

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 01 班

年 级: 2019

学 生: 魏才讓 宁可馨

学 号: 20194181 20194201

完 成 时 间: 2022 年 1 月 9 日

成 绩:

指 导 教 师: 肖春华

重庆大学教务处制

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

指导教师评定成绩:

指导教师签名:

MIPS SOC 设计报告

魏才讓、宁可馨

1 设计简介

1.1 小组分工说明

- 魏才讓: 负责分支跳转指令、访存指令、内陷和特权指令扩展以及 SRAM 接口的 SOC 封装和上板调试。
- 宁可馨: 负责逻辑运算指令、移位指令、数据移动指令和算数运算指令扩展以及实验报告撰写。

2 设计方案

2.1 总体设计思路

本次硬件综合设计在计算机组成原理实验四实现的简易 5 级流水线 CPU 的基础上, 由 10 条指令扩展为 57 条指令。其中包括算数运算指令、逻辑运算指令、移位指令、分支指令、数据移动指令、访存指令共 52 条基础指令, 以及自陷指令、特权指令共 5 条特殊指令。本次设计基于基础数据通路, 以添加逻辑运算指令、移位运算指令、数据移动指令、算数运算指令、分支跳转指令和访存指令的顺序添加 52 条基础指令并进行必要的数据通路和控制信号调整; 进行 funcTest independent 测试并确保 52 条扩展指令完成后, 添加 5 条特殊指令。之后进行 SRAM 接口的 SOC 封装并进行功能测试和上板演示。

2.2 指令添加设计

本节将主要介绍各类指令添加过程中, 对数据通路和控制信号量的设计和更改, 整体模块设计将在下节详述。

2.2.1 逻辑运算指令

对八条逻辑运算指令, 计组实验四对 and 和 or 指令进行了实现, 需要继续添加 xor、nor 和 andi、xori、lui、ori 六条指令。

对于普通逻辑运算指令 xor、nor, 其数据通路 with 计组实验四实现的 and、or 指令相同, 无需调整; 其控制信号所做改动如下: 在 main_decode 模块中添加这两条指令的控制信息, 在

alu_decode 模块中添加相应的 alu_control 信号并在 alu 模块中根据 alu_control 信号执行相应操作。

对于立即数逻辑运算指令 andi、ori、xori 和 lui, 其数据通路需要重点关注无符号扩展的实现, 其控制信号需要添加对符号扩展类型的判断: 在 main_decode 模块中添加 sign_extD 信号量, 若为 1 则为有符号扩展。又因为这四条指令的操作码均为 0011xx, sign_extD 信号量的值可以由操作码来确定。在数据通路中, 根据 sign_extD 信号量的值, 即可进行立即数扩展。除此之外, 同样需要像普通逻辑运算指令的添加一样进行 main_decode、alu_decode、alu 模块的相关修改。

以上设计的关键代码如下:

(1) main_decode 模块:

```
module main_decode(
    ...
    input wire [31:0] instrD,
    output wire      sign_extD,          // 立即数是否为符号扩展
    ...
);
...
assign opcode = instrD[31:26]; // 操作码
assign rs = instrD[25:21]; // rs
assign rt = instrD[20:16]; // rt
assign funct = instrD[5:0]; // funct
assign sign_extD = ~(opcode[5:2] ^ 4'b0011);
assign {reg_write_enD, reg_dstD, alu_imm_selD, mem_to_regD, mem_read_enD, mem_write_enD} = sigs;
...
always @(*) begin
    case(opcode)
        'EXE_R_TYPE:
            case(funct)
                ...
                'EXE_AND, 'EXE_NOR, 'EXE_OR, 'EXE_XOR: begin
                    sigs = 7'b1_00_0_000;
                end
                ...
            endcase

        'EXE_ANDI, 'EXE_LUI, 'EXE_XORI, 'EXE_ORI: begin
            sigs = 7'b1_01_1_000;
        end
    end
    ...
end
```

(2) alu_decode 模块:

```
module alu_decode(
    ...
    input wire [31:0] instrD,
    output reg [4:0] alu_controlD,
    ...
);
...
always @* begin
    case(opcode)
        'EXE_R_TYPE:
            case(funct)
                // 逻辑运算
                'EXE_AND:      alu_controlD <= 'ALU_AND; //1
                'EXE_OR:       alu_controlD <= 'ALU_OR;
            endcase
    endcase
end
```

```

        'EXE_XOR:      alu_controlD <= 'ALU_XOR;
        'EXE_NOR:      alu_controlD <= 'ALU_NOR;
        ...
    endcase

    'EXE_ANDI:  alu_controlD <= 'ALU_AND;
    'EXE_XORI:  alu_controlD <= 'ALU_XOR;
    'EXE_LUI:   alu_controlD <= 'ALU_LUI;
    'EXE_ORI:   alu_controlD <= 'ALU_OR;
    ...

```

(3) alu 模块

```

module ALU (
    ...
    input wire [4:0] alu_controlE, //alu 控制信号
    input wire [31:0] src_aE, src_bE, //操作数
    output wire [63:0] alu_outE, //alu 输出
    ...
);

always @(*) begin
    case(alu_controlE)
        ...
        'ALU_AND:      alu_out_simple = src_aE & src_bE;
        'ALU_OR:       alu_out_simple = src_aE | src_bE;
        'ALU_NOR:      alu_out_simple =~(src_aE | src_bE);
        'ALU_XOR:      alu_out_simple = src_aE ^ src_bE;

        'ALU_LUI:      alu_out_simple = {src_bE[15:0], 16'b0}; //取高16位
        ...
    endcase
end

```

(4)datapath 立即数扩展

```

...
// 立即数扩展 1-有符号扩展 0-无符号扩展
assign immD = sign_extD ? {{16{instrD[15]}}, instrD[15:0]}:{16'b0, instrD[15:0]};
...

```

2.2.2 移位运算指令

对六条移位运算指令, 需要实现三条立即数移位运算指令: SLL、SRL 和 SRA, 三条变量移位运算指令: SLLV、SRLV 和 SRAV, 其中 SLL 和 SRL 指令将 rt 寄存器中的值逻辑左/右移 sa 位并将结果保存到 rd 寄存器中, SRA 指令将 rt 寄存器的值算术右移 sa 位并将结果保存到 rd 寄存器中; SLLV 和 SRLV 将 rt 寄存器中的值逻辑左/右移位并将结果保存到 rd 寄存器中, SRAV 将 rt 寄存器中的值算术右移并将结果保存到 rd 寄存器中, 这三条指令的移位位数由 rs 寄存器中值的第 0-4bit 决定。

对立即数移位运算指令, 其数据通路需要将 sa 信号从指令的 [10:6] 位读出并传入 alu 模块, 其控制信号和计算功能进行 main_decode、alu_decode、alu 的常规改动即可。

对变量移位指令, 其数据通路和基本数据通路一致, 无需调整, 其控制信号和计算功能调整同样只需要进行常规改动。

以上设计的关键代码如下：

(1)main_decode 模块：

```
module main_decode(
    ...
    input wire [31:0] instrD,
    output wire      sign_extD,          //立即数是否为符号扩展
    ...
);
...
assign opcode = instrD[31:26]; //操作码
assign rs = instrD[25:21]; //rs
assign rt = instrD[20:16]; //rt
assign funct = instrD[5:0]; //funct
assign {reg_write_enD, reg_dstD, alu_imm_selD, mem_to_regD, mem_read_enD, mem
_write_enD} = sigs;
...
always @(*) begin
case(opcode)
    'EXE_R_TYPE:
        case(funct)
            ...
            'EXE_SLLV, 'EXE_SLL, 'EXE_SRAV, 'EXE_SRA, 'EXE_SRLV, 'EXE_SRL : begin
                sigs = 7'b1_00_0_000; //算数运算指令
            end
            ...
        endcase
    end
    ...
end
...
```

(2) alu_decode 模块：

```
module alu_decode(
    ...
    input wire [31:0] instrD,
    output reg [4:0] alu_controlD,
    ...
);
...
always @* begin
case(opcode)
    'EXE_R_TYPE:
        case(funct)
            //移位指令
            'EXE_SLL:      alu_controlD <= 'ALU_SLL_SA;
            'EXE_SRL:      alu_controlD <= 'ALU_SRL_SA;
            'EXE_SRA:      alu_controlD <= 'ALU_SRA_SA;
            'EXE_SLLV:     alu_controlD <= 'ALU_SLL;
            'EXE_SRLV:     alu_controlD <= 'ALU_SRL;
            'EXE_SRAV:     alu_controlD <= 'ALU_SRA;
            ...
        endcase
    ...
end
...
```

(3) alu 模块

```
module ALU (
    ...
    input wire [4:0] alu_controlE, //alu 控制信号
    input wire [31:0] src_aE, src_bE, //操作数
    output wire [63:0] alu_outE, //alu输出
    ...
);
...
always @(*) begin
    case(alu_controlE)

```

```

...
`ALU_SLL:      alu_out_simple = src_bE << src_aE[4:0]; //移位src a
`ALU_SRL:      alu_out_simple = src_bE >> src_aE[4:0];
`ALU_SRA:      alu_out_simple = $signed(src_bE) >>> src_aE[4:0];

`ALU_SLL_SA:   alu_out_simple = src_bE << sa; //移位sa
`ALU_SRL_SA:   alu_out_simple = src_bE >> sa;
`ALU_SRA_SA:   alu_out_simple = $signed(src_bE) >>> sa;
...
endcase
end

```

(4) datapath sa 信号量与 alu 模块的连接

```

...
assign saD = instrD[10:6]; //从指令中读取sa
...
//将sa的值从D阶段传递到E阶段
D_to_E D_to_E(
    ...
    .saD(saD),
    ...
    .saE(saE),
    ...
);
//将sa信号接入alu
ALU alu0(
    ...
    .sa(saE),
    ...
);

```

2.2.3 数据移动指令

对数据移动指令, 需要实现 MFXX 类型指令: MFHI 和 MFLO 以及 MTXX 类型指令: MTHI 和 MTLO。其中 MFHI 和 MFLO 指令将 HI/LO 寄存器中的值写入到寄存器 rd 中, MTHI 和 MTLO 指令将 rs 寄存器中的值写入到 HI/LO 寄存器中。

对 MFXX 类型指令, 在原数据通路的基础上, 寄存器写回值来源变为 alu 计算输出、读内存结果和 hilo 寄存器读取结果。需要在 main_decode 模块中添加 hilo_to_reg 信号量进行最终的写回值选择。同时, 与寄存器堆类似, hilo 寄存器读取结果来自于 hilo 寄存器, 因此在数据通路中需要添加 hilo 寄存器的实现, 需要将指令传入 hilo 寄存器并根据操作码和功能码确定是否读取 HILO 寄存器以及读取 HI 寄存器还是 LO 寄存器, 并需求输出相应寄存器中的值。

对 MTXX 类型指令, 将 rs 寄存器中的值写入 HI/LO 寄存器时, 需要保证另一个寄存器的值不被改变, 同时后续算数指令中的乘除指令同样需要用到 HILO 寄存器, 所以在我们设计的数据通路中, HILO 的写入值来源于 alu 模块的处理, 其具体实现将在指令数据流中详述; 因而就控制信号而言, 需要在 alu_decode 模块中添加相应 alu_control 信号量, 同时需要在 main_decode 模块中根据指令操作码和功能码进行 HILO 寄存器写使能判断。

