

# 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:

年 级:

学 生:

学 号:

完 成 时 间: 年 月 日

成 绩:

指 导 教 师:

重庆大学教务处制

项目	分值	优秀	良好	中等	及格	不及格	评分
		$100 > x \geq 90$	$90 > x \geq 70$	$80 > x \geq 70$	$70 > x \geq 60$	$x < 60$	
		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:

# MIPS SOC 设计报告

## 1 设计简介

本次硬件综合实训设计了一个基于经典五级流水线可以上板运行 57 条 MIPS 指令并通过 axi 总线连接了 4 路组相联 cache 的 CPU。实验的最初代码来自于小组成员在计算机组成原理课程实验 4 中设计的可以运行 10 条 MIPS 指令的 CPU。在此基础上经过以下 5 个阶段的修改,我们逐步完成了最终的 CPU 设计。

- 第一阶段:在 lab4 的基础上我们按照逻辑运算指令-> 移位运算指令-> 数据移动指令-> 算术运算指令-> 转移指令的顺序,逐步扩展到 52 条指令。这一部分主要修改了 controller 模块、alu 模块和 datapath 模块,除此之外还需要添加 hilo 寄存器,乘除法器等功能模块,至此可以通过 6 组独立测试。
- 第二阶段:为了进行功能测试,我们在扩展到 52 条指令过后就通过 sram 接口连接 soc,这一部分需要修改顶层模块。在功能测试的前 64 个测试点,我们主要调试了 hazard 模块,让我们的 52 条指令的 CPU 比较完备。
- 第三阶段:这个阶段添加的 5 条指令全部和异常处理有关,包括特权指令和自陷指令。为此我们设计了 cp0 协处理器,设计过程中借鉴了参考代码,不过之后也重写了 cp0 寄存器的实现代码,优化了部分逻辑并去掉了冗余功能。至此可以通过功能测试的全部 89 个测试点。
- 第四阶段:在这个阶段,我们需要将 CPU 通过类 sram 接口连接到 axi 总线,这里参考了计算机组成原理实验 5 的代码,复用了部分 axi 接口相关的文件和基础 cache,经过我们的调试后先后通过了 axi 的功能测试和性能测试并上板得到了性能分数,至此我们完成了本次硬件综合设计的基础要求。
- 第五阶段:这个阶段我们开始对能通过性能测试的 CPU 进行优化,以获得更好的性能分数。我们优化了除法器,将其的运行周期数压缩至 8 个周期;优化了乘法执行的逻辑,将其拆分成 2 个周期运行,减小了 WNS,可以提高 CPU 的频率;优化了 cache,采用写回策略和 4 路组相联的形式提升访存的性能。

### 1.1 小组分工说明

在本次硬件综合实训中小组的两名成员各自完成了一个 axi 项目,均通过性能测试,分别实现不同版本的除法器 and cache,比较了 vivado 仿真综合后的 WNS 和性能分数,一起研讨了包括写回 cache 的组相联,写缓冲区,写回缓冲区等优化策略,研究了块多字和 axi

burst 传输的可行性,最后考虑到开发板资源和项目时间限制,综合选择了一个可以上板运行的最优版本。

## 2 设计方案

### 2.1 总体设计思路

本次硬件设计主要设计和完善了一个可以基于 AXI 总线协议运行的 SOC 系统。其中,系统内存 (Ram)、外设 (Confreg), 以及负责管理和仲裁上述从设备与处理器间的数据交互请求, 并进行数据传输的 AXI 1X2 Bridge 模块均已在课程资料包中提供。因此, 本次设计的主要目标是构建一个支持 AXI 协议的 MIPS32 架构的 CPU, 即下图中的 mycpu\_top 模块。

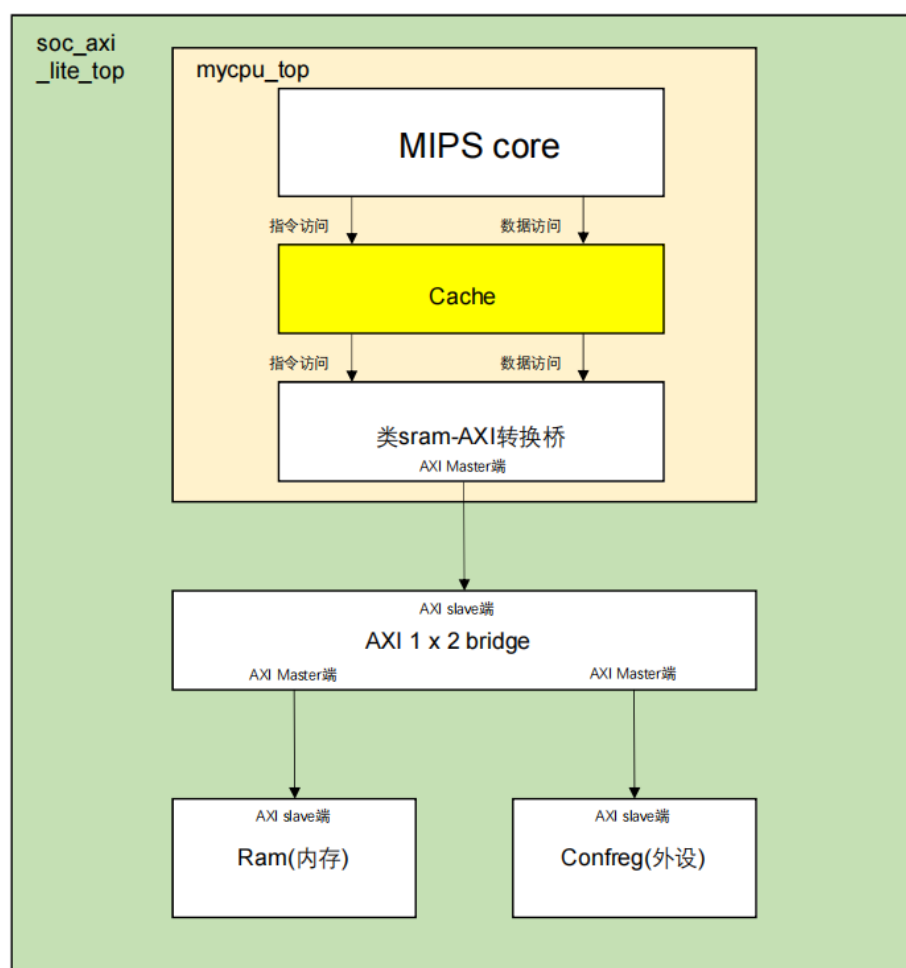


图 1: SOC 顶层结构图

本项目设计的 mycpu\_top 主要包含以下模块组件:

- **中央处理器 MIPS:** 在计算机组成原理-实验四中实现的五级流水线 CPU 基础之上,

拓展实现 MIPS32 指令系统所包含的所有非浮点 MIPS I 指令和 MIPS32 中的 ERET 指令, 并且实现了少量的 CP00 寄存器, 以支持中断和系统调用。不实现 TLB、MMU 和特权等级。

- **内存管理单元 MMU:** 以直接映射方式, 通过读取地址高位将 CPU 需要访问的虚拟地址转换为物理地址, 并判断数据访问请求的类型 (内存或外设)。
- **桥 Bridge 1x2:** 根据 MMU 所给出的数据访问请求类型, 分离或合并内存访问请求和外设访问请求。其中, 仅内存访问请求会经过数据 Cache。
- **高速缓存 Cache:** 用以弥补 CPU 和内存间的速度鸿沟。包含指令缓存 InstCache 和数据缓存 DataCache 两个部分。
- **桥 Bradge 2x1:** Brige 2x1 的逆向工程。不同点在于, 同一时刻 CPU 只能发出内存访问请求和外设访问请求中的一种, 因此 Brige 1x2 对 CPU 发来的请求只进行判断, 不进行仲裁; 而 Bridge 2x1 则可能同时收到来自数据 Cache 的内存访问请求和来自 CPU 的外设访问请求, 此时其内部仲裁策略为内存访问优先。
- **AXI 接口 AXInterface:** mycpu\_top 内部以类 Sram 方式进行数据交互。所有对外的类 Sram 请求都由 AXI 接口统一转换为 AXI 请求。同时, AXI 接口将外层传回的数据重新转换回类 Sram 格式, 以便内层模块使用。其内部仲裁策略为指令访问优先。

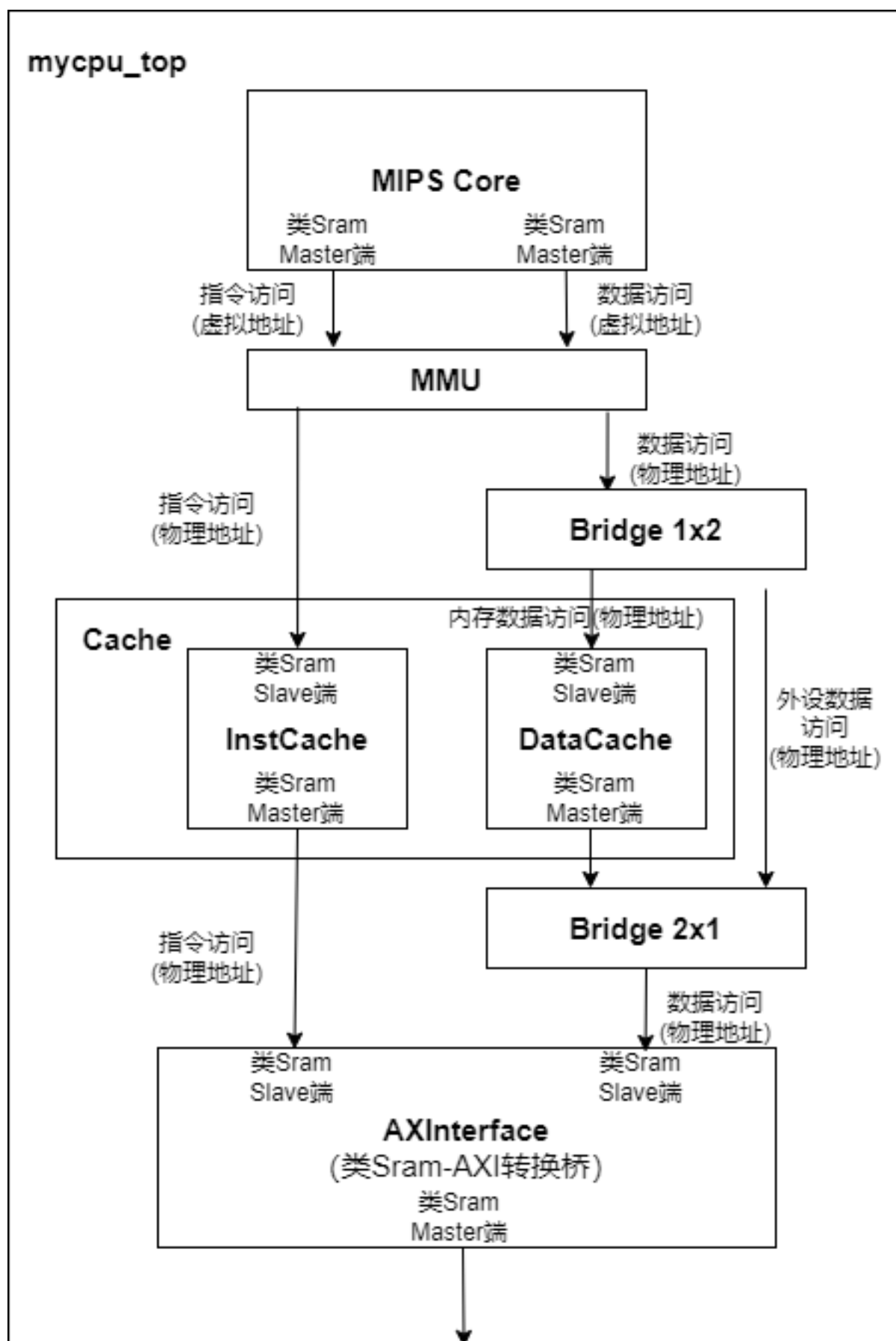


图 2: CPU 顶层结构图

## 2.2 MIPS 模块设计

基于 MIPS 架构的五级流水线 CPU, 本次课程设计的核心部分。其中控制模块 Controller, 数据通路 Datapath 和冒险控制模块 HazardUnit 在实验四中已经基本实现, 在本次设计中需要对其进行适当的功能修改和添加, 以支持前述的全部 57 条指令运行; 类 Sram 接口 Sramlike 则是本次课程设计新增的功能模块, 主要功能为将 CPU 发送的数据请求从 Sram 格式转换为类 Sram 格式, 以实现与外层模块更高效的数据交互。

### 2.2.1 Controller 模块设计

CPU 的主控制单元, 包含主控制译码器 Maindec、ALU 控制译码器 Aludec 以及流水线寄存器的控制信号部分, 整体架构与实验四中完全一致。在此基础上, 添加了若干新的控制信号, 并对流水线进行了相应调整。由于 Aludec 和流水线寄存器的工作过程相较于实验四基本没有变化, 只是简单的堆砌新的对应关系, 因此不再赘述。此处重点阐述 Maindec 模块所产生的各控制信号含义及其解析过程。

表 1: 控制信号定义

信号名	位宽	功能描述
RegWrite	1-bit	写寄存器堆
RegDst	1-bit	目的寄存器号来源,0->Rt 字段,1->Rd 字段
AluSrc	1-bit	第二个 ALU 操作数源,0-> 第二个寄存器堆输出,1->立即数符号扩展
RetSrc	2-bit	MEM 级 Result 源,0->Ex 级 Result,1->HILO 寄存器,2->CP0 寄存器
MemWrite	1-bit	写数据存储器
MemRead	1-bit	读数据存储器, 写入寄存器数据源 (等效 MemtoReg),0-> 来自 Result,1-> 来自数据存储器
HiloWrite	1-bit	写 HILO 寄存器
HilotoReg	1-bit	读/写 HILO 寄存器源,0->Lo 字段,1->Hi 字段
HiloSrc	1-bit	写 HILO 寄存器源,0->ALU,1-> 数据移动指令
Branch	2-bit	高位为 1 表示为 BLTZ 或 BGEZ 指令, 低位为 1 表示为其他非 Link 类型的 Branch 指令
Jump	1-bit	Jump 指令
JumpSrc	1-bit	Jump 的地址源,0-> 立即数,1->rs 寄存器
Link	1-bit	Ex 级 Result 源,0-> 来自 ALUResult,1-> 来自 Xal(r) 指令指定的 PC 值
LinkDst	1-bit	WriteReg 源,0-> 来自目的寄存器号 (rd),1->\$ra
Eret	1-bit	ERET 指令, 直接在 Datapath 中判断 (全字段匹配) 并接入控制器, 以免误判为保留指令
Break	1-bit	BREAK 指令
Syscall	1-bit	SYSCALL 指令
Reserve	1-bit	保留的指令
CP0Write	1-bit	写 CP0 寄存器
DelaySlot	1-bit	延迟槽指令, 在 Controller 中连接 BranchD,JumpD 至 IF 阶段判断

其中需要特别说明的是控制信号 Eret、DelaySlot 和 Reserve。实验设计中所实现的其他所有指令均可以通过 1 至 2 个指令字段 (op, funct, rs 或 rt) 进行识别, 而 Eret 则需要对 ID 阶段的指令进行全字段匹配, 因此直接在 Datapath 中进行判断, 而后接入主控制单元以避免该指令被识别为保留指令。同时, 将其他所有控制信号置 0; 对于 DelaySlot, 与其他所有只与当前 ID 阶段指令相关的控制信号不同, DelaySlot 体现的是分支和跳转与其延迟槽指令间的关系, 因此是整个流水线中唯一产生在 IF 阶段的控制信号; 而对于 Reserve 信号, 当且仅当所有识别条件均不符合时, 该信号有效, 表明当前的 CPU 设计无法识别出给定指令的功能; 除此之外的其他控制信号判断与实验四基本一致, 不再赘述。

```
assign {RegWrite, RegDst, AluSrc, RetSrc,
        MemWrite, MemRead,
        HilowWrite, HilowReg, HilowSrc,
        Branch,
        Jump, JumpSrc,
        Link, LinkDst,
        Break, Syscall, Reserve,
        CP0Write} = Controls;
```

图 3: 控制信号的生成

```
always @(*)
begin
    if (eret)
        Controls <= 20'b00000_00_000_00_00_00_000_0;
    else begin
        case (op)
            R_TYPE:
                case (Funct)
                    "AND", "OR", "XOR", "NOR", "SLL", "SRL", "SRA", "SLLV", "SRLV", "SRVV", "ADD", "ADDU", "SUB", "SUBU", "SLT", "SLTU": Controls <= 20'b11000_00_000_00_00_00_000_0;
                    "MFHI": Controls <= 20'b11001_00_010_00_00_00_000_0;
                    "MFLO": Controls <= 20'b11001_00_000_00_00_00_000_0;
                    "MTHI": Controls <= 20'b00000_00_111_00_00_00_000_0;
                    "MTLO": Controls <= 20'b00000_00_101_00_00_00_000_0;
                    "MULT", "MULTU", "DIV", "DIVU": Controls <= 20'b00000_00_100_00_00_00_000_0;
                    "JR": Controls <= 20'b00000_00_000_00_11_00_000_0;
                    "JALR": Controls <= 20'b11000_00_000_00_11_10_000_0;
                    "BREAK": Controls <= 20'b00000_00_000_00_00_00_100_0;
                    "SYSCALL": Controls <= 20'b00000_00_000_00_00_00_010_0;
                    default: Controls <= 20'b00000_00_000_00_00_00_001_0;
                endcase
            "ANDI", "XORI", "ORI", "LUI", "ADDI", "ADDIU", "SLTI", "SLTIU": Controls <= 20'b10100_00_000_00_00_00_000_0;
            "BEQ", "BNE", "BLEZ", "BGTZ": Controls <= 20'b00000_00_000_01_00_00_000_0;
            "REGIMM_INST": Controls <= (rt == "BLTZ" || rt == "BGEZ") ? 20'b00000_00_000_10_00_00_000_0 : 20'b11000_00_000_10_00_11_000_0;
            "J": Controls <= 20'b00000_00_000_00_10_00_000_0;
            "JAL": Controls <= 20'b11000_00_000_00_10_11_000_0;
            "SW", "SH", "SB": Controls <= 20'b00100_10_000_00_00_00_000_0;
            "LW", "LH", "LHU", "LB", "LBU": Controls <= 20'b10100_01_000_00_00_00_000_0;
            "SPECIAL3_INST":
                case (rs)
                    "MFC0": Controls <= 20'b10010_00_000_00_00_00_000_0;
                    "MTC0": Controls <= 20'b00000_00_000_00_00_00_000_1;
                    default: Controls <= 20'b00000_00_000_00_00_00_001_0;
                endcase
            default: Controls <= 20'b00000_00_000_00_00_00_001_0;
        endcase
    end
end
```

图 4: 控制信号的生成



### 2.2.2 Datapath 模块设计

数据通路 Datapath 是 CPU 功能单元的集合, 还包括数据在各功能部件之间的传输路径。其中, PC 寄存器, 寄存器堆 Regfile 和算术逻辑单元 ALU 等功能部件在实验四中已基本实现, 在本次项目设计无需进行任何更改, 或仅需进行小幅度修改, 因此不再进行说明。以下内容将重点关注本次设计中新增加的以下六个功能模块: BranchCompare、MDU、HILO、Translator、Exception 和 CP0。

