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 01 班
年　　级:	2018
学　　生:	朱海龙　　屈湘钧　　李颀琳
学　　号:	20183044　　20186471　　20185653
完 成 时 间:	2021 年　01 月　10 日
成　　绩:	100
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

朱海龙、屈湘钧、李颀琳

1 设计简介

我们小组设计了一个可以执行 mips32 的 57 条指令的 cpu，包含移位指令、逻辑运算指令、算术运算指令、数据移动指令、分支跳转指令、访存指令、内陷指令和特权指令。并参考实验指导书搭建了 soc_sram 和 soc_axi 环境，同时在 cpu 中加入了 cache 和分支预测，使得 cpu 可以正确且有效率的执行指令。

1.1 小组分工说明

- 朱海龙：负责添加移位、数据移动和分支跳转指令扩展，负责 sram、axi 接口的添加、cache 添加和所有接口添加后的 debug，以及性能测试上板 debug。
- 屈湘钧：负责逻辑运算、算术运算、访存、内陷和特权指令扩展，负责接口添加后的 debug，最后进行上板测试，此外有设计分支预测（包含局部、全局与竞争）。
- 李颀琳: **加入前:** 负责添加移位、数据移动、逻辑运算、算术运算、访存和分支跳转指令扩展，完成单元测试；**加入后:** 参与 sram 接口添加与 debug、cache 添加，编写分支预测和二路组 cache 代码。

2 设计方案

2.1 总体设计思路

本次硬件综合设计的项目是支持 Mips 精简汇编指令集的简单五级流水线 cpu, 需要实现支持 52 条基础 mips 非浮点指令和内陷指令 break,syscall 和 eret,mfc0,mtc0 特权指令的共 57 条指令的 cpu, 并在 soc 环境 (sram 和 axi) 中测试 cpu 功能和进行性能上板 (n4ddr 开发板) 测试。

我们小组的设计基于计算机组成原理实验 4 的五级流水线, 并依次添加前 52 条基本指令。修改数据通路, 处理流水线中的冒险问题, 对于可以前推的数据, 进行必要的前推, 对于无法前推的, 让流水线暂停, 等待数据满足, 使得流水线中的基本指令可以正常执行. 在这个阶段, 可以进行最基本的 independent 测试, 查看基础的六类指令是否执行正确。

对于后面的特权指令, 这些指令主要用于处理异常和中断, 我们小组实现了实验要求中的精确异常, 使得处理器在遇到异常时, 先进行标记, 等到第四阶段访存时再进行处理, 处理过程需要用到 cp0 协处理器和异常异常译码部件 exceptiondec, 异常类型的指令需要刷新掉后续的指令, 并跳转到 bfc00380 的特殊地址,eret 是一个很特殊的指令, 它的作用是跳转回正常的指令, 即 epc 寄存器的值, 特别注意, 在实现一般的跳转指令时, 是默认有延迟槽的, 但 eret 没有延迟槽. 此时,57 条指令已经全部成功加入通路, 由于这些特殊指令已经设计到了地址映射, 在这一阶段, 需要接上 soc 环境, 使用实验提供的 soc_sram_func 包, 加载功能测试 coe 文件, 并使用 trace 文件进行比对, 判断 57 条指令的实现是否正确。

实现了以上的基本指令时, 已经可以接 axi 环境, 进行性能测试了, 由于 axi 环境具有延时, 需要对原本的通路进行一些暂停机制的修改, 让通路在 cpu 访问内存时完全暂停, 以及对异常处理时略作修改, 先进行功能测试确认通路的正确性, 再进行性能测试查看 cpu 性能与龙芯开源的 132cpu 的比较. 并接上 cache 以提升性能。

2.2 指令添加与 soc 接口设计流程

2.2.1 逻辑运算指令

在计组 4 的基础上,这里将 and 和 or 指令添加到 xor、nor 和立即数的逻辑运算 andi,xori,lui,ori 几个指令。

首先对于 xor 和 nor 指令,其数据通路与 and,or 没有什么区别,直接在 main_dec 和 alu_dec 模块中添加处理的控制信号,然后在 alu 中添加其相应的计算项。

对于立即数的逻辑运算指令,其通路图由于计组四有实现运算立即数指令,其通路也可以服用,但是需要注意的是逻辑运算的立即数是无符号扩展,为了减小无符号扩展模块增加数据通路复杂度,直接复用有符号扩展的模块,在 alu 中,判断是逻辑立即数,则将高十六位直接置 0 即可。其他的直接在 main_dec 和 alu_dec 模块中添加处理的控制信号。

此外,注意 lui 是将立即数低十六位放置在结果的高十六位,然后结果的低十六位全为 0。

以上设计的核心代码如下。

```
// main_dec
case(op)
    `EXE_NOP: case(funct)
        //logic inst
        `EXE_AND, `EXE_OR, `EXE_XOR, `EXE_NOR: main_signal <= 6'b110000; // R-type
     endcase
    `EXE_ANDI ,`EXE_XORI, `EXE_LUI, `EXE_ORI: main_signal <= 6'b101000; // Immediate
endcase

//alu_dec
case (op)
    // R-type
    `EXE_NOP : case (funct)
        //logic inst
        `EXE_AND :alucontrol <= `EXE_AND_OP ;
        `EXE_OR :alucontrol <= `EXE_OR_OP ;
```

```
        `EXE_XOR :alucontrol <= `EXE_XOR_OP ;
        `EXE_NOR :alucontrol <= `EXE_NOR_OP ;
    endcase
    `EXE_ANDI :alucontrol <= `EXE_ANDI_OP ;
    `EXE_XORI :alucontrol <= `EXE_XORI_OP ;
    `EXE_LUI :alucontrol <= `EXE_LUI_OP ;
    `EXE_ORI :alucontrol <= `EXE_ORI_OP ;
endcase

// alu
case (alucontrol)
    //logic op
    `EXE_AND_OP :ans <= num1 & num2 ;
    `EXE_OR_OP :ans <= num1 | num2 ;
    `EXE_XOR_OP :ans <= num1 ^ num2 ;
    `EXE_NOR_OP :ans <= ~(num1 | num2) ;
    //TODO 由于传进来的immediate是有符号扩展，这里为了节省一个zero_extend,直接在alu中修改高16位
    `EXE_ANDI_OP :ans <= num1 & { {16{1'b0}} , num2[15:0]} ;
    `EXE_XORI_OP :ans <= num1 ^ { {16{1'b0}} , num2[15:0]} ;
    `EXE_LUI_OP :ans <= {num2[15:0] , {16{1'b0}} } ;
    `EXE_ORI_OP :ans <= num1 | { {16{1'b0}} , num2[15:0]} ;
```

2.2.2 移位指令

MIPS 指令集架构中定义的移位操作指令有 6 条：ssl、sllv、sra、srav、srl、srlv。这六条指令都是 R 型指令，因此需要指令中的第 0-5 位的功能码进一步判断是哪一种指令。移位操作的目的都是将当前输入的数据向左或者右进行移动，但是移动的方式（逻辑移、算术移）与具体移动的位数都需要通过指令类型决定。

这些指令的译码过程如下，首先要通过 alu_dec 解码出当前执行指令的类型，接着找出当前要读取寄存器的情况，将需要被读的寄存器的值传入 ALU 中进行进一步运算，默认 Regfile 模块的读端口 2 就是 rt 寄存器的值。接下来，通过译码的结果找到所对应指令需要进行的运算，ALU 会根据 alucontrol 的值决定当前使用哪个值作为偏移量，以及决定移动方式，最后将计算的结果输出，通过触发器传到第五阶段写入寄存器内。

sll 指令和 srl 指令是逻辑移动，其作用是将地址为 rt 的通用寄存器中的值移动 sa 位，其中 sa 来自指令的 10~6 位。逻辑移动的方式是直接把所给数据的二进制数向左或者右进行移动，空余的部分使用 0 填充。

sllv 指令和 srlv 指令也是逻辑移动，但是决定其移动的数值是来自当前指令的 rs 寄存器中的值。如果寄存器内的值过大，可能会让移位超过 32，出现归零现象，因此只用了寄存器内值的低 5 位，就不会存在归零问题。

sra 指令和 srav 指令都是算术右移。算术移动与逻辑移动的方式不一样，它是要将源操作数数据（即 rt 寄存器内的值）先向右边移动一指定位（sra 指令移动位数由 sa 指定、srav 指令由 rs 寄存器内值决定）空余的位置需由 rt 的最高位进行填充，这样就保证了寄存器内值的算术意义。

在进行移位操作的时候，首先在 decode 阶段解析出指令所需要做的操作类型，以及得到操作数，通过触发器传到 execute 阶段。在 execute 阶段，根据操作数以及指令类型，对传入的数据（rd 寄存器的值）进行移位操作。如果是采用 sa 部分作为偏移量，则直接对寄存器内的值偏移对应的量；如果是采用寄存器 rs 的值作为偏移量，就取出其值对 rd 寄存器的值做偏移操作。完成后，等到 write back 阶段写寄存器，将移位后的结果写入到指定寄存器号中。整体上，在 alu_dec 和 main_dec 中添加了相应的解码操作，并在 alu 中添加了移位操作相关的计算功能。

2.2.3 数据移动指令

这里的数据移动指令是指对 hilo 寄存器的移动操作，这两个寄存器主要用于后面运算指令中的乘除法运算，同时支持将寄存器的值存进 hilo 和将 hilo 的值存进普通寄存器，即读和写操作。需要在控制器模块里面添加 main_dec 中对 hilo_move 指令的译码部分，对于 mfhi 和 mflo 指令，需要在控制器里将 regwrite 和 regdst 信号置为 1，其余信号都是 0，mthi 和 mtlo 的信号都是 0。由于我们小组使用了课程发的 defines.vh 中的宏定义，所以可以直接利用 alucontrol 的信号指示 alu 操作，让 E 阶段可以进行 hilo 的移动操作。hilo 部分在计组 4 的通路中没有，为了避免数据冒险，我们小组最终确定将 hilo 寄存器放置在第三阶段运行 E 阶段，使得读 hilo 和写 hilo 都是在一阶段，恰好可以解决 hilo 的数据冒险问题，不会存在读的值还没写入的情况。

hilo 指令的执行流程依次是：F 阶段取出指令；D 阶段译码，这一阶段需要获得读或写 hilo 的寄存器号，如果是 mthi 或 mtlo 指令，还需要读出 rd 寄存器的值；E 阶段执行，如果是 mthi 和 mtlo，需要特别注意，rd 的值可能还未写入，这里需要的用的值是经过前推选择的值，并将值写入 hilo，如果是 mfhi 和 mtlo 指令，则从 hilo 中读出值；M 阶段不需要操作；W 阶段如果是 mfhi 和 mflo 指令则将读出的 hilo 值写入寄存器堆中。

注意，由于 hilo 的读写都在 E 阶段，但是我们小组的异常处理是在 M 阶段，需要让 M 阶段的异常可以阻止 E 阶段，所以需要 except 信号也放进 hilo 的处理过程。

```

always@(clk) begin
    if (rst) begin hilo <= {64{1'b0}}; end // reset
    else case(exceptionoccur)
        1'b1: begin hilo <= hilo; end // 出现异常 hilo不能被修改
        1'b0: begin
        if(div_res_valid == 1'b1) begin hilo <= hilo_out_div;end // 乘法
        else if(mul_valid == 1'b1) begin hilo <= hilo_out_mul;end // 除法
        else if(alucontrol == `EXE_MTHI_OP) begin hilo <= {num1,hilo[31:0]}; end // 写hi指令
        else if(alucontrol == `EXE_MTLO_OP) begin hilo <= {hilo[63:32],num1}; end // 写lo指令
        else begin hilo <= hilo; end // 没有写操作,保持原状态
        end
    endcase
end
```

2.2.4 算术运算指令

算术指令可以被分为二类:简单算术操作指令与除法指令,这些指令内又分有符号运算、无符号运算两种。根据操作数是否有立即数,又分 I 型指令和 R 型指令。
首先,对于 add、addu、sub、subu、slt、sltu 六条指令,他们都是 R 型指令。当功能码对应到不同的指令的时候,ALU 需要根据 alucontrol 的值选择进行不同的算术运算。如果是无符号运算类型,直接进行加和减。但是如果有符号运算还需要考虑溢出的情况。如果加减法溢出了,则需要将 overflow 位设置为 1,并且输出为 0. 对于比较运算也需要区分有无符号的情况,关键实现代码如下:

```
// ALU
‘EXE_SLTI_OP :ans <= $signed(num1) < $signed(num2) ;
‘EXE_SLTIU_OP :ans <= num1 < num2 ;
```

对于 I 型指令的算术运算指令,输入的值是来自指令中的 15 到 0 位,需要对其进行扩展,立即数仍然是符号扩展至 32 位后再参与运算。

对于乘法和除法运算,我们组参考了 github 上的乘法器除法器版本。乘法器的使用,首先通过当前的 opcode 判断是有符号还是无符号乘法,再决定是否将乘法器模块中的符号运算信号设置成 1. 乘法器是组合逻辑不会引起流水线的暂停,但是除法器需要停顿直到除法的运算结果被算出,因此需要一个 div_stall 信号来表示当前数据通路是否因为除法运算而被暂停。

当除法开始运算的时候,F 阶段 D 阶段不再读入新的指令被暂停,而 E 阶段持续做除法运算直到算出结果,发出一个 div_valid 信号表示运算结果准备好,于是将结果也传出,下一个阶段流水线开始继续正常读取。在 hazard 模块中,我们将 E 阶段由于做除法而产生的 div_stall 信号传入,并控制 F 阶段、E 阶段的停顿。

在乘除法的计算结果完成后,需要把运算结果存入到 hilo 寄存器中,其中乘法是直接将 64 位结果放入到 hilo 寄存器中,除法是 hilo 寄存器的高 32 位放余数,低 32 位放商。由于我们的数据通路设计将 hilo 寄存器放入到了 alu 中,所以一旦计算完结果就立刻将计算结果写入 hilo。

处理非除法的算术运算指令的流程是:首先在 Fetch 阶段取出指令,decode 阶段译码,然后将两个算术运算的操作数传入到 alu 中,在 execute 阶段,ALU 根据 alucontrol 执行对应的加减法操作,如果是有符号计算,判定溢出位,如果溢出,输出结果为 0。如果是乘法,需要将计算结果写入 hilo 寄存器。其余指令在 write back 阶段写回到指定的寄存器中。

对于除法指令的运算流程,前面两个周期都和其他指令一样,但是在 execute 阶段,首先要通过 alucontrol 的值判断当前是否是有符号除法,将两个操作数(除数和被除数)以及有符号除法控制位传给除法器模块,除法器运算开始。在除法运算的时候,前两个阶段

(fetch 和 decode)都被暂停,execute 阶段等待除法器模块计算完结果并返回 res_ready 信号。当收到该信号以后,流水线不再暂停,将计算结果写入到 hilo 寄存器中,接着进行下一个阶段。

2.2.5 分支跳转指令

根据我们小组的通路,分支跳转指令可以分为三类,branch 类,jump 类,al 类。

首先描述 branch 类,包含 beq、bgtz、blez、bne、bltz 和 bgez 指令,由于我们小组的通路有分支预测模块,所以 branch 类执行流程如下:F 阶段取指;D 阶段根据分支预测结果判断是否跳转;E 阶段判断 D 阶段的预测是否正确;M 阶段和 W 阶段没有特殊操作。我们小组的分支预测模块只会预测是否发生跳转,不预测跳转的具体地址,关于分支预测模块的介绍将在后面进行,这里进行通路的分析。如果预测结果完全正确,那么就不需要进行特殊的操作,这种情况下是比较理想的,但是如果预测错误,我们的通路则需要处理这种错误的结果,具体有两种错误:如果预测跳,且预测错误,则需要将这一条指令 flush 掉,即 flushD,并且将正确的 PC 值 PCadd4E+4 传入 PC,这里 +4 是因为延迟槽的指令必须执行,所以需要再 +4;如果预测不跳且预测错误,还是需要将 flushD,并将正确的 PC 值 PCbranchE 传入 PC。

jump 类指令是必跳转的指令,包含 j 和 jr 指令,执行流程如下:F 阶段取指;D 阶段跳转,普通 jump 跳转的地址是立即数左移两位加上原 pc+4 的前三位组成的地址,特别注意,jr 指令需要跳转的地址存储在寄存器 rs 中,这可能带来数据冒险 RAW,需要读取的 rs 是经过数据前推的 rs。

al 类指令是最特殊的,包含 jal、jalr、bltzal 和 bgezal 指令,这些指令的跳转部分和前面一样,特殊之处在于不管这些指令是否发生跳转,必须将 PC+8 的值写入特定寄存器 31 号,其中 jalr 需要将 PC+8 写入寄存器 rd 中。

