

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 2、6、2、1 班

年级: 2020

学生: 谢琉晨、曾子瑄、徐小龙、陈鹏宇

学号: 20204249、20204231、20204137、20204227

完成时间: 2023 年 1 月 13 日

成绩:

指导教师: 肖春华

重庆大学教务处制

MIPSSOC 设计报告

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。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

第一阶段 12.27-1.6

谢疏晨：逻辑运算指令、位移指令、算术运算指令（不包括乘除法）

曾子瑄：分支跳转指令

徐小龙：访存指令

陈鹏宇：乘除法、数据移动指令

第二阶段 1.7-1.12

谢疏晨：sram 功能测试

曾子瑄：异常处理与测试

徐小龙：仿真功能测试 debug、两路组相联写回 cache

陈鹏宇：axi 模块、上板验证结果

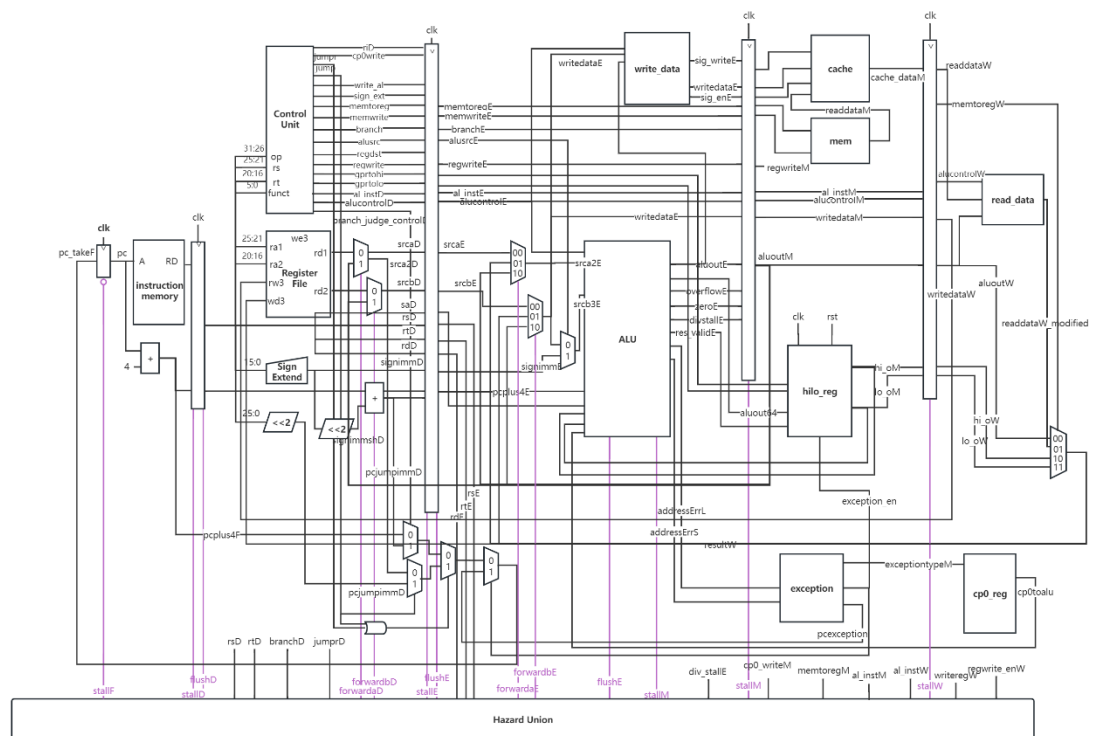
2 设计方案

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。

我们小组基于之前的计算机组成原理实验 4 的五级流水线，依次添加 52 条基本指令，修改数据通路，通过数据前推以及暂停等待处理流水线冒险问题。由于 controller 模块的接口改比较多，我们将 controller 中的 alu 和 aludecoder 合并到 datapath 里面。我们

将根据实验材料的数据通路来编写自己的代码，增加相应的模块，在原来的基础上添加其他模块。最终得到了一下的数据通路图。



2.2 指令设计

2.2.1 逻辑运算指令

逻辑运算指令包括 and、or、xor、nor 和立即数的逻辑运算 andi、xori、lui、ori。对于 and、or、xor、nor 指令，需要在 main_dec 和 alu_dec 模块中用 op 和 funct 两个字段来确定具体指令，然后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，通过 op 字段就能确定。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。

关键代码如下：

maindec.v

```
always@(*)begin
    case(op)
        `EXE_NOP:
            case(funct)
                //logicinstr
                `EXE_AND,`EXE_OR,`EXE_XOR,`EXE_NOR:main_signal<=9'b11000_00_00;//R-type
            endcase
        `EXE_ANDI,`EXE_XORI,`EXE_LUI,`EXE_ORI:main_signal<=9'b10100_00_00;//Immediate
    end
```

```

endcase

aludec.v
always@(*)begin
    case(op)
        //R-type
        `EXE_NOP:case(funcnt)
            //逻辑运算指令
            `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
endcase

alu.v
always@(*)begin
    case(alucontrol)
        `EXE_AND_OP:alu_ans=alu_num1&alu_num2;
        `EXE_OR_OP:alu_ans=alu_num1|alu_num2;
        `EXE_XOR_OP:alu_ans=alu_num1^alu_num2;
        `EXE_NOR_OP:alu_ans=~(alu_num1|alu_num2);
        `EXE_ANDI_OP:alu_ans=alu_num1&{{16{1'b0}},alu_num2[15:0]};
        `EXE_ORI_OP:alu_ans=alu_num1|{{16{1'b0}},alu_num2[15:0]};
        `EXE_XORI_OP:alu_ans=alu_num1^{{16{1'b0}},alu_num2[15:0]};
        `EXE_LUI_OP:alu_ans={alu_num2[15:0],{16{1'b0}}};
    endcase
endcase

```

2.2.2 位移指令

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。

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 writeback 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。

关键代码如下：

```
`EXE_SLL_OP:alu_ans=alu_num2<<sa;
`EXE_SRL_OP:alu_ans=alu_num2>>sa;
`EXE_SRA_OP:alu_ans=$signed(alu_num2)>>>sa;
`EXE_SLLV_OP:alu_ans=alu_num2<<alu_num1[4:0];
`EXE_SRLV_OP:alu_ans=alu_num2>>alu_num1[4:0];
`EXE_SRAV_OP:alu_ans=$signed(alu_num2)>>>alu_num1[4:0];
```

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。我们需要在 maindec 模块中对添加 hilo 相关写信号 gpptohi 和 gpptolo 代表写 hi 位有效和写 lo 位有效，10 代表 mthi，01 代表 mtlo；而读信号只需将 memtoreg 信号拓展为两位，因为写到 GPR 的来源还需考虑是从 hi 或 lo，memtoreg 作为四路选择器的选择信号，选择是哪个值写入 00 代表运算单元计算结果，01 代表读数据内存数据，10 代表 mfhi，11 代表 mflo。为了避免由于 hilo 读写指令带来的新冒险，如 div 后跟 mflo 或 mtlo 后跟 mflo，我们将 hilo 寄存器的写操作放在 EX 阶段，而 hilo 的读操作放在 MEM 阶段，使得读 hilo 时数据已经写入进去，恰好可以解决 hilo 的数据冒险问题。

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；M 阶段会读出 hiloM 的值并传入 alu 中；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。

关键代码如下：

maindec.v

```
`EXE_MFHI:main_signal<=9'b11000_10_00;//hi->gpr
`EXE_MFLO:main_signal<=9'b11000_11_00;//lo->gpr
`EXE_MTHI:main_signal<=9'b00000_00_10;
```

```

`EXE_MTLO:main_signal<=9'b00000_00_01;

alu.v
`EXE_MFHI_OP:alu_ans=hilo[63:32];
`EXE_MFLO_OP:alu_ans=hilo[31:0];
`EXE_MTHI_OP:alu_out_64={alu_num1,hilo[31:0]};
`EXE_MTLO_OP:alu_out_64={hilo[63:32],alu_num1};

```

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：

```

assign overflow_add=((alu_ans[31]&(~alu_num1[31]&~alu_num2[31]))
||(~alu_ans[31]&(alu_num1[31]&alu_num2[31]))&&(alucontrol==`EXE_ADD_OP||alucontrol==`EXE_ADDI_OP);
assign overflow_sub=((alucontrol==`EXE_SUB_OP)&&
((alu_ans[31]&(~alu_num1[31]&~num2_reg[31]))||(~alu_ans[31]&(alu_num1[31]&num2_reg[31]))));
assign overflowE=overflow_add||overflow_sub;

```

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，乘法器的使用，首先通过当前的 alucontrol 判断是有符号还是无符号乘法，再决定是否添加 signed 符号。而除法我们使用的参考的除法器版本。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，因此需要对除法器进行一个简单的多周期握手操作。新增信号 div_valid，div_res_ready，div_res_valid 和 div_stall，考虑到目标组件比较简单，我们省去了 div_ready，div_valid 直接作为除法器的使能信号，只要当前运算指令为有符号或无符号除法，div_valid 即被拉高；div_res_ready 代表 alu 可以接收除法器的计算结果；div_res_valid 代表除法器计算结果已经准备好；div_stall 代表除法器运行阶段的停顿，即除法器使能且结果未出来阶段。

