重庆大学课程设计报告

课程设计题目: MIPS SOC 设计与性能优化

学　　院: 计算机学院

专 业 班 级: 计算机科学与技术 03、06 班

年　　级: 2021

学　　生: 刘成　丁超　余肖杨

学　　号: 20215005　20214075　20214067

完 成 时 间: 2024 年 1 月 12 日

成　　绩:

指 导 教 师: 冯永

重庆大学教务处制

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

指导教师评定成绩：指导教师名：

MIPS SOC 设计报告

刘成、丁超、余肖杨

1 设计简介

本次硬件综合设计选择 MIPS32 指令集架构，以计算机组成原理实验四的五级流水线 CPU 为基础进行后续扩展[1][2]。按照先分析指令在数据通路中的路径，再将其添加到控制器和数据通路模块的思路，将之前已实现的十条指令先扩展为五十二条基础指令（包括 14 条算术运算指令、8 条逻辑运算指令、6 条移位指令、12 条分支指令、4 条数据移动指令、8 条访存指令）。完成后再添加 3 条特权指令和 2 条自陷指令完成完整的五十七条指令的实现。

随后加入异常处理模块并完善冒险模块。最后连接 AXI 接口，构造 AXI 版 SoC 系统。然后使用资料包中给出的 cache 模块，连接了基本 cache。最终实现的功能有五十七条指令、连接 SoC 与 AXI 接口功能，实现基础 cache 并通过了功能测试的 89 个测试点及 10 个性能测试，基本完成了一个具有一定规模指令集的 CPU[4]。

1.1 小组分工说明

- 刘成：从52条指令扩展到57条指令、连接AXI接口、实现基础cache，共同进行指令单独仿真测试、功能测试、性能测试和上板性能测试；
- 丁超：实现乘除法指令和数据移动指令、访存指令、连接SARAM-SOC接口，共同进行指令单独仿真测试、功能测试、性能测试和上板性能测试；
- 余肖杨：实现逻辑指令、移位指令、普通算术指令、分支跳转指令；共同进行指令单独仿真测试、功能测试、性能测试和上板性能测试；

2 设计方案

2.1 总体设计思路

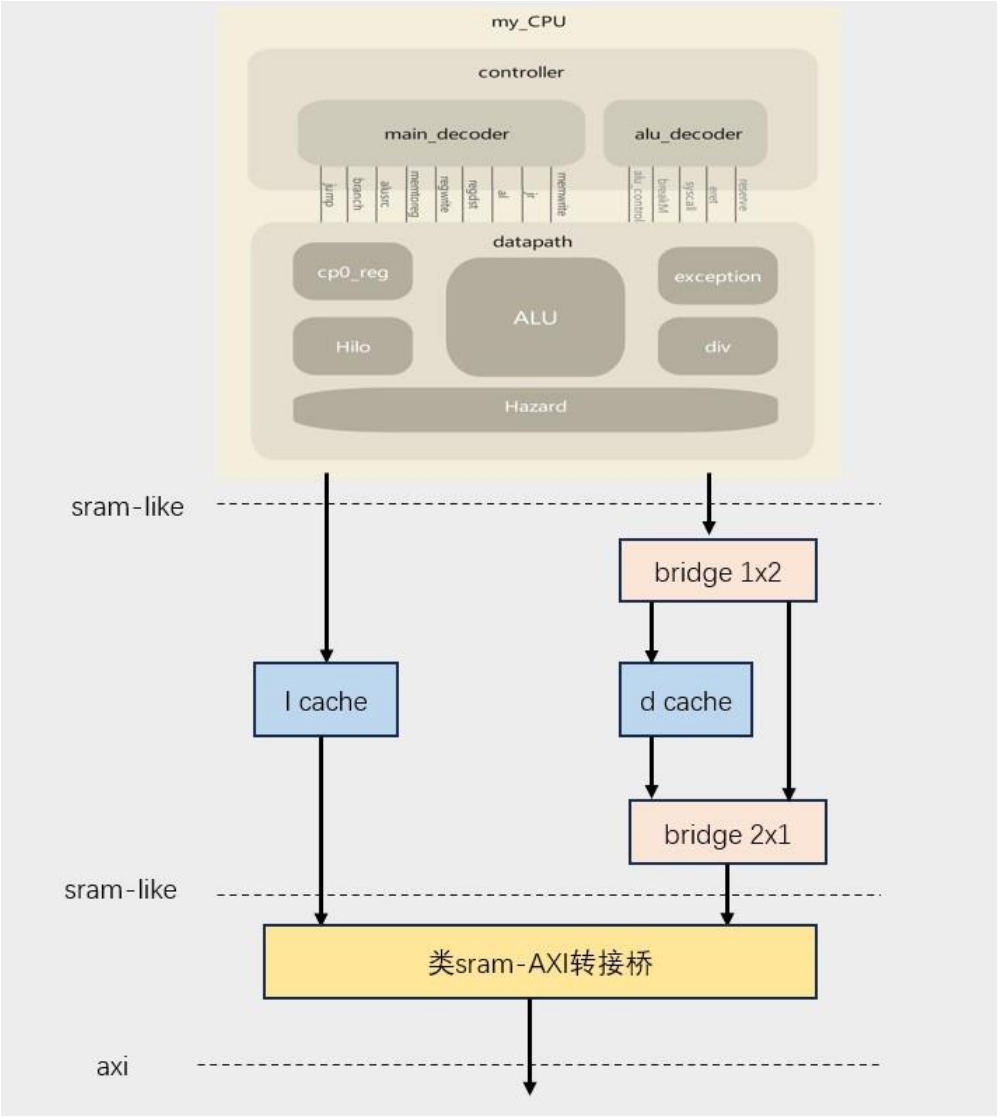


图 1: 总体

本次课设最终实现了连接 AXI 接口和基础 cache 的 SoC 系统，将 SRAM 接口转为类 SRAM 接口，然后使用资料包中的转接桥，实现基本 cache[3]。

对于 CPU 设计，我们将下层实现封装成控制单元 controller 和数据通路 datapath 两个部分。controller 模块负责译码产生控制信号，控制信号控制运算器 ALU 和整个 cpu 的运行，五级流水线的每一级也引入带有使能信号和清除信号的触发器。datapath 模块负责连接 cpu 底层各部件的各个输入输出接口，包括取指，译码，执行，访存，写回五个阶段。其中包含了 hazard 冒险模块和异常处理模块，hazard 模块负责处理数据冒险，控制冒险，用数据前推和阻塞解决相关的冒险，同时异常的刷新，乘除法的阻塞也在此模块解决；异常处理模块负责判断异常和处理异常，工作机制是将

取指、译码、执行、访存阶段发生的异常统一进行标记，在访存阶段进行处理，相应的 cp0 寄存器也放在了流水线的访存阶段[5]。

2.2 datapath 模块设计

IF：取指阶段根据 PC 指令计数器从指令存储器中取出对应指令。其中，需要根据信号量（pcsrcD，jumpD）对 PC 的取值进行选择。pc 模块与加法器协同实现每个周期 pc+4 的功能，pcbranch 与 pcjump 由后续步骤传入跳转部分 pc 的取值。

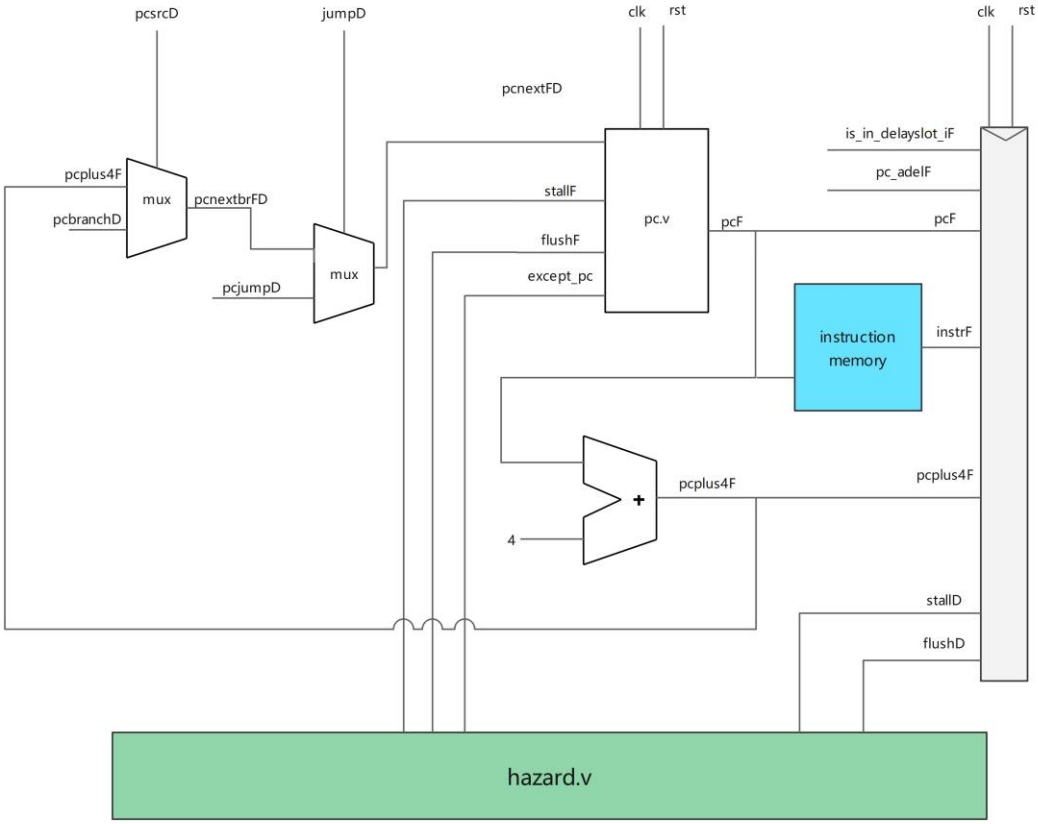


图 2: IF 阶段

ID：译码阶段得到 32 位的指令 instrD，在控制单元 controller 模块（图中未画出）中将指令中将相应字段所对应的值进行提取，并且生成 datapath 中各种元件的控制信号，如 pcbranchD、equalD 等。译码阶段的重点是通过对 instrD 的分解得到对应寄存器中的值，或者对立即数进行扩展和处理。eqcmp 判断分支指令是否符合跳转条件，进行分支指令的跳转判断，涉及分支指令操作数的数据前推。

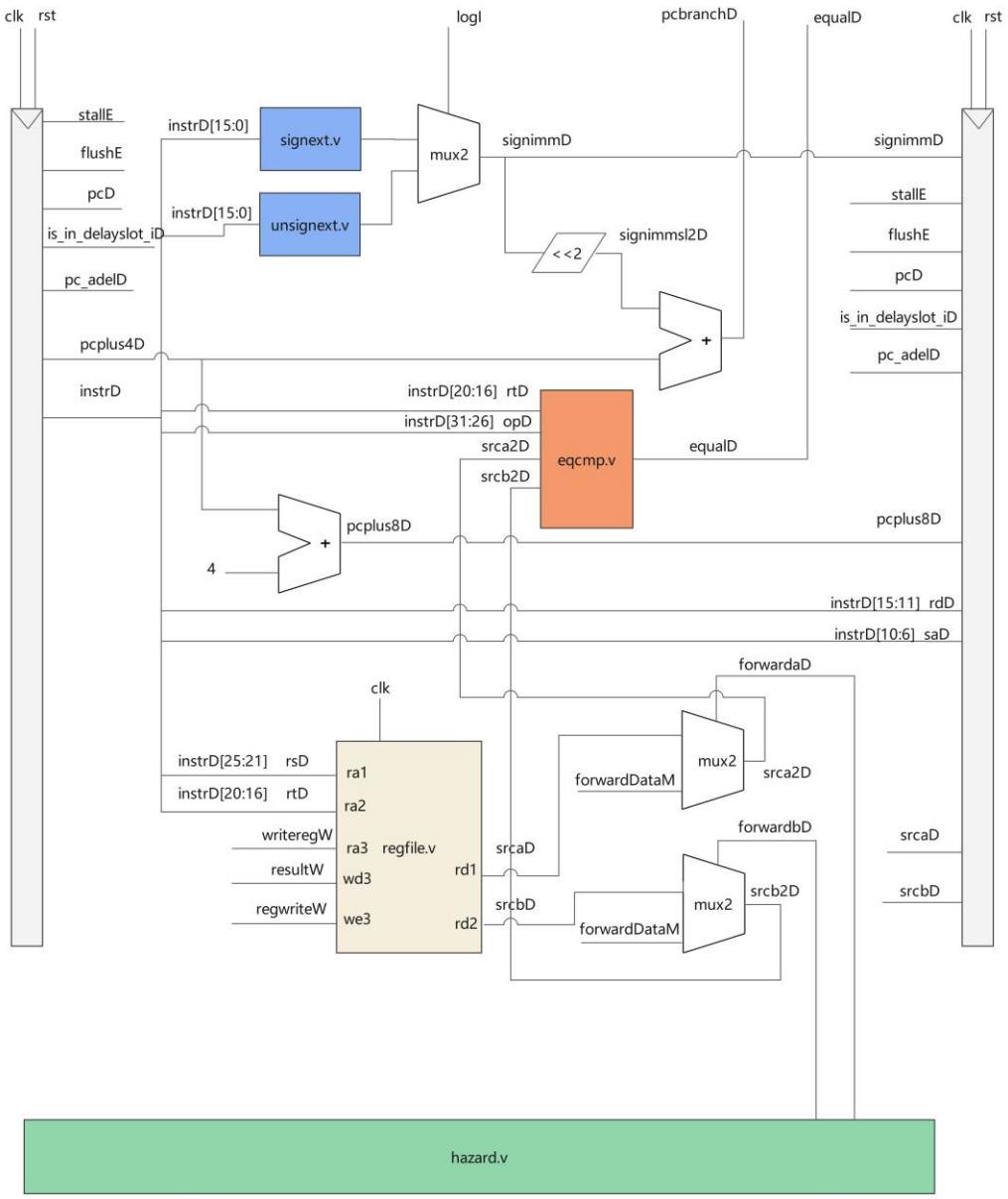


图 3: ID 阶段

EX：执行阶段的主要功能是执行指令指定的 ALU 模块的相关操作，涉及输入 ALU 的两个操作数的数据前推。其中需要根据 controller 模块的信号量执行相应的 ALU 操作。由于 div 操作比较复杂，div 模块独立于 alu 模块，由于除法操作产生的额外的 stall 会交给竞争冒险模块处理。

