

**2023~2024学年 第一学期**

**《系 统 硬 件 综 合 设 计》**

设 计 报 告

班 级 \_\_\_\_计科21-3班\_\_\_\_

学 号 \_\_\_\_2021218152\_\_\_\_

姓 名 \_\_\_\_\_\_\_陈嘉乐\_\_\_\_\_\_\_

2024年1月7日

**目 录**

[1 设计要求 3](#_Toc156132213)

[1.1 CPU处理指令的过程 3](#_Toc156132214)

[1.1.1 指令格式 4](#_Toc156132215)

[1.1.2 指令处理流程概述 5](#_Toc156132216)

[2 指令集设计 5](#_Toc156132217)

[2.1 算数指令设计 5](#_Toc156132218)

[2.1.1 ADD & ADDU 5](#_Toc156132219)

[2.1.2 SUB & SUBU 7](#_Toc156132220)

[２.１.３SLT & SLTU 9](#_Toc156132221)

[2.1.4 CLO & CLZ 10](#_Toc156132222)

[2.2 逻辑指令设计 11](#_Toc156132223)

[2.2.1 and & or & xor & nor 11](#_Toc156132224)

[2.2.2 andi & xori 13](#_Toc156132225)

[3.五级流水线设计 14](#_Toc156132226)

[3.1流水线的基本概念和原理 14](#_Toc156132227)

[3.2五级流水线的设计内容和细节 15](#_Toc156132228)

[3.2.1原理 15](#_Toc156132229)

[3.2.2 取指阶段 17](#_Toc156132230)

[3.2.3 译码阶段 21](#_Toc156132231)

[3.2.4 执行阶段 23](#_Toc156132232)

[3.2.5 访存阶段 24](#_Toc156132233)

[3.2.6 回写阶段 24](#_Toc156132234)

[4.具体模块设计 24](#_Toc156132235)

[4.1 Instruction Memory 24](#_Toc156132236)

[4.2 Data Memory 25](#_Toc156132237)

[4.3 RegFile 25](#_Toc156132238)

[4.4 ALU 26](#_Toc156132239)

[4.5 Decoder 27](#_Toc156132240)

[4.6 Control 27](#_Toc156132241)

[5.冒险与冲突解决策略 28](#_Toc156132242)

[6.实验调试及结果展示 32](#_Toc156132243)

[6.1测试代码 32](#_Toc156132244)

[6.2仿真结果 33](#_Toc156132245)

[7. 总结 34](#_Toc156132246)

[8. 课程建议 35](#_Toc156132247)

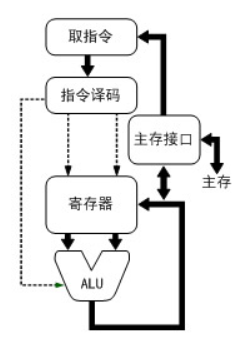
# 1 设计要求

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基于MIPS32或RISC-V或ARM等精简指令集架构的微处理器（CPU）。

要求：完成多周期流水线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，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。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]。

## 1.1 CPU处理指令的过程

冯•诺伊曼型计算机[2]的CPU将指令和数据不加区分放在存储中，指令的处理过程需要访问存储。如图1所示，一条指令的处理通常可以分为5个阶段：取指令、指令译码、执行指令、访存取数和结果写回。



**图1 指令的典型处理过程**

### 指令格式

MIPS32架构中的所有指令都是32位，也就是32个0、1编码连在一起表示一条指令，有三种指令格式。如表格所示：其中op是指令码、func是功能码。

**R类型**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| op | rs | rt | rd | sa | func |

**I类型**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 0 |
| op | rs | rt | immediate |

**J类型**

|  |  |
| --- | --- |
| 31 26 | 25 0 |
| op | Address |

1. R类型:

该类型指令从寄存器堆(Register File)中读取两个源操作数,计算结果写回寄存器堆。具体操作由op,func 结合指定,s和rt 是源寄存器的编号,rd是目的寄存器的编号。例如,假设目的寄存器是 $3,那么对应的rd就是 00011(此处是二进制)。MIPS32架构中有 32个通用寄存器,使用5位编码就可以全部表示,所以rsrtrd 的宽度都是5位sa 只有在移位指令中使用,用来指定移位位数。

1. I类型:

该类型指令使用一个16位立即数作为源操作数.具体操作由op指定，指令的低16位是立即数，运算时要将其扩展至32位，然后作为其中一个源操作数参与运算。

1. J类型:

该类型指令使用一个26位立即数作为跳转的目的地址,具体操作由op指定，一般是跳转指令，低26位是字地址，用于产生跳转的目标地址。

### 指令处理流程概述

指令处理是CPU工作的核心过程，负责执行存储在内存中的程序指令。指令处理流程包括从取指令、解码、执行到访存、回写等阶段的一系列操作。

取指令：从内存中获取下一条要执行的指令。

解码：将取回的指令进行解码，将其转化为可执行的微操作。

执行：根据解码结果，生成控制信号，驱动运算器和内存进行实际操作。

访存：根据需要，访问主存以获取数据或写入结果。

回写：将执行结果写回寄存器或内存。

多周期流水线设计将上述过程分为多个阶段，每个阶段并行执行，从而提高指令处理速度。然而，流水线处理也带来了数据相关和分支冒险等问题，需要进行相应的解决策略。

CPU的核心组件如控制单元、算术逻辑单元和寄存器等在指令处理过程中发挥关键作用。控制单元负责产生控制信号，算术逻辑单元负责执行运算，寄存器用于存储中间结果和数据。

指令执行时间与CPU时钟周期密切相关。优化时钟周期设计可以提高指令处理速度，进而提升CPU性能。

指令处理流程是CPU设计的核心部分，涉及多个阶段和组件的协同工作。理解指令处理流程对于理解CPU工作原理、优化性能和解决潜在问题至关重要。在后续部分中，我们将深入探讨每个阶段的具体实现和技术挑战。

# 2 指令集设计

## 2.1 算数指令设计

### 2.1.1 ADD & ADDU

ADD指令是MIPS32指令集中的加法指令，用于将两个操作数相加并将结果存储在目标寄存器中。在设计ADD指令时，我们需要考虑指令的格式、操作数和目标寄存器。

指令格式：ADD rd, rs, rt

指令功能：将寄存器 rs 和 rt 中的值相加，并将结果存储在寄存器 rd 中。

操作数：ADD指令的操作数可以是立即数、寄存器或内存地址。在本设计中，我们只考虑寄存器操作数，因为寄存器操作数在实际应用中更为常见。

目标寄存器：ADD指令的结果将被存储在目标寄存器 rd 中。在设计指令时，我们需要指定目标寄存器的名称或编号。在本设计中，我们使用一个32位的通用寄存器作为目标寄存器。

实现方式：ADD指令的实现方式是将两个操作数相加，并将结果存储在目标寄存器中。在MIPS32架构中，加法操作可以使用简单的加法指令完成。具体实现过程如下：

1. 取指令: 先将pc的地址送入imem, 然后对imem发出一个读信号(IM\_R), 将指令从imem中取出, 完成取指令后, 将pc送入NPC完成增值.
2. 执行指令: 先将rs和rt内容送入ALU, 完成 + 运算后, 写回rd, RF\_W(regfile写信号有效).
3. 将NPC中的pc完成加4后送回pc.

