

# 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：\_\_\_\_李晓坤\_\_\_\_

专    业：\_\_\_\_信息安全\_\_\_\_

班    级：\_\_\_\_信安 211\_\_\_\_

学    号：\_\_\_\_U202141863\_\_\_\_

指导教师：\_\_\_\_阿孜古丽\_\_\_\_

报告成绩：\_\_\_\_\_

实验地点：\_\_\_\_机电楼 301、304、320\_\_\_\_

实验时间：2023 年 9 月 18 日----2023 年 11 月 26 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：信息安全

班级：信安 211

姓名：李晓坤

学号：U202141863

实验日期：2023 年 10 月 20 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

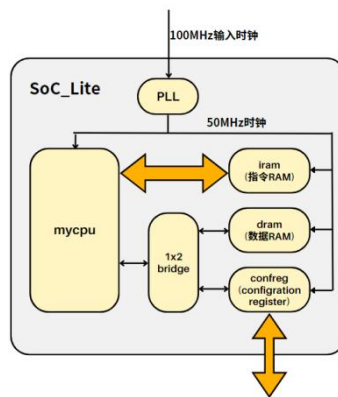
Software: Vivado2018.3 开发工具

VirtualBox 虚拟机+Ubuntu16.04.6（正文用五号宋体）

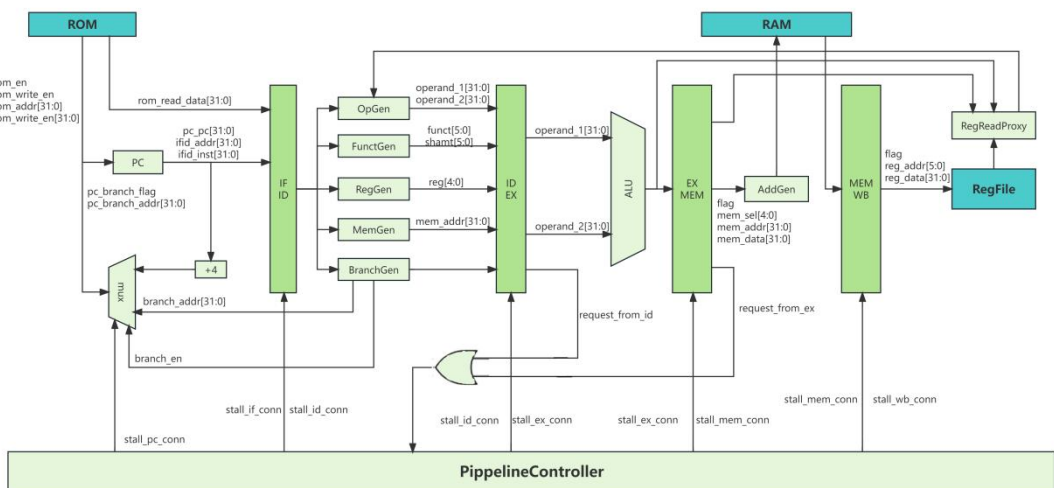
## 三、设计过程与结果分析

### 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)



总体结构框图



总体数据通路图

## 2 ADDIU 指令（典型指令）的设计过程

### 2.1 指令格式

31	26	25	21	20	16	15	0
001001	Rs				Rt		Imm

汇编风格：ADDIU rt, rs, imm

功能描述：ADDIU 是一个典型的 I 型指令，三个操作数分别是来自 rs 寄存器、rt 寄存器和指令中的立即数。其功能是将寄存器 rs 的值与有符号拓展至 32 位的立即数 imm 相加，结果写入 rt 寄存器中。

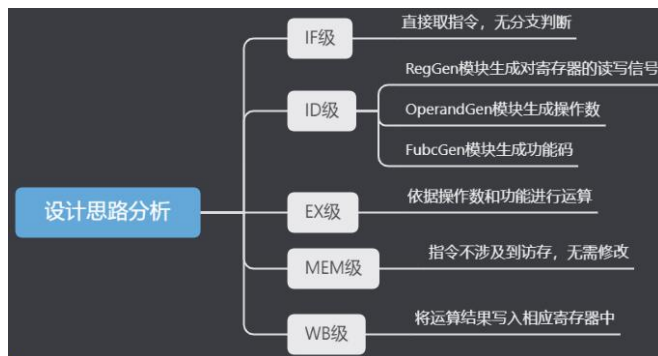
操作定义：GPR[rt] ← GPR[rs] + sign\_extend(imm)

### 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）

#### 2.2.1 设计分析

首先，ADDIU 是一条 I 型指令，因此需要在 opcode.v 头文件中添加相应的操作码宏定义；借由指令设计分析过程图可以看到，由于 ADDIU 指令没有涉及到访存、跳转等操作，因此需要在 FuncGen.v、OperandGen.v、RegGen.v 中添加相应的控制信息；在 ex 级，需要进行操作数相加运算；在 wb 级写回相应的寄存器中。

因此，可以画出相应的指令设计分析过程图如下：



ADDIU 指令的设计分析过程图

在 IF 级，直接取指令，由于不需要跳转，因此 PC 无需进行特殊判断；在 ID 级，需要根据指令内容生成操作数与功能码；在 EX 级，根据操作数和功能码进行相应运算；在 MEM

级由于指令不涉及访存，因此无需进行修改，将信号传递进入 WB 级；在 WB 级，将运算结果写入相应寄存器中。

### 2.2.2 实现代码

受限于报告篇幅，只展示代码中的关键部分，详细代码见工程文件。

#### (1) 操作码宏定义

```
`define OP_ADDIU    6'b001001//addiu
```

#### (2) 在 ID 级的 RegGen 模块，生成针对寄存器的读写信号

```
// arithmetic & logic (immediate)
`OP_ADDIU,
// memory accessing
`OP_LB, `OP_LW, `OP_LBU: begin
    reg_read_en_1 <= 1;//读 rs
    reg_read_en_2 <= 0;//另一个操作数来自立即数
    reg_addr_1 <= rs;//读 rs 寄存器的地址
    reg_addr_2 <= 0;//不需要读寄存器
end

// immediate
`OP_ADDIU, `OP_LUI: begin
    reg_write_en <= 1;//写入 rt 使能信号
    reg_write_addr <= rt;//写入 rt 寄存器的地址
end
```

#### (3) 在 ID 级的 FunctGen 模块，生成操作的功能码

```
case (op)
    `OP_SPECIAL: funct <= funct_in;//i 型指令
    `OP_LUI: funct <= `FUNCT_OR;//逻辑或
    `OP_LB, `OP_LBU, `OP_LW,
    `OP_SB, `OP_SW, `OP_ADDIU: funct <= `FUNCT_ADDU;//逻辑加
    `OP_JAL: funct <= `FUNCT_OR;
    default: funct <= `FUNCT_NOP;//6'b111111
endcase
```

#### (4) 在 ID 级的 OperandGen 模块，生成操作数

```
// immediate
`OP_ADDIU, `OP_LUI,
// memory accessing
`OP_LB, `OP_LW, `OP_LBU, `OP_SB, `OP_SW: begin
    operand_1 <= reg_data_1;//操作数 1 来自寄存器
end

// arithmetic & logic (immediate)
`OP_ADDIU,
// memory accessing
```

```

`OP_LB, `OP_LW, `OP_LBU, `OP_SB, `OP_SW: begin
    operand_2 <= sign_ext_imm; //操作数 2 来自符号拓展
end

```

#### (5) 在 EX 级，进行运算

```

case (funct)
    // jump with link & logic
    `FUNCT_JALR, `FUNCT_OR: result <= operand_1 | operand_2; //按位或
    `FUNCT_AND: result <= operand_1 & operand_2; //按位与
    `FUNCT_XOR: result <= operand_1 ^ operand_2; //按位异或
    // comparison
    `FUNCT_SLT, `FUNCT_SLTU: result <= {31'b0, operand_1 < operand_2};
    // arithmetic
    `FUNCT_ADDU, `FUNCT_SUBU: result <= result_sum; //加法和减法
    // shift 移位操作
    `FUNCT_SLL: result <= operand_2 << shamt;
    `FUNCT_SLLV: result <= operand_2 << operand_1[4:0];
    `FUNCT_SRLV: result <= operand_2 >> operand_1[4:0];
    `FUNCT_SRAV: result <= ({32{operand_2[31]}} << (6'd32 - {1'b0, operand_1[4:0]})) | operand_
2 >> operand_1[4:0];
    default: result <= 0;
endcase

```

#### (6) 在 WB 级的 RegFile 模块，将运算结果写回寄存器 rt 中

```

// writing //写数据操作同步时钟上升沿
always @(posedge clk) begin
    if (rst) begin
        for (i = 0; i < 32; i = i + 1) begin
            registers[i] <= 0;
        end
    end
    else if (write_en && |write_addr) begin //排除写地址为 0 号寄存器
        registers[write_addr] <= write_data;
    end
end

```

### 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)

Trace 比对是指首先使用已经实现全部指令的 CPU 来执行指令，按照时间进行采样并将采样结果整理为 golden\_trace 文件；然后使用我们自行设计的 CPU 执行相同的指令，也按照时间进行采样，并将采样结果与 golden\_trace 文件中的采样结果进行比对，若比对结果一致，则说明我们自行设计的 CPU 能够完整执行该指令，若比对结果不一致，则会发出提示，进一步修改 CPU。

