

# 北京科技大学 计算机与通信工程学院

## 课程设计报告

课程名称: 计算机组成原理课程设计

学生姓名: 刘伟浩

专 业: 计算机科学与技术

班 级: 计 202

学 号: 42024093

指导教师: 张磊

报告成绩: \_\_\_\_\_

实验地点: 腾讯会议+线下机电楼 304

实验时间: 2022 年 9 月 11 日----2022 年 11 月 6 日

# 北京科技大学实验报告

学院： 计算机与通信工程学院      专业： 计算机科学与技术      班级： 计 202

---

姓名： 刘伟浩      学号： 42024093      实验日期： 2022 年 9 月 11 日

---

## 一、课设目的与要求

- 学会处理器的设计方法：单周期/流水线。
- 掌握处理器设计过程中指令扩展的方法。
- 能够运用现代工具独立实现一个完整的处理器。
- 了解处理器功能测试的方法：仿真测试及 FPGA 测试。
- 计算机系统观的建立，对所设计的处理器在整个计算机系统的位置有所了解。

## 二、实验设备（环境）及要求

龙芯实验箱一体化实验平台/CG 平台/流水线 CPU 关键技术虚拟仿真实验平台

OS: Win10 64 位

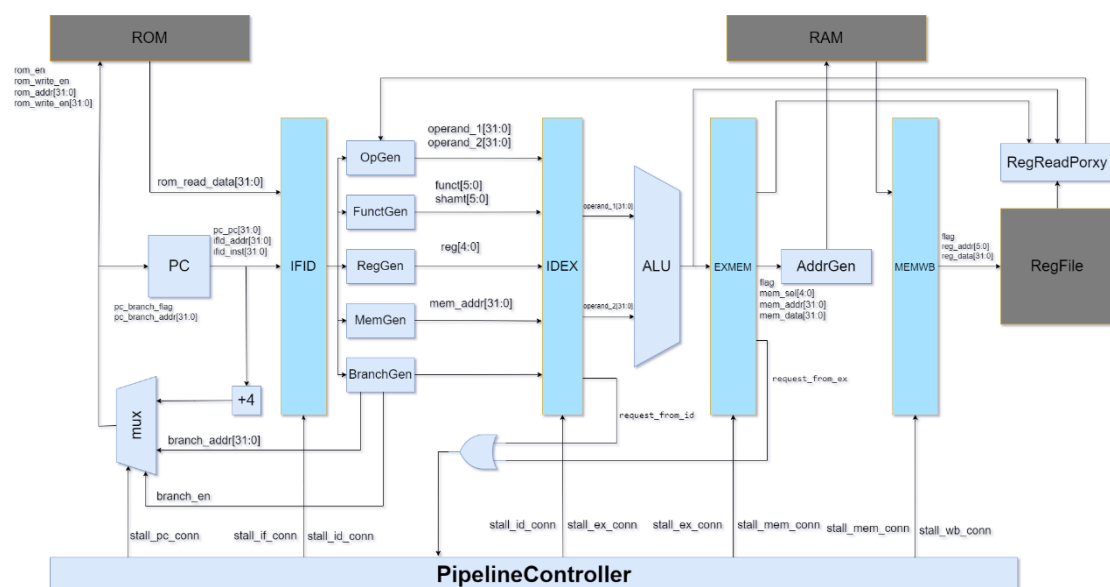
Software: Vivado2018.3 开发工具

VirtualBox 虚拟机+Ubuntu16.04.6（正文用五号宋体）

## 三、设计过程与结果分析

### Part 1 个人任务，代码阅读分析

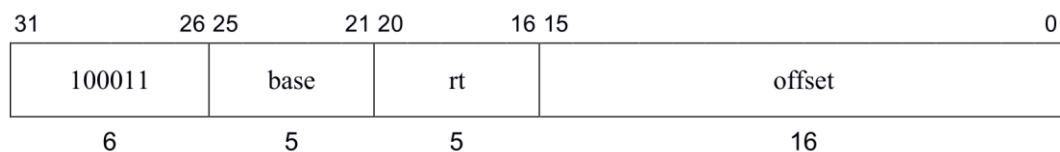
1 TinyMIPS 总体结构框图（自己动手重新画图，推荐画图工具：gliffy 或 draw.io 或者其它）



## 2 LW 指令（典型指令）的设计过程

### 2.1 指令格式

这是一条 I 型指令，格式如下：



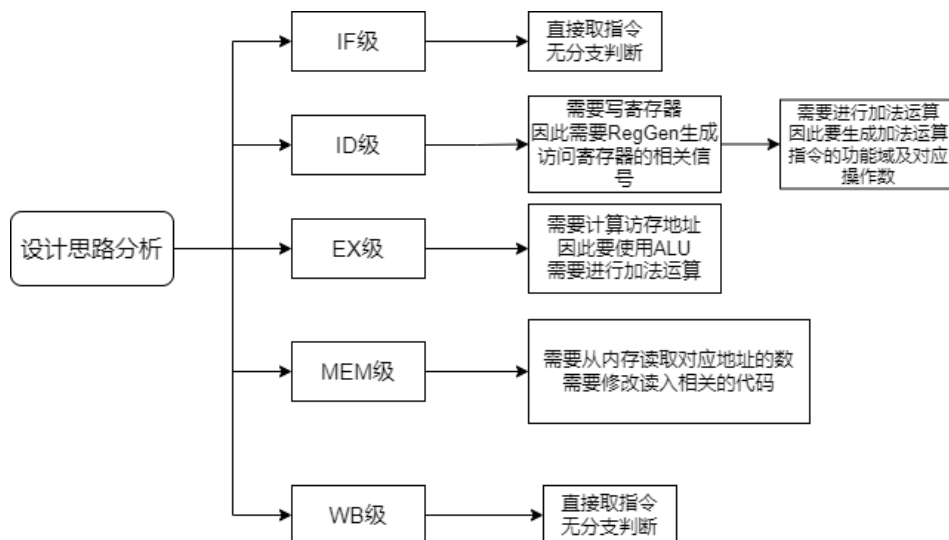
汇编格式：LW rt, offset(base)

功能描述：将 base 寄存器的值加上符号扩展后的立即数 offset 得到访存的虚地址，如果地址不是 4 的整数倍则触发地址错例外，否则据此虚地址从存储器中读取连续 4 个字节的值并进行符号扩展，写入到 rt 寄存器中。

### 2.2 设计分析与实现代码(需有指令设计分析过程图, 实现代码需有恰当的注释)

LW 指令是一条访存指令，并且其寻址方式是寄存器间接寻址，因此也涉及 ALU 运算。从 MEM 级取得存储器数后需要在 WB 级写回 rt 寄存器。综上，LW 指令涉及全部的 5 个流水级，需要在每个流水级考虑该级需要完成的功能。

设计分析过程图如下：



首先在 IF 级，直接读取指令，由于不需要跳转，因此 PC 级无特殊判断。

在 ID 级，对于从内存读入的数据，需要写回到寄存器堆中，因此我们需要设置寄存器堆的写使能信号。同时需要读出作为 base 地址的 rs 寄存器的值。

FunctGen 中对应代码如下：

```

// generate read address
always @(*) begin
case (op)
...
// memory accessing
`OP_LW: begin
    reg_read_en_1 <= 1; // rs 寄存器读使能
    reg_read_en_2 <= 0;
    reg_addr_1 <= rs; // 读寄存器中的 rs 寄存器
    reg_addr_2 <= 0;
end
...
end

// generate write address
always @(*) begin
case (op)
...
`OP_LW: begin
    reg_write_en <= 1; // rt 寄存器写使能
    reg_write_addr <= rt; // 写入 rt 寄存器
end
...
end

```

同时，由于需要计算读取内存的地址  $\text{signed\_ext}(\text{offset}) + \text{GPR}(\text{base})$ ，因此需要操作数生成相关信号与 FUNCT 域相关信号。

如下是 OperandGen 中的相关代码：

```


```

```

// generate operand_1
always @(*) begin
case (op)
...
    // memory accessing
    `OP_LW: begin
        operand_1 <= reg_data_1; //运算的数来自 rs 寄存器
    end
    ...
endcase
end

// generate operand_2
always @(*) begin
case (op)
...
    // memory accessing
    `OP_LW: begin
        operand_2 <= sign_ext_imm; //运算数 offset 的符号扩展
    end
    ...
endcase
end

```

如下是 FunctGen 中的相关代码:

```

// generating FUNCT signal in order for the ALU to perform operations
always @(*) begin
case (op)
...
    `OP_LW: funct <= `FUNCT_ADDU; // 生成无符号数加的信号
    ...
endcase
end

```

EX 级直接执行两个数的相加, 因此直接考虑 MEM 级对内存的访问, 这里要生成访存的读使能型号, 读取的符号信号与选择信号。

```

always @(*) begin
case (op)
    `OP_LW: mem_read_flag <= 1; // 读使能信号
    ...
endcase
end

always @(*) begin
case (op)
    `OP_LW: mem_sign_ext_flag <= 1; // 读入有符号数
    ...
endcase
end

// mem_sel: lb & sb -> 1, lw & sw -> 1111
always @(*) begin
case (op)
...

```

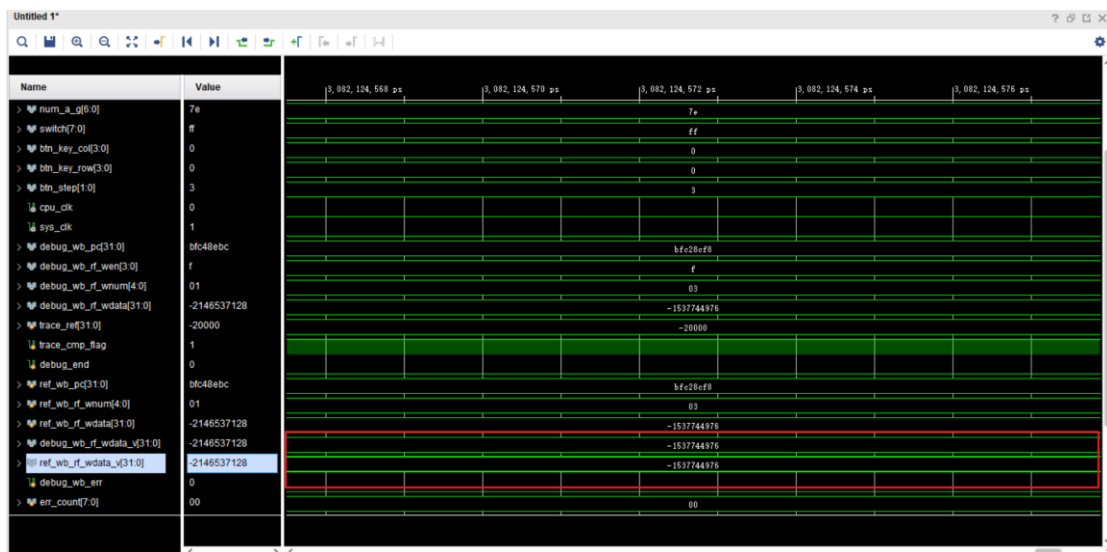
```

    OP_LW: mem_sel <= 4'b1111; // 读取的是一个字，所以选取 4 个存储体
    ...
endcase
end

```

在写回级根据之前生成的访存信号将从内存读入的数据扩充为 32 位以写入对应寄存器。至此整条指令执行完成。

## 2.3 Trace 比对的方法进行仿真测试（需列出指令的测试波形以及程序段的测试结果并分别说明）



LW 测试点对应的写回至寄存器的值和 trace 比对中的值相等，说明结果是正确的。

## 2.4 FPGA 上板验证过程



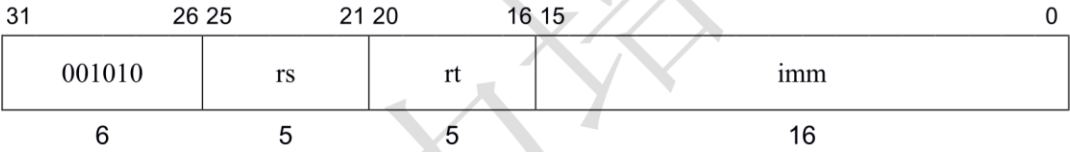
上板后会将参考值输出在 7 段数码管上，当有错误测试点时会停止。结果表明我们的上板测试是正确的。

**Part 2 个人任务--扩展实验(在 TinyMIPS 基础上扩展指令, 建议运算类、跳转类、访存类均有)**

总述:共实现了 5 条指令扩展，分别是:SLTI, SLTIU, LH, LHU, SH 其中指令 SLTI 的设计实现过程如下：

**1 指令格式**

SLTI 是一条 I 型指令，用于比较，指令格式如下：



汇编格式: SLTI rt, rs, imm

功能描述: 将寄存器 rs 的值与有符号扩展至 32 位的立即数 imm 进行有符号数比较，如果寄存器 rs 中的值小，则寄存器 rt 置 1；否则寄存器 rt 置 0。

**2 分析指令功能及执行过程，画出数据通路图**

IF 级：取出该指令。

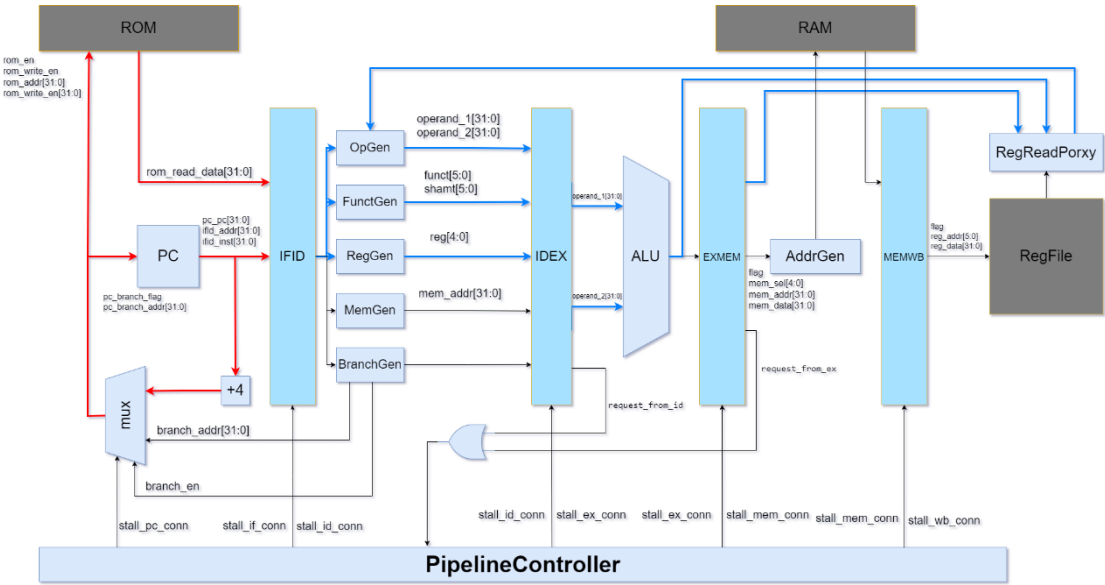
ID 级：译码产生相关信号。涉及到 rs 号寄存器的读操作，rt 寄存器的读操作；FUNCT 域设为 SLT 对应的 FUNCT 域，指示 ALU 实现有符号比较运算；操作数生成上使用 rs 寄存器操作数和 imm 立即数的符号扩展。