- **BranchCompare:** 位于 ID 级的分支比较模块。在实验四中, 由于仅实现了 beq 指令, 分支跳信号 PCSrc 当且仅当比较数 A 和 B 相等时有效。对于扩展后的 57 条指令, 分支指令的数量已增长至 8 种, 并且需要同时检测 ID 级指令的 op 和 rt 字段以识别指令类型, 因此需要增设该模块以完成相应的比较处理。

```
//PCSrc:为1表示跳转地址来源于Branch

always@(*)
begin
    case(op)
        ~BEQ: PCSrc <= SrcA == SrcB;
        ~BNE: PCSrc <= SrcA != SrcB;
        ~BLEZ: PCSrc <= (SrcA == `ZeroWord) || SrcA[31];
        ~BGTZ: PCSrc <= (SrcA != `ZeroWord) && ~SrcA[31];
        ~REGIMM_INST:
            case(rt)
                ~BLTZ, ~BLTZAL: PCSrc <= SrcA[31];
                ~BGEZ, ~BGEZAL: PCSrc <= ~SrcA[31];
                default: PCSrc <= 1'b0;
            endcase
        default: PCSrc <= 1'b0;
    endcase
end
```

图 5: Branch 指令比较模块

- **MDU:** 位于 EX 级的乘除法 (Multiplication and Division Units) 模块。与实验四中实现的算术指令 add, 逻辑指令 and 等可以在单周期内计算完成的指令不同, 乘除法往往需要多个时钟周期才能完成一次计算, 在过程中需要对流水线各级进行阻塞或刷新, 而此前的 ALU 设计则是纯组合逻辑的。因此, 有必要增设该模块来实现乘除法指令的时序控制。相较于 ALU, MDU 同样需要通过控制字段 CONTROL 来识别指令类型, 操作数 A 和 B 也同样可以直接复用 ALU 的输入; 不同之处在于, 由于 MDU 实际上是一个时序逻辑单元, 因此需要接入阻塞信号 StallE 和刷新信号 FlushE, 以维持与外部流水线的相对同步。

对于**除法**, 由于 Vivado 本身提供的除法实现性能较为糟糕, 而资料包中所提供的除法实现需要 36 个周期才能完成一次除法计算, 因此本次项目设计尝试实现了乘性快速除法器 Goldschmidt, 其可以在 8 个周期内完成一次除法计算。相关的功能信号

如下表所示:

表 2: 除法功能信号定义

信号名	位宽	功能描述
clk	1-bit	时钟信号
rst	1-bit	复位信号
A,B	32-bit	(被) 除数
Signed	1-bit	有符号除法
Start	1-bit	除法开始, 在除法结束 (Ready 信号拉高) 前, 该输入始终拉高
Enable	1-bit	除法结束时, 当且仅当该信号为 1 时 Ready 信号可以拉低, 除法器可以执行新的除法计算
Annual	1-bit	除法终止
Ready	1-bit	除法结束
Claim	1-bit	除法结束声明
Divon	1-bit	除法进行中
Count	3-bit	除法迭代次数
lastall	1-bit	使能信号刚来
divA,divB	32-bit	(被) 除数绝对值补码
clzA,clzB	32-bit	divA,divB 的二进制表示前导 0 数
mantissA,mantissB	32-bit	divA,divB 的规格化尾数表示
normQ	32-bit	商的规格化表示
tmpQ	32-bit	商的整数表示
TwoMinusYi	64-bit	迭代方程
Xi,TwoMinusYi	128-bit	迭代方程
regA,regB	64-bit	迭代寄存器
regSigned	1-bit	保留除法开始时的 Signed, 避免计算过程中输入发生变化 (下同)
OffsetA,OffsetB	32-bit	clzA,clzB
Divdend	32-bit	A
Divisor	32-bit	B
mulD,mulQ,tmpQ	32-bit	已知除数, 被除数, 商, 求余数
Quotient	32-bit	商
Remainder	32-bit	余数

该算法基于麦克劳林公式, 将除法转换为乘法进行迭代计算。对于 32 位整数, 迭代次数不超过  $\log(32) = 5$  次。下面对状态机设计进行简要说明:

- **除法开始:** 在整个除法计算的过程中, Start 信号始终拉高, 表明除法指令正在执行。
- **除法执行:** 在第一个周期对除法输入进行保存后, Divon 信号拉高, 表明除法正式开始, 进行 5 个周期的商数计算和 1 个周期的余数计算。
- **除法完成:** 经过上述若干周期的计算后, 商和余数已经迭代至正确结果。若此时外部流水线未发生其他类型的阻塞 (例如取指或访存导致的阻塞), 则使能信号有效, 除法结束, Divon 和 Ready 信号拉低。
- **除法挂起:** 若此时外部流水线仍然阻塞, 则除法器进入挂起状态, 除法结果保持不变, Ready 信号持续拉高, 以避免同一条除法指令的重复执行甚至错误执行。
- **除法声明:** 若此时外部流水线未发生其他类型的阻塞, 使能信号有效, 但使能信号刚好在上一个时钟沿抵达除法器, 则 Ready 信号拉低, Claim 信号拉高, 表明

除法刚刚结束。

下面对状态机设计作必要说明：

(1) 首先, 当除法器与外部流水线时钟正接, 除法指令进入 EX 级时, 其内部寄存器与 EX-MEM 级的流水线寄存器处于相同层级。当除法指令因计算结束进入 EX-MEM 级流水线时, 除法器会将已经结束的除法指令错误识别需要执行的除法指令, 导致除法指令的无限执行。因此, 除法器不能与外部流水线时钟正接。

(2) 当除法器与外部流水线时钟反接时, 以取指暂停为例, 当外部流水线发生阻塞时, 虽然除法指令阻塞在 EX 级, 但除法器内部的多周期计算并不发生阻塞。当取指暂停持续时间超过除法暂停时间时, 如果不挂起除法器, 除法器将进行第二次除法计算。极端情况下, 如果第一次除法开始时的操作数由 MEM 级或 WB 级数据前推而来, 当第二次除法开始时, 上述操作数已经写回寄存器堆, 却被阻塞在 ID 级而无法到达 EX 级除法器的输入端口, 因此除法器将产生错误的除法结果, 并覆盖原本的正确结果。因此, 除法器必须挂起, 等待外部使能信号到来, 则表明流水线重新开始工作。此时除法器可以拉低 Ready 信号并等待下一条除法指令。

(3) 然而, 更加极端的情况下, 当取指暂停信号在外部时钟上行沿拉低, 而 Ready 信号恰好在紧随其后的下行沿拉低时, 下一个外部时钟上升沿, 流水线会误认为除法器尚未计算完成而阻塞流水线, 在下一个外部时钟下降沿, 除法器仍然会产生前述错误计算问题; 而当取指暂停信号在若干个时钟前的时钟上升沿已经拉低, 则在上述第一个外部时钟上行沿, 流水线便重新开始工作, 除法指令离开 EX 级, 避免在上述第二个外部时钟下行沿的重复计算。因此, 需要检测外部使能信号的到达时间, 当上述巧合发生时进行除法声明, 多进行半个周期的除法结束声明, 避免阻塞流水线。遗憾的是, 由于该除法器在单周期内需要执行 64 位乘法, 严重降低了 CPU 的时钟频率, 且除法指令在所有指令中的占比仅不到 5%。综合考量下, 运行开发板性能测试时仍然修改采用参考代码中给出的加性除法器。

对于乘法, Vivado 提供的乘法实现已经具有较为良好的时序, 可以近似认为是单周期计算, 但 WNS 值较差。因此, 参考前述除法器的状态机, 将乘法分为 2-3 个周期计算, 以实现“WNS 值处于安全范围内”这一大前提下的更高的 CPU 时钟频率。

- **HILO:** 与 EX\_MEM 级流水线同级的 HILO 寄存器。与其他所有算术运算指令不同的是, 乘除法的计算结果保留在 HILO 寄存器中, 并不直接写回寄存器堆, 而是需要调用 mfhi, mflo 指令才能取回计算结果。对于乘法, HILO 寄存器分别保存乘法结果的高 32 位和低 32 位; 对于除法, HILO 寄存器则分别保存除法的余数和商。相关信号定义如下:

表 3: HILO 寄存器相关信号定义

信号名	位宽	功能描述
HiloEn	1-bit	流水线使能信号
HiloWrite	1-bit	HILO 寄存器写信号
HilotoReg	1-bit	HILO 寄存器选择信号。0 表示 LO 寄存器,1 表示 HI 寄存器
HiloSrc	1-bit	HILO 寄存器写控制信号。0 表示数据移动指令需要写入 HI 寄存器或 LO 寄存器,1 表示将乘法或除法结果写入 HI 寄存器和 LO 寄存器
Hiloin	32-bit	数据移动指令需要写入的数据
MDUResult	64-bit	乘除法的计算结果
Hilout	32-bit	HILO 寄存器的输出

这一设计的优势在于,任何需要从 MEM 级进行数据前推的相关指令,都可以快速地从流水线上的 HILO 寄存器中直接读取所需要的操作数;与同级流水线上的其他寄存器的不同之处在于,对于 HILO 寄存器,由于其本身并不传输至下一流水级,因此 Stall 信号和 Flush 信号实际上是等价的,即避免 EX 级的任何结果被写入 HILO 寄存器中。

```
//MEM流水线刷新的时候,Hilo应该是阻塞
HILO Hilo_EX_MEM(clk,rst,~HiloStallM,HiloWriteE,HilotoRegE,HiloSrcE,
SrcAE,MDUResultE,HiloutM);
assign HiloStallM = StallM || FlushM;
```

图 6: HILO 寄存器的阻塞控制

```
reg [31:0] Hi,Lo;
always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        Hi <= 32'b0;
        Lo <= 32'b0;
        Hilout <= 32'b0;
    end
    else if(HiloEn) begin
        if(HilotoReg)
            Hilout <= Hi;
        else Hilout <= Lo;
        if(HiloWrite)
        begin
            if(HiloSrc)
            begin
                if(HilotoReg)
                    Hi <= Hiloin;
                else Lo <= Hiloin;
            end
            else begin
                {Hi,Lo} <= MDUResult;
            end
        end
    end
end
```

图 7: HILO 寄存器

- **Translator:** MEM 级的地址转换模块。通过 op 字段识别访存指令的类型, 将外部的 Sram 类型的数据输入转换为 CPU 寄存器类型的数据, 或将 CPU 寄存器类型的数据转换为 Sram 类型的接口输出到外部存储器, 并检测可能出现的地址异常。其相关信号如下表所示:

表 4: Translator 相关信号定义

信号名	位宽	功能描述
MemWrite	1-bit	数据存储器读使能信号
MemRead	1-bit	数据存储器写使能信号
MemEnable	1-bit	数据存储器使能信号
MemWen	4-bit	数据存储器写选择信号,0 表示读该字节,1 表示写该字节
MemAddr	32-bit	访存地址 (虚拟地址)
ReadData	32-bit	从数据存储器读入的数据
TReadData	32-bit	转换后的读入数据
WriteData	32-bit	需要写入数据存储器的输出数据
TWriteData	32-bit	转换后的输出数据

该模块为简单的组合逻辑映射。对于读数据操作,4 位的 MemWen 信号全 0,CPU 从外部存储器获取整个字的数据内容, 然后根据需要保留其中对应的若干字节内容; 对于写数据操作,4 为 MemWen 信号为全 1,1100/0011 或者单个 1, 分别表示写入 4 字节,2 字节或 1 字节数据, 并使用目标字节数据填充无效字节。

```

always@(*)
begin
  case(op)
    `SW:
      begin
        TWriteData <= WriteData;
        MemWen <= 4'b1111;
      end
    `SH:
      begin
        TWriteData <= {2{WriteData[15:0]}};
        case(addr)
          2'b00: MemWen <= 4'b0011;
          2'b10: MemWen <= 4'b1100;
          default: MemWen <= 4'b0;
        endcase
      end
    `SB:
      begin
        TWriteData <= {4{WriteData[7:0]}};
        case(addr)
          2'b00: MemWen <= 4'b0001;
          2'b01: MemWen <= 4'b0010;
          2'b10: MemWen <= 4'b0100;
          2'b11: MemWen <= 4'b1000;
          default: MemWen <= 4'b0;
        endcase
      end
    default:
      begin
        TWriteData <= 32'b0;
        MemWen <= 4'b0;
      end
    endcase
end

always@(*)
begin
  case(op)
    `LW: TReadData <= ReadData;
    `LH:
      case(addr)
        2'b00: TReadData <= {{16{ReadData[15]}},ReadData[15:0]};
        2'b10: TReadData <= {{16{ReadData[31]}},ReadData[31:16]};
        default: TReadData <= 32'b0;
      endcase
    `LHU:
      case(addr)
        2'b00: TReadData <= {{16{1'b0}},ReadData[15:0]};
        2'b10: TReadData <= {{16{1'b0}},ReadData[31:16]};
        default: TReadData <= 32'b0;
      endcase
    `LB:
      case(addr)
        2'b00: TReadData <= {{24{ReadData[7]}},ReadData[7:0]};
        2'b01: TReadData <= {{24{ReadData[15]}},ReadData[15:8]};
        2'b10: TReadData <= {{24{ReadData[23]}},ReadData[23:16]};
        2'b11: TReadData <= {{24{ReadData[31]}},ReadData[31:24]};
        default: TReadData <= 32'b0;
      endcase
    `LBU:
      case(addr)
        2'b00: TReadData <= {{24{1'b0}},ReadData[7:0]};
        2'b01: TReadData <= {{24{1'b0}},ReadData[15:8]};
        2'b10: TReadData <= {{24{1'b0}},ReadData[23:16]};
        2'b11: TReadData <= {{24{1'b0}},ReadData[31:24]};
        default: TReadData <= 32'b0;
      endcase
    default: TReadData <= 32'b0;
  endcase
end

```

图 8: Translator

特别注意,对于读操作,因为类 Sram 接口是按字节访问数据的,而 CPU 是按字访问数据的,因此无论 CPU 读取的是整字,半字还是字节,统一读取整字数据内容后再进行相应处理,此时访存地址的低 2 位必须清空;对于写操作则需要保留访存地址的低 2 位,以便外部存储器从正确的字节基址开始写入数据:

```
assign addr = EXResult[1:0];
assign MemError = Adel || Ades;
assign MemEnable = (~MemError) && (MemWrite | MemRead);
assign MemAddr = (MemWrite)? EXResult : {EXResult[31:2],2'b0};
```

图 9: 访存地址

- **Exception:** MEM 级的例外处理模块,该模块负责识别除中断异常外的所有异常类型,并产生对应的例外编码 EXcCode 以供 CP0 模块进行精确异常处理。

```
always@(*)
begin
    if(rst)
        EXcCode <= `Ne;
    else begin
        if(ExcepType[4] || Adel)//读地址例外
            EXcCode <= `AdEL;
        else if(ExcepType[5] || Ades)//写地址例外
            EXcCode <= `AdES;
        else if(ExcepType[8])//系统调用
            EXcCode <= `Sys;
        else if(ExcepType[9])//断点
            EXcCode <= `Bp;
        else if(ExcepType[10])//保留指令
            EXcCode <= `RI;
        else if(ExcepType[12])//算术溢出
            EXcCode <= `Ov;
        else if(ExcepType[14])//ERET
            EXcCode <= `Eret;
        else EXcCode <= `Ne;
    end
end
end
```

图 10: 异常类型

各异常的含义,产生的流水级和原因如下表所示,按处理的优先级从上到下排序:

表 5: 异常类型

异常名称	产生的流水级	异常描述	异常原因
Int	MEM	软硬件中断	mtc0 指令产生的软件中断或外部硬件产生的硬件中断
AdEL	IF 或 MEM	读指令或读数据地址异常	地址低位与访存指令类型不匹配
AdES	MEM	写数据地址异常	地址低位与访存指令类型不匹配
Sys	ID	系统调用	Syscall 指令
Bp	ID	断点	Break 指令
Remainder	ID	保留指令	主控制单元无法识别的指令
Ov	EX	算术溢出	有符号加减法结果溢出
Eret	EX	异常处理返回	Eret 指令

● **CP0: MEM 级的 CP0 寄存器模块。**本次项目设计实现的 CP0 寄存器类型如下:

- **BadVAddr:** 记录最新地址相关例外 (Adel 或 Ades) 的出错地址。BadVAddr 寄存器是一个只读寄存器, 用于记录最近一次导致发生地址错例外的虚地址, 不能通过 (mtc0) 指令对 BadVAddr 寄存器进行写操作。
- **Count:** 处理器内部计数器, 两个时钟计数一次。用于触发时钟中断, 本次项目设计不实现时钟中断。
- **Status:** 处理器状态与控制寄存器。寄存器各比特位含义如下图所示:

域名称	位	功能描述	读/写	复位值
0	31..23	只读恒为 0。	0	0
Bev	22	针对大赛, 恒设为 1	R	1
0	21..16	只读恒为 0。	0	0
IM7..IM0	15..8	中断屏蔽位。每一位分别控制一个外部中断、内部中断或软件中断的使能。 1: 使能; 0: 屏蔽。	R/W	无
0	7..2	只读恒为 0。	0	0
EXL	1	例外级。当发生例外时该位被置 1。0: 正常级; 1: 例外级。 当 EXL 位置为 1 时: ◆ 处理器自动处于核心态 ◆ 所有硬件与软件中断被屏蔽 ◆ EPC、Cause 在发生新的例外时不做更新。	R/W	0x0
IE	0	全局中断使能位。 0: 屏蔽所有硬件和软件中断; 1: 使能所有硬件和软件中断。	R/W	0x0

图 11: Status 寄存器

● **Cause:** 存放上一次例外原因。寄存器各比特位含义如下图所示:

域名称	位	功能描述	读/写	复位值
IP7..IP2	15..10	待处理硬件中断标识。每一位对应一个中断线, IP7~IP2 依次对应硬件中断 5~0。 1: 该中断线上有待处理的中断; 0: 该中断线上无中断。	R	0x0
IP1..IP0	9..8	待处理软件中断标识。每一位对应一个软件中断, IP1~IP0 依次对应软件中断 1~0。 软件中断标识位可由软件设置和清除。	R/W	0x0
0	7	只读恒为 0。	0	0
ExcCode	6..2	例外编码。详细描述请见表 6-5。		
0	1..0	只读恒为 0。	0	0

图 12: Cause 寄存器

● **EPC:** 存放上一次发生例外指令的 PC。当直接触发例外的指令位于分支延迟槽时, EPC 寄存器保存其前一条指令单地址, 同时将 Cause 寄存器的 BD 字段置 1。



当直接触发例外的指令不位于分支延迟槽时,处理器向 EPC 寄存器中写入例外处理完成后继续执行的指令地址。

精确异常的基本概念和实现原理如下:

- **精确异常处理的一般过程:** 当 MEM 级判断异常处理开始前,处理器状态尚未被前三级流水线错误修改,且 MEM 级是本次项目设计中可能产生异常的最后一个流水级。因此无论当前指令导致的异常在哪一流水级产生,统一在 MEM 级进行处理,以保证精准的处理最先发生异常或例外的指令。当异常处理开始后,流水线将刷新所有流水线寄存器,以避免异常指令或其后续指令错误修改处理器状态,同时 PC 寄存器将跳转至统一的异常处理地址入口,EPC 寄存器写入例外返回地址,Cause 和 Status 寄存器的对应字段发送修改,记录例外处理情况和例外原因。
- **中断处理:** 最高级别例外。本次项目设计中只实现软件中断。软件中断的产生原理为软件调用 mtc0 指令,该指令在 MEM 阶段不经过 Exception 模块,而是直接对 Cause 寄存器的对应比特位进行写操作,因此无论 mtc0 之后的一条指令执行情况如何,该指令都将被认为触发了软件中断,PC 寄存器将跳转至异常处理程序入口地址;特别强调,与其他异常指令的不同之处在于,触发中断异常处理的是 mtc0 指令的下一条指令,而不是其本身。
- **Eret 指令的处理:** 异常处理返回。严格来说,Eret 并不是一种异常,但同样需要对 CP0 寄存器的状态进行部分修改,因此采用类似的处理。该异常触发时,PC 寄存器将跳转至 EPC 寄存器中保存的指令地址,即从异常处理程序中返回并继续往后执行;特别强调,与其他分支跳转指令的不同之处在于,由于 Eret 指令理论上是异常处理程序的最后一条指令,其延迟槽地址内存放的可能是来自其他程序的指令,因此 MEM 级的 Eret 指令不能执行其延迟槽指令或后续的任何其他指令。而是需要像其他异常指令一样对流水线进行彻底的刷新,以避免程序的错误运行。
- **BadVAddr 寄存器处理:** 当且仅当异常类型为地址异常时,向 BadVAddr 寄存器中写入错误地址。
- **EPC 寄存器处理:** 当异常指令不是延迟槽指令时,EPC 中写入该指令的 PC 地址,以便异常处理程序返回后重现从该指令继续运行;当异常指令是延迟槽指令时,EPC 中写入的是延迟槽指令的上一条指令,即分支跳转指令的 PC 地址。因为跳转后的程序在异常处理开始前尚未执行,因此异常处理完成后需要重新从此处开始执行。



```

if(Interrupt)
begin
    if(DelaySlot)
    begin
        EPC <= PC - 32'd4;
        Cause[31] <= 1'b1;
    end
    else begin
        EPC <= PC;
        Cause[31] <= 1'b0;
    end
    Cause[6:2] <= `Int;
    Status[1] <= 1'b1;
    ExceptDeal <= 1'b1;
end

else begin
    case(EXcCode)
        `AdEL, `AdES, `Sys, `Bp, `RI, `Ov:
        begin
            if(DelaySlot)
            begin
                EPC <= PC - 32'd4;
                Cause[31] <= 1'b1;
            end
            else begin
                EPC <= PC;
                Cause[31] <= 1'b0;
            end
            Cause[6:2] <= EXcCode;
            Status[1] <= 1'b1;
            ExceptDeal <= 1'b1;
        end
        `Eret:
        begin
            Cause[6:2] <= EXcCode;
            Status[1] <= 1'b0;
            ExceptDeal <= 1'b1;
        end
        default: ExceptDeal <= 1'b0;
    endcase
end

case(EXcCode)
    `AdEL, `AdES: BadVAddr <= BadAddr;
    default ;
endcase

```

图 13: 精确异常处理

添加异常处理后的 PC 寄存器地址计算如下图所示, 其中 ExceptDeal 表示产生了任意类型异常, 需要刷新流水线; 当异常类型不为 Eret 时, 需要跳转至异常处理地址 0xbfc00380。

```

assign PC = (EretM)? EPC :
            (ExceptDealM)? 32'hbfc00380 :
            (JumpD)? ((JumpSrcD)? SrcAD : Jump_AddrD) :
            (PCSrcD)? Branch_AddrD : PCplus4F;

```

图 14: 添加异常处理后的 PC 寄存器

mtc0 和 mfc0 指令的实现与数据移动指令 mfhi,mflo 大同小异, 此处不再赘述。

### 2.2.3 HazardUnit 模块设计

冒险控制模块。该模块的大部分实现与实验四一致, 此处重点阐述需要进行修改或添加的部分。

- **ForwardXD:** 在实验四中, ForwardXD 控制 ALU 的计算结果从 MEM 级前推至 ID 级进行 Branch 比较。而对于访存结果则未从 WB 级前推至 ID 级, 而是等待其在 WB 级的时钟下行沿写入寄存器堆, 然后直接从寄存器堆中读取以进行比较。然而,

这种方式将导致 Branch 比较模块计算以及依赖寄存器值的 jr 指令跳转地址计算必须在半个时钟周期内完成,不利于 CPU 时钟频率的提高,因此模仿 ForwardXE,增设 WB 级结果至 ID 级的前推控制。

```
//数据冒险
assign ForwardAE = (RegWriteM && (WriteRegM != 0) && (WriteRegM == RsE)) ? 2'b10 : //EX冒险前推信号
                  (RegWriteW && (WriteRegW != 0) && (WriteRegW == RsE)) ? 2'b01 : 2'b00; //MEM冒险前推信号
assign ForwardBE = (RegWriteM && (WriteRegM != 0) && (WriteRegM == RtE)) ? 2'b10 : //EX冒险前推信号
                  (RegWriteW && (WriteRegW != 0) && (WriteRegW == RtE)) ? 2'b01 : 2'b00; //MEM冒险前推信号

assign ForwardAD = (RegWriteM && (WriteRegM != 0) && (WriteRegM == RsD)) ? 2'b10 :
                  (RegWriteW && (WriteRegW != 0) && (WriteRegW == RsD)) ? 2'b01 : 2'b00;
assign ForwardBD = (RegWriteM && (WriteRegM != 0) && (WriteRegM == RtD)) ? 2'b10 :
                  (RegWriteW && (WriteRegW != 0) && (WriteRegW == RtD)) ? 2'b01 : 2'b00;
```

图 15: ForwardXD

- **JumpStall:** 扩展后的跳转指令可能需要在 ID 级读取寄存器值 (jr,jalr), 因此需要新增相应的数据前推和阻塞控制逻辑。实验四中已经实现了 Branch 指令的数据前推和阻塞控制, 因此此处除了指令类型标识不同外, 其余逻辑直接复用即可。

```
assign jumpstall = JumpSrcD && ((RegWriteE && (WriteRegE != 0) && (WriteRegE == RsD)) ||
                               ((MemReadM || RetSrcM[1]) && (WriteRegM == RsD)));
assign branchstall = BranchD[1] ? (RegWriteE && (WriteRegE != 0) && (WriteRegE == RsD)) ||
                                   ((MemReadM || RetSrcM[1]) && (WriteRegM == RsD)) :
                                   BranchD[0] ? (RegWriteE && (WriteRegE != 0) && ((WriteRegE == RsD) || (WriteRegE == RtD))) ||
                                   ((MemReadM || RetSrcM[1]) && ((WriteRegM == RsD) || (WriteRegM == RtD))) : 1'b0; //EX冒险阻塞信号
```

图 16: Jumpstall

- **CP0Stall:** 扩展后的异常处理指令需要读写 MEM 级的 CP0 寄存器, 因此同样需要新增控制逻辑。实验四中已经实现了 MEM 级访存指令的数据前推和阻塞控制, 而 CP0 寄存器的读写和访存指令的实现逻辑基本一致, 因此同上直接进行复用即可。

```
//lw冒险阻塞信号(充分不必要条件,在后一条指令为某些指令(见"指令及对应机器码")
//时可能会造成不必要的阻塞,前述的前推控制信号大概也有类似问题)
assign lwstall = ((RtE != 0) && (RsD == RtE) || (RtD == RtE)) && MemReadE;
assign cp0stall = ((RtE != 0) && (RsD == RtE) || (RtD == RtE)) && RetSrcE[1];
```

图 17: CP0Stall

- **MDUReady:** 本次项目设计新增的阻塞控制信号。当 EX 级未执行乘除法指令时,MDUReady 的值恒定为 1; 当 EX 级执行乘除法指令且未计算完毕,即前述 Ready 信号和 Claim 信号均未拉高时,MDUReady 的值为 0, 表明乘除法尚未产生完整的计算结果,因此需要持续阻塞流水线的前三级,同时刷新 EX\_MEM 级流水线,避免不正确的乘除法结果对处理器状态产生影响。

```

always@(*)
begin
    case(MDUControl)
        `MULT_CONTROL, `MULTU_CONTROL:
            begin
                MulStart <= ~MulReady;
                DivStart <= 1'b0;
            end
        `DIV_CONTROL, `DIVU_CONTROL:
            begin
                MulStart <= 1'b0;
                DivStart <= ~DivReady;
            end
        default:
            begin
                MulStart <= 1'b0;
                DivStart <= 1'b0;
            end
    endcase
end
assign MDUReady = (MDUControl == `MULT_CONTROL || MDUControl == `MULTU_CONTROL)? MulReady || MulClaim :
(MDUControl == `DIV_CONTROL || MDUControl == `DIVU_CONTROL)? DivReady || DivClaim : 1'b1;

```

图 18: MDUReady

- **MemStall:** 本次项目设计新增的阻塞控制信号。当 IF 级取指或 MEM 级访存未结束时, 全流水线阻塞, 等待上述过程执行结束。对于取指阻塞 InstStall, 由于 PC 寄存器的值无法进行更新, 因此全流水线阻塞既不影响性能, 也便于实现; 而对于访存阻塞 DataStall, 流水线的前四级显然必须阻塞, 而最后一级也必须阻塞, 而不是刷新, 因为与计算开始便保留所有输入的 MDU 不同, 此时被阻塞在 EX 级的 ALU 指令可能需要 WB 级的前推结果, 如果 WB 级此时写回寄存器堆并刷新, 则 ALU 既不能从 WB 级取得正确的操作数, 又不能从寄存器堆取得正确的操作数, 因而将产生错误的计算结果。

完整的冒险控制信号如下图所示。当因 MemStall 导致全流水线暂停时, 所有的刷新信号都不能有效, 否则可能导致流水线内的信号被清空但未传递至下一流水级, 从而导致指令丢失; 当进行 ExceptDeal 处理时, 除了 IF 级的 PC 寄存器外, 全流水线寄存器都需要进行刷新, 以避免指令的错误执行。

```

//阻塞与刷新
assign stalls = lwstall || jumpstall || branchstall || cp0stall;
//触发异常处理时已经进入流水线的指令可能会阻塞PC, 导致PC无法取得异常处理程序地址; 由于这些指令此后将被清除, 因此其触发的stallf需要一并清除
assign Stallf = MemStall || (~ExceptDealM) && (stalls || (~MDUReadyE)); //lw, beq和乘法指令均需阻塞ID级和IF级指令
assign StallD = MemStall || stalls || (~MDUReadyE);
assign StallE = MemStall || (~MDUReadyE); //乘法指令还需要阻塞EX级
assign StallM = MemStall;
assign StallW = MemStall;
//beq指令分支发生时或j指令执行时, 如果编译器没有调度合适的延迟槽指令, 则需要清空ID级指令(样例默认有)
// assign FlushD = PCSrcD | JumpD;
assign FlushD = (~MemStall) && ExceptDealM;
assign FlushE = (~MemStall) && (ExceptDealM || stalls); //lw数据冒险或beq(数据)控制冒险均在EX级判断, 需清空ID级指令影响
assign FlushM = (~MemStall) && (ExceptDealM || (~MDUReadyE)); //若乘法法未结束, 则需清空其对MEME级的影响
assign FlushW = (~MemStall) && ExceptDealM;

```

图 19: HazardUnit

### 2.2.4 Sramlike 模块设计

将 CPU 发来的取指或访存请求从单周期的 Sram 类型转换为任意周期的类 Sram 类型, 其状态机分为请求, 地址收到, 数据收到并等待 CPU 确认共 3 种状态, 与实验五完全一致, 此处不再赘述。状态机结构如下图所示 (以访存转换为例, 取指转换基本一致):

```
reg addr_rcv;//地址收到
reg data_rcv;//读:收到cache数据,等待cpu接受 写:cache数据已写入
reg [31:0] data_buffer;//数据被接收前的缓存

always@(posedge clk)
begin
    addr_rcv <= (rst)? 1'b0 :
                (data_data_ok)? 1'b0 :
                (data_req && data_addr_ok)? 1'b1 : addr_rcv;
end

always@(posedge clk)
begin
    data_rcv <= (rst)? 1'b0 :
                (data_data_ok)? 1'b1 :
                (~StallM)? 1'b0 : data_rcv;
end

//读:收到数据确认信号时同步保留数据 写:无意义
always@(posedge clk)
begin
    data_buffer <= (rst)? 32'b0 :
                  (data_data_ok)? data_rdata : data_buffer;
end
```

图 20: Sramlike

## 2.3 MMU 模块设计

将 CPU 发出的虚拟地址转换为物理地址。MIPS 虚拟地址空间的内存映射的 32 位视图如下:

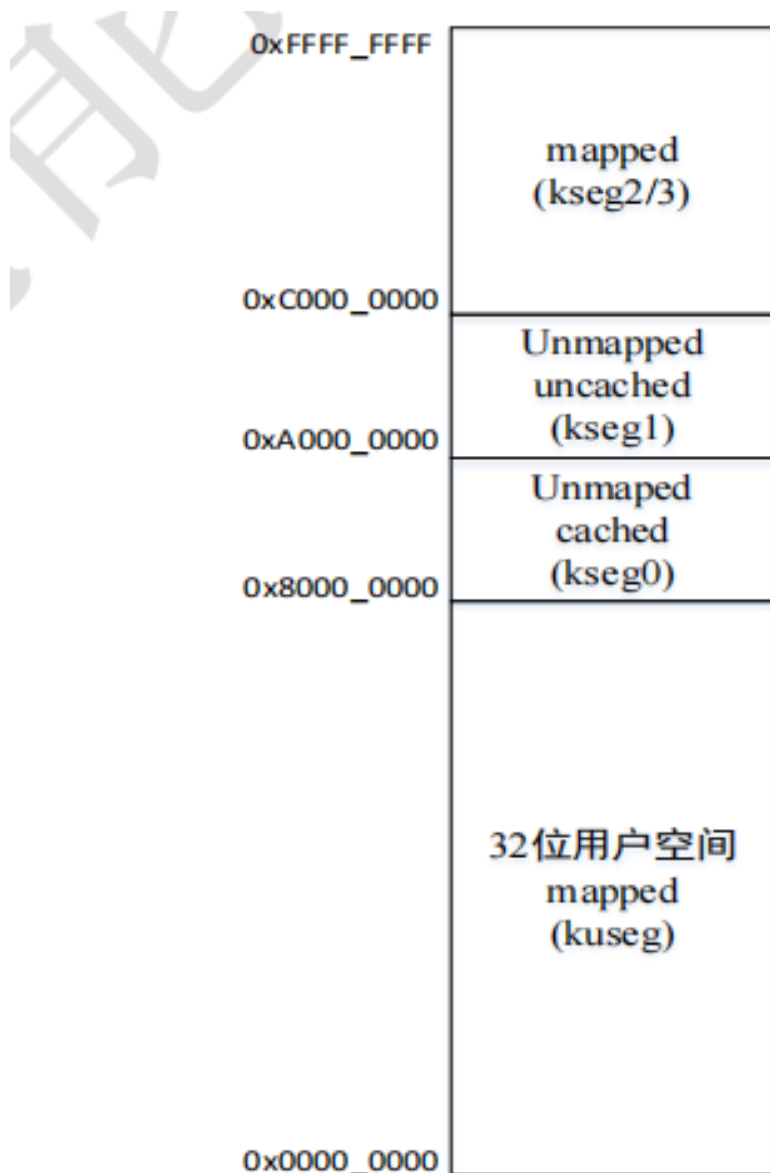


图 21: 地址映射

mapped 和 unmapped 表明是否需要通过 MMU 进行地址转换。kseg0 和 kseg1 属于 unmapped 段, 永远指向物理地址 0x0000\_0000 0x1fff\_fff。kuseg 和 kseg2/3 则需要经过 MMU 单元进行地址翻译的。处理器刚启动时,MMU 处于未设置状态, 因此无法对 kuseg 和 kseg2/3 进行地址翻译, 这同时解释了 MIPS32 处理器为什么需要从 0xbfc0\_0000(kseg1 段) 启动。

本次课程设计中实现的是直接映射方式的 MMU, 同时外设 (Confreg) 也被认为是虚拟地址空间的一部分,MMU 通过地址高位直接判断访存请求是否前往外设, 并生成 now\_dcache 信号控制 Bridge 模块进行数据请求的分离和合并。

```

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 now_dcache = data_kseg0;

```

图 22: MMU

## 2.4 Bridge 模块设计

### 2.4.1 Bridge 1x2 模块设计

基于 now\_dcache 信号区分数据请求, 两类请求不会同时产生, 不涉及仲裁逻辑。

### 2.4.2 Bridge 2x1 模块设计

本次课程设计中使用的数据 Cache 实现了写回缓冲区, 因此 Bridge 2x1 可能同时收到来自 CPU 的外设访问请求和来自数据 Cache 的内存写请求, 此时模块的内部仲裁策略为数据 Cache 请求优先。

```

wire no_dcache;

assign no_dcache = (~now_dcache) && (~ram_data_wr);

//up
assign ram_data_rdata = (no_dcache) ? 32'b0 : wrap_data_rdata;
assign ram_data_addr_ok = (no_dcache) ? 1'b0 : wrap_data_addr_ok;
assign ram_data_data_ok = (no_dcache) ? 1'b0 : wrap_data_data_ok;

assign conf_data_rdata = (no_dcache) ? wrap_data_rdata : 32'b0;
assign conf_data_addr_ok = (no_dcache) ? wrap_data_addr_ok : 1'b0;
assign conf_data_data_ok = (no_dcache) ? wrap_data_data_ok : 1'b0;

//down
assign wrap_data_req = (no_dcache) ? conf_data_req : ram_data_req;
assign wrap_data_wr = (no_dcache) ? conf_data_wr : ram_data_wr;
assign wrap_data_size = (no_dcache) ? conf_data_size : ram_data_size;
assign wrap_data_addr = (no_dcache) ? conf_data_addr : ram_data_addr;
assign wrap_data_wdata = (no_dcache) ? conf_data_wdata : ram_data_wdata;

```

图 23: Bridge 2x1



## 2.5 Cache 模块设计

### 2.5.1 DataCache 模块设计

本次项目设计实现了写回策略的数据 Cache, 并实现了以下 3 种时间局部性优化策略: 组相联, 写缓冲区和写回缓冲区。由于四路组相联占用了过多开发板资源, 因此未能找到足够的 FPGA 资源用以实现空间局部性优化策略: 块多字, 算是很遗憾。

除了部分组合逻辑关键路径优化外, 本次项目设计的数据 Cache 在实验五中已经基本完成, 因此不再展开叙述。此处对各优化策略进行重点说明:

- **组相联:** 本次项目设计实现的是两路组相联和四路组相联的 Cache, 两者具有相同的数据存储规模, 后者在多选器, 替换策略方面需要消耗更多的 LUT 资源以进行并行判断。对于两路组相联, 每组的替换策略实现仅需 1 个比特位, 记录最后一次访问命中的路即可, 当替换发生时, 另外一路将被替换出 Cache; 对于四路组相联, 采用的是实验五:Cache 实验指导书提到的类二叉树型伪 LRU 算法, 原理和实现如下:

#### 8.2.2 伪 LRU 替换算法

如上所述, 采用 LRU 算法需要消耗过多的资源, 因此常使用伪 LRU 算法替代。一种伪 LRU 算法通过, 对 cache 的路进行分组, 每组使用一个位来表示最近是否使用过。对于有  $2^n$  路的 cache, 每个 cache set 需要  $2^n - 1$  bit 位来存储最近访问信息。

图 10 展示了伪 LRU 算法的过程。对于 4 路的结构, 每一个 cache set 需要 3bit 来存储最近使用信息。将这 3 个比特组织成树状结构, 每一 bit 可以表示其左子树或右子树是否被访问过。为了方便说明, 对每个 bit 节点进行编号, 按照从上往下, 从左往右的顺序分别编号为 1, 2, 3。

伪 LRU 算法确定替换哪一路的算法类似于一个二分查找, 每次通过 1bit 的信息可以将范围缩小到原本的一半, 若干次迭代后便可以确定到具体的某一路。具体算法如下: 从根节点(1 号节点)开始, 如果其值为 0, 则表示其右子树(对应的路, 即 0 路, 1 路)最近没有访问过。反之, 则表示其左子树最近没有被访问。假设 1 号节点为 0, 则我们通过判断 3 号节点的值来判断替换哪一路: 如果为 0, 替换 0 路, 如果为 1, 替换 1 路。假设 1 号节点为 1, 我们便通过 2 号节点来判断替换哪一路, 过程类似。

当访问了 cache 的某一路后, 需要对 bit 信息进行更新。更新的算法一句话来说就是保持每一 bit 的语义不变, 即每一 bit 代表其左右子树最近是否被访问过。具体来说其实只需要更新访问路径(从根节点到某一路的连线)上的节点的状态值即可。(并非直接取反, 而是保持语义)