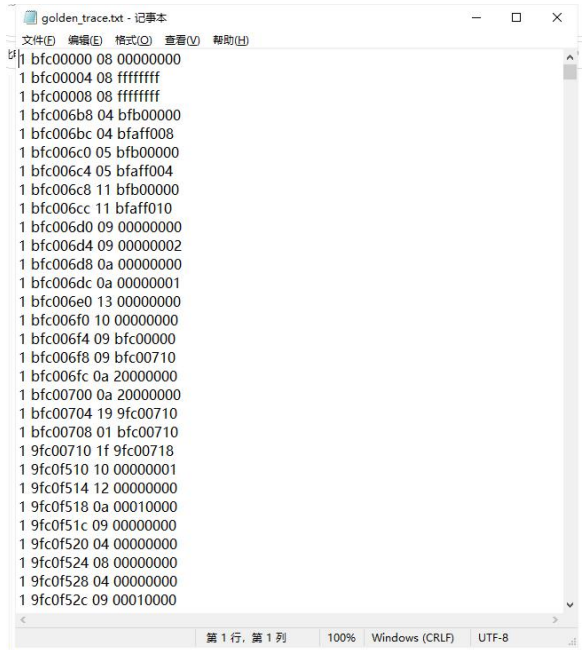
Trace 比对的实现过程具体如下：（1）修改 start.S 文件，将所要测试的指令写入其

中；（2）将 start.S 进行交叉编译，生成 object 文件夹；（3）使用 132 项目，进行仿真测试生成 golden\_trace 采样文件；（4）使用 axi 项目，进行仿真测试，将采样结果与 golden\_trace 比较，判断指令是否通过测试。

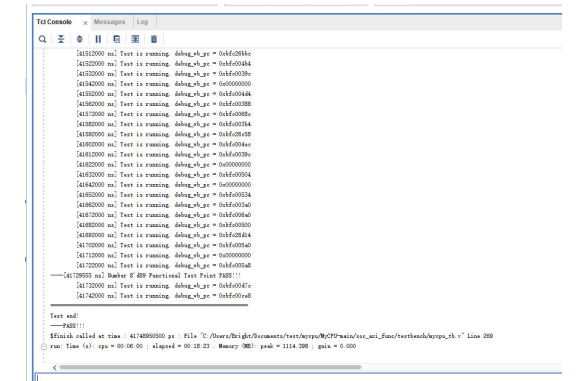
### 2.3.1 修改 start.S 文件

```
1 #include <asm.h>
2 #include <regdef.h>
3 #include <cpu_cmn.h>
4
5 #define TEST_NUM 36
6
7
8 #mch, number
9 #mch, number, address
10 #mch, exception use
11 #mch, score
12 #mch, exception pr
13
14 .set noreorder
15 .globl _start
16 .globl _main
17
18 _start:
19     li t0, 0xffffffff
20     addiu t0, zero, 0xffff
21     b locals
22     nop
23
24 #Mnemonic "j locals" not taken
25     lui t0, 0x8000
26     addiu t1, t1, 1
27     or t2, t0, zero
28     addu t3, t5, t6
29     lw t4, 0(t0)
30     nop
31
32 #Mnemonic rpu run error
33 .org 0x100
34     lui t0, 0x8000
35     addiu t1, t1, 1
36     or t2, t0, zero
37     addu t3, t5, t6
38     lw t4, 0(t0)
39     .org 0x100
40 _test_finish
```

### 2.3.2 生成 golden\_trace 文件



### 2.3.3 trace 比对查看结果



通过该 trace 比对结果图可以看到，所测试的指令成功通过了 trace 比对测试。

### 2.3.4 编写汇编代码

```
lui $8,0x1234
addiu $8,$8,0x5678
```

### 2.3.5 编写仿真激励

```
`timescale 1ns / 1ps
module cpu_tb();
  reg clk,rst;
  initial begin
    clk = 1;
    rst = 1;
    #7 rst = 0;
  end
  always #5
  begin
    clk = ~clk;
  end
  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;
  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;
  Core core(
    .clk(clk),
    .rst(rst),
    .stall(0),
    // ROM control
    .rom_en(rom_en),
    .rom_write_en(rom_write_en),
    .rom_addr(rom_addr),
    .rom_read_data(rom_read_data),
    .rom_write_data(rom_write_data),
    // RAM control
```

```

.ram_en(ram_en),
.ram_write_en(ram_write_en),
.ram_addr(ram_addr),
.ram_read_data(ram_read_data),
.ram_write_data(ram_write_data),
// debug signals
.debug_reg_write_en(debug_reg_write_en),
.debug_reg_write_addr(debug_reg_write_addr),
.debug_reg_write_data(debug_reg_write_data),
.debug_pc_addr(debug_pc_addr)
);
RAM ram(
.clk(clk),
.ram_en(ram_en),
.ram_write_en(ram_write_en),
.ram_addr(ram_addr),
.ram_write_data(ram_write_data),
.ram_read_data(ram_read_data)
);
ROM rom(
.clk(clk),//时钟信号
.rom_en(rom_en),//rom 使能信号
.rom_write_en(rom_write_en),//rom 写使能信号
.rom_addr(rom_addr),//rom 写地址
.rom_write_data(rom_write_data),//rom 写数据
.rom_read_data(rom_read_data)//rom 读数据
);
endmodule

```

### 2.3.6 仿真测试分析

执行指令 `lui $8, 0x1234` 的仿真结果如下。

debug_reg_write_en	1	
debug_reg_w..._addr[4:0]	01	01
debug_reg_...data[31:0]	12340000	12340000
debug_pc_addr[31:0]	c000f800	c000f800

执行指令 `addiu $8, $8, 0x5678` 的仿真结果如下。

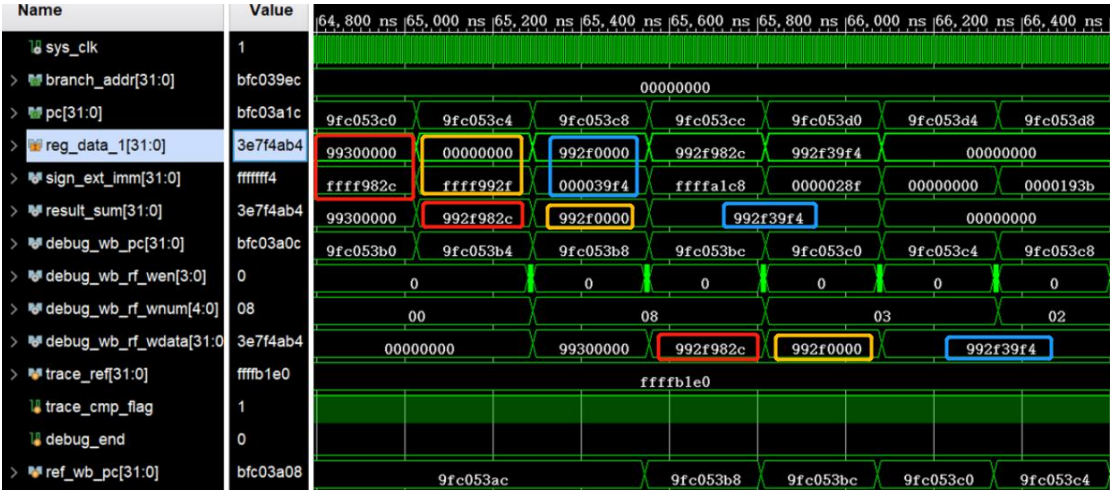
debug_reg_write_en	1	
debug_reg_w..._addr[4:0]	01	01
debug_reg_...data[31:0]	12345678	12345678
debug_pc_addr[31:0]	bfc00404	bfc00404

说明：（1）首先执行指令 `lui $8, 0x1234`，把 0x1234 写入 8 号寄存器高 16 位，看到



debug\_reg...data[31:0] 信号为 0x12340000，说明将该数写入寄存器；（2）执行指令 addiu \$8,\$8,0x5678，把 0x5678 和 8 号寄存器相加，结果写入 8 号寄存器。看到 debug\_reg...data[31:0] 信号为 0x12345678，说明将该数写入寄存器。

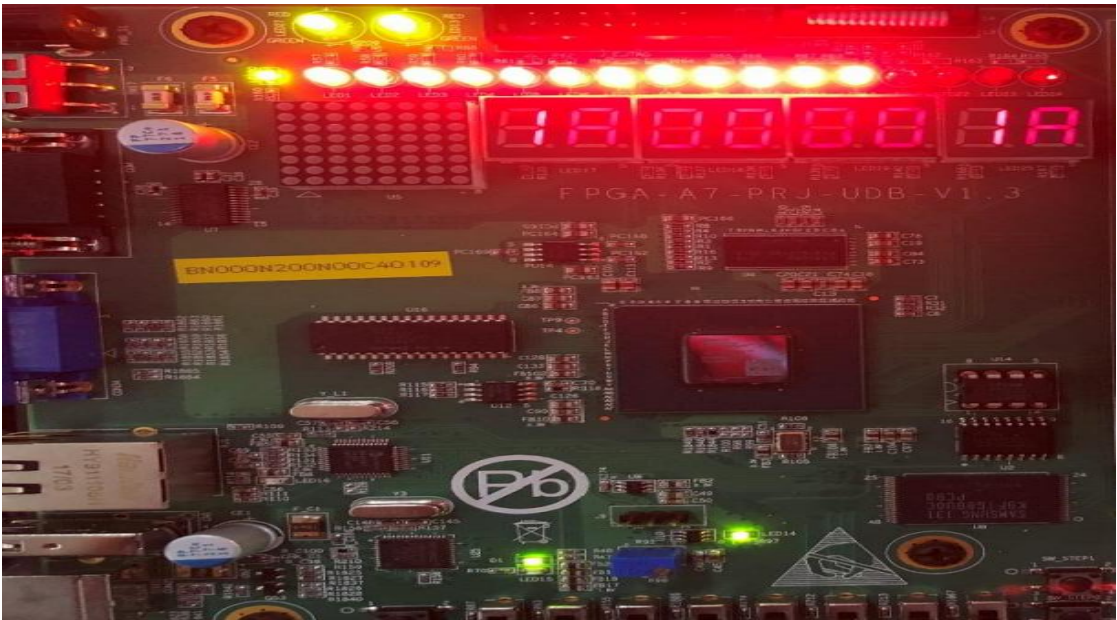
2.3.7 程序段测试



说明：由仿真可以看出，第 n 个 PC 的 Reg\_data\_1 和 sign\_ext\_imm，它们之和在第 n+1 个 PC 的 result\_sum[31:0]（EX 级）中体现，并且在第 n+3 个 pc 的时候被写回（WB 级）。

