

重庆大学课程设计报告

课程设计题目： MIPS SOC

学 院： 计算机学院

专业班级： 计算机卓越 1、2 班

年 级： 2020 级

姓 名： 陈仕广 吴卓宣

学 号： 20204327 20204343

完成时间： 2023 年 1 月 13 日

成 绩：

指导教师： 钟将

重庆大学教务处制

综合设计指导教师评定成绩表

项目	分值	优秀 (100>x≥90)	良好 (90>x≥80)	中等 (80>x≥70)	及格 (70>x≥60)	不及格 (x<60)	评分
		参考标准	参考标准	参考标准	参考标准	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：

年 月 日

MIPS SOC 设计报告

组员 1 陈仕广、组员 2 吴卓宣

一、设计简介

本次硬件综合实验课程，我们所提交的是一个实现了 57 条指令、连接了 AXI 接口、实现了基础 cache 的 5 级流水线 MIPS 处理器。此设计能跑通功能测试 89 个测试点、以及 10 个性能测试，仿真、上板均通过。我们组基于计算机组成原理的 lab4：简单 MIPS 5 级流水线 CPU 开始，采用分模块的方式逐步添加指令，最终成功完成设计。本次设计完成了硬件综合设计的基本要求，并有所扩展，所实现的处理器效果较好。

（一）小组分工说明

陈仕广：

乘除法与数据移动指令；访存指令；从 52 条指令扩展到 57 条指令；AXI 接口和基础 Cache；共同进行指令单独测试、功能调试。共同书写文档。

吴卓宣：

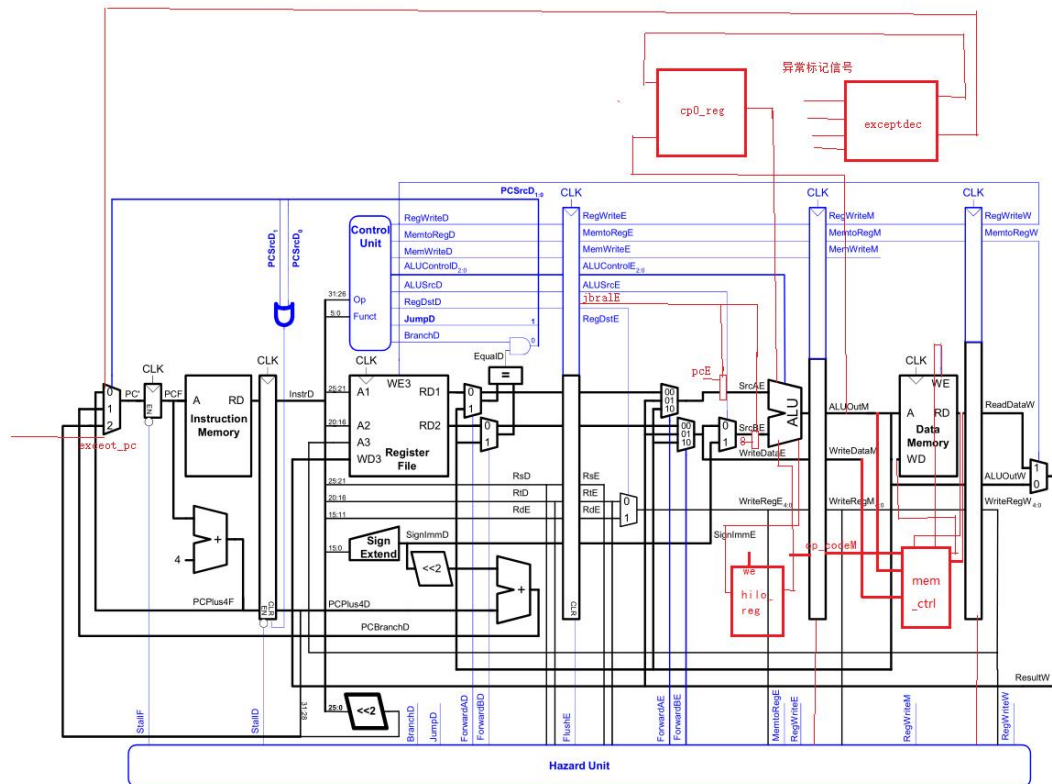
除乘法外的所有运算指令；分支跳转指令；aludec 和 maindec；连接 SRAM-SOC；共同进行指令单独测试、功能调试。共同书写文档。

二、设计方案（30%）

（一）总体设计思路

总体设计思路是先实现 mips 处理器，再向外实现 AXI 接口。5 级流水线 MIPS CPU 共有取指、译码、执行、访存、回写五个阶段，处理器的工作通过从指令存储器中取值，通过译码产生指令的控制信号，用于控制指令按规定的的数据通路执行，执行阶段进行运算、访存阶段访问数据存储器、回写阶段写入寄存器堆。

整体通路图如下：



为实现 57 条指令，需要完成的设计有控制信号的设计、数据通路的设计、处理数据冒险等。我们采用对指令分类，分模块逐步实现的方式，具体的设计路详细描述如下：

1. 8 条逻辑运算指令

1.1 指令机器码概览

	31:26	25:21	20:16	15:11	10:6	5:0	
逻辑运算指令	000000	rs	rt	rd	00000	100100	and rd,rs,rt
	000000	rs	rt	rd	00000	100101	or rd,rs,rt
	000000	rs	rt	rd	00000	100110	xor rd,rs,rt
	000000	rs	rt	rd	00000	100111	nor rd,rs,rt
	001100	rs	rt	immediate			andi rt,rs,immediate
	001110	rs	rt	immediate			xori rt,rs,immediate
	001111	00000	rt	immediate			lui rt,immediate
	001101	rs	rt	immediate			ori rs,rt,immediate

1.2 立即数 ANDI ORI XORI LUI

1.2.1 指令功能

ANDI ORI XORI: 寄存器 `rs` 中的值与 0 扩展至 32 位的立即数 `imm` 按位逻辑运算，结果写入寄存器 `rt` 中。

LUI: 将 16 位立即数 `imm` 写入寄存器 `rt` 的高 16 位，寄存器 `rt` 的低 16 位置 0。

1.2.2 数据通路调整

52 条指令中仅 ANDI ORI XORI 这三条指令是 0 扩展，其他需要扩展的均为有符号扩展（LUI 指令如何扩展都不影响结果），所以根据这三条指令的 `op_code` 的中间两位，传入符号扩展模块 `signext`，用于判断是否 0 扩展。

001100	rs	rt	immediate	andi rt,rs,immediate
001110	rs	rt	immediate	xori rt,rs,immediate
001111	00000	rt	immediate	lui rt,immediate
001101	rs	rt	immediate	ori rs,rt,immediate

`signext` 模块代码:

```
module signext(  
    input wire[15:0] a,  
    input wire[1:0] inst_type,  
    output wire[31:0] y  
);  
  
    assign y = (inst_type == 2'b11)? {{16{1'b0}}},a : {{16{a[15]}}},a;  
endmodule
```

然后需在 ALU 中处理运算。

1.2.3 控制信号调整

数据通路与计组 Lab4 相同，无需调整，只需扩展 `alucontrol` 信号的种类，用于识别并控制这些指令在 ALU 中执行何种操作。

1.3 普通逻辑运算 AND OR XOR NOR

1.3.1 指令功能

寄存器 `rs` 中的值与寄存器 `rt` 中的值按位逻辑运算，结果写入寄存器 `rd` 中。

1.3.2 数据通路调整

数据通路与计组 Lab4 相同，无需调整，只需在 ALU 中对应处理运算即可。

1.3.3 控制信号调整

数据通路与计组 Lab4 相同，无需调整，只需扩展 `alucontrol` 信号的种类，用于识别并控制这些指令在 ALU 中执行何种操作。

2. 6 条移位运算指令

2.1 指令机器码概览

移位指令	000000	00000	rt	rd	sa	000000	sll rd,rt,sa
	000000	00000	rt	rd	sa	000010	srl rd,rt,sa
	000000	00000	rt	rd	sa	000011	sra rd,rt,sa
	000000	rs	rt	rd	00000	000100	slv rd,rt,rs
	000000	rs	rt	rd	00000	000110	srld rd,rt,rs
	000000	rs	rt	rd	00000	000111	srad rd,rt,rs