如图 10 所示, 根据上述的确定替换路的算法, 刚开始该 cache set 的状态表示应该替换 way0, 但在访问了一次 way0 后, 需要更新访问 way0 路径上的节点也就是 0 号和 3 号节点的状态。对 0 号节点来说, 由于访问了其右半子树, 因此更新为 1, 对 3 号节点同理更新为 1。在更新之后, 该 cache set 的状态变为应该替换 way2。

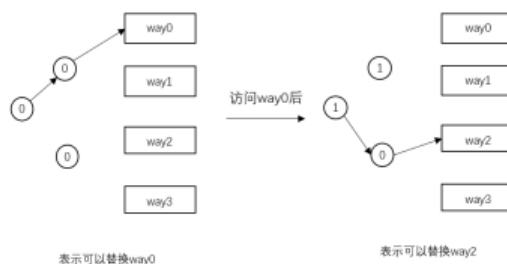


图 10: 伪 LRU 替换算法过程

图 24: FLRU 的工作原理

```

module FLRU(
    input clk,rst,
    input enable,
    input [1:0] target,
    output [1:0] replace
);

    reg rt;
    reg [1:0] sn;

    always@(posedge clk or posedge rst)
    begin
        if(rst)
        begin
            rt <= 1'b0;
            sn <= 2'b0;
        end
        else if(enable)
        begin
            rt <= ~target[1];
            sn[target[1]] <= ~target[0];
        end
    end

    assign replace = {rt,sn[rt]};
endmodule

```

图 25: FLRU 的四路组相联实现

其中,enable 在 Cache 命中且命中的是本组时有效,target 表示本次命中的路,replace 表示替换策略更新后的可能被替换的路。

- **写缓冲区:** 在写回 Cache 中,由于无法重写块,store 操作需要两个周期(一个周期用来检查命中情况,下一个周期真正执行写操作)。本次项目设计提供一个写缓冲区来保存数据,通过流水线有效地使存储操作只花费一个周期。处理器在正常的 Cache 访问周期内查找 cache 并将数据同步存储在存储缓冲区中,如果 cache 命中,在下一个时钟周期,数据从存储缓冲区中写入 cache,而 cache 同步进行下一次命中的判断。



```

//写缓冲区
reg store_dirty;
reg [BLOCK_WIDTH-1:0] store_buffer;
reg [31:0] store_addr;
wire [TAG_WIDTH-1:0] store_tag;
wire store_write;

assign store_tag = store_addr[31:INDEX_WIDTH+OFFSET_WIDTH];
assign store_index = store_addr[INDEX_WIDTH+OFFSET_WIDTH-1:OFFSET_WIDTH];
assign store_write = (state == RM)? cache_data_data_ok : (cpu_data_req && cpu_data_wr && hit);
assign store_target = (state == RM)? replace[store_index] : target;

always@(*)
begin
    store_addr <= cpu_data_addr;
    if(state == RM)
    begin
        if(cpu_data_wr)
        begin
            store_dirty <= 1'b1;
            store_buffer <= rwrite_data;
        end
        else begin
            store_dirty <= 1'b0;
            store_buffer <= cache_data_rdata;
        end
    end
    else begin
        store_dirty <= 1'b1;
        if(cpu_data_req)
            store_buffer <= write_data;
        else store_buffer <= 32'b0;
    end
end
end

```

图 26: 写缓冲区的实现

```

integer j,k;
always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        for(j = 0;j < CACHE_DEPTH;j = j + 1)
        for(k = 0;k < ASSOCIATIVITY;k = k + 1)
        begin
            cache_valid[j][k] <= 1'b0;
            cache_dirty[j][k] <= 1'b0;
            cache_tag[j][k] <= 1'b0;
            cache_block[j][k] <= 32'b0;
        end
    end
    else if(store_write)
    begin
        cache_dirty[store_index][store_target] <= store_dirty;
        cache_valid[store_index][store_target] <= 1'b1;
        cache_tag[store_index][store_target] <= store_tag;
        cache_block[store_index][store_target] <= store_buffer;
    end
end
end

```

图 27: 写缓冲区的实现

- **写回缓冲区:** 本次项目设计使用写回缓冲区, 用于在发生缺失并替换一个被修改的

块时降低缺失代价。在这种情况下, 被修改的块并不立刻写回主存, 而是被移入写回缓冲区, 同时从主存中读出所需要的块, 并向 CPU 直接报告操作完成, 以使 CPU 可以继续运行, 同时写回缓冲区中的数据被写回主存。如果下一次缺失没有立刻发生, 那么当一个脏块必须被替换时, 这种方法可以将缺失代价减半。

```
//写回缓冲区
reg [BLOCK_WIDTH-1:0] write_buffer;
reg [31:0] write_addr;

always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        write_buffer <= 32'b0;
        write_addr <= 32'b0;
    end
    else if((state == RM) && cache_dirty[store_index][store_target])
    begin
        write_buffer <= cache_block[store_index][store_target];
        write_addr <= {cache_tag[store_index][store_target], store_index, 2'b00};
    end
end
end
```

图 28: 写回缓冲区的实现

### 2.5.2 InstCache 模块设计

指令 Cache 可以由数据 Cache 去除写操作逻辑直接得到, 因此不再赘述。在优化策略方面, 与访存相比, 取指操作的空间局部性特征更加明显, 然而数据 Cache 业已未解决开发板资源不足的问题, 指令 Cache 的优化便也无从谈起。

## 2.6 AXInterface 模块设计

对来自 InstCache 的取指请求和来自 Bridge 2x1 的访存请求进行仲裁, 并转换为 AXI 格式。仲裁逻辑为指令请求优先。本次项目设计使用的是参考资料包中提供的类 Sram-AXI 转换桥, 其并未实现 AXI 的 Burst 传输。但由于前述原因, 块多字策略无法实现, 因而 AXI 的 Burst 传输也失去了实现的意义, 因此本次课程设计未对该模块作出过多修改, 不再进行说明。

## 3 设计过程

### 3.1 设计流水账

2023 年 12 月 25 日 8:30~10:30

... 和... 到教室上课, 听... 老师讲解硬件综合设计课程的大致规划和任务目标。

2023 年 12 月 25 日 11:00~23:30

... 和... 分别配置了 vivado 2019.2 的环境,并将 vivado 的编辑器连接到了 vscode,这样可以获得更清晰的代码高亮和补正。考虑到... 的 lab4 的代码更加规范和完善,决定将... 的 lab4 代码作为我们小组的初版代码,将其代码连接到了资料包中的 lab4 工程并成功运行。剩下的时间开始各自学习重庆大学硬件综合设计实验文档的内容,修改代码。

2023 年 12 月 26 日 10:00~23:30

... 添加了逻辑运算指令和移位指令,通过了第一组和第二组独立测试,记录了添加过程中的思路和遇到的 bug。

... 添加了逻辑运算指令和移位指令,通过了第一组和第二组独立测试,记录了添加过程中的思路和遇到的 bug。

2023 年 12 月 27 日 10:00~23:30

... 添加了数据移动指令,和... 讨论了 hilo 寄存器的放置位置,最终决定将其放置在 MEM 流水线级。在添加数据移动指令的通路时,考虑过通过 alu 进行数据移动,但是和... 讨论过后发现没有必要,通过了第三组独立测试。

... 添加了数据移动指令和除乘除法外的算术运算指令。添加数据移动指令时和... 讨论了 hilo 寄存器的放置位置,最终决定将其放置在 MEM 流水线级,通过了第三组独立测试。

2023 年 12 月 28 日 10:00~23:30

... 添加了算术运算指令,通过了第四组独立测试。

... 添加了算术运算指令,通过了第四组独立测试。... 注意到拓展任务中的除法器的优化,开始查阅资料,寻找优化除法器的方法。

2023 年 12 月 29 日 10:00~23:30

... 配置了 verilator 的运行环境,在 Win10 系统上安装了 gtkwave。

... 找到了优化除法器的可行方法,选择了能找到的最快的除法器优化方法开始优化,成功用优化过后的除法器通过第四组独立测试。

2023 年 12 月 31 日 10:00~23:30

... 调整了自己控制除法器的状态机,使其既能运行参考代码中的除法器也能运行... 优化后的除法器。还添加了分支跳转指令,通过了第五组独立测试。

... 添加了分支跳转指令,通过了第五组独立测试。

2024 年 1 月 1 日 10:00~23:30

... 添加了访存指令,通过了第六组独立测试,至此所有独立测试全部通过。

... 添加了访存指令,通过了第六组独立测试,至此所有独立测试全部通过。

2024 年 1 月 2 日 10:00~23:30

... 连接了 sram-soc 运行功能测试,发现参考代码中的除法器不能通过功能测试中的除法,经过调试后通过,同时也解决了一些其他冒险模块的 bug 后通过功能测试的前 64 个测试点。

... 连接了 sram-soc 运行功能测试,调试分支跳转指令的冒险模块过后,通过功能测试的前 64 个测试点。

2024 年 1 月 3 日 10:00~23:30

... 和... 观看了前几届学长录制在 B 站的视频,学习了异常处理有关的知识 and 注意事项,讨论了异常处理的相关设计思路,... 决定直接调用参考代码中的 cp0,... 认为参考代码中的 cp0 过于冗余,决定自己重写。

2024 年 1 月 4 日 10:00~23:30

... 完成了异常处理相关的数据通路的添加,开始调试通过功能测试,但是由于冒险模块的 bug 过多,卡在了第 72 个测试点。

... 完成了 cp0 的重写和相关数据通路的添加,开始调试通过功能测试,但是由于冒险模块的 bug 过多,卡在了第 78 个测试点。

2024 年 1 月 5 日 10:00~23:30

... 和... 在互相交流调试心得后,分别完成了自己代码的调试,成功通过 89 个功能测试点。

2024 年 1 月 6 日 10:00~23:30

... 和... 观看了 B 站学长的视频,准备开始连接 axi 总线,但是发现本次实验的资料包中没有提供相关的转接口代码,为了加快速度避免重复的工作,决定复用计算机组成与结构实验 5 的接口代码。

2024 年 1 月 6 日 10:00~23:30

... 和... 连接好 axi 总线后开始测试 axi 项目的功能测试。... 发现优化后的除法器不能通过部分测试点,开始优化除法器,... 发现自己的 pc 不跳转,而且 cp0 寄存器出错,开始调试。最后分别通过了功能测试。

2024 年 1 月 7 日 10:00~23:30

... 和... 分别调试自己的代码后通过了 verilator 上的性能测试。

2024 年 1 月 8 日 10:00~23:30

... 和... 比较了两人代码的质量和性能得分,决定采用... 的代码来进行后续的开发。连接了计组实验 5 中编写的 4 路组相联写回 cache,成功通过性能测试并上板,得到了性能得分,但是此时的 WNS 只能在 30MHz 时为正值。

2024 年 1 月 9 日 10:00~23:30

在确定代码的大致通路后,... 开始用亿图图示绘制数据通路图,... 发现 WNS 的关键路径出现在自己优化后的除法器上,优化除法器。

2024 年 1 月 10 日 10:00~23:30

... 绘制完数据通路图,和... 一起优化关键路径,将 40MHz 时的 WNS 优化为正值。

2024 年 1 月 11 日 10:00~23:30

... 和... 发现优化后的除法器不能支持更高频率的时钟周期,决定换回参考代码中的除法器。让除法器不再是关键路径,同时将乘法拆成两个周期执行,现在的关键路径是 cache。

2024 年 1 月 12 日 10:00~23:30

... 和... 尝试使用 bram 优化 cache,受限于时间条件未完成,最终的成果为 50MHz 时 WNS 为 0.004。

2024 年 1 月 13 日 9:30~15:30

... 和... 现场添加指令和答辩。

2024 年 1 月 14 日 ~15 日

... 和... 撰写报告。

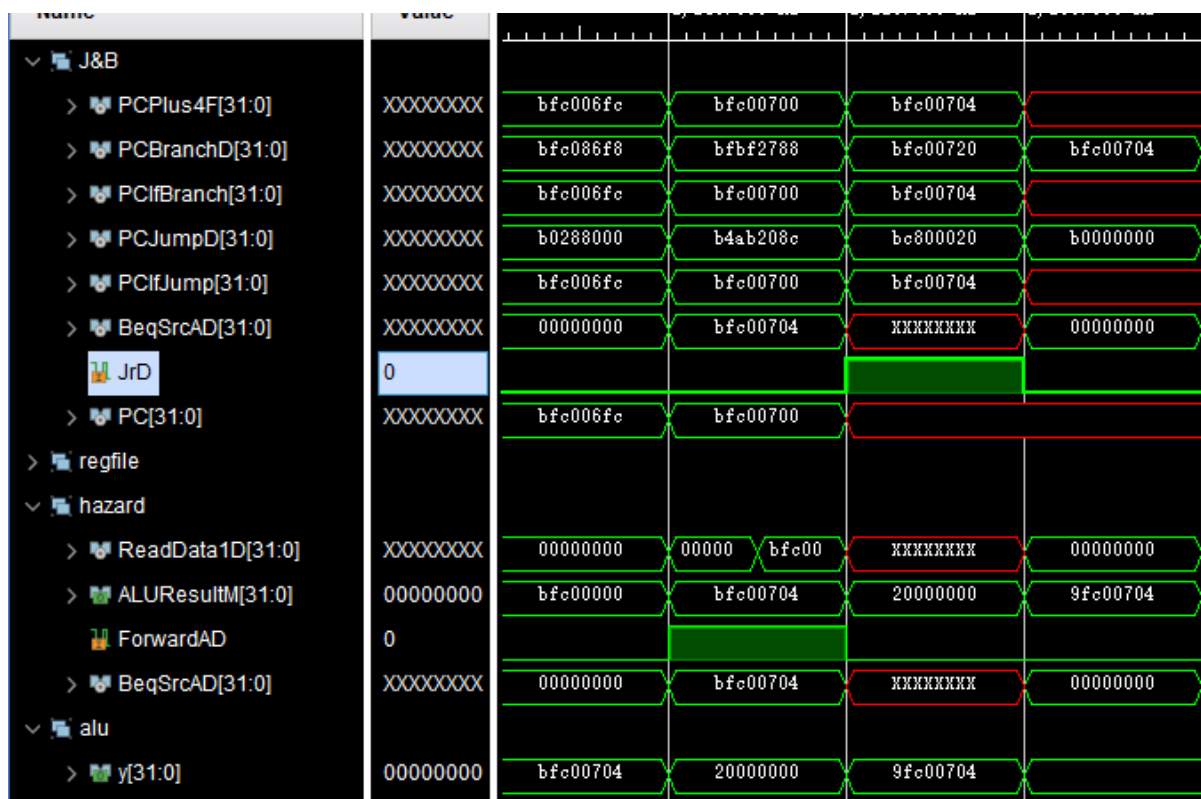
## 3.2 错误记录

### 3.2.1 错误 1

(1) 错误现象:CPU52 功能测试 bfc006fc: 03200008 jr t9 报错

(2) 分析定位过程: 首先我们需要分析这是一条什么类型的指令,发现它一条 jump and

link 类型的指令,那么它会导致 pc 的跳转,而我们的 pc 跳转会和下图中的 7 个信号值有关,现在把这些信号值全部拉出来一一对比,查看哪一步信号值跳转出错。



- (3) 错误原因: 添加 `jr` 型指令后没有修改 `hazard` 模块, 需要添加 `jrstall` 类型的阻塞。
- (4) 修正效果: 如图所示, 在添加了 `jrstall` 信号后, `jr` 指令阻塞了一个周期, 获得了正确的数据前推值, 完成了跳转。
- (5) 归纳总结(可选): 这个错误的出现是因为没有考虑到新加入的指令与之前指令直接的冒险现象, 其实添加指令时以我们的水平很难考虑周全, 所以这类错误几乎是无法避免的, 只能在测试时及时发现, 及时修改。

- (1) 错误现象:CPU52 功能测试 bfc5f7b8: 15560112 bne t2,s6,bfc5fc04 报错
- (2) 分析定位过程:bne 指令出错需要看前后指令的关联,我们先把相关指令都截图展示一下。  
发生错误的 bne 指令的前两条指令都是 lw 指令,具体错误的原因从波形图可以看出:

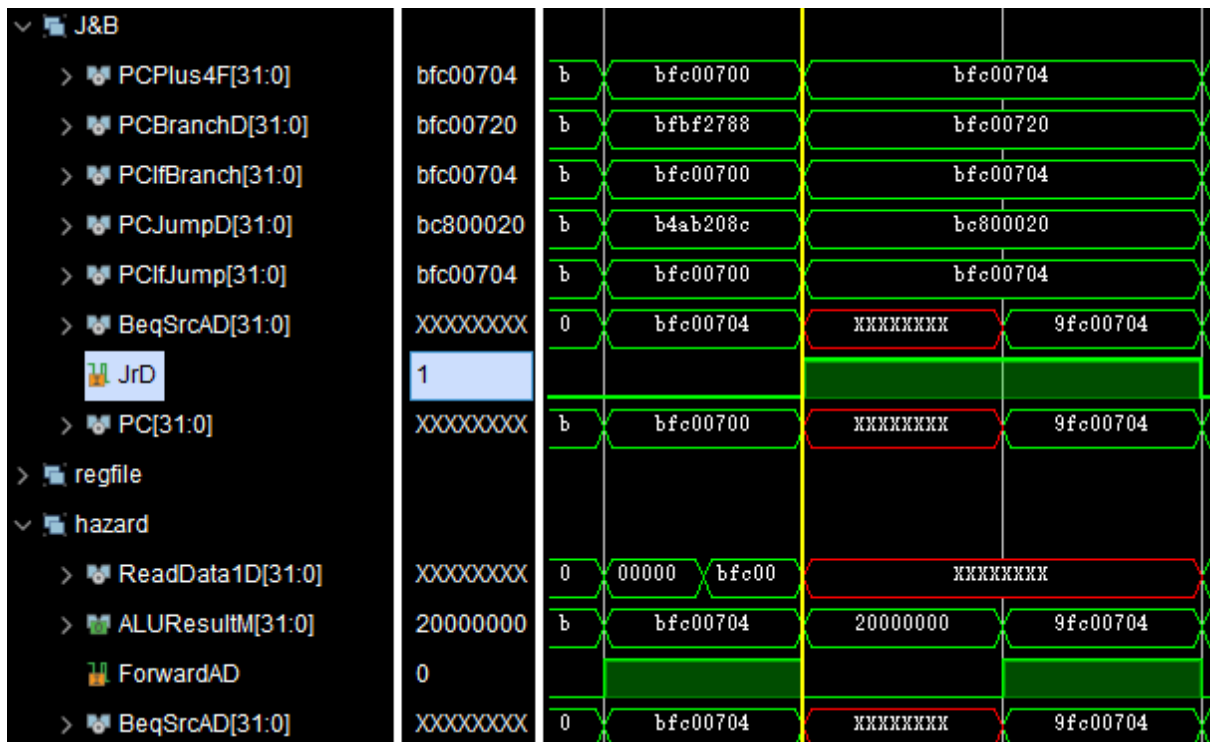


图 30: bfc005fc 正确结果

```

bfc5f7b0: 8d0a8ee0 lw t2,-28960(t0)
bfc5f7b4: 8d168ee0 lw s6,-28960(t0)
bfc5f7b8: 15560112 bne t2,s6,bfc5fc04 <inst_error> # 这里不应该跳转

```

图 31: bfc5f7b8 相关指令

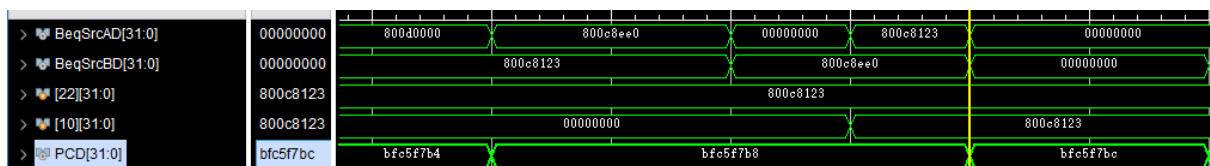


图 32: bfc5f7b8 出错波形图

可以看到在进行 **bne** 比较的时候,两端读入的数据不一样,会发生跳转,但是实际上,t2 寄存器和 s6 寄存器最终存入的值是一样的,出现这个问题的原因在于 **lw** 指令最快取得数据在 **MEM** 阶段,而使用数据在 **bne** 指令的 **ID** 阶段,这里需要 **stall** 两拍。

