华中科技大学

2017

计算机组成原理　　课程设计报告

题　　目：	5 段流水 CPU 设计
专　　业：	计算机科学与技术
班　　级：	CS1409
学　　号：	U201414810
姓　　名：	杨卿
电　　话：	15272773793
邮　　件：	185554981@qq.com
完成日期：	2017-03-25 周五下午

计算机科学与技术学院

目 录

1 课程设计概述

1.1 课设目的

计算机组成原理是计算机专业的核心基础课。该课程力图以“培养学生现代计算机系统设计能力”为目标，贯彻“强调软/硬件关联与协同、以 CPU 设计为核心/层次化系统设计的组织思路，有效地增强对学生的计算机系统设计与实现能力的培养”。课程设计是完成该课程并进行了多个单元实验后，综合利用所学的理论知识，并结合在单元实验中所积累的计算机部件设计和调试方法，设计出一台具有一定规模的指令系统的简单计算机系统。所设计的系统能在 LOGISIM 仿真平台和 FPGA 实验平台上正确运行，通过检查程序结果的正确性来判断所设计计算机系统正确性。

课程设计属于设计型实验，不仅锻炼学生简单计算机系统的设计能力，而且通过进行中央处理器底层电路的实现、故障分析与定位、系统调试等环节的综合锻炼，进一步提高学生分析和解决问题的能力。

1.2 设计任务

本课程设计的总体目标是利用 FPGA 以及相关外围器件，设计五段流水 CPU，要求所设计的流水 CPU 系统能支持自动和单步运行方式，能正确地执行存放在主存中的程序的功能，对主要的数据流和控制流通过 LED、数码管等适时的进行显示，方便监控和调试。尽可能利用 EDA 软件或仿真软件对模型机系统中各部件进行仿真分析和功能验证。在学有余力的前提下，可进一步扩展相关功能。

1.3 设计要求

（1） 根据课程设计指导书的要求，制定出设计方案；

（2） 分析指令系统格式，指令系统功能。

（3） 根据指令系统构建基本功能部件，主要数据通路。

（4） 根据功能部件及数据通路连接，分析所需要的控制信号以及这些控制信号的有效形式；

（5） 设计出实现指令功能的硬布线控制器；

（6） 调试、数据分析、验收检查；

（7） 课程设计报告和总结。

1.4 技术指标

（8） 支持表 1.1 前 27 条基本 32 位 MIPS 指令；

（9） 支持教师指定的 4 条扩展指令；

（10）支持多级嵌套中断，利用中断触发扩展指令集测试程序；

（11）支持 5 段流水机制，可处理数据冒险，结构冒险，分支冒险；

（12）能运行由自己所设计的指令系统构成的一段测试程序，测试程序应能涵盖所有指令，程序执行功能正确。

（13）能运行教师提供的标准测试程序，并自动统计执行周期数

（14）能自动统计各类分支指令数目，如不同种类指令的条数、冒险冲突次数、插入气泡数目、load-use 冲突次数、动态分支预测流水线能自动统计预测成功与失败次数。

表 1.1 指令集

#	指令助记符	简单功能描述	备注
1	ADD	加法	指令格式参考 MIPS32 指令集，最终功能以 MARS 模拟器为准。
2	ADDI	立即数加	
3	ADDIU	无符号立即数加	
4	ADDU	无符号数加	
5	AND	与	
6	ANDI	立即数与	
7	SLL	逻辑左移	
8	SRA	算数右移	
9	SRL	逻辑右移	
10	SUb	减	
11	OR	或	
12	ORI	立即数或	
13	NOR	或非	

#	指令助记符	简单功能描述	备注
14	LW	加载字	
15	SW	存字	
16	BEQ	相等跳转	
17	BNE	不相等跳转	
18	SLT	小于置数	
19	STI	小于立即数置数	
20	SLTU	小于无符号数置数	
21	J	无条件转移	
22	JAL	转移并链接	
23	JR	转移到指定寄存器	If $v0==10 halt(停机指令)
24	SYSCALL	系统调用	else 数码管显示$a0 值
25	MFC0	访问 CP0	中断相关，可简化，选做
26	MTC0	访问 CP0	中断相关，可简化，选做
27	ERET	中断返回	异常返回，选做
28	XORI	XORI	
29	异或立即数	异或立即数	
30	SLTIU	SLTIU	
31	小于立即数置 1(无符号)	小于立即数置 1(无符号)	

2 总体方案设计

2.1 中断机制设计

2.1.1 总体设计

中断是指 cpu 停下当前正在执行的程序转去执行中断服务子程序，当执行完中断服务子程序后再转回继续执行主程序。中断大概可以分为四个过程：中断请求，中断响应，中断服务，中断返回，具体流程如图 2.1。在现代微机系统中中断请求一般由 82C59A 中断控制器发出，我们直接由按钮产生一个正脉冲表示发出中断请求信号。中断响应则需 cpu 经过判优等后在决定是否立即响应当前中断请求，具体实现可以参照中断仲裁电路实现。中断服务首先应实现中断隐指令，中断隐指令实现关中断、保存断点、中断识别等任务。中断服务包括保护现场、设置新的屏蔽字、执行设备中断服务子程序、恢复现场等操作，其间单级中断与多级中断不同之处在于中断是否可以嵌套，所以对于多级中断在中断服务子程序中适当的时候会多次开、关中断。最后执行完中断服务子程序后就是中断返回，主要实现恢复 PC 值，从而能够返回中断前 PC 的地址继续执行主程序。

MIPS 中断控制一般通过协处理器 CP0 完成，有两条指令负责 CP0 通用寄存器的读写（MFC0，MTC0），中断使能位 IE，中断屏蔽位均可以利用 CP0 完成，在 CP0 中设置 3 个寄存器：status、cause 和 epc，分别用来保存中断使能位、中断源编号和主程序 PC 值。开关中断可以直接通过 status 置位、复位实现。用 cause 寄存器保存当前执行中断服务子程序的中断源编号是为了实现多级中断嵌套的时候各级中断的不同优先级。保存现场需要将程序现场压入栈中，直接在数据存储器中开辟一段空间用于实现栈，保存寄存器文件中的现场可以直接用 sw 指令实现，而保存 cp0 中现场则需要先将 cp0 中现场写入寄存器文件（mfc0 指令），然后在将对应寄存器写入栈中。恢复现场则与保存现场是相反的过程，若要恢复寄存器文件的内容，则直接用 lw 从数据存储器中读取对应地址数据重新加载到寄存器文件中，若要恢复 cp0 内寄存器内容，则需要先用 lw 指令将内容保存到寄存器文件中，然后用 mtc0 指令将寄存器文件内容写入 cp0 对应寄存器中。前面一直没有提到 lw 和 sw 指令从数据存储器中读、写

数据的地址，其实只要在恢复现场时与保存现场时寄存器的顺序保持是逆序即可，这是因为栈先进后出的特性造成的。

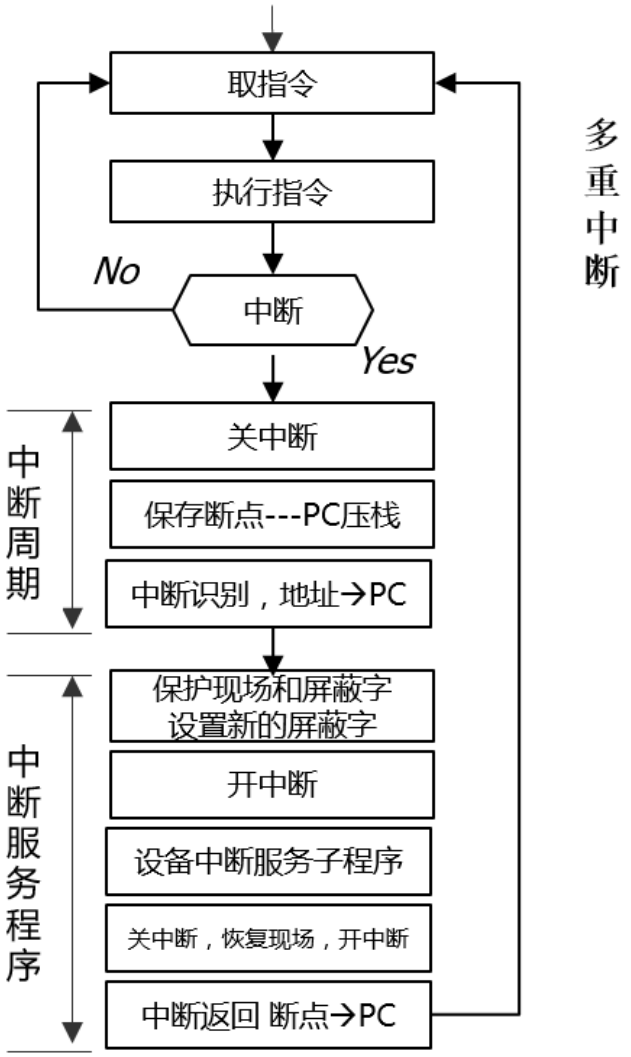

图 2.1 中断服务子程序流程图

有关多级中断嵌套具体是如何实现的，参考了课程上面介绍的中断仲裁电路如图2.2，核心是中断屏蔽字的设置和清除，这样可以实现同级中断的屏蔽，然后对于多个中断请求信号同时到来时，采用优先编码器实现先处理高优先级中断，而低优先级中断需要等待。

关于中断服务子程序的存储地址，其实 DOS 系统的处理方式是通过向量中断的方式存放中断服务子程序入口地址，中断向量表存储在主存地址 0000H~03FFH，但是我为了寻址方便，增加了一个指令存储器，在响应中断时，选择这个存储器，执行预设在其中的中断服务子程序。关于软件方面，我在里面实现了保存现场，中断服务程序，恢复现场等功能。

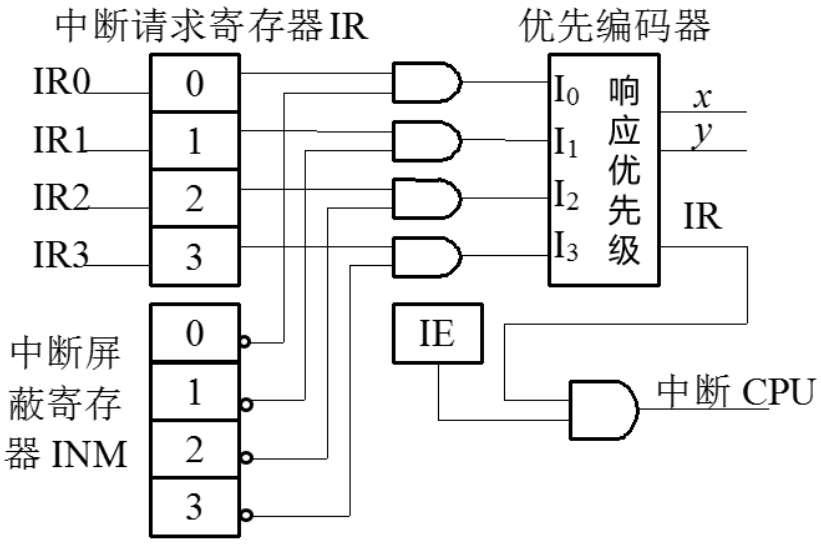

图 2.2 中断仲裁电路

2.2 流水 CPU 设计

2.2.1 总体设计

单周期时即使指令之后的执行不再需要使用前面用到的部件，还仍然霸占着那些部件。合理而高效的做法自然是让指令执行时不再需要使用前面的部件时，应该调入下一条指令执行。所以关键问题是：如何将单周期数据通路进行划分，当指令未执行完而调入新的指令的必须要使用寄存器保存指令当前的执行信息，如何设置流水寄存器内容。当完成理想流水线之后，在统计时钟周期数时会遇到 sys 指令的处理，sys 指令涉及到读寄存器 v0 内容，而其前一条指令的写寄存器正好是 v0，在那个阶段处理 sys 指令才会得到正确的统计结果。从 sys 指令的处理就会引出数据相关，除了数据相关，流水线中还存在结构相关和分支相关，另外还有一种特殊的 load-use 冲突，如何解决这些流水线中的相关。所以主要问题有三个：流水寄存器的设置、执行指令时存在着相关的情况和 load-use 冲突的解决方法，解决了上面所有问题的 cpu 叫做全冒险处理机制的流水 cpu。

意识到理想流水中存在数据冒险，当执行 sys 指令时需要获取 v0 寄存器的最新值以决定是否停机。理想流水停机信号应在那个阶段处理？何时停机？在 WB 阶段处理 sys 指令，因为此时上一条指令已经写回（已修改寄存器文件为下降沿写回），此时再取 vo 寄存器值已是最新，且当 sys 指令执行到 WB 阶段时其已执行完毕，此