2.2 立即数移位运算 SLL SRL SRA

2.2.1 指令功能

SLL SRL：由立即数 `sa` 指定移位量，对寄存器 `rt` 的值进行逻辑移位，结果写入寄存器 `rd` 中。

SRA：由立即数 `sa` 指定移位量，对寄存器 `rt` 的值进行算术右移（即左边补上符号位），结果写入寄存器 `rd` 中。

2.2.2 数据通路调整

新增 `sa` 信号，从指令的[10:6]位直接连入 ALU 中。需在 ALU 中对应处理运算。

2.2.3 控制信号调整

数据通路与计组 Lab4 相同，无需调整，只需扩展 alucontrol 信号的种类，用于识别并控制这些指令在 ALU 中执行何种操作。

2.3 变量移位运算 SLLV SRLV SRAV

2.3.1 指令功能

由寄存器 rs 中的值指定移位量，对寄存器 rt 的值进行逻辑移位，结果写入寄存器 rd 中。

2.3.2 数据通路调整

数据通路与计组 Lab4 相同，无需调整。只需在 ALU 中对应处理运算。

2.3.3 控制信号调整

数据通路与计组 Lab4 相同，无需调整，只需扩展 alucontrol 信号的种类，用于识别并控制这些指令在 ALU 中执行何种操作。

3. 4 条数据移动指令

3.1 指令机器码概览

数据移动指令	000000	00000	00000	rd	00000	010000	mfhi rd
	000000	00000	00000	rd	00000	010010	mflo rd
	000000	rs	00000	00000	00000	010001	mthi rs
	000000	rs	00000	00000	00000	010011	mtlo rs

3.2 数据移动指令实现方案

3.2.1 指令功能

指令名称格式	指令功能简述
MFHI rd	HI 寄存器→通用寄存器 rd
MFLO rd	LO 寄存器→通用寄存器 rd
MTHI rs	通用寄存器 rs→HI 寄存器

MTLO rs	通用寄存器 rs→L0 寄存器
---------	-----------------

MFXX 类型将 HI、L0 寄存器中的值写入到通用寄存器，MTXX 类型将通用寄存器中的值写入到 HI、L0 寄存器中。

3.2.2 数据通路调整

从寄存器堆或 HI、L0 寄存器中读出数据后，沿着流水线向后传递，中间不需要做额外的运算，最终写入对应的寄存器中。需要在计组实验四的基础上增加 HI、L0 寄存器模块 **hilo_reg**。需要在寄存器堆和 HI、L0 寄存器之间新增数据通路。

hilo_reg 模块如下图：

```
module hilo_reg(
    input wire clk,rst,
    input wire we, //写使能
    input wire[63:0] hilo_in, //写入值
    output wire[63:0] hilo_out //读出值
);
    reg [63:0] hilo_reg; //HILO寄存器
    always @(negedge clk) begin
        if(rst)begin
            hilo_reg <= 0;
        end
        else if(we) begin
            hilo_reg <= hilo_in;
        end
    end
    assign hilo_out = hilo_reg;
endmodule
```

3.2.3 控制信号调整

新增 HI、L0 寄存器的写信号 **hilo_write**。

3.2.4 具体做法

指令 MTHI、MTLO：在执行阶段（EX）写 HI、L0 寄存器，复用 ALU，ALU 的输出新增 64 位的 hilo_out 信号，接到 HI、L0 寄存器的输入端口，此时 HI、L0 寄存器的写控制信号是拉高的。在时钟下降沿，写入 HI、L0 寄存器。

指令 MFHI、MFLO：在回写阶段（WB）写入寄存器堆，ALU 新增 64 位的 hilo_in

输入，更改 ALU 内逻辑，复用 ALU 的 32 位输出信号作为写入寄存器堆的值。

数据前推：考虑如 MTHI-->MFHI 的执行顺序（RAW 写后读），MTHI 执行到 EX 阶段时，MFHI 到译码阶段（它在 EX 阶段才需要读 HI、LO 寄存器）。可见在 EX 阶段写 HI、LO 寄存器，无需数据前推。

4. 14 条算术运算指令

4.1 指令机器码概览

算术运算指令	000000	rs	rt	rd	00000	100000	add rd,rs,rt
	000000	rs	rt	rd	00000	100001	addu rd,rs,rt
	000000	rs	rt	rd	00000	100010	sub rd,rs,rt
	000000	rs	rt	rd	00000	100011	subu rd,rs,rt
	000000	rs	rt	rd	00000	101010	slt rd,rs,rt
	000000	rs	rt	rd	00000	101011	sltu rd,rs,rt
	000000	rs	rt	00000	00000	011000	mult rs,rt
	000000	rs	rt	00000	00000	011001	multu rs,rt
	000000	rs	rt	00000	00000	011010	div rs,rt
	000000	rs	rt	00000	00000	011011	divu rs,rt
	001000	rs	rt	immediate			addi rt,rs,immediate
	001001	rs	rt	immediate			addiu rt,rs,immediate
	001010	rs	rt	immediate			slti rt,rs,immediate
	001011	rs	rt	immediate			sltiu rt,rs,immediate

4.2 指令功能

指令名称格式	指令功能简述
ADD rd, rs, rt	加（可产生溢出例外）
ADDU rd, rs, rt	加（不产生溢出例外）
ADDI rt, rs, immediate	加立即数（可产生溢出例外）
ADDIU rt, rs, immeidate	加立即数（不产生溢出例外）
SUB rd, rs, rt	减（可产生溢出例外）
SUBU rd, rs, rt	减（不产生溢出例外）
ST rd, rs, rt	有符号小于置 1
SLTU rd, rs, rt	无符号小于设置 1
SLTI rt, rs, immediate	有符号小于立即数设置 1
SLTIU rt, rs, immediate	无符号小于立即数口设置 1

DIV rs, rt	有符号字除
DIVU rs, rt	无符号字除
MULT rs, rt	有符号字乘
MULTU rs, rt	无符号字乘

4.3 算数运算指令实现方案

4.3.1 数据通路调整

将这 14 条指令两两分组，ADD 和 ADDU、ADDI 和 ADDIU、SUB 和 SUBU、SLT 和 SLTU、SLTI 和 SLTIU、DIV 和 DIVU、MULT 和 MULTU；每一组中的两条指令的功能是完全一样的，唯一的区别在于操作数是看成有符号还是无符号。

将 ALU 看成一个黑盒，那么不同组指令执行的操作是完全一样的，都是从寄存器或立即数中取得源操作数，送入 ALU 中进行运算，然后将结果写回 rd 或 rt 寄存器中，所以区别仅在于 ALU 内部执行的操作。

4.3.2 控制信号调整

数据通路与计组 Lab4 相同，无需调整，只需扩展 alucontrol 信号的种类，用于识别并控制这些指令在 ALU 中执行何种操作。

4.4 乘法具体实现

在 ALU 中，使用了 Verilog 中的乘号：

```
`MULT_CONTROL : hilo_out = $signed(a) * $signed(b); //指令MULT
`MULTU_CONTROL : hilo_out = {32'b0, a} * {32'b0, b}; //指令MULTU
```

乘法结果：复用数据移动指令中添加的 hilo_out 信号，在时钟下降沿时，rs*rt 的低 32 位和高 32 位分别写入 L0 寄存器和 HI 寄存器。（意味着半个时钟周期内要算出乘法结果）

4.5 除法具体实现

在 ALU 中，接入《自己动手写 CPU》中的除法器。除法器接口：

信号	方向	类型	位宽	作用
clk	input	wire	1	时钟信号
rst	input	wire	1	复位信号
signed_div_i	input	wire	1	是否为有符号除法 (1 为有符号)
opdata1_i	input	wire	32	被除数
opdata2_i	input	wire	32	除数
start_i	input	wire	1	是否开始除法
annul_i	input	wire	1	是否结束除法
result_o	output	reg	64	除法结果
ready_o	output	reg	1	除法运算是否结束

