2025~2026 学年 第一学期

《系统硬件综合设计》

设计报告

510

目录

1. 设计要求.....1
   1.1. 具体实现情况.....1
   1.2. CPU 处理指令的过程.....2
2. 设计思路.....4
   2.1. 五级流水线设计.....4
   2.2. 数据相关问题解决.....5
   2.3. HILO 寄存器实现.....8
   2.4. 流水线暂停机制的设计与实现.....8
   2.5. 乘累加、乘累减指令实现.....9
   2.6. 除法指令实现.....9
   2.7. 延迟槽以及转移指令实现.....11
   2.8. 加载存储指令的实现.....12
   2.9. Load 相关问题及解决.....14
   2.10. 协处理器访问指令实现.....14
   2.11. 异常相关指令实现.....16
3. 设计方案基本实现效果.....18
   3.1. 各阶段对应模块文件功能.....18
   3.2. 各模块连接图（总数据流图）.....27
   3.3. 各模块接口设计（接口表格）.....27
4. 性能优化与分析.....49
   4.1. 流水线效率优化：五级架构与多周期指令处理.....49
   4.2. 消除流水线冒险（Hazard Mitigation）.....49
   4.3. 控制流优化：延迟槽与分支预测.....49
   4.4. 存储与异常处理性能.....50
   4.5. 特殊寄存器与原子操作优化.....50
5. 实现指令说明及仿真测试展示.....51
   5.1. 实现指令说明.....51
   5.2. 仿真测试展示.....52
6. 总 结.....55
7. 参考文献.....56
8. 附录一（测试程序）.....57
   斐波那契数列测试仿真版.....57
   斐波那契数列测试实体硬件展示版.....57
   其余指令分类别挑选测试.....58

1. 设计要求

基于先修课程，根据系统设计思想，使用硬件描述语言设计实现一款基于MIPS32，ARM，RISC-V 或者自定义指令集的微处理器（CPU）。要求：完成单周期 CPU 设计，或多周期 CPU 设计，或多级流水线 CPU 设计（递进式、难度依次提升。所有学生必须至少完成单周期 CPU 的设计工作），可以将设计的 CPU 下载至 FPGA 开发板（ego-1）上运行。以此贯穿数字逻辑、计算机组成原理、计算机体系结构课程，实现从逻辑门至完整 CPU 处理器的设计。

1.1.具体实现情况

1.1.1.选择课题

基于精简指令集架构的多周期流水线 CPU 的设计，所设计的各类指令条数不少于 25 条，其中应当包含乘除法指令。**对于指令执行时可能产生的冒险与冲突，能够采取相应的方法合理解决；对于如何提高 CPU 性能有一定的策略并实现**。所设计的结构可以下载至 FPGA 芯片上，并在开发板上可以运行自己设计的测试程序并验证所有设计的指令。例如：斐波拉契数列的显示，游戏的运行。

1.1.2.具体实现情况

表 1-1 实现指令类型和对应条数

- **五级整数流水线：**取指、译码、执行、访存、回写。

指令类型	指令条数
逻辑与移位指令	14 条
算数指令	18 条
数据移动与 HILO 寄存器指令	6 条
转移与分支指令	14 条
访存指令	14 条
系统、特权与异常指令	17 条
总计	**83 条**

- **实现指令:**
  - **总计条数：**总计约 83 条，
  - **乘除法指令包含情况：**已经包含 MULT/MULTU（乘法）、DIV/DIVU（除法）以及 MADD/MSUB（乘累加/减）等乘除法指令。
- **对于指令执行时可能产生的冒险与冲突，能够采取相应的方法合理解决：**
  - **数据冒险：**在 id.v 和 ex.v 中实现了完整的 数据前推（Data Forwarding/Bypassing）机制，能够从 EX/MEM 和 MEM/WB 级间寄存器将结果直接转发给 ID 阶段，解决了大部分数据冒险。
  - **Load-Use 冒险：**针对 Load 指令紧跟使用其结果的指令（Load-Use 冲突），id.v 中有明确的逻辑通过 stallreq_for_reg1_loadrelate 信号触发 1 个时钟周期的暂停（Stall），解决了这一关键冲突。
  - **控制冒险：**采用了 MIPS 架构中常见的 分支延迟槽（Branch Delay Slot）机制来处理分支和跳转指令的控制冒险。
- **对于如何提高 CPU 性能有一定的策略并实现：**
  - **流水线：**采用流水线本身就是最主要的性能提升策略（提高吞吐率）。
  - **数据前推：**避免了大部分数据冒险导致的流水线暂停，是关键的性能优化实现。
  - **分支延迟槽：**减少了分支预测失败或等待分支目标计算完成的开销。
  - **乘法/除法：**乘法在 EX 阶段完成（可能通过组合逻辑或单周期实现），除法通过独立的 div.v 模块以多周期迭代方式实现，避免了阻塞整个流水线。
- **所设计的结构可以下载至 FPGA 芯片上，并在开发板上可以运行自己设计的测试程序并验证所有设计的指令。**
  - **对于设计的指令均已进行仿真测试验证**
  - **可在开发板上运行显示斐波拉契数列**

1.2.CPU 处理指令的过程

哈佛结构计算机：指令存储器与数据存储器物理上独立，拥有两套独立的总线。

可以同时取指令和取数据，并行程度更高，减少了取指和访存之间的等待。

- **取指阶段：** 从指令存储器读出指令，同时确定下一条指令地址。
- **译码阶段：** 对指令进行译码，从通用寄存器中读出要使用的寄存器的值，如果指令中含有立即数，那么还要将立即数进行符号扩展或无符号扩展。如果是转移指令，并且满足转移条件，那么给出转移目标，作为新的指令地址。
- **执行阶段：** 按照译码阶段给出的操作数、运算类型，进行运算，给出运算结果。如果是 Load/Store 指令，那么还会计算 Load/Store 的目标地址。
- **访存阶段：** 如果是 Load/Store 指令，那么在此阶段会访问数据存储器，反之，只是将执行阶段的结果向下传递到回写阶段。同时，在此阶段还要判断是否有异常需要处理，如果有，那么会清除流水线，然后转移到异常处理例程入口地址 处继续执行。
- **回写阶段：** 将运算结果保存到目标寄存器

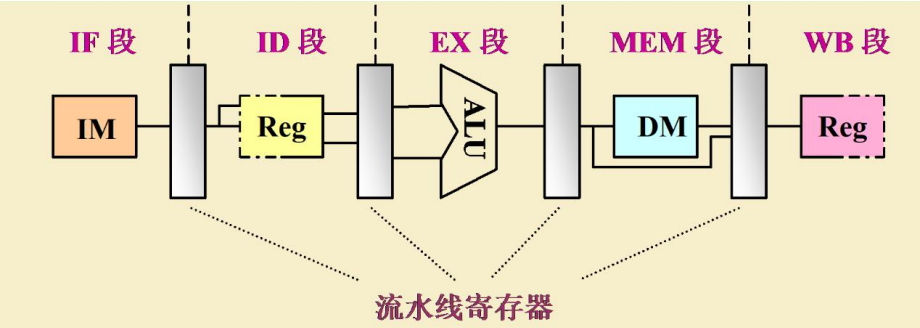

图 1.2 五段式流水线构成简图

1.2.1.指令格式

表 1-2 MIPS 三种基本指令类型

指令类型	操作	常用示例
R 型指令 (Register)	用于寄存器与寄存器之间的操作，包括算术、逻辑运算及移位操作。	add, sub, and, sll
I 型指令 (Immediate)	用于立即数操作、加载/存储（Load/Store）访存操作以及条件分支跳转。	addi, lw, sw, beq, lui
J 型指令 (Jump)	用于实现大范围的无条件跳转。	j, jal

2. 设计思路

2.1. 五级流水线设计

2.1.1. 经典三级流水线设计

2.1.1.1. 理想设计情况

指令的处理从直观上分析至少可以拆分为三步：从存储器取出指令、解释指令、按照解释的结果执行，简单地说就是：取指、译码、执行。如果我们只有一个硬件处理单元，这个单元既要取指，又要译码，还要执行，假设上述三种操作都可以在时间 T 完成，那么一条指令的处理时间为 3T，n 条指令的处理时间就为 3nT，但是如果我们设计有三个硬件单元，分别做这三项工作的一项，那么就可以在执行的同时对下一条指令译码，在对下一条指令译码的同时还可以再取一条指令，这就是经典的三级流水线

图 2-1 经典三级流水线示意图

2.1.1.2. 实际存在缺陷

当某阶段执行时间过长，则后续阶段即会出现停滞，让整个流程处理速度均减慢

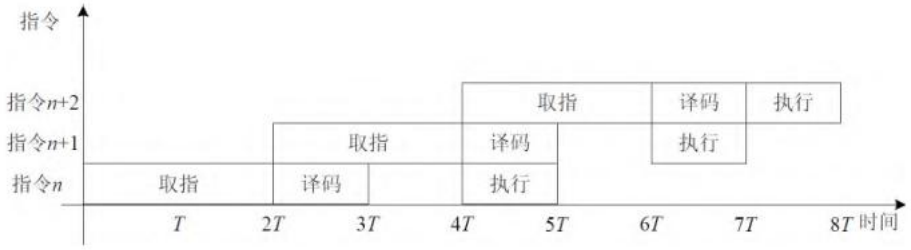

图 2-2 取指时间长于其他阶段时间示意图

2.1.2.五级流水线设计

某一阶段时间过长的情况较多，在设计流水线级数时，本设计尽可能解决指令为加载/存储指令（Load/Store）时，由于涉及访问存储器，执行阶段所需的时间就可能大于 T，导致流水线停滞的情况，故选择设计五级流水线：取指、译码、执行、访存、回写。

其中访存阶段（Memory Access）的作用是从存储器装载数据到寄存器或者将寄存器数据保存到存储器，当然，如果不是 Load/Store 指令则不需要这一步，此时在访存阶段就只是将执行阶段的运算结果送到下一级回写阶段。回写阶段（Write Back）的作用是将数据写入目的寄存器。

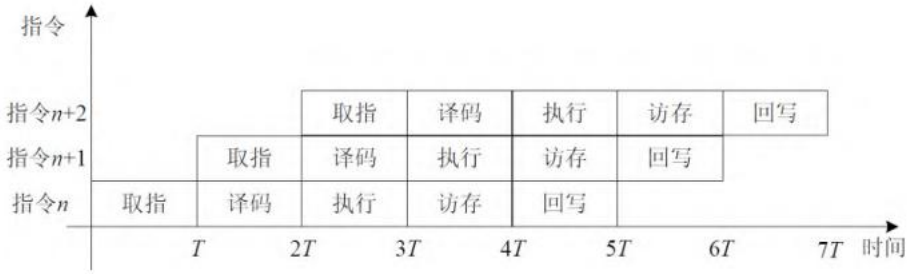

图 2-3 五级流水线示意图

2.2.数据相关问题解决

2.2.1.基本概念

数据相关：指的是在流水线中执行的几条指令中，一条指令依赖于前面指令的执行结果。流水线数据相关又分为三种情况：RAW、WAR、WAW。

RAW，即 Read After Write，假设指令 j 是在指令 i 后面执行的指令，RAW 表示指令 i 将数据写入寄存器后，指令 j 才能从这个寄存器读取数据。如果指令 j 在指令 i 写入寄存器前尝试读出该寄存器的内容，将得到不正确的数据。

WAR，即 Write After Read，假设指令 j 是在指令 i 后面执行的指令，WAR 表示指令 i 读出数据后，指令 j 才能写这个寄存器。如果指令 j 在指令 i 读出数据前就写该寄存器，将使得指令 i 读出的数据不正确。

WAW，即 Write After Write，假设指令 j 是在指令 i 后面执行的指令，WAW 表示指令 i 将数据写入寄存器后，指令 j 才能将数据写入这个寄存器。如果指令 j 在指令 i 之前写该寄存器，将使得该寄存器的值不是最新值。