时产生停机信号停止时钟便可得到正确的时钟数统计。结构冒险指的是流水线中的指令在执行时同一时刻不同阶段的指令会访问相同的部件，从而造成争用，比如最先考虑到的就是取指和访存阶段都会用到存储器，所以老师在一开始的时候就介绍了哈佛结构：指令存储器和数据存储器分开（实际商用的处理器是分开的指令 cache 与数据 cache）。数据相关可以采用重定向的方法解决，真数据相关也就是 RAW，只需要检测相邻指令读寄存器编号与写寄存器编号一致时就采用定向技术（旁路），将正确的数据（还未来得及写回）通过新增的数据通路送到需要读该寄存器的地方。但是有一种数据相关出现 load-use 冲突时不能简单地重定向，而且有关数据冲突或者说 load-use 冲突必须要在 ID 段监测，否则会造成额外的硬件开销，具体原因在后面实现时具体介绍。实际上定向中还存在一些值得注意的问题，也在详细实现的时候在具体介绍。最后介绍分支相关的解决办法，其实我们是采用静态分支预测中的预测失败的方法，当监测到分支指令时并不做特别的处理，等到分支成功或者失败的结果计算出来时，若是分支失败则分支指令和普通指令没有任何区别，一旦分支成功，就要清楚误取深度条指令，从新从分支地址出取出正确的指令。当处理完所有的冒险后我们就实现了全冒险处理的五段流水 cpu，才能正确地运行 benchmark，全冒险处理机制的流水 CPU 电路图如图 2.3。

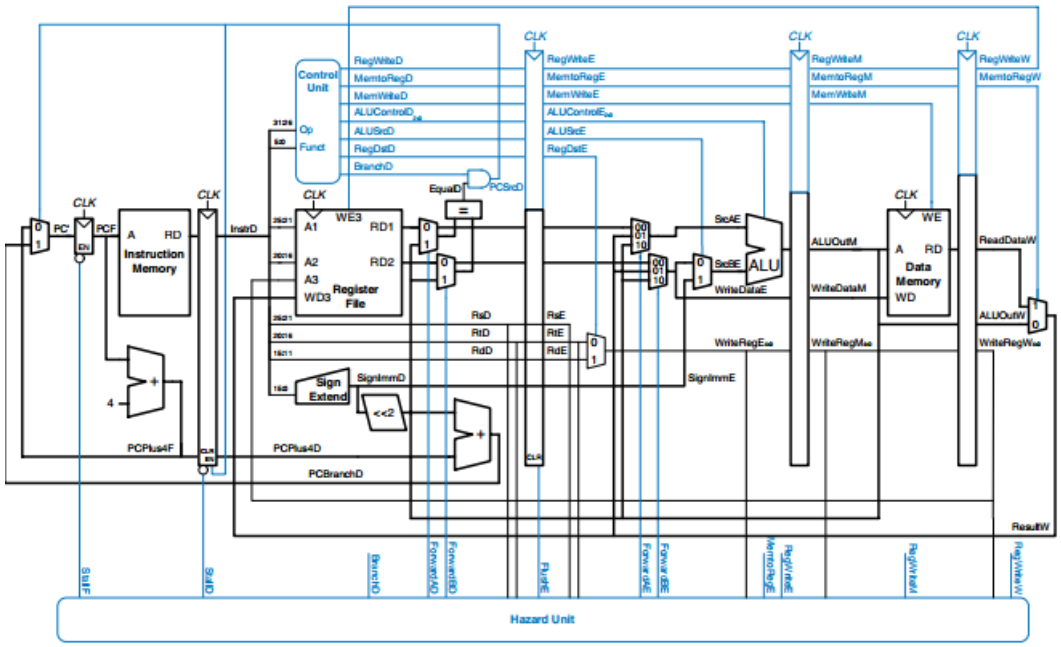

图 2.3 全冒险处理机制的流水 CPU

2.2.2 流水接口部件设计

指令的解释执行过程可以很清晰地分为 5 个阶段：取指令、译码、执行、存储器访问、写回，并且这五个阶段在不同的功能部件上执行，故可以让五个子任务同时并发执行。流水寄存器中应该包含指令在后续解释执行阶段所需的所有信号，并且为了便于调试最好将 PCPlus4、IR 信号也全部传递。如图可以看到 IF/ID 寄存器中保存着 PCPlus4 和 IR 信号，而 ID/EX 寄存器中包含着指令在 EX 执行所需要的全部信号和指令在 MEM 和 WB 阶段需要的所有信号，EX/MEM 寄存器包含着指令在 MEM 段执行所需要的全部信号和指令在 WB 段执行的全部信号，MEM/WB 寄存器则包含着指令在 WB 段执行所需要的全部信号，因此流水寄存器不仅需要包含指令在本段执行所需要的信号，还要将指令在后续阶段所需要的信号逐步传递给后续流水寄存器（相当于实现了控制器的复用）。考虑到后面需要实现流水线的停顿和指令的清空，需要流水寄存器中需要有使能信号和清零信号，流水寄存器中传递的信号及在各段的使用情况如图 2.4，流水寄存器信号在各段的使用情况如图 2.5。

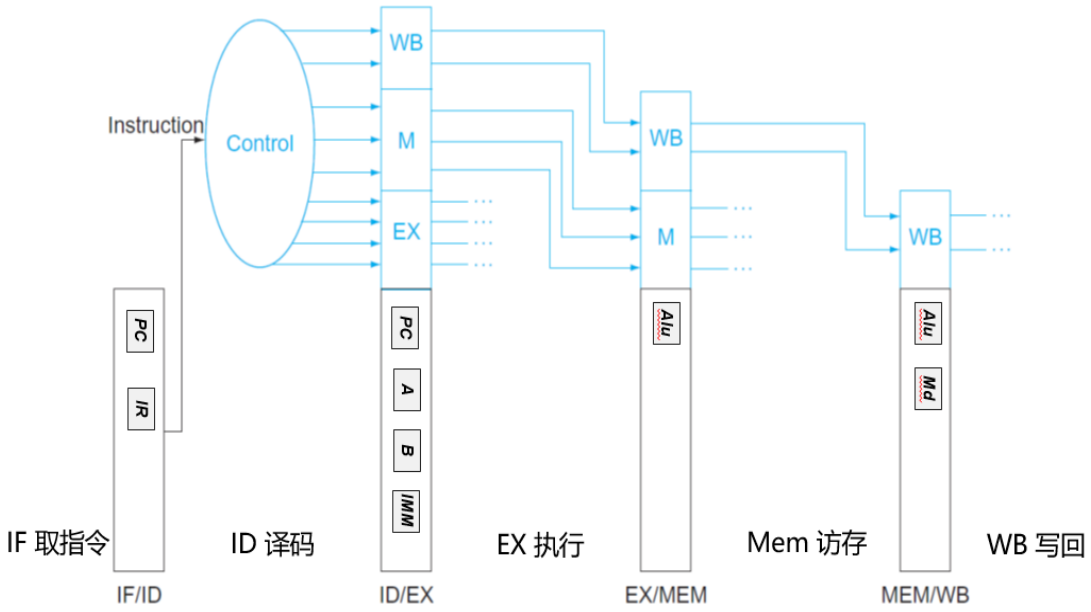

图 2.4 流水寄接口设计及信号传递示意

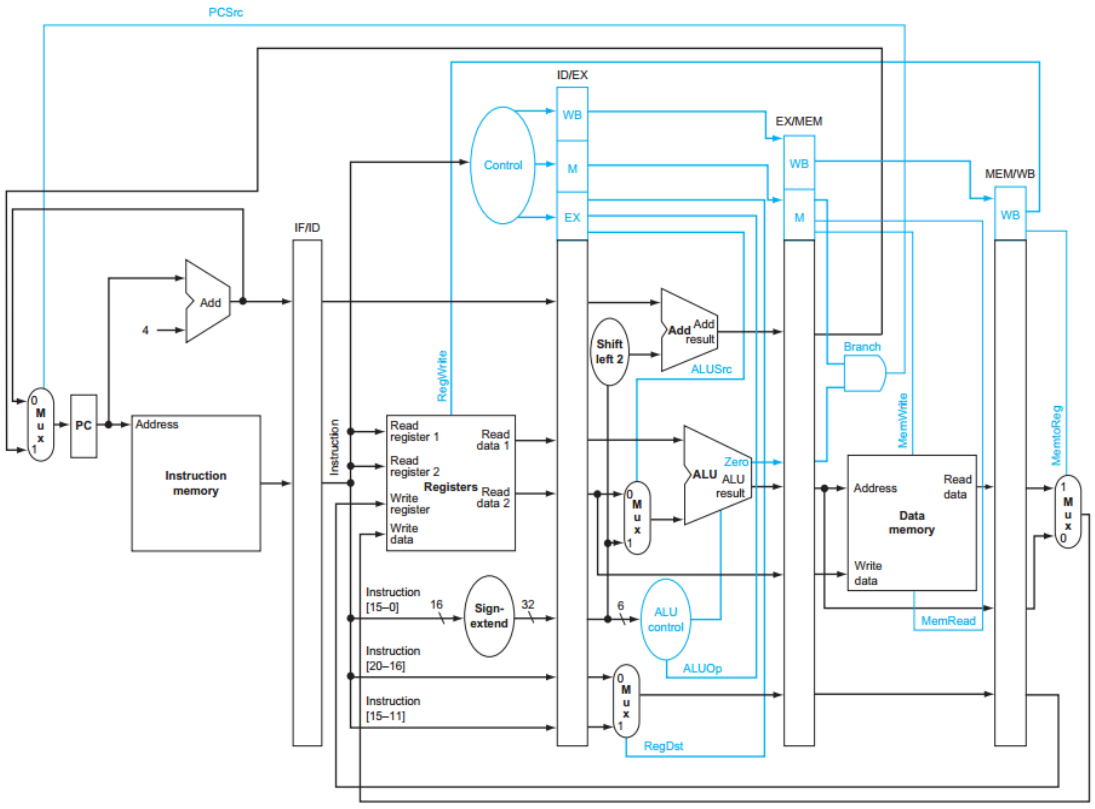

图 2.5 流水寄存器信号在各段使用情况

2.2.3 理想流水线设计

对单周期数据通路进行划分，形成流水线中不同的段，由于采用统一时钟，最终的 cpu 频率取决与五个阶段中最慢的那个阶段（取最慢的同步），所以在划分五个阶段时尽量做到各段传输时延一致。根据上面的分析，五段流水线的最大吞吐率是单周期的 5 倍，那么为什么不把流水线划分成更多段呢？理论上理想流水线最大吞吐率与段数成正比，但是在实际中寄存器存在读、写延迟实际和时钟便宜开销问题，所以理想流水线的段数在实际中不是可以无限大的（不能无限细分的）。之前提到由于理想流水线的最大吞吐率取决与瓶颈段，所以在划分各段时一定要保持各段时延基本一致，如果由于不可抗力原因存在存在瓶颈（比如耦合度高的逻辑不易解耦），通常可以采用细分瓶颈段和重复设置瓶颈段，当然我们的 cpu 逻辑相对于商业 cpu 来说比较简单，并不会出现这样的问题。理想流水线需要正确实现 sys 指令以完成对周期数的统计功能，sys 指令需要读取寄存器 v0 的值，然而当 sys 指令在 ID 段取 v0 寄存器值的时候，前面的指令（目的寄存器也为 v0）的结果此时还未写回，因此 sys 取了错误的没有及时更新的 v0，所以 sys 的判断条件不成功，因此不会正确停机，一个简单的解

决方法是在 WB 阶段再处理 sys 指令，也就意味着 sys 信号要一直传递到 MEM/WB 流水寄存器中，当 sys 指令执行到 WB 段时在从寄存器文件中读取寄存器 v0 的值，前面的所有指令一定已经执行完了，所以取得的 v0 值一定是在正确的。当构建好了流水寄存器，正确处理 sys 指令后就可以正确运行理想流水线测试程序得到正确的执行结果和周期统计数了，理想流水线顶层设计框图如图 2.6。