对 HILO 寄存器实现, 其输入信号有时钟、置位、写使能信号、指令和存入 hilo 的数据, 输

出信号为读寄存器值。其中存入 hilo 的数据为 64 位,这是因为在我们的设计中 hilo 寄存器存储 64 位数据,高 32 位为 hi 寄存器的值,低 32 位为 lo 寄存器中的值。当置位信号为 1 时,将 hilo 中的数据置 0,当写使能信号为 1 时,说明执行 MTXX 指令要进行写寄存器操作,直接将存入 hilo 的 64 位数据写入 hilo 寄存器中。同时,要通过传入指令的操作码和功能码确定是否是 MFXX 指令以及读取什么寄存器,并据此输出寄存器高 32 位值或低 32 位值。

上述存入 hilo 的 64 位数据由 alu 模块确定,其逻辑为:若 alu 接收到的 alu_control 信号量代表 MTHI 指令,则将 rs 寄存器中的值作为高 32 位与 LO 寄存器中的值进行拼接作为 alu 输出传入 HILO 寄存器模块;若 alu 接收到的 alu_control 信号量代表 MTLO 指令,则将 rs 寄存器中的值作为低 32 位与 HI 寄存器中的值进行拼接作为输出传入 HILO 寄存器模块。

基于上述实现,MTXX 类型指令的实现过程为:在 IF 阶段取指;在 ID 阶段译码确认 hilo 寄存器的写使能信号 hilo_wenD,生成 alu 的控制信号 alu_controlD;在 EX 阶段根据从 rs 中读取的或者经过数据前推的 src_aE 与 alu_controlE 生成写入 HILO 寄存器中的值 alu_outE 并根据写使能 hilo_wenE 写入。MFXX 类型指令的实现过程为:在 IF 阶段取指;在 ID 阶段译码确认寄存器写回值的选择信号 hilo_to_regD;在 MEM 阶段根据指令从 hilo 中读数据并根据 hilo_to_regM 确定寄存器写回值;在 W 阶段将数据写入寄存器 rd 中。

以上设计的关键代码如下:

(1)main_decode 模块:

```
module main_decode(
    ...
    input wire [31:0] instrD,
    output reg      hilo_wenE,
    output reg      hilo_to_regM,
    ...
);
...
assign opcode = instrD[31:26]; //操作码
assign funct = instrD[5:0]; //funct

assign hilo_wenD = ~(|(opcode ^ 'EXE_R_TYPE)) & (~(|(funct[5:2] ^ 4'b0110)
    ) // div divu mult multu
    |( ~(|(funct[5:2] ^ 4'b0100)) & funct[0]) //mthi mtlo
);
assign hilo_to_regD = ~(|(opcode ^ 'EXE_R_TYPE)) & (~(|(funct[5:2] ^ 4'b0100))
    & ~funct[0]);
// 00--alu_outM; 01--hilo_o; 10 11--rdataM;
...
```

(2) alu_decode 模块:

```
module alu_decode(
    ...
    input wire [31:0] instrD,
```



```

        output reg [4:0] alu_controlD,
        ...
    );

    always @* begin
    case(opcode)
    'EXE_R_TYPE:
        case(funct)
            //hilo
            'EXE_MTHI:    alu_controlD <= 'ALU_MTHI;
            'EXE_MTL0:    alu_controlD <= 'ALU_MTL0;
            ...
        endcase
    ...
    end

```

(3) alu 模块

```

module ALU (
    ...
    input wire [4:0] alu_controlE, //alu 控制信号
    input wire [31:0] src_aE, src_bE, //操作数
    input wire [63:0] hilo, //hilo值
    output wire [63:0] alu_outE, //alu 输出
    ...
);

assign alu_outE = ({64{(alu_controlE == 'ALU_MTHI)}} & {src_aE, hilo[31:0]}) // 若
                  为mthi/mtlo 直接取Hilo的低32位和高32位
                  | ({64{(alu_controlE == 'ALU_MTL0)}} & {hilo[63:32], src_aE});
...

```

(4) hilo 模块

```

module hilo(
    input wire          clk,rst,we, //both write lo and hi
    input wire [31:0] instrM,
    input wire [63:0] hilo_in, //存入hilo的值

    output wire [31:0] hilo_out
);
// hilo寄存器
reg [63:0] hilo;
// 更新
always @(posedge clk) begin
    if(rst)
        hilo <= 0;
    else if(we)
        hilo <= hilo_in;
    else
        hilo <= hilo;
end
// 若为mfhi指令 读hilo高32位 若为mflo指令读hilo低32位
wire mfhi;
wire mflo;
assign mfhi = ~(|(instrM[31:26] ^ 'EXE_R_TYPE)) & ~(|(instrM[5:0] ^ 'EXE_MFHI));
assign mflo = ~(|(instrM[31:26] ^ 'EXE_R_TYPE)) & ~(|(instrM[5:0] ^ 'EXE_MFLO));

assign hilo_out = ({32{mfhi}} & hilo[63:32]) | ({32{mflo}} & hilo[31:0]);
endmodule

```

(5) datapath regfile 模块、main_decode 模块、alu_decode 模块、alu 模块与 hilo 模块的连接

```

...
wire          hilo_wenE; //hilo写使能
wire [63:0] hilo_oM; //hilo输出
wire          hilo_to_regM; //写寄存器值选择信号
...

```

```

main_decode main_decode0(
    ...
    .hilo_wenE(hilo_wenE),
    .hilo_to_regM(hilo_to_regM)
    ...
);
...
alu_decode alu_decode0(
    ...
    .alu_controlD(alu_controlD),
    ...
);
...
ALU alu0(
    ...
    .src_aE(src_aE),
    .hilo(hilo_oM),
    .alu_outE(alu_outE),
    ...
);
...
hilo hilo(
    ...
    .instrM(instrM),    // 用于识别mfhi, mflo, 决定输出;
    .we(hilo_wenE),     // both write lo and hi & ~flush_exceptionM
    .hilo_in(alu_outE),
    .hilo_out(hilo_oM)
);

```

2.2.4 算数运算指令

对算数运算指令,在计组四实现 add, sub, slt, addi 的基础上继续实现 addu, subu, sltu, mult, multu, div, divu, addiu, slti 和 sltiu 指令。将 14 条指令两两分组: add 和 addu, addi 和 addiu, sub 和 subu, slt 和 sltu, slti 和 sltiu, div 和 divu, mult 和 multu。每一组中的两条指令的区别只在于操作数是有符号还是无符号。那么可以在复用原数据通路的基础上进行 alu 内部运算的更改。

对 R 型指令 add 和 addu, sub 和 subu, slt 和 sltu, 复用原有数据通路, 添加常规控制信号以及 alu 模块无符号运算即可, 其中 alu 无符号运算还要特别关注溢出问题。

对 I 型指令 addi 和 addiu, slti 和 sltiu, 也可以直接复用原数据通路, 添加常规控制信号以及 alu 无符号运算。

对乘法指令 mult 和 multu, 与原始数据通路相比, 运算结果要写入 HILO 寄存器中, 在控制信号上, 需要额外根据乘除法指令对 HILO 寄存器写使能赋值并添加常规控制信号, 在 alu 中直接使用 Verilog 中的乘号进行乘法指令的实现, 值得一提的是, 对两个 32 位操作数运算的结果为 64 位, 高 32 位和低 32 位分别存入 HI 和 LO 寄存器中。

对除法指令 div 和 divu, 本设计中实现了一个除法器并在 alu 运算中使用。除法器 div 模块的逻辑为: 输入时钟信号、置位信号、flush 信号、有效信号、有符号信号以及 32 位的被除数和除数, 根据输入信号计算 64 位除法结果 result 并输出, 其中高 32 位为余数, 低 32 位为商, 该值作为 alu_outE 写入 HILO 寄存器, 高 32 位写入 HI 寄存器, 低 32 位写入 LO

寄存器。同时需要输出 `div_stall` 信号表明当前数据流是否因为除法运算而暂停,并将该信号传入 `hazard` 模块控制数据流在 IF,ID,EXE 阶段的停顿。在 mips 中 32 位除法运算需要 32 个时钟周期来完成。

对除法指令,除常规控制信号添加之外,还需要在 `alu` 模块根据除法指令添加除法器有效信号 `div_vaild` 和有符号运算信号 `div_sign`。同时,由于除法运算结果也要写入 HILO 寄存器中,同样需要在 `main_decode` 模块中对 HILO 寄存器的写使能赋值。

以上设计的关键代码如下:

(1)`main_decode` 模块:

```
module main_decode(
    ...
    input wire [31:0] instrD,
    output reg      hilo_wenE,
    ...
);
...
assign opcode = instrD[31:26]; //操作码
assign funct = instrD[5:0]; //funct
assign {reg_write_enD, reg_dstD, alu_imm_selD, mem_to_regD, mem_read_enD, mem
_write_enD} = sigs;
assign hilo_wenD = ~(|(opcode ^ 'EXE_R_TYPE)) & ~(|(funct[5:2] ^ 4'b0110)
) // div divu mult multu
|( ~(|(funct[5:2] ^ 4'b0100)) & funct[0]) //mthi mtlo
);
...
always @(*) begin
case(opcode)
'EXE_R_TYPE:
case(funct)
...
'EXE_ADD, 'EXE_ADDU, 'EXE_SUB, 'EXE_SUBU, 'EXE_SLTU, 'EXE_SLT
: begin
sigs = 7'b1_00_0_000; //算数运算指令
end
'EXE_ADDI, 'EXE_SLTI, 'EXE_SLTIU, 'EXE_ADDIU
: begin
sigs = 7'b1_01_1_000; //I型信号: 写回rs 操作立即数 不访存
end
...
endcase
end
end
...
```

(2)`alu_decode` 模块:

```
module alu_decode(
    ...
    input wire [31:0] instrD,
    output reg [4:0] alu_controlD,
    ...
);
...
always @* begin
case(opcode)
'EXE_R_TYPE:
case(funct)
'EXE_ADD:      alu_controlD <= 'ALU_ADD; //4
'EXE_SUB:      alu_controlD <= 'ALU_SUB;
'EXE_ADDU:     alu_controlD <= 'ALU_ADDU;
```

```

        'EXE_SUBU:    alu_controlD <= 'ALU_SUBU;
        'EXE_SLT:    alu_controlD <= 'ALU_SLT;
        'EXE_SLTU:    alu_controlD <= 'ALU_SLTU;

        //div and mul
        'EXE_DIV:      alu_controlD <= 'ALU_SIGNED_DIV;
        'EXE_DIVU:     alu_controlD <= 'ALU_UNSIGNED_DIV;
        'EXE_MULT:     alu_controlD <= 'ALU_SIGNED_MULT;
        'EXE_MULTU:    alu_controlD <= 'ALU_UNSIGNED_MULT;
        ...
    endcase
    'EXE_ADDI:    alu_controlD <= 'ALU_ADD;
    'EXE_ADDIU:   alu_controlD <= 'ALU_ADDU;
    'EXE_SLTI:    alu_controlD <= 'ALU_SLT;
    'EXE_SLTIU:   alu_controlD <= 'ALU_SLTU;
    ...