(3) 错误原因: 这个错位发生的原因类似于上一个错位,branch 指令没有考虑到与 **lw** 指令的关联,hazard 模块没有修改。

(4) 修正效果: 最初写的 **branchstall** 信号如下所示:

```
assign branchstall = (BranchD & RegWriteE & ((WriteRegE == RsD) | (WriteRegE == RtD))
); //MEM冒险阻塞信号
```

这里只阻塞了一个周期,就会导致数据来不及被 **bne** 指令读取。所以需要修改 **branchstall** 信号如下所示:

```
assign branchstall = (BranchD & RegWriteE & (WriteRegE != 0) & ((WriteRegE == RsD) |
(WriteRegE == RtD))) | (MemReadM & ((WriteRegM == RsD) | WriteRegM == RtD));
```

这样就可以成功读取到正确的数据,正确的波形图如下所示:

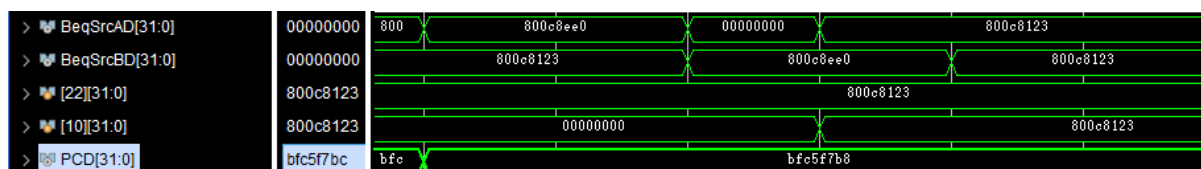


图 33: bfc5f7b8 正确结果

(5) 归纳总结(可选): 这个错误的出现和上一个错误一样,没有考虑全指令之间的联系,也是遇到修改即可。

### 3.2.3 错误 3

(1) 错误现象:CPU52 功能测试 bfc3a938: 0109001a div zero,t0,t1 报错

(2) 分析定位过程: 这是一条除法指令,我们的检查机制是检查写回寄存器堆的时候,写入的寄存器号和写入的寄存器的值是否与参考的一致,而除法指令的值并不会直接写回寄存器堆中,而是写回 **hilo** 寄存器中,而 **hilo** 寄存器的值最后会通过 **mfhi** 和 **mflo** 写回寄存器堆,所以我们这里报错的时候,实际上是报在 **mf** 指令。然后通过向上查找指令可以发现是除法器的计算结果出错了。

这里使用的除法器是参考代码提供的除法器,在查看参考的代码过后,我发现出问题是因为下面这一行代码:

```
if ((signed_div_i == 1'b1) && ((opdata1_i[31] ^ opdata2_i[31]) == 1'b1)) begin
    dividend[31:0] <= (~dividend[31:0] + 1);
end
if ((signed_div_i == 1'b1) && ((opdata1_i[31] ^ dividend[64]) == 1'b1)) begin
    dividend[64:33] <= (~dividend[64:33] + 1);
end
```

表面上看不出问题,但是在波形图中可以注意到操作数 2 也就是 **opdata2\_i** 有一次跳变,这是由于虽然我们阻塞了 **IF**, **ID**, **EX** 三个周期,但是 **MEM** 和 **WB** 阶段的指令



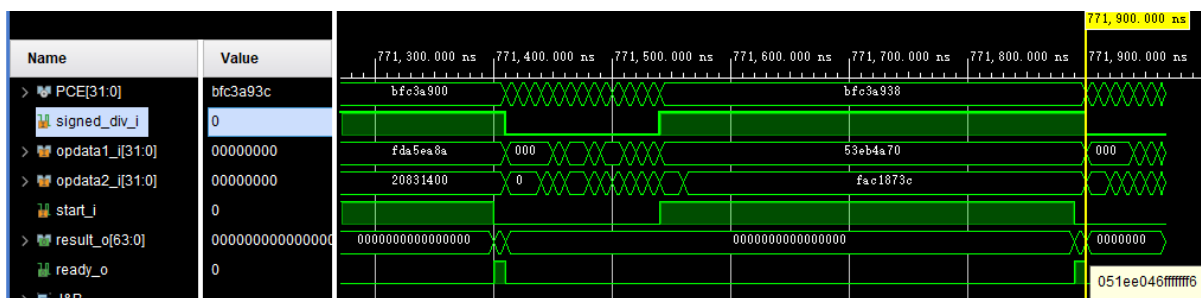


图 34: bfc3a938 出错波形图

还在执行,由于数据前推改变了我们除法器的输入。如果只是改变输入其实不会出现问题,因为除法器内部用 `reg` 存了输入的值,但是在最后判断符号的时候又用了直接输入的值,导致结果出错。

- (3) 错误原因: 除法器的数值虽然在操作过程中用 `reg` 保存下来了。但是在最后判断符号的时候,仍然用的是输入时的操作数,就导致了输入改变的话,最后的结果就会出错。
- (4) 修正效果: 改正的方法也比较简单,只需要把输入的值用 `reg` 存下来,只能在除法器的开始阶段允许赋值一次即可。

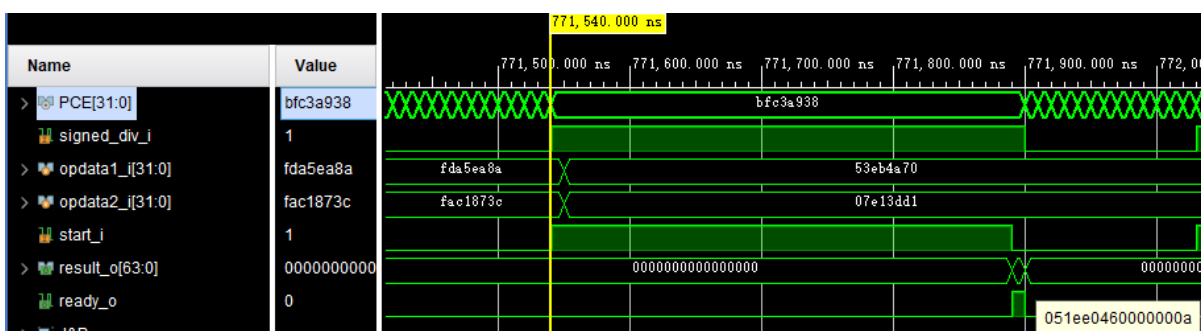


图 35: bfc3a938 正确结果

- (5) 归纳总结(可选): 这一类错误属于是参考代码自身导致的一些错误,这和我们没有十分了解参考代码的实现有关。这种错误需要我们认真研读参考代码才能解决。

### 3.2.4 错误 4

- (1) 错误现象: CPU57 功能测试 bfc42470: 0000000c syscall 报错
- (2) 分析定位过程: 先来看错误波形图: 这里的第一次执行 syscall 指令, 跳转到

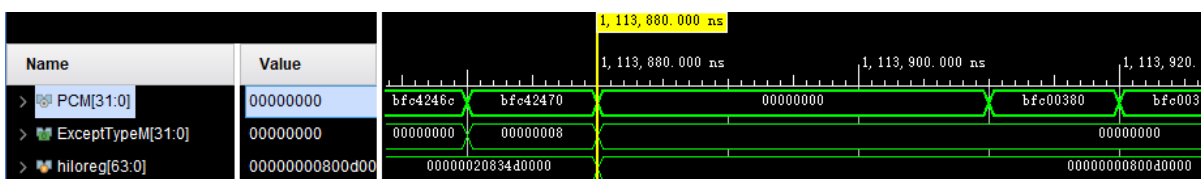


图 36: bfc42470 错误波形图

0xbfc00380 执行异常处理。异常处理的第一条指令是从 hilo 寄存器中取值到 k0 和

k1 寄存器。但是这里出错了,从波形图可以看出,出错的原因在于在异常处理跳转过后,hilo 寄存器的值被覆盖了,所以在这里需要为我们的 hilo 寄存器添加一个 FlushE 接口,但是这个信号不是刷新 hilo 寄存器,而是阻塞 hilo 寄存器。

(3) 错误原因: 异常处理时 hilo 寄存器没有被阻塞,导致值被刷新。

(4) 修正效果: 调试后的波形图如下所示:

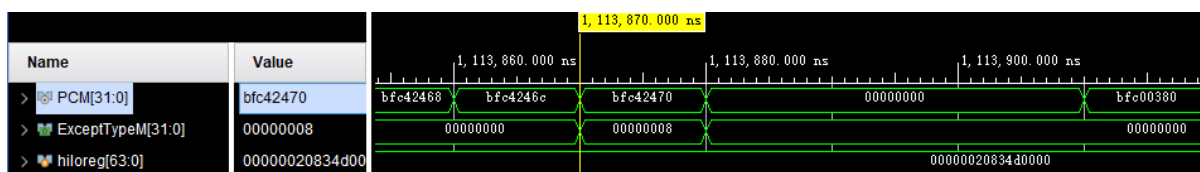


图 37: bfc42470 正确结果

(5) 归纳总结(可选): 这个问题的出现是因为我们填写控制冒险信号的时候忘记考虑其他的时序模块,因为在我们之前的设计中所有的控制冒险都只是控制流水线寄存器,但是现在我们的 hilo 寄存器并没有放在流水线寄存器当中。所以导致我们在这里的时候忽视了。之后考虑流水冒险的时候,我们应该考要考虑所有的有关于时序的寄存器。

### 3.2.5 错误 5

(1) 错误现象:CPU57 功能测试 bfc30ae0: 02f2001b divu zero,s7,s2 报错

(2) 分析定位过程: 先看错误的波形图:

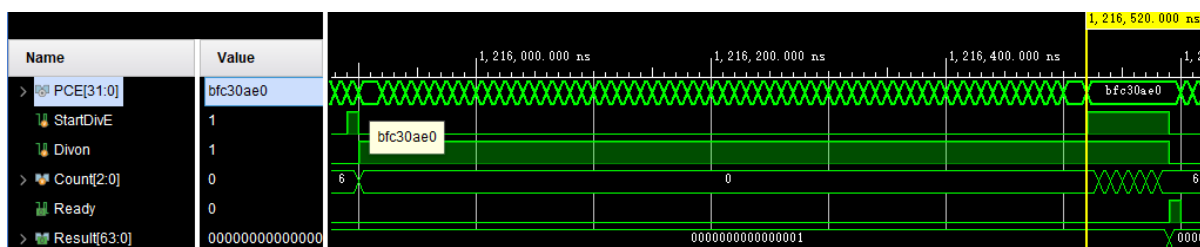


图 38: bfc30ae0 错误波形图

如果仅看最后执行除法的地方,可能会觉得是除法器的问题,但是实际上除法器没有问题,需要把这个波形图拉通看才会发现除法器的 Start 信号被拉高了两次,这就导致了除法器的两次执行,最终结果出错。而为了分析为什么会出错,还需要看指令的执行顺序。

除法指令之前是一条 sw 指令,这一条指令访存地址错误,会执行异常处理,而在处理 sw 异常是在流水线的 MEM 级,此时下一条指令 divu 正处于 EX 级,此时我们会清空 CPU 的五级流水线,转而去执行异常处理程序,在异常处理结束后再返回执行 divu。但是这里发生了问题,在 divu 第一次执行时会拉高一次 Start 信号,但是在返回后还会拉高一次,这时两次拉高 Start 会让除法器重新取数,出现错误。

```

/home/rain/loongson/func_board/inst/n79_bne_ds_ex.S:135
bfc30adc:  ad170002    sw  s7,2(t0)
/home/rain/loongson/func_board/inst/n79_bne_ds_ex.S:136
bfc30ae0:  02f2001b → divu zero,s7,s2

```

图 39: bfc30ae0 相关指令

- (3) 错误原因: 由于传输给除法器的信号错误,导致除法器执行了两次,在第二次执行时除法器的输入改变,导致最终的结果出错。
- (4) 修正效果: 正确的处理逻辑应该是除法器在执行期间如果被阻塞,需要清空除法器,然后在返回后重新执行,所以需要给除法器也接入 FlushE 信号。
- 正确的结果如下图所示:

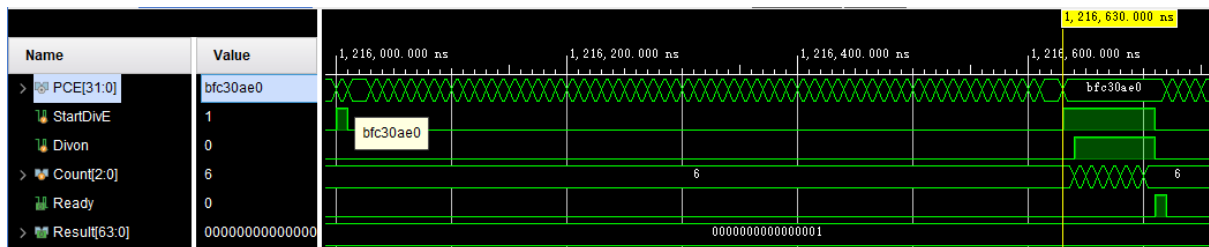


图 40: bfc30ae0 正确结果

修改过后尽管 ‘Start’ 也会拉高两次,但是由于我们重置了除法器的状态,所以不会出现重复运算的问题。

- (5) 归纳总结(可选): 这个错误的类型其实也可以和之前的错误类型归纳在一起。都是在时序执行的模块上没有接入控制冒险。这就导致了除法器在不应该执行的时候反复执行。

### 3.2.6 错误 6

- (1) 错误现象: axi 功能测试 bfc00000 报错
- (2) 分析定位过程: 报错之后先看报错的波形图:

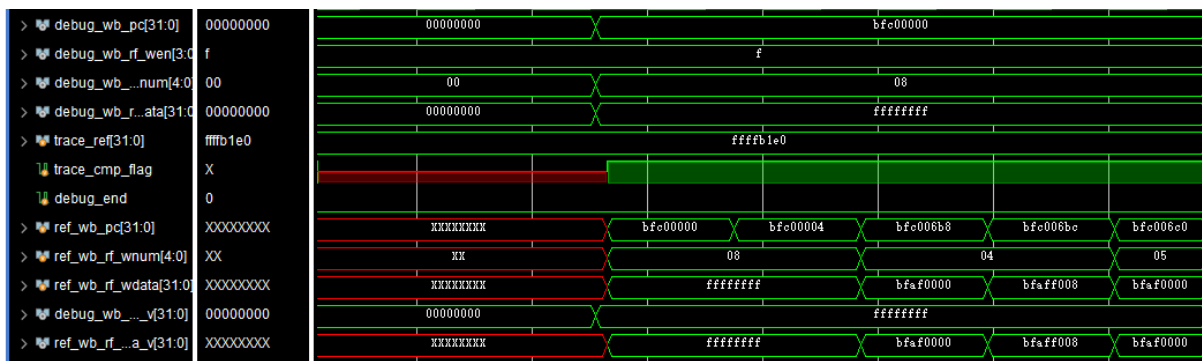


图 41: bfc00000 错误波形图



- (2) 分析定位过程: 再次研究可以发现出问题的原因是 FlushE 信号把 PCE 指令刷新了。
- (3) 错误原因: FlushE 信号把 PCE 指令刷新了。
- (4) 修正效果: 需要修改 hazard 模块, 在 stall\_all 阻塞的时候不能刷新 EX 级信号。成功通过测试。

### 3.2.9 错误 9

- (1) 错误现象:axi 功能测试 bfc3a8cc: 0000a812 mflo s5 报错
- (2) 分析定位过程: 先来看错误的波形图:从图中可以看到 hilo 寄存器的值在 EX 级就

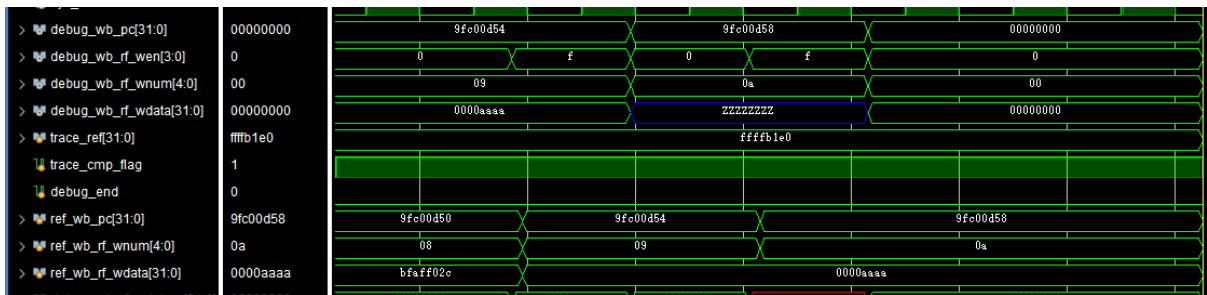


图 43: bfc00d58 错误波形图

可以取到了,之后步步提前了一个周期,导致错误。既然提前了一个周期,那么可以放入流水线寄存器中多存一个周期,就可以得到正确的结果了。

- (4) 修正效果: 可以放入流水线寄存器中多存一个周期, 就可以得到正确的结果了。

### 3.2.10 错误 10

- (1) 错误现象: 上一个问题的解决方法有问题。
- (2) 分析定位过程: 先来看错误的波形图:

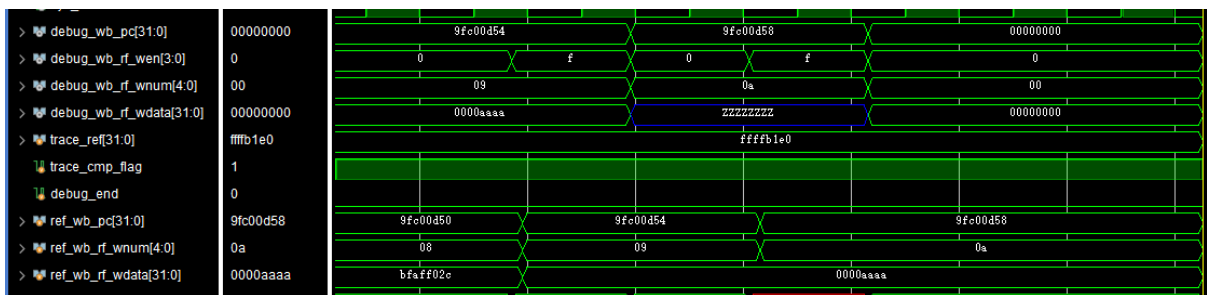


图 44: bfc00d58 错误波形图

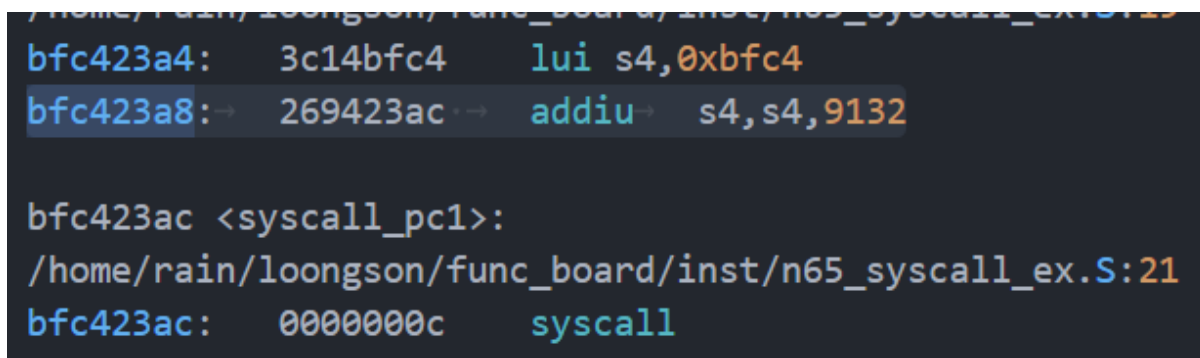
- (3) 错误原因:hilo 寄存器没有被 stall\_all 信号阻塞。
- (4) 修正效果: 正确的解决办法是在全阻塞的时候一同阻塞 hilo 寄存器,这样获得的周期自然是正确的。而不是通过流水线寄存器调整周期。
- 在这里还发现了一个问题,除法器的执行时间是长于其他的普通指令的。这个问题