EX 级：完成比较运算并生成结果输出。

MEM 级：该指令不涉及访问内存，因此本级只完成信号和结果的传递。

WB 级：将 EX 级生成的运算结果写回至寄存器堆。

数据通路如下：



如图，红色为指令流，蓝色为数据流部分数据通路。

### 3 代码实现

ID 级修改部分：

```
(RegGen.v)
always @(*) begin
  case (op)
    ...
    `OP_SLTI: begin
      reg_read_en_1 <= 1; // 寄存器读使能信号
      reg_read_en_2 <= 0;
      reg_addr_1 <= rs; // 读入 rs 号寄存器
      reg_addr_2 <= 0;
    end
    ...
  endcase
end

always @(*) begin
  ...
  case (op)
    `OP_SLTI: begin
      reg_write_en <= 1; // 寄存器写使能信号
      reg_write_addr <= rt; // 写入 rt 号寄存器
    end
    ...
  endcase
end

(OperandGen.v)
always @(*) begin
  case (op)
    ...
    `OP_SLTI: begin
      operand_1 <= reg_data_1; // 第一个操作数是 rs 寄存器操作数
    end
    ...
  endcase
end

// generate operand_2
always @(*) begin
  case (op)
    ...
    `OP_SLTI: begin
      operand_2 <= sign_ext_imm; // 第二个操作数是立即数的符号扩展
    end
    ...
  endcase
end

(FnctGen.v)
always @(*) begin
  case (op)
```



```

...
`OP_SLT: funct <= `FUNCT_SLT; // 设置有符号比较运算功能域
...
    endcase
end

```

EX 级修改部分：

```

wire[`DATA_BUS] operand_2_mux =
    (funct == `FUNCT_SUB || funct == `FUNCT_SUBU || funct == `FUNCT_SLT)
    ? (~operand_2) + 1 : operand_2;
// 比较运算时第二个操作数取立即数的相反数，因为使用减法后判断结果正负来比较大
// 小

// sum of operand_1 & operand_2
wire[`DATA_BUS] result_sum = operand_1 + operand_2_mux;

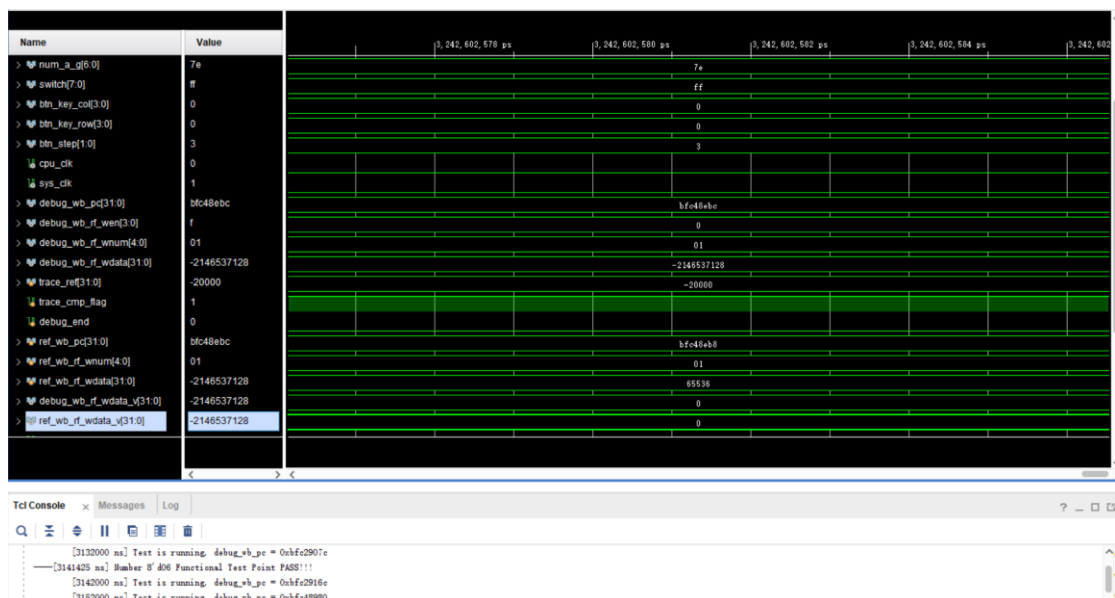
// flag of operand_1 < operand_2
wire operand_1_lt_operand_2 = funct == `FUNCT_SLT ?
    // op1 is negative & op2 is positive
    ((operand_1[31] && !operand_2[31]) ||
    // op1 & op2 is positive, op1 - op2 is negative
    (!operand_1[31] && !operand_2[31] && result_sum[31]) ||
    // op1 & op2 is negative, op1 - op2 is negative
    (operand_1[31] && operand_2[31] && result_sum[31]))
    : (operand_1 < operand_2);
// 通过符号位和运算结果实现比较运算，得到对应结果

// calculate result
always @(*) begin
    case (funct)

        // comparison
        `FUNCT_SLT, `FUNCT_SLTU: result <= {31'b0, operand_1_lt_operand_2};
        // 比较运算的结果选择
    endcase
end
end

```

**4 Trace 比对方法进行指令功能测试**（阐述采用 Trace 比对进行测试的过程，该测试部分可将所有扩展指令放在一起做。）



通过了所有的扩展指令。

## 5 自行编写 Testbench 进行指令功能仿真测试(需有测试激励, 仿真波形截图及分析, 该测试部分可将所有扩展指令放在一起做。)

使用的指令的汇编代码如下, 测试了个人扩展的所有 5 条指令。

```

nop
addiu $8, $0, 258
addiu $9, $0, 1
addiu $13, $0, 2
sh $8, 2($13)
lh $9, 2($13)
slti $7,$9, 255
sltiu $6,$9, 259
lhu $7, 2($13)

```

测试激励如下:

```

`timescale 1ns / 1ps

module core_tb_top;
    reg clk;
    reg rst;
    reg stall;

    wire rom_en;
    wire [ `MEM_SEL_BUS ] rom_write_en;
    wire [ `ADDR_BUS ] rom_addr;
    wire [ `DATA_BUS ] rom_read_data;
    wire [ `DATA_BUS ] rom_write_data;

    // RAM control
    wire ram_en;

```

```

wire  [`MEM_SEL_BUS]  ram_write_en;
wire  [`ADDR_BUS]    ram_addr;
wire  [`DATA_BUS]    ram_read_data;
wire  [`DATA_BUS]    ram_write_data;
// debug signals
wire                                debug_reg_write_en;
wire  [`REG_ADDR_BUS] debug_reg_write_addr;
wire  [`DATA_BUS]    debug_reg_write_data;
wire  [`ADDR_BUS]    debug_pc_addr;

initial begin
    stall <= 0;
    clk <= 0;
    rst <= 1;
    #6 rst <= 0;
end

always #5 clk<=~clk; // T = 10

Core core(clk,rst,stall,
    rom_en,
    rom_write_en,
    rom_addr,
    rom_read_data,
    rom_write_data,

    ram_en,
    ram_write_en,
    ram_addr,
    ram_read_data,
    ram_write_data,

    debug_reg_write_en,
    debug_reg_write_addr,
    debug_reg_write_data,
    debug_pc_addr);

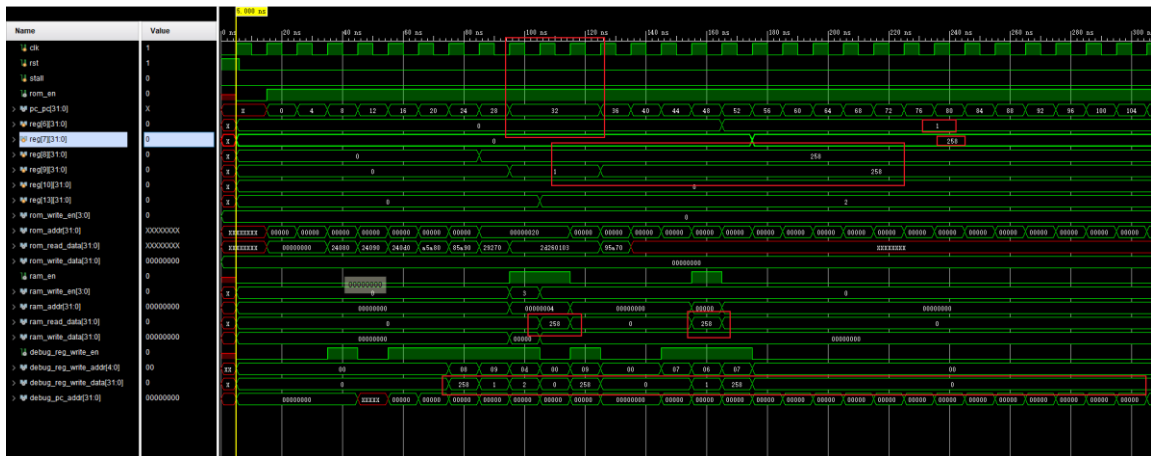
RAM ram(
    clk,
    ram_en,
    ram_write_en,
    ram_addr,
    ram_write_data,
    ram_read_data
);

ROM rom(
    clk,
    rom_en,
    rom_write_en,
    rom_addr,
    rom_write_data,
    rom_read_data
);

endmodule

```

对应的仿真波形图如下：



我们重点关注红框标注部分。首先要说明的是，为了方便测试，将起始 PC 的值设置为了 4。

分析上述指令，首先执行了：

```
addiu $8, $0, 258
addiu $9, $0, 1
addiu $13, $0, 2
```

即将 8, 9, 13 号寄存器分别设置为 258, 1, 2。在仿真波形图中可以看到当 PC 等于 28 时，寄存器的值才开始陆续被赋值成正确的值，这符合流水线工作的原理。

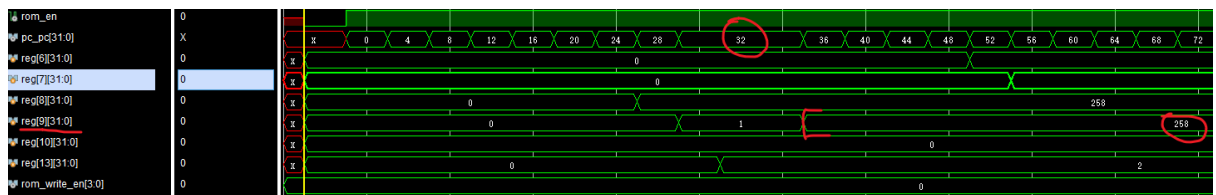
接下来执行下列指令：

```
sh $8, 2($13)
lh $9, 2($13)
```

测试 sh 指令，即在内存地址 (\$13)+2=4 处写入 8 号寄存器低半字的值，也就是 258 的低半字，写成 16 进制就是 0x0102。查看内存后可以发现成功在对应的内存地址处写入了该数值，如下图：

data_mem0[63...	00,00,00,...	Array
> [1][7:0]	02	Array
> [0][7:0]	00	Array
data_mem1[63...	00,00,00,...	Array
> [3][7:0]	00	Array
> [2][7:0]	00	Array
> [1][7:0]	01	Array
> [0][7:0]	00	Array

之后测试了 lh 指令，其将在 (\$13)+2=4 内存地址处读入半字并写入到 9 号寄存器内。这里我们从仿真波形中看到一个特殊现象，就是 PC 在 32 处停留了很长时间。产生这个的原因就是下一条指令要从寄存器中读入这次的数值，因此会从寄存器代理模块读入，但是该数值要在 MEM 级才会被读回，可是此时该指令还处于 EX 级，因此引发了流水线暂停以等待数据被真正读回。如下图：



当暂停了两个时钟周期后流水线才恢复正常工作,这时寄存器的值也变为了读入的数值即 258。这说明 sh 和 lh 都正确执行了。

再往后测试 slti 和 sltiu 指令:

```
slti $7,$9, 255
sltiu $6,$9, 259
```

这里依次将 9 号寄存器中的数值与立即数 255, 259 进行有符号/无符号比较,并将结果分别写回 7 号和 6 号寄存器。在前面的波形中可以看到两个寄存器的值分别被赋值为 0 和 1,这符合预期结果。

最后一条指令测试了 lhu 指令,它也能正确将内存对应处的值读入到 7 号寄存器。从波形中可以看到此指令正确读入了数据并将其写回至寄存器。

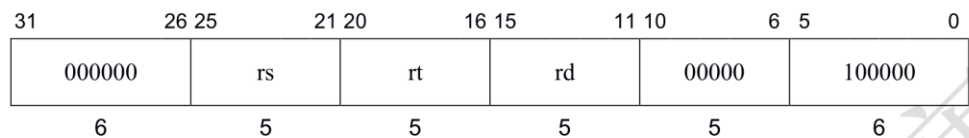
## 6 其他类型指令的分析

上面的部分分析的是一条 I 型运算指令。接下来简单分析一下 R 型运算指令和 J 型指令的执行过程。

**R 型运算指令: 以 ADD 为例**

指令格式如下:

### 3.3.1 ADD



汇编格式: ADD rd, rs, rt

该指令功能是将 rs 寄存器与 rt 寄存器中的值进行无符号相加,并将结果送入 rd 寄存器中。

首先其不会产生分支,因此 PC、IF 级无修改。

在 ID 级,由于是 R 型指令, FUNCT 域无需在此生成而是在指令中已指定,我们只需要生成 rs 和 rt 这两个寄存器操作数,其会从 RegReadProxy 模块中得到并沿着流水线向下传递至 EX 级。