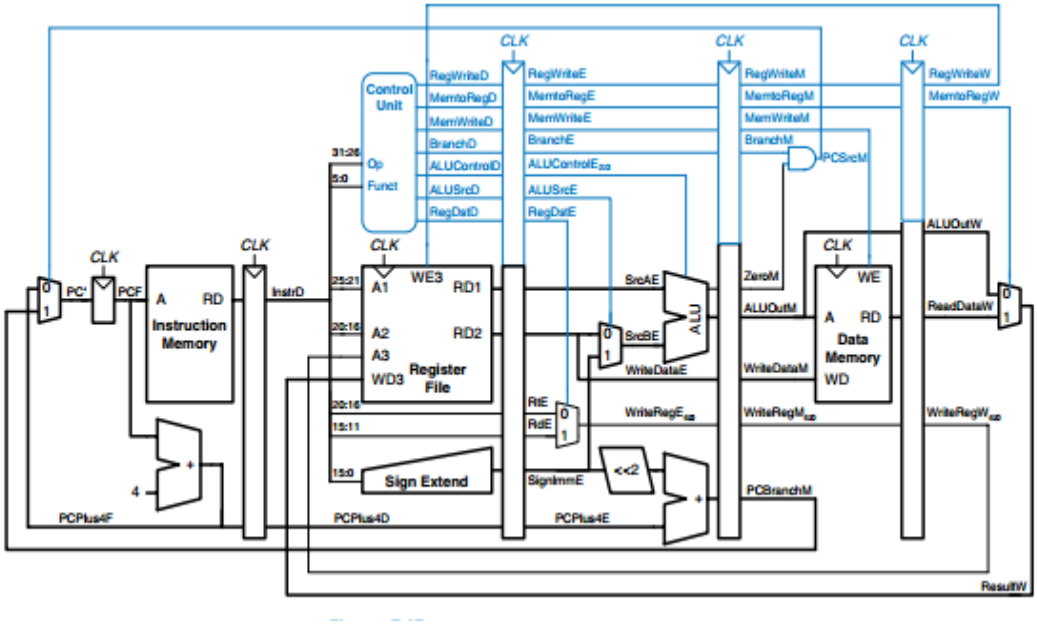

图 2.6 理想流水线顶层设计框图

2.3 数据转发流水线设计

我把 2.4 和 2.5 的题目交换了，因为重定向可以解决大部分的数据冲突，只有在存在 load-use 冲突时需要插入气泡，而如果采用出现数据冲突就停顿、插入气泡的方法，那是效率极低的做法。所以我觉得逻辑关系应该是重定向可以解决大部分数据冲突，而插气泡只是针对重定向无法解决的 load-use 冲突的一个修正，所以先介绍数据转发技术。

首先数据转发技术和重定向、旁路等所指的概念是一样的，是为了解决流水线中存在的 RAW 写后读冲突，其实在理想流水线的 sys 指令处理过程中已经体现了这个问题会带来的程序执行可能会出现错误的后果。后面指令读寄存器编号和前面指令写寄存器编号相同时就会出现真数据相关，该问题存在的原因是执行结果未写回寄存器文件，后续指令就需要从寄存器文件中读取还未来得及写入的寄存器的值，但是实际上这些需要写入的值都已经在数据通路中计算出来了，只是没有写回而已，所以我们

可以通过构建额外的数据通路（旁路）来将这些数据送到需要使用它们的地方，原理如图 2.7。

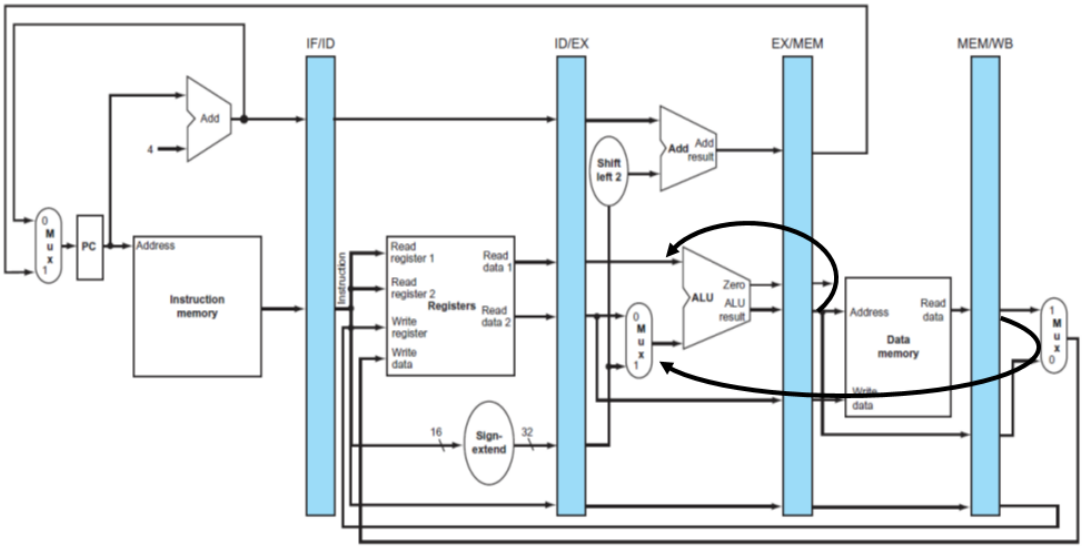

图 2.7 重定向示意图

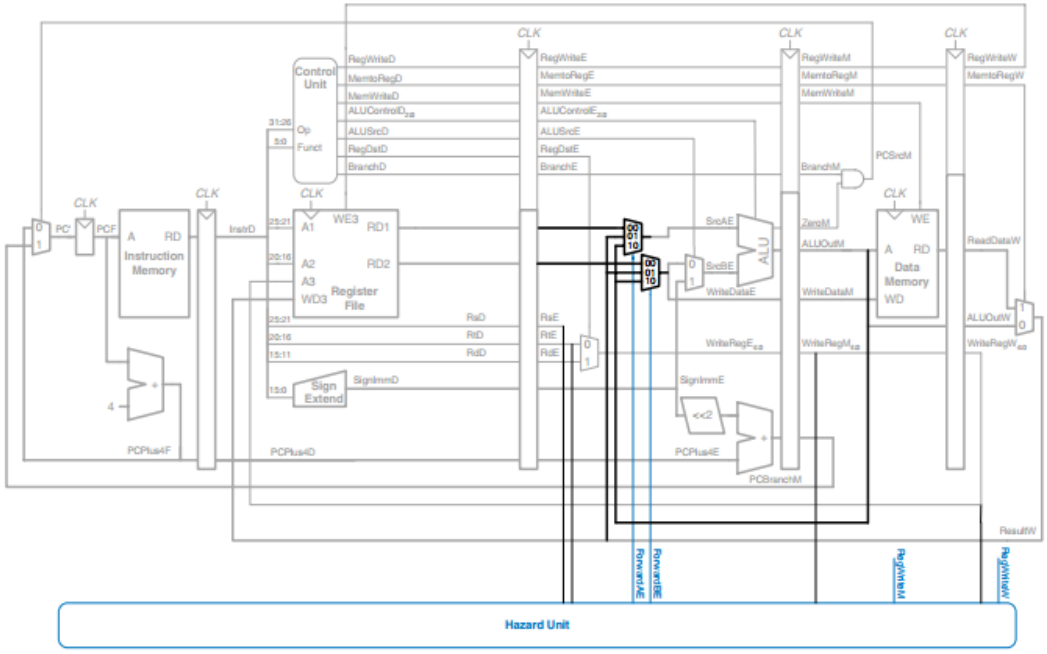

图 2.8 实现 MEM 和 WB 重定向的数据旁路

冒险单元应该检测 EX 段读的寄存器编号（最多可能是两个），与 MEM 和 WB 段指令写寄存器编号是否出现相同的情况，若出现了，则数据冲突就发生了，所以需要将数据送到 EX 段对应位置，如果编号不相同，则正常执行，无需重定向。这里有两个问题需要注意，当读、写 0 号寄存器时，是否需要重定向，答案是不需要，因为 0 号寄存器内容始终未 0，一定不会出现数据冲突。如果读寄存器编号与 MEM 和 WB

段写寄存器编号均相同，应该重定向那个段的数据，显示应该重定向 MEM 段的的数据，因此 MEM 段的数据重定向具有更高的优先级，在实现的时候只需将输入端为 MEM 段数据的二路选择器放在后面即可。当上述问题解决之后，数据转发流水线也就完成了，重定向具体实现如图 2.8。

2.4 气泡式流水线设计

在完成了数据转发流水线之后观察这样的现象，当连续两条指令，其中前一条是 lw 指令，而后面的一条指令的源寄存器编号与 lw 指令的目标寄存器编号相同时（load-use 冲突）如图 2.9，这属于重定向的解决范围—RAW，但是请思考 EX 段指令的时延，那将是访问与 ALU 运算串行进行的总时间，因此如果在这种情况下仍然直接使用重定向，在不出错的情况下，会造成 cpu 频率变低，而这对 cpu 特别是商用 cpu 是无法接受的，因此必须采用其他的方法来解决这种情况。

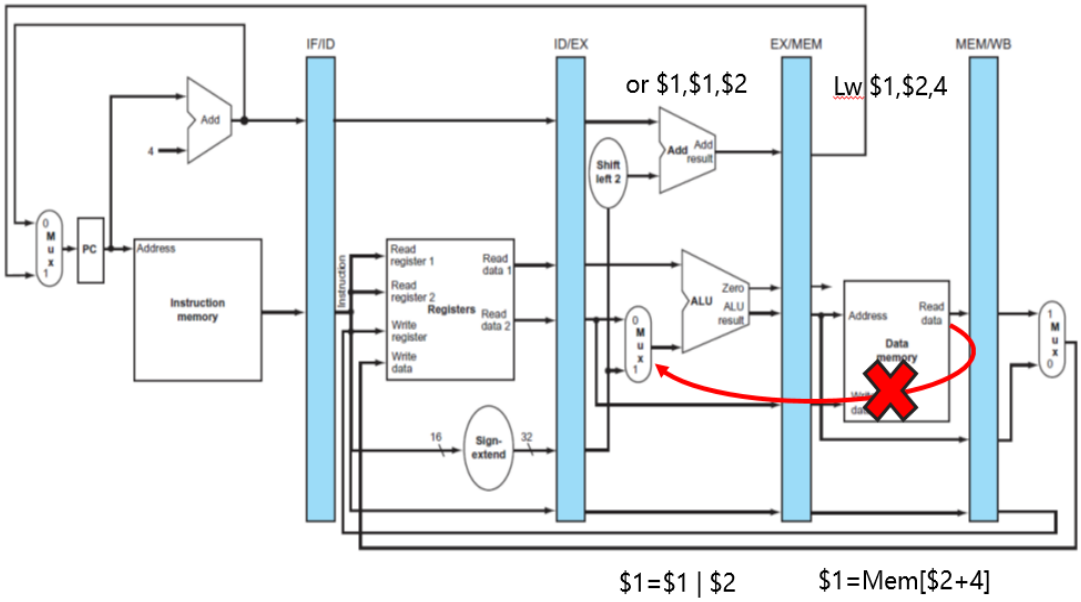

图 2.9 load-use 冲突示意图

直观的想法是等到 MEM 段的指令在经历一个时钟周期，让数据锁入流水寄存器中，然后重定向，这样就不会出现执行路径时间过长的问题了了，具体来说就是插入一个气泡并停下流水线一个时钟周期。load-sue 冲突的检测就是在重定向的检测上加上判定需要重定向的指令的前一条指令是否为 lw 指令即可，但是这里存在一个非常关键的问题：到底在那个段检测 load-use 冲突。答案是必须在 ID 段，否则将必须以增加硬件开销为基础来保证程序不出错。可以假设在 EX 段检测，考虑这样的情况，

MEM 段和 WB 段执行的指令都为 lw 指令，且两条 lw 指令的目的寄存器编号正好与 EX 段指令源寄存器编号相同，若插入一个气泡，WB 段 lw 指令将被“挤走”，重定向的数据被丢失，而 EX 段指令又未从寄存器文件中取得正确的源寄存器值，因此就造成了程序执行可能出现错误。考虑在 ID 段检测 load-use 冲突，如果在 EX 段与 MEM 段的指令出现与在 EX 段检测 load-use 冲突相同的情况，插入一个气泡并站听 ID 段及 IF 段流水线，那么最先的 lw 指令将会流动到 WB 段，由于改造了寄存器文件为下降沿写入，因此下一个时钟上升沿来临时，正确的源寄存器值将被锁入 ID/EX 流水寄存器中。

综上所述，load-use 冲突需要插入气泡并暂停流水线，且关于 load-use 冲突的检测必须在 ID 段。气泡式流水线原理图如图 2.10。

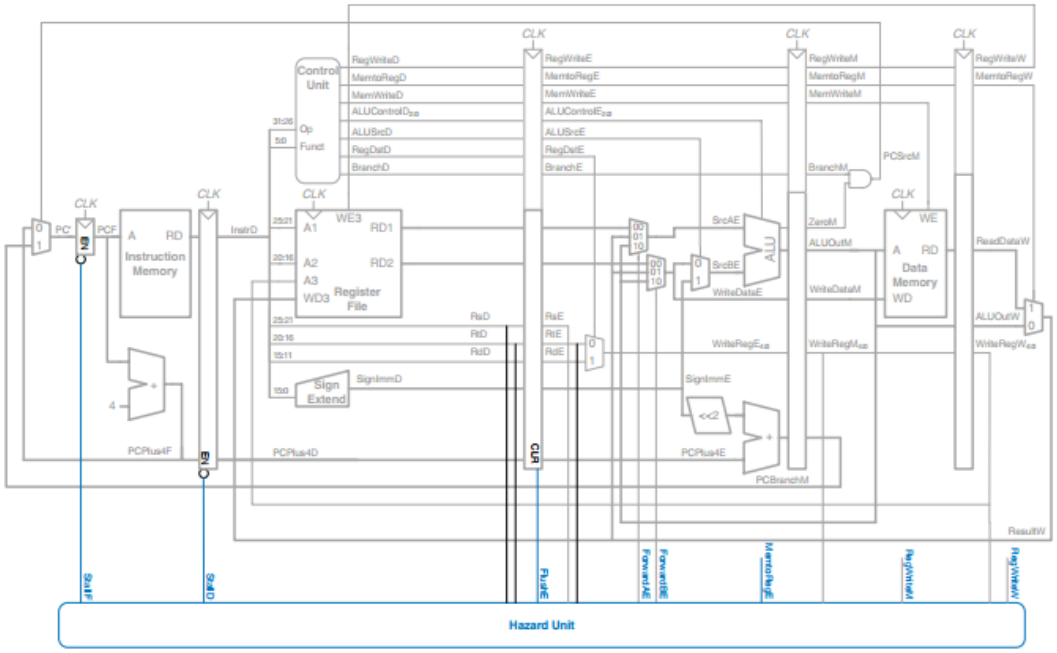

图 2.10 气泡式流水线原理图

3 详细设计与实现

3.1 中断机制实现

3.1.1 中断硬件实现

1) 中断响应和中断返回

在我的 cpu 里面没有使用向量中断的方法寻找中断服务子程序，而是新建了另一个指令存储器，在这个指令存储器里面专门存储中断服务子程序。

① Logism 实现：

使用一个只读存储器 ROM2 实现中断指令的存储。设置该只读存储器的地址位宽为 9 位，数据位宽为 32 位。因为 PC 中存储的指令地址有 32 位，而 ROM 地址线宽度有限，仅为 9 位，故将 32 位指令地址高位部分和字节偏移部分直接屏蔽，使用分线器只取 32 位指令地址的 2-10 位作为指令存储器的输入地址。取 PC 第 11 位作为两个存储器的寻址信号，当 PC 第 11 位为高电平是选中下面存储中断指令的存储器 ROM1，当 PC 第 11 位为低电平时则正常选择上面存储着 benchmark 主程序的指令存储器。另外当中断响应信号 HasExp 到来时，PC 值变为 0800H，选中 ROM2 起始地址。当中断返回 Eret 信号来临时，将会把 CP0 中 EPC 寄存器的值锁入 PC 中以实现中断返回，具体实现电路图如图 3.1。

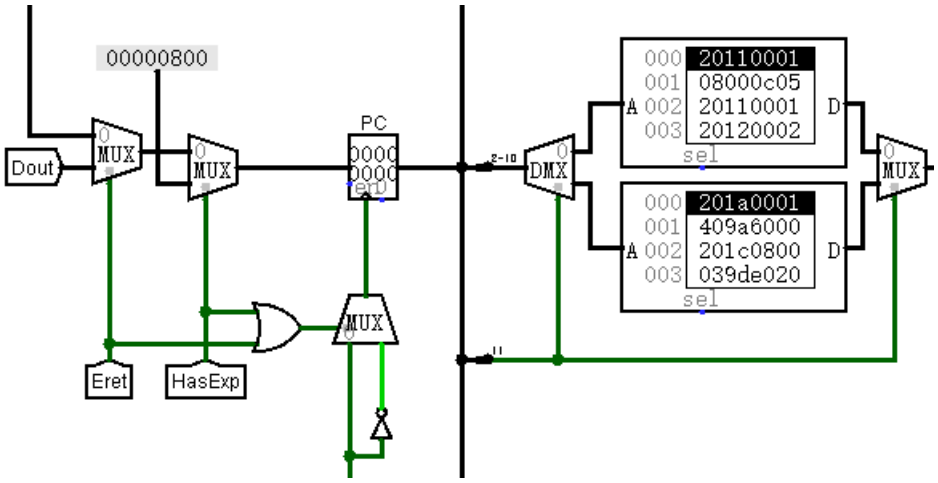

图 3.1 中断响应和中断返回硬件实现

② FPGA 实现：

直接使用 verilog 中的 assign 和三目选择运算符实现二路选择器和赋值。

中断响应和中断返回功能实现的 Verilog 代码如下：

```
assign clk1=(HasExp|EretEX)?(~clk):clk;
assign PC_sel=HasExp?32'h800:(EretEX?DoutEX:PC_next);
```

2) 中断仲裁

中断仲裁电路主要实现 CPU 对设备中断请求作出响应，其中要实现的细节还包括：中断识别，中断屏蔽和中断请求信号的及时清零。

① Logisim 实现：

为了消除按钮可能带来的毛刺，先将中断请求信息锁入寄存器内，然后在时钟来临时将将请求信号锁入 IR 寄存器中，从 Logisim 电路图中可以非常明显地看到系统只支持三个中断源，在 IE 中断使能位有效的情况下 CPU 才可能处理该中断请求，如果出现多个中断请求同时发出时，则通过优先编码器找到优先级最高的那个中断源，同时与当时系统中正在运行的中断的优先级进行比较，如果比较器发现当前中断请求优先级比系统正在运行的中断优先级更高，那么当前中断背会被打断。通过 cause 将终端号送给 CP0，而为了实现同步清零（一定是同步），等待一个时钟周期再将当前最高优先级的中断请求寄存器清零，实现电路图如图 3.2。

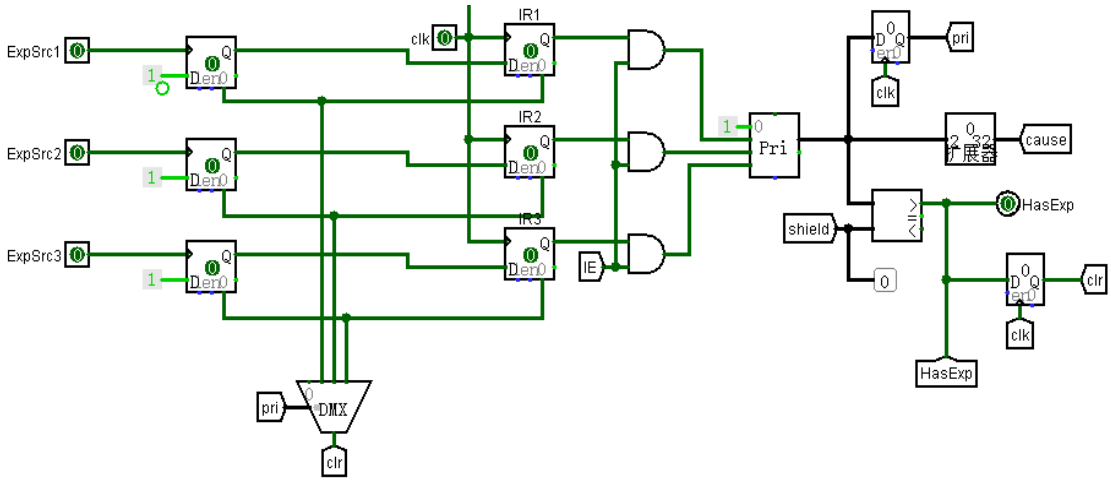

图 3.2 中断仲裁电路

② FPGA 实现

直接使用 verilog 中的 assign 实现组合逻辑的赋值，而对于包含时序关系的寄存器赋值则使用 always，三目选择运算符实现二路选择器。中断仲裁电路实现的 Verilog 代码如下：

```
always@(posedge ExpSrc1 or posedge clear[1])
```

```
begin
    if(clear[1]>0)
    begin
        reg1<=0;
    end
    else if(ExpSrc1>0)
    begin
        reg1<=1;
    end
end

always@(posedge ExpSrc2 or posedge clear[2])
begin
    if(clear[2]>0)
    begin
        reg2<=0;
    end
    else if(ExpSrc2>0)
    begin
        reg2<=1;
    end
end

always@(posedge ExpSrc3 or posedge clear[3])
begin
    if(clear[3]>0)
    begin
        reg3<=0;
    end
    else if(ExpSrc3>0)
```

```
    begin
        reg3<=1;
    end
end

always@(posedge clk)
begin
    if(clear[1])
    begin
    IR1<=0;
    end
    else
    begin
    IR1<=reg1;
    end
end

always@(posedge clk)
begin
    if(clear[2])
    begin
        IR2<=0;
    end
    else
    begin
    IR2<=reg2;
    end
end

always@(posedge clk)
```

```
begin
    if(clear[3])
    begin
    IR3<=0;
    end
    else
    begin
    IR3<=reg3;
    end
end

assign DoutID=EretID?EPC:(cp0regID[1]?EPC:(cp0regID[0]?cause:status));
assign shield=cause[1:0];
assign pri=(IE&IR3)?3:((IE&IR2)?2:((IE&IR1)?1:0));
assign exp_cause={30'b0,pri};
assign HasExp=(pri>shield);
assign clear[3]=(pri_reg[1]&pri_reg[0])&clr_reg;
assign clear[2]=(pri_reg[1]&(~pri_reg[0]))&clr_reg;
assign clear[1]=((~pri_reg[1])&pri_reg[0])&clr_reg;

always@(negedge clk)
begin
    pri_reg<=pri;
    clr_reg<=HasExp;
end
```

3) 协处理器 CP0 内寄存器设置

为了保存有关中断的信息，我设置了三个寄存器：status，cause，epc，分别用来实现中断使能寄存器（与开关中断指令 STI,CLI 相关，实际通过 Mtc0 指令实现），保存正在执行中断的中断号，保存中断发生了主程序的 PC 值。

① Logisim 实现：

对于 CP0 寄存器组来说和 CPU 寄存器文件类似，主要的还是寄存器的读写操作，CP0 寄存器的读、写分别是通过 Mfc0 和 Mtc0 实现的。从 Logisim 电路图中可以看到，当 HasExp 中断响应信号来临时，会将引起中断的中断号锁入 cause 寄存器中，同时也会将主程序 PC 值锁入 EPC 寄存器中。当然需要对 CP0 寄存器组进行写操作时，通过寄存器编号 cp0reg 进行端口寻址，Mtc0 指令使得 CP0 寄存器组时钟有效可以正常进行写操作。Mfc0 读 CP0 寄存器组指令同样的需要 cp0reg 提供端口地址，在中断返回信号 Eret 来临时，会直接读取 EPC 的值，重新装入 PC 寄存器中，使得主程序可以返回断点，继续执行。可能注意到两个隧道：IE 和 shield，其中 IE 是 status 寄存器的最低一位，而且只有等到时钟来临时才会将其值锁入寄存器中，这是为了在中断返回时让开中断与断电返回能够在硬件上并行执行，而指令的并行执行在传统串行编程中是不易实现的。如果不进行硬件上的并行处理，其实可以发现开中断与中断返回在多级中断嵌套时其实是矛盾的，无论那个先进行都可能会导致程序运行出现异常。其实 MIPS 处理器在真正实现的时候是因为管态和用户态的不同所以没有这个问题，但是在我们的实验没有涉及到。shield 隧道是将当前正在执行的中断的中断号送入屏蔽寄存器中，以实现屏蔽优先级比当前正在运行的中断优先级低或者相同的中断请求，寄存器组具体实现如图 3.3。