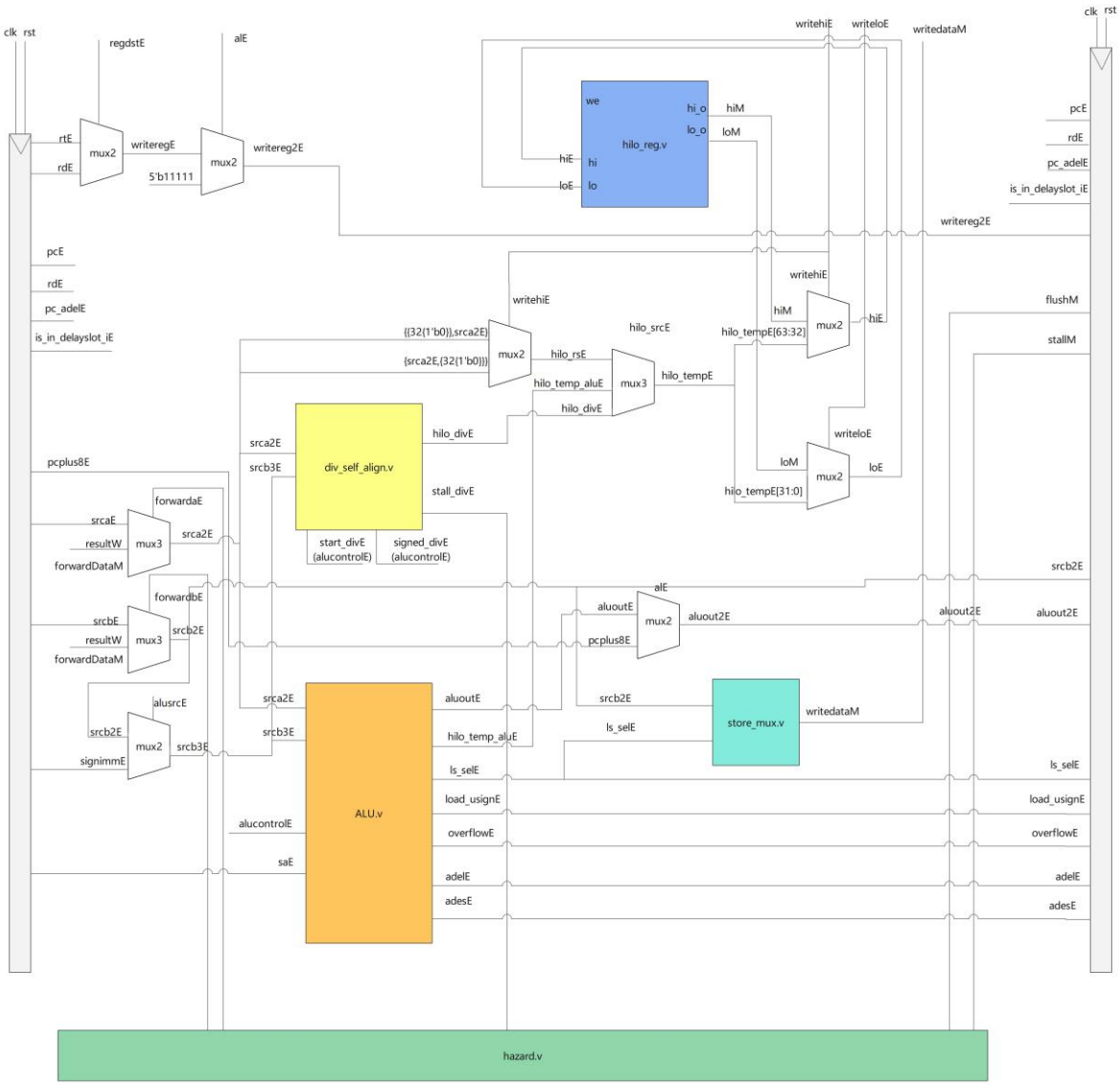


图 4: EX 阶段

MEM：访存阶段的主要功能是访存指令进行数据存储器的读写，另外前面阶段的所有异常也需要在本阶段进行处理。图中 cp0 reg 模块和 exception 模块进行异常处理。另外，与其他阶段相似的，需要接收上一级流水线执行阶段 E 传递下来的控制信号，并输出相对应的本级流水线所需的控制信号；根据阻塞 stall 或刷新 flush 的控制信号，以决定是否阻塞本级流水线继续向下一级流水线传递信号或刷新本阶段流水线信号。

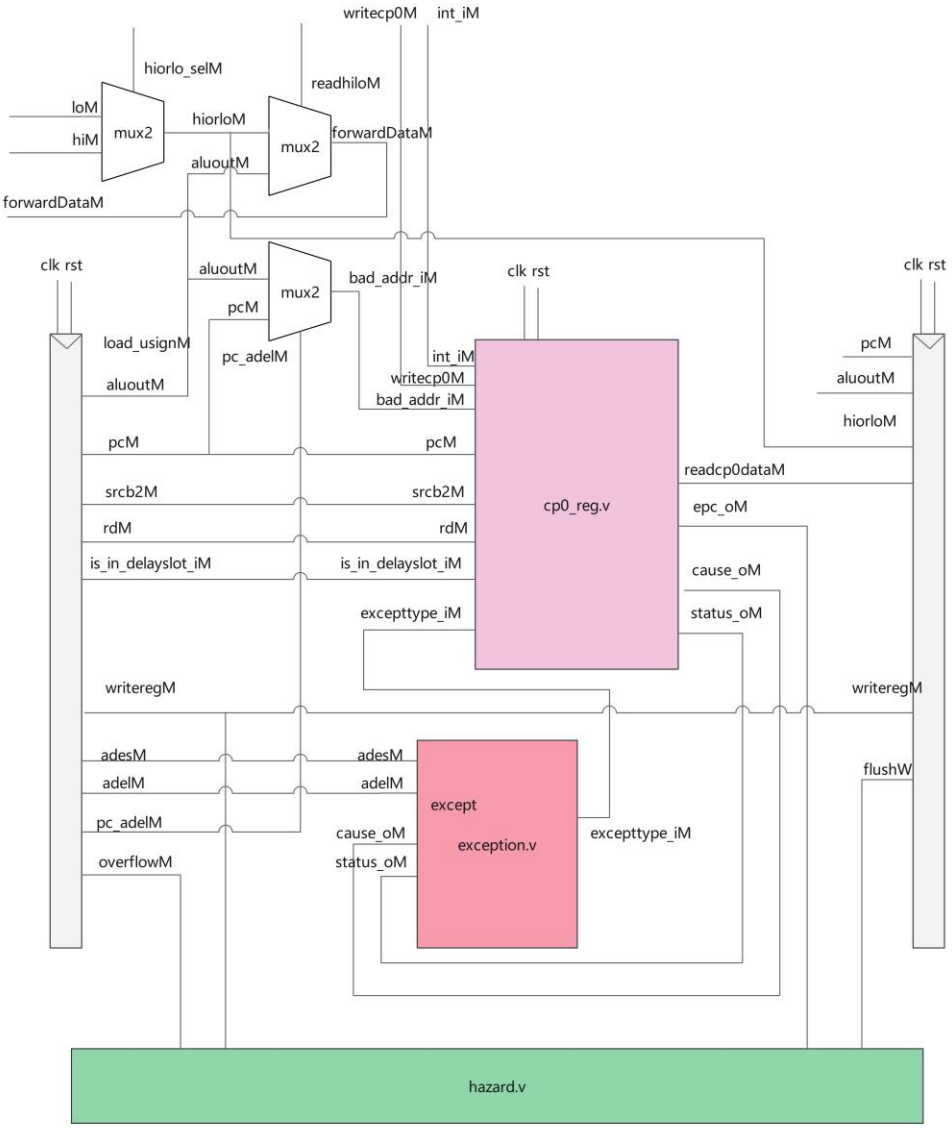


图 5: MEM 阶段

WB：写回阶段主要功能是将数据写入寄存器堆。图中的多个多路选择器对写回寄存器堆的数据进行选择：ALU 计算结果、Data RAM 中读取的数据、CP0 寄存器中读出的数据、HILO 寄存器中读出的数据。最终将 resultW 写入寄存器堆。

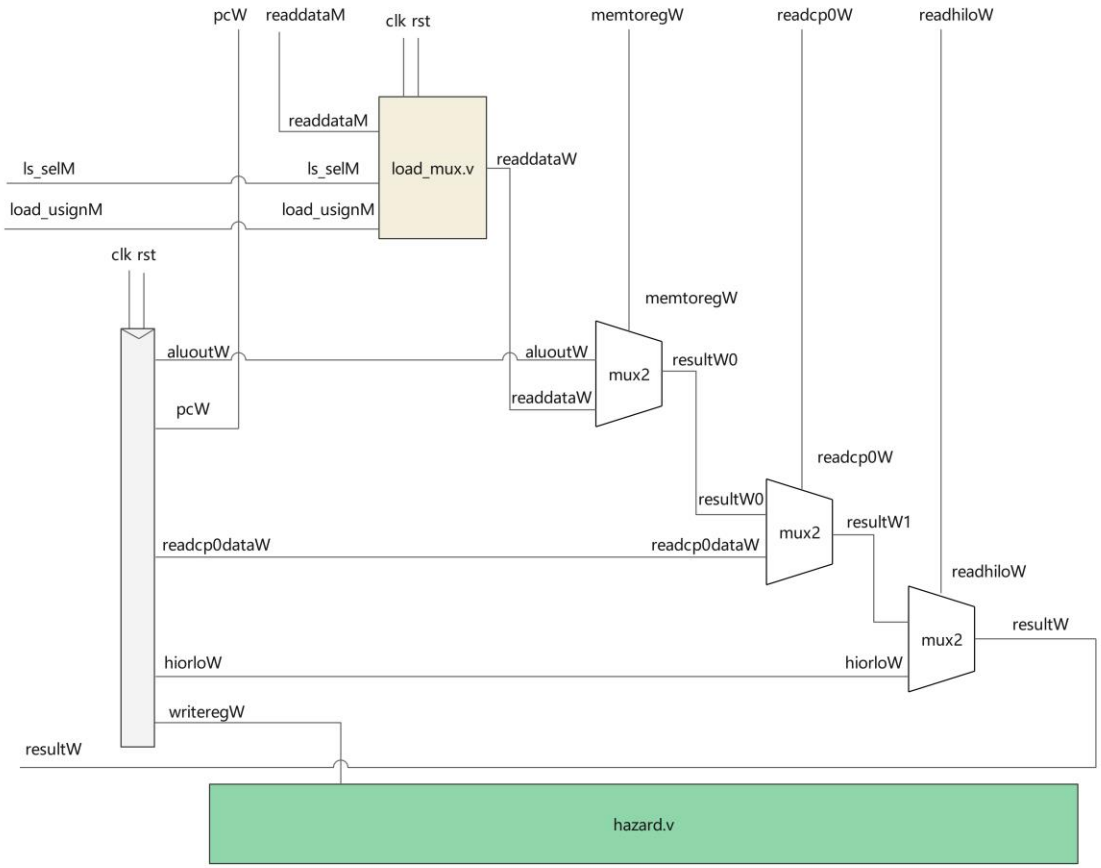


图 6: WB 阶段

接口定义如下：

信号名	方向	位宽	功能描述
clk	input	1bit	时钟信号
rst	input	1bit	复位信号：0 代表不复位，1 代表复位
int iM	input	6bit	硬件中断
pcF	output	32bit	取指阶段的 pc
instrF	input	32bit	取指阶段：指令二进制表示
pcsrcD	input	1bit	译码阶段程序计数器值的来源
branchD	input	1bit	译码阶段：表示是否为分支跳转类型指令
jumpD	input	1bit	译码阶段：表示是否为跳转类型指令
logI	input	1bit	表示立即数选择无符号填充还是符号填充
jrD	input	1bit	译码阶段：表示是否为跳转寄存器类型指令

equalD	output	1bit	译码阶段：是否满足分支指令的跳转条件
opD	output	6bit	指令的 [31:26] 操作码
functD	output	6bit	指令的 [5:0] 功能码
rtD	output	5bit	指令的 [20:16] 位
rsD	output	5bit	指令的 [25:21] 位
memtoregE	input	1bit	执行阶段：将要写回寄存器堆的数据的来源
alusrcE	input	1bit	执行阶段：表示传入 alu 模块的第二个源操作数的来源
regdstE	input	1bit	执行阶段将要写入的寄存器号是 rd 或是 rt 字段
regwriteE	input	1bit	执行阶段：寄存器堆写使能信号
alE	input	1bit	执行阶段：选择写入寄存器堆的数据、地址
alucontrolE	input	8bit	当前指令类型对应的 alu 控制信号
flushE	output	1bit	执行阶段：表示是否需要刷新流水线
writehiE	input	1bit	Hilo 寄存器 hi 部分写使能
writeloE	input	1bit	Hilo 寄存器 lo 部分写使能
hilo srcE	input	2bit	hilo rsE、hilo temp aluE、hilo divE 数据选择
stallE	output	1bit	执行阶段：表示是否需要阻塞流水线
readcp0E	input	1bit	执行阶段：cp0 寄存器读使能
memtoregM	input	1bit	访存阶段：将要写回寄存器堆的数据的来源
regwriteM	input	1bit	访存阶段：寄存器堆写使能信号
aluoutM	output	32bit	ALU 模块计算的值
writedataM	output	32bit	访存阶段：需要写入数据存储器的数据
readdataM	input	32bit	访存阶段：数据存储器读出的数据
hiorlo selM	input	1bit	Hilo 寄存器读取数据选择
readhiloM	input	1bit	访存阶段：hilo 寄存器的读使能
writecp0M	input	1bit	访存阶段：cp0 寄存器写使能
readcp0M	input	1bit	访存阶段：cp0 寄存器读使能
ls selM2	output	4bit	数据存储器字节写使能

breakM	input	1bit	访存阶段：break 处理
syscallM	input	1bit	访存阶段：syscall 处理
eretM	input	1bit	访存阶段：eret 处理
reserveM	input	1bit	访存阶段：reserve 处理
flushM	output	1bit	访存阶段：表示是否需要刷新流水线
excepttype iM	output	32bit	例外类型
stallM	output	1bit	访存阶段：表示是否需要阻塞流水线
memtoregW	input	1bit	写回阶段：将要写回寄存器堆的数据的来源
regwriteW	input	1bit	写回阶段：寄存器堆写使能信号
readhiloW	input	1bit	hilo 寄存器读使能
readcp0W	input	1bit	写回阶段：cp0 寄存器读使能
resultW	output	32bit	寄存器堆读出的数据
writeregW	output	5bit	写入寄存器堆的数据地址
pcW	output	32bit	写回阶段的 pc
flushW	output	1bit	写回阶段：表示是否需要刷新流水线
stallreq from if	input	1bit	取指阶段的 stall 请求
stallreq from mem	input	1bit	访存结果的 stall 请求

表 1: datapath 控制信号含义

2.2.1 ALU 模块设计

(1) 功能描述：alu 模块根据 controller 模块中 alu decoder 输出的 alu 控制信号 alu control 来决定需要运算的类型，进行相应的运算后，输出结果 result。

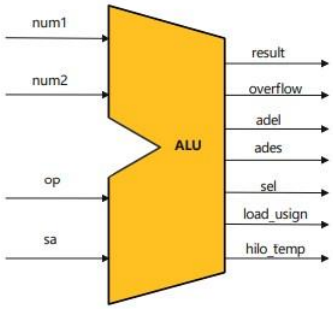


图 7: ALU 模块

(2) 实现逻辑：模块中包含逻辑运算指令、移位指令、数据移动指令、算术运算指令、访存指令、内陷指令和特权指令，每个指令通过输入的 op 分类，进行相应的运算。

- 对于逻辑运算、移位、算术逻辑指令，直接运算得到结果 result，其中加减法有溢出情况，输出 overflow，与 exception 模块连接，用于异常处理。

Algorithm 1 Overflow

```
assign overflow = (op ==‘EXE_ADD_OP || op == ‘EXE_ADDI_OP) ? ( (num1[31] & num2
    [31] & (~result[31])) | ((~num1[31]) & (~num2[31]) & result[31]) ) : (op ==
        ‘EXE_SUB_OP) ? ( (num1[31] & (~num2[31]) & (~result[31])) | ((~num1[31]) & num2[31] & result[31]) ):1’b0;
```