4.5.1 数据通路调整

ALU 内部接入 div.v 模块，使用状态机控制除法模块。所以 clk、rst 信号也要引入 ALU 中。

对于除法结果：复用数据移动指令中添加的 hilo_out 信号，rs/rt 的值存入 L0 寄存器，rs%rt 的值存入 HI 寄存器。

4.5.2 控制信号调整

完善 hilo_reg 模块在除法指令时的写信号 hilo_writeE 的逻辑：在除法没完成时，hilo_writeE 应该拉低，在除法完成时（信号 div_ready==1），拉高 hilo_writeE。

因为除法要 36 个时钟周期，所以要进行流水线暂停。新增 stallE 控制信号，控制 译码-->执行阶段 的流水线暂停，在除法进行时，拉高 stallE。（当然 stallF、stallD 也拉高）。

stallE 控制信号经由 hazard 模块生成，要接入 ID-->EX 阶段的流水线寄存器中（stallE 取反后接到使能端），包括控制信号和数据信号。

5. 12 条分支跳转指令

5.1 指令机器码概览

分支跳转指令	000000	rs	00000	00000	00000	001000	jr rs
	000000	rs	00000	rd	00000	001001	jalr rs/jalr rd,rs
	000010	instr_index					j target
	000011	instr_index					jal target
	000100	rs	rt	offset			beq rs,rt,offset
	000111	rs	00000	offset			bgtz rs,offset
	000110	rs	00000	offset			blez rs,offset
	000101	rs	rt	offset			bne rs,rt,offset
	000001	rs	00000	offset			bltz rs,offset
	000001	rs	10000	offset			bltzal rs,offset
	000001	rs	00001	offset			bgez rs,offset
	000001	rs	10001	offset			bgezal rs,offset

5.2 指令功能

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

5.3 BNE、BGEZ、BGTZ、BLEZ、BLTZ 指令

5.3.1 数据通路调整

这 5 条指令功能和 BEQ 类似，区别在于它们判断是否跳转的方式不一样。只需要对生成是否跳转信号的逻辑进行扩展即可，其余功能可直接复用 BEQ 的通路。

扩展后的 eqcmp.v 模块如下：

```
module eqcmp(  
    input wire [31:0] a,b,  
    input wire [5:0] opD,  
    input wire [4:0] rtD,  
    output reg y  
);  
always@(*) begin  
    case(opD)  
        `BEQ : y = (a == b);  
        `BNE : y = (a != b);  
        `BGTZ : y = ((a[31] == 1'b0) & (a != `ZeroWord));  
        `BLEZ : y = ((a[31] == 1'b1) | (a == `ZeroWord));  
        `REGIMM_INST : begin  
            case(rtD)  
                `BGEZ,`BGEZAL : y = (a[31] == 1'b0);  
                `BLTZ,`BLTZAL : y = (a[31] == 1'b1);  
                default : y = 1'b0;  
            endcase  
        end  
        default : y = 1'b0;  
    endcase  
end  
endmodule
```

5.3.2 控制信号调整

与计组 Lab4 相同，只需正确生成各指令相关的控制信号。

5.4 JR 指令

5.4.1 数据通路调整

JR 指令的功能与 J 指令完全相同，都是无条件跳转到目标地址，区别在于 JR 指令的跳转目标地址来自通用寄存器堆的第 rs 项。所以数据通路只需新增寄存器堆的 rdata1 端口连接到生成下一个 PC 的多选器的输入。

5.4.2 控制信号调整

添加新的控制信号 jr，用于控制下一个 PC 的选择。其余相关的控制信号能够正确对应的产生即可。

```
mux2 #(32) pc_jr_mux(pcnextFD,srca2D,jrD,pcnextjrD);
```

5.5 JAL 指令

5.5.1 数据通路调整

JAL 指令分为跳转操作和 Link 操作。跳转操作可以复用 J 指令的数据通路，而 Link 操作需要增加新的数据通路。写寄存器涉及到写端口的地址和写数据，对于写地址，需要调整通用寄存器堆写端口的地址输入 waddr 的生成逻辑，增加一个固定数值 31 作为新的输入，即将二路选择器改为三路选择器。而对于写数据，需要得到 PC+8，可以复用 ALU 的加法器，只是第一个源操作数需要添加一个二路选择器，使其可以选到 PC，第二个源操作数需要再添加一个二路选择器，使其可以选到 8。

```
mux3 #(5) wrmux(rtE,rdE,5'd31,regdstE,writeregE);
```

5.5.2 控制信号调整

由于新增了 ALU 两个源操作数的多选器，所以需要添加控制信号 jbral，使得 JAL 指令执行时这两个多选器可以选择到 PC 和 8。其余相关的控制信号能够正确对应的产生即可

```
//跳转链接类指令,复用ALU,ALU源操作数选择分别为pcE and 8
mux2 #(32) alusrcamux(srca2E,pcE,jbralE,srca3E);
mux2 #(32) alusrcbmux(srcb3E,32'h00000008,jbralE,srcb4E);
```

5.6 JALR 指令

5.6.1 数据通路调整

JALR 指令可以看作 JR 和 JAL 的结合。对于跳转操作，可以复用 JR 指令的数据通路。对于 Link 操作，可以复用 JAL 指令的数据通路。

5.6.2 控制信号调整

对于 Link 操作，相比于 JAL 指令，JALR 不是写入 31 号寄存器，而是第 rd 项寄存器，所以调整控制信号 regdst。并复用 JR 指令新增的控制信号 jr 和 JAL 指令新增的控制信号 jbral。其余相关的控制信号能够正确对应的产生即可。

5.7 BLTZAL 和 BGEZAL 指令

5.7.1 数据通路调整

这两条指令可以看作是 BLTZ、BGEZ 指令与 JAL 指令的结合。对于跳转操作，可以复用 BLTZ、BGEZ 指令的数据通路。对于 Link 操作，可以复用 JAL 指令的数据通路。

5.7.2 控制信号调整

复用 JAL 指令新增的控制信号 jbral，其余相关的控制信号能够正确对应的产生即可。

6. 8 条访存指令

6.1 指令机器码概览

访存指令	100000	base	rt	offset	lb rt,offset(base)
	100100	base	rt	offset	lbu rt,offset(base)
	100001	base	rt	offset	lh rt,offset(base)
	100101	base	rt	offset	lhu rt,offset(base)
	100011	base	rt	offset	lw rt,offset(base)
	101000	base	rt	offset	sb rt,offset(base)
	101001	base	rt	offset	sh rt,offset(base)
	101011	base	rt	offset	sw rt,offset(base)

6.2 指令功能

指令名称格式	指令功能简述
LB rt, offset(base)	根据 (base+offset) 地址, 取字节, 进行有符号扩展, 写入 rt 号寄存器
LBU rt, offset(base)	根据 (base+offset) 地址, 取字节, 进行无符号扩展, 写入 rt 号寄存器
LH rt, offset (base)	根据 (base+offset) 地址, 取半字, 进行有符号扩展, 写入 rt 号寄存器
LHU rt, offset (base)	根据 (base+offset) 地址, 取半字, 进行无符号扩展, 写入 rt 号寄存器
LW rt, offset(base)	根据 (base+offset) 地址, 取字, 写入 rt 号寄存器
SB rt, offset(base)	根据 (base+offset) 地址, 把 rt 号寄存器的值的最低字节, 写入数据存储器
SH rt, offset(base)	根据 (base+offset) 地址, 把 rt 号寄存器的值的低半字, 写入数据存储器
SW rt, offset (base)	根据 (base+offset) 地址, 把 rt 号寄存器的值, 写入数据存储器

6.3 Load 类指令

6.3.1 数据通路调整

根据这些指令的功能, LB、LBU、LH、LHU 这四条指令在译码、执行、写回阶段的数据通路和控制逻辑可以复用 LW 指令的实现。

Load 类指令每次从数据存储器取出一个字, 然后再根据指令类型处理出最后要回写的值: 对于 LB、LBU 指令, 需要根据偏移量选择 1 个字节, LH、LHU 指令需要选择低 2 字节或高 2 字节。

上面选择出来的内容可能是 8 位或 16 位, 需要扩展到 32 位才能写回到寄存

器中。LB、LH 进行有符号扩展，LBU、LHU 进行无符号扩展。

所以访存阶段添加一个 `mem_ctrl.v` 模块，在此模块内根据指令类型对从数据存储器读出的字做处理。又因为在此模块内要判断指令类型，所以需要把 `op_code` 信号传递到访存阶段。