```

(3) alu 模块

```

module ALU (
    ...
    input wire [4:0] alu_controlE, //alu 控制信号
    input wire [31:0] src_aE, src_bE, //操作数
    output wire [63:0] alu_outE, //alu 输出
    output wire div_stallE,
    output wire overflowE//算数溢出
    ...
);
wire [63:0] alu_out_div; //乘除法结果
reg [63:0] alu_out_mul;
reg carry_bit; //进位 判断溢出
//乘法信号
assign mul_sign = (alu_controlE == 'ALU_SIGNED_MULT);
assign mul_valid = (alu_controlE == 'ALU_SIGNED_MULT) | (alu_controlE == 'ALU_
    UNSIGNED_MULT);
//除法信号
assign div_sign = (alu_controlE == 'ALU_SIGNED_DIV);
assign div_vaild = (alu_controlE == 'ALU_SIGNED_DIV || alu_controlE == 'ALU_UNSIGNED_
    DIV);

assign alu_outE = ({64{div_vaild}} & alu_out_div)
    | ({64{mul_valid}} & alu_out_mul)
    | ({64{~mul_valid & ~div_vaild}} & {32'b0, alu_out_simple})
    | ({64{(alu_controlE == 'ALU_MTHI)}} & {src_aE, hilo[31:0]}) // 若为
        mthi/mtlo 直接取Hilo的低32位和高32位
    | ({64{(alu_controlE == 'ALU_MTL0)}} & {hilo[63:32], src_aE});
// 加减且溢出位与最高位不等时 算数溢出
assign overflowE = (alu_controlE=='ALU_ADD || alu_controlE=='ALU_SUB) & (carry_bit ~
    alu_out_simple[31]);
always @(*) begin
    carry_bit = 0; //溢出位取0
    case(alu_controlE)
        'ALU_ADD:    {carry_bit, alu_out_simple} = {src_aE[31], src_aE} + {src_bE
            [31], src_bE};
        'ALU_ADDU:    alu_out_simple = src_aE + src_bE;
        'ALU_SUB:     {carry_bit, alu_out_simple} = {src_aE[31], src_aE} - {src_bE
            [31], src_bE};
        'ALU_SUBU:    alu_out_simple = src_aE - src_bE;

        'ALU_SIGNED_MULT: alu_out_mul = $signed(src_aE) * $signed(src_bE); //乘法
        'ALU_UNSIGNED_MULT: alu_out_mul = src_aE * src_bE;

        'ALU_SLT:     alu_out_simple = $signed(src_aE) < $signed(src_bE); //有符号
            比较
        'ALU_SLTU:    alu_out_simple = src_aE < src_bE; //无符号比较
        ...
    endcase
end
//与div模块连接
div div(
    .clk(~clk),

```

```

        .rst(rst),
        .flush(flushE),
        .a(src_aE), //divident
        .b(src_bE), //divisor
        .valid(div_vaild),
        .sign(div_sign), //1 signed
        .div_stall(div_stallE),
        .result(alu_out_div)
    );

    ...

```

(4) datapath main_decode 模块、alu_decode 模块、alu 模块与 hilo 模块、hazard 模块的连接

```

...
wire        hilo_wenE; //hilo 写使能
wire [63:0] hilo_oM;  //hilo 输出
wire        hilo_to_regM; //写寄存器值选择信号
...
main_decode main_decode0(
    ...
    .hilo_wenE(hilo_wenE),
    ...
);
...
alu_decode alu_decode0(
    ...
    .alu_controlD(alu_controlD),
    ...
);
...
ALU alu0(
    ...
    .src_aE(src_aE),
    .hilo(hilo_oM),
    .alu_outE(alu_outE),
    .div_stallE(alu_stallE),
    ...
);
...
hilo hilo(
    ...
    .instrM(instrM), // 用于识别mfhi, mflo, 决定输出;
    .we(hilo_wenE), // both write lo and hi & ~flush_exceptionM
    .hilo_in(alu_outE),
    .hilo_out(hilo_oM)
);
...
hazard hazard0(
    .alu_stallE(alu_stallE),
    .stallF(stallF), .stallD(stallD), .stallE(stallE),
    .flushE(flushE),
    ...
);

```

2.2.5 分支跳转指令

对 12 条分支跳转指令,在计组四实现 j 和 beq 指令的基础上,添加 bne, bgez, bgtz, blez, bltz, bltzal, bgezal, jal, jr 和 jalr 指令。

对 branch 类指令,bgtz, blez, bne, bltz 和 bgez 指令与 beq 的区别在于判断是否跳转的方式,可以在实现 beq 的原数据通路基础上对生成跳转信号的逻辑进行拓展,在设计中,我们添加了分支预测模块,数据通路描述如下:在 D 阶段,进行分支跳转预测生成信号量

branch_D,除 main_decode 模块添加基础控制信息外,在 alu_control 模块对 branch 指令单独使用分支判断 branch_judge_control 信号量,并在 E 阶段将其传入 branch_check 模块中,该模块根据分支判断信号量进行对应逻辑运算确认真实的是否跳转信号量,在 M 阶段将其与分支预测信号量传入 pc_reg 模块确认预测是否正确并确认下一条 pc 地址。如果预测正确,则不进行额外操作;如果预测不正确,若预测跳实际不跳则将下一条指令地址指向指向 branch 指令的 PC+8,若预测不跳实际上跳则将下一条指令地址指向 pc_branchD 传到 M 阶段的值。具体分支预测模块的实现将在模块设计中详述。

对 jr 指令,其功能与 j 指令一样是无条件跳转到目标地址,区别在于 jr 指令的跳转目标地址来自 rs 寄存器中存储的操作数。我们小组通过 jump_control 模块来控制跳转逻辑:在 D 阶段,jump_control 模块根据指令码判断 jr/j 的指令类型,并据此判断是否需要跳转并确定跳转目标。尤其对 jr 指令需要重点关注 rs 寄存器读取导致的 RAW 类型数据冒险;在 M 阶段将跳转目标地址和跳转信号以及数据冒险信号传入 pc_reg 模块确定下一条指令的跳转地址。

对 jal 指令,其在 j 指令功能的基础上将 PC+8 地址写入 31 号寄存器中。在数据通路上需要在 j 指令实现通路上添加对写回 31 号寄存器的选择以及 pc+8 的运算。其数据流为:D 阶段在 main_decode 模块中根据指令码确定写回寄存器的选择信号,在 alu_decode 模块中确定 alu 的操作控制信号,在 jump_control 模块中确定跳转地址以及跳转控制信号;E 阶段根据跳转控制信号选择 pc+8 作为 alu 第一个操作数来源并根据 alu 操作信号以该操作数作为 aluoutE,根据写回寄存器的选择信号选择第 31 号寄存器作为写回寄存器;在 M 阶段 pc_reg 模块根据跳转目标地址和跳转信号确定下一条指令的跳转地址;W 阶段将 pc+8 写回 31 号寄存器。

对 jalr 指令,其功能类似于 jr 和 jar 的结合:跳转到 rs 寄存器中存储的目标地址中并将 pc+8 写入 rd 寄存器中。其数据通路无需进行添加和改动,只需改动写回寄存器的控制信号即可。

对 bltzal 和 bgezal 指令,其功能类似于 bltz,bgez 指令与 jal 指令的集合,其跳转判断与 bltz、bgez 一致,与 jal 指令一样需要将 pc+8 的值写入 31 号寄存器,同样其数据通路无需进行添加和改动,只需改动相关控制信号即可。

以上设计的关键代码如下,其中 jump_control,pc_reg,branch_predecit 模块将在模块设计中详述:

(1)main_decode 模块:

```
module main_decode(  
    ...  
    input wire [31:0] instrD,
```

```

...
);
...
assign opcode = instrD[31:26]; //操作码
assign funct = instrD[5:0]; //funct
assign {reg_write_enD, reg_dstD, alu_imm_selD, mem_to_regD, mem_read_enD, mem
_write_enD} = sigs;
...
always @(*) begin
case(opcode)
'EXE_JR: begin
sigs = 7'b0_00_0_000; //跳转指令 信号为0
end
'EXE_JALR: begin
sigs = 7'b1_10_0_000; //jalr指令需要将分支之后的pc存入rd中
end
'EXE_BEQ, 'EXE_BNE, 'EXE_BLEZ, 'EXE_BGTZ: begin
sigs = 7'b0_00_0_000;
end
// 分支指令: BGEZ BLTZ BGEZAL BLTZAL
'EXE_BRANCHS: begin
case(rt[4:1])
// BGEZAL BLTZAL rt:10001 10000
// 无论转移与否 需要将延迟槽后pc保存至31号寄存器
4'b1000: begin
sigs = 7'b1_10_0_000;
end
// BLTZ rt:00000
4'b0000: begin
sigs = 7'b0_00_0_000;
end
end
// 不属于任何指令
end
endcase
end
...

```

(2) alu_decode 模块:

```

module alu_decode(
...
input wire [31:0] instrD,
...
);

always @* begin
case(opcode)
'default:
alu_controlD <= 'ALU_DONOTHING;
endcase
end
...

```

(3) alu 模块

```

module ALU (
...
input wire [4:0] alu_controlE, //alu 控制信号
input wire [31:0] src_aE, src_bE, //操作数
output wire [63:0] alu_outE, //alu输出
...
);
always @(*) begin
case(alu_controlE)
'ALU_DONOTHING: alu_out_simple = src_aE;
...
endcase
end
...

```

(4) datapath 模块对写回寄存器、alu 第一个操作数的选择控制

```
// 写回寄存器选择
mux4 #(5) mux4_regdst(rdE,rtE,5'd31,5'b0,reg_dstE,reg_writeE);
// alu 操作数选择, 当ex阶段是jal或alr指令, 或者bxxzal时, jumpE | branchE== 1选择pc_plus4D;
mux4 #(32) mux4_forward_aE(rd1E,resultM,resultW,pc_plus4D,{2{jumpE | branchE}} | forward_aE,src_aE);
```

2.2.6 访存指令

对 8 条访存指令,在计组 4 实现 lw,sw 的基础上继续实现 lb,lb,lb,lb,lb,lb,lb,lb 和 sh 指令。

对 load 类指令 lb,lb,lb,lb,lb 指令,其数据通路和控制信号可以直接复用 lw 指令的实现,但从 data ram 中读取的数据宽度有所不同,所以通过实现 mem_control 在 M 阶段根据指令码以及访存地址对从 data ram 中读取的原数据进行处理产生实际输出数据。

对 store 类指令 sb 和 sh 指令,其数据通路可以复用 sw 指令的实现,写入 data ram 数据的宽度需要通过控制四位写使能信号进行限制。仍然在 M 阶段利用 mem_control 模块根据指令码以及访存地址输出写使能信号和实际写入数据。

以上设计在 mem_control 模块中的的关键代码如下:

```
assign mem_wenM = ( {4{( instr_sw & addr_W0 )}} & 4'b1111) // 写字
                  | ( {4{( instr_sh & addr_W0 )}} & 4'b0011) // 写半字 低位
                  | ( {4{( instr_sh & addr_B2 )}} & 4'b1100) // 写半字 高位
                  | ( {4{( instr_sb & addr_W0 )}} & 4'b0001) // 写字节 四个字
                  | ( {4{( instr_sb & addr_B1 )}} & 4'b0010)
                  | ( {4{( instr_sb & addr_B2 )}} & 4'b0100)
                  | ( {4{( instr_sb & addr_B3 )}} & 4'b1000);

// data ram 按字寻址
assign mem_wdataM = ({ 32{instr_sw}} & data_wdataM) //
                    | ( {32{instr_sh}} & {2{data_wdataM[15:0]}} ) // 低位高位均为数据 具体根据操作
                    | ( {32{instr_sb}} & {4{data_wdataM[7:0]}} ); //

// rdata
assign data_rdataM = ( {32{instr_lw}} & mem_rdataM) //lw 直接读取字
                    | ( {32{ instr_lh & addr_W0}} & { {16{mem_rdataM[15]}}, mem_rdataM[15:0] } })
                    //lh 分别从00 10开始读半字 读取后进行符号扩展
                    | ( {32{ instr_lh & addr_B2}} & { {16{mem_rdataM[31]}}, mem_rdataM[31:16] } })
                    | ( {32{ instr_lhu & addr_W0}} & { 16'b0, mem_rdataM[15:0] } })
                    //lhb 分别从00 10开始读半字 读取后进行0扩展
                    | ( {32{ instr_lhu & addr_B2}} & { 16'b0, mem_rdataM[31:16] } })
                    | ( {32{ instr_lb & addr_W0}} & { {24{mem_rdataM[7]}}, mem_rdataM[7:0] } })
                    //lb 分别从00 01 10 11开始取bytes 读取后进行符号扩展
                    | ( {32{ instr_lb & addr_B1}} & { {24{mem_rdataM[15]}}, mem_rdataM[15:8] } })
                    | ( {32{ instr_lb & addr_B2}} & { {24{mem_rdataM[23]}}, mem_rdataM[23:16] } })
                    | ( {32{ instr_lb & addr_B3}} & { {24{mem_rdataM[31]}}, mem_rdataM[31:24] } })
                    | ( {32{ instr_lbu & addr_W0}} & { 24'b0, mem_rdataM[7:0] } })
                    //lbu 分别从00 01 10 11开始取bytes 读取后进行0扩展
                    | ( {32{ instr_lbu & addr_B1}} & { 24'b0, mem_rdataM[15:8] } })
                    | ( {32{ instr_lbu & addr_B2}} & { 24'b0, mem_rdataM[23:16] } })
                    | ( {32{ instr_lbu & addr_B3}} & { 24'b0, mem_rdataM[31:24] } });
```