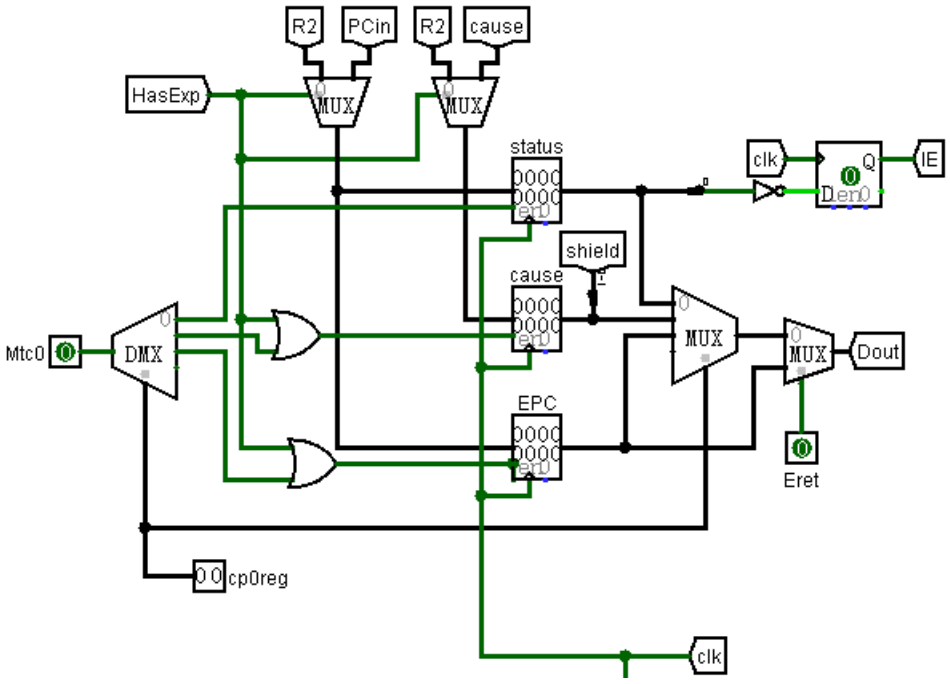

图 3.3 CP0 内寄存器组设置

② FPGA 实现：

对于包含时序关系的寄存器赋值使用 always 语句进行赋值，三目选择运算符实现二路选择器。实现 CP0 内寄存器组的 Verilog 代码如下：

```
always@(negedge clk)
begin
if((cp0regwriteEX==0)&Mtc0EX)
      begin
        status<=HasExp?PCin:R_R2EX;
      end
if(((cp0regwriteEX==1)&Mtc0EX)|HasExp)
      begin
      cause<=HasExp?exp_cause:R_R2EX;
      end
if(((cp0regwriteEX==2)&Mtc0EX)|HasExp)
      begin
      EPC<=HasExp?PCin:R_R2EX;
      end
end

always@(negedge clk)
begin
      IE<=~(status[0]);
end
```

当完成中断硬件部分后，剩余工作就需要软件来实现了，我们回顾一下，硬件部分实现了中断仲裁电路与 CP0 中寄存器文件的配置，其中中断仲裁电路实现了中断嵌套的优先级判断基础，并且其中中断请求寄存器的清零一定是同步清零，CP0 寄存器组用来实现中断使能位标识、保存断点信息和标识正在运行的中断源的中断号，顶层电路图如图 3.4。

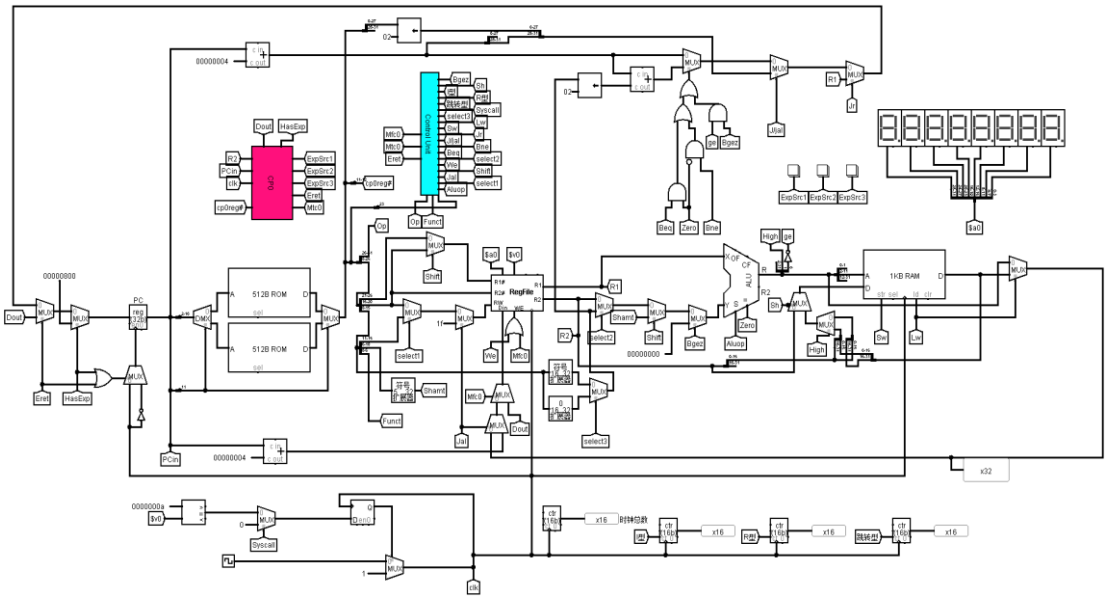

图 3.4 多级中断嵌套硬件实现电路图

3.1.2 中断软件实现

中断需要设置的硬件基础已经通过较为详细的介绍说明清楚，接下来介绍中断过程需要软件完成的部分。首先中断隐指令完成关中断，保存断点和中断识别，在中断服务子程序中需要完成的流程是：保护现场并设置新的屏蔽字，开中断，执行设备中断服务子程序，关中断，恢复现场，开中断，中断返回。其中需要特别注意的问题的由于栈先进先出的特性，出栈寄存器的顺序应该是进展寄存器的逆序。

```
#####################################################################
#中断测试,系统支持 3 个按键中断源，3>2>1>CPU,
#由按键触发中断演示程序，中断服务程序是对应按键号的走马灯
#####################################################################
.text
#close interrupt
addi $k0,$0,1
mtc0 $k0,$12

#store pc

addi $gp,$0,0x800
```

```
add    $gp,$gp,$sp
mfc0 $k1,$14
sw      $k1,0($gp)
addi $gp,$gp,-4

#store environment
sw      $a0,0($gp)
addi $gp,$gp,-4
sw      $s1,0($gp)
addi $gp,$gp,-4
sw      $s2,0($gp)
addi $gp,$gp,-4
sw      $s3,0($gp)
addi $gp,$gp,-4
sw      $s4,0($gp)
addi $gp,$gp,-4
addi $sp,$0,0x800
sub    $sp,$gp,$sp

#equal to identify    interrupt
mfc0 $s4,$13

#open interrupt
add    $k0,$0,$0
mtc0 $k0,$12

#interrupt program
add    $a0,$0,$0
add    $s1,$0,$0
add    $s2,$0,$0
```

```
add    $s3,$0,$0
addi $s3,$0,1
sll    $s1,$s4,28
addi $s2,$0,4
label2:
add    $a0,$0,$s4
label1:
sll    $a0,$a0,4
bne    $a0,$s1,label1
sub    $s2,$s2,$s3
bne    $s2,$0,label2

#close interrupt
addi $k0,$0,1
mtc0 $k0,$12

#restore environment
addi $gp,$gp,4
lw      $s4,0($gp)
addi $gp,$gp,4
lw      $s3,0($gp)
addi $gp,$gp,4
lw      $s2,0($gp)
addi $gp,$gp,4
lw      $s1,0($gp)
addi $gp,$gp,4
lw      $a0,0($gp)

#restore pc
mtc0 $s4,$13
```

```
#open interrupt
add    $k0,$0,$0
mtc0 $k0,$12

#return
sll    $0,$0,0
addi $gp,$gp,4
lw      $k1,0($gp)
mtc0 $k1,$14
eret
```

① Logisim 实现：

只需编写好中断的软件部分，通过 MARS 模拟器进行编译，然后加载到 ROM2 中即可。

② FPGA 实现：

同样地，只需编写好中断的软件部分然后通过 readmemh 指令读取到 ROM2 中即可。

```
initial
begin
      $readmemh("D:\\zhongduan.hex",rom2,0,511);
end
```

3.2 流水 CPU 实现

3.2.1 流水接口部件实现

流水接口部件就是指的流水寄存器，由于我们实现的是经典五段流水线，因此总共需要设置 4 个流水寄存器，分别是：IF/ID，ID/EX，EX/MEM，MEM/WB。ID/EX 流水寄存器中应该包括指令在 EX/MEM，MEM/WB 段需要使用到的信号，以实现控制器的复用。同时为了便于流水线的额调试，应当将 PCPlus4 和 IR 的内容全部传递。置于每一段的流水寄存具体传递了什么信号，这里给出内部图片，不再详细说明，因

