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(卓越)02 班
年　　级:	2020
学　　生:	黄锐　　王世豪
学　　号:	20204112　　20204111
完 成 时 间:	2023 年　1 月　6 日
成　　绩:	
指 导 教 师:	钟将

重庆大学教务处制

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

指导教师评定成绩：

指导教师签名：

MIPS SOC 设计报告

黄锐、王世豪

1 设计简介

本小组的设计是一个简单 MIPS CPU,实现了 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、2 条自陷指令、8 条访存指令、3 条特权指令,总计 57 条指令,实现了连接类 SRAM 接口转 AXI 总线转接桥,构造 AXI 版 SOC, 且通过完整版指令测试、性能测试,同时使用 Cache, 成功提高 CPU 性能,且功能测试、性能测试通过。

1.1 小组分工说明

本小组完成功能时,分为两个阶段,首先两人分工进行不同模块的设计,然后在交流思路之后,一起写代码,一个人写,另一个人监督写代码的过程并记录发生修改的地方和出错的地方。

- 黄锐:
  - 设计阶段:在 52 条指令期间,设计 hilo 寄存器,完成数据移动指令的设计和乘除法器的设计,完成访存指令的设计,和 hazard 处理,在 57 条指令与 AXI 期间,设计 cp0 和异常处理逻辑。
  - 代码阶段:写代码。
- 王世豪:
  - 设计阶段:在 52 条指令期间,负责设计除乘除法以外的运算指令和分支跳转指令,在 57 条指令与 AXI 期间,负责设计 AXI 接口,和 Cache 的接入。
  - 代码阶段:监督写代码的过程并记录发生修改的地方和出错的地方。

2 设计方案

2.1 总体设计思路

总体设计思路分为两部分,第一部分为 AXI 的总体设计思路,主要包含 mips 向外部 ram 和外设进行交互的设计,第二部分为 mips 的总体设计思路,包含 mips 的流水线的设计[2]。

2.1.1 AXI 总体设计思路

总体设计思路图如图1, 这里以 datapath 向 RAM 请求数据为例,介绍总体设计思路。

首先 datapath 要请求数据时,将数据分为 instruct 数据和 data 数据,分别由 i_sram_to_sram_like 和 d_sram_to_sram_like 转换为 sram_like 标准,也就是将 inst_req 和 data_req 设为 1,并且等到地址握手后改变状态机,等待数据握手,并且将流水线 stall。

对于 inst_req,只需要转换为物理地址后传给 I_Cache 请求数据,若 I_Cache 命中,则直接返回数据,若缺失,就将 inst_req 传给 cpu_axi_interface 请求外部 ram 中的数据。

对于 data_req,首先要经过 mmu,判断是否经过 cache,若不经过 cache,则通过 1x2bridge 将请求转给 confreg_data,直接请求外部 Confreg,否则类似 inst_req,经过 D_Cache,后续和 I_Cache 一致。

当外部传回数据的时候,也就是 i_sram_to_sram_like 和 d_sram_to_sram_like 接收到地址握手后,更改状态机,并将 stall 解除,然后由 datapath 把数据读出来。

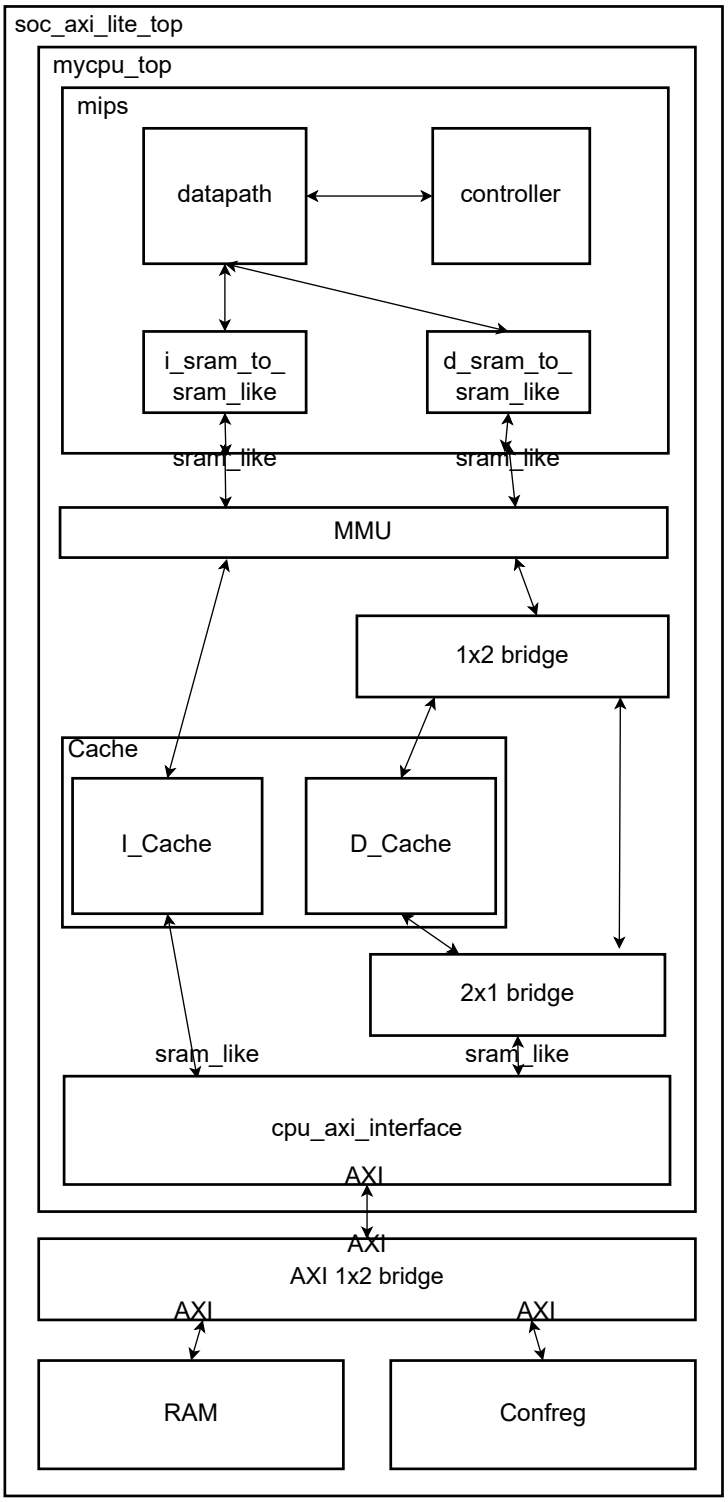

图 1: 总体设计图

2.1.2 mips 总体设计思路

mips 设计总体思路如图2,总体上分为 datapath,controller,i_sram_to_sram_like, d_sram_to_sram_like。