综上所述,需要在 D 阶段的通路增加数据前推的选择器,来为 jr 指令获取正确的跳转地址,并添加分支预测模块 branch_predict 模块,进行分支预测;在 E 阶段的通路添加 branch_judge 组件 (我们小组为了方便,直接写在了 alu 里面) 判断 branch 是否正确,并添加一个选择器,选择普通的写寄存器号还是单写 31 号寄存器 (al 类指令)。关于控制器模块,由于计组 4 已经有很多信号了,这里只需要添加一个 jumpr 信号,来代表 jr 指令,并添加信号 al_regdst 信号,代表是否是 al 类指令,是否需要写 31 号寄存器。

```

// 特殊的控制信号
assign jump = ((op == `EXE_J) || (op == `EXE_JAL)) ? 1 : 0;
assign jumpr = ((op == `EXE_NOP) && ((funct == `EXE_JR) || (funct == `EXE_JALR))) ? 1 : 0;

assign al_regdst = (((op == `EXE_REGIMM_INST) && (rt == `EXE_BLTZAL || rt == `EXE_BGEZAL)) // 两条bzal指令
                    || (op == `EXE_JAL)) ? 1 : 0; // jal指令

// PC前面的五个选择器结构, 可以完全处理jump和jumpr和分支(包含分支预测)
mux2 #(32) before_pc_which_wrong(pc_temp1,pc_branchE,pc_add4E+4, predictE);
```

```
mux2 #(32) before_pc_wrong(pc_temp2,pc_add4F,pc_branchD, branchD & predictD);
mux2 #(32) before_pc_predict(pc_temp3,pc_temp2,pc_temp1,predict_wrong & branchE);

// 前面三条选择器都是在处理分支预测，下面两条在处理jump和jumpr
mux2 #(32) before_pc_jump(pc_temp4,pc_temp3,{pc_add4D[31:28],instrD[25:0],2'b00},jumpD);
mux2 #(32) before_pc_jumpr(pc_temp5,pc_temp4,eq1,jumprD);

// 处理al型指令的选择，write_alE就表示需要选择pc_al_dst即31号寄存器,否则选择原本的WriteRegTemp寄存器
mux2 #(5) mux_for_al(WriteRegE,WriteRegTemp,pc_al_dst,write_alE);

// 在alu里面需要添加al类指令的值
//J type
`EXE_JAL_OP :ans <= pc_add4E + 32'b100 ; // 需要写pc+8到31号ra寄存器
`EXE_JALR_OP :ans <= pc_add4E + 32'b100 ; // 需要写pc+8到31号rd寄存器

//b type
`EXE_BLTZAL_OP :ans <= pc_add4E + 32'b100 ; // 需要写pc+8到31号ra寄存器
`EXE_BGEZAL_OP :ans <= pc_add4E + 32'b100 ; // 需要写pc+8到31号ra寄存器
```

这里在 pc 前面的选择器有点多，将在后面的模块设计中的 pc 模块处进行绘图说明。

2.2.6 访存指令

访存指令在计组四的基础上从 lw(load word) 扩充到 lb,lbu,lh,lhu, 即加载字节、加载无符号字节、加载半字、加载无符号半字，从 sw(store word) 扩充到 sb,sh, 即存储字节，存储半字。

在扩充后的执行逻辑为，用地址最低两位来为字节与半字选择写的信号，对于读直接读入整个字，然后用地址的最低两位取出需要读的内容即可。同时需要注意对于半字和字是由地址错误的例外，半字的二进制地址最低位必须为 0，字的二进制地址最低两位必须为 00。

具体的设计流程为：

第一步的是修改 blockmemory ip 核的参数，选择写字节的模式。

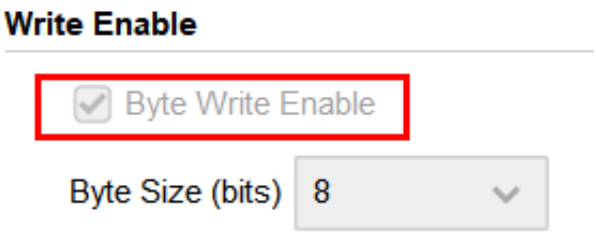

图 1: blockmemory_ip 配置

第二步然后需要清楚写字节模式下 blockmemory 的写接口使用方法，其中读写都需要将 memory 的 en 使能信号置为 1；写的时候，写字节，写片选 0001,0010,0100,1000 分别表示四个字节在字中的位置，写片选 1100,0011 表示两种半字的位置，1111 表示写字，0000 表示读字出来，此处对于读字节和半字是读出整个字然后再选取其中内容。

```
wire[31:0] readdata;
wire [3:0] selM;
```

```
data_mem dmem(
    ~clk,
    memwrite,
    selM,
    dataadr[11:2],
    writedata,
    readdata);
```

第三步，为了在使得写的信号生成，阶段即在 memroy 之前加一个 wirte_data_handle 的模块来生成片选写入的信号，为了得到正确的读出数据，在写回阶段选择写回数据之前加一个 read_data_handle 模块处理读出来的数据。其示例核心代码如下。

```
//w irte_data_handle
assign handled_WriteDataE = (sel == 4'b0000 || sel == 4'b1111)?WriteDataE:
                          (sel == 4'b1100 || sel == 4'b0011)? {WriteDataE[15:0],WriteDataE[15:0]} :
                          {WriteDataE[7:0],WriteDataE[7:0],WriteDataE[7:0],WriteDataE[7:0]} ;
// read_data_handle
always @ (*)
begin
    case (alucontrolW)
        `EXE_LW_OP:handled_readdataW <= readdataW;
        `EXE_LH_OP:
        begin
            case (dataadrW[1:0])
                2'b10: handled_readdataW <= {{16{readdataW[31]}},readdataW[31:16]};
                2'b00: handled_readdataW <= {{16{readdataW[15]}},readdataW[15:0]};
                default: handled_readdataW <= readdataW;
            endcase
        end
        `EXE_LHU_OP:
        begin
            case (dataadrW[1:0])
                2'b10: handled_readdataW <= {{16{1'b0}},readdataW[31:16]};
                2'b00: handled_readdataW <= {{16{1'b0}},readdataW[15:0]};
                default: handled_readdataW <= readdataW;
            endcase
        end
        `EXE_LB_OP:
        begin
            case (dataadrW[1:0])
                2'b11: handled_readdataW <= {{24{readdataW[31]}},readdataW[31:24]};
                2'b10: handled_readdataW <= {{24{readdataW[23]}},readdataW[23:16]};
                2'b01: handled_readdataW <= {{24{readdataW[15]}},readdataW[15:8]};
                2'b00: handled_readdataW <= {{24{readdataW[7]}},readdataW[7:0]};
                default: handled_readdataW <= readdataW;
            endcase
        end
        `EXE_LBU_OP:
        begin
            case (dataadrW[1:0])
                2'b11: handled_readdataW <= {{24{1'b0}},readdataW[31:24]};
                2'b10: handled_readdataW <= {{24{1'b0}},readdataW[23:16]};
                2'b01: handled_readdataW <= {{24{1'b0}},readdataW[15:8]};
                2'b00: handled_readdataW <= {{24{1'b0}},readdataW[7:0]};
                default: handled_readdataW <= readdataW;
            endcase
        end
    endcase
end
```

第四步是得到地址异常，在 alu 模块中生成读写地址时就判断地址是否对应异常，然后发出标记信息。

```
// addressError
```

```
assign laddressError = ( (alucontrol == `EXE_LH_OP || alucontrol == `EXE_LHU_OP) && (ans[0] != 0)
    )? 1:
                       (alucontrol == `EXE_LW_OP && ans[1:0] != 2'b00)? 1: 0;
assign saddressError = ( (alucontrol == `EXE_SH_OP) && (ans[0] != 0) )? 1:
                       (alucontrol == `EXE_SW_OP && ans[1:0] != 2'b00)? 1: 0;
```

2.2.7 特权与内陷指令

在计组 4 中是无与中断和异常相关的器件通路的,所以为了添加特权和内陷指令首先必须要清楚异常处理通路的设计。

在通路设计中,首先需要在 memory 阶段添加 cp0 模块 [课程参考代码] 和与 cp0 相适配的异常加工处理模块。然后在 main_dec 模块中添加必要控制信号。

在特权指令中的 mtc0,mfc0 两条是对 cp0 的中的值进行移进移出,需要在 main_dec 模块中添加 cp0write 控制信号表示写 cp0 的信号,可以在 alu 处理中将要输入 cp0 的 rs 值或要写回的 cp0 中的值都在 alu 中处理,统一用 alu 中的 aluout 进行写回 regfile。此外还要注意 mtc0 与 mfc0 的前后顺序容易产生数据冒险 [3],故使用刚写入 cp0 的值前推到 alu 的输入,用 cp0write 信号和数据冒险信号控制选择器。其核心代码如下。

```
// flowmips
mux2 #(32) forwardcp0datamux (cp0aluin,cp0dataoutE,aluoutM,forwardcp0dataE);
// alu
case (alucontrol)
       // sink in inst
    `EXE_MTC0_OP : ans <= num2 ;
    `EXE_MFC0_OP : ans <= cp0aluin ;
    default: ans <= 32'b0;
endcase
```

然后是特权指令中的 eret 指令,这个指令的含义是将 cp0 中的 epc 值作为新的 pc 值进行执行,一般是在异常结束的位置,用于返回原本执行代码块的对应地址。其只需要在 decode 阶段判断指令等于 eret 的 32 位二进制值,然后将此判断译码后传入 cp0 中,cp0 就可以正常产生中断信号并刷新所有的寄存器,跳转到指定的 pc 地方。其核心代码如下

```
// flowmips
assign eretD = (instrD == 32'b01000010000000000000000000011000);

// exception decode
if (exception[4])    // eret
    exceptiontype <= 32'h0000_000e;

// 传回pc
case (exceptiontype)
32'h0000_000e: pcexception <= cp0epc;
       default: ;
    endcase
```

对于内陷指令,其处理逻辑极为简单,只需要在 decode 阶段对其译码判断为 systemcall 或者 break 指令,标记传输到 w 阶段,进行例外译码,在 cp0 中做相应处理,然后跳转 pc

到 bfc00380 的入口欧地址即可。其核心代码如下。

```

// 异常判断
assign syscallD = (instrD[31:26] == 6'b000000 && instrD[5:0] == 6'b001100);
assign breakD = (instrD[31:26] == 6'b000000 && instrD[5:0] == 6'b001101);

// 译码
           else if (exception[6])    // system call
               exceptiontype <= 32'h0000_0008;
       else if (exception[5])    // break
               exceptiontype <= 32'h0000_0009;
// 跳转
    32'h0000_0008: pcexception <= 32'hbfc0_0380;
    32'h0000_0009: pcexception <= 32'hbfc0_0380;
```

此外,还要考虑软中断的问题,软中断是由 mtc0 产生的,其将 cp0 中的 cause 改变后,会影响译码得到软中断异常类型,但是由于 clk 激发的缘故,M 阶段改变 cp0 的 cause,在 W 阶段才会激发软中断,相对于其他异常的激发晚了一个周期。所以在 axi 接口面对握手机制,就可能产生不必要的冲突,所以在这本项目的决策是前推软中断,在 e 阶段就直接用组合逻辑单独判断出是否出现软中断,在 m 阶段同时改变 cp0 的 cause 和前推的 epc, 并同时得到异常类型和异常 pc 值,执行异常跳转。其核心代码如下。

```
wire [31:0] real_causeout,real_pcM;
reg [31:0] cause_o;

always@(*) begin
    if(rst) cause_o = 32'b0;
    else begin
        cause_o[9:8] = aluoutM[9:8];
        cause_o[23] = aluoutM[23];
        cause_o[22] = aluoutM[22];
    end
end
assign real_causeout = (RdM == 5'b01101 && cp0writeM) ? cause_o:causeout;
assign real_pcM = (RdM == 5'b01101 && cp0writeM) ? pcE : pcM;
assign bad_addr = (exceptM[7])? pcM : aluoutM; //pc错误时, bad_addr_i为pcM, 否则为计算出来的load
     store地址
```

2.2.8 sram 接口的 soc 设计

由于我们小组的 cpu 接口在实现访存指令时本来就是 sram 的接口,这里只需要将接口放置在数据通路上即可,特别注意,sram 需要进行地址映射,这里使用课程发 [课程代码] 的 mmu 模块进行地址映射。

```

// 转换cpu的地址到sram的地址
mmu my_addr_translate(.inst_vaddr(cpu_inst_addr),.inst_paddr(inst_sram_addr),
.data_vaddr(cpu_data_addr),.data_paddr(data_sram_addr),.no_dcache(no_dcache));
```

由于指令内存不需要写,可以直接将指令的 wen 信号和 wdata 信号直接置为全 0,其它的信号都从通路中接出来即可,对于 debug 信号,需要将通路中第五阶段写回时的 sram 信号单独放出来。这几个信号将用于测试时读 trace 文件来判断 cpu 是否正确执行了代码。

```

```

```
// DEBUG OUTPUT
assign debug_wb_pc = pcW;
assign debug_wb_rf_wen = {4{regwriteW}};
assign debug_wb_rf_wnum = WriteRegW;
assign debug_wb_rf_wdata = ResultW;
```

特别注意，mycpu_top 中的 reset 信号是 0 表示 reset，1 表示正常的，所以接数据通路时要将这个信号的反接入数据通路

连接完 sram 接口，就可以进行基础的 89 个测试点的功能测试，这些测试中会对 57 条指令一一进行测试，并在 tcl 命令行中打印通过的测试点的个数。下图是 sram 的连接结构：

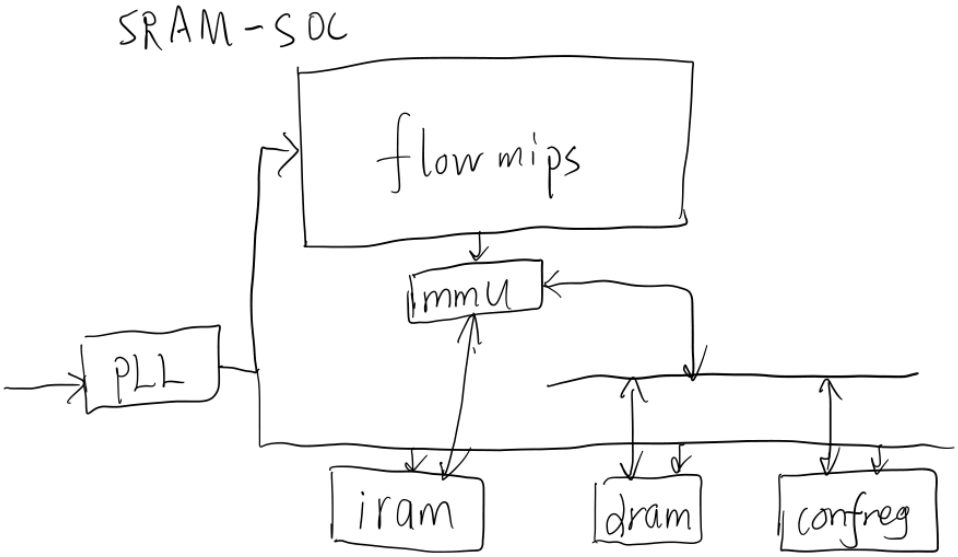

图 2: sram 结构图

2.2.9 axi 接口的 soc 设计

了 sram 的测试后，可以进行 axi 的接口连接，由于课程 [课程代码] 已经提供了龙芯杯的类 sram 转 axi 的接口，我们小组参考了课程提供了的 sram 转类 sram 的转换桥，将原本的 sram 的 cpu 转换成类 sram 的接口 cpu，最后利用龙芯的接口，将 cpu 连接上去，实现完整的 soc_axi 环境，并进行功能测试和性能测试。

