****

**东北大学秦皇岛分校**

**计算机与通信工程学院**

**计算机组成原理课程设计**

**设计题目** 数字跑表模块和流水线CPU的设计与实现

|  |  |
| --- | --- |
| 专业名称 | 计算机科学与技术 |
| 班级学号 | 计科2205-31-202212701 |
| 学生姓名 | 陈宇帆 |
| 指导教师 | 沈哲 |
| 设计时间 | **2025年1月6日—2025年1月12日** |

**课程设计任务书**

**专业：**计算机科学与技术 **学号：**202212701 **学生姓名（签名）：**

**设计题目：** 数字跑表模块与流水线CPU的设计与实现

**一、设计实验条件**

计算机与通信工程学院实验室，综合楼1419

**二、设计任务及要求**

1. **电路模块设计**：用Verilog实现一个电路模块；
2. **流水线CPU设计**：至少实现21条指令ADD、SUB、OR、AND、XOR、NOR、SLT、SLTU、SLL、SRL、SRA、LUI， ADDU、ADDIU、SUBU、LW、SW、BEQ、BNE、JAL、JR。

要求如下：

1. CPU微结构为静态5级流水。
2. 实现MIPS架构的延迟槽技术，延迟槽可以是任意指令。
3. 控制相关由分支指令造成，通过延迟槽技术可以完美解决。
4. 结构相关即某一级流水停顿了，会阻塞上游的流水级。
5. 要求仿真运行测试程序通过。

**三、设计报告的内容**

1. **设计题目与设计任务（设计任务书）**
2. **前言（绪论）(设计的目的、意义等)**
3. **设计主体（各部分设计内容、分析、结论等）**
4. **结束语（设计的收获、体会等）**
5. **参考资料**

**四、设计时间与安排**

**1、设计时间： 1周**

**2、设计时间安排：**

熟悉实验设备、收集资料： 1 天

设计图纸、实验、计算、程序编写调试： 4 天

编写课程设计报告： 1 天

答辩： 1 天

目录

[【课程设计内容】 3](#_Toc187600993)

[【系统设计与实现】 4](#_Toc187600994)

[1. 数字跑表电路模块设计 4](#_Toc187600995)

[1.1 数字跑表电路模块介绍 4](#_Toc187600996)

[1.2 数字跑表电路模块原理 4](#_Toc187600997)

[1.3 数字跑表电路模块的顶层设计原理图 5](#_Toc187600998)

[1.4 数字跑表电路模块的关键代码 5](#_Toc187600999)

[1.5 数字跑表电路模块的综合结果 9](#_Toc187601000)

[2. 五级流水CPU介绍 10](#_Toc187601001)

[2.1 实现的指令列表（包括指令描述） 10](#_Toc187601002)

[2.2 流水线数据流图 13](#_Toc187601003)

[2.3 流水线模块结构 13](#_Toc187601004)

[2.4 流水线模块接口描述（包括关键代码） 16](#_Toc187601005)

[2.5 总结 35](#_Toc187601006)

[【系统测试】 37](#_Toc187601007)

[1. 数字跑表系统测试 37](#_Toc187601008)

[1.1 数字跑表的功能仿真 37](#_Toc187601009)

[1.2 数字跑表的RTL级逻辑电路 39](#_Toc187601010)

[2. 五级流水系统测试 39](#_Toc187601011)

[2.1 五级流水CPU的功能仿真 39](#_Toc187601012)

[2.2 五级流水CPU的RTL级逻辑电路 47](#_Toc187601013)

[【总结】 48](#_Toc187601014)

[1. 数字跑表 48](#_Toc187601015)

[2. 五级流水CPU 48](#_Toc187601016)

[【心得体会】 49](#_Toc187601017)

[【参考文献（资料）】 50](#_Toc187601018)

# 【课程设计内容】

**题目**：数字跑表和MIPS32五级流水CPU设计

**设计内容如下**：

1. 电路模块设计：用Verilog实现一个数字跑表电路模块；

2. 流水线CPU设计：至少实现21条指令ADD、SUB、OR、AND、XOR、NOR、SLT、SLTU、SLL、SRL、SRA、LUI， ADDU、ADDIU、SUBU、LW、SW、BEQ、BNE、JAL、JR。

要求如下：

（1） CPU微结构为静态5级流水。

（2） 实现MIPS架构的延迟槽技术，延迟槽可以是任意指令。

（3） 控制相关由分支指令造成，通过延迟槽技术可以完美解决。

（4） 结构相关即某一级流水停顿了，会阻塞上游的流水级。

（5） 要求仿真运行测试程序通过。

# 【系统设计与实现】

## 数字跑表电路模块设计

### 数字跑表电路模块介绍

使用Verilog HDL设计一个数字跑表，具有复位、暂停、计数、读表等功能，编写测试代码进行仿真验证。

### 数字跑表电路模块原理

数字跑表的核心功能是通过高频时钟信号（通常由晶振提供）来生成准确的时间基准，并基于用户的操作（如启动、停止、复位）来控制计时器的启停或复位。计时器的计数值通常与时间成比例，最终通过数码管或液晶屏将时间显示出来。

1. 高频时钟信号。当前单片机主板时钟周期多为50MHZ，假设输入系统时钟CLK为50MHZ，时钟周期为20ns，与最低进位10ms的时间不符，需设计一个分频器，将50MHZ的输出分频为100HZ。
2. 时钟计数。然后作为百分秒的进位时钟信号，让它进一步作为秒、分钟的输入时钟信号。例如：通过100个百分秒的输入得到1个1秒的输出，再通过60个1秒得到1个1分钟的输出。
3. 按键控制。通过复位键与暂停键对数据跑表进行逻辑控制。复位键按下后，数据复位，并暂停计数；暂停键按下后，跑表暂停计数，当再次按下后，跑表在原数据上继续计数。
4. 数码显示。本次采用七位数码管进行数据显示，将数字跑表的每一位通过译码输出七个电平信号，控制数码管进行显示。

通过分频器、时钟计数器、控制器、数字七位译码器实现上述功能，将四个模块整合为数字跑表。

### 数字跑表电路模块的顶层设计原理图



图 1 顶层设计原理图

### 数字跑表电路模块的关键代码

设计一个数字跑表的系统结构通常包括四个主要模块：控制模块、计数模块、分频模块、显示模块。每个模块的功能和作用如下：

1. 将系统内部时钟转换为一个毫秒信号时钟的分频模块。
2. 通过按键控制计数状态的按键控制模块；
3. 需要将输入毫秒ms时钟信号转化能输出秒s，分钟m的计数模块。
4. 需要能将毫秒ms、秒s、分钟m在数码管上显示的译码模块。

最后通过顶层模块top将所有模块串联起来实现功能。通过以上设计，数字跑表系统可以实现稳定的计时功能。

#### 分频模块代码

分频功能的作用是通过降低输入时钟信号的频率，生成一个较低频率的时钟信号，以满足系统中不同模块对时钟频率的需求。假设系统的的频率为50MHZ，需要通过分频转换100HZ，用于百分秒计数。

|  |
| --- |
| 模块：分频控制 |
| 输入：分屏系数DIVIDE\_BY，初始信号clk,  输出：分频信号clk\_out  1. module clk\_divider #(parameter DIVIDE\_BY = 5)(  2. input clk,  3. input reset,  4. output reg clk\_out  5. );  6. reg [5:0] counter; // 计数器，支持计数到 50（log2(50) = 6 位）  7.  8. always @(clk ) begin  9. if (reset) begin  10. counter <= 0;  11. clk\_out <= 0; // 初始输出低电平  12. end else begin  13. if (counter == (DIVIDE\_BY - 1)) begin  14. clk\_out <= ~clk\_out; // 翻转输出时钟  15. counter <= 0; // 清零计数器  16. end else begin  17. counter <= counter + 1; // 计数器递增  18. end  19. end  20. end  21. endmodule  22. |

#### 控制模块代码

状态机是一种设计数字电路的有效方法，特别适用于具有顺序逻辑的系统，如数字跑表。它将系统分为有限个状态，每个状态对应于系统的特定行为。通过状态转移，系统可以在不同的状态之间切换，从而实现复杂的控制逻辑。

数字跑表控制模块可以分为以下3个状态：

* 1. 初始状态：所有计数器清零，此时为特殊的暂停状态。
  2. 计数状态：计数器正常计数，显示当前时间。
  3. 暂停状态：计数器停止计数，显示暂停时间。
  4. 复位状态：所有计数器清零，状态转换为初始状态。

可以用以下的状态转移图表示：



图 2 按键状态转移图

|  |
| --- |
| 模块：控制按键 |
| 输入：输入start\_stop, reset\_key  输出: 运行标志run\_flag,置位标志reset\_flag  1. module control();  2.     always @(posedge clk or posedge rst) begin  3.         if (rst) begin  4.             state <= RESET;  5.         end else begin  6.             case (state)  7.                 STOP: begin  8.                     if (start\_stop\_edge) state <= RUNNING;  // 响应按键上升沿  9.                     else if (reset\_key\_edge) state <= RESET;    // 响应复位按键  10.                 end  11.                 RUNNING: begin  12.                     if (start\_stop\_edge) state <= STOP;    // 响应按键上升沿  13.                     else if (reset\_key\_edge) state <= RESET;   // 响应复位按键  14.                 end  15.                 RESET: begin  16.                     state <= STOP;                        // 复位后返回停止状态  17.                 end  18.             endcase  19.         end  20.     end  21. endmodule |

#### 计数模块代码

计数器模块的核心功能是实现时间计数，具体包括毫秒、秒和分钟的逐级进位计时。其主要功能总结如下：

1. 重置功能：支持通过复位信号将毫秒、秒、分钟的计数清零。
2. 毫秒计数：实现从 00 到 99 毫秒的计数，当毫秒达到 99 时自动清零，并触发秒计数器进位。
3. 秒计数：实现从 00 到 59 秒的计数，当秒达到 59 时自动清零，并触发分钟计数器进位。
4. 分钟计数：实现从 00 到 59 分钟的计数，当分钟达到 59 时自动清零，形成一个完整的时间计时周期。
5. 启停控制：通过启动信号控制计时器的运行或暂停，适用于可控计时场景。

|  |
| --- |
| 模块：计数 |
| 输入：分频信号clk\_out,  输出：计数结果dms, sec, min  1. always @(posedge clk )begin  2.     if(rst)begin//重置  3.         dms\_l=4'b0000;  4.         dms\_h=4'b0000;  5.     end else if(start)begin//正常计数时未暂停  6.         if(dms\_l==9)begin//低位进位  7.                 dms\_l=0;  8.                 if(dms\_h==9)begin//高位进位  9.                     dms\_h=0;  10.                     dms\_c=1;//进位标志  11.                 end else  12.                     dms\_h=dms\_h+1;//高位+1  13.         end else begin//低位+1  14.             dms\_l=dms\_l+1;  15.             dms\_c=0;  16.         end  17.     end  18. end  19. always @(posedge clk or posedge rst) begin  20.     if(rst)begin//重置  21.         sec\_l=4'b0000;  22.         sec\_h=4'b0000;  23.     end else if (start) begin  24.         if (dms\_l == 9 && dms\_h == 9) begin  25.             if (sec\_l == 9) begin  26.                 sec\_l <= 0;  27.                 if (sec\_h == 5) begin  28.                     sec\_h <= 0;  29.                 end else begin  30.                     sec\_h <= sec\_h + 1;  31.                 end  32.             end else begin  33.                 sec\_l <= sec\_l + 1;  34.             end  35.         end  36.     end  37.  38. end  39. always @(posedge clk or posedge rst) begin  40.     ……  41.     分钟计数方式  42. end |

#### 译码模块

译码模块将计数结果译码出七位的译码结果，用于连接七位数码管，译码模块由6各4-7译码器组成。

|  |
| --- |
| 模块：译码模块 |
| 1. module display(  2. input [7:0] dms,  3. input [7:0] sec,  4. input [7:0] min,  5. output wire[6:0] dms\_l\_seg,  6. output wire[6:0] dms\_h\_seg, // 秒十位  7. output wire[6:0] sec\_l\_seg, // 分钟个位  8. output wire[6:0] sec\_h\_seg, // 分钟十位  9. output wire[6:0] min\_l\_seg, // 毫秒个位  10. output wire[6:0] min\_h\_seg // 毫秒十位  11. );  12. digit\_to\_7seg dms\_l(dms[3:0], dms\_l\_seg);  13. digit\_to\_7seg dms\_h(dms[7:4], dms\_h\_seg);  14. digit\_to\_7seg sec\_l(sec[3:0], sec\_l\_seg);  15. digit\_to\_7seg sec\_h(sec[7:4], sec\_h\_seg);  16. digit\_to\_7seg min\_l(min[3:0], min\_l\_seg);  17. digit\_to\_7seg min\_h(min[7:4], min\_h\_seg);  18. endmodule |

### 数字跑表电路模块的综合结果

数字跑表的各个模块通过微控制器紧密连接，控制模块负责接收用户输入并管理系统状态，计数模块根据分频信号计算时间，显示模块实时显示计时结果，分频模块提供稳定的时钟脉冲信号。整个系统协调工作，确保计时准确，并能够响应用户输入进行暂停、复位等操作。



图 3 数字跑表综合结果

1. 初始化阶段：
   1. 系统启动时，微控制器初始化所有模块，确保分频模块开始生成时钟信号。
   2. 控制模块、计数模块、分频模块和显示模块相互配合，准备好接收后续的输入信号。
2. 按下“开始/暂停”按钮：
   1. 按键输入被控制模块检测到。
   2. 控制模块切换系统状态到“计数”模式，启动计数器。
   3. 分频模块继续生成时钟信号，计数模块开始根据时钟信号进行计数。
   4. 计数模块将当前计时值传递给显示模块，更新显示信息。
3. 按下“复位”按钮：
   1. 按键输入被控制模块检测到。
   2. 控制模块将复位信号发送给计数模块，计数模块清零当前计时数据。
   3. 显示模块将时间显示为 00:00:00，回到暂停状态。
4. 计时过程中：
   1. 分频模块持续输出时钟脉冲（或每百分秒一次），控制计数器递增。
   2. 计数模块计算计时数据，并将其传输给显示模块。
   3. 显示模块根据收到的计时数据（分钟：秒：百分秒）实时更新显示内容。

## 五级流水CPU介绍

### 实现的指令列表（包括指令描述）

MIPS指令集是极为精简的RISC架构设计，指令长度固定为32位；仅采用Load/Store指令进行访存，所有算术和逻辑操作仅针对寄存器，数据在内存与寄存器之间的传输通过专门的加载（Load）和存储（Store）指令完成；支持丰富的算术、逻辑、移位、分支跳转和特权指令，指令集简洁高效，易于硬件实现和优化。

以下是func\_test1, func\_test2中含有指令的指令描述，另外，实现CPU需要特权指令、中断指令等指令的实现，所以在课程设计要求外实现了中断、特权指令等共计61条指令，实现CPU的基本功能。

表1是func\_test1中包含的指令格式，共计21条。

表 1 func\_test121条指令描述

|  |  |  |
| --- | --- | --- |
| 指令 | 功能描述 | Func\_test1 |
| lui | 加载上半字，将立即数左移 16 位后存入寄存器 | lui $1, 0x1234 |
| add | 加法，将两个寄存器的值相加，结果存入目标寄存器 | add $3, $1, $2 |
| sub | 减法，将两个寄存器的值相减，结果存入目标寄存器 | sub $4, $1, $2 |
| or | 或运算，将两个寄存器的值按位或，结果存入目标寄存器 | or $5, $1, $2 |
| and | 与运算，将两个寄存器的值按位与，结果存入目标寄存器 | and $6, $1, $2 |
| xor | 异或运算，将两个寄存器的值按位异或，结果存入目标寄存器 | xor $7, $1, $2 |
| nor | 非或运算，将两个寄存器的值按位或后取反，结果存入目标寄存器 | nor $8, $1, $2 |
| bne | 不相等跳转，如果两个寄存器的值不相等，则跳转到指定标签 | bne $1, $0, s2 |
| beq | 相等跳转，如果两个寄存器的值相等，则跳转到指定标签 | beq $2, $1, s1 |
| slt | 有符号比较，第一个寄存器小于第二个寄存器，则结果为 1否则为 0 | slt $9, $1, $2 |
| sltu | 无符号比较，第一个寄存器小于第二个寄存器，则结果为 1否则为 0 | sltu $10, $1, $2 |
| sll | 逻辑左移，将寄存器的值左移指定位数 | sll $11, $1, 2 |
| srl | 逻辑右移，将寄存器的值右移指定位数 | srl $12, $1, 16 |
| sra | 算术右移，将寄存器的值右移指定位数，符号位扩展 | sra $13, $1, 2 |
| addu | 无符号加法，将两个寄存器的值相加，结果存入目标寄存器 | addu $14, $1, $2 |
| addiu | 无符号立即数加法，将寄存器的值加上一个立即数 | addiu $15, $1, 0x2020 |
| subu | 无符号减法，将两个寄存器的值相减，结果存入目标寄存器 | subu $16, $1, $2 |
| sw | 存储字，将寄存器的值存储到内存中 | sw $14, 0x8($0) |
| lw | 加载字，将内存中的值加载到寄存器中 | lw $18, 0x8($0) |
| jal | 跳转并链接，将返回地址存入 $ra，然后跳转到指定地址 | jal 0x80 |
| jr | 跳转到寄存器所指的地址 | jr $3 |

表2是func\_test2中包含的指令格式，共计17条。

表 2 func\_test2中19条指令描述

|  |  |  |
| --- | --- | --- |
| 指令类别 | 功能 | Func\_test2指令 |
| addi | 将立即数与寄存器值相加，结果存储在目标寄存器 | addi $1, $0, 0x1234 |
| slti | 比较寄存器值与立即数（有符号），小于则置 1 | slti $4, $3, 0x1 |
| sltiu | 比较寄存器值与立即数（无符号），小于则置 1 | sltiu $5, $3, 0x1 |
| andi | 寄存器值与立即数按位与，存储在目标寄存器 | Andi $6,$1,0x5678 |
| ori | 寄存器值与立即数按位或，存储在目标寄存器 | ori $7, $1, 0x5678 |
| xori | 寄存器值与立即数按位异或，结果存储在目标寄存器 | xori $8, $1, 0x5678 |
| sllv | 将寄存器值左移，移位量由另一个寄存器指定 | sllv $9, $2, $1 |
| srav | 将寄存器值右移（算术）移位量由另一个寄存器指定 | srav $10, $3, $31 |
| srlv | 将寄存器值右移（逻辑）移位量由另一个寄存器指定 | srlv $11, $3, $31 |
| mult | 将两个寄存器值相乘，结果存入 HI 和 LO 寄存器 | mult $1, $3 |
| mfhi | 从 HI 寄存器读取值到目标寄存器 | mfhi $12 |
| mflo | 从 LO 寄存器读取值到目标寄存器 | mflo $13 |
| multu | 无符号乘法，结果存入 HI 和 LO 寄存器 | multu $1, $3 |
| div | 有符号除法，将商存入 LO，余数存入 HI 寄存器 | div $zero, $3, $1 |
| divu | 无符号除法，将商存入 LO，余数存入 HI 寄存器 | divu $3, $1 |
| mthi | 将寄存器值存入 HI 寄存器 | mthi $1 |
| mtlo | 将寄存器值存入 LO 寄存器 | mtlo $2 |

**另外，**实现CPU需要特权指令、中断指令等指令的实现，以下是课程设计外完成的23条指令。

表 3 其余23条指令的指令描述

|  |  |  |
| --- | --- | --- |
| 指令类别 | 功能 | 示例指令 |
| BGEZ | 大于等于 0 转移 | BGEZ rs, offset |
| BGTZ | 大于 0 转移 | BGTZ rs, offset |
| BLEZ | 小于等于 0 转移 | BLEZ rs, offset |
| BLTZ | 小于 0 转移 | BLTZ rs, offset |
| BLTZAL | 小于 0 调用子程序并保存返回地址 | BLTZAL rs, offset |
| BGEZAL | 大于等于 0 调用子程序并保存返回地址 | BGEZAL rs, offset |
| J | 无条件直接跳转 | J target |
| JALR | 无条件寄存器跳转至子程序并保存返回地址下 | JALR rd, rs |
| LB | 取字节有符号扩展 | LB rt, offset(base) |
| LBU | 取字节无符号扩展 | LBU rt, offset(base) |
| LH | 取半字有符号扩展 | LH rt, offset(base) |
| LHU | 取半字无符号扩展 | LHU rt, offset(base) |
| LWL | 非对齐地址取字至寄存器左部 | LWL rt, offset(base) |
| LWR | 非对齐地址取字至寄存器右部 | LWR rt, offset(base) |
| SB | 存字节 | SB rt, offset(base) |
| SH | 存半字 | SH rt, offset(base) |
| SWL | 寄存器左部存入非对齐地址 | SWL rt, offset(base) |
| SWR | 寄存器右部存入非对齐地址 | SWR rt, offset(base) |
| ERET | 例外处理返回 | ERET |
| MFC | 读 CP0 寄存器值至通用寄存器 | MFC0 |
| MTC | 通用寄存器值写入 CP0 寄存器 | MTC0 |
| BREAK | 断点 | BREAK |
| SYSCALL | 系统调用 | SYSCALL |

### 流水线数据流图

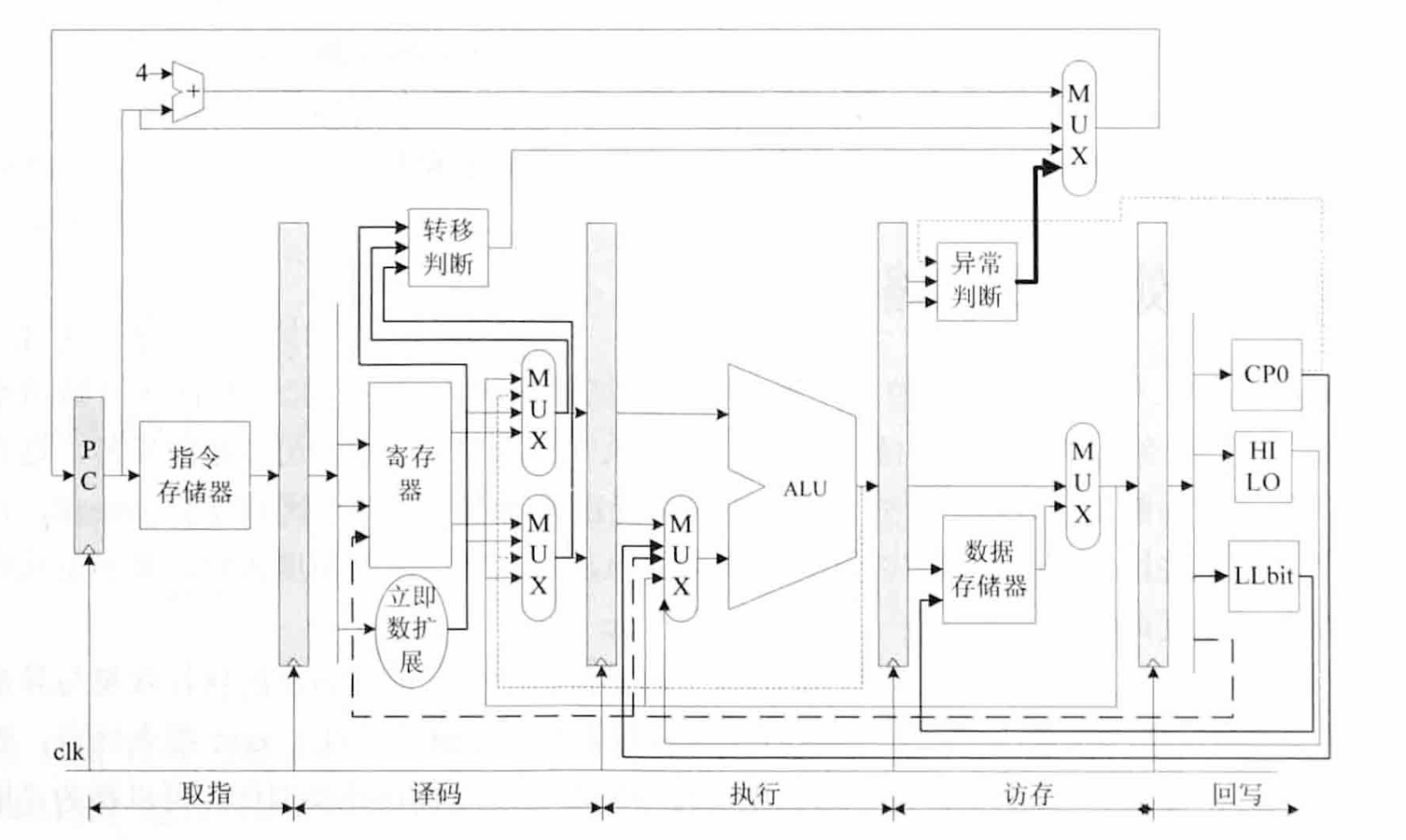


图 4 五级流水CPU数据流图[2]

### 流水线模块结构

#### 五级流水CPU系统结构图

五级流水CPU 的功能模块为pc、id、alu、mem、wb五个模块组成，控制模块为四个锁存器if\_id、id\_ex、ex\_mem、mem\_wb，以及指令暂停控制模块ctrl，数据模块有data\_ram、inst\_rom，以及数据寄存器堆regfiles和状态控制寄存器堆cpu0\_reg，乘商寄存器hilo\_reg、锁存控制器llbit\_reg。



图 5 各模块连接关系[2]

#### 上层模块连接结构

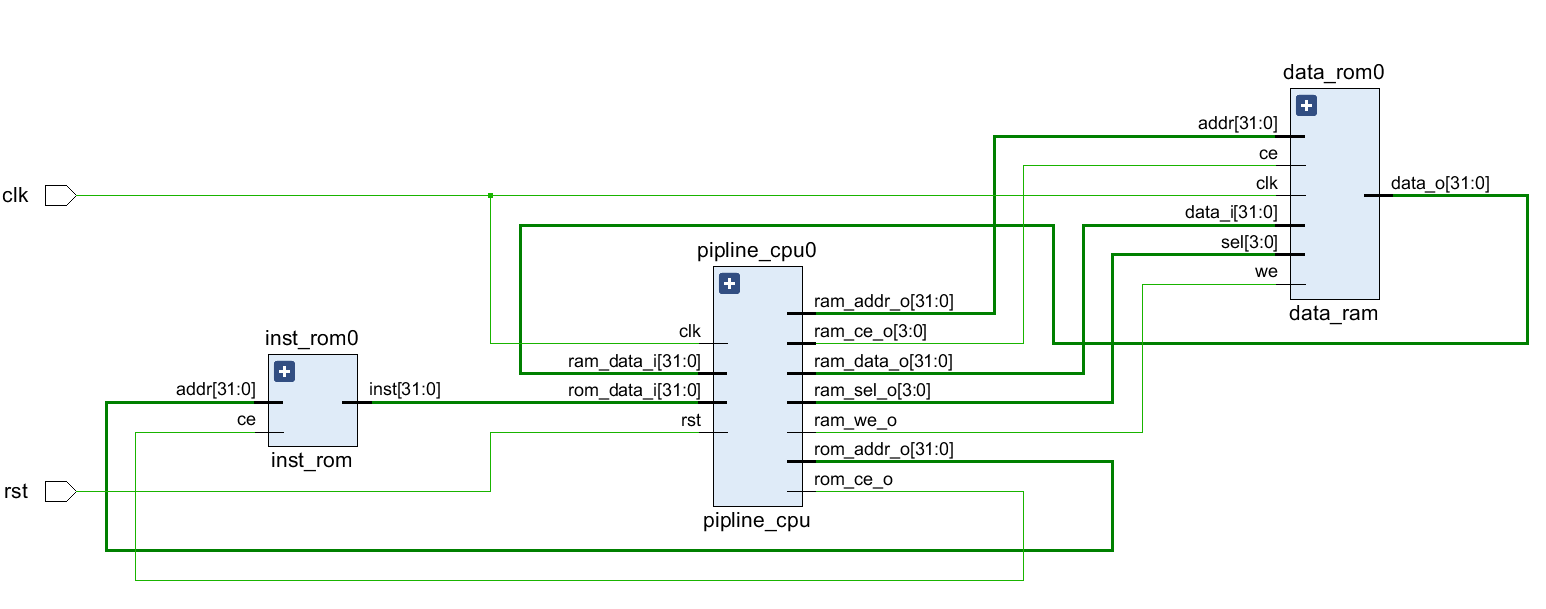


图 6 上层功能模块

#### 功能模块PC、id、alu、mem

五级流水中主要有PC、id、alu、mem、wb五个功能模块，其中wb模块由寄存器堆regfile实现，regfile模块在数据模块中展示，以下为PC、id、alu、mem模块接口电路。

|  |  |  |
| --- | --- | --- |
| 图 7 PC功能模块 | 图 8 id功能模块 | |
| 图 9 alu功能模块 | 图 10 mem功能模块 |

#### 控制模块if\_id、id\_ex、ex\_mem、mem\_wb、ctrl

在五级流水线处理器中，if\_id、id\_ex、ex\_mem、mem\_wb 和 ctrl 是非常重要的锁存器或寄存器组，用于管理指令在流水线各阶段之间的数据传递和控制流。ctrl 是流水线处理器中的控制单元，负责根据当前指令生成各阶段所需的控制信号，协调流水线工作。由if\_id、id\_ex、ex\_mem、mem\_wb等锁存上一个模块的数据，并跟俊ctrl中发出的信号对下一个模块进行控制。

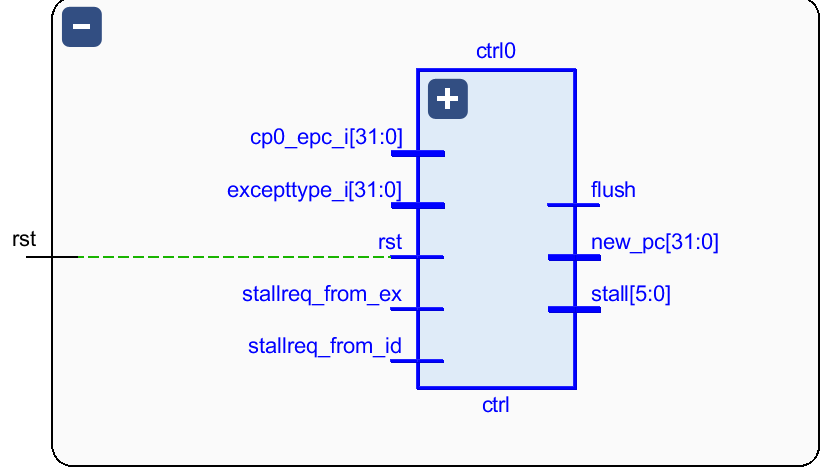


图 11 ctrl控制模块

|  |  |
| --- | --- |
| 图 12 ex\_mem控制模块 | 图 13 id\_ex控制模块 |
| 图 14 if\_id控制模块 | 图 15 mem\_wb控制模块 |

#### 数据模块data\_ram、inst\_rom、 regfiles、 cpu0\_reg、hi,lo、llbit

在 CPU 设计中，模块如 data\_ram、inst\_rom、regfiles、cp0\_reg、hilo、llbit 等是非常重要的组成部分，负责数据存储、指令存储、寄存器管理等功能。cp0\_reg 是用于处理异常和中断的专用寄存器组。在 MIPS 处理器中，协处理器是一个负责管理异常、状态和控制的特殊模块，管理状态与异常，并输出状态信息。llbit 是一个用于支持同步机制的标志位，主要用于实现负载链接/条件存储 (LL/SC) 操作。

|  |  |
| --- | --- |
| 图 16 inst\_rom数据模块 | 图 17 data\_ram数据模块 |
| 图 18 regfile数据模块 | 图 19 cp0\_reg寄存器 |
| 图 20 乘商寄存器 | 图 21 llsc锁存状态寄存器 |

### 流水线模块接口描述（包括关键代码）

#### 程序计数器pc模块

程序计数器PC,在取指阶段取出指令存储器中的指令，当有复位信号rst时，输出指令存储器使能信号为ChipDisable(0),表示指令存储器禁用，此时PC的值保持为0。其余时刻指令存储器使能信号为ChipEnale(1),PC可能有三种取值：

① PC = PC + 4（顺序流动）

② PC不变（流水线暂停stall[0]==`Stop）

③ PC等于转移判断的结果（如果是转移指令且满足转移条件）

表 4 pc模块接口描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | rst | 1 | 输入 | 复位信号 |
| 2 | clk | 1 | 输入 | 时钟信号 |
| 3 | pc | 32 | 输出 | 要读取的指令地址 |
| 4 | ce | 1 | 输出 | 指令存储器使能信号 |
| 5 | branch\_flag\_i | 1 | 输入 | 是否发生转移，如果发生转移，就将pc设置为跳转地址 |
| 6 | branch\_target\_address\_i | 32 | 输入 | 指令转移目标地址 |
| 7 | stall | 6 | 输入 | 是否暂停流水线（由ctrl发出信号决定） |

代码部分，其中重要的PC转移与PC自增。

|  |
| --- |
| 模块：PC |
| 1. always @ (posedge clk) begin  2.     if (ce == `ChipDisable) begin  3.         pc <= 32'h00000000;  4.     end else begin  5.         if(flush == 1'b1) begin  6.             pc <= new\_pc;  7.         end else if(stall[0] == `NoStop) begin  8.             if(branch\_flag\_i == `Branch) begin  9.                 pc <= branch\_target\_address\_i;  10.             end else begin  11.                 pc <= pc + 4'h4;  12.             end  13.         end  14.     end  15. end |

#### 锁存if\_id模块

if\_id模块负责锁存PC与inst\_rom中的指令信号，同时接收控制信号，通过控制输出数据控制id模块的进行。

表 5 if\_id接口描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | rst | 1 | 输入 | 复位信号 |
| 2 | clk | 1 | 输入 | 时钟信号 |
| 3 | if\_inst | 32 | 输入 | 取指阶段的指令 |
| 5 | id\_inst | 32 | 输出 | 指令送入译码阶段 |
| 5 | if\_pc | 32 | 输入 | 取指阶段的地址 |
| 6 | id\_pc | 32 | 输出 | 地址送入译码阶段 |
| 7 | stall | 6 | 输入 | 流水线是否暂停 |

代码部分，重要的是PC锁存与id控制数据。

|  |
| --- |
| 模块：if\_id |
| 1. always @ (posedge clk) begin  2. if (rst == `RstEnable) begin  3. id\_pc <= `ZeroWord;  4. id\_inst <= `ZeroWord;  5. end else if(flush == 1'b1 ) begin  6. id\_pc <= `ZeroWord;  7. id\_inst <= `ZeroWord;  8. end else if(stall[1] == `Stop && stall[2] == `NoStop) begin  9. id\_pc <= `ZeroWord;  10. id\_inst <= `ZeroWord;  11. end else if(stall[1] == `NoStop) begin  12. id\_pc <= if\_pc;  13. id\_inst <= if\_inst;  14. end  15. end |

#### 译码id模块

ID模块的作用是对指令进行译码，ID模块可以将指令种类以及子种类译码出来，并通过指令种类和子种类，进一步译码出访存地址，读写寄存器地址，立即数等。在实验课的基础上，ID模块还实现了延迟槽的功能，通过设计延迟槽，CPU可以处理转移指令。同时，如果为条件转移指令，那么在ID模块就要进行条件判断，并将跳转到的地址通过branch\_target\_address接口传给PC模块，如果此转移指令需要保存地址，可以通过ID模块的link\_addr\_o模块传给后面的模块。。

其中id模块由于去操作数需解决数据相关问题的冲突，主要为RAM，可以使用数据旁路处理大部分指令相关，但是LOAD/STORE指令结果计数的特殊性，可以采用指令暂停结束数据相关冲突。

表 6 id接口描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | rst | 1 | 输入 | 复位信号 |
| 2 | inst\_i | 32 | 输入 | 译码阶段的指令 |
| 3 | reg1\_data\_i | 32 | 输入 | 从regfile读入寄存器数据1 |
| 4 | reg2\_data\_i | 32 | 输入 | 从regfile读入寄存器数据2 |
| 5 | aluop\_o | 4 | 输出 | 译码阶段运算类型 |
| 6 | reg1\_o | 32 | 输出 | 译码阶段源操作数1 |
| 7 | reg2\_o | 32 | 输出 | 译码阶段源操作数2 |
| 8 | wd\_o | 5 | 输出 | 目的寄存器地址 |
| 9 | wreg\_o | 1 | 输出 | 是否要写入目的寄存器 |
| 10 | reg2\_addr\_o | 5 | 输出 | regfile第二个寄存器地址 |
| 11 | reg2\_read\_o | 1 | 输出 | regfile第二个寄存器读使能信号 |
| 12 | reg1\_addr\_o | 5 | 输出 | regfile第一个寄存器地址 |
| 13 | reg1\_read\_o | 1 | 输出 | regfile第一个寄存器读使能信号 |
| 14 | inst\_o | 32 | 输出 | 指令输出端口，用于alu计算访存地址mem\_addr\_o |
| 控制相关——延迟槽相关接口 | | | | |
| 15 | is\_in\_delayslot\_i | 1 | 输入 | 当前处于译码阶段的指令是否在延迟槽 |
| 16 | next\_inst\_in\_delayslot\_o | 1 | 输出 | 下一条进入译码阶段的指令是否在延迟槽 |
| 17 | branch\_flag\_o | 1 | 输出 | 是否发生转移 |
| 18 | branch\_target\_address\_o | 32 | 输出 | 转移到的目标地址 |
| 19 | link\_addr\_o | 32 | 输出 | 转移指令保存的返回地址 |
| 20 | is\_in\_delayslot\_o | 1 | 输出 | 当前处于译码阶段的指令是否在延迟槽 |
| 数据相关解决接口 | | | | |
| 21 | ex\_waddr\_i | 5 | 输入 | 执行阶段指令写入寄存器地址 |
| 22 | ex\_wreg\_i | 1 | 输入 | 执行阶段指令是否写寄存器 |
| 23 | ex\_wdata\_i | 32 | 输入 | 执行阶段写入寄存器数据 |
| 24 | mem\_wreg\_i | 1 | 输入 | 访存阶段指令是否写寄存器 |
| 25 | mem\_waddr\_i | 5 | 输入 | 访存阶段指令写入寄存器地址 |
| 26 | mem\_wdata\_i | 32 | 输入 | 访存阶段写入寄存器数据 |

代码部分，主要分为译码、数据旁路（数据相关处理）、指令暂停load/store指令数据相关控制。

|  |
| --- |
| 模块：指令译码（部分） |
| 1. always @ (\*) begin  2.     if (rst == `RstEnable) begin  3.         ……  4.     end else begin  5.         ……  6.     case (op)  7.         `EXE\_SPECIAL\_INST:      begin  8.             case (op2)  9.                 5'b00000:           begin  10.                     case (op3)  11.                         `EXE\_OR:    begin  12.                             wreg\_o <= `WriteEnable;     aluop\_o <= `EXE\_OR\_OP;  13.                             alusel\_o <= `EXE\_RES\_LOGIC;     reg1\_read\_o <= 1'b1;    reg2\_read\_o <= 1'b1;  14.                             instvalid <= `InstValid;  15.                         end  16.                         `EXE\_AND:   begin  17.                             wreg\_o <= `WriteEnable;     aluop\_o <= `EXE\_AND\_OP;  18.                             alusel\_o <= `EXE\_RES\_LOGIC;   reg1\_read\_o <= 1'b1;  reg2\_read\_o <= 1'b1;  19.                             instvalid <= `InstValid;  20.                         end  21.                         `EXE\_XOR:   begin  22.                             wreg\_o <= `WriteEnable;     aluop\_o <= `EXE\_XOR\_OP;  23.                             alusel\_o <= `EXE\_RES\_LOGIC;     reg1\_read\_o <= 1'b1;    reg2\_read\_o <= 1'b1;  24.                             instvalid <= `InstValid;  25.                         end  26.                         ……  27.                     endcase  28.                 end  29.             endcase  30.         end  31.     endcase  32.     end  33. end |

数据旁路解决相邻4条数据的RAW的问题

|  |
| --- |
| 模块：数据旁路 |
| 1. always @ (\*) begin  2.         stallreq\_for\_reg1\_loadrelate <= `NoStop;  3.     if(rst == `RstEnable) begin  4.         reg1\_o <= `ZeroWord;  5.     end else if(pre\_inst\_is\_load == 1'b1 && ex\_wd\_i == reg1\_addr\_o  6.                             && reg1\_read\_o == 1'b1 ) begin  7.         stallreq\_for\_reg1\_loadrelate <= `Stop;  8.     end else if((reg1\_read\_o == 1'b1) && (ex\_wreg\_i == 1'b1)  9.                             && (ex\_wd\_i == reg1\_addr\_o)) begin  10.         reg1\_o <= ex\_wdata\_i;  11.     end else if((reg1\_read\_o == 1'b1) && (mem\_wreg\_i == 1'b1)  12.                             && (mem\_wd\_i == reg1\_addr\_o)) begin  13.         reg1\_o <= mem\_wdata\_i;  14.     end else if(reg1\_read\_o == 1'b1) begin  15.         reg1\_o <= reg1\_data\_i;  16.     end else if(reg1\_read\_o == 1'b0) begin  17.         reg1\_o <= imm;  18.     end else begin  19.         reg1\_o <= `ZeroWord;  20.     end  21. end |

指令暂停计数解决4条数据的RAW的问题。如果alu判断时LOAD指令时，将信号传入ID模块中，ID模块输出控制信号stallreq。

|  |
| --- |
| 模块：指令暂停 |
| 1. assign pre\_inst\_is\_load = ((ex\_aluop\_i == `EXE\_LB\_OP) ||  2.                 (ex\_aluop\_i == `EXE\_LBU\_OP)||  3.                 (ex\_aluop\_i == `EXE\_LH\_OP) ||  4.                 (ex\_aluop\_i == `EXE\_LHU\_OP)||  5.                 (ex\_aluop\_i == `EXE\_LW\_OP) ||  6.                 (ex\_aluop\_i == `EXE\_LWR\_OP)||  7.                 (ex\_aluop\_i == `EXE\_LWL\_OP)||  8.                 (ex\_aluop\_i == `EXE\_LL\_OP) ||  9.                 (ex\_aluop\_i == `EXE\_SC\_OP)) ? 1'b1 : 1'b0;  10. assign stallreq = stallreq\_for\_reg1\_loadrelate |stallreq\_for\_reg2\_loadrelate; |

#### 锁存id\_ex模块

id\_ex仅用锁存数据，模块负责锁存id的数据信号，同时接收控制信号，通过控制输出数据控制id模块的进行。

表 7 id\_ex接口描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | rst | 1 | 输入 | 复位信号 |
| 2 | clk | 1 | 输入 | 时钟信号 |
| 3 | id\_aluop | 6 | 输入 | 运算类型 |
| 4 | id\_reg1 | 32 | 输入 | 写寄存器1数据 |
| 5 | id\_reg2 | 32 | 输入 | 写寄存器2数据 |
| 6 | id\_wd | 5 | 输入 | 写寄存器地址 |
| 7 | id\_wreg | 1 | 输入 | 是否写寄存器 |
| 8 | ex\_aluop | 6 | 输出 | 运算类型 |
| 9 | ex\_reg1 | 32 | 输出 | 写寄存器1数据 |
| 10 | ex\_reg2 | 32 | 输出 | 写寄存器2数据 |
| 11 | ex\_wd | 5 | 输出 | 写寄存器地址 |
| 12 | ex\_wreg | 1 | 输出 | 是否写寄存器 |
| 13 | id\_inst | 32 | 输入 | 当前指令 |
| 14 | ex\_inst | 32 | 输出 | 当前指令 |
| 分支跳转——延迟槽相关接口 | | | | |
| 15 | id\_link\_address | 32 | 输入 | 返回地址 |
| 16 | id\_is\_in\_delayslot | 1 | 输入 | 当前指令是否在延迟槽 |
| 17 | next\_inst\_in\_delayslot\_i | 1 | 输入 | 下条指令是否在延迟槽 |
| 18 | ex\_link\_address | 32 | 输出 | 返回地址 |
| 19 | ex\_is\_in\_delayslot | 1 | 输出 | 当前指令是否在延迟槽 |
| 20 | is\_in\_delayslot\_o | 1 | 输出 | 下条指令是否在延迟槽 |

代码部分，重要的是id锁存与ex控制数据。

|  |
| --- |
| 模块:id\_ex |
| 1. else if(stall[2] == `NoStop) begin  2.     ex\_aluop <= id\_aluop;  3.     ex\_alusel <= id\_alusel;  4.     ex\_reg1 <= id\_reg1;  5.     ex\_reg2 <= id\_reg2;  6.     ex\_wd <= id\_wd;  7.     ex\_wreg <= id\_wreg;  8.     ex\_link\_address <= id\_link\_address;  9.     ex\_is\_in\_delayslot <= id\_is\_in\_delayslot;  10.     is\_in\_delayslot\_o <= next\_inst\_in\_delayslot\_i;  11.     ex\_inst<=id\_inst;  12.     ex\_excepttype <= id\_excepttype;  13.     ex\_current\_inst\_address <= id\_current\_inst\_address;  14. end |

#### 运算alu模块

alu模块是 CPU 流水线的执行阶段，它的主要功能是完成指令的算术运算、逻辑操作、乘法/除法、地址计算等，并将结果传递给下一流水线阶段（MEM 阶段）。这是流水线执行的核心计算部分。

1. 依据输入的运算子类型进行运算，并将相应的结果保存在暂时的符号中，主要有逻辑运算，算术运算，移动，跳转等类型。
2. 给出最终的运算结果，包括是否要写目的寄存器wreg\_o、 要写的目的寄存器地址wd\_o、 要写入的数据wdata\_o。
3. LOAD/STORE指令在alu模块中需计算写入地址mem\_addr\_o与写入的初始数据reg2\_o。
4. 在计算DIV与MUL等需多周期计算指令时，需要输出信号，进行指令暂停。

表 8 运算器alu接口描述表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | rst | 1 | 输入 | 复位信号 |
| 2 | alu\_control | 12 | 输入 | 运算类型 |
| 3 | alu\_src1 | 32 | 输入 | 操作数1 |
| 4 | alu\_src2 | 32 | 输入 | 操作数2 |
| 5 | alu\_result | 32 | 输出 | 运算结果 |
| 6 | wd\_i | 5 | 输入 | 写入寄存器地址 |
| 7 | wreg\_i | 1 | 输入 | 是否写入寄存器 |
| 8 | wd\_o | 5 | 输出 | 写入寄存器地址 |
| 9 | wreg\_o | 1 | 输出 | 是否写入寄存器 |
| 10 | inst\_i | 32 | 输入 | 当前指令 |
| 11 | aluop\_o | 6 | 输出 | 运算类型 |
| 12 | mem\_addr\_o | 32 | 输出 | 访存地址 |
| 13 | reg2\_o | 32 | 输出 | 存入存储器的数 |
| 14 | stallreq | 1 | 输出 | 流水线暂停请求->ctrl |
| 延迟槽相关接口 | | | | |
| 15 | link\_address\_i |  | 输入 | 转移指令保存的返回地址 |
| 16 | is\_in\_delayslot\_i | 1 | 输入 | 当前指令是否在延迟槽 |
| 乘法相关接口 | | | | |
| 17 | mul\_result\_i | 64 | 输入 | 乘法运算结果 |
| 18 | mul\_ready\_i | 1 | 输入 | 乘法器是否运算完毕 |
| 19 | mul\_opdata1\_o | 32 | 输出 | 被乘数 |
| 20 | mul\_opdata2\_o | 32 | 输出 | 乘数 |
| 21 | mul\_start\_o | 1 | 输出 | 启动乘法器 |
| 22 | signed\_mul\_o | 1 | 输出 | 有无符号 |
| 除法相关接口 | | | | |
| 23 | div\_result\_i | 64 | 输入 | 除法运算结果 |
| 24 | div\_ready\_i | 1 | 输入 | 除法器是否运算完毕 |
| 25 | div\_opdata1\_o | 32 | 输出 | 被除数 |
| 26 | div\_opdata2\_o | 32 | 输出 | 除数 |
| 27 | div\_start\_o | 1 | 输出 | 启动除法器 |
| 28 | signed\_div\_o | 1 | 输出 | 有无符号 |
| 乘商寄存器数据相关的接口 | | | | |
| 29 | hi\_i | 32 | 输入 | HI寄存器内容 |
| 30 | lo\_i | 32 | 输入 | LO寄存器内容 |
| 31 | wb\_hi\_i | 32 | 输入 | 回写时是否要写HI |
| 32 | wb\_lo\_i | 32 | 输入 | 回写时是否要写LO |
| 33 | wb\_whilo\_i | 1 | 输入 | 回写时是否写HI/LO |
| 34 | mem\_hi\_i | 32 | 输入 | 访存时是否要写HI |
| 35 | mem\_lo\_i | 32 | 输入 | 访存时是否要写LO |
| 36 | mem\_whilo\_i | 1 | 输入 | 访存时是否写HI/LO |
| 37 | hi\_o | 32 | 输出 | 执行阶段写入HI的值 |
| 38 | lo\_o | 32 | 输出 | 执行阶段写入LO的值 |
| 39 | whilo\_o | 1 | 输出 | 执行阶段是否要写HI/LO |

代码部分，核心为alu算术逻辑运算、地址计算、指令暂停控制。

|  |
| --- |
| 模块：alu算术逻辑计算 |
| 1. always@(\*) begin  2.     wd\_o<= wd\_i;  3.     if(((aluop\_i == `EXE\_ADD\_OP) || (aluop\_i == `EXE\_ADDI\_OP) ||  4.         (aluop\_i == `EXE\_SUB\_OP)) && (ov\_sum == 1'b1)) begin  5.         wreg\_o <= `WriteDisable;  6.         ovassert<=  1;  7.     end else begin  8.         wreg\_o <= wreg\_i;  9.         ovassert<=  0;  10.     end  11.     case(alusel\_i)  12.         `EXE\_RES\_LOGIC: begin  13.             wdata\_o <= logicout;  14.         end  15.         `EXE\_RES\_SHIFT:begin  16.             wdata\_o<=shiftres;  17.         end  18.             `EXE\_RES\_MOVE:      begin  19.             wdata\_o <= moveres;  20.         end  21.         `EXE\_RES\_ARITHMETIC: begin  22.             wdata\_o<= arithmeticres;  23.         end  24.         `EXE\_RES\_MUL: begin  25.             wdata\_o<=mulres[31:0];  26.         end  27.         `EXE\_RES\_JUMP\_BRANCH: begin  28.             wdata\_o<= link\_address\_i;  29.         end  30.         default: begin  31.             wdata\_o<=`ZeroWord;  32.         end  33.     endcase  34. end |

地址计算与写入初始值

|  |
| --- |
| 模块：地址计算 |
| 1. assign mem\_addr\_o= reg1\_i+ {{16{inst\_i[15]}},inst\_i[15:0]};  2.assign reg2\_o= reg2\_i; |

除法等多周期指令，进行指令暂停控制

|  |
| --- |
| 模块：指令暂停（除法） |
| 1. always@(\*) begin  2.     stallreq=stallreq\_for\_madd\_msub||stallreq\_for\_div;  3. end  4. //DIV、DIVU指令  5. always @ (\*) begin  6.     if(rst == `RstEnable) begin  7.         ……  8.     end else begin  9.         stallreq\_for\_div <= `NoStop;  10.         div\_opdata1\_o <= `ZeroWord;  11.         div\_opdata2\_o <= `ZeroWord;  12.         div\_start\_o <= `DivStop;  13.         signed\_div\_o <= 1'b0;  14.         case (aluop\_i)  15.             `EXE\_DIV\_OP:        begin  16.                 if(div\_ready\_i == `DivResultNotReady) begin  17.                 div\_opdata1\_o <= reg1\_i;  18.                     div\_opdata2\_o <= reg2\_i;  19.                     div\_start\_o <= `DivStart;  20.                     signed\_div\_o <= 1'b1;  21.                     stallreq\_for\_div <= `Stop;  22.                 end else if(div\_ready\_i == `DivResultReady) begin  23.                 div\_opdata1\_o <= reg1\_i;  24.                     div\_opdata2\_o <= reg2\_i;  25.                     div\_start\_o <= `DivStop;  26.                     signed\_div\_o <= 1'b1;  27.                     stallreq\_for\_div <= `NoStop;  28.                 end else begin  29.                 div\_opdata1\_o <= `ZeroWord;  30.                     div\_opdata2\_o <= `ZeroWord;  31.                     div\_start\_o <= `DivStop;  32.                     signed\_div\_o <= 1'b0;  33.                     stallreq\_for\_div <= `NoStop;  34.                 end  35.             end  36.         endcase  37.     end  38. end |

#### 锁存ex\_mem模块

EX\_MEM模块是处理器流水线中的一个阶段寄存器，用于连接执行阶段（EX） 和访存阶段（MEM）。其主要作用是保存从 EX 阶段传递过来的数据和控制信号，以便在访存阶段使用。作为流水线寄存器，EX\_MEM模块的设计确保了流水线的同步性和数据的准确传递。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | rst | 1 | 输入 | 复位信号 |
| 2 | clk | 1 | 输入 | 时钟信号 |
| 3 | ex\_wd | 5 | 输入 | 写入寄存器地址 |
| 4 | ex\_wreg | 1 | 输入 | 是否写入寄存器 |
| 5 | ex\_wdata | 32 | 输入 | 写入寄存器数据 |
| 6 | mem\_wd | 5 | 输出 | 写入寄存器地址 |
| 7 | mem\_wreg | 1 | 输出 | 是否写入寄存器 |
| 8 | mem\_wdata | 32 | 输出 | 写入寄存器数据 |
| 9 | stall | 6 | 输入 | 流水线暂停 |
| load/store相关接口 | | | | |
| 10 | ex\_aluop | 6 | 输入 | 运算类型 |
| 11 | ex\_mem\_addr | 32 | 输入 | 访存地址 |
| 12 | ex\_reg2 | 32 | 输入 | 放入存储器的数据 |
| 13 | mem\_aluop | 6 | 输出 | 运算类型 |
| 14 | mem\_mem\_addr | 32 | 输出 | 访存地址 |
| 15 | mem\_reg2 | 32 | 输出 | 放入存储器的数据 |
| 移动指令相关接口 | | | | |
| 16 | ex\_hi | 32 | 输入 | 执行阶段写入HI的值 |
| 17 | ex\_lo | 32 | 输入 | 执行阶段写入LO的值 |
| 18 | ex\_whilo | 1 | 输入 | 执行阶段是否要写HI/LO |
| 19 | mem\_hi | 32 | 输出 | 执行阶段写入HI的值 |
| 20 | mem\_lo | 32 | 输出 | 执行阶段写入LO的值 |
| 21 | mem\_whilo | 1 | 输出 | 执行阶段是否要写HI/LO |

关键代码

|  |
| --- |
| 模块：ex\_mem接口 |
| 1. mem\_wd <= ex\_wd;  2. mem\_wreg <= ex\_wreg;  3. mem\_wdata <= ex\_wdata;  4. mem\_hi <= ex\_hi;  5. mem\_lo <= ex\_lo;  6. mem\_whilo <= ex\_whilo;  7. hilo\_o <= {`ZeroWord, `ZeroWord};  8. cnt\_o <= 2'b00;  9. mem\_aluop <= ex\_aluop;  10. mem\_mem\_addr <= ex\_mem\_addr;  11. mem\_reg2 <= ex\_reg2; |

#### 寻址mem模块

MEM模块是处理器流水线中的访存阶段模块，主要负责执行指令涉及的访存操作，同时处理其他非访存相关的数据传递任务综合结果，主要流程如下。

* 1. 接收来自 EX 模块的运算结果、访存地址和控制信号。
  2. 根据控制信号决定是否进行数据存储器的读写操作。
  3. 对于读操作，将数据从 data\_ram 中取出；对于写操作，将数据写入指定地址。
  4. 将访存结果或运算结果传递至写回阶段，同时处理可能的异常或中断。
  5. 统计异常信息，传输值CP0\_reg与ctrl中

表 9 mem接口描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | rst | 1 | 输入 | 复位信号 |
| 2 | wd\_i | 12 | 输入 | 写入寄存器地址 |
| 3 | wreg\_i | 1 | 输入 | 是否写入寄存器 |
| 4 | wdata\_i | 32 | 输入 | 写入寄存器数据 |
| 5 | wd\_o | 32 | 输出 | 写入寄存器地址 |
| 6 | wreg\_o | 1 | 输出 | 是否写入寄存器 |
| 7 | wdata\_o | 32 | 输出 | 写入寄存器数据 |
| load/store相关接口 | | | | |
| 8 | aluop\_i | 6 | 输入 | 运算类型 |
| 9 | mem\_addr\_i | 32 | 输入 | 访存地址 |
| 10 | reg2\_i | 32 | 输入 | 送入存储器的数 |
| RAM相关接口 | | | | |
| 11 | mem\_data\_i | 32 | 输入 | 来自RAM的读取数据 |
| 12 | mem\_addr\_o | 32 | 输出 | 写入RAM的地址 |
| 13 | mem\_we\_o | 1 | 输出 | RAM读还是写 1写0读 |
| 14 | mem\_sel\_o | 4 | 输出 | 字节选择信号 |
| 15 | mem\_data\_o | 32 | 输出 | 存入RAM数据 |
| 16 | mem\_ce\_o | 1 | 输出 | 使能信号 |
| HI/LO相关接口 | | | | |
| 17 | hi\_i | 32 | 输入 | HI寄存器内容 |
| 18 | lo\_i | 32 | 输入 | LO寄存器内容 |
| 19 | hi\_o | 32 | 输出 | 执行阶段写入HI的值 |
| 20 | lo\_o | 32 | 输出 | 执行阶段写入LO的值 |
| 21 | whilo\_o | 1 | 输出 | 执行阶段是否要写HI/LO |

取数指令(LW)代码部分

|  |
| --- |
| 模块：取数指令（LW） |
| 1. `EXE\_LW\_OP:     begin  2.     mem\_addr\_o <= mem\_addr\_i;  3.     mem\_we <= `WriteDisable;  4.     wdata\_o <= mem\_data\_i;  5.     mem\_sel\_o <= 4'b1111;  6.     mem\_ce\_o <= `ChipEnable;  7. end |

存数指令(SW)部分

|  |
| --- |
| 模块：存数指令(SW) |
| 1. `EXE\_SW\_OP:     begin  2.     mem\_addr\_o <= mem\_addr\_i;  3.     mem\_we <= `WriteEnable;  4.     mem\_data\_o <= reg2\_i;  5.     mem\_sel\_o <= 4'b1111;  6.     mem\_ce\_o <= `ChipEnable;  7. end |

#### 锁存mem\_wb模块

mem\_wb 模块是处理器流水线中的访存到写回（Memory to Write Back）阶段的流水线寄存器。它的主要作用是从访存阶段 (MEM) 接收数据，并将其保存和传递到写回阶段 (WB)。这个模块在流水线设计中起着承上启下的作用，确保数据正确、有序地流向下一阶段。

表 10 mem\_wb模块

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | rst | 1 | 输入 | 复位信号 |
| 2 | clk | 1 | 输入 | 时钟信号 |
| 3 | stall | 6 | 输入 | 流水线暂停 |
| 4 | mem\_hi | 32 | 输入 | 访存时是否要写HI |
| 5 | mem\_lo | 32 | 输入 | 访存时是否要写LO |
| 6 | mem\_whilo | 1 | 输入 | 访存时是否写HI/LO |
| 7 | wb\_hi | 32 | 输出 | 回写时是否要写HI |
| 8 | wb\_lo | 32 | 输出 | 回写时是否要写LO |
| 9 | wb\_whilo | 1 | 输出 | 回写时是否写HI/LO |
| 数据相关解决方案相关接口 | | | | |
| 10 | mem\_wd | 5 | 输入 | 写入寄存器地址 |
| 11 | mem\_wreg | 1 | 输入 | 是否写入寄存器 |
| 12 | mem\_wdata | 32 | 输入 | 写入寄存器数据 |
| 13 | wb\_wd | 5 | 输出 | 写入寄存器地址 |
| 14 | wb\_wreg | 1 | 输出 | 是否写入寄存器 |
| 15 | wb\_wdata | 32 | 输出 | 写入寄存器数据 |

关键代码实现如下：

|  |
| --- |
| Mem\_wb模块 |
| 1. wb\_wd <= mem\_wd;  2. wb\_wreg <= mem\_wreg;  3. wb\_wdata <= mem\_wdata;  4. wb\_hi <= mem\_hi;  5. wb\_lo <= mem\_lo;  6. wb\_whilo <= mem\_whilo; |

#### 寄存器regfile模块

regfile 模块是 CPU 的寄存器堆，包含 32 个 32 位通用寄存器，支持两个寄存器的并发读操作和一个寄存器的同步写操作。

1. 定义寄存器数组：采用二维向量实现，数组大小为 32×32 位，表示 32 个 32 位寄存器。
2. 写寄存器操作：
   1. 在复位信号无效 (rst==0) 且写使能信号有效 (we==1)，同时写入目标寄存器地址不为 0 时，将输入数据写入指定寄存器。
   2. 写操作为同步写，即在时钟信号上升沿时完成。
3. 第一个读寄存器端口：
   1. 复位信号有效时，输出固定为 0。
   2. 复位信号无效时，读取 $0 时输出为 0。
   3. 若读取目标寄存器与写入目标寄存器地址相同，直接输出待写入数据（写后读冲突解决）。
   4. 其他情况下，从寄存器数组中读取对应地址的数据。
   5. 若读端口不可用，输出固定为 0。
4. 第二个读寄存器端口：与第一个读端口逻辑相同，复用同样的判断流程。

表 11 regfile接口描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | rst | 1 | 输入 | 复位信号 |
| 2 | clk | 1 | 输入 | 时钟信号 |
| 3 | re1 | 1 | 输入 | 读使能信号1 |
| 4 | raddr1 | 5 | 输入 | 读取的寄存器地址1 |
| 5 | re2 | 1 | 输入 | 读使能信号2 |
| 6 | raddr2 | 5 | 输入 | 读取的寄存器地址2 |
| 7 | we | 1 | 输入 | 写使能信号 |
| 8 | waddr | 5 | 输入 | 写入的寄存器地址 |
| 9 | wdata | 32 | 输入 | 写入的数据 |
| 10 | rdata1 | 32 | 输出 | 读出的32位数据1 |
| 11 | rdata2 | 32 | 输出 | 读出的32位数据2 |

关键代码如下:

|  |
| --- |
| 模块：regfile |
| 1. always @ (\*) begin  2.     if(rst == `RstEnable) begin  3.             rdata1 <= `ZeroWord;  4.     end else if(raddr1 == `RegNumLog2'h0) begin  5.         rdata1 <= `ZeroWord;  6.     end else if((raddr1 == waddr) && (we == `WriteEnable)  7.                 && (re1 == `ReadEnable)) begin  8.         rdata1 <= wdata;  9.     end else if(re1 == `ReadEnable) begin  10.         rdata1 <= regs[raddr1];  11.     end else begin  12.         rdata1 <= `ZeroWord;  13.     end  14. end |

#### 控制ctrl模块

ctrl 模块是流水线 CPU 的控制中心，用于管理流水线的状态，控制流水线暂停（stall）、清空（flush），以及处理异常和中断信号。它在确保数据流正确性、处理异常以及优化指令执行过程中起着关键作用。

1. 根据来自各阶段的暂停请求信号（如 stallreq\_from\_id, stallreq\_from\_ex），决定是否需要暂停流水线的部分或全部阶段。
2. 通过生成 stall 信号，指示流水线中的每个阶段是继续工作还是保持当前状态.

表 12 ctrl模块接口描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | rst | 1 | 输入 | 复位信号 |
| 2 | stallreq\_from\_id | 1 | 输入 | 译码阶段流水线暂停请求 |
| 3 | stallreq\_from\_ex | 1 | 输入 | 执行阶段流水线暂停请求 |
| 4 | stall | 6 | 输出 | 流水线暂停信号 |

关键代码如下：

|  |
| --- |
| 模块：ctrl |
| 1. always @ (\*) begin  2.     if(rst == `RstEnable) begin  3.         stall <= 6'b000000;  4.     end else if(stallreq\_from\_ex == `Stop) begin  5.         stall <= 6'b001111;  6.     end else if(stallreq\_from\_id == `Stop) begin  7.         stall <= 6'b000111;  8.         flush <= 1'b0;  9.         stall <= 6'b000000;  10.         new\_pc <= `ZeroWord;  11.     end    //if  12. end      //always |

#### 乘商hilo寄存器

hilo 模块是处理器中用于保存乘法和除法操作结果的特殊寄存器单元。该模块包含两个寄存器，分别为 HI 和 LO，通常用于存储乘法结果的高 32 位和低 32 位，或者除法操作的余数和商。hilo 模块在流水线中用于实现算术指令的结果传递。

表 13 hilo接口描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | rst | 1 | 输入 | 复位信号 |
| 2 | clk | 1 | 输入 | 时钟信号 |
| 3 | we | 1 | 输入 | 写使能信号 |
| 4 | hi\_i | 32 | 输入 | 输入的HI值 |
| 5 | lo\_i | 32 | 输入 | 输入的LO值 |
| 6 | hi\_o | 32 | 输入 | 输出的HI值 |
| 7 | lo\_o | 32 | 输出 | 输出的LO值 |

关键代码如下：

|  |
| --- |
| 模块：hilo寄存器 |
| 1. always@(posedge clk) begin  2.     if(rst==`RstEnable) begin  3.         hi\_o<=`ZeroWord;  4.         lo\_o<=`ZeroWord;  5.     end else if( we==`WriteEnable)begin  6.         hi\_o<= hi\_i;  7.         lo\_o<= lo\_i;  8.     end  9. end |

#### inst\_rom模块

inst\_rom 模块是一个指令存储器 (Instruction ROM)，用于存储处理器运行所需的指令。它作为处理器指令取指阶段 (IF) 的输入源，在处理器启动或运行时提供指令数据。只提供读端口，没有写入端口

1. 当 ce 为无效时，输出空指令 (NOP 或 ZeroWord)，表示当前没有指令被读取。
2. 根据 addr 从指令存储器中取出对应的指令数据。

表 14 inst\_rom模块接口描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | ce | 1 | 输入 | 使能信号 |
| 2 | addr | 32 | 输入 | 要读取的指令地址 |
| 3 | inst | 32 | 输出 | 读出的指令 |

关键代码如下：

|  |
| --- |
| 模块：inst\_rom模块 |
| 1. initial begin  2.     $readmemh("inst\_rom.data", regs);  3. end  4. always@(\*)begin  5.     if(ce==0) inst<=31'h0;  6.     else inst<=regs[addr[31:2]];  7. end |

#### 数据存储data\_ram模块

在设计中使用 4x8位存储器代替一个32位存储器，是一种灵活且资源高效的实现方式

1. 地址 addr 被分为两部分：
   1. addr[DataMemNumLog2+1:2]：用于计算行号，确定访问哪一行。
   2. sel：用于计算列号，选择当前哪个存储器（0-3）。
2. 读写操作:
   1. 从 4 个 8 位存储器中，由sel信号选择读写的8位存储器。
   2. 将读取的字节按照sel选择方法拼接，组成一个完整的 32 位数据输出。
   3. 写入时，按照sel选择的8位存储器将mem中拼接的原始数据写入。

表 15 data\_ram接口描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度（bit） | 输入/输出 | 作用 |
| 1 | ce | 1 | 输入 | 使能信号 |
| 2 | clk | 1 | 输入 | 时钟信号 |
| 3 | data\_i | 32 | 输入 | 要写入的数据 |
| 4 | addr | 32 | 输入 | 要读取的地址 |
| 5 | we | 1 | 输入 | 是否是写操作 |
| 6 | sel | 4 | 输入 | 字节选择信号 |
| 7 | data\_o | 32 | 输出 | 读出的数据 |

关键代码如下：

|  |
| --- |
| 模块：写入存储器 |
| 1. always @ (posedge clk) begin  2.     if (ce == `ChipDisable) begin  3.         //根据片选信号sel,将数据写存储器芯片  4.     end else if(we == `WriteEnable) begin  5.             if (sel[3] == 1'b1) begin  6.             data\_mem3[addr[`DataMemNumLog2+1:2]] <= data\_i[31:24];  7.         end  8.             if (sel[2] == 1'b1) begin  9.             data\_mem2[addr[`DataMemNumLog2+1:2]] <= data\_i[23:16];  10.         end  11.         if (sel[1] == 1'b1) begin  12.             data\_mem1[addr[`DataMemNumLog2+1:2]] <= data\_i[15:8];  13.         end  14.             if (sel[0] == 1'b1) begin  15.             data\_mem0[addr[`DataMemNumLog2+1:2]] <= data\_i[7:0];  16.         end  17.     end  18. end |

### 总结

在设计与优化 MIPS 五级流水线的过程中，数据相关、控制相关与结构相关是性能瓶颈的核心挑战。以下是针对这三类相关性问题的解决思路与实现方法：

#### 数据相关

流水线数据相关又分为三种情况：RAW、WAR、WAW。

对于五级流水CPU来说，只有在流水线回写阶段才会写寄存器，因此不存在WAW相关。又因为只能在流水线译码阶段读寄存器、回写阶段写寄存器，不存在WAR相关。所以五级流水CPU只存在RAW相关，RAW相关又有三种情况：

1. 相邻指令间存在数据相关。
2. 相隔1条指令的指令间存在数据相关。
3. 相隔2条指令的指令间存在数据相关。

##### 数据旁路(前推)

实现 ALU 结果的快速转发，将 EX/MEM 和 MEM/WB 阶段的结果直接回送给 ID 或 EX 阶段，减少数据等待周期。具体实现时，通过比较目标寄存器地址和源寄存器地址，选择性转发。如图中所示

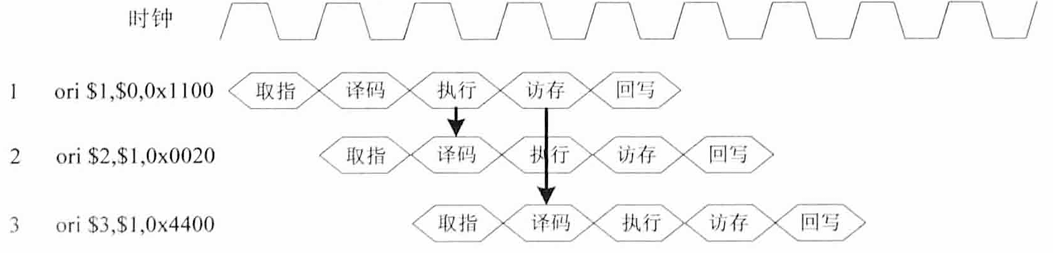


图 22 数据前推解决相邻指令数据相关问题

同时，在回写阶段会发生第三种数据相关情况，在寄存器设计避免读写冲突时候已经间接解决回写时候访存问题，此时将写入数据直接传入读取数据，即可解决读写冲突与数据相关问题。

##### 指令暂停

对于LOAD指令，指令结果在访存时候才可以获得，此时，相邻的一条指令已进入运算阶段，此时使用数据旁路已经来不及将数据传入源操作数中，所以当进入LOAD指令时，需暂停下一条指令的执行。

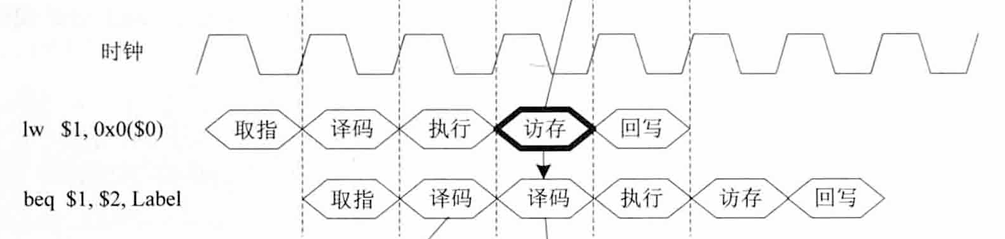


图 23 LOAD指令数据相关时指令暂停

当检测到LOAD指令时，暂停流水线以下行为：PC（程序计数器）保持当前值，阻止取指ID/EX流水寄存器保持不变，防止错误指令流入执行阶段。插入一个 NOP（无操作指令）以延迟后续指令执行，等待加载数据写回。

经过一个周期后，LOAD指令的数据写回寄存器，后续指令可以正常使用该数据。

#### 控制相关

控制相关（Control Hazard）指由于分支指令或跳转指令的目标地址在流水线运行过程中无法及时确定，导致指令流水线中断或延迟的问题。例如，在跳转指令执行前，处理器无法提前知道下一条需要执行的指令地址。

使用延迟槽解决控制相关，通过在跳转指令后插入可执行的指令，将跳转目标计算和更新的延迟周期转化为有意义的执行周期。

1. bne $1,$0,s2          #如果$1和$0不相等则跳转到s2

2.     nop          #2c

3.     lui $1,0xf760

4.     lui $2,0x1234

5. s2:

6.     nop

7.     nop         #3c

8.     beq $2,$1,s1             #如果$2和$1相等则跳转到s1

9.     nop

10.     slt $9,$1,$2

如上述代码中所示，bne $1,$0,s2后的nop指令为延迟槽指令，在后续的仿真中我们可以看到，即使发生了转移，转移指令bne后的延迟槽指令仍然执行，这是一种为减少气泡、提高效率而进行的控制冒险。

这种方法在早期处理器中有效减少了流水线停顿，提高了指令执行效率。不过，这种方法需要采用编译器优化，选择合适的指令加入在延迟槽中。

#### 结构相关

结构冲突（Structural Hazard）指由于硬件资源不足，导致多条指令在同一时钟周期内无法同时访问同一资源。例如，在单存储器架构中，取指（IF 阶段）和数据访问（MEM 阶段）需要同时访问存储器，导致冲突。

此次设计的CPU采用哈佛结构，可以有效地避免结构相关的冲突问题。

* 1. 独立的指令存储器和数据存储器：指令存储器仅用于存储和读取指令，而数据存储器仅用于存储和访问数据。
  2. 独立的总线：两套存储器分别配有独立的读写通路，可并行访问，无需排队。

将取指与访存的使用的存储器结构分为两个存储器，实现取指和数据访存的并行操作，提升流水线性能，同时加快了数据读写的速度。

# 【系统测试】

## 数字跑表系统测试

### 数字跑表的功能仿真

下面对数字跑表的仿真波形图进行分析。根据功能，数字跑表外连接口为reset，start\_stop信号，所以在仿真测试中，将测试reset置位功能，start\_stop暂停与开始功能

#### reset置位

当 reset 按键 按下后，数字跑表的计时信号需要置为 0，包括 百分秒、秒、和 分钟 的计时信号。仿真波形图如图中所示。

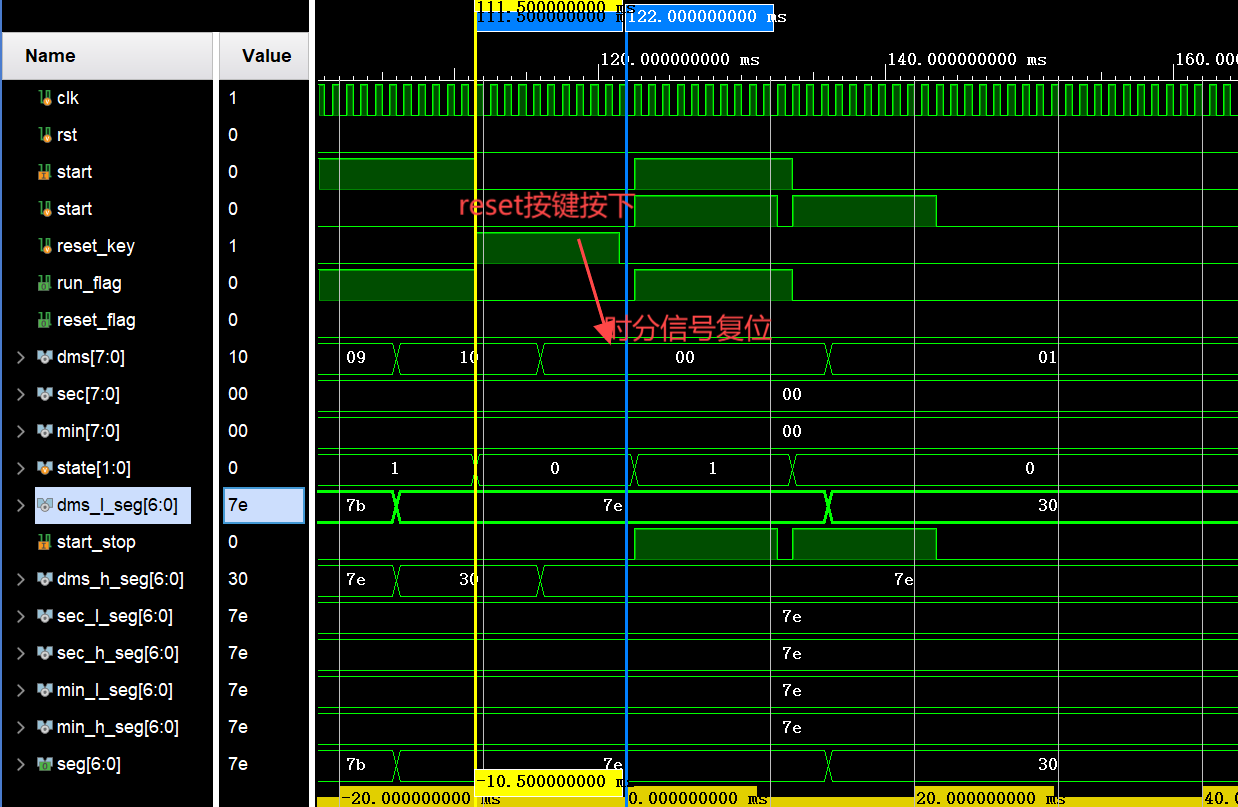


图 24 数字跑表reset按键仿真波形

当reset\_key信号出现一个上升沿（仿真reset按键按下）时，时钟信号复位。

1. 百分秒dms，原数据10，复位为0；
2. 秒sec，原数据0，复位为0；
3. 分钟min，原数据0，复位为0；

#### start\_stop暂停按键

按下 start\_stop 按键时，计时器从当前时间开始计时。初始状态下，计时从零开始。再次按下 start\_stop 按键时，计时器暂停计时，保持当前时间值不变。暂停期间显示模块显示静止时间。仿真波形图如图中所示。

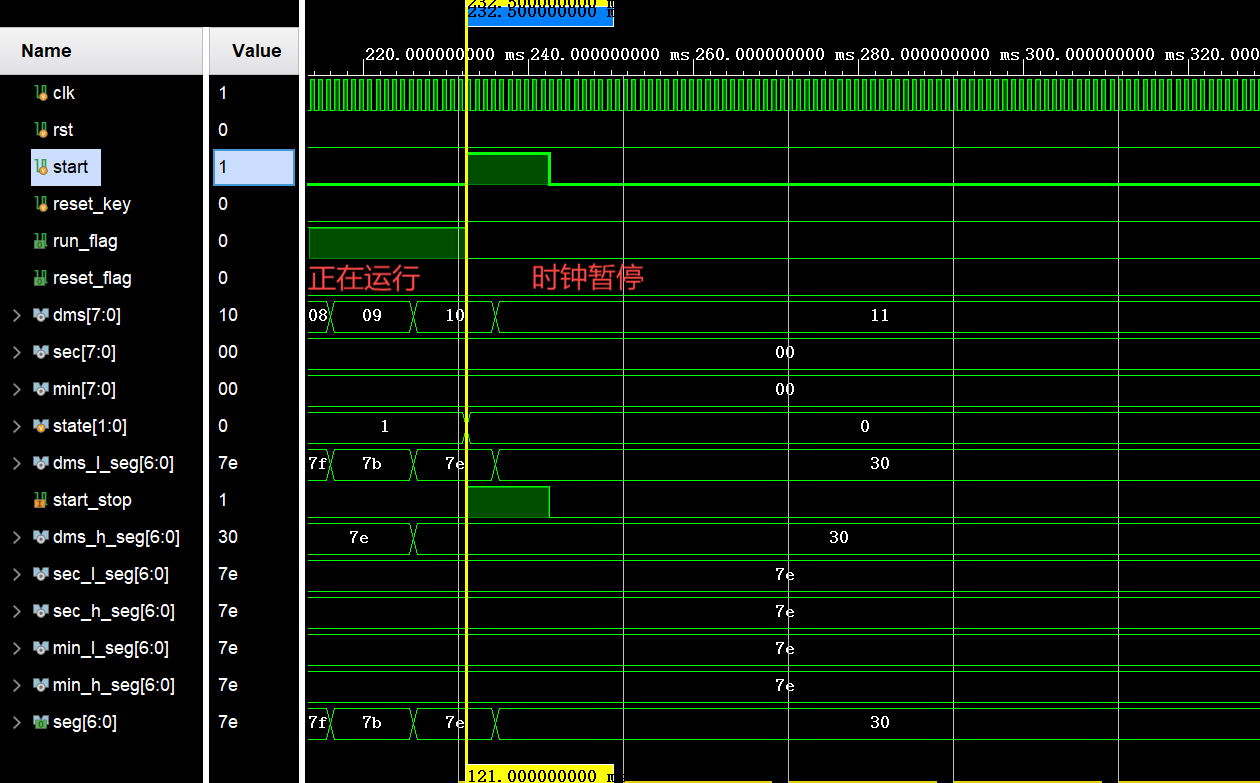


图 25 start\_stop按键暂停

当此时状态为运行状态时，start\_stop信号出现一个上升沿（仿真reset按键按下）时，时钟信号暂停。

1. 百分秒dms，原数据11，后续暂停为11；
2. 秒sec，原数据0，后续暂停为0；
3. 分钟min，原数据0，后续暂停为0；

#### start\_stop开始按键

按下 start\_stop 按键时，计时器从当前时间开始计时。初始状态下，计时从零开始。再次按下 start\_stop 按键时，计时器暂停计时，保持当前时间值不变。暂停期间显示模块显示静止时间。仿真波形图如图中所示。

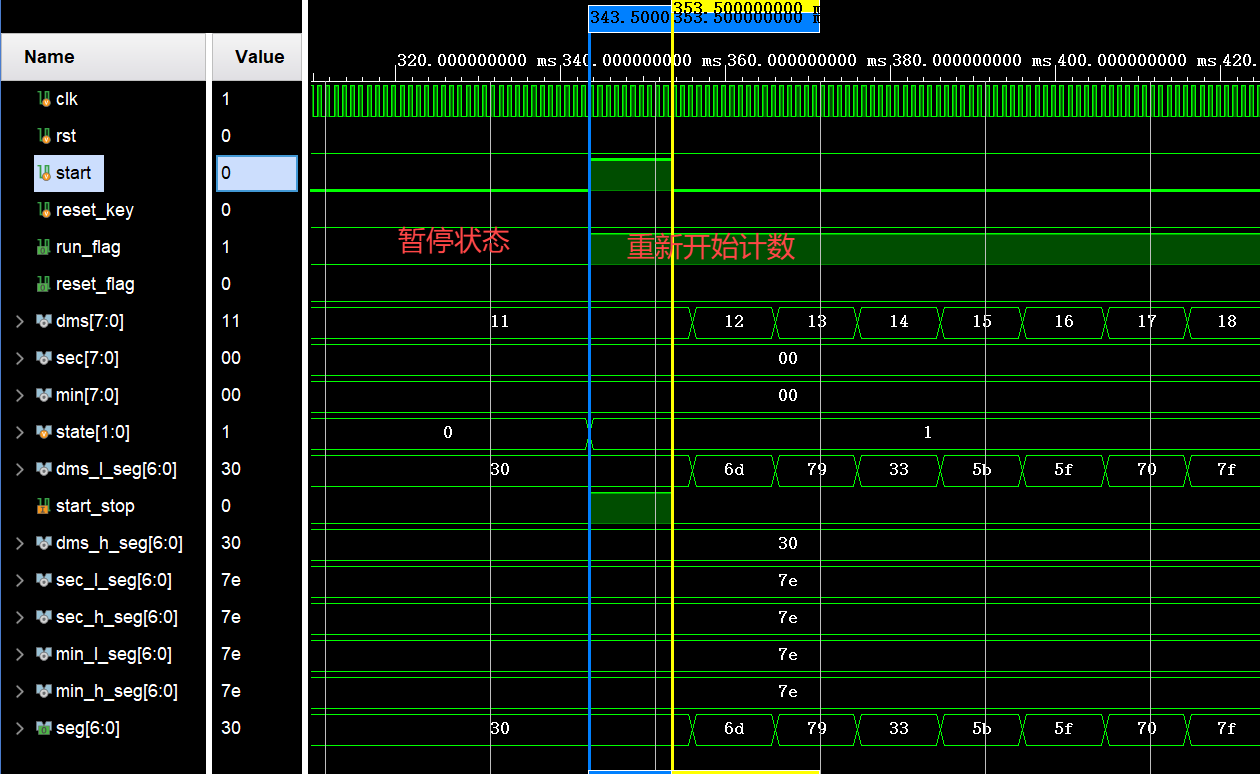


图 26 start\_stop开始按键

1. 百分秒dms，原数据11，后续继续计数为12；
2. 秒sec，原数据0，后续继续计数；
3. 分钟min，原数据0，后续继续计数.

### 数字跑表的RTL级逻辑电路

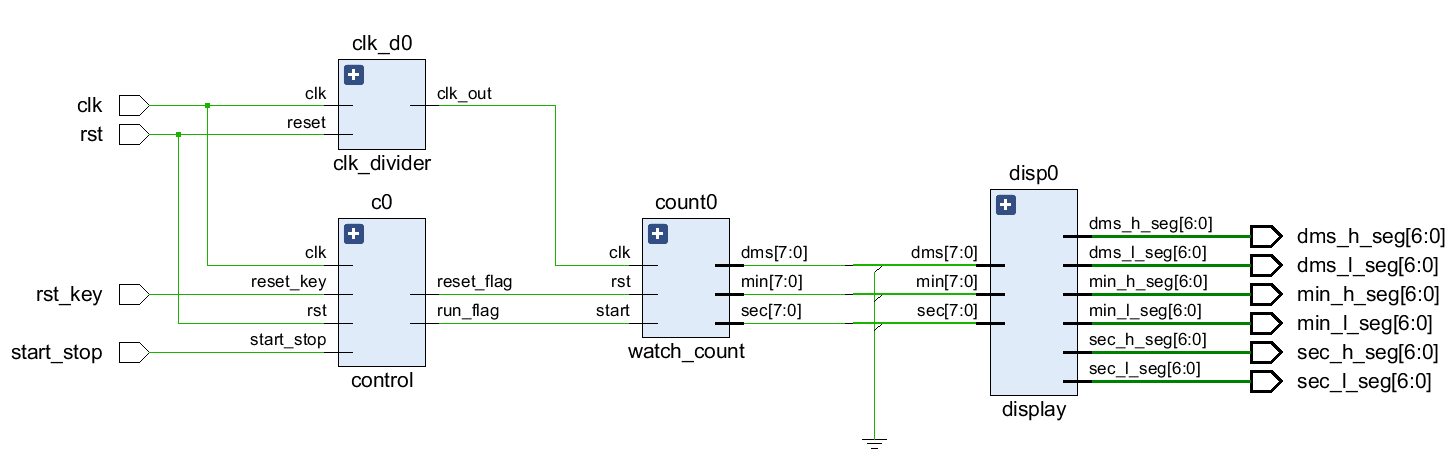


图 27 数字跑表的RTL级逻辑电路

## 五级流水系统测试

### 五级流水CPU的功能仿真

下面将以课设文件的func\_test1进行仿真测试。

表 16 func\_test1汇编指令与十六进制机器指令

|  |  |
| --- | --- |
| 汇编程序 | 对应机器指令 |
| .org 0x0  .set noat  .set noreorder  .global \_start  \_start:  lui $1,0x1234 #$1=0x12340000  lui $2,0x5678 #$2=0x56780000  nop  nop #c  s1:  add $3,$1,$2 #$3=0x68ac0000  sub $4,$1,$2 #$4=0xbbbc0000  or $5,$1,$2 #$5=0x567c0000  and $6,$1,$2 #1c #$6=0x12300000  xor $7,$1,$2 #$7=0x444c0000  nor $8,$1,$2 #$8=0xa983ffff  bne $1,$0,s2 #如果$1和$0不相等则跳转到s2  nop #2c  lui $1,0xf760  lui $2,0x1234  s2:  nop  nop #3c  beq $2,$1,s1 #如果$2和$1相等则跳转到s1  nop  slt $9,$1,$2 #$9 =0x00000001  sltu $10,$1,$2 #4c $10=0x00000001  sll $11,$1,2 #$11=0x48d00000  srl $12,$1,16 #$12=0x00001234  sra $13,$1,2 #$13=0x048d0000  addu $14,$1,$2 #5c $14=0x68ac0000  addiu $15,$1,0x2020 #$15=0x12342020  subu $16,$1,$2 #$16=0xbbbc0000  sltu $17,$1,$2 #$17=0x00000001  sw $14,0x8($0) #6c #将$14的值(0x68ac0000)存入物理地址为0x00000008的字  lw $18,0x8($0) #$18=0x68ac0000  jal 0x80 #$31=0x0000007c  nop  addiu $1,$0,0x2020 #$1=0x00002020  addiu $2,$0,0x2010 #80 $2=0x00002010  addiu $3,$0,0x00a0 #$3=0x000000a0  nop  nop  jr $3 #90 #跳转到$3所指位置  nop  add $3,$1,$2  sub $4,$1,$2  or $5,$1,$2 #a0 #jr target 0xa0  lui $31,0x0101 #$31=0x01010000  finish:  nop  nop | 3c011234  3c025678  00000000  00000000  00221820  00222022  00222825  00223024  00223826  00224027  14200003  00000000  3c01f760  3c021234  00000000  00000000  1041fff3  00000000  0022482a  0022502b  00015880  00016402  00016883  00227021  242f2020  00228023  0022882b  ac0e0008  8c120008  8c000020  00000000  24012020  24022010  240300a0  00000000  00000000  00600008  00000000  00221820  00222022  00222825  3c1f0101  00000000  00000000 |

首先，我们预先观察五级流水仿真的整体情况以及指令运行的结果，指令结果最终运算结果保存在5号与31号寄存器中;

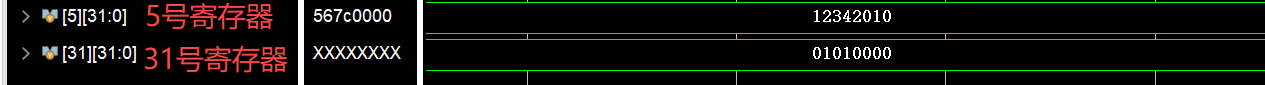


图 28 Func\_test1指令运算结果

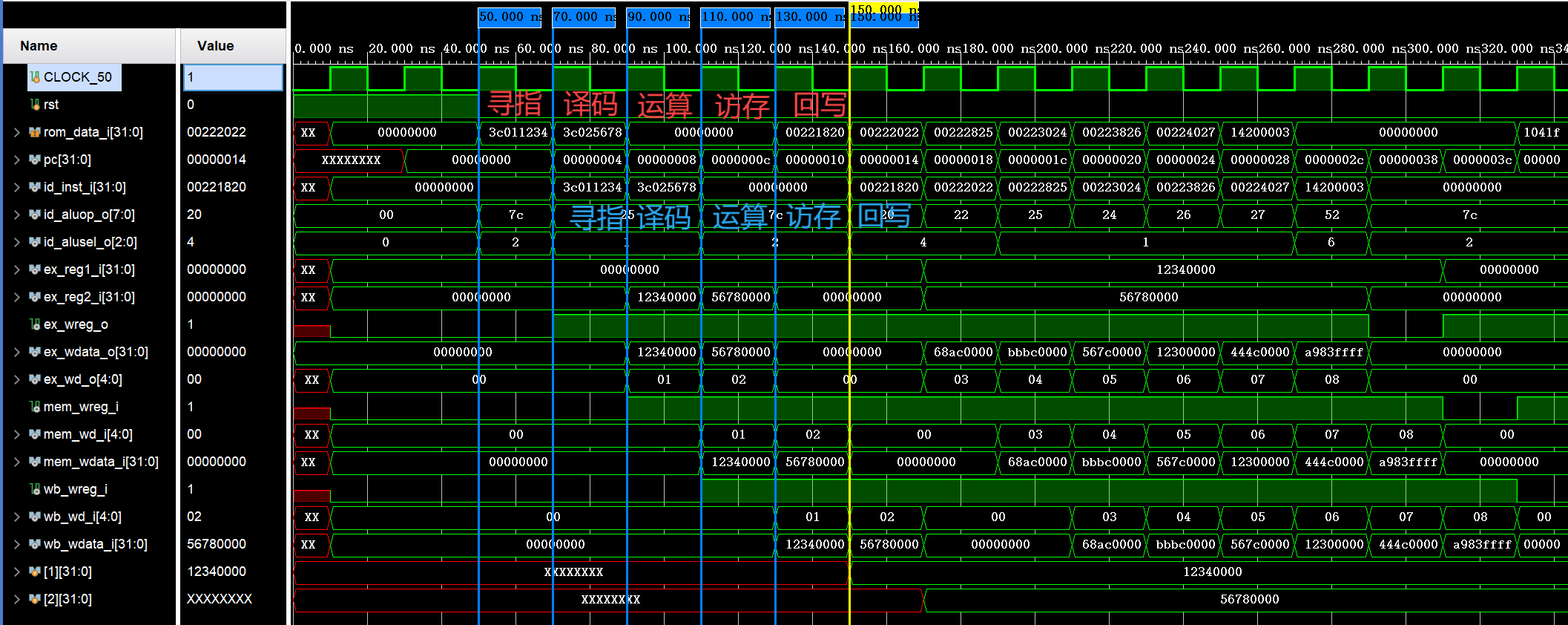


图 29 五级流水仿真波形图

下面对 func\_test1 中指令的仿真波形图进行分析。根据功能，指令可以分为以下五类：算术运算指令、立即数指令、访存指令、转移指令 和 延迟槽指令。以下将针对这五种类型分别进行分析：

#### 算术运算指令add $3,$1,$2:

分析涉及基本的算术操作（如加法、减法等）对寄存器的影响，重点观察寄存器内容变化以及计算结果是否符合预期。

指令含义：将寄存器 $1 和 $2 的值相加，并将结果存入寄存器 $3。仿真波形如下：

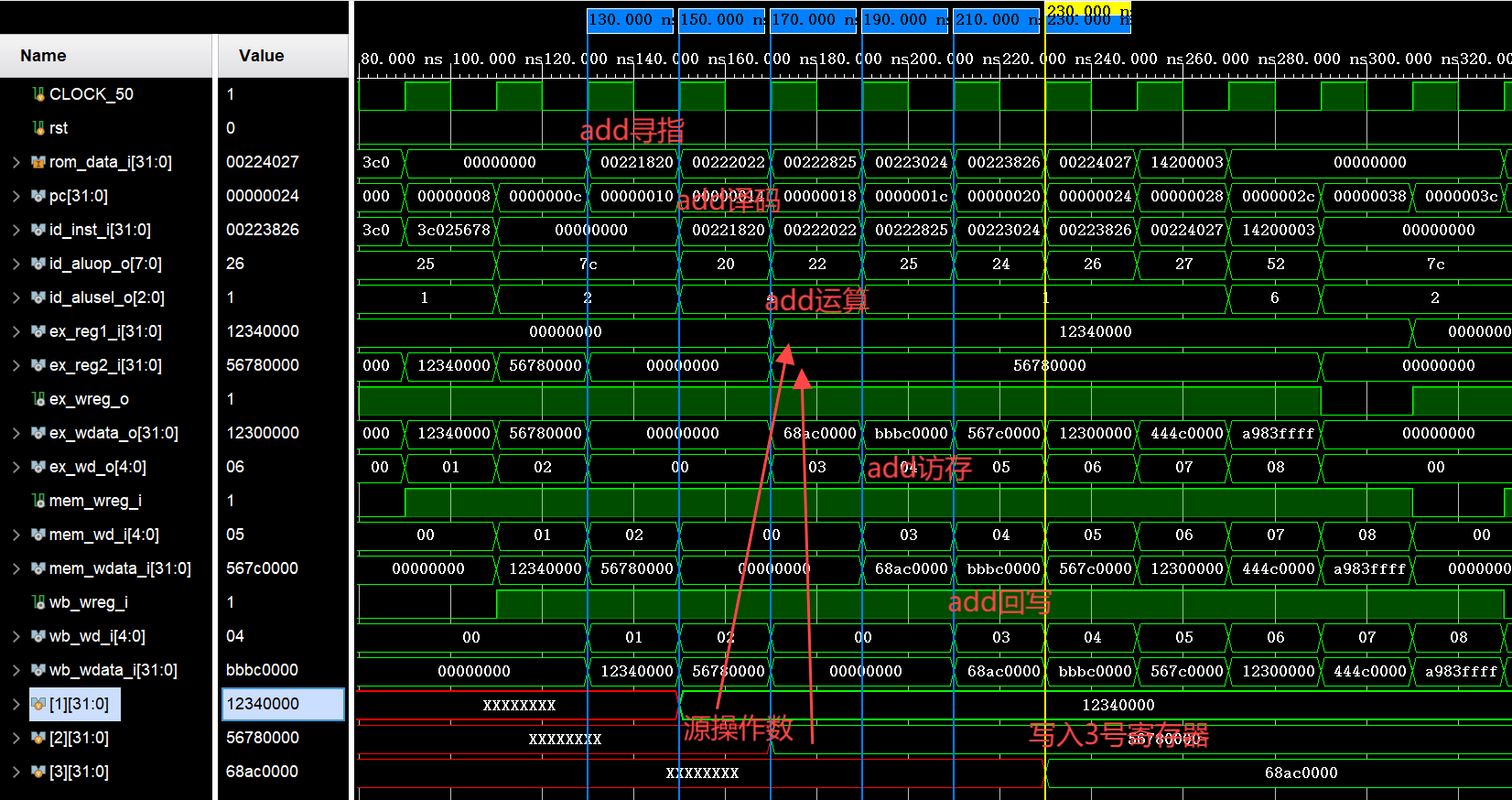


图 30 算术指令仿真波形图

1. IF 操作：取指令 add $3, $1, $2（00221820）。结果：指令从指令存储器中加载到流水线。
2. ID 操作：解析指令，读取 $1 = 0x12340000 和 $2 = 0x56780000。结果：生成控制信号，准备操作数。
3. EX操作：ALU 执行加法 $3 = $1 + $2 = 0x68AC0000。结果：计算结果 0x68AC0000 写入流水线寄存器。
4. MEM 操作：访存阶段，无内存操作。结果：结果保持，直接进入写回阶段。
5. WB 操作：将结果 0x68AC0000 写入寄存器 $3。结果：寄存器 $3 = 0x68AC0000。

#### 立即数指令lui $1,0x1234

针对操作数包含立即数的指令，分析立即数在指令执行过程中是否被正确加载，寄存器结果是否符合指令定义。

指令含义：将立即数 0x1234 加载到寄存器 $1 的高 16 位，而低 16 位清零。仿真结果如下：

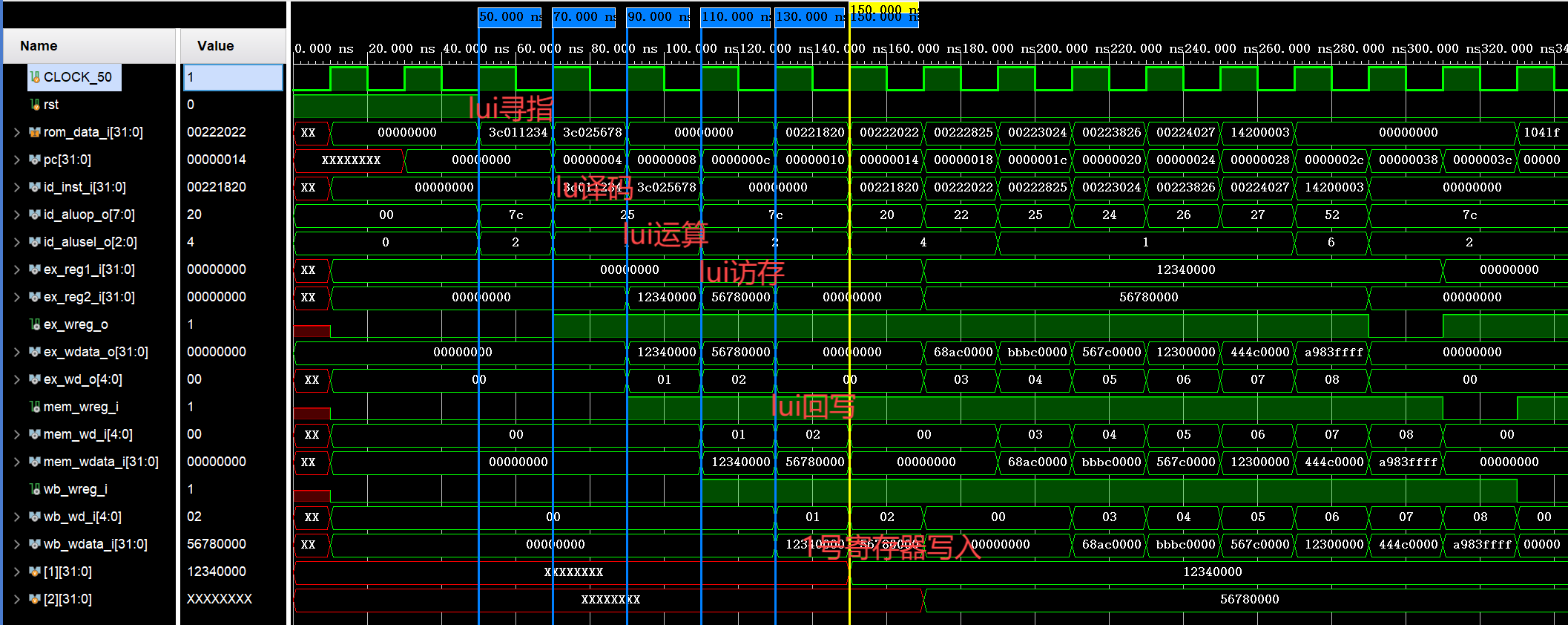


图 31 立即数指令仿真波形图

1. IF（取指阶段）。从指令存储器（inst\_rom）中取出指令3c011234PC（程序计数器）增加4，以准备取下一条指令。
2. ID（指令译码阶段）
   1. 对指令进行译码，识别为 lui 指令。
   2. 解析操作码，提取目标寄存器 $1 和立即数 0x1234。
   3. 控制信号生成：准备后续流水级所需的信号，如写寄存器使能、ALU 操作类型等。
3. EX（执行阶段）。ALU 将立即数 0x1234 左移 16 位，形成 0x12340000。ALU 结果准备写入目标寄存器。
4. MEM（访存阶段）。此指令无需访问数据存储器，因此该阶段被跳过或直接通过。
5. WB（写回阶段）。将 EX 阶段计算的结果 0x12340000 写入寄存器 $1。

#### 访存指令sw $14,0x8($0)

主要关注对存储器的访问操作，分析数据存取的正确性，例如加载指令（Load）和存储指令（Store）对内存和寄存器的影响。

指令含义：将$14的值(0x68ac0000)存入物理地址为0x00000008的字。仿真波形图如下：

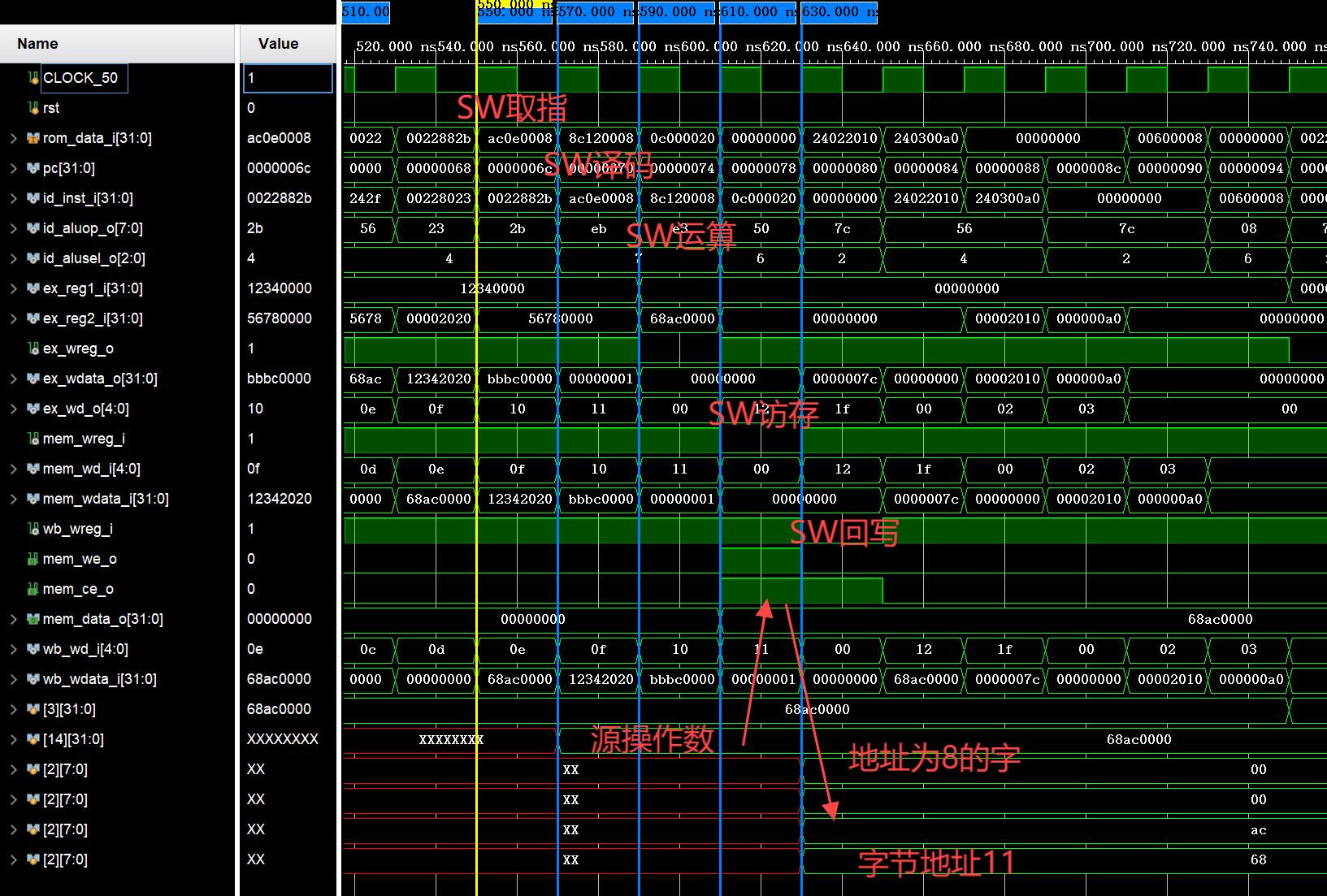


图 32 访存指令仿真波形图

1. IF（取指阶段）：从指令内存中取出指令 ac0e0008。获取当前 PC 指向的指令。
2. ID（指令解码阶段）：
   1. 解码指令并读取寄存器 $0 和 $14 的值：$0 = 0x00000000，$14 = 0x68ac0000。
   2. 计算目标地址：0x00000000 + 0x8 = 0x8。
   3. 将目标地址和要存储的数据传递到 EX 阶段。
3. EX（执行阶段）：
   1. 地址计算已在 ID 阶段完成，传递数据到 MEM 阶段。
4. MEM（访存阶段）：
   1. 存储数据 0x68ac0000 到地址 0x8，按照大端格式拆分并写入内存：0x68 存储到地址 0x8。0xac 存储到地址 0x9。0x00 存储到地址 0xa。0x00 存储到地址 0xb。
5. WB（写回阶段）：sw 指令没有写回寄存器，只进行内存写操作，因此 WB 阶段不涉及任何数据写回

#### 转移指令bne $1,$0,s2及延迟槽分析

对条件跳转（Branch）和无条件跳转（Jump）指令进行分析，验证程序计数器（PC）是否按照预期发生了跳转。

指令含义：bne $1, $0, s2，它表示如果 $1 不等于 $0，则跳转到地址 s2，仿真波形图如下：

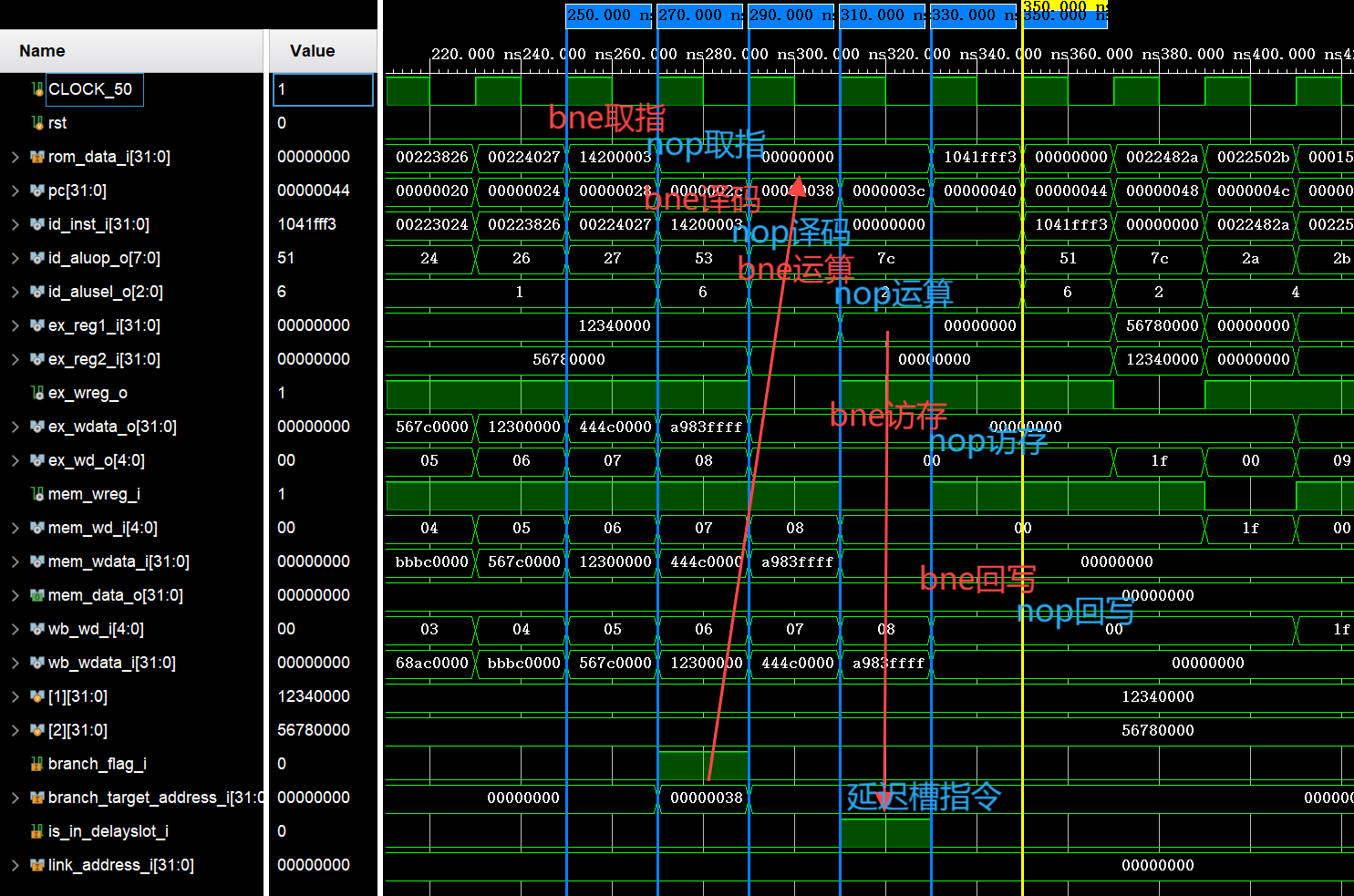


图 33 跳转指令与延迟槽指令分析

1. 指令的五级流水顺序分析：
   1. IF（取指阶段）：取指令 14200003，对应十六进制 14200003。PC 指向当前指令的地址（假设为 PC）。
   2. ID（指令解码阶段）：解码指令 bne，并读取 $1 和 $0 寄存器的值：$1 = 0x12340000，$0 = 0x00000000。比较 $1 和 $0 的值：$1 != $0，因此条件满足，跳转成立。计算跳转地址：s2 = 0x00000038。
   3. EX（执行阶段）：bne 指令的比较操作已经在 ID 阶段完成，跳转地址已计算出。在 EX 阶段，bne 指令将跳转地址传递到 MEM 阶段。
   4. MEM（访存阶段）：bne 指令在 MEM 阶段并不涉及内存读取或写入操作，只是继续传递跳转信息。因此，在 MEM 阶段，跳转目标地址 0x00000038 被传递到 WB 阶段。延迟槽指令 nop 也会被执行，但它并不影响跳转。
   5. WB（写回阶段）：bne 指令本身没有数据写回操作，因此 WB 阶段无操作。
2. 延迟槽指令：延迟槽指令：nop，即无操作，它在跳转指令后立即执行，用来消耗流水线中的周期，避免跳转时的流水线冲突。信号ex\_is\_in\_delayslot\_i为高电平，表示此时正在执行的指令nop为延迟槽指令。

可以看到信号ex\_aluop\_i=0x7c,ex\_aluop\_i=0x2，即延迟槽指令nop的运算类型。由此可见，即使发生了转移，转移指令bne后的延迟槽指令仍然执行，这是一种为减少气泡、提高效率而进行的控制冒险。

#### Func\_test2仿真结果展示

我们预先观察五级流水仿真的整体情况以及指令运行的结果，指令结果最终运算结果保存在18号与19号寄存器中，以及乘商hi、lo寄存器中。



图 34 Func\_test2指令仿真波形图

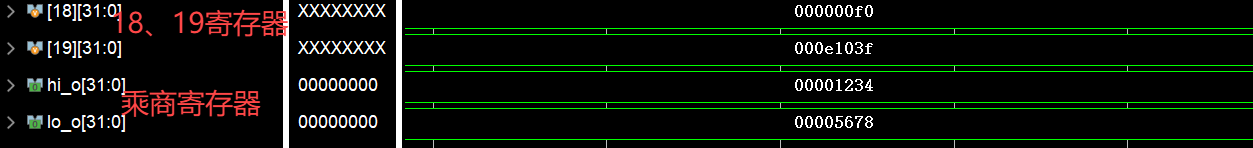


图 35 Func\_test2中18、19、hi、lo寄存器仿真结果

四种指令的波形在Func\_test1中完成分析，指令的仿真波形图可以参考Func\_test1中的指令分析结果。

##### 除法指令div $zero,$3,$1

下面介绍一条新的指令种类，除法指令div、divu，这种指令在执行过程中会暂停数据的执行，本次采用恢复余数法，div指令会占用32个时钟周期，在运算阶段中中断其余指令的取指、译码、执行。

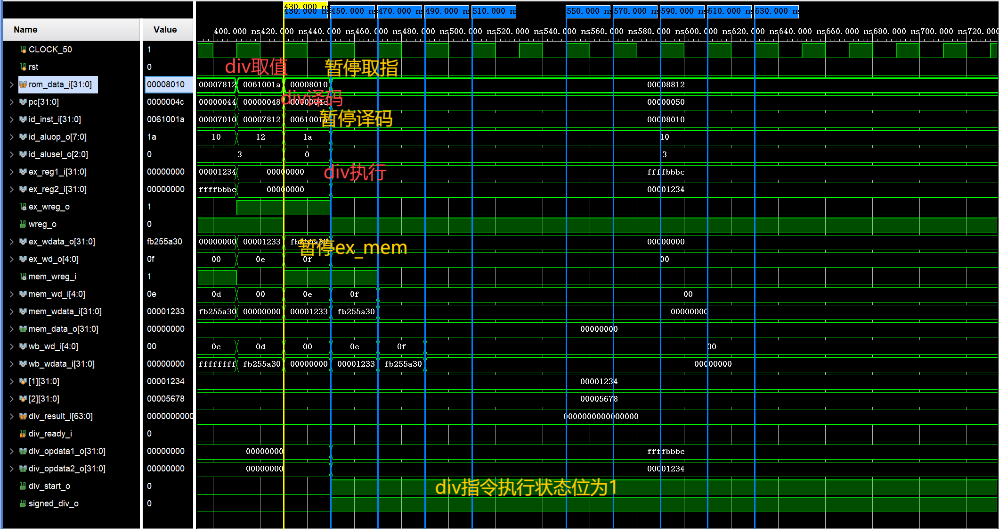


图 36 div指令执行仿真波形

1. 取指（IF）。从指令存储器中取出 div $zero, $3, $1 指令，将其送入流水线，PC 增加 4，为下一条指令做好准备。
2. 译码（ID）。解析指令为 MIPS 的整型除法操作。从寄存器文件中读取操作数：$3 = 0xFFFFBBBC（有符号值为 -17636）。$1 = 0x00001234（有符号值为 4660）。
3. 执行（EX）。
   1. 其余指令暂停32个时钟周期；
   2. 在执行阶段，进行有符号除法运算：商：-17636 ÷ 4660 = -4。余数：-17636 % 4660 = -2596。

4. 访存（MEM）无访存操作，此阶段直接将数据通过流水线传递到写回阶段。

5. 写回（WB）将计算的商和余数分别写入特殊寄存器：lo = 0xFFFFFFFC（-4 的二进制表示）。hi = 0xFFFFF5BC（-2596 的二进制表示）。

### 五级流水CPU的RTL级逻辑电路

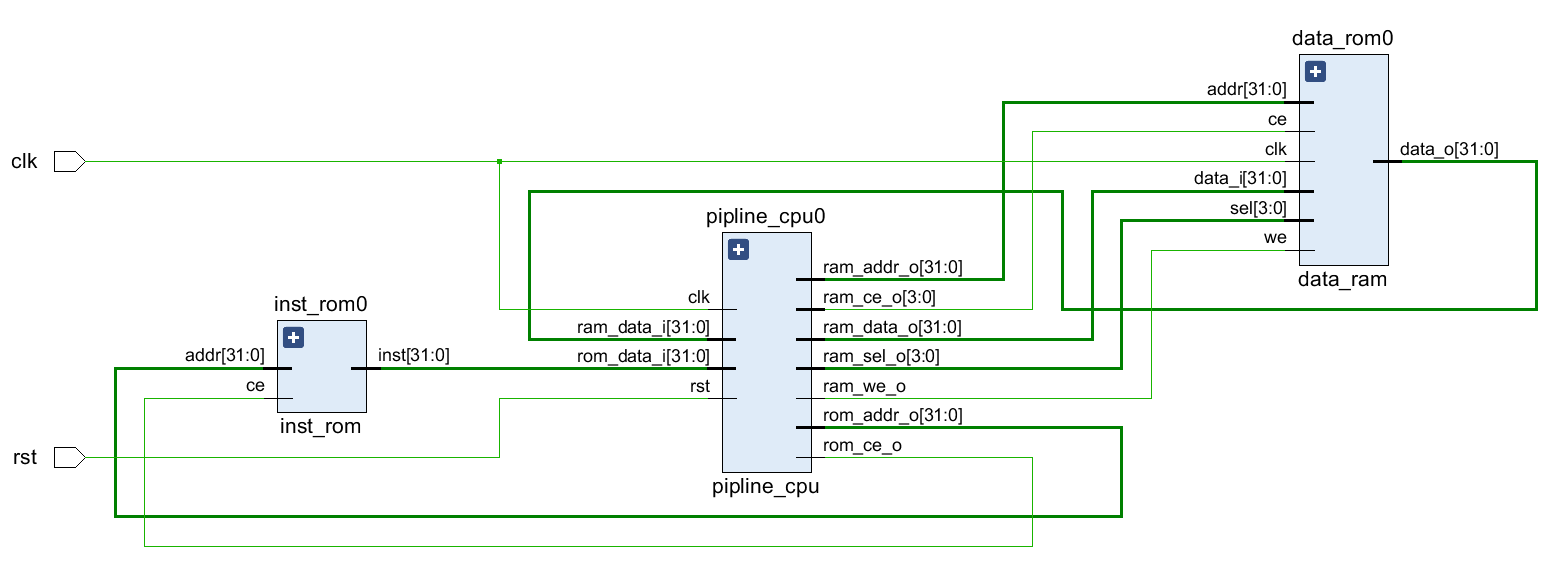


图 37 上层RTL级电路

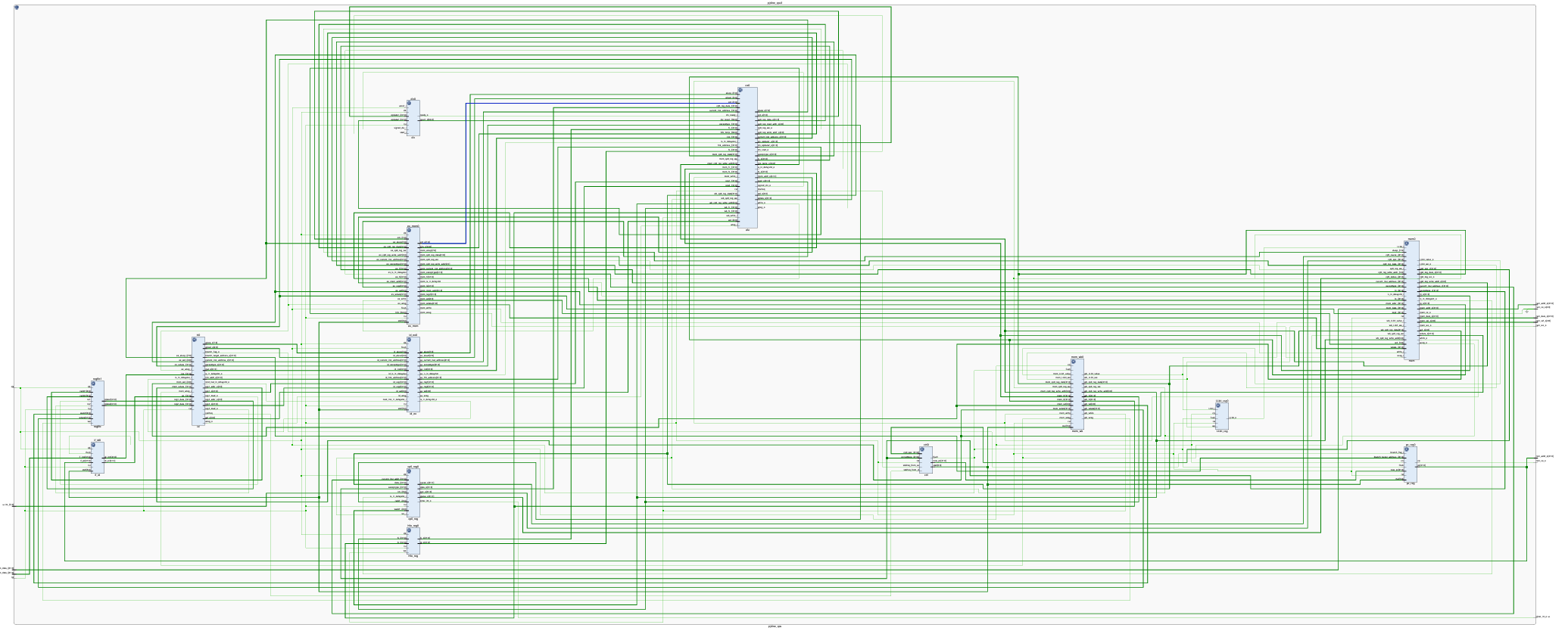


图 38 详细RTL级电路

# 【总结】

在设计与实现五级流水CPU和数字跑表的过程中，我们结合了数字电路设计和计算机体系结构的核心理念，完成了一次从理论到实践的完整探索。

## 数字跑表

数字跑表作为一种功能简单但逻辑复杂的计数器电路，实现了秒表计时和多功能控制。设计中通过状态机控制了计时开始、暂停、复位等功能，计数器模块则实现了从秒到毫秒的精确计时。核心亮点包括：

状态机设计：采用有限状态机（FSM）设计，简化了复杂逻辑的实现，通过状态转移控制计数器的启动和复位，确保了逻辑的稳定性。

模块化实现：计时模块和显示模块独立设计，通过分频技术实现时间单位的精确划分，并通过七段数码管驱动模块实现人性化的时间显示.

## 五级流水CPU

五级流水CPU的设计围绕取指（IF）、译码（ID）、执行（EX）、访存（MEM）、写回（WB）五个阶段展开。在设计中，我们重点解决了三类流水线相关问题：

数据相关：通过数据前推和指令暂停等策略有效解决了数据冒险。例如，对于LOAD指令的相关问题，采用暂停流水线的方法，确保下一条指令获取正确的数据。

控制相关：通过引入延迟槽和分支预测优化了跳转指令的执行效率。延迟槽技术延缓跳转指令的执行，使其执行期间的控制信号保持稳定，从而消除了控制冒险对流水线的影响。

结构相关：采用哈佛结构的指令存储器和数据存储器分离设计，彻底消除了访存阶段的结构冲突，提高了流水线的吞吐率。

整个CPU的模块化设计，包括寄存器堆（regfile）、ALU、控制模块（ctrl）、乘商寄存器（hilo）等子模块的协作，使指令执行的每个阶段都实现了高效和稳定。通过仿真与调试，我们验证了诸如算术运算指令、访存指令、分支跳转指令在五级流水中的正确性和性能表现。

本次设计不仅帮助我们深入理解了计算机体系结构与数字电路的设计原理，还培养了我们在复杂系统中发现问题、分析问题、解决问题的能力。从CPU流水线的冒险解决到跑表状态机的精确控制，每一步都离不开模块化的设计思维和严谨的逻辑验证。这种实践经验为后续深入研究嵌入式系统和数字电路打下了坚实的基础，同时也展现了数字逻辑设计的艺术与科学结合的魅力。

# 【心得体会】

在完成五级流水CPU和数字跑表的设计与实现过程中，我深刻体会到了将计算机电路设计与计算机组成原理的核心理念以及它们在实际工程应用中的意义。这次实践不仅帮助我掌握了技术细节，更让我在理论联系实际的过程中收获了成长。

五级流水CPU的设计是一个非常经典但也充满挑战的任务。从理论上看，流水线的分阶段执行提高了指令执行的吞吐量，但在实际实现中，各阶段之间的协调与资源冲突的解决却需要非常精细的设计。例如：

数据相关：数据冒险是流水线中最常见的问题之一。为了避免后续指令使用未更新的数据，我们引入了数据前推和指令暂停机制。这让我认识到，虽然硬件资源可以通过设计扩展，但调度策略的优化是提高效率的关键所在。

控制相关：跳转指令和分支指令的执行对流水线造成的控制冒险问题，通过延迟槽的设计得到了有效解决。在仿真的过程中，我意识到延迟槽虽然在实现上比较复杂，但它能极大提高流水线的连续性，这种取舍体现了设计中的工程智慧。

结构相关：采用哈佛结构解决了指令存储器与数据存储器的冲突问题。这一设计让我深刻体会到，硬件结构上的创新往往能从根本上解决性能瓶颈。

通过模块化的设计思路，我构建了寄存器堆、ALU、控制模块等多个子模块，并通过信号的传递与组合，实现了复杂功能的流水线执行。这种自下而上的设计方法让我更加理解了复杂系统的分解与集成过程。

数字跑表的设计虽然功能相对简单，但对状态机的理解和应用起到了很好的补充作用。通过有限状态机（FSM）控制计时功能，我掌握了如何在时序逻辑中对状态进行切换和维持。同时，七段数码管的驱动设计让我感受到，从原理到实际硬件展示是一个非常令人兴奋的过程。

从书本知识到硬件实现的过程是理论和实践不断碰撞的过程。设计中遇到的问题往往无法通过理论直接解决，需要通过仿真和调试一步步优化。例如，在解决LOAD指令的数据冒险问题时，通过反复验证前推和暂停的时序条件，我加深了对流水线控制逻辑的理解。

同时，数字跑表的实现让我对状态机的逻辑设计有了更深入的认识。起初状态切换存在卡顿问题，通过优化状态转移条件和调整时钟频率，最终实现了稳定的计时功能。这种从问题出发，寻找解决方案并验证效果的过程，让我体会到设计逻辑电路时的成就感.

在实现的过程中，我深刻认识到软硬件结合的重要性。例如，CPU的设计既依赖硬件逻辑的实现，也需要理解汇编语言和指令集的特点，才能设计出高效的流水线架构。而跑表设计则是硬件逻辑和驱动显示的结合，从时间分频到数码管显示都体现了软硬件协同的思想。

这次实践让我深刻认识到，CPU电路设计不仅是技术的体现，更是思维方式和创新能力的综合反映。从解决CPU流水线的冒险问题到设计稳定的数字跑表，我对计算机体系结构有了更全面的理解，也更加坚定了在计算机电路设计与计算机组成原理领域深入学习的信心。

# 【参考文献（资料）】

1. 汪文祥,邢金璋.CPU设计实战.机械工业出版社.2021.
2. 雷思磊.自己动手写CPU.电子工业出版社.2014.
3. 高小鹏,计算机组成与实现,高等教育出版社,2018
4. 袁静波等，《计算机组成与结构》，机械工业出版社，2011
5. 王爱英，《计算机组成与结构》，清华大学出版社，2013
6. 白中英，《计算机组成原理》，科学出版社，2008