2.2.7 内陷与特权指令

对 2 条内陷指令 `syscall`,`break` 和 3 条特权指令 `mtc0`,`mfc0`,`eret`。需要在原数据通路基础上添加 `cp0` 模块和异常处理 `exception` 模块。

对内陷指令 `break`, `syscall`。发生断点异常时,将控制权转到异常处理程序。其数据通路为:在 D 阶段根据指令码判断是否为内陷指令;在 M 阶段将标志信号传递到 `exception` 模块生成中断信号传递给 `cp0` 模块进行处理。

对特权指令中的 `eret` 指令,其在中断、异常或错误处理完成时返回中断指令,数据通路与内陷指令基本一致。对特权指令中的 `mtc0`,`mtc0` 指令将寄存器 `rt` 中的数据写入 CP0 寄存器中,其数据通路为:在 D 阶段,根据指令码确定 `cp0` 的写使能 `cp0_wen` 以及其他常规控制信息,在 M 阶段根据 `cp0` 的写使能和 `rd` 指定的存储器 (CP0 寄存器暂不支持 `sel` 字段) 将数据写入 CP0 的相应寄存器中。对特权指令中的 `mfc0` 指令,其将 `rd` 指定的 CP0 寄存器 (同样 CP0 寄存器暂不支持 `sel` 字段) 中的数据写入 `rt` 寄存器中,其数据通路为:在 D 阶段,根据指令码确定 CP0 的读使能 `cp0_to_reg`, 写回寄存器数据的选择信号 `is_mfc` 以及其他基本控制信号,在 M 阶段根据 `rd` 从 CP0 指定的寄存器读出数据与 `is_mfc` 信号一起确定要写回 `rt` 寄存器中的值,在 W 阶段写回。

以上设计的关键代码如下:

(1)`main_decode` 模块:

```
module main_decode(
    ...
    input wire [31:0] instrD,
    output reg        breakM, syscallM, eretM,
    output reg        cp0_wenM,    // 写 cp0
    output reg        cp0_to_regM, // 读 cp0
    output reg        is_mfcM     // 为 mfc0
    ...
);
...
assign opcode = instrD[31:26]; // 操作码
assign funct = instrD[5:0];   // funct
assign {reg_write_end, reg_dstD, alu_imm_selD, mem_to_regD, mem_read_end, mem
    _write_end} = sigs;
// cp0 写使能: 为 MTC0 指令
assign cp0_wenD = ~(|(opcode ^ 'EXE_ERET_MFTC)) & ~(|(rs ^ 'EXE_MTC0));
// 读 cp0: 为 MFC0 指令
assign cp0_to_regD = ~(|(opcode ^ 'EXE_ERET_MFTC)) & ~(|(rs ^ 'EXE_MFC0));
// 判断是否为 break syscall eret
assign breakD = ~(|(opcode ^ 'EXE_R_TYPE)) & ~(|(funct ^ 'EXE_BREAK));
assign syscallD = ~(|(opcode ^ 'EXE_R_TYPE)) & ~(|(funct ^ 'EXE_SYSCALL));
// eret 的 32 位 固定
assign eretD = ~(|(instrD ^ {'EXE_ERET_MFTC, 'EXE_ERET}));
...
always @(*) begin
    case(opcode)
        'EXE_SYSCALL, 'EXE_BREAK: begin
            sigs = 7'b0_00_0_000; // 跳转指令 信号为 0
        end
    end
// 3 条 特权 指令
    'EXE_ERET_MFTC: begin
        case(instrD[25:21])
```

```

        'EXE_MTC0: begin
            sigs = 7'b0;
        end
        'EXE_MFC0: begin
            sigs = 7'b1_01_0_000;
            is_mfcD = 1'b1;
        end
        default: begin
            riD = |(instrD[25:0] ^ 'EXE_ERET); //为eret指令: 0 不为eret指令: 1
            sigs = 7'b0;
        end
    end
endcase
end
...

```

(2) CP0 模块读寄存器逻辑

```

//read
// 读cp0组合逻辑
wire count, compare, status, cause, epc, prid, config1, badvaddr;
// 确定寄存器
assign count      = (~rst & ~(|( raddr_i ^ 'CP0_REG_COUNT      )));
assign compare    = (~rst & ~(|( raddr_i ^ 'CP0_REG_COMPARE    )));
assign status     = (~rst & ~(|( raddr_i ^ 'CP0_REG_STATUS     )));
assign cause      = (~rst & ~(|( raddr_i ^ 'CP0_REG_CAUSE      )));
assign epc        = (~rst & ~(|( raddr_i ^ 'CP0_REG_EPC        )));
assign badvaddr   = (~rst & ~(|( raddr_i ^ 'CP0_REG_BADVADDR   )));

// 读出相应寄存器
assign data_o = ( {32{rst}      } & 32'd0 )
| ( {32{count}    } & count_o )
| ( {32{compare}  } & compare_o )
| ( {32{status}   } & status_o )
| ( {32{cause}    } & cause_o )
| ( {32{epc}      } & epc_o )
| ( {32{badvaddr}} & badvaddr_o )
;

```

(3) CP0 模块写寄存器逻辑

```

// mtc0指令
if(we_i) begin
    case (waddr_i) //rd指定存储器 (暂不支持sel) 然后把rt的内容存入
        'CP0_REG_COUNT:begin
            count_o <= data_i;
        end
        'CP0_REG_COMPARE:begin
            compare_o <= data_i;
            cause_o[30] <= 'InterruptNotAssert;
        end
        'CP0_REG_STATUS:begin
            status_o[0] <= data_i[0]; //全局中断使能: 0-屏蔽 1-使能
            status_o[15:8] <= data_i[15:8]; //中断屏蔽位: 每一位控制一个中断的使能
        end
        'CP0_REG_CAUSE:begin //仅有9-8位可写
            cause_o[9:8] <= data_i[9:8]; //待处理软件中断标识 每一位对应软件中断
        end
        'CP0_REG_EPC:begin
            epc_o <= data_i; //例外程序计数器
        end
    endcase
end
end

```

(4) datapath 模块对写回寄存器值的选择控制

```

mux4 #(32) mux4_mem_to_reg(alu_outM, mem_ctrl_rdataM, hilo_oM, cp0_data_oW, {hilo_to_regM, mem_to_regM} | {2{is_mfcM}}, resultM);

```

2.3 SRAM 接口的 SOC 封装

利用硬件综合设计材料 test_func_test_v0.03/soc_sram_func/run_vivado 进行 sram 接口的 SoC 封装,对 mycpu_top 模块接口分为:CLK,RST,INT 信号;连接 INST_RAM 信号;连接 DATA_RAM 信号;调试 DEBUG 信号。应注意 CLK 信号对于 INST_RAM 是下降沿取值,与当前 CLK 相反。由于在访存指令实现时即使用 INST_RAM 和 DATA_RAM 的相关接口,只需要进行接口传递即可。对于调试 DEBUG 信号,是为了便于在 SOC 层调试增加的写入阶段信号,其中 debug_wb_pc 为 pcW, debug_wb_rf_wen 为写寄存器使能信号, debug_wb_rf_wnum 为写回 regfile 的寄存器值 WriteRegW, debug_wb_rf_wdata 为写入 regfile 的数据值 ResultW,其信号传递如下:

(1)mycpu_top 模块

```
datapath datapath(  
    ....  
    .debug_wb_pc(debug_wb_pc),  
    .debug_wb_rf_wen(debug_wb_rf_wen),  
    .debug_wb_rf_wnum(debug_wb_rf_wnum),  
    .debug_wb_rf_wdata(debug_wb_rf_wdata)  
);
```

(2)datapath 模块

```
assign debug_wb_pc          = pcM;  
assign debug_wb_rf_wen      = {4{reg_write_enM & ~flush_exceptionM }};//  
assign debug_wb_rf_wnum     = reg_writeM;  
assign debug_wb_rf_wdata    = resultM;
```

SOC_SRAM 结构如下所示:

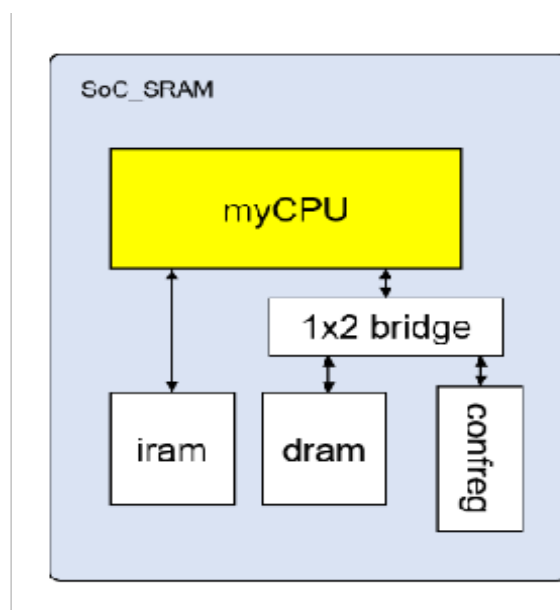


图 1: SOC_SRAM

2.4 模块设计

基于上述指令添加过程中数据通路和控制信号的增加和修改,整体数据通路图如下:

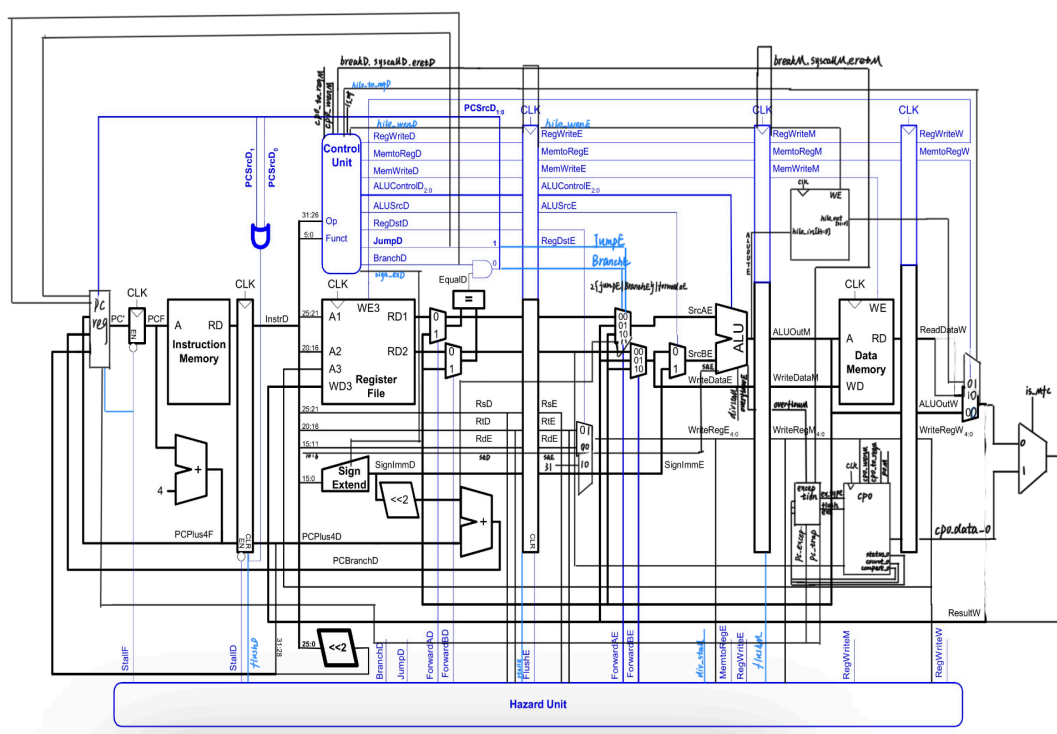


图 2: 数据通路图

本节将着重介绍分支预测模块顶层模块 mycpu_top, 数据通路模块 datapath, 分支预测模块 branch_predict, j 型指令跳转控制模块 jump_control, 异常处理模块 exception 以及 cp0, 冒险处理模块 hazard, pc 地址处理模块 pc_reg 以及存取控制模块 mem_control, 由于 main_decode, alu_decode, alu 模块已在上节指令添加描述中进行了相应展示, 本节只对这些模块进行信号描述, 不再赘述内部逻辑。

2.4.1 mycpu_top 模块与 datapath 模块

mycpu_top 模块主要负责提供对外连接 soc 的接口,其接口定义如下:

信号名	方向	位宽	功能
clk	input	1	时钟信号
rst	input	1	置位信号
int	input	6	常量 6'd0
inst_sram_en	output	1	使能信号
inst_sram_wen	output	4	默认为 4'b0, 无写入数据
inst_sram_addr	output	32	指令地址
inst_sram_wdata	output	32	默认 32'b0, 无写入数据
inst_sram_rdata	input	32	读出的指令
data_sram_en	output	1	使能信号
data_sram_wen	output	4	片选信号
data_sram_addr	output	32	数据的读写地址
data_sram_wdata	output	32	数据的写入数据
data_sram_rdata	input	32	数据的读出值
debug_wb_pc	output	32	pcW
debug_wb_rf_wdata	output	32	写入 regfile 的数据值 ResultW
debug_wb_rf_wen	output	4	写寄存器使能信号
debug_wb_rf_wnum	output	5	写回 regfile 的寄存器值 WriteRegW

datapath 模块是 mycpu_top 的子模块, 是对数据通路图的实现, 在 F,D,E,M,W 五级流水间进行数据流和控制信号的传递, 调用 main_decode, alu_decode, alu, hazard, pc_reg 等模块进行模块间的信号的数据传递, 其接口定义如下:

信号名	方向	位宽	功能
clk	input	1	时钟信号
rst	input	1	置位信号
int	input	6	常量 6'd0
inst_addrF	output	32	指令地址
inst_enF	output	1	使能信号
instrF	input	32	指令
mem_enM	output	1	使能信号
mem_wenM	output	4	片选信号
mem_addrM	output	32	读写地址
mem_wdataM	output	32	写入数据
mem_rdataM	input	32	数据的读出值
debug_wb_pc	output	32	pcW
debug_wb_rf_wdata	output	32	写入 regfile 的数据值 ResultW
debug_wb_rf_wen	output	4	写寄存器使能信号
debug_wb_rf_wnum	output	5	写回 regfile 的寄存器值 WriteRegW

2.4.2 branch_predict 模块

分支预测模块内部维护了 BHT 和 PHT,并借此在 D 阶段预测指令是否跳转,在 M 阶段根据实际跳转情况进行 BHT 和 PHT 的更新,其接口定义如下:

信号名	方向	位宽	功能
clk	input	1	时钟信号
rst	input	1	置位信号
flushD	input	1	D 阶段的刷新信号
stallD	input	1	D 阶段的暂停信号
instrD	input	32	指令
immD	input	32	立即数
pcF	input	32	F 阶段地址
pcM	input	32	M 阶段地址
branchM	input	1	M 阶段 branch 信号
actual_takeM	output	1	实际跳转情况
branchD	output	1	branch 指令信号
pred_takeD	output	1	branch 指令跳转信号

程序在译码阶段根据 pc 的值完成预测,因此,根据 pc 索引 BHT 得到对应 BHR,再用 BHR 的值索引 PHT 得到对应饱和计数器,最后得到预测结果。在程序运行到访存阶段时,可以得到是否为跳转指令 brachM 和真实跳转情况 actual_takenM,此时根据二者完成

BHT 和 PHT 的更新。BHT_index 取 pc 的 [14:5] 作为地址, 当为跳转指令时, 将跳转情况 actual_takenM 存入 BHT。同时 PHT 根据以下状态转移图实现更新:

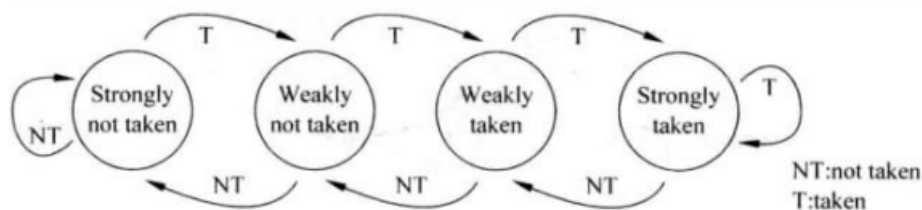


图 3: 状态转移图

BHT 初始化及更新逻辑如下:

```

wire [(PHT_DEPTH-1):0] update_PHT_index;
wire [(BHT_DEPTH-1):0] update_BHT_index;
wire [(PHT_DEPTH-1):0] update_BHR_value;

assign update_BHT_index = pcM[11:2];
assign update_BHR_value = BHT[update_BHT_index];
assign update_PHT_index = update_BHR_value;

always@(posedge clk) begin
    if(rst) begin
        for(j = 0; j < (1<<BHT_DEPTH); j=j+1) begin
            BHT[j] <= 0;
        end
    end
    else if(branchM) begin
        BHT[update_BHT_index] <= {BHT[update_BHT_index] << 1, actual_takeM};
    end
end

```

PHT 初始化及更新逻辑如下:

```

always @(posedge clk) begin
    if(rst) begin
        for(i = 0; i < (1<<PHT_DEPTH); i=i+1) begin
            PHT[i] <= Weakly_taken;
        end
    end
    else begin
        case(PHT[update_PHT_index])
            Strongly_not_taken : PHT[update_PHT_index] <= actual_takeM & branchM
                                ? Weakly_not_taken : Strongly_not_taken;
            Weakly_not_taken   : PHT[update_PHT_index] <= actual_takeM & branchM
                                ? Weakly_taken   : Strongly_not_taken;
            Weakly_taken       : PHT[update_PHT_index] <= actual_takeM & branchM
                                ? Strongly_taken  : Weakly_not_taken;
            Strongly_taken     : PHT[update_PHT_index] <= actual_takeM & branchM
                                ? Strongly_taken  : Weakly_taken;
        endcase
    end
end

```

2.4.3 jump_control 模块

jump_control 模块的大致功能为: 在 D 阶段对 j,jr,jarl,jal 指令进行判断并输出是否需要跳转的控制信号 jumpD 以及需要跳转的目标地址。对于 j 和 jal 指令跳转地址即为立即数,

对于另外两条指令的跳转地址来自 rs 寄存器。另外,该模块还对 rs 寄存器操作数导致的数据冲突进行了检测,当 rs 是在 E 或 M 阶段需要写回的寄存器时,则会输出冲突信号。

该模块的接口信号如下:

信号名	方向	位宽	功能
instrD	input	32	D 阶段指令码
pc_plus4D	input	32	pc+4
rd1D	input	32	rs 寄存器中的数据
reg_write_enE	input	1	E 阶段寄存器写使能
reg_write_enM	input	1	M 阶段的寄存器写使能
reg_writeE	input	5	E 阶段写回寄存器号
reg_writeM	input	5	M 阶段写回寄存器号
jumpD	output	1	D 阶段 jump 信号
jump_conflictD	output	1	写 rs 寄存器导致 jump 冲突
pc_jumpD	output	1	跳转目标

其 jumpD 信号的输出逻辑如下:

```
assign jr = ~(|instrD[31:26]) & ~(|(instrD[5:1] ^ 5'b00100)); //判断jr, jalr
assign j = ~(|(instrD[31:27] ^ 5'b00001)); //判断j, jal
assign jumpD = jr | j; //需要 jump
```

其 jump_conflictD 的输出逻辑如下:

```
assign jump_conflictD = jr &&((reg_write_enE && rsD == reg_writeE)
|| (reg_write_enM && rsD == reg_writeM));
```

其跳转目标 pc_jumpD 的选择逻辑如下:

```
wire [31:0] pc_jump_immD;
//instr_index左移2位 与pc+4高四位拼接
assign pc_jump_immD = {pc_plus4D[31:28], instrD[25:0], 2'b00};
assign pc_jumpD = j ? pc_jump_immD : rd1D; //j 和 jr 的跳转目标 立即数:寄存器的值
```

2.4.4 exception 与 cp0 模块

exception 模块对各阶段产生的精确异常信号以及指令信号,CP0 寄存器信息进行接收并处理内陷和特权指令以及中断和例外,在此基础上输出异常类型,异常清空信号,异常处理地址,异常处理地址选择信号和 pc 修正处理信息。

CP0 模块根据 exception 模块产生的异常类型(中断、地址错地外,系统调用例外,断点例外,保留指令例外,算数溢出例外等)对内部寄存器进行处理,同时 cp0 模块也实现了 mfc0 指令和 mtc0 指令的相关操作,其实现过程已在指令添加部分说明,在此不作赘述。

exception 模块的接口信号如下:

信号名	方向	位宽	功能
rst	input	1	置位
int	input	6	默认为 0
ri	input	1	保留指令例外
break	input	1	break 指令
syscall	input	1	syscall 指令
overflow	input	1	算数溢出例外
addrErrorSw	input	1	sw 地址异常
addrErrorLw	input	1	lw 地址异常
pcError	input	1	地址错误例外
eretM	input	1	eret 指令
cp0_status	input	32	status 寄存器
cp0_cause	input	32	cause 寄存器
cp0_epc	input	32	epc 寄存器
pcM	input	32	M 阶段地址
alu_outM	input	32	alu 输出(出错时 pc)
except_type	output	32	异常类型
flush_exception	output	1	是否有异常
pc_exception	output	32	pc 异常处理地址
pc_trap	output	32	是否跳转到 pc 异常处理地址
badvaddrM	output	32	pc 修正

cp0 模块的接口信号如下：

信号名	方向	位宽	功能
rst	input	1	置位
clk	input	1	时钟
we _i	input	1	写使能
waddr _i	input	5	rd 指定存储器
raddr _i	input	5	rd 指定存储器
data _i	input	32	rt 寄存器中的数据
data _o	output	32	读出的寄存器值
except_type _i	input	32	异常类型
current_inst_addr _i	input	32	M 阶段的地址
is_in_delayslot _i	input	1	是否在延迟槽
badvaddr _i	input	32	出错的虚地址
status _o	output	32	status 寄存器
cause _o	output	32	cause 寄存器
epc _o	output	32	epc 寄存器

其中 exception 模块生成异常类型的逻辑为：

```

assign int =    cp0_status[0] && ~cp0_status[1] && (
                //IM                                //IP
                ( |(cp0_status[9:8] & cp0_cause[9:8]) ) ||      //软件中断
                ( |(cp0_status[15:10] & ext_int) )              ||      //硬件中断
                (|(cp0_status[30] & cp0_cause[30]))              //计时器中断
);
// 全局中断开启,且没有例外在处理,识别软件中断或者硬件中断

assign except_type =    (int)                                ? 'EXC_TYPE_INT :      //中断
                        (addrErrorLw | pcError) ? 'EXC_TYPE_ADEL :      //地址错误例外 (
                        lw地址 pc错误
                        (ri)                                ? 'EXC_TYPE_RI :      //保留指令例外
                        (指令不存在
                        (syscall)                            ? 'EXC_TYPE_SYS :      //系统调用例外 (
                        syscall指令
                        (break)                              ? 'EXC_TYPE_BP :      //断点例外 (
                        break指令
                        (addrErrorSw)                        ? 'EXC_TYPE_ADES :      //地址错误例外 (
                        sw地址异常
                        (overflow)                          ? 'EXC_TYPE_OV :      //算数溢出例外
                        (eretM)                             ? 'EXC_TYPE_ERET :      //eret指令
                        'EXC_TYPE_NOEXC;                //无异常

```

exception 模块生成异常处理信号的逻辑为：

```

assign pc_exception =    (except_type == 'EXC_TYPE_NOEXC) ? 'ZeroWord:
                        (except_type == 'EXC_TYPE_ERET)? cp0_epc :
                        32'hbfc0_0380; //异常处理地址
assign pc_trap =        (except_type == 'EXC_TYPE_NOEXC) ? 1'b0:
                        1'b1; //表示发生异常处理pc
assign flush_exception =    (except_type == 'EXC_TYPE_NOEXC) ? 1'b0:
                        1'b1; //异常时的清空信号
assign badvaddrM =        (pcError) ? pcM : alu_outM; //出错时的pc

```

2.4.5 hazard 模块

hazard 模块主要处理数据通路中的数据冒险和控制冒险,仍然先对模块接口进行介绍,之后将根据具体情况进行具体逻辑阐述。

hazard 模块的接口如下：

信号名	方向	位宽	功能
alu_stallE	input	1	alu 导致的 stall, 源自除法
flush_jump_conflictE	input	1	跳转冲突信号
flush_pred_failedM	input	1	分支预测失败
flush_exceptionM	input	5	异常信号
rsE	input	5	rs 寄存器值
rtE	input	32	rt 寄存器值
reg_write_enM	input	32	M 阶段的写寄存器信号
reg_write_enW	input	32	W 阶段的写寄存器信号
reg_writeM	input	32	M 阶段的写寄存器号
reg_writeW	input	1	W 阶段的写寄存器号
mem_read_enM	input	32	读 mem 信号
stallF	output	32	F 阶段暂停信号
stallD	output	32	D 阶段暂停信号
stallE	output	32	E 阶段暂停信号
stallM	output	1	M 阶段暂停信号
stallW	output	1	W 阶段暂停信号
flushF	output	1	F 阶段刷新信号
flushD	output	5	D 阶段刷新信号
flushM	output	5	M 阶段刷新信号
flushW	output	32	W 阶段刷新信号
forward_aE	output	32	src_aE 的数据前推信号
forward_bE	output	32	src_bE 的数据前推信号

对于读取寄存器但目标值已经求出确未写回的情况,可以通过数据前推解决数据冒险,直接将上一条指令的 alu_outE 作为下一条指令的操作数,其前推逻辑如下:

```
assign forward_aE = rsE != 0 && reg_write_enM && (rsE == reg_writeM) ? 2'b01 :
                    rsE != 0 && reg_write_enW && (rsE == reg_writeW) ? 2'b10 : 2'b00;
assign forward_bE = reg_write_enM && (rtE == reg_writeM) ? 2'b01 :
                    reg_write_enW && (rtE == reg_writeW) ? 2'b10 : 2'b00;
```

对于 alu 除法运算,由于其运算周期即为 32 个周期,无法通过数据前推来解决数据冒险,需要进行 D,E,F 阶段的停顿直到除法运算结束,其逻辑如下:

```
assign stallF = ~flush_exceptionM & alu_stallE; // 如果异常直接 flush 不 stall
assign stallD = alu_stallE;
assign stallE = alu_stallE;
```

另外,如果有异常的存在,需要对整个流水线进行刷新;如果分支预测错误,需要对 D 进行刷新,其逻辑如下:

```
assign flushD = flush_exceptionM | flush_pred_failedM | flush_jump_conflictE ;
assign flushE = flush_exceptionM | (flush_pred_failedM & ~alu_stallE);
assign flushM = flush_exceptionM | alu_stallE; //
```

2.4.6 pc_reg 模块

该模块根据暂停、异常、分支预测、跳转冲突等一系列信号量确定下一条指令的地址,其接口如下:

信号名	方向	位宽	功能
clk	input	1	时钟
rst	input	1	置位
stallF	input	1	F 阶段的暂停信号
branchD	input	1	D 阶段跳转信号
branchM	input	1	M 阶段跳转信 M
pre_right	input	1	预测正确
actual_takeM	input	1	实际是否跳转
pred_takeD	input	1	D 阶段预测是否跳转
pc_trapM	input	1	异常 pc 选择
jumpD	input	1	D 阶段 jump 信号
jump_conflictD	input	1	D 阶段 jump 冲突
jump_conflictE	input	1	E 阶段 jump 冲突
pc_exceptionM	input	32	异常情况的跳转地址
pc_plus4E	input	32	PC+8
pc_branchM	input	32	M 阶段的跳转 pc
pc_jumpE	input	32	jump 在 E 阶段冲突的跳转地址
pc_jumpD	input	32	jump 在 D 阶段不冲突的跳转地址
pc_branchD	input	32	D 阶段预测跳转地址
pc_plus4F	input	32	pc+4
pc	output	32	下一条指令的 pc 地址

其下一条 pc 的选择类似于多个多路选择器累加的结果:如果发生异常,则下一条 pc 地址是 pc_exceptionM;若不发生异常但预测错误,若预测跳但实际不跳则下一条 pc 地址是 pc_plus4E,若预测不跳但实际跳则下一条 pc 地址是 pc_branchM;若在 E 阶段检测到 jump 读寄存器未写回的冲突则下一条 pc 地址是 pc_jumpE;若是 jump 指令且不冲突,则下一条 pc 地址是 pc_jumpD;若 D 和 M 阶段都是分支信号且 M 预测正确,则下一条 pc 地址是 pc_branchD;其余普通情况取下一条指令地址 pc_plus4F 即可。

2.4.7 mem_control 模块

mem_control 模块主要进行访存指令的执行以及地址异常的判断,其接口定义如下:

信号名	方向	位宽	功能
instrM	input	32	M 阶段的指令码
addr	input	32	访存地址
data_wdataM	input	32	写入的数据
mem_wdataM	output	32	真正写入的数据
mem_wenM	output	32	写使能
mem_rdataM	input	32	内存读出
data_rdataM	output	32	实际读出
addr_error_sw	output	1	sw 地址异常
addr_error_lw	output	1	lw 地址异常

其访存逻辑为:根据指令码判断访存指令类型,对 store 类型指令根据访存地址后两位生成写使能,将该信号传递给 data sram,写入指令对应的位宽,实际写入数据是根据指令类型截取要写数据的一部分写入 data sram。对 load 型指令根据数据类型和访存地址后两位,对实际读入的数据选取一部分进行有符号或无符号扩展作为最终读入值。

访存指令类型判断逻辑如下:

```

assign instr_lw = ~(!(op_code ^ 'EXE_LW));
assign instr_lb = ~(!(op_code ^ 'EXE_LB));
assign instr_lh = ~(!(op_code ^ 'EXE_LH));
assign instr_lbu = ~(!(op_code ^ 'EXE_LBU));
assign instr_lhu = ~(!(op_code ^ 'EXE_LHU));
assign instr_sw = ~(!(op_code ^ 'EXE_SW));
assign instr_sh = ~(!(op_code ^ 'EXE_SH));
assign instr_sb = ~(!(op_code ^ 'EXE_SB));

```

写使能生成逻辑如下:

```

assign mem_wenM = ( {4{( instr_sw & addr_W0 )}} & 4'b1111) // 写字
                  | ( {4{( instr_sh & addr_W0 )}} & 4'b0011) // 写半字 低位
                  | ( {4{( instr_sh & addr_B2 )}} & 4'b1100) // 写半字 高位
                  | ( {4{( instr_sb & addr_W0 )}} & 4'b0001) // 写字节 四个字
                  | ( {4{( instr_sb & addr_B1 )}} & 4'b0010)
                  | ( {4{( instr_sb & addr_B2 )}} & 4'b0100)
                  | ( {4{( instr_sb & addr_B3 )}} & 4'b1000);

```

实际写入数据生成逻辑如下:

```

assign mem_wdataM = ({ 32{instr_sw}} & data_wdataM) //
                  | ( {32{instr_sh}} & {2{data_wdataM[15:0]}} ) // 低位高位均为
                  | ( {32{instr_sb}} & {4{data_wdataM[7:0]}} ); // 数据 具体根据操作

```

实际读取数据生成逻辑如下:

```

assign data_rdataM = ({ 32{instr_lw}} & mem_rdataM) //lw 直接读取字
                  | ( {32{ instr_lh & addr_W0}} & { {16{mem_rdataM[15]}}, mem_rdataM[15:0] }} //lh 分别从00 10开始读半字 读取后进行符号扩展
                  | ( {32{ instr_lh & addr_B2}} & { {16{mem_rdataM[31]}}, mem_rdataM[31:16] }} //lhb 分别从00 10开始读半字 读取后进行0扩展
                  | ( {32{ instr_lhu & addr_W0}} & { 16'b0, mem_rdataM[15:0] }} //lhb 分别从00 10开始读半字 读取后进行0扩展
                  | ( {32{ instr_lhu & addr_B2}} & { 16'b0, mem_rdataM[31:16] }} //lhb 分别从00 01 10 11开始取bytes 读取后进行符号扩展
                  | ( {32{ instr_lb & addr_W0}} & { {24{mem_rdataM[7]}}, mem_rdataM[7:0] }} //lhb 分别从00 01 10 11开始取bytes 读取后进行符号扩展

```

```

| ( {32{ instr_lb    & addr_B1}} & { {24{mem_rdataM[15]}}, mem_rdataM[15:8]    })
| ( {32{ instr_lb    & addr_B2}} & { {24{mem_rdataM[23]}}, mem_rdataM[23:16]   })
| ( {32{ instr_lb    & addr_B3}} & { {24{mem_rdataM[31]}}, mem_rdataM[31:24]   })
| ( {32{ instr_lbu    & addr_W0}} & { 24'b0 , mem_rdataM[7:0]    })
//lbu 分别从 00 01 10 11 开始取 bytes 读取后进行 0 扩展
| ( {32{ instr_lbu    & addr_B1}} & { 24'b0 , mem_rdataM[15:8]    })
| ( {32{ instr_lbu    & addr_B2}} & { 24'b0 , mem_rdataM[23:16]   })
| ( {32{ instr_lbu    & addr_B3}} & { 24'b0 , mem_rdataM[31:24]   });