为过于繁杂。

① Logisim 实现：

IF/ID 段流水寄存器内部设置如图 3.5：

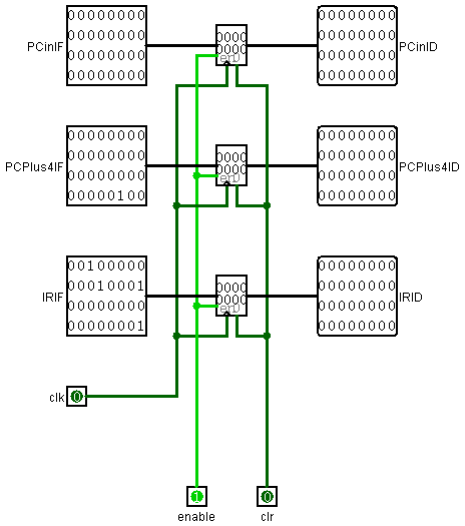

图 3.5 IF/ID 段流水寄存器内部

ID/EX 段流水寄存器内部设置如图 3.6：

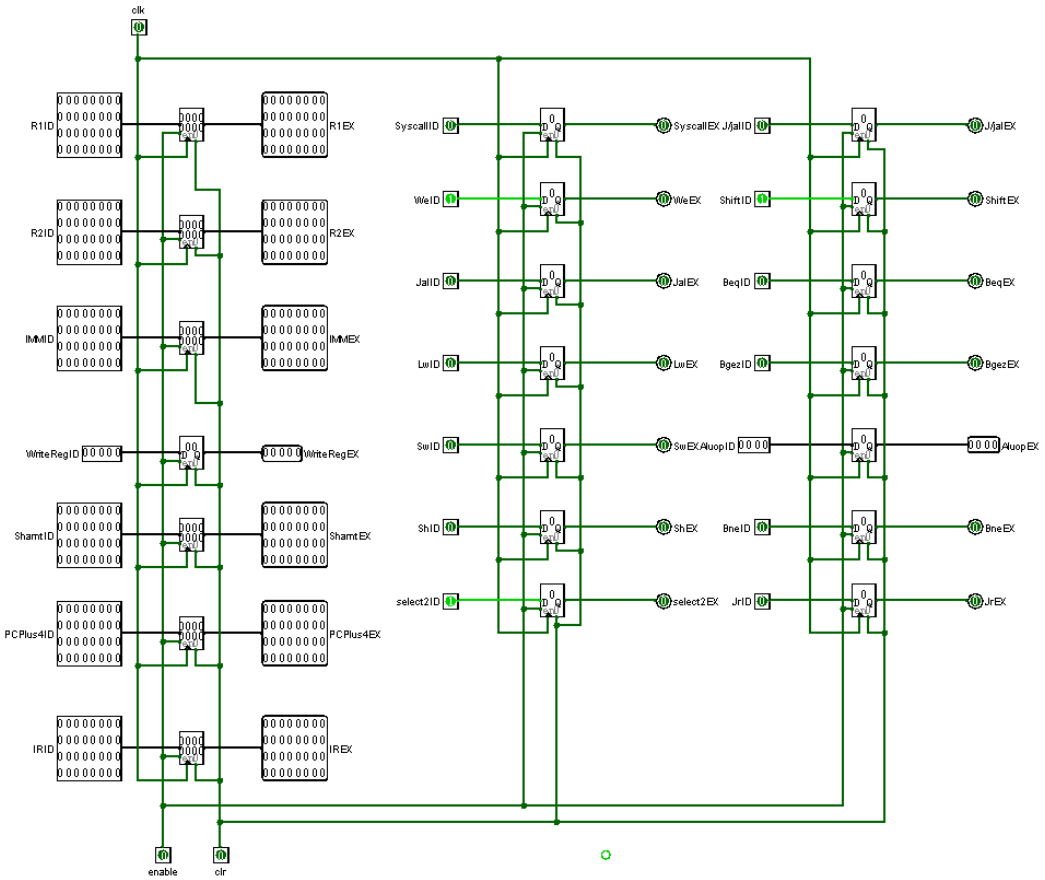

图 3.6 ID/EX 段流水寄存器内部

EX/MEM 段流水寄存器内部设置如图 3.7：

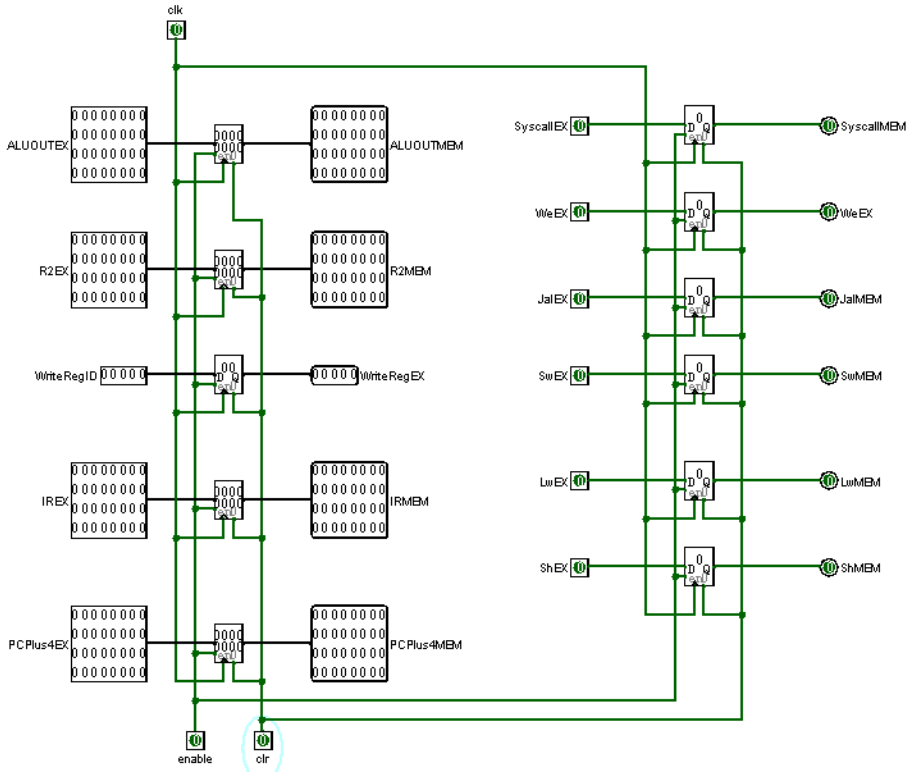

图 3.7 EX/MEM 段流水寄存器内部

MEM/WB 段流水寄存器内部设置如图 3.8：

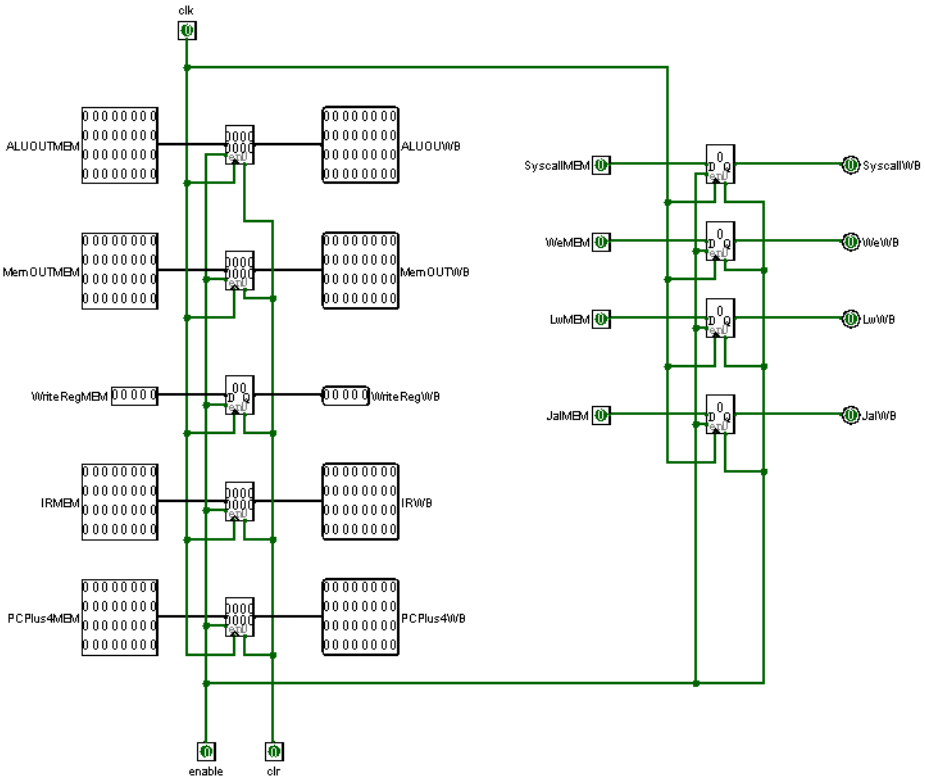

图 3.8 MEM/WB 段流水寄存器内部

② FPGA 实现：

由于 4 个流水寄存器的思想和实现方式都基本类似，只是其中传递的信号不同，为了报告的简洁性，这里只给出 IF/ID 段流水寄存器的 FPGA 实现。

```
module IF_ID(clk,enable,clr,IRIF,PCPlus4IF,IRID,PCPlus4ID);
input    wire clk,enable,clr;
input    wire [31:0] PCPlus4IF,IRIF;
output reg    [31:0] PCPlus4ID,IRID;

initial
begin
      PCPlus4ID<=0;
      IRID<=0;
end

always@(posedge clk)
begin
      if(clr)
      begin
            PCPlus4ID<=0;
            IRID<=0;
        end
        else if(enable)
        begin
            IRID<=IRIF;
            PCPlus4ID<=PCPlus4IF;
        end
end

endmodule
```

3.2.2 理想流水线实现

当做完流水接口部件—流水寄存器之后剩余的工作就是用 4 个流水寄存器将单周期数据通路划分为 5 段，实现控制信号和数据信号的锁存。然后就是要正确处理 sys 指令，前面已经做了诸多分析：sys 在 WB 段处理就不会出现数据相关。最终成型的理想流水线电路图如图 3.9。

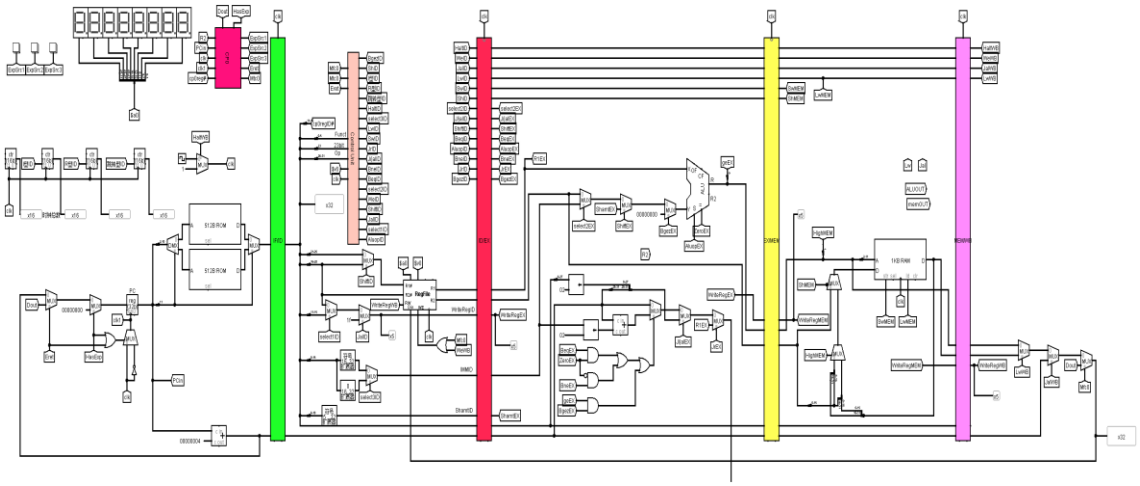

图 3.9 理想流水线顶层电路图

3.3 数据转发流水线实现

和前面的设计一样，这里在实现的时候仍然是先实现数据转发流水线，而气泡式流水线是作为数据转发流水线中存在 load-use 执行路径时间过长问题的一个修正。

1) 数据冲突检测

对于 RAW 真数据相关，采用旁路（重定向技术）可以避免流水线暂停一直等到数据写回。关于读后写数据冲突的检测这里需要用到 6 路并发比较，其中 4 路是为了检测 ID 段读寄存器（最多两个）的编号和 EX 段、MEM 段写寄存器编号是否相同，另外 2 路并发比较是如果 ID 段读寄存器编号为 0 的话，即使检测到与前面两条指令的写寄存器编号相同也不必重定向，因为 0 号寄存器内容永远为 0。

① Logisim 实现如图 3.10：

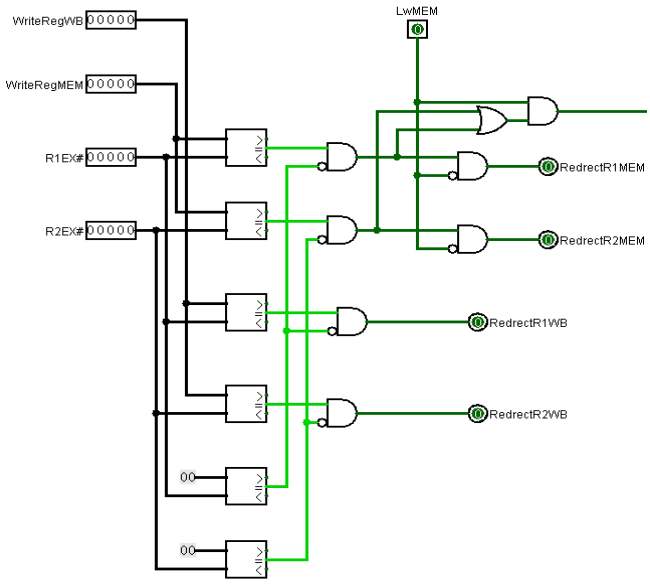

图 3.10 数据冲突检测电路图

② FPGA 实现：

```
assign signal1=(WriteRegEX==R1_EX)&(R1_EX!=0);
assign signal2=(WriteRegEX==R2_EX)&(R2_EX!=0);
assign RedrectR1MEM=signal1&(!LwMEM);
assign RedrectR2MEM=signal2&(!LwMEM);
assign RedrectR1WB=(WriteRegMEM==R1_EX)&(R1_EX!=0);
assign RedrectR2WB=(WriteRegMEM==R2_EX)&(R2_EX!=0);
```

2) 数据旁路

通过数据冲突检测机制得到的重定向信号增加新的数据旁路以实现重定向的目的，通过加入 4 个二路选择器将 MEM 段和 WB 段数据在发生数据冒险时通过旁路送到对应位置。

① Logisim 实现：

非常值得说明的一点是图中的两组多路选择器都是把选择 WB 段数据的多路选择器放在左侧，而选择 MEM 段数据的多路选择器放在右侧，他们的位置关系不是随机的，而是一种确定关系，因为当 MEM 段和 WB 段目的寄存器相同且和 EX 段源寄存器相同时（WAW），应当以 MEM 段的数据为准，故而 MEM 段数据应当具有比 WB 段更高的优先级，而这种优先级的设置只需要将 MEM 段数据放在 WB 段数据的右侧多路选择器上即可实现了，具体实现电路图如图 3.11。

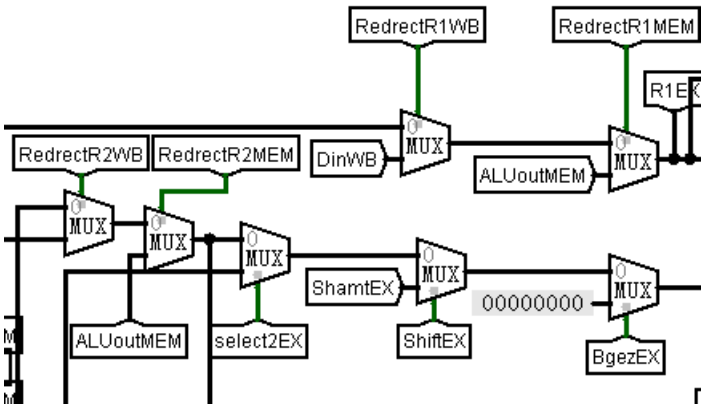

图 3.11 数据旁路电路图

② FPGA 实现：

```
assign ALUsrcx=RedrectR1MEM?(ALUoutMEM):(RedrectR1WB?(DinWB):R1EX);
assign R_R2EX=RedrectR2MEM?(ALUoutMEM):(RedrectR2WB?(DinWB):R2EX);
```