在 EX 级,由于 FUNCT 域设置为有符号数的加法,因此 ALU 执行有符号数加法,并将结果沿流水线传递。

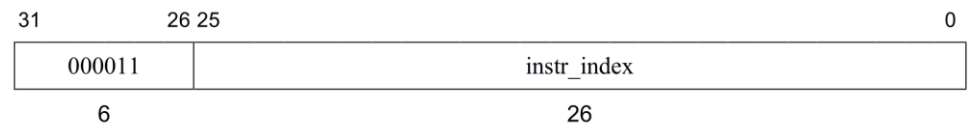
在 MEM 级,由于不涉及访存操作,因此没有影响。

在 WB 级,指定了要回写的数是 ALU 的输出结果,将计算结果写回至 rd 寄存器中,至此指令执行完毕。

**J 型指令: 以 JAL 为例**

其指令格式如下:

### 3.6.10 JAL



汇编格式: JAL target

无条件跳转，跳转目标由该分支指令对应的延迟槽指令的 PC 的最高 4 位与立即数 instr\_index 左移 2 位后的值拼接得到。同时将该分支对应延迟槽指令之后的指令的 PC 值保存至第 31 号通用寄存器中。

那么在 IF、PC 级，由于约定，在 J 型指令后会插入延迟槽指令，因此不用特别修改。而是当 JAL 指令执行到 ID 级给出跳转信号和对应地址后，将 PC 修改为对应的数值。

ID 级，需要计算出跳转到的地址并给出跳转信号。同时给出 FUNCT 域为无符号加法，使 ALU 能计算出要保存到寄存器中的返回。

EX 级，计算出返回地址。

MEM 级，无需访存，无修改，只进行数据传递。

WB 级，将返回地址保存到 31 号通用寄存器中。

至此指令结束。

该部分我们采用了类似流水线的结构以提高效率，具体结构图如下：

# Mutipiler

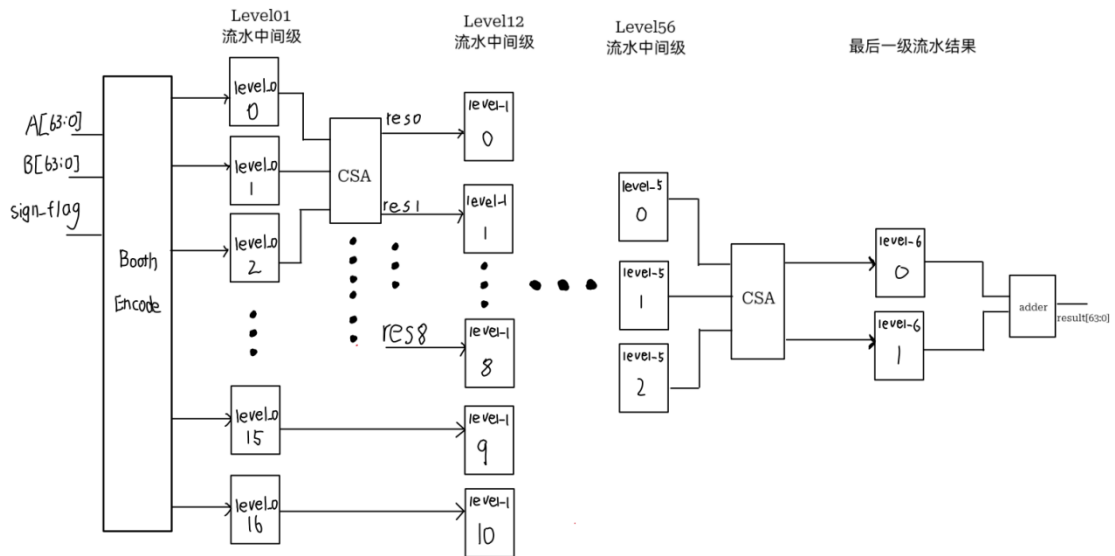


图 3.2 流水化华莱士树结构简图

之后考虑将乘法器集成至 CPU 中，我们的方法是约定好乘法器将要进行的时钟周期数。开始计算时首先给出乘法器的使能信号，这样在下一个时钟上升沿乘法器检测到使能信号有效便会开始进行计算。同时在乘法器外计时，当约定好的时钟周期到时才从乘法器的输出端口取得结果。在乘法器计算期间，EX 级前的流水线都是暂停的。

## 实验现象及分析：

我们采用了单独编写 testbench 的方法首先对乘法器进行单独测试，波形图如下：

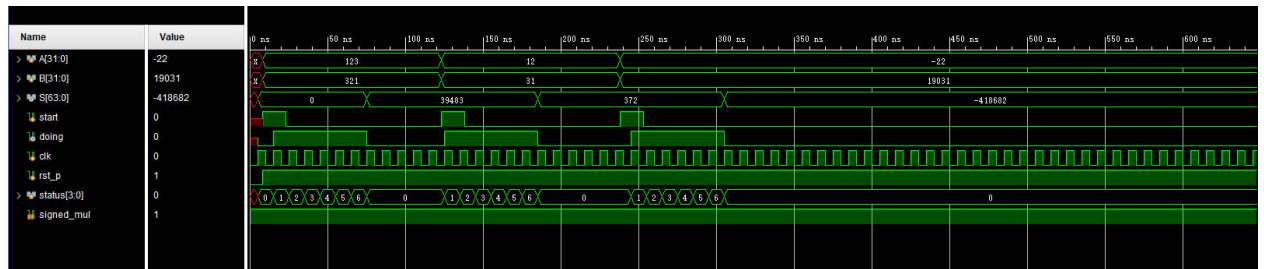


图 3.3 乘法器测试波形图

这个过程采用的 testbench 代码如下：



```

module tb_top;
    reg [31:0]A;
    reg [31:0]B;
    wire [63:0]S;
    reg start;
    wire doing;
    reg clk;
    reg rst_p;
    initial begin
        clk <= 0;
        rst_p <= 0;
        #8 rst_p <= 1;
        start <= 1;
        A<=123;
        B<=321;
        #15 start <= 0;
        #100 A<=12;
        B<=31;
        start<=1;
        #15 start<=0;
        #100 A<=0-32'd22;
        B<=19031;
        start<=1;
        #15 start<=0;
    end
    always #5 clk=~clk;
    multiplier mul (clk,rst_p,A,B,1,start,doing,S);
endmodule

```

由波形图可以看出，乘法器可以在 7 个时钟上升沿后输出正确的结果，并且可以正常的连续工作，说明我们的乘法器的功能是正确的。

### 3 除法器部分

#### a. 设计原理

我们小组设计的迭代除法器原理方法为试商法，其核心思想是将除法转为减法进行处理。换言之，便是将纸笔运算的流程用程序来实现。试商法的原理如下：

首先为方便阐述，谁当下述数据的最低位的索引为 1。假设除数为  $a=1101$ ，被除数为  $b=0100$ ，商为  $c$ ，辅助参数  $k$  初始值为 4。首先取  $a[k]$  作为被减数，取  $b$  作为减数，进行减法运算，得到减法结果  $d$ 。如果  $d>0$ ，则赋值商  $c[k]=1$ ，并且更新被减数为 {减法结果,  $a[k-1]$ }；如果  $d<0$ ，则赋值商  $c[k]=0$ ，并更新被减数为  $\{a[k], a[k-1]\}$ ；

然后赋值  $k = k - 1$ ，利用上一轮的更新值进行下一轮的减法运算。

当  $k=0$  时，循环结束，至此也获得了四位的商与余数（恰为被减数的值）。

#### b. 程序设计

首先是模块的输入输出说明：

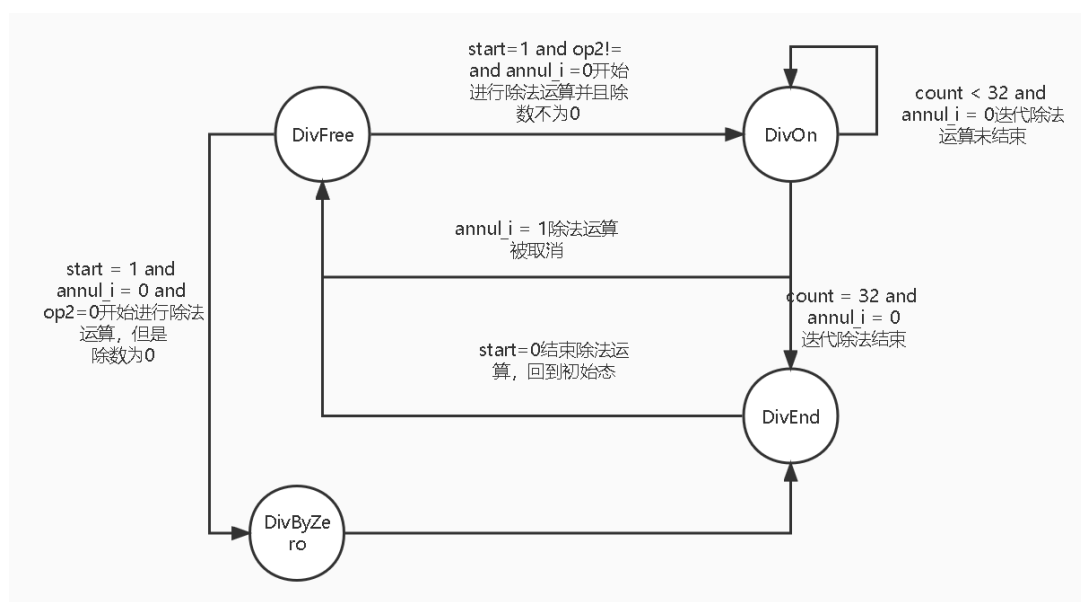
序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位
2	clk	1	输入	时钟
3	signed_div	1	输入	为 1 代表有符号
4	op1	32	输入	被除数
5	op2	32	输入	除数

6	start	1	输入	是否开始运算
7	annul	1	输入	是否取消运算
8	result_o	64	输出	除法结果, 高 32 位为余数, 低 32 为商
9	ready_o	1	输出	除法是否结束

从输入输出端口的角度来看, 整个运行流程为: 首先是 rst 有效, 使触发器状态机进入初始状态, 然后 start 为高电平有效, 除法器接收 op1, op2, 并根据 signed\_div 判断是否进行有符号运算, 在 33 个周期的运算内, 如果 annul=1, 说明异常除法, 需要清空流水线, 状态机回到初始状态, 否则则输出运算结果 result\_o, 并向外给出运算完成的信号 ready\_o。

然后是状态机说明:

为实现上述原理, 需要设计一个四状态的状态机, 具体如下图所示:



状态机说明: 首先是 DivFree 状态, 这是 rst 有效后进入的初始状态, 它的作用是初始化各个变量 (第一轮的被减数, 减数, 计数变量等), 并根据 signed\_div 判断是否对负数的操作数进行变补操作。以及根据 start, annul, 以及除数是否为 0 判断下一个时钟上升沿进入哪个状态。

然后时 DivByZero 状态, 它代表除数为 0, 在这个状态中, 会将 next\_state 置为 DivEnd。

下一个是 DivOn 状态, 这是除法运算的核心状态, 在此状态中, 将根据 cnt 的值, 进行 32 个周期的运算, 运算内容便是实验原理中提到的试商法不断迭代。这里不再赘述。最终的结果会存在 dividend 中, 除此之外, 在第 33 个上升沿时, 此状态会根据 signed\_div 以及除数, 被除数, 商, 余数的符号对结果的符号进行调整, 规则便是, op1, op2 异号, 则商为负数, 余数的符号与 op1 保持一致。

最后一个状态时 DivEnd 状态，在这个状态中，会将结果输出，并给出运算结束的信号，当再接收 start=0 后，其会将输出信号全部置为 0 并且进入 DivFree 状态。

### c. 最后说明与其他模块的交互

与其他模块的交互为三部分：运算结果写入 HILO 寄存器；流水线暂停；异常处理，取消除法运算。

对于第一部分而言，其与除法器关系其实不大，这里不再赘述。

第二部分，因为除法运算需要 33 个周期，所以在进行除法运算的时候需要将流水线暂停。这个操作在 EX 级实现，首先 EX 会判断运算类型是否为 DIVU, DIV，如果是，stall\_all 置为 1 暂停流水线，并且 start 置为 1，除法器开始工作，当除法器完成运算后，EX 级别，根据 ready\_o 将 stall\_all 置为 0，开启流水线，与此同时 start 变为 0，除法器回到 DivFree 状态。

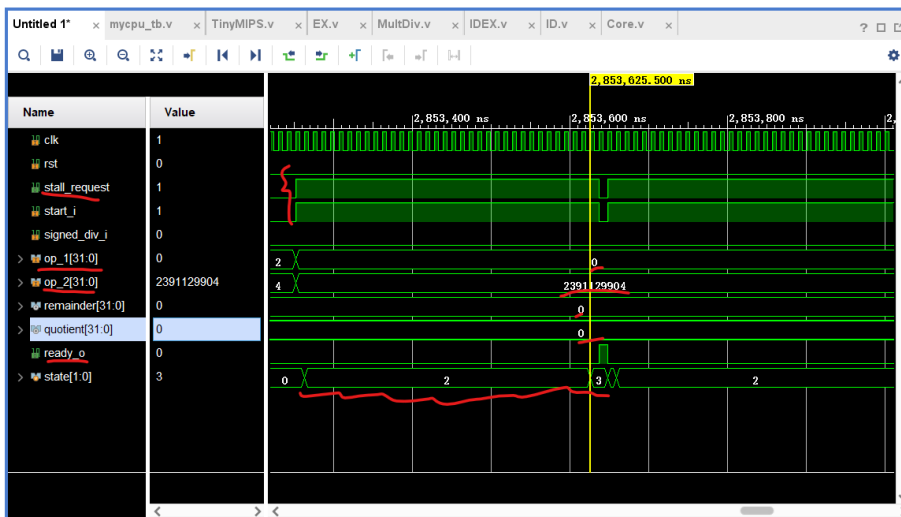
第三部分主要涉及 annual 信号，当某处发生异常时，需要将当前流水线执行的指令全部清空，所以此时 annual 的值会变为 1，除法器回到 DivFree 状态，并且所有变量值清空。

### d. 实验现象及分析

直接使用 gold\_trace 进行仿真并查看波形图进行分析：

波形图 1:

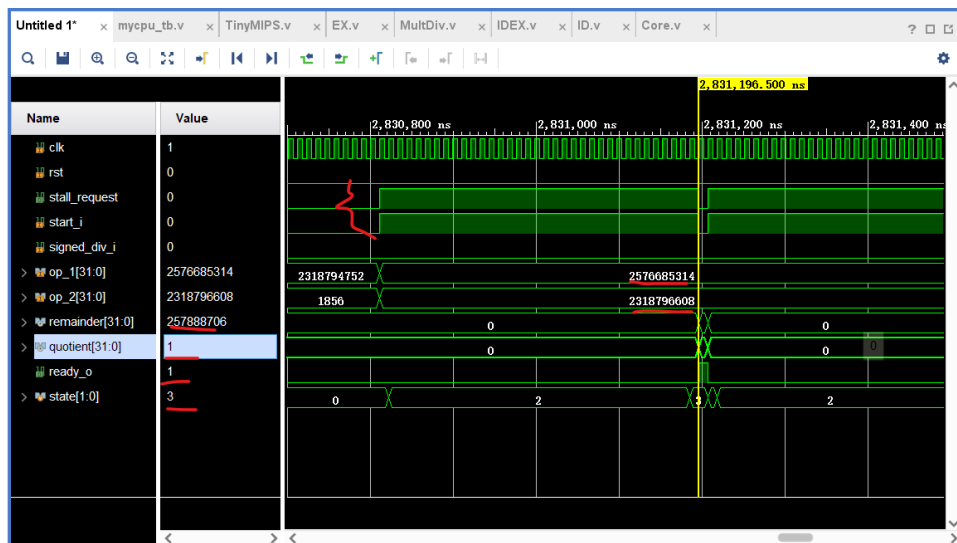
被除数为 0，除数为 2391129904，无符号运算。



通过上述波形图，我们可以看到除法器的余数，商均正确，并且 ready\_o 也正确给出，状态 state 的变化也正确。流水线的暂停也暂停与开启也顺利实现。

波形图 2:

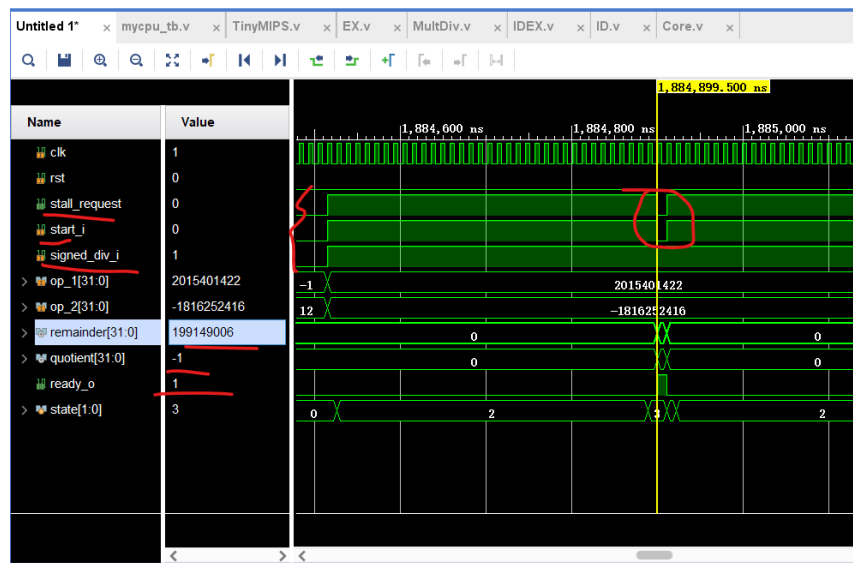
被除数为 2576685314，除数为 2318796608，无符号运算。



通过上述波形图，我们可以看到除法器的余数为 257888706，商为 1，均正确，并且 ready\_o 也正确给出，状态 state 的变化也正确。流水线的暂停也暂停与开启也顺利实现。

波形图 3:

被除数为 2015401422，除数为 -1816252416，有符号运算。



通过上述波形图，我们可以看到除法器的余数为 199149006，商为 -1，均正确，并且 ready\_o 也正确给出，状态 state 的变化也正确。流水线的暂停也暂停与开启也顺利实现。

## 4 HILO 寄存器部分

### 设计过程:

首先是创建一个 hilo 寄存器模块，用于将 hi 和 lo 寄存器的值输出以及将乘

除法器的值进行保存。

```
module HILO (
    input                clk,
    input                rst,
    input                write_en,
    input    [`DATA_BUS] hi_i,
    input    [`DATA_BUS] lo_i,
    output   [`DATA_BUS] hi_o,
    output   [`DATA_BUS] lo_o
);
    reg [`DATA_BUS] hi;
    reg [`DATA_BUS] lo;
    assign hi_o = hi;
    assign lo_o = lo;

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

endmodule // HILO
```

然后在顶层模块创建 hilo 相关的信号和变量进行数据和信号传递，主要在 EX 级。定义：

```
wire [`DATA_BUS] hilo_rp_hi, hilo_rp_lo, ex_hi, ex_lo, exmem_hi, exmem_lo;
```

```
wire ex_hilo_write_en, exmem_hilo_write_en;
```

以及在 MEM 阶段定义：

```
wire [`DATA_BUS] mem_hi, mem_lo, memwb_hi, memwb_lo;
```

```
wire mem_hilo_write_en, memwb_hilo_write_en;
```

还需要创建数据前递的模块 hiloreadproxy：

```
module HILOReadProxy (
    input    [`DATA_BUS] hi_i,
    input    [`DATA_BUS] lo_i,
    input                mem_hilo_write_en,
    input    [`DATA_BUS] mem_hi_i,
    input    [`DATA_BUS] mem_lo_i,
    input                wb_hilo_write_en,
    input    [`DATA_BUS] wb_hi_i,
    input    [`DATA_BUS] wb_lo_i,
    output   [`DATA_BUS] hi_o,
    output   [`DATA_BUS] lo_o
);
    assign hi_o = mem_hilo_write_en ? mem_hi_i :
                  wb_hilo_write_en ? wb_hi_i :
                  hi_i;
    assign lo_o = mem_hilo_write_en ? mem_lo_i :
                  wb_hilo_write_en ? wb_lo_i :
                  lo_i;
endmodule // HILOReadProxy
```

最后再顶层模块创建实例，并在 EX 级传送回去：

```
wire [`DATA_BUS] hilo_hi, hilo_lo;

HILO u_HILO (
    .clk            ( clk            ),
    .rst            ( rst            ),
    .write_en       ( wb_hilo_write_en ),
    .hi_i           ( wb_hi          ),
    .lo_i           ( wb_lo          ),
    .hi_o           ( hilo_hi        ),
    .lo_o           ( hilo_lo        )
);
HILOReadProxy u_HILOReadProxy (
    .hi_i           ( hilo_hi        ),
    .lo_i           ( hilo_lo        ),
    .mem_hilo_write_en ( mem_hilo_write_en ),
    .mem_hi_i       ( mem_hi         ),
    .mem_lo_i       ( mem_lo         ),
    .wb_hilo_write_en ( wb_hilo_write_en ),
    .wb_hi_i        ( wb_hi          ),
    .wb_lo_i        ( wb_lo          ),
    .hi_o           ( hilo_rp_hi     ),
    .lo_o           ( hilo_rp_lo     )
);
```

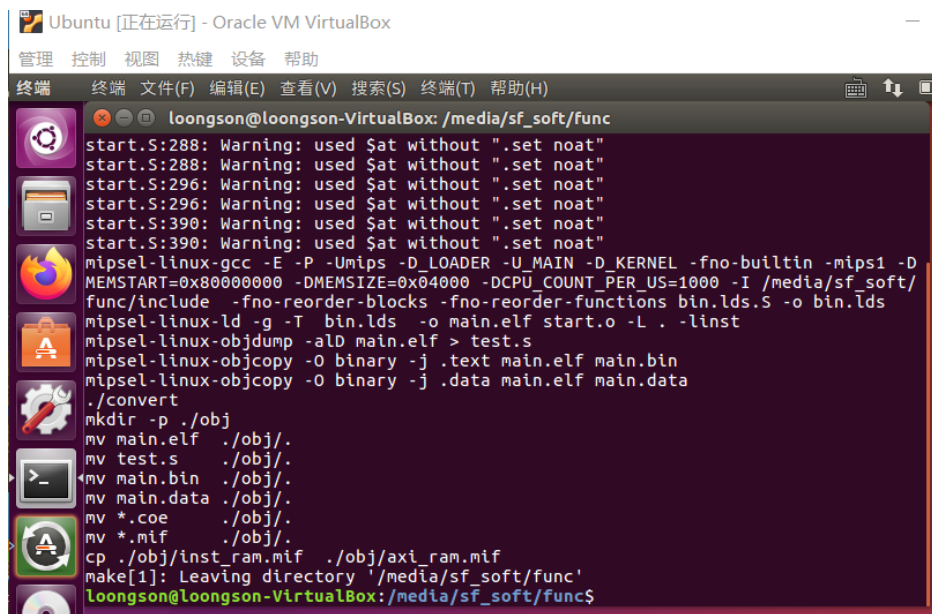
在 EX 级输入的 hilo 即为数据前递模块的输出 hilo\_rp\_hi 和 hilo\_rp\_lo。

### 实验现象及分析：

通过了 MTHI,MFHI,MTLO,MFLO 测试点,如下是测试 start.s 部分：

```
kseg0_kseg1:
    jal n48_mfhi_test
    nop
    jal wait_1s
    nop
jal n49_mflo_test
    nop
    jal wait_1s
    nop
jal n50_mthi_test
    nop
    jal wait_1s
    nop
jal n51_mtlo_test
    nop
    jal wait_1s
    nop
```

下面是测试过程截图：



这里为生成 gold\_trace:

```
Test begin!
----[ 11935 ns] Number 8'd01 Functional Test Point PASS!!!
----[ 21495 ns] Number 8'd02 Functional Test Point PASS!!!
      [ 22000 ns] Test is running, debug_wb_pc = 0xbfc01eec
----[ 30435 ns] Number 8'd03 Functional Test Point PASS!!!
      [ 32000 ns] Test is running, debug_wb_pc = 0xbfc018f4
----[ 35615 ns] Number 8'd04 Functional Test Point PASS!!!
=====
gettrace end!
----Succeed in generating trace file!
$finish called at time : 36355 ns : File "E:/CDE/cpu132_gettrace/testbench/tb_top.v" Line 221
```

最后为测试结果:

```
      [ 732000 ns] Test is running, debug_wb_pc = 0xbfc02af4
      [ 742000 ns] Test is running, debug_wb_pc = 0xbfc02b8c
----[ 750665 ns] Number 8'd03 Functional Test Point PASS!!!
      [ 752000 ns] Test is running, debug_wb_pc = 0xbfc00754
      [ 762000 ns] Test is running, debug_wb_pc = 0xbfc0172c
      [ 772000 ns] Test is running, debug_wb_pc = 0xbfc017c0
      [ 782000 ns] Test is running, debug_wb_pc = 0xbfc01858
      [ 792000 ns] Test is running, debug_wb_pc = 0xbfc018f0
      [ 802000 ns] Test is running, debug_wb_pc = 0xbfc01988
      [ 812000 ns] Test is running, debug_wb_pc = 0xbfc01a20
      [ 822000 ns] Test is running, debug_wb_pc = 0xbfc01ab8
      [ 832000 ns] Test is running, debug_wb_pc = 0xbfc01b50
      [ 842000 ns] Test is running, debug_wb_pc = 0xbfc01be8
      [ 852000 ns] Test is running, debug_wb_pc = 0xbfc01c7c
      [ 862000 ns] Test is running, debug_wb_pc = 0xbfc01d14
      [ 872000 ns] Test is running, debug_wb_pc = 0xbfc01dac
      [ 882000 ns] Test is running, debug_wb_pc = 0xbfc01e44
----[ 887025 ns] Number 8'd04 Functional Test Point PASS!!!
      [ 892000 ns] Test is running, debug_wb_pc = 0x00000000
      [ 902000 ns] Test is running, debug_wb_pc = 0xbfc007cc
=====
Test end!
----PASS!!!
$finish called at time : 906170500 ps : File "E:/CDE/soc_axi_func/testbench/mycpu_tb.v" Line 269
```

## 6 CP0 协处理器和异常扩展部分

设计过程:

### 1. 协处理器部分:

本部分首先介绍 MIPS32 架构中的协处理器, 说明了协处理器的作用。由于 OpenMIPS 计划实现其中的一个协处理器——CP0, 其实现方式有点类似 HI、LO 寄存器的实现方式。接着说明协处理器访问指令 mfc0、mtc0 的格式、作用、用法。最后节给出了协处理器访问

指令的实现思路,以及对系统结构的修改。通过修改 OpenMIPS, 实现了协处理器访问指令, 最后编写测试程序, 在 ModelSim 中进行仿真验证。协处理器一词通常用来表示处理器的一个可选部件, 负责处理指令集的某个扩展, 具有与处理器核独立的寄存器。MIPS32 架构提供了最多 4 个协处理器, 分别是 CP0 CP3, 作用如下图:

协处理器	作用
cp0	系统控制
cp1	fpu
cp2	特定实现
cp3	fpu

协处理器 CP0 用作系统控制, CP1、CP3 用作浮点处理单元, 而 CP2 被保留用于特定实现。除 CP0 外的协处理器都是可选的, OpenMIPS 没有实现浮点运算, 所以 CP1、CP3 不用实现, CP2 也没有作用, 不用实现。而 CP0 是不可选的, 需要实现, 所以下面重点介绍协处理器 CP0 . 配置 CPU 工作状态: 符合 MIPS32 架构的硬件通常是很灵活的, 可以通过读/写一个或一些内部寄存器来改变 一些很根本的 CPU 特性( 如: 将字节次序从 MSB 变为 LSB, 或者从 LSB 变为 MSB)。高速缓存控制: 符合 MIPS32 架构的 CPU 一般会集成缓存控制器, 用来控制、读、写缓存。 异常控制: 异常发生时的检测和处理都由 CP0 的一些控制寄存器来定义和控制。存储管理单元控制: 对系统的存储区域进行合理的控制管理分配, 主要对 MMU、TLB 的一些配置、管理、访问。

2. 协处理器 CP0 中的寄存器:

OpenMIPS 的设计目标是一个轻量级的处理器, 并不打算实现缓存、MMU、TLB、调试等复杂功能, 所以相关的寄存器都可以不用实现。下面依次介绍这几个主要寄存器的格式、作用。

1. Count 寄存器

Count 寄存器是一个不停计数的 32 位寄存器, 计数频率一般与 CPU 时钟频率相同, 当计数达到 32 位无符号数的上限时, 会从 0 升始重新计数。Count 寄存器可读、可写。

```
// COUNT
always @(posedge clk) begin
    if (rst) begin
        reg_count <= 33'h0;
    end
    else if (cp0_write_en && cp0_write_addr == `CP0_REG_COUNT) begin
        reg_count <= {cp0_write_data, 1'b0};
    end
    else begin
        reg_count <= reg_count + 1;
    end
end
end
```



CP0.v - 30行.