在设计ADD指令时，我们还需要考虑指令的编码方式和在流水线中的执行阶段。在本设计中，我们将ADD指令编码为单字节指令，并在执行阶段执行加法操作。同时，我们还需要确保ADD指令与其他指令的兼容性和互操作性，以便在多周期流水线中正确地执行。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 000000 | rs | rt | rd | 00000 | 100000 |
| 000000 | rs | rt | rd | 00000 | 100001 |

①当功能码是 6b100000 时，表示 add 指令，加法运算。

指令用法为:addrd,rs,rt。

指令作用为:rd <- rs +rt，将地址为 rs 的通用寄存器的值与地址为 的通用寄存器的值

进行加法运算，结果保存到地址为 rd 的通用寄存器中。但是有一种特殊情况:如果加法运算

溢出，那么会产生溢出异常，同时不保存结果。

②当功能码是6b100001 时，表示addu 指令，加法运算。

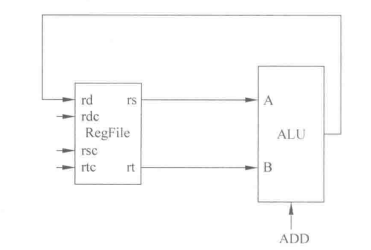
指令用法为:addu rd,rs,rt。

指令作用为: rd < rs +t，将地址为 rs 的通用寄存器的值与地址为 的通用存器的值

进行加法运算，结果保存到地址为 rd 的通用寄存器中与add 指令的不同之处在于addu 指令

不进行溢出检查，总是将结果保存到目的寄存器。

**图2.1 ADDU执行数据通路**



### 2.1.2 SUB & SUBU

SUB指令是MIPS32指令集中的减法指令，用于将第一个操作数减去第二个操作数，并将结果存储在目标寄存器中。在设计SUB指令时，我们需要考虑指令的格式、操作数和目标寄存器。

指令格式：SUB rd, rs, rt

指令功能：将寄存器 rs 中的值减去寄存器 rt 中的值，并将结果存储在寄存器 rd 中。

操作数：SUB指令的操作数可以是立即数、寄存器或内存地址。在本设计中，我们只考虑寄存器操作数，因为寄存器操作数在实际应用中更为常见。

目标寄存器：SUB指令的结果将被存储在目标寄存器 rd 中。在设计指令时，我们需要指定目标寄存器的名称或编号。在本设计中，我们使用一个32位的通用寄存器作为目标寄存器。

实现方式：SUB指令的实现方式是将第一个操作数减去第二个操作数，并将结果存储在目标寄存器中。在MIPS32架构中，减法操作可以使用简单的减法指令完成。具体实现过程如下：

1. 取指令: 先将pc的地址送入imem, 然后对imem发出一个读信号(IM\_R), 将指令从imem中取出, 完成取指令后, 将pc送入NPC完成增值.
2. 执行指令: 先将 rs 和 rt 的内容送入ALU, 完成 - 运算后, 写回 rd, RF\_W(regfile写信号有效).
3. 将NPC中的pc加4后送回pc.

在设计SUB指令时，我们还需要考虑指令的编码方式和在流水线中的执行阶段。在本设计中，我们将SUB指令编码为单字节指令，并在执行阶段执行减法操作。同时，我们还需要确保SUB指令与其他指令的兼容性和互操作性，以便在多周期流水线中正确地执行。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 000000 | rs | rt | rd | 00000 | 100010 |
| 000000 | rs | rt | rd | 00000 | 100011 |

①当功能码是 6b100010 时，表示 sub 指令，减法运算。

指令用法为:sub rd,rs,rt。

指令作用为:rd <- rs t，将地址为 rs 的通用寄存器的值与地址为 的通用寄存器的值

进行减法运算，结果保存到地址为 rd 的通用寄存器中。但是有一种特殊情况:如果减法运算溢出，那么产生溢出异常，同时不保存结果。

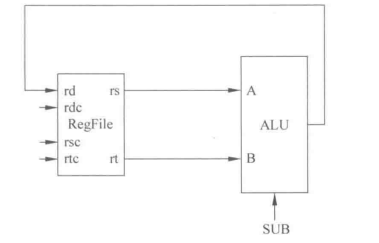
②当功能码是6b100011 时，表示 suu 指令，减法运算

指令用法为:subu rd,rs,rt。

指令作用为:rd < rs t，将地址为 s 的通用寄存器的值与地址为t 的通用寄存器的值

进行减法运算，结果保存到地址为 rd 的通用寄存器中。与 sub 指令的不同之处在于:subu 指令不进行溢出检查，总是将结果保存到目的寄存器。

**图2.2 SUBU执行数据通路**



### ２.１.３SLT & SLTU

SLT指令是MIPS32指令集中的符号位传送指令，用于将一个操作数的符号位复制到目标寄存器的位字段中。在设计SLT指令时，我们需要考虑指令的格式、操作数和目标寄存器。

指令格式：SLT rd, rs, rt

指令功能：将寄存器 rs 中的值的符号位复制到寄存器 rt 中的值的位字段中，并将结果存储在寄存器 rd 中。

操作数：SLT指令的操作数可以是立即数、寄存器或内存地址。在本设计中，我们只考虑寄存器操作数，因为寄存器操作数在实际应用中更为常见。

目标寄存器：SLT指令的结果将被存储在目标寄存器 rd 中。在设计指令时，我们需要指定目标寄存器的名称或编号。在本设计中，我们使用一个32位的通用寄存器作为目标寄存器。

实现方式：SLT指令的实现方式是将一个操作数的符号位复制到目标寄存器的位字段中。在MIPS32架构中，符号位传送操作可以使用SLT指令完成。具体实现过程如下：

判断寄存器 rs 中的值的符号位，并将符号位复制到寄存器 rt 中的值的位字段中。

将结果存储在目标寄存器 rd 中。

在设计SLT指令时，我们还需要考虑指令的编码方式和在流水线中的执行阶段。在本设计中，我们将SLT指令编码为单字节指令，并在执行阶段执行符号位传送操作。同时，我们还需要确保SLT指令与其他指令的兼容性和互操作性，以便在多周期流水线中正确地执行。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 000000 | rs | rt | rd | 00000 | 101010 |
| 000000 | rs | rt | rd | 00000 | 101011 |

①当指令码是 6b001010 时，表示 slti 指令，比较运算。

指令用法为:sltu rt,rs,immediate。

指令作用为:rt <-(rs <(sign extended)immediate)，将指令中的16位立即数进行符号扩展，与地址为rs 的通用寄存器的值按照有符号数进行比较，如果前者大于后者，那么将 1保存到地址为rt 的通用寄存器中;反之，将0保存到地址为t 的通用寄存器中。

②当指令码是 6b001011 时，表示 sltu 指令，比较运算。

指令用法为:sltu rt,rs,immediate。

指令作用为:rt<-(rs <(sign extended)immediate),将指令中的16 位立即数进行符号扩展,与地址为rs 的通用寄存器的值按照无符号数进行比较，如果前者大于后者，那么将1 保存到地址为t 的通用寄存器中:反之，将0保存到地址为 的通用寄存器中。

### 2.1.4 CLO & CLZ

CLO指令是MIPS32指令集中的计数器清零指令，用于将计数器的值清零。在设计CLO指令时，我们需要考虑指令的格式和目标寄存器。

指令格式：CLO rd

指令功能：将计数器的值清零，并将结果存储在目标寄存器 rd 中。

目标寄存器：CLO指令的结果将被存储在目标寄存器 rd 中。在设计指令时，我们需要指定目标寄存器的名称或编号。在本设计中，我们使用一个32位的通用寄存器作为目标寄存器。

实现方式：CLO指令的实现方式是将计数器的值清零，并将结果存储在目标寄存器中。在MIPS32架构中，计数器清零操作可以使用CLO指令完成。具体实现过程如下：

将计数器的值清零。

将结果存储在目标寄存器 rd 中。

在设计CLO指令时，我们还需要考虑指令的编码方式和在流水线中的执行阶段。在本设计中，我们将CLO指令编码为单字节指令，并在执行阶段执行计数器清零操作。同时，我们还需要确保CLO指令与其他指令的兼容性和互操作性，以便在多周期流水线中正确地执行。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 011100 | rs | rt | rd | 00000 | 100000 |
| 011100 | rs | rt | rd | 00000 | 100001 |

①当功能码是 6b100000 时，表示 clz 指令，计数运算。

指令用法为:clz rd,rs。

指令作用为:rd <- coun leading zeros rs，对地址为rs 的通用寄存器的值，从其最高位开始向最低位方向检查，直到遇到值为“1”的位，将该位之前“0”的个数保存到地址为 d的通用寄存器中，如果地址为 rs 的通用寄存器的所有位都为 0(即0x0000000)，那么将 32保存到地址为 rd 的通用寄存器中。

②当功能码是6b100001 时，表示clo 指令，计数运算。

指令用法为:clo rd,rs。

指令作用为:rd <- coun leading ones rs，对地址为 rs 的通用寄存器的值，从其最高位开始向最低位方向检查，直到遇到值为“0”的位，将该位之前“1”的个数保存到地址为 d 的通用寄存器中，如果地址为 rs 的通用寄存器的所有位都为 (即 XFFFFFFFF)，那么将 32保存到地址为rd的通用寄存器中。

## 2.2 逻辑指令设计

### 2.2.1 and & or & xor & nor

指令集中的“与”、“或”、“异或”和“非或”是常见的算术逻辑指令，它们在计算机中用于执行基本的逻辑运算。以下是这四个指令的实现方式：

AND指令：AND指令用于执行逻辑与操作，即当两个操作数都为真时，结果才为真。在大多数处理器中，AND指令通过位与操作实现。它将两个操作数作为输入，然后按位执行与操作。具体来说，如果两个相应的位都为1，则结果位为1，否则为0。

OR指令：OR指令用于执行逻辑或操作，即当至少有一个操作数为真时，结果就为真。在处理器中，OR指令通过位或操作实现。它将两个操作数作为输入，然后按位执行或操作。具体来说，如果两个相应的位至少有一个为1，则结果位为1，否则为0。

XOR指令：XOR指令用于执行逻辑异或操作，即当两个操作数不相等时，结果为真。在处理器中，XOR指令通过位异或操作实现。它将两个操作数作为输入，然后按位执行异或操作。具体来说，如果两个相应的位不同，则结果位为1，否则为0。

NOR指令：NOR指令用于执行逻辑非或操作，即当两个操作数都为假时，结果才为真。在处理器中，NOR指令通常通过先对两个操作数执行逻辑与操作，然后再对结果执行逻辑非操作来实现。这样可以在一个步骤中完成非或操作。

这些指令的实现方式基于二进制数的位运算。通过按位执行与、或、异或和非操作，处理器能够快速高效地执行这些逻辑运算。这些指令在处理器中的实现方式因不同的处理器架构而有所不同，但基本原理是相同的。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 000000 | rs | rt | rd | 00000 | 100100 |
| 000000 | rs | rt | rd | 00000 | 100101 |
| 000000 | rs | rt | rd | 00000 | 100110 |
| 000000 | rs | rt | rd | 00000 | 100111 |

①当功能码是6b100100 时，表示是and 指令，逻辑“与”运算。

指令用法为:andrd,rs,rt。

指令作用为: rd <- rs AND rt,将地址为 rs 的通用存器的值与地址为t 的通用寄存器的值进行逻辑“与”运算，运算结果保存到地址为 rd 的通用寄存器中。

②当功能码是6b100101 时，表示是or 指令，逻辑“或”运算

指令用法为:or rd,rs,rt。

指令作用为:rd <- rs R t，将地址为 rs 的通用存器的值与地址为t 的通用寄存器的值进行逻辑“或”运算，运算结果保存到地址为 rd 的通用寄存器中。

③当功能码是6b100110 时，表示是 xor 指令，异或运算。

指令用法为:xor rd,rs,rt。

指令作用为: rd<-rs XOR t,将地址为 rs 的通用存器的值与地址为 t 的通用寄存器的值进行逻辑“异或”运算，运算结果保存到地址为 rd 的通用寄存器中。

④当功能码是6b100111 时，表示是nor 指令，或非运算。

指令用法为:norrd,rs,rt。

指令作用为: rd<- rs NORt，将地址为s 的通用寄存器的值，与地址为t 的通用寄存器的值进行逻辑“或非”运算，运算结果保存到地址为 rd 的通用寄存器中。

### 2.2.2 andi & xori

ANDI和XORI是MIPS32指令集中的两条指令，它们分别用于执行与（AND）和异或（XOR）运算。

ANDI指令用于将一个立即数与寄存器中的值进行逻辑与运算。具体来说，它将立即数与寄存器中的值进行零扩展，然后执行逻辑与运算，将结果存储在目标寄存器中。例如，如果目标寄存器的值为0x0000FFFF，立即数为0x00000001，则执行ANDI指令后，目标寄存器的值将为0x00000001。

XORI指令用于将一个立即数与寄存器中的值进行异或运算。具体来说，它将立即数与寄存器中的值进行零扩展，然后执行异或运算，将结果存储在目标寄存器中。例如，如果目标寄存器的值为0x0000FFFF，立即数为0x00000001，则执行XORI指令后，目标寄存器的值将为0xFFFFFFF0。

ANDI和XORI指令在MIPS32指令集中是I型指令，它们的操作码部分可以通过特定的位模式来识别。在MIPS32架构中，这些指令被用于执行基本的逻辑运算，可以应用于各种场景，包括整数运算、位操作和数据传输等。

|  |  |  |  |
| --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 0 |
| 001100 | rs | rt | immediate |
| 001110 | rs | rt | immediate |

①当指令码是6b001100，表示是andi 指令，逻辑“与”运算。

指令用法为:andirt,rs,immediate。

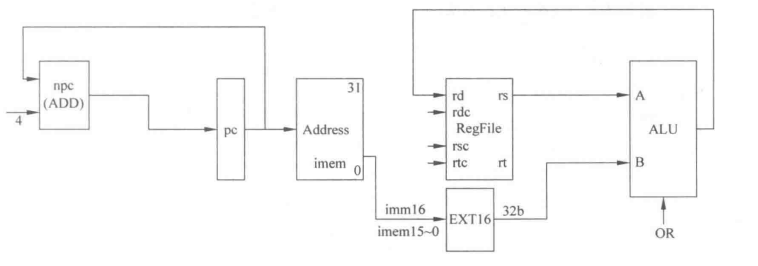
指令作用为:rt<-rsANDzero extended(immediate)，将地址为rs的通用寄存器的值与指令中立即数进行零扩展后的值进行逻辑“与”运算，运算结果保存到地址为 t 的通用寄存器中

②当指令码是6b001110，表示是 xori 指令，异或运算。

指令用法为:xorirt,rs,immediate。

指令作用为:rt<-rsXOR zero extended(immediate)，将地址为rs 的通用寄存器的值与指令中立即数进行零扩展后的值进行逻辑“异或”运算，运算结果保存到地址为 t 的通用寄存器中。

**图2.5 ANDI执行数据通路**



# 3.五级流水线设计

## 3.1流水线的基本概念和原理

流水线技术是一种显著提高指令执行速度与效率的技术。其基本原理是将指令的执行过程划分为多个子过程，每个子过程由一个子部件来完成，每个子部件都在同一时间处理一条指令的一个子过程。子部件之间通过一定的方式连接起来，形成一条指令流水线。当一条指令被取入流水线时，各子部件分别处理该指令的各个子过程，当所有子过程都完成后，该指令即执行完毕并输出结果。

流水线技术可以显著提高指令执行速度和效率，主要原因在于它充分利用了处理器的资源，避免了资源的空闲和等待。同时，流水线技术还可以通过并行处理的方式，同时处理多条指令的各个子过程，进一步提高处理器的吞吐量和效率。

流水线设计是将指令执行过程划分为多个阶段，每个阶段完成特定的任务，如取指、译码、执行、访存和写回等。这些阶段按照一定的顺序排列，形成一个流水线结构。当一条指令经过流水线时，每个阶段都在不同的时钟周期内完成其任务。通过这种方式，流水线设计可以同时处理多条指令，从而提高处理器的吞吐量和效率。

流水线设计的优势包括：

并发性：通过将指令执行过程划分为多个阶段，流水线可以在同一时刻处理多条指令，提高了处理器的并行处理能力。

均衡性：流水线中的各个阶段可以按照固定的时间间隔完成其任务，使得处理器各部分的工作负载更加均衡，减少了资源浪费。

连续性：流水线设计使得指令执行过程更加连续，减少了处理器等待时间，提高了指令的执行效率。

高效性：流水线设计通过重叠执行多个指令，提高了处理器的吞吐量和效率。

流水线技术广泛应用于计算机体系结构中，包括CPU、GPU、DSP等处理器的设计。通过采用流水线技术，处理器可以在保持高吞吐量和效率的同时，降低功耗和成本。

## 3.2五级流水线的设计内容和细节

### 3.2.1原理

五级流水线是指将计算机指令处理过程拆分为多个步骤，并通过多个硬件处理单元并行执行来加快指令执行速度。此处有两个关键词:(1) 拆分:(2)并行。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，如图3-1 所示。

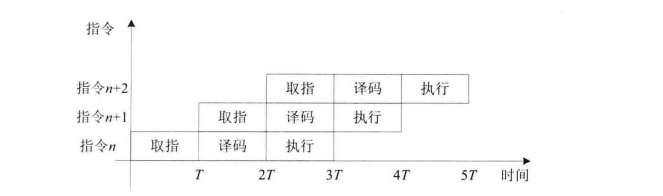


图3-1

从图 3-1 可知，在三级流水线上执行3条指所需时间为 5T，而如果没有使用流水线则需要 9T，流水线确实加快了指令执行。ARM7 采用的就是三级流水线。但世间是没有这么简单完美的，上面假设取指、译码、执行需要的时间都是 T，实际并非如此，比如取指的时间就可能很长，假设取指需要 2T时间，那么如图3-2 所示。

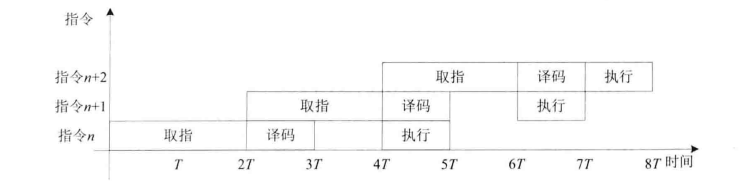


图3-2

可见在 3T~4T的时间段、5T~6T的时间段，流水线在等待取指结束，此时译码阶段执行阶段都停滞，这样一来自然就慢下来，最后，执行 3 条指令所需时间为 8T。解决取指时间过长的措施是引入缓存(Cache)，处理器从缓存读取指令只需要1个时钟周期。还有一种情况是执行阶段时间过长，比如指令为加载/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，此时也会导致流水线停滞。为了解决这种情况下的流水线停滞问题，引入了五级流水线，分别是:取指、译码、执行、访存、回写。如图3-3 所示。

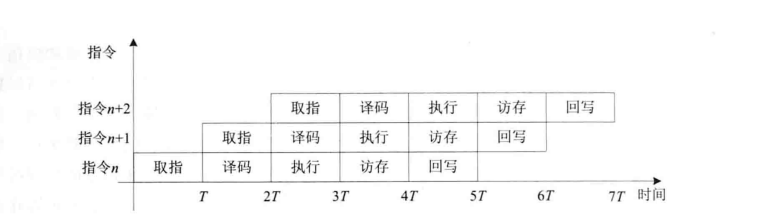


图3-3

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。ARM9就采用了这种五级流水线,OpenMIPS 的设计目标也是五级流水线。具体而言OpenMIPS五级流水线各个阶段的主要工作如下。

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。

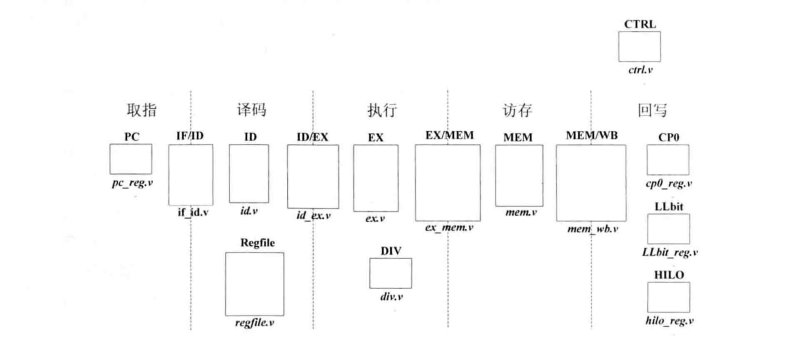


图3-2-1 流水线各个阶段对应模块和文件

### 3.2.2 取指阶段

取指阶段是CPU执行指令过程中的一个重要阶段，主要负责从内存中取出指令并加载到指令寄存器中。在取指阶段，CPU需要完成以下任务：

1.确定要执行的指令的地址。这通常是由程序计数器（PC）提供的，它会指向下一条要执行的指令的内存地址。

2.从内存中读取指令。根据程序计数器的值，CPU从内存中读取指令并存储到指令寄存器中。

3.更新程序计数器的值。在取出指令后，程序计数器通常会增加，指向下一条要执行的指令的地址。

4.解码指令。取出的指令需要被解码，以确定需要执行的操作和操作数。

PC 模块: 给出指令地址，其中实现指令指针寄存器 PC，该寄存器的值就是指令地址，对应pc\_reg.v 文件。

IF/ID 模块:实现取指与译码阶段之间的寄存器，将取指阶段的结果 (取得的指令、指令地址等信息)在下一个时钟传递到译码阶段，对应 if id.v 文件。

module pc\_reg(

// 定义模块的输入

input wire clk, // 时钟信号

input wire rst, // 复位信号

input wire[5:0] stall, // 来自控制模块的信息，表示是否需要停止

// 定义模块的输出

output reg[`InstAddrBus] pc, // 32位程序计数器

output reg ce // 使能信号

);

// 在每个时钟周期的上升沿执行以下操作

always @ (posedge clk) begin

// 如果使能信号被禁用，将程序计数器重置为0

if (ce == `ChipDisable) begin

pc <= 32'h00000000;

// 如果输入信号表示不需要停止，将程序计数器增加4

end else if(stall[0] == `NoStop) begin

pc <= pc + 4'h4;

end

end

// 在每个时钟周期的上升沿执行以下操作

always @ (posedge clk) begin

// 如果复位信号被激活，将使能信号重置为禁用状态

if (rst == `RstEnable) begin

ce <= `ChipDisable;

// 否则，将使能信号设置为启用状态

end else begin

ce <= `ChipEnable;

end

end

endmodule

这个模块的功能是：在每个时钟周期的上升沿，检查使能信号（ce）和输入信号（stall）。如果使能信号被禁用，则将程序计数器（pc）重置为0；如果输入信号表示不需要停止，则将程序计数器增加4。复位信号（rst）用于在每个时钟周期的上升沿重置使能信号（ce）。

module if\_id(

// 定义模块的输入

input wire clk, // 时钟信号

input wire rst, // 复位信号

// 来自控制模块的信息

input wire[5:0] stall, // 表示是否需要停止的5位宽输入信号

// 来自上一个模块的输入

input wire[`InstAddrBus] if\_pc, // 32位程序计数器输入

input wire[`InstBus] if\_inst, // `InstBus`宽的指令输入

// 定义模块的输出

output reg[`InstAddrBus] id\_pc, // 32位程序计数器输出

output reg[`InstBus] id\_inst // `InstBus`宽的指令输出

);

