

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： 信息安全 1 班

年 级： 2021

姓 名： 刘静婷 皮悦颖 邓茜 覃倩瑶

学 号： 20214816 20214600 20215186 20215213

完成时间： 2024 年 1 月 12 日

成 绩：

指导教师： 吴长泽

重庆大学教务处制

综合设计指导教师评定成绩表

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

指导教师评定成绩：

指导教师签名：

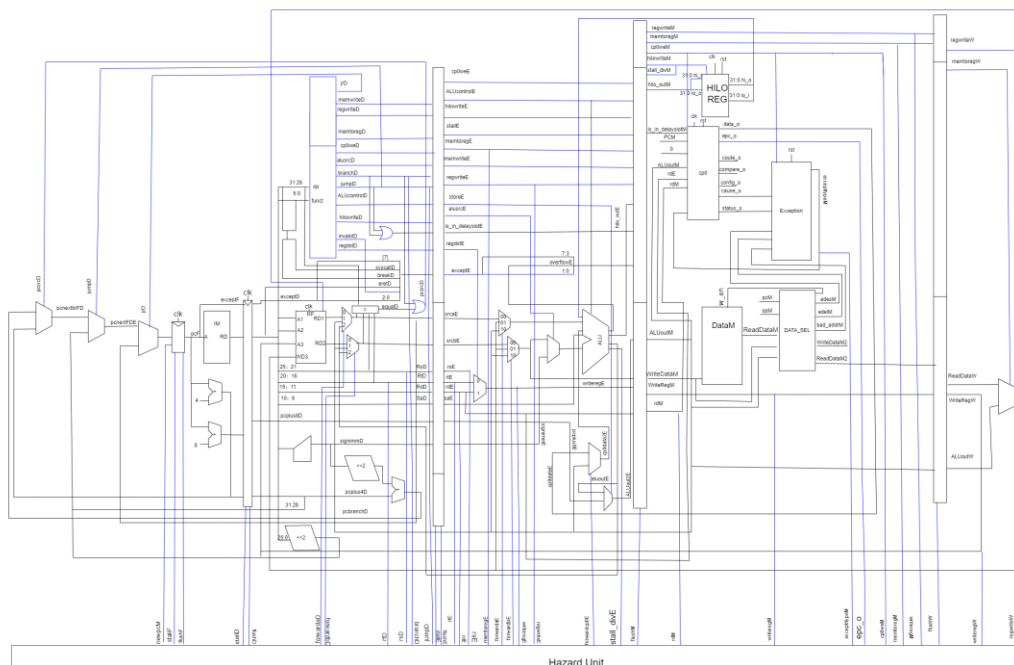
年 月 日

MIPS SOC 设计报告

刘静婷 皮悦颖 邓茜 覃倩瑶

1 设计简介

在这次项目中，我们小组在《计算机组成原理》课程 lab4 的基础上进行了扩展。通过将原本只有少数指令的简易五级流水线 CPU 扩展为支持 57 条指令，并实现了异常处理模块并成功地提升了 CPU 的功能和性能。其中这 57 条指令包括了 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 条访存指令和 3 条特权指令。通过这些指令的支持，能够更加灵活地进行计算和控制流程。此外，小组还实现了基于 SRAM 接口的 SOC 连接，并通过了完整版指令的 Trace 测试和下板测试。为了进一步提升 CPU 的性能，我们将 CPU 封装成 axi 接口，并成功通过了给出的性能测试仿真。这些测试包括了基础测试、延迟槽测试和异常测试，验证了 CPU 在各种情况下的正确性和稳定性。总的来说，本次项目不仅扩展了 CPU 的指令集，还实现了异常处理模块，并通过了多项测试和仿真，这些改进使得 myCPU 在功能和性能上都得到了显著提升。（下图为完整的数据通路图）



1.1 小组分工说明

成员	主要工作内容
刘静婷	数据移动指令、内线和特权指令，连接 SOC，连接 AXI，设计数据通路，功能测试、性能测试
皮悦颖	访存指令、分支指令，连接 SOC、AXI，写透 cache，设计数据通路图，功能测试、性能测试
邓茜	乘法、除法指令，连接 SOC、AXI，设计数据通路图，功能测试、性能测试
覃倩瑶	部分算术指令、逻辑、移位指令，设计数据通路图，功能测试、性能测试

2 设计方案

2.1 总体设计思路

2.1.1 乘法指令

乘法指令分为有符号乘法 `mult` 和符号乘法 `umult`。对于有符号乘法，需要在运算之前，对于乘数中的负数求补，再将得到的数带入进行运算。

```
assign mult_a=((op == `EXE_MULT_OP) && (a[31] == 1'b1)) ? (~a + 1): a;
assign mult_b=((op == `EXE_MULT_OP) && (b[31] == 1'b1)) ? (~b + 1): b;
```

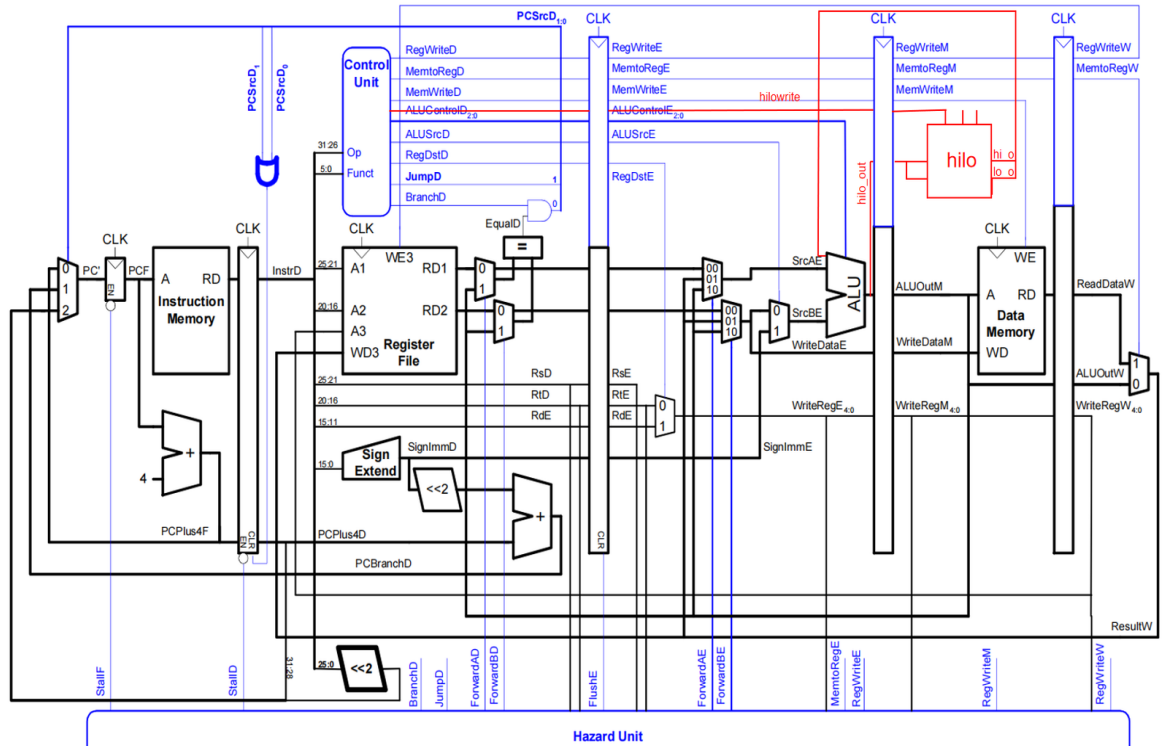
乘法运算我们选择直接采用乘号“*”进行运算，不额外使用乘法器。

```
`EXE_MULT_OP: mult_result <= (a[31] ^ b[31] == 1'b1)? ~(mult_a * mult_b) + 1 : mult_a * mult_b;
`EXE_MULTU_OP : mult_result <= mult_a * mult_b;
```

乘法的运算结果 64 位，alu 原本的输出 `aluout` 为 32 位，所以我们需要将乘法得出的结果传入到 `hilo` 寄存器中。我们在 `alu` 中另设一个 `mult_result` 来存乘法计算出的结果，再将该结果赋值给 `hilo_out`，接着 `hilo_out` 传到下一级，成功将结果传入 `hilo` 寄存器中。

```
always @(*) begin
    case(op)
        `EXE_DIVU_OP, `EXE_DIV_OP: hilo_out<= div_result;
        `EXE_MULT_OP, `EXE_MULTU_OP: hilo_out<= mult_result;
    endcase
end
```

乘法需要在数据通路中添加 `hilo` 寄存器，具体通路改变情况如下图所示：



2.1.2 除法指令

除法指令通过在 alu 内调用资料包内所给的 div 除法器进行运算。

```
div div(clk,rst,signed_div,a,b,start_div,1'b0,div_result,div_ready);
```

除法的结果有余数和商，所以除法也需要将结果传入到 hilo 寄存器中。

```
always @(*) begin
  case(op)
    `EXE_DIVU_OP,`EXE_DIV_OP: hilo_out<= div_result;
    `EXE_MULT_OP,`EXE_MULTU_OP: hilo_out<= mult_result;
  endcase
end
```

除法器中会涉及到 signed_div_i, start_i, annul_i, ready_o。signed_div_i 表示是否有符号，div 时置为 1，udiv 为 0；start_i 表示除法运算是否开始，开始了则为 1，还未开始或者结束了则置为 0；annul_i 在本次实验中直接设置为 0 即可；ready_o 表示除法的结果是否计算出来（准备好），结果尚未计算出则为 0，反之则为 1。

除法无法在一个周期内完成运算，因此我们暂停流水线，等待除法器计算，引入 stall_divE。除法的暂停分为两种：暂停 F、D、E 阶段和暂停所有阶段。但是因为单独暂停 F、D、E 阶段会导致 M、W 阶段数据流动之后影响前推，情况比较复杂不好处理，所以这里采用第二种即暂停整个流水线的方法。

alu 则通过改变上述信号的来控制除法器，类似于状态转换。

为了防止除法器的状态影响到其他指令的运行，我们先将 alu 内除法 sign、start、stall 信号都设置为 0。

```

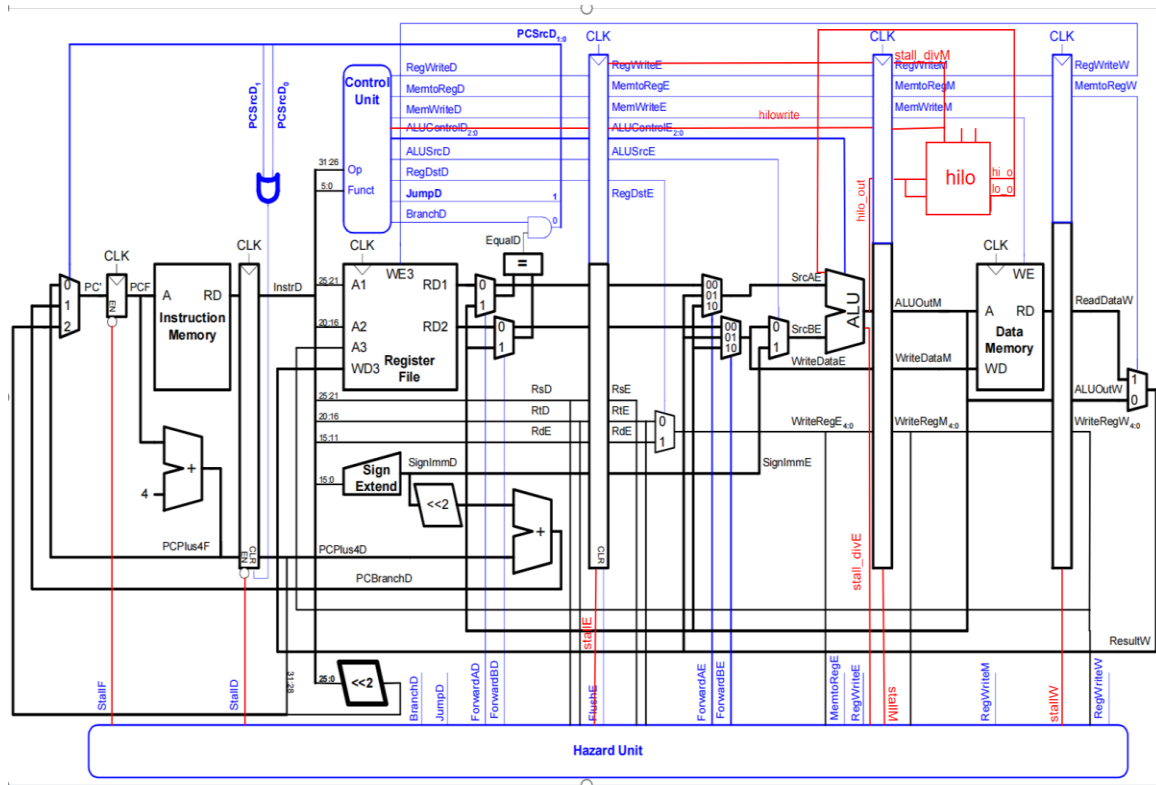
always @(*) begin
    start_div <= 1'b0;
    signed_div <= 1'b0;
    stall_div <= 1'b0;
    case (op)
        .
    end
end

```

sign 在 div 时置为 1, udiv 时置为 0。start、stall 信号取决于 ready, 即除法的结果是否已经准备好: 当 ready 为 0 时, start 为 1 除法器运行, stall 为 1 流水线全线暂停等待除法结果; 当 ready 为 1 时, start 为 0 结果已出除法器停止运行, stall 为 0 流水线继续运行。

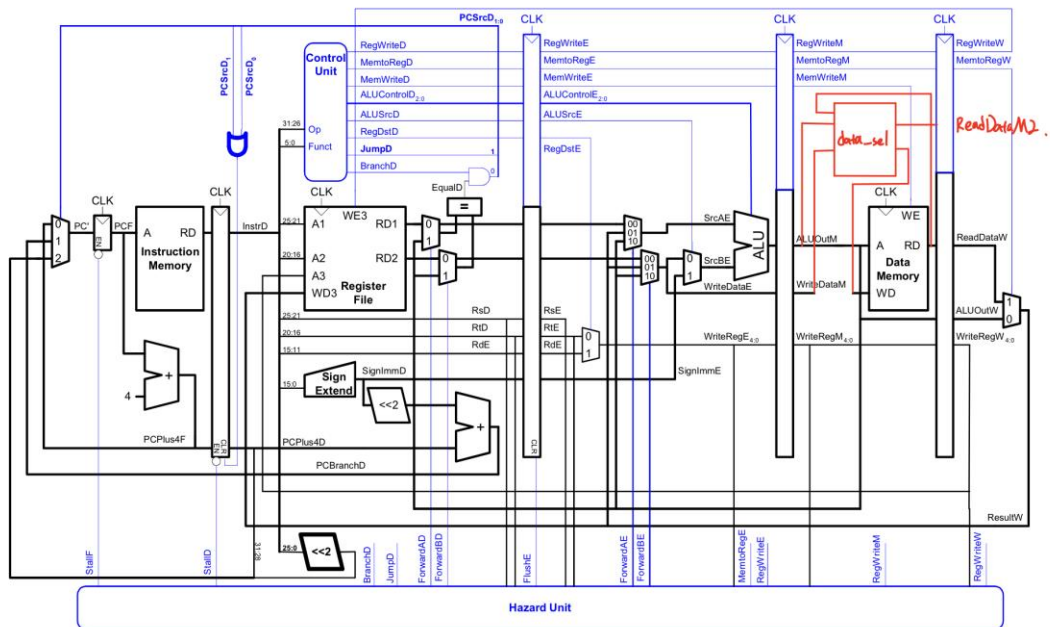
<pre> `EXE_DIV_OP: begin if(div_ready == 1'b0) begin start_div <= 1'b1; signed_div <= 1'b1; stall_div <= 1'b1; end else if (div_ready == 1'b1) begin start_div <= 1'b0; signed_div <= 1'b1; stall_div <= 1'b0; end else begin start_div <= 1'b0; signed_div <= 1'b0; stall_div <= 1'b0; end end end </pre>	<pre> `EXE_DIVU_OP: begin if(div_ready == 1'b0) begin start_div <= 1'b1; signed_div <= 1'b0; stall_div <= 1'b1; end else if (div_ready == 1'b1) begin start_div <= 1'b0; signed_div <= 1'b0; stall_div <= 1'b0; end else begin start_div <= 1'b0; signed_div <= 1'b0; stall_div <= 1'b0; end end end </pre>
---	--

除法需要在数据通路中添加 hilo 寄存器, 并向各阶段触发器添加暂停信号, 具体通路改变情况如下图所示:



2.1.3 访存指令

访存指令 datapath:



访存地址均为 rs 寄存器中的值加上立即数偏移量，故均需启用立即数，同时，根据不同的存取指令，存取数据的长度也不相同，需要在 MEM 阶段增加一个字节选择器，

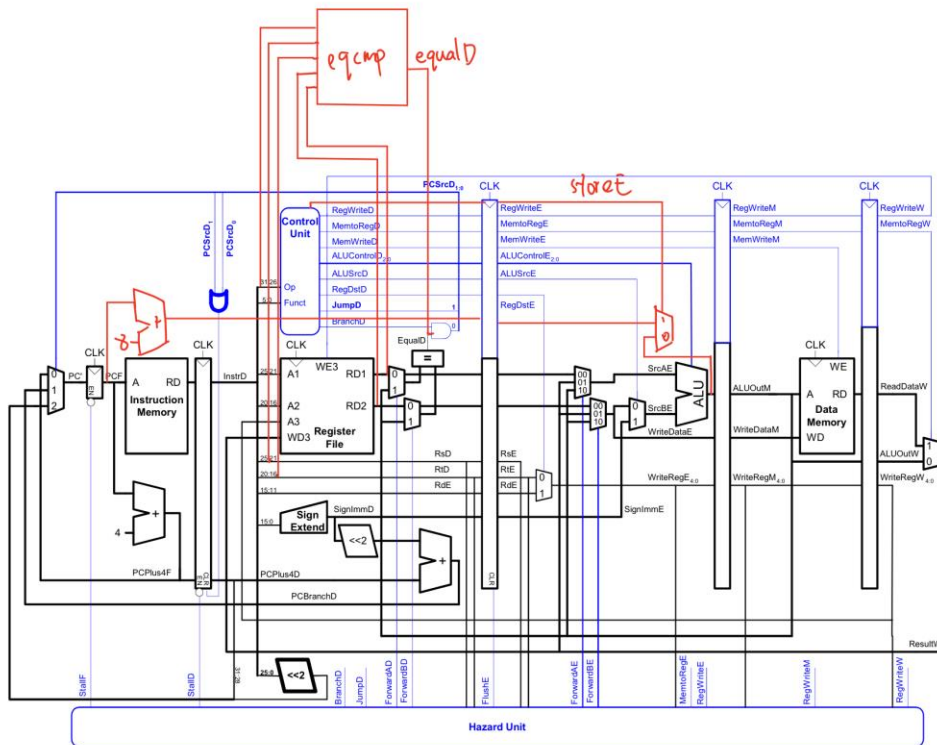
```
mux2 #(32) srcbmux(srcb2E, signimmE, alusrcE, srcb3E);
```

```
data_sel sel_m(pcM, aluoutM, writedataM, readdataM, opM, writedataM2,
readdataM2, w_en, adesM, adelM, bad_addrM);
```

对于取出内存的数，数据长度均为 4 个字节，通过选择器选择不同长度；而对于存入内存中的数，要增添一个长度为 4 位的写使能信号，控制不同长度的数据写入。

2.1.4 跳转分支指令

跳转分支指令 datapath:



增加控制信号 jr(跳转地址在 rs 寄存器中)、store(保存延迟槽指令之后的指令的 PC)

在译码阶段判断当前指令是否是跳转或分支指令，并且通过 eqcmp 判断是否分支，从而改变下一条 pc 值，减少执行多余指令。

```
//next PC logic (operates in fetch an decode) (控制信号决定分支或跳转)
mux2 #(32) pcbrmux(pcplus4F, pcbranchD, pcsrcD, pcnextbrFD); //是否有分支
mux2 #(32) pcmux(pcnextbrFD,
    {pcplus4D[31:28], instrD[25:0], 2'b00},
    jumpD, pcnextFD1); //是否有跳转
mux2 #(32) pcjump(pcnextFD1, srca2D, jrD, pcnextFD); //判断是否是跳转中的jr或jalr指令
pc #(32) pcf(clk, rst, ~stallF, flushF, pcnextFD, newpcM, pcf);
```

如果是 jal、jalr、bltzal、bgezal 指令，还需将 pc+8 存入 31 号寄存器或 rd 寄存器，需要在 EX 阶段增加一个多选器，选择最终写回寄存器的值。


```
mux2 #(32) pc8mux(aluoutE, pcplus8E, storeE, aluout2E);
```

2.1.5 数据移动指令

数据移动指令 MFHI, MFLO, MTHI, MTLO 指令格式及对应机器码如下图所示：

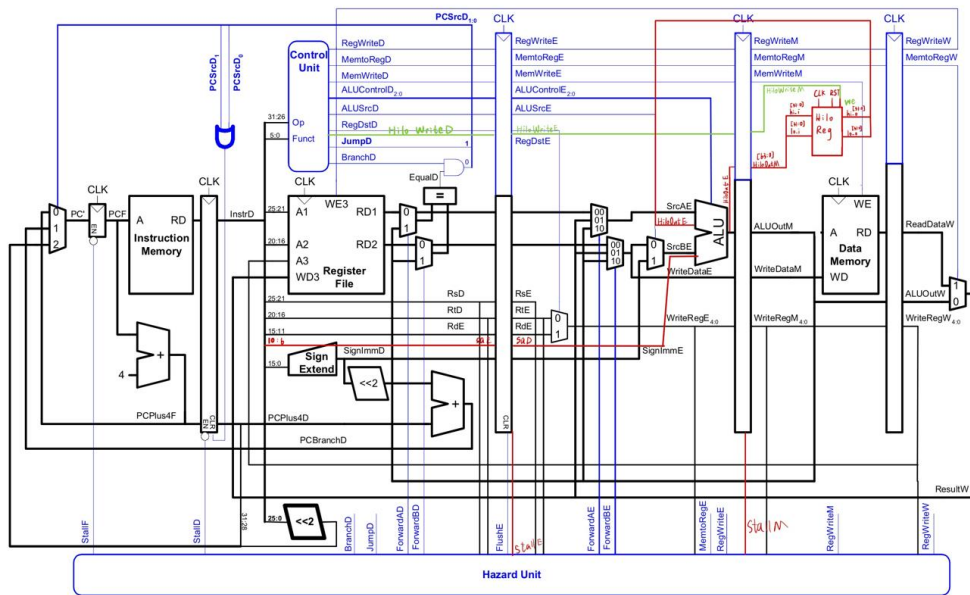
数据移动指令	000000	00000	00000	rd	00000	010000	mfhi rd
	000000	00000	00000	rd	00000	010010	mflo rd
	000000	rs	00000	00000	00000	010001	mthi rs
	000000	rs	00000	00000	00000	010011	mtlo rs

高六位 op 均为 000000，所以在 maindec.v 和 aludec.v 对这四条指令进行译码时需要使用低六位 func 进行联合判断，如下图所示：

```
6'b000000:case(func)
// 数据移动指令
    `EXE_MFHI:alucontrol <= `EXE_MFHI_OP;
    `EXE_MTHI:alucontrol <= `EXE_MTHI_OP;
    `EXE_MFLO:alucontrol <= `EXE_MFLO_OP;
    `EXE_MTLO:alucontrol <= `EXE_MTLO_OP;

6'b000000:case(func)
// 数据移动指令
    `EXE_MFHI:controls <= 12'b1100000000010;
    `EXE_MTHI:controls <= 12'b0000000000110;
    `EXE_MFLO:controls <= 12'b1100000000010;
    `EXE_MTLO:controls <= 12'b0000000000110;
```

为了实现数据移动指令我们首先需要在 cpu 中接入 HILO 寄存器，我们起初设计将其放置在 EX 阶段进行写操作，但是在后续 debug 阶段发现，如果 HILO 寄存器在执行阶段写会导致额外的数据冲突，需要进行冒险处理。查阅参考文献和往届的教学视频后得知，把 HILO 的输入流水一级至 MEM 阶段再进行写操作会规避掉冒险的问题，最终修改过的数据通路图如下：



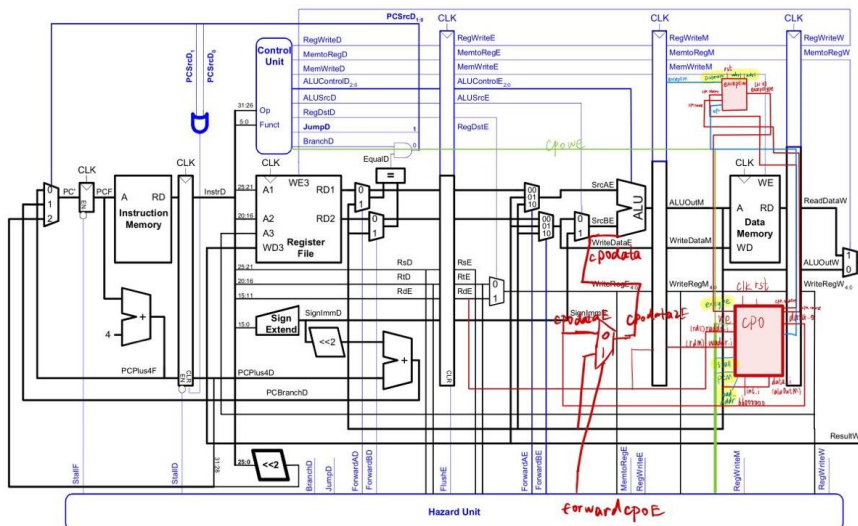
为了指明何时对HILO寄存器进行写操作,我们设置了一个使能信号hilo_write(1位),当hilo_write为1时表示对HILO寄存器进行写操作,当为0时不写。数据移动指令中MTHI,MTLO两条指令的hilo_write信号置为1。MFHI,MFLO,MTHI,MTLO指令的计算方式如下,这里hilo_in对应着HILO寄存器的输出,hilo_out对应着alu输出的hilo_outE,流水至MEM阶段以hilo_outM作为HILO寄存器的输入。

数据移动指令在alu中实现,具体代码如下:

```
//数据移动指令
`EXE_MFHI_OP: y <= hilo_in[63:32];
`EXE_MFLO_OP: y <= hilo_in[31:0];
`EXE_MTHI_OP: hilo_out <= {a[31:0],{hilo_in[31:0]}};
`EXE_MTLO_OP: hilo_out <= {{hilo_in[63:32]},a[31:0]};
```

2.1.6 特权指令

对于最后五条异常指令,其主要实现思路是在流水线的前四个阶段即取指、译码、执行和访存进行异常的搜集,然后统一在访存阶段进行异常类型的判断。判断异常类型这里我们使用了新增的exception.v模块以及资料包中cp0_reg.v,数据通路图如下:



(1) 内陷指令

内陷指令包括 break 和 syscall 指令，指令格式如下：

内陷指令	000000	code	001101	break
	000000	code	001100	syscall

高六位 op 均为 000000，所以我们在 datapath 中对 syscall 和 break 使用低六位 func 进行联合赋值，如下图所示：

```
assign syscallD = (opD == `EXE_SPECIAL_INST && functD == `EXE_SYSCALL);
assign breakD = (opD == `EXE_SPECIAL_INST && functD == `EXE_BREAK);
```

在 maindec.v 文件中添加控制信号，如下图所示：

```
`EXE_BREAK:controls <= 12'b000000000000;
`EXE_SYSCALL:controls <=12'b000000000000;
```

判断异常类型时，我们把 syscallD、breakD、eretD 及 invalidD 信号在 ID 阶段从 controller 中输出结合成 exceptD 信号，流水至 EXE 阶段为 exceptE。

```
flopencrc #(8) r13E(clk,rst,~stallE,flushE,
| {exceptD[7],syscallD,breakD,eretD,invalidD,exceptD[2:0]},
| exceptE);
```

在 EXE 阶段将 exceptE 和 overflowE 结合继续流水至 MEM 阶段。

```
flopencrc #(8) r9M(clk,rst,flushM,{exceptE[7:3],overflowE,
| | | | _ exceptE[1:0]},exceptM);
```

exceptM 作为 exception 模块的输入信号,用于判断是何种异常类型，如下图所示：

```

exception exp(
    .rst(rst),
    .ades(adesM),.adel(adelM),
    .except(exceptM),
    .cp0_status(status_o),
    .cp0_cause(cause_o),
    .excepttype(excepttypeM)
);

end
else if except[7]==1'b1 || adel) begin
    excepttype <= 32'h0000_0004;
end
else if(ades) begin//Data address error
    excepttype <= 32'h0000_0005;
end
else if(except[6]==1'b1 begin
    excepttype <= 32'h0000_0008;
end
else if except[5]==1'b1 begin
    excepttype <= 32'h0000_0009;
end
else if except[4]==1'b1 begin
    excepttype <= 32'h0000_000e;
end
else if except[3]==1'b1 begin
    excepttype <= 32'h0000_000a;
end
else if except[2]==1'b1 begin
    excepttype <= 32'h0000_000c;
end

```

exception 模块输出异常类型传给 cp0 寄存器，处理过程如下图：

```

32'h00000008:begin // Syscall异常
    if(is_in_delayslot_i == `InDelaySlot) begin
        /* code */
        epc_o <= current_inst_addr_i - 4;
        cause_o[31] <= 1'b1;
    end else begin
        epc_o <= current_inst_addr_i;
        cause_o[31] <= 1'b0;
    end
    status_o[1] <= 1'b1;
    cause_o[6:2] <= 5'b01000;
end
32'h00000009:begin // BREAK异常
    if(is_in_delayslot_i == `InDelaySlot) begin
        /* code */
        epc_o <= current_inst_addr_i - 4;
        cause_o[31] <= 1'b1;
    end else begin
        epc_o <= current_inst_addr_i;
        cause_o[31] <= 1'b0;
    end
    status_o[1] <= 1'b1;
    cause_o[6:2] <= 5'b01001;
end
end

```

break 指令和 syscall 指令还涉及冒险处理，在后文的 hazard 模块中会详细阐述。

(2) 特权指令

特权指令包括 mtc0, mfc0 和 eret 指令，这三条指令都是在 alu 中实现的，指令格式如下：

特权指令	31:26	25:21	20:16	15:11	10:3	2:0	
	010000	00100	rt	rd	00000000	sel	
	010000	00000	rt	rd	00000000	sel	
	31:26	25	24:6			5:0	
	010000	1	0000 0000 0000 0000 000			011000	eret

这三个指令高六位 op 均为 010000，所以在 maindec.v 和 aludec.v 对这三条指令进行译码时需要使用低六位 func 进行联合判断，如下图所示：

```

`EXE_CP0: case(rsD)
    `RS_MTC0: alucontrol <= `EXE_MTC0_OP;
    `RS_MFC0: alucontrol <= `EXE_MFC0_OP;
    `RS_ERET: alucontrol <= `EXE_ERET_OP;
    default: alucontrol <= 8'b00000000;
endcase

```

```

`EXE_CP0: case(rs)
|
|
|
|
| `RS_MTC0: controls <= 12'b000000000000;
| `RS_MFC0: controls <= 12'b100000000000;
| `RS_ERET: controls <= 12'b000000000000;
|     default: begin
|         controls <= 12'b000000000000;
|         invalid <= 1;
|     end//illegal op
endcase

```

mfc0 指令的功能是把由 rd 和 sel 组合指定的 CP0 寄存器的数据加载到通用寄存器 rt 中。mtc0 指令的功能是把通用寄存器 rt 的内容加载到由 rd 和 sel 组合指定的协处理器 CP0 寄存器中。这两条指令都是在 alu 中实现的，实现代码如下：

```

`EXE_MFC0_OP: y <= cp0data;
`EXE_MTC0_OP: y <= b;

```

eret 指令的功能是从中断、例外处理返回中断指令，eret 指令没有延迟槽。我们在 datapath 中直接使用指令序列对 eretD 信号进行赋值，然后和 syscallD、breakD 信号一起合成 exceptD 信号。

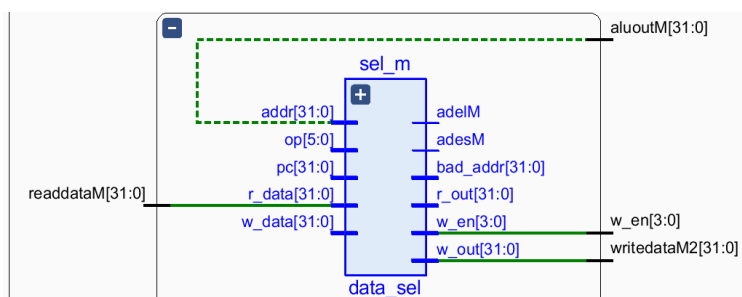
```
assign eretD = (instrD == `EXE_ERET);
```

cp0 寄存器处理过程如下图:

```
32'h0000000e:begin // eret异常
    status_o[1] <= 1'b0;
end
```

特权指令也还涉及冒险处理，在后文的 hazard 模块中会详细阐述。

2.2 data sel 模块设计



对于所有访存指令，alu 计算结果均为内存地址，需作为输入通过最后两位判断内存地址是否对齐，操作数 op 作为输入判断最终输出操作数的长度；从内存中读出的数据和准备写入内存的数据也均需作为输入，通过该选择器得出最后读出和写入的数据；后续还加入了异常处理机制，用于应对地址未对齐和无效地址的异常情况。

```

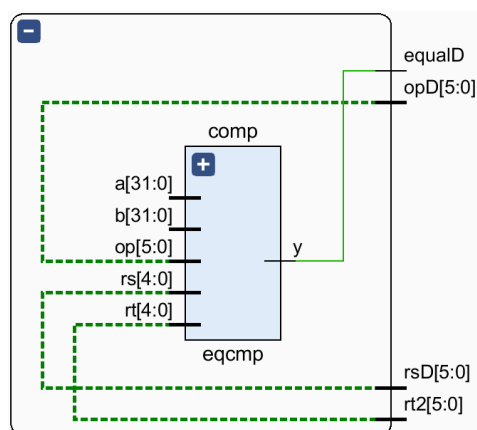
module data_sel(
input [31:0] pc,
input [31:0] addr, //内存地址
input [31:0] w_data, //写数据
input [31:0] r_data, //读数据
input [5:0] op,
output reg [31:0] w_out, //写数据
output reg [31:0] r_out,
output reg [3:0] w_en,
output reg adesM, adeIM,
output reg [31:0] bad_addr
);

case(op)
`EXE_LB:begin
case(addr[1:0])
2'b00:r_out<={{24{r_data[7]}},r_data[7:0]};
2'b01:r_out<={{24{r_data[15]}},r_data[15:8]};
2'b10:r_out<={{24{r_data[23]}},r_data[23:16]};
2'b11:r_out<={{24{r_data[31]}},r_data[31:24]};
endcase
end

case(op)
`EXE_LW: begin
if(addr[1:0]!=2'b00) begin
adesM <= 1'b1;
bad_addr <= addr;
end
end
end

```

2.3 eqcmp 模块设计



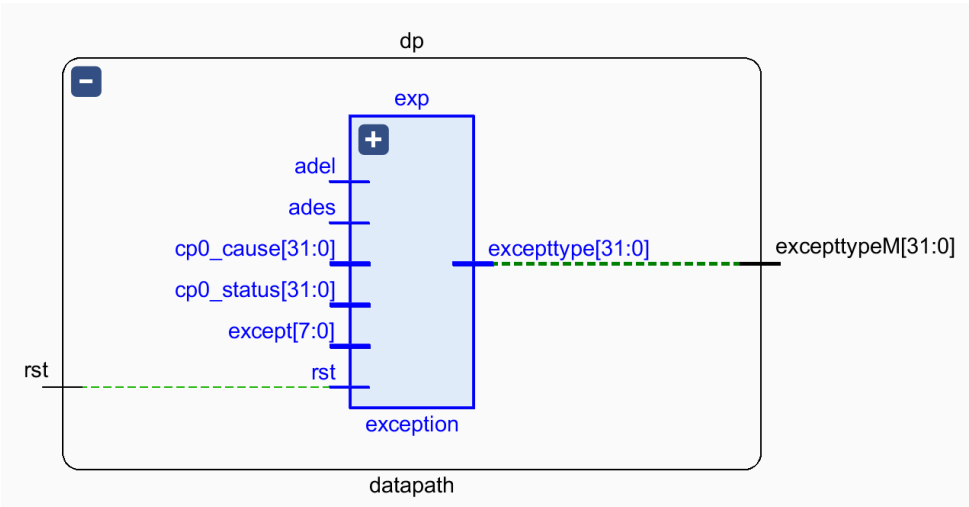
因为不同的分支指令有不同的 op（指令前 6 位），故可根据不同的 op 对操作数进行判断是否分支，并且所有分支指令的结构均一致，均为 6 位操作符、两个 5 位操作数（寄存器）、16 位立即数，eqcmp 模块还需将两个操作数作为输入，与零或另一个操作数进行比较，

输出一位是否跳转标志。

同时，还存在两种特殊情况，即无条件跳转 b (rs、rt 均为 0) 和 bal (rs 为 0)，故还需寄存器 rs、rt 作为输入

```
module eqcmp(  
    input [5:0] op,  
    input [4:0] rs,  
    input [4:0] rt,  
    input wire [31:0] a,b,  
    output reg y  
);  
  
case(op)  
`EXE_BEQ:begin  
if(rs==5'b00000&&rt==5'b00000) //b  
y<=1'b1;  
else if(a==b) y<=1'b1;  
else y<=1'b0;  
end
```

2.4 exception 模块设计



在异常处理中，流水线前 4 个阶段可能会产生一系列异常，于是我们在这 4 个阶段进行异常的搜集，然后统一在第 4 个阶段对搜集的异常进行异常类型的判断。一共有 8 种异常类型：

种类	异常类型	描述
1	0x01	软件中断

2	0x04	地址错例外（取指或读数据）
3	0x05	地址错例外（写数据地址）
4	0x08	系统调用例外 syscall
5	0x09	断点例外 break
6	0x0e	eret
7	0x0a	保留指令例外 invalid
8	0x0c	整型溢出 overflow

第一种，cp0_cause 和 cp0_status 联合控制，异常类型应该为 0x00000001。

第二种，取数据指令异常 adelM=1, 则 except 信号第 8 位置为 1, 异常类型应该为 0x00000004;

第三种，存数据指令异常 adesM=1, 异常类型应该为 0x00000005;

第四种，控制信号 syscallD=1, 则 except 信号第 7 位置为 1, 异常类型应该为 0x00000008;

第五种，控制信号 breakD=1, 则 except 信号第 6 位置为 1, 异常类型应该为 0x00000009;

第六种，控制信号 eretD=1, 则 except 信号第 5 位置为 1, 异常类型应该为 0x0000000e;

第七种，控制信号 invalidD=1, 则异常搜集第 4 位置为 1, 异常类型应该为 0x0000000a;

第八种，alu 溢出 overflowE=1, 则异常搜集第 3 位置为 1, 异常类型应该为 0x0000000c;

exception 模块实现代码如下：

```

module exception(
    input wire rst,
    input wire ades, adel,
    input wire [7:0] except,
    input wire [31:0] cp0_status, cp0_cause,
    output reg [31:0] excepttype
);

always @(*) begin
    if(rst) begin
        excepttype <= 32'b0;
    end
    else begin
        excepttype <= 32'b0;
        if (((cp0_cause[15:8] & cp0_status[15:8]) != 8'h00) && (cp0_status[1] == 1'b0) && (cp0_status[0] == 1'b1)) begin
            excepttype <= 32'h0000_0001;
        end
        else if(except[7] == 1'b1 || adel) begin
            excepttype <= 32'h0000_0004;
        end
        else if(ades) begin //Data address error
            excepttype <= 32'h0000_0005;
        end
    end
end

```

```

else if(except[6]==1'b1) begin
|   excepttype <= 32'h0000_0008;
end
else if (except[5]==1'b1) begin
|   excepttype <= 32'h0000_0009;
end
else if (except[4]==1'b1) begin
|   excepttype <= 32'h0000_000e;
end
else if (except[3]==1'b1) begin
|   excepttype <= 32'h0000_000a;
end
else if (except[2]==1'b1) begin
|   excepttype <= 32'h0000_000c;
end
end
end

```

2.5 写透 cache 设计

我们在硬件综合设计中借鉴了计算机组成原理及体系结构第五次实验的内容,使用了实验五中提供的 bridge_1x2 和 bridge_2x1 模块,在连接了 AXI 的基础上进行设计。

总体的设计思路是首先通过 datapath 请求数据,将待请求的数据分为 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,等到地址握手后改变状态机,等待数据握手,并将流水线暂停。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 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。结构图如下:

冒险模块包括数据前推和流水线暂停，数据前推包括前推至 ID 阶段和前推至 EX 阶段，将 EX 阶段和 MEM 阶段产生的数据从旁路发送给 reg 模块、分支判断模块、ALU 计算模块，需要判断是否对同一个寄存器进行读写，输入和输出为各个阶段的控制信号：

```
//前推到ID
assign forwardaD = ((rsD != 0) && (rsD == writeregM) && regwriteM) ? 2'b10 :
                  ((rsD != 0) && (rsD == writeregE) && regwriteE) ? 2'b01 :
                  2'b00;
assign forwardbD = ((rtD != 0) && (rtD == writeregM) && regwriteM) ? 2'b10 :
                  ((rtD != 0) && (rtD == writeregE) && regwriteE) ? 2'b01 :
                  2'b00;
assign flush_except = (excepttypeM != 32'b0);
assign forwardcpOE = ((rdE!=0)&(rdE == rdM)&(cp0weM)) ? 1'b1 : 1'b0;
assign flushF = (flush_except);
assign flushD = (flush_except);

//前推到EX
always @(*) begin
    forwardaE = 2'b00;
    forwardbE = 2'b00;
    if(rsE != 0) begin
        if(rsE == writeregM & regwriteM) begin
            forwardaE = 2'b10;
        end else if(rsE == writeregW & regwriteW) begin
            forwardaE = 2'b01;
        end
    end
    if(rtE != 0) begin
        if(rtE == writeregM & regwriteM) begin
            forwardbE = 2'b10;
        end else if(rtE == writeregW & regwriteW) begin
            forwardbE = 2'b01;
        end
    end
end
end
```

对于分支和跳转指令，需要在 ID 阶段得到寄存器中的值，故如果存在数据相关型冒险，需要对流水线进行暂停：

```

assign lwstallD = memtoregE & (rtE == rsD | rtE == rtD);

assign branchstallD = ((branchD||jumpD) && regwriteE && (writeregM == rsD || writeregM == rtD) || (branchD||jumpD)
&& memtoregM && (writeregM == rsD || writeregM == rtD));
assign stallD = lwstallD | branchstallD|stall_divE;
assign stallF = flush_except?1'b0:stallD;
assign stallE = branchstallD | stall_divE;

assign flushE = lwstallD | branchstallD|flush_except;

assign flushM = stall_divE|flush_except;
assign flushW = flush_except;

```

3 实验过程

3.1 设计工作日志

刘静婷: 12.25 配置 WSL2, 安装 vivado2019.2; 观看指导视频

12.26 安装 vivado, 配置实验环境; 复习流水线和数据通路相关知识

12.27 分析逻辑运算指令数据通路, 并添加部分逻辑运算指令代码

12.29 添加部分逻辑运算指令、移位指令 以及访存指令

12.30 添加数据移动指令

12.31 调试并修改数据移动指令错误

1.2 学习特权指令和 cp0 寄存器

1.3 添加 cp0 寄存器和特权指令

1.4 调试并修改特权指令错误

1.5 学习添加 soc

1.6 添加 soc, 调试 bug, 通过 64 个测试点

1.7 调试并通过 89 条剩余功能测试

1.8 连接 axi 接口, 调试通过部分 axi 功能测试

1.9 调试并通过 axi 剩余功能测试

1.10 连接直写 cache

1.11 完成 cache 上板性能测试, 完善整个实验设计

皮悦颖: 12.26 安装 vivado, 配置实验环境; 复习流水线和数据通路相关知识

12.27 分析逻辑运算指令数据通路, 并添加部分逻辑运算指令代码

12.29 添加部分逻辑运算指令、移位指令以及访存指令

12.30 添加剩余访存指令

12.31 调试并修改访存指令错误

1.2 学习跳转指令数据通路并添加部分跳转指令

1.3 添加剩余分支跳转指令

1.4 调试并修改分支指令错误

- 1.5 调试并修改跳转指令错误；学习添加 soc
- 1.6 添加 soc，调试 bug，通过 64 个测试点
- 1.7 调试并通过 89 条剩余功能测试
- 1.8 连接 axi 接口，调试通过部分 axi 功能测试
- 1.9 调试并通过 axi 剩余功能测试
- 1.10 连接直写 cache
- 1.11 完成 cache 上板性能测试，完善整个实验设计

邓茜：12.26 配置实验环境

- 12.27 学习 2022mips 视频，复习 lab4
- 12.28 看 2019 视频并跟做部分 sllv，复习 lab4
- 12.29 观看视频，学习乘法除法相关原理
- 12.31 添加部分乘除法指令
- 1.2 学习 hilo 寄存器，梳理其和乘除法相关的部分
- 1.3 继续添加除法相关代码
- 1.4 学习除法涉及的流水线暂停，添加代码
- 1.5 学习 soc
- 1.6 绘制数据通路草图
- 1.7 学习 axi，并根据反馈调整数据通路图
- 1.8 继续细化数据通路图
- 1.9 学习 cache
- 1.10 完善数据通路图

覃倩瑶：12.26 8:30-11:30 安装 vivado、WSL2

14:25-17:30 解决配置环境出现的问题

- 12.27 观看视频复习，并跟做部分内容，复习前面的知识
- 12.29 学习部分移位指令并进行添加指令
- 12.31 学习分支指令
- 1.1 添加部分算数指令
- 1.2 添加部分逻辑移位指令
- 1.3 开始设计、绘制数据通路图
- 1.4 学习 soc
- 1.5 添加 soc 并尝试修改 debug 错误的问题
- 1.6 尝试修改 sram 测试中的 debug 并继续绘制数据通路草图
- 1.7 尝试修改 sram 测试中的 debug 并学习 axi 相关知识
- 1.8 继续绘制数据通路图并连接 axi
- 1.9 尝试修改 axi 测试中的 debug
- 1.10 观看视频学习 cache

3.2 主要的错误记录

1、错误 1

(1) 错误现象: Hilo 寄存器取值问题

在写数据移动指令时仿真遇到 mfhi 和 mflo 指令取不到值的情况。

(2) 分析定位过程

编写 hilo.v 和 alu.v 等文件, 添加数据移动指令的测试 coe 文件, 对波形图进行分析。

(3) 错误原因

因为把 hilo 寄存器放在 exe 阶段写, 但是没有进行冒险处理, 导致数据还没来得及写, 读操作就发生了。

(4) 修正效果

查阅 ppt 后得知如果把 Hilo 在 MEM 阶段上升沿写, 在 EXE 阶段读取, 可以不用处理冒险。由于当时对冒险处理理解的不够深入所以直接对数据通路进行了修改, 在 MEM 阶段进行写操作, 问题消除。

2、错误 2

(1) 错误现象: 乘法指令 aluoutE 为 0

在进行算数指令测试时发现乘法指令的两个乘数正确但是结果只有低 32 位的值。

(2) 分析定位过程

编写 alu.v 和 datapath.v 等文件, 添加算数指令的测试 coe 文件, 对代码进行检查, 对波形图进行分析。

(3) 错误原因

因为之前 debug 的时候把写 hilo 移动到了 MEM 阶段, 导致 alu 输出的 hilo_outE 流水了一级, 但是 datapath 中对 alu 的 hilo_in 信号传参没有改为 hilo 寄存器输出的 hilo_outM 信号, 还是原来的 hilo_outE。

(4) 修正效果

传参修改正确后, 仿真正确。

3. 错误 3

(1) 错误现象: MFC0 指令异常进入到 BFC00380

(2) 分析定位过程: 和从 GitHub 上找到的模板代码仿真波形图进行对比, 仔细分析代码哪里逻辑不对

(3) 错误原因: exception 判断使能信号逻辑不对

(4) 修正效果: 修改逻辑后通过仿真测试

4. 错误 4

(1) 错误现象: 从内存中指定虚拟地址中读数据的时候发现不是之前存进去的数据

(2) 分析定位过程: 和从 GitHub 上找到的模板代码仿真波形图进行对比, 仔细分析代码哪里逻辑不对

(3) 错误原因: 发现在 data_sel 模块中对 w_en 的赋值逻辑有缺陷, 对 SW, SB, SH 指令添加如下代码:

```

`EXE_SW:
begin
    if(adesM==1'b1)
        w_en<=4'b0;
    else begin
        w_en <= 4'b1111;
        final_writed <= w_data;
    end
end

```

(4) 修正效果：修改逻辑后，通过 89 条功能测试

5、错误 5

(1) 错误现象：无法得到除法结果

在写除法指令时仿真遇到 div 和 udiv 指令无阶段结果的情况。

(2) 分析定位过程

检查 div.v 和 alu.v 等文件，添加除法指令的测试 coe 文件，对波形图进行分析。

(3) 错误原因

因为 alu.v 无 clk、rst 信号，在 alu 中直接调用了有 clk、rst 信号的 div，但是没有在 alu 添加，导致 div 模块无法正常进行。

(4) 修正效果

在 alu 添加 clk、rst，结果显示。

6、错误 6

(1) 错误现象：只能显示部分指令的除法结果。

在写除法指令时仿真遇到 div 和 udiv 指令暂停正常但无阶段结果的情况。

(2) 分析定位过程

检查 div.v 和 alu.v 等文件，添加除法指令的测试 coe 文件，对波形图进行分析。

(3) 错误原因

除法指令在测试文件的最后进行且除法本身因为暂停所需时间会高于其余指令很多，所以原本设定运行时间不足以让除法得出结果。

(4) 修正效果

延长运行时间，所有除法指令结果均可见。

7、错误 7

(1) 错误现象：指令执行过程中该跳转时没有正确跳转。

(2) 分析定位过程

发现从 reg 中读出的第一操作数为 X，意味着并未从寄存器中读出正确的数值。并且阅读指令源文件发现前一条指令为 jal 指令，对 31 号寄存器进行了写入操作，而下一条 beq 指令需要在 ID 阶段对 31 号寄存器中的数进行比较，判断跳转。

(3) 错误原因

存在控制冒险，前一条指令的 MEM 阶段计算出的结果并未写入寄存器中，需要增加一个数据前推，从 MEM 阶段推至 ID 阶段。


```

assign forwardaD = ((rsD != 0) && (rsD == writeregM) && regwriteM) ? 2'b10 :
                  ((rsD != 0) && (rsD == writeregE) && regwriteE) ? 2'b01 :
                  2'b00;
assign forwardbD = ((rtD != 0) && (rtD == writeregM) && regwriteM) ? 2'b10 :
                  ((rtD != 0) && (rtD == writeregE) && regwriteE) ? 2'b01 :
                  2'b00;

```

(4) 修正效果

reg 读出的数据正常，不再为 X，测试点通过。

8、错误 8

(1) 错误现象：分支指令附近波形图时序混乱，后续指令提前执行。

(2) 分析定位过程

发现从 reg 中读出的第一操作数为 X，意味着并未从寄存器中读出正确的数值。并且阅读指令源文件发现前一条指令为 lw 访存指令，将内存中读出的数据在 WB 阶段存入相同寄存器中，而下一条 beq 指令需要在 ID 阶段对寄存器中的数进行比较，判断跳转。

(3) 错误原因

存在控制冒险，前一条指令的 MEM 阶段写回的数并未写入寄存器中，需要对流水线进行暂停操作，修改对 branchstallD 置 1 的条件。

```

assign branchstallD = ((branchD || jumpD) && regwriteE && (writeregM == rsD || writeregM == rtD) || (branchD || jumpD)
&& memtoregM && (writeregM == rsD || writeregM == rtD));

```

(4) 修正效果

reg 读出的数据正常，不再为 X，测试点通过。

9、错误 9

(1) 错误现象：访存数据与参考数据不符。

(2) 分析定位过程

观察访存指令附近仿真波形图，发现写使能信号异常，不该写时进行了写操作。

(3) 错误原因

MEM 和 WB 阶段的流水线寄存器和 controller 中的流水线寄存器中未加入 flush 信号，导致后续信号并未清空，写使能信号不为零。

```

flopdc #32) r1M(clk,rst,flushM,srcb2E,writedataM);
flopdc #32) r2M(clk,rst,flushM,aluout2E,aluoutM);
flopdc #5) r3M(clk,rst,flushM,writeregE,writeregM);
flopdc #6) r4M(clk,rst,flushM,opE,opM);
flopdc #64) r5M(clk,rst,flushM,hilo_inE,hilo_inM);
flopdc #5) r6M(clk,rst,flushM,rdE,rdM);
flopdc #1) r7M(clk,rst,flushM,is_in_delayslotE,is_in_delayslotM);
flopdc #32) r8M(clk,rst,flushM,pcE,pcM);
flopdc #8) r9M(clk,rst,flushM,{exceptE[7:3],overflowE,exceptE[1:0]},exceptM);

flopdc #9) regM(
    clk,rst,flushM,
    {memtoregE,memwriteE,regwriteE,hilo_writeE,stallE,cp0weE},
    {memtoregM,memwriteM,regwriteM,hilo_writeM,stall_divM,cp0weM}
);

```

(4) 修正效果

内存读写操作正常，测试点通过。

10、错误 10

(1) 错误现象：内存读写数据时序异常，与参考波形图不符。

(2) 分析定位过程

观察内存写使能信号，发现数据被提前写入。

(3) 错误原因

在 data_sel 模块使用 opD 作为输入，使结果提前，应该通过流水线逐级传递 op 信号，在 data_sel 模块使用 opM，防止流水线混乱

```
flopenrc #(6) r8E(clk,rst,~stallE,flushE,opD,opE);
```

```
floprrc #(6) r4M(clk,rst,flushM,opE,opM);
```

(4) 修正效果

内存读写操作正常，测试点通过。

4 设计结果

4.1 设计交付物说明

设计交付物包括项目报告、项目代码源文件，其中项目源文件是 axi 框架下的 mycpu 文件包，一个 readme 的 txt 文档其中各个.v 文件会做一个具体的介绍说明。

4.2 设计演示结果

4.2.1 功能测试通过 89 个测试点：

```
-----[28312000 ns] Test is running, debug_wb_pc = 0xbfc22530
[28322000 ns] Test is running, debug_wb_pc = 0xbfc22578
-----[28322535 ns] Number 8'd88 Functional Test Point PASS!!!
[28332000 ns] Test is running, debug_wb_pc = 0x00000000
[28342000 ns] Test is running, debug_wb_pc = 0xbfc003a4
[28352000 ns] Test is running, debug_wb_pc = 0xbfc15358
[28362000 ns] Test is running, debug_wb_pc = 0xbfc003a0
[28372000 ns] Test is running, debug_wb_pc = 0xbfc153d0
[28382000 ns] Test is running, debug_wb_pc = 0x00000000
[28392000 ns] Test is running, debug_wb_pc = 0xbfc15458
[28402000 ns] Test is running, debug_wb_pc = 0xbfc154e4
[28412000 ns] Test is running, debug_wb_pc = 0xbfc154e8
[28422000 ns] Test is running, debug_wb_pc = 0xbfc1551c
[28432000 ns] Test is running, debug_wb_pc = 0xbfc15558
[28442000 ns] Test is running, debug_wb_pc = 0xbfc15594
-----[28443015 ns] Number 8'd89 Functional Test Point PASS!!!
[28452000 ns] Test is running, debug_wb_pc = 0xbfc00cf4

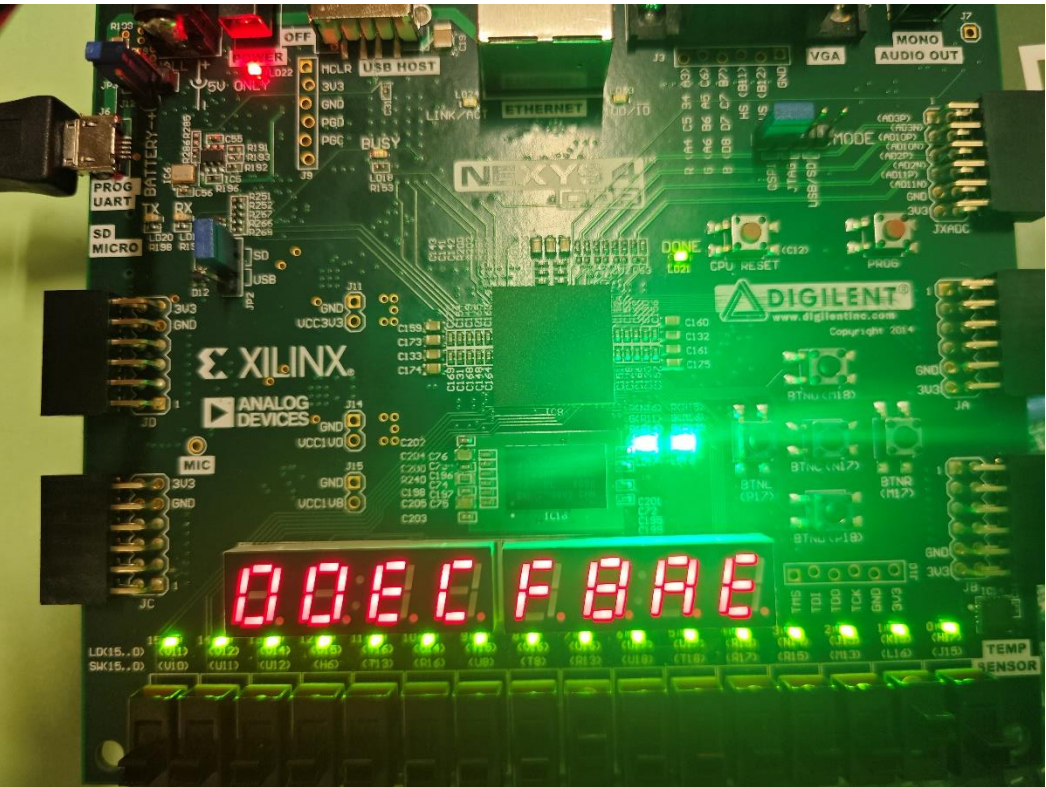
Test end!
-----PASS!!!
$finish called at time : 28454801500 ps : File "D:/1.6.2/AYZ/CO-lab-material-CQU/vivado/func_test/v0.01/soc_axi_func_sim/testbench/mycpu_tb.v" Line 269
run: Time (s): cpu = 00:00:41 ; elapsed = 00:05:16 . Memory (MB): peak = 1271.391 ; gain = 0.000
```

4.2.2 性能测试上板分数：

二、性能测试分数计算

序号	测试程序	myCPU	gs132	T_{gs132}/T_{mycpu}
		上板计时(16进制)	上板(16进制)	
		数码管显示	数码管显示	
cpu_clk : sys_clk		60MHz : 100MHz	50MHz : 100MHz	-
1	bitcount	15C8BD	13CF7FA	14.55043656
2	bubble_sort	CC6317	7BDD47E	9.696455116
3	coremark	1A03589	10CE6772	10.33712107
4	erc32	E58876	AA1AA5C	11.85740119
5	dhrystone	38CBA2	1FC00D8	8.944437967
6	quick_sort	C9AAC3	719615A	9.011786447
7	select_sort	6B2E2F	6E0009A	16.42093399
8	sha	AFF295	74B8B20	10.61420335
9	stream_copy	120A77	853B00	7.38492143
10	stringsearch	B86663	50A1BCC	6.996252411

性能分 10.230



4.2.3 现场添加指令 MAX

```
//MAX
8'b11111111:begin
  if(a[31]!=b[31])
    y<=(a[31]==0)?a:b;
  else if(a[31]==0 && b[31]==0)
    y<=(a[30:0]>=b[30:0])?a:b;
  else if(a[31]==1 && b[31]==1)
    y<=(a[30:0]>=b[30:0])?a:b;
end
```

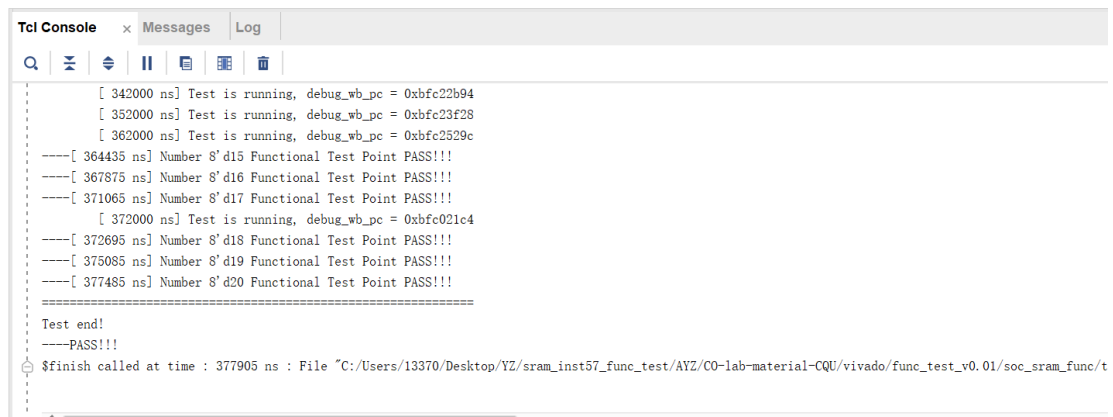
```

//MAX
6'b111111:case(funcnt)
|
|   6'b000000:controls <=12'b110000000010;
|   default: begin
|       |
|       |   controls <= 12'b000000000000;
|       |   invalid <= 1;
|       |
|       |   end
|   endcase

6'b111111:case(funcnt)
|
|   6'b000000:alucontrol <= 8'b11111111;
|   default: alucontrol <=8'b00000000;
|   endcase

```

测试通过:



```

Tcl Console x Messages Log
[ 342000 ns] Test is running, debug_wb_pc = 0xbfc22b94
[ 352000 ns] Test is running, debug_wb_pc = 0xbfc23f28
[ 362000 ns] Test is running, debug_wb_pc = 0xbfc2529c
----[ 364435 ns] Number 8'd15 Functional Test Point PASS!!!
----[ 367875 ns] Number 8'd16 Functional Test Point PASS!!!
----[ 371065 ns] Number 8'd17 Functional Test Point PASS!!!
[ 372000 ns] Test is running, debug_wb_pc = 0xbfc021c4
----[ 372695 ns] Number 8'd18 Functional Test Point PASS!!!
----[ 375085 ns] Number 8'd19 Functional Test Point PASS!!!
----[ 377485 ns] Number 8'd20 Functional Test Point PASS!!!
=====
Test end!
----PASS!!!
$finish called at time : 377905 ns : File "C:/Users/13370/Desktop/YZ/sram_inst57_func_test/AYZ/CO-lab-material-CQU/vivado/func_test_v0.01/soc_sram_func/t

```

5 参考设计说明

- 1、数据通路图参考给了综合设计压缩包中给出的示例 pdf 文件
2. 写透 cache 参考了计组实验五示例代码

6 总结

这门硬件综合设计课程的内容非常丰富，以循序渐进的方式引导我们小组逐步掌握各种关键概念和技能。该课程从添加指令开始，让我们了解如何在设计中引入新的指令，并理解指令的功能和实现方式，并学习了如何连接 SOC，进一步学习了如何连接 AXI，调试对应的 debug，此外，还涵盖了添加 cache 的内容，让我们了解如何优化存储器访问和数据处理效率。通过这些学习和实践，我们小组不仅对 Verilog 语言和 vivado 工具有了更深入的了解和熟练运用，还对五级流水线、7 类指令和 SOC 等概念有了更加深刻的认识。学会了如何进行代码的调试和排错，在实验过程中，逐步添加指令功能，从基础开始一点一点地构建

项目，在这个过程中积累了丰富的经验和技能。总的来说，这门硬件综合设计课程提供了一个全面的学习平台，让我们在实践中不断提升自己的硬件设计能力。通过逐步学习和实践，不仅掌握了关键的知识和技能，还培养了解决问题和持续学习的能力。

7 参考文献

- [1] 李亚民. 计算机原理与设计 [M]. 北京: 清华大学出版社, 2011.
- [2] 雷思磊. 自己动手写 CPU. 电子工业出版社, 2014.
- [3] MIPS 基准指令集手册.pdf
- [4] 附录 A_coe 文件涉及指令一查表.pdf [5] 附录 A09_CPU 仿真调试说明 _v1.00.pdf