2. Count, 软件可读寄存器, 数值以固定频率↑

Count 数值增加作为不受 CPU 中行为 (流水线刷新 暂停!)

两周期 加 1. 在系统测试功能中, 可以实现断点计数

发生异常开始计数

33位, 低位补0, 不停计数的32位寄存器

写数据32位

## 2. Status 寄存器

Status 寄存器也是个 32 位、可读、可写的寄存器, 用来控制处理器的操作模式、中断使能以及诊断状态。

```
// STATUS
always @(posedge clk) begin
    if (rst) begin
        reg_status <= 32'h0040ff00;
    end
    else if (exc == `EXC_ERET) begin
        reg_status[1] <= 0;
    end
    else if (exc != `EXC_NULL) begin
        reg_status[1] <= 1;
    end
    else if (cp0_write_en && cp0_write_addr == `CP0_REG_STATUS) begin
        reg_status[22] <= cp0_write_data[22];
        reg_status[15:8] <= cp0_write_data[15:8];
        reg_status[1:0] <= cp0_write_data[1:0];
    end
    else begin
        reg_cause <= reg_cause;
    end
end
```

Compare 没有  
可读字 写入不变 与 Count 低 32 位比较  
两数相等 计时器中断

3. Status. CP0.V 31 行 37 行赋值 1 状态  
Status 是软件可读写的寄存器 32 位 第 5 个

**MEM 级**

使能以及诊断状态。其字段如表 10-5 所示。

表 10-5 Status 寄存器的各个字段

Bit	31-28	27	26	25	24-23	22	21	20	19
标志名	CU3-CU0	RP	R	RE	0	BEV	TS	SR	NMI
Bit	18-16	15-8	7-5	4	3	2	1	0	
标志名	0	IM7-IM0	R	UM	R	ERL	EXL	IE	

31-28 位 分别是 CP3 到 CP0. 为 0 时表示协处理器不可用.  
对于 OPEN MIPS 处理器. 只有协处理器 CP0 可以设置本字段 4'b0001  
15-8 位 IM7-IM0. 表示是否屏蔽相应中断.  
(mips 处理器的 8 个中断源. 6 个硬件. 2 个软件)  
处理器相应中断: Status IM 字段与 Cause IP 字段相应位  
条件: 为 1, Status IE 字段为 1 时  
处理器进入内核模式.  
EXL: 是否处于异常级. 异常发生时置 1, 禁止中断.  
IE: 全局中断使能.

IS: 10 9-8  
6 个 2 个  
硬件中断 软件

表中标识为 R 的字段是保留字段，下面逐一介绍其中的非保留字段。读者朋友如果没有时间，可以只理解其中使用灰色背景的字段，TinyMIPS 处理器也只实现了这些字段。

CU3-CU0 表示协处理器是否可用 (Coprocessor Usability)，分别控制协处理器 CP3、CP2、CP1、CP0。为 0 时，表示相应的协处理器不可用；为 1 时，表示相应的协处理器可用。对于 OpenMIPS 处理器而言，只有协处理器 CP0，所以可以设置本字段为 4'b0001。

- SR: 表示是否是软重启 (Soft Reset)，为 1 表示重启异常是由软重启引起的。
- NMI: 表示是否是不可屏蔽中断 (Non-Maskable Interrupt)，为 1 表示重启异常是由不可屏蔽中断引起的。
- IM7-IM0: 表示是否屏蔽相应中断 (Interrupt Mask)，0 表示屏蔽，1 表示不屏蔽，MIPS 处理器可以有 8 个中断源，对应 IM 字段的 8 位，其中 6 个中断源是处理器外部硬件中断，另外 2 个是软件中断，中断是否能够被处理器响应是由 Status 寄存器与 Cause 寄存器共同决定的，如果 Status 寄存器的 IM 字段与 Cause 寄存器的 IP 字段的相应位都为 1，而且 Status 寄存器的 IE 字段也为 1 时，处理器才响应相应中断。
- ERL: 表示是否处于错误级，当处理器接收到坏的数据时设置本字段为 1。有一些 MIPS 处理器在接收来自缓存或内存中的数据块时，能够检验数据中附带的奇偶校验位或纠错码，当发现数据错误且无法纠正时，处理器就设置 ERL 字段为 1，并进入奇偶校验/ECC 错误的异常处理过程，这是一个特殊的异常处理过程（有别于一般的异常处理过程）。读者只需知道，OpenMIPS 处理器没有对奇偶校验位或纠错码的检验过程，所以不用考虑 ERL 字段。
- EXL: 表示是否处于异常级 (Exception Level)，当异常发生时，会设置本字段为 1，表示处理器处于异常级，此时，处理器会进入内核模式下工作，并且禁止中断。
- IE: 表示是否使能中断 (Interrupt Enable)，这是全局中断使能标志位。为 1 表示中断使能，为 0 表示中断禁止。

### 3. Cause 寄存器(标号 13)

Cause 寄存器主要记录最近一次异常发生的原因，也控制软件中断请求。Cause 寄存器的各字段如表，除了 IV 和 WP，其余字段都是只读的。

4. Cause 用于描述最近一次异常的原因

包括 IP7-IP0 共 8 个中断位。有 1 表示有待处理中断。

对这两位写入可以进行中断。IP7 在定时器中断。

IP7-IP2 为外部中断。

IP1: 是软件。  
IP0: 是硬。

表 10-6 Cause 寄存器的各个字段

标志	31	30	29-28	27	26	25-24	23	22	21-16
标志	BD	R	CE	DC	PCI	0	IV	WP	0
标志	15-10	9-8	7	6-2	1-0				
标志	IP[7:2]		IP[1:0]	ExcCode	0				

是否处于延迟槽。

记录哪种异常

```
// CAUSE
always @(posedge clk) begin
    if (rst) begin
        reg_cause <= 32'h0;
    end
    else if (exc == `EXC_INT) begin
        reg_cause[31] <= delayslot_flag;
        reg_cause[6:2] <= `CP0_EXCCODE_INT;
    end
    else if (exc == `EXC_RI) begin
        reg_cause[31] <= delayslot_flag;
        reg_cause[6:2] <= `CP0_EXCCODE_RI;
    end
    else if (exc == `EXC_BP) begin
        reg_cause[31] <= delayslot_flag;
        reg_cause[6:2] <= `CP0_EXCCODE_BP;
    end
    else if (exc == `EXC_SYS) begin
        reg_cause[31] <= delayslot_flag;
        reg_cause[6:2] <= `CP0_EXCCODE_SYS;
    end
    else if (exc == `EXC_OV) begin
        reg_cause[31] <= delayslot_flag;
        reg_cause[6:2] <= `CP0_EXCCODE_OV;
    end
    else if (exc == `EXC_ADEL) begin
        reg_cause[31] <= delayslot_flag;
        reg_cause[6:2] <= `CP0_EXCCODE_ADEL;
    end
    else if (exc == `EXC_ADES) begin
        reg_cause[31] <= delayslot_flag;
        reg_cause[6:2] <= `CP0_EXCCODE_ADES;
    end
    else if (cp0_write_en && cp0_write_addr == `CP0_REG_CAUSE) begin
        reg_cause[9:8] <= cp0_write_data[9:8];
    end
end
```

```

    else begin
        reg_cause <= reg_cause;
    end
End

```

BD: 当发生异常的指令处于分支延迟槽 (Branch DelaySlot) 时, 该字段被置为 1

CE: 当协处理器不可用异常发生时, 将发生协处理器错误 (Coprocessor Error) 的协处理器序号存储到本字段。

PCI: 这是在 MIPS32/64 架构中新增加的字段, 当协处理器 CP0 的性能计数器溢出时 (Performance Count Interrupt), 设置本字段为 1, 以产生中断。

WP: 观测挂起 (Watch Pending) 字段, 该字段与调试有关, 为 1 表示有一个观测点被触发, 处理器处于异常模式。

IP[7:2]: 中断挂起 (Interrupt Pending) 字段, 相应位用来指明外部硬件中断是否发生, 1 表示发生, 0 表示没有发生。本字段的 6 位与外部硬件中断的对应关系如下。

IP[7] —— 5 号硬件中断 IP[6] —— 4 号硬件中断

IP[5] —— 3 号硬件中断 IP[4] —— 2 号硬件中断

IP[3] —— 1 号硬件中断 IP[2] —— 0 号硬件中断

IP[1:0] 也是中断挂起字段, 但是对应的是软件中断。

IP[1] —— 1 号软件中断 IP[0] —— 0 号软件中断

#### 4. EPC 寄存器

EPC 是异常程序计数器 (Exception Program Counter), 用来存储异常返回地址, 一般情况下, 存储发生异常的指令的地址, 但是, 如果发生异常的指令位于延迟槽中, 那么 EPC 存储的是前一条转移指令的地址。该寄存器可读、可写。其字段如表所示。

```

// EPC
always @(posedge clk) begin
    if (rst) begin
        reg_epc <= 32'h0;
    end
    else if (exc != `EXC_NULL && exc != `EXC_ERET) begin
        reg_epc <= exc_epc;
    end
    else if (cp0_write_en && cp0_write_addr == `CP0_REG_EPC) begin
        reg_epc <= cp0_write_data;
    end
    else begin
        reg_epc <= reg_epc;
    end
end
End

```

5. EPC: 存的是系统异常发生时, 系统正在执行的指令的地址。

<sup>CSDN</sup>  
保存上次异常时的程序计数器。

用来寄存异常返回地址, 一般情况下有前一条转移指令的地址。  
发生异常的指令在延迟槽的话

#### 5. BADVADDR 寄存器

```

// BADVADDR
always @(posedge clk) begin
    if (rst) begin
        reg_badvaddr <= 32'h0;
    end
    else if (exc == `EXC_ADEL || exc == `EXC_IF || exc == `EXC_ADES) begin
        reg_badvaddr <= cp0_badvaddr;
    end
    // only read
    else begin
        reg_badvaddr <= reg_badvaddr;
    end
end
End

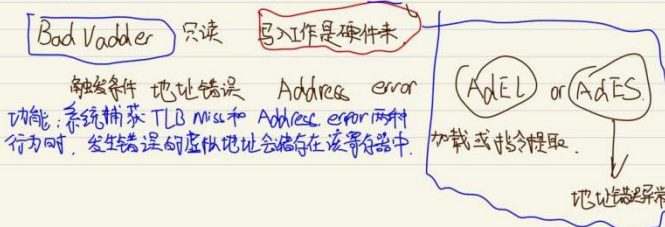
```

MIPS CPO 协处理器.

是一系列寄存器和其读写控制逻辑组合而成的寄存器堆,其中的寄存器反映着处理器运行的状态,也是上层软件对协处理器行为的控制接口.

其实和通用 PC 寄存器一样.

总共 5 个 Bad Vaddr, Count, Status, Cause, EPC



### 3. 协处理器 CP0 中的寄存器:

名称	类型	宽度	方向	用途
clk	wire		1i	时钟信号
rst	wire		1i	复位
cp0_write_en	wire		1i	cp0 寄存器写使能
cp0_read_addr	wire		8i	cp0 寄存器读地址
cp0_write_addr	wire		8i	cp0 寄存器写地址
cp0_write_data	wire		32i	cp0 寄存器写数据
data_o	reg		32o	cp0 读出结果

### 4. 协处理器访问指令

要实现 CP0 的控制功能, 需要对 CP0 中的有关寄存器进行设置, 这涉及对 CP0 中寄存器的访问, 需要使用协处理器访问指令。MIPS32 指令集架构中定义了 2 条协处理器访问指令: mtc0、mfc0, 前者实现修改 CP0 中的寄存器, 后者实现读取 CP0 中的寄存器。指令格式如:

MFC0

```

31 26 25 21 20 16 15 11 10 3 2 0
010000 00000 Rt rd 00000000 sel
6 5 5 5 8 3

```

汇编格式: MFC0 rt, rd, sel

功能描述: 从协处理器 0 的寄存器取值

操作定义:  $GPR[rt] \leftarrow CP0[rd, sel]$

MTC0

31 26 25 21 20 16 15 11 10 3 2 0

010000 00100 Rt rd 00000000 sel

6 5 5 5 8 3

汇编格式: MTC0 rt, rd, sel

功能描述: 向协处理器 0 的寄存器存值

操作定义:  $CP0[rd, sel] \leftarrow GPR[rt]$

```
always @(*) begin
    case (op)
        `OP_CP0: begin
            if (rs == `CP0_MTC0 && inst[10:3] == 0) begin
                cp0_write_en <= 1;
                cp0_read_en <= 0;
                cp0_write_data <= reg_data_1;
                cp0_addr <= {rd, inst[2:0]};
            end
            else if (rs == `CP0_MFC0 && inst[10:3] == 0) begin
                cp0_write_en <= 0;
                cp0_read_en <= 1;
                cp0_write_data <= 0;
                cp0_addr <= {rd, inst[2:0]};
            end
            else begin
                cp0_write_en <= 0;
                cp0_read_en <= 0;
                cp0_write_data <= 0;
                cp0_addr <= 0;
            end
        end
    end
    default:begin
        cp0_write_en <= 0;
        cp0_read_en <= 0;
        cp0_write_data <= 0;
        cp0_addr <= 0;
    end
endcase
end
```