// 在每个时钟周期的上升沿执行以下操作

always @ (posedge clk) begin

// 如果复位信号被激活

if (rst == `RstEnable) begin

// 将程序计数器输出重置为0

id\_pc <= `ZeroWord;

// 将指令输出重置为0

id\_inst <= `ZeroWord;

end else if(stall[1] == `Stop && stall[2] == `NoStop) begin

// 如果停止和无停止标志同时被激活

id\_pc <= `ZeroWord;

id\_inst <= `ZeroWord;

end else if(stall[1] == `NoStop) begin

// 如果仅无停止标志被激活

id\_pc <= if\_pc; // 将程序计数器输出设置为输入程序计数器值

id\_inst <= if\_inst; // 将指令输出设置为输入指令值

end

end

endmodule

这个模块的功能是：在每个时钟周期的上升沿，检查复位信号、输入信号（stall）以及来自上一个模块的输入程序计数器和指令。如果复位信号被激活，或者输入信号表示需要停止，则将输出程序计数器和指令重置为0。否则，将输出程序计数器和指令设置为上一个模块的输入程序计数器和指令。

### 3.2.3 译码阶段

译码阶段是CPU执行指令过程中的一个重要阶段，主要负责对指令进行解码，以确定需要执行的操作和操作数。在译码阶段，CPU需要完成以下任务：

1.解码指令：根据指令寄存器中的指令操作码和其他信息，将其解码为相应的操作和操作数。

2.确定操作数：根据指令的操作码和其他信息，确定需要执行的操作数。这些操作数可能是寄存器、内存地址或立即数等。

3.更新状态寄存器：根据指令的操作和操作数，更新CPU的状态寄存器中的相应位。这包括标志位、状态位等。

4.发送控制信号：根据解码后的指令和操作数，向控制单元发送相应的控制信号，以控制CPU的执行流程。

ID 模块:对指令进行译码，译码结果包括运算类型、运算所需的源操作数、要写入的目的寄存器地址等，对应id.v 文件。

Regfile 模块:实现了32个32位通用整数存器，可以同时进行两个寄存器的读操作和一个寄存器的写操作，对应regfile.v 文件。

ID/EX 模块:实现译码与执行阶段之间的寄存器，将译码阶段的结果在下一个时钟周期传递到执行阶段，对应id ex.v 文件。

module id\_ex(

// 时钟信号输入

input wire clk,

// 复位信号输入

input wire rst,

// 来自控制模块的信息

input wire[5:0] stall,

// 来自译码阶段的信息

input wire[`AluOpBus] id\_aluop, // ALU操作输入

input wire[`AluSelBus] id\_alusel, // ALU选择输入

input wire[`RegBus] id\_reg1, // 第一个寄存器操作数输入

input wire[`RegBus] id\_reg2, // 第二个寄存器操作数输入

input wire[`RegAddrBus] id\_wd, // 写地址输入

input wire id\_wreg, // 写使能输入

// 传递到执行阶段的信息

output reg[`AluOpBus] ex\_aluop, // ALU操作输出

output reg[`AluSelBus] ex\_alusel, // ALU选择输出

output reg[`RegBus] ex\_reg1, // 第一个寄存器操作数输出

output reg[`RegBus] ex\_reg2, // 第二个寄存器操作数输出

output reg[`RegAddrBus] ex\_wd, // 写地址输出

output reg ex\_wreg // 写使能输出

);

// 在每个时钟周期的上升沿执行以下操作

always @ (posedge clk) begin

// 如果复位信号被激活

if (rst == `RstEnable) begin

// 将ALU操作、ALU选择、寄存器操作数、写地址和写使能重置为相应的NOP值

ex\_aluop <= `EXE\_NOP\_OP;

ex\_alusel <= `EXE\_RES\_NOP;

ex\_reg1 <= `ZeroWord;

ex\_reg2 <= `ZeroWord;

ex\_wd <= `NOPRegAddr;

ex\_wreg <= `WriteDisable;

end else if(stall[2] == `Stop && stall[3] == `NoStop) begin

// 如果停止和无停止标志同时被激活，执行与复位相同的操作

ex\_aluop <= `EXE\_NOP\_OP;

ex\_alusel <= `EXE\_RES\_NOP;

ex\_reg1 <= `ZeroWord;

ex\_reg2 <= `ZeroWord;

ex\_wd <= `NOPRegAddr;

ex\_wreg <= `WriteDisable;

end else if(stall[2] == `NoStop) begin

// 如果仅无停止标志被激活，将ALU操作、ALU选择、寄存器操作数、写地址和写使能设置为译码阶段的相应值

ex\_aluop <= id\_aluop;

ex\_alusel <= id\_alusel;

ex\_reg1 <= id\_reg1;

ex\_reg2 <= id\_reg2;

ex\_wd <= id\_wd;

ex\_wreg <= id\_wreg;

end

end

endmodule

注：if (rst == RstEnable) begin`: 如果复位信号被激活，执行以下操作：

将ALU操作设置为NOP操作。

将ALU选择设置为NOP选择。

将第一个和第二个寄存器操作数重置为0。

将写地址设置为NOP寄存器地址。

将写使能设置为禁用状态。

### 3.2.4 执行阶段

EX模块:依据译码阶段的结果，进行指定的运算，给出运算结果。对应 ex.v 文件

DIV模块:进行除法运算的模块，对应 div.v 文件。

EX/MEM 模块:实现执行与访存阶段之间的寄存器，将执行阶段的结果在下一个时钟周期传递到访存阶段，对应ex mem.v 文件。

### 3.2.5 访存阶段

MEM 模块:如果是加载、存储指令，那么会对数据存储器进行访问。此外，还会在该模块进行异常判断。对应mem.v 文件。

MEM/WB 模块:实现访存与回写阶段之间的寄存器，将访存阶段的结果在下一个时钟周期传递到回写阶段，对应mem wb.v 文件。

### 3.2.6 回写阶段

CPO模块:对应MIPS架构中的协处理器CPO。

LLbit 模块:实现寄存器 LLbit，在链接加载指令11、条件存储指令 sc 的处理过程中会使用到该寄存器。

HILO 模块:实现寄存器 HI、LO，在乘法、除法指令的处理过程中会使用到这两个寄存器。

# 4.具体模块设计

## 4.1 Instruction Memory

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| addr | 32 | 输入 | 指令地址输入端口 |
| IM\_R | 1 | 输入 | 读指令信号输入端口 |
| Rd | 32 | 输出 | 存储器数据输出端口 |

表4.1 Instruction Memory模块的接口描述

addr是指令地址的输入端口，指令存储器根据输入PC的值取出相应的32位的指令，并输出，同时吧PC加4。

## 4.2 Data Memory

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| DM\_W | 1 | 输入 | 写使能信号 |
| CS | 1 | 输入 | 片选信号 |
| Clk | 1 | 输入 | 时钟信号 |
| rst | 1 | 输入 | 复位信号 |
| DM\_R | 1 | 输入 | 读使能信号 |
| Rdata | 32 | 输出 | 读数据 |
| addr | 32 | 输出 | 地址 |
| Wdata | 32 | 输出 | 写数据 |

表4.2 Data Memory模块的接口描述

Data Memory是与存储器相关的模块。在进行读操作时WR信号置0，addr端口输入需要访问的地址，由Data Out端口输出访问的数据；在进行写操作时WR信号置1，数据由Data In端口读入并写入到addr端口输入的地址中。

## 4.3 RegFile

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| we | 1 | 输入 | 写使能信号 |
| Clk | 1 | 输入 | 时钟信号 |
| Rst | 1 | 输入 | 复位信号 |
| Raddr1 | 5 | 输入 | 读入的第一个寄存器编号 |
| Raddr2 | 5 | 输入 | 读入的第二个寄存器编号 |
| waddr | 5 | 输入 | 写入的寄存器编号 |
| wdata | 32 | 输入 | 写入的数据 |
| Rdata1 | 32 | 输出 | 从第一个寄存器中读出的数据 |
| Rdata2 | 32 | 输出 | 从第二个寄存器中读出的数据 |

表4-3 Regfile模块的接口描述

Regfile模块是寄存器相关模块，实现寄存器的读写功能。因为共有32个寄存器，需要5位来对寄存器编号进行编码，故有关编号的输入都是5位的。其中，Read Reg1、Read Reg2寄存器的信息分别由Read Data1、Read Data2端口输出，因有些指令会利用到立即数，故Read Data2端口与立即数通过控制信号进行二选一的选择。RW、RegWr、Write Data是与写有关的端口，RegWr控制是否进行写操作；RegWr端口的输入是一个二选一选择器的输出，该二选一选择器是通过控制信号判断rt为需要输入的寄存器编号还是rs为需要输入的寄存器编号。

## 4.4 ALU

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| ALUC | 2 | 输入 | 控制ALU中进行的操作 |
| A | 32 | 输入 | 操作数1 |
| B | 32 | 输入 | 操作数2 |
| result | 32 | 输出 | 运算结果 |
| zero | 32 | 输出 | 运算结果 |

表4-4 ALU模块的接口描述

ALU模块是运算相关的模块，由ALUOp控制信号控制两个操作数进行的数据操作，result与zero均为答案输出，只是继续传递的方向不同，zero是向Instruction模块传递，主要是为在beq指令时根据运算结果控制PC的变化情况，ALU功能运算表如下：

|  |  |  |
| --- | --- | --- |
| ALUOp[2..0] | 功能 | 描述 |
| 000 | Y=A+B | 加 |
| 001 | Y=A-B | 减 |
| 010 | Y=(A-B)?1:0 | 比较A与B不带符号 |
| 011 | Y=(((ra<rb)&&(ra[31]==rb[31]))||((ra[31)==1&&rb[31]==0)))?1:0 | 比较A与B带符号 |
| 100 | Y=B<<A | B左移A位 |
| 101 | Y = A ∨ B | 或 |
| 110 | Y = A ∧ B | 与 |
| 111 | Y = A ⊕ B | 异或 |

**表4.4.1 ALU功能控制表**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ALUC2 | ALUC1 | ALUC0 | 备注 |
| ADDU | 0 | 0 | 0 | ALU完成+ |
| SUBU | 0 | 0 | 1 | ALU完成- |
| ORI | 0 | 1 | 0 | ALU完成| |
| ANDI | 0 | 1 | 1 | ALU完成& |
| SLL | 1 | 0 | 0 | ALU完成<< |

## 4.5 Decoder

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| Clk | 1 | 输入 | 时钟信号 |
| zero | 1 | 输入 | 运算结果 |
| Instr\_index | 32 | 输入 | 指令码 |
| result | 32 | 输出 | 运算结果 |
| zero | 32 | 输出 | 运算结果 |
| IM\_R | 1 | 输出 | 读指令信号 |
| M1 | 1 | 输出 | 选择器控制端 |
| M2 | 1 | 输出 | 选择器控制端 |
| M3 | 1 | 输出 | 选择器控制端 |
| M4 | 1 | 输出 | 选择器控制端 |
| M5 | 1 | 输出 | 选择器控制端 |
| M6 | 1 | 输出 | 选择器控制端 |
| DM\_R | 1 | 输出 | 数据存储器读信号 |
| DM\_W | 1 | 输出 | 数据存储器写信号 |
| CS | 1 | 输出 | 片选信号 |
| Sign\_ext | 1 | 输出 | 数据扩展信号 |
| RF\_W | 1 | 输出 | 寄存器写信号 |

表4-5 Decoder模块的接口描述

## 4.6 Control

|  |  |  |
| --- | --- | --- |
| **控制信号** | 状态“0” | 状态“1” |
| RST | 对于 PC，初始化 PC 为程序首地址 | 对于 PC，PC 接收下一条指令地址 |
| PCWre | PC 不更改，相关指令：halt | PC 更改，相关指令：除指令 halt 外 |
| ALUSrc | 来自寄存器堆 data 输出，相关指令：add、sub、and、slt、sll | 来自 sign 或 zero 扩展的立即数，相关指令：add、ani、ori、beq lw、sw |
| IRWre | IR(指令寄存器)不更改 | IR 寄存器写使能 |
| ExtSel | (zero-extend)immediate，相关指令：andi、xori、ori | (sign-extend)immediate，相关指令： addiu、slti、lw、sw、beq |
| WrRegData | 写入寄存器组寄存器的数据来自pc+4(pc4），相关指令：jal | 写入寄存器组寄存器的数据来自 ALU 运算结果或存储器读出的数据 |
| InsMemRW | 写指令存储器 | 读指令存储器 |
| RegOut[1..0] | 写寄存器组寄存器的地址 |  |

表4-6 control模块控制信号与指令对应关系

control模块是整个CPU运行的核心，表格中列出了控制信号与相关指令的对应关系。CU通过改变控制信号来控制状态转移，每个状态代表一个时钟周期，同时需要D触发器保存当前状态。

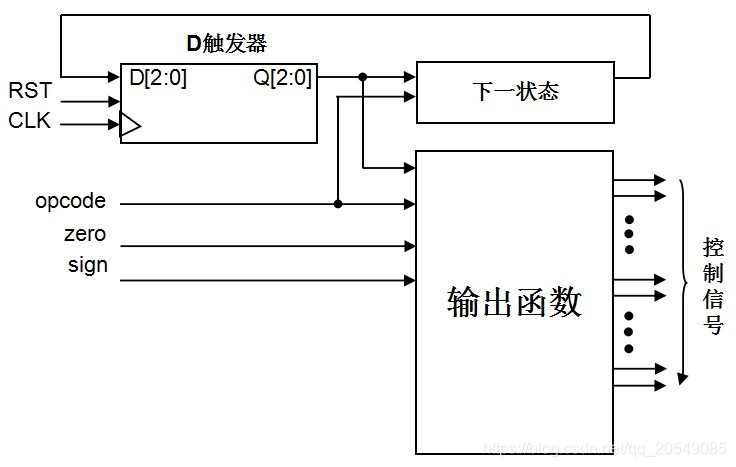


图4-6-1 D触发器电路图

三个 D 触发器用于保存当前状态，是时序逻辑电路，RST 用于初始化状态“000“，另外两个部分都是组合逻辑电路，一个用于产生下一个阶段的状态，另一个用于产生每个阶段的控制信号。从图上可看出，下个状态取决于指令操作码和当前状态；而每个阶段的控制信号取决于指令操作码、当前状态和反映运算结果的状态 zero 标志和符号 sign 标志。

# 5.冒险与冲突解决策略

流水线数据相关问题是指在流水线中执行的多条指令之间存在对某些寄存器或内存单元的读写依赖关系，导致指令的执行顺序和结果与预期不符的问题。流水线数据相关问题主要有以下三种类型：

RAW（Read After Write）：写后读相关，指一条指令需要读取前面一条或多条指令写入的寄存器或内存单元的值，但是前面的指令还没有完成写操作，导致读取的值不正确。例如：

add $1, $2, $3 # $1 <- $2 + $3

sub $4, $1, $5 # $4 <- $1 - $5

第二条指令需要读取第一条指令写入的$1的值，但是第一条指令还没有完成写回阶段，所以第二条指令读取的$1的值是错误的。

WAR（Write After Read）：读后写相关，指一条指令需要写入前面一条或多条指令读取的寄存器或内存单元的值，但是前面的指令还没有完成读操作，导致写入的值覆盖了前面的指令需要的值。例如：

lw $1, 0($2) # $1 <- M[$2]

sw $3, 0($2) # M[$2] <- $3

第二条指令需要写入第一条指令读取的内存单元M[$2]的值，但是第一条指令还没有完成访存阶段，所以第二条指令写入的值覆盖了第一条指令需要的值。

WAW（Write After Write）：写后写相关，指一条指令需要写入前面一条或多条指令写入的寄存器或内存单元的值，但是前面的指令还没有完成写操作，导致写入的值不是最新的值。例如：

add $1, $2, $3 # $1 <- $2 + $3

sub $1, $4, $5 # $1 <- $4 - $5

第二条指令需要写入第一条指令写入的$1的值，但是第一条指令还没有完成写回阶段，所以第二条指令写入的值不是最新的值。

但是所有指令在流水线回写阶段才会写寄存器，因此没有WRW问题；又因为只能在译码阶段读寄存器，回写阶段写寄存器，不存在WAR问题，所以只需要考虑RAW问题。

相邻指令间存在数据相关

l ori $1,$0,0x1100 #$1 =$010x1100= 0x1100

2 ori $2,$1,0x0020 #$2 =$1| 0x0020 = 0x1120

第1条ori指令将写寄存器$1,随后的第2条ori指令需要读出$1的数据，但是第1条ori指令在回写阶段才会将其运算结果写入$1,而第2条ori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。如图所示，这种情况可以称为相邻指令间存在数据相关.

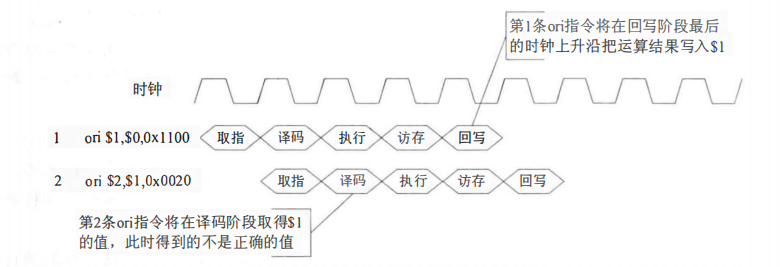


图 5-1 相邻指令间存在数据相关.

相隔1条指令的指令间存在数据相关

考虑如下代码。

l ori $1,$0,0x1100 #$1=$0I 0x1100 = 0x1100

2 ori $3,$0,0xffff #$3 =$0| 0xffff = 0xffff

3 ori $2,$1,0x0020 #$2 =$1I 0x0020= 0x1120

第1条ori指令将写寄存器$1,第3条ori指令在译码阶段需要读取寄存器S1,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。如图所示，这种情况可以称为相隔1条指令的指令间存在数据相关。

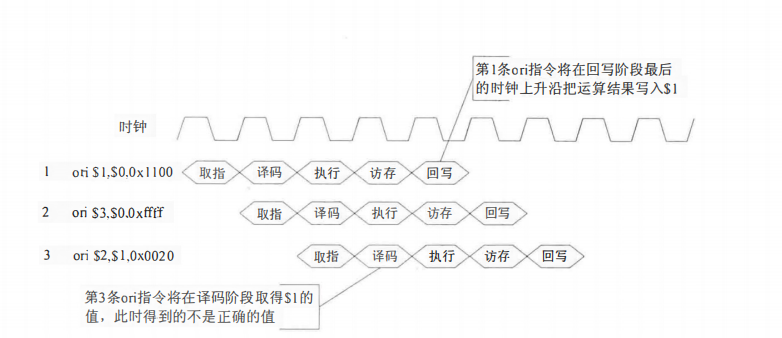


图 5-2 相隔1条指令的指令间存在数据相关

相隔2条指令的指令间存在数据相关

l ori $1,$0,0x1100 #$1 =$010x1100 = 0x1100

2 ori $3,$0,0xffff #$3 =$0I 0xffff = 0xffff

3 ori $4,$0,0xffff #$4 =$0I 0xffff = 0xffff

4 ori $2,$1,0x0020 #$2 =$1|0x0020= 0x1120

第1条ori指令将写寄存器$1,第4条ori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ori指令得到的不是正确的寄存器S1的值。如图所示，这种情况可以称为相隔2条指令的指令间存在数据相关。

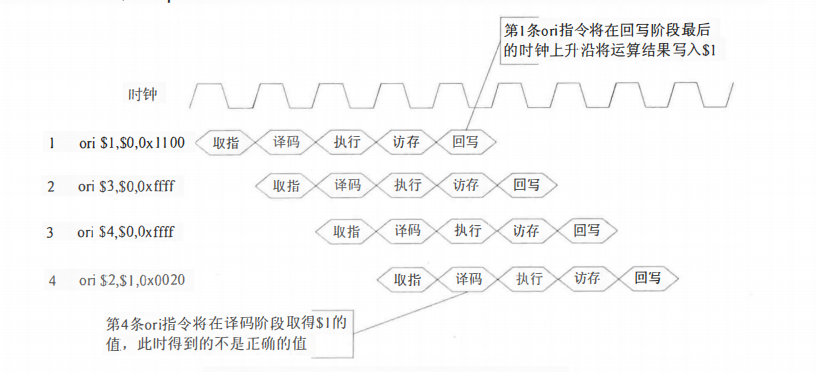


图 5-3 相隔2条指令的指令间存在数据相关

流水线数据相关问题的具体解决思路有以下几种：

数据前推（Data Forwarding）：即在数据计算完成后，将数据直接传递给需要使用它的指令，而不用等到写入寄存器或内存后再读取。它的基本思路是将流水线中后面的指令所需的数据直接从前面的指令中获取，而不用等待前面的指令写回寄存器。这样可以避免因为数据相关而导致的流水线暂停，提高流水线的效率。

举个例子，假设有以下两条指令：

add $t0, $s0, $s1 # I1: $t0 = $s0 + $s1

sub $t1, $t0, $s2 # I2: $t1 = $t0 - $s2

如果按照正常的流水线执行，I2需要等待I1的结果写回寄存器才能执行，这样就会产生数据冲突。但是如果使用数据前推，I2可以直接从I1的执行阶段获取$t0的值，而不用等待I1的写回阶段，这样就可以避免数据冲突。

暂停（Stalling）：即在检测到数据相关问题时，暂停流水线的运行，等待数据产生后再继续执行。这种方法可以解决所有的数据相关问题，但是会降低流水线的性能，因为暂停期间流水线不能接受新的指令。

重排序（Reordering）：即在编译或运行时，对指令序列进行重新排列，使得相关的指令之间有足够的间隔，避免数据相关问题的发生。这种方法可以提高流水线的性能，但是需要保证指令重排序不会改变程序的语义，也不能解决所有的数据相关问题，因为有些指令之间的依赖关系是必须的。

# 6.实验调试及结果展示

当复位信号关闭时，流水线开始运作。在每个时钟上升沿，流水线从pc\_reg文件中读取地址，然后进入inst\_rom指令存储器，从中取出相应的指令。取出的指令进入if\_id文件，实现取指与译码阶段的转换，然后根据指令的操作码和功能码进行译码。译码完成后，将源寄存器、目的寄存器等信息传递到id\_ex文件，实现译码与执行阶段的转换。在执行阶段，根据不同指令的格式进行相应的运算，并生成结果。如果指令是加载或存储指令，还会与数据存储器进行交互。执行完成后，将结果传递到ex\_mem文件，实现执行与访存阶段的转换。访存完成后，将结果传递到mem文件，如果是加载、存储指令，会对数据存储器进行访问。最后，将访存结果传递到mem\_wb文件，实现访存与回写阶段的转换，等待在下一个时钟上升沿被写入目标寄存器。至此，一条完整的指令就被处理完毕并执行完毕。

## 6.1测试代码

inst\_mem[0] = 32'h34021100;

inst\_mem[1] = 32'h34420020;

inst\_mem[2] = 32'h34424400;

inst\_mem[3] = 32'h34420044;//数据前推解决数据冲突测试

inst\_mem[4] = 32'h3402000f;//R2 被赋值为15

//inst\_mem[7] = 32'h3442fff1;//R2 被赋值为-15

inst\_mem[5] = 32'h34030010;//R3 被赋值为16

//乘法指令，乘累加指令

inst\_mem[6] = 32'h00620018; //乘法 结果240

inst\_mem[7] = 32'h70620000; //乘累加 结果 480

inst\_mem[8] = 32'h34020000;

inst\_mem[9] = 32'h3402ffff;//R2,R3賦值

inst\_mem[10] = 32'h00021400;

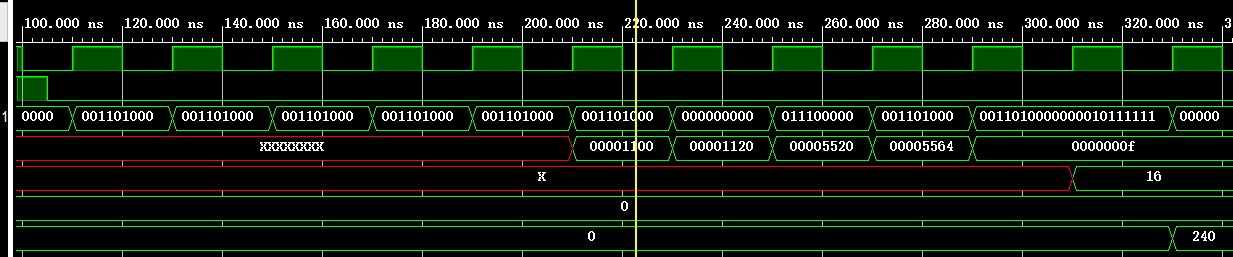
inst\_mem[11] = 32'h3442fff1;//R2 被赋值为-15

inst\_mem[12] = 32'h0043001a;//3 种除法

inst\_mem[13] = 32'h0043001b;

inst\_mem[14] = 32'h0062001a;

## 6.2仿真结果



**图6.2 数据冲突测试的仿真结果图**

ori $1,$0,0x1100 #$1=$O 0x1100=0x1100  
ori $1,$1,0x0020 #$1=$1 0x0020=0x1120  
ori $1,$1,0x4400 #$1=$1 0x4400=0x5520  
ori $1,$1,0x0044 #$1=$1 0x0044=0x5564

以上四条指令，分别涉及相邻指令的数据相关，隔一条指令的数据相关，以及隔两条指令的数据相关，用来测试是否真正处理好数据相关。

从开始取码，经过四个周期，得到该指令的结果。

测试结果与预计结果相同，说明成功处理好数据相关。

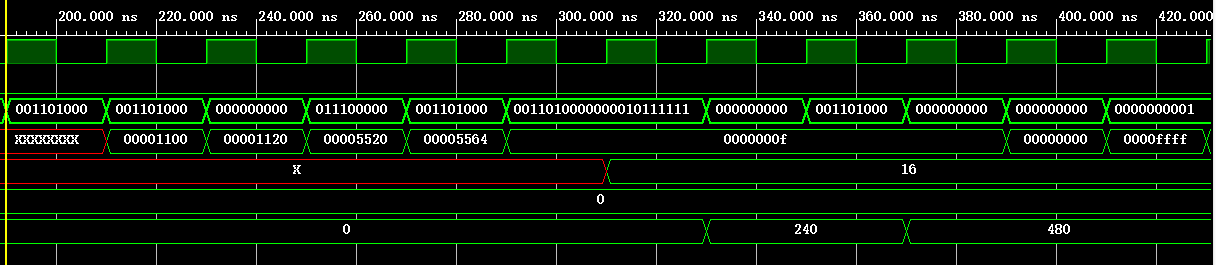


图6.3 乘法，乘累加测试

将R2赋值15，R3赋值16,采用乘法指令和乘累加指令

15\*16=240

240+240=480

从开始取码，经过五个周期（由于乘法，乘累加流水线要暂停一个周期），得到该指令的结果。

测试结果符合预计结果



图 6.4 除法测试

给寄存器$2赋初值-15,寄存器$3 赋初值17,然后分别使用 div 、divu 、div 指令进行运算，结果保存在 HI 、LO 寄存器

测试结果符合预计结果

# 7. 总结

通过这次CPU的课程设计，我得以一窥CPU设计的深奥之处，并对多周期CPU和单周期CPU有了更加清晰的认识。这让我想起了在计算机组成原理课程中学习的知识，那些曾经抽象的概念现在变得栩栩如生。我深入地理解了CPU执行的各个阶段，从取指令、指令译码、指令执行，到存储器访问和结果写回，每一个步骤都让我深感震撼。

更让我印象深刻的是多周期CPU中每条指令执行的各个状态间的转移。控制器发出的不同控制信号，使CPU能够按照取出指令的含义执行，这其中的复杂性和精密性让我对计算机的工作原理有了更加深入的认识。

同时，我也感受到了流水线设计在实际应用中的重要性和优势。通过合理地划分指令执行阶段，我们可以并行处理多个指令，大大提高了处理器的吞吐量和效率。这种设计理念不仅增强了系统的性能，还为解决冒险和冲突提供了有效的策略，从而提高了系统的稳定性和可靠性。

在实践过程中，我学会了使用Vivado软件，以及如何在仿真图中观测各个寄存器变量的值。我也开始接触并学习Verilog硬件描述语言，虽然它与我们之前所学的编程语言有所不同，但其中的逻辑和结构却有着异曲同工之妙。这个过程增强了我对硬件描述语言的兴趣，并提高了我的动手实践能力。

最让我激动的是最后通过Vivado连接开发板，通过编写约束文件将程序中的寄存器变量与开发板的某些端口对应起来。当我看到LED灯按照Verilog程序的设定闪烁时，那一刻我深刻地感受到了硬件的魅力。

整个设计过程历时一周，虽然紧张且充满挑战，但每当我解决一个问题或实现一个功能时，那种成就感都让我觉得所有的努力都是值得的。这次课程设计不仅让我学到了很多知识，也让我对多周期CPU有了更加深入的理解。

我希望未来的CPU课程设计能够继续发展壮大，为更多的学生提供学习和实践的机会。同时，我也期待自己能够在这一领域继续深入探索，不断追求卓越。

# 8. 课程建议

对于本课程，我有以下几点建议：

①加强理论与实践的结合：硬件课程往往涉及大量的理论，但理论只有在实际应用中才能发挥其真正的价值。因此，建议在课堂上引入更多的实际项目和案例，让学生在实践中理解和掌握硬件设计的原理。

②增加现代硬件设计的介绍：随着技术的快速发展，硬件设计的方法和工具也在不断演进。建议课程中增加对现代硬件设计方法（如高级综合、硬件优化工具等）的介绍，使学生能够紧跟时代的步伐。

③强化对硬件描述语言的学习：硬件描述语言是硬件设计的核心工具。为了使学生更好地掌握这一技能，建议加强对Verilog和VHDL等硬件描述语言的实践教学，包括编写、仿真和综合等环节。

④注重培养学生的创新思维：硬件设计不仅仅是实现功能，更重要的是设计思维。建议在课堂上鼓励学生提出自己的想法和解决方案，培养他们的创新思维和批判性思维。

⑤关注最新的技术动态：硬件设计是一个快速发展的领域，新的技术和产品不断涌现。为了使学生能够跟上时代的步伐，建议在教学中关注最新的技术动态，并邀请业界专家进行分享和交流。

|  |
| --- |
| 一、原创性声明 |
| 本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。  特此声明！  **作者签字: 嵌入签名图片** |
| |  |  |  |  | | --- | --- | --- | --- | |  | 评价内容 | 权重 | 得分 | | **验收** |  | 0.4 |  | | **设计报告** |  | 0.6 |  | | 合计 |  | | | | 指导教师（签章）： 2024 年 1 月 日 | | | | |