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、2 班

年　　级：2022

姓　　名：姚凡　吴明军　冉倬樾

学　　号：20221879　20221928　20220999

完成时间：2025 年 1 月 10 日

成　　绩：

指导教师：谭玉娟

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

姚凡、吴明军、冉倬樾

1 设计简介

本次硬件综合设计是设计并实现一个基于五级流水线的 CPU，支持 57 条 MIPS 指令，并实现异常处理模块，以及通过 AXI 总线连接了 Cache。在这一过程中，我们小组以《计算机组成原理》课程中的 Lab4 作为起点，逐步完成了从基础 CPU 构建到性能优化的全过程。

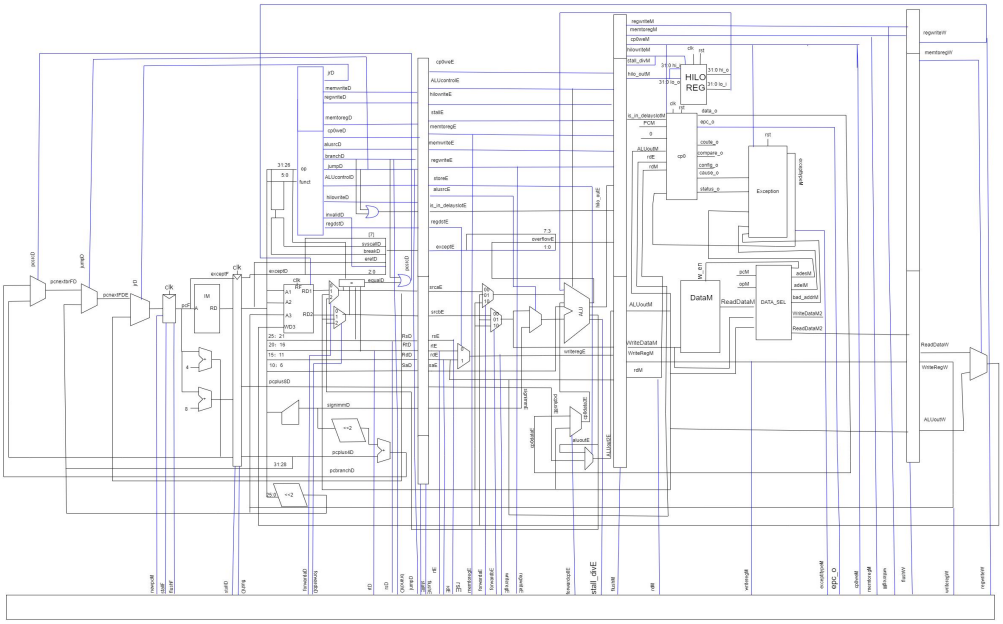
第一阶段，我们逐步将指令集扩展至 52 条。通过对控制器（controller）模块、算术逻辑单元（ALU）模块以及数据路径（datapath）模块的修改和优化，实现了 52 条指令的扩展。同时，为了支持更多复杂的计算和数据处理，我们还增加了 hilo 寄存器和除法器等功能模块。

第二阶段，在扩展至 52 条指令的基础上，我们使用 SRAM 接口连接 SOC。并在 hazard 模块进行了大量的调试，确保了 CPU 在执行 52 条指令时不会发生数据冒险等问题。成功通过前 64 个测试点的功能测试和完整版指令的 Trace 测试，验证了系统在不同工作负载下的稳定性。

第三阶段，我们将 52 条指令扩展为 57 条指令。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，这些指令使得 CPU 在计算和控制流程方面具备了更高的灵活性和多样性。我们还优化了 CP0 协处理器，提高了模块的效率。至此，CPU 不仅能够正确执行异常处理指令，还顺利通过了 89 个功能测试点。

第四阶段，我们将 CPU 通过类 SRAM 接口连接到 AXI 总线，复用了部分以往实验代码，实现了与 AXI 接口的对接以及添加 cache，并成功完成通过了相应的功能测试和性能测试，其中性能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

姚凡：负责算数指令、自陷指令、连接 SOC、AXI，主要负责实现写透 Cache，设计数据通路，实现性能测试并上板。

吴明军：负责访存指令、分支指令、连接 SOC、AXI，实现写透 Cache，设计数据通路，功能测试。

冉倬樾：负责数据移动指令、移位指令、特权指令和逻辑运算指令，连接 SOC、AXI，实现写透 Cache，设计数据通路，功能测试。

2 设计方案

这部分内容我们对 7 类指令中比较复杂的、特殊的指令和异常、例外模块等相关模块进行了进一步介绍，未介绍的部分指令与之前 Lab4 实现逻辑类似，在这里就不再赘述。

2.1 总体设计思路

我们的 CPU 基于五级流水线设计，每个阶段之间有相应的触发器来实现阶段之间信号量的传递，流水线由取指(Fetch)，译码(Decode)，执行(Execute)，访存(Memory)，写回(WriteBack)五个阶段组成。mips 模块主要由 maindec 和 datapath 模块组成，其中 maindec 模块用于判断不同指令并进行译码操作，包括 alu 参与的指令的译码以及所有指令所对应译码以产生相应的控制信号。datapath 模块包括阶段取指 fetch，译码 decode，运算 execute，访存 memory，写回 write_back 5

个模块构成，同时在流水线文件 pipeline_reg 中包括他们的暂存区，以及包括全局控制流水线暂停的 ctrl 模块。同时，我们还连接了 SOC、AXI，通过类 sram 接口将 CPU 发送的数据请求转换为类 Sram 格式，并加入写透 cache，以实现与外层模块更高效的数据交互和性能优化。

2.2 算术指令

2.2.1 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;
```

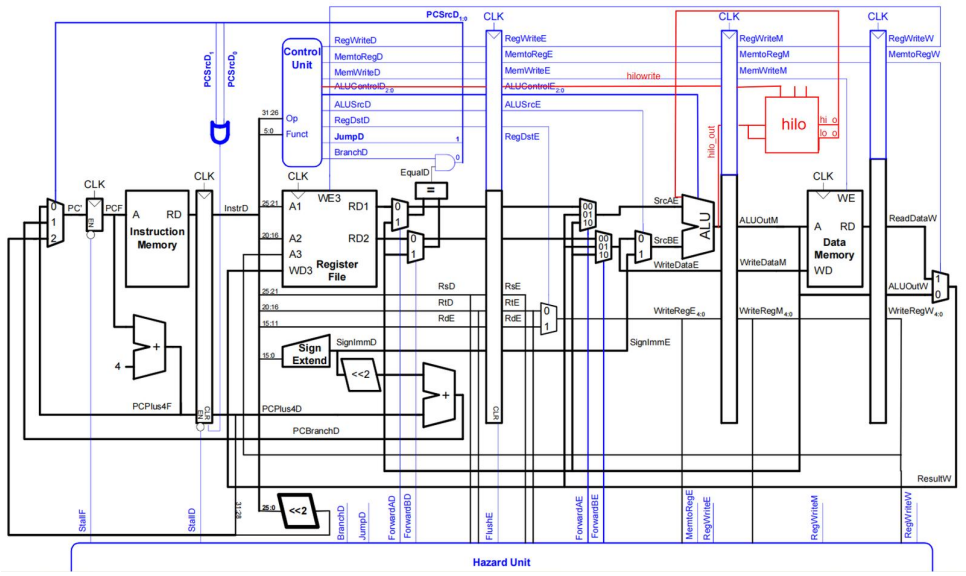
乘法运算我们选择直接采用乘号“*”进行运算，不额外使用乘法器。

```
`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：



2.2.2 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。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，开始则为 1，未开始或结束则为 0；annul_i 在本次实验中直接设置为 0；ready_o 表示除法的结果是否计算出来（准备好）结果未计算出则为 0，反之则为 1。由于除法无法在一个周期内完成运算，因此引入 stall_div 暂停流水线，等待除法器计算。除法的暂停分为两种：暂停 F、D、E 阶段和暂停所有阶段。由于单独暂停 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。

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)
```

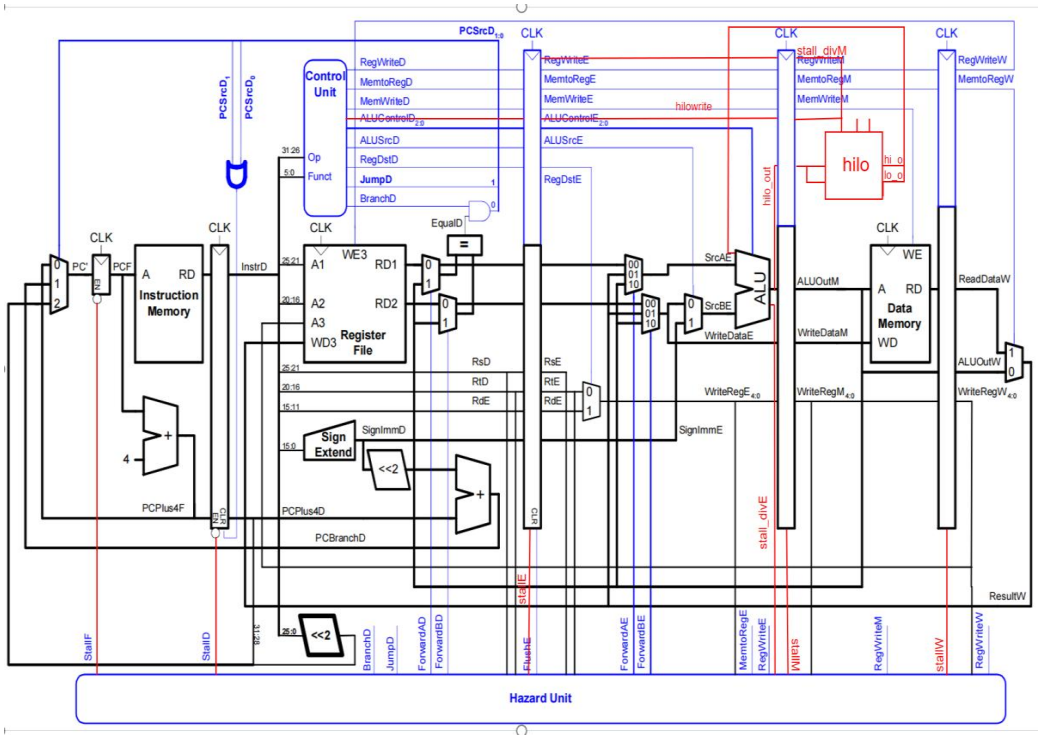
sign 在 div 时置为 1，udiv 时置为 0 。start、stall 信号取决于 ready，即除法的结果是否已经准备好：当 ready 为 0 时，start 为 1 除法器运行，stall 为 1 流水线全线暂停等待除法结果；当 ready 为 1 时，start 为 0 结果已出除法器停止运行，stall 为 0 流水线继续运行。

```
`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
```

```
`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
```

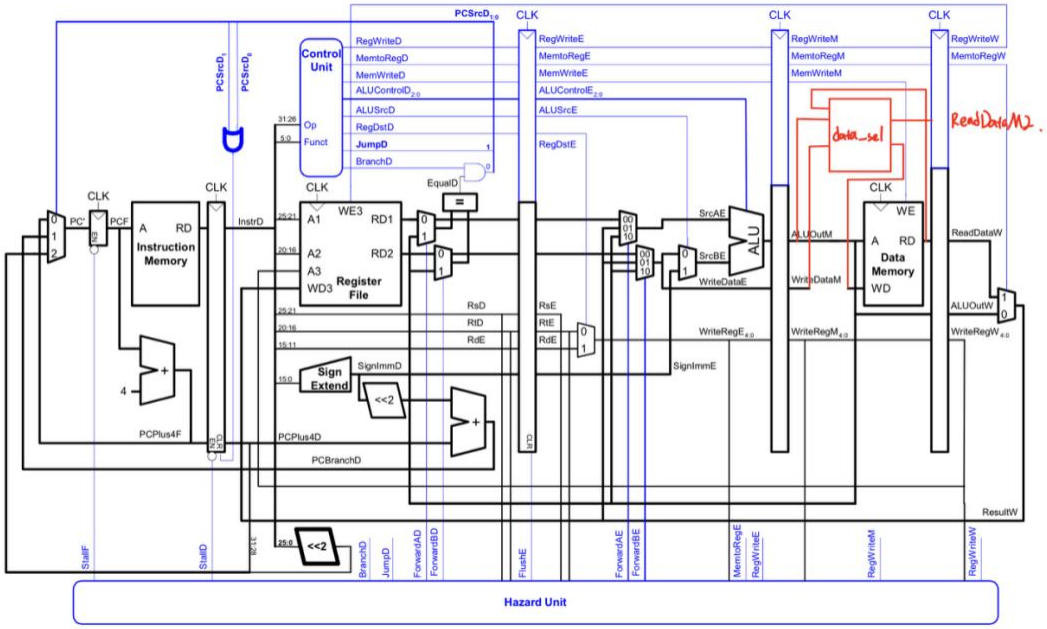
除法需要在数据通路中添加 hilo 寄存器，并向各阶段触发器添加暂停信号，具体通路改变

情况如下图所示：



2.3 访存指令

访存指令 datapath：



访存指令的目标地址由基址寄存器（rs）和立即数偏移量（offset）共同决定，因此在指令执行的 EX 阶段，需要启用立即数扩展和加法操作来完成目标地址的计算。不同的访存指令对应不同的数据长度，内存中的数据通常以 4 字节为一个存储单元，而访存指令可能需要访问其中

的 1 字节、2 字节或 4 字节数据。因此，在访存指令的 MEM 阶段，需要设计一个字节选择器来处理不同长度的数据。

对于取数据的指令（如 LB、LH、LW 等），内存中返回的数据长度固定为 4 字节，而访存指令只需其中的一部分数据（如 1 字节或 2 字节），因此需要通过字节选择器选出对应的数据片段，并根据指令类型进行有符号或无符号扩展。

对于存数据的指令（如 SB、SH、SW 等），需要将寄存器中的数据按要求写入到内存中的某些字节。这就需要一个长度为 4 位的写使能信号（w_en），用于控制哪些字节被写入。

同时对于访存指令需要在计算访存地址后，检查其是否满足对齐要求。不同类型的访存指令有不同的对齐约束，字（Word）访问：地址必须是 4 的倍数，即地址的低两位为 00；

半字（Halfword）访问：地址必须是 2 的倍数，即地址的最低位为 0；

字节（Byte）访问：无对齐要求。

如地址不符合对齐约束，则需触发地址异常信号（adelM 或 adesM）。

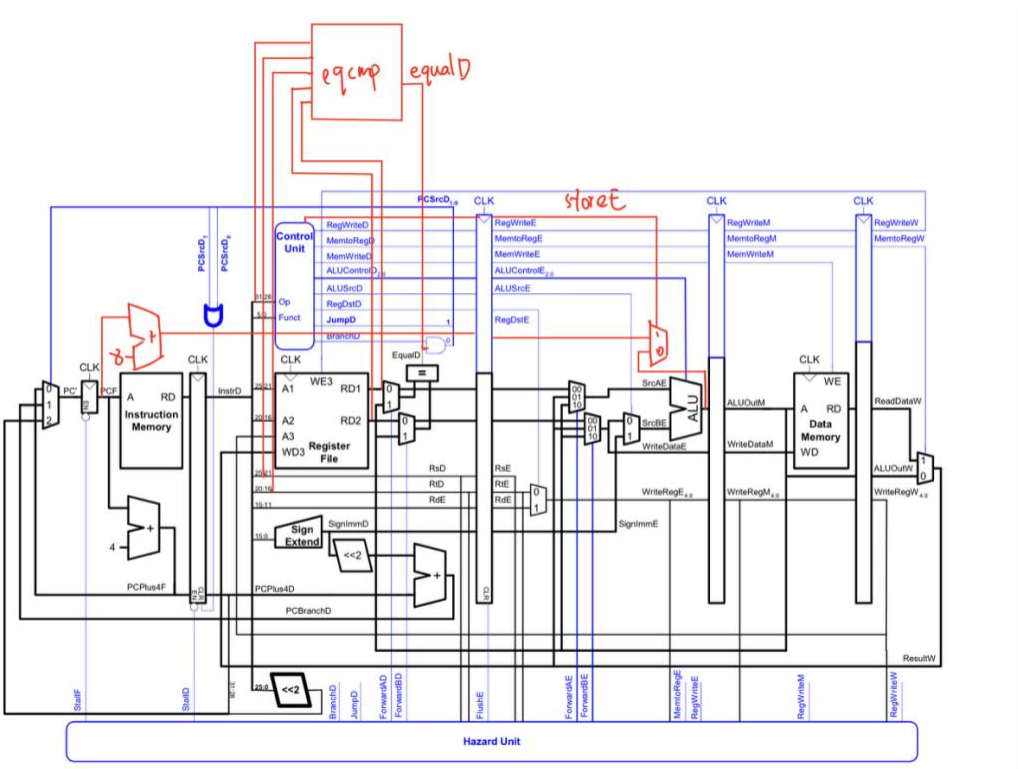
异常处理与字节选择的功能在 data_sel 模块中实现，具体的功能将在 data_sel 模块设计板块进行详细的描述。

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


data_sel lm(
    .pc(pcM),
    .addr(aluoutM),
    .op(opM),
    .r_data(readdataM),
    .w_data(writedataM),
    .final_readd(final_readdM),
    .final_writed(final_writedM),
    .w_en(w_en),
    .adesM(adesM),
    .adelM(adelM),
    .bad_addr(bad_addrM)
);
```

2.4 分支跳转指令

分支指令 datapath：



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);
```

2.5 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

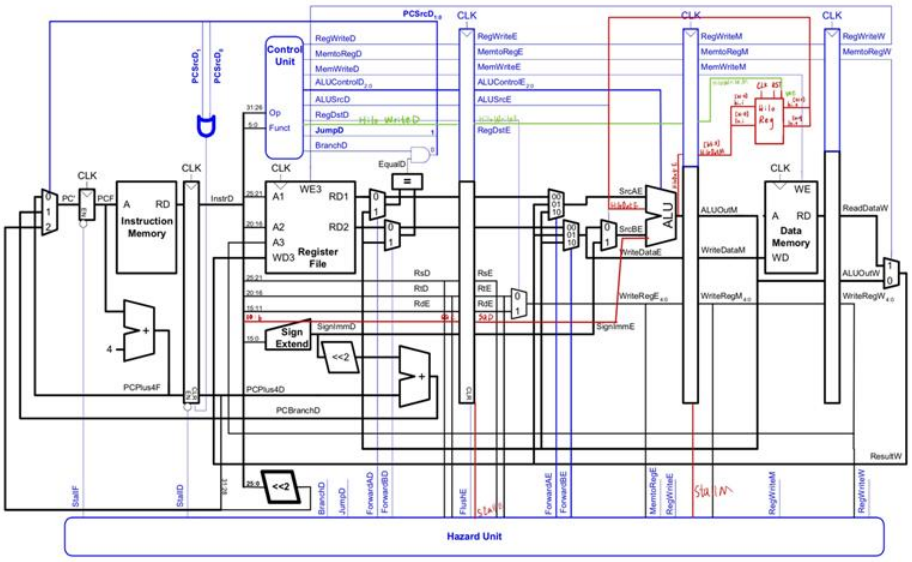
在 maindec.v 和 aludec.v 对四条指令进行译码，由于高六位 op 均为 000000，需要使用低六位

func 进行联合判断，如下图所示：

```
6'b000000:case(funct)
  `EXE_MFHI:controls <= 12'b110000000010;
  `EXE_MTHI:controls <= 12'b000000000110;
  `EXE_MFLO:controls <= 12'b110000000010;
  `EXE_MTLO:controls <= 12'b000000000110;

6'b000000:case(funct)
    `EXE_MFHI:alucontrol <= `EXE_MFHI_OP;
    `EXE_MTHI:alucontrol <= `EXE_MTHI_OP;
    `EXE_MFLO:alucontrol <= `EXE_MFLO_OP;
    `EXE_MTLO:alucontrol <= `EXE_MTLO_OP;
```

为了实现数据移动指令，需要在 cpu 中接入 HILO 寄存器，数据通路图如下：



在 alu.v 中添加实现代码，其中 hilo_in 对应 HILO 寄存器的输出，hilo_out 对应 alu 输出的 hilo_outE，代码如下图所示：

```
`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]};
```

2.6 自陷指令和特权指令

2.6.1 自陷指令

自陷指令包括 break 和 syscall 指令，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：

内陷指令	000000	code	001101	break
	000000	code	001100	syscall

```
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。

```
flopenrc #(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。

```
floprc #(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)
);
```

exception 模块输出异常类型传给 cp0 寄存器：

```
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
```

```
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
```

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。

2.6.2 特权指令

特权指令包括 mtc0,mfc0 和 eret 指令，指令格式如下：

	31:26	25:21	20:16	15:11	10:3	2:0	
特权指令	010000	00100	rt	rd	00000000	sel	mtc0 rt,rd
	010000	00000	rt	rd	00000000	sel	mfc0 rt,rd
	31:26	25	24:6			5:0	
	010000	1	0000 0000 0000 0000 000			011000	eret

在 maindec.v 和 aludec.v 中对三条指令进行译码，由于三条指令高六位 op 均为 010000，译码时需要使用低六位 func 进行联合判断，如下图所示：

maindec.v：

```
`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
```

aludec.v：

```
`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,在 alu.v 中添加实现代码，如下图所示：

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

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，在 alu.v 中添加实现代码，如下图所示：

```
`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);
```

exception.v 根据 except 信号产生 eret 异常类型输出：

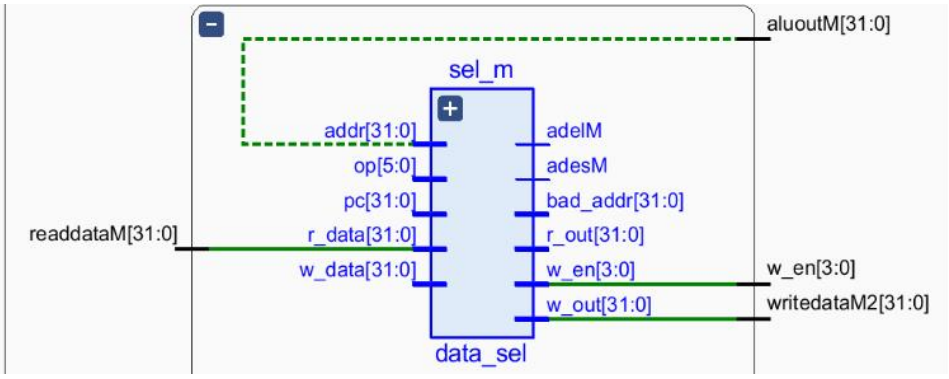
```
else if (except[4]==1'b1) begin
    excepttype <= 32'h0000_000e;
end
```

CP0 寄存器根据异常类型信号处理如下：

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

2.7 Data_sel 模块设计

主要用于处理数据存储器的操作，包括字节选择、地址对齐检查和异常处理。



输入和输出信号说明：

```
module data_sel(
    input wire [31:0] pc,//当前的程序计数器值（用于异常报告）
    input wire [31:0] addr,//目标存储器地址
    input wire [31:0] r_data, //从数据存储器中读出的值
    input wire [31:0] w_data,//要写入数据存储器的值
    input wire [5:0] op,//操作码，用于确定内存访问的类型（如字节、半字或字的加载/存储）
    output reg [31:0] final_readd,//经过字节选择后最终写入regfile的值
    output reg [31:0] final_writed,//经过字节选择后最终要写入data_ram的值
    output reg [3:0] w_en,//写使能信号，用于控制写入哪个字节
    output reg adesM,adelM,//存储地址错误标志,加载地址错误标志
    output reg [31:0] bad_addr //异常地址
    );
```

该模块核心逻辑主要分为以下三个部分：

1. 地址对齐检查（异常处理）：通过操作码 op 对内存地址 addr 进行对齐检查，并设置异常标志 adesM 或 adelM：

```
always @(*) begin
    adesM <= 1'b0;
    adelM <= 1'b0;
    bad_addr <= pc;
    case(op)
        `EXE_LW: begin
            if(addr[1:0]!=2'b00) begin
                adelM <= 1'b1;
                bad_addr <= addr;
            end
        end
        `EXE_SW: begin
            if(addr[1:0]!=2'b00) begin
                adesM <= 1'b1;
                bad_addr <= addr;
            end
        end
        `EXE_LH: begin
            if(addr[0]!=0) begin
                adelM <= 1'b1;
                bad_addr <= addr;
            end
        end
        `EXE_LHU: begin
            if(addr[0]!=0) begin
                adelM <= 1'b1;
                bad_addr <= addr;
            end
        end
        `EXE_SH: begin
            if(addr[0]!=0) begin
                adesM <= 1'b1;
                bad_addr <= addr;
            end
        end
    endcase
end
```

加载操作（EXE_LW、EXE_LH、EXE_LHU、EXE_LB 等）：

检查地址是否符合操作对齐要求，例如：

对于 EXE_LW（加载字），地址必须是 4 字节对齐（addr[1:0] == 2'b00）。

对于 EXE_LH 和 EXE_LHU（加载半字），地址必须是 2 字节对齐（addr[0] == 0）。

如果地址未对齐，则设置 adelM = 1，并将错误地址记录到 bad_addr。

存储操作（EXE_SW、EXE_SH 等）：

检查存储地址是否对齐，例如：

对于 EXE_SW（存储字），地址必须是 4 字节对齐。

对于 EXE_SH（存储半字），地址必须是 2 字节对齐。

如果地址未对齐，则设置 adesM = 1，并记录 bad_addr。

2. 写入数据选择（写使能信号生成）：根据操作码和对齐异常标志生成写使能信号 w_en

和写入数据 final_writed：

```
always @(*) begin
    case (op)
        `EXE_SW:
            begin
                if(adesM==1'b1)
                w_en<=4'b0;
            else begin
            w_en <= 4'b1111;
            final_writed <= w_data;
            end
            end
        `EXE_SB:begin
            if(adesM==1'b1)
                w_en<=4'b0;
            else begin
            case(addr[1:0])
                2'b00:w_en<=4'b0001;
                2'b01:w_en<=4'b0010;
                2'b10:w_en<=4'b0100;
                2'b11:w_en<=4'b1000;
            endcase
            final_writed <= {4{w_data[7:0]}};
            end
            end
        `EXE_SH:
            begin
            if(adesM==1'b1)
                w_en<=4'b0;
            else begin
            case (addr[1:0])
                2'b10: w_en <= 4'b1100;
                2'b00: w_en <= 4'b0011;
                default: w_en <= 4'b0000;
            endcase
            final_writed <= {2{w_data[15:0]}};
            end
            end
        default:begin
         w_en <= 4'b0000;
         final_writed <= w_data;
        end

    endcase
end
```

SW：如果地址未对齐（adesM == 1），写使能设置为 4'b0000，否则写使能为 4'b1111，写入完整的 w_data。

SB：根据地址低两位 addr[1:0] 确定需要写入的字节：

2'b00：写使能 4'b0001（写入最低字节）。

2'b01：写使能 4'b0010（写入次低字节）。

2'b10：写使能 4'b0100（写入次高字节）。

2'b11：写使能 4'b1000（写入最高字节）。

写入数据为 w_data[7:0] 扩展为 4 个相同的字节 {4{w_data[7:0]}}。

SH：根据地址低两位 addr[1:0] 确定需要写入的半字：

2'b00：写使能 4'b0011（写入低两个字节）。

2'b10：写使能 4'b1100（写入高两个字节）。

默认：写使能 4'b0000（不写入）。

写入数据为 w_data[15:0] 扩展为两个相同的半字 {2{w_data[15:0]}}。

3. 读取数据选择：根据操作码和地址低两位选择最终写入寄存器文件的数据 final_readd：

```
always @(*)
if(adelM!=1'b1)
begin
    case (op)
        `EXE_LB:
            case (addr[1:0])
                2'b00: final_readd <={{24{r_data[7]}},r_data[7:0]};
                2'b01: final_readd <={{24{r_data[15]}},r_data[15:8]};
                2'b10: final_readd <={{24{r_data[23]}},r_data[23:16]};
                2'b11: final_readd <={{24{r_data[31]}},r_data[31:24]};
                default: final_readd <= r_data;
            endcase
        `EXE_LBU:
            case (addr[1:0])
                2'b00: final_readd <={{24{1'b0}},r_data[7:0]};
                2'b01: final_readd <={{24{1'b0}},r_data[15:8]};
                2'b10: final_readd <={{24{1'b0}},r_data[23:16]};
                2'b11: final_readd <={{24{1'b0}},r_data[31:24]};
                default: final_readd <= r_data;
            endcase
        `EXE_LH:
            case (addr[1:0])
                2'b00: final_readd <={{16{r_data[15]}},r_data[15:0]};
                2'b10: final_readd <={{16{r_data[31]}},r_data[31:16]};
                default: final_readd <= r_data;
            endcase
        `EXE_LHU:
            case (addr[1:0])
                2'b00: final_readd <={{16{1'b0}},r_data[15:0]};
                2'b10: final_readd <={{16{1'b0}},r_data[31:16]};
                default: final_readd <= r_data;
            endcase
        `EXE_LW:
            final_readd <= r_data;
        default: final_readd <= r_data;
    endcase
end
endmodule
```

LB：根据 addr[1:0] 从 r_data 中提取目标字节，并进行符号扩展：

2'b00：选择 r_data[7:0]，符号扩展为 32 位。

2'b01：选择 r_data[15:8]，符号扩展为 32 位。

2'b10：选择 r_data[23:16]，符号扩展为 32 位。

2'b11：选择 r_data[31:24]，符号扩展为 32 位。

LBU：与 LB 类似，但不进行符号扩展，而是零扩展。

LH：根据 addr[1:0] 提取目标半字，并进行符号扩展：

2'b00：选择 r_data[15:0]。

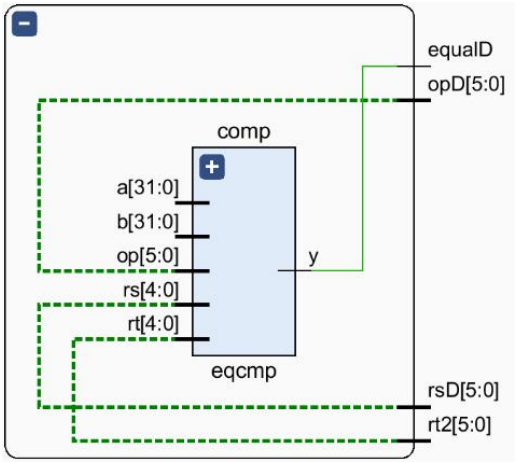
2'b10：选择 r_data[31:16]。

默认：返回原始数据 r_data。

LHU：与 LH 类似，但不进行符号扩展，而是零扩展。

LW：直接返回读取的数据 r_data。

2.8 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

模块的输入输出：

```
module eqcmp(
    input wire [31:0] a,b,
    input wire [5:0] op,
    input wire [4:0] rt,
    input wire [4:0] rs,
    output reg y
    );
```

模块内部的数据通路

模块的核心数据通路是一个 case 语句，通过检查 op（操作码）来判断指令类型，并基于输入信号 a 和 b 的值生成输出 y。以下是数据通路的具体设计：

```
 always @(*) begin
case(op)
`EXE_BEQ:begin
if(rs==5'b0&&rt==5'b0)  //b
y<=1'b1;
else if(a==b) y<=1'b1;
else y<=1'b0;
end

`EXE_BNE:begin
if(a!=b) y<=1'b1;
else y<=1'b0;
end

`EXE_BGTZ:begin
y = ((a[31] == 0) && (a != 32'b0)) ? 1:0;
end

`EXE_BLEZ:begin
y = ((a[31] == 1) || (a == 32'b0)) ? 1:0;
end

6'b000001:case(rt)
`EXE_BLTZ:begin
y = (a[31] == 1) ? 1:0;
end

`EXE_BLTZAL:begin
y = (a[31] == 1) ? 1:0;
end

`EXE_BGEZ:begin
y = (a[31] == 0) ? 1:0;
end

`EXE_BGEZ:begin
y = (a[31] == 0) ? 1:0;
end

`EXE_BGEZAL:begin
if(rs==5'b00000) y<=1'b1;  //bal
else y = (a[31] == 0) ? 1:0;
end
endcase

default:y<=1'b0;
endcase
end

endmodule
```

1.判断条件逻辑：模块通过 case(op) 检测当前指令类型，针对每种指令，检查是否满足条件：

相等比较：BEQ：判断 a == b 是否为真。

不等比较：BNE：判断 a != b 是否为真。

大小关系比较：

BGTZ：判断 a > 0（a[31] == 0 且 a != 0）。

BLEZ：判断 a <= 0（a[31] == 1 或 a == 0）。

BLTZ：判断 a < 0（a[31] == 1）。

BGEZ：判断 a >= 0（a[31] == 0）。

Link 类指令：

BLTZAL 和 BGEZAL：逻辑与 BLTZ 和 BGEZ 类似，同时考虑特殊情况下需要处理的 bal 指令（根据 rs 判断）。

2.特殊字段处理

某些指令（例如 BLTZAL 和 BGEZAL）会根据 rt 和 rs 的值做额外处理：

如果 rs == 5'b00000，则直接跳转（特殊情况，例如 bal 指令）。

通过对 op 和 rt 进一步 case 语句嵌套处理，增加逻辑判断的灵活性。

2.9 Exception 模块设计

excption 模块负责识别除中断异常外的所有异常类型,并产生对应的例外编码 excepttype 给 CP0 模块，使之可以进行精确异常处理，相关信号如下所示：

信号名	功能描述
rst	异常模块的复位信号，用于在复位状态下清除异常类型
ades	数据写入地址错误信号，表明数据写入时发生了地址错误
adel	指令或数据读取地址错误信号，表明取指或取数据时发生了地址错误
except	8 位异常标志信号，表示各种异常条件
cp0_status	CP0 寄存器中的状态寄存器，包含中断和异常处理的状态信息
cp0_cause	CP0 寄存器中的原因寄存器，包含异常发生的原因和中断号信息
excepttype	异常类型输出信号，表示发生的具体异常类型

异常类型如下所示：

异常类型	异常描述
0x01	软硬件中断
0x04	读指令或读数据地址异常
0x05	写数据地址异常
0x08	系统调用
0x09	断点
0x0a	保留指令
0x0c	算数溢出
0x0e	异常处理返回

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：

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.v 部分代码如下：

```
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; //软件中断
        end
        else if(except[7]==1'b1 || adel) begin
            excepttype <= 32'h0000_0004;    //地址错例外（取指或读数据）
        end
        else if(ades) begin//Data address error
            excepttype <= 32'h0000_0005;    //地址错例外（写数据地址）
        end
         else if(except[6]==1'b1) begin
             excepttype <= 32'h0000_0008;    //系统调用例外syscall
         end
         else if (except[5]==1'b1) begin
             excepttype <= 32'h0000_0009;    //断点例外1break
         end
         else if (except[4]==1'b1) begin
             excepttype <= 32'h0000_000e;    //eret
         end

        else if (except[3]==1'b1) begin
            excepttype <= 32'h0000_000a;    //保留1指令例外invalid
        end
        else if (except[2]==1'b1) begin
            excepttype <= 32'h0000_000c;    //整型溢出overflow
        end
    end
```

2.10 Hazard 模块设计

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；以及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：

```
//forwarding sources to D stage (branch equality)
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 forwardcp0E = ((rdE!=0)&(rdE == rdM)&(cp0weM))?1'b1:1'b0;

//forwarding sources to E stage (ALU)
assign forwardaE = ((rsE != 0) && (rsE == writeregM) && regwriteM) ? 2'b10 :
                   ((rsE != 0) && (rsE == writeregW) && regwriteW) ? 2'b01 :
                   2'b00;
assign forwardbE = ((rtE != 0) && (rtE == writeregM) && regwriteM) ? 2'b10 :
                   ((rtE != 0) && (rtE == writeregW) && regwriteW) ? 2'b01 :
                   2'b00;

assign lwstallD = ((rsD == rtE) || (rtD == rtE)) && memtoregE;
assign branchflushD = branchD && !balD;
assign branchstall = ((branchD||jumpD) && regwriteE && (writeregE == rsD || writeregE == rtD) || (branchD||jumpD) && memtoregM && (writeregM == rsD || writeregM == rtD));

assign flushF = ~(i_stall |d_stall) & (flush_except);
assign flushD = ~(i_stall |d_stall)& (flush_except);
 //stalling D stalls all previous stages
 assign flushE = ~(i_stall |d_stall)& (lwstallD ||jumpD||branchstall||flush_except);

 assign stallF = (i_stall |d_stall) | (flush_except?1'b0:(lwstallD|stall_divE|branchstall));
 assign stallD = (i_stall |d_stall) |(lwstallD | branchstall | stall_divE|i_stall |d_stall);
 assign stallE = (i_stall |d_stall) |(stall_divE|branchstall);
 assign flushM = ~(i_stall |d_stall)& (stall_divE|flush_except);

 assign flushW = ~(i_stall |d_stall) & flush_except;
 assign stallM = (i_stall |d_stall) ;
 assign stallW = (i_stall |d_stall) ;
 assign longest_stall = stall_divE| i_stall |d_stall;
```

2.11 HILO 模块设计

HILO 寄存器用于存储乘法和除法等复杂算术运算的结果。对于乘法,HILO 寄存器分别保存乘法结果的高 32 位和低 32 位;对于除法,HILO 寄存器则分别保存除法的余数和商。相关信号如下所示：

信号名	功能描述
hilo_writeD	用于指示在解码阶段是否需要写入 HILO 寄存器。
hilo_writeE	用于在执行阶段传递 HILO 写入信号。
hilo_writeM	用于在内存阶段传递 HILO 写入信号。
hilo_inE	在执行阶段生成的用于写入 HILO 寄存器的数据，包括高位和低位结果
hilo_inM	在内存阶段传递的用于写入 HILO 寄存器的数据。
hilo_outM	从 HILO 寄存器读取的输出数据

we	写使能信号，结合了 hilo_writeM 和 stall_divM 信号，以确定是否允许写入操作。
hi_i	写入 HILO 寄存器高位的数据
lo_i	写入 HILO 寄存器低位的数据
hi_o	输出 HILO 寄存器中高位的数据
lo_o	输出 HILO 寄存器中低位的数据
clk	时钟信号
rst	复位信号

具体的 hilo 寄存器实现代码如下图所示：

```
module hilo_reg(
    input  wire clk,rst,we,
    input  wire [31:0] hi_i,lo_i,
    output wire [31:0] hi_o,lo_o
    );

    reg [31:0] hi, lo;

   always @(posedge clk) begin
      if(rst) begin
          hi <= 0;
          lo <= 0;
      end else if (we) begin
          hi <= hi_i;
          lo <= lo_i;
      end
   end

   assign hi_o = hi;
   assign lo_o = lo;
endmodule
```

在解码阶段，控制单元识别当前指令是否涉及 HILO 寄存器的操作，根据指令类型，生成 hilo_writeD 信号以标记后续的执行阶段可能需要写入 HILO 寄存器。

在执行阶段，运算执行后将计算结果暂存在 hilo_inE 信号中，用于传递给下一个阶段，hilo_writeD 信号通过流水线寄存器传递到 hilo_writeE，用于指示本阶段计算结果准备写入 HILO。

在存储阶段，hilo_inE 通过流水线寄存器传递到 hilo_inM，准备在本阶段更新 HILO 寄存器，hilo_writeE 信号通过流水线寄存器传递到 hilo_writeM, we 信号指示是否在此阶段对 HILO 寄存器进行写操作，we 信号结合了 hilo_writeM 和 stall_divM 信号，根据是否由写入信号以及是否有流水线暂停进行判断是否进行写操作。

设计数据旁路机制，使得在执行阶段能够直接使用最新的 HILO 结果，而不必等到写回阶段，HILO 寄存器的更新在时钟信号的上升沿同步进行，这保证了数据的一致性。通过引入暂停机制和旁路路径，降低了因多周期运算引起的冒险。

设计实现了在不同的流水线阶段对 HILO 寄存器进行控制和数据传递，这样做的好处是允许在不同的流水线周期中有效利用 HILO 寄存器，从而提高整体指令吞吐率。通过在流水线的解码、执行和内存阶段传递控制信号 hilo_writeD、hilo_writeE、hilo_writeM，HILO 寄存器的写入操作被精确控制，确保了在执行阶段对 HILO 寄存器进行写操作的指令不会与前一周期使用 HILO 数据的指令产生冲突；任何需要从 MEM 级进行数据前推的相关指令,都可以快速地从流水线上的 HILO 寄存器中直接读取所需要的操作数，无需等待其写入和再次读取，减少了数据冒险带来的流水线停顿；we 信号可以用于处理由长时间运算带来的流水线暂停问题，在需要时暂停流水线以等待 HILO 寄存器的计算完成，从而避免数据冒险；HILO 寄存器在时钟上升沿被同步更新，数据一致性得到了严格保证，这种同步机制减少了由于非同步更新导致的控制冒险。

2.12 写透 Cache 设计

我们在 Cache 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，连接 AXI，在此基础上进行设计。设计的核心思想是通过合理的流水线和缓存机制来优化数据访问，同时确保数据的一致性和有效性。

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 类 sram 标准，也就是将 inst_req 和 data_req 设为 1，发送请求，利用缓存机制（i_cache 和 d_cache）提高访问速度，等到地址握手后改变状态机，并将流水线暂停。

对于 inst_req（指令请求），首先需要将虚拟地址转换为物理地址。然后，这个物理地址被传递给 i_Cache 来查询是否缓存命中。若 i_Cache 命中，则直接返回数据，若缺失就将 inst_req 传给 cpu_axi_interface ，再通过 AXI 总线请求外部 ram 中的数据。

对于 data_req（数据请求）首先，数据请求会经过 MMU，判断请求是否需要通过缓存。如果请求不需要通过 D_Cache，则通过 bridge_1x2 模块将请求传递给 confreg_data，并直接从外部

存储器请求数据。如果数据需要经过 D_Cache，数据请求会进入 D_Cache，与指令请求处理类似，首先在缓存中查找，若命中则直接返回数据，若未命中则请求外部 RAM。

通过 AXI 总线统一与外部存储器通信，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，并解除流水线的暂停状态，然后由 datapath 把数据读出来。

其中，对于 Cache 的写直达（write-through）策略，若 Cache 行有效（c_valid == 1）且 Cache 行的标签与地址标签相等（c_tag == tag，则命中（hit）。否则，未命中对命中信号取反（miss = ~hit）。

```
wire hit, miss;
assign hit = c_valid & (c_tag == tag);
assign miss = ~hit;
```

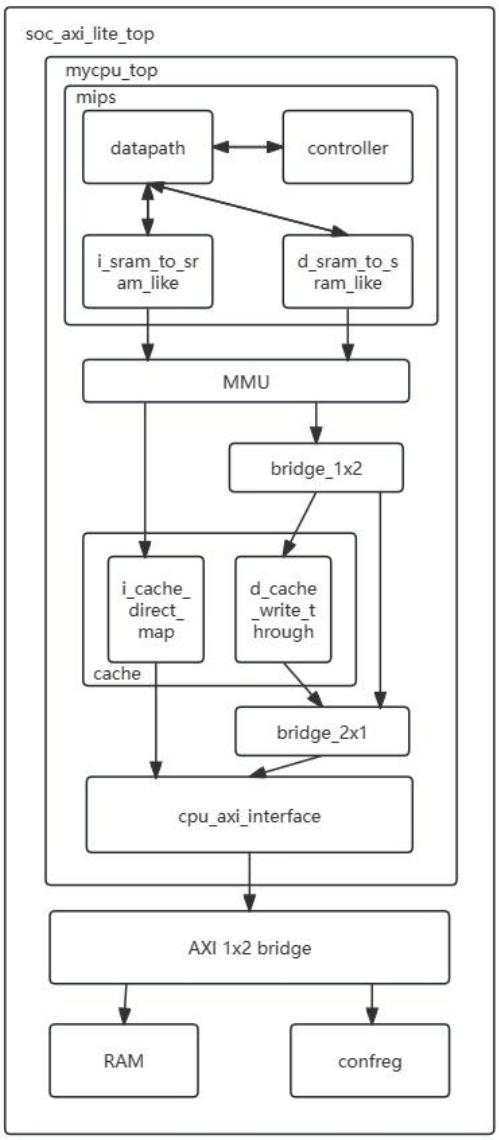
并且定义了 3 种状态：

在 IDLE（空闲状态）时，如果有读未命中请求，则进入 RM 状态，如果是写请求，则进入 WM 状态，如果是读命中，则保持 IDLE 状态。

在 RM（Read Miss）时，处理读未命中状态，向 SRAM 请求数据，如果 SRAM 数据完成传输（cache_data_data_ok），回到 IDLE；读未命中完成时，将 SRAM 返回的数据写入对应的 Cache 行，更新有效位和标签。

在 WM（Write Miss）时，处理写请求，向 SRAM 写入数据。如果 SRAM 写完成 cache_data_data_ok，回到 IDLE。写命中时，更新对应 Cache 行的数据块。

结构图如下：



3 实验过程

3.1 设计工作日志

姚凡：12.23 下载 vivado2019.2，配置实验环境

12.24 了解实验目的和提供的资料包的使用

12.25 观看往届讲解视频，复习 lab4

12.26 观看视频，学习乘法除法相关原理和简单算术指令

12.27 添加部分除乘除法指令外的算术指令

12.28 学习 hilo 寄存器，梳理其和乘除法相关的部分

12.29 继续添加除法相关代码，完善剩余的算数指令

12.30 学习除法涉及的流水线暂停，添加剩余乘除法指令

12.31 完善两条自陷指令，学习 soc、axi

1.1 绘制数据通路草图

1.2 学习 axi，并根据反馈调整数据通路图

1.3 继续细化数据通路图，学习 cache、类 sram 接口的实现

1.4 学习 cache，尝试连接直写 cache

1.5 完善数据通路图，进行完整的功能测试

1.6 完善 cache，并实现性能测试和上板

1.7 完善整个实验设计，确实功能测试、性能测试的实现无误

1.8 回顾设计内容，完善细节问题，准备答辩

1.9 正式答辩

吴明军：12.23 下载 vivado2019.2

12.24 观看 2022 重庆大学硬件综合设计-MIPS 组资源包的使用 视频，学习资源包的使用，配置 verilator 与 gtkwave

12.25 观看往届讲解视频，复习 lab4

12.26 复习流水线和数据通路相关知识

12.27 添加部分访存指令

12.28 添加剩余访存指令

12.29 调试并修改访存指令错误

12.30 学习调准指令数据通路并添加部分跳转指令

12.31 添加剩余分支跳转指令

1.1 调试并修改分支跳转指令错误

1.2 调试并通过 89 条功能测试

1.3 学习 axi

1.4 调整数据通路图

1.5 连接 axi 接口，调试通过部分 axi 功能测试

1.6 调试并通过 axi 剩余功能测试

1.7 对整个项目进行最终测试，确保无误。记录测试结果

1.8 回顾设计内容，准备答辩

1.9 正式答辩

冉倬樾：12.23 配置 wsl，安装 vivado

12.24 观看指导视频，复习 lab4

12.25 学习添加逻辑运算指令

12.26 添加逻辑运算指令

12.27 学习数据移动指令，学习 Hilo 寄存器

12.28 设计 hilo 寄存器并开始添加

12.29 完善 hilo 寄存器

12.30 学习并添加位移指令

12.31 学习并添加特权指令

1.1 绘制数据通路图

1.2 学习 soc

1.3 添加 soc

1.4 学习 axi 并尝试连接

1.5 连接 axi 并进行功能测试

1.6 学习 cache，辅助进行性能测试

1.7 对整个项目进行调试，确保无误

1.8 完善细节，准备答辩

1.9 正式答辩

3.2 主要的错误记录

1、错误 1

（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。

2、错误 2

（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。

3、错误 3

（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。

4、错误 4

（1）错误现象:生成比特流失败

在进行准备上板进行性能分评估的时候，生成比特流时出现多种错误

（2）分析定位过程

根据 vivado 中的 error 报错和 Tcl 控制台中的信息判断可能是与开发板型号不匹配，以及某些操作导致。

（3）错误原因

因为忽略资料包中的 readme 和 log 文档，对顶部模块和约束文件等某些操作未能修改，以及开发板型号不匹配，所以导致多种报错

（4）修正效果

修改对应文件以及运行时开发板型号后。生成比特流成功并且成功上板。

5、错误 5

（1）错误现象:仿真没问题，但是在综合阶段 vivado 出现报错

在通过仿真阶段后，进行下一阶段综合时，vivado 综合失败，出现 error。

（2）分析定位过程

根据报错信息发现是跟文件路径相关。

（3）错误原因

因为文件路径含有中文名称，以及文件路径过长，导致失败。

（4）修正效果

修改文件名称无中文出现，以及将文件路径缩短后，再次综合，问题解决。

6、错误 6

（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 模块中使用 opD 作为输入，使结果提前。应该通过流水线逐级传递 op 信号，datas_sel 位于 MEM 阶段，所以在 data_sel 中应该使用 opM 作为输入，防止流水线混乱。

（4）修正效果

内存读写操作正常，测试点通过

7、错误 7

（1）错误现象：当从内存中读取指定虚拟地址的数据时，数据与之前写入的数据不一致。

（2）分析定位过程

与模版代码仿真波形图进行对比，分析哪里的逻辑有问题

（3）错误原因

在 data_sel 模块中对 w_en 的赋值逻辑有缺陷，没有对地址未对齐的情况进行处理。对 SW,SB,SH 指令添加如下代码，使得在地址未对齐即 adesM 信号为 1 时，禁止写入。

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

（5）修正效果

修正赋值逻辑后，通过测试点

8、错误 8

（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 中读出的第一个操作数为 X，意味着并未从寄存器中读出正确的数值。阅读指令源文件发现前一条指令是 jal 指令，对 31 号寄存器进行了写入操作，而下一条指令为 beq 指令需要在 ID 阶段对 31 号寄存器中的数进行比较，判断是否要执行跳转操作

（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

从 reg 读出的数据正确，不再为 X，测试点通过。

9、错误 9

(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。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 branchstall 置 1 的条件。

```
assign branchstall = ((branchD||jumpD) && regwriteE && (writeregE == rsD || writeregE == 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。

10、错误 10

(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。

```
flopenrc #(32) r1M(clk,rst,~stallM,flushM,srcb2E,writedataM);
flopenrc #(32) r2M(clk,rst,~stallM,flushM,aluout2E,aluoutM);
flopenrc #(5) r3M(clk,rst,~stallM,flushM,writereg2E,writeregM);

flopenrc #(64) r4M(clk,rst,~stallM,flushM,hilo_inE,hilo_inM);
flopenrc #(32) r5M(clk,rst,~stallM,flushM,pcE,pcM);
flopenrc #(6) r6M(clk,rst,~stallM,flushM,opE,opM);
flopenrc #(6) r7M(clk,rst,~stallM,flushM,rdE,rdM);
flopenrc #(1) r8M(clk,rst,~stallM,flushM,is_in_delayslotE,is_in_delayslotM);
flopenrc #(8) r9M(clk,rst,~stallM,flushM,{exceptE[7:3],overflowE,exceptE[1:0]},exceptM);
```

```
flopenrc #(7) regM(
    clk,rst,~stallM,flushM,
    {memtoregE,memwriteE,regwriteE,hilo_writeE,stallE,memenE,cp0weE},
    {memtoregM,memwriteM,regwriteM,hilo_writeM,stall_divM,memenM,cp0weM}
);
```

（4）修正效果

内存读写操作正常，测试点通过。

11、错误 11

（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。

4 设计结果

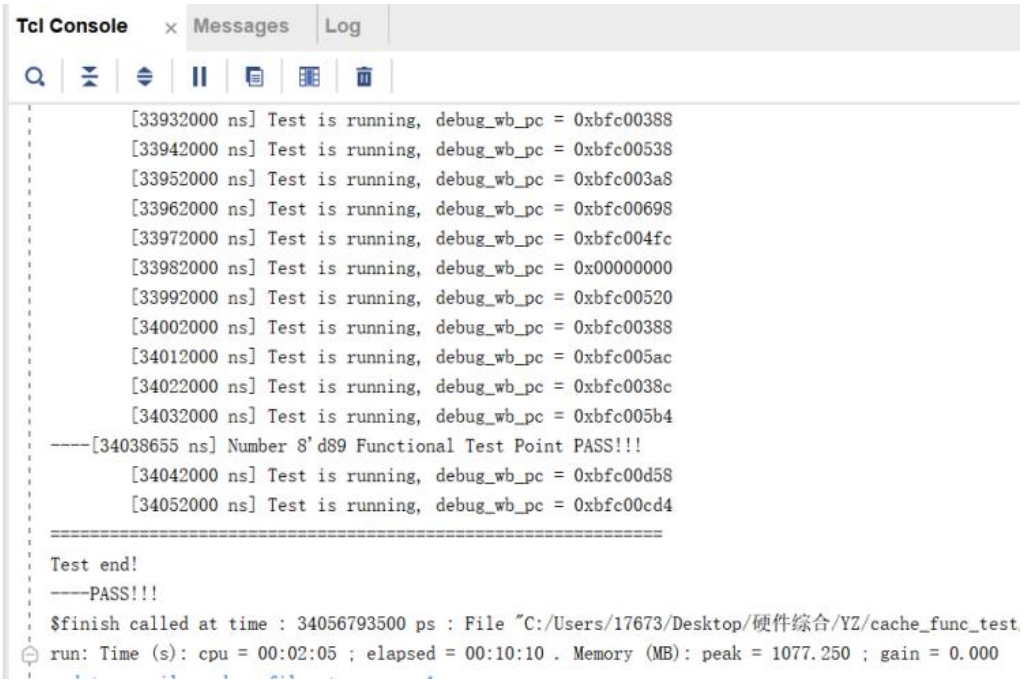
4.1 设计交付物说明

设计交付物包括项目报告、项目代码源文件、bit 流文件（包括功能测试和性能测试）、sorce 性能分表格。

```
├─score.xls
├─ 19-姚凡-吴明军-冉倬樾-实验报告.pdf
|  |
├─bit
|   ├─func.bit
|   └─ perf.bit
|
└─src
   ├─axi
   |   └─mycpu
   |
   └─sram
      └─inst57
         └─mycpu
```

4.2 设计演示结果

4.2.1 功能测试通过 89 个测试点



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 上板计时(16进制) 数码管显示	gs132 上板(16进制) 数码管显示	T_{gs132}/T_{mycpu}
cpu_clk : sys_clk		60MHz : 100MHz	50MHz : 100MHz	-
1	bitcount	15C8BD	13CF7FA	14.55043656
2	bubble_sort	CC6317	7BDD47E	9.696455116
3	coremark	1A03589	10CE6772	10.33712107
4	crc32	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

注1：根据myCPU运行时间填写黄色区域，**不要更改单元格格式**，填写格式为16进制数，分别填写：上板cpu_clk与sys_clk的频率，上板计时；

注2：表格中计时结果为上板时性能测试通过时的数码管显示。

注3：如果某性能测试程序不通过，则对应仿真计时和上板计时项填为0，此时该项与gs132比值按0.1算，最终预赛评分时可能会调整；

注4：灰色部分为固定项，不可修改；

注5：橙色部分为自动计算，不可修改；

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
3. 宏定义参考资料中提供的 define2.vh 文件
4. 除法器在资料包提供的 div.v 文件基础上进行修改优化
5. hilo 寄存器基于资料包提供的 hilo_reg.v 文件上进行修改设计
6. 内存管理模块 mmu 使用资料包提供的 mmu.v 文件
7. AXI 连接参考了 github 上的代码

6 总结和答辩记录

6.1 项目总结

硬件综合设计课程内容丰富,通过循序渐进的方式带领我们小组深入理解和掌握了现代硬件设计的核心概念和关键技能。在整个学习和实践过程中，我们从添加指令入手，逐步掌握了指令的功能、实现方式及其在硬件设计中的具体应用。我们通过设计和实现算数运算指令、逻辑运算指令、分支跳转指令、访存指令、移位指令、数据移动指令、自陷指令与特权指令共 57 条指令，深刻理解了指令在五级流水线架构中的运行机制和实现细节。

在课程中，我们还学习了如何连接 SOC 系统，并成功实现了基于 AXI 协议的连接设计。通过这一部分的学习，我们深入理解了总线协议的实现和在复杂系统中的运用，同时也通过调试 AXI 接口和 SOC 连接，锻炼了硬件系统的调试能力。此外，课程还涵盖了缓存（Cache）设计内容，让我们掌握了如何优化存储器的访问效率与系统性能，并通过实现缓存的功能，进一步体会到硬件设计中性能优化的重要性。

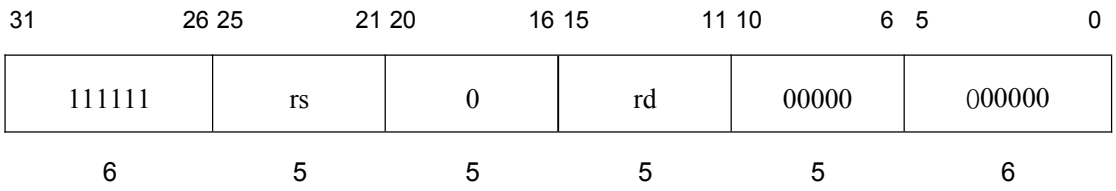
在这一系列实践任务中，我们对 Verilog 硬件描述语言和 Vivado 工具的使用更加熟练，对硬件设计的调试和排错能力也得到了显著提升。从基础的功能模块设计到复杂的系统集成，我们逐步添加指令功能、实现流水线设计、构建完整的硬件架构，在实践中不断提升自己的硬件设计能力。每一次调试、每一个问题的解决都帮助我们积累了宝贵的经验，为未来的硬件开发打下了坚实的基础。

通过这门课程，我们不仅对五级流水线、SOC、AXI 总线等核心概念有了更加全面和深刻的理解，也通过持续实践提升了解决问题和应对复杂设计需求的能力。同时，我们在团队合作中学会了分工与协作，能够高效地完成项目任务。这门课程为我们提供了一个全面的学习平台，让我们在硬件设计领域得到了全方位的锻炼，也为未来的学习和实践积累了信心和宝贵的经验。

6.2 现场添加指令

6.2.1 指令信息和添加过程

ABS



汇编格式： RELU　rd, rt

功能描述： 由op段和func段判断指令，判断rs寄存器的值，若大于等于0则将原值写入rd寄存器中，若小于0则向rd寄存器中写入rs寄存器的绝对值。

操作定义： tmp ← ABS(GPR[rs])

GPR[rd]←tmp31..0

在 maindec.v 和 aludec.v 中进行译码,使用低六位进行联合判断，如下图所示

```
6'b111111:case(funct)
      6'b000000:controls <=12'b110000000010;
      default: begin
                    controls <= 12'b000000000000;
                    invalid <= 1;
              end
endcase
```

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

在 alu.v 中添加实现代码，如下图所示：

```
8'b11111111: y<= ( a[31] == 1 ) ? -a : a; //abs
```

6.2.2 指令测试情况

14：25 开始，14：40 完成

```
----[ 359495 ns] Number 8'd15 Functional Test Point PASS!!!
        [ 362000 ns] Test is running, debug_wb_pc = 0xbfc1f06c
----[ 362935 ns] Number 8'd16 Functional Test Point PASS!!!
----[ 366125 ns] Number 8'd17 Functional Test Point PASS!!!
----[ 367755 ns] Number 8'd18 Functional Test Point PASS!!!
----[ 370145 ns] Number 8'd19 Functional Test Point PASS!!!
        [ 372000 ns] Test is running, debug_wb_pc = 0xbfc252b0
----[ 372545 ns] Number 8'd20 Functional Test Point PASS!!!
================================================================
Test end!
----PASS!!!
```

6.3 现场答辩记录

6.3.1 问题 1

助教：分支指令的实现是通过分支预测还是分支判断?

回答：分支判断

助教：那分支指令是在哪一个阶段进行分支判断的?

回答：译码阶段。译码阶段会对指令进行解码，同时通过专门的模块判断是否满足分支条件，决定下一步的指令流向。在设计中，我们使用了 eqcmp 模块 来完成分支条件的判断。eqcmp 模块根据不同的分支指令操作码和两个操作数的值，判断是否满足分支条件，并通过输出信号 y 表示结果。如果 y=1，表示条件满足，跳转到目标地址；否则，继续顺序执行下一条指令。

6.3.2 问题 2

助教：讲一下 CP0

回答：CP0 包含的寄存器有：count_o 计时器；data_o，用于存放读取的寄存器的数据；compare_o 比较寄存器；status_o 状态寄存器，存储 CPU 运行状态等；cause_o 原因寄存器，中断和异常的原因；epc_o 存放异常的指令地址；config_o 配置寄存器，存放系统配置信息；prid_o 处理器标识寄存器,存放处理器标识信息;badvaddr 存放地址发生错误时的虚拟地址等寄存器。通过 exception 识别异常类型并输出异常类型信号传给 CP0，同时还有记录当前指令地址、是否在延迟槽中，发生地址错误时错误的地址的信号输入 CP0,CP0 根据异常类型等信号对异常进行相应的处理。

6.3.3 问题 3

助教：ERET 有延迟槽吗?

回答：没有

6.3.4 问题 4

助教：精确异常是什么?

回答：发生异常时，不立即进行处理，仅进行标记，统一将处理放到访存或写回阶段。

6.3.5 问题 5

助教：简单说一下 cache?

回答：实现的是一个简单的 i_cache 直接映射和 D_cache 写直达写透缓存，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 类 sram 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，若不经过 cache，则通过 1x2bridge 将请求转给 confreg_data，直接请求外部 Confreg，否则类似 inst_req，经过 D_Cache，后续和 I_Cache 一致。 当外部传回数据的时候，也就是 i_sram_to_sram_like 和 d_sram_to_sram_like 接收到地址

握手后，更改状态机，并将 stall 解除，然后由 datapath 把数据读出来。保证缓存和主存中的数据是一致的。

6.3.6 问题 6

助教：cache 的 hit 是怎么设置的?

回答：每个主存地址根据 index 直接映射到缓存的某一行，每个缓存块对应唯一的主存块，如果多个主存块的 index 相同，它们只能存储在相同的缓存行中，发生替换。其实就是分解主存的地址，提取出 index 和 tag，使用 index 选择对应的缓存行，检查 valid 和 tag 看看是否命中。如果 tag = c_tag，则命中。写入数据时，数据会更新到缓存，也会同步写入主存。

```
wire hit, miss;
assign hit = c_valid & (c_tag == tag);
assign miss = ~hit;
```

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