对CP0的所有操作都在回写阶段

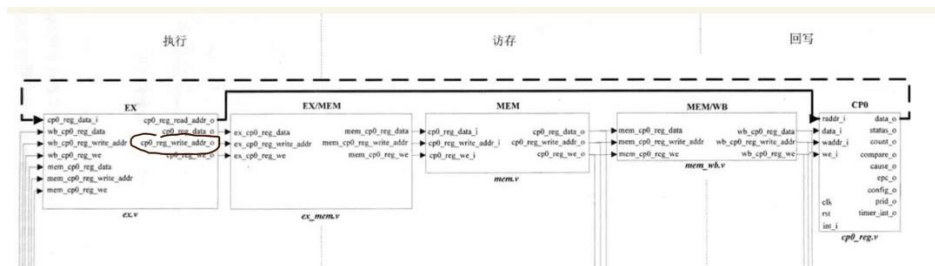
1° mtc0 译码 读通用寄存器  
执行阶段 确定写入CP0寄存器的值

回写阶段 改CP0中地址为rd值

2° mfc0 <sup>Ex</sup> 执行阶段获取 CP0中指定寄存器 (写入通用寄存器)

IF ID EX MEM WB

取址 用地址寄存器 译码寄存器 执行 ALU 访存 数据存储器 写回 寄存器堆



Ex

读CP0的mfc0 在执行阶段通过 cp0\_reg\_read\_addr\_o

输出要读取的CP0地址

(CP0通过 data\_o接口输出数据) 通过 Ex的 cp0\_reg\_data\_i 进入Ex

改CP0的指令 mtc0 执行阶段Ex通过 cp0\_reg\_we\_o

mtc0 实现思路

- (1) 在译码阶段依据指令，读出地址为 rt 的通用寄存器的值。
- (2) 在执行阶段确定要写入 CP0 中寄存器的值，其实就是译码阶段读出的地址为 rt 的通用寄存器的值，将这些信息传递到访存阶段。
- (3) 访存阶段再将这些信息传递到回写阶段。
- (4) 回写阶段依据这些信息修改 CP0 中的地址为 rd 的寄存器。

mfc0 实现思路

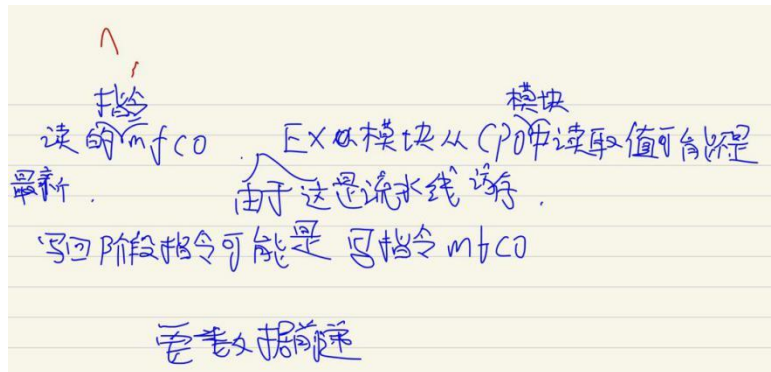
- (1) 在执行阶段获取 CP0 中指定寄存器的值，作为要写入目的通用寄存器的数据，并将这些信息传递到访存阶段。
- (2) 访存阶段再将这些信息传递到回写阶段。
- (3) 回写阶段依据这些信息修改地址为 rt 的通用寄存器。

## 5. 数据前递模块 CP0proxy

```
assign cp0_read_data_o = (mem_cp0_write_en && mem_cp0_write_addr == cp0_read_addr) ? mem_cp0_write_data :  
                          (wb_cp0_write_en && wb_cp0_write_addr == cp0_read_addr) ? wb_cp0_write_data : cp0_read_data_i;  
  
//在方寸之前要读  
// generate data output of cp0 registers (MEM stage)  
assign cp0_status_o = (wb_cp0_write_en && wb_cp0_write_addr == 'CP0_REG_STATUS') ?  
                      wb_cp0_write_data : cp0_status_i;  
                      (wb_cp0_write_en && wb_cp0_write_addr == 'CP0_REG_CAUSE') ?  
                      wb_cp0_write_data : cp0_cause_i;  
                      (wb_cp0_write_en && wb_cp0_write_addr == 'CP0_REG_EPC') ?  
                      wb_cp0_write_data : cp0_epc_i;
```



对于读取 CP0 中寄存器的指令 mfc0, EX 模块从 CP0 模块中读取的值可能不是最新的值, 因为此时处于流水线访存、回写阶段的指令可能是 mtc0, 也就是说可能会修改 CP0 中的寄存器。将访存、回写阶段对 CP0 中寄存器的写信息前推到执行阶段的 EX 模块, 由 EX 模块判断得到最新的值。这也就是图 10-4 中, MEM 模块、MEM/WB 模块的输出会回送到 EX 模块的原因。



修改译码阶段, 在 RegGen 中实现:

```
`OP_CP0: begin
    reg_read_en_1 <= 1;
    reg_read_en_2 <= 0;
    reg_addr_1 <= rt;
    reg_addr_2 <= 0;
End

`OP_CP0: begin
    if (rs == `CP0_MFC0) begin
        reg_write_en <= 1;
        reg_write_addr <= rt;
    end
    else begin
        reg_write_en <= 0;
        reg_write_addr <= 0;
    end
end
end
```

执行阶段:

```
// cp0 signal
input cp0_write_en_in,
input cp0_read_en_in,
input [ `CP0_ADDR_BUS ] cp0_addr_in,
input [ `DATA_BUS ] cp0_write_data_in,
input [ `DATA_BUS ] cp0_read_data_in,

// to cp0
assign cp0_write_en_out = cp0_write_en_in;
assign cp0_write_data_out = cp0_write_data_in;
assign cp0_addr_out = cp0_addr_in;

// cp0 signal
output cp0_write_en_out,
output [ `DATA_BUS ] cp0_write_data_out,
output [ `CP0_ADDR_BUS ] cp0_addr_out,
```

修改 EX/MEM 模块:

```
// cp0
input cp0_write_en_in,
input [ `DATA_BUS ] cp0_write_data_in,
input [ `CP0_ADDR_BUS ] cp0_addr_in,

// cp0
output cp0_write_en_out,
output [ `DATA_BUS ] cp0_write_data_out,
output [ `CP0_ADDR_BUS ] cp0_addr_out,
```



```

PipelineDeliver #(1) ff_cp0_write_en(
    clk, rst, flush,
    stall_current_stage, stall_next_stage,
    cp0_write_en_in, cp0_write_en_out
);

PipelineDeliver #(CPO_ADDR_BUS_WIDTH) ff_cp0_addr(
    clk, rst, flush,
    stall_current_stage, stall_next_stage,
    cp0_addr_in, cp0_addr_out
);

```

修改访存阶段：

MEM 模块会将执行阶段传递过来的，对 CP0 中寄存器的写信息继续传递到流水线下一级。

```

//cp0
input          cp0_write_en_in,
input  [`DATA_BUS] cp0_write_data_in,
input  [`CPO_ADDR_BUS] cp0_addr_in,
input  [`EXC_BUS] exc_in,
input  [`DATA_BUS] cp0_status_in,
input  [`DATA_BUS] cp0_cause_in,
input          stall_all,

// cp0
output          cp0_write_en_out,
output  [`DATA_BUS] cp0_write_data_out,
output  [`CPO_ADDR_BUS] cp0_addr_out,
output reg [`EXC_BUS] exc_out,
output reg [`DATA_BUS] cp0_badvaddr

always @(*) begin
    if(exc_in[`EXC_POS_ADE]) begin
        cp0_badvaddr <= current_pc_addr_in;
    end
    else if((adel_flag, ades_flag) != 2'b00) begin
        cp0_badvaddr <= address;
    end
    else begin
        cp0_badvaddr <= 0;
    end
end
end

```

改 MEM/WB 模块：

MEM/WB 模块会将 MEM 模块传递过来的，对 CP0 中寄存器的写信息传递到回写阶段：

```

// cp0
input          cp0_write_en_in,
input  [`DATA_BUS] cp0_write_data_in,
input  [`CPO_ADDR_BUS] cp0_addr_in,

// cp0
output          cp0_write_en_out,
output  [`DATA_BUS] cp0_write_data_out,
output  [`CPO_ADDR_BUS] cp0_addr_out

PipelineDeliver #(1) ff_cp0_write_en(
    clk, rst, flush,
    stall_current_stage, stall_next_stage,
    cp0_write_en_in, cp0_write_en_out
);

PipelineDeliver #(CPO_ADDR_BUS_WIDTH) ff_cp0_addr(
    clk, rst, flush,
    stall_current_stage, stall_next_stage,
    cp0_addr_in, cp0_addr_out
);

PipelineDeliver #(DATA_BUS_WIDTH) ff_cp0_write_data(
    clk, rst, flush,
    stall_current_stage, stall_next_stage,
    cp0_write_data_in, cp0_write_data_out
);

```

协处理器访问指令接口：

```

module CP0Gen (
    input          [`INST_BUS] inst,
    input          [`INST_OP_BUS] op,
    input          [`REG_ADDR_BUS] rs,
    input          [`REG_ADDR_BUS] rd,
    input          [`DATA_BUS] reg_data_1,

    output reg          cp0_write_en,

```

```

        output reg                                cp0_read_en,
        output reg [`CP0_ADDR_BUS]                cp0_addr,
        output reg [`DATA_BUS]                    cp0_write_data
    );

```

## 6. 异常处理部分

### 异常基础:

首先先实现对 eret、break 和 syscall 指令的判断。这三条指令在译码阶段就可以被识别判断出来，因此分别设置 flag 作为输出，先将异常的信息保存下来，沿流水线逐级后递。这是因为当一个异常发生后，系统的顺序执行会被中断掉，此时有若干条指令处在流水线上不同阶段，处理器会转移到异常处理例程，异常处理结束后返回原程序继续执行，因为不希望异常处理例程破坏原程序的正常执行，所以对于异常发生时，流水线上没有执行完的指令，必须记住它处于流水线的哪一个阶段，以便异常处理结束后能恢复执行，这便是精确异常。如此便能保证依照指令顺序逐条对指令异常进行处理。

```

        output          eret_flag,
        output          syscall_flag,
        output          break_flag,
        output          ov_flag,
        output          ri_flag
    );

```

```

    assign eret_flag = (inst == `CP0_ERET_FULL) ? 1 : 0;
    assign syscall_flag = (op == `OP_SPECIAL && funct == `FUNCT_SYSCALL) ? 1 : 0;
    assign break_flag = (op == `OP_SPECIAL && funct == `FUNCT_BREAK) ? 1 : 0;

```

此外，还需要注意异常发生时的 pc 地址。具体而言，当异常发生时，cp0 的 epc 寄存器需要记录下此时的 pc 地址，在异常处理结束后返回到原来的 pc 地址。一种特殊情况是当发生的异常指令为延迟槽指令时，即异常指令为跳转指令的下一条指令时，返回的地址并不是当前 pc 地址。这是因为延迟槽指令不一定是实际要执行的指令，它可能因为跳转指令而被跳过。但是儒过 epc 存的地址为延迟槽指令所在地址，则异常处理完后返回到延迟槽指令执行，再执行延迟槽指令的下一条，这与原本的指令运行过程不一。为避免出现这种情况，需要让 epc 存储的地址为延迟槽的上一条指令的地址，即 pc-4。这样在异常处理完后会返回到跳转指令所在的地址运行，进行是否跳转的判断。

延迟槽 flag 的判断在 id 级的 branchgen 文件下，这里只截取了其中一条判断实现，具体为根据给定的指令是否为跳转指令来判断：

```

`OP_BNE: begin
    if (reg_data_1 != reg_data_2) begin
        branch_flag <= 1;
        branch_addr <= addr_plus_4 + sign_ext_imm_sll2;
    end
    else begin
        branch_flag <= 0;
        branch_addr <= 0;
    end
    next_inst_delayslot_flag <= 1;
end

```

Pc0 中根据传递下来的指令是否为延迟槽指令的 flag，判断 exc 存储的地址为 pc 还是 pc-4

```
assign exc_epc = delayslot_flag ? current_pc_addr - 4 : current_pc_addr;
```

增加 next\_inst\_delayslot\_flag 信号后，需要在下一个周期再传输回 ID 阶段，用于告知 D 模块当前指令是延迟槽内的指令，因此在 ID 阶段加入 delayslot\_flag\_in 接口，用于接收来自 EX 阶段的 next\_inst\_delayslot\_flag 信号。

```
// delayslot flag
input          delayslot_flag_in
input          ft_adel_flag
```

## 扩展异常：

扩展异常主要需要实现在 ex 级的溢出判断和在 mem 级的地址异常判断，并在 mem 级对所有异常信息作统一的处理，判断是否要进入异常处理例程，并判断处理的异常类型。

Id 级新增一个无效指令的判断，具体即当判断给的指令不在 case 的项目中时，即认为该指令并没有被实现，为无效指令。这里截取了一部分无效指令 ri 的判断

```
always @(*) begin
  case (op)
    'OP_SPECIAL: begin
      case (funct)
        'FUNCT_SLL, 'FUNCT_SRL, 'FUNCT_SRA, 'FUNCT_SLLV,
        'FUNCT_SRLV, 'FUNCT_SRAV, 'FUNCT_JR, 'FUNCT_JALR,
        'FUNCT_ADD, 'FUNCT_SUB,
        'FUNCT_MFHI, 'FUNCT_MTHI, 'FUNCT_MFLO, 'FUNCT_MTLO,
        'FUNCT_MULT, 'FUNCT_MULTU, 'FUNCT_DIV, 'FUNCT_DIVU,
        'FUNCT_ADDU, 'FUNCT_SUBU, 'FUNCT_AND, 'FUNCT_OR,
        'FUNCT_XOR, 'FUNCT_NOR, 'FUNCT_SLT, 'FUNCT_SLTU,
        'FUNCT_SYSCALL, 'FUNCT_BREAK: begin
          ri_flag <= 0;
        end
        default: ri_flag <= 1;
      endcase
    end
    'OP_R: begin
      case (rt)
        'RT_RLZ, 'RT_RLZAL,
        'RT_RGEZ, 'RT_RGEZAL: begin
          ri_flag <= 0;
        end
        default: ri_flag <= 1;
      endcase
    end
    'OP_CPO: begin
      case (rs)

```

首先看 ex 级，ex 级根据运算数和运算结果为正负数来判断是否出现了运算溢出的情况，并将该 flag 给到 exc\_out 中作为异常信息存储起来。

```
wire ov_flag = ((!operand_1[31] && !operand_2_mux[31]) && result_sum[31])
|| ((operand_1[31] && operand_2_mux[31]) && (!result_sum[31]));
wire ov_exc = exc_in[`EXC_POS_OV] ? ov_flag : 0;
assign exc_out = {exc_in[7:3], ov_exc, exc_in[1:0]};
```