6.3.2 控制信号调整

数据通路 with 计组 Lab4 相同，无需调整。

6.4 Store 类指令

6.4.1 数据通路调整

根据这三条 Store 指令的功能可知，SB 和 SH 指令可以复用 SW 指令的译码、执行、写回阶段的数据通路。

因为 SB 指令写入寄存器的永远是 `rt` 寄存器的最低字节，SH 指令则是低 2 字节，因此需要先扩展成 32 位的数据，再传递给数据 RAM。所以在新增的 `mem_ctrl` 模块进行扩展处理。

扩展逻辑：因为字节写使能中 0 对应的字节不会被写入，所以是写数据中这个字节是什么值不会对结果造成影响，只要保证字节写使能和写数据可以对应上即可。

6.4.2 控制信号调整

三条 Store 指令写入的宽度是不同的，所以需要控制哪些字节需要写入。故新增位宽为 4 的控制信号字节写使能 `mem_wen`，这个控制信号直接由 `mem_ctrl` 模块产生即可。

字节写使能 `mem_wen` 的确定逻辑：实现 SB 和 SH 指令的关键在于如何在 4 字节宽的 RAM 上完成字节或半字的写入，这可以通过字节写使能来完成。

字节写使能就是每个字节都有自己对应的写使能，所以 4 字节宽 RAM 的字节写使能就有 4 位，某位为 1 就会写入对应的字节，为 0 则保持不变，例如 `4'b0001` 的字节写使能表示只有最低字节会修改，`4'b1100` 表示高 2 字节会修改。

字节写使能的生成需要由访存操作的类型信息和访存地址中包含的偏移量来共同控制，例如，SB 指令的访存地址最低 2 位如果等于 `0'b00`，则字节写使能

就是 4' b0001。

7. 2 条自陷+3 条特权指令（精确异常）

7.1 指令机器码概览

内陷指令	000000	code				001101	break
	000000	code				001100	syscall
特权指令	31:26	25:21	20:16	15:11	10:3	2:0	mtc0 rt,rd
	010000	00100	rt	rd	00000000	sel	
	010000	00000	rt	rd	00000000	sel	mfc0 rt,rd
	31:26	25	24:6			5:0	eret
	010000	1	0000 0000 0000 0000 000			011000	

7.2 精确异常介绍：

原因：流水线的 IF, ID, EXE, MEM 阶段都可能发送例外，考虑 指令 A-->指令 B 的执行顺序，假如指令 A 在 MEM 访存阶段发生例外，而指令 B 在 ID 译码阶段发生例外，如果一发生例外就立即处理的话，由于指令 B 的例外先发生，此时进入指令 B 的异常处理程序，从而导致指令 A 的例外被吞掉（例外处理时屏蔽其他例外）。

定义：在流水级的 CPU 中，精准的处理最先发生异常或例外的指令。

实现：为了实现精确异常，在 IF, ID, EXE, MEM 检测到的例外，都是只进行标记，并将所有的例外标记信号传递到 MEM 阶段，在 MEM 阶段统一进行处理。

7.3 延迟槽指令检测

发生例外的指令是否处于延迟槽，二者写入 CP0 的内容略有不同，所以要检测指令是否处于延迟槽。检测指令是否处于延迟槽，在 IF（取指令）和 ID（译码）阶段完成。

将 ID 阶段译码后的 jumpD（无条件跳转）、jrD（无条件跳转寄存器）、jbra1D（无条件链接跳转、无条件跳转寄存器并链接、条件分支并链接）、branchD（条件分支）信号作为 IF 阶段的输入，通过判断当前 CPU 的 ID 阶段是否为这些跳转指令，从而判断当前 IF 级的指令是否在延迟槽中。

同时也需要将指令地址 pcF 传递到 MEM 级，因为发生异常后需要保存原本例

外处理完成后继续开始执行的指令的 PC。

数据通路：‘是否在延迟槽中’的信号在 IF 级产生，需要传递到 MEM 级使用。

7.4 BREAK 指令

7.4.1 指令功能

发生断点异常，无条件地将控制权转到异常处理程序。

7.4.2 数据通路调整

新增标记信号 `is_breakD`，并将此标记信号传递到 MEM 阶段。

在译码阶段通过判断指令的高 6 位以及最低 6 位来产生此标记信号。

7.4.3 控制信号调整

与计组 Lab4 相同，无需调整。

7.5 SYSCALL 指令

7.5.1 指令功能

发生系统调用异常，无条件地将控制权转到异常处理程序。

7.5.2 数据通路调整

新增标记信号 `is_syscallD`，并将此标记信号传递到 MEM 阶段。

在译码阶段通过判断指令的高 6 位以及最低 6 位来产生此标记信号。

7.5.3 控制信号调整

与计组 Lab4 相同，无需调整。

7.6 ERET 指令

7.6.1 指令功能

在中断、异常或错误处理完成时返回中断指令。ERET 不执行下一条指令（即，它没有延迟槽）。

7.8.2 数据通路调整

新增读 CP0 寄存器的地址信号 cp0_raddrD（来自指令的 rd 字段），并将此地址信号传递到 EX 阶段。

扩展 alucontrol 信号的种类，用于识别并控制指令在 ALU 中执行何种操作。

7.8.3 控制信号调整

译码阶段通过检测指令的 OP 为是否为 6'b010000 和 rs 是否为 5'b00000，由此产生指令 MFC0 的控制信号。

7.9 MTC0、MFC0 指令注意

在 EX 阶段读取 CP0（复用 ALU 数据通路）。在 MEM 阶段写入 CP0。

数据前推：当 EX 级读 CP0，MEM 级写 CP0 的时候，如果读写的是同一地址 (cp0_raddrE == cp0_waddrM)，产生数据冒险，需要进行数据前推。

7.10 响应例外

7.10.1 新增异常标记信号

除 syscall、break、eret 外，数据通路还需新增一些异常的标记信号（如下表），并传递到 MEM 阶段：

异常类型	标记信号名	产生阶段	产生原因
地址错例外	is_AdEL_pcF	1, 取指阶段	取指或读数据： 1, 取指 PC 不对齐于字边界。 2, 字（半字）Load 指令，其访问地址不对齐于字（半字）边界。
	is_AdEL_dataM	2, 访存阶段	
	is_AdESM	访存阶段	字（半字）Store 指令，其访问地址不对齐于字（半字）边界。
整型溢出例外	is_overflowE	执行阶段	当一条 ADD、ADDI 或 SUB

			指令执行结果溢出时，触发整型溢出例外。
保留指令例外	is_invalidD	译码阶段	当执行一条未实现的指令时，触发保留指令例外。

7. 10. 2 新增 exceptdec.v 模块

为了响应例外，我们需要在 MEM 阶段识别出例外的类型，所以新增 exceptdec.v 模块，通过各种例外标记信号译码出例外类型。

在 exceptdec.v 模块确定 pc 的下一个地址 except_pc。取指阶段新增 2x1 多路选择器 mux2，用于确定下一个 pc 是正常的 pc 还是 except_pc。此 mux2 的控制信号 is_exception 也由 exceptdec.v 模块产生。

7. 10. 3 新增 cp0_reg.v 模块

此模块来自资料包中 ref_code 文件夹里的参考代码。CP0 寄存器模块实现的功能：1、根据例外类型，写相应的 CP0 寄存器；2、完成其他读写 CP0 内寄存器的需求。然后根据 cp0_reg 的接口，增加相应的数据通路。

7. 10. 4 流水线刷新

当一条指令引发例外处理时，其之后的 3 条指令已进入流水线，我们要废弃掉这 3 条指令的执行，所以要刷新 ID、EX、MEM 三阶段的流水线寄存器。

通过信号 is_exception 传入 hazard 模块，辅助判断产生 flushD、flushE、flushM 刷新控制信号。注意，hilo 寄存器的写信号也要在处理例外时冲刷（置 0）。