- 对于访存类指令，包括 load 和 store 两种，load 从 RAM 中读取一个字、半个字、一个字节的数据，如果不足一个字，则进行有符号或无符号扩展，写入到 rt 寄存器中。store 指令则将 rt 寄存器中的值最低字节、低半字、整字存入 RAM 中。

  在 load 指令中，我们需要将最终写入寄存器堆的数 resultW 通过选择器选择为 readdataW。在 lab4 中，这里的 readdataW 就是从 Data RAM 中读出的数据，但是现在，我们需要对从 Data RAM 中读出的数据进行一定的处理，才能得到 readdataW. 这一功能在模块 flopr readData 中实现，其实就是根据指令的不同，分别取出原始读出数据的某个字节、某半字、整个字，并根据指令进行有符号扩展或无符号扩展。其中，重要的选择信号 ls sel(load store selection) 就是从 ALU 中获得的。ALU 中的 sel 对应选择器的控制信

号, 从数据 RAM 返回的数据中选择到正确的字节或控制 RAM（4 字节宽）上的字节写使能, 如 2。

Algorithm 2 sel

```
assign sel = ((op ==‘EXE_LB_OP || op ==‘EXE_LBU_OP || op ==‘EXE_SB_OP) && (result [1:0] == 2’b00))? 4’b0001:
((op ==‘EXE_LB_OP || op ==‘EXE_LBU_OP || op ==‘EXE_SB_OP) && (result[1:0] == 2’b01))?
    4’b0010:
((op ==‘EXE_LB_OP || op ==‘EXE_LBU_OP || op ==‘EXE_SB_OP) && (result[1:0] == 2’b10))?
    4’b0100:
((op ==‘EXE_LB_OP || op ==‘EXE_LBU_OP || op ==‘EXE_SB_OP) && (result[1:0] == 2’b11))?
    4’b1000:
((op ==‘EXE_LH_OP || op ==‘EXE_LHU_OP || op ==‘EXE_SH_OP) && (result[1:0] == 2’b00))?
    4’b0011:
((op ==‘EXE_LH_OP || op ==‘EXE_LHU_OP || op ==‘EXE_SH_OP) && (result[1:0] == 2’b10))?
    4’b1100:
((op ==‘EXE_LW_OP || op ==‘EXE_SW_OP))? 4’b1111: 4’b0000;
```

在访存指令的执行路径中，ALU 的计算结果即为我们需要读取数据的地址 PC，这一数值以字节为单位，因此需要根据指令的不同对其进行判断，比如对于 LB 指令，读取的是一个字节，而从 DataRAM 中根据 PC 读出的是一个字 (readdataM)，也就是四个字节，因此，如果 PC 模 4 为 0，则应该读取的字节为 readdataM[7:0]，再根据指令进行有符号或无符号扩展得到的结果 readdataW 即为写回阶段真正写入到 regfile 中的数据；如果 PC 模 4 为 1，则应该读取的字节为 readdataM[15:8]；如果 PC 模 4 为 2，则应该读取的字节为 readdataM[23:16]；如果 PC 模 4 为 3，则应该读取的字节为 readdataM[31:24]。再如 LH 指令，读取的是半字，如果 PC 模 4 为 0，则读取下半字，即 readdataM[15:0]，如果 PC 模 4 为 2，则读取上半字，即 readdataM[31:16].

在 store 指令中，我们需要将 rt 寄存器中的值存入 Data RAM 中，写地址与上述读地址的计算方式一样，通过 ALU 计算得到 (这一步与 addi 执行逻辑相似). 由于 rt 寄存器中的值为 32 位，因此与上述 load 指令一样，需要对 rt 寄存器中的值进行处理，处理方法和 load 一样。

此外，ALU 中的 adel 与 ades 判断取值和存储指令是否出现地址错误例外，与例外处理模块相连，如 3。

Algorithm 3 adel&ades

```
//判断取值或加载过程中是否出现地址错误例外（取一个字节时（lb），pc 值不受限制；取半个字时，PC 值是 2 的倍数；
    取一个字时，PC 值是 4 的倍数）

assign adel = ((op ==‘EXE_LH_OP || op ==‘EXE_LHU_OP) && (result[0] == 1’b1)) ||

(op ==‘EXE_LW_OP&& result[1:0] != 2’b00);

//判断存储指令是否出现地址错误例外

assign ades = (op ==‘EXE_SH_OP&& result[0] == 1’b1) || (op ==‘EXE_SW_OP&& result [1:0] != 2’b00);
```

- 对于乘法指令，先将两个乘数转为相应的补码，再直接相乘，所得结果后，根据乘数的符号，选择是否将结果取补码。最后将乘法结果写入 hilo 寄存器，并通过 hilo temp 输出，如 4。

Algorithm 4 mul

```
//乘法操作时，将操作数转为补码

    assign mult_a = ((op ==‘EXE_MULT_OP) && (num1[31] == 1’b1)) ? (~num1 + 1) : num1; assign mult_b = ((op
                ==‘EXE_MULT_OP) && (num2[31] == 1’b1)) ? (~num2 + 1) : num2;

//乘法指令时，根据结果符号判断是否取补码,写入 hilo 寄存器并通过 hilo_temp 输出

assign hilo_temp = ((op ==‘EXE_MULT_OP) && (num1[31] ^ num2[31] == 1’b1)) ? ~(mult_a * mult_b)+ 1: mult_a * mult_b;
```

- 对于除法指令，除法器 div 是一个单独的模块，详情见 div 与 div1 模块设计

(3) 接口定义：

信号名	方向	位宽	功能描述
opcode	input	6bit	指令二进制表示 op 字段的值
funct	input	6bit	指令二进制表示 funct 字段的值
alu control	output	8bit	alu 控制信号，决定 alu 执行运算的类型
is break	output	8bit	判断是否断点
is syscall	output	1bit	判断是否系统调用
is eret	output	1bit	判断是否为例外处理

is reserve	output	1bit	判断是否为保留指令

表 2: alu decoder 接口定义

信号名	方向	位宽	功能描述
num1	input	32bit	第一个源操作数
num2	input	32bit	第二个源操作数
op	input	8bit	alu 的控制信号
sa	input	5bit	移位量，从指令的 [10:6] 位直接连入 ALU 中
result	output	32bit	输出的结果
overflow	output	1bit	溢出信号
adel	output	1bit	取值或加载过程中是否出现地址错误例外
ades	output	1bit	存储指令是否出现地址错误例外
sel	output	4bit	访存类指令对应选择器的控制信号
load usign	output	1bit	取指令的无符号扩展
hilo temp	output	64bit	乘法指令结果的符号判断

表 3: alu 接口定义

2.2.2 div 模块设计

(1) 功能描述：div 是除法运算单元，由于除法运算周期长，因此不能放入 alu 模块中，在此设计了单独的模块 div。

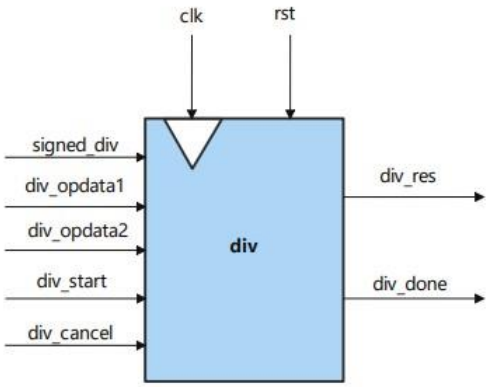


图 8: div 模块图

（2）实现逻辑：除法器通过 signed div 信号，判断是有符号除法还是无符号除法。div opdata1 为被除数，div opdata2 为除数，div res 为运算结果。div start 表示除法开始运算，div cancel 表示是否取消除法运算，这里默认设置不取消，div done 表示除法的运算结果是否算好。

除法实现使用了状态机，分别是初始化状态、计算状态、符号处理状态、运算结束状态。在初始化状态中，将被除数与除数先取补码，再优化除法周期；在计算状态，根据周期用试减法计算，逐步移位，用 64 位的 dividend 存商和余数；在符号处理阶段，判断被除数的符号，确定是否将结果取反加一；在运算结束状态，将 div done 置为 1。

（3）接口定义：

信号名	方向	位宽	功能描述
clk	input	1bit	时钟信号：0 表示下降沿，1 表示上升沿
rst	input	1bit	复位信号：0 表示不复位，1 表示复位
signed div	input	1bit	符号标志，0 表示无符号除法，1 表示有符号除法
div opdata1	input	32bit	源操作数 1
div opdata2	input	32bit	源操作数 2
div start	input	1bit	开始除法操作
div cancel	input	1bit	取消除法操作
div res	output	64bit	除法运算的结果
div done	output	1bit	除法完成的控制信号

表 4: div 接口定义

2.2.3 eqcmp 模块设计

(1) 功能描述：在分支跳转指令中，首先进行分支跳转指令条件是否成立的判断，其中：分支指令的成立条件各不相同，我们通过修改计组 lab4 标准代码中已有的 eqcmp 模块实现。在 lab4 中，eqcmp 模块实现的功能是判断两个操作数是否相等，用于 BEQ 指令的分支条件判断。现在，我们加入了对其他 7 条分支指令是否跳转进行判断，最终 eqcmp 可以判断所有分支指令是否满足跳转条件，辅助分支指令的执行。

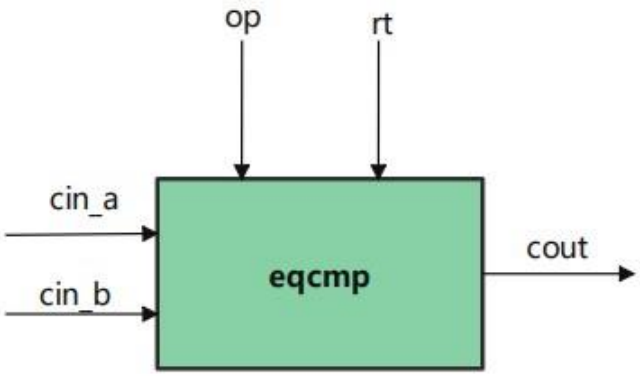


图 9: eqcmp 模块图

实现逻辑：如图 9 所示，输入 cin a 和 cin b 两个源操作数后，eqcmp 就根据 op 和 rt 字段判断需要执行哪一个分支指令的比较判断操作。最后输出是否满足跳转条件信号, 即 cout。

Algorithm 5 Eqcmp

```
assign cout = (op ==‘EXE_BEQ) ? (cin_a == cin_b) :
(op ==‘EXE_BNE) ? (cin_a != cin_b):
// 非负且不等于 0
(op ==‘EXE_BGTZ) ? ((cin_a[31] == 1’b0) && (cin_a != 32’b0)):
// 非正
(op == ‘EXE_BLEZ) ? ((cin_a[31] == 1’b1) || (cin_a == 32’b0)): // opcode 不能唯一识别, 需要继续判断 rt 的值
((op ==‘EXE_REGIMM_INST) && ((rt ==‘EXE_BGEZ) || (rt ==‘EXE_BGEZAL))) ?(cin_a[31] == 1’b0) :
((op ==‘EXE_REGIMM_INST) && ((rt == ‘EXE_BLTZ) || (rt ==‘EXE_BLTZAL))) ? ((cin_a[31] == 1’b1)):
0;
```

(3) 接口定义：

信号名	方向	位宽	功能描述
cin a	input	32bit	源操作数 1
cin b	input	32bit	源操作数 2
op	input	6bit	op 字段的二进制表示
rt	input	5bit	rt 字段的二进制表示
cout	output	1bit	满足跳转条件与否

表 5: eqcmp 接口定义

2.2.4 hilo 寄存器设计

(1) 功能描述：hilo 寄存器用于乘除法的计算。

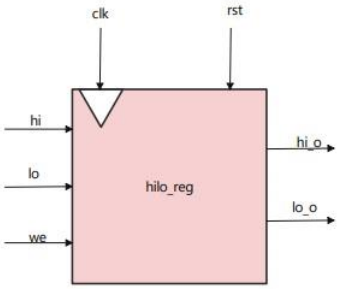


图 10: hilo 模块图

(2) 实现逻辑:：hilo 寄存器可以输入 32 位的高位 hi 或低位 lo，输出 32 位的 hi_o 和 lo_o。当要写 hilo 寄存器时，使能信号 we 为 1，并将 hi 和 lo 分别赋值给 hi_o 和 lo_o。hi 寄存器用于存储乘法的运算结果的高 32 位，lo 则是低 32 位。hi 存储的是除法运算的余数，lo 存储的是除法的商。MFXX 类型将 hi_o、lo_o 写入到通用寄存器，MTXX 类型将通用寄存器中的值写入到 hi、lo 寄存器中。

(3) 接口定义：

信号名	方向	位宽	功能描述
clk	input	1bit	时钟信号：0 代表下降沿，1 代表上升沿
rst	input	1bit	复位信号：0 代表不复位，1 代表复位
we	input	1bit	hilo 寄存器写使能信号
hi	input	32bit	输入的 hi 寄存器中存储的值
lo	input	32bit	输入的 lo 寄存器中存储的值
hi o	output	32bit	输出的 hi 寄存器中存储的值
lo o	output	32bit	输出的 lo 寄存器中存储的值

表 6: hilo reg 接口定义

2.2.5 处理器例外处理

完成 52 条指令后，我们可以需要为处理器增加例外处理的功能。在 MIPS32 中，中断、陷阱、系统调用和任何可以中断程序正常执行流的情况都称为例外 (exception)。例外处理的过程在操作系统中已经有过介绍，可以使用下面的流程图说明：

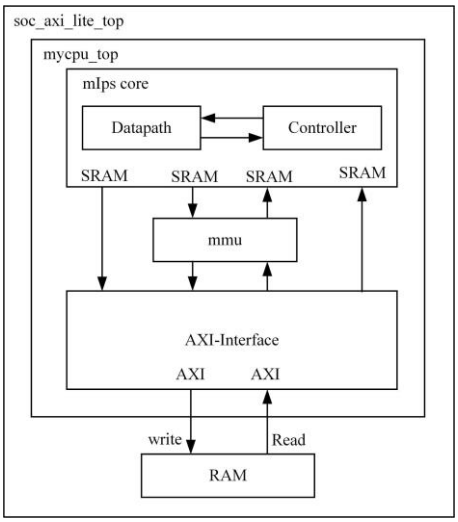


图 11: 例外处理流程图

总体来说，即发生例外后，处理器进入例外入口地址 (在 MIPS32 中统一为 0xBFC00380)，从例外入口重新取指，软件开始执行例外处理程序。在操作系统课程中，我们主要实现右半部分的 handler，即软件处理例外。而在硬件综合设计课程中，在设计处理器时主要实现左半部分，即处理器硬件响应例外的过程。

在介绍 MIPS32 例外处理之前，我们首先需要学习协处理器 CP0(Co-Processor 0).MIPS32 架构提供了最多 4 个协处理器，其作用不同，CP0 主要负责系统控制，我们需要完成的例外控制也是系统控制的一部分。例外发生时的检测和处理都由 CP0 中的一些控制寄存器来定义和控制。