Mem 级则首先判断是否有地址的读写异常，分别为读地址异常 adel 和写地址异常 ades。Mem\_sel\_in 是一个存储了调用具体哪几位地址的寄存器。这两条 assign 实现了当读写最后一字节地址时，无需判断；读写末两字节地址时，判断地址是否为 2 的倍数；读写四位地址时，判断是否为 4 的倍数的功能。如此可以判断读写的地址是否出现异常。当判断地址异常时，flag 会被置为 1。

```

assign adel_flag = (mem_read_flag_in && (~((mem_sel_in == 4'b0000) ||
(mem_sel_in == 4'b0001) ||
((mem_sel_in == 4'b0011) && (address[1:0] == 2'b00 || address[1:0]
== 2'b10))) ||
((mem_sel_in == 4'b1111) && (address[1:0] == 2'b00)))) ? 1: 0;

assign ades_flag = (mem_write_flag_in && (~((mem_sel_in == 4'b0000) ||
(mem_sel_in == 4'b0001) ||
((mem_sel_in == 4'b0011) && (address[1:0] == 2'b00 || address[1:0]
== 2'b10))) ||
((mem_sel_in == 4'b1111) && (address[1:0] == 2'b00)))) ? 1: 0;

```

此外，pc 级还有一条 pc 地址异常的判断：

```

assign ft_adel_flag = pc[1:0] != 2'b00 ? 1: 0;

```

这些异常信息最后都会 mem 级被处理，根据发生的异常，进入对应的异常处理地址，进行异常的处理，并在处理完后返回到原有的 pc 地址。

```

always @(*) begin//异常
    if(int_occured && int_enabled && !stall_all) begin
        exc_out <= `EXC_INT;
    end
    else if (exc_in[`EXC_POS_OV]) begin
        exc_out <= `EXC_OV;
    end
    else if (exc_in[`EXC_POS_BP]) begin
        exc_out <= `EXC_BP;
    end
    else if (exc_in[`EXC_POS_SYS]) begin
        exc_out <= `EXC_SYS;
    end
    else if (exc_in[`EXC_POS_ERET]) begin
        exc_out <= `EXC_ERET;
    end
    else if (exc_in[`EXC_POS_ADE] || adel_flag) begin
        exc_out <= `EXC_ADEL;
    end
    else if (ades_flag) begin
        exc_out <= `EXC_ADES;
    end
    else if (exc_in[`EXC_POS_RI]) begin
        exc_out <= `EXC_RI;
    end
    else begin
        exc_out <= `EXC_NULL;
    end
end
end

```

判断异常发生后，流水线中的数据会被全部丢弃，等待异常处理完后重新运行各条指令。在访存阶段会将各异常相关信号传输给 PipelineController 模块，并通过 PipelineController 模块生成 exc\_pc 和 flush 信号，实现对流水线的控制。在 pipelinecontrol 中，首先会判断是否出现了异常。如若出现异常，则将 flush 置为 1

```

assign flush = stall_all ? 0 : (exc != `EXC_NULL) ? 1 : 0;

```

而 pipelinedeliver 检测到 flush 被置为 1 后，会对数据作一个清除处理，如此便实现了异常时的数据清空。此外，flush 也负责给 pc 一个跳转地址的信号。

```

always @(*) begin
    if (flush) begin
        next_pc <= exc_pc;
    end
    else if (!stall_pc) begin
        if (branch_flag) begin
            next_pc <= branch_addr;
        end
        else begin
            next_pc <= pc + 4;
        end
    end
    else begin

```

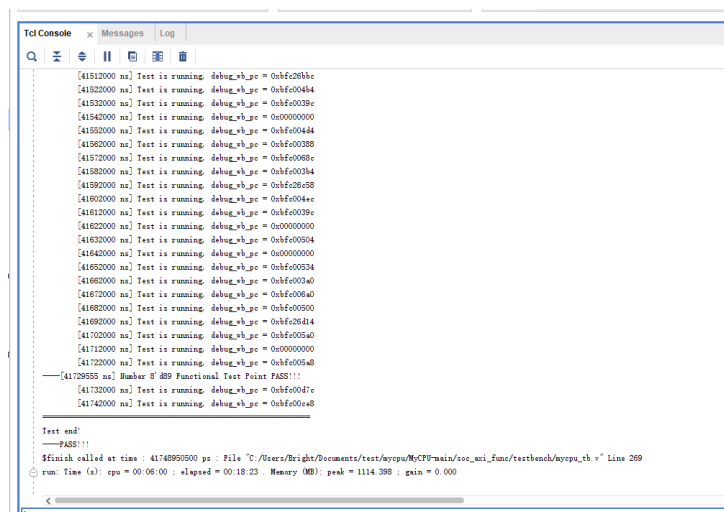
Pc 读取到 flush 后便将异常处理的地址给到下一地址，在下一跳跳转过去。

```

        else if (flush || !stall_pc) begin
            pc <= next_pc;
        end
    end
end

```

实验现象及分析：



成功通过了全 89 条指令的测试。

## 5 Cache 部分

### 设计过程

这部分只实现了 cache 的控制器模块，实际上还没有设计专门的 cache 寄存器模块。

首先定义 cache 行的格式和内存地址的格式：内存地址为 12 位，其中 [1:0] 是块内偏移，[6:2] 是索引，[11:7] 是 Tag。Cache 行为  $cache = V + D + Tag + Data = 1 + 1 + 5 + 128 = 135$  位。且 Cache 缓存中 cache 大小为 32 块，主存大小为 1024 块，1 块=4 字，1 字=32bit。

然后是 cache 控制状态机：

分别有 4 个状态，IDLE 为空闲状态，CompareTag 为通过 Tag 比较判断是否命中，执行两种情况的状态，Allocate 为没有命中时的情况从内存取出数据放到 cache 里面，然后更新修改位 D 和有效位 V。在 WriteBack 阶段执行将 cache 的修改行写入到内存中。

其中写回的时候，只有当没有命中且查询 cache 的那一行的修改位为 1 时，才将原本的 cache 行写入到内存里。然后去内存里面通过 cpu 发送过来的 tag 和 index 寻找 cache 行放入 cache 中，再次命中 cpu 即可取出。

测试方面，在这个部分中需要测试 6 种情况，分别是写未命中 (writemiss)，写命中 (writehit)，读命中 (readhit)，读未命中 (readmiss)，读写回 (readdirtymiss)，写写回 (writedirtymiss)。在 test.v 里面测试。

这里我们主要是使用 cache 第一行和第二行以及 cache 第一行对应于内存里面的第二块进行测试。且设置 cache 第二行第一个对应的内存里面存的是 1，其余内存里面都是 0，也就是 mem[4]=32'd1。首先是 writemiss 部分，此时 cache 行未命中，首先通过 cpu 那边给出的 tag 和 index 将内存里面的那一行数据都取出，然后放到 cache 行里，并将有效位设置为 1，修改位设置为 0，然后将 cpu 那边给出的数据写入对应 cache 行里面，并且将修改位设置为 1，先使用以下代码测试：

```
WriteMiss:begin
    cpu_req_rw<=1'b1;
    cpu_req_addr<=12'd0;
    cpu_data_write<=32'd8;
    cpu_req_valid<=1'b1;
end
```

cpu\_req\_rw 表示写有效，cpu\_req\_addr 表示 CPU 发过来的内存访问地址，cpu\_data\_write 表示 CPU 写入内存的数据，cpu\_req\_valid<=1'b1 表示从 cpu 发过来的这些数据和信号有效，即 cache 控制器中状态机开始从 IDLE 空闲状态转变为下一状态 comparetag。

然后是 writehit 部分：

```
WriteHit:begin
    cpu_req_rw<=1'b1;
    cpu_req_addr<=12'd0;
    cpu_data_write<=32'd9;
    cpu_req_valid<=1'b1;
End
```

接下来是 readmiss：

```
ReadMiss:begin
    cpu_req_rw<=1'b0;
    cpu_req_addr<=12'd4;
    cpu_data_write<=cpu_data_write;
    cpu_req_valid<=1'b1;
End
```

readhit 部分：

```
ReadHit:begin
    cpu_req_rw<=1'b0;
    cpu_req_addr<=12'd0;
    cpu_data_write<=cpu_data_write;
    cpu_req_valid<=1'b1;
End
```

writedirtymiss 部分：

```
WriteDirtyMiss:begin
    cpu_req_valid<=1'b1;
    cpu_req_rw<=1'b1;
```

```

        cpu_data_write<=32'd6;
        cpu_req_addr<=12'd128;
    end

```

readDirtyMiss 部分:

```

ReadDirtyMiss:begin
    cpu_req_valid<=1'b1;
    cpu_req_rw<=1'b0;
    cpu_req_addr<=12'd0;
    cpu_data_write<=cpu_data_write;
End

```

## 实验现象与分析

readmiss 的情况:

Cache 部分:

cpu_req_valid	0	Logic
> cpu_data_write[31:0]	00000006	Array
> cpu_data_read[31:0]	XXXXXXXX	Array
cpu_ready	0	Logic
> mem_req_addr[11:0]	000	Array
mem_req_rw	0	Logic
mem_req_valid	0	Logic
> mem_data_write[127:0]	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	Array
> mem_data_read[127:0]	00000000000000000000000000000000	Array
mem_ready	0	Logic
cache_data[0:31][134:0]	600000000000000000000000000000008,000000000000000000000000...	Array
> [0][134:0]	600000000000000000000000000000008	Array
> [1][134:0]	00000000000000000000000000000000	Array
> [2][134:0]	00000000000000000000000000000000	Array
> [3][134:0]	00000000000000000000000000000000	Array
> [4][134:0]	00000000000000000000000000000000	Array
> [5][134:0]	00000000000000000000000000000000	Array
> [6][134:0]	00000000000000000000000000000000	Array
> [7][134:0]	00000000000000000000000000000000	Array
> [8][134:0]	00000000000000000000000000000000	Array
> [9][134:0]	00000000000000000000000000000000	Array
> [10][134:0]	00000000000000000000000000000000	Array
> [11][134:0]	00000000000000000000000000000000	Array

Mem 的情况:

> mem_data_write[127:0]	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	Array
> mem_data_read[127:0]	00000000000000000000000000000000	Array
mem_ready	0	Logic
mem[0:4095][31:0]	00000000,00000000,00000000,00000000,00000001,00000000,00000...	Array
> [0][31:0]	00000000	Array
> [1][31:0]	00000000	Array
> [2][31:0]	00000000	Array
> [3][31:0]	00000000	Array
> [4][31:0]	00000001	Array
> [5][31:0]	00000000	Array
> [6][31:0]	00000000	Array
> [7][31:0]	00000000	Array
> [8][31:0]	00000000	Array
> [9][31:0]	00000000	Array
> [10][31:0]	00000000	Array
> [11][31:0]	00000000	Array

此时 cache 第一行中 6 位 0110，由于我们是 135 位 cache 行所以最高位 0 无效，第一位为 1 表示 V=1, 有效，第二位为 1 表示 D=1, 已修改，最后为 8 表示数据 8 以及写入，此时 cache 行第一个字对应与 mem[0]，此时并未写入。

然后是命中的情况:

Cache 部分:

>	cpu_req_addr[11:0]	080	Array
	cpu_req_rw	1	Logic
	cpu_req_valid	0	Logic
>	cpu_data_write[31:0]	00000006	Array
>	cpu_data_read[31:0]	XXXXXXXX	Array
	cpu_ready	0	Logic
>	mem_req_addr[11:0]	000	Array
	mem_req_rw	0	Logic
	mem_req_valid	0	Logic
>	mem_data_write[127:0]	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	Array
>	mem_data_read[127:0]	00000000000000000000000000000000	Array
	mem_ready	0	Logic
▼	cache_data[0:31][134:0]	60000000000000000000000000000009,00000...	Array
>	[0][134:0]	60000000000000000000000000000009	Array
>	[1][134:0]	00000000000000000000000000000000	Array
>	[2][134:0]	00000000000000000000000000000000	Array
>	[3][134:0]	00000000000000000000000000000000	Array
>	[4][134:0]	00000000000000000000000000000000	Array
>	[5][134:0]	00000000000000000000000000000000	Array
>	[6][134:0]	00000000000000000000000000000000	Array

Mem 部分：

>	mem_req_addr[11:0]	000	Array
	mem_req_rw	0	Logic
	mem_req_valid	0	Logic
>	mem_data_write[127:0]	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	Array
>	mem_data_read[127:0]	00000000000000000000000000000000	Array
	mem_ready	0	Logic
▼	mem[0:4095][31:0]	00000000,00000000,00000000,00000000,000000...	Array
>	[0][31:0]	00000000	Array
>	[1][31:0]	00000000	Array
>	[2][31:0]	00000000	Array
>	[3][31:0]	00000000	Array
>	[4][31:0]	00000001	Array
>	[5][31:0]	00000000	Array
>	[6][31:0]	00000000	Array
>	[7][31:0]	00000000	Array
>	[8][31:0]	00000000	Array
>	[9][31:0]	00000000	Array
>	[10][31:0]	00000000	Array
>	[11][31:0]	00000000	Array
>	[12][31:0]	00000000	Array

此时 cache 中的数据部分值修改为了 9，此时 V 和 D 均为 1，但是没有写回到 mem 中，因为此时 tag 是命中的。只有当不命中的时候且 cache 那一行是修改过的时候才写回。

Readmiss 的情况：

Cache 部分：

>	cpu_req_addr[11:0]	080	Array
	cpu_req_rw	1	Logic
	cpu_req_valid	0	Logic
>	cpu_data_write[31:0]	00000006	Array
>	cpu_data_read[31:0]	00000001	Array
	cpu_ready	0	Logic
>	mem_req_addr[11:0]	004	Array
	mem_req_rw	0	Logic
	mem_req_valid	0	Logic
>	mem_data_write[127:0]	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	Array
>	mem_data_read[127:0]	00000000000000000000000000000001	Array
	mem_ready	0	Logic
▼	cache_data[0:31][134:0]	60000000000000000000000000000009,400000...	Array
>	[0][134:0]	60000000000000000000000000000009	Array
>	[1][134:0]	40000000000000000000000000000001	Array
>	[2][134:0]	00000000000000000000000000000000	Array
>	[3][134:0]	00000000000000000000000000000000	Array
>	[4][134:0]	00000000000000000000000000000000	Array
>	[5][134:0]	00000000000000000000000000000000	Array
>	[6][134:0]	00000000000000000000000000000000	Array
>	[7][134:0]	00000000000000000000000000000000	Array