3.4 气泡式流水线实现

当实现了数据转发之后解决了真数据相关，接下来需要解决 load-use 冲突和分支冲突。前面已经把这两种冲突解释地非常清楚了，这里只展示是如何解决这两种冲突的。load-use 冲突需要在 ID 段检测，处理方式就是在 EX 段插入一个气泡，然后暂停 IF 和 ID 段的时钟，这样就能够让 LW 指令访存结束后在 WB 段进行重定向，这样就解决了 load-use 冲突中的执行路径时间太长的问题了。而对于分支指令是采用静态分支预测分支失败的方法处理，先假定分支失败，在 EX 段得到分支结果，如果分支确实失败，那么分支指令和普通指令一般无二，但是如果分支成功，则需要清除误取的指令并从分支目标处重新取值执行。

对经典五段流水线增加了重定向和插气泡等机制后就实现了全冒险处理机制的五段流水 CPU 了，全冒险处理机制的五段流水线可以正确运行由所支持 31 条指令组成的程序，全冒险处理的 CPU 顶层电路图如图 3.12。

① Logisim 实现：

通过 LwMEM 信号加上多路并发比较的结果判断当前是否出现 load-use 冲突，检测到时输出 loaduse 信号便于统计，同时输出使能信号 enable，同时作用于 PC、IF/ID 和 ID/EX，是的时钟暂停同时输出 ID/EX 段流水寄存器的清零信号插入一个气泡。当分支成功时需要同时清空 IF/ID 段和 ID/EX 段流水寄存器，因此当分支成功时

输出清零信号 clr1 和 clr2。可以注意到在图中对 LwMEM 进行了同步清零处理，实现电路图如图 3.13。

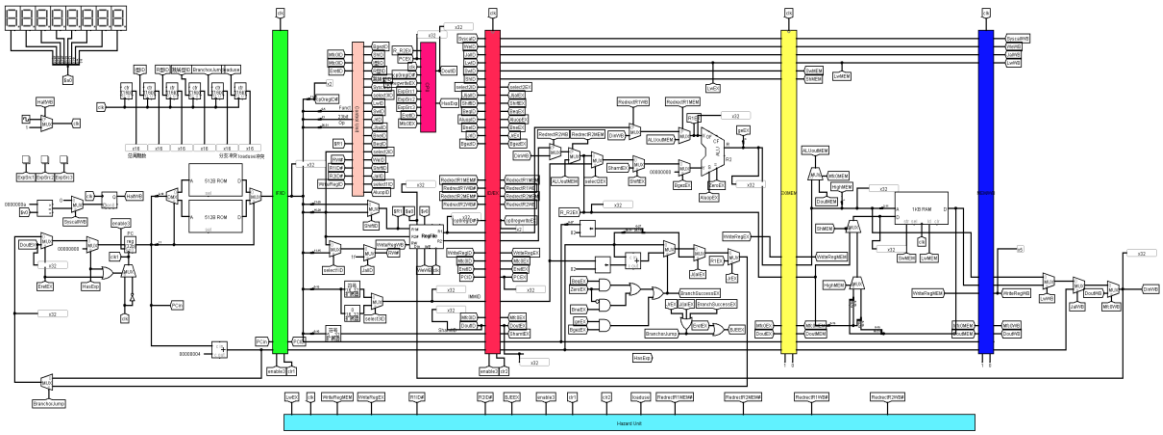

图 3.12 全冒险处理机制的 CPU 顶层电路图

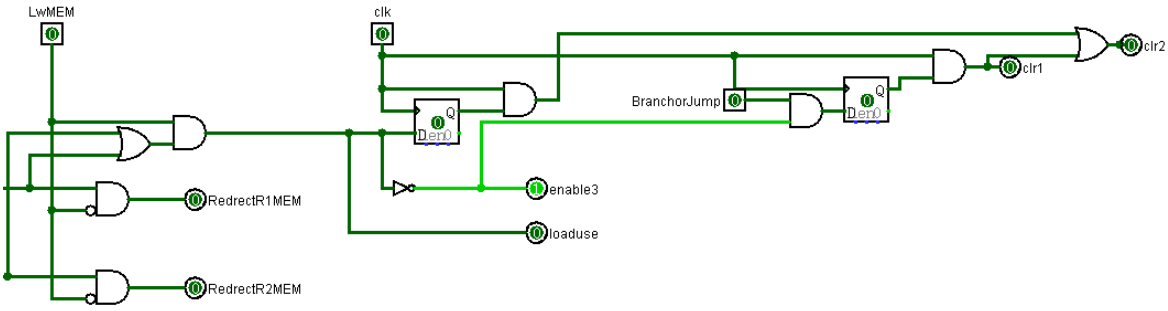

图 3.13 气泡插入及暂停流水线信号的生成

② FPGA 实现：

```
assign loaduse=LwMEM&(signal1|signal2);
assign enable3=~loaduse;

always @(negedge clk)
begin
reg1<=loaduse;
reg2<=BranchorJump&enable3;
end

assign clr1=clk&reg2;
assign clr2=clk&reg1|clr1;
```

4 实验过程与调试

4.1 测试用例和功能测试

按照实验的流程先后完成了多级中断嵌套，理想流水线，全冒险处理机制的流水线和流水中断，下面分别进行展示

4.1.1 多级中断嵌套测试

依次按下按钮 1,2,3,1，通过三级中断的优先级关系从理论上判断执行结果应该是：主程序被中断 1 打断，转而执行中断 1，而中断 2 优先级比中断 1 高故而中断 1 将会被打断转而执行中断 2，而中断 3 优先级比中断 2 高故中断 2 将会被打断转而执行中断 3，当中断 3 执行完毕后返回中断 2 执行，中断 2 执行完之后返回中断 1 执行，中断 1 执行完毕后返回主程序，而此时最后按下的中断 1 将会被响应，故再转去执行中断 1 服务子程序，执行完毕后返回主程序。

下面展示几个关键时刻 a0 的值，以及最后主程序执行完毕内存的值。

按下中断 1 时主程序 a0 值如图 4.1：

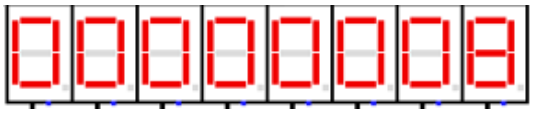

图 4.1 按下中断按钮 1 时主程序 a0 值

执行中断 1 的中断服务子程序—1 的走马灯如图 4.2：

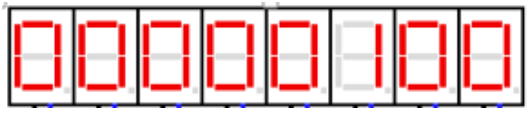

图 4.2 执行中断 1 的服务子程序

执行中断 2 的中断服务子程序—2 的走马灯如图 4.3：

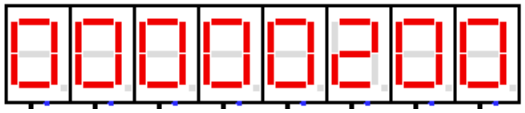

图 4.3 执行中断 2 的服务子程序

执行中断 3 的中断服务子程序—3 的走马灯如图 4.4：

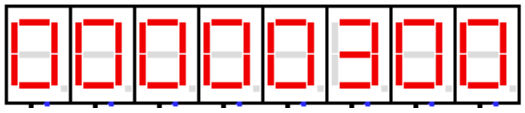

图 4.4 执行中断 3 的服务子程序

执行完中断 3 的服务子程序后返回中断 2 的服务子程序执行如图 4.5：

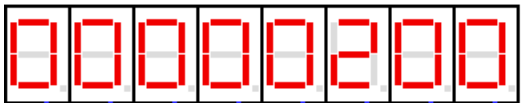

图 4.5 返回中断 2 的服务子程序

执行完中断 2 的服务子程序后返回中断 1 的服务子程序执行如图 4.6：

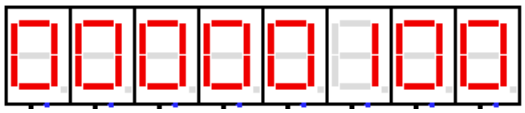

图 4.6 返回中断 1 的服务子程序

执行完中断 1 的服务子程序后返回主程序执行如图 4.7：

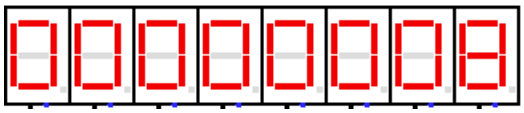

图 4.7 返回主程序

执行主程序时响应最后一次按下的中断 1 如图 4.8：

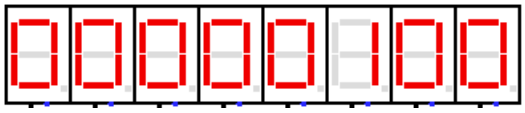

图 4.8 再次响应中断 1

执行完中断 1 的服务子程序后返回主程序执行如图 4.9：

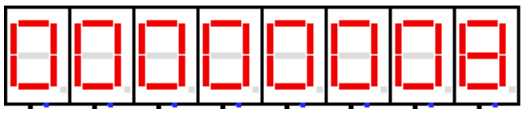

图 4.9 返回主程序

执行完主程序后 a0 寄存器如图 4.10 和内存中数据排序的结果如图 4.11：

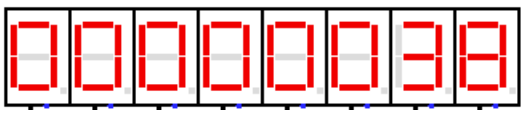

图 4.10 主程序执行完毕的 a0

```
000  0000000e 0000000d 0000000c 0000000b  0000000a 00000009 00000008 00000007
008  00000006 00000005 00000004 00000003  00000002 00000001 00000000 ffffffff
```

图 4.11 主程序执行完毕内存排序情况

4.1.2 理想流水线测试

这里使用老师提供的理想流水线测试程序，其中共 17 条指令，所以指令执行需要的总时钟周期数为 21，且数据存储器中应该依次存储了 0，1，2，3 四个数据。

理想流水线执行完毕后，周期数统计如图 4.12 与内存中的情况如图 4.13。

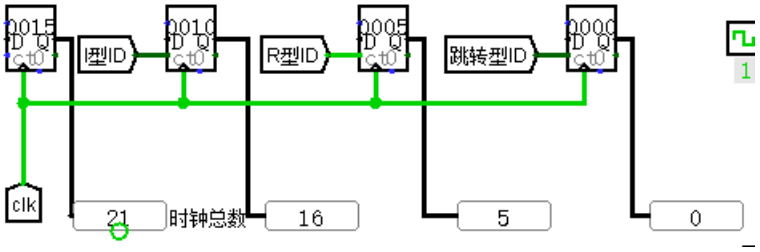

图 4.12 理想流水线测试周期数统计

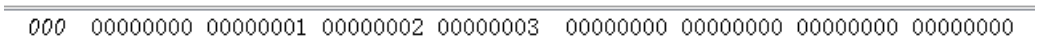

图 4.13 理想流水线测试内存情况

4.1.3 全冒险处理流水线测试

解决了数据冒险，结果冒险和分支冲突后我们的 CPU 就可以正确运行 benchmark 测试程序，测试结果如下，周期数、load-use 冲突、分支冲突数的统计如图 4.14，存储器中排序结果如图 4.15，统计数与内存排序结果均符合预期。