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，考虑到 mtlo 和 mthi 指令定义过 hilo 的写信号，所以乘除法指令也直接将其复用，如果 gp_rtohi 和 gp_rtlo 同时为 1 代表的乘法写入。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。

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 writeback 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。

```
// branch : jump check
assign zeroE = (alucontrol == `EXE_BEQ_OP) ? (alu_num1 == alu_num2): // == 0
              (alucontrol == `EXE_BNE_OP) ? (alu_num1 != alu_num2): // != 0
              (alucontrol == `EXE_BGTZ_OP) ? ((alu_num1[31]==1'b0) && (alu_num1!=32'b0)): // > 0
              (alucontrol == `EXE_BLEZ_OP) ? ((alu_num1[31]==1'b1) || (alu_num1==32'b0)): // = 0
              (alucontrol == `EXE_BLTZ_OP) ? (alu_num1[31] == 1'b1): // < 0
              (alucontrol == `EXE_BGEZ_OP) ? (alu_num1[31] == 1'b0): // >= 0
              (alucontrol == `EXE_BLTZAL_OP) ? (alu_num1[31] == 1'b1): // < 0
              (alucontrol == `EXE_BGEZAL_OP) ? (alu_num1[31] == 1'b0): // >= 0
              (alu_ans == 32'b0);

`EXE_J_OP:      alu_ans = alu_num1 + alu_num2;
`EXE_JR_OP:     alu_ans = alu_num1 + alu_num2;
`EXE_JAL_OP:    alu_ans = pcplus4E + 32'h4;
```



```

`EXE_JALR_OP:   alu_ans = pcplus4E + 32'h4;
//b type
`EXE_BEQ_OP:    alu_ans = alu_num1 - alu_num2;
`EXE_BNE_OP:    alu_ans = alu_num1 - alu_num2;
`EXE_BLTZAL_OP: alu_ans = pcplus4E + 32'h4 ;
`EXE_BGEZAL_OP: alu_ans = pcplus4E + 32'h4 ;

```

2.2.6 访存指令

访存指令在计组四的基础上从 lw(loadword)扩充到 lb,lbw,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(storeword)扩充到 sb,sh,即存储字节,存储半字。

在扩充后的执行逻辑为:对于写操作,用地址最低两位来为字节与半字操作选择写使能信号:

```

// 为 sig_enE、sig_write 信号赋值
always @ (*)
begin
    case (alucontrolE)
        `EXE_LW_OP,`EXE_LB_OP,`EXE_LBU_OP,`EXE_LH_OP,`EXE_LHU_OP:
            // load 指令 使能信号=1, 写使能=0000
            begin
                sig_writeE <= 4'b0000;
                sig_enE <= 1;
            end
        `EXE_SW_OP:
            // SW 使能=1, 写使能=1111
            begin
                case (aluoutE[1:0])
                    2'b00: sig_writeE <= 4'b1111;
                    // 地址有误
                    default: sig_writeE <= 4'b0000;
                endcase
                sig_enE <= 1;
            end
        `EXE_SH_OP:
            // SH 使能=1, 写使能=1100/0011
            begin
                case (aluoutE[1:0])
                    //根据地址末尾两位判断写入的位置
                    2'b10: sig_writeE <= 4'b1100;
                    2'b00: sig_writeE <= 4'b0011;
                    // 地址有误
                    default: sig_writeE <= 4'b0000;
                endcase
            end
    endcase
end

```



```

        endcase
        sig_enE <= 1;
    end
    `EXE_SB_OP:
    // SB 使能=1, 写使能=1000/0100/0010/0001
    begin
        case (aluoutE[1:0])
            //根据地址末尾两位判断写入的位置
            2'b11: sig_writeE <= 4'b1000;
            2'b10: sig_writeE <= 4'b0100;
            2'b01: sig_writeE <= 4'b0010;
            2'b00: sig_writeE <= 4'b0001;
        endcase
        sig_enE <= 1;
    end
    // 其他无关指令 使能=0, 写使能=0000
    default:
    begin
        sig_writeE <= 4'b0000;
        sig_enE <= 0;
    end
end
endcase
end

```

对于读操作，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：

```

// 根据指令读取对应数据
always @ (*)
begin
    case (alucontrolW)
        `EXE_LW_OP:
        // LW 指令 无需修改数据
        readdataW_modified <= readdataW;
        `EXE_LH_OP:
        // LH 指令 根据地址低两位判断，作符号拓展
        begin
            case (dataadrW[1:0])
                2'b10: readdataW_modified <= {{16{readdataW[31]}},readdataW[31:16]};
                2'b00: readdataW_modified <= {{16{readdataW[15]}},readdataW[15:0]};
                default: readdataW_modified <= readdataW;
            endcase
        end
        `EXE_LHU_OP:
        // LHU 指令 根据地址低两位判断，作无符号拓展
        begin
            case (dataadrW[1:0])

```

```

                2'b10: readdataW_modified <= {{16{1'b0}},readdataW[31:16]};
                2'b00: readdataW_modified <= {{16{1'b0}},readdataW[15:0]};
                default: readdataW_modified <= readdataW;
            endcase
        end
        `EXE_LB_OP:
        // LB 指令 根据地址低两位判断，作符号拓展
        begin
            case (dataadrW[1:0])
                2'b11: readdataW_modified <= {{24{readdataW[31]}},readdataW[31:24]};
                2'b10: readdataW_modified <= {{24{readdataW[23]}},readdataW[23:16]};
                2'b01: readdataW_modified <= {{24{readdataW[15]}},readdataW[15:8]};
                2'b00: readdataW_modified <= {{24{readdataW[7]}},readdataW[7:0]};
                default: readdataW_modified <= readdataW;
            endcase
        end
        `EXE_LBU_OP:
        // LBU 指令 根据地址低两位判断，作无符号拓展
        begin
            case (dataadrW[1:0])
                2'b11: readdataW_modified <= {{24{1'b0}},readdataW[31:24]};
                2'b10: readdataW_modified <= {{24{1'b0}},readdataW[23:16]};
                2'b01: readdataW_modified <= {{24{1'b0}},readdataW[15:8]};
                2'b00: readdataW_modified <= {{24{1'b0}},readdataW[7:0]};
                default: readdataW_modified <= readdataW;
            endcase
        end
    endcase
end

```

2.2.7 内陷指令

对于 system 和 break 指令，并不需要添加太多的内容，只需要在译码时这两条指令进行判断，再到 cp0 中做相应寄存器处理，然后将 pc 下一跳的地址改为 bfc00380 异常处理入口地址即可。

2.2.8 特权指令

eret 指令的操作仍然相似，这条指令是再异常处理完成时进行返回的中断指令，同样只需要在译码阶段对指令进行判断，然后在 M 阶段进行处理的时候，再把 epc 的值传入下一跳的 pc，继续执行异常触发前的指令段。

mtc0 和 mfc0 指令是对 cp0 寄存器的值进行数据移动操作, 于是在译码阶段需要加入写 cp0 的控制信号, 然后在 alu 中统一处理, 通过 aluout 进行输出, 再写回 regfile 或者 cp0。此外, 这两个指令前后发生可能会引起数据冒险, 具体在模块设计中详述。

2.2.9 sram 接口 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, 这里使用课程发的 mmu 模块进行地址映射。

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。

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 命令行中查看通过的测试点的个数。

2.2.10 axi 接口 soc 设计

完成 sram 的 89 条功能测试后, 进行 axi 接口连接。我们小组参考了 github 上 sram 转类 sram 的接口完成了 sram 到 sramlike 的转换, 再利用提供的 sramlike 转 axi 的转接桥, 将 cpu 连接上去, 实现完整的 soc_axi 环境, 通过 89 条功能测试。

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 环境中有一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。我们借鉴了计算机体系结构实验二中的相关代码, 通过一个 1×2bridge 和一个 2×1bridge, 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。

2.3 模块设计

2.3.1 datapath 模块

datapath 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。

表格 1datapath 模块接口

信号名	方向	位宽/bit	功能
clk	input	1	时钟信号
rst	input	1	重置信号
int	input	6	中断接口
instrF	input	32	取指阶段取到的指令
readdataM	input	32	从 data_sram 输入的数据
i_stall	input	1	指令访存时的 stall 信号
d_stall	input	1	数据访存时的 stall 信号
inst_enF	output	1	instram 使能
pcF	output	32	当前 pc 地址
sig_writeM	output	1	写内存的写使能信号
memwriteM	output	1	写内存信号
longest_stall	output	1	全局 stall 信号
debug_wb_pc	output	32	写回阶段的 pc 值, debug 用
debug_wb_rf_wen	output	4	写回阶段的存储器使能, debug 用
debug_wb_rf_wnum	output	5	写回阶段的寄存器号, debug 用
debug_wb_rf_wdata	output	32	写回阶段的写存储器数据, debug 用

2.3.2 PC 模块

pc 本质是一个 D 触发器, 如果 reset 信号为 1, 那么就将 pc 置为 0。如果没有 stall, 那么就将输入的 pc 作为下一个 pc 的值。其他情况 pc 不变。sram 和 axi 测试时, 需要将 pc 设为 0xbfc00000。

Table1

信号名	方向	位宽/bit	功能
clk	input	1	时钟信号
rst	input	1	重置信号
en	input	1	pc 使能信号
pcnext	input	32	输入的下一个 pc 地址
pc	output	32	输出的 pc 地址
ce	output	1	pc 使能信号

2.3.3 maindec 模块

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。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。

Table2

信号名	方向	位宽/bit	功能
op	input	6	当前指令的 opcode 字段。
rs	input	5	当前指令的 rs 字段
rt	input	5	当前指令的 rt 字段
funct	input	6	当前指令的 funct 字段
memtoreg	output	2	寄存器数据来源 00→aluresult,01→readdata,10→hi,11→lo
memwrite	output	1	写内存信号
branch	output	1	指令为跳转指令
alusrc	output	1	选择立即数还是 reg 中的值输入 ALU。
regdst	output	1	选择 rd 还是 rt 作为写的目标。
regwrite	output	1	regfile 的写使能，表示是否要写入数据。
gprtohi	output	1	将寄存器值传入高位寄存器的使能信号
gprtolo	output	1	将寄存器值传入低位寄存器的使能信号
al_instD	output	1	判断 jal 等 al 类型的指令的信号
write_al	output	1	al 类型指令写入 31 号寄存器的信号
jump	output	1	立即数跳转信号
jumpr	output	1	寄存器跳转信号
rid	output	1	保留指令控制信号
cp0write	output	1	写入 CP0 信号

2.3.4 aludec 模块

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。

Table3

信号名	方向	位宽/bit	功能
op	input	6	当前指令的 opcode 字段。
rs	input	5	当前指令的 rs 字段
rt	input	5	当前指令的 rt 字段
funct	input	6	当前指令的 funct 字段
alucontrol	output	8	ALU 控制信号。
branch_judge_controlID	output	8	分支跳转信号

2.3.5 alu 模块

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。

Table4

信号名	方向	位宽/bit	功能
clk	input	1	时钟信号
rst	input	1	重置信号
alu_num1	input	32	alu 操作数 1
alu_num2	input	32	alu 操作数 2
alucontrol	input	8	alu 控制信号
hilo	input	64	拓展 64 位高低寄存器
sa	input	5	指令的 10-6 字段
flushE	input	1	刷新运行阶段触发器的信号
stallM	input	1	暂停访存阶段触发器的信号
pcplus4E	input	32	第三阶段的 PC+4 的值。
cp0toalu	input	32	cp0 寄存器输入到 alu 的数据
alu_out	output	32	计算输出
alu_out_64	output	64	乘除法计算输出
overflowE	output	1	计算结果是否溢出
div_stall	output	1	除法暂停
res_validE	output	1	乘除法结果是否有效
addressErrL	output	1	load 地址错误
addressErrS	output	1	store 地址错误

2.3.6 hazard 模块

Table5

信号名	方向	位宽/bit	功能
rsD	input	5	译码阶段的 rs 寄存器号
rtD	input	5	译码阶段的 rt 寄存器号
rdE	input	5	运行阶段 rd 信号
rdM	input	5	访存阶段 rd 信号
branchD	input	1	译码阶段 branch 信号
jumprD	input	1	译码阶段寄存器跳转信号
predict_wrong	input	1	预测地址错误
rsE	input	5	运行阶段 rs 信号
rtE	input	5	运行阶段 rt 信号
writeregE	input	5	运行阶段写寄存器地址
branchE	input	1	运行阶段 branch 信号
regwrite_enE	input	1	运行阶段写寄存器信号

memtoregE	input	2	运行阶段从内存写到寄存器的信号
div_stallE	input	1	运行阶段除法暂停信号
cp0_writeM	input	1	访存阶段是否写 cp0 寄存器的信号
writeregM	input	5	访存阶段写内存地址
regwrite_enM	input	1	访存阶段写寄存器信号
memtoregM	input	2	访存阶段从内存写到寄存器的信号
writeregW	input	5	写回阶段写寄存器地址
regwrite_enW	input	1	写回阶段写寄存器信号
al_instW	input	1	写回阶段 al 类型指令信号
al_instM	input	1	访存阶段 al 类型指令信号
i_stall	input	1	指令访存时的 stall 信号
d_stall	input	1	数据访存时的 stall 信号
exception_en	input	1	异常信号
stallF	output	1	取指阶段 stall 信号
flushF	output	1	取指阶段 flush 信号
forwardaD	output	1	译码阶段数据 1 前推信号
forwardbD	output	1	译码阶段数据 2 前推信号
stallD	output	1	译码阶段 stall 信号
flushD	output	1	译码阶段 flush 信号
forwardaE	output	2	运行阶段数据 1 前推信号
forwardbE	output	2	运行阶段数据 2 前推信号
flushE	output	1	运行阶段 flush 信号
stallE	output	1	运行阶段 stall 信号
forwardcp0E	output	1	运行阶段 cp0 数据前推信号
stallM	output	1	访存阶段 stall 信号
flushM	output	1	访存阶段 flush 信号
stallW	output	1	写回阶段 stall 信号
flushW	output	1	写回阶段 flush 信号
all_stall	output	1	全局 stall 信号

2.3.7cache 模块

实现了两路组相联的写回机制 cache，接口信息如下：

信号名	方向	位宽/bit	功能
clk	input	1	时钟信号
rst	input	1	复位信号
mips core cpu<-->cache			
cpu_data_req	input	1	读写请求信号
cpu_data_wr	input	1	写请求信号
cpu_data_size	input	2	由地址最低两位，确定有效字节长度 (即写掩码)
cpu_data_addr	input	32	数据地址

cpu_data_wdata	input	32	待写入数据
cpu_data_rdata	output	32	待读出数据
cpu_data_addr_ok	output	1	cache 间接传递 内存的地址确认
cpu_data_data_ok	output	1	cache 间接传递 内存的数据确认(已写入/已读出)
axi interface cache<-->mem			
cache_data_req	output	1	运行阶段 branch 信号
cache_data_wr	output	1	运行阶段写寄存器信号
cache_data_size	output	2	运行阶段从内存写到寄存器的信号
cache_data_addr	output	32	运行阶段除法暂停信号
cache_data_wdata	output	32	访存阶段是否写 cp0 寄存器的信号
cache_data_rdata	input	32	访存阶段写内存地址
cache_data_addr_ok	input	1	访存阶段写寄存器信号
cache_data_data_ok	input	1	访存阶段从内存写到寄存器的信号

2.3.8 exception 模块

五条特权指令是用于处理处理器异常的, 在 MIPS 中主要通过协处理器 CP0 来实现对异常的处理。在本设计中, 对于特权指令的支持主要依靠课程资料给出的 CP0 参考代码以及异常处理数据通路。因为 cp0_reg.v 中已经给出较完整的对于常见的 7 种异常中涉及到关于 CP0 寄存器的读写, 因此只需要构建相应的数据通路, 接上 cp0_reg 模块的接口, 就能完成异常处理, 传入该模块的最核心的信号是 excepttype, 就是说要确定异常的类型。在本设计中, 设计了针对于 7 种异常类型都构建了相应的数据通路, 并在流水线的每一级把产生的异常信号向后传递, 直到 M 阶段进行处理。我们设计了 exception 模块来捕获异常, 根据 M 阶段得到的信号, 以组合逻辑的方式输出异常使能信号、异常类型、异常跳转 pc。

信号名	方向	位宽/bit	功能
rst	input	1	复位信号
exception	input	8	异常信号
addrErrL	input	1	取地址错
addrErrS	input	1	写地址错
cp0status	input	32	cp0status 寄存器的输出
cp0cause	input	32	cp0cause 寄存器的输出
cp0epc	input	31	cp0epc 寄存器的输出
exception_en	output	1	是否产生异常
exceptiontype	output	32	异常类型
pcexception	output	32	异常处理 pc

2.3.9 cp0 模块

信号名	方向	位宽/bit	功能
clk	input	1	时钟信号
rst	input	1	复位信号
we_i	input	1	cp0 写控制信号
waddr_i	input	5	写入数据的地址
raddr_i	input	5	读数据的地址
data_i	input	32	输入 cp0 的写数据
int_i	input	6	外部中断
excepttype_i	input	32	异常类型
current_inst_addr_i	input	32	当前指令 pc
is_in_delayslot_i	input	1	指令是否处于延迟槽
bad_addr_i	input	32	触发异常的地址
data_o	output	32	cp0 中读出的数据
count_o	output	32	count 寄存器输出（内部计数器）
compare_o	output	32	compare 寄存器输出（设计中未使用）
status_o	output	32	status 寄存器输出（描述状态与控制）
cause_o	output	32	cause 寄存器输出（描述例外发生原因）
epc_o	output	32	epc 寄存器输出（例外指令 pc）
config_o	output	32	config 寄存器输出（设计中未使用）
prid_o	output	32	prid 寄存器输出（设计中未使用）
badvaddr	output	32	badvaddr 寄存器输出（描述最新地址相关例外的出错地址）
timer_int_o	output	1	时钟中断寄存器（设计中未使用）

具体来说，采用精确异常的思路，在 F 阶段进行指令 pc 的判断，如果读到的 pc 不是按字对齐的，则触发地址错异常，将记录的异常值传入 D 阶段，经过译码可以判断 systemcall、eret、break 三条特权指令，若指令无法正常译码，则触发保留指令异常，在 E 阶段判断溢出和存取地址异常。进入 M 阶段就进行异常的处理，传入 exception 模块产生异常类型等信息，再在 cp0 中进行相应寄存器的读写。

另外值得注意的是，如果在 E 阶段读 cp0，M 阶段写 cp0，如果读写发生在同一地址，那么需要进行数据前推。同时，需要对触发异常的指令是否处于延迟槽做判断，因为延迟槽的指令是必须执行的，所以此时 epc 将会存入延迟槽的上一条分支跳转指令的 pc，保证异常处理完成后能正确执行原指令。

3 设计过程

3.1 设计流水账

3.1.1 个人流水账

Table6

日期	时间	人员	工作内容
20221226	10:00-17:00	谢琉晨	阅读文档，回顾计组实验四，整理思路
20221228	9:00-12:00	谢琉晨	完成 aludec 部分，完成 alu、maindec 部分的逻辑指令、位移指令部分
20221229	10:00-20:00	谢琉晨	完成添加算术运算指令（不包括乘除）、分支跳转指令、访存指令部分（未测试）
20221231	10:00-15:00	谢琉晨	不能仿真，debug
20230101	10:00-20:00	谢琉晨	pc 一直不变，寻找 bug，未解决
20230102	10:00-24:00	谢琉晨	debug，捋清其他数据通路
20230103	10:00-24:00	谢琉晨	解决 pc 不变的问题，在 alu 中添加剩余的指令
20230104	10:00-24:00	谢琉晨	核对实验材料的数据通路图与自己写的数据通路图，debug
20230105	10:00-20:00	谢琉晨	对逻辑运算指令、位移指令、算术运算指令进行功能测试
20230106	15:00-3:00	谢琉晨	修复功能测试的 bug，跑出正确结果
20220107	10:00-23:00	谢琉晨	完成其它基本指令，开始功能测试，未测试成功
20220109	15:00-22:00	谢琉晨	修复 maindec 中的 bug
20230110	10:00-18:00	谢琉晨	开始写报告
20230110	19:00-2:00	谢琉晨	辅助 debug，通过 sram 前 40 个测试点
20230111	14:00-15:30	谢琉晨	辅助 debug，通过 sram 41-65 个测试点
20230111	15:30-1:00	谢琉晨	编写报告
20230112	17:00-2:00	谢琉晨	完善报告
20230113	11:00-17:00	谢琉晨	完善报告

日期	时间	人员	工作内容
20221227	9:00-17:30	徐小龙	下载实验资料，阅读实验材料与要求
20221228	9:00-17:30	徐小龙	根据讲解视频配置实验环境，并测试运行
20221229	9:00-17:30	徐小龙	回顾计组实验四，整理思路
20221230	9:00-17:30	徐小龙	熟悉 gitee 使用方法
20221231	9:00-17:30	徐小龙	阅读指导文档，熟悉访存指令逻辑
20230101	9:00-17:30	徐小龙	编写访存指令所需的读处理模块
20230102	9:00-17:30	徐小龙	编写访存指令所需的写处理模块
20230103	9:00-17:30	徐小龙	译码器对应部分以及 ALU 对应部分
20230104	9:00-17:30	徐小龙	调整数据通路(sig_write 信号传递、写入数据传递等)
20230105	9:00-17:30	徐小龙	继续调整数据通路(readdataW_modified 等)
20230106	9:00-17:30	徐小龙	修改访存使能信号(原本恒定为 1)

20220107	9:00-24:00	徐小龙	进行独立功能测试，基本完成(sw 指令存在多占用一周期的情况)
20220109	9:00-17:30	徐小龙	debug(sw 指令多占用一周期)，通过独立测试
20230110	10:00-24:00	徐小龙	新建 func_test 分支，进行 52 条功能测试 debug，通过前 22 测试点
20230110	10:00-24:00	徐小龙	52 条功能测试 debug，通过前 40 测试点
20230111	10:00-14:30	徐小龙	52 条功能测试 debug，修复 NOP 指令引起 branchstall 的 bug
20230111	14:30-2:00	徐小龙	52 条功能测试 debug，通过前 64 测试点 辅助 axi 接口功能测试 debug
20230112	10:00-20:00	徐小龙	整理 57 条 sram 功能测试通过版本，发布分支 func_sram_pass 57 条 axi 接口功能测试 debug(修改 al_instW 的使能信号)，通过
20230112	20:00-24:00	徐小龙	整理代码，删除冗余信号，准备添加 cache
20200113	0:00-2:00	徐小龙	添加两路组相联 cache，并修改部分注释，撰写部分文档

日期	时间	人员	工作内容
20221227	10:00-17:00	曾子瑄	阅读指导文档，学习指导视频，掌握了解硬件综合设计整体流程与工作
20221228	9:00-12:00	曾子瑄	合并计组实验四中 datapath 与 controller 模块
20221229	10:00-19:00	曾子瑄	完成分支跳转指令的添加，未测试
20230101	10:00-23:59	曾子瑄	修复 IP 核错误设置导致第一条指令执行两周期的 bug
20230103	10:00-20:00	曾子瑄	完成分支跳转指令的单元测试，修改 hazard 模块
20230104	10:00-24:00	曾子瑄	协助队友进行调试，分析问题
20230105	10:00-22:00	曾子瑄	去掉分支预测模块，修改分支跳转指令支持
20230106	10:00-22:00	曾子瑄	连接 sram 接口，进行功能测试
20230107	11:00-21:00	曾子瑄	添加 5 条异常处理指令
20230108	10:00-24:00	曾子瑄	解决异常指令测试中无法冲刷触发异常指令后面执行指令的 bug
20220109	10:00-23:00	曾子瑄	完成异常指令的功能测试
20220110	14:00-20:00	曾子瑄	协助进行 axi 接口测试
20230112	10:00-22:00	曾子瑄	写报告

日期	时间	人员	工作内容
20221227	9:00-17:30	陈鹏宇	下载实验资料，阅读实验材料与要求
20221228	9:00-17:30	陈鹏宇	根据讲解视频配置实验环境，并测试运行
20221229	9:00-17:30	陈鹏宇	回顾计组实验四，整理思路
20221230	9:00-17:30	陈鹏宇	阅读文档
20230103	9:00-17:30	陈鹏宇	添加乘除法，未通过独立测试
20230104	9:00-17:30	陈鹏宇	处理除法器异常，未解决，先改为组合逻辑
20230105	9:00-17:30	陈鹏宇	补充数据移动指令
20230106	9:00-17:30	陈鹏宇	Debug 数据移动，通过独立测试
20220107	9:00-19:30	陈鹏宇	阅读多周期握手文档，改写多周期除法器，通过独立测试
20220109	9:00-17:30	陈鹏宇	协助队友完善通路问题
20230110	12:00-24:00	陈鹏宇	阅读 axi 接口和类 sram 接口文档
20230111	10:00-24:00	陈鹏宇	尝试接入 axi，失败
20230112	10:00-24:00	陈鹏宇	Debug axi_func，通过 89 条功能测试点

20230113	10:30-4:00	陈鹏宇	完善报告
----------	------------	-----	------

3.1.2 gitee 提交记录

Jeffrey / MIPS_CPU

Watching 4
Star 2
Fork 0

代码

Issues 0

Pull Requests 0

Wiki

统计

流水线

服务

管理

func_test
分支 6
标签 0

+ Pull Request
+ Issue
文件
Web IDE
克隆/下载

当前分支与 master 相比，领先 70 个 Commit，落后 17 个 Commit。

贡献代码
同步代码

159*****52
删除冗余文件
82ada50
35分钟前

245 次提交

rtl
删除冗余文件
35分钟前

LICENSE
t
11天前

README.md
t
11天前

README.md

MIPS_CPU

RTL源码仓库

搜索分支

分支 (6)
管理

func_test

axi_pass

func_axi_pass

func_sram_pass

master

test_branch

简介

重庆大学硬件综合设计项目——基于MIPS32整数指令集精简版的五级流水线CPU设计

暂无标签

Verilog 等 2 种语言

GPL-3.0

发行版

暂无发行版，创建

贡献者 (4)

全部

B

C

1

近期动态

34分钟前推送了新的提交到 func_test 分支，2854bd9...82ada50

1小时前推送了新的提交到 func_test 分支，cc06b98...2854bd9

1小时前推送了新的提交到 func_test 分支，8e0afff...cc06b98



3.2 错误记录

3.2.1 错误 1

(1)错误现象:仿真时 pc 一直不变。

(2)分析定位过程:按实验资料的数据通路进行修改代码, 添加模块, 改动的地方挺多的, pc 信号传递得有点复杂。于是重新分析了一下正常的计组实验 4 的 pc 数据通路和自己修改后的代码, 请教了其他组正常的 pc 通路。都没发现通路有啥问题。最后在 pc 模块的使能信号上发现了原因。

(3)错误原因:pc 模块中, 由于要把 stallID 传给 pc, 作为使能信号, 在 datapath 里面取了 ~stallID, 但是在 pc 里面也对使能信号取反, 导致一直为 0。

(4)修正效果:在 pc 模块中, clear 为 1, 则清零。传入~stallID, 之后正常。

3.2.2 错误 2

(1)错误现象:波形图中, 有很多信号是 xxx。

(2)分析定位过程:查找第一个出现 xxx 的地方, 把相关的信号找出来, 一步一步往前找, 找第一个出现 xxx 的地方。

(3)错误原因:发现是 alu 里面的输入为 xxx, 代码编写的问题。由于有多个同学修改同一个文件, 添加指令的时候添加到了相同的指令, 没有测试就直接上传到 gitee 上, ①有两行代码给同一个信号赋值, ②某些信号, 比如在不同模块里面, regdst 信号的长度不一样。

(4)修正效果:删去多余的代码, 注意定义信号的问题, 之后就能正常显示。

3.2.3 错误 3

(1)错误现象:第 1 个测试点报错。功能测试时, pc 比测试点的参考 pc 慢一个周期。

(2)分析定位过程:最开始的时候, pc 就比参考值慢。根据 pc 的数据通路, 找到 pc 最开始变化的地方。

(3)错误原因:rst 变为 1 时,开始执行程序,此时先将信号 ce 置为 1,再让 ce 去启动 pc 这里就相当于延迟了一个周期。

(4)修正效果:删掉 ce 后,pc 值就能和测试点的 pc 值对得上。

3.2.4 错误 4

(1)错误现象:执行第一条指令的结果正常,但是后面的指令结果为 xxx。

(2)分析定位过程:冒险冲突的问题。检查了前后指令,发现是在先写后读的时候,还没把数据写到寄存器,就开始读寄存器的数据出来,此时读出来的是 xxx。

(3)错误原因:数据前推的信号没接好,一直是 x。

(4)修正效果:对数据进行数据前推,原来只有 forwardaD、forwardbD,需要加上 forwardaE、forwardbE,在译码和执行阶段加上数据前推,就可以解决数据冲突的问题。

3.2.5 错误 5

(1)错误现象:执行第一条指令的结果正常,但是后面的指令结果为 xxx。

(2)分析定位过程:冒险冲突的问题。检查了前后指令,发现是在先写后读的时候,还没把数据写到寄存器,就开始读寄存器的数据出来,此时读出来的是 xxx。

(3)错误原因:数据前推的信号没接好,一直是 x。

(4)修正效果:对数据进行数据前推,原来只有 forwardaD、forwardbD,需要加上 forwardaE、forwardbE,在译码和执行阶段加上数据前推,就可以解决数据冲突的问题。

3.2.6 错误 6

(1)错误现象:第 40 个测试点,BGEZAL 指令对应的 pc 值很奇怪。

(2)分析定位过程:到 alu 中查看 EXE_BLTZAL_OP 对应的操作,
 $alu_ans = pcplus4E + 32'b100$;此时的 pc 本应该是 $pcplus4E + 4$,但是波形图的结果和 pc 相差很大。

(3)错误原因:猜测是 $32'b100$ 与 $pcplus4E$ 相加时把 1 当作符号位,出了问题。由于测试的时候忘记记录具体数字了,想不起来当时波形图的 pc 值是多少。暂时还没搞清原因。

(4)修正效果:将 $32'b100$ 改成 $32'h4$,pc 正常。

3.2.7 错误 7

(1)错误现象:还是第 40 个测试点,计算出的跳转 pc 值正常,但是跳转的时候慢了一个周期。

(2)分析定位过程:查看波形图,发现在跳转之前有连续两个 stall,导致跳转地址慢了一个周期。这里本应只需要一个 stall。

(3)错误原因:jal、beqzal 指令遇到需要数据前推,之前的写寄存器指令来不及将结果前推,需要 stall 的情况,会有两个周期都有写寄存器操作,会有误判。

(4)修正效果:在 maindec 里面添加 al_instD 信号,作为 jal、bnezal 等 al 类型的指令的信号。在遇到连续两个写信号的时候,第二次写入的时候关闭写使能,方便 pc 提前跳转。

如果不需要数据前推，会误伤。如果不需要数据前推 forwardaD，就不传给 stallE，ID→EX 阶段，该信号的传递方式为 $al_instE \leftarrow al_instD \& forwardaD$ 。此时能通过测试。

3.2.8 错误 8

- (1)错误现象:功能测试 243340ns 的时候出现迭代次数超过上限的错误。
- (2)分析定位过程:出错的指令为判断一个数是否小于 0。之前的指令涉及减法。查找减法相关的信号，发现了问题。
- (3)错误原因:在判断一个数小于 0 的时候，数据流没问题，但是 alu 中 $num2_reg \leq 0$ 出现了问题。
- (4)修正效果:这时候应该是组合逻辑，而不是时序逻辑。应该将 \leq 变成 $=$ 。修改之后，结果就正常。

3.2.9 错误 9

- (1)错误现象:测试到 nop 指令的时候，出现问题。
- (2)分析定位过程:查找判断 nop 指令的条件，发现 maindec 里面并没有专门判断 nop 指令。因为 nop 指令是将 0 写入 0 寄存器，相当于 sll 指令。
- (3)错误原因:最开始把 nop 指令当做 sll 指令处理，写寄存器的使能信号为 1，但是 0 寄存器的结果永远是 0，可以直接读，分支跳转 branchstall 不需要变成 1，不需要数据前推。
- (4)修正效果:在冒险冲突模块，在 branchstall 添加新的信号 $\&(writeregE! = 5'b00000)$ ，当写入的寄存器为 0 寄存器时，不跳转。此时正常跳转，nop 指令的测试通过。

3.2.10 错误 10

- (1)错误现象:第 57 条指令，BGEZAL 指令出错。
- (2)分析定位过程:此时又涉及到 stall 的问题。在测试时，cpu_inst_wen 信号为 0，本不应该触发测试判断，但是在不该触发的时候触发了，而且后面的 pc 也不正确。
- (3)错误原因:对于 al 类型的指令，需要有特殊的 stall 处理。原来是 $al_instE \leftarrow al_instD \& forwardaD$ ， $al_instM \leftarrow al_instE$ ，需要添加其他条件。
- (4)修正效果:最终改成 $al_instE \leftarrow al_instD \& stallID$ ， $al_instM \leftarrow al_instE \& forwardaD$ ，然后这个测试点通过，指令全都能对得上，一气呵成直接通过 64 个测试点。

3.2.11 错误 11

- (1)错误现象:第 41 测试点，sram 测试通过，但是 axi 不能通过。
- (2)分析定位过程:对比 debug 和 ref，得出 B 指令未跳转
- (3)错误原因: al_instM to al_instW ，使能信号一直为 1，导致 B 类指令得到目标 pc 无法 stall，从而导致后续跳转失败
- (4)修正效果:正常跳转且通过 89

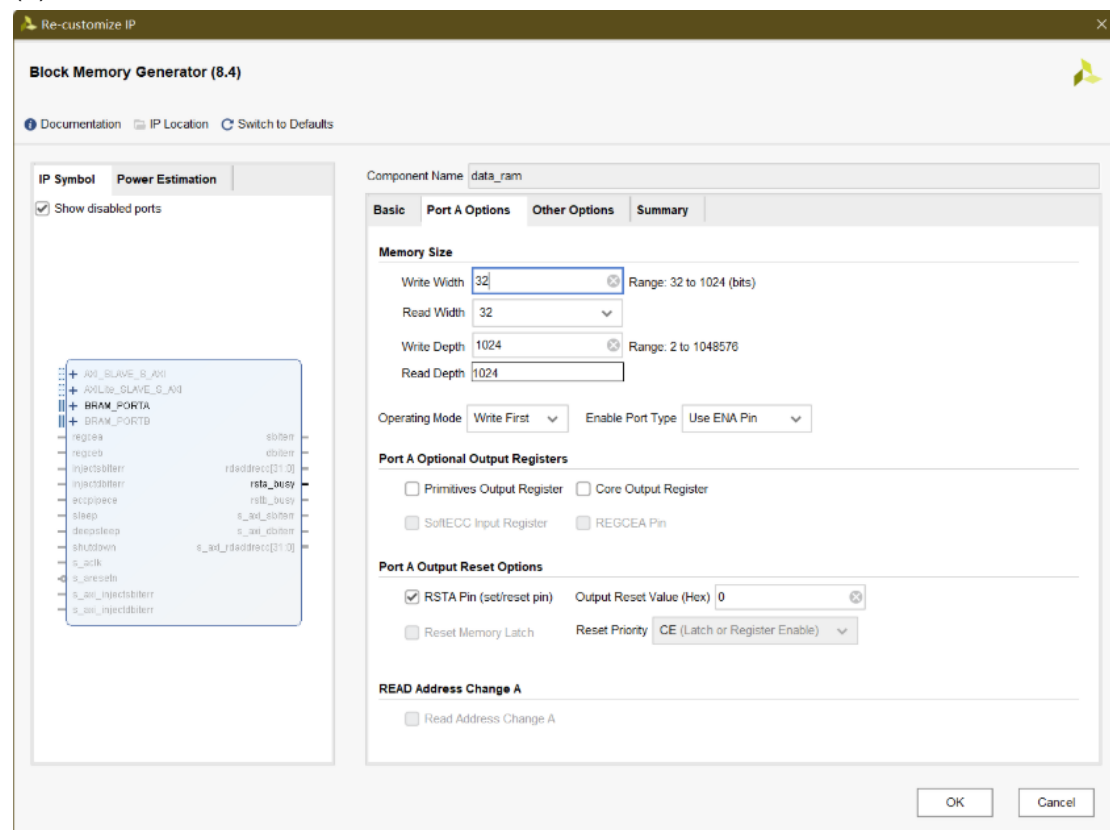
3.2.12 错误 12

(1)错误现象: ip 核设置问题, 困扰很久, 导致输出慢了一拍

(2)分析定位过程: 排除所有可能结果, 检查代码都没有问题, 想到可能是 ip 核的问题, 于是与队友核对, 发现果然有一处设置不同。

(3)错误原因：当初发现 ip 核有一个 `rsta_busy` 信号没有连上，也不需要这个信号，于是便勾选了 `core output register` 选项，把这个输出信号取消掉，但是没想到这样会使内部处理逻辑改变，引起了第一条指令的输出会延长一拍。

(4)修正效果: 能正常进行取指令



3.2.13 错误 13

(1)错误现象：功能测试 debug，pc2040ns 才变化，本来该 2030ns 处变化，延迟了一个周期。

(2)分析定位过程: 根据 pc 下一跳不断往前回溯, 观察几个 pc 中间值以及相应控制信号是否正确, 最后在 pc 模块中发现, rst 信号复位 ce 信号, ce 信号控制了 pc 的复位, 导致 pc 晚了一拍。

(3)错误原因:如上。

(4)修正效果: 取消 ce 对 pc 的复位, 能在正常周期取到 pc 值。

3.2.14 错误 14

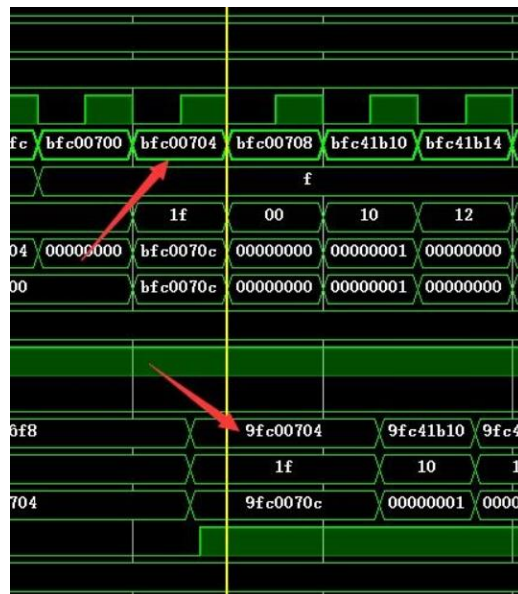
(1)错误现象：出现 jr 指令不跳转的情况

(2)分析定位过程：观察如下波形图可见并没有正常跳转，通过检查 jump 控制指令发现，在选择 jump pc 时控制信号有错误

(3)错误原因：是因为之前加了分支预测，而 cpu132 的 reference 里面没有，导致有些指令提前，去掉分支预测之后，导致之前 jump 和 jumpr 的逻辑消失，因此改变如下选择信号，使 jumprD 能正常跳转。

```
mux2 #(32) pcmux(pcnxtbrFD, pcjumpD, jumpD | jumprD, pcnextFD);
```

(4)修正效果：能正常跳转。



3.2.15 错误 15

(1)错误现象：异常测试中，从同一个地址读取刚才写入该地址的值，结果竟然与写入不一致。

(2)分析定位过程：lw 指令出错，于是在测试汇编文件中，找到上一条在该地址写入的指令，发现是一条系统调用后面的指令。

(3)错误原因：原来中间有另外一条指令写了该地址，这条指令位于 syscall 指令的后一个位置，然而 M 阶段处理系统调用例外，竟没有把错误执行的几条指令刷掉，再更改了 flush 信号的控制逻辑后仍不奏效，排查发现是数据通路中流水线之间信号传递有一部分根本没有通过 flush 信号控制，在仔细检查后问题解决。

(4)修正效果：正常冲刷了流水线，lw 读取正确

3.2.16 错误 16

(1)错误现象：异常测试中，处理系统调用，第一条指令 mfhi 出错，取的 hi 寄存器的值是错误的。

- (2)分析定位过程: 首先找到 hilo 寄存器的写入位置, 再找到上一条写入 hilo 寄存器的指令。
- (3)错误原因: 系统调用指令后面执行了一条 div 指令, div 指令的结果会写入 hilo, 这个过程是不应该发生的, 应该被 flush 掉, 但原来的 hilo 是在 datapath 中的一个单独模块实现的。
- (4)修正效果: 把异常使能添加进该模块, 控制异常时 hilo 信号不进行传递, 问题得以解决。

3.2.17 错误 17

- (1)错误现象:写 hilo 寄存器慢了一周期
- (2)分析定位过程:观察波形图, 与预期延迟一周期
- (3)错误原因:最开始在 MEM 阶段传入写 hilo 值, 因为没有考虑到写入寄存器再读出会延迟一周期, 导致 hilo 读出的结果时上一周期传入的值, 会导致数据冒险, 需在 EX 阶段传入写 hilo 值。
- (4)修正效果:写 hilo 在 EX 阶段, 读 hilo 在 MEM 阶段。

3.2.18 错误 18

- (1)错误现象:数据移动指令中, 写 GPR 位置出错
- (2)分析定位过程:观察波形图, 写寄存器并没有写到对应位置, 且修改了零号寄存器值, 得出两个问题, 写位置出错, 寄存器堆逻辑有问题。
- (3)错误原因:写位置出错是因为 regdst 信号与 mips 信号相反, 导致写位置取的是 rt 的值即为 0; 而零号寄存器被修改是因为没有固定零号寄存器, 没有对写位置进行判断。
- (4)修正效果:零号寄存器固定, mfhi、mflo 指令写位置正常

3.2.19 错误 19

- (1)错误现象:改写多周期除法器时, 在正确结果出来前一直对 hiloreg 进行写
- (2)分析定位过程:观察波形图, hilo 寄存器中在除法器运行期间值一直在变化
- (3)错误原因:除法器的握手信号并没有传到 hiloreg 模块, 导致 hiloreg 一直写使能有效。将 res_validE 信号传入 hilofile 模块, 只有当其有效时才对其写使能。但考虑到 mthi、mtlo 指令 res_validE 始终为 0, 所以需要通过 gphtohi、gprtolo 来辅助写使能。
- (4)修正效果:只会写入除法最终结果。

3.2.20 错误 20

- (1)错误现象:第 1 个测试点, 接上 axi 接口后无法跑通

(2)分析定位过程:观察波形图可以发现, 是 lw 指令出错, 读出的数据与 ref 不同。通过对波形图追踪相关信号, 从 datapath 中的 aluout 到 core 中的 data_sram_addr, 再到 mycpu_top 中的 data_addr, 最终发现 top 中并没有接上 mmu 导致读出来的数据不正确。

(3)错误原因:mmu 未接通

(4)修正效果:通过 axi_func40 个测试点

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, 并通过上板进行了功能验证。

4.1 设计交付物说明

说明所提交设计的目录层次, 各目录下对应的内容是什么。提供所提交设计进行仿真、综合、上板演示的必要操作提示步骤。

4.1.1 目录层次

```
--mycpu
|--RTL
|   |--ascii
|   |   |--defines2.vh    #指令译码定义
|   |   |--instdec.v      #指令 ascii 码译码模块
|   |--adder.v            #加法器
|   |--alu.v              #计算单元
|   |--aludec.v           #alu 译码器
|   |--branch_judge.v     #判断分支指令
|   |--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            #cache 数据通路
|   |--core.v             #数据握手
|   |--cp0_reg.v          #cp0 寄存器
|   |--cpu_axi_interface.v #cpu 的 axi 接口
|   |--d_cache_WB_2way.v  #数据 cache, 直接映射+写回+两路组相联
|   |--d_cache.v          #数据 cache 顶层模块
|   |--d_sram2sraml.v     #数据地址握手
|   |--datapath.v         #数据通路文件
|   |--defines.vh         #指令中间编码定义
|   |--div.v              #除法器
|   |--exception.v       #异常处理模块
|   |--flopenr.v          #使能触发器
|   |--flopenrc.v         #使能清零触发器
|   |--hazard.v           #冒险冲突模块
|   |--hilo_reg.v         #高低寄存器
```

```

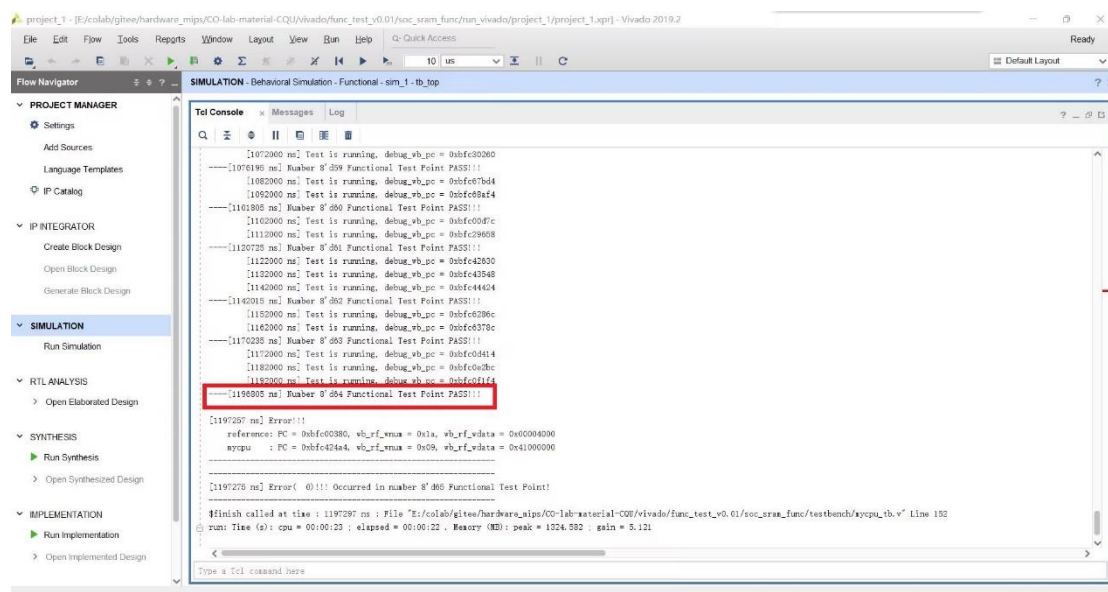
|--i_cache_WB_2way.v #指令 cache, 直接映射+写回+两路组相联
|--i_cache.v         #指令 cache 顶层模块
|--i_sram2sraml.v    #指令地址握手
|--maindedc.v        #主译码器
|--mmu.v             #地址转换
|--mul.v             #乘法器
|--mux2.v            #二选一选择器
|--mux3.v            #三选一选择器
|--mux4.v            #四选一选择器
|--mycpu_top_axi.v   #cpu-axi 接口
|--mycpu_top.v       #cpu 模块
|--pc.v              #pc 寄存器
|--read_data.v       #修正读出的数据
|--regfile.v         #寄存器堆
|--signext.v         #有符号拓展
|--sl2.v             #左移两位
|--top.v             #顶层模块
|--write_data.v      #修正需要写入的数据

```

4.2 设计演示结果

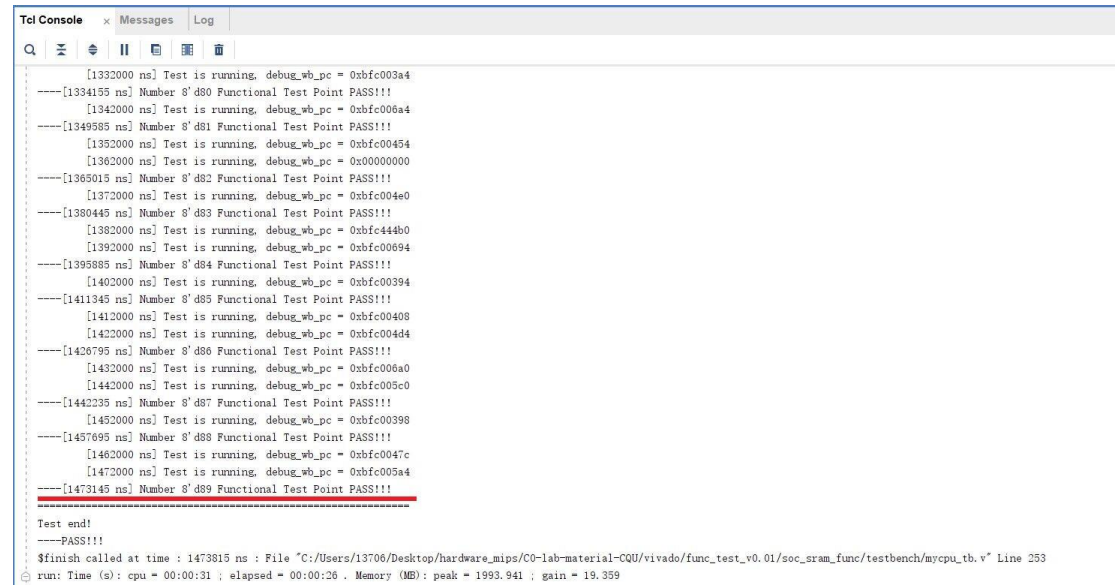
4.2.1 Sram 接口：52 条指令，通过 64 个测试点

第一次功能测试时，经过了晚上接近 6 小时的 debug 之后，通过了 40 个测试点。第二次功能测试时，经过了 1 小时的 debug，通过了剩余的 41-64 个测试点。



4.2.2 Sram 接口：57 条指令，通过 89 个功能测试点

通过 52 条基本指令的测试之后，当天晚上进行异常处理部分的功能测试，通过了 sram 的所有测试点。

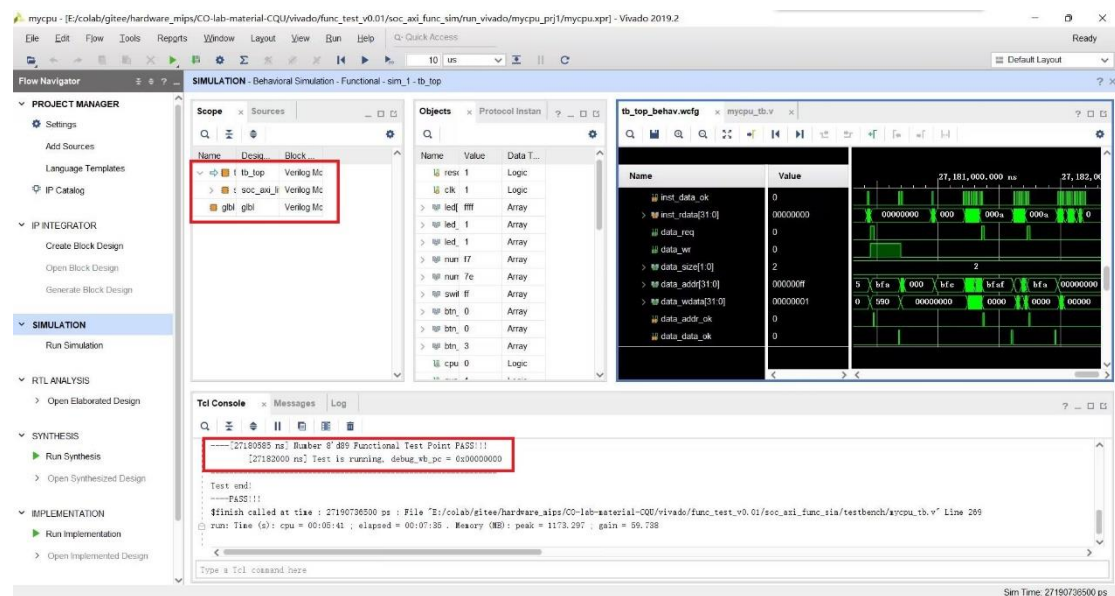


```
Tcl Console x Messages Log
[1332000 ns] Test is running, debug_wb_pc = 0xbfc003a4
----[1334155 ns] Number 8'd80 Functional Test Point PASS!!!
[1342000 ns] Test is running, debug_wb_pc = 0xbfc006a4
----[1349585 ns] Number 8'd81 Functional Test Point PASS!!!
[1352000 ns] Test is running, debug_wb_pc = 0xbfc00454
[1362000 ns] Test is running, debug_wb_pc = 0x00000000
----[1365015 ns] Number 8'd82 Functional Test Point PASS!!!
[1372000 ns] Test is running, debug_wb_pc = 0xbfc004e0
----[1380445 ns] Number 8'd83 Functional Test Point PASS!!!
[1382000 ns] Test is running, debug_wb_pc = 0xbfc044b0
[1392000 ns] Test is running, debug_wb_pc = 0xbfc00694
----[1395885 ns] Number 8'd84 Functional Test Point PASS!!!
[1402000 ns] Test is running, debug_wb_pc = 0xbfc00394
----[1411345 ns] Number 8'd85 Functional Test Point PASS!!!
[1412000 ns] Test is running, debug_wb_pc = 0xbfc00408
[1422000 ns] Test is running, debug_wb_pc = 0xbfc004d4
----[1426795 ns] Number 8'd86 Functional Test Point PASS!!!
[1432000 ns] Test is running, debug_wb_pc = 0xbfc006a0
[1442000 ns] Test is running, debug_wb_pc = 0xbfc005c0
----[1442235 ns] Number 8'd87 Functional Test Point PASS!!!
[1452000 ns] Test is running, debug_wb_pc = 0xbfc00398
----[1457695 ns] Number 8'd88 Functional Test Point PASS!!!
[1462000 ns] Test is running, debug_wb_pc = 0xbfc0047c
[1472000 ns] Test is running, debug_wb_pc = 0xbfc005a4
----[1473145 ns] Number 8'd89 Functional Test Point PASS!!!

Test end!
----PASS!!!
$finish called at time : 1473815 ns : File "C:/Users/13706/Desktop/hardware_mips/CO-lab-material-CQU/vivado/func_test_v0.01/soc_sram_func/testbench/mycpu.tb.v" Line 253
run: Time (s): cpu = 00:00:31 ; elapsed = 00:00:26 ; Memory (MB): peak = 1993.941 ; gain = 19.359
```

4.2.3 通过 AXI 接口+两路组相联写回 cache 功能测试

添加 axi 接口，成功解决接线的 bug 之后，我们成功跑通的 axi 功能测试。并在跑通之后，加上了体系结构编写的两路组相联写回 cache 部分代码。下图为 axi+cache 的运行结果。



https://github.com/TheRainstorm/PiplineMIPS/tree/master/src/PipelineMIPS_only_axi 中的接口，而之后类 sram 转 axi 的桥是直接引用课程中提供的转接桥。

5.4 Cache 连接的 axi 接口

本项目借鉴了计算机体系结构中的 cache 实验，并结合 https://github.com/TheRainstorm/PiplineMIPS/tree/master/src/PipelineMIPS-sram_like 中实现接口完成 cache 的接入。

6 现场添加指令和答辩记录

6.1 现场添加指令

+func_1 和 trace_1 供接上SOc的工程使用，可使用trace对比并且上版。

+test_1 供没有接上Soc的同学使用，需要自行看仿真结果对比，汇编文件里面的结果。

MAX

31	26	25	21	20	16	15	11	10	6	5	0
111111			rs		rt		rd		00000		000000
6			5		5		5		5		6

汇编格式: **MAX rd, rs, rt**

功能描述: 由op段和func段判断指令，将寄存器rs的值与寄存器rt的值比较大小，值较大的写入寄存器rd中。

操作定义: $tmp \leftarrow MAX(GPR[rs], GPR[rt])$

$GPR[rd] \leftarrow tmp_{31:0}$

6.1.1 分工情况

谢琉晨主要负责添加指令，曾子瑄补充 alu 中正数、负数减法的细节。

6.1.2 完成情况

在原有的数据通路的基础上，在 defines.vh 中添加指令的标识符。MAX 指令的 op 字段为 111111，与基本指令的 op 字段对比之后，该指令不会与其他指令冲突。按要求来，

本应在判断时，现根据 **op** 来判断指令类型，再根据 **funct** 字段判断具体哪种指令。但是在不冲突的情况下，我们直接使用 **op** 字段来判断这是 **MAX** 指令。

该指令需要读寄存器、写寄存器，数据来源是寄存器，所以 **maindec** 中，将写寄存器使能信号设为 1；数据写回地址 **regdst** 设为 1，表示写回到 **rt** 中；操作数都来源于寄存器，**alusrc** 设为 0；其它字段都为 0。

在 **alu** 译码器上加上判断，指明这是哪种指令。在 **alu** 中具体实现，输入的操作数有 3 种情况：①正数与负数，根据符号位直接判断。②都是正数，用 **num1** 减 **num2**，如果结果为正，则取 **num1**，如果结果为负，则取 **num2**。③都是负数，两个数相减之后，需要取反，再按②的方式判断。

6.2 现场答辩记录

6.2.1 问题 1

在 **alu** 模块中，如果输出错误，有哪些信号可能会出现问题？

谢琉晨回答：在 **alu** 中，如果计算结果出错，①可能原因是输入的操作数 **num1** 和 **num2** 是 **xxx**，从寄存器中读入的是 **xxx**，没有进行数据前推。②也可能是运算过程中，实现的运算代码有问题。③可能是 **maindec** 里面的信号赋值有问题，导致传进来的 **num1** 和 **num2** 是错的。④在进行指令判断的时候，**alucontrol** 信号的赋值错误也会出现问题。

6.2.2 问题 2

添加指令的时候，需要修改哪些模块？

谢琉晨回答：添加指令时，首先要在 **maindec** 里面针对不同的指令设置相应的操作码。**aludec** 的判断方式和 **maindec** 一样，根据不同的指令设置相应的指令信号。最后在 **alu** 里面实现这条指令的具体功能。

6.2.3 问题 3

问题：pc 的来源有哪些

曾子瑄回答

答：pc 的主要来源大体上可以分为四类，第一类是正常取指的 **pc+4**，第二类是 **branch** 类指令的 **pc+4+offset**，就是延迟槽指令 **pc** 加上地址偏移，第三类是 **jump** 类指令，**jump** 类指令的下一跳 **pc** 来源又分为两种，一种是指令中直接给出的立即数左移两位与 **pc** 的高四位进行拼接，另外一个来源比如 **jr** 指令，是来自寄存器，最后一类就是异常处理的 **pc**，也有两种情况，一种是异常处理地址 **bfc00380**，一种是 **eret** 指令退出异常处理时，从 **epc** 寄存器中取出的发生异常的指令地址。

6.2.4 问题 4

问题：branch 在 D 阶段会不会 stall

曾子瑄回答

答：branch 在 D 阶段是不会 stall 的，我们的项目是把 branch 跳转判断放在 D 阶段来做的。

6.2.5 问题 5

问题：什么是延迟槽？异常处理的分支跳转跟普通的跳转有什么区别？

曾子瑄回答

答：延迟槽就是转移指令的下一条指令，地址就是当前转移指令的 pc+4。异常处理的分支跳转与普通的跳转最大的区别就是延迟槽，异常处理是没有延迟槽的，比如 systemcall 指令跳到异常处理地址 bfc00380，在 systemcall 后面执行的几条指令都会被刷掉，不会执行，包括在 eret 指令异常退出的时候，也是没有延迟槽的，而普通的分支跳转延迟槽是必须执行的。还有就是跳转的 pc 不同，大概就是这些区别。

6.2.6 问题 6

访存中 en 与 wen 信号有什么作用？

徐小龙：en 是内存模块的使能信号，当指令为 load/save 这类访存指令时拉高，其余无关指令置 0；wen 是内存模块的两位写使能信号，当指令不是 save 指令时置 0，是 save 指令时，根据是字、半字还是字节操作以及对应地址的末尾两位来为写使能信号赋值，从而控制写入数据的位置。

6.2.7 问题 7

运行 Load 指令时，数据地址正确，但是得到的数据不正确，需要检查什么？

徐小龙：首先需要查看内存的使能信号是否被拉高，倘若未拉高则无法读出数据；

若使能信号正常，鉴于地址正确，应查看之前对于该地址的 save 操作中写入的数据是否有误，以及写入操作是否有效。

6.2.8 问题 8

在乘除法器阶段遇到过什么问题吗？

陈鹏宇回答：一开始在接入除法器时，由于没有了解多周期的握手机制，导致除法结果一直异常，由于花费太多时间，就先用 verilog 的除法通过独立测试。后面看到了多周期握手的文档，了解其中信号机制后，成功接上多周期除法器。后面遇到的问题主要是在乘法写 hilo 寄存器时，由于没有对 hilo 进行握手，导致每个时钟周期都会写 hilo，将 div_res_ready 信号引入 hilo 中，作为其写使能信号后，能保证在最后一个周期写入正确的除法结果。

6.2.9 问题 9

你能讲一下 axi 中的信号吗？

陈鹏宇回答：其实我没有怎么取了解 axi 信号，我是直接使用的类 sram 转 axi 的接口。类 sram 信号主要大致分为 req, wr, size, addr, wdata, addr_ok, data_ok 和 rdata。就读事务来说，首先需要拉高 req 信号，放入 size, addr, wdata 后等待上层传来的 addr_ok 信号，表示地址握手成功。之后拉低 req 和 addr_ok，等待 data_ok 信号和读取到的值，完成数据握手，至此读事务结束。

6.2.10 问题 10

在实现 axi 时遇到过什么问题吗？

陈鹏宇回答：在接 axi 接口时没有遇到什么问题，但在第一次仿真的时候出错，当时花了很长时间。主要问题是当时是 lw 指令，但是 load 的值不对，我们主要观察波形图，一直沿着该指令的通路最后找到顶层的 data_addr 不对应，是因为没有接上 mmu 模块导致读取错误。

7 供同学们吐槽之用。有什么问题都可以直接写在这。

7.1 徐小龙

- 1.GTKwave 运行不稳定，win11 下仍会无法打开
- 2.（与实验无直接关系）gitee 协同开发时，中文注释时常乱码
- 3.vivado 中更换 coe 文件有时并不生效

7.2 谢琉晨

1、软件环境问题，vivado 编译慢，界面清晰，功能强大。verilator 编译快，但是功能测试的时候，只显示三行错误，需要另外查找 bug，特别麻烦，需要很多额外操作。最后还是选择了 vivado 进行仿真。

- 2、线上一个人写代码效率特别低，感觉线下一起讨论的效率会更高一点。

7.3 曾子瑄

工具链的问题不再多说，暂且说一下对硬综课程的建议吧，我觉得课程任务设置还不够平滑，导致项目初期开展工作比较困难，同时在任务的量化上可能不够精确，包括有些 bug 可能会卡个一两天，这样对实际项目的推进和进度的把握还是有一定的影响。最后希望硬件综合设计课程建设越来越好，通过这样一次实践，对计算机专业的学生的知识体系构建还是很有帮助的，整体收获很大。

8 参考文献

参考文档

- [1] A03_“系统能力培养大赛”MIPS 指令系统规范_v1.01.pdf;
- [2] 指令及对应机器码_2018.pdf;
- [3] 《计算机组成原理实验指导书》,重庆大学计算机学院编;
- [4] 《SOC 结构介绍》
- [5] 《MIPS 基准指令集手册》
- [6] 《功能测试说明》
- [7] 《性能测试说明》
- [8] 《超标量处理器设计》, 姚永斌著, 清华大学出版社