2.4 FPGA 上板验证过程（阐述对上板测试过程的理解）

在进行上板测试时，每通过一个测试点，数码管上的示数便会加 1，并且是以十六进制的形式进行呈现。将工程生成比特流后上板测试，测试结果如下图所示，数码管示数为 1A 即十进制数 26，说明成功通过 FPGA 测试点。



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“18+异常”条指令扩展，分别是：

slti,sltiu,j,add,addi,sub,nor,xori,sra,bgez,bgtz,blez,bltz,bltzal,bgezal,lh,lhu,sh,srl, 受限于报告篇幅,不能一一详细解释,因此三类指令各选一条详细介绍,其中 SLTI、BGEZ、SH 指令的设计实现过程如下:

1 指令格式

1.1 运算类: SLTI 指令

31	26 25	21 20	16 15	11 10	6 5	0
000000	rs	rt	rd	00000		101011

汇编风格: SLTI rt, rs, imm

功能描述: SLTI 是一条 I 型指令,三个操作数分别来自 rs 寄存器、rt 寄存器和指令中的立即数。其功能是将寄存器 rs 的值与有符号拓展至 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。

操作定义: if GPR[rs]<Sign\_extend(imm) then  
          GPR[rt]<-1  
          else  
          GPR[rt]<-0  
          endif

1.2 跳转类: BGEZ 指令

31	26 25	21 20	16 15	0
000001	rs	00001	offset	

汇编风格: BGEZ rs, offset

功能描述: 如果寄存器 rs 的值大于等于 0 则转移, 否则顺序执行。转移目标由立即数 offset 左移两位并进行有符号拓展的值加上该分支指令对应的延迟槽指令的 PC 计算得到。

操作定义: I: condition<-GPR[rs]>0  
          target\_offset<-sign\_extend(offset||0^2)  
          I+1:if condition then  
          PC<-PC+target\_offset  
          endif

1.3 访存类: SH 指令

31	26 25	21 20	16 15	0
101001	base	rt	offset	

汇编风格: SH rt, offset(base)

功能描述: 将 base 寄存器的值加上符号拓展后的立即数 offset 得到访存的虚地址, 如果地址不是 2 的整数倍则触发地址错例外, 否则据此虚地址将 rt 寄存器的半字。

操作定义: vAddr<-GPR[base]+sign\_extend(offset)  
          if vAddr0≠0 then  
          SignalExpection(AddressError)  
          endif  
          (pAddr, CCA)<-AddressTranslation(vAddr, DATA, STORE)  
          datahalf<-GPR[rt]15..0  
          StoreMemory(CCA, HALFWORD, datahalf, pAddr, vAddr, DATA)

## 2 分析指令功能及执行过程，画出数据通路图

### 2.1 运算类：SLTI 指令

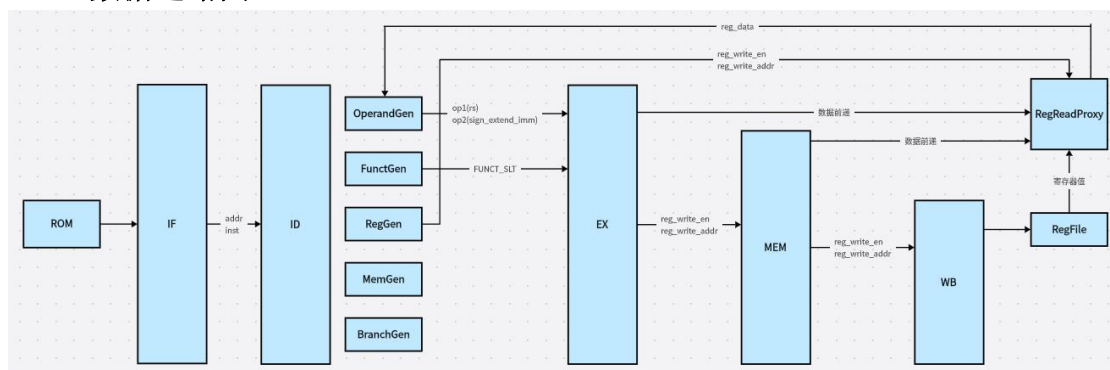
#### 2.1.1 指令功能

SLTI 是一条 I 型指令，三个操作数分别来自 rs 寄存器、rt 寄存器和指令中的立即数。其功能是将寄存器 rs 的值与有符号拓展至 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。

#### 2.1.2 指令执行过程

在取指级取出 SLTI 指令；在译码级生成相应的读写信号和操作数（rs 读使能，rt 写使能，rs 和立即数符号拓展为操作数，生成 funct 码）；在执行级将操作数的比较结果进行零拓展；在写回级将结果写入 rt 寄存器。

#### 2.1.3 数据通路图



### 2.2 跳转类：BGEZ 指令

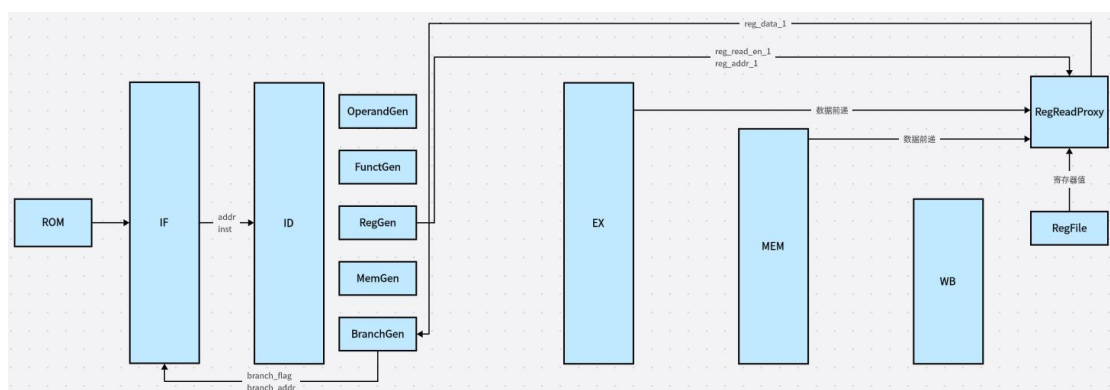
#### 2.2.1 指令功能

如果寄存器 rs 的值大于等于 0 则转移，否则顺序执行。转移目标由立即数 offset 左移两位并进行有符号拓展的值加上该分支指令对应的延迟槽指令的 PC 计算得到。

#### 2.2.2 指令执行过程

在取指级取出 BGEZ 指令；在译码级生成相应的读写信号（rs、rt 读使能，生成 funct 码，产生分支标志、分支地址）；根据分支标志和分支地址进行跳转。

#### 2.2.3 数据通路图



## 2.3 访存类：SH 指令

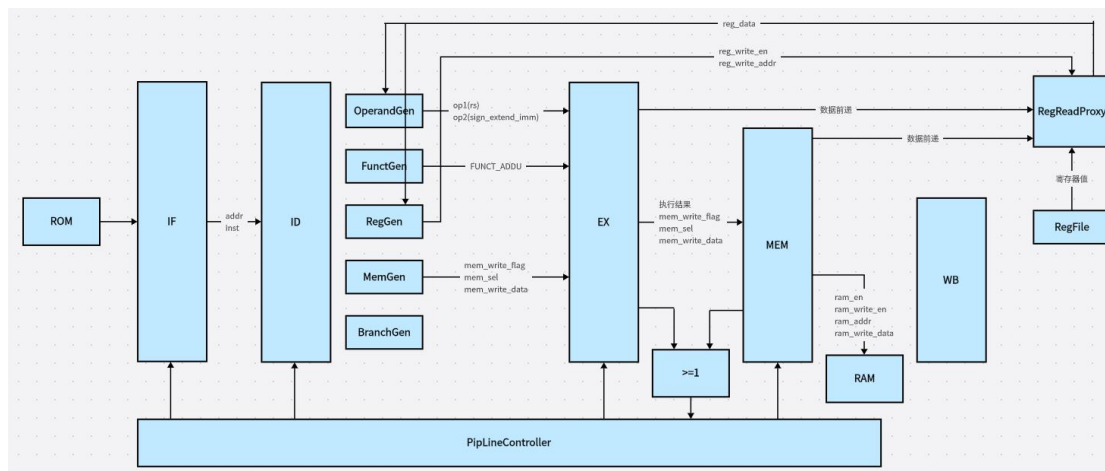
### 2.3.1 指令功能

将 base 寄存器的值加上符号拓展后的立即数 offset 得到访存的虚地址，如果地址不是 2 的整数倍则触发地址错例外，否则据此虚地址将 rt 寄存器的半字。