```

2.4.8 main_decode,alu_decode,alu 模块

在此只叙述这三个模块的信号接口,其内部逻辑已在指令添加描述中进行了较为详细的叙述。

main_decode 模块			
信号名	方向	位宽	功能
clk	input	1	时钟
rst	input	1	置位
instrD	input	32	D 阶段指令码
stallE	input	1	E 阶段停顿信号
stallM	input	1	M 阶段停顿信号
stallW	input	1	W 阶段停顿信号
flushE	input	1	E 阶段的刷新信号
flushM	input	1	M 阶段的刷新信号
flushW	input	1	W 阶段的刷新信号
sign_extD	output	1	立即数是有符号扩展
reg_dstE	output	2	写寄存器选择信号
alu_imm_selE	output	1	alu 第二个操作数是否来源于立即数
reg_write_enE	output	1	E 阶段寄存器写使能
hilo_wenE	output	1	hilo 写使能
mem_read_enM	output	1	mem 读使能
mem_write_enM	output	1	mem 写使能
reg_write_enM	output	1	M 阶段寄存器写使能
mem_to_regM	output	1	寄存器数据来自访存
hilo_to_regM	output	1	寄存器数据来自 hilo
riM	output	1	指令不存在异常
breakM	output	1	break 指令信号
syscallM	output	1	syscall 指令信号
eretM	output	1	eret 指令信号
cp0_wenM	output	1	cp0 写使能
cp0_to_regM	output	1	寄存器写入数据来自 cp0
is_mfcM	output	1	mfc 指令信号

alu_decode			
信号名	方向	位宽	功能
instrD	input	32	D 阶段指令码
alu_controlD	output	1	alu_control 信号
branch_judge_controlD	output	1	branch 指令信号
alu			
信号名	方向	位宽	功能
clk	input	1	时钟
rst	input	1	置位
flushE	input	1	E 阶段刷新信号
src_aE	input	32	第一个操作数
src_bE	input	32	第二个操作数
alu_controlE	input	5	alu 操作控制信号
sa	input	5	移位信号
hilo	input	64	hilo 寄存器
div_stallE	output	1	除法运算导致的停顿
alu_outE	output	64	alu 运算结果
overflowE	output	1	算数溢出信号

3 设计过程

3.1 设计流水账

魏才讓：

2022.1.1 20:00-2:00 : 在网络中搜寻各种相关材料和参考文件并开始学习,初步了解各种指令的实现方式、初步设计一些数据通路;

2022.1.2 10:00-1:30 : 再次学习相关材料,并分配小组内工作,准备分支指令的实现。由于在体系结构实验中实现过分支预测,所以在本次设计中也考虑加入分支预测,这一天主要是实现几条分支指令以及分支预测的逻辑,还没有调试;

2022.1.3 10:00-2:00 : 加入完成条件分支指令并调试,(由于分支指令需要处理冒险,以及分支预测需要控制流水线的刷新,调试较久);

2022.1.4 10:00-2:00 : 加入 j 类指令(主要处理 jr 的冲突)并调试。加入访存指令并调试

(访存指令逻辑较为清晰,相对好实现,主要是加入一个访存控制单元);

2022.1.5 10:00-2:30: 合并前 52 条指令,进行 SOC 封装,进行功能测试、不断调试直到通过前 64 个功能测试点。开始学习异常处理模块以及特权指令和内陷指令;

2022.1.6 10:00-3:30: 添加异常处理模块和 cp0 寄存器,在原有的通路中加入判断异常的一些信号,实现特权指令和内陷指令的逻辑并调试。最终通过 89 个测试点。尝试上板,优化内部逻辑后,最终上板也终于通过 89 个测试点。

宁可馨:

2022.1.1 20:00-23:00: 学习 2019 年与 2020 年的硬综 ppt 和视频材料,了解硬件综合设计的大致内容和流程;

2022.1.2 10:00-12:00: 复习 mips 指令以及计算机组成原理实验四的数据通路实现;

2022.1.2 16:00-18:00: 阅读硬综材料,进一步明确硬综要求和测试方法;

2022.1.2 21:00-22:00: 在计组实验四的基础上添加逻辑运算指令;

2022.1.3 10:00-12:00: 在计组实验四的基础上添加移位运算指令、熟悉数据移动指令;

2022.1.3 16:00-22:00: 继续添加数据移动指令,添加部分算数运算指令;

2022.1.4 14:00-22:00: 测试前三组指令,调试数据移动指令;

2022.1.5 10:00-11:30: 继续添加算数运算指令,学习乘除法器的实现;

2022.1.5 16:00-22:00: 熟悉所有指令的功能和大致数据通路,开始撰写实验报告;

2022.1.6 10:00-11:30: 学习分支指令和异常处理的实现过程;

2022.1.6 16:00-22:00: 继续撰写实验报告;

3.2 错误记录

在写报告的时候已经忘记了一些错误,再此列出一些印象深刻的错误。

3.2.1 错误 1

(1) 错误现象: 分支指令跳转到了不该跳转的地方。

(2) 分析定位过程: 首先是控制台中与 trace 的比较可以精确找到错误的 pc,对比.s 文件比较前后指令,然后对照波形图何时发生跳转,回溯路过的每一个信号判断为何发生这种异常跳转。

- (3) 错误原因: 分支预测跳转,但实际不用跳转,由于分支预测在 D 阶段,而判断需要在 E 阶段,所以当预测错误时需要刷新流水线 D、E,然后把 M 阶段的 pc 重新返回 pc reg,比对后发现,忘记把 E 阶段也刷新。
- (4) 修正效果: 在 hazard 模块中对 flushE 加入分支预测失败的信号。
- (5) 归纳总结: 这种错误属于小错误,不应该犯的马虎问题。

3.2.2 错误 2

- (1) 错误现象: 无条件跳转时的冲突
- (2) 分析定位过程: 操作大同小异,首先是控制台中与 trace 的比较可以精确找到错误的 pc,对比.s 文件比较前后指令,然后对照波形图何时发生跳转,回溯路过的每一个信号判断为何发生这种冲突。
- (3) 错误原因:jr 类指令虽然也是无条件跳转,但是如果前面一条指令刚好要写回相同的寄存器,则会发生冲突。
- (4) 修正效果: 在 hazard 模块中检测冲突,让流水线暂停。
- (5) 归纳总结: 对于 j 类指令,想当然地认为无条件跳转便不会冲突,于是还是遇到了。

3.2.3 错误 3

- (1) 错误现象:hilo 寄存器的值异常
- (2) 分析定位过程: 操作大同小异,首先是控制台中与 trace 的比较可以精确找到错误的 pc,对比.s 文件比较前后指令,然后对照波形图何时产生异常,回溯路过的每一个信号判断为何发生这种异常。
- (3) 错误原因: 在实现 hilo 寄存器的时候,是实现了一个 64 位的寄存器,高位为 hi 低位为 lo,在写入 hi 的时候默认低位为 0 写入,导致了把原来 hilo 寄存器中的值低位覆盖为 0 这样尴尬的事情发生。
- (4) 修正效果: 在写入 hi 或者 lo 时,先将 hilo 中的值取出,然后与对应半字拼接,在把新值存入 hilo 寄存器中。
- (5) 归纳总结: 属于逻辑上没有问题,但实际运行时才能发现的一个小问题。

3.2.4 错误 4

- (1) 错误现象:syscall 不跳转
- (2) 分析定位过程: 操作大同小异,首先是控制台中与 trace 的比较可以精确找到错误的 pc,对比.s 文件比较前后指令,然后对照波形图何时产生异常,回溯路过的每一个信号判断为何发生这种异常。
- (3) 错误原因: 第一个异常处理测试就是 syscall 语句,但它并没有跳转,初步判定是流水

线刷新处理的问题。

(4) 修正效果: 再次分析, 在 syscall 的时候, 应该刷新 D E M 阶段, stall F 阶段。

(5) 归纳总结: 流水线刷新逻辑上的问题。

3.2.5 错误 5

(1) 错误现象: MFC0 指令没有把值存入寄存器

(2) 分析定位过程: 操作大同小异, 首先是控制台中与 trace 的比较可以精确找到错误的 pc, 对比.s 文件比较前后指令, 然后对照波形图何时产生异常, 回溯路过的每一个信号判断为何发生这种异常。

(3) 错误原因: 在写回的四选中片选信号没有加入这一条的选择。

(4) 修正效果: 在片选信号中加入如果为 mfc0 指令时的选择, 这个选择信号来自于 decode 时发现为 mfc0 指令, 并一直传到写回时的选择器。

(5) 归纳总结: 选择信号的处理问题。

4 设计结果

4.1 设计交付物说明

4.1.1 目录层次

```
—mycpu_top           //顶层模块

  —datapath           //数据通路模块

    —main_decode       //控制信号译码模块

    —alu_decode        //alu 控制信号译码模块

    —hazard            //数据冒险处理模块

    —pc_reg            //pc 控制模块

    —F_to_D            //F->D 信号传递

    —inst_ascii_decoder //指令译码模块, 便于 debug

    —regfile           //寄存器

    —branch_predict    //分支预测模块

    —jump_control      //跳转控制模块
```

```

—D_to_E          //D->E 信号传递

—alu             //alu 模块

    —div          //除法器

—mux4            //四路选择器

—branch_check    //分支控制模块

—E_to_M          //E->M 信号传递

—mem_control     //存储控制模块

—hilo            //hilo 寄存器实现

—execption       //异常处理模块

—cp0             //cp0 寄存器模块

—M_to_W          //M->W 信号传递

```

4.1.2 仿真

使用硬件综合设计材料进行 sram 接口的 SoC 功能测试,打开 test_func_test_v0.03/soc_sram_func/run_vivado 下的工程文件,利用 add_source 将上述目录层次中的文件添加到工程中选择 inst_ram 的 coe 文件为 test_func_test_v0.03/soft/func_full/obj/inst_ram.coe。进行 Run Simulation, 仿真成功后点 Rull all, 通过功能测试的 89 个测试点并出现正确波形图。