CP0 及 exception type mux 模块 (用于确定例外类型) 数据通路图如下：

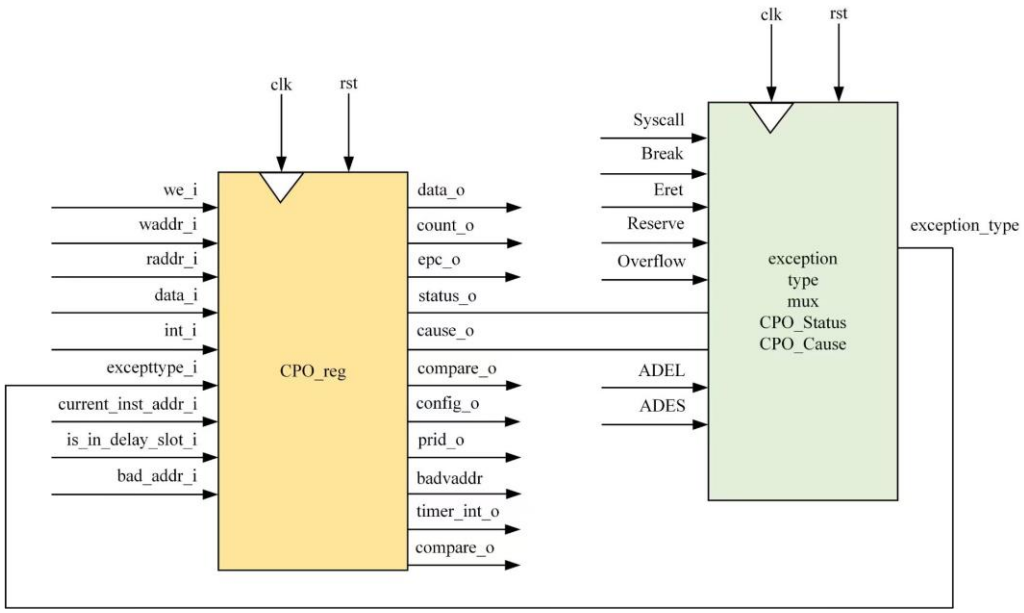


图 12: 例外处理数据通路图

2.2.6 发生例外后根据例外类型对 CP0 的修改

CP0 在我们的设计中类似一个寄存器堆 (regfile)，可以通过根据输入的异常控制信号对其进行读写，更新或读取 Count、EPC、Compare、Status、Cause 等寄存器。

MIPS 32 中的例外分为以下几种，在具体介绍例外的处理过程之前，首先介绍一个概念：精准异常，即处理器会精准地最先发生例外的指令，实现精准异常需要使用 EPC(Exception PC) 寄存器，EPC 中存储的是上一次发生例外指令的 PC 值，但如果发生例外的指令位于延迟槽内，则 EPC 中存储其上一条分支跳转指令的 PC。

代码中写 EPC 寄存器的逻辑如下：

Algorithm 6 写 EPC 寄存器

```
if(is_in_delayslot_i == 'InDelaySlot) begin
epc_o <= current_inst_addr_i - 4;     // 发生例外的指令位于分支延迟槽中，将其上一条分支跳转指令的 PC 存入 EPC 寄存器
cause_o[31] <= 1'b1; // Cause.BD 当发生异常的指令位于分支延迟槽时，该字段被置为 1 end
else begin
epc_o <= current_inst_addr_i; // 发生例外的指令不位于分支延迟槽内，将该指令的 PC 存入 EPC 寄存器
    cause_o[31] <= 1'b0;                                    // Cause.BD 当发生异常的指令不位于分支延迟槽时，该字段被置为
      0
end
```

代码中的 cause o[31] 对应 Cause 寄存器中的 BD(Branch DelaySlot)，只有当发生例外的指令位于分支延迟槽内，其值被置为 1。

当发生例外时，我们需要将 Cause.ExcCode 修改为对应的例外编码，这也是每一种例外发生时都要进行的。本次项目实现的例外对应的编码如下表：

ExcCode 编码	助记符	描述
0	Interrupt	中断例外
4	AdEL	加载或取指过程中，地址错误例外
5	AdES	存储过程中，地址错误例外

8	Syscall	系统调用例外
9	Break	断点例外
10	Reserve	保留指令例外
12	Overflow	整数溢出例外
13	Trap	自陷指令引起的例外

表 7: 例外对应编码

除此之外，发生例外后，还需要将 Status.EXL(Exception Level) 修改为 1，表示处理器处于异常级，此时会禁止新的中断。

1. 中断例外

Algorithm 7 中断例外

```
status_o[1] <= 1'b1;                    // 写 Status.EXL 表示是否位于异常级 为 1 则处理器进入内核模式工作
                                        并且禁止中断
                                        // 写 Cause.ExcCode 记录发生了哪种异常
cause_o[6:2] <= 5'b00000;
```

2. 地址错误例外

(1) 加载或取指过程

Algorithm 8 加载或取指过程

```
status_o[1] <= 1'b1; cause_o[6:2] <=
5'b00100; badvaddr <= bad_addr_i;
```

(2) 存储过程

Algorithm 9 存储过程

```
status_o[1] <= 1'b1; cause_o[6:2] <=
5'b00101; badvaddr <= bad_addr_i;
```

地址错误指令还需额外将 BadVAddr 寄存器的值修改为最近一次地址相关例外的地址。

3. 系统调用例外

Algorithm 10 系统调用例外

```
status_o[1] <= 1'b1; cause_o[6:2] <=
5'b01000;
```

4. 断点例外

Algorithm 11 断点例外

```
status_o[1] <= 1'b1; cause_o[6:2] <=
5'b01001;
```

5. 保留指令例外

Algorithm 12 保留指令例外

```
status_o[1] <= 1'b1; cause_o[6:2] <=
5'b01010;
```

6. 整形溢出例外

Algorithm 13 整形溢出例外

```
status_o[1] <= 1'b1; cause_o[6:2] <=
5'b01100;
```

7. 自陷指令例外

Algorithm 14 自陷指令例外

```
status_o[1] <= 1'b1; cause_o[6:2] <=
5'b01101;
```

2.2.7 确定发生例外的类型

我们需要在数据通路中判断当前指令是否发生例外，以及例外类型。

1. 中断例外中断例外的发生有 CP0 中多个寄存器决定:

Algorithm 15 中断例外

```
if (((CP0_Cause[15:8]&CP0_Status[15:8])!=8'b00000000)&&(CP0_Status[1]!=1'b1)&&(
      CP0_Status[0]==1'b1)) begin
exception_type <= 32'h00000001;
// 中断例外 按规范中断的 ExcCode 编码应该为 0，这里为了方便后面直接根据 exception_type(!=0) 判断是否有例外发生，因
      此将其改为 1
    // CP0.Status.IM                    Interrupt Mask 中断屏蔽信号 0 表示屏蔽 1 表示不屏蔽
    // CP0.Cause.IP                     Interrupt Pending 中断挂起字段 1 表示发生 0 表示不发生
// CP0.Status.EXL Exception Level 为 1 时表示正处于异常级，处理器会进入内核模式工作，此时禁止中断
    // CP0.Status.IE                    Interrupt Enable 中断使能信号 1 中断使能
end
```

即在处理器挂起中断、且处理器不屏蔽相应中断、中断使能为 1、处理器不位于异常级时，发生中断例外。

2. 地址错误例外

(1) 加载或取指过程

Algorithm 16 加载或取指过程

```
if (AdEL) begin

// 加载或取指过程中，地址错误例外 AdEL (Address Exception Load) exception_type <=
32'h00000004; end
```

加载或取指过程发生地址错误例外信号 AdEL 是由数据通路中的 pc adelM 和 load adelM 信号做或运算得到的。 a、pc adelM 信号是由 IF 阶段的 pc adelF 通过流水线传过来的:

Algorithm 17 IF 阶段

```
assign pc_adelF = (pcF[1:0] != 2'b00); address // 加载或取指过程中，地址错误例外 ADEL (
    exception load)
```

b、load adelM 信号是由 EXE 阶段的 load adelE 通过流水线传过来的，load adelE 是在 ALU 中计算出的：

Algorithm 18 EXE 阶段

```
assign load_adel = ((op ==`EXE_LH_OP || op ==`EXE_LHU_OP) && (result[0] == 1'b1)) || (op ==`EXE_LW_OP&&
result[1:0] != 2'b00);

// ALU 中的 result 在 load 指令中最后会成为读 Data RAM 的读地址(PC)
```

load adelE 表示在 load 指令中是否出现地址错误例外: 如果 load 指令取一个字节，即 LB 类型，那么 PC 值任意，不会发生地址错误例外；如果 load 指令取半字，那么 PC 值只能是 2 的倍数，否则发生地址错误例外；如果 load 指令取整字，那么 PC 值只能是 4 的倍数，否则发生地址错误例外。以上 pc adelM load adelM 任一为 1，则说明加载或取指过程中发生地址错误例外。

Algorithm 19 存储过程

```
if (AdES) begin

// 存储过程中，地址错误例外 ADES (address exception store) exception_type <=
32'h00000005; end
```

存储过程中发生地址错误例外信号 AdES 信号是由数据通路中的 adesM 传入； adesM 信号是由 EXE 阶段的 adesE 信号通过流水线传过来的；adesE 信号是在 ALU 中计算得到的:

Algorithm 20 存储过程

```
assign ades = (op ==‘EXE_SH_OP&& result[0] == 1’b1) || (op ==‘EXE_SW_OP&& result[1:0]
!= 2’b00);
```

判断方法与上述 load 指令相同。

3. 系统调用例外

Algorithm 21 系统调用例外

```
if (Syscall==1’b1) begin
// 系统调用例外 Syscall
exception_type <= 32’h00000008; end
```

系统调用例外信号由数据通路中的 syscallM 传入，而 syscallM 是由 ID 阶段的 syscallD 信号通过流水线传过来的，syscallD 信号是在 alu decoder 模块译码时得到:

Algorithm 22 系统调用例外

```
‘EXE_SYSCALL: begin
// 系统调用例外
alu_control <= ‘EXE_SYSCALL_OP; is_syscall <= 1’b1;
end
```

4. 断点例外

Algorithm 23 断点例外

```
if (Break==1'b1) begin
// 断点例外 Break
exception_type <= 32'h00000009; end
```

系统调用例外信号由数据通路中的 breakM 传入，而 breakM 是由 ID 阶段的 breakD 信号通过流水线传过来的，breakD 信号是在 alu decoder 模块译码时得到:

Algorithm 24 断点例外

```
'EXE_BREAK: begin
// 断点例外
alu_control <='EXE_BREAK_OP; is_break <= 1'b1;
end
```

5. 保留指令例外保留指令例外是指传入的指令在 MIPS 32 中没有定义，即处于保留状态。

Algorithm 25 保留指令例外

```
if (Reserve==1'b1) begin
// 保留指令例外 Reserve
exception_type <= 32'h0000000a; end
```

保留指令例外信号由数据通路中的 reserveM 传入，而 reserveM 是由 ID 阶段的 reserveD 信号通过流水线传过来的，reserveD 信号是在 alu decoder 模块译码时得到:

Algorithm 26 保留指令例外

```
default: begin
// 保留指令例外
alu_control <= 8'bxxxxxxxx;
is_reserve <= 1'b1; end
```

即当传入的指令不是我们添加的 57 条指令其中之一时触发保留指令例外。

6. 整形溢出例外整形溢出例外是指在有符号加法或减法中发生溢出的情况:

Algorithm 27 整形溢出例外

```
if (Overflow==1'b1) begin
// 整数运算溢出例外 Overflow
exception_type <= 32'h0000000c; end
```

整数溢出例外信号由数据通路中的 overflowM 传入，在 overflowM 是由 EXE 阶段的 overflowE 信号通过流水线传过来的，overflowE 信号是在 ALU 中计算出来的:

Algorithm 28 整形溢出例外

```
assign overflow =
(op ==‘EXE_ADD_OP || op ==‘EXE_ADDI_OP) ?
((num1[31]&num2[31]&(~result[31]))|((~num1[31])&(~num2[31])&result[31])) : (op ==‘EXE_SUB_OP) ?
((num1[31]&(~num2[31])&(~result[31]))|((~num1[31])&num2[31]&result[31])) : 1'b0;
```

7. 例外返回

Algorithm 29 例外返回

```
if (Eret==1'b1) begin
// 例外返回 Eret
exception_type <= 32'h0000000e; end
```

例外返回信号由数据通路中的 eretM 传入，在 eretM 是由 ID 阶段的 eretD 信号通过流水线传过来的，eretD 信号是在 alu decoder 中译码时得到的:

Algorithm 30 例外返回

```
case(opcode):
6'b010000:begin
case(funct) // 例外返
回

6'b011000:

is_eret      <=      1'b1;      endcase
alu_control<= 8'b00000000;

end
```

相关接口定义如下：

信号名	方向	位宽	说明
clk	Input	1-bit	时钟信号
rst	Input	1-bit	复位信号
we i	Input	1-bit	CP0 寄存器的写使能
waddr i	Input	5-bits	写 CP0 中寄存器的地址
raddr i	Input	5-bits	读 CP0 中寄存器的地址
data i	Input	32-bits	写入 CP0 中寄存器的数据
int i	Input	6-bits	6 个硬件外部中断输入
excepttype i	Input	32-bits	发生例外类型

current inst addr i	Input	32-bits	发生例外的指令的地址
is in delayslot i	Input	1-bit	发生例外的指令是否位于延迟槽
bad addr i	Input	32-bits	记录最近一次地址相关例外的地址
data o	Output	32-bits	从 CP0 中读出的某寄存器的值
count o	Output	32-bits	Count 寄存器的值处理器计数周期
compare o	Output	32-bits	Compare 寄存器的值定时终断控制
status o	Output	32-bits	Status 寄存器的值处理器状态和控制寄存器
cause o	Output	32-bits	Cause 寄存器的值保存上一次例外原因
epc o	Output	32-bits	EPC 寄存器的值保存上一次例外时的 PC
config o	Output	32-bits	Config 寄存器的值配置寄存器
prid o	Output	32-bits	PRId 寄存器的值处理器标志和版本
badvaddr	Output	32-bits	BadVAddr 寄存器的值记录最近一次地址相关异常的地址
timer int o	Output	1-bit	时钟中断

表 8: cp0 reg 接口定义

信号名	方向	位宽	说明
rst	Input	1-bit	时钟信号
clk	Input	1-bit	复位信号
Syscall	Input	1-bit	是否发生系统调用例外
Break	Input	1-bit	是否发生断点例外
Eret	Input	1-bit	是否发生例外返回
Reserve	Input	1-bit	是否发生保留指令例外
Overflow	Input	1-bit	是否发生整形溢出例外
AdEL	Input	1-bit	加载或取指过程是否发生地址错误例外
AdES	Input	1-bit	存储过程是否发生地址错误例外
CP0 Status	Input	32-bits	CP0 中的 Status 寄存器
CP0 Cause	Input	32-bits	CP0 中的 Cause 寄存器
exception type	Output	32-bits	例外类型

表 9: exception type mux 接口定义

2.2.8 冒险模块设计

(1) 主要功能：在流水线中，若当前指令取决于前一条指令的结果，但此时前一条指令并未产生结果，此时就会产生冒险现象。冒险可分为：(1) 数据冒险：寄存器中的值还未写回到寄存器堆中，下一条指令已经需要从寄存器堆中读取数据，主要通过数据前推和暂停流水线解决。(2) 控制冒险：下一条要执行的指令还未确定，就按照 PC 自增顺序执行了本不该执行的指令 (由分支指令引起)。冒险模块主要通过数据前推及流水线暂停来解决以上问题，并在判断输入的异常类型后为例外入口地址赋值。