mips 中,主要部分为 datapath,datapath 分为五级流水线,所有器件时钟同步,包含

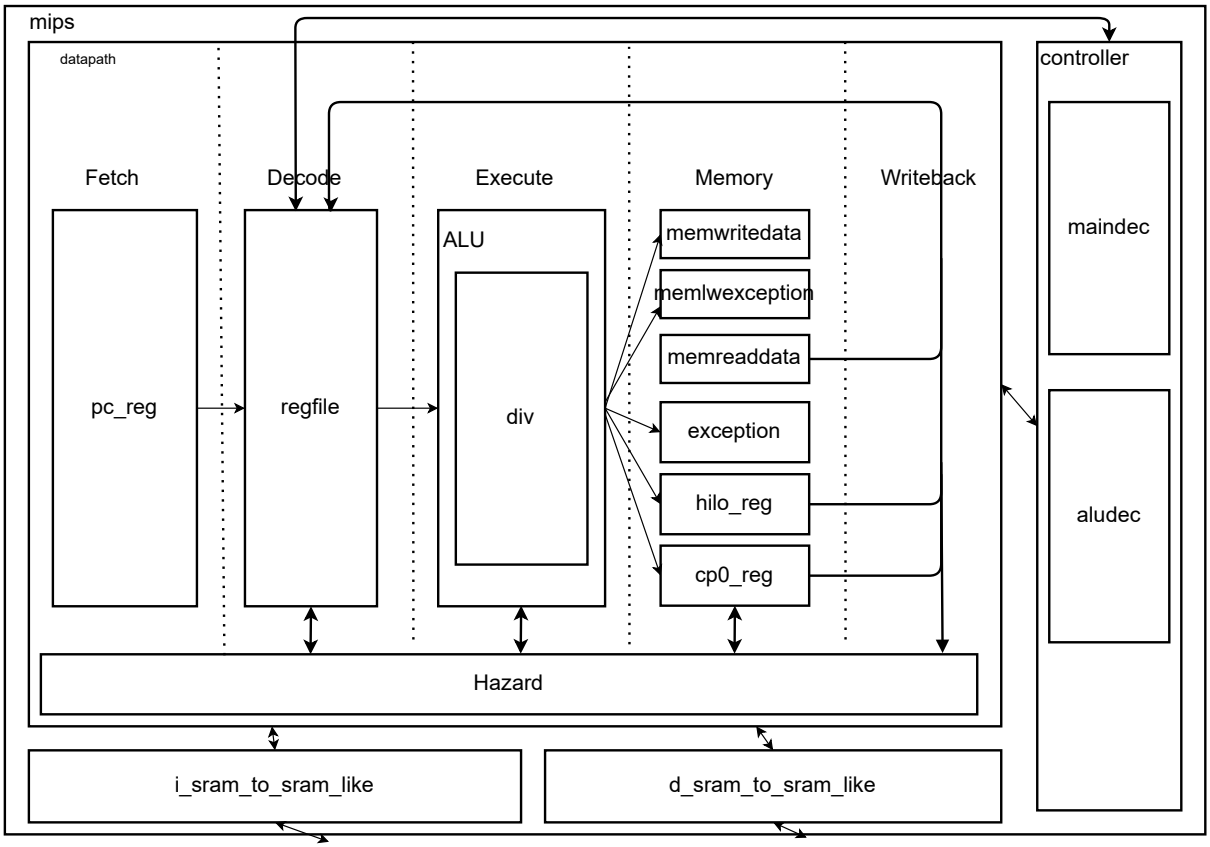

图 2: mips 设计图

Fetch,Decode,Execute,Memory,Writeback 五个阶段,以及冒险处理模块。Fetch 阶段负责取指,指令地址由 pcreg 控制;Decode 阶段将取到的指令交给 controller 进行译码,regfile 负责读取相应寄存器的数据;Execute 阶段根据 controller 译码的结果和 regfile 读取到的数据使用 ALU 进行计算,对于非除法指令,使用组合逻辑得到结果,对于除法,使用 36 周期除法器 div 得到结果,此时需要暂停流水线的全部阶段;Memory 阶段负责 memory 数据,hilo 数据,cp0 数据的存储和读取,memwritedata 负责写入数据的处理,主要处理的是 sb 和 sh 指令,memreaddata 负责处理 memory 得到的数据,主要处理的是 lb(u) 和 lh(u) 指令,memlwexception 负责处理读取指令的地址例外,exception 模块将前面得到的异常汇总,通过组合逻辑得出 exception 的类型传给 cp0 处理,并且确定 Fetch 阶段下一跳的地址;Writeback 阶段,将相应数据写入到 regfile 中。

在 controller 中,maindec 根据 decode 阶段传入的指令翻译出各个组件的控制型号,aludec 根据 decode 阶段传入的指令翻译出 alu 的运算类型。

2.2 写透 d_cache 模块设计

cache 部分理解了 ref_code 中的 cache 后直接使用了其中的写透 cache,由于 i_cache 只需要读,设计是 d_cache 的子集,因此此处只对 d_cache 进行说明。

其接口定义如下:

信号名	方向	接口类型	位宽	说明
clk	input	wire	1	时钟
rst	input	wire	1	复位信号
cpu_data_req	input	wire	1	发送的请求
cpu_data_wr	input	wire	1	是否是写请求
cpu_data_size	input	wire	2	配合 addr 的最后两位决定有效字节
cpu_data_addr	input	wire	32	写或读地址
cpu_data_wdata	input	wire	32	写数据
cpu_data_rdata	output	wire	32	读出来的数据
cpu_data_addr_ok	output	wire	1	地址握手
cpu_data_data_ok	output	wire	1	数据握手
cache_data_req	output	wire	1	发送的请求
cache_data_wr	output	wire	1	是否是写请求
cache_data_size	output	wire	2	配合 addr 的最后两位决定有效字节
cache_data_addr	output	wire	32	写或读地址
cache_data_wdata	output	wire	32	写数据
cache_data_rdata	input	wire	32	读出来的数据
cache_data_addr_ok	input	wire	1	地址握手
cache_data_data_ok	input	wire	1	数据握手

这里需要注意的是 cpu_ 开头的代表 cpu 与 cache 之间的交互,cache_ 开头的是 cache 与外部 axi 的交互。

cache 中的状态机定义如下:

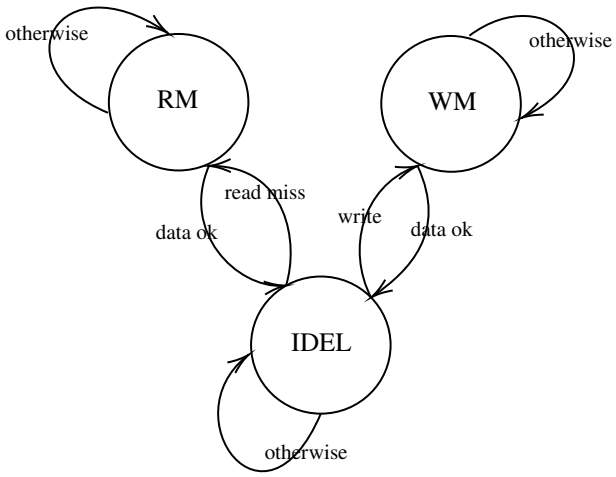

图 3: dcache 状态机

cache 的组织方式(通路图)如下:

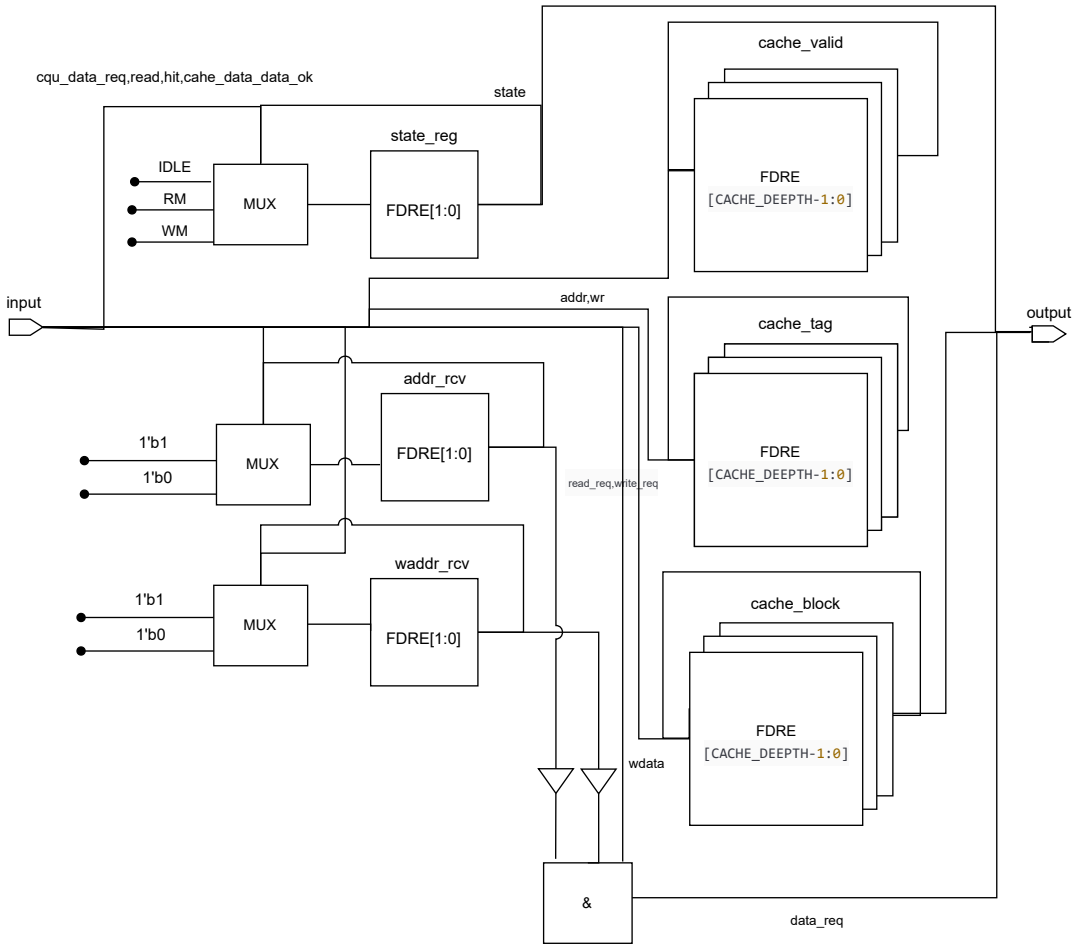

图 4: dcache 通路图

2.3 sram_to_sram_like 模块设计

该部分参考系统结构实验二的结构,其模块接口定义如下(X 代表 inst 或者 data):

信号名	方向	接口类型	位宽	说明
clk	input	wire	1	时钟
rst	input	wire	1	复位信号
X_sram_en	input	wire	1	ram 使能
X_sram_addr	input	wire	32	读 ram 的地址
X_sram_rdata	output	wire	32	ram 读出来的数据
X_stall	output	wire	1	操作 ram 时带来的 stall
X_req	output	wire	1	发起的数据请求
X_wr	output	wire	1	是否是写请求
X_size	output	wire	2	配合 addr 的最后两位决定操作 ram 的有效字节
X_addr	output	wire	32	读或写的地址
X_wdata	output	wire	32	写入的值
X_addr_ok	input	wire	1	地址握手信号
X_data_ok	input	wire	1	数据握手信号
X_rdata	input	wire	32	读出的数据
all_stall	input	wire	1	是否正在全阶段 stall

在模块中使用 addr_rcv 来记录地址握手的状态,利用 do_finish 来记录是否拿到了有效数

据。模块内部判断地址握手和数据握手的状态机如下:

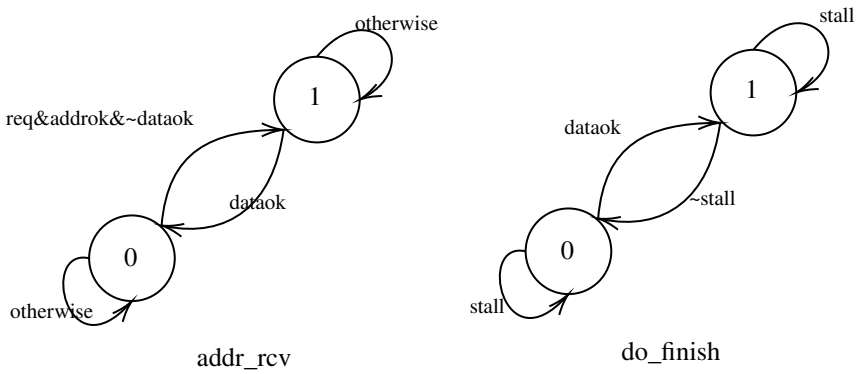

图 5: sram_to_sram_like 状态机

模块通路图如下:

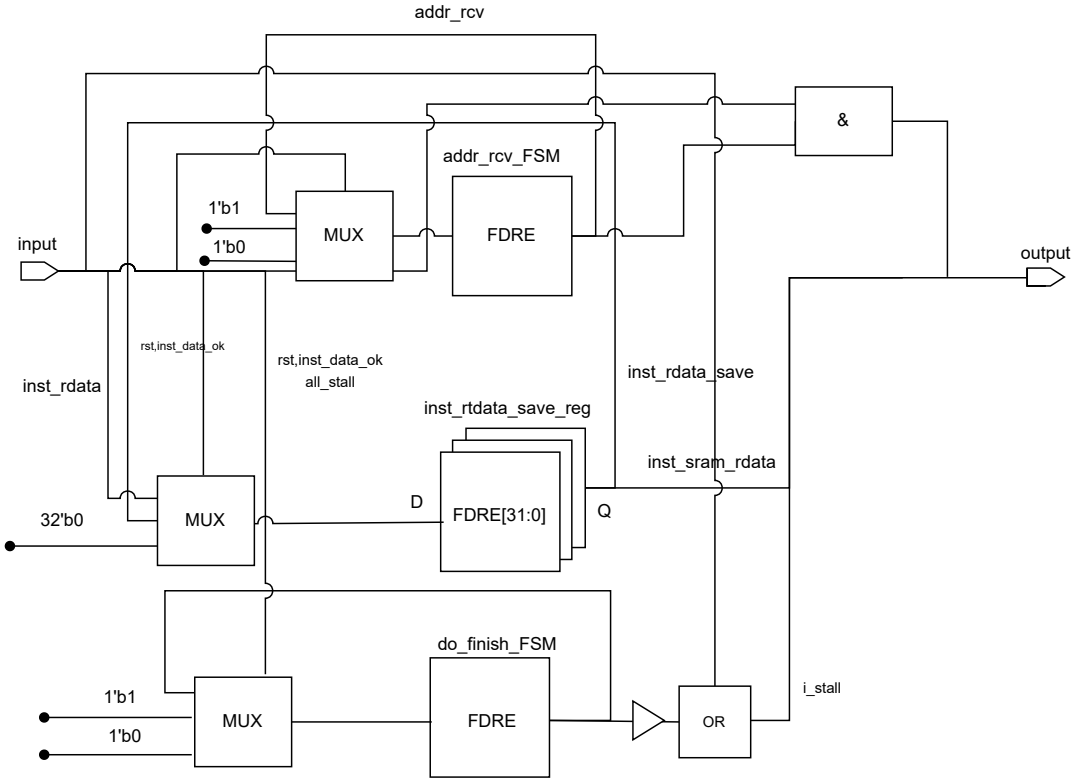

图 6: sram 转类 sram 接口通路图

2.4 乘法器模块设计

乘法器是我们小组自己设计的模块,其接口定义如下:

信号名	方向	接口类型	位宽	说明
clk	input	wire	1	时钟
rst	input	wire	1	复位信号
signed_mul_i	input	wire	1	是否为有符号乘法
opdata1_i	input	wire	32	乘数 1
opdata2_i	input	wire	32	乘数 2
start_i	input	wire	1	开始信号
result_o	output	reg	64	结果
ready_o	output	reg	1	是否计算完成
mul_stall	output	reg	1	乘法全流水线 stall

我们的乘法器分为三个状态,00 表示空闲,01 表示正在算第一拍,10 表示算第二拍并且得到了结果,具体状态转移图如下:

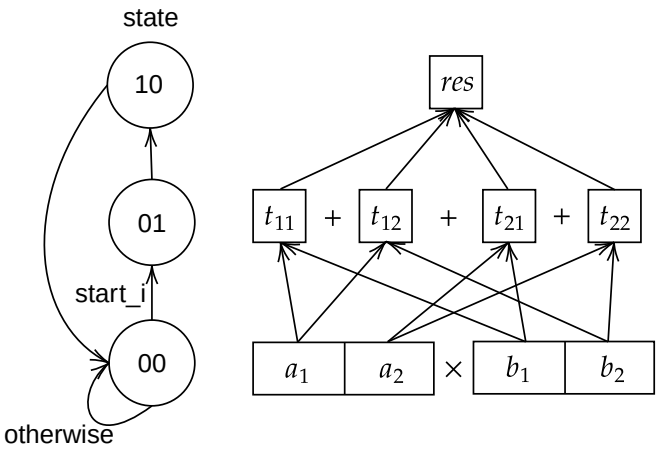

图 7: 乘法器设计

乘法器的通路图如下:

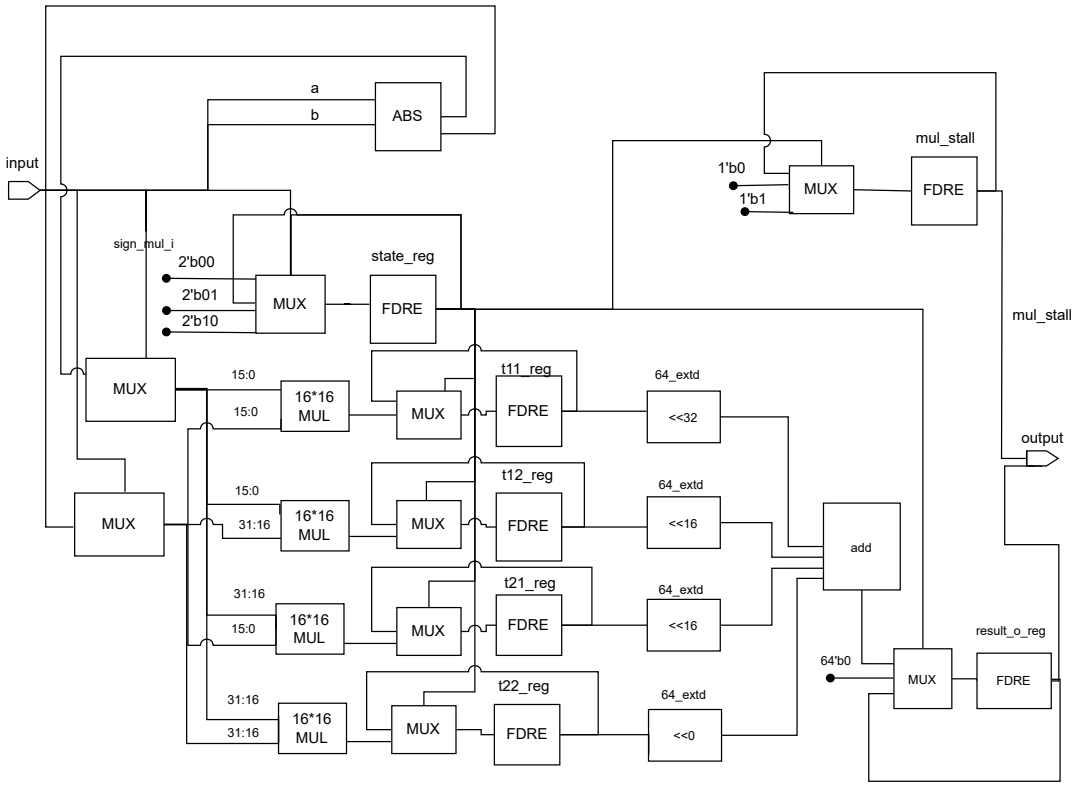

图 8: 乘法器通路图

3 设计过程

3.1 设计流水账

1. 2022 年 12 月 22 日 19:00~ 凌晨 1 点:两人一起配置环境(Vivado2019.2,Verilator,gtkwave,实验包),复习 Vivado 的使用,学习 Verilator gtkwave 等新工具的使用,结果是学会了。
2. 2022 年 12 月 23 日 15:00~18:00,两人一起修改 ALU 架构(去掉了 aluop),重写原来的 12 条指令,结果是改好了。
3. 2022 年 12 月 24 日 15:00~18:00,王世豪添加 8 条逻辑指令和 6 条移位运算指令,黄锐设计 HILO 方案,结果是添加好了,并且设计好了。
4. 2022 年 12 月 25 日

   (1) 14:00~19:00,两人一起添加 hilo 和 14 条算术运算指令。

   (2) 20:00~24:00,两人一起添加 12 条转移指令。

   结果是添加成功。
5. 2022 年 12 月 26 日

   (1) 10:00~12:00,两人一起设计访存指令方案。

   (2) 14:00~15:00,两人一起添加访存指令。

   结果是该日,52 条指令完成,但没有通过功能测试。

6. 2022 年 12 月 28 日 20:00~ 凌晨 1 点,两人一起连接 SRAMSOC,并 debug 前 64 个功能测试,最终通过。
7. 2022 年 12 月 30 日
   (1) 9:00~15:00,两人一起完成 52 条指令 SRAMSOC 并各自上板成功。
   (2) 15:00~19:00,两人一起完成 57 条指令功能测试通过并各自上板成功。
8. 2022 年 12 月 31 日
   (1) 9:00~12:00,两人一起讨论研究 AXI 和 Cache,结果是讨论了一个方案出来。
   (2) 15:00~18:00,两人一起做系统结构实验(提前把硬件综合的 Cache 准备好),结果是完成了系统结构的实验。
9. 2023 年 1 月 1 日(元旦)
   (1) 14:00~18:00,两人一起连接 AXI 和 Cache,写转接桥,修改 mycpu_top,并 debug,结果是没 debug 完。
   (2) 20:00~24:00,两人继续一起 debug,结果是 AXI 和 Cache 功能测试通过,性能测试通过。
10. 2023 年 1 月 2 日
    (1) 9:30~11:00,两人一起写报告,结果是没写完。
    (2) 14:00~17:00,两人各自完成一些模块的通路图,写报告,结果是通路图画完了,报告没写完。
    (3) 19:00~22:00,写报告,结果是写完了。
11. 2023 年 1 月 3 日 19:00~24:00,发现 1 月 2 日半场开香槟,之前没测 perfdiff,现在发现 perfdiff 炸了,debug 一晚上未果,确定了是 dcache 带来的问题。
12. 2023 年 1 月 4 日 22:00~ 凌晨 2 点,发现 perdiff 过不了是因为 sramto 类 sram 转接口读取的时候有非法组合,改了之后就过了 perfdiff,但此时性能低于 10 分,为了让性能高于 10 分,写了一个双周期乘法器,把性能提高到了 11 分。同时尝试了用系统结构实验的四路组相联 cache 和普通写回 cache,因不同的问题而放弃。
13. 2023 年 1 月 6 日 15:00~17:00,补充报告。

3.2 错误记录

3.2.1 错误 1

(1) 错误现象:bgtz 触发了不该触发的异常

(2) 分析定位过程: 根据波形图,能看到 bgtz 指令发生了异常,仔细检查后发现是 bgtz 被 aludec 译码出来的结果是有符号减法,而触发了 overflow。

(3) 错误原因:aludec 译码错了,bgtz 被 aludec 译码出来的结果是有符号减法,而触发了 overflow。

(4) 修正效果: 将 aludec 译码 bgtz 改为 subu(无符号)。

(5) 归纳总结: 修改文件后,没有全面考虑,导致某些信号没有修改到。

3.2.2 错误 2

(1) 错误现象:div 算出来的结果,与正确结果互为相反数。

(2) 分析定位过程: 通过波形图可以发现 div 计算的过程中,前面的部分都是正确的,直到最后一个周期的时候,原本的正确答案突然变成负的正确答案。仔细查看 div 的写法,我们发现在最后一个周期的时候,会根据 op_data 的值改变得到的结果,而 op_data 在整个过程中发生了变化,而 op_data 是 alu 的 srca2E 传进来的,因此是 srca2E 出了问题。

(3) 错误原因: 刚开始的 srca2E 是 M 阶段前推过来的,但是我们的 div 只 stall 了前三个阶段,因此,在 M 阶段的指令流走了之后,前推也没了。虽然 M 阶段的那条指令已经写入了 regfile,但是由于 stall 的存在,不能把正确的值读到 E 阶段。(当时讨论的截图如下)

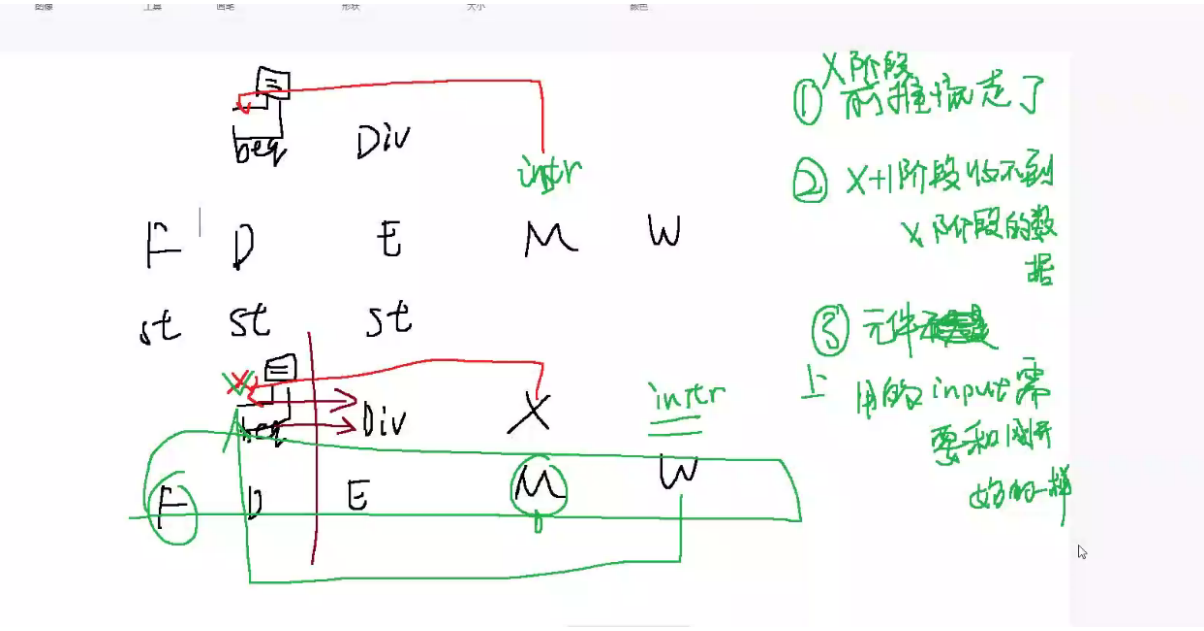

图 9: 错误原因

(4) 修正效果: 在陈泱宇学长的指导下,修正有两种方案,第一种是直接修改 div 的 stall 为全周期 stall,第二种是修改 div 模块,在里面添加一个 reg,用来存放第一个周期传进来的除数和被除数,这样最后一个周期就不会因为 op_data 改变而结果出错。我们采用了第二种方法。

(5) 归纳总结: 此错误属于时钟同步问题,在添加组件的时候应仔细考虑时钟的影响。

3.2.3 错误 3

(1) 错误现象:lb 指令本来应该改第二个字节,但是却改了第四个字节。
(2) 分析定位过程:vivado 仿真得到的错误信息中,与正确答案的第二个和第四个字节不一样,并且修改写入的数据之后,仍然是第二位和第四位不一样。
(3) 错误原因: 类 sram 接口的地址写是由 data_size 和 data_waddr 同时控制的,我们传出的 data_waddr 永远最后两位是 2'b00。
(4) 修正效果: 传入正确的 data_waddr 后,lb 指令可以正常运作。
(5) 归纳总结: 对类 sram 接口的理解不够透彻。

3.2.4 错误 4

(1) 错误现象: 从某个时间点开始,一直 stall,永远也不停。
(2) 分析定位过程: 在跑功能测试的时候,地址永远卡在了 0xbfc006d8,查看波形图发现是 data_req 发出了请求,并且进行了地址握手,但是永远也等不到它的数据握手。
(3) 错误原因:mycpu_top 模块往 mips 里面传时钟的时候,没有取反,导致 axi 接口没有接收到 data_req。
(4) 修正效果: 对 mycpu_top 传给 mips 的时钟取反。
(5) 归纳总结: 修改文件后,没有全面考虑,导致某些信号没有修改到。

3.2.5 错误 5

(1) 错误现象: 有一个 pc 在取指阶段没有发出 inst_req 请求。
(2) 分析定位过程: 查看波形图,发现该 pc 没有发出 inst_req 请求,且发现 i_sram_to_sram_like 中的状态机一直处于数据握手 ok 的状态,发现这个 ok 是因为 d_stall 是置位的,导致数据握手状态没变。
(3) 错误原因: 在上升沿的时候,i_stall 变成了 0,在下降沿的时候,d_stall 变成了 1,导致在下一个上升沿的时候,状态机不会变为空闲状态。
(4) 修正效果: 首先考虑了将 d_stall 与 X_sram_to_sram_like 同步,但是发现实现上可能会导致无法综合的问题。于是考虑将所有组件全部与流水线同步,这样做不仅可以解决上述问题,还可以提升整个 mips 的性能。
(5) 归纳总结: 此错误属于时钟同步问题,在添加组件的时候应仔细考虑时钟的影响。

3.2.6 错误 6

(1) 错误现象: 修改全时钟同步之后,cp0 处理 int 中断时,写入的 epc 提前了一周期。
(2) 分析定位过程: 首先根据错误提示的指令地址查看 test.S 文件中的对应指令,发现是 mfc0 v0,cp0_epc,这个是把 epc 取出来的意思,我们发现此处的 epc 和 ref 中的 epc

差了 4,于是我们定位到上次发生例外的地方,观察其波形图。发现是 mtc0 向 cp0 中写入了一个 int 中断,写入的一瞬间,exception 就发生了,但是此时 i_stall 也是置位的,因此发生错误的那一条指令并没有流入 Mem 阶段,导致 cp0 没有正确把中断指令的地址读到 epc 中。因此找到了原因。

(3) 错误原因:mtc0 向 cp0 中写入了一个 int 中断,写入的一瞬间,exception 就发生了,但是此时 i_stall 也是置位的,因此发生错误的那一条指令并没有流入 Mem 阶段,导致 cp0 没有正确把中断指令的地址读到 epc 中。

(4) 修正效果: 向 cp0 传入一个 stallM,使得其一同 stall,在 i_stall 结束后,再把 mtc0 的数据写进 cp0,这样就可以正常的让 cp0 接收到正确的 mtc0 地址。

(5) 归纳总结: 此错误属于时钟同步问题,在添加组件的时候应仔细考虑时钟的影响。

4 设计结果

4.1 设计交付物说明

本设计所有源文件都在 mycpu(AXI 接口)中,将文件 add 到 vivado 中即可综合实现上板或仿真,将文件放入 CO-lab-material-CQU/mycpu 文件夹中即可使用 verilator/axi 进行测试。

4.2 设计演示结果

4.2.1 功能测试

功能测试结果（Vivado）

```
============================================================
Test begin!
        [  22000 ns] Test is running, debug_wb_pc = 0x9fc41bc8
        [  32000 ns] Test is running, debug_wb_pc = 0x9fc41c68
        [  42000 ns] Test is running, debug_wb_pc = 0x9fc41d04
        [  52000 ns] Test is running, debug_wb_pc = 0x9fc41da4
        [  62000 ns] Test is running, debug_wb_pc = 0x9fc41e40
        [  72000 ns] Test is running, debug_wb_pc = 0x9fc41ee0
        [  82000 ns] Test is running, debug_wb_pc = 0x9fc41f7c
        [  92000 ns] Test is running, debug_wb_pc = 0x9fc4201c
        [ 102000 ns] Test is running, debug_wb_pc = 0x9fc420b8
        [ 112000 ns] Test is running, debug_wb_pc = 0x9fc42158
        [ 122000 ns] Test is running, debug_wb_pc = 0x9fc421f4
        [ 132000 ns] Test is running, debug_wb_pc = 0x9fc42294
        [ 142000 ns] Test is running, debug_wb_pc = 0x9fc42330
----[ 145755 ns] Number 8'd01 Functional Test Point PASS!!!
        [ 152000 ns] Test is running, debug_wb_pc = 0x9fc00d8c
        [28272000 ns] Test is running, debug_wb_pc = 0x00000000
        [28282000 ns] Test is running, debug_wb_pc = 0x00000000
        [28292000 ns] Test is running, debug_wb_pc = 0xbfc22558
----[28296095 ns] Number 8'd88 Functional Test Point PASS!!!
        [28302000 ns] Test is running, debug_wb_pc = 0xbfc152d4
        [28312000 ns] Test is running, debug_wb_pc = 0xbfc15310
        [28322000 ns] Test is running, debug_wb_pc = 0xbfc1534c
        [28332000 ns] Test is running, debug_wb_pc = 0xbfc00390
        [28342000 ns] Test is running, debug_wb_pc = 0xbfc004ac
        [28352000 ns] Test is running, debug_wb_pc = 0xbfc15438
        [28362000 ns] Test is running, debug_wb_pc = 0x00000000
        [28372000 ns] Test is running, debug_wb_pc = 0x00000000
        [28382000 ns] Test is running, debug_wb_pc = 0xbfc00528
        [28392000 ns] Test is running, debug_wb_pc = 0x00000000
        [28402000 ns] Test is running, debug_wb_pc = 0xbfc005ac
----[28409235 ns] Number 8'd89 Functional Test Point PASS!!!
        [28412000 ns] Test is running, debug_wb_pc = 0xbfc00d94
============================================================
Test end!
----PASS!!!
```

图 10: vivado 功能测试

功能测试结果（Verilator）

```
khoray@LAPTOP-KHORAY:/mnt/e/colab/CO-lab-material-CQU/verilator/axi$ ./obj_dir/Vmycpu_top -func
Number 1 Functional Test Point PASS!
Number 2 Functional Test Point PASS!
Number 3 Functional Test Point PASS!
Number 4 Functional Test Point PASS!
Number 5 Functional Test Point PASS!
Number 6 Functional Test Point PASS!
Number 7 Functional Test Point PASS!
Number 8 Functional Test Point PASS!
Number 9 Functional Test Point PASS!
Number 10 Functional Test Point PASS!
Number 11 Functional Test Point PASS!
Number 12 Functional Test Point PASS!
Number 13 Functional Test Point PASS!
Number 14 Functional Test Point PASS!
Number 15 Functional Test Point PASS!
Number 16 Functional Test Point PASS!
Number 17 Functional Test Point PASS!
Number 18 Functional Test Point PASS!
Number 19 Functional Test Point PASS!
Number 20 Functional Test Point PASS!
Number 21 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!
```

图 11: verilator 功能测试

4.2.2 性能测试

图 12: vivado 性能测试

图 13: verilator 性能测试

性能测试结果（板子）

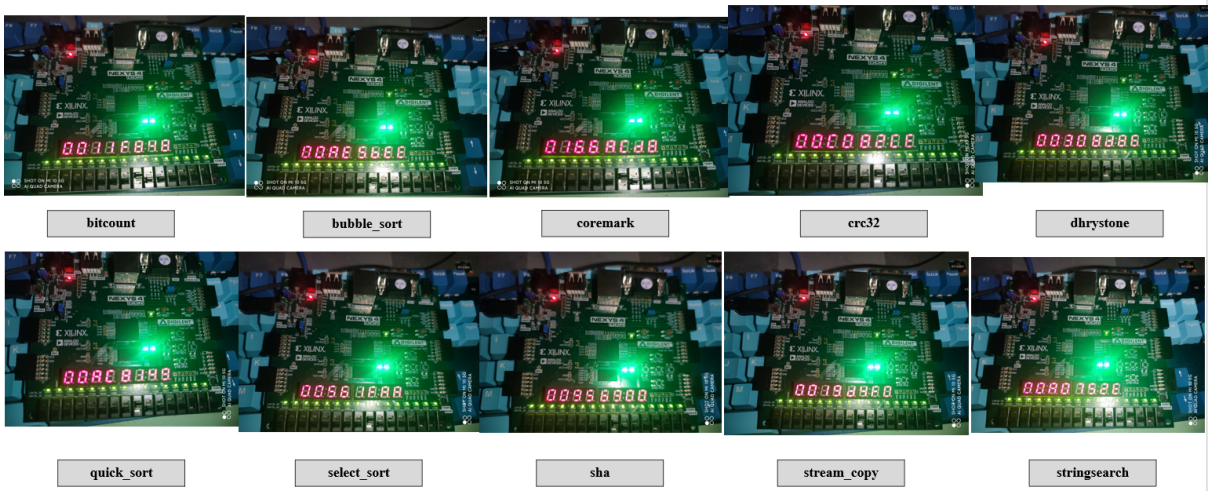

图 14: 开发板性能测试

性能测试分数

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	11f848	13CF7FA	17.63891644
2	bubble_sort	ae5bee	7BDD47E	11.36637
3	coremark	166acd8	10CE6772	11.995266
4	crc32	c082ce	AA1AA5C	14.13771728
5	dhrystone	308d86	1FC00D8	10.46289863
6	quick_sort	ac8149	719615A	10.53523125
7	select_sort	561faa	6E0009A	20.4357523
8	sha	956900	74B8B20	12.49944116
9	stream_copy	19d4f0	853B00	5.157617171
10	stringsearch	a0752e	50A1BCC	8.040177037

性能分	11.478

图 15: 性能测试分数

5 参考设计说明

1. 本项目建立在吕昱峰学长的 lab4 基础上 [1]。
2. opdefines.vh 和 defines2.vh 中的宏定义借鉴实验资料包中的 ref_code 中的 defines.vh。
3. 36 周期 div 除法器来源于实验资料包中的 ref_code[3]。

4. mmu 来源于实验资料包中的 ref_code。
5. 直接映射 icache 和写透 dcache 来源于实验资料包中的 ref_code/cache。
6. cp0_reg 来源于实验资料包中的 ref_code。
7. sram 转类 sram(X_sram_to_sram_like.v)来源于系统结构实验二的示例代码。
8. mycpu_top.v 和 mips.v 的接口参考了系统结构实验二的示例代码中的 mycpu_top.v 的接口。
9. cpu_axi_interface.v 直接使用了龙芯提供的转接代码。
10. 将走 cache 和不走 cache 的信号分开的模块 bridge1x2 和 bridge2x1 的接口参考了系统结构实验二的示例代码中的 bridge1x2.v 和 bridge2x1.v 的接口。
11. AXI 1x2 bridge,clk_pll,axi_clock_converter 为 vivado ip 核。

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

两人一起添加指令。

6.1.2 完成情况

修改 aludec(添加 RELU 的 alucontrol),maindec(添加 RELU 的译码),opcodedefines.vh 文件中添加 RELU 的 opcode 和 aludefines.vh 文件中添加 RELU 的 alucontrol,在 alu 中添加 RELU 操作。添加指令逻辑思路清晰正确,编译通过。

6.2 现场答辩记录

6.2.1 问题 1

- 问题:cp0 的 BadVaddr 寄存器是放什么的?
- 王世豪回答:地址错例外的那一条指令的虚地址,地址错例外是 pc 或 lw 或 sw 的时候地址没有对齐导致的。

6.2.2 问题 2

- 问题:cp0 的 epc 寄存器是放什么的?
- 王世豪回答:异常处理执行完后的返回地址,如果异常在延迟槽就要返回上一条,否则就是该条。

- 黄锐补充:异常分为异步异常和同步异常,前者的返回地址是例外地址的下一条,后者的返回地址是例外的地址。

6.2.3 问题 3

- 问题:AXI 是不是用的转接桥?
- 王世豪回答:是的,从 mips 中的 sram_to_sram_like 将 sram 接口转为 sram_like 接口,然后经过 mmu 模块确定是否要过 dcache。对于指令来说必须要经过 icache,经过了 icache miss,进入 axi 转接桥。对于数据来说,首先根据是否经过 dcache 分为两条路,一条经过 cache,另一条直接传出去。如果 cachemiss 或者直接传出去,就通过 axi 转接桥送给外面的组件。

6.2.4 问题 4

- 问题:跳转放在了哪个阶段?
- 王世豪回答:Decode 阶段。

6.2.5 问题 5

- 问题:整个设计的冒险有哪些?
- 王世豪回答:分为数据冒险和控制冒险,对于数据冒险来说,要用到之前没有写回的数据,有的需要 stall 有的需要前推,stall 分为两类:一类是上一条指令还在 E 阶段,M 阶段才能拿到,D 阶段娶不到 M 阶段的值,必须 stall 一个周期;另一类是我们没有实现 E 阶段向 D 阶段的前推,参考 lab4 的设计,stall 了一个周期。前推部分也有两类,一类是 W 向 E 前推的,还有一类是 M 向 E 和 D 的前推的。

6.2.6 问题 6

- 问题:sram_like 的握手信号是什么逻辑?
- 王世豪回答:时序逻辑。

6.2.7 问题 7

- 问题:请你描述一下 d_cache 的状态机。
- 王世豪回答:d_cache 状态机分为三个状态,空闲、读 ram 的 RM 状态和写 ram 的 WM 状态,他们各自转移的逻辑是,当空闲状态遇到读 miss 时,会进入到 RM 状态,遇到写时无论是 hit 还是 miss,都要进入 WM 状态,其他情况保持不变,RM 在握手之后转到空闲状态,WM 也是,握手是指数据握手。

6.2.8 问题 8

- 问题:请问遇到异常 flush 的时候,div 已经进入到除法状态了,这个时候怎么处理呢?
- 黄锐回答:我们的设计中,flush 的时候 div_stall 不会置位,div 运算的结果按理来说会被 flush,以后都用不到了。

6.2.9 问题 9

- 问题:请你们各自说说过程中遇到令你影响深刻的问题。
- 黄锐回答:我们发现在除法最后一个周期的时候,会根据 op_data 的值改变得到的结果,而 op_data 在整个过程中发生了变化,而 op_data 是 alu 的 srca2E 传进来的,因此是 srca2E 出了问题,刚开始的 srca2E 是 M 阶段前推过来的,但是那个时候我们的 div 只 stall 了前三个阶段,因此,在 M 阶段的指令流走了之后,前推也没了。虽然 M 阶段的那条指令已经写入了 regfile,但是由于 stall 的存在,不能把正确的值读到 E 阶段。
- 王世豪回答:axi 的 sram 转类 sram 的接口中,我们的接口用的是系统结构提供的那个接口,但是我们发现那个接口加上 dcache 就炸了 perfdiff,对着波形图找了很久,最后发现 d_sram_to_sram_like 那个地方 wen 是 0000,但是出来的 size 是 10,地址低两位不是 00 而是 10,但 1010 不是一个合法组合,所以就把低两位直接改成 0,就可以过 perfdiff 了。

7 总结

7.1 黄锐总结感想

在我们的设计中,使用了 AXI 转接桥来连接 CPU 和其他系统组件,写透 Cache 来减少内存访问的延迟,多周期乘除法器来提高乘除法的运算效率。在实现过程中,我还很好的处理了竞争冒险现象和通路设计,从而提升了 CPU 的性能。我们的设计思路主要是利用了时钟频率的提升和 Cache 的局部性原理来提升 CPU 的性能。首先,我将一个周期的事情细化为多个周期,这样就可以提升时钟频率,从而提升 CPU 的总体运算效率。其次,我利用 Cache 的局部性原理,即程序往往是顺序执行的,所以如果一个数据被使用了一次,那么很可能在不久的将来还会被使用到。因此,我在设计 Cache 时采用了写透 Cache 的方式,使得 Cache 中的数据能够及时更新,从而提高 CPU 的平均 CPI。对于这个设计对我们在计算机组成原理和系统结构方面的收获,我们认为这次设计对我的收获非常大。

在这次设计中,我们不仅学习了很多关于 CPU 设计的基本知识,如 AXI 转接桥的工作原理、Cache 的设计方法、乘除法器的实现方式等,还了解了很多关于竞争冒险现象和通路设计的知识,并学会了如何处理这些问题。此外,我们还学会了如何利用时钟频率的提升和 Cache 的局部性原理来提升 CPU 的性能。总的来说,这次设计对我在计算机组成原理和系统结构方面的收获非常大,也提升了我们对计算机 CPU 设计的理解。我们觉得在计算机领域,实践经验是非常重要的,而这次设计就是一次很好的实践机会。我们相信,通过这次设计,我们的计算机组成原理和系统结构方面的知识将会得到进一步的巩固,并为我们以后的学习和工作打下了坚实的基础。同时感谢老师助教在我们设计遇到困难的时候的指导和支持,祝老师和助教身体健康、工作顺利、青春常在、桃李满园、平安幸福、事事顺利、健康快乐、心想事成、美梦成真、万事如意、鹏程万里、合家欢乐、春风得意、马到成功、事业有成!

7.2 王世豪总结感想

1. BUG 太难调了,稍不注意就进入了无法 Debug 的神秘空间。
2. 硬综期间可以培养废寝忘食的习惯,有利于减肥。
3. 做好硬综需要相信奇迹。

参考文献

[1] 吕昱峰. 一步一步写 mips cpu. https://github.com/lvyufeng/step_into_mips, 2017.

[2] 重庆大学计算机学院. 重庆大学硬件综合设计实验文档. https://co.ccslab.cn/, 2022.

[3] 重庆大学计算机学院. 重庆大学计算机组成原理、硬件综合设计实验材料. https://github.com/CQU-CS-LABs/CO-lab-material-CQU, 2022.