在连接完 axi 后，我们小组完成了实验基本要求的最后一部分连接 cache，我们小组连接了写回的 dcache 和读 icache，由于 axi 环境中有部分地址的数据不来自 ram，而是来自 confreg，所以这一部分的数据不能经过 cache，这里可以借鉴体系结构 cache 实验的 bridge 方法 (简单的选择组合逻辑，就是利用 mmu 的 nodcache 信号，不经过 cache 的信号直接传输，经过 cache 的信号先传到 cache 再传到类 sram 转 axi 的转接口)，这样就完成了一个带 cache 的 axi 接口。完整的 soc_axi 模型如下所示：

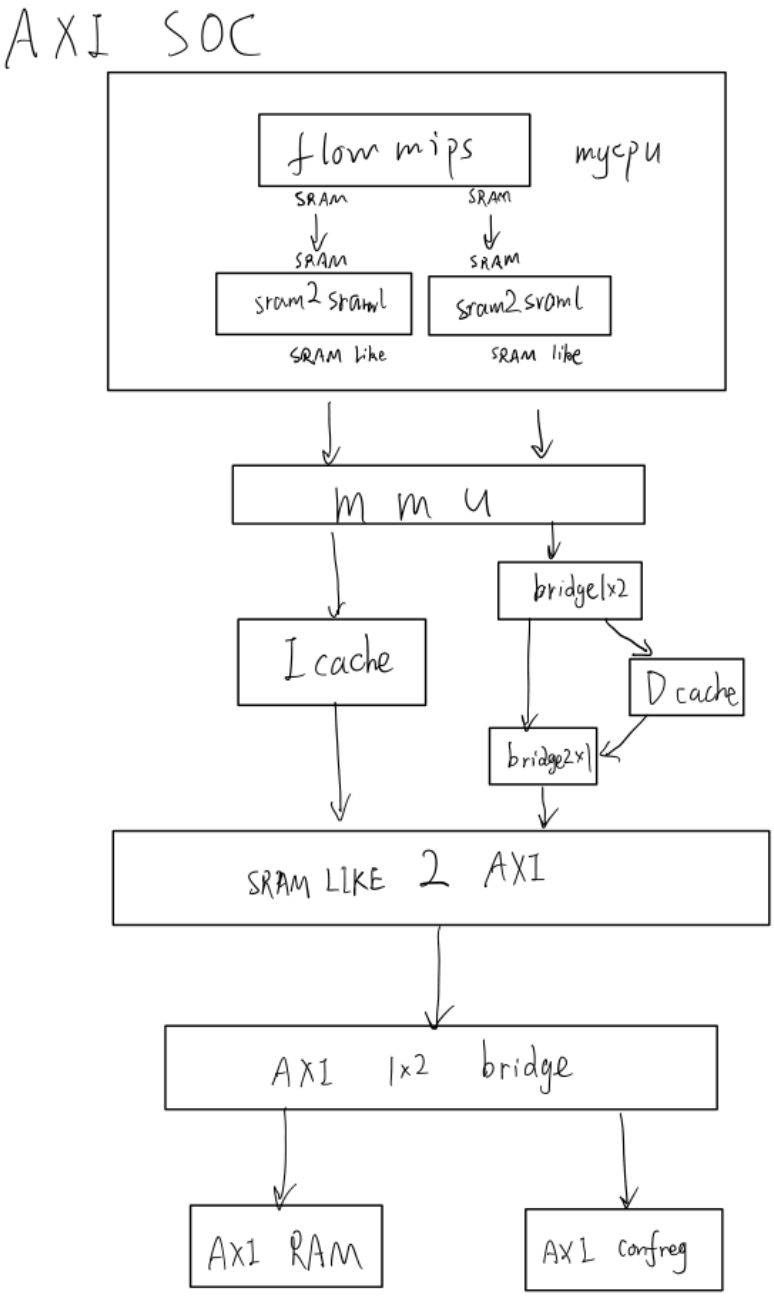
AXI SOC
flow mips
mycpu
SRAM
SRAM
SRAM
SRAM
sram2 sram1
sram2 sram1
SRAM like
SRAM like
mmu
bridge1x2
I cache
D cache
bridge2x1
SRAM LIKE 2 AXI
AXI 1x2 bridge
AXI RAM
AXI confreg

图 3: axi 结构图

cache 分为指令 icache 和数据 dcache,这两种 cache 在没获得数据时,不会发出 data_ok 信号,因此 sram 转类 sram 转换桥就不会结束 stall 信号,这样就实现了让原本的流水线通路完全暂停,阻止未获得数据的指令继续执行。关于 cache 的设计,由于我们小组之前都先修了体系结构课程 [1],在这门课的实验中已经完成了写直达 (实验自带), 写回,二路组相连,四路组相连的 cache,这里可以直接引用。经过 89 条功能测试点仿真验证,发现代码执行正确,说明 cache 设计无问题。这里简单介绍一下写直达和写回的关键点: 写直达就是写透,直接将 cpu 传过来的数据既写到 ram 里又写到 cache 去,状态图较简单; 对于写回 cache,需要设置一个标记 cache 是否为 dirty 的状态位,并且由于我们的 cpu 还支持短的访存指令 lb,sh 等,所以实现起来比一般教材的描述复杂,只要没命中,就需要先读 ram 再写 cache,因为如果是短指令 sh,如果没命中就直接写 cache,那么就会写入一个不属于它的 16bit 中,所以需要先将它所在的完整 32bit 读出来,再更改其中的 16bit,并且如果写缺失,且原来的 cache 已经是 dirty 状态,需要先将 dirty 的数据先写回 ram,再读出该条指令所在的 32bit,最后再写 cache。可以将总结成以下的状态图,其中 IDLE 是空闲状态,RM 是读 ram 状态,WD 是写 dirty 的数据的状态:

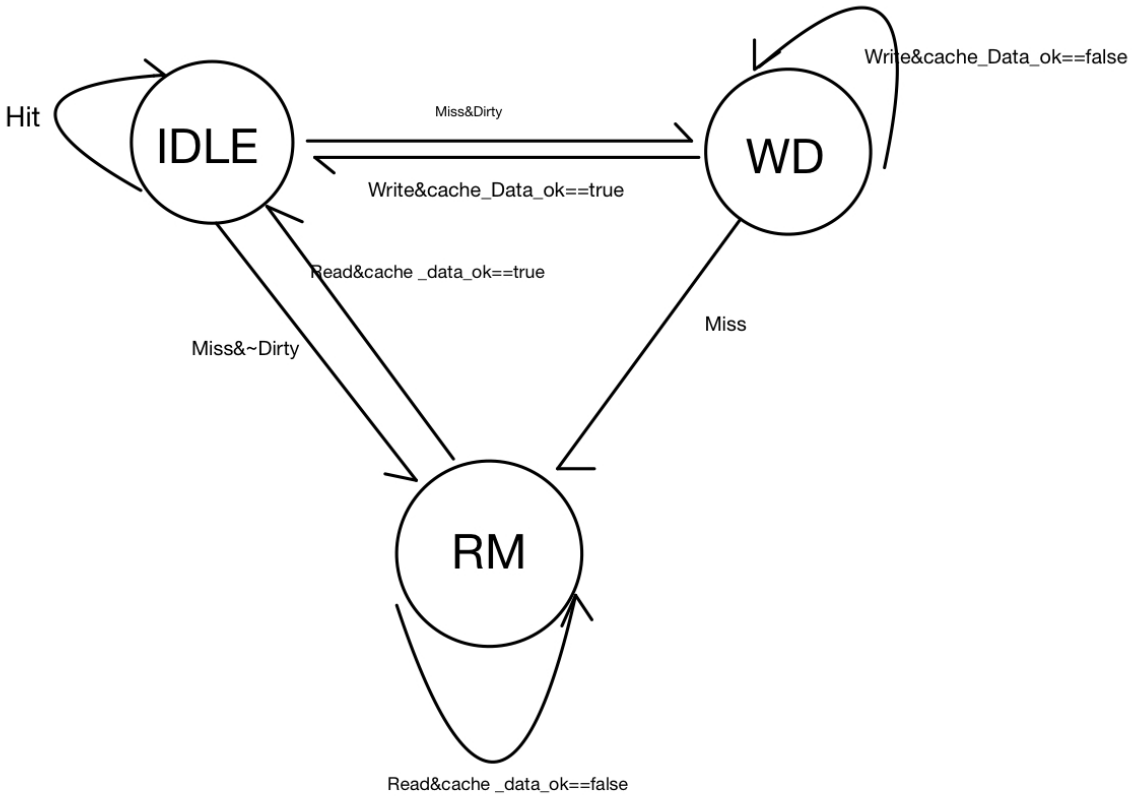

图 4: cache 状态机

2.3 模块设计

在如上的指令的添加后，整个的数据通路模型已经成型，总的数据通路是 flowmips 模块，接下来将重点介绍其中的 pc 处理模块,main_dec 模块,alu_dec 模块,alu 计算模块，存储处理模块，异常处理模块,hazard 模块。最终的数据通路雏形如下图。

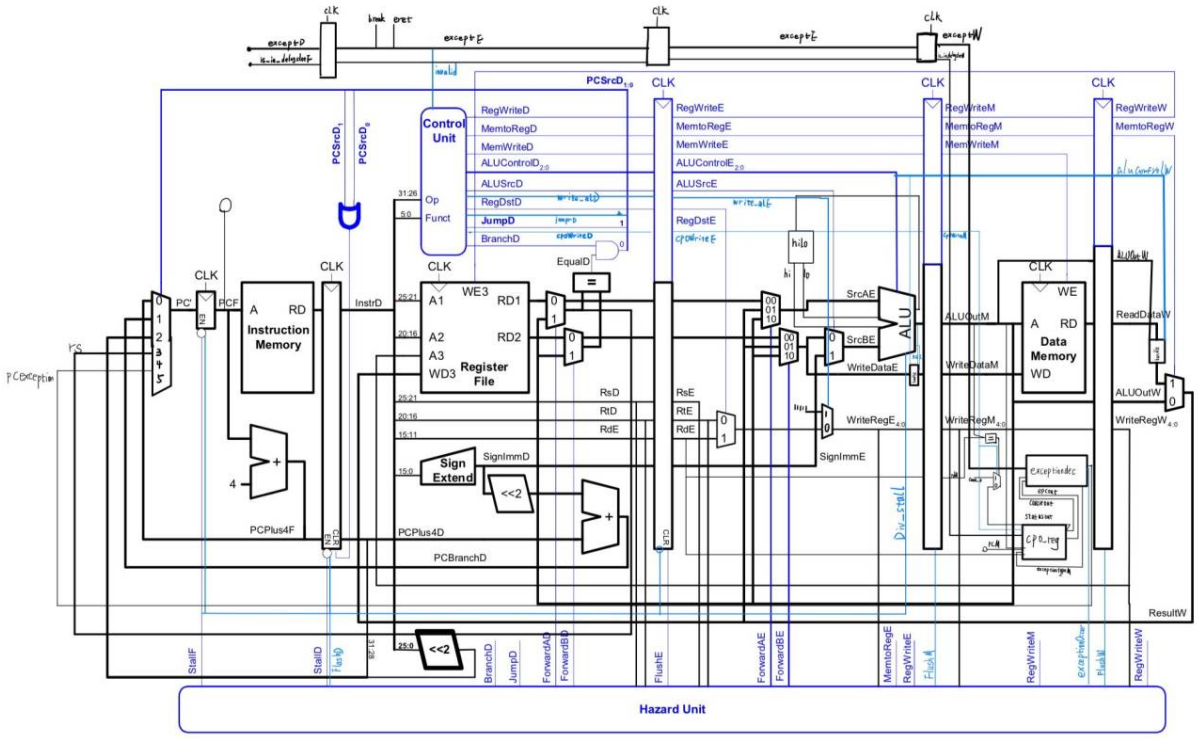

图 5: 数据通路图

2.3.1 flowmips 顶层模块设计

flowmips 文件中,主要是连接如上数据通路图的各个部件,和定义各种连线变量,用 assign 语句来执行一些简单的信号合并与分离操作。其对外是连接指令存储器和数据存储器。其接口定义如下表。

表 1: flowmips 接口定义

信号名	方向	位宽	功能描述
clk	input	1-bit	时钟
rst	input	1-bit	复位
instr	input	32-bit	指令编码
readdata	input	32-bit	数据存储器的读出数据
i_stall	input	1-bit	指令存储器 cache 的运行中 stall 信号
d_stall	input	1-bit	数据存储器 cache 的运行中 stall 信号
pc	output	32-bit	pc 地址
memwriteM	output	1-bit	数据存储器使能信号
aluoutMk	output	32-bit	alu 在 M 阶段的计算结果
WriteDataM	output	32-bit	写入数据存储器的数据
selM	output	4-bit	写数据存储器的片选信号
longest_stall	output	1-bit	全局 stall 信号
debug_wb_pc	output	32-bit	写回阶段的 pc 值,debug 用
debug_wb_rf_wen	output	4-bit	写回阶段的存储器使能,debug 用
debug_wb_rf_wnum	output	5-bit	写回阶段的寄存器号,debug 用
debug_wb_rf_wdata	output	32-bit	写回阶段的写存储器数据,debug 用

2.3.2 PC 模块设计

pc 就是一个 D 触发器,如果有 reset 信号,就将 pc 输出置为 0(**特别注意,如果是 sram 或 axi 版本,需要将 pc 置为 0xbfc00000**),否则置为 pc 的输入。这里的输出是 pc_out 代表 pc 值和 inst_ce 代表指令的地址,由于在 soc 中指令地址就是 pc,所以这两个值等价 (在计组 4 中,指令的地址 inst_ce 等于 pc_out[:2],即后两位不要的 pc 值)

表 2: PC 接口定义

信号名	方向	位宽	功能描述
clk	input	1-bit	时钟信号。
rst	input	1-bit	重置信号。
en	input	1-bit	使能信号,表示可以将输入赋值给输出。
pc_in	input	32-bit	pc 的输入。
pc_out	output	32-bit	pc 的输出。
inst_ce	output	32-bit	指令的地址。

由于跳转,分支预测,分支判定,异常等都会影响 PC 的输入,在通路中我们是一步一步加的选择器,所以这些选择器就可以抽象成一个 PC 的输入模块。下图展示了 pc 的输入值的选择:

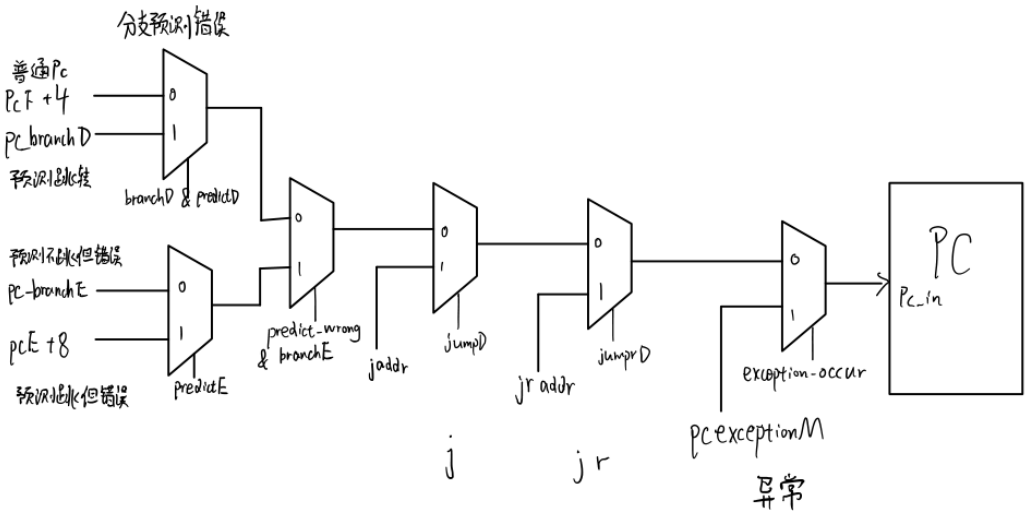

图 6: pc 及其输入选择

2.3.3 ALU 模块设计

ALU 模块实现了对不同指令所需要的数值进行运算。其信号的完整定义与下表 1 所示。在 ALU 的内部,根据当前阶段指令所对应的 ALU 控制信号(即 alucontrol)对输入的操作数进行对应的操作,并对指令需要的信号量进行值的计算。

ALU 模块需要处理有符号运算的溢出问题,所以需要 overflow 信号来表示当前运算的溢出情况,并且对于有符号的加法和减法,判断溢出的方式都不一样,加法是相同位相加所得结果符号位改变则溢出,而减法则是先把第二个操作数取反,再进行符号位溢出的判断。实现如下:

```
assign overflow_add = ( (ans[31] & (~num1[31] & ~num2[31]))
              || (~ans[31] & (num1[31] & num2[31]))) &&(alucontrol == ‘EXE_ADD_OP || alucontrol ==
                  ‘EXE_ADDI_OP );
assign overflow_sub = ( (alucontrol == ‘EXE_SUB_OP ) &&
              ((ans[31] & (~num1[31] & ~num2_reg[31])) || (~ans[31] & (num1[31] & num2_reg[31])))
              );
```

除了基本指令运算以外,还加入了乘除法模块。其中乘法运算调用了 my_mul 模块,将输入的操作数信号导入乘法模块进行运算。乘法运算的实现是组合逻辑,不涉及流水线暂停,但除法运算所调用的 div_radix2 模块需要在运算的过程中对流水线进行暂停。当除法执行完毕以后,暂停结束,通过除法器的握手信号告知 ALU 计算完成,此时,因为除法而导致的流水线暂停结束,暂停信号设置为 0.