2.2.2.设计分析

在本设计流水线中，仅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。所以本设计只存在 RAW 相关的情况。

- **相邻指令间存在数据相关：**第 1 条指令将写寄存器$1，随后的第 2 条指令需要读出$1 的数据，但是第 1 条指令在回写阶段才会将其运算结果写入$1，而第 2 条指令在译码阶段就需要读取$1 的值，此时第 1 条 ori 指令还处于执行阶段，所以得到的必然不是第 1 条 ori 指令计算得出的结果，按这个值运算，必然会出错。
- **相隔 1 条指令的指令间存在数据相关：**第 1 条指令将写寄存器$1，第 3 条指令在译码阶段需要读取寄存器$1，此时第 1 条 ori 指令还处于访存阶段，所以得到的必然也不是正确的值。
- **相隔 2 条指令的指令间存在数据相关：**第 1 条指令将写寄存器$1，第 4 条指令在译码阶段需要读取寄存器$1，此时第 1 条指令处于回写阶段，在回写阶段最后的时钟上升沿才会将运算结果写入$1，所以第 4 条指令得到的不是正确的寄存器$1 的值。

2.2.3.解决方案

使用数据前推的方法来解决流水线数据相关问题。

- 将处于流水线执行阶段的指令的运算结果，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、要写入目的寄存器的数据 wdata_o 等信息送到译码阶段
- 将处于流水线访存阶段的指令的运算结果，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、要写入目的寄存器的数据 wdata_o 等信息送到译码阶段。

2.2.4.实现

在译码阶段 ID 模块，添加两处接口，接收执行阶段和访存阶段的传入结果

表 2-1 ID 模块数据相关实现处理

接口名	输入/输出	作用
ex_wreg_i	输入	处于执行阶段的指令是否要写目的寄存器
ex_wd_i	输入	处于执行阶段的指令要写的目的寄存器地址
ex_wdata_i	输入	处于执行阶段的指令要写入目的寄存器的数据
mem_wreg_i	输入	处于访存阶段的指令是否要写目的寄存器
mem_wd_i	输入	处于访存阶段的指令要写的目的寄存器地址

2.3.HILO 寄存器实现

2.3.1.实现用处

专门用于处理乘法与除法结果的特殊寄存器，在 MIPS 架构中，两个 32 位数相乘会得到一个 64 位的结果。为了存放这个双倍宽度的结果，MIPS 定义了两个独立的寄存器：HI 存放高 32 位，LO 存放低 32 位，它们与 32 个通用寄存器（Regfile）是物理隔离的，主要用于以下指令：

- 乘法 (MULT/MULTU)：将 64 位结果存入 HI 和 LO。
- 除法 (DIV/DIVU)：将余数存入 HI，商存入 LO。
- 数据移动 (MFHI/MFLO/MTHI/MTLO)：在通用寄存器与 HI/LO 之间交换数据。

2.3.2.新数据相关出现及处理

指令 1、2、3 均要修改 HI 寄存器，当指令 4 处于执行阶段时，指令 3 处于访存阶段，指令 2 处于回写阶段，而此时 HI 寄存器的值是指令 1 刚刚写入值，HILO 模块正是将该值传到执行阶段，如果采用这个值，那么就会出错，偏离程序设想，正确的值应该是当前处于访存阶段的指令 3 要写的数据。

为此，照例使用数据前推的方式将访存阶段、回写阶段的信息反馈到执行阶段，输入到执行阶段的选择模块。

2.4.流水线暂停机制的设计与实现

2.4.1.实现用处

因为设计乘累加、乘累减、除法指令在流水线执行阶段占用多个时钟周期，因此需要暂停流水线，以等待这些多周期指令执行完毕

2.4.2.实现思路

假如位于流水线第 n 阶段的指令需要多个时钟周期，进而请求流水线暂停，那么需保持取指令地址 PC 的值不变，同时保持流水线第 n 阶段、第 n 阶段之前的各个阶段的寄存器不变，而第 n 阶段后面的指令继续运行。比如：流水线执行阶段的指令请求流水线暂停，那么保持 PC 不变，同时保持取指、译码、执行阶段的寄存器不变，但是可以允许访存、回写阶段的指令继续运行

为此，设计添加 CTRL 模块，其作用是接收各阶段传递过来的流水线暂停请求信号，从而控制流水线各阶段的运行。

CTRL 模块的输入来自 ID、EX 模块的请求暂停信号 stallreq。CTRL 模块对暂停请求信号进行判断，然后输出流水线暂停信号 stall。stall 输出到 PC、IF/ID、ID/EX、EX/MEM、 MEM/WB 等模块，从而控制 PC 的值，以及流水线各个阶段的寄存器。

2.5.乘累加、乘累减指令实现

2.5.1.实现思路

在流水线执行阶段采用两个时钟周期完成运算，第一个时钟周期进行乘法运算，第二个时钟周期将乘法结果与 HI、LO 寄存器进行加/减法。

为了实现乘累加、乘累减指令的实现思路，必须要保存两个信息：

- 当前是第几个时钟周期;
- 乘法结果;通过在 EX/MEM 模块中添加两个寄存器 cnt、hilo，分别保存上述信息

2.6.除法指令实现

2.6.1.实现方法（试商法）

设被除数是 m，除数是 n，商保存在 s 中，被除数的位数是 k，其计算步骤如下（为了便于说明，在此处将所有数据的最低位称为第 1 位，而不称为第 0

位）。

1．取出被除数的最高位 m[k]，使用被除数的最高位减去除数 n，如果结果大于等于 0，则商的 s[k]为 1，反之为 0。

2．如果上一步得出的结果是 0，表示当前的被减数小于除数，则取出被除数剩下的值的最高位 m[k-1]，与当前被减数组合作为下一轮的被减数；如果上一步得出的结果是 1，表示当前的被减数大于除数，则利用上一步中减法的结果与被除数剩下的值的最高位 m[k-1]组合作为下一轮的被减数。然后，设置 k 等于 k-1。

3．新的被减数减去除数，如果结果大于等于 0，则商的 s[k]为 1，否则 s[k]为 0，后面的步骤重复 2-3，直到 k 等于 1

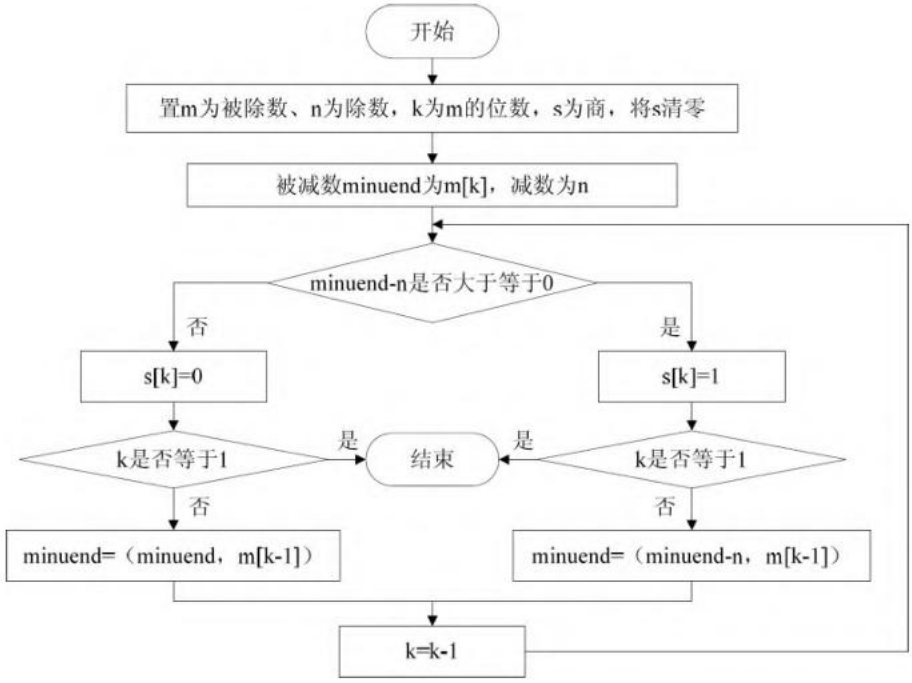

图 2-4 试商法运算过程

2.6.2. 具体实现

新建一个模块 DIV，在其中实现采用试商法的 32 位除法运算。当流水线执行阶段的 EX 模块发现当前指令是除法指令时，首先暂停流水线，然后将被除数、除数等信息送到 DIV 模块，开始除法运算。DIV 模块在除法运算结束后，通知 EX 模块，并将除法结果送到 EX 模块，后者依据除法结果设置 HI、LO 寄存器的写信息，同时取消暂停流水线。对于 32 位的除法，至少需要 32 个时钟周期才能得到

除法结果。

2.7.延迟槽以及转移指令实现

2.7.1.控制相关问题

控制相关是指流水线中的转移指令或者其他需要改写 PC 的指令造成的相关。这些指令改写了 PC 的值，所以导致后面已经进入流水线的几条指令无效，比如：如果转移指令在流水线的执行阶段进行转移条件判断，在发生转移时，会导致当前处于取指、译码阶段的指令无效，需要重新取指。也就是说，在流水线执行阶段进行转移判断，并且转移发生，那么会有 2 条无效指令，导致浪费了两个时钟周期。

为了减少损失，规定转移指令后面的指令位置为“延迟槽”，延迟槽中的指令被称为“延迟指令”（也可称之为“延迟槽指令”）。延迟指令总是被执行，与转移发生与否没有关系。即使引入延迟槽，在转移发生时仍然会导致已经进入取指阶段的指令无效，也就是说，仍浪费一个时钟周期，要解决这个问题，本设计选择在译码阶段进行转移判断，这样就可以避免浪费时钟周期。

2.7.2.具体实现

- 如果处于译码阶段的指令是转移指令，并且满足转移条件，那么 ID 模块设置转移发生标志 branch_flag_o 为 Branch，同时通过 branch_target_address_o 接口给出转移目的地址，送到 PC 模块，后者据此修改取指地址
- 如果处于译码阶段的指令是转移指令，并且满足转移条件，那么 ID 模块还会设置 next_inst_in_delayslot_o 为 InDelaySlot，表示下一条指令是延迟槽指令，其中 InDelaySlot 是一个宏定义。

  next_inst_in_delayslot_o 信号会送入 ID/EX 模块，并在下一个时钟周期通过 ID/EX 模块的 is_in_delayslot_o 接口送回到 ID 模块，ID 模块可以据此判断当前处于译码阶段的指令是否是延迟槽指令。

- 如果转移指令需要保存返回地址，那么 ID 模块还要计算返回 地址，并通过 link_addr_o 接口输出，该值最终会传递到 EX 模块，作为 要写入目的寄存器的值。

2.8.加载存储指令的实现

2.8.1.Data_ram 设计

独立的 data_ram 模块实现了数据存储与指令存储的解耦。data_ram 负责程序运行过程中的变量存取与堆栈维护，支持大端模式下的字、半字、字节访问，配合流水线 MEM 阶段完成高效的访存操作

2.8.2.设计概念

2.8.2.1. 加载指令

- 执行阶段 (EX)：ALU 计算出内存地址。例如 Addr = Reg[rs] + offset。
- 访存阶段 (MEM)：
  - mem 模块向 data_ram 发送地址和读使能信号。
  - data_ram 返回 32 位数据。
  - 字节对齐处理：如果是 LB（加载字节），mem 模块会根据地址的低两位，从 32 位字中截取出特定的 8 位，并进行符号扩展。
- 回写阶段 (WB)：将最终处理好的数据写回目标寄存器（rt）。

2.8.2.2. 存储指令

- 执行阶段 (EX)：
  - 计算内存地址。
  - 准备待写入的数据（从寄存器 rt 中读出的值）。
- 访存阶段 (MEM)：

- mem 模块向 data_ram 发送地址、数据和写使能信号。
- 字节掩码 (Select Signals)：如果是 SB（存储字节），mem 模块会生成一个特殊的“选通信号”（如 4'b0001），告诉存储器只更新该字中的某一个字节。

- 回写阶段 (WB)：Store 指令不需要写回通用寄存器，因此该阶段对 Store 指令无效（不做任何操作）。

2.8.2.3. 链接加载指令 ll、条件存储指令 sc

在多线程系统中，需要 RMW（Read-Modify-Write）操作序列保证对某个资源的独占性，RMW 操作序列的含义是，读取内存某个地址的数据，读取的数据经过修改，然后再保存回内存原地址，这个过程不能有任何打扰，因此需要建立一个临界区域（Critical Region），临界区域中完成的操作通常称为原子操作，原子操作不被打扰。操作系统建立临界区域的方式通常是信号量机制

```
wait（semaphore）；

原子操作；

signal（semaphore）；
```

图 2-5 信号量机制示意图

semaphore 是一个信号量，为 1 表示信号量使用中，为 0 表示信号量空闲。进行原子操作前，使用 wait 函数查询 semaphore 的值，如果为 1，则等待，否则，将其置为 1，开始执行原子操作。操作结束后，signal 函数将 semaphore 置为 0，这样其他线程就可以执行原子操作了。

ll 指令同一般的加载指令一样，从内存中加载一个字，但是，有一点不同，ll 指令还会将处理器内部的一个链接状态位 LLbit 置为 1，表明发生了一个链接加载操作。

ll 指令执行完毕后，会进行一定的操作（如：修改加载得到的数据），然后执行 sc 指令，这可以认为是一个 RMW 序列。有如下情况干扰这个 RMW 序列，受到干扰后，处理器会设置链接状态位 LLbit 为 0。

- 在 ll、sc 指令之间产生异常，从而进入异常处理例程，或者发生线程切

换，导致 RMW 序列受到干扰。

执行 sc 指令时，会对从 ll 指令开始的 RMW 序列进行检查，判断是否 受到干扰，实际就是判断 LLbit 是否为 1，如果没有受到任何干扰，LLbit 保持为 1，那么操作是原子的，sc 指令会对 ll 指令加载数据的地址进行写回操作，并设置一个通用寄存器的值为 1，表示成功，反之不进行写回操作，并设置一个通用寄存器的值为 0，表示失败。

2.9.Load 相关问题及解决

2.9.1.Load 相关问题

假设有如下指令序列：

```
LW  $t1, 0($t2)  # 指令 A：从内存读数据到 $t1
ADD $t3, $t1, $t4 # 指令 B：立即使用 $t1 进行加法
```

图 2-6 Load 相关指令举例

当指令 B 到达 EX 阶段准备计算时，指令 A 刚刚进入 MEM 阶段。此时 data_ram 还没返回数据给$t1。

即使通过数据前推的方法，将访存阶段加载得到的数据前推，也解决不了问题，因为数据加载时，指令 B 已经处于执行阶段了，已经进行了比较判断，这种情况称为 load 相关。

2.9.2.解决方法

在译码阶段检查当前指令与上一条指令是否存在 load 相关，如果存在 load 相关，那么就让流水线的译码、取指阶段暂停，而执行、访存、回写阶段继续，相当于插入一个空指令，这样处于执行阶段的加载指令会继续运行，不受影响，当其运行到访存阶段时，将加载得到的数据前推到译码阶段，然后，流水线可以继续运行。

2.10. 协处理器访问指令实现

2.10.1. 协处理器

协处理器一词通常用来表示处理器的一个可选部件，负责处理指令集的某个扩展，具有与处理器核独立的寄存器。

MIPS32 架构提供了 最多 4 个协处理器，分别是 CP0～CP3

协处理器	作　用
CP0	系统控制
CP1	FPU
CP2	特定实现
CP3	FPU

图 2-7 MIPS 架构定义的协处理器及其作用

本设计中主要实现 CP0 的工作：

- 配置 CPU 工作状态：符合 MIPS32 架构的硬件通常是很灵活的，可以通过读/写一个或一些内部寄存器来改变一些很根本的 CPU 特性（如：将字节次序从 MSB 变为 LSB，或者从 LSB 变为 MSB）。
- 高速缓存控制：符合 MIPS32 架构的 CPU 一般会集成缓存控制器，用来控制、读、写缓存。
- 异常控制：异常发生时的检测和处理都由 CP0 中的一些控制寄存器来定义和控制。
- 存储管理单元控制：对系统的存储区域进行合理的控制、管理和分配，主要是对 MMU、TLB 的一些配置、管理、访问。
- 其他：当要把额外的功能集成在 CPU 中，但又不方便当作外设访问时，常常在 CP0 中增加一些模块以实现这些功能。例如：时钟、时间计数器、奇偶校验错误检测等。

2.11. 异常相关指令实现

2.11.1.异常相关的概念

在 MIPS32 架构中，有一些事件要打断程序的正常执行流程，这些事件有中断（Interrupt）、陷阱（Trap）、系统调用（System Call）以及其他任何可以打断程序正常执行流程的情况，统称为异常。

本设计主要实现以下 6 种异常情况处理

- 硬件复位；
- 中断（包含软中断、硬中断）；
- syscall 系统调用；
- 无效指令；
- 溢出；
- 自陷指令引发的异常。

2.11.1.1. 精确异常

当一个异常发生后，系统的顺序执行会被中断，此时有若干条指 令处于流水线上的不同阶段，处理器会转移到异常处理例程，异常处 理结束后返回原程序继续执行，因为不希望异常处理例程破坏原程序 的正常执行，所以对于异常发生时，流水线上没有执行完的指令，必 须记住它处于流水线的哪一个阶段，以便异常处理结束后能恢复执 行，这便是精确异常。

2.11.1.2. 按指令执行的顺序处理异常

先发生的异常并不立即处理，异常事件只是被标记，并继续运行流水线。在大多数处理器中，会设计一个特殊的流水线阶段，专门用于处理异常。如果某一条指令的异常事件到达了流水线的这个阶段，那么会进行异常处理，并且当前处于流水线其余阶段的指令的异常事件都会被忽略。

2.11.2.异常处理过程

（1）检测 CP0 中 Status 寄存器的 EXL 字段，分两种情况。

- 如果 EXL 为 1，表示当前已经处于异常处理过程中了，此时，如果当前发生的异常类型是中断，那么不处理，忽略该异常，因为在异常处理过程中会禁止中断。如果当前发生的异常类型不是中断，那么将异常原因保存到 CP0 中 Cause 寄存器的 ExcCode 字段，转到步骤（4）。
- 如果 EXL 为 0，那么将异常原因保存到 CP0 中 Cause 寄存器的 ExcCode 字段，进入步骤（2）。

（2）检查发生异常的指令是否在延迟槽中，如果在延迟槽中，那么设置 EPC 寄存器的值为该指令的地址减 4，同时设置 Cause 寄存器的 BD 字段为 1，反之，设置 EPC 寄存器的值就为该指令的地址，同时设置 Cause 寄存器的 BD 字段为 0。

（3）设置 Status 寄存器的 EXL 字段为 1，表示进入异常处理过程，禁止中断。

（4）处理器转移到事先定义好的一个地址，在那个地址中往往有异常处理例程，在其中进行异常处理，这个地址称为异常处理例程入口地址。

2.11.3.异常处理的实现

在流水线的各个阶段收集异常信息，并传递到流水线访存阶段，在访存阶段统一处理异常信息。流水线各个阶段需要收集的异常信息如下：

- 在流水线译码阶段判断是否是系统调用异常、是否是返回指令、无效指令。
- 在流水线执行阶段判断是否有自陷异常、溢出异常。
- 在流水线访存阶段检查是否有中断发生。

在流水线访存阶段，处理器将结合协处理器 CP0 中相关寄存器的值，判断异常是否需要处理，如果需要处理，那么转移到该异常对应的处理例程入口地址，清除流水线上除回写阶段外的全部信息，同时，修改协处理器 CP0 中相关寄存器的值。

3. 设计方案基本实现效果

3.1. 各阶段对应模块文件功能

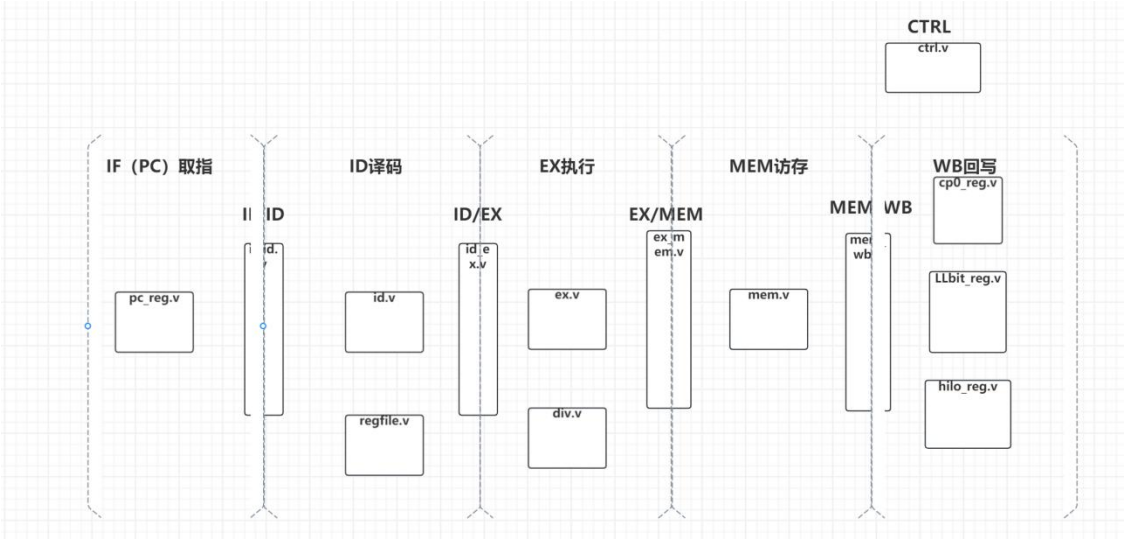

图 3-1 各阶段对应模块/文件

3.1.1. PC_reg.v 文件功能

3.1.1.1. 核心任务

它的本质是一个带控制逻辑的 32 位寄存器。它的基本逻辑是：

- 默认： 顺序执行（PC = PC + 4）。
- 跳转： 遇到分支指令（PC = BranchTarget）。
- 强制重置： 遇到异常或异常处理（PC = NewPC）。

3.1.1.2. 控制逻辑优先级

- 最高优先级：ce == ChipDisable

  如果芯片没启动，PC 强制为 0。

- 第二优先级：flush == 1'b1

  通常用于异常（Exception）或中断（Interrupt）。当流水线发生严重错误或需要强制跳转到内核入口时，直接覆盖所有逻辑，跳转到 new_pc。

- 第三优先级：stall[0] == NoStop

这是暂停机制。如果取指阶段被暂停（例如内存还没准备好数据），PC 就会“冻结”，保持当前值不变。

- 最低优先级：branch_flag_i

在不暂停、不冲刷的情况下，判断是执行跳转（branch_target_address_i）还是老老实实地加 4。

3.1.2.If_id.v 文件功能

3.1.2.1. 核心任务

在没有干扰的正常情况下，它的工作非常简单：

- 输入： if_pc（当前地址）, if_inst（刚读出来的指令）。
- 动作： 每个时钟上升沿，把输入锁存到寄存器里。
- 输出： id_pc, id_inst。

3.1.2.2. 流水线冲突处理

- 冲刷机制 (Flush)

触发场景： 预测跳转失败、发生系统调用（Syscall）、外部中断。

逻辑： 只要 flush 为 1，无论输入是什么，输出全部变 0（即 NOP 空指令）。

目的： 把已经取出来但其实不该执行的指令扔掉。

- 暂停与气泡 (Stall & Bubble)

逻辑： 如果 ID 阶段要停下来（比如等前面的数据写回），但后面的 EX 阶段还在跑。

结果： 它向后面发送一个全 0 的指令。

- 全线冻结

逻辑： 如果 stall[1] 和 stall[2] 都是 Stop。

结果： 寄存器保持原值。

3.1.3.Id.v 文件

3.1.3.1. 核心任务

它接收 if_id 传来的 32 位机器码，根据操作码（op, op2, op3 等）确定这是一条什么指令（是加法、逻辑或、还是跳转）。

输出控制信号：

aluop_o：告诉执行阶段（EX）具体做什么运算（如 EXE_OR_OP）。

alusel_o：告诉执行阶段结果属于哪一类（逻辑、算术、移位等）。

wreg_o：这指令是否需要写回寄存器。

立即数处理： 将指令中的 16 位立即数进行符号扩展或零扩展（imm），变成 32 位供后续使用。

3.1.3.2. 特殊处理逻辑

- 处理数据相关：数据前推（Forwarding）

检查 ex_wreg_i 和 ex_wd_i。如果 EX 阶段要写的寄存器正是我们要读的，直接把 ex_wdata_i 给 reg1_o。

同样的逻辑检查 MEM 阶段。

- 冲突检测与流水线暂停（Stall）

内部信号 pre_inst_is_load 检测前一指令。

如果满足 pre_inst_is_load && ex_wd_i == reg1_addr_o，触发 stallreq。

结果： 这个信号会发给控制单元（CTRL），让取指和译码阶段原地踏步。

- 跳转逻辑与延迟槽（Branch & Delay Slot）

分支计算： 例如 BEQ 指令，它在这里比较 reg1_o 和 reg2_o 是否相等。如果相等，立即设置 branch_flag_o 并计算 branch_target_address_o

延迟槽（Delay Slot）： 跳转指令后的那条指令（延迟槽指令）总是会被执行。模块通过 is_in_delayslot_o 信号管理这一逻辑。

- 异常处理初探

instvalid：如果解析发现是不认识的指令，标记为无效指令异常。

excepttype_is_syscall：如果是 syscall 指令，标记为系统调用异常。

3.1.4.Id_ex.v 文件功能

将译码阶段（ID）解析出来的各种指令信息（计算方法、操作数、地址等），在时钟上升沿“锁存”并传递给执行阶段（EX）。

基本功能和 If_id.v 相似

3.1.5.Ex.v 文件功能

3.1.5.1. 核心任务

该模块根据译码阶段传来的 aluop_i（运算子类型）和 alusel_i（运算大类），对操作数 reg1_i 和 reg2_i 进行处理：

逻辑运算：支持 AND、OR、NOR、XOR。

移位运算：支持逻辑左移 (SLL)、逻辑右移 (SRL) 和算术右移 (SRA)。

算术运算：

加减法（ADD, SUB, ADDU, SUBU）。

比较运算（SLT, SLTU）。

计数指令：实现了 CLZ（从高位起第一个 0 前导个数）和 CLO（第一个 1 前导个数）。

乘法与除法：

乘法：支持有符号/无符号乘法。

除法：通过接口与外部除法器模块通信。由于除法耗时较长，模块会通过 stallreq_for_div 向流水线控制器发出暂停请求。

3.1.5.2. 特殊指令处理

- **特殊寄存器处理 (HI/LO 寄存器)**

乘除法的结果存储在专门的 HI 和 LO 寄存器中。

数据前推 (Data Forwarding)：为了解决数据相关问题，模块会检查访存阶段（MEM）和回写阶段（WB）是否正在改写 HI/LO。如果是，则直接从后面的流水级取值，而不是读寄存器组的旧值。

乘累加 (MADD/MSUB)：支持 MADD（乘加）和 MSUB（乘减）。这类指令通常需要两个时钟周期完成，模块内部使用了状态计数器 cnt_i 来控制流水线的暂停和计算。

- **CP0 (协处理器 0) 接口**

CP0 主要负责异常处理和状态控制。

读写支持：支持 MFC0（读取 CP0 寄存器到通用寄存器）和 MTC0（将通用寄存器的值写入 CP0）。

前推逻辑：同样针对 CP0 寄存器实现了数据前推，防止因流水线延迟导致的读错数据。

- **异常检测逻辑**

溢出检测 (Overflow)：在执行 ADD、ADDI、SUB 等有符号运算时，如果结果溢出，会置位 ovassert，并屏蔽寄存器的写使能（wreg_o），防止错误结果写入。

自陷指令 (Trap)：实现了 TEQ、TNE、TGE 等自陷指令。如果满足比较条件，会置位 trapassert。

异常打包：最后将译码阶段传来的异常信息与本级产生的溢出、自陷异常合并，打包成 excepttype_o 送往下一级。

- **加载/存储 (Load/Store) 预处理**

地址计算：通过 reg1_i（基址）加立即数（偏移量）计算出物理地址 mem_addr_o。

透传：将 aluop 和存储数据 reg2_o 传递给访存阶段。

- **流水线控制 (Stall Request)**

当执行 DIV 指令且除法器未就绪时，stallreq 拉高。

当执行 MADD/MSUB 的第一个周期时，stallreq 拉高。

3.1.6. Ex_mem 文件功能

在每个时钟上升沿，将 EX 阶段产生的运算结果“捕捉”下来，并在下一个时钟周期提供给 MEM 阶段。

基本功能和 If_id.v 相似

3.1.7. Mem.v 文件功能

3.1.7.1. 核心任务实现

- **内存读写控制 (Load / Store)**

根据 aluop_i 判定具体的指令类型，并根据计算出的地址 mem_addr_i 生成对内存的操作信号。

加载指令 (Load)：支持字节 (LB/LBU)、半字 (LH/LHU) 和字 (LW)。模块通过 mem_sel_o（字节选择信号）告诉内存要读哪几位，并将读到的数据进行符号扩展或零扩展后放入 wdata_o。

存储指令 (Store)：支持 SB、SH、SW。它会根据地址的低两位自动调整数据在 32 位总线上的位置。

非对齐加载/存储：实现了 LWL/LWR 和 SWL/SWR，用于处理 MIPS 中特有的非对齐内存访问。

原子操作：实现了 LL (Load Linked) 和 SC (Store Conditional)，这是实现多处理器同步锁的关键指令。模块会维护一个 LLbit 来判定存储是否成功。

- **异常判定与处理 (Exception Handling)**

异常优先级排查：模块会检查 excepttype_i（译码和执行阶段传来的异常）以及当前是否存在外部中断 (Interrupt)。

中断判定：结合 CP0 的 Status 寄存器（允许中断位）和 Cause 寄存器（挂起的中断位）进行逻辑判断。

屏蔽写操作：如果判定发生异常（excepttype_o 非 0），模块会强制将 mem_we_o 置为无效，防止因异常指令错误地修改内存数据。

- **系统寄存器 (CP0) 维护**

数据前推 (Data Forwarding)：它会检查回写（WB）阶段是否正在写 CP0 寄存器（如 Status、EPC、Cause）。如果是，则直接使用 WB 阶段的新值，解决数据相关性问题。

信息打包：将异常类型、指令是否在延迟槽中、以及当前指令地址打包，交给异常处理模块（通常是 ctrl 模块）进行跳转决策

3.1.8. Mem_wb.v 文件功能

在时钟上升沿将访存阶段处理完的数据锁存，并在下一个时钟周期传递给回写阶段，最终由回写阶段将数据写入通用寄存器堆、HILO 寄存器或 CP0 协处理器寄存器。

基本功能和 If_id.v 相似

3.1.9. Hilo_reg.v 文件功能

一个 64 位的专用同步寄存器：

HI 寄存器：通常存放乘法结果的高 32 位，或者除法运算的余数（Remainder）。

LO 寄存器：通常存放乘法结果的低 32 位，或者除法运算的商（Quotient）。

3.1.10. Ctrl.v 文件功能

整个 CPU 的调度中心。在流水线 CPU 中，各个阶段（取指、译码、执行、访存、回写）是并行工作的，但当出现资源冲突、耗时运算或异常情况时，就需要这个模块来协调。

- **流水线暂停机制**

该模块根据 stall 信号的 6 位掩码来控制不同的阶段（通常对应：PC, IF, ID, EX, MEM, WB）：

来自译码阶段的暂停 (stallreq_from_id)：

通常是因为 Load 相关（Load-Use Hazard）。

stall <= 6'b000111：暂停 PC、IF、ID 阶段，EX 及其之后的阶段继续流动。

来自执行阶段的暂停 (stallreq_from_ex)：

主要用于除法（DIV）或多周期的乘累加（MADD）。

stall <= 6'b001111：暂停 PC、IF、ID、EX 阶段，只有 MEM 和 WB 继续流动。

- **流水线刷新与异常处理**

刷新信号 (flush)：一旦 excepttype_i 不为 0，flush 立即置 1。这会通知所有流水线寄存器（如之前看的 ex_mem）清空当前内容。

异常跳转地址 (new_pc)：模块根据异常类型决定 CPU 下一步去哪里取指：

中断 (Interrupt)：跳转到 0x20。

Syscall / Invalid Inst / Trap / Overflow：统一跳转到异常处理入口 0x40。

ERET (从异常返回)：将 PC 恢复为 CP0 寄存器中保存的地址 cp0_epc_i。

3.1.11.Div.v 文件功能

3.1.11.1. 试商法实现

基本逻辑：每一轮将被除数左移一位，减去除数。如果减法结果为正，说明“够减”，该位商 1；如果为负，说明“不够减”，该位商 0 并保持/恢复原值。

迭代次数：通过 cnt 计数器进行 32 次迭代，覆盖 32 位的被除数

3.1.12. LLbit_reg.v 文件功能

用于实现 MIPS 架构中的原子操作

LL 指令：从内存读取数据，并将 LLbit 置为 1，表示“我已经锁定这个地址了”。

SC 指令：在写入内存前检查 LLbit。如果 LLbit 仍为 1，说明期间没有发生异常或切换，写入成功并返回 1；如果 LLbit 变为了 0，说明这次操作是不安全的，

写入失败并返回 0。

3.1.13. Cp0_reg.v 文件功能

处理中断、异常、操作系统特权指令或定时任务。

- 自动更新逻辑

模块中不仅有被动的读写，还有主动的硬件行为：

时钟递增：count_o <= count_o + 1; 确保了硬件计时功能。

中断判定：一旦 count_o == compare_o，立即拉高 timer_int_o，这是操作系统实现分时调度的基础。

外部中断输入：cause_o[15:10] <= int_i; 实时捕捉 6 路外部硬件中断信号。

- 异常现场保存 (Exception Entry)

当 excepttype_i 有效时（非 0），模块会自动进入“保护模式”：

保存返回地址：如果指令在延迟槽中（is_in_delayslot_i），EPC 必须保存为跳转指令的地址（current_inst_addr_i - 4），并将 Cause 寄存器的第 31 位（BD 位）置 1。

锁定状态：将 Status[1] (EXL 位) 置 1。这表示正在处理异常，此时硬件会屏蔽其他中断，防止嵌套。

记录原因：将异常代码写入 Cause[6:2]。例如 0x08 代表系统调用 (Syscall)。

- 异常返回 (Exception Return - ERET)

当执行 eret 指令时，excepttype_i 为 0x0000000e：

它会将 Status[1] 清零，告诉硬件异常处理已结束，可以重新响应中断。

3.1.14. Regfile.v 文件功能

寄存器堆 (Register File) 实现读写端口。

3.2. 各模块连接图（总数据流图）

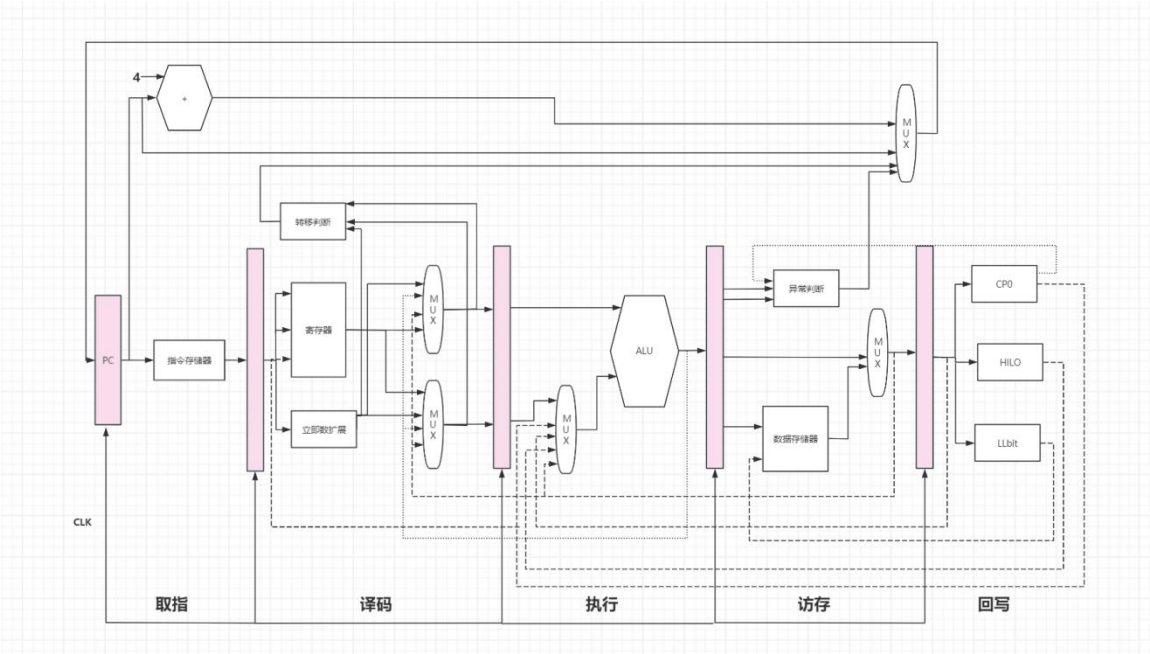

图 3-2 各模块链接图（数据流图）

3.3. 各模块接口设计（接口表格）

3.3.1. pc 模块

表 3-1 pc 模块接口表格

信号名称	位宽	I/O 类型	来源/去向	功能描述
clk	1	Input	全局时钟	系统的基准时钟信号，上升沿触发 PC 更新。
rst	1	Input	复位模块	全局复位信号（高有效）。置位时 PC 复位至初始地址。

信号名称	位宽	I/O 类型	来源/去向	功能描述
stall	6	Input	CTRL (控制模块)	流水线暂停控制。若 stall[0] 有效，则 PC 保持不变，取指级挂起。
flush	1	Input	CTRL (控制模块)	异常/中断冲刷信号。有效时丢弃当前取指路径，跳转至异常入口。
new_pc	32	Input	CTRL (控制模块)	异常处理程序的入口地址。当 flush 有效时强制赋给 PC。
branch_flag_i	1	Input	ID (译码阶段)	分支/跳转使能信号。为 1 时表示发生了指令跳转。
branch_target_address_i	32	Input	ID (译码阶段)	由 ID 级提前计算出的分支/跳转目标物理地址。
pc	32	Output	Inst_ROM / IF_ID	下一条指令的地址。输出到指令存储器取指，并传给 ID 级计算。

信号名称	位宽	I/O 类型	来源/去向	功能描述
ce	1	Output	Inst_ROM	指令存储器使能信号。指示当前 PC 地址是否有效。

3.3.2.If_id 模块

表 3-2 if_id 模块接口表格

信号名称	位宽	I/O 类型	来源/去向	功能描述
clk	1	Input	全局时钟	触发段寄存器更新的时钟信号。
rst	1	Input	复位模块	全局复位信号。有效时，清除当前段寄存器中的指令（置 0）。
stall	6	Input	CTRL (控制模块)	流水线暂停信号。stall[1] 有效时，保持 id_pc/inst 不变。
flush	1	Input	CTRL (控制模块)	流水线冲刷信号。有效时，将当前指令清空（通常变为 NOP 指令）。
if_pc	32	Input	pc_reg	来自 IF 阶段的指令地址。
if_inst	32	Input	Inst_ROM	从指令存储器中刚取出的原始指令。
id_pc	32	Output	ID 阶段 (id.v)	输出到译码阶段的指令地址。
id_inst	32	Output	ID 阶段 (id.v)	输出到译码阶段的指令内容。

3.3.3.id 模块

表 3-3 id 模块输入接口表格

信号名称	位宽	I/O 类型	来源	功能描述
rst	1	Input	全局复位	复位信号，清空所有译码输出和标志位。
pc_i	32	Input	IF/ID	当前待译码指令的地址。
inst_i	32	Input	IF/ID	当前待译码的 32 位原始机器指令。
ex_aluop_i	8	Input	EX 阶段	处于执行阶段指令的运算类型，用于 Load-Use 冲突检测。
ex_wreg_i	1	Input	EX 阶段	执行阶段是否要写回寄存器。用于 EX 级数据前推。
ex_wdata_i	32	Input	EX 阶段	执行阶段指令算出的结果。用于 EX 级数据前推。
ex_wd_i	5	Input	EX 阶段	执行阶段指令要写入的目标寄存器地址。
mem_wreg_i	1	Input	MEM 阶段	访存阶段是否要写回寄存器。用于 MEM 级数据前推。
mem_wdata_i	32	Input	MEM 阶段	访存阶段要写回的数据。用于 MEM 级数据前推。
mem_wd_i	5	Input	MEM 阶段	访存阶段指令要写入的目标寄存器地址。
reg1_data_i	32	Input	Regfile	通用寄存器堆端口 1 读出的原始数据。
reg2_data_i	32	Input	Regfile	通用寄存器堆端口 2 读出的原始数据。

信号名称	位宽	I/O 类型	来源	功能描述
is_in_delayslot_i	1	Input	IF/ID	表示当前指令是否处于分支指令的延迟槽中。

表 3-4 id 模块输出接口表格

信号名称	位宽	I/O 类型	去向	功能描述
reg1_read_o	1	Output	Regfile	寄存器堆端口 1 读使能信号。
reg2_read_o	1	Output	Regfile	寄存器堆端口 2 读使能信号。
reg1_addr_o	5	Output	Regfile	寄存器堆端口 1 读地址。
reg2_addr_o	5	Output	Regfile	寄存器堆端口 2 读地址。
aluop_o	8	Output	ID/EX	译码出的子操作码（如 EXE_OR_OP）。
alusel_o	3	Output	ID/EX	译码出的运算类型（如逻辑、算术、移位等）。
reg1_o	32	Output	ID/EX	最终确定的操作数 1（已处理前推）。

信号名称	位宽	I/O 类型	去向	功能描述
reg2_o	32	Output	ID/EX	最终确定的操作数 2（已处理前推或立即数）。
wd_o	5	Output	ID/EX	当前指令准备写入的目标寄存器地址。
wreg_o	1	Output	ID/EX	当前指令是否有写回寄存器的需求。
branch_flag_o	1	Output	pc_reg	跳转标志，为 1 时通知 PC 发生跳转。
branch_target_address_o	32	Output	pc_reg	计算出的跳转目标地址。
stallreq	1	Output	CTRL	暂停请求信号，由 Load-Use 相关触发。
excepttype_o	32	Output	后续流水级	异常类型码（如系统调用 syscall、非法指令等）。

3.3.4.regfile 模块

表 3-5 regfile 模块接口表格

信号名称	位宽	I/O 类型	来源/去向	功能描述
clk	1	Input	全局时钟	写入触发信号。数据在时钟上升沿且 we 有效时存入。
rst	1	Input	全局复位	高电平有效。复位时所有寄存器清零。
we	1	Input	WB (回写阶段)	写使能信号。为 1 时允许将数据写入 waddr 指定的寄存器。
waddr	5	Input	WB (回写阶段)	写地址。指定要更新的寄存器编号 (0-31)。
wdata	32	Input	WB (回写阶段)	写数据。即最终计算或访存得到的结果。
re1	1	Input	ID (译码阶段)	读使能 1。控制是否开启第一个读端口。
raddr1	5	Input	ID (译码阶段)	读地址 1。指定源寄存器 rs 的编号。
rdata1	32	Output	ID (译码阶段)	读数据 1。输出 raddr1 对应寄存器的值。

信号名称	位宽	I/O 类型	来源/去向	功能描述
re2	1	Input	ID (译码阶段)	读使能 2。控制是否开启第二个读端口。
raddr2	5	Input	ID (译码阶段)	读地址 2。指定源寄存器 rt 的编号。
rdata2	32	Output	ID (译码阶段)	读数据 2。输出 raddr2 对应寄存器的值。
s0_data_o	32	Output	FPGA 外设	寄存器监控引脚。专门引出 16 号寄存器 ($s0$) 的值供数码管显示。

3.3.5.Id_ex 模块

表 3-6 id_ex 模块接口表格

信号名称	位宽	I/O 类型	来源/去向	功能描述
clk / rst	1 / 1	Input	全局信号	同步时钟信号与异步复位信号（高电平有效）。
stall	6	Input	CTRL 模块	流水线暂停信号。stall[2] 有效时，保持 EX 级数据不变。
flush	1	Input	CTRL 模块	冲刷信号。有效时，清空当前段寄存器（插入 NOP 或清除异常数据）。

信号名称	位宽	I/O 类型	来源/去向	功能描述
id_aluop	8	Input	ID 阶段	译码出的子操作码。
id_alusel	3	Input	ID 阶段	译码出的运算类型（算术、逻辑、移位等）。
id_reg1 / id_reg2	32 / 32	Input	ID 阶段	经过前推处理后的源操作数 1 和 2。
id_wd	5	Input	ID 阶段	当前指令的目的寄存器写入地址。
id_wreg	1	Input	ID 阶段	当前指令是否需要写回目的寄存器使能。
id_link_address	32	Input	ID 阶段	专门用于跳转连接指令（如 JAL/JALR）的返回地址。
id_is_in_delayslot	1	Input	ID 阶段	标记当前正在执行阶段的指令是否处于延迟槽中。
id_inst	32	Input	ID 阶段	传递原始指令内容（用于加载存储指令的偏移计算）。
id_excepttype	32	Input	ID 阶段	传递 ID 级识别出的异常类型。
ex_aluop / ex_alusel	8 / 3	Output	EX 阶段	传递给执行阶段的运算控制信号。
ex_reg1 / ex_reg2	32 / 32	Output	EX 阶段	传递给执行阶段的操作数。
ex_wd / ex_wreg	5 / 1	Output	EX 阶段	传递给执行阶段的写回控制信息。
ex_link_address	32	Output	EX 阶段	传递给执行阶段的返回地址（用于链接跳转）。
ex_is_in_delayslot	1	Output	EX 阶段	告知 EX 级当前指令是否在延迟槽

信号名称	位宽	I/O 类型	来源/去向	功能描述
				中。
ex_inst	32	Output	EX 阶段	传递给执行阶段的原始指令。
ex_excepttype	32	Output	EX 阶段	传递给执行阶段的异常信息。

3.3.6.ex 模块

表 3-7 运算输入与基本控制信号表格

信号名称	位宽	I/O 类型	来源/去向	功能描述
aluop_i / alusel_i	8 / 3	Input	ID/EX	运算子操作码与运算类型（控制 ALU 做何种计算）。
reg1_i / reg2_i	32 / 32	Input	ID/EX	源操作数 1 和 2。
wd_i / wreg_i	5 / 1	Input	ID/EX	目的寄存器地址及写使能。
inst_i	32	Input	ID/EX	当前执行的指令源码，用于提取加载/存储的偏移量。

表 3-8 HI/LO 寄存器接口（含数据前推）表格

信号名称	位宽	I/O 类型	来源/去向	功能描述
hi_i / lo_i	32 / 32	Input	HILO 寄存	从 HI/LO 寄存器读出的

信号名称	位宽	I/O 类型	来源/去向	功能描述
			器	当前值。
mem_hi_i / lo_i	32 / 32	Input	MEM 阶段	前推信号：来自访存级，解决 HI/LO 的数据相关。
wb_hi_i / lo_i	32 / 32	Input	WB 阶段	前推信号：来自回写级，解决 HI/LO 的数据相关。
hi_o / lo_o / whilo_o	32/32/1	Output	EX/MEM	最终算出的 HI/LO 新值及写使能信号。

表 3-9 除法器与多周期运算接口表格

信号名称	位宽	I/O 类型	来源/去向	功能描述
div_result_i / ready_i	64 / 1	Input	DIV 模块	除法运算结果及运算完成标志。
div_start_o / signed_o	1 / 1	Output	DIV 模块	启动除法信号及是否为有符号除法。
hilo_temp_o / cnt_o	64 / 2	Output	EX/MEM	暂存乘法中间结果和周期计数（用于多周期乘法）。

表 3-10 访存与异常控制（Load/Store/CP0）接口表格

信号名称	位宽	I/O 类型	来源/去向	功能描述
mem_addr_o	32	Output	MEM 阶段	有效地址计算：基址 + 偏移量的计算结果。

信号名称	位宽	I/O 类型	来源/去向	功能描述
aluop_o	8	Output	MEM 阶段	传递操作码，告知访存级是 LB, LW 还是 SB 等。
cp0_reg_data_i	32	Input	CP0 模块	读取特权寄存器的值。
excepttype_o	32	Output	EX/MEM	最终确定的异常类型（整合 ID 级与 EX 级的异常）。
stallreq	1	Output	CTRL	暂停请求：当除法未完成或多周期运算时请求停顿。

3.3.7. Ex_mem 模块

表 3-11 ex_mem 模块接口表格

信号名称	位宽	I/O 类型	来源/去向	功能描述
clk / rst	1 / 1	Input	全局信号	同步时钟与异步复位信号（高有效）。
stall / flush	6 / 1	Input	CTRL 模块	流水线暂停（检查 stall[3]）与冲刷信号。

信号名称	位宽	I/O 类型	来源/去向	功能描述
ex_wd / ex_wreg	5 / 1	Input	EX 阶段	执行阶段算出的目的寄存器地址及写使能。
ex_wdata	32	Input	EX 阶段	执行阶段算出的寄存器写回数据（如加法结果）。
ex_hi / ex_lo	32 / 32	Input	EX 阶段	执行阶段算出的 HI/LO 寄存器新值。
ex_whilo	1	Input	EX 阶段	执行阶段是否需要更新 HI/LO 寄存器。
ex_aluop	8	Input	EX 阶段	传递操作码，告知 MEM 阶段具体的访存类型（如 LW, SB）。
ex_mem_addr	32	Input	EX 阶段	访存物理地址（由 EX 阶段的加法器算出）。
ex_reg2	32	Input	EX 阶段	存储指令的数据（如 SW 指令要存入内存的值）。
ex_cp0_reg_...	混合	Input	EX 阶段	传递执行阶段涉及 CP0 寄存器写的控制信号与数据。
ex_excepttype	32	Input	EX 阶段	传递当前指令产生的最终异常类型编码。
mem_wd / mem_wreg	5 / 1	Output	MEM 阶段	向访存级传递目的寄存器地址与写使能。
mem_wdata	32	Output	MEM 阶段	向访存级传递计算结果。
mem_hi / mem_lo	32 / 32	Output	MEM 阶段	向访存级传递待写入 HI/LO 的值（用于数据前推）。

信号名称	位宽	I/O 类型	来源/去向	功能描述
mem_mem_addr	32	Output	MEM 阶段	向访存级（及外部 RAM）输出目标地址。
mem_reg2	32	Output	MEM 阶段	向访存级输出准备写入存储器的数据。
hilo_o / cnt_o	66	Output	EX 阶段	反馈信号：将多周期乘法的中间结果传回 EX 级。

3.3.8.mem 模块

表 3-12 来自执行级（EX）与回写级（WB）的数据输入

信号名称	位宽	I/O 类型	来源	功能描述
rst	1	Input	全局复位	模块复位。
wd_i / wreg_i	5 / 1	Input	EX/MEM	目的寄存器地址及其写使能信号。
wdata_i	32	Input	EX/MEM	来自执行级的通用运算结果。
hi_i / lo_i / whilo_i	32x2 / 1	Input	EX/MEM	待写入 HI/LO 寄存器的数据及使能。
aluop_i	8	Input	EX/MEM	运算子操作码，用于判定具体的访存指令类型。
mem_addr_i	32	Input	EX/MEM	访存物理地址（基址 + 偏移量）。
reg2_i	32	Input	EX/MEM	存储指令（Store）要写入内存的原始数据。

信号名称	位宽	I/O 类型	来源	功能描述
wb_LLbit_...	1 / 1	Input	WB 阶段	前推信号：来自回写级最新的 LLbit 状态。
wb_cp0_reg_...	混合	Input	WB 阶段	前推信号：来自回写级最新的 CP0 寄存器状态。

表 3-13 与外部数据存储器（Data RAM）的交互接口

信号名称	位宽	I/O 类型	去向	功能描述
mem_data_i	32	Input	Data RAM	从内存中读出的原始 32 位字数据。
mem_addr_o	32	Output	Data RAM	输出到内存的访问地址。
mem_we_o	1	Output	Data RAM	内存写使能。对于 Store 指令为有效，Load 为无效。
mem_sel_o	4	Output	Data RAM	字节选通信号。决定对 32 位字中的哪几个字节进行读写。
mem_data_o	32	Output	Data RAM	要写入内存的数据（经过字节对齐调整）。
mem_ce_o	1	Output	Data RAM	存储器芯片使能信号。

表 3-14 输出到回写级（WB）与异常处理

信号名称	位宽	I/O 类型	去向	功能描述

wdata_o	32	Output	MEM/WB	最终写回数据。若是 Load 指令，则为经对齐处理后的内存数据。
excepttype_o	32	Output	CTRL/CP0	最终确定的异常类型。在此处结合外部中断进行最终判定。
cp0_epc_o	32	Output	CTRL	输出 EPC 寄存器的值，用于异常跳转返回。
LLbit_we/value_o	1 / 1	Output	LLbit 寄存器	输出到 LLbit 寄存器的写控制。

3.3.9. Mem_wb 模块

表 3-15 mem_wb 模块接口描述表

信号名称	位宽	I/O 类型	来源/去向	功能描述
clk / rst	1 / 1	Input	全局信号	同步时钟与异步复位信号（高有效）。
stall / flush	6 / 1	Input	CTRL 模块	流水线暂停（检查 stall[4]）与冲刷信号。
mem_wd / mem_wreg	5 / 1	Input	MEM 阶段	访存级确定的目的寄存器地址及写使能信号。
mem_wdata	32	Input	MEM 阶段	访存级确定的最终写回数据（运算结果或 Load 数据）。
mem_hi / mem_lo	32 / 32	Input	MEM 阶段	访存级确定的待写入 HI/LO 寄存器的数据。
mem_whilo	1	Input	MEM 阶段	访存级确定的 HI/LO 写使能信号。
mem_LLbit_...	1 / 1	Input	MEM 阶段	传递原子指令所需的 LLbit 写使能及数值。

信号名称	位宽	I/O 类型	来源/去向	功能描述
mem_cp0_reg_...	混合	Input	MEM 阶段	传递特权指令要写入 CP0 寄存器的地址与数据。
wb_wd / wb_wreg	5 / 1	Output	WB (Regfile)	输出到通用寄存器堆的写地址与写使能。
wb_wdata	32	Output	WB (Regfile)	输出到通用寄存器堆的最终写入数据。
wb_hi / wb_lo	32 / 32	Output	HILO 寄存器	输出到 HI/LO 寄存器的最终写入数据。
wb_whilo	1	Output	HILO 寄存器	输出到 HI/LO 寄存器的最终写使能。
wb_cp0_reg_...	混合	Output	CP0 模块	输出到协处理器 CP0 的最终写控制信号。

3.3.10. Hilo_reg 模块

表 3-16 hilo_reg 模块接口描述表

信号名称	位宽	I/O 类型	来源/去向	功能描述
clk	1	Input	全局时钟	寄存器更新的时钟参考，上升沿触发写入。
rst	1	Input	复位模块	全局复位信号。有效时将 HI 和 LO 寄存器清零。
we	1	Input	WB 阶段	写使能信号。由回写阶段控制是否更新寄存器。
hi_i	32	Input	WB 阶段	待写入 HI 寄存器的数据（通常为乘法的 位或除法的余数）。

lo_i	32	Input	WB 阶段	待写入 LO 寄存器的数据（通常为乘法的 [31:0] 位或除法的商）。
hi_o	32	Output	EX 阶段	HI 寄存器的当前值，送往执行阶段用于 MFHI 指令。
lo_o	32	Output	EX 阶段	LO 寄存器的当前值，送往执行阶段用于 MFLO 指令。

3.3.11.div 模块

表 3-17 div 模块接口描述表

信号名称	位宽	I/O 类型	来源/去向	功能描述
clk / rst	1 / 1	Input	全局信号	同步工作时钟与异步复位信号。
signed_div_i	1	Input	EX 阶段	符号控制：为 1 表示有符号除法（DIV），为 0 表示无符号除法（DIVU）。
opdata1_i	32	Input	EX 阶段	被除数（Dividend）。
opdata2_i	32	Input	EX 阶段	除数（Divisor）。
start_i	1	Input	EX 阶段	启动信号：为 1 时开始除法迭代运算。
annul_i	1	Input	EX 阶段	中止信号：发生异常或冲刷时，强制停止当前的除法运算。
result_o	64	Output	EX 阶段	运算结果：[63:32] 位为余数，[31:0] 位为商。
ready_o	1	Output	EX 阶段	就绪信号：运算完成标志。为 1 时 EX 阶段才停止申请暂停。

表 3-18 LLbit_reg 模块接口描述表

信号名称	位宽	I/O 类型	来源/去向	功能描述
clk / rst	1 / 1	Input	全局信号	同步工作时钟与异步复位信号（高有效）。
flush	1	Input	CTRL 模块	冲刷信号：发生异常或中断跳转时，通常需清空 LLbit 以保证原子性。
LLbit_i	1	Input	MEM/WB 阶段	准备写入 LLbit 的新值。
we	1	Input	MEM/WB 阶段	写使能：由 LL 指令（置 1）或 SC 指令（结果写入后处理）控制。
LLbit_o	1	Output	MEM 阶段	读取状态：输出给访存阶段，决定 SC 指令是否执行成功。

3.3.12. CP0 模块

表 3-19 cp0_reg 模块接口描述表

信号名称	位宽	I/O 类型	来源/去向	功能描述
clk / rst	1 / 1	Input	全局信号	同步时钟与异步复位信号（高有效）。
we_i	1	Input	WB 阶段	写使能。控制是否更新特权寄存器。
waddr_i / raddr_i	5 / 5	Input	WB / EX	读写地址。指定访问哪个 CP0 寄存器（如 Status, Cause 等）。
data_i	32	Input	WB 阶段	准备写入特权寄存器的数据（来自 MTC0 指令）。
excepttype_i	32	Input	MEM 阶段	异常类型。告知 CP0 当前发生了什么异常（系统调用、非法指令等）。

信号名称	位宽	I/O 类型	来源/去向	功能描述
int_i	6	Input	外部设备	外部中断输入。来自硬件设备的 6 路异步中断信号。
current_inst_addr_i	32	Input	MEM 阶段	触发异常的指令地址，用于保存到 EPC。
is_in_delayslot_i	1	Input	MEM 阶段	标记触发异常的指令是否处于分支延迟槽中。
data_o	32	Output	EX 阶段	读数据。输出被读取寄存器的值（供给 MFC0 指令）。
status_o / cause_o	32 / 32	Output	MEM / CTRL	核心状态输出。反映当前中断屏蔽位及异常原因。
epc_o	32	Output	CTRL / PC	异常返回地址。存储异常处理完后应返回的指令位置。
timer_int_o	1	Output	内部计数器	定时器中断。当 Count 达到 Compare 时产生的内部时钟中断。

3.3.13. Inst_rom 模块

表 3-20 inst_rom 模块接口描述表

信号名称	位宽	I/O 类型	来源/去向	功能描述
ce	1	Input	pc_reg	芯片使能信号。为 1 时模块工作，为 0 时输出空指令（或高阻态）。
addr	32	Input	pc_reg	指令地址信号。由程序计数器 PC 提供，决定读取哪一单元。
inst	32	Output	IF/ID	指令数据输出。输出对应地址处的 32 位机器码。

3.3.14. Data_ram 模块

表 3-21 data_ram 模块接口描述表

信号名称	位宽	I/O 类型	来源/去向	功能描述
clk	1	Input	全局时钟	存储器工作时钟。在 we 有效时，数据在时钟上升沿写入。
ce	1	Input	MEM 阶段	芯片使能信号。只有该信号有效时，才允许进行读写操作。
we	1	Input	MEM 阶段	写使能信号。为 1 时执行存储（Store），为 0 时执行加载（Load）。
addr	32	Input	MEM 阶段	访存物理地址。由执行阶段（EX）计算所得。
sel	4	Input	MEM 阶段	字节选择信号（Byte Enable）。决定 32 位字中哪些字节被读写。
data_i	32	Input	MEM 阶段	待写入数据。存储指令（如 SW, SB）传来的数据。
data_o	32	Output	MEM 阶段	读出数据。传回给 MEM 模块进行对齐和符号扩展。

3.3.15. Seg_driver 模块

表 3-22 Seg_driver 模块接口表格

信号名称	方向	位宽	描述	硬件/逻辑连接建议
clk	input	1	系统主时钟	接开发板 100MHz 晶振
rst	input	1	异步复位信号	低电平有效。建议连接到 SOPC 的全局复位信号
data_in	input	32	待显示数据	连接到 SOPC 中的 display_reg (锁存 0x3000 地址的数据)
seg_o	output	8	数码管段选信号	连接到数码管引脚 (a, b, c, d, e, f, g, dp)
dig_en_o	output	8	数码管位选使能	连接到 8 个数码管的公共端 (Common Anode/Cathode)

4. 性能优化与分析

4.1. 流水线效率优化：五级架构与多周期指令处理

深度的平衡（五级流水线）：将经典的三级流水线扩展为五级（取指、译码、执行、访存、回写）。这种设计有效地拆分了耗时较长的“访存”阶段，使得每个阶段的时钟周期（T）能够进一步缩短，从而提高了主频上限。

多周期指令暂停机制（Stall Unit）：引入了 CTRL 模块。针对乘加（MADD）、乘减（MSUB）及除法（DIV）等无法在一个周期内完成的复杂运算，设计了局部暂停策略：仅保持流水线前段（PC、IF、ID、EX）不动，允许后段（MEM、WB）继续执行。这种“精准停顿”避免了全流水线的盲目等待。

硬件专用除法器（DIV）：采用“试商法”并独立成模块。虽然 32 位除法需要 32 个周期，但通过独立模块配合流水线暂停，确保了高功耗/高复杂操作不会破坏指令流的正确性。

4.2. 消除流水线冒险（Hazard Mitigation）

数据相关：数据前推（Data Forwarding）：这是性能优化的核心。针对 RAW（写后读）相关，并没有简单地采用“暂停等待”的方法，而是将 EX 阶段和 MEM 阶段的结果直接反馈（前推）给 ID 阶段。

优化效果：这消除了相邻指令、相隔 1 条、相隔 2 条指令之间绝大部分的数据相关，使得流水线在遇到数据依赖时依然能保持满载运行。

Load 相关：检测与单周期气泡：由于 Load 指令的数据在 MEM 阶段末尾才能产生，无法通过纯前推解决（Load-Use Hazard）。设计了检测机制，仅在必要时插入一个时钟周期的气泡（Stall）。

性能权衡：这种设计平衡了硬件复杂度与执行效率，比完全依赖软件插入 NOP 指令要高效得多。

4.3. 控制流优化：延迟槽与分支预测

分支判断前移（Early Branch Decision）：将转移指令的判断从 EX 阶段提前到了 ID 阶段。

优化分析：在执行阶段判断转移会导致 2 个周期的损失（2 条无效指令），而提前到译码阶段配合延迟槽，将损失降低到了零周期或一周期，极大地提高了分支密集型程序的执行速度。

延迟槽（Branch Delay Slot）机制：通过“延迟槽指令总是被执行”的原则，充分利用了分支跳转时的时钟周期，确保取指单元在改变 PC 地址时，流水线依然在做有用功。

4.4. 存储与异常处理性能

哈佛架构思想的应用：通过独立的 data_ram 设计，实现了指令存储与数据存储的解耦。这允许 CPU 在同一时钟周期内进行取指（IF）和访存（MEM），避免了单总线架构下的结构冒险（Structural Hazard）。

精确异常处理（Precise Exception）：采用“异常标记+访存阶段统一处理”的策略。

性能优势：异常不会立即打断流水线，而是随指令流动。这样可以确保异常处理前，较早进入流水线的指令能完成写回，而较晚进入的指令被撤销，保证了异常恢复后的状态一致性，减少了因异常导致的大规模重算损失。

4.5. 特殊寄存器与原子操作优化

HILO 独立路径：HILO 寄存器与通用寄存器物理隔离，并配备了专门的数据前推路径。这意味着连续的乘法运算不会因为竞争通用寄存器堆（Regfile）的端口而造成阻塞。

原子操作（LL/SC）支持：引入链接加载和条件存储，为多任务/多线程环境下的同步提供了底层硬件支持。虽然不直接提升单核跑分，但极大地优化了多任务操作系统中的临界区性能（避免了繁重的关中断操作）。

5. 实现指令说明及仿真测试展示

5.1. 实现指令说明

5.1.1. 逻辑与移位类指令

包括 AND、OR、XOR、NOR 及其立即数形式，以及 SLL、SRL、SRA 等移位操作。这些指令主要由 ALU 的逻辑与移位单元完成，对应的 AluSel 通常设置为逻辑或移位结果。

5.1.2. 数据移动指令

如 MOVZ、MOVN、MFHI、MTHI、MFLO、MTLO 等，用于通用寄存器与 HI/LO 特殊寄存器之间的数据传输，部分指令带条件写回特性。

5.1.3. 算术与比较指令

包括 ADD、ADDU、SUB、SLT、SLTU 及其立即数版本。这类指令由算术逻辑单元完成，支持有符号与无符号运算，并可能涉及溢出判断。

5.1.4. 乘除法与扩展算术指令

包含 MULT、MULTU、DIV、DIVU 以及 MADD、MSUB 等扩展指令。结果通常写入 HI/LO 寄存器，部分指令需要多周期执行与状态控制。

5.1.5. 跳转与分支指令

如 J、JAL、JR、BEQ、BNE、BGEZ 等，用于控制程序流。该类指令会影响 PC 更新，并可能涉及延迟槽处理。

5.1.6. 访存指令

包括 LB、LH、LW、SB、SH、SW 等，用于 CPU 与数据存储器之间的数据交互，对齐方式和符号扩展方式依指令不同而变化。

5.1.7. 异常与系统指令

如 SYSCALL、ERET、TRAP 类指令，用于系统调用、异常返回和调试控制，通常与 CP0 协处理器配合使用。

5.1.8. CP0 寄存器相关指令

通过 MFC0、MTC0 指令访问 CP0 中的状态、异常原因、EPC 等寄存器，用于异常管理与中断控制。

5.1.9. ALU 操作码与结果选择

AluOp 用于区分具体运算类型，AluSel 用于选择结果通路，如逻辑结果、算术结果、访存地址或跳转目标。

5.2. 仿真测试展示

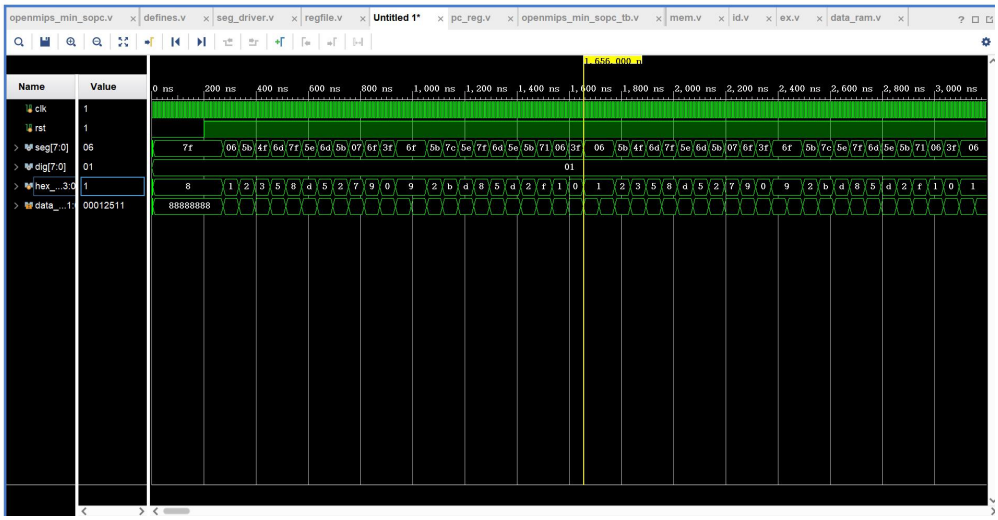

图 5-1 斐波那契数列测试程序仿真波形图

观察 hex_val[3:0] 的波形标签，可以清晰地看到数列的产生：

序列验证：图中显示了 1 -> 2 -> 3 -> 5 -> 8 -> d (13) -> 15 (21) -> 27...（注意：波形显示的是十六进制）。

可见 CPU 运算逻辑、数据前推和寄存器更新运行非常快，每隔几个时钟周期就计算出一个新值。

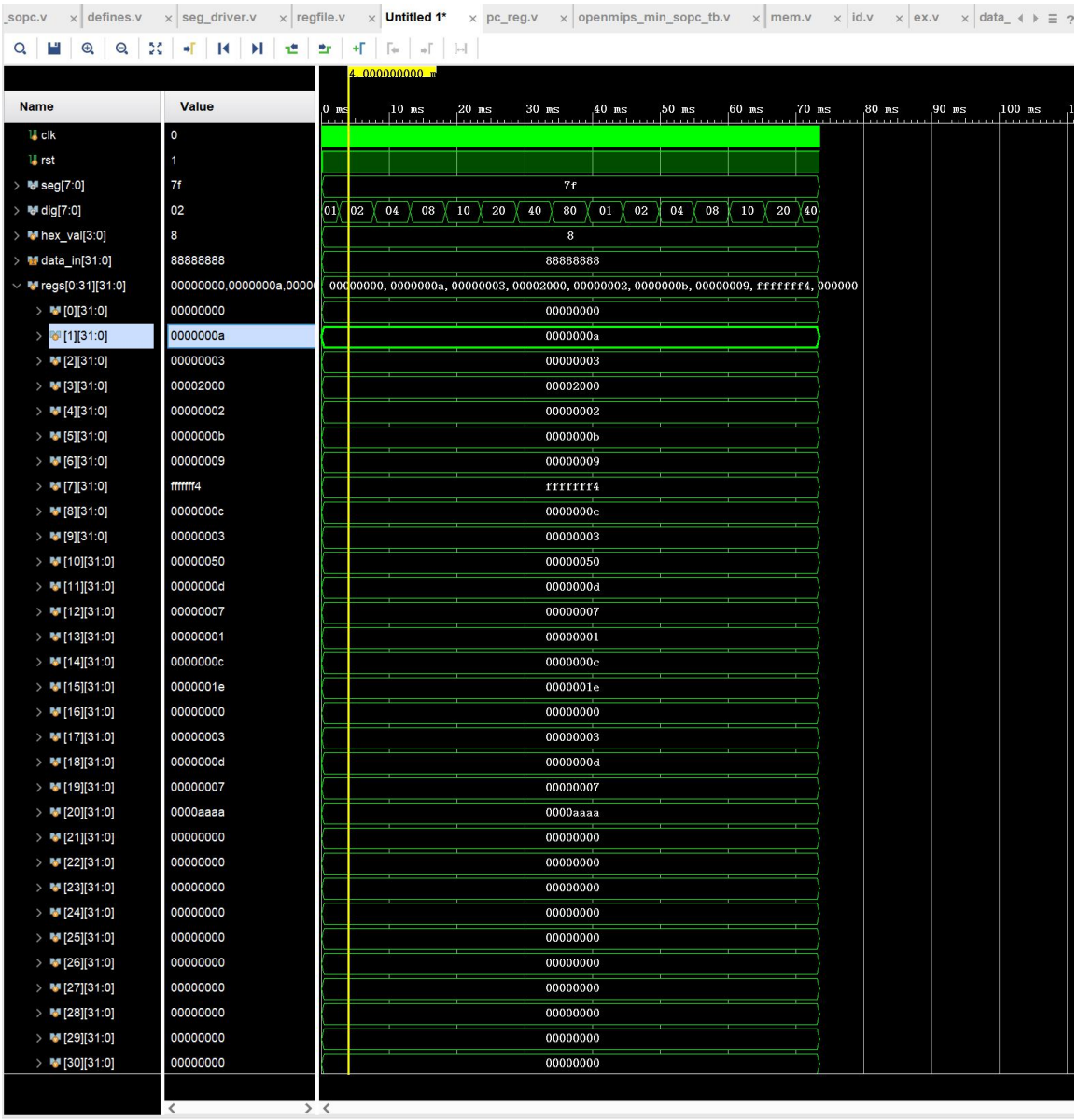

图 5-2 其余指令分类别挑选测试

相关测试代码请见附录一测试程序部分，总体思路为挑选每个类别实现的指令进行统一测试，仿真效果图如上，对比测试程序想要得到仿真结果（寄存器中赋值改变），可见指令执行后的寄存器均满足测试要求。

表 5-1 测试程序执行完成后各寄存器值

寄存器	十六进制值	十进制值	来源/计算过程
$1	0x0000000A	10	初始化及 mfc0 读回
$2	0x00000003	3	初始化
$3	0x00002000	8192	基地址初始化 (lui + ori)
$4	0x00000002	2	10 & 3 (AND)
$5	0x0000000B	11	`10
$6	0x00000009	9	10 ^ 3 (XOR)
$7	0xFFFFFFF4	-12	`~(10
$8	0x0000000C	12	3 << 2 (SLL)
$9	0x00000006	6	12 >> 1 (SRL)
$10	0x00000050	80	10 << 3 (SLLV)
$11	0x0000000D	13	10 + 3 (ADD)
$12	0x00000007	7	10 - 3 (SUB)
$13	0x00000001	1	3 < 10 (SLT) 为真
$14	0x0000000C	12	7 + 5 (ADDI)
$15	0x0000001E	30	10 * 3 的低位结果 (MFLO)
$16	0x00000000	0	10 * 3 的高位结果 (MFHI)
$17	0x00000003	3	10 / 3 的商 (MFLO)
$18	0x0000000D	13	从内存 0x2000 读出的值 (LW)
$19	0x00000007	7	从内存 0x2004 读出的值 (LW)
$20	0x0000AAAA	43690	beq 失败后执行的 ori

6. 总结

以前学组成原理只是看图说话，这次是自己用 Verilog 把 IF、ID、EX、MEM、WB 这五级流水线一个个模块写出来的。我弄清楚了 32 位指令是怎么从存储器里取出来，又是怎么在译码阶段被拆解成寄存器地址和操作码的。特别是写各个模块间的流水线寄存器时，我明白了数据是怎么在时钟驱动下一级一级往后传的。

在处理数据前推（Forwarding）部分，我实现了 EX 和 MEM 阶段的结果反馈路径。原本后面指令要等前面指令写回寄存器才能读，现在通过旁路逻辑直接把结果“截获”给下一条指令，解决了写后读（RAW）的问题。

针对 Load 指令后面紧跟相关指令的情况，我设计了冲突检测单元，让流水线强行“歇一个周期”，防止读到错数据。

设计延迟槽（Delay Slot），搞清楚了为什么跳转指令后的那条指令也会执行，并在译码阶段提前进行了分支判断，把跳转带来的开销降到了最低。

我独立实现了 HI/LO 寄存器，并写了一个状态机来管理除法器的运行。当除法没算完时，我会让整个流水线停下来等着（Stall），直到结果出来。这让我理解了长周期指令在流水线里是怎么“占位”的。

我设计了 CP0 寄存器组，实现了像 syscall 这样的系统调用。我学会了如何通过逻辑捕捉异常信号，并让 PC 指针强行跳转到异常入口地址（0x40000040）。当看到程序报错能跳到预定位置，而不是直接跑飞时。

这次课设有一半的时间是在看 Vivado 的仿真波形。当寄存器里的数不对时，我要从 WB 阶段倒着往回查，看是哪一级的信号在传递时断了或者变了。这种“查线”的过程很枯燥，但通过波形图定位到逻辑漏洞的那一刻，确实很有成就感。

以前觉得指令集、流水线这些词都很抽象，但当我把 83 条指令全部写完，并转成机器码存进 ROM，最后在开发板上看到它真的能算出斐波那契数列时，我才意识到这些理论是怎么变成实实在在的电路的。

自己造过一遍 CPU 后才发现，看似简单的加减乘除，在底层都要经过异常检测、数据前推、时序对齐等一系列极其复杂的配合。这次课设让我不再觉得 CPU 是一个黑盒，而是由一个个严谨的逻辑门拼出来的精密系统。

7. 参考文献

[1] 雷思磊．自己动手写 CPU [M]．北京：电子工业出版社，2014．

[2] 高亚军．Vivado 从此开始 [M]．北京：电子工业出版社，2017．

[3] 依爱普（北京）科技有限公司．EGO-1 FPGA 硬件指南 [R]．北京：依爱普科技，2016．

[4] 依爱普（北京）科技有限公司．EGO-1 FPGA 开发板使用手册 [R]．北京：依爱普科技，2018．

[5] Xilinx Inc. Vivado Design Suite User Guide: Design Flows Overview (UG892) [R]. San Jose: Xilinx, 2022.

[6] Xilinx Inc. Vivado Design Suite 典型设计流程指南 [EB/OL]. [2024-12-01]

510

8. 附录一（测试程序）

● 斐波那契数列测试仿真版

```
34043000   // [0x00] ori  $4, $0, 0x3000 (初始化显示地址)
34020001   // [0x04] ori  $2, $0, 1      (f(n) 初始值)
34030001   // [0x08] ori  $3, $0, 1      (f(n-1) 初始值)

ac820000   // [0x0C] loop: sw $2, 0($4)  (将当前项写入显示寄存器)
00430821   // [0x10] addu $1, $2, $3     (计算新项: $1 = $2 + $3)
00021821   // [0x24] addu $3, $0, $2     (更新旧项: $3 = $2)
00011021   // [0x28] addu $2, $0, $1     (更新当前项: $2 = $1)

08000003   // [0x1C] j 0x0C              (跳回 0x0C 继续循环)
00000000   // [0x20] nop                 (跳转延迟槽)
```

● 斐波那契数列测试实体硬件展示版

```
34043000   // [0x00] ori  $4, $0, 0x3000
34020001   // [0x04] ori  $2, $0, 1
34030001   // [0x08] ori  $3, $0, 1
ac820000   // [0x0C] sw   $2, 0($4)
3c0500F0   // [0x10] lui  $5, 0x00F0
24a5ffff   // [0x14] delay: addiu $5, $5, -1
14a0fffe   // [0x18] bne  $5, $0, delay
00000000   // [0x1C] nop
00430821   // [0x20] addu $1, $2, $3
00021821   // [0x24] addu $3, $0, $2
00011021   // [0x28] addu $2, $0, $1
08000003   // [0x2C] j 0x0C
00000000   // [0x30] nop
```

● 其余指令分类别挑选测试

```
// --- 第一阶段：寄存器初始化与逻辑运算 ---
3401000A    // [00] ori  $1, $0, 10      ; $1 = 10 (0xA)
34020003    // [04] ori  $2, $0, 3       ; $2 = 3
3C030000    // [08] lui  $3, 0x0000      ; $3 = 0x00000000
34632000    // [0C] ori  $3, $3, 0x2000  ; $3 = 0x2000 (基地址)
00222024    // [10] and  $4, $1, $2      ; $4 = 10 & 3 = 2
00222825    // [14] or   $5, $1, $2      ; $5 = 10 | 3 = 11
00223026    // [18] xor  $6, $1, $2      ; $6 = 10 ^ 3 = 9
00223827    // [1C] nor  $7, $1, $2      ; $7 = ~(10 | 3) = 0xFFFFFFF4

// --- 第二阶段：移位运算 ---
00024080    // [20] sll  $8, $2, 2       ; $8 = 3 << 2 = 12
00084882    // [24] srl  $9, $8, 1       ; $9 = 12 >> 1 = 6
00415004    // [28] sllv $10, $1, $2     ; $10 = 10 << 3 = 80

// --- 第三阶段：算术运算 ---
00225820    // [2C] add  $11, $1, $2     ; $11 = 10 + 3 = 13
00226022    // [30] sub  $12, $1, $2     ; $12 = 10 - 3 = 7
0041682A    // [34] slt  $13, $2, $1     ; $13 = (3 < 10) = 1
218E0005    // [38] addi $14, $12, 5     ; $14 = 7 + 5 = 12

// --- 第四阶段：HILO 寄存器与乘除法 ---
00220018    // [3C] mult $1, $2          ; HI/LO = 10 * 3 = 30
00007812    // [40] mflo $15             ; $15 = 30
00008010    // [44] mfhi $16             ; $16 = 0
0022001A    // [48] div  $1, $2          ; LO=3, HI=1 (10/3=3...1)
00008812    // [49] mflo $17             ; $17 = 3
```

```
// --- 第五阶段：存储与加载 (Load/Store) ---
AC6B0000    // [50] sw   $11, 0($3)     ; mem[0x2000] = 13
AC6C0004    // [54] sw   $12, 4($3)     ; mem[0x2004] = 7
8C720000    // [58] lw   $18, 0($3)     ; $18 = 13
8C730004    // [5C] lw   $19, 4($3)     ; $19 = 7

// --- 第六阶段：跳转与分支控制 ---
12530002    // [60] beq  $18, $19, +2    ; 13 == 7 ? 假，不跳转
00000000    // [64] nop                  ; 延迟槽
3414AAAA    // [68] ori  $20, $0, 0xAAAA ; 执行到此处说明 beq 没跳
16530002    // [6C] bne  $18, $19, +2    ; 13 != 7 ? 真，跳转至 [7C]
00000000    // [70] nop                  ; 延迟槽
3415BBBB    // [74] ori  $21, $0, 0xBBBB ; 该指令被跳过

08000021    // [78] j    0x84            ; 跳转到 [84]
00000000    // [7C] nop                  ; 延迟槽 (由于 bne 跳转，PC 来到这里)
3416CCCC    // [80] ori  $22, $0, 0xCCCC ; 只有 J 指令没执行前才会运行

// --- 第七阶段：系统与特权 (CP0) ---
40816000    // [84] mtc0 $1, $12         ; 将 10 写入 CP0_Status ($12)
40016000    // [88] mfc0 $1, $12         ; 从 CP0 读回数据到 $1
0000000C    // [8C] syscall              ; 触发异常

// --- 结束循环 ---
08000024    // [90] j    0x90            ; 死循环
00000000    // [94] nop
```