在我们优化除法器又加入了 AXI 之后其实不应该会出现了,因为访存时间大大增加,这段时间是足够除法器完成运算的。但是由于除法器的信号有一些冲突,导致除法器会反复运算,而且访存结束后也要等最后正在进行的除法完成才会结束。这里存在一定的性能浪费,有待优化。

- (5) 归纳总结(可选): 这个问题的两种解决方法告诉我们,条条大路通罗马,一个问题不只有一种解决方法。如果是按照上一步的解决方法来解决这个问题,那么实际上是把 hilo 寄存器移到了下一级流水线,而如果是添加了阻塞信号,就是把它作为流水线寄存器一级。

### 3.2.11 错误 11

- (1) 错误现象:axi 功能测试 bfc423a8: 269423ac addiu s4,s4,9132 报错  
(2) 分析定位过程:



```

bfc423a4: 3c14bfc4 lui s4,0xbfc4
bfc423a8: 269423ac addiu s4,s4,9132

bfc423ac <syscall_pc1>:
/home/rain/loongson/func_board/inst/n65_syscall_ex.S:21
bfc423ac: 0000000c syscall

```

图 45: bfc423a8 错误波形图

出问题的指令是异常处理的第一条指令,出现异常的原因是在异常处理的时候如果阻塞了信号,会刷新所有流水线寄存器,导致结果指令丢失。

- (3) 错误原因: 在异常处理的时候如果阻塞了信号,会刷新所有流水线寄存器,导致结果指令丢失。  
(4) 修正效果: 在所有的刷新信号上与上 ~stall\_all,这样就可以在阻塞时不刷新流水线寄存器。  
(5) 归纳总结(可选): 这个问题是属于控制冒险的问题,我们在加入了 AXI 总线过后,存储器的取值从单个周期就能取到值,变为了多个周期的握手式的取值,这就导致了我们需要暂停整个 CPU 来等待存储器返回值,而这其中就涉及到了很多时序的电路的阻塞。这要求我们重新考虑 hazard 模块。

### 3.2.12 错误 12

- (1) 错误现象:axi 功能测试 bfc005d0: 401a7000 mfc0 k0,c0\_epc 报错  
(2) 分析定位过程: 先来看错误的波形图:



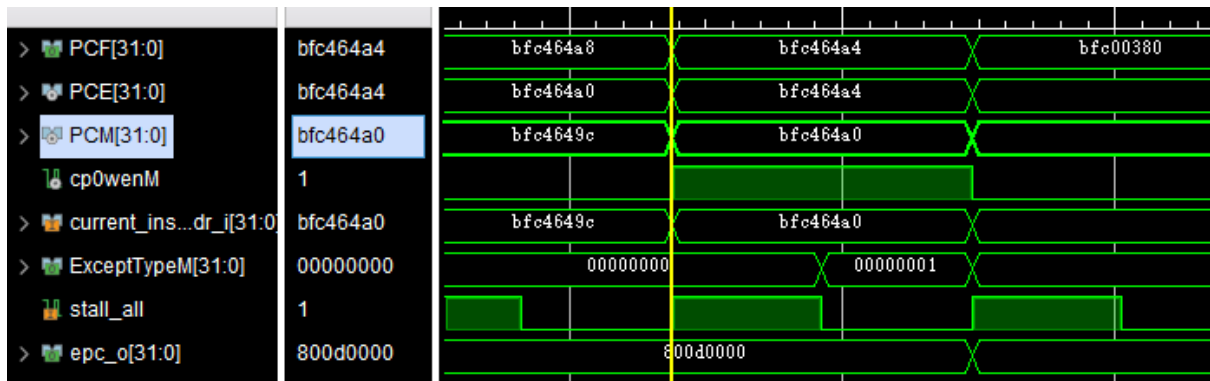


图 46: bfc005d0 错误波形图

波形图中可以看到出现问题的原因是刷新信号提前出现了,在下个周期到来了之前,cp0 已经写入了 PCM 的值,导致需要的 PCM 的值比预期的少 4。这是由于我们的协处理器 cp0 也是时序的, stall\_all 信号并没有暂停 cp0 的工作,现在需要给 cp0 也接入 stall\_all 信号,阻塞它在等待访存数据阶段的写入操作。

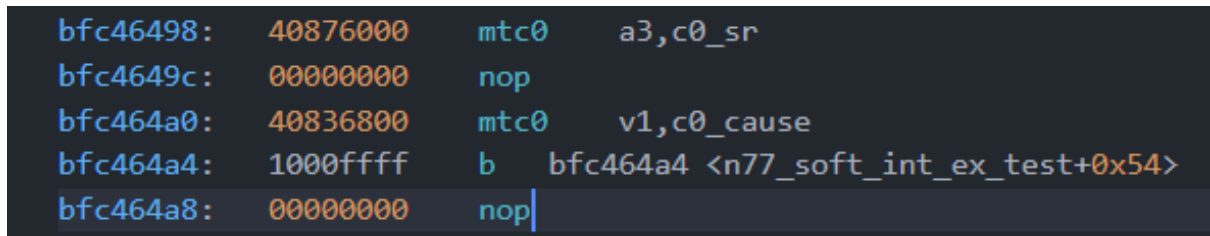


图 47: bfc005d0 相关指令

(3) 错误原因:cp0 没有接入控制冒险。

(4) 修正效果: 我们的 cp0 分为读和写,其中只有写是时序的,读是组合的,所以只需要在写阶段加入 stall\_all 信号。修改过后正确的结果如下所示:

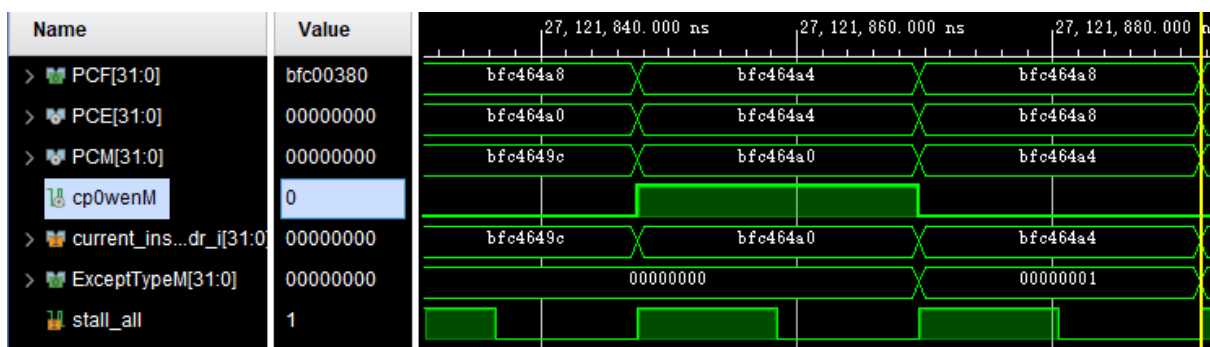


图 48: bfc005d0 正确结果

(5) 归纳总结(可选): 这个问题和 hilo 寄存器,除法器一样,属于是没有接入控制冒险信号,说明我们考虑时考虑的还不够全面。

## 4 设计结果

## 4.1 设计交付物说明

设计交付物分为两次提交,第一次提交代码相关资料,提交时文件的名称为学号-姓名-11-submit.pdf,需要先修改后缀名为.zip 然后解压。解压后可以获得一个以学号-姓名-11-submit 命名的文件夹。文件夹下的目录树如图所示:

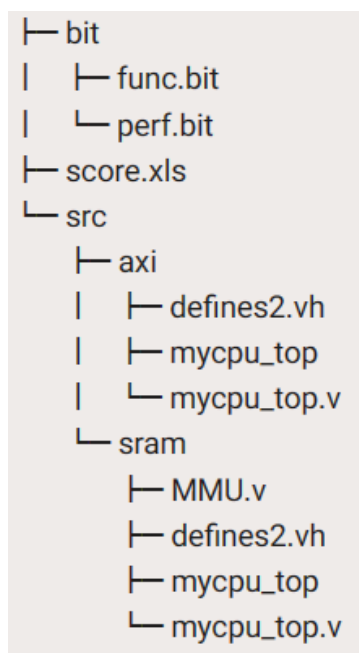


图 49: 提交文件目录树

bit 文件夹下存放了两个.bit 文件,func.bit 是在\vivado\func\_test\_v0.01\soc\_sram\_func 工程下生成的功能测试比特流文件;perf.bit 是在\vivado\perf\_test\_v0.01\soc\_axi\_perf 工程下生成的性能测试比特流文件。如果想要运行上板测试,需要将这两个文件通过 vivado 2019.2 烧录到开发板上运行。

src 文件夹下存放了我们项目工程的源代码,源代码分为功能测试版本和性能测试版本,sram 文件夹存放了对应\vivado\func\_test\_v0.01\soc\_sram\_func 工程的代码版本;axi 文件夹存放了对应\vivado\perf\_test\_v0.01\soc\_axi\_perf 工程的代码版本。如果想要运行源代码需要打开对应工程,通过 vivado 的 Add Directories 分别将这两组源代码加入对应工程,即可进行仿真、综合、实现等一系列操作。

score.xls 文件记录了我们一次最高的性能测试得分。

第二次提交的文件为课程设计报告,命名方式与第一次提交相同。

## 4.2 设计演示结果



### 4.2.1 52 条指令独立测试结果

我们将 52 条指令分为 6 组进行添加,按照添加的顺序,每添加完一组指令我们都会运行对应的独立测试,只有在这一组的独立测试通过后我们才会继续下一组的添加。6 组独立测试没有提供 trace 比对机制,需要我们根据 inst\_rom.S 文件中的注释手动比对波形图,所以在截图时将展示预期结果和实际运行结果。

ArithmeticTest	2023/1/12 0:02	文件夹
DataMoveInstTest	2023/1/12 0:02	文件夹
j_BTest	2023/1/12 0:02	文件夹
LogicInstTest	2023/1/12 0:02	文件夹
S_LInstTest	2023/1/12 0:02	文件夹
ShiftInstTest	2023/1/12 0:02	文件夹

图 50: 6 组独立测试

#### 1. 第一组独立测试:对逻辑运算指令进行测试

```
_start:
    lui    $1,0x0101          ## $1 = 0x01010000
    ori    $1,$1,0x0101       ## S1 = 0x01010101
    ori    $2,$1,0x1100       ## $2 = $1 | 0x1100 = 0x01011101
    or     $1,$1,$2           ## $1 = $1 | $2 = 0x01011101
    andi   $3,$1,0x00fe       ## $3 = $1 & 0x00fe = 0x00000000
    and     $1,$3,$1          ## $1 = $3 & $1 = 0x00000000
    xori   $4,$1,0xff00       ## $4 = $1 ^ 0xff00 = 0x0000ff00
    xor     $1,$4,$1          ## $1 = $4 ^ $1 = 0x0000ff00
    nor    $1,$4,$1          ## $1 = $4 ~^ $1 = 0xffff00ff    nor is "not or"
```

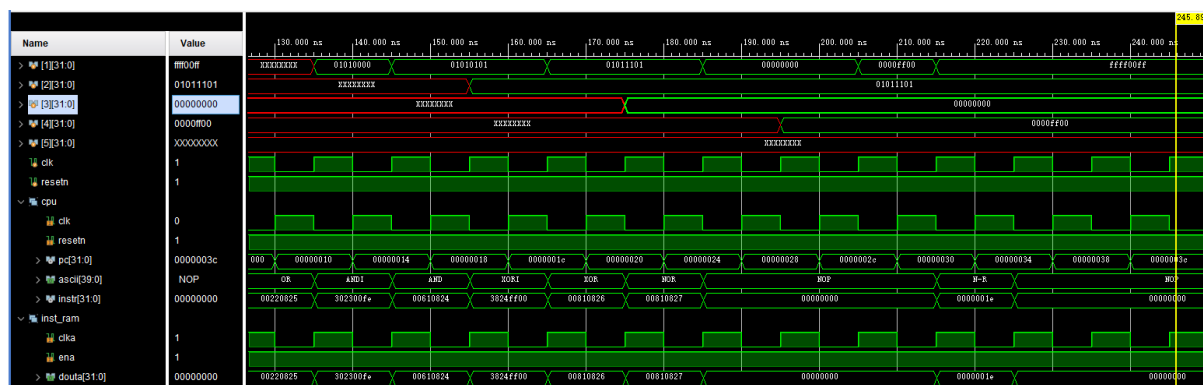


图 51: 逻辑运算指令运行结果

图中写入寄存器堆的值和顺序都与预期结果一致,表明第一组独立测试通过。

#### 2. 第二组独立测试:对移位指令进行测试

```
_start:
    lui    $2,0x0404
    ori    $2,$2,0x0404
    ori    $7,$0,0x7
    ori    $5,$0,0x5
    ori    $8,$0,0x8
    sll    $2,$2,8    ## $2 = 0x04040404    sll 8  = 0x04040400
```

```

sllv $2,$2,$7    ## $2 = 0x04040400 sll 7 = 0x02020000
srl $2,$2,8      ## $2 = 0x02020000 srl 8 = 0x00020200
srlv $2,$2,$5    ## $2 = 0x00020200 srl 5 = 0x00001010
nop              ##
sll $2,$2,19     ## $2 = 0x00001010 sll 19 = 0x80800000
sra $2,$2,16     ## $2 = 0x80800000 sra 16 = 0xffff8080
srav $2,$2,$8    ## $2 = 0xffff8080 sra 8 = 0xffffff80

```

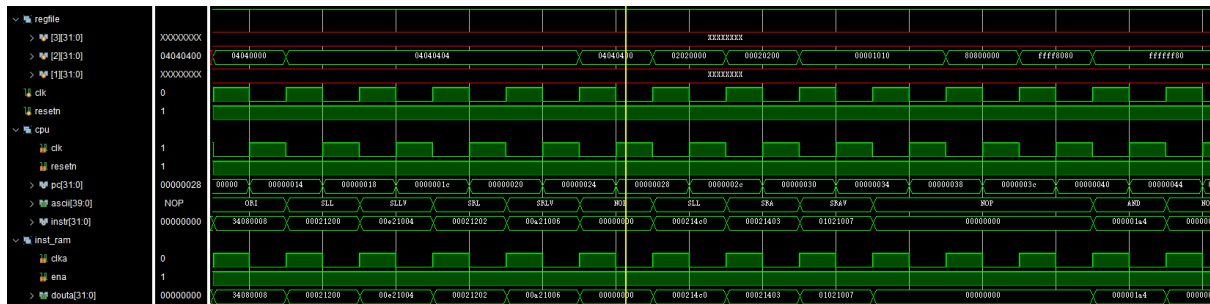


图 52: 移位指令运行结果

图中写入寄存器堆的值和顺序都与预期结果一致,表明第二组独立测试通过。

### 3. 第三组独立测试:对数据移动指令进行测试

```

_start:
    lui $1,0x0000    # $1 = 0x00000000
    lui $2,0xffff    # $2 = 0xffff0000
    lui $3,0x0505    # $3 = 0x05050000
    lui $4,0x0000    # $4 = 0x00000000

    mthi $0          ## hi = 0x00000000
    mthi $2          ## hi = 0xffff0000
    mthi $3          ## hi = 0x05050000
    mfhi $4          ## $4 = 0x05050000

    mtlo $3          ## lo = 0x05050000
    mtlo $2          ## lo = 0xffff0000
    mtlo $1          ## lo = 0x00000000
    mflo $4          ## $4 = 0x00000000

```

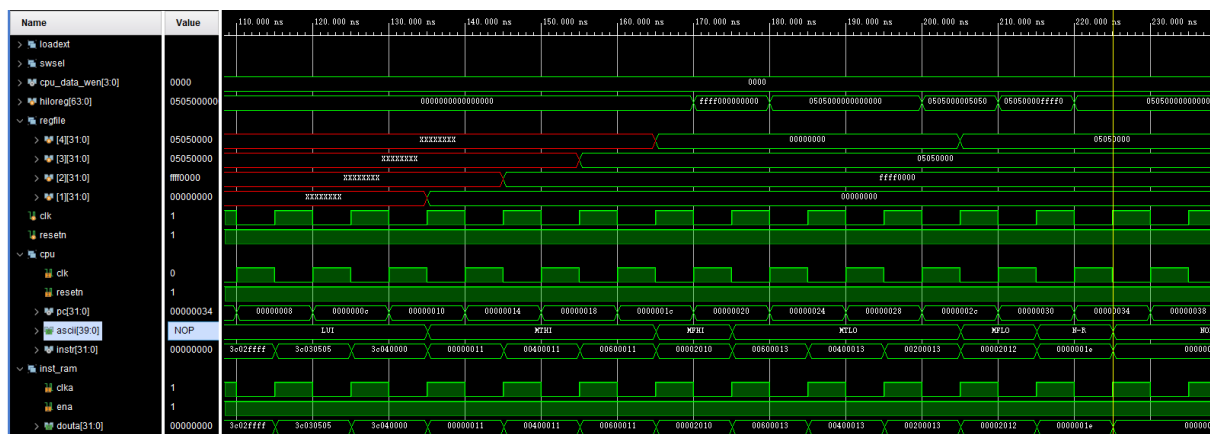


图 53: 数据移动指令运行结果

图中写入寄存器堆和 hilo 寄存器的值及顺序都与预期结果一致,表明第三组独立测试通过。

#### 4. 第四组独立测试:对算术运算指令进行测试

对算术运算的测试比较多,如果只在一致图中展示信号值不能看清,故分为 4 组进行展示,分别是加减法、slt 类指令、乘法和除法。

```

_start:
##### add\addi\addiu\addu\sub\subu #####
ori $1,$0,0x8000    ## $1 = 0x00008000
sll $1,$1,16        ## $1 = 0x80000000
ori $1,$1,0x0010    ## $1 = 0x80000010
ori $2,$0,0x8000    ## $2 = 0x00008000
sll $2,$2,16        ## $2 = 0x80000000
ori $2,$2,0x0001    ## $2 = 0x80000001
ori $3,$0,0x0000    ## $3 = 0x00000000
addu $3,$2,$1        ## $3 = 0x00000011
ori $3,$0,0x0000    ## $3 = 0x00000000
sub $3,$1,$3         ## $3 = 0x80000010
subu $3,$3,$2        ## $3 = 0x0000000F
addi $3,$3,2         ## $3 = 0x00000011
ori $3,$0,0x0000    ## $3 = 0x00000000
addiu $3,$3,0x8000   ## $3 = 0xffff8000
sll $1,$1,1         ## $1 = 0x00000020
add $3,$2,$1         ## $3 = 0x80000021

```

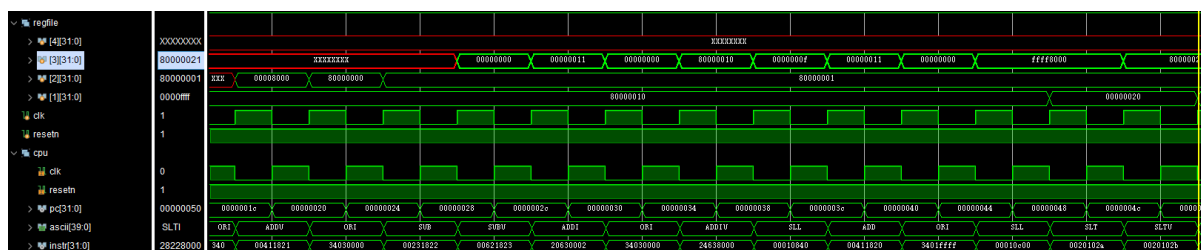


图 54: 加减法运行结果

```

##### slt\sltu\slti\sltiu #####
or $1,$0,0xffff     ## $1 = 0x0000ffff
sll $1,$1,16        ## $1 = 0xffff0000
slt $2,$1,$0         ## $2 = 0x00000001
sltu $2,$1,$0       ## $2 = 0x00000000
slti $2,$1,0x8000   ## $2 = 0x00000001
sltiu $2,$1,0x8000  ## $2 = 0x00000001

```

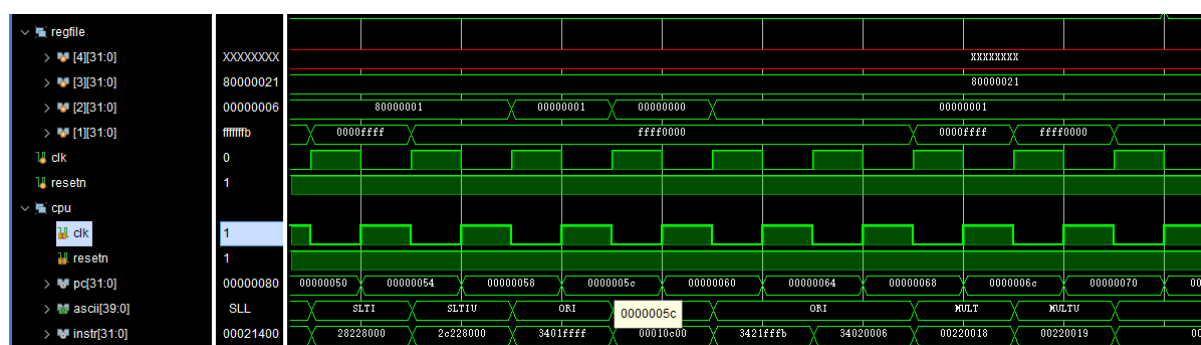


图 55: slt 类指令运行结果

```

##### mult\multu #####
ori $1,$0,0xffff    ## $1 = -5
sll $1,$1,16        ## $1 = -5
ori $1,$1,0xffffb   ## $1 = -5
ori $2,$0,6         ## $2 = 6
mult $1,$2          ## hi = 0xffffffff
                    ## lo = 0xffffffe2
multu $1,$2         ## hi = 0x00000005
                    ## lo = 0xffffffe2

```

nop  
nop

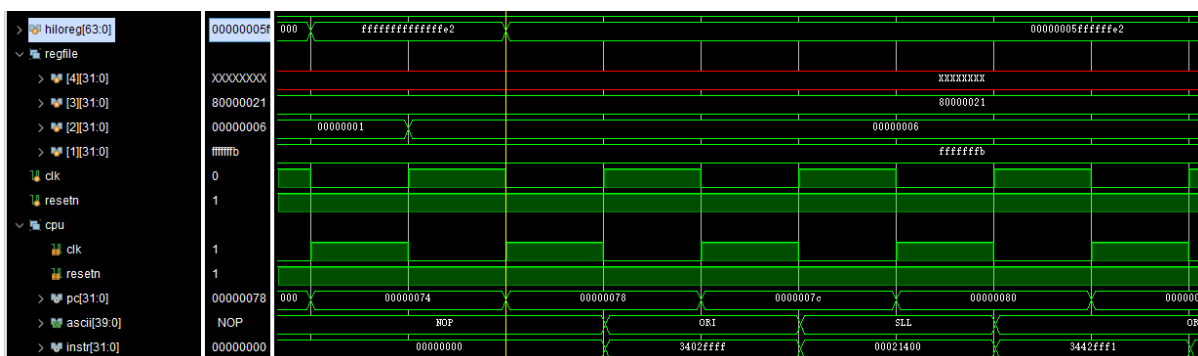


图 56: 乘法运行结果

```
##### div\divu #####
ori $2,$0,0xffff
sll $2,$2,16
ori $2,$2,0xfff1      ## $2 = -15
ori $3,$0,0x11        ## $3 = 17
div $zero,$2,$3       ## hi = 0xffffffff
                        ## lo = 0x00000000
divu $zero,$2,$3      ## hi = 0x00000003
                        ## lo = 0x0f0f0f0e
div $zero,$3,$2       ## hi = 0x00000002
                        ## lo = 0xffffffff
```

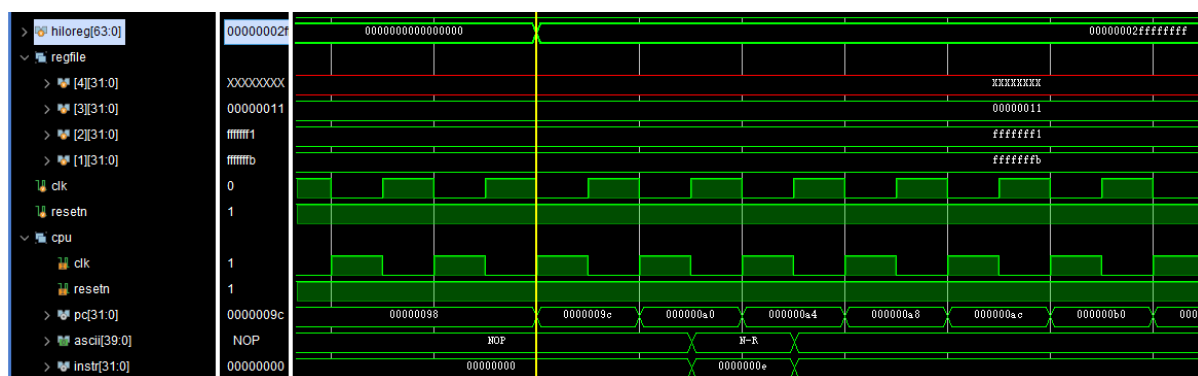


图 57: 除法运行结果

图中写入寄存器堆和 hilo 寄存器的值及顺序都与预期结果一致,表明第四组独立测试通过。

## 5. 第五组独立测试:对跳转指令进行测试

跳转指令的测试分为两类,分别是 J 型指令和 B 型指令。下面依次展示两类指令的测试代码和测试结果。

```
_start:
    addiu $1,$0,0x0001    ## $1 = 0x1
    j      0x20
    addiu $1,$1,0x0001    ## $1 = 0x2
    addiu $1,$1,0x1111
    addiu $1,$1,0x1100

    .org 0x20
    addiu $1,$1,0x0001    ## $1 = 0x3
```

```

jal 0x40          ## 这一条指令会把PC+8的值放入$31中(0x2c)
nop
addiu $1,$1,0x0001 ## r1 = 0x4   addr = 0x2c
addiu $1,$1,0x0001 ## r1 = 0x5
j 0x60
nop

.org 0x40
jalr $2,$31      ## 这一条指令会把$31寄存器的值作为下一跳的地址，然后把PC+8的值
                放入$2中
or $3,$2,$0      ## $3 = 0xb0000048
addiu $1,$1,0x0001 ## $1 = 0x8   addr = 0x48
addiu $1,$1,0x0001 ## $1 = 0x9
addiu $1,$1,0x0001 ## $1 = 0xa
j 0x80
nop

.org 0x60
addiu $1,$1,0x0001 ## $1 = 0x6
jr $3
addiu $1,$1,0x0001 ## $1 = 0x7
addiu $1,$1,0x1111
addiu $1,$1,0x1100

.org 0x80
nop

_loop:
j _loop
nop

```

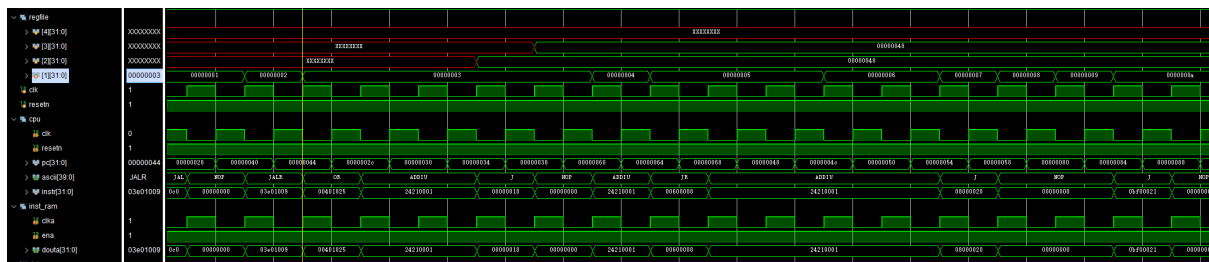


图 58: J 型指令运行结果

```

_start:
ori $3,$0,0x8000
sll $3,16          # $3 = 0x80000000
ori $1,$0,0x0001   ## $1 = 0x1
b sec1
ori $1,$0,0x0002   ## $1 = 0x2
1:
ori $1,$0,0x1111
ori $1,$0,0x1100

.org 0x20
sec1:
ori $1,$0,0x0003   ## $1 = 0x3
bal s2
ori $1,$0,0x1100
ori $1,$0,0x1111
bne $1,$0,s3
nop
ori $1,$0,0x1100
ori $1,$0,0x1111

.org 0x50
s2:
ori $1,$0,0x0004   ## $1 = 0x4
beq $3,$3,s3
or $1,$31,$0       ## $1 = 0x2c
ori $1,$0,0x1111
ori $1,$0,0x1100

```



```

_start:
    ori $3,$0,0xeeff
    sb $3,0x3($0)      # [0x3] = 0xff
    srl $3,$3,8
    sb $3,0x2($0)      # [0x2] = 0xee
    ori $3,$0,0xccdd
    sb $3,0x1($0)      # [0x1] = 0xdd
    srl $3,$3,8
    sb $3,0x0($0)      # [0x0] = 0xcc
    lb $1,0x3($0)      ## $1 = 0xffffffff
    lbu $1,0x2($0)     ## $1 = 0x000000ee
    nop

    ori $3,$0,0xaabb
    sh $3,0x4($0)      # [0x4] = 0xaa, [0x5] = 0xbb
    lhu $1,0x4($0)     ## $1 = 0x0000aabb
    lh $1,0x4($0)      ## $1 = 0xffffaabb

    ori $3,$0,0x8899
    sh $3,0x6($0)      # [0x6] = 0x88, [0x7] = 0x99
    lh $1,0x6($0)     ## $1 = 0xffff8899
    lhu $1,0x6($0)    ## $1 = 0x00008899

    ori $3,$0,0x4455
    sll $3,$3,0x10
    ori $3,$3,0x6677
    sw $3,0x8($0)      # [0x8] = 0x44, [0x9] = 0x55, [0xa] = 0x66, [0xb] = 0x77
    lw $1,0x8($0)     ## $1 = 0x44556677

_loop:
    j _loop
    nop

```

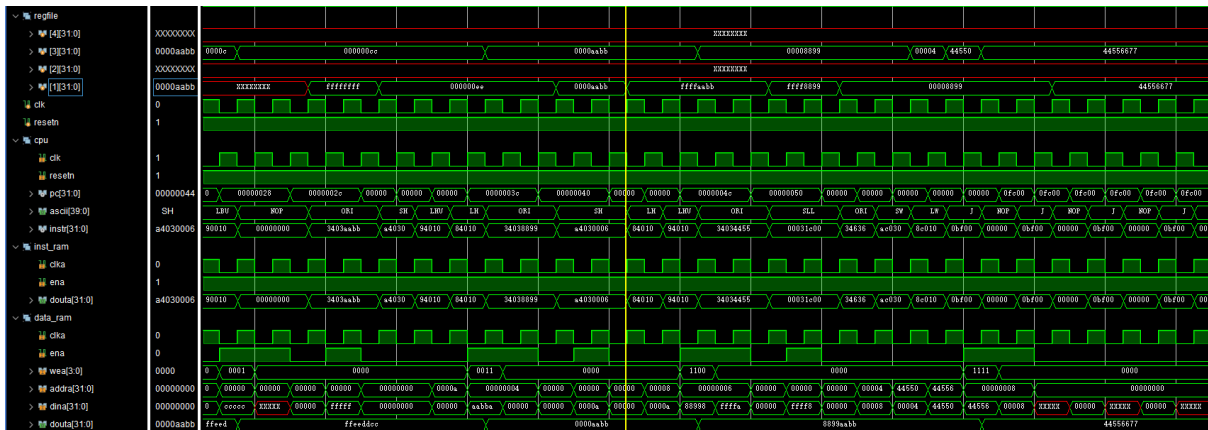


图 60: 访存指令运行结果

图中写入寄存器堆的值及顺序都与预期结果一致, data\_ram 的数据与访存预期一致, 表明第六组测试通过。

#### 4.2.2 52 条指令功能测试结果

我们选择在扩展 52 条指令之后就连接 sram-soc 运行功能测试, 由于我们没有完成 57 条指令, 所以只能运行前 64 个测试点, 下面是 sram 功能测试的前 64 个测试点通过截图。从截图可以看到我们通过了前 64 个测试点, 在第 65 个测试点的 0xbfc00380 出错, 这是第一条调用 syscall 的指令的位置, 由于 52 条指令的 CPU 还没有实现任何和异常处理相关的指令, 所以无法跳转到这一地址进行异常处理。至此我们完成了 52 条指令的扩展和

```

——[1120795 ns] Number 8' d61 Functional Test Point PASS!!!
    [1122000 ns] Test is running, debug_wb_pc = 0xbfc42604
    [1132000 ns] Test is running, debug_wb_pc = 0xbfc43468
    [1142000 ns] Test is running, debug_wb_pc = 0xbfc44284
——[1143195 ns] Number 8' d62 Functional Test Point PASS!!!
    [1152000 ns] Test is running, debug_wb_pc = 0xbfc625b0
    [1162000 ns] Test is running, debug_wb_pc = 0xbfc633b8
    [1172000 ns] Test is running, debug_wb_pc = 0xbfc64130
——[1173685 ns] Number 8' d63 Functional Test Point PASS!!!
    [1182000 ns] Test is running, debug_wb_pc = 0x00000000
    [1192000 ns] Test is running, debug_wb_pc = 0xbfc0ead4
    [1202000 ns] Test is running, debug_wb_pc = 0x00000000
——[1202325 ns] Number 8' d64 Functional Test Point PASS!!!

[1202787 ns] Error!!!
    reference: PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00004000
    mycpu    : PC = 0xbfc424a4, wb_rf_wnum = 0x09, wb_rf_wdata = 0x41000000

[1202805 ns] Error( 0)!!! Occurred in number 8' d65 Functional Test Point!

$finish called at time : 1202827 ns : File "E:/Program/Verilog/vivado/func_test_v0.01/soc_sram_func/testbench/mycpu_tb.v" Line 152
run: Time (s): cpu = 00:00:15 ; elapsed = 00:00:14 . Memory (MB): peak = 1086.277 ; gain = 10.594

```

图 61: sram 功能测试前 64 个测试点通过

sram-soc 的连接。

#### 4.2.3 57 条指令 sram 功能测试结果

在上一阶段 52 条指令连接 sram 通过前 64 个功能测试点的基础上我们进一步完善了 CPU, 添加了 5 条特权指令和自陷指令, 然后再次运行功能测试, 这次通过了全部的 89 个功能测试点。

```

Tcl Console x Messages Log
[1302000 ns] Test is running, debug_wb_pc = 0xbfc004dc
——[1310085 ns] Number 8' d86 Functional Test Point PASS!!!
    [1312000 ns] Test is running, debug_wb_pc = 0xbfc00680
    [1322000 ns] Test is running, debug_wb_pc = 0xbfc529f8
——[1325035 ns] Number 8' d87 Functional Test Point PASS!!!
    [1332000 ns] Test is running, debug_wb_pc = 0xbfc00678
——[1339995 ns] Number 8' d88 Functional Test Point PASS!!!
    [1342000 ns] Test is running, debug_wb_pc = 0xbfc006a4
    [1352000 ns] Test is running, debug_wb_pc = 0xbfc00380
——[1354945 ns] Number 8' d89 Functional Test Point PASS!!!

Test end!
——PASS!!!

$finish called at time : 1355605 ns : File "E:/Program/Verilog/vivado/func_test_v0.01/soc_sram_func/testbench/mycpu_tb.v" Line 253
run: Time (s): cpu = 00:00:18 ; elapsed = 00:00:16 . Memory (MB): peak = 1127.141 ; gain = 0.000

```

图 62: sram 功能测试 89 个测试点通过



#### 4.2.4 57 条指令 axi 功能测试结果

在 sram 通过功能测试过后,我们通过类 sram 接口连接到 axi 总线。从波形图来看两者的一大区别就是访存时间大大增加,每一条指令 stall 的时间占了大部分,这点从一条指令执行的时钟周期数可以看出。

```
——[28271965 ns] Number 8'd88 Functional Test Point PASS!!!
[28272000 ns] Test is running, debug_wb_pc = 0xbfc22580
[28282000 ns] Test is running, debug_wb_pc = 0xbfc00410
[28292000 ns] Test is running, debug_wb_pc = 0x00000000
[28302000 ns] Test is running, debug_wb_pc = 0xbfc15360
[28312000 ns] Test is running, debug_wb_pc = 0x00000000
[28322000 ns] Test is running, debug_wb_pc = 0xbfc153d0
[28332000 ns] Test is running, debug_wb_pc = 0x00000000
[28342000 ns] Test is running, debug_wb_pc = 0xbfc1544c
[28352000 ns] Test is running, debug_wb_pc = 0x00000000
[28362000 ns] Test is running, debug_wb_pc = 0xbfc154d8
[28372000 ns] Test is running, debug_wb_pc = 0x00000000
[28382000 ns] Test is running, debug_wb_pc = 0x00000000
[28392000 ns] Test is running, debug_wb_pc = 0x00000000
——[28395305 ns] Number 8'd89 Functional Test Point PASS!!!
[28402000 ns] Test is running, debug_wb_pc = 0x00000000

Test end!
——PASS!!!
$finish called at time : 28406874500 ps : File "E:/Program/Verilog/vivado/func_test_v0.01/soc_axi_func_sim/testbench/mycpu_tb.v" Line 269
) run: Time (s): cpu = 00:00:43 ; elapsed = 00:01:31 . Memory (MB): peak = 1124.184 ; gain = 28.812
```

图 63: axi 功能测试 89 个测试点通过

由于仿真时间增加,vivado 运行功能测试的时间也大大增加,对比以上两张图,sram 运行功能测试的时间是 1354945ns,而 axi 运行功能测试的时间是 28402000ns,后者是前者的 20 倍,调试的成本大大增加。

所以之后我们开始使用 verilator 进行调试。

#### 4.2.5 axi 性能测试仿真结果

性能测试共有 10 个测试点,下面依次展示每个测试点通过的结果。

#### 4.2.6 axi 性能测试上板结果

在我们的疏忽下我们忘记拍摄开发板结果,只记录了开发板上运行性能测试获得的性能分数,获得的分数如图所示。

不过我们在最后答辩时已经由助教验收了我们的上板情况。

## 5 参考设计说明

1. 实验的整体进展方向和介绍参考了重庆大学硬件综合设计实验文档  
<https://co.ccslab.cn/>。

```

1hx@DESKTOP-MJUK20P:/mnt/e/Program/Verilog/verilator/axi$ ./obj_dir/Vmycpu_top -perfdiff -prog 1 -uart
set performance test program to 1
bitcount test begin.
Bit counter algorithm benchmark

Bit counter algorithm benchmark

Bit counter algorithm benchmark

Bit counter algorithm benchmark

Bit counter algorithm benchmark

Bit counter algorithm benchmark

Bit counter algorithm benchmark

Bit counter algorithm benchmark

Bit counter algorithm benchmark

bitcount PASS!Bits: 811
bitcount: Total Count(SoC count) = 0xaba00
bitcount: Total Count(CPU count) = 0x27fea
27fea
total ticks = 1450491

```

图 64: 性能测试 1 号测试点通过

```

1hx@DESKTOP-MJUK20P:/mnt/e/Program/Verilog/verilator/axi$ ./obj_dir/Vmycpu_top -perfdiff -prog 2 -uart
set performance test program to 2
bubble sort test begin.
bubble sort PASS!
bubble sort: Total Count(SoC count) = 0x386805
bubble sort: Total Count(CPU count) = 0xed0b2
ed0b2
total ticks = 7436563

```

图 65: 性能测试 2 号测试点通过

```

Total ticks      : 0
Total time (secs): 0
Iterations       : 1
Compiler version : GCC4.3.0
Compiler flags   :
Memory location  : Please put data memory location here
                   (e.g. code in flash, data on heap etc)
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xe714
Correct operation validated. See readme.txt for run and reporting rules.
coremark PASS!
coremark: Total Count(SoC count) = 0xb8f803
coremark: Total Count(CPU count) = 0x22f2d9
22f2d9
total ticks = 24281371

```

图 66: 性能测试 3 号测试点通过

```

198
199
200
ffffffff
end
1601645211, 00000200
crc32 PASS!
crc32: Total Count(SoC count) = 0x6dee03
crc32: Total Count(CPU count) = 0x183e02
183e02
total ticks = 14442209

```

图 67: 性能测试 4 号测试点通过

```

Enum_Loc:          1
    should be:      1
Str_1_Loc:          DHRYSTONE PROGRAM, 1'ST STRING
    should be:      DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:          DHRYSTONE PROGRAM, 2'ND STRING
    should be:      DHRYSTONE PROGRAM, 2'ND STRING

Begin ns: 14937120
End ns: 15124300
Total ns: 187180
Dhrystones per ms:          53

dhrystone PASS!
dhrystone: Total Count(SoC count) = 0x18fc52
dhrystone: Total Count(CPU count) = 0x4436e
4436e
total ticks = 3312093

```

图 68: 性能测试 5 号测试点通过

```

1hx@DESKTOP-MJUK20P:/mnt/e/Program/Verilog/verilator/axi$ ./obj_dir/Vmycpu_top -perfdiff -prog 6 -uart
set performance test program to 6
quick sort test begin.
quick sort PASS!
quick sort: Total Count(SoC count) = 0x589a15
quick sort: Total Count(CPU count) = 0xf172f
f172f
total ticks = 11659851

```

图 69: 性能测试 6 号测试点通过

```

1hx@DESKTOP-MJUK20P:/mnt/e/Program/Verilog/verilator/axi$ ./obj_dir/Vmycpu_top -perfdiff -prog 7 -uart
set performance test program to 7
select sort test begin.
select sort PASS!
select sort: Total Count(SoC count) = 0x3a7ee5
select sort: Total Count(CPU count) = 0xf80ac
f80ac
total ticks = 7713805

```

图 70: 性能测试 7 号测试点通过

```

437358104 2057077515 2988414705 3742976831 2079096471
437358104 : 437358104
2057077515 : 2057077515
2988414705 : 2988414705
3742976831 : 3742976831
2079096471 : 2079096471
sha PASS!
sha: Total Count(SoC count) = 0x3d79c0
sha: Total Count(CPU count) = 0xf1326
f1326
total ticks = 8092783

```

图 71: 性能测试 8 号测试点通过

```

lhx@DESKTOP-MJUK20P:/mnt/e/Program/Verilog/verilator/axi$ ./obj_dir/Vmycpu_top -perfdiff -prog 9 -uart
set performance test program to 9
stream copy test begin.
stream copy PASS!
stream copy: Total Count(SoC count) = 0x16df9e
stream copy: Total Count(CPU count) = 0xf587
f587
total ticks = 3044879

```

图 72: 性能测试 9 号测试点通过

```

"guide" is not in "and it will recommend guiding principles"
"regard" is in "in this regard. The University's" ["regard. The University's"]
"officers" is not in "Executive Officers and I will then decide"
"implement" is in "whether and how to implement such" ["implement such"]
"principalities" is not in "principles."
string search PASS!
string search: Total Count(SoC count) = 0x353f5b
string search: Total Count(CPU count) = 0x9fcef
9fcef
total ticks = 7022295

```

图 73: 性能测试 10 号测试点通过

序号	测试程序	myCPU	gs132	T <sub>gs132</sub> /T <sub>mycpu</sub>
		上板计时(16进制)	上板(16进制)	
		数码管显示	数码管显示	
cpu_clk : sys_clk		50MHz : 100MHz	50MHz : 100MHz	-
1	bitcount	f9a99	13CF7FA	20.31342917
2	bubble_sort	531b27	7BDD47E	23.84699691
3	coremark	128e405	10CE6772	14.49154205
4	crc32	a4b1f7	AA1AA5C	16.52547176
5	dhrystone	22877c	1FC00D8	14.71226581
6	quick_sort	985c07	719615A	11.92823878
7	select_sort	55e1c4	6E0009A	20.49328685
8	sha	766b85	74B8B20	15.7705074
9	stream_copy	28cf6f	853B00	3.264629509
10	stringsearch	4e0a43	50A1BCC	16.53135801

性能分	14.290
-----	--------

图 74: 性能测试上板结果

2. lab4 的数据通路和 hazard 模块参考了计算机组成与结构课程的教材《计算机组成与设计》[6] 以及参考资料包中的数据通路图 [3]。
3. 设计 MIPS 52 条指令和 57 条指令时参考了资料包中的 MIPS 指令系统规范 [5] 和指令及对应机器码 [4]。
4. 设计新的数据通路和冒险模块时参考了《自己动手写 CPU》[14]。
5. 宏定义参考了资料中提供的 defines2.vh 文件。
6. 基础的除法器模块调用了资料包中的 div.v 文件,后面优化的除法器待添加。
7. hilo 寄存器参考了资料包中提供的 hilo\_reg.v 文件,不过后面根据我们自己设计的需要重新实现了 hilo 寄存器。
8. cp0 协处理器参考了资料包中提供的 cp0\_reg.v 文件,但是我们发现参考代码的实现过于冗余,而且部分逻辑有误,所以我们重写了 cp0 模块。
9. 地址映射模块 mmu 使用了资料包中提供的 mmu.v 文件。
10. 类 sram 接口设计参考了计算机组成与结构课程实验 5 的资料包 cache\_lab\_v0.06.rar 中的 sram\_to\_sram\_like 转接口,使用了龙芯提供的 axi\_interface 相关文件。
11. cache 设计使用了上学期计算机组成与结构课程实验 5 时设计的 4 路组相联写回 cache,并对其做出了部分优化,参考了资料 Cache 实验补充说明 [2] 和 Cache 实验指导书 [1],设计 LRU 算法时参考了《超标量处理器设计》[11] 和 [7]。
12. 快速除法器设计参考了以下资料: [8], [13] [12] [9] [10]

## 6 现场添加指令和答辩记录

## 6.1 现场添加指令

本次添加指令我们小组抽取到的题目是 func\_6 添加一条 RELU 指令。指令的具体要求如下图所示：

### RELU

31	26	25	21	20	16	15	11	10	6	5	0
111111	rs	0	rd	00000	000000						
6	5	5	5	5	6						

汇编格式: RELU rd, rt

功能描述: 由 op 段和 func 段判断指令, 判断 rs 寄存器的值是否大于 0, 若大于等于 0 则将原值写入 rd 寄存器中, 若小于 0 则向 rd 寄存器中写入 32 位 0。

操作定义:  $\text{tmp} \leftarrow \text{RELU}(\text{GPR}[\text{rs}])$

$\text{GPR}[\text{rd}] \leftarrow \text{tmp}_{31..0}$

图 75: 现场添加指令详情

### 6.1.1 分工情况

由于添加的指令比较简单, 而我们最终的代码版本是... 的代码, 所以由... 动手添加, ... 和... 一起分析指令结构, 确定需要修改的部分。

### 6.1.2 完成情况

我们在早上 10:00 领取到指令开始添加, 10:26 分添加完成指令并跑通 axi 仿真测试。添加过程中我们首先分析指令, 发现这一条指令既不是 R 型指令 (没有两个操作数, op 字段也不是 6'b000000), 也不是 I 型指令 (没有立即数字段), 更不是 J 型指令 (没有跳转地址), 所以它基本上不会与我们现有指令的编码冲突, 我们先在 define2.vh 文件中为这条指令分配了一个 op 字段的宏定义, 如下图所示:

```

`define R_TYPE 6'b000000
`define REGIMM_INST 6'b000001
`define SPECIAL3_INST 6'b010000
//change the SPECIAL2_INST from 6'b011100 to 6'b010000
`define RELU 6'b111111
`define MTC0 5'b00100
`define MFC0 5'b00000

```

图 76: 添加 op 字段宏定义

因为这一条指令需要 ALU 参与运算,所以还需要添加 ALU 控制信号,如图所示:

```

`define MULT_CONTROL      5'b11000
`define MULTU_CONTROL    5'b11001
`define DIV_CONTROL      5'b11010
`define DIVU_CONTROL     5'b11011

`define RELU_CONTROL      5'b11111

```

图 77: 添加 ALU 控制信号

然后在 Maindec 和 Aludec 中进行对应译码。

```

`SPECIAL3_INST:
case (rs)
  `MFC0: Controls <= 20'b10010_00_000_00_00_00_000_0;
  `MTC0: Controls <= 20'b00000_00_000_00_00_00_000_1;
  default: Controls <= 20'b00000_00_000_00_00_00_001_0;
endcase
`RELU: Controls <= 20'b11000_00_000_00_00_00_000_0;
default: Controls <= 20'b00000_00_000_00_00_00_001_0;
endcase

```

图 78: 修改 Maindec

```

`ANDI: ALUControl <= `AND_CONTROL;
`ORI: ALUControl <= `OR_CONTROL;
`XORI: ALUControl <= `XOR_CONTROL;
`LUI: ALUControl <= `LUI_CONTROL;
`ADDI: ALUControl <= `ADD_CONTROL;
`ADDIU: ALUControl <= `ADDU_CONTROL;
`SLTI: ALUControl <= `SLT_CONTROL;
`SLTIU: ALUControl <= `SLTU_CONTROL;
`SW, `SH, `SB, `LW, `LH, `LHU, `LB, `LBU: ALUControl <= `ADDU_CONTROL;
`RELU: ALUControl <= `RELU_CONTROL;
default: ALUControl <= `NO_CONTROL;
endcase

```

图 79: 修改 Aludec

最后还需要在 ALU 中执行对应的操作:

```

`SLLV_CONTROL: ALUResult <= B << A[4:0];
`SRLV_CONTROL: ALUResult <= B >> A[4:0];
`SRAV_CONTROL: ALUResult <= (B >> A[4:0]) | ({32{B[31]}} << (6'd32 - {1'b0, A[4:0]}));
`ADD_CONTROL, `ADDU_CONTROL, `SUB_CONTROL, `SUBU_CONTROL: ALUResult <= tmp[31:0];
`SLT_CONTROL: ALUResult <= (opposite) ? A[31] : tmp[31];
`SLTU_CONTROL: ALUResult <= (opposite) ? B[31] : tmp[31];
`RELU_CONTROL: ALUResult <= A[31] ? 32'b0 : A;
default: ALUResult <= {32{1'b0}};
endcase

```

图 80: 修改 ALU 模块

这样就完成了指令的添加。

下面是我们仿真运行的结果图:

```

Tcl Console x Messages Log
[8612000 ns] Test is running, debug_wb_pc = 0xbfc04a90
[8622000 ns] Test is running, debug_wb_pc = 0xbfc04b68
----[8630745 ns] Number 8'd18 Functional Test Point PASS!!!
[8632000 ns] Test is running, debug_wb_pc = 0xbfc008f4
[8642000 ns] Test is running, debug_wb_pc = 0xbfc0a6fc
[8652000 ns] Test is running, debug_wb_pc = 0xbfc0a7cc
[8662000 ns] Test is running, debug_wb_pc = 0xbfc0a890
[8672000 ns] Test is running, debug_wb_pc = 0xbfc0a960
[8682000 ns] Test is running, debug_wb_pc = 0xbfc0aa2c
----[8683225 ns] Number 8'd19 Functional Test Point PASS!!!
[8692000 ns] Test is running, debug_wb_pc = 0xbfc24eb0
[8702000 ns] Test is running, debug_wb_pc = 0x00000000
[8712000 ns] Test is running, debug_wb_pc = 0xbfc25044
[8722000 ns] Test is running, debug_wb_pc = 0xbfc25110
[8732000 ns] Test is running, debug_wb_pc = 0xbfc251d8
----[8735745 ns] Number 8'd20 Functional Test Point PASS!!!
[8742000 ns] Test is running, debug_wb_pc = 0xbfc00980
=====
Test end!
---PASS!!!
$finish called at time : 8742493500 ps : File "E:/Hardware/CO-lab-material-CQU-2022/vivado/func_tes
run: Time (s): cpu = 00:00:44 ; elapsed = 00:01:38 . Memory (MB): peak = 1301.324 ; gain = 18.121

```

图 81: 仿真通过



## 6.2 现场答辩记录

现场回答由两人共同完成, 不作区分。

### 6.2.1 问题 1:bram 实现和实验指导书中提供的 cache 写法的区别

实验指导书中的 cache 采用 reg 型变量实现, 使用的是开发板上的 LUT 资源模拟实现。FPGA 中提供了专门用于实现 ram 的 bram 资源 (Block RAM), 可以使用 Block Memory Generator 提供的 ip 核生成各种类型的 ram。采用 bram 实现的 cache 容量大, 性能好, 但缺点在于读数据时, 无法像 dram 那样是组合逻辑, 需要至少一个时钟的延迟。

### 6.2.2 问题 2: 两路组相联 LRU 策略

对于两路组相联的 lru, 每组只需要寄存器类型的 1 个 bit 位, 记录最后一次访问的那一路即可, 则另一路即为替换目标。

### 6.2.3 问题 3:1X2 桥的作用

mmu 根据虚拟地址的高位判断数据访问请求访问的对象是内存还是外设, 并生成控制信号 no\_dcach。1X2 桥根据上述控制信号对数据访问请求进行区分, 接入数据 cache 或者直接接入 2X1 的桥。

### 6.2.4 问题 4: 乘法器的优化策略

vivado 自带的乘法实现时序较好, 因此只需要将其执行时间延长到 2-3 个时钟周期, 便可以大幅降低这部分的关键路径长度。

### 6.2.5 问题 5: 除法优化的必要性和除法指令占比

除法指令在全部指令中大约只占到 4% 左右, 降低其运算周期数对 CPU 性能的提升幅度很小, 同时优化后变得复杂的组合逻辑可能会成为关键路径。

### 6.2.6 问题 6: 各个异常分别是什么

表 6: 异常类型

异常名称	产生的流水级	异常描述	异常原因
Int	MEM	软硬件中断	mtc0 指令产生的软件中断或外部硬件产生的硬件中断
AdEL	IF 或 MEM	读指令或读数据地址异常	地址低位与访存指令类型不匹配
AdES	MEM	写数据地址异常	地址低位与访存指令类型不匹配
Sys	ID	系统调用	Syscall 指令
Bp	ID	断点	Break 指令
Remainder	ID	保留指令	主控制单元无法识别的指令
Ov	EX	算术溢出	有符号加减法结果溢出
Eret	EX	异常处理返回	Eret 指令

### 6.2.7 问题 7:cp0 延迟槽怎么处理

对于延迟槽指令,EPC 取 PC-4, 否则取 PC。

## 7 总结(可选)

### 7.1 课程成果总结

1. 成功添加了 52 条指令,通过了 6 组独立测试。
2. 成功将 CPU52 连接 sram-soc 进行功能测试,通过了前 64 个测试点。
3. 成功添加了异常处理相关的 5 条指令拓展至 57 条指令,通过了 89 个功能测试点。
4. 成功完成类 sram 接口连接 axi 总线,并通过了 89 个功能测试点。
5. 成功通过 axi 的 10 个性能测试仿真。
6. 成功优化 cache,实现了四路组相联写回 cache。
7. 成功优化除法器,在不考虑时钟周期长短的情况下将除法器的运行时间缩短至 8 个周期。
8. 成功优化关键路径,将性能得分提升至 14 分左右。

### 7.2 课程心得体会

这次我们小组成功地设计了一个基于经典五级流水线可以上板运行 57 条 MIPS 指令并通过 axi 总线连接了 4 路组相联 cache 的 CPU。通过对 MIPS 简单五级流水线的深入研究,我们对 CPU 与外部设备通过总线交互的过程有了清晰的认识。这不仅提升了我们的硬件设计能力,而且让我们能够以硬件的角度思考问题。在使用 Verilog 语言设计电路和 vivado 进行调试的过程中,我们的技能也更加熟练。同时还认识了 vivado 之外的调试工具 verilator。

在这次课程设计中我们综合运用了数字逻辑、计算机组成原理课程中学到的理论知识。通过亲自实践,我们对这些理论的理解更加深入。同时,我们还学到了当时课堂之外的新的知识,特别是对处理器进行精确异常处理的每一个步骤有了清晰的理解。

这门课程带给我们的收获并不仅限于知识本身,还在于我们在团队合作和解决问题的过程中培养了良好的沟通能力。我们学会了有效地分工合作,充分发挥每个成员的专长,使得整个项目顺利推进。在面对挑战时,我们能够共同探讨解决方案,不断优化设计,最终取得了成功。

## 8 供同学们吐槽之用。有什么问题都可以直接写在这。

### 8.1

.....

### 8.2

(1) 真别优化除法器。指令占比少,还卡关键路径。加上 Cache 之后一次 Implementation 要跑半小时,关键路径优化的一头雾水,效率很低。给出的 WNS 的 Timing Summary 完全看不懂,到底为啥寄存器堆会连出一条电路到 MEM\_WB 级的流水线,然后又连到 PC 寄存器上呢? 为啥除法器里声明的变量会出现在流水线寄存器的内部电路里呢? 诸如此类的问题真是让人摸不着头脑,还搜不到任何资料。啥时候才能有国产替代呀 =\_=

(2) 没能赶在答辩之前实现 bram 版本的 cache, 结果助教老师说把四路组相联的三维数组拆成四个二维的资源占用就会好很多,真是反人类的特性。能写循环不让用循环,非要展开写。虽然理论上确实可能对硬件比较友好,但作为一个编译器还是显得太古板了; 于是也没来得及写 AXI 的 burst 传输,毕竟让人感觉写了也上不去板 =\_= 感觉性能测试分数还大有可为,真的很遗憾。

(3) 助教老师们都很尽职尽责。回答问题很快,讲得也很清楚。有些问题回头来看真是问的自己都看不下去 =\_= 在此对助教老师们表示由衷的感谢!

## 9 参考文献

### 参考文献

- [1] Cache 实验指导书.pdf.
- [2] Cache 实验补充说明.pdf.
- [3] 原始数据通路图\_2018.pdf.

- [4] 指令及对应机器码 \_2018.pdf.
- [5] "系统能力培养大赛" MIPS 指令系统规范 \_v1.01.pdf.
- [6] John L.Hennessy David A.Patterson. *Computer Organization and Design*. 机械工业出版社, 2015.
- [7] [美]John L. Hennessy. 计算机体系结构-量化研究方法 5th-中文. 人民邮电出版社, 2021.
- [8] [日]Yamin Li. *Computer Principles and Design in Verilog HDL*. 清华大学出版社, 2015.
- [9] 何婷婷; 彭元喜; 雷元武. 基于 *Goldschmidt* 算法的高性能双精度浮点除法器设计. 计算机应用, 2015.
- [10] 华东. *SRT* 除法器及其算法的研究. 计算机工程与设计, 2007.
- [11] 姚永斌. 超标量处理器设计. 清华大学出版社, 2014.
- [12] 朱建银; 沈海斌. 高性能单精度除法器的实现. 微电子学与计算机, 2007.
- [13] 王县; 倪晓强; 邢座程. 浮点除法算法的分析与研究. 第十二届计算机工程与工艺学术年会, 2008.
- [14] 雷思磊. 自己动手写 *CPU*. 电子工业出版社, 2014.