ALU 模块还需要将乘除法的结果写入到 hilo 寄存器中。我们的数据通路设计将 hilo 寄存器放置在了第三阶段,因此乘除法的结果在 ALU 中运算结束后会直接存入。

由于 ALU 模块还对 Load(加载)和 Store(存储)两类指令的存取内容格式进行了计算,因此还要对计算的结果进行检验是否正确,如果检查发现错误,则需要将输出信号量 ladressError 或 saddressError 设置为 1。

表 3: ALU 接口定义

信号名	方向	位宽	功能描述
clk	input	1-bit	时钟信号。
rst	input	1-bit	重置信号。
num1	input	32-bit	alu 运算的第一个操作数。
num2	input	32-bit	alu 运算的第二个操作数。
sa	input	5-bit	立即数,决定了移位操作的偏移量。
alucontrol	input	8-bit	alu 操作序号。
hi	input	32-bit	hilo 寄存器的高 32 位数据。
lo	input	32-bit	hilo 寄存器的低 32 位数据。
flushE	input	1-bit	清除第二到第三阶段寄存器的信号，同时用于打断清除除法运算。
stallM	input	1-bit	三至四阶段中间寄存器的停顿信号，用于除法器的接收信号。
pc_add4E	input	32-bit	第三阶段的 PC+4 的值。
cp0aluin	input	32-bit	MFC0 指令的输入。
exceptionoccur	input	1-bit	出现异常信号。
real_ans	output	32-bit	处理完溢出后的实际计算输出。
hilo_out	output	64-bit	hilo 寄存器的输出，用于放乘除法计算结果。
overflowE	output	1-bit	计算结果是否溢出。
zero	output	1-bit	处理 B 指令的跳转结果。
div_stallE	output	1-bit	除法运算的停顿。
laddressError	output	32-bit	读地址错误例外。
saddressError	output	32-bit	写地址错误例外。

2.3.4 main_dec 模块设计

main_dec 模块的功能通过输入的指令对指令类型进行解码,并得到对应的控制信号的值。其完整输入输出信号量定义如下表 2。在 main_dec 模块中,根据输入的指令 opcode 以及 funct 两段数据,对指令操作进行分析,得到当前指令所需要进行的操作,用输出信号量表示。

例如,对于需要写寄存器的指令类型(如逻辑指令、移位指令以及 JAL 类型指令等),将 regwrite 信号和 regdst 信号都掷为 1,而对于 Branch 类的跳转指令,需要把 branch 信号掷为 1.

对于 Jump 类无条件跳转指令,考虑到三种情况(跳转地址由立即数计算、跳转地址由寄存器给、跳转后还要写寄存器),将 jump 相关的信号分成三个,分别为 jump、jumpr 以及 al_regdst. 在不同的跳转指令中,其所对应的 jump 信号将会被设置成 1. 实现代码如下:

```
assign jump = ((op == `EXE_J) || (op == `EXE_JAL)) ? 1 : 0;
assign jumpr = ((op == `EXE_NOP) && ((funct == `EXE_JR) || (funct == `EXE_JALR))) ? 1 : 0;
```

```
3
4 assign al_regdst = (((op == `EXE_REGIMM_INST) && (rt == `EXE_BLTZAL || rt == `EXE_BGEZAL)) // 两条
      bzal指令
5                       || (op == `EXE_JAL)) ? 1 : 0; // jal指令
```

对于异常类型的指令，main_dec 模块也要根据异常指令的 opcode 进行解码，将异常处理相关的操作信号如 cp0write、invalid 信号的值进行更新。如，如果当前指令为 mtc0 则需要将 cp0write 信号掷为 1.

表 4: main_dec 接口定义

信号名	方向	位宽	功能描述
op	input	6-bit	当前指令的 opcode 信号。
funct	input	6-bit	当前指令的 funct 信号。
rs	input	4-bit	当前指令的源寄存器 1。
rt	input	4-bit	当前指令的源寄存器 2。
regwrite	output	1-bit	regfile 的写使能，表示是否要写入数据。
regdst	output	1-bit	选择 rd 还是 rt 作为写的目标。
alusrc	output	1-bit	选择立即数还是 reg 中的值输入 ALU。
branch	output	1-bit	跳转(branch 信号)。
memtoreg	output	1-bit	控制写入 regfile 的是来自 alu 计算结果还是 data_mem 中取出的数据。
memwrite	output	1-bit	写内存。
al_regdst	output	1-bit	是否需要写寄存器，针对 AL 型指令。
jump	output	1-bit	是否跳转，针对 J 跳转指令。
jumpr	output	1-bit	是否用寄存器值计算跳转目的地址，针对 JR 类型指令。
invalid	output	1-bit	保留地址异常。
cp0write	output	1-bit	写入 CP0 信号。

2.3.5 alu_dec 模块设计

alu_dec 的功能是通过当前指令中的 funct 和 op，来解码出当前的 alucontrol 值，进一步传输到 ALU 中，对 ALU 的操作进行控制，其完整的输入输出信号定义如下表所示。

表 5: alu_dec 接口定义

信号名	方向	位宽	功能描述
op	input	6-bit	当前指令的 opcode 信号。
funct	input	6-bit	当前指令的 funct 信号。
rs	input	4-bit	当前指令的源寄存器 1。
rt	input	4-bit	当前指令的源寄存器 2。
alucontrol	output	8-bit	ALU 控制信号。

在本次硬件设计中，与计组实验 4 不同之处在于，我们取消了 aluop 的中间信号连接 main_decode 和 alu_dec，直接使用 [5:0]funct + [5:0]op 来判断 [7:0] alucontrol。我们使用了 defines.vh 中的宏定义对输入进行判断、对 alucontrol 进行赋值。

alu_dec 得到 ALU 操作的控制信号，main_dec 获得了指令在整个数据通路里需要用到的

数据信号,两者都以当前正在执行的第二阶段收到的指令为输入,根据内部的逻辑结构产生输出,并且连接到 controller 模块上,由 controller 进一步的控制这些信号在哪一流水线阶段起到作用。

2.3.6 分支预测之局部预测模块

branch_predict_local 模块是分支预测中的局部预测模块,其信号量如下表格所示。局部历史更新即在预测模块中维护一个 BHT 和 PHT。他的核心技术在于用一个寄存器保留了当前指令过去 N 次跳转的记录。而这个跳转序列,往往对应着一个跳转结果的规律(这个规律由我们设计训练得到)我们所训练的结果被存放在了 PHT 中。通过跳转序列在 PHT 中检索,可以找到对于当前这种序列,下一次结果的预测值。[2]

表 6: branch_predict_local 接口定义

信号名	方向	位宽	功能描述
clk	input	1-bit	时钟信号。
rst	input	1-bit	重置信号。
flushD	input	1-bit	决定 decode 阶段是否被冲掉所有数据。
stallD	input	1-bit	决定 decode 阶段是否暂停。
pcF/pcM	input	1-bit	Fetch 阶段或 Memory 阶段的 PC 值。
branchD/branchM	input	1-bit	decode 阶段或 memory 阶段跳转判断信号。
actual_takeM/actual_takeE	input	1-bit	当前指令实际是否跳转
pred_takeD/ pred_takeF	output	1-bit	预测是否跳转, decode 阶段的结果是基于 fetch 的信号与 branchD 得到的。

对于一条跳转指令,使用一个寄存器 BHR(Branch History Register) 来记录该指令的前面 N 次的历史跳转记录,并可以针对每一种序列,训练一个两位饱和计数器,然后根据这个饱和计数器的状态预测跳转方向。(饱和计数器的内容是前文所提到的状态机)。

可以用 PC 的一部分来寻址 BHT 的表项。同时,由图可知,并不是 PHT 的每一个表项都被使用,所以我们可以使用 PC 的一部分来寻址到哪一个 PHT。并且,更为极端的一种情况,所有分支指令可以共用一个 PHT。

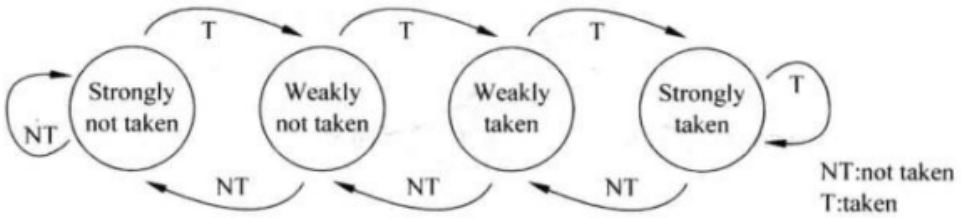

图 7: 分支预测状态机示意图

2.3.7 分支预测之全局预测模块

branch_predict_global 模块式全局预测模块。全局历史更新中需要一个全局历史寄存器(Global History Register)来记录程序中所有的分支指令在过去的执行情况。GHR 的宽度不能是无穷大,只能用有限的宽记录最近执行的结果。本模块的信号定义与局部预测模块相同,可以参见表 5.

在每当要对一条分支指令进行分支预测的时候,就会根据当前的 GHR 寄存器的值进行预测。这个时候依旧会借助 PHT(依旧是两位的饱和计数器)进行预测结果的存储。我们在实验中只用了一个 GHR 寄存器,用来记录最近执行的所有分支指令的结果。同时,使用 GHR 的值和 PC 的值一起去 PHT 中查找预测结果。

在 fetch 阶段就得到对跳转分支的预测。在接到了 branch 信号以后,对 GHR 的值进行更新。为了避免连续跳转分支指令对 GHR 存储的值所造成的影响,我们一共引入了三个变量(GHR_value_old_X)分别表示不同阶段处理单元中存放的值,从而避免了错误。

2.3.8 分支预测之竞争预测模块

compete_predict 模块是竞争分支预测模块。前面所提到的两种方法(局部历史和全局历史)都各有优缺点,所以最后引入竞争分支预测。所谓竞争,就是前面两种方法竞争,取更准确的结果。本模块的信号定义也与局部预测模块相同,可以参见表 5.

与前面不同的是,竞争预测有一个 CPHT,它是由分支指令的 PC 来寻找一个表格。这个表格也是由两位的饱和寄存器组成,是当其中一种分支预测方法失败的时候,同时会使用另一种方法再次预测。他所用到的状态机与前面不同,如下图所示:

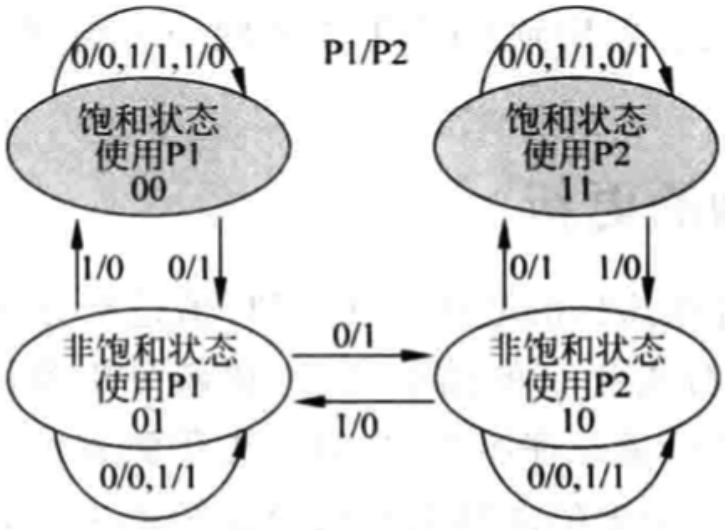

图 8: 竞争分支预测状态机

2.3.9 存储处理模块设计

存储处理模块中涉及控制信号生成、memoryblock、写入数据处理模块和输出数据处理模块。其相关的局部数据通路图如下。

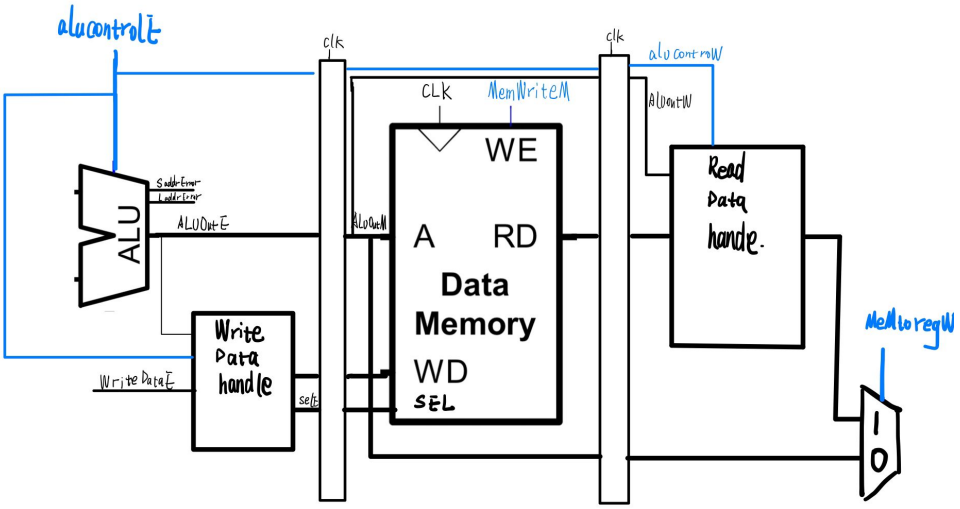

图 9: 存储数据通路图

可以看到图中最左边是 alu 模块在计算存储指令的地址，在计算的同时，其中就判断了与半字和字相关指令的地址错误例外，并输出两种例外错误信号 saddrError 与 laddrError。然后在 E 阶段的 alu 计算出地址后，需要根据指令类型和地址信息来生成对 memory 读写操作的片选信号，这就是 Write_data_handle 模块。写的时候，写字节，写片选 0001,0010,0100,1000 分别表示四个字节在字中的位置，然后将写入的信息所有字节都替换为需要写入的字节；写片选 1100,0011 表示两种半字的位置，然后替换所有半字都等于需要写入的半字；1111 表示写字，0000 表示读字出来，此处对于读字节和半字是读出整个字然后再选取其中内容。此模块的接口信息表如下。

表 7: Write_data_handle 模块接口定义

信号名	方向	位宽	功能描述
alucontrolE	input	[7:0]	指令类型编码
aluoutE	input	[31:0]	alu 输出，即计算出的内存地址
WriteDataE	input	[31:0]	写入存储的数据
sel	output	[3:0]	存储的片选信号
handled_WriteDataE	output	[31:0]	处理后的写入数据

在通过 memory 模块读出数据后，这里的策略是对于所有读都是首先读出一整个字 32bit，然后在用图上的 Read_Data_handle 模块，根据指令类型来处理读出的整个字数据。对于 lh 即将对应的半字取出放在低 16 位，然后有符号扩展；对于 lhu 将对应半字取出放在低 16 位，然后 0 符号扩展；lb 即取出字节放在低 8 位，高 24 位有符号扩展；lbu 即取出字节放入低 8 位，然后 0 扩展高 24 位。该模块的接口表如下。

表 8: Read_data_handle 模块接口定义

信号名	方向	位宽	功能描述
alucontrolW	input	[7:0]	指令类型编码
readdataW	input	[31:0]	memory 读出的整字数据
dataadrW	input	[31:0]	读数据的地址
handled_readdataW	output	handled_readdataW	处理后的正确读出数据

2.3.10 异常处理模块设计

异常处理模块需要完成对于内限指令和特权指令的处理，此外还要处理软中断和其他异常的处理。对于这一模块的主要思路是，实现精确异常，在 F 阶段判断指令地址异常，在 D 阶段判断 systemcall、eret 和 break，在 E 阶段判断溢出例外和判断访存地址错误例外，最后在 M 阶段进行例外处理。由此在 Memory 阶段添加异常处理的 cp0 和 exceptiondec，其局部数据通路图如下。

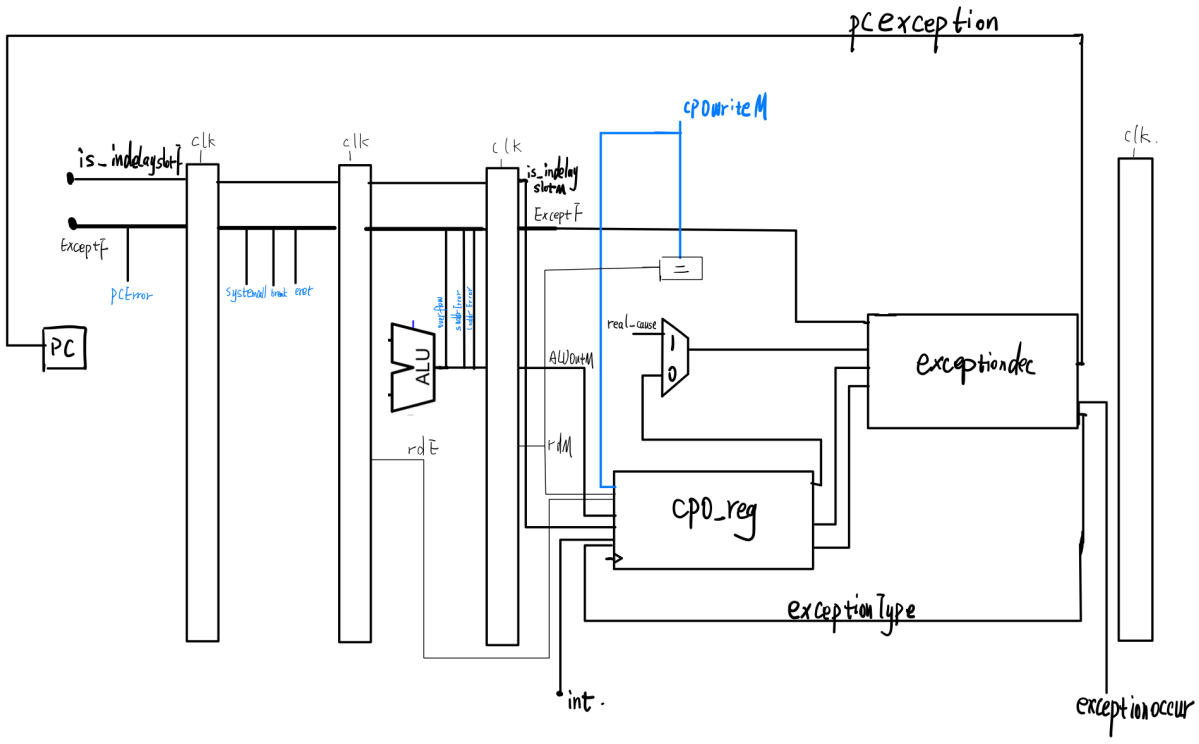

图 10: 异常数据通路图

其中从左到右首先是为每个阶段的异常处理打上标签，依次传递到 M 阶段，然后传入 cp0_reg 和 exceptiondec 模块。然后 cp0 会对内部寄存器进行一些处理，同时 exceptiondec 模块会生成异常类型，传入 cp0 进行后一步的操作，同时 exceptiondec 根据异常类型传出异常发生信号和跳转的异常 pc 地址。

对于 cp0 模块，其主要功能是通过是否写信号来实现 Mtc0 指令，同时也组合逻辑用 E 阶

段的 rdE 来进行 mfc0 的指令执行。此外,其还根据异常的标记来实现异常处理与更具 mtc0 来达到软中断的效果。其接口如下表。

表 9: cp0 接口定义

信号名	方向	位宽	功能描述
clk	input	1-bit	时钟
rst	input	1-bit	复位信号
we_i	input	1-bit	写控制信号
waddr_i	input	5-bit	写入数据的地址
raddr_i	input	5-bit	读数据的地址
data_i	input	32-bit	读出的数据
int_i	input	6-bit	外部中断
excepttype_i	input	32-bit	异常类型
current_inst_addr_i	input	32-bit	当前指令地址
is_in_delayslot_i	input	1-bit	是否位于延迟槽
bad_addr_i	input	32-bit	例外出错地址
data_o	output	32-bit	读出的数据
count_o	output	32-bit	cp0 内部寄存器
compare_o	output	32-bit	cp0 内部寄存器
status_o	output	32-bit	cp0 内部寄存器,存放 cp0 状态
cause_o	output	32-bit	cp0 内部寄存器
epc_o	output	32-bit	cp0 内部寄存器,存放异常 pc
config_o	output	32-bit	cp0 内部寄存器
prid_o	output	32-bit	cp0 内部寄存器
badvaddr	output	32-bit	cp0 内部寄存器,存放异常的内存地址
timer_int_o	output	1-bit	p0 内部寄存器

对于 exceptiondec 模块,其主要功能是配合 cp0,为其异常类型进行译码,同时为异常产生后发出异常信号和生成异常跳转 pc 地址。其接口的定义如下。

表 10: exceptdec 接口定义

信号名	方向	位宽	功能描述
rst	input	1-bit	复位信号
exception	input	8-bit	异常标记
laddrerror	input	1-bit	load 地址错误
saddrerror	input	1-bit	store 地址错误
cp0status	input	32-bit	cp0 状态寄存器
cp0cause	input	32-bit	cp0cause 寄存器
cp0epc	input	32-bit	cp0epc 寄存器
exceptionoccur	output	1-bit	异常发生信号
exceptiontype	output	32-bit	异常类型
pcexception	output	32-bit	异常跳转 pc

此外,在此模块中需要特别注意**软中断**, 软中断是由 mtc0 产生的,其将 cp0 中的 cause 改变后,会影响译码得到软中断异常类型,但是由于 clk 激发的缘故,M 阶段改变 cp0 的 cause,在 W 阶段才会激发软中断,相对于其他异常的激发晚了一个周期。所以在 axi 接口面对握手机制,就可能产生不必要的冲突,所以在本项目的决策是前推软中断,在 E 阶段就直接用组合逻辑单独判断出是否出现软中断,在 M 阶段同时改变 cp0 的 cause 和前推的 epc, 并同时得到异常类型和异常 pc 值,执行异常跳转。所以如上图的局部数据通

路即可看出在 exception 前有选择器选择前推的 cause 寄存器值。

2.3.11 hazard 模块设计

hazard 模块用来处理代码中的冒险情况,具体来说,我们小组的 cpu 将面临数据冒险和控制冒险这两种冒险,hazard 的定义如下:

表 11: Hazard 接口定义

信号名	方向	位宽	功能描述
rsD	input	5-bit	译码阶段的 rs 寄存器号
rtD	input	5-bit	译码阶段的 rt 寄存器号
rsE	input	5-bit	运行阶段的 rs 寄存器号
rtE	input	5-bit	运行阶段的 rt 寄存器号
rdE	input	5-bit	运行阶段的 rd 寄存器号
rdM	input	5-bit	访存阶段的 rd 寄存器号
writeregE	input	5-bit	运行阶段的写入寄存器号
writeregM	input	5-bit	访存阶段的写入寄存器号
writeregW	input	5-bit	写回阶段的写入寄存器号
regwriteE	input	1-bit	运行阶段的写寄存器信号
regwriteM	input	1-bit	访存阶段的写寄存器信号
regwriteW	input	1-bit	写回阶段的写寄存器信号
memtoregD	input	1-bit	译码阶段的从内存写到寄存器信号
memtoregE	input	1-bit	运行阶段的从内存写到寄存器信号
memtoregM	input	1-bit	访存阶段的从内存写到寄存器信号
branchD	input	1-bit	译码阶段的 branch 信号
jumprD	input	1-bit	译码阶段是否是 jr 类指令信号
cp0writeM	input	1-bit	访存阶段是否写 cp0 寄存器的信号
exceptionoccur	input	1-bit	是否发生异常的信号
div_stall	input	1-bit	除法 stall 信号
i_stall	input	1-bit	指令访存时的 stall 信号
d_stall	input	1-bit	数据访存时的 stall 信号
branchE	input	1-bit	运行阶段是否是 branch 类信号
predict_wrong	input	1-bit	分支预测错误的信号
forwardAE	output	2-bit	运行阶段的数据前推选择信号
forwardBE	output	2-bit	运行阶段的数据前推选择信号
forwardAD	output	1-bit	branch 数据前推选择信号
forwardBD	output	1-bit	branch 数据前推选择信号
foewardcp0dataE	output	1-bit	cp0 数据前推
stallF	output	1-bit	暂停 PC 和取指的信号
stallD	output	1-bit	暂停译码阶段触发器的信号
stallE	output	1-bit	暂停执行阶段触发器的信号
stallM	output	1-bit	暂停访存阶段触发器的信号
stallW	output	1-bit	暂停写回阶段触发器的信号
flushF	output	1-bit	刷新 PC 阶段触发器的信号
flushD	output	1-bit	刷新译码阶段触发器的信号
flushE	output	1-bit	刷新运行阶段触发器的信号
flushM	output	1-bit	刷新访存阶段触发器的信号
flushW	output	1-bit	刷新写回阶段触发器的信号
longest_stall	output	1-bit	全局 stall 信号

这个模块相当重要,可以处理冒险,下面进行具体的讨论。

首先是读写普通寄存器的冒险，可能读出来的寄存器还未写入，对于 jr 型指令，如果上一条是计算指令，且计算指令需要写的寄存器恰好就是 jr 的目标，则在 jr 的 D 阶段，计算结果还未出现，需要 stall 一个周期，并且将 M 阶段的 alu 结果前推到 D 阶段，即两个 forwardD 信号，如果恰好是访存指令，则需要 stall 更多周期，具体依据访存的情况。

```
assign jrstall = (jumprD & regwriteE & ((writeregE == rsD) | (writeregE == rtD))) | (jumprD &
    memtoregM & ((writeregM == rsD) | (writeregM == rtD)));

assign forwardAD = (rsD != 5'b0) & (rsD == writeregM) & regwriteM;
assign forwardBD = (rtD != 5'b0) & (rtD == writeregM) & regwriteM;
```

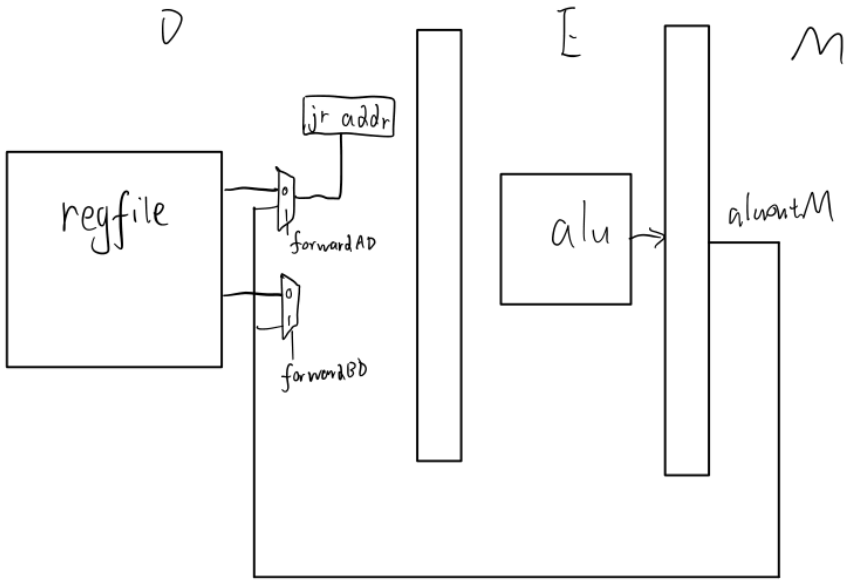

图 11: forwardD 演示图

对于普通的计算指令，不需要 stall，但是需要进行数据前推，将计算的结果 aluoutM 传回 alu 的输入那里进行前推，因为这些数据需要进入 alu，所以前推到 E 阶段 alu 前即可，不必推到 D 阶段，这就是两个 forwardE 信号。

```
assign forwardAE = ((rsE != 5'b0) & (rsE == writeregM) & regwriteM) ? 2'b10: ((rsE != 5'b0) & (rsE
    == writeregW) & regwriteW) ? 2'b01: 2'b00;
assign forwardBE = ((rtE != 2'b0) & (rtE == writeregM) & regwriteM) ? 2'b10: ((rtE != 5'b0) & (rtE
    == writeregW) & regwriteW) ? 2'b01: 2'b00;
```

对于 div_stall、i_stall 和 d_stall，将它们统一综合成 longest_stall，全流水线都暂停。

```
assign longest_stall = i_stall | d_stall | div_stall;
```

对于 cp0 的读写，如果前一条是 mtc0 后一条是 mfc0，则需要前推 cp0 的数据到 E 阶段。

```
// mtc0 mfc0冲突
assign forwardcp0dataE = (rdE && (rdE == rdM) && cp0writeM);
```

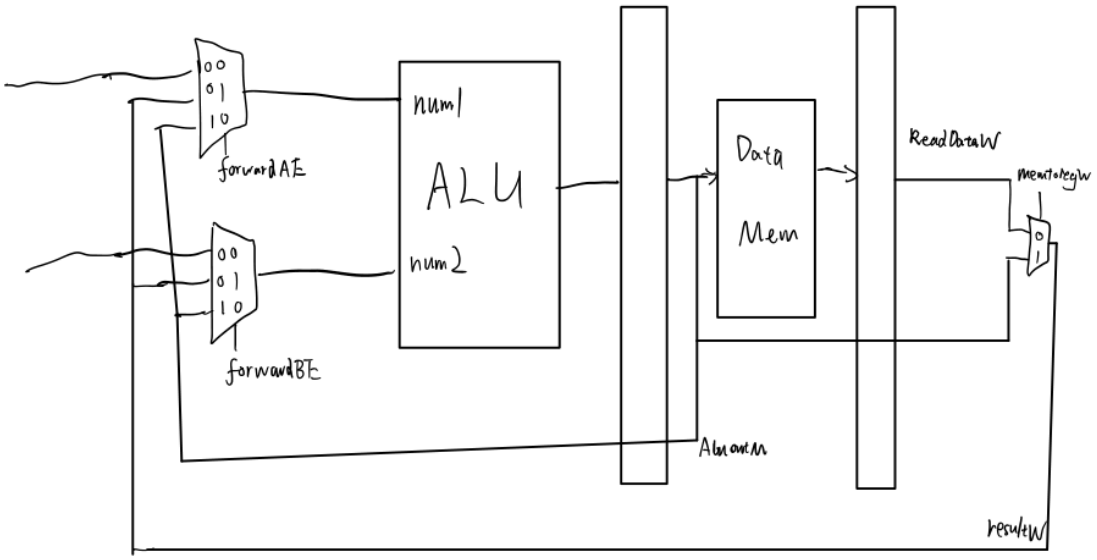

图 12: forwardE 演示图

对于 load 型指令,这些指令在 M 阶段结束才可以得到值,所以如果有数据相关,则需要进行 stall,将 FD 都 stall,并且用 flushE 刷新 E 阶段。

```
assign lwstall = (((rsD == rtE) | (rtD == rtE)) & memtoregE);
```

此外,注意异常的存在,如果出现异常,需要对流水线进行清空,并且由于 axi 接口的特殊性,异常的清空不能同时与 axi 访存暂停同时发生。如果分支预测错误,还进行需要 flushD。

综上所述,我们小组的 hazard 模块对 cpu 的数据通路的控制信号设置主要代码如下

```
assign stallF = (longest_stall | lwstall | jrstall) & ~exceptionoccur;
assign stallD = longest_stall | lwstall | jrstall;
assign stallE = longest_stall;
assign stallM = longest_stall;
assign stallW = longest_stall;

assign flushF = 1'b0;
assign flushD = ((branchE & predict_wrong) | exceptionoccur) & (~longest_stall);
assign flushE = (lwstall | jrstall | exceptionoccur) & (~longest_stall);
assign flushM = (exceptionoccur) & (~longest_stall);
assign flushW = (exceptionoccur) & (~longest_stall);
```

3 设计过程

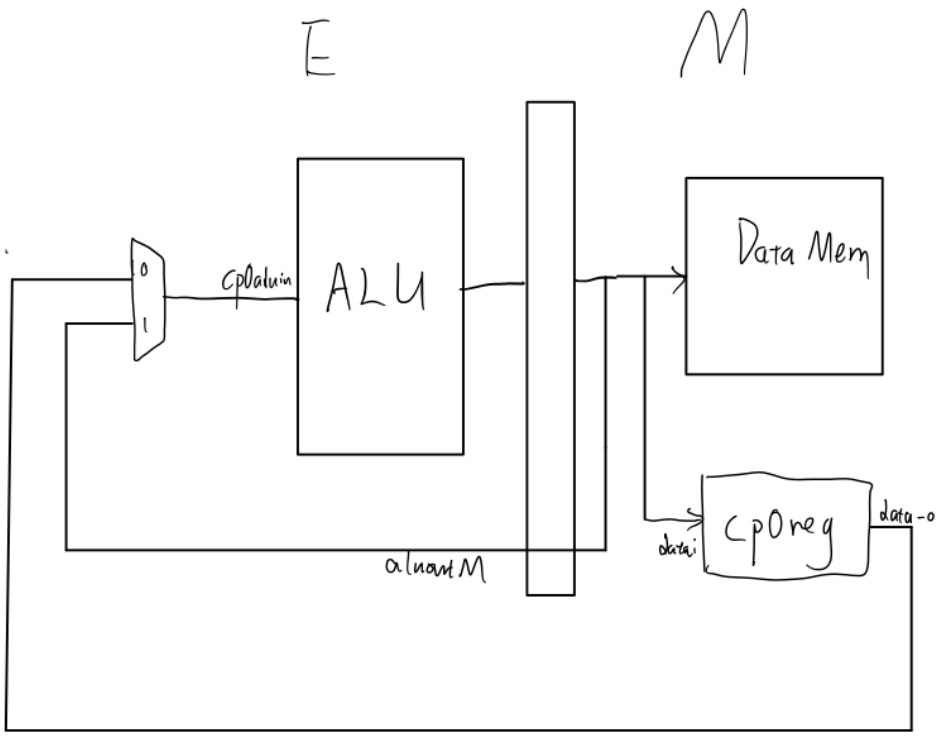

图 13: cp0forward 演示图

3.1 设计流水账

项目的设计过程每个人的工作日志如下,其中朱海龙和屈湘钧是项目全程参与;李颀琳是 1 月 5 日加入本项目,故其工作日志表格单独列出,1 月 5 日后的工作日志是参与本项目的内容。我们小组有一个自己的 github 地址,可以在这里看到我们所有的提交记录和工作日志等等`https://github.com/marxoffice/MIPS_HARDWARE`

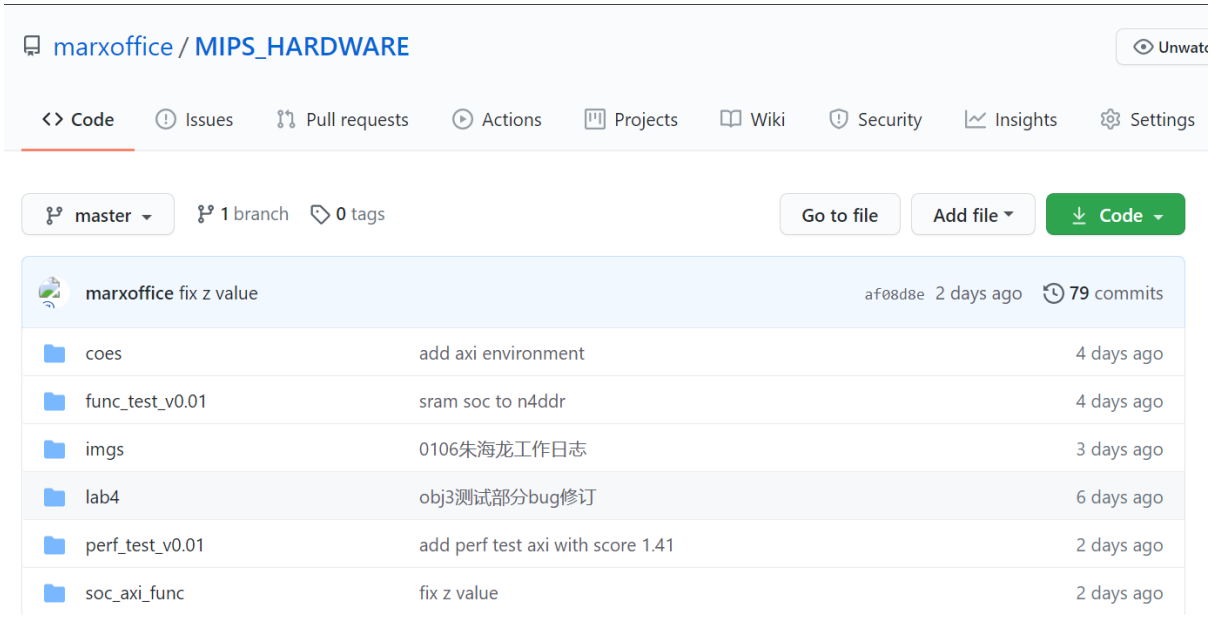

图 14: github 展示

表 12: 工作日志

日期	时间	人员	工作内容
2020.12.25	10:00-12:00	屈湘钧	复习计组 4 的简单五级流水线 cpu，并大致整理任务目标和初步的上手想法。
2020.12.25	15:00-18:00	屈湘钧	学习吕学长教学 57 条指令添加教程，大致思考添加指令的第一步译码做法。
2020.12.25	8:00-18:00	朱海龙	分析实验通路，阅读 soc，axi，sram 资料
2020.12.26	10:10-12:00	屈湘钧	基本写完 alu_dec.v 文件
2020.12.26	14:20-18:00	屈湘钧	完成大部分逻辑运算指令添加工作；变更指令编码后的计组 4 测试调试通过
2020.12.26	8:50-17:30	朱海龙	阅读测试文档，初步完成 main_dec 和 controller 的编写
2020.12.27	14:40-18:00	屈湘钧	完成逻辑运算指令 8 条的添加和测试工作
2020.12.27	9:00-18:30	朱海龙	完成移位指令添加，完成部分 hilo_move 指令添加
2020.12.28	14:30-24:00	屈湘钧	听课，做算术指令的添加，已完成初步代码，正在测试
2020.12.28	8:30-12:00	朱海龙	完成数据移动指令，修复移位指令 bug，完成这两类指令的测试
2020.12.29	13:00-17:45	屈湘钧	完成算术指令 14 条的添加与测试通过，初步想好了访存指令的构建
2020.12.29	8:30-12:00	朱海龙	完成分支 branch 指令 (包含分支预测)，并完成分支 branch 测试
2020.12.30	14:30-19:30	屈湘钧	基本完成访存指令代码，测试通过计组 4，还未做单元测试
2020.12.30	8:30-12:00	朱海龙	完成跳转 jump 指令 (包含 jal 的数据前推)，并完成跳转 jump 测试
2021.01.01	12:00-24:00	朱海龙	修改分支预测机制，将延迟槽指令先运行，并重新测试 branch 指令
2021.01.01	12:00-24:00	屈湘钧	完成访存指令的所有单元测试，分析解决访存冒险，并开始看中断
2021.01.02	12:00-24:00	朱海龙	尝试连接 sram 接口，出现无法读取 trace 文件 bug
2021.01.02	13:30-23:00	屈湘钧	添加异常相关的代码，重复测试以往 coe
2021.01.03	10:00-24:00	朱海龙	sram 接口连接，修复部分指令 bug，完成 52 条指令测试
2021.01.03	13:30-23:00	屈湘钧	内陷 & 特权 5 条指令单元测试完成，开始功能测试
2021.01.04	10:00-24:00	朱海龙	协助队友调错，尝试封装 axi 接口和修改体系结构实验的 cache(未成功)
2021.01.04	13:30-24:00	屈湘钧	完成功能测试 3 的内陷 & 特权，完成全部功能测试 89 条
2021.01.05	9:00-24:00	朱海龙	axi 接口连接，修复部分 bug，完成 52 条功能测试
2021.01.05	13:30-24:00	屈湘钧	学习 axi，sram soc 上板子成功
2021.01.06	9:00-4:00	朱海龙	axi 接口连接，修复异常处理的 bug，完成 axi 的 57 条功能测试
2021.01.06	13:30-24:00	屈湘钧	axi 处理异常 bug，axi 性能上板
2021.01.07	9:00-4:00	朱海龙	axi 上 cache，修复异常，完成性能仿真
2021.01.07	13:30-24:00	屈湘钧	axi 的 cache 上板，完成性能仿真
2021.01.08	10:00-24:00	屈湘钧	写报告
2021.01.08	10:00-24:00	朱海龙	写报告

表 13: 李颀琳工作日志

日期	时间	人员	工作内容
2020.12.28	9:00-12:00	李颀琳	看了吕学长的视频,大致了解了整个设计流程和步骤
2020.12.29	9:00-12:00	李颀琳	复习计组 4 实验内容,回忆数据通路相关知识点,了解硬综设计需求
2020.12.30	14:00-17:00	李颀琳	开始根据 b 站视频以及参考资料写逻辑运算指令
2020.12.30	19:00-24:00	李颀琳	完成逻辑运算指令,并且通过单元测试
2021.12.31	9:00-12:00	李颀琳	完成移位指令代码编写并通过测试
2021.12.31	14:00-17:00	李颀琳	根据参考学习 HILO 寄存器内容,完成编写 alu 模块
2021.12.31	19:00-22:00	李颀琳	完成数据移动指令代码编写并通过测试
2021.01.01	14:00-22:00	李颀琳	编写算术运算指令,除除法指令都通过测试
2021.01.02	10:00-22:00	李颀琳	除法器错误修改,通过除法指令单元测试
2021.01.03	10:00-22:00	李颀琳	编写访存代码,实现读写的两个模块,并处理大小端问题
2021.01.04	10:00-15:00	李颀琳	修改跳转 b 指令在运行中出现的问题,修改 hazard 模块
2021.01.04	18:00-22:00	李颀琳	修改访存代码中遇到的问题,重新生成 IP 核,通过访存单元测试
2021.01.05	10:00-17:00	李颀琳	阅读新参与项目代码,熟悉已经实现的架构以及内容
2021.01.05	18:00-22:00	李颀琳	协助组员做 SRAM 上板的 debug 工作
2021.01.06	13:00-17:00	李颀琳	参与 AXI cache 的连线以及测试
2021.01.06	19:00-24:00	李颀琳	协助替换了源代码中的除法器
2021.01.07	13:00-24:00	李颀琳	协助队友完成 AXI 的上板和性能仿真
2021.01.08	10:00-24:00	李颀琳	写报告

3.2 测试日志

表 14: 测试日志

时间	人员	测试
2020.12.27	屈湘钧	logic8 条指令部分单元测试完成
2020.12.28	朱海龙	shift6 条指令部分单元测试完成
2020.12.28	朱海龙	move_hilo 四条指令单元测试完成
2020.12.29	屈湘钧	运算指令 14 条单元测试完毕
2020.12.29	朱海龙	分支指令 branch 单元测试
2020.12.30	朱海龙	跳转指令 jump 单元测试
2021.01.01	屈湘钧	访存指令 8 条完成单元测试
2021.01.02	屈湘钧	访存指令冒险单元测试完成
2021.01.03	屈湘钧	内陷 & 特权 5 条指令单元测试完成
2021.01.03	朱海龙	功能测试 52 条完成 (trace1,trace2)
2021.01.04	屈湘钧	功能测试特权内陷指令通过 (trace3)
2021.01.04	屈湘钧	功能测试 89 个测试点全部通过
2021.01.05	朱海龙	axi 接口功能测试 52 条完成 (trace1,trace2)
2021.01.05	屈湘钧	sram 版本 57 条指令功能测试上板成功
2021.01.06	朱海龙	axi 接口功能测试 89 全部完成 (trace1,2,3)
2021.01.07	屈湘钧	axi 接口性能测试全部通过
2021.01.07	朱海龙	axi 接口性能测试全部通过
2021.01.07	屈湘钧	axi 加 cache 上板测试通过 5 个点

表 15: 李颀琳测试日志

时间	人员	测试
2020.12.30	李颀琳	通过逻辑运算 8 条指令部分单元测试
2020.12.31	李颀琳	通过移位运算 6 条指令单元测试
2021.12.31	李颀琳	通过数据移动 4 条指令单元测试
2021.01.02	李颀琳	通过算术指令单元测试
2021.01.04	李颀琳	通过跳转指令单元测试
2021.01.04	李颀琳	通过访存 8 条指令单元测试
2021.01.04	李颀琳	通过访存冒险指令单元测试
2021.01.05	李颀琳	协助尝试 sram 版本 57 条指令上板测试
2021.01.07	李颀琳	协助 axi 加 cache 上板测试通过 5 个点

3.3 错误记录

3.3.1 错误 1

(1) 错误现象:alu 无法计算出正确的结果
(2) 分析定位过程: 查看目前 alu 计算的结果,分析这是 alu 中的哪一种指令计算,发现是 alucontrol 错误,进一步向前推,观察是否 alu_dec 译码错误,发现不是。分析 alucontrol 信号在通路中的流动过程,结果发现由于现在版本的 alucontrol 是 8 位的,计组 4 的 aluop 位数小于它,我们直接复用了原本的寄存器,且在寄存器之间传递时忘记修改位数,导致传到 alu 的是错的 alucontrol
(3) 错误原因:alucontrol 的位数在寄存器传递时位数错误
(4) 修正效果: 修改寄存器传递位数,结果 alu 计算正确了

3.3.2 错误 2

(1) 错误现象: 访存指令完成后,测试计组 4 代码发现 sw 写入有问题
(2) 分析定位过程: 查看仿真图中写入时是否有使能信号成功,然后查看写入的地址是否正确,然后确认地址错误。
(3) 错误原因: 最终查询发现计组 4 的 sw 写入地址不是 4 的整数倍
(4) 修正效果: 存储访问正常

3.3.3 错误 3

(1) 错误现象: 读 load 指令每次读出的都错误,一直和前面写入指令时有读出。
(2) 分析定位过程: 查看仿真图时发现读指令后,mem 的读出值一直保持不变,猜测是写入未有使能信号的原因。
(3) 错误原因: 误认为 we 信号只用写使能,maindec 中的 writedata 信号置为 1 即可,但是读信号也需要 we 使能为 1。
(4) 修正效果: 更改 man_dec, 读写在 blockmemory 中都要加 en 信号为 1,读出正确。

3.3.4 错误 4

(1) 错误现象:blockmemory 的地址 load 和 store 与计组四的有区别,用 lb、lh、sb、sh 的非整字读写,其虽然用了四位的 we 表示写入字节位置,但是其写入地址 01、02 等对应的字位置,不是字节位置,所以地址 01、02 表示的是两个不同的 32 位地址,导致写入时写到了两个不同的地址位置。
(2) 分析定位过程: 在写入信号后立即加一个 load 的指令,看仿真图中 load 的结果,发现写同一字的位置结果是不同的,猜测是需要将地址右移两位才行;但找了大多数

的参考代码,发现其都为右移,最终没有方法,所以尝试一下吧。

(3) 错误原因: 计组 4 和现在的读写机制不同,需要加片选信号。

(4) 修正效果: 在 blockmemory 地址信号为,用 aluout[x:2] 的第三位开始,忽略最低两位,则 01 或 02 或 03 或 00 是表示同一 32 位的不同字节偏移。

3.3.5 错误 5

(1) 错误现象: lb 1, 0*x*0(0)、lb 1, 0*x*1(0) 两个 load 指令连着时,由于写入的 reg 相同,则有 stall 一次,但其并无数据冒险。

(2) 分析定位过程: 发现仿真时有 lh 和 lhu 的两个连续 load 信号,其写入寄存器都是 3 号,结果中间莫名的有 00000000 的 nop 类型指令结果,故查看 hazard 模块信号仿真,发现确实有 stall 信号传出。

(3) 错误原因: 错误的使用的数据冒险暂停机制,未排除 load 接着 load 的情况

(4) 修正效果: 在 hazard 模块中判断 lwsatll 信号时加上与上非 memtoregD 信号表示两个连着的 load 指令无冲突。

3.3.6 错误 6

(1) 错误现象:load 指令后接 jr 或者 beq 一类指令,如果出现数据冒险,原来的数据前推是不支持 memory 读出的结果前推,导致在 Decode 阶段会出现 jr 和 beq 的数据冒险。

(2) 分析定位过程: 分析 hazard 模块时,考虑以上 lwlw 连着 load 指令,突然想到这个好像没处理,然后看电路图的连线,确实没有 mem 读出后的前推连线。

(3) 错误原因: 未考虑 jr 或者 beq 在 decode 阶段需要数据冒险的前推

(4) 修正效果: 对于 beq 类型的指令,由于我们已经实现了分支预测机制,可以在 D 阶段预测,E 阶段判定预测是否正确,所以在 load 时,先进行一个 lwstall,然后在 alu 阶段进行了一个前推,不会影响我们的判定预测是否正确的模块。但对于 jr 指令,是在 Decode 阶段需要 rs,所以前推 load 的数据是不可行的,故 stall 两次等到 load 的 w 阶段写会正常。lw 后解 beq 和 jr 均完全正确,延迟槽也正常执行

3.3.7 错误 7

(1) 错误现象: 在增加上 cp0 寄存器,需要异常的 clear 所有阶段的寄存器,但修改代码后发现运行 lw 后 jr 指令错误,未正确跳转,只执行了一个 stall。

(2) 分析定位过程: 定位 hazard 模块定位二次 stall 的判断条件发现 writeregW 一直为 00,然后查看代码 writereW,发现其出现的其中流水线寄存器位置信号写错了,en 位置一直写的 0.

(3) 错误原因: 流水线寄存器位置信号写错了,en 位置一直写的 0

(4) 修正效果: 查看和修改所有改了的寄存器 en 为 0 的修改为 1,lw 后解 beq 和 jr 均完全正确,延迟槽也正常执行

3.3.8 错误 8

(1) 错误现象: 在功能测试时,sub 总是错误。

(2) 分析定位过程: 将错误的代码拿出用 mars 编码得到二进制执行代码,自己写一个 coe 文件取加载执行,发现 sub 减的时候触发了加法的溢出,实际减法溢出未做。

(3) 错误原因:sub 溢出计算错误

(4) 修正效果: 将减法计算变更为加法计算,然后复用加法的溢出判断;减法成功

3.3.9 错误 9

(1) 错误现象: 进行对象 3 测试时发现跳入 bfc380 执行异常处理必定会 mfhi,但是 mfhi 是 xxx,结果错误。

(2) 分析定位过程: 查看 hilo 相关寄存器发现异常之前未执行过 hilo 相关指令,则未初始化 hilo

(3) 错误原因:hilo 寄存器未初始化,若第一条指令未 mf 则出现问题

(4) 修正效果:hilo 器件中 initial 初始化,同时 rst 也需要初始化为 0 解决效果:测试通过,无报错

3.3.10 错误 10

(1) 错误现象: 进行对象测试时发现 systemcall 指令接 div 指令,异常跳转到 bfc380 未正确跳转 pc

(2) 分析定位过程: 查看 pc 模块的信号,发现 pc_in 是正确变化为 bfc380,异常信号也出现,但是 pc_out 却两个周期相同,则出现了 stall。发现 div 的 stall 干扰了 systemcall 的 pc 处理跳转。

(3) 错误原因: 异常处理指令后接 div 指令未给你暂停信号解除,导致流水线被除法优先暂停

(4) 修正效果: 在 pc 的 stall 信号位或 exceptionoccur 信号,exceptionoccur 时,pc 正常使能测试通过

3.3.11 错误 11

(1) 错误现象: 由于我们的 hilo 放在 alu 中,导致每次异常的时候,没有保持 hilo 的值不变,被异常指令后的指令改变了 hilo。

(2) 分析定位过程: 查看 hilo 相关寄存器发现在前一条指令执行 mul 时改变了
(3) 错误原因: 异常指令之后执行的指令在流水线中未完全阻止,为给 hilo 写入加暂停功能
(4) 修正效果: 用 always@(clk) 来触发,加入 exception 的信号来使 hili=hilo 不变,测试通过

3.3.12 错误 12

(1) 错误现象:cp0 中保存的 pc 错误值不对
(2) 分析定位过程: 查看 cp0 寄存器的输入发现对应的 pc 值不正确,是由于判断 pc 指令错误的 pc 为 pc_in, 而传入 cp0 的是 pc_out,两者不是同一个周期指令,
(3) 错误原因: 判断错误的 pc 地址和标记的 pc 不匹配
(4) 修正效果: 判断 pc 错误的指令改变为判断 pc_out,测试通过

3.3.13 错误 13

(1) 错误现象:jr 的跳转 pc 地址错误,引发 pc 地址错误例外,在异常处理代码一长串后,mfc0 的结果错误
(2) 分析定位过程: 从 jr 的例外开始查看 cp0 中的所有变化地址,对比别人组过了的这个指令处理发现没有将 pc 存入 cp0 中 bad_addr。即此 cp0 的 epc 值在传入时,即异常发生时错误,然后查看到错误发生时事 jr 的跳转地址有误,需要保存此 pc 值到 epc,但是未存入 cp0 中 bad_addr
(3) 错误原因: 为了解 pc 跳转地址错误需要同时写入 epc 和 bad_addr 两个寄存器。
(4) 修正效果: 在 datapath 中加一个选择,当出现 pc 地址错误时,bad_addr=pcM,存储 cp0 的 bad_addr,然后测试通过

3.3.14 错误 14

(1) 错误现象: 一直无法读取 sram 的 trace 文件中的 ref 信号,这些信号一直是 X
(2) 分析定位过程: 一个信号一个信号的查看,发现 sram 的指令使能没有连接,是 Z
(3) 错误原因: 信号为 Z,未连接
(4) 修正效果: 将指令读取设置为 1'b1。解决效果:通过

3.3.15 错误 15

(1) 错误现象: 使用类 sram 转 axi 的 interface 连接 axi 接口时,一直无法读取 axi 的数据,只有 addr_ok,没有 data_ok

(2) 分析定位过程: 打开 axi interface 的波形图,与体系结构 cache 实验的波形图 (也是 axi 接口) 每个信号一个一个去比较,发现 axi 的 reset 在使用时是高电位,说明连 cpu 时,reset 需要置反,但是连 axi_interface 不需要置反

(3) 错误原因:reset 的使用电平错误

(4) 修正效果: 直接把 mycpu_top 的 reset 传给 axi_interface 的 reset 信号。信号出现正常。

3.3.16 错误 16

(1) 错误现象: 使用类 sram 转 axi 的 interface 连接 axi 接口时,addr_ok 信号一直处于高电位,且 rcv 信号一直是 X 状态

(2) 分析定位过程: 打开 axi_interface 的波形图,与体系结构 cache 实验的波形图 (也是 axi 接口) 每个信号一个一个去比较,发现把 cpu 的 sram 转换为类 sram 的转换桥,rcv 信号需要一个激发,否则一直处于 X 状态

(3) 错误原因: 未激发信号

(4) 修正效果: 将 inst_sram_en 改成随 reset 变化的形式,在时钟下降沿进行变化。最后信号仿真正常。

3.3.17 错误 17

(1) 错误现象: 连完 axi 后,无法处理除法结束,除法无法停止,导致 cpu 一直处于除法状态

(2) 分析定位过程: 查看 div_stall 信号,发现 div_stall 所需的 flush_endE 信号处于 Z 状态,由于之前重整 hazard 时,未处理 flush_endE 信号,而是直接删除了,所以导致未连接

(3) 错误原因: 重整代码丢失 div_stall 的信号。

(4) 修正效果: 把 flush_endE 改成 hazard 中的 flushE,然后除法测试正常。

3.3.18 错误 18

(1) 错误现象: 在进行 axi 功能测试时,pc 一直和 trace 文件的 pc 对不上

(2) 分析定位过程: 查看波形图,发现 debugpc 不变时,refpc 一直在变,推测是 debugwen 信号在 stall 时一直是 en 状态,所以测试不断读取 trace 文件,导致 pc 和 trace 对不上

(3) 错误原因:debugwen 信号在 stall 时一直为 1,不停读取 trace

(4) 修正效果: 将 debugwen 信号修改为 {4{regwriteW & ~stallW}}; 这样在 stall 的时候就不会一直读 trace 文件,测试通过

3.3.19 错误 19

(1) 错误现象: 在 axi 软件中断功能测试时,debug 的 data 一直不对
(2) 分析定位过程: 查看波形图,发现在 M 阶段时,写入了 cp0,但是并未触发 exceptdec 发现异常,这是因为对 cp0 的写是 clk 激发的,写完后将 cause 寄存器的值传入 exceptdec 译码,译码后的结果传回 cp0,但是此时已经错过时钟沿,导致无法二次激发,无法当时处理软中断,所以在这一阶段不触发异常,等到后面触发异常的时候,地址已经不对了
(3) 错误原因: 软件中断无法在 M 阶段一周期内进行写入 cp0 的 cause 和检查出错并处理
(4) 修正效果: 在通路中修改异常的部分,如果发现指令在修改 ep0 的 cause 寄存器,将指令想要修改的值先计算出来,并传入 exceptdec 去进行异常译码,同时对 cp0 的写正常执行,这里由于我们将两周期缩为一周期,所以需要将 epc 的值写为 PCE 而不是 PCM,这样经过异常译码后的异常状态就会传入 cp0,此时正常的对 cp0 的写也会进行,由于 cp0 内部语句先进行写后进行异常处理,所以刚好可以正常进行异常处理,发现软中断,进行正确的处理

3.3.20 错误 20

(1) 错误现象: 在 axi 软件性能测试仿真时,即使更换了 coe 文件,测试时还是原来的 coe 文件
(2) 分析定位过程: 每次重新启动打开 vavado 工程,更换不同的 ceo 文件,都可以正常执行,但是再更换 coe 文件就还是执行的第一次的 ceo 文件内容。
(3) 错误原因:ceo 文件并未真正的完全替换
(4) 修正效果: 每次替换 coe 文件后,选择不同的 global 或 local 的综合方式,然后再点击 run simulation 而非仿真页面的重置按钮即可。最终 coe 文件替换成功

4 设计结果

本项目最终得到了 myCPU 整个文件的代码,其构成了整个数据通路,然后在课程提供的各种接口文件和 vivado 环境下能够实现 57 条指令(逻辑、移位、数据移动、算术、分支跳转、访存、内陷和特权指令)的独立测试和通过基于 sram 接口环境的 89 个功能测试点,并在 axi 接口环境下可以通过 89 个功能测试点和 10 个性能仿真测试,最终上板能实现 bubble_sort,quick_sort,select_sort,stream_copy,stringsearch 五个性能测试,主频能达到 60MHZ 以上。

4.1 设计交付物说明

4.1.1 目录层次

所提交的 mycpu 文件中包含如下的代码文件,其功能都写在注释中,其总的实现是对应 axi 接口下的带 cache 和分支预测的 cpu 设计文件。

```
.
‘-- myCPU
    |-- adder.v # 加法器
    |-- alu_dec.v # alu编码译码
    |-- alu.v # 计算单元
    |-- branch_predict_global.v # 全局分支预测
    |-- branch_predict_local.v # 局部分支预测
    |-- bridge_1x2.v # 1*2转接桥，data数据转化为ram和config数据
    |-- bridge_2x1.v # 2*1转接桥，把ram数据和config数据转化为data数据
    |-- cache.v # i_cache和d_cache的封装
    |-- compete_predict.v # 竞争分支预测
    |-- controller.v # 控制信号译码控制器
    |-- cp0_reg.v # cp0寄存器
    |-- cpu_axi_interface.v # cpu的axi接口
    |-- d_cache.v # 数据存储器cache
    |-- defines2.vh # 指令译码定义
    |-- defines.vh # 指令中间编码定义
    |-- div_radix2.v # 除法模块
    |-- d_sram2sraml.v # 数据存储器cache的sram接口转类sram接口
    |-- exceptiondec.v # 异常译码模块
    |-- flopenrc.v # 使能触发器
    |-- flowmips.v # 数据通路文件
    |-- hazard.v # 冒险处理模块
    |-- i_cache.v # 指令cache
    |-- instdec.v # 指令assic码译码模块
    |-- i_sram2sraml.v # 指令存储器cache的sram接口转类sram接口
    |-- main_dec.v # 控制信号译码
    |-- mmu.v # 地址转换
    |-- mux2.v # 二选一选择器
    |-- mux3.v # 三选一选择器
    |-- mycpu_top.v # cpu模块的顶层接口转换
    |-- mycpu.v # cpu模块
    |-- my_mul.v # 乘法器
    |-- pc.v # pc寄存器
    |-- ReadData_handle.v # 读数据存储器的数据处理模块
    |-- regfile.v # 寄存器堆
    |-- signext.v # 有符号扩展
    |-- sl2.v # 左移两位
    ‘-- WriteData_handle.v # 写存储器的数据与信号处理模块
1 directory, 37 files
```

4.1.2 仿真

做 axi 的性能仿真,需要将如上的 cpu 代码放入课程提供的性能测试 vivado 功能的环境中,然后启动 vivado 工程,用 add sources 添加这些代码到工程中, 然后在文件 source 中选择 axi_ram,双击打开设置,然后选择 otheroptions 中选择 coe 文件,在 soft 文件夹中选择第一个性能测试样例,其后一次如下步骤选择其他仿真样例。其步骤图如下

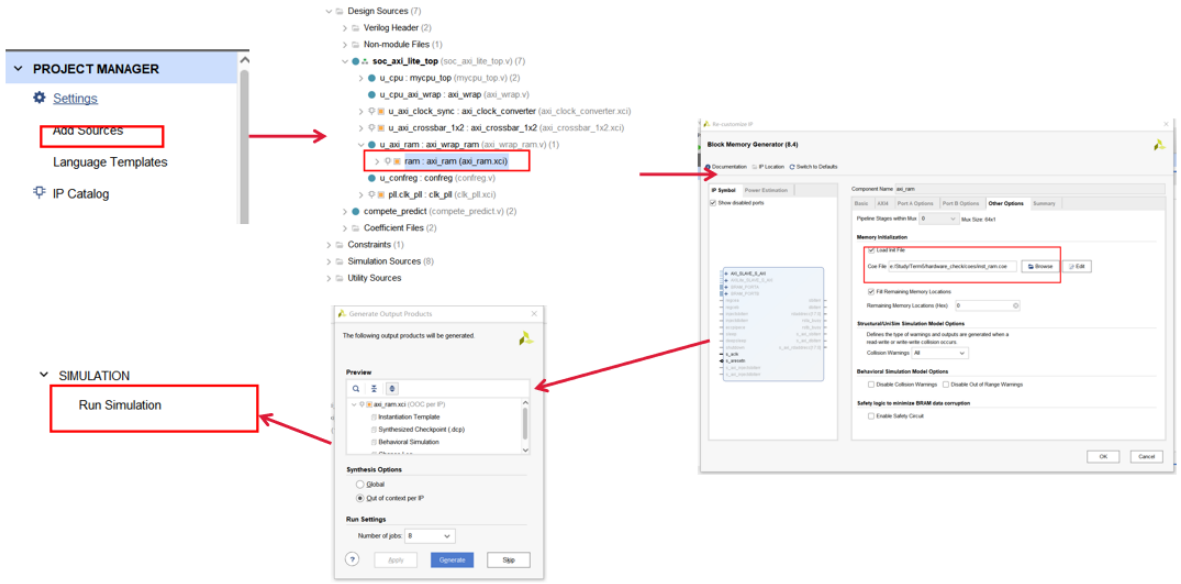

图 15: axi 仿真过程

4.1.3 综合

在仿真完成后，没有任何错误，即可进行综合看有无时序错误等，然后在综合后可查看一下报告如时序报告和 warning 提示等。其综合步骤如下为在左栏 project manager 中先选择 run synthesis, 然后依次选择提示的选项，完成 run 后点击 run implementation，继续按照提示点击即可完成综合。

4.1.4 上板

在综合完成后，即可点击 project manager 中的 generate bitstream 生成 bit 流，然后根据提示点击生成完成后，点击 open hardware manager，然后将板子连接上电脑，然后自动连接板子，右键选择 program device 即可上传 bit 流到板子，板子即可开始工作。

4.2 设计演示结果

4.2.1 sram 功能仿真

在添加完成 57 条指令后，并接上 sram 接口，如下图是 sram 接口版本通过 89 个功能点测试 Tcl console 截图。

4.2.2 axi 带 cache 的功能仿真

然后再将 cpu 加 cache 后接上 axi 接口，进行功能测试，其通过截图如下。

```
       [1632000 ns] Test is running, debug_wb_pc = 0x00000000
----[1633385 ns] Number 8'd85 Functional Test Point PASS!!!
       [1642000 ns] Test is running, debug_wb_pc = 0xbfc004c0
----[1647655 ns] Number 8'd86 Functional Test Point PASS!!!
       [1652000 ns] Test is running, debug_wb_pc = 0xbfc46b14
----[1661885 ns] Number 8'd87 Functional Test Point PASS!!!
       [1662000 ns] Test is running, debug_wb_pc = 0x00000000
       [1672000 ns] Test is running, debug_wb_pc = 0xbfc27f70
----[1676115 ns] Number 8'd88 Functional Test Point PASS!!!
       [1682000 ns] Test is running, debug_wb_pc = 0xbfc00678
----[1690355 ns] Number 8'd89 Functional Test Point PASS!!!
================================================================
Test end!
----PASS!!!
$finish called at time : 1691005 ns : File "E:/Study/Term5/hardware/MIPS_HARDWARE/MIPS_HARDWARE/soc_sram_func/testbench/mycpu_tb.v" Line 261
run: Time (s): cpu = 00:01:24 ; elapsed = 00:01:15 . Memory (MB): peak = 1469.824 ; gain = 0.000
```

图 16: sram 接口的 89 个功能测试点仿真通过

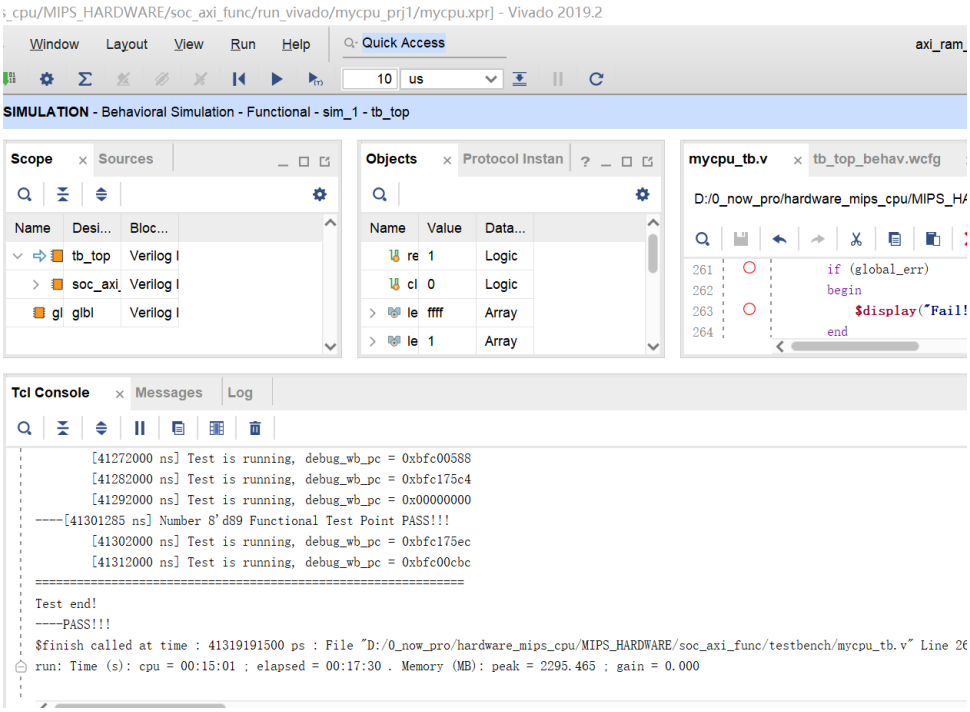

图 17: axi89 个测试点的功能仿真通过

4.2.3 axi 带 cache 的性能仿真

功能仿真通过后,将有 cache 的 mycpu 代码放入 axi 的性能仿真环境,其仿真结果如下图。

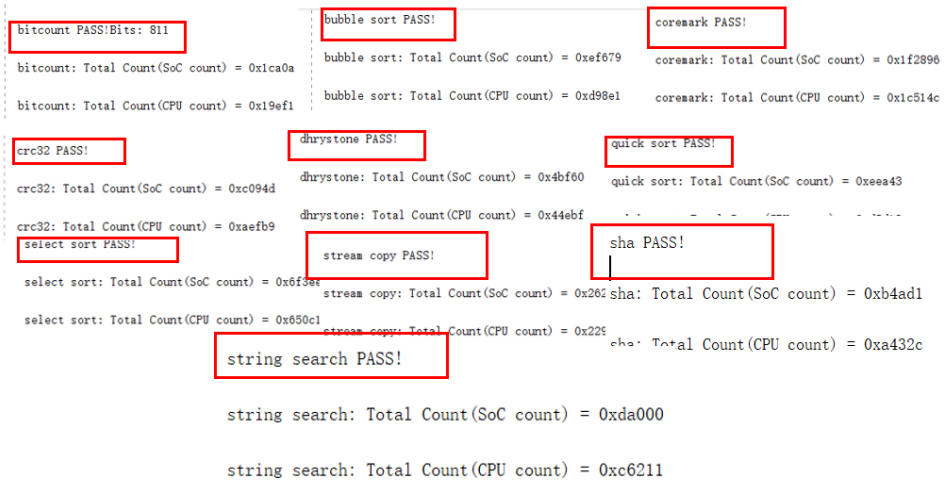

图 18: axi 性能仿真通过

4.2.4 axi 带 cache 的性能测试上板

然后综合生产 bit 流文件,上传到 N4ddr 板子上,最终实现如下的五个测试点(bubble_sort,quick_sort,select_sort,stream_copy,stringsearch)上板测试。

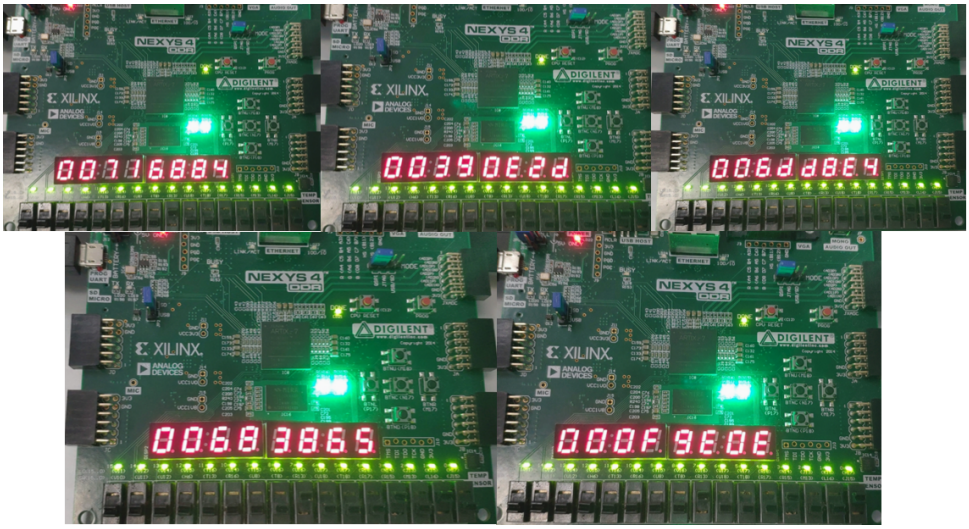

图 19: n4ddr 上板五个测试通过

4.2.5 axi 带 cache 的性能测试得分

最终的得分为 1.251 分,其得分表格计算如下

序号	测试程序	myCPU	gs132	Tgs132/Tmycpu	性能分	1.251
		上板计时 (16 进制)	上板 (16 进制)			
		数码管显示	数码管显示			
cpu_clk : sys_clk		50MHz : 100MHz	50MHz : 100MHz	-		
1	bitcount		13CF7FA	0.1		
2	bubble_sort	716884	7BDD47E	17.47518192		
3	coremark		10CE6772	0.1		
4	crc32		AA1AA5C	0.1		
5	dhrystone		1FC00D8	0.1		
6	quick_sort	6dd8e4	719615A	16.54461652		
7	select_sort	390e2d	6E0009A	30.84726682		
8	sha		74B8B20	0.1		
9	stream_copy	f9e0e	853B00	8.530898816		
10	stringsearch	683865	50A1BCC	12.37866991		

5 参考设计说明

5.1 指令 asscii 码译码器引用

本项目中为方便调试,引用了题目提供的 asscii 码译码相关的文件

5.2 乘法器引用

本项目中乘法器引用了 https://github.com/14010007517/2020NSCSCC.git 中的乘法器 1.0 版本

5.3 除法器引用

本项目中除法器引用了 https://github.com/14010007517/2020NSCSCC.git 中的除法器 1.4 版本,在 axi 版本中引用了 https://github.com/barryZZJ/Hardware_Design.git 的除法器。

5.4 字节写入读出信号译码器件参考

对于 sel 信号和 mem 读出字节的选中引用了
https://github.com/lingcraft/HardwareIntegratedDesign.git 中的 memsel.v 文件设计

5.5 cp0 的错误译码部分参考

对于 cp0 的译码部分结构与信号参考了
https://github.com/lingcraft/HardwareIntegratedDesign.git 的 exceptiondec.v 文件

5.6 sram 转类 sram 接口和 axi 桥的参考

我们小组在连接 axi 时,使用了袁福焱学长的硬综讲解 2ppt 中的 sram 转类 sram 接口,并在转完 sram 后用来龙芯的类 sram 转 axi 的桥来连接 axi 环境。

5.7 cache 连接的借鉴

我们小组先修了体系结构,并完成了 cache 实验,这里参考了 cache 实验中使用 bridge 先分开 ram 和 confreg 的模块 bridge1X2 和 bridge2X1 来连接 cache。

6 总结

在本次硬件综合设计项目中,我们小组体验了一次相对完整的 cpu 设计,并尝试性的搭建了 soc_sram 和 soc_axi 环境,第一次了解到 axi 这种一周期无法读出数据且带有随机延时的系统环境。我们小组原本有两人,进行了基本的 cpu 的设计和 sram 的搭建,1 月 5 日小组中加入了新成员李颀琳,大家一起搭建了 axi 环境,接上了 cache,完成了最后的 cpu 设计,基本把本次硬综项目必选项目做完了。

在这个过程中,我们遇到了很多困难,进行了很多次的 debug,在遇到无法解决的问题或者不清楚的概念时,我们一些网上的博客和 github 的代码资料,并在同学那里获得了帮助,最终才完成了这个并不完善的 cpu,很遗憾,这个 cpu 并不能通过所有的性能上板测试,仅可以通过功能测试和性能测试的仿真,但是,我们在设计和编码的过程中,对于计算机的基本知识更加了解了,例如我们组的屈湘钧同学在设计异常处理模块时主件了解到 cp0 寄存器的各项功能,也在 debug 中对各个寄存器熟悉,并联系操作系统的中断处理收获更多。这样的新认识和对旧的知识的回顾让我们感觉过去书本上的知识生动了起来。

虽然硬件综合设计的这几天很累很难熬,但是回想起来,做 cpu 其实还挺有趣的。我们最终的设计结果还可以,让我们还有一点点的成就感,尤其是看到自己的 cpu 的指令一条一条的增加时,感觉自己真的做了一点有意义有价值的事。

7 供同学们吐槽之用。有什么问题都可以直接写在这。

7.1 朱海龙

希望指导能多一点。

vivado 对写代码的帮助好少,无法动态提示,甚至对于没有连的信号 Z 都不主动提示未

连接,甚至未定义的变量也可以使用,而不是提示 notdefined 啥的。
vivado 的 bug 好多,日常崩溃:(

7.2 屈湘钧

cp0 的相关知识相对普通指令的处理有一些陌生,在 debug 过程中容易由于不知道 cp0 的有些知识而陷入僵局。此外,vivado 的各种玄学让我们浪费大量时间,上板不成功,无法进一步的挑战性能。

7.3 李颀琳

本次硬件综合设计,让我从最基本的原理出发,了解了一个完整的 CPU 的架构与设计,并且也通过写代码和 debug,看每一个信号的波形图找错体验了硬件设计的特色。中途遇到了很多困难,找不出 bug、想不通设计,昼夜难眠梦里都是都是红绿蓝色的线,但是都通过参考和学习找到了合适的解决方案,这也很锻炼人的学习能力和耐心。

8 参考文献

参考文献

[1] Hennessy and JohnL. *Computer Architecture A Quantitative Approach:* 计算机体系结构量化研究方法. China Machine Press, 2002.

[2] 姚永斌. 超标量处理器设计. 清华大学出版社, 2014.

[3] 雷思磊. 自己动手写 *CPU*. 电子工业出版社, 2014.