(2) 实现逻辑：主要通过 forward、stall 及 flush 信号来控制数据前推、流水线的暂停和阻塞。

- Forward 信号用于判断是否有数据冒险的情况发生，可以控制流水线执行数据前推操作，实现代码如下：

Algorithm 31 Forward 信号

```
assign forwardaD = (rsD != 0) && (rsD == writeregM) && regwriteM; assign forwardbD = (rtD != 0) && (rtD
== writeregM) && regwriteM; assign forwardaE = ((rsE != 0) && (rsE == writeregM) && regwriteM) ? 2'b10 :
((rsE != 0) && (rsE == writeregW) && regwriteW) ? 2'b01 :

2'b00;

assign forwardbE = ((rtE != 0) && (rtE == writeregM) && regwriteM) ? 2'b10 : ((rtE != 0) && (rtE == writeregW)
&& regwriteW) ? 2'b01 :

2'b00;
```

- Stall 信号可控制流水线阻塞，主要用于数据前推无法解决冒险时，将流水线暂停至可继续的时间。需要 stall 的情况可具体分为以下几种：(1) 访存指令与其下一条指令发生数据冒险 (2) 分支指令及 JR 指令 D 阶段需要从寄存器读取数据，但寄存器的值正在 E 阶段被计算 (3) 除法指令进行到 E 阶段 需要从指令和数据存储器读取数据，但存储器还未准备就绪时需要暂停。stall 信号实现代码如下：

Algorithm 32 Stall 信号

```
assign lwstall = ((rsD == rtE) || (rtD == rtE)) && (memtoregE | readcp0E); assign brachstall = (branchD && regwriteE &&
(writeregE == rsD || writeregE == rtD))|| ( branchD && (memtoregM | readcp0M) && (writeregM == rsD || writeregM == rtD));

assign jrstall = (jrD && regwriteE && (writeregE == rsD))|| (jrD && (memtoregM | readcp0M) && (writeregM == rsD));

assign stallF=lwstall || brachstall || stall_divE || jrstall || stallreq_from_if || stallreq_from_mem;

assign stallD = lwstall || brachstall || stall_divE || jrstall || stallreq_from_if || stallreq_from_mem;
```

```
assign flushE = stall_divE ? 0 : (lwstall || brachstall || jrstall || flush_except || stallreq_from_if);

assign stallE = stall_divE || stallreq_from_mem; assign stallM =
stallreq_from_mem;
```

- flush 信号可控制流水线的刷新操作，主要在产生异常时进行刷新或在有错误数据时刷新流水线，实现代码如下：

Algorithm 33 flush 信号

```
wire flush_except; assign flush_except = (excepttypeM != 0);
assign flushF = flush_except; assign flushD = flush_except;
assign flushM = flush_except;

assign flushW = flush_except || stallreq_from_mem;
```

(3) hazard 接口定义

信号名	方向	位宽	功能描述
stallF	Output	1bit	取指阶段阻塞流水线判断信号
flushF	Output	1bit	取指阶段刷新流水线判断信号
rsD	Input	5bit	译码阶段信号，指令的 [25:21] 位
rtD	Input	5bit	译码阶段信号，指令的 [20:16] 位
branchD	Input	1bit	译码阶段分支指令判断信号
forwardaD	Output	1bit	译码阶段信号，第一个操作数的前推信号
forwardbD	Output	1bit	译码阶段信号，第二个操作数的前推信号
stallD	Output	1bit	译码阶段阻塞流水线判断信号
jrD	Input	1bit	译码阶段信号跳转寄存器指令判断信号
flushD	Output	1bit	译码阶段信号刷新流水线判断信号
rsE	Input	5bit	执行阶段信号，指令的 [25:21] 位
rtE	Input	5bit	执行阶段信号，指令的 [20:16] 位
writeregE	Input	5bit	执行阶段信号，要写回的寄存器号

regwriteE	Input	1bit	执行阶段信号，寄存器堆写使能
memtoregE	Input	1bit	执行阶段寄存器写入数据来源信号
stall divE	Input	1bit	执行阶段信号，除法指令的阻塞指令
forwardaE	Output	2bit	执行阶段第一个操作数的前推信号
forwardbE	Output	2bit	执行阶段第二个操作数的前推信号
flushE	Output	1bit	执行阶段刷新流水线判断信号
stallE	Output	1bit	执行阶段阻塞流水线判断信号
readcp0E	Input	1bit	执行阶段信号，Cp0 寄存器读使能
writeregM	Input	5bit	访存阶段信号，要写回的寄存器号
regwriteM	Input	1bit	访存阶段信号，寄存器堆写使能
memtoregM	Input	1bit	访存阶段写入寄存器数据来源信号
readcp0M	Input	1bit	访存阶段信号，Cp0 寄存器读使能
excepttypeM	Input	32bit	访存阶段信号，例外类型
flushM	Output	1bit	访存阶段刷新流水线判断信号
except pc	Output	32bit	访存阶段信号，例外入口地址
epc oM	Input	32bit	访存阶段信号，例外处理返回地址
stallM	Output	1bit	访存阶段阻塞流水线判断信号
writeregW	Input	5bit	写回阶段信号，要写回的寄存器号
regwriteW	Input	1bit	写回阶段信号，寄存器堆写使能
flushW	Output	1bit	写回阶段刷新流水线判断信号
stallreq from if	Input	1bit	指令存储器阻塞信号
stallreq from mem	Input	1bit	数据存储器阻塞信号

表 10: hazard 接口定义

2.3 controller 模块设计

(1) 主要功能: 在本模块中生成流水线需要的元件控制信号。主要包含两部分，分别为 main decoder 和 alu decoder 模块。main decoder 负责判断指令类型，并生成相应的控制信号。alu decoder 负责 ALU 模块控制信号的译码。模块图如下图所示：

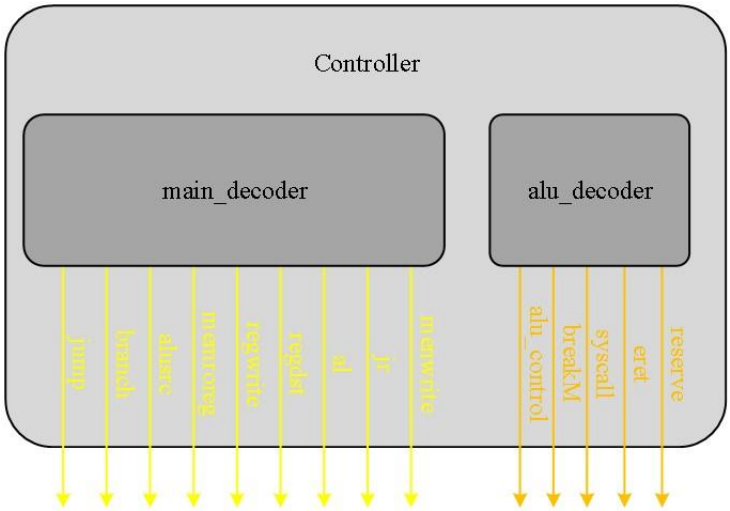


图 13: controller 模块图

(2) 实现逻辑 ● controller 模块执行过程中除了调用 main decoder 和 alu decoder 模块，还需要为译码阶段的 readhiloD、hilo selD 等控制信号赋值，实现代码如下：

Algorithm 34 controller

```
assign logI = (opD[3:2] == 2'b11 );

assign readhiloD = ((opD ==‘EXE_NOP) && ((functD == 6'b010000) || (functD == 6'b010010)

    ) ); assign hilo_selD = ((opD ==‘EXE_NOP) && (functD == 6'b010000));//选择 hi 还是 lo assign hilo_srcD = ((opD
==‘EXE_NOP) && (functD == 6'b010001 || functD == 6'b010011)) ? 2'b00://rs

((opD ==‘EXE_NOP) && (functD == 6'b011000 || functD == 6'b011001)) ? 2'b01://乘法

2'b10;//除法

assign writehiD = ((opD ==‘EXE_NOP) && (functD == 6'b010001 || functD == 6'b011000 || functD == 6'b011001 || functD ==
    6'b011010 || functD == 6'b011011) );

assign writeloD = ((opD ==‘EXE_NOP) && (functD == 6'b010011 || functD == 6'b011000 || functD == 6'b011001 ||functD ==
    6'b011010 || functD == 6'b011011) );

assign readcp0D = (opD == 6'b010000) &&(rsD == 5'b00000); assign writecp0D = (opD
== 6'b010000) && (rsD == 5'b00100);
```

并通过触发器将控制信号传入下一阶段，实现代码如下：

Algorithm 35 controller

```
flopenrc                                                                                    #(27)
        regdE(.clk(clk) ,.rst(rst) ,.enable(~stallE) ,.clear(flushE) ,.cin({memtoregD ,memwriteD,alusrcD,regdstD,regwriteD,alucontr
        olD,writehiD,writeloD,hilo_srcD,  hilo_selD,readhiloD,alD,jrD,  writecp0D,readcp0D,breakD,syscallD,eretD,reserveD}) ,.
        cout({memtoregE,memwriteE,alusrcE,regdstE,regwriteE,alucontrolE,writehiE,writeloE,
        hilo_srcE,hilo_selE,readhiloE,alE,jrE,writecp0E,readcp0E,breakE,syscallE,eretE, reserveE}));

flopenrc #(11) regM(.clk(clk) ,.rst(rst) ,.enable(~stallM) ,.clear(flushM) ,.cin({memtoregE,
        memwriteE,regwriteE,readhiloE,hilo_selE,writecp0E,readcp0E,breakE,syscallE,eretE,

        reserveE}) ,.cout({memtoregM,memwriteM,regwriteM,readhiloM,hilo_selM,writecp0M,

        readcp0M,breakM,syscallM,eretM,reserveM}));

floprc #(8) regW(.clk(clk) ,.rst(rst) ,.clear(flushW) ,.cin({memtoregM,regwriteM,readhiloM,
        readcp0M,breakM,syscallM,eretM,reserveM}) ,.cout({memtoregW,regwriteW,readhiloW,
        readcp0W,breakW,syscallW,eretW,reserveW}));
```

- main decoder 模块通过译码得到指令的类型，并生成相应的各元件控制信号，控制信号可作为多个信号输出，也可以拼接后作为一个多位信号输出。这里采用代码内部赋值时将信号拼接为一个多位信号进行赋值，但输出时多个信号独立输出的方式。主要控制信号信息如下表所示：

元件控制信号	信号取 0	信号取 1
Regwrite	寄存器堆不写入数据	寄存器堆要写入数据
Regdst	目的寄存器是 rt	目的寄存器是 rd
Alusrc	alu 的 b 操作数是寄存器堆	alu 的 b 操作数是立即数
Branch	不是分支跳转指令	是分支跳转指令
Memwrite	data 存储器不写入数据	data 存储器要写入数据
Memtoreg	将 alu 结果的值存到寄存器	将 data 存储器的值存到寄存器
Jump	当前指令不是 j 指令	当前指令是 j 指令
al	当前指令不是 jal 指令	当前指令是 jal 指令
jr	当前指令不是 jr 指令	当前指令是 jr 指令

表 11: main decoder 控制信号含义

部分实现代码如下：

Algorithm 36 main decoder

```
assign {regwrite,regdst,alusrc ,branch,memWrite,memtoReg,jump,al,jr} = output_assemble; always @(*)begin case(opcode)

‘EXE_ANDI: output_assemble <= 9’b101000000;

‘EXE_XORI: output_assemble <= 9’b101000000;
```

```
‘EXE_LUI: output_assemble <= 9’b101000000;

‘EXE_ORI: output_assemble <= 9’b101000000;

...

‘EXE_REGIMM_INST: case(rt)

‘EXE_BGEZ: output_assemble <= 9’b000100000;

‘EXE_BLTZ: output_assemble <= 9’b000100000;

‘EXE_BLTZAL: output_assemble <= 9’b100100010; ‘EXE_BGEZAL:
output_assemble <= 9’b100100010; endcase
```

- alu decoder 模块通过译码得到 ALU 模块控制信号，该信号可控制 ALU 模块对输入的数据执行对应操作。为了提高程序可读性，在 defines.vh 文件中通过宏定义指令将信号的值转换为标识符，故需要在模块开头引入该文件。部分实现代码如下：

Algorithm 37 alu decoder

```
always@(*)begin
breakM <= 1’b0; syscall
<= 1’b0; eret <= 1’b0;
reserve <= 1’b0;
case(op) ‘EXE_NOP:
case(funct)

‘EXE_AND:alu_control <=‘EXE_AND_OP; ‘EXE_OR:alu_control
<=‘EXE_OR_OP;

‘EXE_XOR:alu_control <=‘EXE_XOR_OP;

‘EXE_NOR:alu_control <=‘EXE_NOR_OP;

‘EXE_SLL:alu_control <= ‘EXE_SLL_OP;

‘EXE_SRL:alu_control <= ‘EXE_SRL_OP; ‘EXE_SRA:alu_control
                        <= ‘EXE_SRA_OP;

‘EXE_SLLV:alu_control <= ‘EXE_SLLV_OP;
```

(3) 相关接口定义

信号名	方向	位宽	功能描述
clk	Input	1bit	时钟信号
Rst	Input	1bit	复位信号，取 0 为不复位，取 1 为复位
opD	Input	6bit	取指阶段信号，指令 [31:26] 位
functD	Input	6bit	取指阶段信号，指令 [5:0] 位

pcsrcD	Output	1bit	取指阶段程序计数器取值信号
branchD	Output	1bit	取指阶段分支指令判断信号
equalD	Input	1bit	取指阶段是否满足跳转条件判断信号
rtD	Input	5bit	取指阶段信号，指令的 [20:16] 位
rsD	Input	5bit	取指阶段信号，指令的 [25:21] 位
jumpD	Output	1bit	取指阶段跳转指令判断信号
logI	Output	1bit	逻辑运算指令有无立即数判断信号
jrD	Output	1bit	取指阶段跳转寄存器指令判断信号
flushE	Input	1bit	译码阶段刷新流水线判断信号
stallE	Input	1bit	译码阶段阻塞流水线判断信号
memtoregE	Output	1bit	译码阶段写入寄存器数据来源信号
alusrcE	Output	1bit	译码阶段 alu 操作数判断信号
regdstE	Output	1bit	译码阶段目标寄存器选择信号
regwriteE	Output	1bit	译码阶段信号，寄存器堆写使能
alucontrolE	Output	8bit	译码阶段信号，Alu 控制信号
writehiE	Output	1bit	译码阶段信号，hi 寄存器写使能
writeloE	Output	1bit	译码阶段信号，lo 寄存器写使能
hilo srcE	Output	2bit	译码阶段指令类型判断信号
alE	Output	1bit	译码阶段 jal 指令判断信号
jrE	Output	1bit	译码阶段 jr 指令判断信号
readcp0E	Output	1bit	译码阶段信号，Cp0 寄存器读使能
memtoregM	Output	1bit	访存阶段寄存器写入数据来源判断信号
memwriteM	Output	1bit	访存阶段数据存储器是否写入数据判断信号
regwriteM	Output	1bit	访存阶段信号，寄存器堆写使能
readhiloM	Output	1bit	访存阶段信号，hilo 寄存器读使能
hilo selM	Output	1bit	访存阶段 hi、lo 寄存器选择信号
writecp0M	Output	1bit	访存阶段信号，cp0 寄存器写使能

readcp0M	Output	1bit	访存阶段信号，cp0 阶段读使能
breakM	Output	1bit	访存阶段 break 指令判断信号
syscallM	Output	1bit	访存阶段 syscall 指令判断信号
eretM	Output	1bit	访存阶段 eret 指令判断信号令
reserveM	Output	1bit	访存阶段范围内指令判断信号
flushM	Input	1bit	访存阶段刷新流水线判断信号
stallM	Input	1bit	访存阶段阻塞流水线判断信号
memtoregW	Output	1bit	写回阶段写入寄存器数据来源信号
regwriteW	Output	1bit	写回阶段信号，寄存器堆写使能
readhiloW	Output	1bit	写回阶段信号，hilo 寄存器读使能
readcp0W	Output	1bit	写回阶段信号，cp0 阶段读使能
flushW	Input	1bit	写回阶段刷新流水线判断信号

表 12: Controller 接口定义

信号名	方向	位宽	功能描述
opcode	Input	6bit	指令 [31:26] 位
funct	Input	6bit	指令 [5:0] 位
jump	Output	1bit	取 0 为不是跳转指令，取 1 是跳转指令
branch	Output	1bit	取 0 为当前指令不是分支指令，取 1 为当前指令是分支指令
alusrc	Output	1bit	alu 第二个操作数选择信号
memtoreg	Output	1bit	写入寄存器数据来源选择信号
regwrite	Output	1bit	寄存器堆写使能
regdst	Output	1bit	目标寄存器选择信号
al	Output	1bit	取 0 为当前指令不是 jal 指令，取 1 为是 jal 指令
jr	Output	1bit	取 0 为当前指令不是 jr 指令，取 1 为是 jr 指令
memwrite	Output	1bit	数据存储器写使能
rt	Input	5bit	指令的 [20:16] 位

rs	Input	5bit	指令的 [25:21] 位

表 13: main decoder 接口定义

信号名	方向	位宽	功能描述
op	Input	6bit	指令 [31:26] 位
funct	Input	6bit	指令 [5:0] 位
alu control	Output	8bit	Alu 控制信号
breakM	Output	1bit	取 0 为当前指令不是 break 指令，取 1 为当前指令是 break 指令
syscall	Output	1bit	取 0 为当前指令不是 syscall 指令，取 1 为当前指令是 syscall 指令
eret	Output	1bit	取 0 为当前指令不是 eret 指令，取 1 为当前指令是 eret 指令
reserve	Output	1bit	取 0 为当前指令是范围内的指令，取 1 为当前指令是范围外的指令

表 14: alu decoder 接口定义

2.4 AXI 接口模块设计

(1) 主要功能：AXI 包含 Master 端和 Slave 端，它将读/写请求与读/写结果相互分离、将数据写入和数据读出的信号相分离，可以同时进行写入和读出动作，从而最大限度地提高总线的数据吞吐率。可分为五个通道：写地址通道，信号以 AW 开头、写数据通道，信号以 W 开头、写响应通道，信号以 R 开头、读地址通道，信号以 AR 开头、读数据操作，信号以 R 开头、全局信号，包括时钟信号 ACLK，和 ARESETn。

(2) 实现逻辑：AXI 读请求在 ARREADY 和 ARVALID 同时为 1 时完成地址握手，当 RREADY 和 RVALID 同时为 1 的每个上升沿进行一次数据传输，具体实现流程图如下图所示：

图 14: AXI 读请求流程图

部分实现代码如下：

Algorithm 38 AXI 读请求

```
  assign arid = 4'b0;

assign araddr = read_addr; assign
arlen = 8'b0; assign arsize =
read_size; assign arburst = 2'b01;
assign arlock = 2'b0; assign arcache
= 4'b0; assign arprot = 3'b0;

assign arvalid = read_req && ~read_addr_finish && ~flush && ~flush_reg; assign rready = 1'b1;
```

写请求在 AWREADY 和 AWVALID 同时为 1 时实现地址握手，当 WREADY 和 WVALID 同时为 1 的每个上升沿进行一次数据传输，具体实现流程与读请求相似，但在读请求的基础上改变了信号并添加了响应信号部分：

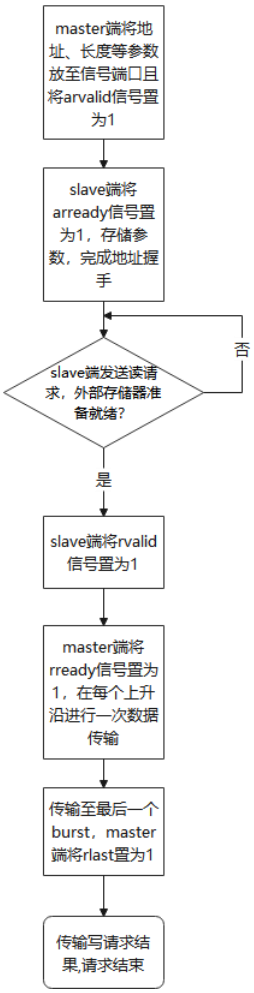


图 15: AXI 写请求流程图

部分实现代码如下：

Algorithm 39 AXI 写请求

```
assign awid = 4'b0; assign awaddr =
write_addr; assign awlen = 8'b0;
assign awsize = write_size; assign
awburst = 2'b01; assign awlock =
2'b0; assign awcache = 4'b0; assign
awprot = 3'b0;

assign awvalid = write_req && ~write_addr_finish; assign wid = 4'b0;
assign wdata = write_data; assign wstrb = write_wen; assign wlast =
1'b1;

assign wvalid = write_req && ~write_data_finish; assign bready = 1'b1;
```

(3) axi interface 接口定义

信号名	方向	位宽	功能描述
clk	Input	1bit	时钟信号
resetn	Input	1bit	重置信号
mem a	Input	32bit	取 0 为可用存储地址，取 1 为指令地址
mem access	Input	1bit	取 0 表示数据存储器连通性，取 1 表示指令存储器连通性
mem write	Input	1bit	数据存储器写使能，默认取 1'b0
mem size	Input	2bit	数据存储器的大小，默认取 2'b10
mem sel	Input	4bit	扩展控制信号, 默认取 4'b1111
mem ready	Output	1bit	存储器是否准备就绪
mem st data	Input	32bit	待写入的数据
mem data	Output	32bit	读出的数据
flush	Input	1bit	流水线刷新信号
arid	Output	4bit	读请求 ID，默认 4'b0
araddr	Output	32bit	读请求最低字节的地址
arlen	Output	8bit	读请求传输长度
arsize	Output	3bit	读请求传输字节大小

arburst	Output	2bit	传输类型，默认 2’b01
arlock	Output	2bit	加锁信号，默认 2’b0
arcache	Output	4bit	Cache 属性，默认 4’b0
arprot	Output	3bit	保护属性，默认 3’b0
arvalid	Output	1bit	读请求地址是否有效
arready	Input	1bit	Slave 端是否做好接收地址的准备
rid	Input	4bit	与 arid 相同（可忽略）
rdata	Input	32bit	从 slave 端读出的数据
rresp	Input	2bit	读请求响应信号（可忽略）
rlast	Input	1bit	传输结束信号（可忽略）
rvalid	Input	1bit	Slave 端是否准备好发送数据（读请求的数据是否有效）
rready	Output	1bit	Master 端是否准备好接收数据
awid	Output	4bit	写请求 ID，默认 4’b0
awadd	Output	32bit	写请求最低字节的地址
awlen	Output	4bit	写请求传输长度
awsize	Output	3bit	写请求传输数据大小
awburst	Output	2bit	传输类型，默认 2’b00
awlock	Output	2bit	加锁信号，默认 2’b00
awcache	Output	4bit	Cache 类型，默认 4’b00
awprot	Output	3bit	保护信号，默认 3’b00
awvalid	Output	1bit	写请求地址是否有效
awready	Input	1bit	Slave 端是否准备好接收地址
wid	Output	4bit	与 awid 相同（可忽略），默认 4’b0
wdata	Output	32bit	写入 salve 的数据
wstrb	Output	4bit	写字节使能，每一个 bit 代表 4 个字节中其中一位的使能信号
wlast	Output	1bit	传输的结束信号（可忽略），默认 1’b1
wvalid	Output	1bit	Master 端是否完成发送数据（写请求数据有效）
wready	Input	1bit	Slave 端准备好接受数据传输

bid	Input	4bit	与 wid、awid 相同，可忽略
bresp	Input	2bit	写响应（可忽略）
bvalid	Input	1bit	写响应是否有效
bready	Output	1bit	Master 端准备好接受写响应

表 15: axi interface 接口定义

2.5 基础 cache 实现

(1) 主要功能：在 cpu 实现中加入 cache, 可以提高数据访问速度，减少对主存的访问次数，提高指令执行效率并且缓解内存带宽压力和提高系统的可扩展性。

(2) 实现逻辑：资料包中已经给出了 icache 和 dcache 的状态机实现，主要有 IDLE: 等待请求状态，WRIBK: 正在向内存写回数据，READIN: 正在向内存读入数据，

FIN:cache 读写完成。直接使用资料包中给出的 cache 模块，通过转接桥，与 cpu 和 axi 接口相连，实现基础 cache。

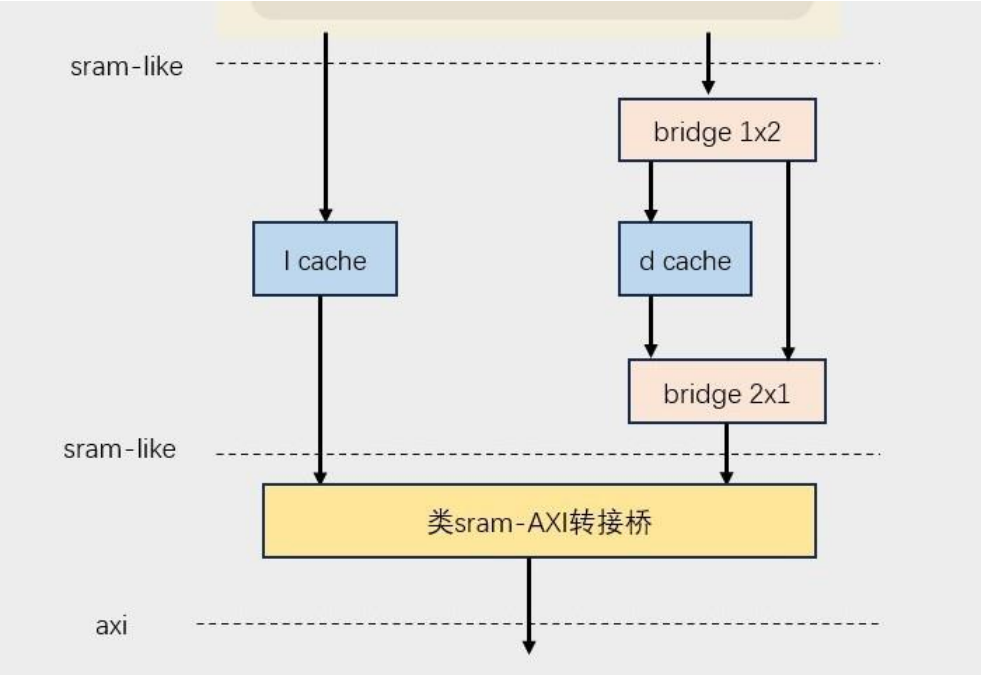


图 16: cache 与 axi 接口的连接

3 设计过程

3.1 设计日志

2023.12.26 共同阅读了实验要求及资料并配置完成实验环境，观看了教学视频。

2023.12.27 对实验 4 进行回顾，继续观看教学视频，对实验的整体进行初步了解。确定了 MIPS 的整体方向，并打算从 52 条指令开始做起，初步分工如下：刘成负责扩展到 57 条指令、连接 AXI 接口、实现基础 cache；丁超负责实现乘除法指令和数据移动指令、访存指令、连接 SARAM-SOC 接口；余肖杨负责实现逻辑指令、移位指令、普通算术指令、分支跳转指令；然后，我们共同进行指令单独仿真测试、功能测试、性能测试和上板性能测试。

2023.12.28 小组开始阅读《自己动手写 CPU》这本书，对整体流程有了大致的了解。小组成员首先尝试完成自己负责部分指令的添加，并将代码放入共享文档中以便后期整合。

2023.12.29-2024.1.3 继续阅读相关文档，完成指令；涂仁桦生病住院，分支跳转指令交由李晓璐添加和测试。

2024.1.4 代码合并大致完成，但是分支跳转和涉及冒险模块的变量部分仍然存在问题，继续修改。当天晚上通过分支跳转部分的测试，例外处理未完全解决。

2024.1.5 52 条指令添加完成，并成功通过测试，尝试扩展到 57 条指令，包括内陷指令和特权指令。继续阅读指导书和查找相关资料。同时，小组开始绘制流水线各个阶段的数据通路图，整理信号量及其说明。

2024.1.6 绘制完成流水线各个阶段的数据通路图，57 条指令成功添加，SRAM 接口连接成功，功能测试通过，性能测试却出现问题。当天继续尝试 AXI 接口的替换，未成功。

2024.1.7-2024.1.8 成功连接 AXI 接口和实现 cache，跑通功能测试和性能测试，综合项目，完成上板。

2024.1.9 对代码进行调整优化，熟悉代码，整理思路，准备答辩。

2024.1.11-2024.1.16 撰写实验报告，对实验文件进行整理打包，提交。

3.2 错误记录

3.2.1 错误 1

(1) 错误现象: 在添加乘法指令时，HILO 寄存器对应的输出全部是不定态 x。

(2) 分析定位过程: 在 vivado 中拉取更多的控制信号，逐一进行检查核对。

(3) 错误原因: 拉取到 alucontrol 时，发现之前将其宽度加了 1 位，但是代码中有的地方未改，这导致 alu 中得到的 alucontrol 信号少了一位，其他运算正确的原因是其即使少了一位也并不重复，但乘法的 alucontrol 少了一位后与上面的重复了，因此其并不执行乘法操作，导致 HILO 寄存器对应的输出全部是不定态。

(4) 修正效果: 将 alucontrol 的宽度调整正确，最终 HILO 寄存器的输出符合预期。

(5) 归纳总结: 即便大部分指令可以成功运行，其中的信号量依旧可能出错；避免的办法其实就是细心，在出现问题后，依次检查对应的控制信号即可。

3.2.2 错误 2

(1) 错误现象: 在添加除法指令时，测试程序跑不动，只有第一条指令被读出。

(2) 分析定位过程: 进行原理分析，程序一直不跑，可能是由于某个流水线停顿信号导致整个流水线一直处于阻塞状态，而此时正是刚刚加入除法指令，因此我们将怀疑的目光投向除法对应的停顿信号: stall divE。

(3) 错误原因: 分析代码发现，除法对应的停顿信号: stall divE，我们从未对其进行赋 0 操作，故而测试程序无法运行。

(4) 修正效果: 对除法对应的停顿信号: stall divE 进行赋 0 操作，测试程序成功跑通。

3.2.3 错误 3

(1) 错误现象: 在使用 trace 功能测试时，reference 的 PC 是非零数值，而 mycpu 的 PC 则是 0。

```
Number 71 Functional Test Point PASS!
Number 72 Functional Test Point PASS!
Number 73 Functional Test Point PASS!
Number 74 Functional Test Point PASS!
Number 75 Functional Test Point PASS!
Number 76 Functional Test Point PASS!
Error!
last     : PC = 0xbfc005cc, wb_rf_wnum = 0x12, wb_rf_wdata = 0x00000000
reference: PC = 0xbfc005d0, wb_rf_wnum = 0x1a, wb_rf_wdata = 0xbfc464a4
mycpu    : PC = 0xbfc005d0, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00000000
```

图 17: 错误 3 对应图

(2) 分析定位过程: 进行逻辑分析，考虑刷洗逻辑是否有影响。

(3) 错误原因: 我们在 ID 到 EXE 阶段并未进行根据 flushE 对 PC 进行了刷洗。

(4) 修正效果: 将刷洗逻辑删除后，reference 和 mycpu 的 PC 符合预期。修正后对应代码如下：

Algorithm 40 flushE

```
“‘verilog if(flushE) begin

srcaE <= 32’b0; srcbE <=
32’b0; signimmE <= 32’b0;
pcplus8E <= 32’b0;

// pcE <= 32’b0; 错误逻辑
```

```
pcE <= pcD; rsE <= 5'b0;
rtE <= 5'b0; rdE <= 5'b0;
saE  <=  5'b0;  pc_adelE
<= 1'b0;

is_in_delayslot_iE <= 1'b0;

end

```

3.2.4 错误 4

(1) 错误现象: 除法无法正确暂停，导致后续指令执行出错。

(2) 分析定位过程: 因为除法在 E 阶段发出暂停信号，此时流水线已经进入了紧跟着的后面两条指令，PC 也指向除法下面第三条指令；但是，因为暂停错误问题，当除法以及除法下一条指令执行结束，除法下面第二条指令丢失。

(3) 错误原因: 除法暂停信号发出的时机不对，导致第三条指令被跳过执行。

(4) 修正效果: 保存被冲掉的 pc，当除法结束时，将 pc 返回流水线，从而顺利完成除法。

3.3 调试心得

在前期进行单类指令的功能测试中，由于指令数少，因此我们直接选择在 vivado 中观察波形。一般方法是观察代表当前指令最重要的输出结果的信号及其在流水线中传递的同一信号，再对照给出的 inst rom.s 中的结果进行检查。

对于 8 条逻辑运算指令，我们会将 ALU 的输出结果 aluoutE，随着流水线传递的 aluoutM、aluoutW，相关控制信号 memtoReg、regwrite 等的波形在 vivado 上拉取出来; 对于 6 条移位运算指令，方法同上；

对于 4 条数据移动指令，这时初步引入了 HILO 寄存器，其中对于 MFHI 和 MFLO 指令，我们需要将 HILO 寄存器中的值的波形图拉取出来，对于 MTHI 和 MTLO 指令，我们需要对比观察 ALU 的输出与 HILO 寄存器中存储值是否相同；

对于 14 条算术运算指令，前 10 条非乘除法的算术运算，只需观察 ALU 的输出及其相关信号，乘法指令，我们不仅要观察 ALU 的输出结果，更要观察 HILO 寄存器的值，即确定乘法结果是否成功写入 HILO 寄存器中，除法指令，由于除法运算需要多周期，因此需要暂停流水线，因此我们还需除法指令的暂停信号 stall divE 以及 IF、ID、EXE 阶段寄存器的暂停信号 stallF、stallD 和 stallE；

对于 12 条分支跳转指令，我们主要拉取出其跳转地址 pcbranchD 或 pcjumpD 以及分支指令的跳转条件是否满足信号 equalD，而对于带有 link 的指令，则还需要观察其写回寄存器堆的地址和数据是否正确；

对于 8 条访存指令，由于加入了对字节和半字的读写，因此我们需要拉取字节选择信号，再观察写入 Data RAM 的地址 aluoutM 和数据 writedataM、读 Data RAM 的地址 aluoutM 和读出的数据 readdataM 、以及 Data RAM 的写使能等控制信号是否正确。再使用 trace 机制进行功能测试中，如果发生错误，我们的一般处理思路分为三步:

第一步，直接根据错误提示分析出错原因，有一些错误是可以从报错信息中推测一二的，如 mycpu 的 debug wb pc = 32’b0，这时出错的原因大概率是流水线对 PC 值的刷新出了问题，因此需要着重检查各阶段 flush 信号的设置问题，如果在代码层面找不出错误，再调出相应 PC 位置附件的波形图，着重观察 flush 信号与 PC 信号的变化；再如有时 mucpu 的 PC 值会比参考 PC 小 4，那么这时的原因大概率处在延迟槽指令、例外返回、link 指令这几方面，因为这几部分都有对 PC 进行加 4、加 8 或减 4 的操作，找到其对于信号对 PC 的影响，结合波形图再做进一步分析。

第二步，仔细分析波形图，由上到下把相关信号都拉取出来，仔细比对。第三步，到了这一步，证明我们很难通过自己 debug 的方式找出错误，这时我们会通过阅读参考代码的方式，学习其在对应部分采取的逻辑，再重新审视我们自己的实现，对其进行修改。

4 设计结果

4.1 设计交付物说明

设计的目录层次如下：

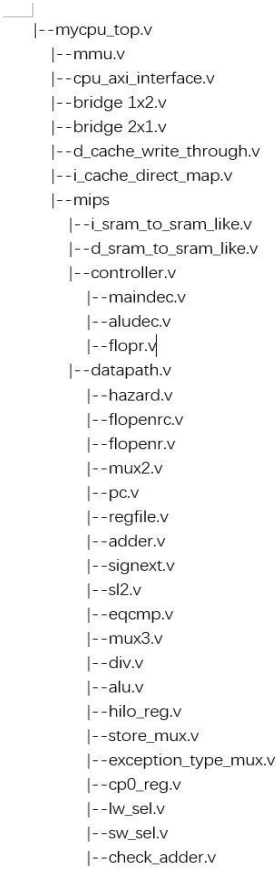


图 18: 目录层次

功能测试: 将实现了 cache 和 axi 接口的 cpu 的所有文件放到 CO-lab-material-CQU 中的 mycpu 文件夹下并在 VS code 中打开 CO-lab-material-CQU 文件夹。之后，在终端中打开 verilator 文件夹下的 axi 文件并按照 readme 文件修改 verilator 不支持的语法。最后，输入 make 进行编译，再输入./obj dir/Vmycpu top -func 进行功能测试观察是否通过 89 个测试点。

性能测试: 根据 axi 功能测试的结果，我们已经通过了所有功能测试点，接着便可在 VS code 打开的终端中输入./obj dir/Vmycpu top -perf 进行性能测试。同时，可以打开 CO-lab-material-CQU 中的 perf-test v0.01 对 10 个测试点进行单独的测试, 测试方法为导入相应的 cpu 文件并更换 axi ram 文件进行测试。最终，我们的测试结果为 10 个测试点全部通过。

4.2 设计演示结果

SRAM-SOC 功能测试：全部通过。

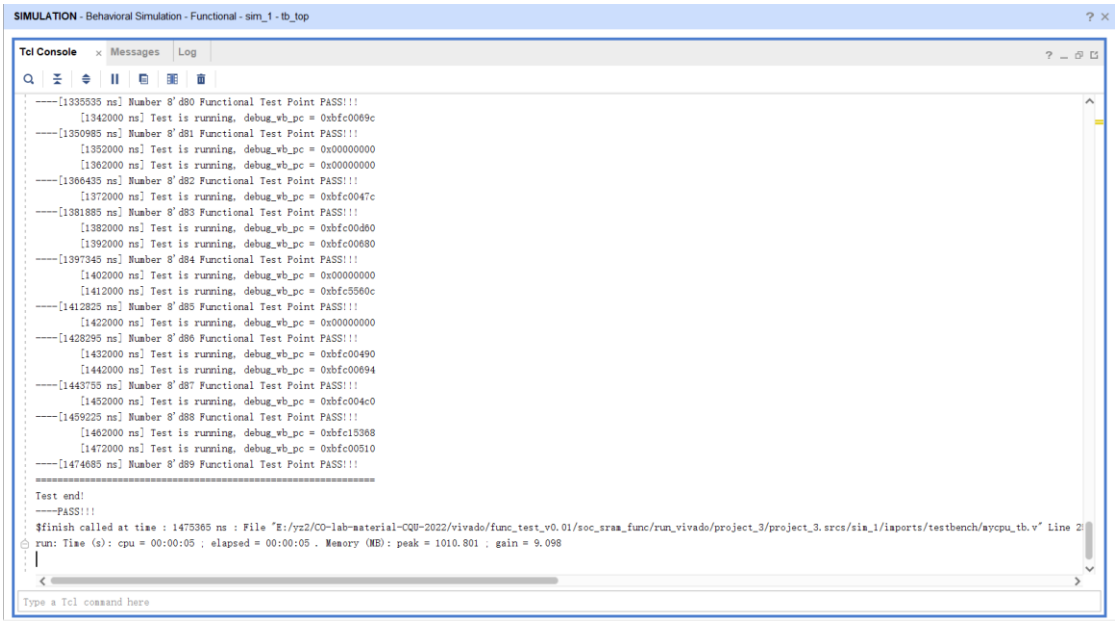


图 19: SRAM-SOC 功能测试

AXI 功能测试：全部通过。

```
        [56862000 ns] Test is running, debug_wb_pc = 0xbfc00398
        [56872000 ns] Test is running, debug_wb_pc = 0xbfc005a4
        [56882000 ns] Test is running, debug_wb_pc = 0xbfc15588
        [56892000 ns] Test is running, debug_wb_pc = 0xbfc00588
        [56902000 ns] Test is running, debug_wb_pc = 0xbfc15598
----[56903535 ns] Number 8'd89 Functional Test Point PASS!!!
        [56912000 ns] Test is running, debug_wb_pc = 0xbfc00d90
        [56922000 ns] Test is running, debug_wb_pc = 0x00000000
==============================================================
Test end!
----PASS!!!
$finish called at time : 56927674500 ps : File "E:/colab3/CO-lab-material-CQU/vivado/func_test_v0.01/soc_axi_func_sim/testbench/mycpu_tb.v" Line 269
run: Time (s): cpu = 00:03:59 ; elapsed = 00:10:18 . Memory (MB): peak = 1401.781 ; gain = 83.430
```

图 20: AXI-SOC 功能测试

AXI 性能测试——allbench：通过。

```
run all
==============================================================
Test begin!
        [  22000 ns] Test is running, debug_wb_pc = 0xbfc0004c
==============================================================
Test end!
----PASS!!!
$finish called at time : 78662500 ps : File "E:/colab4/CO-lab-material-CQU/vivado/perf_test_v0.01/soc_axi_perf/testbench/mycpu_tb.v" Line 268
```

图 21: AXI 性能测试——allbench

AXI 性能测试——bitcount：通过。

```
Test begin!
        [  22000 ns] Test is running, debug_wb_pc = 0xbfc0004c
bitcount test begin.

Bit counter algorithm benchmark



bitcount PASS!Bits: 811

bitcount: Total Count(SoC count) = 0x18c4c9

bitcount: Total Count(CPU count) = 0x168176

================================================================
Test end!
----PASS!!!
$finish called at time : 18225252500 ps : File "E:/colab4/CO-lab-material-CQU/vivado/perf_test_v0.01/soc_axi_perf/testbench/mycpu_tb.v" Line 268
run: Time (s): cpu = 00:02:16 ; elapsed = 00:04:26 . Memory (MB): peak = 1482.738 ; gain = 0.000
```

图 22: AXI 性能测试——bitcount

AXI 性能测试——bubble sort：通过。

```
run all
================================================================
Test begin!
        [  22000 ns] Test is running, debug_wb_pc = 0xbfc0004c
bubble sort test begin.

bubble sort PASS!

bubble sort: Total Count(SoC count) = 0x8be99b

bubble sort: Total Count(CPU count) = 0x7f2e7f

================================================================
Test end!
----PASS!!!
$finish called at time : 93595492500 ps : File "E:/colab4/CO-lab-material-CQU/vivado/perf_test_v0.01/soc_axi_perf/testbench/mycpu_tb.v" Line 268
run: Time (s): cpu = 00:09:54 ; elapsed = 00:19:30 . Memory (MB): peak = 1463.500 ; gain = 0.000
```

图 23: AXI 性能测试——bubble sort

AXI 性能测试——coremark：通过。

```
[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) = 0x1554d1f

coremark: Total Count(CPU count) = 0x136428b

================================================================
Test end!
----PASS!!!
$finish called at time : 225441976500 ps : File "E:/colab4/CO-lab-material-CQU/vivado/perf_test_v0.01/soc_axi_perf/testbench/mycpu_tb.v" Line 268
run: Time (s): cpu = 00:17:16 ; elapsed = 00:41:37 . Memory (MB): peak = 1445.039 ; gain = 12.770
```

图 24: AXI 性能测试——coremark

AXI 性能测试——crc32：通过。

```
end

1601645211, 00000200

crc32 PASS!

crc32: Total Count(SoC count) = 0xe0f88a

crc32: Total Count(CPU count) = 0xcc81f9

============================================================
Test end!
----PASS!!!
$finish called at time : 149024822500 ps : File "E:/colab4/CO-lab-material-CQU/vivado/perf_test_v0.01/soc_axi_perf/testbench/mycpu_tb.v" Line 268
run: Time (s): cpu = 00:11:11 ; elapsed = 00:26:16 . Memory (MB): peak = 1390.426 ; gain = 16.840
```

图 25: AXI 性能测试——crc32

AXI 性能测试——dhrystone：通过。

```
Begin ns: 2442790

End ns: 4889670

Total ns: 2446880

Dhrystones per ms:                    4


dhrystone PASS!

dhrystone: Total Count(SoC count) = 0x265a14

dhrystone: Total Count(CPU count) = 0x22d9f6

============================================================
Test end!
----PASS!!!
$finish called at time : 26916286500 ps : File "E:/colab4/CO-lab-material-CQU/vivado/perf_test_v0.01/soc_axi_perf/testbench/mycpu_tb.v" Line 268
run: Time (s): cpu = 00:02:05 ; elapsed = 00:04:59 . Memory (MB): peak = 1371.590 ; gain = 0.000
```

图 26: AXI 性能测试——dhrystone

AXI 性能测试——quick sort：通过。

```
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_top_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:06 ; elapsed = 00:00:18 . Memory (MB): peak = 1156.898 ; gain = 0.000
run all
============================================================
Test begin!
        [  22000 ns] Test is running, debug_wb_pc = 0xbfc0004c
quick sort test begin.

quick sort PASS!

quick sort: Total Count(SoC count) = 0x945026

quick sort: Total Count(CPU count) = 0x86d0dc

============================================================
Test end!
----PASS!!!
$finish called at time : 99089772500 ps : File "E:/colab4/CO-lab-material-CQU/vivado/perf_test_v0.01/soc_axi_perf/testbench/mycpu_tb.v" Line 268
run: Time (s): cpu = 00:06:15 ; elapsed = 00:19:04 . Memory (MB): peak = 1180.828 ; gain = 23.930
```

图 27: AXI 性能测试——quick sort

AXI 性能测试——select sort：通过。

```
Test begin!
        [  22000 ns] Test is running, debug_wb_pc = 0xbfc0004c
select sort test begin.

select sort PASS!

select sort: Total Count(SoC count) = 0x99ad63

select sort: Total Count(CPU count) = 0x8bb1ef

============================================================
Test end!
----PASS!!!
$finish called at time : 102616416500 ps : File "E:/colab4/CO-lab-material-CQU/vivado/perf_test_v0.01/soc_axi_perf/testbench/mycpu_tb.v" Line 268
run: Time (s): cpu = 00:04:48 ; elapsed = 00:16:58 . Memory (MB): peak = 1335.641 ; gain = 0.000
```

图 28: AXI 性能测试——select sort

AXI 性能测试——sha：通过。

```
3742976831 : 3742976831

2079096471 : 2079096471

sha PASS!

sha: Total Count(SoC count) = 0x8c5956

sha: Total Count(CPU count) = 0x7f9426

============================================================
Test end!
----PASS!!!
$finish called at time : 93507162500 ps : File "E:/colab4/CO-lab-material-CQU/vivado/perf_test_v0.01/soc_axi_perf/testbench/mycpu_tb.v" Line
run: Time (s): cpu = 00:06:29 ; elapsed = 00:17:30 . Memory (MB): peak = 1335.641 ; gain = 3.500
update_compile_order -fileset sources_1
```

图 29: AXI 性能测试——sha

AXI 性能测试——stream copy：通过。

```
run all
============================================================
Test begin!
        [  22000 ns] Test is running, debug_wb_pc = 0xbfc0004c
stream copy test begin.

stream copy PASS!

stream copy: Total Count(SoC count) = 0x94c24

stream copy: Total Count(CPU count) = 0x870cd

============================================================
Test end!
----PASS!!!
$finish called at time : 7961042500 ps : File "E:/colab4/CO-lab-material-CQU/vivado/perf_test_v0.01/soc_axi_perf/testbench/mycpu_tb.v" Line 268
run: Time (s): cpu = 00:00:35 ; elapsed = 00:01:20 . Memory (MB): peak = 1335.641 ; gain = 0.000
```

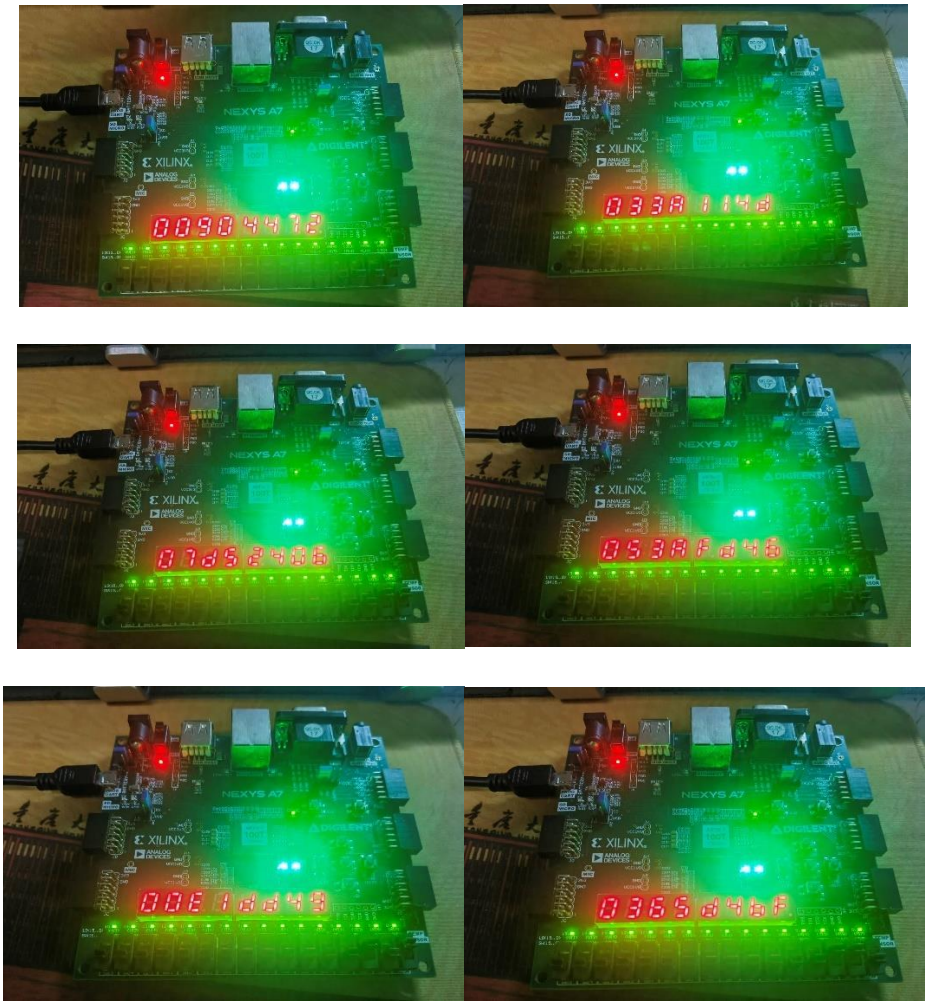
图 30: AXI 性能测试——stream copy

AXI 性能测试——stringsearch：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) = 0x5a1558

string search: Total Count(CPU count) = 0x51e1f8

============================================================
Test end!
----PASS!!!
$finish called at time : 61001612500 ps : File "E:/colab4/CO-lab-material-CQU/vivado/perf_test_v0.01/soc_axi_perf/testbench/mycpu_tb.v" Line 268
run: Time (s): cpu = 00:04:41 ; elapsed = 00:11:24 . Memory (MB): peak = 1338.617 ; gain = 0.000
```

图 31: AXI 性能测试——stringsearch

AXI 性能测试——上板



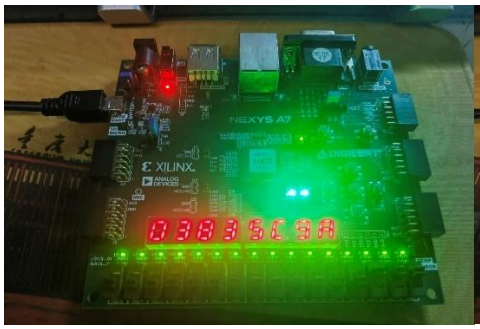
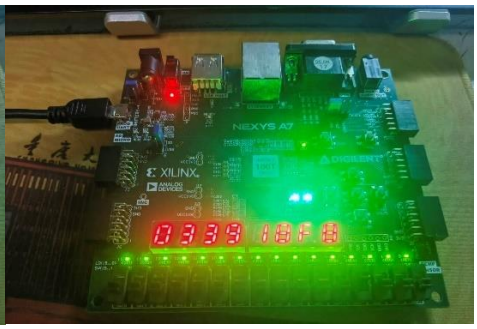
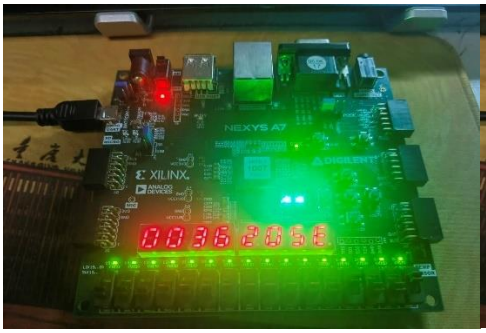
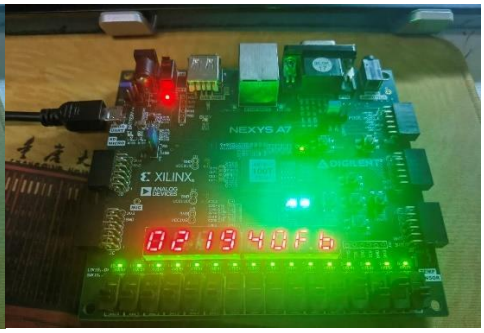





图 32: AXI 性能测试——上板

AXI 性能测试——性能测试得分

二、性能测试分数计算

序号	测试程序	myCPU 上板计时(16进制) 数码管显示	gs132 上板(16进制) 数码管显示	T_{gs132}/T_{mycpu}
cpu_clk : sys_clk		50MHz : 100MHz	50MHz : 100MHz	-
1	bitcount	904472	13CF7FA	2.197091903
2	bubble_sort	33A114D	7BDD47E	2.399113696
3	coremark	7D52406	10CE6772	2.145686858
4	crc32	53AFD46	AA1AA5C	2.032626673
5	dhrystone	E1DD49	1FC00D8	2.249151734
6	quick_sort	365D4DF	719615A	2.089347444
7	select_sort	3835C9A	6E0009A	1.956946022
8	sha	33918F9	74B8B20	2.263421408
9	stream_copy	36205E	853B00	2.46146768
10	stringsearch	21940FD	50A1BCC	2.40130165

性能分	2.214

图 33: AXI 性能测试得分

5 参考设计说明

本次课设基于计算机组成原理课程实验 4 完成。

6 现场添加指令和答辩记录

6.1 现场添加指令

6.1.1 分工情况

刘成负责编写代码，丁超、余肖杨负责思考指令应当如何添加，怎样改动以及添加的代码是否错误，最后大家一起对仿真结果进行分析。

6.1.2 完成情况

根据抽到的 ABS 指令的指令格式，分析得出需要在以下文件添加以下代码：

Algorithm 41 maindecoder 中需要添加的代码

```
6'b111111: case(funct)

//1.ABS

6'b000000:controls = 17'b1100_0000000_000000; default: controls =
17'b0000_0000000_000000; endcase
```

Algorithm 42 aludecoder 中需要添加的代码

```
6'b111111: case(funct)

//1.ABS

6'b000000:alucontrol = 'EXE_ABS_OP; default:
alucontrol ='EXE_NOP_OP; endcasee
```

Algorithm 43 alu 中需要添加的代码

```
//1.ABS

        'EXE_ABS_OP:begin result = (a[31]==0)? a : ~a+1 ;          overflow = 0; end
```

仿真运行结果如图，测试通过：

```
       [ 332000 ns] Test is running, debug_wb_pc = 0xbfc21a5c
       [ 342000 ns] Test is running, debug_wb_pc = 0xbfc22dec
       [ 352000 ns] Test is running, debug_wb_pc = 0xbfc2415c
----[ 359365 ns] Number 8'd15 Functional Test Point PASS!!!
       [ 362000 ns] Test is running, debug_wb_pc = 0x00000000
----[ 363045 ns] Number 8'd16 Functional Test Point PASS!!!
----[ 366985 ns] Number 8'd17 Functional Test Point PASS!!!
----[ 368845 ns] Number 8'd18 Functional Test Point PASS!!!
----[ 370975 ns] Number 8'd19 Functional Test Point PASS!!!
       [ 372000 ns] Test is running, debug_wb_pc = 0xbfc25148
----[ 373355 ns] Number 8'd20 Functional Test Point PASS!!!
================================================================
Test end!
----PASS!!!
$finish called at time : 373775 ns : File "H:/hardware/CO-lab-material-CQU-new/CO-lab-material-CQU/vi
```

图 34: ABS 指令添加运行结果

6.2 现场答辩记录

6.2.1 问题 1

问：跳转指令在什么阶段实现，为什么不刷新取值阶段的指令以及延迟槽是什么？

余肖杨:

跳转在译码阶段实现，因为跳转后面的指令在这个阶段已经被取进 CPU，如果刷新会影响执行的结果。延迟槽是跳转指令后面紧跟的那一条指令，也就是指令地址为 PC+4 的指令。延迟槽是无论分支条件是否成立都会执行的指令，如果刷新，则延迟槽指令不会执行。

6.2.2 问题 2

问：异常有哪些异常？

刘成：中断：中断是由外部设备或其他事件触发的异常。当中断发生时，CPU 会暂停当前的指令执行，并跳转到中断处理程序来处理中断事件。系统调用：系统调用是由程序主动发起的异常，用于请求操作系统提供特定的服务或功能。无效指令：当 CPU 遇到无效或未定义的指令时，会触发无效指令异常。断点：断点是用于调试程序, 比如 syscall 和 break 指令，在 MEM 阶段存在读写数据和取指令地址异常，对于读指令地址异常，在 IF 阶段取指令的时候判断指令地址的最低两位是否为 0，如果不为 0，则触发异常。在读写数据的时候，通过判断读写类型和读写地址进行判断是否触发异常。

6.2.3 问题 3

问：访问内存的最小单位是什么？访存具体怎么实现？

丁超：访问内容的最小单位是字。访存指令就是实现对主存中的数据进行访问或存储。所有运算都在寄存器上进行,访存指令包含各种宽度数据的读写、无符号读、非对齐访存和原子访存等。MIPS 中访存指令如下：L 开头的都是从主存加载数据操作，S 开头的都是存储数据操作。操作数据的大小有 B（Byte 8bit）、H（Halfword 16bit）、W（Word 32bit）、D（Double 64bit）。加载地址都是 base+offset 方式，offset 取值范围在在-32767 至 32768。

7 总结

7.1 刘成

这次硬件综合设计实践不仅是一次技术上的挑战，更是对知识广度和深度的全面考验。从 52 条基础指令到连接 SRAM 和基础 Cache 再到 AXI 的实现，每个阶段都涉及到了深入的硬件知识。在访存指令的运转细节、寄存器与内存之间数据存储的区别、异常处理等方面，我逐渐建立了更为具体而深刻的认识。在解决 AXI 实现的难题时，对信号量的理解更是得到了加深。整个过程不仅加深了我对 MIPS 架构的 CPU 的理解，还让我对计算机体系结构有了更为直观和具

体的认识。特别是在处理数据冒险和控制冒险等指令添加问题时，我深入思考了指令对整体结构的影响，学到了在修改现有代码时尽量保持结构清晰的原则。此外，通过实践我还学到了一些关于硬件处理器例外响应、虚实地址转换、以及 cache 的原理与实现等方面的知识。也学习到了转换桥的作用，也加深了 cache 在 cpu 中的结构（mips 核与 2x1 转换桥之间）以及作用（高速缓存）等等。这次实践不仅是技术水平的提升，更是对计算机体系结构的认知和理解的全面拓展。

7.2 丁超

这次硬件综合设计的实践是一场充满挑战的征程。整个过程充满了不确定性与困难。最初的阶段，面对陌生的硬件设计，我深感自己的无知，仿佛站在巨人的肩膀上仍然难以望见远处。然而，通过学习学长的视频、查阅实验文档以及研究网上的代码，我逐渐摸索出设计的方向。一些基础指令的逐步实现，再到连接 SRAM 的设计，每个阶段都如同攀登一座高峰，需要我们克服各种技术难题。在 52 条指令阶段，我负责访存部分。通过学习各方面知识以及实验文档，我明白了访存指令的运转细节，以及寄存器与内存之间存储数据的区别——内存地址是按照字节编址的，但是寄存器是按照字（32 位）存储的，而访存指令其实主要区分就在于数据字节截取以及有无符号扩展，同时还需要注意访存指令的地址错例外——取指操作的访存地址必须 4 字节边界对齐，否则将触发取指地址错例外，在后续异常处理部分，需要捕捉此处的异常情况。

7.3 余肖杨

在本次硬综的实践过程中，最大的感觉就是困难。在整合阶段，我们经历了信号命名不一致、仿真失败等问题，为了解决这些问题，我们选择了推倒重来。这个决定让我们更加深刻地认识到团队合作的重要性。在重新开始的过程中，我们明确了分工，小心翼翼地进行每个模块的测试，确保思路的清晰和记录的完整。在处理指令添加时的数据冒险和控制冒险问题时，团队成员之间的高效沟通和协作变得至关重要。我们需要共同思考解决方案，及时交流问题，确保每个模块的设计都能够顺利地融入整体系统。在这次硬件综合设计的过程中我也学到课很多新的东西。通过自身的实践，我对 mips 架构的 cpu 有了更深地认识，同时，对于数据冒险，流水线刷新，流水线暂停等也有了更加深刻的认识。这次实践让我深刻认识到，在硬件设计中，不仅要有扎实的技术能力，更需要良好的团队协作和沟通能力。

参考文献

[1] 雷思磊. 自己动手写 CPU. 电子工业出版社, 2014.

[2] 《计算机组成原理实验指导书》，重庆大学计算机学院编；

[3] 《MIPS 基准指令集手册》

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

[5] 《计算机系统综合课程设计》，杨全胜主编，清华大学出版社；