### 2.3.2 指令执行过程

在取指级取出指令；在译码级生成相应的读写信号（rs、rt 读使能，生成 funct 码，rs 和立即数符号拓展为操作数，内存写使能，mem\_sel 后两位为 1，rt 写入内存）；在执行级将操作数相加得到地址；在写回级将相应数据写入内存。

### 2.3.3 数据通路图



## 3 代码实现

受限于报告篇幅，在此部分只列出关键代码，详细代码见工程文件。

### 3.1 运算类：SLTI

(1) opcode.v

```
`define OP_SLTI    6'b001010
```

(2) functGen.v

```
case (op)
  `OP_SPECIAL: funct <= funct_in;
  `OP_ORI, `OP_LUI, `OP_JAL, `OP_J: funct <= `FUNCT_OR;
  `OP_LB, `OP_LBU, `OP_LW, `OP_LH, `OP_B, `OP_LHU,
  `OP_SH, `OP_ADDI,
  `OP_SB, `OP_SW, `OP_ADDIU: funct <= `FUNCT_ADDU;
  `OP_ANDI: funct <= `FUNCT_AND;
  `OP_XORI: funct <= `FUNCT_XOR;
  `OP_SLTIU: funct <= `FUNCT_SLTU;
  `OP_SLTI: funct <= `FUNCT_SLT;
  default: funct <= `FUNCT_NOP;
endcase
```

### (3) RegGen. v

```
`OP_ADDI, `OP_BLEZ,
`OP_SLTI, `OP_SLTIU, `OP_BGTZ,
`OP_XORI, `OP_LH, `OP_LHU,
// arithmetic & logic (immediate)
`OP_ADDIU, `OP_ORI, `OP_ANDI,
// memory accessing
`OP_LB, `OP_LW, `OP_LBU: begin
    reg_read_en_1 <= 1;
    reg_read_en_2 <= 0;
    reg_addr_1 <= rs;
    reg_addr_2 <= 0;
end

`OP_ADDI,
`OP_XORI, `OP_SLTI, `OP_SLTIU,
// immediate
`OP_ADDIU, `OP_LUI, `OP_ORI, `OP_ANDI: begin
    reg_write_en <= 1;
    reg_write_addr <= rt;
end
```

### (4) operandGen. v

```
`OP_ADDI,
`OP_SLTI, `OP_SLTIU,
`OP_XORI, `OP_LH, `OP_LHU, `OP_SH,
// immediate
`OP_ADDIU, `OP_LUI, `OP_ANDI,
// memory accessing
`OP_LB, `OP_LW, `OP_LBU, `OP_SB, `OP_SW, `OP_ORI: begin
    operand_1 <= reg_data_1;
end

`OP_SLTI, `OP_ADDI, `OP_LHU, `OP_SH,
// arithmetic & logic (immediate)
`OP_ADDIU, `OP_LH, `OP_SLTIU,
// memory accessing
`OP_LB, `OP_LW, `OP_LBU, `OP_SB, `OP_SW: begin
    operand_2 <= sign_ext_imm;
end
```

### (5) EX. v

```
// 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);

// comparison
`FUNCT_SLT, `FUNCT_SLTU: result <= {31'b0, operand_1_lt_operand_2};

```

### 3.2 跳转类: BGEZ

(1) opcode.v

```

`define OP_BLEZ    6'b000110

```

(2) RegGen.v

```

case(rt)
  `RT_BLTZ, `RT_BLTZAL,
  `RT_BGEZ, `RT_BGEZAL: begin
    reg_read_en_1 <= 1;
    reg_read_en_2 <= 0;
    reg_addr_1 <= rs;
    reg_addr_2 <= 0;
  end
end

```

(3) BranchGen.v

```

`OP_BGTZ: begin
  if(!reg_data_1[31] && reg_data_1)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

`RT_BGEZ, `RT_BGEZAL: begin
  if(!reg_data_1[31]) begin
    branch_flag <= 1;
    branch_addr <= addr_plus_4 + sign_ext_imm_sll2;
  end
  else begin
    branch_addr <= 0;
    branch_flag <= 0;
  end
  next_inst_delayslot_flag <= 1;
end

```

```
end
```

### 3.3 访存类: SH

(1) opcode. v

```
`define OP_SH      6'b101001
```

(2) FunctGen. v

```
case (op)
  `OP_SPECIAL: funct <= funct_in;
  `OP_ORI, `OP_LUI, `OP_JAL, `OP_J: funct <= `FUNCT_OR;
  `OP_LB, `OP_LBU, `OP_LW, `OP_LH, `OP_B, `OP_LHU,
  `OP_SH, `OP_ADDI,
  `OP_SB, `OP_SW, `OP_ADDIU: funct <= `FUNCT_ADDU;
  `OP_ANDI: funct <= `FUNCT_AND;
  `OP_XORI: funct <= `FUNCT_XOR;
  `OP_SLTIU: funct <= `FUNCT_SLTU;
  `OP_SLTI: funct <= `FUNCT_SLT;
  default: funct <= `FUNCT_NOP;
endcase
```

(3) RegGen. v

```
// memory accessing
`OP_SB, `OP_SW, `OP_SH,
// r-type
`OP_SPECIAL: begin
  reg_read_en_1 <= 1;
  reg_read_en_2 <= 1;
  reg_addr_1 <= rs;
  reg_addr_2 <= rt;
end
```

(4) OperandGen. v

```
`OP_ADDI,
`OP_SLTI, `OP_SLTIU,
`OP_XORI, `OP_LH, `OP_LHU, `OP_SH,
// immediate
`OP_ADDIU, `OP_LUI, `OP_ANDI,
// memory accessing
`OP_LB, `OP_LW, `OP_LBU, `OP_SB, `OP_SW, `OP_ORI: begin
  operand_1 <= reg_data_1;
end
`OP_SLTI, `OP_ADDI, `OP_LHU, `OP_SH,
// arithmetic & logic (immediate)
`OP_ADDIU, `OP_LH, `OP_SLTIU,
```

```
// memory accessing
`OP_LB, `OP_LW, `OP_LBU, `OP_SB, `OP_SW: begin
    operand_2 <= sign_ext_imm;
end
```

#### (5) MemGen. v

```
case (op)
    `OP_SH, `OP_SB, `OP_SW: mem_write_flag <= 1;
    default: mem_write_flag <= 0;
endcase
```

```
case (op)
    `OP_LB, `OP_LBU, `OP_SB: mem_sel <= 4'b0001;
    `OP_SH, `OP_LHU, `OP_LH: mem_sel <= 4'b0011;
    `OP_LW, `OP_SW: mem_sel <= 4'b1111;
    default: mem_sel <= 4'b0000;
endcase
```

```
case (op)
    `OP_SH, `OP_SB, `OP_SW: mem_write_data <= reg_data_2;
    default: mem_write_data <= 0;
endcase
```

#### (6) MEM. v

```
else if (mem_sel_in == 4'b0011) begin
    case (address[1:0])
        2'b00: ram_write_sel <= 4'b0011;
        2'b10: ram_write_sel <= 4'b1100;
        default: ram_write_sel <= 4'b0000;
    endcase
end
```

```
else if (mem_sel_in == 4'b0011) begin
    case (address[1:0])
        2'b00: ram_write_data <= mem_write_data;
        2'b10: ram_write_data <= mem_write_data << 16;
        default: ram_write_data <= 0;
    endcase
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。）

### 4.1 Trace 比对过程分析

Trace 比对是指首先使用已经实现全部指令的 CPU 来执行指令，按照时间进行采样并将采样结果整理为 golden\_trace 文件；然后使用我们自行设计的 CPU 执行相同的指令，也按照时间进行采样，并将采样结果与 golden\_trace 文件中的采样结果进行比对，若比对结果一致，则说明我们自行设计的 CPU 能够完整执行该指令，若比对结果不一致，则会发出提示，





通过该 trace 比对结果图可以看到，所测试的指令成功通过了 trace 比对测试，即所拓展的指令全部通过测试。

## 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。）

### 5.1 编写汇编代码

#### 5.1.1 SLTI

```
addiu $10,$0,0x0008
slti $11,$10,0x0005
slti $11,$10,0x000b
addiu $12,$0,0x0009
slti $11,$12,0x8000
```

#### 5.1.2 BGEZ

```
addiu $10,$0,0x0001
bgez $10,flag_1
nop
addiu $10,$0,0x0008
flag_1: addiu $10,$0,0x0003
```

#### 5.1.3 SH

```
addiu $10,$0,0x000a
addiu $11,$0,0x000a
sh $11,0x1000($10)
lh $12,0x1000($10)
```

### 5.2 编写仿真激励

对不同的指令进行测试时，需要编写不同的汇编代码，但是使用相同的 top\_tb 文件，因此，编写统一的 top\_tb 文件内容如下。

```
`timescale 1ns / 1ps
module cpu_tb();
    reg clk,rst;
    initial begin
        clk = 1;
        rst = 1;
        #7 rst = 0;
    end
    always #5
    begin
        clk = ~clk;
    end
    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;
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;
Core core(
    .clk(clk),
    .rst(rst),
    .stall(0),
    // ROM control
    .rom_en(rom_en),
    .rom_write_en(rom_write_en),
    .rom_addr(ram_addr),
    .rom_read_data(rom_read_data),
    .rom_write_data(rom_write_data),
    // RAM control
    .ram_en(ram_en),
    .ram_write_en(ram_write_en),
    .ram_addr(ram_addr),
    .ram_read_data(ram_read_data),
    .ram_write_data(ram_write_data),
    // debug signals
    .debug_reg_write_en(debug_reg_write_en),
    .debug_reg_write_addr(debug_reg_write_addr),
    .debug_reg_write_data(debug_reg_write_data),
    .debug_pc_addr(debug_pc_addr)
);
RAM ram(
    .clk(clk),
    .ram_en(ram_en),
    .ram_write_en(ram_write_en),
    .ram_addr(ram_addr),
    .ram_write_data(ram_write_data),
    .ram_read_data(ram_read_data)
);
ROM rom(
    .clk(clk), //时钟信号

```

```

.rom_en(rom_en),//rom 使能信号
.rom_write_en(rom_write_en),//rom 写使能信号
.rom_addr(rom_addr),//rom 写地址
.rom_write_data(rom_write_data),//rom 写数据
.rom_read_data(rom_read_data)//rom 读数据
);
endmodule

```

## 5.3 仿真测试及分析

### 5.3.1 SLTI



分析仿真测试结果,可以得到如下结论:(1)立即数8写入\$10;(2)slti \$11,\$10,0x0005,\$10中的数值大于立即数5,因此0写入\$11;(3)slti \$11,\$10,0x000a,\$10中的数值小于立即数b,因此1写入\$11;(4)向\$12写入9;(5)slti \$11,\$12,0x8000,其中0x8000的符号拓展是负数,\$12中的数值9大于负数,因此0写入\$12。仿真结果符合预期。

### 5.3.2 BGEZ



pc=bfc0008对应执行延迟槽中的nop指令。因为\$10的数值大于0,所以执行之前的bgez,跳转到pc=bfc0014即flag\_1处。仿真结果符合预期。

### 5.3.3 SH



pc=0xbfc0000,向\$10写入0x000a。pc=0xbfc0004,向\$11写入0x000a。pc=0xbfc0008,将\$11的内容(值a)写入0x100a地址的内存。pc=0xbfc000c,将0x100a地址的内存读入\$11,数值确实为a。仿真结果符合预期。

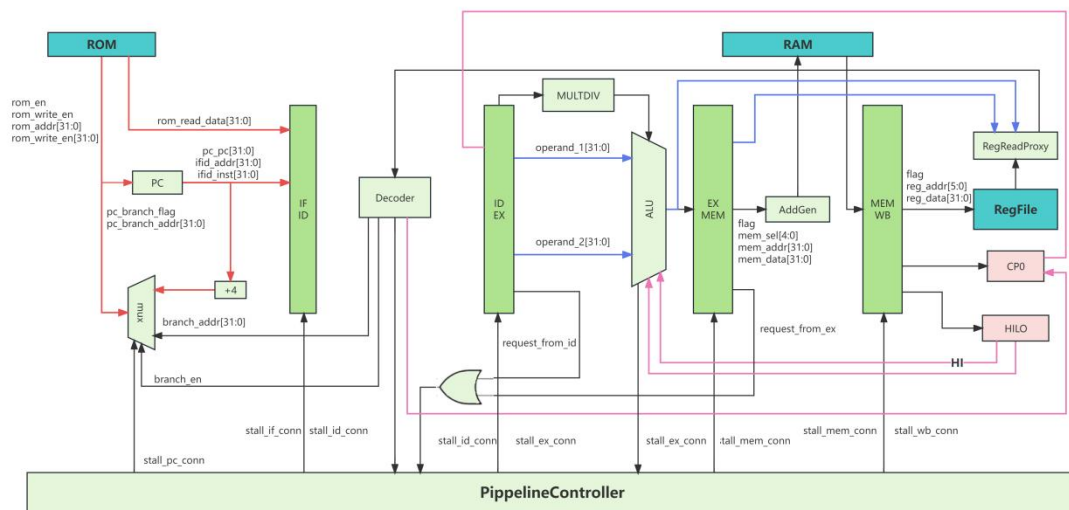
## Part 3 创新实验（演示系统介绍）实验分组编号：50（填写实验箱分组中组号）

**系统概述：**实现了什么功能，扩展了哪些内容，如何测试，有何发现。

我们小组的创新部分实现了HILO寄存器及相关数据转移指令、CP0协处理器及相关数据转移指令、基于华莱士树的多周期乘法器、迭代除法器、异常处理、一级直接映射Cache。

在测试方面,我们对于乘除法器 and Cache 部分首先编写单独的testbench模拟这些部件在工作过程中的信号情况,根据相关的输出信号判断该部件功能是否完善;进一步,将该部件整合进入CPU中,通过标准trace比对的方法来判断功能是否完善。

## 1 系统总体架构图



## 2 设计过程与实验现象及分析

### 2.1 HILO 寄存器

#### 2.1.1 设计过程

(1) 首先创建一个 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。

```

`timescale 1ns / 1ps
`include "../include/bus.v"
//维护两个寄存器，用于放置乘除法运算的结果
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;
        end
    end
end module
    
```

```

        lo <= lo_i;
    end
end
endmodule // HILO

```

(2) 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;

```

(3) 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;

```

(4) 创建数据前递的模块 hiloreadproxy：

```

`timescale 1ns / 1ps
`include "../include/bus.v"
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

```

(5) 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   )
);
```

(6) 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。

### 2.1.2 实验现象及分析

(1) 测试 MTHI, MFHI, MTLO, MFLO 测试点, 修改 start.S 文件内容。

```
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
```

(2) 交叉编译

```
loongson@loongson-VirtualBox: /media/sf_soft/func
start.S:288: Warning: used $at without ".set noat"
start.S:288: Warning: used $at without ".set noat"
start.S:296: Warning: used $at without ".set noat"
start.S:296: Warning: used $at without ".set noat"
start.S:390: Warning: used $at without ".set noat"
start.S:390: Warning: used $at without ".set noat"
mipsel-linux-gcc -E -P -Umps -D_LOADER -U_MAIN -D_KERNEL -fno-builtin -mips1 -D
MEMSTART=0x80000000 -DMEMSIZE=0x04000 -DCPU_COUNT_PER_US=1000 -I /media/sf_soft/
func/include -fno-reorder-blocks -fno-reorder-functions bin.lds.S -o bin.lds
mipsel-linux-ld -g -T bin.lds -o main.elf start.o -L . -lnst
mipsel-linux-objdump -aD main.elf > test.s
mipsel-linux-objcopy -O binary -j .text main.elf main.bin
mipsel-linux-objcopy -O binary -j .data main.elf main.data
./convert
mkdir -p ./obj
mv main.elf ./obj/
mv test.s ./obj/
mv main.bin ./obj/
mv main.data ./obj/
mv *.coe ./obj/
mv *.mif ./obj/
cp ./obj/inst_ram.mif ./obj/axi_ram.mif
make[1]: Leaving directory '/media/sf_soft/func'
loongson@loongson-VirtualBox: /media/sf_soft/func$
```

(3) 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/cpul32_gettrace/testbench/tb_top.v" Line 221

```

#### (4) 最终测试结果

```

      [ 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/zoc_axi_func/testbench/mycpu_tb.v" Line 269

```

## 2.2 CP0 协处理器

### 2.2.1 设计过程

#### (一) 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，作用如下表。

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，CPI、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 CPI、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。



## （二）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 开始重新计数。Count 寄存器可读、可写。

```
// COUNT 处理内部计数器
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
```

### （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 处理器状态和控制寄存器
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]; //8-15 位对应 im0-im7, 1 使能 0 屏蔽
        reg_status[1:0] <= cp0_write_data[1:0];
    end
    else begin
        reg_cause <= reg_cause;
    end
end
```

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 七 0001。

- 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

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。

```
// CAUSE 记录上次例外原因
always @(posedge clk) begin//8-15 位对应 ip0-ip7//15-10 对应的 ip7-ip2 是硬中断标识
    if (rst) begin //9-8 ip1-ip0 是软中断标识 BD 指的是最高位，是否在延迟早中
        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
end
```

```

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
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 上次发生例外的 pc
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 == `CPO_REG_EPC) begin
    reg_epc <= cp0_write_data;
end
else begin
    reg_epc <= reg_epc;
end
end
```

(5) BADVADDR 寄存器

```
// BADVADDR 最新地址相关例外的出错地址
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
```

(三) 协处理器 CP0 中的寄存器

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

(四) 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。指令格式如下。

```
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
    default: begin
      cp0_write_en <= 0;
      cp0_read_en <= 0;
      cp0_write_data <= 0;
      cp0_addr <= 0;
    end
  endcase
end
```



```

reg_write_en <= 0;
reg_write_addr <= 0;
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,
// -----

// 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 #( `CP0_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          [ `CP0_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         [ `CP0_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          [ `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 #(`CP0_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：

```

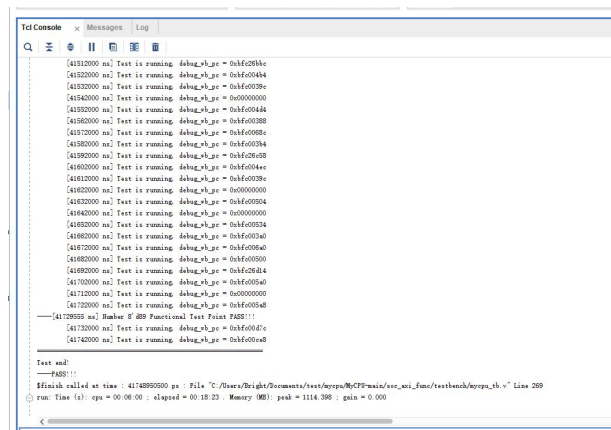
module CPOGen (
    input    [`INST_BUS]    inst,
    input    [`INST_OP_BUS] op,
    input    [`REG_ADDR_BUS] rs,
    input    [`REG_ADDR_BUS] rd,
    input    [`DATA_BUS]    reg_data_1,

    output reg    cp0_write_en,
    output reg    cp0_read_en,
    output reg [`CP0_ADDR_BUS] cp0_addr,
    output reg [`DATA_BUS]    cp0_write_data
);

```

## 2.2.2 实验现象及分析

由于 CP0 协处理器涉及到的指令数量较多，因此将所有指令集中，统一进行 trace 比对，通过了 89 个 trace 比对点，说明关于 CP0 协处理器的指令均已经成功实现。



## 2.3 乘法器

### 2.3.1 设计过程

乘法器的设计上，我们计划实现的是一个在 7 个时钟上升沿后得到计算结果的华莱士树乘法器。其设计的核心思想是将乘法转换为多个数的加法。

首先我们需要得到这些相加的数，也就是部分积。该部分通过 booth 编码解决，基于以



下公式，我们可以将原本的 32 个部分积转换为仅 17 个部分积。

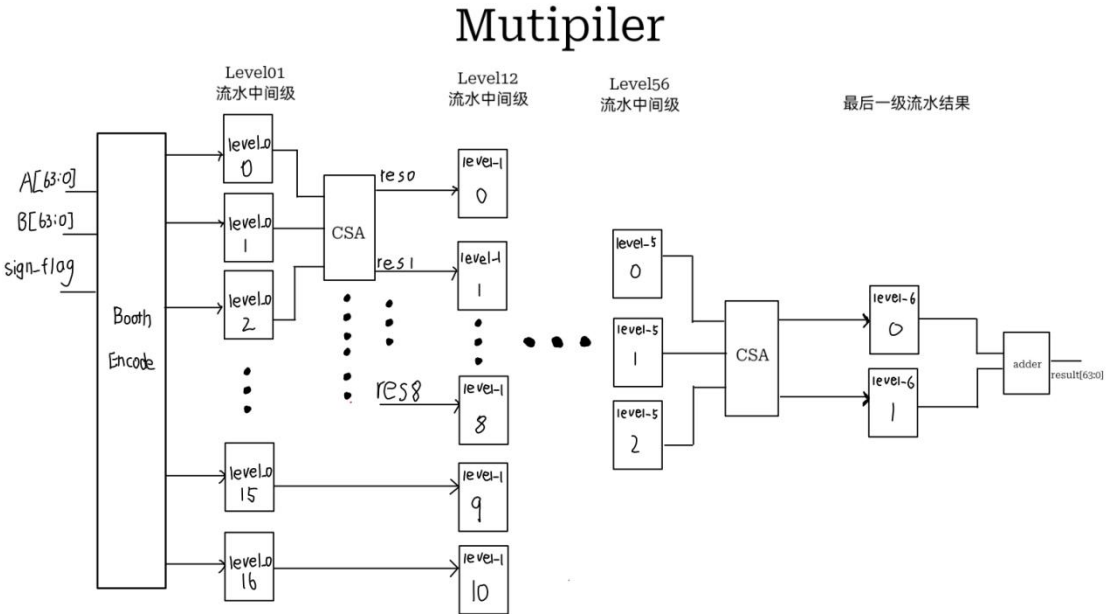
$$Y = -2^{n-1}y_{n-1} + 2^{n-2}y_{n-2} + \dots + y_0 + y_{-1}, \quad y_{-1} = 0$$

$$Y = (-y_{n-1} + y_{n-2}) \cdot 2^{n-1} + (-y_{n-2} + y_{n-3}) \cdot 2^{n-2} + \dots + (-y_0 + y_{-1}) \cdot 2^0$$

$$Y = (-2y_{n-1} + y_{n-2} + y_{n-3}) \cdot 2^{n-2} + (-2y_{n-3} + y_{n-4} + y_{n-5} \cdot 2^{n-4}) + \dots + (-2y_1 + y_0 + y_{-1}) \cdot 2^0$$

我们的加法用到的并不是普通的加法器，而是进位保留加法器，这也是华莱士树乘法器的核心部件。他可以将三个数相加转换为两个数相加，换言之其是一个有三个输入，两个输出的部件，可以将相加的数的个数规模减小。据此我们可以通过一级级的进位保留加法器将要相加的数的个数不断减少，直至只有两个加数，最后再通过一个普通的 64 位全加法器输出最后的结果。

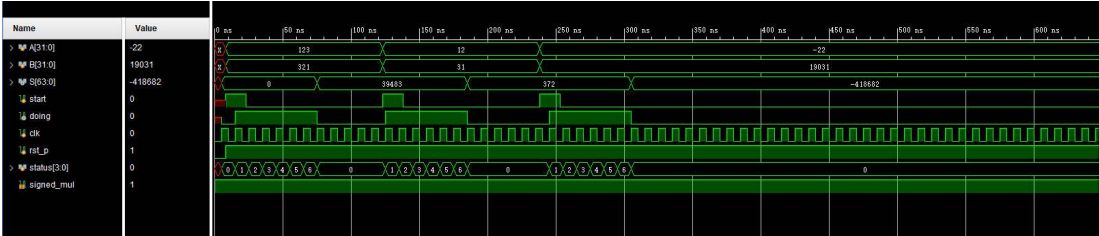
该部分我们采用了类似流水线的结构以提高效率，具体结构图如下。



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。

### 2.3.2 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。



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。

## 2.4 除法器

### 2.4.1 设计过程

#### (1) 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）。

#### (2) 程序设计

首先是模块的输入输出说明：

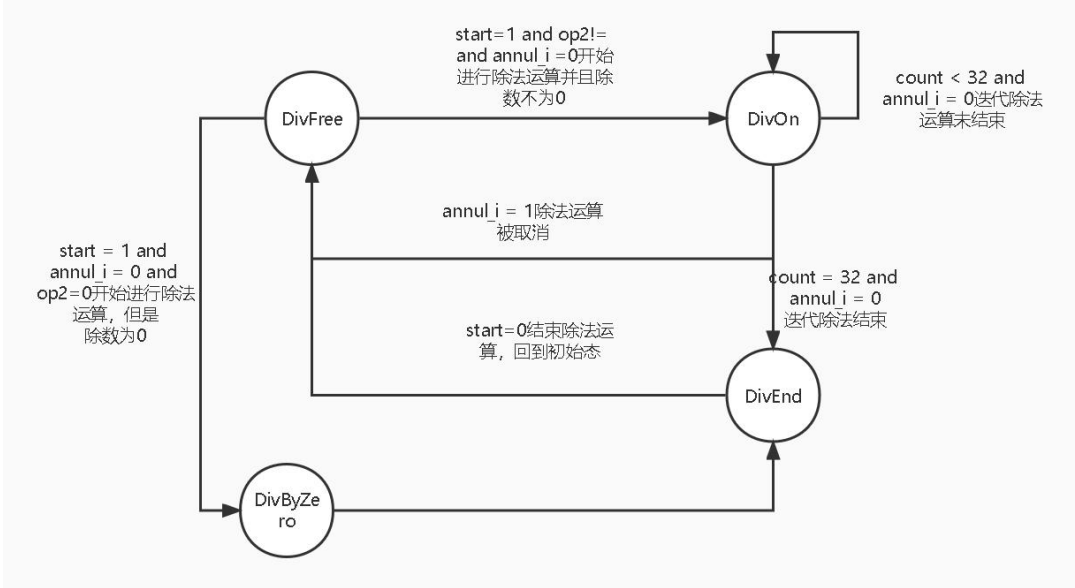
序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位
2	clk	1	输入	时钟
3	signed_div	1	输入	为 1 代表有符号
4	op1	32	输入	被除数
5	op2	32	输入	除数
6	start	1	输入	是否开始运算
7	annul	1	输入	是否取消运算
8	result_o	64	输出	除法结果，高 32 位为余数，低 32 为商
9	ready_o	1	输出	除法是否结束

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：（1）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。（2）然后是 DivByZero 状态，它代表除数为 0，在这个状态中，会将 next\_state 置为 DivEnd。（3）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。（4）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。

（3）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。

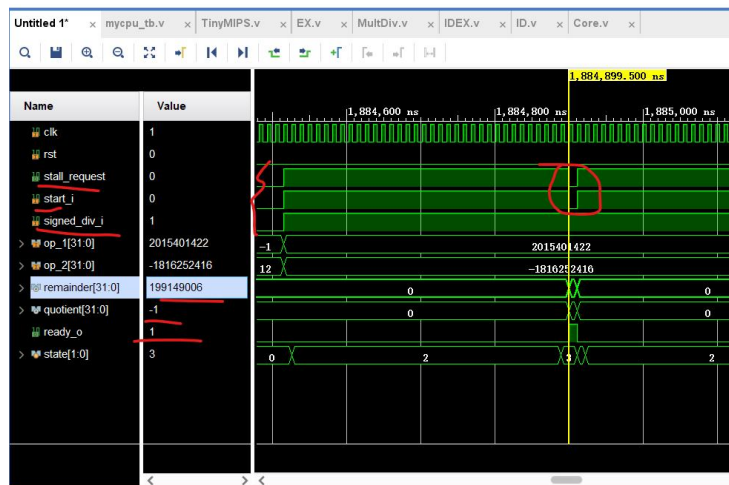
第三部分主要涉及 annul 信号，当某处发生异常时，需要将当前流水线执行的指令全部清空，所以此时 annul 的值会变为 1，除法器回到 DivFree 状态，并且所有变量值清空。

2.4.2 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如下。

### (1) 波形图 1

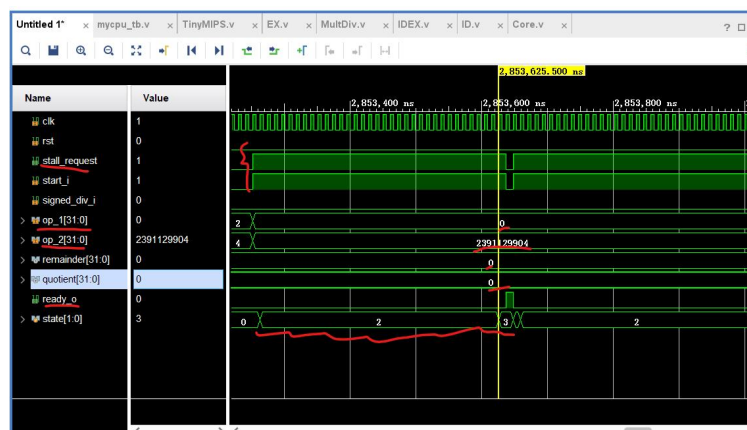
被除数为 2015401422，除数为-1816252416，有符号运算。



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。

### (2) 波形图 2

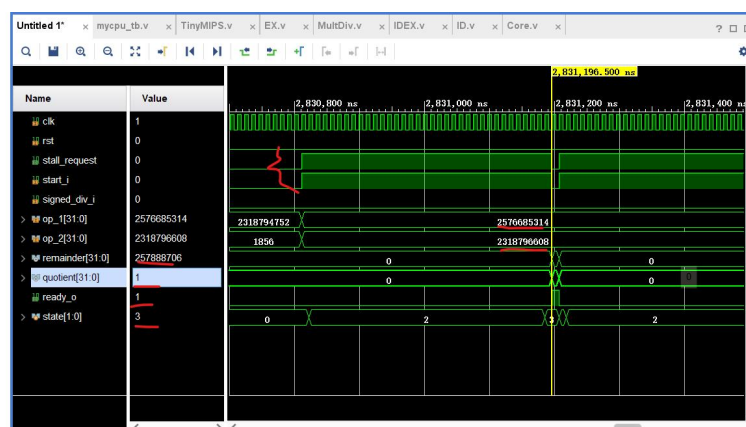
被除数为 0，除数为 2391129904，无符号运算。



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。

### (3) 波形图 3

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。

## 2.5 异常处理

### 2.5.1 设计过程

#### (一) 异常基础

首先实现了对 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 == `CPO_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
```

CP0 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,
```

## (二) 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_XOR, 'FUNCT_SLT, 'FUNCT_SLTU,
        'FUNCT_SYSCALL, 'FUNCT_BREAK: begin
          ri_flag <= 0;
        end
        default: ri_flag <= 1;
      endcase
    end
    'OP_B: begin
      case (rt)
        'RT_BLTZ, 'RT_BLTZAL,
        'RT_BEZ, 'RT_BEZAL: 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]));
//001/110
//在 ex 真正判断是否有溢出例外
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。

```
wire adel_flag, ades_flag;//与 load 和 store 相关的地址例外信号
assign adel_flag = (mem_read_flag_in && ~(mem_sel_in == 4'b0000) || (mem_sel_in == 4'b0
```

```

001) ||
    ((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'b0
001) ||
    ((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; //这是一个可能发生的地址错误例外，pc 未对对齐
字边界

```

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
    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

```

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。

//若存在异常，则进行冲刷

```
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。

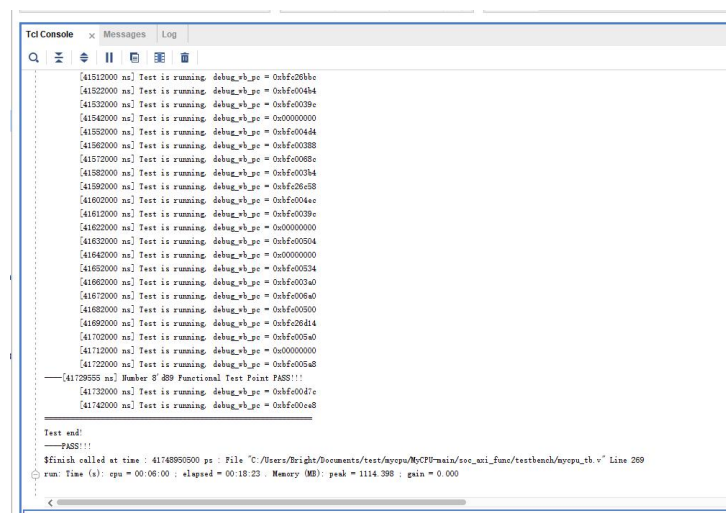
```
always @(*) begin
    if (flush) begin
        next_pc <= exc_pc;
    end
    else if (!stall_pc) begin
        if (branch_flag) begin
            next_pc <= branch_addr;
        end
        else begin
            next_pc <= pc + 4;
        end
    end
end
else begin
```

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
```

## 2.5.2 实验现象及分析

由于该部分涉及到的指令数量过多，不便于逐条测试，因此将异常处理部分的测试直接以 trace 比对的方式进行，trace 比对结果如下，trace 比对通过说明异常处理部分实现的功能完整。



## 2.6 控制外设完成流水灯

### 2.6.1 设计过程

#### (1) 设计思路



我们基于 TinyMIPS 指令和部分自己扩展的指令，通过修改 start.S 汇编文件，实现控制实验箱外设的功能。具体而言，我们实现了综合控制龙芯实验箱的双色 LED 灯、单色 LED 灯、数码管、拨码开关、4\*4 键盘等部件，各个外设之间相互配合，最终形成了单向循环流水灯的功能。

具体功能如下：

- a、控制 LED 灯循环地从右向左移动。
- b、按下 4\*4 小键盘的右下角按键后让 LED 灯重新回到最右端开始流动，并使数码管计数+1。
- c、当 LED 灯流动到最左端时，数码管计数清零。
- d、通过 8 个拨码开关可以控制 LED 灯的流动速度。

## (2) 关键代码

这一部分中，我们参考了指导书的部分代码与设计思路，在此基础上进行代码与功能的创新设计，从而实现单向循环流水灯的功能。

部分关键代码如下：

### A、寄存器和地址的初始化：

```
LI (a0, LED_RG1_ADDR)
LI (a1, LED_RG0_ADDR)
LI (s1, NUM_ADDR)
LI (s7, LED_ADDR)
LI (s4, BTN_KEY_ADDR)
```

这部分设置程序中使用的各种寄存器和地址的初始值。将 LED 灯、七段数码管、小键盘按键的地址加载到寄存器中。LED\_RG1\_ADDR、NUM\_ADDR 等使用了 cpu\_cde.h 头文件来获取地址。

### B、为一些寄存器赋初值：

```
LI (t1, 0x0002)
LI (t2, 0x0001)
LI (t7, 0x0001)
LI (t8, 0xffff)
LI (s8, 0x8000)
LI (s6, 1)
LI (a3, 0)
lui s3, 0
sw t1, 0(a0)
sw t2, 0(a1)
sw s3, 0(s1)
sw t7, 0(s2)
lui s0, 0          ## initial run number
LI(t4, 0)
LA (t1, kseg1_kseg0)
LI (t2, 0x20000000)
subu t9, t1, t2    #kseg1 -> kseg0
JR (t9)
nop
```

这部分实现了各个外设的初始化设置，令初始状态为：单色 LED 灯全灭，双色 LED 灯一红一绿。并设置寄存器负责存储 LED 灯移动的位置、后续用来异或取反的数（因为 LED

与开关都是共阳极）、单向循环方向的临界位置、小键盘的值、七段数码管的初始值等。最后 JR 跳转到主函数部分。

### C、主函数：

```
kseg1_kseg0:
    inst_test:
        LA (t9, kseg0_kseg1)
        JR (t9)
        nop
kseg0_kseg1:
    jal toleft
    nop
```

这段代码定义了两个标签：kseg1\_kseg0 和 kseg0\_kseg1，并且在 kseg1\_kseg0 标签中通过 LA 汇编指令加载了 kseg0\_kseg1 的地址到寄存器 t9 中，然后通过 JR 指令跳转到 kseg0\_kseg1 标签的地址。

在 kseg0\_kseg1 标签中，使用了 jal 指令调用了 toleft 函数。

这段代码的作用是实现程序的逻辑跳转和函数调用，完成 LED 灯控制逻辑的执行。

### D、toleft 函数：

```
toleft:
    SLL t7, t7, 1 #将寄存器 t7 中的值逻辑左移一位，把 LED 的位置左移一格
    xor t9, t8, t7 #使用异或操作，对 t7 和 t8 异或，实现了 LED 灯的亮灭效果
    sw t9, 0(s7) # 将 t9 存储到地址 s7 对应的位置，控制 LED 灯的亮灭状态
    jal wait_1s #拨码开关控制 CPU 的延迟（若干次循环，实时检测开关的值）
    nop
    lw a3, 0(s4) #从地址 s4 加载数据到寄存器 a3 中，用于检测按键开关的状态
    LI (t5, 0x8000) #将常数 0x8000 加载到寄存器 t5 中
    beq a3, t5, againleft # 流动到最左端，a3 和 t5 相等则跳转到 againleft
    nop
    bne t7, s8, toleft # 未到最左边，继续通过 toleft 左移一位
    nop
```

这段代码实现了每执行一次，控制 LED 灯向左移动一位的效果。整体功能为：首先使用 SLL 逻辑左移实现 LED 灯的位置左移一格的效果；然后通过异或操作使得 LED 灯当前位置点亮，前一位置熄灭；wait\_1s 内部通过若干次循环实时检测拨码开关的值，从而控制 CPU 的延迟效果，即 LED 灯的流动速度；通过检测 4\*4 小键盘的按键状态来确定是否需要将流水灯恢复到最右端并让数码管显示数值加一；若检测到 LED 灯当前位置是流水灯的最左端则跳转到 againleft 重新回到最右端并初始化，否则继续跳转到 toleft 开头实现 LED 灯左移一位。

### E、againleft 函数：

```
againleft:
    LI (a0, LED_RG1_ADDR)
    LI (a1, LED_RG0_ADDR)
    LI (t7, 0x0001)
    lui t4, 0x100
    addu s3, s3, t4
    sw s3, 0(s1)
    LA (t9, kseg0_kseg1)
```

JR (t9)

nop

这段代码的主要功能可以总结为重新初始化 LED 灯的地址到最右端,并重新设置寄存器的初始值,使得状态初始化。完成初始化后,通过 JR 指令跳转到主函数部分,从而继续调用 toleft 函数,完成 LED 灯左移的操作。

初始化 LED 地址:通过 LI 指令将 LED 灯的地址初始化为最右端的地址,使用了 LED\_RG1\_ADDR 和 LED\_RGO\_ADDR 作为地址的常数值。

重新初始化寄存器值:使用 LI 指令将寄存器 t7 重新初始化为 0x0001,以及对寄存器 s3 进行加法操作,并将结果存储到地址 s1 对应的位置。

### 2.6.2 实验现象及分析

为了便于展示单向循环流水灯的效果,我们录制了演示视频 show.mp4,并将其作为附件对象嵌入到本报告中,可以通过双击下面图标来观看视频。



show.mp4

视频的操作流程如下:

- 观察流水灯从右向左单向循环的效果,此时数码管计数为 0。
- 多次按下 4\*4 键盘右下角的按键,观察到按键能成功控制流水灯回到最右端开始流动,并且每按一次数码管计数会加一。
- 当停止按键操作时,流水灯会正常流动到最左端,然后数码管数值清零,流水灯回到最右端继续循环向左流动。

然而,本视频也存在一定的疏忽。我们仅仅通过调节拨码开关为流水灯设置了一个合适的流动速度,忘记展示通过拨码开关改变流水灯流动速度的情景,这也提醒我们以后在视频录制中要事先做好整体规划,从而覆盖到所有可能的测试情况。

### 3 组内成员主要工作及贡献比例

李晓坤	负责实现 CP0 协处理器及相关指令、特权指令和异常处理部分	20%
王若凡	负责实现基于汇编语言的单向流水灯	20%
施耀民	负责实现 HILO 寄存器及基于华莱士树的多周期乘法器	20%
成豪	负责实现 HILO 寄存器及迭代除法器	20%
王宠爱	负责实现基础指令的拓展,完成各个部分的测试	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)

## 1、结论（实验总结）

### （1）李晓坤

在课程设计期间，我主要负责实现 CP0 协处理器及相关指令、特权指令和异常处理部分。其中我的主要工作包括：（1）实现 CP0 协处理器，为处理器增加功能提供基础；（2）实现特权指令和异常处理，确保系统能够在出现异常情况时正确地响应和处理，保证系统的稳定性和安全性。在课程设计中，我对处理器的设计方法有了深入的理解，并培养了自己对一个复杂系统分层设计的思维；基本掌握了指令的拓展方法，能够根据需求实现指令的拓展；熟悉了诸如 Verilog、Mars、GCC 等工具的使用，能够独立使用这些工具对处理器进行设计和改进；最重要的，我了解了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了深入的了解，意识到处理器与其他组件之间的交互和协作。

除此之外，作为小组组长，在课程设计过程中我承担了领导和组织工作，负责整个团队的协调和管理。我在团队管理、沟通协调等非技术方面也得到了锻炼，其中所积累的经验对于未来的生涯发展也是极为重要的。

### （2）王若凡

在完成基于汇编语言的单向循环流水灯实验过程中，我们深入探索了底层系统编程和硬件控制，取得了一系列实质性的收获和能力的提升。我们将从主要工作和收获两部分进行总结。

我们成功实现了对龙芯实验箱多个外设的综合控制，包括双色 LED 灯、单色 LED 灯、数码管、拨码开关、4\*4 键盘等。通过修改 start.S 汇编文件，我们使这些外设协同工作，最终形成了单向循环流水灯的整体功能。本次创新项目使用的是 MIPS 汇编语言，而且代码逻辑清晰简捷，方便人们进一步理解 MIPS 指令。我们不仅使用了 TinyMIPS 中的部分指令，还运用了自行扩展的一些指令，将自己以前的工作任务运用到创新场景中，使得我们很有成就感。

通过创新设计和修改代码，我们学到了如何在汇编语言中实现复杂的功能。在调试过程中，我们解决了一系列问题，提升了我们的调试能力和解决问题的技巧。

### （3）施耀民

在课程设计期间，我完成的工作如下：设计了三条乘法指令（mul、mult、multu），包括指令格式和功能码的定义。对设计的乘法器进行了原理分析，涵盖了华莱士树乘法器和 Booth 编码在乘法器中的优化作用。详细说明了华莱士树法的分解、部分积生成、部分积的合并等步骤，以及 Booth 编码的减少加法操作次数、部分积压缩、提高并行性等优势。描述了乘法器和除法器的系统结构设计，包括模块接口、输入输出、与其他模块的交互等方面的设计。给出了模块的接口定义和功能说明。

通过实践，对乘法器设计进行详细分析，加深了对乘法器原理和优化技术的理解，包括华莱士树法和 Booth 编码的应用。通过设计乘法器和除法器的系统结构，提升了模块设计的能力，包括接口定义、模块功能划分、与其他模块的交互等方面的考虑。

### （4）成豪

在本次计算机组成原理课程设计中，我在小组中负责完成除法器的实现，以及相应部分 ppt 的编写，在这其中收获颇丰，更深刻的与之前课上的知识相结合，了解了流水线实现的原理，对指令集的设计包括指令格式、指令类型、寻址模式等更加深刻，同时加深了同学之间的友谊，锻炼了自己的动手和表达能力。

### （5）王宠爱

在进行课程设计的过程中，我理解了 TinyMIPS 的设计过程，逐渐了解流水线 CPU 的工作原理和设计方式以及指令执行过程，以及各种指令的运行所需要的基础的元器件；同时在

进行虚拟仿真实验时，理解了流水线处理器中数据冲突的解决技术，即数据前递和流水线暂停。基于理论知识基础我逐步完成了 28 条基础指令的扩展，这 28 条基础指令包括逻辑运算指令、算术运算指令、分支跳转指令、访存指令，同时协助完成了流水灯的测试过程。在这个过程中，我的理论知识不够完善，让我在进行指令扩展等操作时容易遇到难题，需要进一步完善理论知识，同时在相应的实验环境中动手操作以便更好的理解相关原理。

## 2、讨论（问题归纳，课程建议等）

### （1）李晓坤

在课程设计期间我们遇到的最大问题就是技术上的难题和挑战，处理器设计是一项复杂的任务，涉及到设计方法、指令拓展、异常处理等方面。为了解决这个问题，我们团队成员之间积极交流，寻求解决方案，并利用现代工具和资源进行支持和辅助，如查找网络资料等。

对于本课程，希望能够在课堂给予我们更多的提示和指导，以此更加出色的完成该课程设计。

### （2）王若凡

在本次小组作业中，由于我负责的流水灯创新部分需要多次连接实验箱检验结果，而工程文件进行一次综合和生成比特流大约需要 10-20 分钟，因此每次结果验证都需要等待较长的时间，使得完成最终的代码调试消耗时间较长。但是当我们看到单向循环流水灯完整地实现了我们设计的所有功能的时候，确实会感觉到十足的成就感和喜悦。

我认为本门课程提供了详细的学习资料，能够逐步引导我们完成课程的学习和任务，整体学习效果较好，收获很多，没有特别的建议。

### （3）施耀民

问题：代码片段未提供完整上下文，文档中插入的代码片段缺乏完整的上下文，难以全面理解；硬件实现细节不明确，文档中提到了乘法器的硬件实现，但未详细说明具体的硬件结构和实现细节，这使得理解和复现的难度增加。

改进方向：硬件实现详述，对乘法器的硬件实现进行更详细的描述，包括具体的电路结构和信号传输方式，使读者更容易理解。

课程建议：如果文档是为课程或培训提供的，可以添加一些课程建议，例如推荐的进一步学习资源、实践项目等。

### （4）成豪

在我负责的除法器部分，前期也遇到了很大的困难，包括对于程序整体的理解，以及相关流水级的实现，还有对 div 模块中各个状态机程序的实现都存在较大疑问，最终在查阅指导书和相关资料得到了解决。对本课程我建议可以在时间上与其他实验课分开，可以降低一些难度。

### （5）王宠爱

在进行课程设计初期，对各指令的功能不够熟悉，编写 Verilog 代码的能力也需要进一步加强，需要在实验之前复习相关知识，以便顺利进行后面的操作。通过本次课程，我对计算机组成原理有了更深入的理解，提高了编写 Verilog 代码的能力，理论实践的结合让我对 CPU 等硬件设备更加了解。

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 data-bbox="240 365 1093 521">(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> <div data-bbox="764 1043 855 1084">签名：</div> <div data-bbox="767 1211 855 1252">日期：</div>	