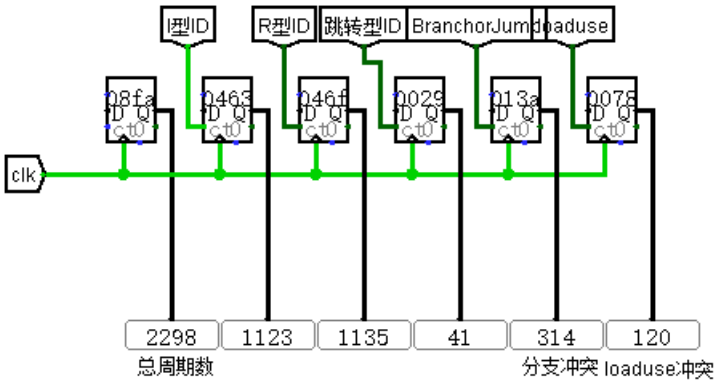

图 4.14 benchmark 各数据统计数

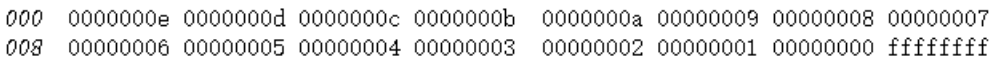

图 4.15 内存排序结果

4.1.4 流水中断 FPGA 测试

流水中断已当面进行演示，由于写次报告时已无开发板，故这里仅截取 a0 寄存器和内存排序结果最终值，FPGA 最终运行结果如图 4.16。

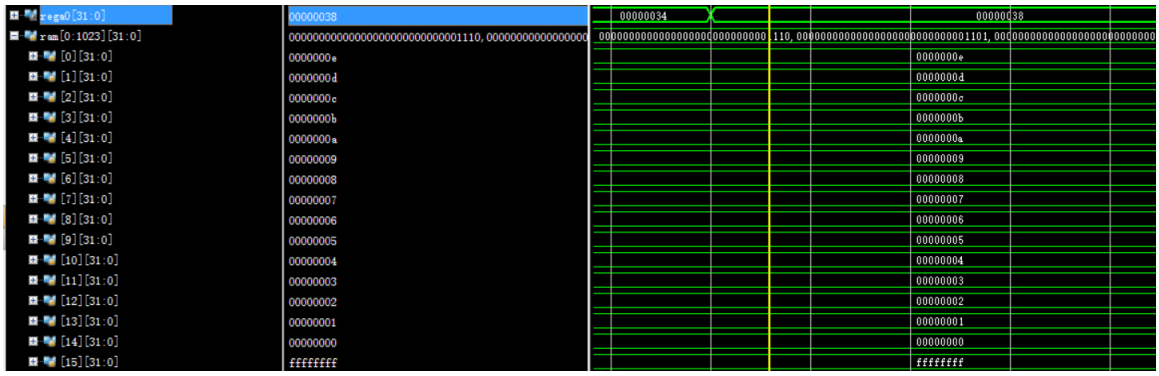

图 4.16 FPGA 流水中断最终执行结果

4.2 性能分析

采用单周期方式下执行 benchmark 时钟周期数为：1546，而采用静态分支预测分支失败的五段流水方式 benchmark 时钟周期数为：2298。假定单周期方式时钟周期为五段流水方式的 5 倍，那么加速比：$S = \frac{1546 \times 5\Delta t}{2298 \times \Delta t} \approx 3.36$。

4.3 主要故障与调试

4.3.1 中断故障

中断：当发生多级中断时，不能到达预期走马灯效果。

故障现象：中断程序编写出现问题。

原因分析：在中断的实现上硬件上并没有出现太多的问题，需要构建支持新指令的数据通路，在此前已有大量经验，主要问题出现在软件的编写上，出现了一些小问题，但都很快得以解决。

解决方案：依据走马灯的异常现象，判断出对应寄存器或堆栈中值不正确，再返回中断程序中修改对应指令。应该考虑栈帧指针的赋值在中断嵌套时的合理性，比如每次进入中断均设置 addi $gp,$0,0x800，然后单独使用 gp 访问堆栈，结果当出现嵌套时自然会导致无法正常恢复现场，原因是后续中断保存的现场由于 gp 的错误赋值覆盖了之前中断的现场，解决方案是下一次中断时的 gp 继承上一次中断时的 gp，而不是赋一个固定的值。导致现场恢复不正确还有一个原因：恢复现场时寄存器的先后

顺序应该是保存现场时的逆序，这与栈 FILO 的特性是相关的。还有开中断与返回断点的矛盾性，若先返回断点则不会执行后续开中断操作，若先开中断则会造成现场恢复不完整（pc 无法正常恢复），解决方法是采取加入一个寄存器，延期一个时钟才将将中断使能寄存器置位，即实现了返回断点和开中断在硬件级别达到并行，实现电路图如图 4.17，而这对传统串行编程时无法实现的。还有就是对应这样的中断嵌套问题，当要使用一个寄存器的值的时候，应该先置位，不要觉得没有使用该寄存器就默认寄存器初值为 0，这样做是不太合适的。

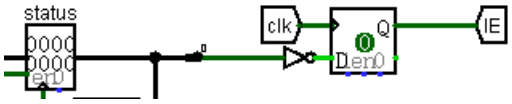

图 4.17 延迟开中断

4.3.2 load-use 冲突未处理好

开始检查 load-use 冲突是在 EX 段进行的，应该提前检测。

故障现象：当 benchmark 执行到，排序段的时候程序出现与预期不一致的情况。

原因分析：benchmark 中存在两条 LW 指令的目的寄存器编号分别与后一条指令两个源寄存器编号相同的情况，如果在 EX 段才检测 load-use 冲突，则因为插入一个气泡，将第一天 LW“挤”出流水线，故 EX 段时钟有效时，其需要的数据已经正确写回，所以无法通过定向技术获得了。

解决方案：把 load-use 检测提前到 ID 段，那么当检测到 ID 段与 EX 和 MEM 段均发生数据相关且存在 load-use 冲突时，首先在 EX 段插入一个气泡，且停下 ID 段及之前段的时钟，此时第一条 LW 指令已执行到 WB 段。由于将寄存器文件改造成下降沿写入，故而当上升沿到来时，处理 ID 段指令可以从寄存器文件中获得正确的相应寄存器的值。

4.3.3 实验中其他故障

故障现象：实验中当然还存在一些其他的故障，但大部分都由于是很小的问题，随手就解决了。

原因分析：比如最容易出错的 Logisim 平台到 FPGA 平台的转换时，会集中出现较多的错误，比如因为前后因为打错导致相同变量名字不相同，或者手误导致控制器中某个信号译码写错了等，这些错误在编程时由于粗心，犯的概率并不低。

解决方案：我通过一步一步看 FPGA 仿真图，与 Logisim 平台中执行结果相比较，一旦出现不相同就去找为什么会导致不相同，一般来说是某个信号出错了，那么就去看生成该信号的表达式是否正确，一步一步跟着信号图走下去，我尽力处理了所有的 warnning 之后，我根据仿真图改正了三个逻辑错之后我就可以在开发板上正确执行 benchmark 测试程序了。调试的时候，一直在结合着 vivado 仿真图、Logisim、MARS 模拟器同步调试，即使一些出现在控制器中译码时的错误，也很快地得以纠正。

4.4 实验进度

表 4.1 课程设计进度表

时间	进度
第一天	复习组成原理 CPU 相关理论知识，阅读课设任务书，阅读 MIPS 指令手册，修改单周期电路使之支持新增的 3 条指令。
第二天	修改电路使之支持新增的四条扩展指令并完成测试，思考多级中断实现方法。
第三天	实现 CP0 部分功能，单级中断的实现上仍都存在问题。
第四天	已实现单级中断，多级中断的嵌套上存在一些问题，研究经典五段流水线。
第五天	实现多级中断的嵌套，但存在小 bug，正在设计流水四个寄存器。
第六天	解决中断嵌套存在的问题，完全实现了多级中断的嵌套，完成理想流水线，在处理冒险。
第七天	完成重定向，分支跳转存在问题。
第八天	完成全冒险处理的流水 cpu，正在实现 FPGA 流水线。
第九天	完成 FPGA 五段流水 CPU，正确执行 benchmark。
第十天	完成流水中断，成功在 FPGA 上通过测试。

5 设计总结与心得

5.1 课设总结

本次课程设计实现了支持三级中断嵌套的全冒险处理机制的五段流水 CPU，可以正常运行由支持的 31 条 MIPS 指令构成的程序，同时由于采用了流水线，CPU 的频率得到了显著的提升，作了如下几点工作：

1） 在单周期数据通路上修改了电路使之支持四条扩展指令，有了之前单周期数据通过构建的经验，这个过程并不难，但是对于 SH 存储半子指令的实现其实有些复杂，我们的数据存储器一次访问只能存取 32 位，而 SH 要求存储一个字节，直接存储势必要对数据存储器其他字节内容进行破坏，所以可以采用先读后写的方法，先将整个字节读出，然后与写入内容拼接后再写入数据存储器。

2） 实现了多级中断嵌套，外界有三个中断源可以进行中断请求，根据其固定的优先级，CP0 会正确响应某一中断请求序列。多级中断卡了我比较久，我建议一定要先实现单级中断，而不要空想多级中断如何屏蔽，如何恢复（我就是这样的反面例子）。单级中断的实现并不太难，主要是中断跳转和保存、恢复现场，在实现了单级中断后，会主动去想在一个中断已经按下之后如何能够使高优先级中断可以打断当前中断，从而设计出实现中断屏蔽的中断仲裁电路，中断请求寄存器的清零最好采用同步清零，否则在 FPGA 实现时会存在 always 触发条件不好写的问题。

3） 首先在 Logisim 平台实现了全冒险处理机制的 CPU，而后成功转移至 FPGA 平台，测试通过。理想流水线对指令中存在的冲突无法处理，会导致程序运行出现错误，通过重定向可以解决数据冒险，而以插气泡的方式解决 load-use 冲突和分支冲突，结构冲突我们采用哈佛结构解决访存指令和取值均需用到存储器的内存争用，采用下降沿写寄存器文件和上升沿读寄存器文件的方法解决了 ID 段读寄存器文件和 WB 段写寄存器文件的冲突，但是结构冲突不是我们考虑的重点。FPGA 就是把电路图转化成 verilog 语言实现，本质上没有什么。

5.2 课设心得

组原课程设计给我带来更多的可能还是很多思想上的收获：只要你努力，很多比较困难的事情就会变得不是那么难了。当遇到我们没有做过的事情的时候，分治法真的是一个有效的方法：可以先实现单级中断，在考虑我还需要什么才能实现多级中断，不断添加功能直至完全实现最初那个认为比较复杂的功能。虽然整个实验中出现了比较多的 bug，但是没有一次他们让我变得毛躁，而是一直耐心地去想要解决他们，我想这也是大学的生活给我带来的性格上的成长。

我自己感觉上个季度组成原理理论课程学习地还是很认真的，对于课程也有很多自己的理解，去年即使是相对较难的实验三，刚开始的时候我就有很清晰的思路，所以做起来也非常的快。然而组成原理课程设计最初让我再一次质疑：我是不是真的比别人差。修改电路支持 4 条扩展指令在有了之前单周期的基础上是非常简单的，然而之后的多级中断的嵌套我开始就想一下子直接实现，导致思考地非常多，而实际去做却非常少，我一心只想想之前的实验一样有了一个大体的思路之后开始做，这也导致我长达几天的无作为，看到班上别的同学都纷纷有了不小进展，我开始怀疑自己。直到第一周周五我还是没有实质性的进展，当天晚上我们开了一个班会，我们班主任在上面表扬了我，我当时下定决心：我一定要把多级中断的嵌套做出来。回去之后改变了思路，先实现了单级中断，单级中断的实现其实不难，主要是现场的保护。完成单级中断之后，我开始思考，为了实现多级中断还需要那些功能，我后面逐步加入了中断屏蔽，中断仲裁等功能，加入这些部件完全是因为当前就是因为没有实现这些，所以才无法实现中断嵌套。说实话，当实现了单级中断之后后面多级中断的思路是非常流畅的。

完成了多级中断之后其实后面真的是越来越简单，但是由于我在中断上浪费了太多时间，所以即使两个周末我全部加班最后也才只做到流水中断 FPGA 实现。整个过程非常累，但是我觉得收获也不小，特别是掌握了利用 vivado、Logsim、MARS 三个平台同时进行调试，很多错误跟着仿真图其实非常方便的就可以调试出来。

整个课设我基本上没有向别人请教，一部分原因是我想自己解决遇到的问题，我觉得收获更大，而且我一直都是这么做的。所以班上同学前期在中断上的进度都比我快，虽然我最后完全自己解决了所有问题，但是耗费的时间是巨大的，在之后的实验中我可能会更多地与同学交流。

我觉得我们学校硬件系列课程开设地比较多，总共有：数字逻辑、计算机组成原理、接口技术、计算机系统结构、嵌入式系统，我都非常喜欢这些课程，但是现在存在的一个问题是每个课程的实验都是相对独立的，如果说最后能把这些课程进行一个综合性的设计，我相信对计算机的硬件组成应该会加深很多理解。

最后真的感谢所有为组成原理课程设计耗费了大量心血的老师，我觉得组成原理课程组真的是非常有上进心，每年都会更新上课的 ppt、课程实验和课程设计，这在别的学科中是我没有见过的，希望学校有相应的制度：奖励那些大力推进课程改革的老师，真的可以尝试惩罚那些上课就是一字不落照念 PPT 的人，希望再过几年华科的学生可以接受到更好的教育，主动学习的动力也会更大了。

参考文献

[1] DAVID A.PATTERSON(美).计算机组成与设计硬件/软件接口(原书第 4 版).北京：机械工业出版社.

[2] David Money Harris(美).数字设计和计算机体系结构（第二版）. 机械工业出版社

[3] 秦磊华，吴非，莫正坤.计算机组成原理. 北京：清华大学出版社，2011 年.

[4] 袁春风编著. 计算机组成与系统结构. 北京：清华大学出版社，2011 年.

[5] 张晨曦，王志英. 计算机系统结构. 高等教育出版社，2008 年.

一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字: