同济大学

TONGJI UNIVERSITY

《计算机系统实验》

实验报告

实验名称	CPU 改造：MIPS 89 条指令动态流水线 CPU
实验成员	冯羽芯（2251206）
日　　期	二零二五年 三 月 二十八 日

装 订 线

1、实验目的

- 回顾 MIPS 五阶段流水线 CPU 的基本结构
- 学习教学用 CPU 各模块的编写以及模块之间的连接结构，将原有的 MIPS 54 条指令动态流水线 CPU 进行改造，使其支持 89 条指令，并支持异常处理和 CP0
- 加深对 Verilog 语言的理解以及对硬件开发的理解
- 为之后移植操作系统做准备工作

2、实验内容

(1) 实验环境与硬件配置

- 处理器： 13th Gen Intel(R) Core(TM) i7-13650HX，2600 Mhz
- 开发平台：Vivado 2022.2
- 仿真环境：Vivado 2022.2 自带仿真器
- 测试环境：MARS4.5
- 文档管理：Office Word 2021
- 开发板： NEXYS4DDRAtrix-7

(2) 实验方案

参考《自己动手写 CPU》，改造 54 条指令 MIPS 流水线 CPU，支持到 89 条指令，实现 CP0、异常处理，需要新增的 35 条指令如下：

- 移动操作指令： `movn、movz`
- 算术操作指令： `clo、madd、maddu、msub、msubu`
- 转移指令： `b、bal、bgezal、bgtz、blez、bltz、bltzal`
- 加载存储指令： `ll、lwl、lwr、sc、swl、swr`
- 异常相关指令： `tge、tgeu、tlt、tltu、tne、teqi、tgei、tgeiu、tlti、tltiu、tnei`
- 其他指令： `nop、ssnop、sync、pref`

此次实验中实现的 MIPS CPU 为五级流水线 CPU，分别有取指、译码、执行、访存、回写五个阶段。存储模式采取大端模式。整体采用哈佛结构，有分开的指令存储器和数据存储器。

3、实验步骤

(1) 数据通路设计

新增指令之前，54 条 MIPS 动态流水线 CPU 的总体结构和数据通路设计如下。但由于该结构与《自己动手写 CPU》中的 OpenMIPS 结构相差较大，故后续并未采用此结构，使用了 OpenMIPS 进行改造。

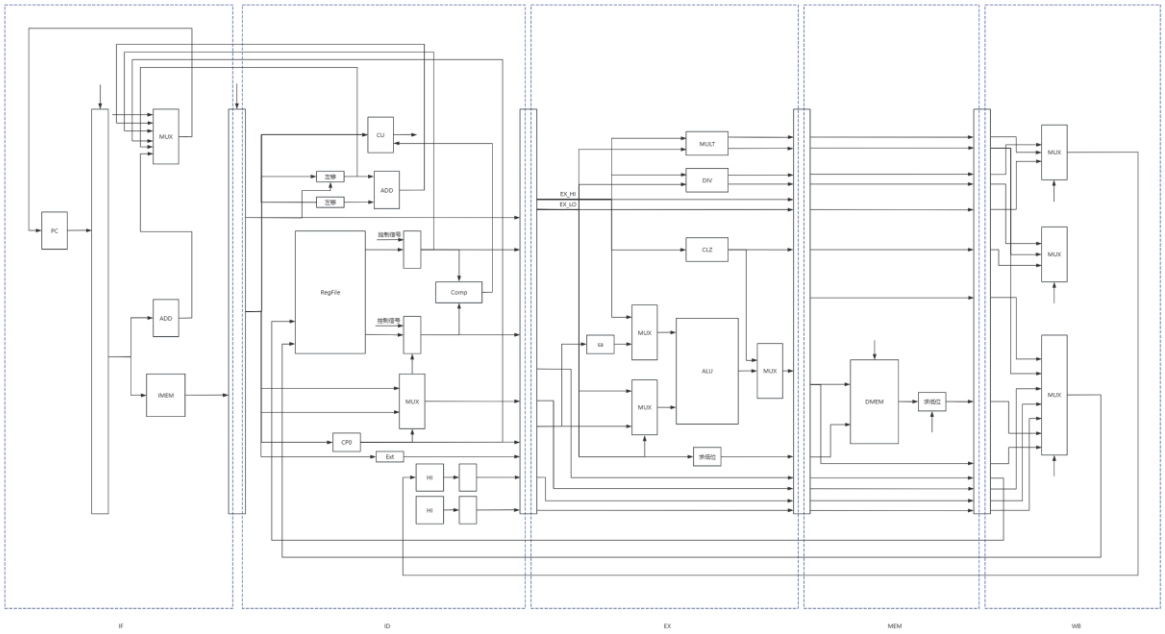

图 1：54 条 MIPS 动态流水线 CPU 的数据通路

在设计 89 条 MIPS CPU 之前，在 54 条 MIPS 动态流水线 CPU 的基础上修改完成了数据通路的设计，其总体结构和数据通路设计如下（图 2）。

(2) 指令设计

1. movn

- 格式：`movn rd,rs,rt`
- 描述：如果 `rt` 不等于 0，则 `rd` 获取 `rs` 的值。如果 `rt` 等于 0，则 `rd` 的值不变，`movn` 指令的功能是在非零条件下移动寄存器的值。
- 涉及的硬件：PC、NPC、IMEM、RegFile。

2. movz

- 格式：`movz rd,rs,rt`
- 描述：如果 `rt` 等于 0，则 `rd` 获取 `rs` 的值。与 `movn` 指令相对，其功能是在零条件下移动寄存器的值。
- 涉及的硬件：PC、NPC、IMEM、RegFile。

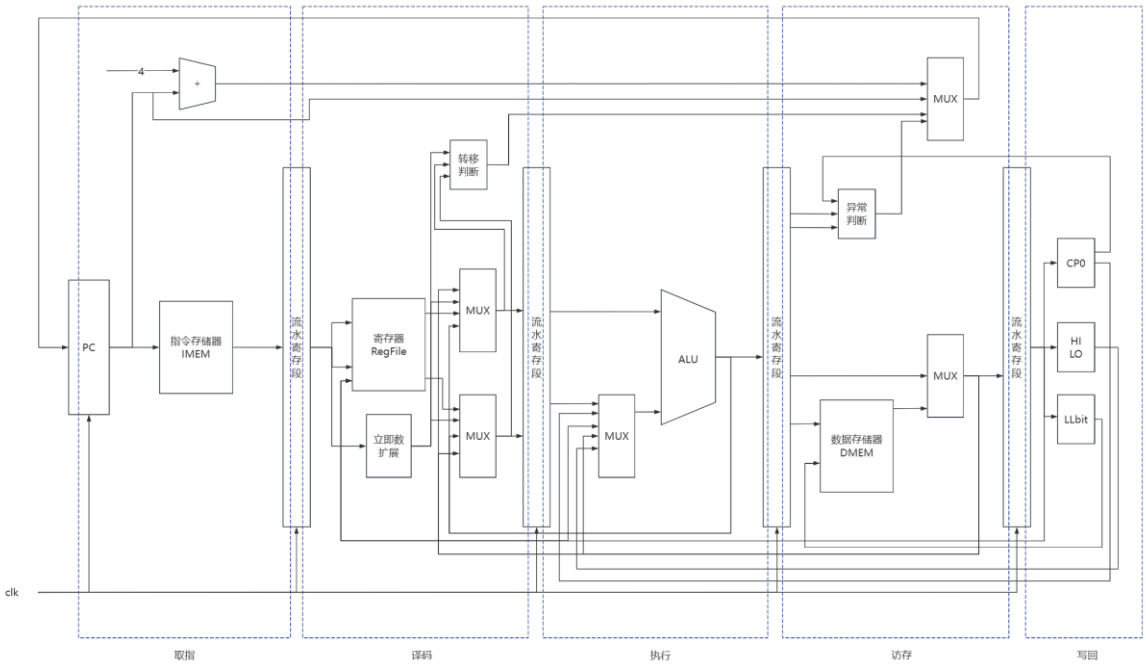

图 2：89 条 MIPS 动态流水线 CPU 的数据通路

3. clo

- 格式：`clo rd,rs`
- 描述：该指令用于计算 `rs` 寄存器中最高位开始连续的“1”的个数。例如，如果 `rs` 寄存器的值为 `0xFFFFFFFF`，则 `clo` 指令会计算出 32，因为该数值的二进制表示是连续 32 个“1”。
- 涉及的硬件：PC、NPC、IMEM、RegFile。

```
`EXE_CLO_OP: begin        // 计数运算 clo
    arithmeticres <= (reg1_i_not[31] ? 0 :
                      reg1_i_not[29] ? 2 :
                      reg1_i_not[28] ? 3 :
                      reg1_i_not[27] ? 4 :
                      reg1_i_not[26] ? 5 :
                      reg1_i_not[25] ? 6 :
                      reg1_i_not[24] ? 7 :
                      reg1_i_not[23] ? 8 :
```

```
                        reg1_i_not[22] ? 9 :
                        reg1_i_not[21] ? 10 :
                        reg1_i_not[20] ? 11 :
                        reg1_i_not[19] ? 12 :
                        reg1_i_not[18] ? 13 :
                        reg1_i_not[17] ? 14 :
                        reg1_i_not[16] ? 15 :
                        reg1_i_not[15] ? 16 :
                        reg1_i_not[14] ? 17 :
                        reg1_i_not[13] ? 18 :
                        reg1_i_not[12] ? 19 :
                        reg1_i_not[11] ? 20 :
                        reg1_i_not[10] ? 21 :
                        reg1_i_not[9] ? 22 :
                        reg1_i_not[8] ? 23 :
                        reg1_i_not[7] ? 24 :
                        reg1_i_not[6] ? 25 :
                        reg1_i_not[5] ? 26 :
                        reg1_i_not[4] ? 27 :
                        reg1_i_not[3] ? 28 :
                        reg1_i_not[2] ? 29 :
                        reg1_i_not[1] ? 30 :
                        reg1_i_not[0] ? 31 : 32) ;
end
```

4. madd

- 格式：madd rs,rt
- 描述：该指令用于执行乘法加法操作，它将 rs 和 rt 寄存器的值相乘，然后将结果加到特殊寄存器 HI,LO 中。如果乘法结果超出了 32 位，HI 寄存器存储高 32 位，而 LO 寄存器存储低 32 位。
- 涉及的硬件：PC、NPC、IMEM、RegFile、HI,LO。

5. maddu

- 格式：maddu rs,rt
- 描述：该指令类似于 madd，但它用于无符号数的乘法加法操作。同样地，乘法结果的高位存储在 HI 寄存器，低位存储在 LO 寄存器。
- 涉及的硬件：PC、NPC、IMEM、RegFile、HI,LO。

```
`EXE_MADD_OP, `EXE_MADDU_OP: begin      // madd、maddu 指令
    if(cnt_i == 2'b00) begin             // 执行阶段第一个时钟周期
        hilo_temp_o           <= mulres;
```

```
        cnt_o                   <= 2'b01;
        hilo_temp1              <= {`ZeroWord,`ZeroWord};
        stallreq_for_madd_msub <= `Stop;
    end
    else if(cnt_i == 2'b01) begin       // 执行阶段第二个时钟周期
        hilo_temp_o             <= {`ZeroWord,`ZeroWord};
        cnt_o                   <= 2'b10;
        hilo_temp1              <= hilo_temp_i + {HI,LO};
        stallreq_for_madd_msub <= `NoStop;
    end
end
```

6. msub

- 格式：msub rs,rt
- 描述：此指令将 rs 和 rt 寄存器的值相乘，然后将结果从 HI,LO 寄存器中的现有值中减去。如果乘法结果超出了 32 位，则 HI 寄存器存储高 32 位，而 LO 寄存器 存储低 32 位。
- 涉及的硬件：PC、NPC、IMEM、RegFile、HI,LO。

7. msubu

- 格式：msubu rs,rt
- 描述：当执行 rs 和 rt 的无符号数乘法操作，结果会从特殊寄存器 HI,LO 中现有的值中减去。如果 rs 和 rt 的乘法结果超过了 32 位，高位存储在 HI 寄存器，低位存储在 LO 寄存器。msubu 指令的作用是执行无符号数乘法后的减法操作。
- 涉及的硬件：PC、NPC、IMEM、RegFile、HI,LO。

```
`EXE_MSUB_OP, `EXE_MSUBU_OP: begin   // msub、msubu 指令
    if(cnt_i == 2'b00) begin           // 执行阶段第一个时钟周期
        hilo_temp_o             <=  ~mulres + 1 ;
        cnt_o                   <= 2'b01;
        stallreq_for_madd_msub <= `Stop;
    end
    else if(cnt_i == 2'b01) begin       // 执行阶段第二个时钟周期
        hilo_temp_o             <= {`ZeroWord,`ZeroWord};
        cnt_o                   <= 2'b10;
        hilo_temp1              <= hilo_temp_i + {HI,LO};
        stallreq_for_madd_msub <= `NoStop;
    end
end
```

8. b

- 格式：`b offset`
- 描述：无条件分支，当指令执行时，处理器会跳转到 `b` 指令的当前位置加上 `offset` 的目标地址。`b` 指令简化了代码流程控制，可以实现无条件的跳转。 `b` 指令可以认为是 `beq` 指令的特殊情况，当 `beq` 指令的 `rs`、`rt` 都等于 0 时，即为 `b` 指令，所以在 OpenMIPS 实现的时候不需要特意实现 `b` 指令，只需要实现 `beq` 指令即可。
- 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT18、ADD。

9. bal

- 格式：`bal offset`
- 描述：无条件分支，同时将返回地址存储在寄存器 `$ra` 中，方便函数调用后返回到原程序继续执行。`bal` 指令等效于 `bgezal` 指令的特殊情况，即 `bgezal` 的条件总是满足的情况，因此总是执行跳转。`bal` 指令是 `bgezal` 指令的特殊情况，当 `bgezal` 指令的 `rs` 为 0 时，就是 `bal` 指令，所以在 OpenMIPS 实现时，不用特意考虑 `bal` 指令，只要实现 `bgezal` 指令即可。
- 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT18、ADD。

10. bgezal

- 格式：`bgezal rs,offset`
- 描述：如果 `rs` 大于等于 0，则执行分支，并且将返回地址保存在寄存器 `$ra` 中，用于支持函数的调用与返回。该指令在实现函数调用时很有用，因为它提供了一个链接到返回地址的机制。
- 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT18、ADD。

```
`EXE_BGEZAL: begin   // bgezal 指令
    wreg_o      <= `WriteEnable;
    aluop_o     <= `EXE_BGEZAL_OP;
    alusel_o    <= `EXE_RES_JUMP_BRANCH;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b0;
    link_addr_o <= pc_plus_8;
    wd_o        <= 5'b11111;
    instvalid   <= `InstValid;
    if(reg1_o[31] == 1'b0) begin
        branch_target_address_o  <= pc_plus_4 + imm_sll2_signedext;
        branch_flag_o            <= `Branch;
        next_inst_in_delayslot_o <= `InDelaySlot;
    end
```

```
end
```

11. bgtz

- 格式：`bgtz rs,offset`
- 描述：如果 `rs` 大于 0，则执行分支操作。`bgtz` 指令用于条件分支，通常与循环或条件语句结合使用，以实现更复杂的控制流程。
- 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT18、ADD。

```
`EXE_BGTZ: begin                // bgtz 指令
    wreg_o      <= `WriteDisable;
    aluop_o     <= `EXE_BGTZ_OP;
    alusel_o    <= `EXE_RES_JUMP_BRANCH;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b0;
    instvalid   <= `InstValid;
    if((reg1_o[31] == 1'b0) && (reg1_o != `ZeroWord)) begin
        branch_target_address_o  <= pc_plus_4 + imm_sll2_signedext;
        branch_flag_o            <= `Branch;
        next_inst_in_delayslot_o <= `InDelaySlot;
    end
end
```

12. blez

- 格式：`blez rs,offset`
- 描述：如果 `rs` 小于等于 0，则执行分支操作，处理器跳转到 `blez` 指令的当前位置加上 `offset` 的目标地址。`blez` 指令用于在 `rs` 寄存器的值不大于零时控制程序流程。
- 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT18、ADD。

```
`EXE_BLEZ:       begin        // blez 指令
    wreg_o      <= `WriteDisable;
    aluop_o     <= `EXE_BLEZ_OP;
    alusel_o    <= `EXE_RES_JUMP_BRANCH;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b0;
    instvalid   <= `InstValid;
    if((reg1_o[31] == 1'b1) || (reg1_o == `ZeroWord)) begin
        branch_target_address_o  <= pc_plus_4 + imm_sll2_signedext;
        branch_flag_o            <= `Branch;
        next_inst_in_delayslot_o <= `InDelaySlot;
    end
end
```

13. bltz

- 格式：`bltz rs,offset`
- 描述：如果 `rs` 小于 0，则执行分支操作，处理器跳转到 `bltz` 指令的当前位 置加上 `offset` 的目标地址。`bltz` 指令专用于在 `rs` 寄存器的值为负数时实现程序的条件分支。
- 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT18、ADD。

```
`EXE_BLTZ: begin     // bltz 指令
    wreg_o      <= `WriteDisable;
    aluop_o     <= `EXE_BLTZ_OP;
    alusel_o    <= `EXE_RES_JUMP_BRANCH;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b0;
    instvalid   <= `InstValid;
    if(reg1_o[31] == 1'b1) begin
        branch_target_address_o  <= pc_plus_4 + imm_sll2_signedext;
        branch_flag_o            <= `Branch;
        next_inst_in_delayslot_o <= `InDelaySlot;
    end
end
```

14. bltzal

- 格式：`bltzal rs,offset`
- 描述：如果 `rs` 小于 0，则执行分支，并且将返回地址保存在寄存器 `$ra` 中。`bltzal` 指令不仅用于条件分支，而且用于支持函数调用与返回，使得程序能够在执行完函数后返回到原来的位置。
- 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT18、ADD。

```
`EXE_BLTZAL: begin   // bltzal 指令
    wreg_o      <= `WriteEnable;
    aluop_o     <= `EXE_BLTZAL_OP;
    alusel_o    <= `EXE_RES_JUMP_BRANCH;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b0;
    link_addr_o <= pc_plus_8;
    wd_o        <= 5'b11111;
    instvalid   <= `InstValid;
    if(reg1_o[31] == 1'b1) begin
        branch_target_address_o <= pc_plus_4 + imm_sll2_signedext;
        branch_flag_o <= `Branch;
        next_inst_in_delayslot_o <= `InDelaySlot;
    end
```

```
end
```

15. ll

- 格式：ll rt, offset(base)
- 描述：ll 指令用于从内存中加载一个字到寄存器 rt，同时设置内存系统的一个监测位，用于后续的 sc（StoreConditional）指令实现原子操作。如果在执行 sc 指令之前该地址被修改，sc 操作将失败。
- 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、DMEM。

```
`EXE_LL_OP: begin                // ll 指令的访存输出
    mem_addr_o    <= mem_addr_i;
    mem_we        <= `WriteDisable;
    wdata_o       <= mem_data_i;
    LLbit_we_o    <= 1'b1;
    LLbit_value_o <= 1'b1;
    mem_sel_o     <= 4'b1111;
    mem_ce_o      <= `ChipEnable;
end
```

16. lwl

- 格式：lwl rt, offset(base)
- 描述：该指令用于加载字的左边部分到寄存器 rt，一般与 lwr 指令组合使用来加载非对齐的字。lwl 指令根据对齐的字的边界，从内存中读取字的左边部分，拼接到寄存器 rt 中。与 lhu、lh 指令的作用不同，后两者是加载半字。注意，大端模式和小端模式下 lwl 指令的效果有所不同。
- 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT16、DMEM。

```
`EXE_LWL_OP: begin              // lwl 指令
    mem_addr_o <= {mem_addr_i[31:2], 2'b00};
    mem_we     <= `WriteDisable;
    mem_sel_o  <= 4'b1111;
    mem_ce_o   <= `ChipEnable;
    case (mem_addr_i[1:0])
        2'b00: begin
            wdata_o <= mem_data_i[31:0];
        end
        2'b01: begin
            wdata_o <= {mem_data_i[23:0], reg2_i[7:0]};
        end
        2'b10: begin
            wdata_o <= {mem_data_i[15:0], reg2_i[15:0]};
```

```
        end
        2'b11: begin
            wdata_o <= {mem_data_i[7:0], reg2_i[23:0]};
        end
        default: begin
            wdata_o <= `ZeroWord;
        end
    endcase
end
```

17. lwr

- 格式：`lwr rt, offset(base)`
- 描述：与 `lwl` 指令相对应，用于加载字的右边部分到寄存器 `rt`，一般与 `lwl` 指令组合使用来加载非对齐的字。 注意，大端模式和小端模式下 `lwr` 指令的效果有所不同。
- 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT16、DMEM。

```
`EXE_LWR_OP: begin              // lwr 指令
    mem_addr_o <= {mem_addr_i[31:2], 2'b00};
    mem_we     <= `WriteDisable;
    mem_sel_o  <= 4'b1111;
    mem_ce_o   <= `ChipEnable;
    case (mem_addr_i[1:0])
        2'b00: begin
            wdata_o <= {reg2_i[31:8],mem_data_i[31:24]};
        end
        2'b01: begin
            wdata_o <= {reg2_i[31:16],mem_data_i[31:16]};
        end
        2'b10: begin
            wdata_o <= {reg2_i[31:24],mem_data_i[31:8]};
        end
        2'b11: begin
            wdata_o <= mem_data_i;
        end
        default: begin
            wdata_o <= `ZeroWord;
        end
    endcase
end
```

18. sc

- 格式：`sc rt, offset(base)`

• 描述：该指令用于实现原子交换操作，也就是 LLbit（LoadLinkedbit）为 1 时，将寄存器 rt 的内容存储到指定的内存地址，并将 LLbit 设为 0。如果 RMW（Read Modify Write）指令组执行了此操作，也就是 LLbit 为 0，则不会执行存储，并将寄存器 rt 中的内容设为 0。

• 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT16、DMEM。

```
`EXE_SC_OP: begin                   // sc 指令的访存输出
    if(LLbit == 1'b1) begin
        LLbit_we_o    <= 1'b1;
        LLbit_value_o <= 1'b0;
        mem_addr_o    <= mem_addr_i;
        mem_we        <= `WriteEnable;
        mem_data_o    <= reg2_i;
        wdata_o       <= 32'b1;
        mem_sel_o     <= 4'b1111;
        mem_ce_o      <= `ChipEnable;
    end
    else begin
        wdata_o       <= 32'b0;
    end
end
```

19. swl

• 格式：swl rt, offset(base)

• 描述：将寄存器 rt 中的字的左边部分存储到内存的指定地址中，通常与 swr 指令配合使用以实现非对齐的字的存储操作。swl 指令根据对齐的字边界，将寄存器 rt 中的左边部分存储到内存。与 sh 指令的作用不同，后者是存储半字。 注意，大端模式和小端模式下 swl 指令的效果有所不同。

• 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT16、DMEM。

```
`EXE_SWL_OP: begin                  // swl 指令
    mem_addr_o <= {mem_addr_i[31:2], 2'b00};
    mem_we     <= `WriteEnable;
    mem_ce_o   <= `ChipEnable;
    case (mem_addr_i[1:0])
        2'b00: begin
            mem_sel_o <= 4'b1111;
            mem_data_o <= reg2_i;
        end
        2'b01: begin
            mem_sel_o <= 4'b0111;
```

```
            mem_data_o <= {zero32[7:0], reg2_i[31:8]};
        end
        2'b10: begin
            mem_sel_o <= 4'b0011;
            mem_data_o <= {zero32[15:0], reg2_i[31:16]};
        end
        2'b11: begin
            mem_sel_o <= 4'b0001;
            mem_data_o <= {zero32[23:0], reg2_i[31:24]};
        end
        default: begin
            mem_sel_o <= 4'b0000;
        end
    endcase
end
```

20. swr

1. 格式：`swr rt, offset(base)`

2. 描述：将寄存器 `rt` 中的字的右边部分存储到内存的指定地址中，通常与 `swl` 指令配合使用以实现非对齐的字的存储操作。 注意，大端模式和小端模式下 `swr` 指令的效果有所不同。

3. 涉及的硬件：PC、NPC、IMEM、RegFile、ALU、EXT16、DMEM。

```
`EXE_SWR_OP:  begin             // swr 指令
    mem_addr_o <= {mem_addr_i[31:2], 2'b00};
    mem_we     <= `WriteEnable;
    mem_ce_o   <= `ChipEnable;
    case (mem_addr_i[1:0])
        2'b00: begin
            mem_sel_o  <= 4'b1000;
            mem_data_o <= {reg2_i[7:0], zero32[23:0]};
        end
        2'b01: begin
            mem_sel_o  <= 4'b1100;
            mem_data_o <= {reg2_i[15:0],zero32[15:0]};
        end
        2'b10: begin
            mem_sel_o  <= 4'b1110;
            mem_data_o <= {reg2_i[23:0],zero32[7:0]};
        end
        2'b11: begin
            mem_sel_o  <= 4'b1111;
            mem_data_o <= reg2_i[31:0];
        end
```

```
        default: begin
            mem_sel_o  <= 4'b0000;
        end
    endcase
end
```

21. tge

- 格式：tge rs, rt
- 描述：如果通用寄存器 rs 的值大于等于寄存器 rt 的值，则执行陷阱（trap） 操作。tge 指令用于在寄存器的值满足特定的条件时产生陷阱，用于程序的异常处理或调试。
- 涉及的硬件：PC、NPC、CPO、IMEM、RegFiles、ALU。

22. tgeu

- 格式：tgeu rs, rt
- 描述：如果通用寄存器 rs 的值在无符号比较下大于等于寄存器 rt 的值，则执行陷阱操作。tgeu 与 tge 指令相似，但专用于无符号数的比较。
- 涉及的硬件：PC、NPC、CPO、IMEM、RegFiles、ALU。

23. tlt

- 格式：tlt rs, rt
- 描述：如果通用寄存器 rs 的值小于寄存器 rt 的值，则执行陷阱操作。tlt 指令用于在寄存器的值不满足特定的条件时产生陷阱，用于程序的异常处理或调试。
- 涉及的硬件：PC、NPC、CPO、IMEM、RegFiles、ALU。

24. tltu

- 格式：tltu rs, rt
- 描述：如果通用寄存器 rs 的值在无符号比较下小于寄存器 rt 的值，则执行陷阱操作。tltu 指令用于在寄存器的值不满足特定的无符号比较条件时产生陷阱，用于程序的异常处理或调试。
- 涉及的硬件：PC、NPC、CPO、IMEM、RegFiles、ALU。

25. tne

- 格式：tne rs, rt
- 描述：如果通用寄存器 rs 的值不等于寄存器 rt 的值，则执行陷阱操作。tne 指令用于在寄存器的值不满足等值条件时产生陷阱，用于程序的异常处理或调试。

- 涉及的硬件：PC、NPC、CPO、IMEM、RegFiles、ALU。

26. teqi

- 格式：`teqi rs, immediate`
- 描述：如果通用寄存器 `rs` 的值等于立即数（立即数被符号扩展到 32 位）则执行陷阱操作。`teqi` 指令用于比较寄存器值和立即数，如果相等，则产生陷阱。
- 涉及的硬件：PC、NPC、CPO、IMEM、RegFiles、ALU。

27. tgei

- 格式：`tgei rs, immediate`
- 描述：如果通用寄存器 `rs` 的值大于等于立即数（立即数被符号扩展到 32 位）则执行陷阱操作。`tgei` 指令用于比较寄存器值和立即数，如果寄存器值大等于立即数，则产生陷阱。
- 涉及的硬件：PC、NPC、CPO、IMEM、RegFiles、ALU。

28. tgeiu

- 格式：`tgeiu rs, immediate`
- 描述：如果通用寄存器 `rs` 的值在无符号比较下大于或等于立即数 `immediate`，则执行陷阱操作。`tgeiu` 指令用于在寄存器的值无符号比较满足或超过特定立即数时产生陷阱，通常用于程序的异常处理或调试。
- 涉及的硬件：PC、NPC、CPO、IMEM、RegFiles、ALU。

29. tlti

- 格式：`tlti rs, immediate`
- 描述：如果通用寄存器 `rs` 的值小于立即数 `immediate`，则执行陷阱操作。`tlti` 指令用于在寄存器的值小于特定立即数时产生陷阱，用于程序的异常处理或调试。
- 涉及的硬件：PC、NPC、CPO、IMEM、RegFiles、ALU。

30. tltiu

- 格式：`tltiu rs, immediate`
- 描述：如果通用寄存器 `rs` 的值在无符号比较下小于立即数 `immediate`，则执行陷阱操作。`tltiu` 指令用于在寄存器的值无符号比较小于特定立即数时产生陷阱，用于程序的异常处理或调试。
- 涉及的硬件：PC、NPC、CPO、IMEM、RegFiles、ALU。

31. tnei

• 格式：tnei rs, immediate

• 描述：如果通用寄存器 rs 的值不等于立即数 immediate，则执行陷阱操作。 tnei 指令用于在寄存器的值不等于特定立即数时产生陷阱，用于程序的异常处理或调试。

• 涉及的硬件：PC、NPC、CPO、IMEM、RegFiles、ALU。

32. nop

• 描述：无操作。 nop 指令的功能码是 6'b000000，与逻辑左移指令 sll 的功能码相同，sll 指令向 $0 寄存器保存移位结果，实际不会有任何效果，因为无论向 $0 写任何数，其值始终为 0，所以效果等同于什么都不做，这也正是空指令 nop 的效果。所以 nop 指令不用特意实现，完全可以当作特殊的逻辑左移指令 sll。

33. ssnop

• 描述：一个特殊类型的空操作，此类指令在某些 MIPS CPU 中的 nop 指令的使用上有特殊的效果，可以防止与 nop 指令相关的潜在执行速度降低的问题。 在每个周期发射多条指令的 CPU 中，使用 ssnop 指令可以确保单独占用一个发射周期。OpenMIPS 设计为标量处理器，也就是每个周期发射一条指令，所以 ssnop 的作用与 nop 相同，可以按照 nop 指令的处理方式来处理 ssnop 指令。ssnop 指令的功能码是 6'b000000，与逻辑左移指令 sll 的功能码相同，sll 指令向 $0 寄存器保存移位结果，实际不会有任何效果，因为无论向 $0 写任何数，其值始终为 0，所以效果等同于什么都不做，这也正是空指令 ssnop 的效果。所以 ssnop 指令不用特意实现，完全可以当作特殊的逻辑左移指令 sll。

34. sync

• 描述：用于保证内存操作的完成，以确保在此指令之前的所有内存操作都已经正确完成，以确保数据的一致性。在多核心或多线程环境中，sync 指令用于在不同的处理器或线程之间同步内存操作。对于 OpenMIPS 而言，是严格按照指令顺序执行的，加载、存储操作也是按照顺序进行的，所以可以将 sync 指令当作 nop 指令处理，在这里将其归纳为空指令。

35. pref

• 描述：用于数据预取操作，此指令在 MIPS CPU 中用于预先将数据加载到缓存中，可以提升后续访问数据的速度。pref 指令可视为 nop 指令的增强版。OpenMIPS 没有实现缓存，所以也可以将 pref 指令当作 nop 指令处理，此处也将其归纳为空指令。

(3) 修改模块说明

1. id.v

由于 OpenMIPS 并没有实现 break 指令，故需要在其基础上增加 break 指令。需要有一个标志位来确定是否有断点异常，用 excepttype_is_break 来表示，并根据指令码进行译码。

```
`include "defines.vh"

module id(
    ...
);

    ...

    reg excepttype_is_break;    // 是否是调试断点指令 break

    ...

    // 新增：第 13 bit 表示 是否有断点异常 break
    assign excepttype_o = {18'b0, excepttype_is_break, excepttype_is_eret,
2'b0, instvalid, excepttype_is_syscall, 8'b0};

    ...

    always @ (*) begin
        if (rst == `RstEnable) begin
            ...
        end
        else begin
            ...
            excepttype_is_break      <= `False_v;      // 默认不是 break 指令
            ...

            case (op)
                `EXE_SPECIAL_INST: begin          // 指令码是 SPECIAL
                    case (op2)
                        5'b00000: begin
                            case (op3)            // 依据功能码判断是哪种指令
                                ...
                                `EXE_BREAK: begin              // break 指令
                                    wreg_o      <= `WriteDisable;
                                    aluop_o     <= `EXE_BREAK_OP;
                                    alusel_o    <= `EXE_RES_NOP;
```

```
                                reg1_read_o <= 1'b0;
                                reg2_read_o <= 1'b0;
                                instvalid   <= `InstValid;
                                excepttype_is_break <= `True_v;
                            end
                            default: begin
                            end
                        endcase
                    end
                    default: begin
                    end
                endcase
            end
            ...
        endcase    //case op
        ...
    end       //if
end         //always

...

endmodule
```

2. mem.v

在 mem 阶段最终确定异常的类型，此处也需要增加未实现的 break 指令，根据 MIPS32 架构，CP0 中的 Cause 寄存器第 2 到 6 位的 ExcCode 字段当异常为断点异常时，值为 9，据此修改 mem 模块。

```
module mem(
    ...
);

...

always @ (*) begin
    if(rst == `RstEnable) begin
        excepttype_o <= `ZeroWord;
    end
    else begin
        excepttype_o <= `ZeroWord;
        if(current_inst_address_i != `ZeroWord) begin
```

```
            if(((cp0_cause[15:8] & (cp0_status[15:8])) != 8'h00) &&
(cp0_status[1] == 1'b0) && (cp0_status[0] == 1'b1)) begin
                excepttype_o <= 32'h00000001;           // interrupt
            end
            ...
            else if(excepttype_i[13] == 1'b1) begin
                excepttype_o <= 32'h00000009;           // break
            end
        end
    end
end

...

endmodule
```

3. cp0_reg.v

同样地，在 CP0 模块，需要根据传入的异常类型来改写 CP0 中的部分寄存器，以新增对 break 指令的处理。

```
`include "defines.vh"

module cp0_reg(
        ...
    );

always @ (posedge clk) begin
    if(rst == `RstEnable) begin
        ...
    end
    else begin
        ...
        case (excepttype_i)
            ...
            32'h00000009: begin           // 断点异常指令 break
                if(status_o[1] == 1'b0) begin
                    if(is_in_delayslot_i == `InDelaySlot ) begin
                        epc_o       <= current_inst_addr_i - 4 ;
                        cause_o[31] <= 1'b1;
                    end
                    else begin
                        epc_o       <= current_inst_addr_i;
                        cause_o[31] <= 1'b0;
```

```
                    end
                end
                status_o[1]  <= 1'b1;
                cause_o[6:2] <= 5'b01001;
            end
        endcase
    end
end

...

endmodule
```

4. ctrl.v

OpenMIPS 中的中断例程起始地址是自定义的，与 MARS 中的不符，故在此需要修改中断例程起始地址为 0x00400004，从而保证最后输出结果符合预期。此外，还需增加 break 指令的中断例程地址，也为 0x00400004。

```
`include "defines.vh"

module ctrl(
    ...
);

always @ (*) begin
    if(rst == `RstEnable) begin
        ...
    end
    else if(excepttype_i != `ZeroWord) begin  // 不为 0，表示发生异常
        flush  <= 1'b1;
        stall  <= 6'b000000;
        case (excepttype_i)
            32'h00000001: begin              // 中断
                new_pc <= 32'h00000020;
            end
            32'h00000008: begin              // 系统调用异常 syscall
                new_pc <= 32'h00400004;      // 中断例程地址 （MARS）
            end
            32'h0000000a: begin              // 无效指令异常
                new_pc <= 32'h00400004;
            end
            32'h0000000d: begin              // 自陷异常
```

```
                new_pc <= 32'h00400004;
            end
            32'h0000000c: begin                 // 溢出异常
                new_pc <= 32'h00400004;
            end
            32'h0000000e: begin                 // 异常返回指令 eret
                new_pc <= cp0_epc_i;
            end
            32'h00000009: begin                 // 断点异常指令 break
                new_pc <= 32'h00400004;         // 自定义
            end
            default: begin
            end
        endcase
    end
    ...
end

endmodule
```

5. data_ram.v

由于 MARS 中的数据段是从 0x10010000 开始的，但是在 data_ram 的寄存器中，地址编址从 0 开始，故需要进行地址转换，将所有的地址减去 17'b0_0100_0000_0000_0000，这里的 17 位是由于真正有效的地址长度只有 17 位，理论上是 32 位，但是这里由于空间不足，将地址空间大大减少（测试程序里也确实没有用到那么多空间）。

其次，需要输出数据段地址空间的前四个字节作为七段数码管的输出，需要新增 output 端口去读取地址空间的第一个字。

```
`include "defines.vh"

module data_ram(
       input  wire                  clk,
       input  wire                  ce,         // 数据存储器使能信号
       input  wire                  we,        // 是否是写操作，为 1 表示是写操作
       input  wire[`DataAddrBus]     addr,       // 要访问的地址
       input  wire[3:0]             sel,        // 字节选择信号
       input  wire[`DataBus]         data_i,     // 要写入的数据
       output reg [`DataBus]         data_o,     // 读出的数据
       output [`DataBus]             seg7x16_data  // 输入给七段数码管的数据
);
```

```
        // 定义四个字节数组
        reg[`ByteWidth]  data_mem0[0:`DataMemNum - 1];
        reg[`ByteWidth]  data_mem1[0:`DataMemNum - 1];
        reg[`ByteWidth]  data_mem2[0:`DataMemNum - 1];
        reg[`ByteWidth]  data_mem3[0:`DataMemNum - 1];

        // mem3 表示模 4 余 0 的地址
        // mem2 表示模 4 余 1 的地址
        // mem1 表示模 4 余 2 的地址
        // mem0 表示模 4 余 3 的地址

        assign seg7x16_data = {data_mem3[0], data_mem2[0], data_mem1[0],
data_mem0[0]};

        // 写操作
         always @ (posedge clk) begin
             if (ce == `ChipDisable) begin
             // data_o <= ZeroWord;
             end
             else if(we == `WriteEnable) begin
                 if (sel[3] == 1'b1) begin // 这不是数组！！！！！！这是左边第
一位！！！！  3.2.1.0
          data_mem3[addr[`DataMemNumLog2 + 1:2] - 17'b0_0100_0000_0000_0000]
<= data_i[31:24]; // 除以 4 // 低地址
                 end
                 if (sel[2] == 1'b1) begin
          data_mem2[addr[`DataMemNumLog2 + 1:2] - 17'b0_0100_0000_0000_0000]
<= data_i[23:16];
                 end
                 if (sel[1] == 1'b1) begin
          data_mem1[addr[`DataMemNumLog2 + 1:2] - 17'b0_0100_0000_0000_0000]
<= data_i[15:8];
                 end
                 if (sel[0] == 1'b1) begin
          data_mem0[addr[`DataMemNumLog2 + 1:2] - 17'b0_0100_0000_0000_0000]
<= data_i[7:0];
                 end
             end
         end

         // 读操作 读出 0 1 2 3 地址的内容
         always @ (*) begin
             if (ce == `ChipDisable) begin
                 data_o <= `ZeroWord;
```


装 订 线

```
            end
            else if(we == `WriteDisable) begin // 低 → 高 0 1 2 3
                data_o <=
{data_mem3[addr[`DataMemNumLog2 + 1:2] - 17'b0_0100_0000_0000_0000],
 data_mem2[addr[`DataMemNumLog2 + 1:2] - 17'b0_0100_0000_0000_0000],
 data_mem1[addr[`DataMemNumLog2 + 1:2] - 17'b0_0100_0000_0000_0000],
 data_mem0[addr[`DataMemNumLog2 + 1:2] - 17'b0_0100_0000_0000_0000]};
       // - 17'b0_0100_0000_0000_0000 是因为 MARS 数据段是从 0x10010000 开始的
        // 17'b0_0100_0000_0000_0000 是 17'b1_0000_0000_0000_0000（0x10000）
右移两位（除以 4）得到的
        // 相当于是 19 位地址除以 4 得到 17 位地址（寄存器地址），最多是 19 位地址
        // 所以 MARS 中的 32 位地址
（32'b0001_0000_0000_0001_0000_0000_0000_0000）没有全部用上
        // 只用了 17'b1_0000_0000_0000_0000（0x10000）
            end
            else begin
                data_o <= `ZeroWord;
            end
        end

endmodule
```

6. seg7x16.v

将指定的 0x10010000 内存单元最终结果显示在数码管上，实现沿用 MIPS54 的七段数码管驱动模块。

```
`timescale 1ns / 1ns

module Seg7x16(
    input Clk,
    input Reset,
    input Cs,
    input [31:0] I_Data,
    output [7:0] O_Seg,
    output [7:0] O_Sel
    );

    reg [14:0] Cnt;
    always @ (posedge Clk, posedge Reset)
    if (Reset)
        Cnt <= 0;
    else
        Cnt <= Cnt + 1'B1;
```

```
wire Seg7_Clk = Cnt[14];

reg [2:0] Seg7_Addr;

always @ (posedge Seg7_Clk, posedge Reset)
if(Reset)
    Seg7_Addr <= 0;
else
    Seg7_Addr <= Seg7_Addr + 1'B1;

reg [7:0] O_Sel_R;

always @ (*)
    case(Seg7_Addr)
        7 : O_Sel_R = 8'B01111111;
        6 : O_Sel_R = 8'B10111111;
        5 : O_Sel_R = 8'B11011111;
        4 : O_Sel_R = 8'B11101111;
        3 : O_Sel_R = 8'B11110111;
        2 : O_Sel_R = 8'B11111011;
        1 : O_Sel_R = 8'B11111101;
        0 : O_Sel_R = 8'B11111110;
    endcase

reg [31:0] I_Data_Store;
always @ (posedge Clk, posedge Reset)
if(Reset)
    I_Data_Store <= 0;
else if(Cs)
    I_Data_Store <= I_Data;

reg [7:0] Seg_Data_R;
always @ (*)
    case(Seg7_Addr)
        0 : Seg_Data_R = I_Data_Store[3:0];
        1 : Seg_Data_R = I_Data_Store[7:4];
        2 : Seg_Data_R = I_Data_Store[11:8];
        3 : Seg_Data_R = I_Data_Store[15:12];
        4 : Seg_Data_R = I_Data_Store[19:16];
        5 : Seg_Data_R = I_Data_Store[23:20];
        6 : Seg_Data_R = I_Data_Store[27:24];
        7 : Seg_Data_R = I_Data_Store[31:28];
    endcase
```


装 订 线

```
    reg [7:0] O_Seg_R;
    always @ (posedge Clk, posedge Reset)
    if(Reset)
        O_Seg_R <= 8'Hff;
    else
        case(Seg_Data_R)
            4'H0 : O_Seg_R <= 8'HC0;
            4'H1 : O_Seg_R <= 8'HF9;
            4'H2 : O_Seg_R <= 8'HA4;
            4'H3 : O_Seg_R <= 8'HB0;
            4'H4 : O_Seg_R <= 8'H99;
            4'H5 : O_Seg_R <= 8'H92;
            4'H6 : O_Seg_R <= 8'H82;
            4'H7 : O_Seg_R <= 8'HF8;
            4'H8 : O_Seg_R <= 8'H80;
            4'H9 : O_Seg_R <= 8'H90;
            4'HA : O_Seg_R <= 8'H88;
            4'HB : O_Seg_R <= 8'H83;
            4'HC : O_Seg_R <= 8'HC6;
            4'HD : O_Seg_R <= 8'HA1;
            4'HE : O_Seg_R <= 8'H86;
            4'HF : O_Seg_R <= 8'H8E;
        endcase

     assign O_Sel = O_Sel_R;
     assign O_Seg = O_Seg_R;

endmodule
```

7. openmips_min_sopc.v

顶层模块中需要实现分频器，保证数码管的显示变化可以被人眼捕捉，其中七段数码管的时钟是原始时钟，CPU 和数据存储器的时钟都是放慢过的。

```
`include "defines.vh"

module openmips_min_sopc(
    input wire    clk,
    input wire    rst,
    output [7:0]  O_Seg,
    output [7:0]  O_Sel
);
```

```
// 连接指令存储器
wire[`InstAddrBus]  inst_addr;
wire[`InstBus]      inst;
wire                rom_ce;

wire                mem_we_i;
wire[`RegBus]       mem_addr_i;
wire[`RegBus]       mem_data_i;
wire[`RegBus]       mem_data_o;
wire[3:0]           mem_sel_i;
wire                mem_ce_i;

wire[5:0]           int;
wire                timer_int;

wire[`DataBus]      Seg7_In;            // 显示在数码管里面的内容

assign int = {5'b00000, timer_int};     // 时钟中断输出作为一个中断输入

reg clk_100;
initial clk_100 = 0;
integer cnt = 0;

// 分频器：clk n 个周期 clk_100 才一个周期
always @(posedge clk) begin
    if (cnt < 100) begin
        cnt = cnt + 1;
    end
    else begin
        cnt = 0;
        clk_100 = ~clk_100;
    end
end

// 例化处理器 OpenMIPS
openmips openmips0(
    .clk(clk_100),
    .rst(rst),
    .rom_addr_o(inst_addr),
    .rom_data_i(inst),
    .rom_ce_o(rom_ce),

    .int_i(int),                // 中断输入
```

```
        .ram_we_o(mem_we_i),
        .ram_addr_o(mem_addr_i),
        .ram_sel_o(mem_sel_i),
        .ram_data_o(mem_data_i),
        .ram_data_i(mem_data_o),
        .ram_ce_o(mem_ce_i),

        .timer_int_o(timer_int)     // 时钟中断输出
    );

    // 例化指令存储器 ROM
    inst_rom inst_rom0(
        .ce(rom_ce),
        .addr(inst_addr),
        .inst(inst)
    );

    data_ram data_ram0(
        .clk(clk_100),
        .we(mem_we_i),
        .addr(mem_addr_i),
        .sel(mem_sel_i),
        .data_i(mem_data_i),
        .data_o(mem_data_o),
        .ce(mem_ce_i),
        .seg7x16_data(Seg7_In)
    );

    Seg7x16 seg7x16_uut(
        .Clk(clk),
        .Reset(rst),
        .Cs(1'b1),
        .I_Data(Seg7_In),
        .O_Seg(O_Seg),
        .O_Sel(O_Sel)
    );

endmodule
```

8. top.xdc 约束文件

将七段数码管的片选信号、时钟、复位信号绑定到开发板的固定端口，并添加时钟约束。

```
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[7]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports rst]
set_property PACKAGE_PIN E3 [get_ports clk]
set_property PACKAGE_PIN N17 [get_ports rst]
set_property PACKAGE_PIN T10 [get_ports {O_Seg[0]}]
set_property PACKAGE_PIN R10 [get_ports {O_Seg[1]}]
set_property PACKAGE_PIN K16 [get_ports {O_Seg[2]}]
set_property PACKAGE_PIN K13 [get_ports {O_Seg[3]}]
set_property PACKAGE_PIN P15 [get_ports {O_Seg[4]}]
set_property PACKAGE_PIN T11 [get_ports {O_Seg[5]}]
set_property PACKAGE_PIN L18 [get_ports {O_Seg[6]}]
set_property PACKAGE_PIN H15 [get_ports {O_Seg[7]}]
set_property PACKAGE_PIN J17 [get_ports {O_Sel[0]}]
set_property PACKAGE_PIN J18 [get_ports {O_Sel[1]}]
set_property PACKAGE_PIN T9 [get_ports {O_Sel[2]}]
set_property PACKAGE_PIN J14 [get_ports {O_Sel[3]}]
set_property PACKAGE_PIN P14 [get_ports {O_Sel[4]}]
set_property PACKAGE_PIN T14 [get_ports {O_Sel[5]}]
set_property PACKAGE_PIN K2 [get_ports {O_Sel[6]}]
set_property PACKAGE_PIN U13 [get_ports {O_Sel[7]}]
create_clock -period 200.000 -name clk_pin -waveform {0.000 100.000}
[get_ports clk]
set_input_delay -clock [get_clocks *] 1.000 [get_ports rst]
set_output_delay -clock [get_clocks *] 0.000 [get_ports -filter { NAME =~
"*" && DIRECTION == "OUT" }]
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets rst_IBUF]
```

(4) 模块说明

1. PC 寄存器模块

- 功能：存储当前指令的地址，并根据控制信号和指令流水线的状态更新 PC 寄存器的值。
- 模块设计

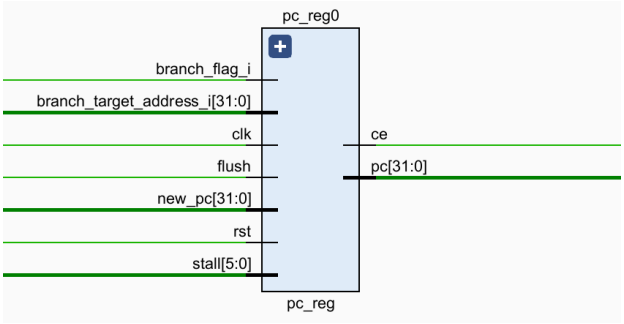

图 3：PC 寄存器模块设计图

- 接口定义

```
module pc_reg(
    input  wire             clk,
    input  wire             rst,
    input  wire[5:0]        stall, // 来自控制模块 ctrl

    // 来自译码阶段 ID 模块的信息
    input wire              branch_flag_i,
    input wire[`RegBus]     branch_target_address_i,

    input wire              flush,   // 流水线清除信号
    input wire[`RegBus]     new_pc,  // 异常处理例程入口地址

    output reg[`InstAddrBus] pc,
    output reg              ce   // 指令存储器使能信号
);
```

2. IF 模块

- 功能：取指模块，主要包含指令寄存器等。
- 模块设计（见图 4）

装 订 线

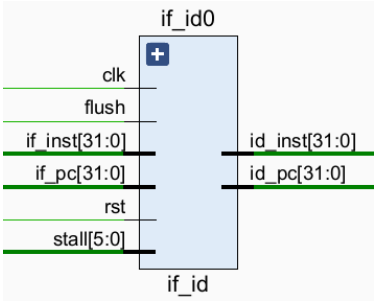

图 4：if_id 模块设计图

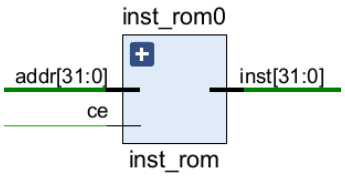

图 5：inst_rom 模块设计图

• 接口定义

```
module if_id(
    input wire                  clk,
    input wire                  rst,

    // 来自取指阶段的信号，其中宏定义 InstBus 表示指令宽度，为 32
    input wire[`InstAddrBus]     if_pc,
    input wire[`InstBus]         if_inst,

    input wire[5:0]              stall,

    input wire                  flush,

    // 对应译码阶段的信号
    output reg[`InstAddrBus]     id_pc,
    output reg[`InstBus]         id_inst
);
```

```
module inst_rom(
    input  wire                 ce,
    input  wire[`InstAddrBus] addr,
    output reg [`InstBus]     inst
);
```

3. ID 模块

• 功能：主要包含控制单元模块（负责控制整体流水线的状态，处理流水线暂停、流水线排

空，并在指令地址跳转时产生相应的延迟槽和新 PC 地址）、通用寄存器堆等。

• 模块设计

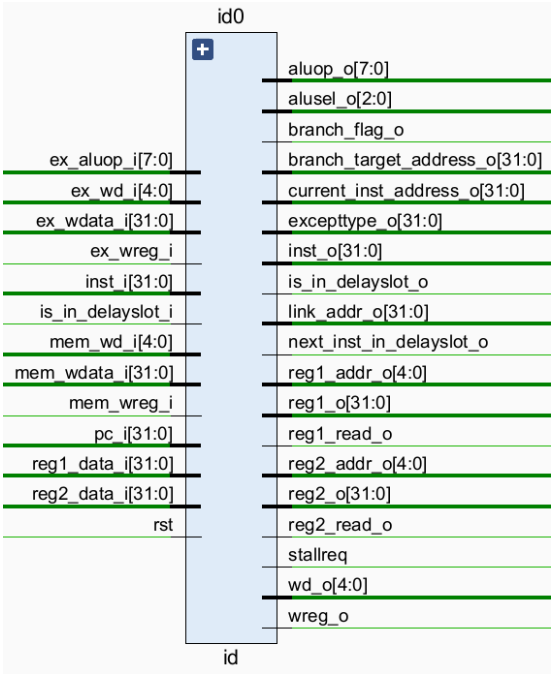

图 6：id 模块设计图

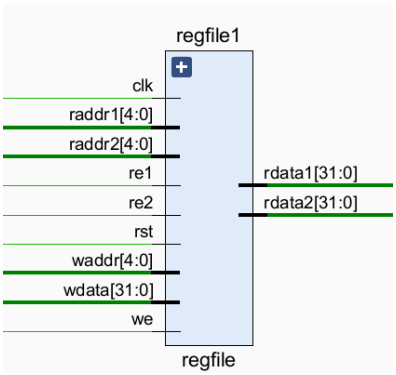

图 7：regfile 模块设计图

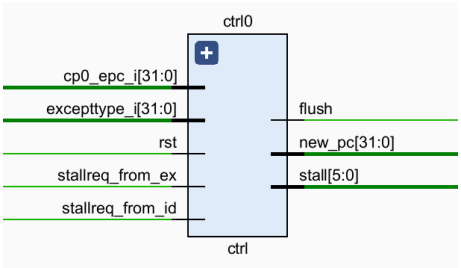

图 8：控制器模块设计图

• 接口定义

```
module id(
    input wire                      rst,
```


装 订 线

```
    input wire[`InstAddrBus]       pc_i,
    input wire[`InstBus]           inst_i,

    // 处于执行阶段的指令的一些信息，用于解决 load 相关
    input wire[`AluOpBus]          ex_aluop_i,

    // 读取的 Regfile 的值
    input wire[`RegBus]            reg1_data_i,
    input wire[`RegBus]            reg2_data_i,

    /* 解决先写后读 */
    // 处于执行阶段的指令的运算结果
    input wire                     ex_wreg_i,
    input wire[`RegBus]            ex_wdata_i,
    input wire[`RegAddrBus]        ex_wd_i,

    // 处于访存阶段的指令的运算结果
    input wire                     mem_wreg_i,
    input wire[`RegBus]            mem_wdata_i,
    input wire[`RegAddrBus]        mem_wd_i,

    // 如果上一条指令是转移指令，那么下一条指令进入译码阶段的时候，输入变量
    // is_in_delayslot_i 为 true，表示是延迟槽指令，反之，为 false
    input wire                     is_in_delayslot_i,

    output reg                     next_inst_in_delayslot_o,

    output reg                     branch_flag_o,              // 是否发生转移
    output reg[`RegBus]            branch_target_address_o,    // 转移到的目标地址
    output reg[`RegBus]  link_addr_o,                // 转移指令要保存的返回地址
    output reg   is_in_delayslot_o,       // 当前处于译码阶段的指令是否位于延迟槽

    // 输出到 Regfile 的信息
    output reg                     reg1_read_o,
    output reg                     reg2_read_o,
    output reg[`RegAddrBus]        reg1_addr_o,
    output reg[`RegAddrBus]        reg2_addr_o,

    // 送到执行阶段的信息
    output reg[`AluOpBus]          aluop_o,
    output reg[`AluSelBus]         alusel_o,
    output reg[`RegBus]            reg1_o,         // 操作数的值
    output reg[`RegBus]            reg2_o,
    output reg[`RegAddrBus]        wd_o,           // 要写入的目的寄存器地址
```

```
    output reg                      wreg_o,         // 是否有要写入的目的寄存器

    output wire[`RegBus]            inst_o,         // 用于存取指令

    output wire[31:0]               excepttype_o,             // 收集的异常信息
    output wire[`RegBus]     current_inst_address_o,  // 译码阶段指令的地址

    output wire                     stallreq
);
```

```
module regfile(
    input  wire              clk,
    input  wire              rst,

    // 写端口
    input  wire              we,
    input  wire[`RegAddrBus]  waddr,
    input  wire[`RegBus]      wdata,

    // 读端口 1
    input  wire              re1,
    input  wire[`RegAddrBus]  raddr1,
    output reg [`RegBus]      rdata1,
    // 读端口 2
    input  wire              re2,
    input  wire[`RegAddrBus]  raddr2,
    output reg [`RegBus]      rdata2
);
```

```
module ctrl(
    input wire          rst,
    input wire          stallreq_from_id,    // 来自译码阶段的暂停请求
    input wire          stallreq_from_ex,    // 来自执行阶段的暂停请求

    // 来自 MEM
    input wire[31:0]    excepttype_i,
    input wire[`RegBus] cp0_epc_i,

    output reg[`RegBus] new_pc,              // 异常处理入口地址
    output reg          flush,               // 是否清除流水线

    output reg[5:0]     stall
);
```

4. EX 模块

• 功能：执行模块（给出后续单元对 HILO 寄存器、 CP0 寄存器和转移状态、转移地址的控制。同时，EX 处理器还负责检测后续状态中存在的数据相关问题，并进行相应处理），包含除法器模块。

• 模块设计

图 9：ex 模块设计图

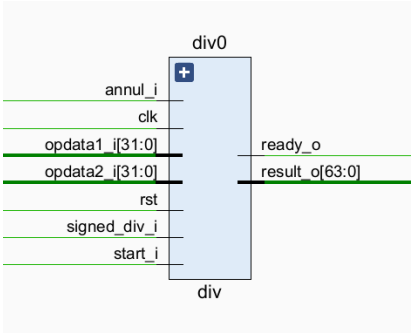

图 10：除法器模块设计图

• 接口定义

```
module ex(
    input wire                  rst,

    input wire[31:0]            excepttype_i,
    input wire[`RegBus]         current_inst_address_i,
```

```
// 译码阶段送到执行阶段的信息
input wire[`AluOpBus]      aluop_i,
input wire[`AluSelBus]     alusel_i,
input wire[`RegBus]        reg1_i,
input wire[`RegBus]        reg2_i,
input wire[`RegAddrBus]    wd_i,
input wire                 wreg_i,

// HILO 模块给出的 HI、LO 寄存器的值
input wire[`RegBus]           hi_i,
input wire[`RegBus]           lo_i,

// 回写阶段的指令是否要写 HI、LO，用于检测 HI、LO 寄存器带来的数据相关问题
input wire[`RegBus]           wb_hi_i,
input wire[`RegBus]           wb_lo_i,
input wire                    wb_whilo_i,

// 访存阶段的指令是否要写 HI、LO，用于检测 HI、LO 寄存器带来的数据相关问题
input wire[`RegBus]           mem_hi_i,
input wire[`RegBus]           mem_lo_i,
input wire                    mem_whilo_i,

input wire[`DoubleRegBus]     hilo_temp_i,  // 第一个执行周期得到的乘法结果
input wire[1:0]               cnt_i,      // 当前处于执行阶段的第几个时钟周期

input wire[`DoubleRegBus]     div_result_i,  // 来自除法模块的输入
input wire                    div_ready_i,

// 处于执行阶段的转移指令要保存的返回地址
input wire[`RegBus]           link_address_i,

// 当前执行阶段的指令是否位于延迟槽
input wire                    is_in_delayslot_i,

// 当前处于执行阶段的指令
input wire[`RegBus]           inst_i,

// 访存阶段的指令是否要写 CP0 中的寄存器，用来检测数据相关
input wire                    mem_cp0_reg_we,
input wire[4:0]               mem_cp0_reg_write_addr,
input wire[`RegBus]           mem_cp0_reg_data,

// 回写阶段的指令是否要写 CP0 中的寄存器，也是用来检测数据相关
```


装 订 线

```
    input wire                      wb_cp0_reg_we,
    input wire[4:0]                 wb_cp0_reg_write_addr,
    input wire[`RegBus]             wb_cp0_reg_data,

    // 与 CP0 直接相连，用于读取其中指定寄存器的值
    input wire[`RegBus]             cp0_reg_data_i,
    output reg[4:0]                 cp0_reg_read_addr_o,

    // 向流水线下一级传递，用于写 CP0 中的指定寄存器
    output reg                      cp0_reg_we_o,
    output reg[4:0]                 cp0_reg_write_addr_o,
    output reg[`RegBus]             cp0_reg_data_o,

    // 处于执行阶段的指令对 HI、LO 寄存器的写操作请求
    output reg[`RegBus]             hi_o,
    output reg[`RegBus]             lo_o,
    output reg                      whilo_o,

    // 执行的结果
    output reg[`RegAddrBus]     wd_o,
    output reg                  wreg_o,
    output reg[`RegBus]         wdata_o,

    output reg[`DoubleRegBus]   hilo_temp_o,
    output reg[1:0]             cnt_o,

    // 到除法模块的输出
    output reg[`RegBus]         div_opdata1_o,
    output reg[`RegBus]         div_opdata2_o,
    output reg                  div_start_o,
    output reg                  signed_div_o,

    // load / store
    output wire[`AluOpBus]      aluop_o,
    output wire[`RegBus]        mem_addr_o,
    output wire[`RegBus]        reg2_o,

    output wire[31:0]       excepttype_o,   // 译码阶段、执行阶段收集到的异常信息
    output wire                 is_in_delayslot_o,
    output wire[`RegBus]        current_inst_address_o,

    output reg                  stallreq
);
```


装 订 线

```
module div(
    input  wire             clk,
    input  wire             rst,
    input  wire             signed_div_i, // 是否有符号除法，为 1 表示有符号除法
    input  wire[31:0]       opdata1_i,     // 被除数
    input  wire[31:0]       opdata2_i,     // 除数
    input  wire             start_i,       // 是否开始除法运算
    input  wire             annul_i,  // 是否取消除法运算，为 1 表示取消除法运算
    output reg[63:0]        result_o,      // 除法运算结果
    output reg              ready_o        // 除法运算是否结束
);
```

5. MEM 模块

- 功能：访存模块，主要包含数据存储器等模块。
- 模块设计

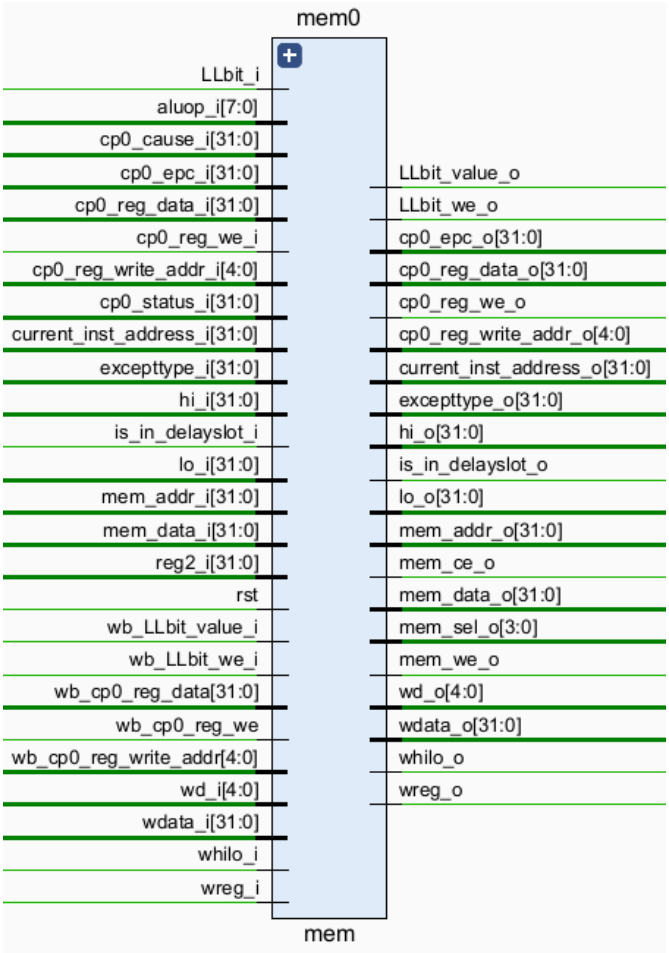

图 11：mem 模块设计图

装 订 线

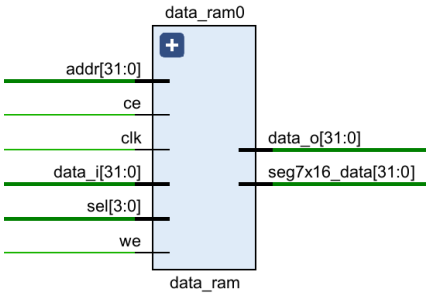

图 12：数据存储器模块设计图

• 接口定义

```
module mem(
    input wire              rst,

    // 来自执行阶段的信息
    input wire[`RegAddrBus] wd_i,
    input wire              wreg_i,
    input wire[`RegBus]     wdata_i,
    input wire[`RegBus]     hi_i,
    input wire[`RegBus]     lo_i,
    input wire              whilo_i,

    input wire[`AluOpBus]   aluop_i,      // 访存阶段的指令要进行的运算的子类型
    input wire[`RegBus]  mem_addr_i, // 访存阶段的加载、存储指令对应的存储器地址
    input wire[`RegBus]     reg2_i,       // 访存阶段的存储指令要存储的数据，或者
lwl、lwr 指令要写入的目的寄存器的原始值

    // 来自外部数据存储器 RAM 的信息，从数据存储器读取的数据
    input wire[`RegBus]   mem_data_i,  // 存储器是 32 位地址，每个地址存 1 字节

    input wire              LLbit_i,     // LLbit 模块给出的 LLbit 寄存器的值
    input wire           wb_LLbit_we_i,  // 回写阶段的指令是否要写 LLbit 寄存器
    input wire            wb_LLbit_value_i,  // 回写阶段要写入 LLbit 寄存器的值

    input wire              cp0_reg_we_i,
    input wire[4:0]         cp0_reg_write_addr_i,
    input wire[`RegBus]     cp0_reg_data_i,

    // 来自执行阶段
    input wire[31:0]        excepttype_i,
    input wire              is_in_delayslot_i,
    input wire[`RegBus]     current_inst_address_i,

    // 来自 CP0 模块
```

```
    input wire[`RegBus]     cp0_status_i,
    input wire[`RegBus]     cp0_cause_i,
    input wire[`RegBus]     cp0_epc_i,

    // 来自回写阶段，是回写阶段的指令对 CP0 中寄存器的写信息，
    // 用来检测数据相关
    input wire              wb_cp0_reg_we,
    input wire[4:0]         wb_cp0_reg_write_addr,
    input wire[`RegBus]     wb_cp0_reg_data,

    // 访存阶段的结果
    output reg[`RegAddrBus] wd_o,
    output reg              wreg_o,
    output reg[`RegBus]     wdata_o,
    output reg[`RegBus]     hi_o,
    output reg[`RegBus]     lo_o,
    output reg              whilo_o,

    // 送到外部数据存储器 RAM 的信息
    output reg[`RegBus]     mem_addr_o,  // 要访问的数据存储器的地址
    output wire             mem_we_o,    // 是否是写操作，为 1 表示是写操作
    output reg[3:0]         mem_sel_o,   // 字节选择信号
    output reg[`RegBus]     mem_data_o,  // 要写入数据存储器的数据
    output reg              mem_ce_o,    // 数据存储器使能信号
    output reg           LLbit_we_o,     // 访存阶段的指令是否要写 LLbit 寄存器
    output reg         LLbit_value_o,  // 访存阶段的指令要写入 LLbit 寄存器的值

    output reg              cp0_reg_we_o,
    output reg[4:0]         cp0_reg_write_addr_o,
    output reg[`RegBus]     cp0_reg_data_o,

    output reg[31:0]        excepttype_o,       // 最终的异常类型
    output wire[`RegBus]    cp0_epc_o,          // CP0 中 EPC 寄存器的最新值
    output wire          is_in_delayslot_o,  // 访存阶段的指令是否是延迟槽指令

    output wire[`RegBus]    current_inst_address_o // 访存阶段指令的地址
);
```

```
module data_ram(
       input  wire                  clk,
       input  wire                  ce,        // 数据存储器使能信号
       input  wire                  we,     // 是否是写操作，为 1 表示是写操作
       input  wire[`DataAddrBus]     addr,      // 要访问的地址
       input  wire[3:0]             sel,       // 字节选择信号
```


装 订 线

```
        input  wire[`DataBus]          data_i,     // 要写入的数据
        output reg [`DataBus]          data_o,     // 读出的数据
        output [`DataBus]              seg7x16_data  // 输入给七段数码管的数据
);
```

6. WB 模块

- 功能：回写模块，主要包含 HI、LO 寄存器、CP0 寄存器、LLbit 寄存器等模块。
- 模块设计

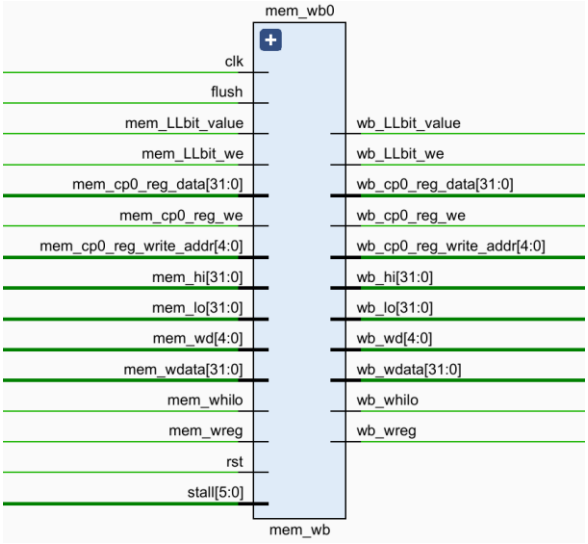

图 13：wb 模块设计图

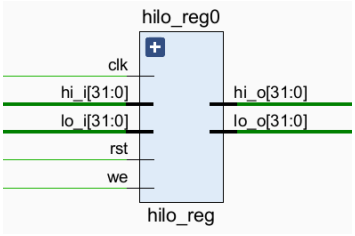

图 14：HI/LO 寄存器模块设计图

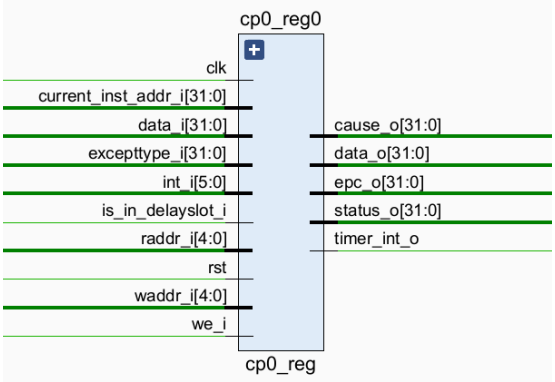

图 15：CP0 模块设计图

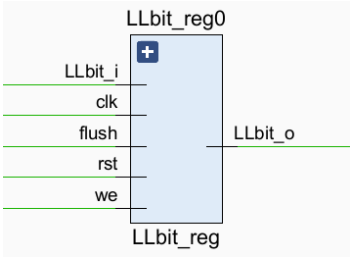

图 16：LLbit 寄存器模块设计图

• 接口定义

```
module mem_wb(
    input wire                  clk,
    input wire                  rst,

    // 访存阶段的结果
    input wire[`RegAddrBus] mem_wd,
    input wire                  mem_wreg,
    input wire[`RegBus]     mem_wdata,
    input wire[`RegBus]     mem_hi,
    input wire[`RegBus]     mem_lo,
    input wire                  mem_whilo,

    input wire[5:0]         stall,       // 来自控制模块的信息

    input wire                  flush,       // 流水线清除信号

    input wire              mem_LLbit_we, // 访存阶段的指令是否要写 LLbit 寄存器
    input wire       mem_LLbit_value,  // 访存阶段的指令要写入 LLbit 寄存器的值

    input wire                  mem_cp0_reg_we,
    input wire[4:0]         mem_cp0_reg_write_addr,
    input wire[`RegBus]     mem_cp0_reg_data,

    // 送到回写阶段的信息
    output reg[`RegAddrBus] wb_wd,
    output reg                  wb_wreg,
    output reg[`RegBus]     wb_wdata,
    output reg[`RegBus]     wb_hi,
    output reg[`RegBus]     wb_lo,
    output reg                  wb_whilo,

    output reg                  wb_LLbit_we, // 回写阶段的指令是否要写 LLbit 寄存器
    output reg      wb_LLbit_value,    // 回写阶段的指令要写入 LLbit 寄存器的值

    output reg                  wb_cp0_reg_we,
```

```
    output reg[4:0]         wb_cp0_reg_write_addr,
    output reg[`RegBus]     wb_cp0_reg_data
);
```

```
module hilo_reg(
        input wire              clk,
        input wire              rst,

        // 写端口
        input wire              we,
        input wire[`RegBus]     hi_i,
        input wire[`RegBus]     lo_i,

        // 读端口
        output reg[`RegBus]     hi_o,
        output reg[`RegBus]     lo_o
);
```

```
module cp0_reg(
        input wire                  clk,
        input wire                  rst,

        input wire                  we_i,           // 是否要写 CP0 中的寄存器
        input wire[4:0]             waddr_i,        // 要写的 CP0 中寄存器的地址
        input wire[4:0]             raddr_i,      // 要读取的 CP0 中寄存器的地址
        input wire[`RegBus]         data_i,         // 要写入 CP0 中寄存器的数据
        input wire[5:0]             int_i,          // 6 个外部硬件中断输入

        input wire[31:0]            excepttype_i,
        input wire[`RegBus]         current_inst_addr_i,
        input wire                  is_in_delayslot_i,

        output reg[`RegBus]         data_o,       // 读出的 CP0 中某个寄存器的值
        output reg[`RegBus]         count_o,        // Count 寄存器的值
        output reg[`RegBus]         compare_o,      // Compare 寄存器的值
        output reg[`RegBus]         status_o,       // Status 寄存器的值
        output reg[`RegBus]         cause_o,        // Cause 寄存器的值
        output reg[`RegBus]         epc_o,          // EPC 寄存器的值
        output reg[`RegBus]         config_o,       // Config 寄存器的值
        output reg[`RegBus]         prid_o,         // PRId 寄存器的值

        output reg                  timer_int_o     // 是否有定时中断发生
    );
```


装 订 线

```
module LLbit_reg(
        input wire      clk,
        input wire      rst,

        // 异常是否发生，为 1 表示异常发生，为 0 表示没有异常
        input wire      flush,

        // 写操作
        input wire      LLbit_i,
        input wire      we,

        // LLbit 寄存器的值
        output reg      LLbit_o
);
```

4、实验结果

(1) 测试与调试方法

本次实验采用 Vivado 2022.2 自带仿真工具进行前后仿真，观察波形变化。采用的测试 testbench 文件实现如下：

```
`include "defines.vh"

// 时间单位是 1ns，精度是 1ps
`timescale 1ns/1ps

module openmips_min_sopc_tb();
    reg     CLOCK_50;
    reg     rst;

    // 每隔 10ns，CLOCK_50 信号翻转一次，所以一个周期是 20ns，对应 50MHz
    initial begin
        CLOCK_50 = 1'b0;
        forever #10 CLOCK_50 = ~CLOCK_50;
    end

    // 最初时刻，复位信号有效，在第 195ns，复位信号无效，最小 SOPC 开始运行
    // 运行 1000ns 后，暂停仿真
    initial begin
        rst = `RstEnable;
        #195 rst= `RstDisable;
//        #1000 $stop;
```

```
    end

    // 例化最小 SOPC
    openmips_min_sopc openmips_min_sopc0(
        .clk(CLOCK_50),
        .rst(rst)
    );

endmodule
```

在调试 Debug 时，可以利用 Vivado 自带的仿真器，将所需信号右键“Add to Wave Window”查看相应的波形图，也可以在相应的.v 文件中鼠标悬停查看当前变量的值。

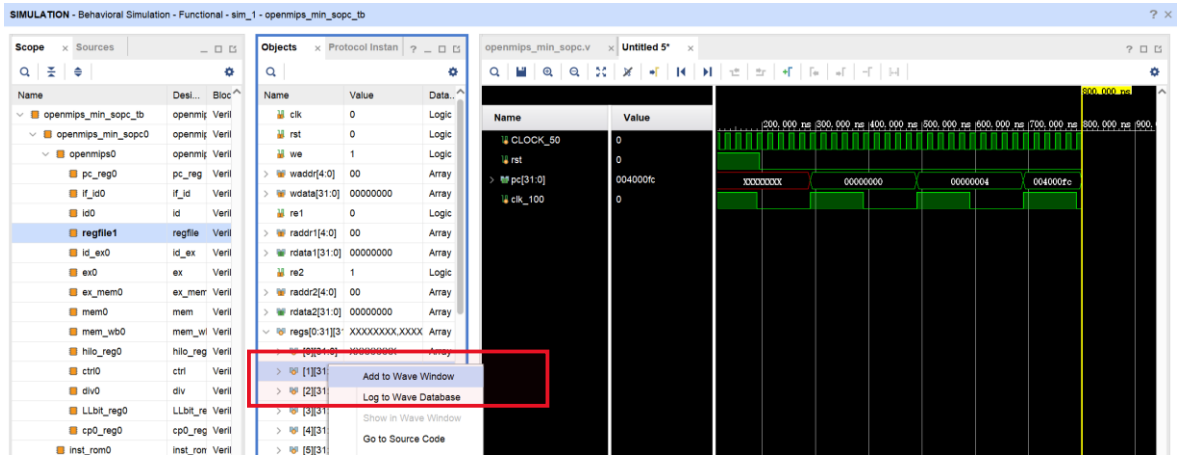

图 17：调试方法 1

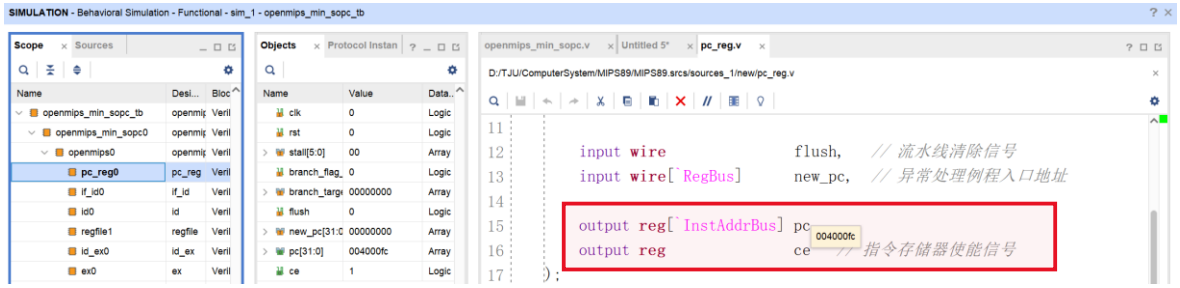

图 18：调试方法 2

(2) 仿真验证与分析

测试指令由老师给出的汇编文件 测试程序(mips_89_mars_board_big).s 经过 MARS 进行反汇编生成 COE 文件，注意此处直接将 COE 文件视作文本文件，在 inst_rom.v 中指定文本文件绝对路径，将指令读取出之后存储于 inst_rom 当中，在测试时将指令逐条取出即可。以下进行前仿真测试。

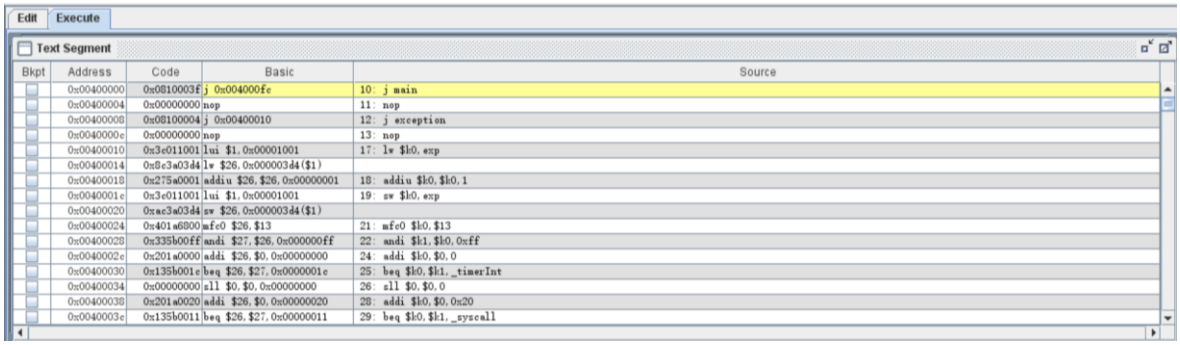

图 19：MARS 进行反汇编

首先观察指令读取情况：可以观察到指令可以正常读取，指令地址一次递增，取指正常。

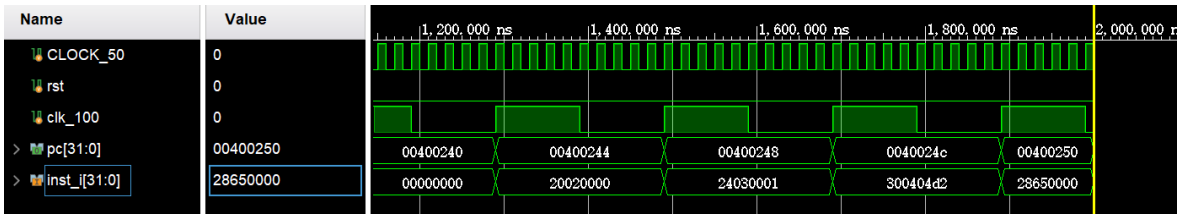

图 20：指令读取情况波形图

其次观察流经各个部件的指令和相关信号，可以观察到流水线能够正常流动。

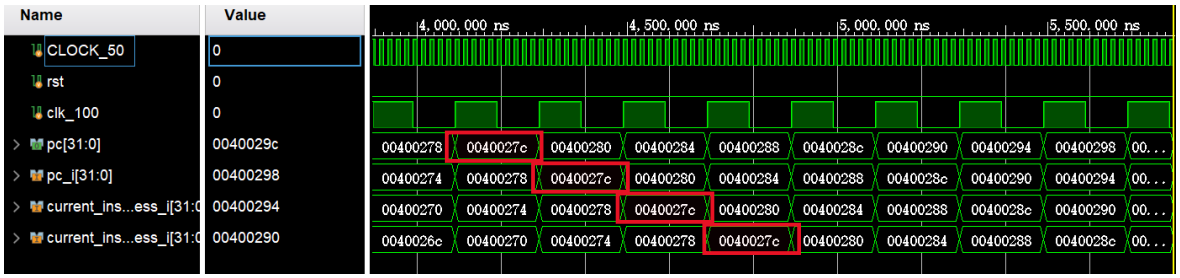

图 21：流水线流动波形图

当执行完 11 个测试函数后，结果正确，DMEM 的 0 号单元低位为 0x0001，然后程序开始测试时钟中断，可以看到，DMEM 的 0 号单元高位不断增加，表示时钟中断次数的计数。

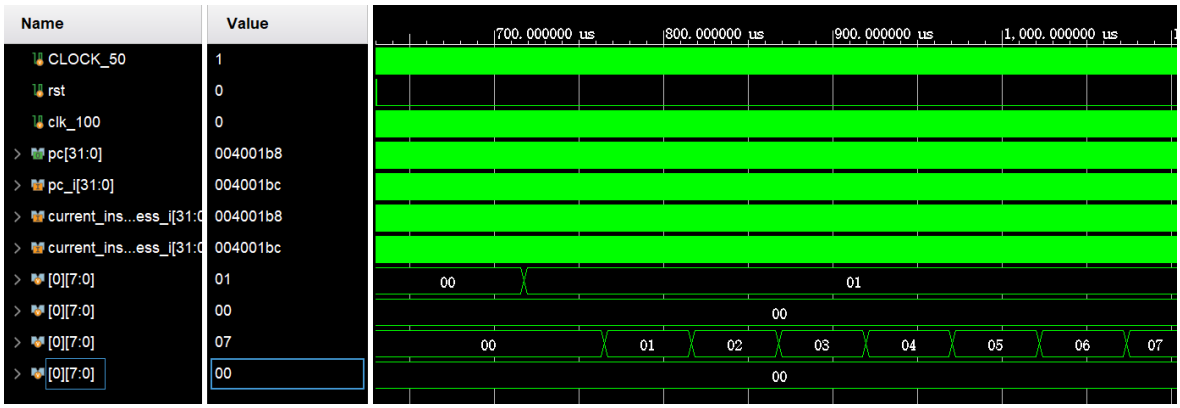

图 22：测试结果波形图

装 订 线

(3) 下板验证

在下板验证之前，先进行时序验证，结果如下：

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	187.440 ns	Worst Hold Slack (WHS):	0.107 ns	Worst Pulse Width Slack (WPWS):	99.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	148	Total Number of Endpoints:	148	Total Number of Endpoints:	88

All user specified timing constraints are met.

图 23：时序验证结果

在 200ns 的时序约束下，WorstNegativeSlack 为 187.440 ns，WorstHoldSlack (WHS) 为 0.107 ns，Worst Pulse Width Slack (WPWS) 为 99.500 ns，都满足了时序要求，这代表本次实验设计的 CPU 在时序约束下不会发生功能性问题。

下板采用的 CPU 频率为 50Mhz，实验结果如下：

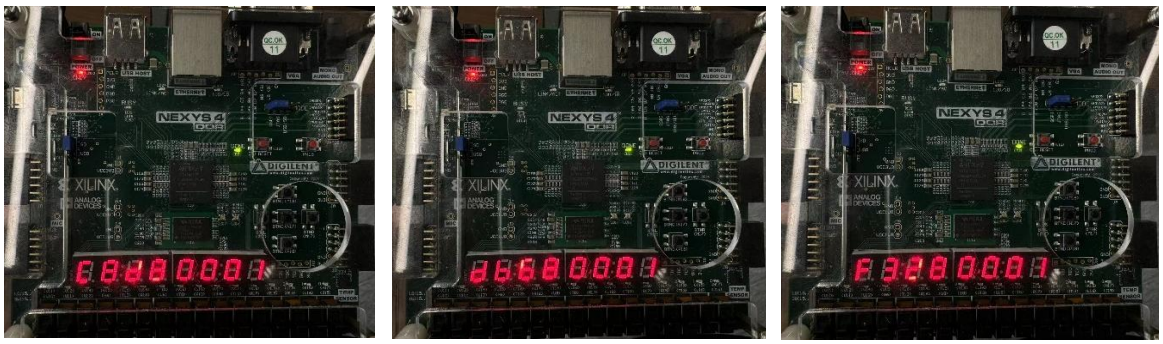

图 24：下板结果演示

可以看到数码管低半字显示 0x0001，高半字随着时钟中断而计数，验证结果正确，这证明了本次试验设计的 CPU 可以在 50Mhz 下平稳运行，在完成 54 条的前提下很好地完成了 HILO 交互、拓展内存读写、CPO 读写在内的一系列任务和中断处理任务，为之后的系统移植打下坚实的基础。

5、实验总结

(1) 遇到的问题

1. break 指令的异常处理例程跳转存疑

通过查询相关资料，包括《自己动手写 CPU》和往届小端模式的测试程序，发现测试程序比较的数值与上述两个资料中不同。

本次所给的测试程序：

```
addi $k0,$0,0x28
beq $k0,$k1,_break
sll $0,$0,0
```

往届测试程序：

```
addi $k0,$0,0x24
beq $k0,$k1,_break
sll $0,$0,0
```

而《自己动手写 CPU》中，对于断点异常 Bp，其在 Cause 寄存器中的 ExcCode 字段值为 9，即 1001，字段与 0x24（00100100）的第 2 至第 6 位对应。综合考虑，本次实验中我将测试程序中的 0x28 改为 0x24，方才测试通过，因此我认为是测试程序中出现了小错误，否则会在最后一个测试例程中出错，最终得到 -10 的错误码。

2. 分频器倍率过大时仿真读不到指令

这和自己的 testbench 很有关系，PC 寄存器在时钟上升沿到来时，如果复位信号是高电平，那么会被初始化为 0，这样后续每次加 4 是显然正确的。但是，如果分频器倍率太大了的话，会导致复位信号时效已经过了时钟上升沿还是没有到来，导致 PC 寄存器中的数值未知，后续的加 4 操作得到的结果也会是未知的，导致 PC 寄存器一直取不到指令。

图 25：PC 取不到指令问题仿真复现

当然，这只是 testbench 没写好导致的，在仿真时减少分频器的倍率即可，到下板时再增加倍率，只是增加一点人力罢了，影响不大。

3. 明明没有更改 PC 的起始地址为 0x00400000，为什么 PC 取值一切正常？

因为 MARS 里面地址是 32 位的，但是由于测试的指令条数不多，不可能用到很多位，且 OpenMIPS 中的地址有效位同样只有 17 位，那个“4”根本没有用到！一旦一开始跳转指令执行

后，那个“4”被写进 PC 寄存器，后续就都抹不掉了！

4. Vivado 2022.2 在综合时发生 synth_design 卡死问题

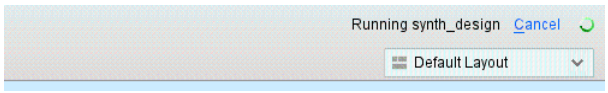

图 26：综合卡死

Vivado 一点 Run Synthesis 就挂死，原来还以为只是综合时间比较长，但等了 30 分钟还是没有综合完成，发现是卡死了。后续发现自己是把 testbench 文件进行综合了，但是按理说这样也可以综合出来的，总之卡死了。更改成真正的顶层文件后可以正常仿真。卡死与否是个玄学问题，网上也没有很好的解决方法，自求多福。

5. Vivado 2022.2 综合失败，但没有详细报错信息

Log 显示：Abnormal program termination (EXCEPTION_ACCESS_VIOLATION)

经过一番搜索，发现的解决办法是关闭 Vivado 综合时的增量编译。关闭后，综合总算成功。

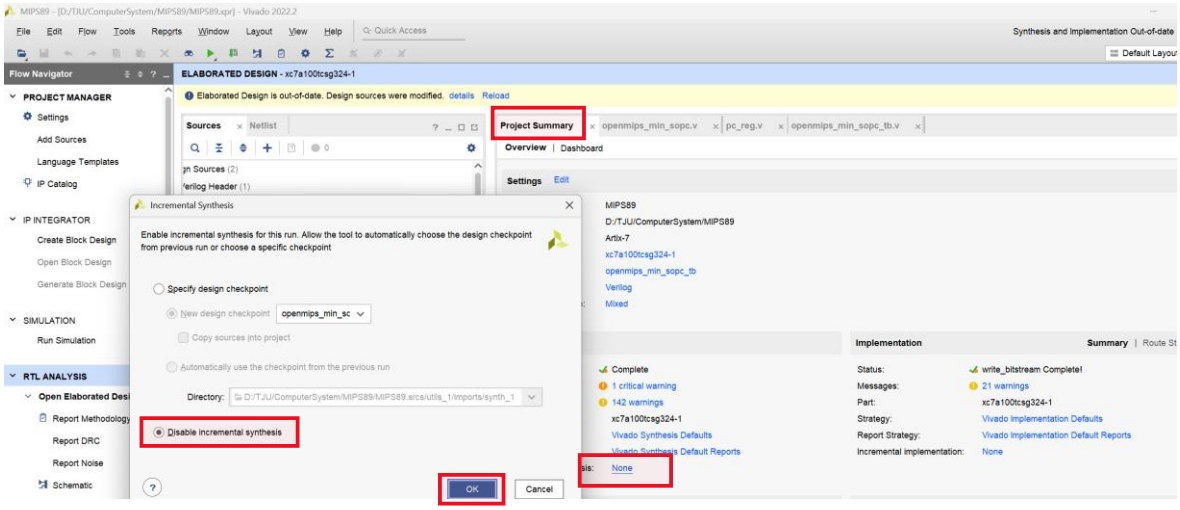

图 27：关闭增量编译的方法

5. Implementation 阶段遇到[DRC MDRV-1] Multiple Driver Nets 报错

这是一个典型的多驱动问题，可能在多个 always 中对同一个变量进行赋值，导致硬件不知道选什么（也可能出现了多个 assign 对同一个变量赋值）。仔细排查代码发现果然有此问题。

6. 增加时钟约束后 Implementation 阶段报错 [Place 30-574]

具体信息如下：

[Place 30-574] Poor placement for routing between an IO pin and BUFG. If this sub optimal condition is acceptable for this design, you may use the CLOCK_DEDICATED_ROUTE constraint in the .xdc file to

demote this message to a WARNING. However, the use of this override is highly discouraged. These examples can be used directly in the .xdc file to override this clock rule.
< set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets I_rst_IBUF] >

报错原因为，编译器在综合时会自动的为工程中的时钟信号生成一个全局时钟 BUF，然后如果管脚分配将这个 BUFG 连接到普通管脚上，就会报以上错误。即使坚持使用 IO 管脚做为全局时钟管脚，这个错误也并不是不能消除的，可以使用 CLOCK_DEDICATED_ROUTE 约束来将这种错误降级为 WARNING。

经查阅资料，可以根据给出的一条语句 set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets I_rst_IBUF]，在 xdc 文件中插入，问题得以解决。注意，左右两侧的尖括号是不要添加的！

7. bit 流无法烧录进板子：End of startup status: LOW

历经千辛万苦，结果在下板的时候还来一个报错！在烧录 bit 流文件时，出现烧录不进去，报以上的错误。最终通过降低 JTAG 下载速率得到解决。具体操作流程见 vivado FPGA 烧录报错_end of startup status:low-CSDN 博客，当然后来几次下板下载速率拉满了也没有报错，可能又是一个玄学问题。

(2) 心得体会

这是手搓的最后一个 CPU，也是最漂亮最优雅的一个 CPU 先生（毕竟是雷老师的心血，帮我们把结构都搭好了代码都写好了！）。秉持着善始善终的原则，我从头开始把雷老师的《自己动手写 CPU》这本书的前 11 章一个字一个字地看了一遍，然后跟着雷老师的思路一步一步添加完善这个代码，从只有一条指令的 CPU，最后实现了 90 条指令，可以说我的代码是网上注释最完善的代码（可能是之一！），特别特别有成就感，这简直是一件艺术品。

在实现雷老师的代码的时候，也出现了一些 Bug，花了很多小时，最终发现是自己的一个小粗心……前前后后花了三个多星期，每天看几页写几页，积少成多。

然后我想吐槽一下 Vivado，在一个堪称完美的艺术品面前狂报错，Vivado 你是不是不服气！从仿真到下板的每一步都在报错，2022 真的可能没有 2016 版本好用。不过我好好佩服自己的检索能力，能在大大小小的网站冗杂的信息中找出有用的解决方法。为了保存这份珍贵的 Debug 经历，我在遇到的问题板块写得特别详细，在课设结束后我会将代码开源，希望我走过的弯路得到的经验可以为学弟学妹们提供那么一点点小小的帮助。

希望接下来的实验可以顺顺利利地完成！（体会写得可能有些少了，但加上遇到的问题有很多很多，嘻嘻）

参考文献

[1] 秦国锋, 王力生, 陆有军, 郭玉臣. 计算机系统结构实验指导, 清华大学出版社,2019.

[2] 雷思磊. 自己动手写 CPU, 电子工业出版社,2014.

[3] 张晨曦, 王志英等. 计算机体系结构, 高等教育出版社,2014.