7. 10. 5 软件中断和硬件中断

软中断要读取 CP0.Cause 寄存器来判断。硬件中断直接由 CPU 外部传入，只需提供数据通路 ext_int 与外部连接。

(二) maindec.v 模块设计

1. 功能描述

该模块对指令进行译码，根据指令的 op、funct、rs 和 rt 识别所有指令，生成各个控制信号。

2. 接口定义

```
module maindec(  
    input wire[5:0] op,  
    input wire[5:0] funct,  
    input wire[4:0] rs,  
    input wire[4:0] rt,  
    output wire memtoreg,memwrite,  
    output wire branch,alusrc,  
    output wire[1:0] regdst,  
    output wire regwrite,  
    output wire jump,  
    output wire hilo_write,  
    output wire jbral,  
    output wire jr,  
    output wire cp0_write,  
    output reg is_invalid  
);
```

信号名	位宽	方向	功能描述
op	6-bit	Input	指令的 opcode
funct	6-bit	Input	指令的 funct
rs	5-bit	Input	指令的 rs
rt	5-bit	Input	指令的 rt
memtoreg	1-bit	Output	回写的数据来自于 ALU 计算的结果还是存储器读取的数据
memwrite	1-bit	Output	是否需要写数据存储器
branch	1-bit	Output	是否为 branch 指令
alusrc	1-bit	Output	送入 ALU B 端口的值是立即数的 32 位扩展还是寄存器堆读取的值
regdst	2-bit	Output	写入寄存器堆的地址是 rt 还是 rd 还是 31

regwrite	1-bit	Output	是否需要写寄存器堆
jump	1-bit	Output	是否为 jump 指令
hilo_write	1-bit	Output	是否需要写 HI、LO 寄存器
jbral	1-bit	Output	是否是 Link 类型指令
jr	1-bit	Output	是否是 jr 指令
cp0_write	1-bit	Output	是否需要写 CP0 寄存器
is_invalid	1-bit	Output	是否是无效指令

3. 逻辑控制

根据 op 区分指令是 R-Type 还是 I-Type 还是 J-Type 还是特权指令。R-Type 指令根据 funct 再作区分；I-Type 指令中的 REGIMM 指令根据 rt 再作区分；特权指令根据 rs 再作区分。然后再为不同指令生成各个控制信号。

```
reg[11:0] controls;
assign {regwrite,regdst,alusrc,branch,memwrite,memtoereg,jump,hilo_write,jbral,jr,cp0_write} = controls;
always @(*) begin
    is_invalid <= 1'b0;
    case (op)
        `R_TYPE:
            case (funct)
                `ADD,`ADDU,`SUB,`SUBU,`SLT,`SLTU,
                `AND,`NOR,`OR,`XOR,
                `SLLV,`SLL,`SRAV,`SRA,`SRLV,`SRL,
                `MFHI,`MFLO: controls <= 12'b1_01_00000000;
                `DIV,`DIVU,`MULT,`MULTU,
                `MTHI,`MTLO: controls <= 12'b0_00_00000100;
                `JR: controls <= 12'b0_00_00000010;
                `JALR: controls <= 12'b1_01_00000110;
                //自陷指令
                `BREAK,`SYSCALL: controls <= 12'b0_00_00000000;
                default: begin
                    controls <= 12'b000000000000;
                    is_invalid <= 1'b1;
                end
            end
        endcase
        // I-type
        `ADDI,`ADDIU,`SLTI,`SLTIU,
        `ANDI,`LUI,`ORI,`XORI: controls <= 12'b1_00_10000000;
        `BEQ,`BNE,`BGTZ,`BLEZ: controls <= 12'b0_00_01000000;
        `REGIMM_INST:
            case (rt)
                `BGEZ,`BLTZ: controls <= 12'b0_00_01000000;
                `BGEZAL,`BLTZAL: controls <= 12'b1_10_01000100;
                default: begin
                    controls <= 12'b000000000000;
                    is_invalid <= 1'b1;
                end
            end
        endcase
    end
end
```



```

`LB,`LBU,`LH,`LHU,`LW:      controls <= 12'b1_00_100100000;
`SB,`SH,`SW:                 controls <= 12'b0_00_101000000;
// J-type
`J:      controls <= 12'b0_00_000010000;
`JAL:    controls <= 12'b1_10_000010100;
//for exception
`SPECIAL3_INST:
    case(rs)
        `MTC0:controls <= 12'b0_00_000000001;
        `MFC0:controls <= 12'b1_00_000000000;
        `ERET:controls <= 12'b0_00_000000000;
        default: begin
            controls <= 12'b000000000000;
            is_invalid <= 1'b1;
        end
    endcase
default: begin
    controls <= 12'b000000000000;
    is_invalid <= 1'b1;
end
endcase
end

```

(三) aludec.v 模块设计

1. 功能描述

该模块对指令进行译码，根据指令的 op、funct、rs 和 rt 识别所有指令，生成 ALU 的控制信号 alucontrol。

2. 接口定义

```

module aludec(
    input wire[5:0] funct,
    input wire[5:0] op,
    input wire[4:0] rs,
    input wire[4:0] rt,
    output reg[4:0] alucontrol
);

```

信号名	位宽	方向	功能描述
op	6-bit	Input	指令的 opcode
funct	6-bit	Input	指令的 funct
rs	5-bit	Input	指令的 rs
rt	5-bit	Input	指令的 rt
alucontrol	5-bit	Output	ALU 控制信号，代表不同的运算类型

3. 逻辑控制

根据 op 区分指令是 R-Type 还是 I-Type 还是 J-Type 还是特权指令。R-Type 指令根据 funct 再作区分；I-Type 指令中的 REGIMM 指令根据 rt 再作区分；特权指令根据 rs 再作区分。然后再为不同指令生成 ALU 控制信号 alucontrol。

```
always @(*) begin
    case (op)
        `R_TYPE:
            case (funct)
                //算术运算
                `ADD:      alucontrol = `ADD_CONTROL;
                `ADDU:     alucontrol = `ADDU_CONTROL;
                `SUB:      alucontrol = `SUB_CONTROL;
                `SUBU:     alucontrol = `SUBU_CONTROL;
                `SLT:      alucontrol = `SLT_CONTROL;
                `SLTU:     alucontrol = `SLTU_CONTROL;
                `DIV:      alucontrol = `DIV_CONTROL;
                `DIVU:     alucontrol = `DIVU_CONTROL;
                `MULT:     alucontrol = `MULT_CONTROL;
                `MULTU:    alucontrol = `MULTU_CONTROL;
                //逻辑运算
                `AND:      alucontrol = `AND_CONTROL;
                `NOR:      alucontrol = `NOR_CONTROL;
                `OR:       alucontrol = `OR_CONTROL;
                `XOR:      alucontrol = `XOR_CONTROL;
                //移位
                `SLLV:     alucontrol = `SLLV_CONTROL;
                `SLL:      alucontrol = `SLL_CONTROL;
                `SRAV:     alucontrol = `SRAV_CONTROL;
                `SRA:      alucontrol = `SRA_CONTROL;
                `SRLV:     alucontrol = `SRLV_CONTROL;
                `SRL:      alucontrol = `SRL_CONTROL;
                //数据移动
                `MFHI:     alucontrol = `MFHI_CONTROL;
                `MTHI:     alucontrol = `MTHI_CONTROL;
                `MFLO:     alucontrol = `MFLO_CONTROL;
                `MTLO:     alucontrol = `MTLO_CONTROL;
                //JALR
                `JALR:     alucontrol = `ADDU_CONTROL; //做加法
                default:   alucontrol = `USELESS_CONTROL;
            endcase
    endcase
end
```

```

//I-type
//算术运算
`ADDI:      alucontrol = `ADD_CONTROL;
`ADDIU:     alucontrol = `ADDU_CONTROL;
`SLTI:      alucontrol = `SLT_CONTROL;
`SLTIU:     alucontrol = `SLTU_CONTROL;
//逻辑运算
`ANDI:      alucontrol = `AND_CONTROL;
`LUI:       alucontrol = `LUI_CONTROL;
`ORI:       alucontrol = `OR_CONTROL;
`XORI:      alucontrol = `XOR_CONTROL;
//访存
`LB, `LBU, `LH, `LHU, `LW, `SB, `SH, `SW: alucontrol = `ADDU_CONTROL;
//跳转链接类
`REGIMM_INST:
    case(rt)
        `BGEZAL, `BLTZAL: alucontrol = `ADDU_CONTROL; //做加法
        default: alucontrol = `USELESS_CONTROL;
    endcase
`JAL : alucontrol = `ADDU_CONTROL; //做加法
//for exception
`SPECIAL3_INST:
    case(rs)
        `MTC0: alucontrol = `MTC0_CONTROL;
        `MFC0: alucontrol = `MFC0_CONTROL;
        default: alucontrol = `USELESS_CONTROL;
    endcase
    default: alucontrol = `USELESS_CONTROL;
endcase
end

```

三、实验过程（40%）

（一）设计工作日志

2022 年 12 月 25 日-2023 年 1 月 2 日，陈仕广和吴卓宣上课、阅读资料，观看往年教学视频，在计组 lab4 的基础上添加指令。

2023 年 1 月 3 日上午，吴卓宣完成逻辑运算指令、移位运算指令，陈仕广完成数据移动指令的设计方案。

2023 年 1 月 3 日下午，吴卓宣完成逻辑运算指令、移位运算指令、数据移动指令的控制器译码；陈仕广完成 ALU 中具体的计算逻辑。

2023 年 1 月 4 日整天，陈仕广、吴卓宣共同进行逻辑运算指令单独测试、移位运算指令单独测试、数据移动指令单独测试，最终调试通过。

2023 年 1 月 5 日整天，陈仕广完成访存指令实现方案设计及代码编写，并完善了设计文档；

吴卓宣完成分支跳转指令，实现方案设计及代码编写，并完善了设计文档。

2023 年 1 月 6 日上午，陈仕广测试通过访存指令单独测试；吴卓宣测试通过分支跳转指令单独测试。

2023 年 1 月 6 日下午，陈仕广完成算数运算指令中乘、除法指令实现，并完善了设计文档。

2023 年 1 月 6 日晚上，陈仕广、吴卓宣共同测试通过算数运算指令单独测试。

2023 年 1 月 7 日，陈仕广完成精确异常 5 条指令实现方案设计，并完善了设计文档。吴卓宣完善设计文档。

2023 年 1 月 8 日整天，陈仕广完成精确异常部分的代码编写。吴卓宣连接上 SRAM-SOC 并开始进行功能测试调试。

2023 年 1 月 9 日整天，陈仕广、吴卓宣继续功能测试调试，并进行错误记录，成功通过 89 个测试点，上板继续调试，最终成功跑通。

2023 年 1 月 10 日整天，陈仕广连接上 AXI 接口。

2023 年 1 月 11 日整天，陈仕广开始进行 AXI 接口的功能测试，并进行调试和错误记录，吴卓宣辅助调试，最终成功通过功能测试。

2023 年 1 月 12 日整天，陈仕广进行 AXI 接口的性能测试，仿真通过、上板也通过。接上基础 cache 后调试后也成功运行。吴卓宣进行 ppt 制作和代码整理。

2023 年 1 月 13 日，进行答辩和报告修缮并提交。

（二）错误记录

1、仿真逻辑运算指令时错

（1）错误现象

regfile 内的寄存器全是 X，仿真失败。

（2）分析定位过程

运行仿真，根据 LUI 指令，拉取关键控制信号如 alusrcD、alusrcE，发现 alusrcD 拉高后下一拍 alusrcE 并没有拉高，检查对应代码部分发现是控制信号

传递错位。

(3) 错误原因

控制信号 regdstD 原来的位宽为 1，扩展指令时更改其位宽为 2，但是与之关联的 regdstE 信号的位宽忘记更改，导致控制信号的传递错位，仿真失败。

(4) 修正效果

对应更改 regdstE 信号的位宽为 2。

2、仿真数据移动指令时报错

(1) 错误现象

运行到 MTLO 指令时，发现 HI 寄存器的值也被更改了。

(2) 分析定位过程

直接检查 alu 内对应代码，发现 MTLO 指令的处理逻辑出错。

(3) 错误原因

```
`MTHI_CONTROL : hilo_out = {a,hilo_in[31:0]}; //指令MTHI
`MTLO_CONTROL : hilo_out = {hilo_in[31:0],a}; //指令MTLO
```

(4) 修正效果

```
`MTHI_CONTROL : hilo_out = {a,hilo_in[31:0]}; //指令MTHI
`MTLO_CONTROL : hilo_out = {hilo_in[63:32],a}; //指令MTLO
```

3、仿真访存指令时

(1) 错误现象

连续两次读数据存储器同一个地址，读出的值不一致。

(2) 分析定位过程

依据指令，跟着仿真波形图一步步查看，发现其他逻辑都没问题，于是怀疑是数据存储器的问题，vivado 打开数据存储器，发现读延迟是两个时钟周期：

Component Name data_mem

Basic Port A Options Other Options Summary

Information

Memory Type: Single Port Memory

Block RAM resource(s) (18K BRAMs): 0

Block RAM resource(s) (36K BRAMs): 64

Total Port A Read Latency : 2 Clock Cycle(s)

Address Width A: 32

果然是数据存储器的問題。

(3) 錯誤原因

數據存儲器讀延遲應該是一個時鐘周期，但實際為兩個時鐘周期。

(4) 修正效果

重新調整數據存儲器，取消內部輸出寄存器：

Component Name data_mem

Basic Port A Options Other Options Summary

Memory Size

Write Width 32 Range: 32 to 1024 (bits)

Read Width 32

Write Depth 65532 Range: 2 to 1048576

Read Depth 65532

Operating Mode Write First Enable Port Type Always Enabled

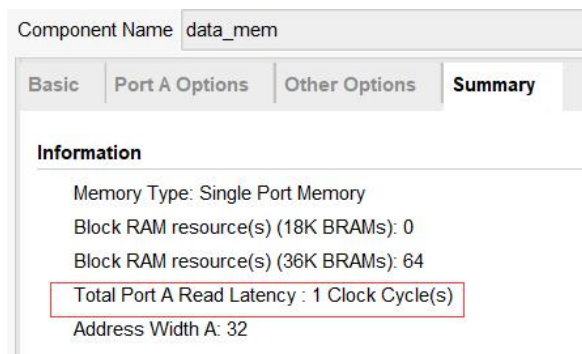
Port A Optional Output Registers

☒ Primitives Output Register ☐ Core Output Register

☐ SoftECC Input Register ☐ REGCEA Pin

Port A Output Reset Options

此時讀時延為 1 個時鐘周期：



问题解决，重新仿真后，经过比对验证结果正确。

4、仿真算数运算指令时

（1）错误现象

程序没跑起来。

（2）分析定位过程

调出 stall 相关信号，发现均为高阻态 X，查看代码，发现是除法指令的暂停信号 div_stall 没有在 rst 时赋予初值 0，导致其为高阻态 X，受其影响，stallF、stallD、stallE 也都为高阻态，所以流水线没跑起来。

（3）错误原因

除法指令的暂停信号 div_stall 没有在 rst 时赋予初值 0，导致其为高阻态 X，流水线被暂停。

（4）修正效果

给 div_stall 信号在 rst 时赋初值 0 即可。

5、功能测试报错 1

（1）错误现象

trace 调试，测试点 1 出错。

（2）分析定位过程

观察波形图和相应指令，发现 jr 指令跳转错误。分析要跳转的地址，发现是数据冒险，即要跳转的寄存器值还没有被写入，所以要进行流水线暂停。

(3) 错误原因

jr 指令数据冒险。

(4) 修正效果

把 jrD 控制信号传入 hazard 模块，跟 branch 指令类似，控制流水线暂停。

6、功能测试报错 2

(1) 错误现象

trace 调试，测试点 16 误报整型溢出异常。

(2) 分析定位过程

查看 alu 逻辑，是整型溢出处理失误。

(3) 错误原因

整型溢出处理失误。

(4) 修正效果

使用的是双符号位的判断方法：正数符号位 00，负数符号位 11，相加（减）后如果双符号位为 01，则正溢出，10 则负溢出。判断是否溢出可以通过判断结果的双符号位是否相同。使用异或逻辑。

7、功能测试报错 3

(1) 错误现象

trace 调试，测试点 44 除法结果错误。

(2) 分析定位过程

观察波形图，发现除法进行过程中，alu 的源操作数输入发生了改变。

(3) 错误原因

改变的原因是除法刷新 M 阶段，继而使数据前推选择器信号改变，导致 alu 的源操作数输入改变，除法的源操作数随之发生改变，所以除法出错。

(4) 修正效果

在 ALU 新增两个 32 位寄存器，用于保存除法的源操作数，使之不随 ALU 输入改变而改变。注意操作数的保存还要采用非阻塞赋值，不然时序也有问题。

8、功能测试报错 4

(1) 错误现象

trace 调试，测试点 65，MFC0 指令错误。

(2) 分析定位过程

观察波形调试发现小 bug。

(3) 错误原因

1. MFC0 指令的数据前推选择器的 32 位的输出信号忘记声明（位宽大于 1 必须显式声明，否则默认位宽为 1）。

2. aludec 模块忘记译码出 MFC0 指令的 alucontrol 信号，导致 alu 输出为默认的全 0。

(4) 修正效果

增加对应声明和信号即可。

9、功能测试报错 5

(1) 错误现象

trace 调试，测试点 65，SYSCALL 指令错误。

(2) 分析定位过程

触发异常处理时，可能有后续指令(无效执行的指令)会暂停流水线，这个暂停会导致 pc 取不到异常处理地址 0xBFC00380。因为暂停时，pc 保持不变，然后下一周期不暂停了，但是 0xBFC00380 也流走了，所以这个异常就得不到处理，出错。

(3) 错误原因

触发异常处理时取指阶段可能可暂停，导致取不到异常处理地址 0xBFC00380 处的指令。

(4) 修正效果

因此，触发异常时，不能暂停取指阶段。修改取指阶段暂停逻辑如下：

```
assign stallF = (~is_exceptM & stallD);
```

10、功能测试报错 6

(1) 错误现象

trace 调试，测试点 65，SYSCALL 指令错误。

(2) 分析定位过程

观察波形图，发现 SYSCALL 指令后跟 DIV 指令，流水线被刷新，并且由于 div_stall 一直为 1，流水线一直暂停。

(3) 错误原因

SYSCALL 指令后跟 DIV 指令，由于原逻辑触发异常后刷新流水线时，并未处理除法刷新，导致除法运行完后也无法停止 div_stall (因为 alucontrol 已经被刷掉了)，所以流水线一直暂停。

(4) 修正效果

在 alu 内，触发异常时，刷新除法暂停，并终止除法。如下：

```
if(rst | is_except) begin
    div_stall = 1'b0;
    div_start = 1'b0;
end
```

```
wire annul; //终止除法信号
assign annul = ((alucontrolE == `DIV_CONTROL)|(alucontrolE == `DIVU_CONTROL)) & is_except;
//接入除法器
div div(clk,rst,div_signed,a_save,b_save,div_start,annul,div_result,div_ready);
```

11、功能测试报错 7

(1) 错误现象

trace 调试，测试点 67，整型溢出的 add 指令错误提交。

(2) 分析定位过程

根据报错 PC 定位，观察波形图，发现一条整型溢出 add 指令也被提交。

(3) 错误原因

异常指令被错误提交。

(4) 修正效果

触发异常时，要进行写回级的刷新，防止异常指令提交。

```
assign flushW = is_exceptM;
```

12、功能测试报错 8

(1) 错误现象

trace 调试，测试点 70，MFC0 指令错误。

(3) 分析定位过程

定位 PC, 观察波形图，即可发现。

(4) 错误原因

地址不对齐异常的 bad_addrM 忘记声明为 31 位宽。导致写入 CP0.badvaddr 错误，所以读出也错误。

(5) 修正效果

解决：增加声明 wire [31:0] bad_addrM;

13、功能测试报错 9

(1) 错误现象

trace 调试，测试点 73，LW 指令错误。

(2) 分析定位过程

观察波形图，找到对应的 SW 指令，检查地址和数据。

(3) 错误原因

发现该 SW 指令是异常地址，但是却把数据写进了数据存储器。

(4) 修正效果

调整数据存储器的使能信号，防止异常地址写入或读出。

```
assign mem_enM = (~is_AdEL_dataM & ~is_AdESM); //存储器使能，防止异常地址写入或读出
```

14、接上 AXI, 功能测试

14.1 测试点 1

debug 1，全停时 pc 的传递也要停，原设计未考虑。

debug 2，全停时，同一条指令多次提交。解决方案：新增一个信号做判断，如果 stallW 拉高，则把连接 debug 写寄存器使能的信号 regwrite_for_debugW 拉低。（注意：不能直接拉低 regwriteW，也不能刷新 WB 阶段，见 debug4）。

判断逻辑：wire regwrite_for_debugW = stallW ? 0 : regwriteW;

debug 3，当全停时，若某一个阶段 A 的后面阶段也暂停了，则不能刷新 A 阶段。原逻辑是 lwstall 和 branchstall 都会刷新 EX 阶段，但是如果现在也处于全停状态，则是不能刷新 EX 的。

debug 4，全停状态下，当 EX 阶段的数据来源于 WB 阶段前推时，由于此前应对 debug 2 的解决方案是：全停时刷新 WB 阶段，而这导致一些寄存器状态改变使数据前推选择信号出错，所以 ALU 的输入中途改变，输出自然也错。解决方案：全停状态不刷新 WB 阶段，debug 2 换另一种方案去解决。

14.2 测试点 65

debug 5，发生异常时，由于 debug 2 的解决方案是：（stallW 拉高时，regwrite_for_debugW 将拉低），但是当 MEM 阶段触发异常时，而 stallW 是拉高的，这就导致 regwrite_for_debugW 拉低，trace 无法采样，所以 trace 比对失败，报错。

解决方案：调整 WB 阶段的暂停逻辑，当异常时取消暂停：

```
assign stallW = longest_stall & ~is_exceptM;
```

14.3 测试点 77

debug 6, 软件中断, 写 CP0_cause 时, 由于是上升沿写, 所以无法同周期检测中断异常, 若是下一周期再检测到中断异常, 此周期内 PC 还不是异常处理的固定地址 BFC00380, 故下一周期时钟跳变取指令时, 取出了错误指令, 而 BFC00380 的指令从波形图来看是被跳过了, 所以出错了。由于我不太了解怎么才是较优的解决方案, 我就按自己的思路来, 修改 CP0 寄存器为时钟下降沿写, 这样就可以同周期检测所有异常或中断。但是这样会引发另一个问题: 触发异常时, 要记录错误指令的地址 PC, 显然如果我们此时还是直接把 pcM 写入 cp0_epc 的话, 由于触发异常时的流水线冲刷, 而我们此时是下降沿写 CP0, 所以 pcM 会变为全 0!!! 因此再次修改, 把 pcE 用一个不被流水线刷新的寄存器传递到 M 阶段, 再接入 CP0。

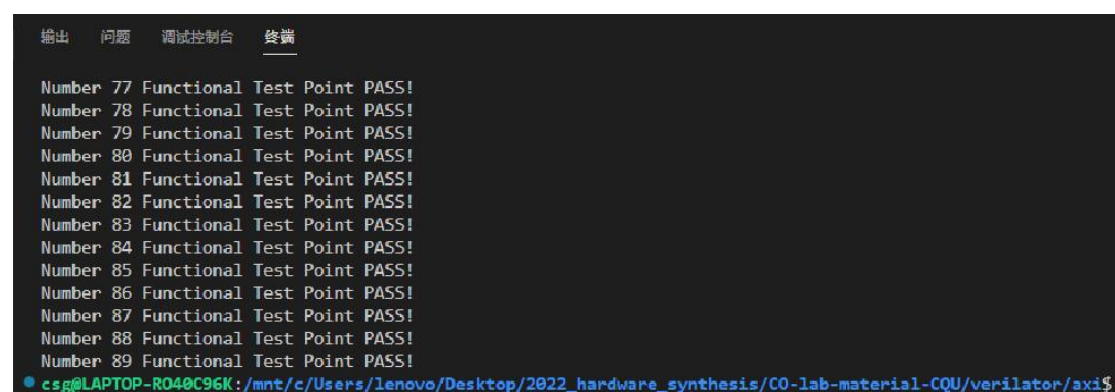
四、设计结果

(一) 设计交付物说明

/src/myCPU_with_axi 文件夹下, 是我们实现的 MIPS 处理器。若要进行仿真, 需要把整个 myCPU_with_axi 文件夹导入实验资料包相应的工程中。即可仿真、综合、上板。

(二) 设计演示结果

2.1 功能仿真测试截图 (AXI)

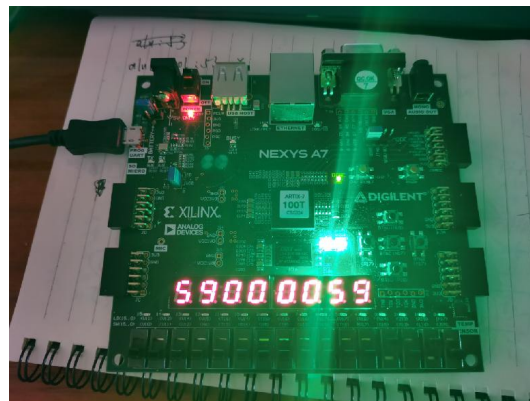


```
输出 问题 调试控制台 终端
Number 77 Functional Test Point PASS!
Number 78 Functional Test Point PASS!
Number 79 Functional Test Point PASS!
Number 80 Functional Test Point PASS!
Number 81 Functional Test Point PASS!
Number 82 Functional Test Point PASS!
Number 83 Functional Test Point PASS!
Number 84 Functional Test Point PASS!
Number 85 Functional Test Point PASS!
Number 86 Functional Test Point PASS!
Number 87 Functional Test Point PASS!
Number 88 Functional Test Point PASS!
Number 89 Functional Test Point PASS!
csg@LAPTOP-R040C96K: /mnt/c/Users/lenovo/Desktop/2022_hardware_synthesis/CO-lab-material-CQU/verilator/axi$
```

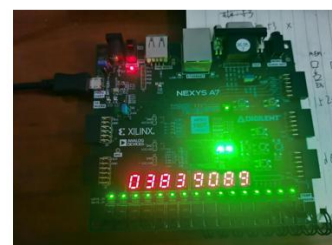
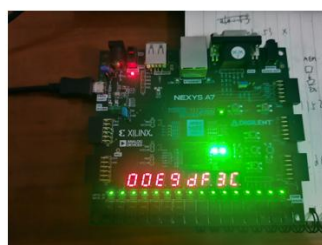
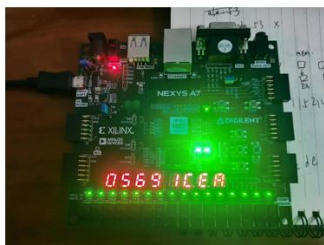
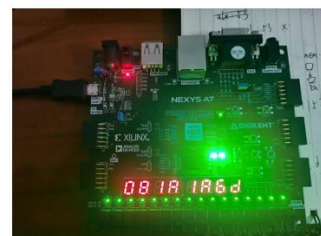
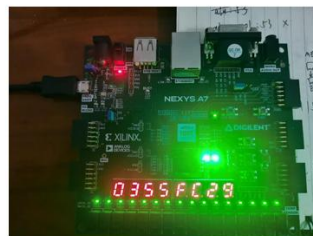
2.2 性能仿真测试截图（AXI）

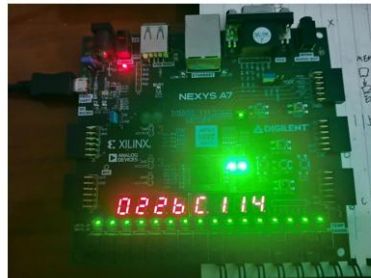
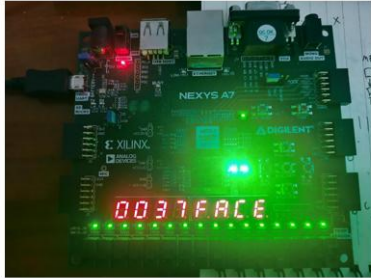
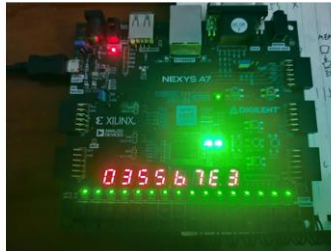
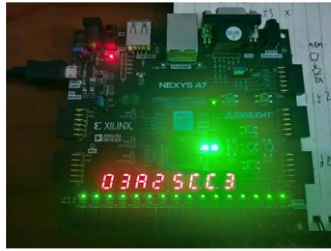
```
cs@LAPTOP-R040C96K:/mnt/c/Users/lenovo/Desktop/2022_hardware_synthesis/CO-lab-material-CQU/verilator/axi$ obj_dir/Vmycpu_top -perfdiff
878abc
3202814
76560c0
4ea5327
dcca6a
339372e
3463b0c
31bac3d
33e460
21c0073
total ticks = 966777401
cs@LAPTOP-R040C96K:/mnt/c/Users/lenovo/Desktop/2022_hardware_synthesis/CO-lab-material-CQU/verilator/axi$
```

2.3 功能测试上板图（SRAM-SOC）



2.4 性能测试上板图（AXI）





2.5 性能得分表（AXI）

二、性能测试分数计算

序号	测试程序	myCPU	gss132	T _{gss132} /T _{mycpu}
		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	953A77	13CF7FA	2.124050814
2	bubble_sort	355FC29	7BDD47E	2.320684844
3	coremark	81A1A6D	10CE6772	2.074344242
4	crc32	5691CEA	AA1AA5C	1.96494118
5	dhrystone	E9DF3C	1FC00D8	2.17214235
6	quick_sort	3839089	719615A	2.020288814
7	select_sort	3A25CC3	6E0009A	1.891738578
8	sha	355B7E3	74B8B20	2.187540598
9	stream_copy	37FACE	853B00	2.379978008
10	stringsearch	22BC114	50A1BCC	2.321365372

性能分	2.140
-----	-------

五、参考设计说明

本次硬件综合模块引用如下：

- 1，计算组成原理 lab4 标准代码：实验资料包提供。
- 2，defines2.vh 宏定义：实验资料包提供。
- 3，div.v 除法模块：实验资料包提供。
- 4，cp0_reg.v：实验资料包提供。
- 5，基础 cache:实验资料包提供。

- 6, d_sram_to_sram_like.v、i_sram_to_sram_like.v 类 SRAM 转换：引用自计算机系统结构实验二资料包。
- 7, 转接桥 bridge_1x2.v, bridge_2x1.v: 引用自计算机系统结构实验二资料包。
- 8, mmu.v: 实验资料包提供。
- 9, cpu_axi_interface.v: 实验资料包提供。
- 10, axi 接口的 mycpu_top.v: 引用自计算机系统结构实验二资料包。

六、总结

(一) 组员：陈仕广

历经磨难，终于完成了硬件综合设计，并达到了较高的要求。虽然 debug 到怀疑人生，但是最终效果是好的，努力值得！万水千山总是情，想拿高分行不行！

(二) 组员：吴卓宣

本次硬件综合设计的经历是惊心动魄的，前有狼后有虎。前面的狼是指 12 月 25 日截止的机器学习和操作系统的课程报告、12 月 30 日截止的计算机网络课程项目以及 12 月 31 日截止的计算机系统结构综述论文。后有虎是指 1 月 9 日母亲突然病重，自从我送她去急诊到今天 1 月 13 日，母亲一直无法自理，我与父亲不得不轮流照顾母亲，留给我做硬件综合设计的时间进一步被压缩。万幸的是我有一个好队友，我非常感谢他的理解和帮助。尽管困难重重，但最终我们还是交出了一个令人较为满意的结果。在过程中，我收获了许多，我的 CPU 设计能力也得到了很大的提升。

七、参考文献

- [1] 《计算机组成原理实验指导书》,重庆大学计算机学院编.
- [2] 《MIPS 基准指令集手册》.

[3] 《功能测试说明》.

[4] 《性能测试说明》.

[5] 《计算机组成与设计-硬件软件接口》,David A.Patterson、John L.Hennessy 编,康继昌、樊晓桢等译,机械工业出版社,2012 年 1 月出版.

[6] 《自己动手写 CPU》,雷思磊著,电子工业出版社.

[7] 重庆大学硬件综合设计实验文档[EB/OL].[2023-01-12].<https://co.ccslab.cn/>