4.1.3 综合

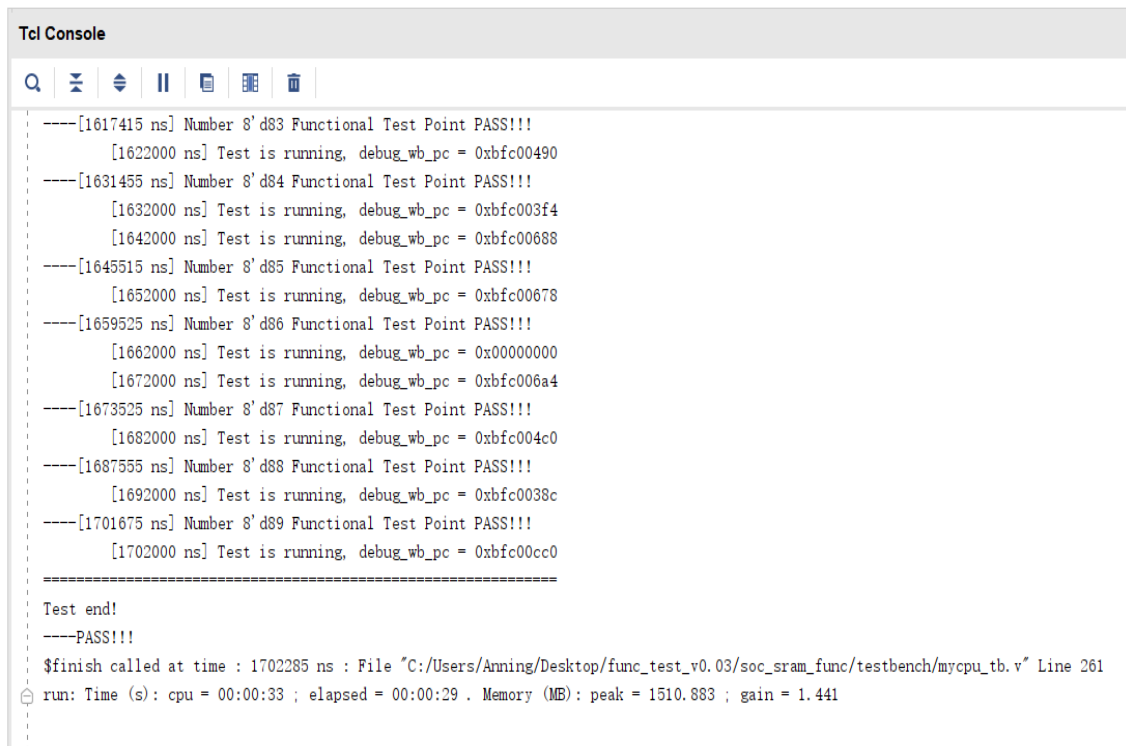
仿真结束无误后,则可以开始进行综合 Run Synthesis, 等待完成后继续 run implementation, 等待完成综合实现。

4.1.4 上板

综合完成后,则可以通过 generate bitstream 生成 bit 流,成功生成后即可 open hardware manager, 使用数据线连接好开发板后进行 program device, 将 bit 流上传到开发板后即可进行开发板演示。

4.2 设计演示结果

4.2.1 sram 接口的 SoC 功能测试结果



```
Tcl Console

[1617415 ns] Number 8'd83 Functional Test Point PASS!!!
[1622000 ns] Test is running, debug_wb_pc = 0xbfc00490
[1631455 ns] Number 8'd84 Functional Test Point PASS!!!
[1632000 ns] Test is running, debug_wb_pc = 0xbfc003f4
[1642000 ns] Test is running, debug_wb_pc = 0xbfc00688
[1645515 ns] Number 8'd85 Functional Test Point PASS!!!
[1652000 ns] Test is running, debug_wb_pc = 0xbfc00678
[1659525 ns] Number 8'd86 Functional Test Point PASS!!!
[1662000 ns] Test is running, debug_wb_pc = 0x00000000
[1672000 ns] Test is running, debug_wb_pc = 0xbfc006a4
[1673525 ns] Number 8'd87 Functional Test Point PASS!!!
[1682000 ns] Test is running, debug_wb_pc = 0xbfc004c0
[1687555 ns] Number 8'd88 Functional Test Point PASS!!!
[1692000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[1701675 ns] Number 8'd89 Functional Test Point PASS!!!
[1702000 ns] Test is running, debug_wb_pc = 0xbfc00cc0
=====
Test end!
---PASS!!!

$finish called at time : 1702285 ns : File "C:/Users/Arning/Desktop/func_test_v0.03/soc_sram_func/testbench/mycpu_tb.v" Line 261
run: Time (s): cpu = 00:00:33 ; elapsed = 00:00:29 . Memory (MB): peak = 1510.883 ; gain = 1.441
```

图 4: 功能测试

4.2.2 sram 接口的 SoC 功能测试仿真局部示意图



图 5: 功能测试仿真图

4.2.3 上板结果

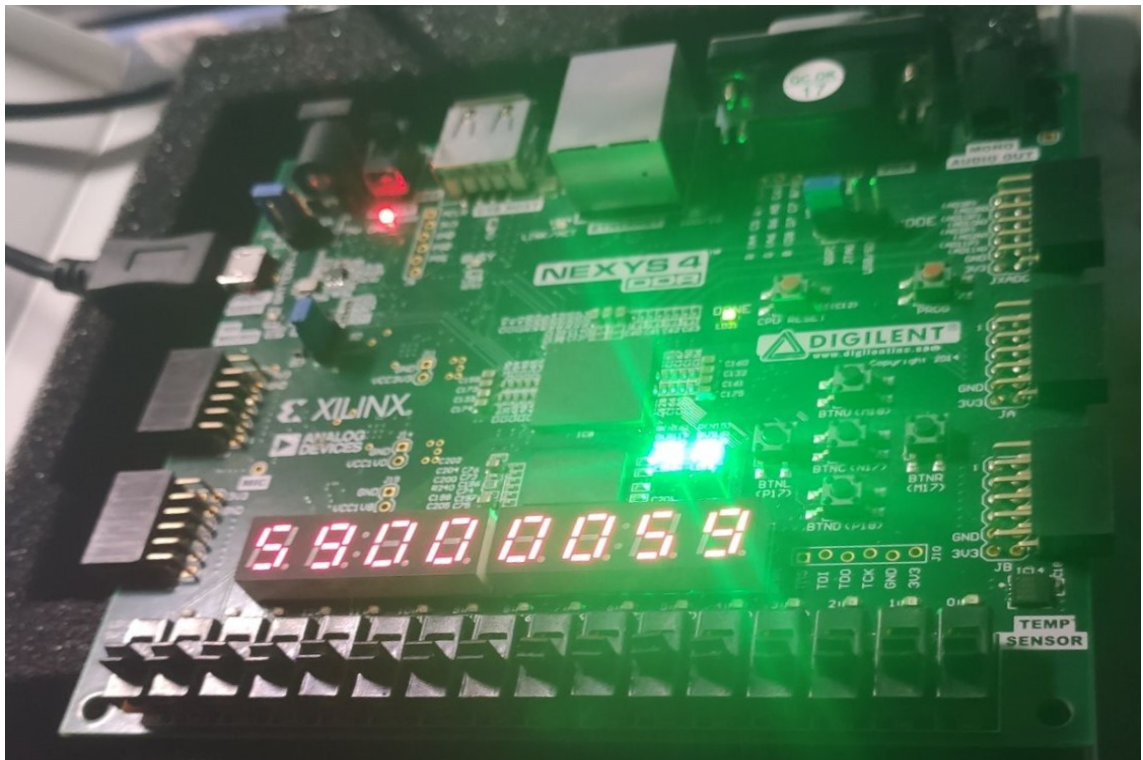


图 6: 上板结果

4.2.4 现场指令添加和答辩情况

4.3 指令添加

我们小组要求添加的指令为 `equal rd, rs, rt`, 其功能为判断 `rs` 寄存器的值是否等于 `rt` 寄存器的值, 若相等则将 1 写入 `rd` 寄存器中, 若不等则将 0 写入 `rd` 寄存器中。其操作码为 111111, 功能码为 000000。这实际上是一个 R 型指令, 只需要在复用原有 R 型指令的基础上进行基础控制信号添加即可。

添加指令的操作为:

- (1) 在宏定义中添加该指令的相关变量, `EXE_EQUAL=6'b111111`,
`ALU_EQUAL=5'b11101`;
- (2) 在 `main_decode` 模块中根据操作码为 `EXE_EQUAL` 确定常规控制信号, 与算数运算指令一致;
- (3) 在 `alu_decode` 模块中根据根据操作码为 `EXE_EQUAL` 确定 `alu_control` 信号量为 `ALU_EQUAL`;
- (4) 在 `alu` 模块中, 若 `alu_control` 信号量为 `ALU_EQUAL`, 则 `alu` 进行两个源操作数 `src_a` 和 `src_b` 是否相等的判断并将判断结果作为输出。

4.4 答辩情况

魏才讓:

问题一: 在实现过程中你们引用了些什么?

回答: 我们在实现的过程中, 主要是参考了《教你做一个 cpu》、龙芯的手册以及一些 github 上的开源资源。比如除法器的实现我们就套用了 github 上现有的除法器。

问题二: SOC 封装是怎么实现的? 回答: 封装其实只需要把对应的各种信号连起来即可。值得注意的是 `pc` 初值要为 `0x3fc00000`。

宁可馨:

问题一: `hilo` 寄存器如何实现?

回答: 在内部有一个 64 位寄存器, 高 32 位是 `hi` 寄存器的值, 低 32 位是 `lo` 寄存器的值, 这是我和参考代码不太一样的地方。

问题二: `hilo` 寄存器写入过程有没有数据冒险?

回答:有,在我们的设计中写入 hilo 寄存器的数据由 alu_outE 产生,写入数据源自 alu 的第一个操作数,可能会出现读取但未写入的情况。

问题三:hilo 寄存器靠什么触发?

回答:写使能,在 main_decode 模块中对指令进行判断,如果是乘除法或者 MT 类型的指令则将写使能置 1。

5 参考设计说明

5.1 ascii 译码模块实现

借鉴引用了/ref_code/ascii 中的相关文件。

5.2 除法器模块实现

借鉴引用了

https://github.com/CherryYang05/OpenMIPS_CPU/blob/master/src/mymips/mymips.srccs/sources_1/new/div.v。

5.3 CP0 模块实现

借鉴引用了/ref_code/cp0_reg.v

5.4 cpu 模块结构借鉴

借鉴引用了 <https://github.com/YC-Vertex/mips> 的结构实现。

6 总结

(1)本次实验对心态和能力而言是一个很大的挑战,由于考试以及课程实现和项目的原因导致硬件综合设计的时间非常紧张,这极大考验了抗压能力和时间利用能力。

(2)通过此次硬件综合设计加深了我们对于硬件知识的理解,提升了对于硬件数据通路的设计能力。我们更加深刻的理解了数据通路在整体上应该完成什么功能,在局部上如何将各个功能模块连接起来在各个时期各个模块间实现数据和信号传递。

7 供同学们吐槽之用。有什么问题都可以直接写在这。

7.1 魏才讓

这次的硬件综合设计主要是由于考试安排的比较晚导致时间非常紧,于是只能没日没夜地去肝,一个小 bug 就可能需要调好几个小时,导致时间更加紧迫。在前 52 条指令中,因为这些普通指令都比较熟悉,所以逻辑清晰,调起来也是顺藤摸瓜,有理有据。硬件调试时的优点就在于逻辑上是完全透明且底层的,虽然需要认真的看每条指令和波形图(看波形图真的好难受),但是逻辑及其清晰。对于后五条关于异常的指令,在课上基本上没有学过,所以首先就需要花时间和精力去学习这部分东西才能做,如果针对这部分能再多给一点详细且易懂的资料就更好了。总得来说,硬件设计确实可以让我对整个体系结构有了更加深刻的认知,对计算机底层逻辑更加熟悉,确实对同学的提升十分巨大。希望对下一届的同学可以再增加难度,也是对同学们的锻炼。

7.2 宁可馨

这次硬综设计让我体会到,指令的数据通路看似简单,但其内部有很多数据冒险、异常等逻辑细节都会影响指令的成功实现,这也是指令添加和调试过程中容易忽略却又必须注意非常折磨人的地方。另外,对 cp0 模块和 exception 模块实现异常处理在理解上存在困难,从而影响了整体设计和理解。令人遗憾的是,由于考试安排和其他课程项目的原因,硬件综合设计的时间并不是很充足,加上对硬件实验的排斥心理,导致在设计中并没有充分发挥个人作用。

[1] [3] [2]

参考文献

- [1] Hennessy and JohnL. *Computer Architecture A Quantitative Approach*: 计算机体系结构量化研究方法. China Machine Press, 2002.
- [2] 姚永斌. 超标量处理器设计. 清华大学出版社, 2014.
- [3] 雷思磊. 自己动手写 CPU. 电子工业出版社, 2014.