Mem 部分：



> mem_req_addr[11:0]	004	Array
mem_req_rw	0	Logic
mem_req_valid	0	Logic
> mem_data_write[127:0]	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	Array
> mem_data_read[127:0]	00000000000000000000000000000001	Array
mem_ready	0	Logic
mem[0:4095][31:0]	00000000,00000000,00000000,00000000,00000000...	Array
> [0][31:0]	00000000	Array
> [1][31:0]	00000000	Array
> [2][31:0]	00000000	Array
> [3][31:0]	00000000	Array
> [4][31:0]	00000001	Array
> [5][31:0]	00000000	Array
> [6][31:0]	00000000	Array
> [7][31:0]	00000000	Array
> [8][31:0]	00000000	Array
> [9][31:0]	00000000	Array
> [10][31:0]	00000000	Array
> [11][31:0]	00000000	Array

此时从 mem[4]那里将数据读取到了 cache 第二行，4 表示 0100，即有效位为 1，修改位为 0。

接下来是 readhit：

Cache 部分：

> cpu_req_addr[11:0]	080	Array
cpu_req_rw	1	Logic
cpu_req_valid	0	Logic
> cpu_data_write[31:0]	00000006	Array
> cpu_data_read[31:0]	00000009	Array
cpu_ready	0	Logic
> mem_req_addr[11:0]	004	Array
mem_req_rw	0	Logic
mem_req_valid	0	Logic
> mem_data_write[127:0]	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	Array
> mem_data_read[127:0]	00000000000000000000000000000001	Array
mem_ready	0	Logic
cache_data[0:31][134:0]	60000000000000000000000000000009,400...	Array
> [0][134:0]	60000000000000000000000000000009	Array
> [1][134:0]	40000000000000000000000000000001	Array
> [2][134:0]	00000000000000000000000000000000	Array
> [3][134:0]	00000000000000000000000000000000	Array
> [4][134:0]	00000000000000000000000000000000	Array
> [5][134:0]	00000000000000000000000000000000	Array
> [6][134:0]	00000000000000000000000000000000	Array
> [7][134:0]	00000000000000000000000000000000	Array

Mem 部分：

> mem_req_addr[11:0]	004	Array
mem_req_rw	0	Logic
mem_req_valid	0	Logic
> mem_data_write[127:0]	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	Array
> mem_data_read[127:0]	00000000000000000000000000000001	Array
mem_ready	0	Logic
mem[0:4095][31:0]	00000000,00000000,00000000,00000000,0000...	Array
> [0][31:0]	00000000	Array
> [1][31:0]	00000000	Array
> [2][31:0]	00000000	Array
> [3][31:0]	00000000	Array
> [4][31:0]	00000001	Array
> [5][31:0]	00000000	Array
> [6][31:0]	00000000	Array
> [7][31:0]	00000000	Array
> [8][31:0]	00000000	Array
> [9][31:0]	00000000	Array
> [10][31:0]	00000000	Array
> [11][31:0]	00000000	Array
> [12][31:0]	00000000	Array





















和上一次的结果一样，其实这里不太好看是否命中，不过可以从 cpu\_data\_read 那得到 cpu 接受的数据是 9，即命中。

writedirty 的情形：

Cache 部分:

>  cpu_req_addr[11:0]	080	Array
>  cpu_req_rw	1	Logic
>  cpu_req_valid	0	Logic
>  cpu_data_write[31:0]	00000006	Array
>  cpu_data_read[31:0]	00000009	Array
>  cpu_ready	0	Logic
>  mem_req_addr[11:0]	080	Array
>  mem_req_rw	0	Logic
>  mem_req_valid	0	Logic
>  mem_data_write[127:0]	00000000000000000000000000000009	Array
>  mem_data_read[127:0]	00000000000000000000000000000000	Array
>  mem_ready	0	Logic
>  cache_data[0:31][134:0]	61000000000000000000000000000006_4...	Array
>  [0][134:0]	61000000000000000000000000000006	Array
>  [1][134:0]	40000000000000000000000000000001	Array
>  [2][134:0]	00000000000000000000000000000000	Array
>  [3][134:0]	00000000000000000000000000000000	Array
>  [4][134:0]	00000000000000000000000000000000	Array
>  [5][134:0]	00000000000000000000000000000000	Array
>  [6][134:0]	00000000000000000000000000000000	Array
>  [7][134:0]	00000000000000000000000000000000	Array

Mem 部分:

>  mem_req_addr[11:0]	080	Array
 mem_req_rw	0	Logic
 mem_req_valid	0	Logic
>  mem_data_write[127:0]	000000000000000000000000000000009	Array
>  mem_data_read[127:0]	000000000000000000000000000000000	Array
 mem_ready	0	Logic
▼  mem[0.4095][31:0]	00000009,00000000,00000000,00000000,00...	Array
>  [0][31:0]	00000009	Array
>  [1][31:0]	00000000	Array
>  [2][31:0]	00000000	Array
>  [3][31:0]	00000000	Array
>  [4][31:0]	00000001	Array
>  [5][31:0]	00000000	Array
>  [6][31:0]	00000000	Array
>  [7][31:0]	00000000	Array
>  [8][31:0]	00000000	Array
>  [9][31:0]	00000000	Array
>  [10][31:0]	00000000	Array
>  [11][31:0]	00000000	Array
>  [12][31:0]	00000000	Array

此时 cache 第一行的数据写回了 mem 中，第一个字的数据为 9，且 cache 第一行的数据为 6，为新写入的数据，此时写入的 cache 第一行的第一个字的数据对应于 mem[128]。

最后是 `readdirtymiss` 部分:

Cache 部分:

[illegible]

Mem 部分：

>	mem_req_addr[11:0]	000	Array
	mem_req_rw	0	Logic
	mem_req_valid	0	Logic
>	mem_data_write[127:0]	00000000000000000000000000000006	Array
>	mem_data_read[127:0]	00000000000000000000000000000009	Array
	mem_ready	0	Logic
✓	mem[0:4095][31:0]	00000009,00000000,00000000,00000000,00...	Array
>	[0][31:0]	00000009	Array
>	[1][31:0]	00000000	Array
>	[2][31:0]	00000000	Array
>	[3][31:0]	00000000	Array
>	[4][31:0]	00000001	Array
>	[5][31:0]	00000000	Array
>	[6][31:0]	00000000	Array
>	[7][31:0]	00000000	Array
>	[8][31:0]	00000000	Array
>	[9][31:0]	00000000	Array
>	[10][31:0]	00000000	Array
>	[11][31:0]	00000000	Array
>	[12][31:0]	00000000	Array
>	[13][31:0]	00000000	Array
>	[14][31:0]	00000000	Array
>	[15][31:0]	00000000	Array
>	[16][31:0]	00000000	Array
>	[17][31:0]	00000000	Array
>	[18][31:0]	00000000	Array
>	[19][31:0]	00000000	Array
>	[20][31:0]	00000000	Array
>	[21][31:0]	00000000	Array
>	[22][31:0]	00000000	Array
>	[23][31:0]	00000000	Array
>	[24][31:0]	00000000	Array
>	[25][31:0]	00000000	Array
>	[26][31:0]	00000000	Array
>	[27][31:0]	00000000	Array
>	[28][31:0]	00000000	Array
>	[29][31:0]	00000000	Array
>	[30][31:0]	00000000	Array
>	[31][31:0]	00000000	Array
>	[32][31:0]	00000000	Array
>	[33][31:0]	00000000	Array
>	[34][31:0]	00000000	Array
>	[35][31:0]	00000000	Array
>	[36][31:0]	00000000	Array

首先此时 cache 第一行写的的数据对应于 mem[128], 由于我们此时读取的数据地址为 12'd0 即 mem[0]的数据, 且上一次即在 writedirtymiss 那写入过, 此时修改位为 1, 所以得需要将 cache 第一行写入到 mem[128:131], 再从内存里面取出 mem[0]=9, 然后将它放到 cache 第一行, 并将修改位设置为 1, 所以 cache 第一行此时的数据中 4 表示只有有效位为 1, 9 表示将 mem[0]取出。mem[128]=6 表示已经将数据写入到内存中。

5 组内成员主要工作及贡献比例

刘伟浩	负责实现并集成多周期华莱士树乘法器	20%
秦铭壕	负责实现并集成迭代除法器模块	20%
郝鑫	负责研究并实现 cache 模块, 完成相关测试	20%
罗奇峰	负责实现 CP0 协处理器及相关指令	20%
吴宇航	负责实现特权指令及扩展所有异常指令	20%

四：结论（讨论）

（对照课设内容，简要总结课设期间完成的主要工作：对照课设目的，着重说明通过课程设计过程所取得的收获和能力的达成情况；存在的问题及可能的改进方向；其它需要说明的问题）

题)

## 1、 结论 (实验总结)

刘伟浩：通过本次实验，我对于乘法器的实现有了比较大的认识并进行了实践。首先是了解了华莱士树乘法器的基本原理，并自己实现了一个多周期的华莱士树乘法。同时也探索出了控制 CPU 读取乘法器结果的实现方式。

郝鑫：通过本次计组设计实验，实现了 hilo 寄存器的相关指令以及其数据前递，深刻理解了数据前递，且通过实现 cache 控制部分以及测试仿真过程中遇到问题并解决，也加深了我对于 cache 的理解，也听了同组成员对于各自工作的汇总，虽然具体细节不是完全懂，但是大致的原理和流程是了解了，提高了我对于计算机运行机制的理解。

秦铭壕：由除法器的仿真分析以及 goldtrace 的测试可知，当前基于试商法的除法器实现成功。但是仍有较大的改进空间，例如当前除法器只能用于整数除法，并且只能得到整数商与余数的形式，更高级的除法器能够实现浮点数的除法，以后我们小组将尝试 Goldschmidt 方法、泰勒级数展开等更高级的算法实现更全面的除法功能

罗奇峰：协处理器部分完成了对多个状态寄存器 CP0 的定义书写，主要寄存器有 BadVaddr、Count、Status、Cause、EPC 等。在实现五个主要寄存器之后，完成对协处理器访问指令 mtc0 和 mfc0，以及为解决数据相关冲突而设计的数据前递部件 cp0ReadProxy，在此基础上开展对异常处理部分的拓展。

吴宇航：通过本次实验，我了解了异常判断和处理的具体流程，对异常类型有了清楚的认知，并对于不同类型的异常分别在哪判断、如何判断是否发生有了深刻的理解。此外，对于精确异常、异常发生后的流水线操作等内容也有了较深的感悟。在小组实验的过程中，我们也通过向其他小组成员阐述的方式来判断自己是否完全明白了自己负责的部分，而在这一过程中，我也对他人所负责的部分有了一些了解。总的来说，这次实验让我受益良多，对 cpu 的运行也有了一些自己的体悟。路漫漫其修远兮，我将继续努力学习相关知识，积极探索，加深对 cpu 的了解和理解。

## 2、讨论 (问题归纳，课程建议等)

刘伟浩：对于本课程，希望能够多在课堂上学习一些指示，而不是让助教用仅有的 4 节课的时间把计算机组成原理这一门庞大的课程的知识灌输给我们。（必修课计算机组成原理课程中讲到的内容实在太少，满足不了课设的需求）

郝鑫：本次实验在 cache 部分只完成了控制器部分，并未整合到工程中，存在以下问题：  
1. cache 得单独设立一个 cache 寄存器模块，用于进行数据和信号交换。  
2. 主存地址在项目中并不是 12 位，需要进行转换，否则无法通过 store 和 load 指令进行对 cache 的存取。  
3. cache 里面的那些从 cpu 发出的信号如主存地址还有数据等还需要与项目进行衔接和转换。只有 cache 在整个框架下运行指令并执行的情况才是完整的 cache。

吴宇航：异常部分的难点主要为内容多且杂，分布在各个不同的文件中，相互调用其他文件的 output。对于各个文件的端口与数据的传递，我仍有一些困惑的地方。不过，感谢于小组成员以及其他组同学的热心帮助，我能够完成这一部分的任务，回想起来仍感到不胜感激。

附：

TinyMIPS 实现的 MIPS 指令：

表 1-1 算术运算指令

指令助记格式	指令功能简述
ADDU rd,rs,rt	无符号加（无溢出异常）
ADDIU rt,rs,imm	无符号加立即数（无溢出异常）
SUBU rd,rs,rt	无符号减（无溢出异常）
SLT rd,rs,rt	有符号小于置 1
SLTU rd,rs,rt	无符号小于置 1

表 1-2 逻辑运算指令

指令助记格式	指令功能简述
AND rd,rs,rt	按位与
LUI rt,imm	寄存器高位置立即数
OR rd,rs,rt	按位或
XOR rd,rs,rt	按位异或

表 1-3 移位指令

指令助记格式	指令功能简述
SLL rd,rt,sa	立即数逻辑左移
SLLV rd,rs,rt	寄存器逻辑左移
SRAV rd,rs,rt	寄存器算术右移
SRLV rd,rt,sa	寄存器逻辑右移

表 1-4 分支跳转指令

指令助记格式	指令功能简述
BEQ rs,rt,offset	相等时分支转移
BNE rs,rt,offset	不等时分支转移
JAL target	无条件直接跳转，并保存返回地址
JALR rd,rs	无条件寄存器跳转，并保存返回地址

表 1-5 访存指令

指令助记格式	指令功能简述
LB rt,offset(base)	访存读字节（8 位），有符号扩展
LBU rt,offset(base)	访存读字节（8 位），无符号扩展
LW rt,offset(base)	访存读字（32 位）
SB rt,offset(base)	访存写字节（8 位）
SW rt,offset(base)	访存写字（32 位）

可选的 MIPS 指令见附件 A02：

## 北京科技大学实验报告

学院：专业：班级：

姓名：学号：实验日期：年 月 日

### 五、教师评审

教师评语	实验成绩
<p>（虽然课设主要侧重于验证问题，但是建议各位老师从解决“工程技术问题”，特别是“复杂工程问题”的角度去评审学生课设过程及代码阅读报告，主要包括提出问题、分析问题、解决问题及验证问题。 要有较详细的评审意见。）</p>	

签名:

日期: