

**2023~2024学年 第一学期**

**《系 统 硬 件 综 合 设 计》**

设 计 报 告

班 级 计算机科学与技术21-1班

学 号 \_\_ 2021214710\_ \_\_ \_

姓 名 \_\_\_ \_\_\_\_ \_杨程锦 \_

2023年 1 月

**目 录**

[1 设计准备 4](#_Toc155668304)

[1.1 设计要求 4](#_Toc155668305)

[1.2 RISC-V指令格式 4](#_Toc155668306)

[2 设计过程 7](#_Toc155668307)

[2.1 数据通路设计 7](#_Toc155668308)

[2.2 模块设计 9](#_Toc155668309)

[2.2.1 instr\_memory 9](#_Toc155668310)

[2.2.2 data\_memory 9](#_Toc155668311)

[2.2.3 main\_control 10](#_Toc155668312)

[2.2.4 alu\_control 11](#_Toc155668313)

[2.2.5 pc\_reg 11](#_Toc155668314)

[2.2.6 instr\_decode 12](#_Toc155668315)

[2.2.7 register 13](#_Toc155668316)

[2.2.8 alu 14](#_Toc155668317)

[2.2.9 branch\_judge 15](#_Toc155668318)

[2.2.10 Shifter 16](#_Toc155668319)

[2.2.11 Adder 17](#_Toc155668320)

[2.2.12 cla\_adder32 18](#_Toc155668321)

[2.2.13 cla\_4 19](#_Toc155668322)

[2.2.14 forward\_unit 20](#_Toc155668323)

[2.2.15 mux 21](#_Toc155668324)

[2.2.16 mux3\_1 21](#_Toc155668325)

[2.3 五级流水线设计 23](#_Toc155668326)

[2.3.1 if\_stage 23](#_Toc155668327)

[2.3.2 if\_id\_regs 24](#_Toc155668328)

[2.3.3 id\_stage 24](#_Toc155668329)

[2.3.4 id\_ex\_regs 25](#_Toc155668330)

[2.3.5 ex\_stage 26](#_Toc155668331)

[2.3.6 ex\_mem\_regs 28](#_Toc155668332)

[2.3.7 mem\_stage 28](#_Toc155668333)

[2.3.8 mem\_wb\_regs 29](#_Toc155668334)

[2.3.9 wb\_stage 29](#_Toc155668335)

[3 相关与冒险 30](#_Toc155668336)

[3.1 结构冒险 30](#_Toc155668337)

[3.2 数据冒险 30](#_Toc155668338)

[3.2.1 寄存器堆前推 31](#_Toc155668339)

[3.2.2 数据前推 32](#_Toc155668340)

[3.2.3 停顿流水线 33](#_Toc155668341)

[3.3 控制冒险 35](#_Toc155668342)

[4 仿真测试 36](#_Toc155668343)

[4.1.1 数据前推测试 36](#_Toc155668344)

[4.1.2 停顿流水线测试 38](#_Toc155668345)

[4.1.3 冲刷流水线测试 39](#_Toc155668346)

[5 总结 40](#_Toc155668347)

[6 参考文献 41](#_Toc155668348)

1. 设计准备
   1. 设计要求

要求：基于MIPS32或RISC-V或ARM等精简指令集架构完成一个多周期流水线CPU的设计，所设计的各类指令条数不少于20条，对于指令执行时可能产生的冒险与冲突，能够采取各种相应的方法合理解决，在EDA软件上可以运行自己设计的测试程序并仿真验证所有设计的指令。

实现：基于RISC-V精简指令集架构的多周期流水线CPU的设计，所设计指令条数为37条，对于指令执行时可能产生的冒险与冲突，采取相应的方法合理解决，设计了32位全加器以及移位器用于ALU计算，在EDA软件上可以运行自己设计的测试程序并仿真验证所有设计的指令。

具体指令如图 1所示。

表格

描述已自动生成

图 1 RV32I基本指令集

* 1. RISC-V指令格式

本次课程设计采用RISC-V指令集架构。采用RISC-V是它对指令集功能做了良好的分割，以此实现更好的兼容性和扩展性， 并且具有简单的Load/Store结构，易于流水线的实现；指令长度固定，解码方便。

RISC-V指令共有6种类型，分别为：R-type; I-type; S-type; B-type; U-type; J-type。

**R-type：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31：25 | 24：20 | 19：15 | 14：12 | 11：7 | 6：0 |
| funct7 | rs2 | rs1 | funct3 | rd | opcode |

表 1 R-type指令格式

如表 1所示，R-type指令[31:25]位为fucnt7，表示7位功能码；[24:20]位为rs2、[19:15]位为rs1，分别表示两个5位源寄存器；[14:12]位为funct3，表示3位功能码；[11:7]位为rd，表示5位目标寄存器；[6:0]位为opcode，表示7位指令操作码。

**I-type：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31:20 | 19:15 | 14:12 | 11:7 | 6:0 |
| imm[11:0] | rs1 | funct3 | rd | opcode |

表 2 I-type指令格式

如表 2所示，I-type指令[31:20]位为imm[11:0],表示12位立即数；[19:15]位为rs1，表示5位源寄存器；[14:12]位为funct3，表示3位功能码；[11:7]位为rd，表示5位目标寄存器；[6:0]位为opcode，表示7位指令操作码。

**S-type：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31:25 | 24:20 | 19:15 | 14:12 | 11:7 | 6:0 |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode |

表 3 S-type指令格式

如表 3所示，S-type指令[31:25]位为imm[11:5],表示12位立即数的前7位；[24:20]位为rs2，[19:15]位为rs1，分别表示5位源寄存器；[14:12]位为funct3，表示3位功能码；[11:7]位为imm[4:0]，表示12位立即数的后5位；[6:0]位为opcode，表示7位指令操作码。

**B-type：**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30:25 | 24:20 | 19:15 | 14:12 | 11:8 | 7 | 6:0 |
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode |

表 4 B-type指令格式

如表 4所示，B-type指令31位为imm[12]，表示立即数的第31位；[30:25]位为imm[10:5],表示12位立即数的第5到10位；[24:20]位为rs2，[19:15]位为rs1，分别表示5位源寄存器；[14:12]位为funct3，表示3位功能码；[11:8]位为imm[4:1]，表示12位立即数的第1到4位；11位为imm[11]，表示12位立即数的第11位；[6:0]位为opcode，表示7位指令操作码。

**U-type：**

|  |  |  |
| --- | --- | --- |
| [31:12] | [11:7] | [6:0] |
| imm[31:12] | rd | opcode |

表 5 U-type指令格式

如表 5所示，U-type指令[31:12]位为imm[31:12]，表示存放高20位的立即数；[11:7]位为rd，表示5位目标寄存器；[6:0]位为opcode，表示7位指令操作码。

**J-type：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | [30:21] | 20 | [19:12] | [11:7] | [6:0] |
| imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode |

表 6 J-type jal指令格式

如表 6所示，jal指令31位为imm[20]，表示立即数的第20位；[30:21]位为imm[10:1]，表示立即数的第1到10位；20位为imm[11]，表示立即数的第11位；[19:12]位为imm[19:12]，表示立即数的第12到19位；[11:7]位为rd，表示5位目标寄存器；[6:0]位为opcode，表示7位指令操作码。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [31:20] | [19:15] | [14:12] | [11:7] | [6:0] |
| imm[11:0] | rs1 | funct3 | rd | opcode |

表 7 J-type jalr指令格式

如表 7所示，jalr指令第[31:20]位为imm[11:0]，表示12位立即数；[19:15]位为rs1，表示5位源寄存器；[14:12]位为funct3，表示3位功能码；[11:7]位为rd，表示5位目标寄存器；[6:0]位为opcode，表示7位指令操作码。

1. 设计过程

本次课程设计我选择RISC-V的理由如下：

（1）上学期的计算机组成原理已经设计过一个自定义指令集的单周期CPU，对于CPU的单周期已经有相对深入的了解，并且对于MIPS指令格式已经有了一定的了解，我想增加一些挑战难度，选择一个完全没有学过的指令集--RISC-V。

（2）无论是MIPS、ARM、RISC-V，都是RISC基于指令架构，没有过大的本质差别，ARM和MIPS指令都是固定长度的，RISC-V指令是变长的，所以我选择更有挑战性的RISC-V来设计CPU。

本次课程设计，我采用的是五段流水的CPU设计，即为IF，ID，EX，MEM，WB五个阶段，并在它们之间设置流水线寄存器。指令方面采用RISC-V指令架构，并通过停顿、冲刷指令、寄存器堆前推和数据前推技术解决相关与冒险。

* 1. 数据通路设计

在设计五段流水线CPU时候，我参考了计算机体系结构课程学习的内容，采用模块化思想。对于该CPU，它的一条指令的执行过程大致如下：

（1）从外部接入CLK信号接至pc\_reg，其中保存的pc值流出，并根据输入信号是顺序执行，跳转或是停顿，对pc值进行相应的算术操作。

（2）根据pc值从instr\_memory取出指令。

（3）在ID阶段对指令进行译码，根据指令类型与寄存器堆进行数据交换，并将译码流出的func3、func7和opcode流入control模块，并流出控制信号。

（4）根据ID段所取得的操作数和控制信号，在EX段执行相应运算。

（5）通过读写存储器指令信号和计算出的存储器地址，读取或写入存储器中对应单元的内容。

（6）根据指令译码结果，最终写回至寄存器组或存储器。

基于以上设计思路，最终在Vivado2018.3上实现的整体架构如图 2所示；vivado数据通路如图 3所示；控制单元如图 4所示。

图片包含 图示

描述已自动生成

图 2 整体架构

图示, 示意图

描述已自动生成

图 3 vivado数据通路

图片包含 图示

描述已自动生成

图 4 控制单元

* 1. 模块设计
     1. instr\_memory

如图 5所示，该模块用于指令的存储和读取。端口含义如表 8所示。

日程表

描述已自动生成

图 5 指令存储器

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | addr[7:0] | 指令地址 |
| output | instr[31:0] | 指令 |

表 8 instr\_memory端口

* + 1. data\_memory

如图 6所示，该模块用于数据的存储和读取。端口含义如表 9所示。

图表

中度可信度描述已自动生成

图 6 数据存储器

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | RW\_type[2:0] | 数据扩展类型使能 |
| input | R\_en | 读使能 |
| input | W\_en | 写使能 |
| input | addr[31:0] | 数据地址 |
| input | clk | 时钟信号 |
| input | din[31:0] | 写入数据 |
| input | rst\_n | 复位信号 |
| output | dout[31:0] | 读取数据 |

表 9 data\_memory端口

* + 1. main\_control

如图 7所示，该模块负责解析从instr\_decode模块流出的opcode和func3。端口含义如表 10所示。

图片包含 图表

描述已自动生成

图 7 main\_control模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | func3[2:0] | 3位指令功能码 |
| input | opcode[6:0] | 7位指令操作码 |
| output | ALUSrc | ALU数据源 |
| output | ALUop[1:0] | ALUctl控制信号 |
| output | MemRead | load指令使能 |
| output | MemWrite | store指令使能 |
| output | MemtoReg | 写入数据来源选择信号 |
| output | RW\_type[2:0] | 数据扩展类型使能 |
| output | U\_type | U-type指令使能 |
| output | beq | beq指令使能 |
| output | bge | bge指令使能 |
| output | bgeu | bgeu指令使能 |
| output | blt | blt指令使能 |
| output | bltu | bltu指令使能 |
| output | bne | bne指令使能 |
| output | jal | jal指令使能 |
| output | jalr | jalr指令使能 |
| output | lui | lui指令使能 |

表 10 main\_control端口

* + 1. alu\_control

如图 8所示，该模块负责解析从main\_control模块流出的ALUop[1:0]，以及从instr\_decode模块流出的func7和func3。端口含义如表 11所示。

图表

低可信度描述已自动生成

图 8 alu\_control模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | ALUop[1:0] | ALUctl控制信号 |
| input | func3[2:0] | 3位指令功能码 |
| input | func7 | 7位指令功能码 |
| output | ALUctl[3:0] | ALU计算控制信号 |

表 11 alu\_control端口

* + 1. pc\_reg

如图 9所示，该模块负责流出下一条指令地址。端口含义如表 12所示。

图表, 箱线图

描述已自动生成

图 9 pc\_reg模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | clk | 时钟信号 |
| input | load\_use\_flag | load-use型冒险使能 |
| input | pc\_new[31:0] | 输入pc值 |
| input | rst\_n | 复位信号 |
| output | pc\_out[31:0] | 输出pc值 |

表 12 pc\_reg端口

* + 1. instr\_decode

如图 10所示，该模块负责指令译码。端口含义如表 13所示。

图表

低可信度描述已自动生成

图 10 instr\_decode模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | instr[31:0] | 指令 |
| output | Rd[4:0] | 目标寄存器 |
| output | Rs1[4:0] | 源寄存器1 |
| output | Rs2[4:0] | 源寄存器2 |
| output | func3[2:0] | 3位指令功能码 |
| output | func7 | 7位指令功能码 |
| output | imme[31:0] | 32位立即数 |
| output | opcode[6:0] | 7位指令操作码 |

表 13 instr\_decode端口

* + 1. register

如图 11所示，该模块负责读出和写回数据。端口含义如表 14所示。

图片包含 图表

描述已自动生成

图 11 registers模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | Rd[4:0] | 目标寄存器 |
| input | Rs1[4:0] | 源寄存器1 |
| input | Rs2[4:0] | 源寄存器2 |
| input | W\_en | 写使能 |
| input | Wr\_data[31:0] | 写入数据 |
| input | clk | 时钟信号 |
| input | rst\_n | 置位信号 |
| output | Rd\_data1[31:0] | 读出数据1 |
| output | Rd\_data2[31:0] | 读出数据2 |

表 14 registers端口

* + 1. alu

如图 12所示，该模块负责计算输入的数据。端口含义如表 15所示。

图表

描述已自动生成

图 12 alu模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | ALU\_CTL[3:0] | alu计算控制信号 |
| input | ALU\_DA[31:0] | alu计算源操作数A |
| input | ALU\_DB[31:0] | alu计算源操作数B |
| output | ALU\_DC[31:0] | alu计算结果C |
| output | ALU\_ZERO | 结果是否为0 |

表 15 alu端口

ALU模块主要进行数据的运算，根据ALU的控制模块产生的控制信ALU\_CTL决定ALU进行的运算类型。

ALU\_CTL对应的运算关系如表 16所示。

|  |  |  |
| --- | --- | --- |
| ALU\_CTL | 运算类型 | 具体的运算 |
| 0000 | 加法运算 | 加 |
| 0011 | 加法运算 | 减 |
| 0100 | 逻辑运算 | 与 |
| 0101 | 逻辑运算 | 或 |
| 0110 | 逻辑运算 | 异或 |
| 0111 | 逻辑运算 | 或非 |
| 1000 | 小于置一 | 无符号小于置一 |
| 1001 | 小于置一 | 有符号小于置一 |
| 1100 | 移位运算 | 逻辑左移 |
| 1101 | 移位运算 | 逻辑右移 |
| 1110 | 移位运算 | 算数右移 |

表 16 ALU\_CTL对应的运算关系

由此可见，ALU运算分为加法运算、逻辑运算、小于置一、移位运算，根据ALU\_CTL高两位便可判断执行的运算类型。

在进行ALU操作时，该模块会将以下四种运算均进行实现，然后根据输入的控制信号进行选择输出数据。

（1）加法运算：由32位超前进位加法器实现。

（2）逻辑运算：由逻辑门实现。

（3）小于置一：实质上为减法运算，根据加法器的运算结果进一步判断是否小于。

（4）移位运算：由移位器实现。

* + 1. branch\_judge

如图 13所示，该模块负责判断指令是否为跳转指令。端口含义如表 17所示。

图表, 条形图

描述已自动生成

图 13 branch\_judge模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | ALU\_result\_sig | alu计算结果C的最低位 |
| input | beq | beq指令使能 |
| input | bge | bge指令使能 |
| input | bgeu | bgeu指令使能 |
| input | blt | blt指令使能 |
| input | bltu | bltu指令使能 |
| input | bne | bne指令使能 |
| input | jal | jal指令使能 |
| input | jalr | jalr指令使能 |
| input | zero | alu计算结果是否为0 |
| output | jump\_flag | 控制冒险使能 |

表 17 branch\_judge端口

* + 1. Shifter

如图 14所示，该模块负责进行移位运算。端口含义如表 18所示。

图片包含 图表

描述已自动生成

图 14 Shifter模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | ALU\_DA[31:0] | 移位数据 |
| input | ALU\_SHIFT[4:0] | 移位量 |
| input | Shiftctl[1:0] | 移位类型控制信号 |
| output | shift\_result[31:0] | 移位输出数据 |

表 18 Shifter端口

移位器用于进行移位操作，在此模块中无论Shifterctl为多少，移位器会先将其进行移位操作，把ALU\_DA进行逻辑右移、逻辑左移、算术右移，移位量由ALU\_SHIFT控制，然后再根据Shiftctl来选择输出的结果。

* + 1. Adder

如图 15所示，该模块负责进行移位运算。端口含义如表 19所示。

图表, 图示

描述已自动生成

图 15 Adder模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | A[31:0] | 加法操作数A |
| input | B[31:0] | 加法操作数B |
| input | ALU\_CTL[3:0] | ALU计算控制信号 |
| input | Cin | 低位进位 |
| output | ADD\_OverFlow | 计算结果溢出信号 |
| output | ADD\_carry | 计算结果进位信号 |
| output | ADD\_result[31:0] | 计算结果 |
| output | ADD\_zero | 计算结果结果是否为0 |

表 19 Adder端口

加法器用于进行加法操作，在此模块中实例化了一个32位超前进位加法器，并且根据两个加法操作数和计算结果进行判断是否溢出，溢出情况如表 20所示。

|  |  |  |  |
| --- | --- | --- | --- |
| 操作数A | 计算符号 | 操作数B | 计算结果 |
| 正数 | + | 正数 | 负数 |
| 负数 | + | 负数 | 正数 |
| 正数 | - | 负数 | 负数 |
| 负数 | - | 正数 | 正数 |

表 20 加法溢出情况

* + 1. cla\_adder32

如图 16所示，该模块负责进行加法，为32位超前进位加法器，可以更快的进行加法。端口含义如表 21所示。

图表

中度可信度描述已自动生成

图 16 cla\_adder32模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | A[31:0] | 加法操作数A |
| input | B[31:0] | 加法操作数B |
| input | cin | 低位进位 |
| output | cout | 高位进位 |
| output | result[31:0] | 计算结果 |

表 21 cla\_adder32端口

32位超前进位加法器用于加法操作，在此模块中实例化了11个4位超前进位加法器。在该模块中，先将传入的A、B两个操作数进行按位与、按位或操作，得到了 32位的“传递”信号、32位的“生成”信号。将这两个信号按4位为一组分成8组，将“传递”信号和“生成”信号两两一组，分别传入8个实例化的4位超前进位加法器中，将计算的结果再次按照4位一组分成两组，将其传入2个4位超前进位加法器中，最后将计算出的最后一组结果再次传入1个4位超前进位加法器中，即可得到最终的计算结果。该32位超前进位加法器的结构图如图 17所示。

图形用户界面

描述已自动生成

图 17 32位超前进位加法器

* + 1. cla\_4

如图 18所示，该模块负责进行加法，为4位超前进位加法器。端口含义如表 22所示。

图表, 箱线图

描述已自动生成

图 18 cla\_4模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | c\_in | 低位进位 |
| input | g[3:0] | 加法操作的传递信号 |
| input | p[3:0] | 加法操作的生成信号 |
| output | c[4:1] | 4个进位信号 |
| output | gx | 最高位的进位生成信号 |
| output | px | 进位是否传递到更高位 |

表 22 cla\_4端口

4位超前进位加法器，有生成和传递两个参数。

生成：如果两个加数的该位都是1，则无论前一位的进位如何，这一位都将生成一个新的进位。在代码中，生成信号g[i]表示为加数A和B在该位的AND操作。

传递：如果两个加数中至少有一个是1，则该位可能会将进位传递到下一位。在代码中，传递信号p[i]表示为加数A和B在该位的 XOR操作。

该公式代表如果某一位能生成进位（a[i]b[i]），或者能传递前一位的进位（(a[i] ^ b[i])c[i-1]），那么该位就会产生一个进位c[i]。

* + 1. forward\_unit

如图 19所示，该模块负责数据前推。端口含义如表 23所示。

表格

低可信度描述已自动生成

图 19 forward\_unit模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | MemRead\_ex\_mem\_o | MemRead来自ex\_mem\_o |
| input | MemRead\_id\_ex\_o | MemRead来自id\_ex\_o |
| input | MemWrite\_id\_ex\_o | MemWrite来自id\_ex\_o |
| input | MemWrite\_id\_ex\_i | MemWrite来自id\_ex\_i |
| input | Rd\_ex\_mem\_o | Rd来自ex\_mem\_o |
| input | Rd\_mem\_wb\_o | Rd来自mem\_wb\_o |
| input | Rd\_id\_ex\_o | Rd来自id\_ex\_o |
| input | RegWrite\_mem\_wb\_o | RegWrite来自mem\_wb\_o |
| input | RegWrite\_ex\_mem\_o | RegWrite来自ex\_mem\_o |
| input | RegWrite\_id\_ex\_o | RegWrite来自id\_ex\_o |
| input | Rs1\_id\_ex\_o | Rs1来自id\_ex\_o |
| input | Rs1\_id\_ex\_i | Rs1来自d\_ex\_i |
| input | Rs2\_id\_ex\_o | Rs2来自id\_ex\_o |
| input | Rs2\_id\_ex\_i | Rs2来自id\_ex\_i |
| output | forwardA[1:0] | ALU第一个操作数来源 |
| output | forwardB[1:0] | ALU第二个操作数来源 |
| output | forwardC | 写入数据存储及数据来源 |
| output | load\_use\_flag | load-use型冒险使能 |

表 23 forward\_unit端口

* + 1. mux

如图 20所示，该模块负责二选一多路选择。端口含义如表 24所示。

图表, 箱线图

描述已自动生成

图 20 mux模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | data1[31:0] | 选择数据1 |
| input | data2[31:0] | 选择数据2 |
| input | sel | 选择使能 |
| output | dout[31:0] | 选择输出数据 |

表 24 mux端口

* + 1. mux3\_1

如图 21所示，该模块负责三选一多路选择。端口含义如表 25所示。

图表

描述已自动生成

图 21 mux3\_1模块

|  |  |  |
| --- | --- | --- |
| 方向 | 名称 | 含义 |
| input | din1[31:0] | 选择数据1 |
| input | din2[31:0] | 选择数据2 |
| input | din3[31:0] | 选择数据3 |
| input | sel[1:0] | 选择使能 |
| output | dout[31:0] | 选择输出数据 |

表 25 mux3\_1端口

* 1. 五级流水线设计

流水线信号传递如图 22所示。该图中展示了在流水线之间需要向下一级传递的数据信号和控制信号。

图形用户界面, 应用程序, 表格, Excel

描述已自动生成

图 22 信号传递表

* + 1. if\_stage

图表, 瀑布图

描述已自动生成

图 23 取指阶段数据通路

取指阶段如图 23所示。取指阶段包括pc\_reg和instr\_memory两个子模块。此处 instr\_memory为一个存储器，将其当作CPU外部的部件而不将其写入if\_stage阶段。在该阶段会根据控制信号得到pc\_out值，并将pc\_out拆分出rom\_addr[7:0]，将其传入instr\_memory来寻找指令。

* + 1. if\_id\_regs

图表

描述已自动生成

图 24 if\_id\_regs

IF/ID寄存器如图 24所示。此模块传递2个信号，分别是pc和读出的32位的指令。load\_use\_flag是用来解决加载-使用型冒险的信号，当该信号为高电平时，要暂停if\_id\_regs的更新，将信号instr\_if\_id\_o赋值给自己，即instr\_if\_id\_o<=instr\_if\_id\_o；jump\_flag是判断是否为跳转指令，如果是则需要将instr\_if\_id\_i在该寄存器内阻塞一个回合因为后续指令已经读出，所以需要将其置0，即instr\_if\_id\_o<=zeroword。

* + 1. id\_stage

示意图

中度可信度描述已自动生成

图 25 译码阶段数据通路

译码阶段如图 25所示。在该阶段会将从if\_id\_regs寄存器中流出的信号传入id\_stage，将instr[31:0]传入instr\_decode模块进行译码；将译码得到的rs1、rs2以及从if\_id\_regs寄存器中流出的信号传入registers，进行读写数据；最后将读出的数据以及译码结束得到的各个信号传至id\_ex\_regs。

* + 1. id\_ex\_regs

表格

描述已自动生成

图 26 id\_ex\_regs

ID/EX寄存器如图 26所示。此模块主要用于传递信号，将数据信号在时钟上升沿到来，rst\_n不为0时向后传递；将控制信号在时钟上升沿到来时，需要冲刷流水线、停顿流水线，即jump\_flag ==1 | load\_use\_flag == 1时置为0；在rst\_n不为0时向后传递。

* + 1. ex\_stage

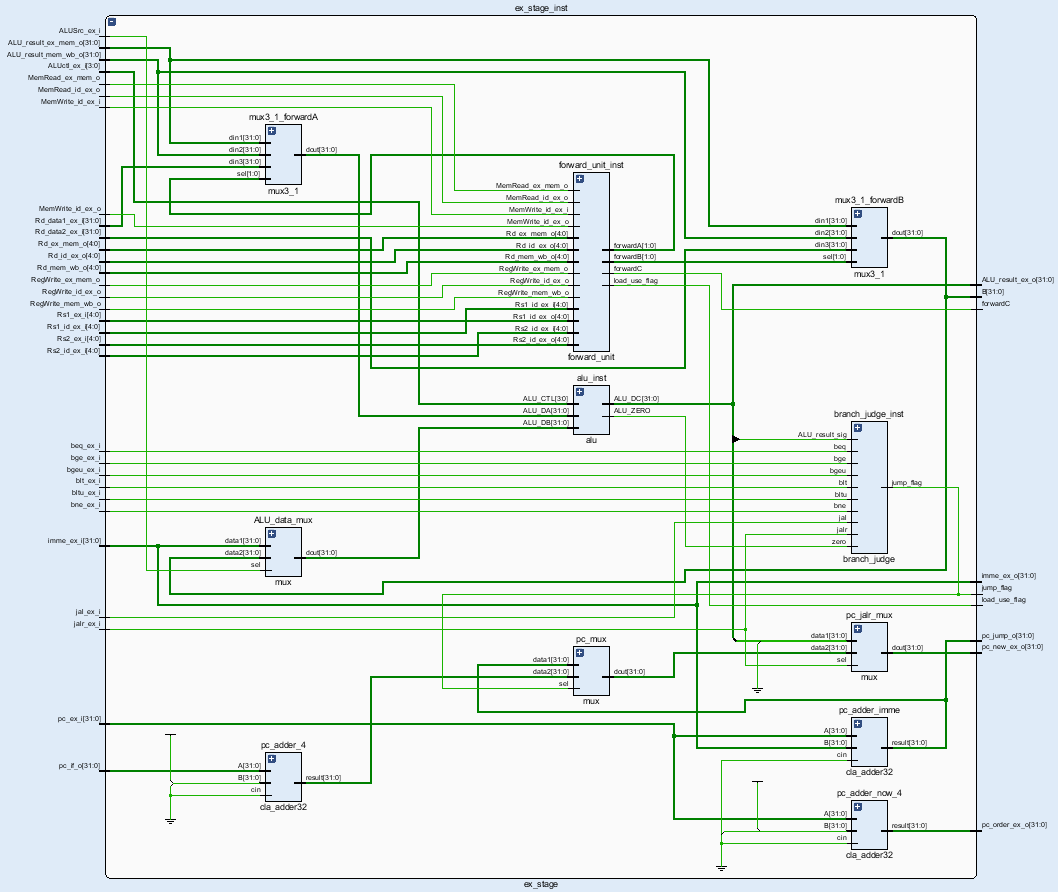


图 27 执行阶段数据通路

执行阶段如图 27所示。该阶段有alu、Shifter、Adder、cla\_adder32、cla\_4、branch\_judge、forward\_unit、pc\_adder\_4、pc\_adder\_now\_4、pc\_adder\_imme、pc\_mux、pc\_jalr、ALU\_data、mux3\_1\_forwardA、mux3\_1\_forwardB模块。接下来具体介绍主要模块。

（1）alu：将传入ALU的两个操作数A、ALU\_DB按照控制信号进行运算。

（2）Shifter：将传入的操作数ALU\_DA按照控制信号Shifterctl进行移位。

（3）Adder：将两个操作数、ALU\_CTL、Cin传入Adder进行加法。

（4）branch\_judge：将条件跳转使能信号传入该模块，生成jump\_flag。

（5）forward\_unit：数据前推单元，根据EX阶段使用的ALU的操作数来源，生成使能信号，控制数据前推。

（6）pc\_adder\_4：将if\_stage流出的pc通过超前进位加法器进行加4.

（7）pc\_adder\_now\_4：将从ex\_stage流出的当前指令的pc通过超前进位加法器进行加4。

（8）pc\_adder\_imme：通过分析指令得出指令为跳转指令，将从ex\_stage流出的当前指令的pc通过超前进位加法器进行加立即数imme。

（9）pc\_mux：根据jump\_flag判断指令是顺序执行指令还是跳转指令。

（10）pc\_jalr：根据jalr\_ex\_i判断指令是为jalr指令还是其他。

（11）ALU\_data：根据ALUSrc\_ex\_i来判断alu的操作数是为立即数还是B。

（12）mux3\_1\_forwardA：根据数据前推信号A来判断ALU的操作数1来自ex\_mem寄存器还是mem\_wb寄存器还是Rd\_data1。

（13）mux3\_1\_forwardB：根据数据前推信号B来判断ALU的操作数2来自ex\_mem寄存器还是mem\_wb寄存器还是Rd\_data2。

* + 1. ex\_mem\_regs

表格

中度可信度描述已自动生成

图 28 ex\_mem\_regs

EX/MEM寄存器如图 28所示。此模块主要用于传递信号，将信号在时钟上升沿到来，rst\_n不为0时向后传递。

* + 1. mem\_stage

图表

描述已自动生成

图 29 访存阶段数据通路

访存阶段如图 29所示。在该阶段会将从ex\_mem\_regs寄存器中流出的信号传入mem\_stage，然后通过二选一多路选择器，根据forwardC使能信号，判断写入数据存储器的数据是来自EX/MEM寄存器还是MEM/WB寄存器。

* + 1. mem\_wb\_regs

应用程序

中度可信度描述已自动生成

图 30 mem\_wb\_regs

MEM/WB寄存器如图 30所示。此模块主要用于传递信号，将信号在时钟上升沿到来，rst\_n不为0时向后传递。

* + 1. wb\_stage

图示

描述已自动生成

图 31 写回阶段数据通路

写回阶段如图 31所示。在该阶段会根据MemtoReg使能信号，判断数据来源是loaddata还是ALU\_result；根据reg\_sel使能信号判断Wr\_reg\_data2数据是来自pc\_order还是WB\_data；根据lui使能信号判断Wr\_reg\_data1数据来自imme还是pc\_jump；最后根据U\_type使能信号，判断写回到寄存器的数据到底是Wr\_reg\_data1还是Wr\_reg\_data2。

1. 相关与冒险
   1. 结构冒险

结构冒险是指因缺乏硬件支持而导致指令不能在预定的时钟周期内执行的情况，即硬件不支持多条指令在同一时钟周期执行。如果该处理器只有一个存储器同时存储指令和数据，那么在访存和取指阶段将同时访问存储器，若该存储器只有一个访问接口，将会发生结构冒险。

在本次课设中，我将指令存储器和数据存储器分开，避免了对存储器同时读写所产生的冲突。为了防止多条指令同时读写存储器，我将其设置了在时钟上升沿写，在任何时间都能读，从而实现同步写，异步读。

* 1. 数据冒险

数据冒险分RAW，WAR 和WAW 三类，因为本次设计的CPU未使用动态分支预测技术，所以只会出现RAW 冒险。

经过分析，RAW冒险大致分为如下5种类型。

（1）在一个周期开始，EX 阶段要使用上一条处在 EX 阶段指令的执行结果，此时我们将 EX/MEM 寄存器的数据前递。

（2）在一个周期开始，EX 阶段要使用上一条处在 MEM 阶段指令的执行结果，此时我们将 MEM/WB 寄存器的数据前递。

（3）在一个周期开始，EX 阶段要使用上一条处在 WB 阶段指令的执行结果，此时不需要前递，利用寄存器堆前递机制。

（4）在第一种情况下，如果是上一条是访存指令，即发生加载—使用型冒险。则需要停顿一个周期。

（5）在发生加载——使用型冒险的时候，如果是load后跟着store指令，并且load指令的rd与store指令的rs1 不同而与rs2相同，则不需要停顿，只需要将MEM/WB 寄存器的数据前递到MEM阶段。

* + 1. 寄存器堆前推

在一个周期开始，EX 阶段要使用上一条处在 WB 阶段指令的执行结果，此时不需要前递，利用寄存器堆前推机制。

表格

中度可信度描述已自动生成

图 32 寄存器堆前递

如图 32所示，寄存器堆前推机制只需要把在时钟上升沿写寄存器改为在时钟下降沿进行写寄存器。

对于下面这部分指令进行分析：

addi x1,x0,1

addi x2,x0,2

addi x3,x0,3

addi x4,x1,3

当这段指令中的前一条指令处于译码阶段时（第5个时钟周期），需要读取x1寄存器值，在之前的设计中，写寄存器发生在上升沿。在上升沿采集信号，得到的信号仍然是这个时钟周期前一时刻的信号，也就是第4个时钟周期末的信号。但是待写回的x1寄存器值在第5个时钟周期才能到来（第一条指令进入WB阶段），所以仍然会发生冒险。

因此在第5个周期的高电平期间，读的x1的值是旧的值，而在第5个周期的低电平期间，读的x1的值是最新的值。这样，在第四条指令的执行阶段，采集到的x1的信号就是最新的值而不会发生冒险。

* + 1. 数据前推

数据前推想要前推正确的数据，则需要分辨三种不同的情况，并且给出标志信号，即设计正确的前推检测单元。

|  |  |  |
| --- | --- | --- |
| 控制信号 | 数据源 | 解释 |
| forwardA =1X | EX/MEM | ALU的第一个源操作数来自于EX/MEM流水线寄存器的数据前递 |
| forwardA =01 | MEM/WB | ALU的第一个源操作数来自于EX/MEM流水线寄存器的数据前递 |
| forwardA =00 | ID/EX | ALU的第一个源操作数来自于ID/EX流水线寄存器的数据 ，不需要前递 |
| forwardB =1x | EX/MEM | ALU的第二个源操作数来自于EX/MEM流水线寄存器的数据前递 |
| forwardB =01 | MEM/WB | ALU的第二个源操作数来自于MEM/WB流水线寄存器的数据前递 |
| forwardB =00 | ID/EX | ALU的第二个源操作数来自于ID/EX流水线寄存器的数据 ，不需要前递 |
| forwardC =1 | MEM/WB | 写入数据存储器的数据来自于MEM/WB流水线寄存器的数据前递 |
| forwardC =0 | EX/MEM | 写入数据存储器的数据来自于EX/MEM流水线寄存器的数据，不需要前递 |

forwardA：ALU第一个操作数来源控制信号

forwardB：ALU第二个操作数来源控制信号

forwardC：写入数据存储器数据来源控制信号

加入数据前递之后，ALU的数据来源变成三个，分别是ID/EX流水线寄存器的数值，EX/MEM流水线寄存器前递的数值和MEM/WB流水线寄存器前递的数值，因此要添加三选一多路选择器。如图 33所示。

文本, 信件

描述已自动生成

图 33 ALU数据来源

还有一组前递发生在访存阶段，由于数据来源只有两种，用forwardC控制一个二选一多路选择器即可。如图 34所示。

文本, 信件

描述已自动生成

图 34 数据存储器数据来源

* + 1. 停顿流水线

对于加载-使用型相关中，访存指令的结果要在访存结束后才能知道，但是对于下一条使用型指令来说，指令处于译码阶段时候，访存指令处于执行阶段；使用型指令在执行阶段开始时就需要使用最新的寄存器值，而此时访存指令刚进入访存阶段，还没来得及读出待写回的寄存器值，因此只能通过停顿一个周期的流水线来等访存指令访存结束之后再进行数据前推。

在执行load指令阶段，判断出加载-使用型相关并产生一个标志信号，如图 35所示；在下一个时钟周期的上升沿采集到这个信号，然后清除ID/EX流水线寄存器的控制信号，如图 36所示；暂停IF/ID流水线寄存器的更新，如图 37所示；暂停pc的更新，如图 38所示。

文本, 信件

描述已自动生成

图 35 生成load\_use\_flag信号

文本

描述已自动生成

图 36 清除ID/EX寄存器控制信号

文本, 信件

描述已自动生成

图 37 暂停IF/ID寄存器更新

文本, 信件

描述已自动生成

图 38 暂停pc更新

* 1. 控制冒险

当遇到控制相关时，采用假设分支不发生的方法，当分支跳转发生时，就需要冲刷掉多余的两条指令。

下面给出一个示例：

第一个时钟周期：取指阶段取出第一条跳转指令。

第二个时钟周期：跳转指令译码，同时取指阶段取出一条新指令instr1。

第三个时钟周期：跳转指令进入执行阶段，同时取指阶段取出一条新指令instr2，在本周期结束时，跳转目标地址已生成。

第四个时钟周期上升沿：取指阶段即将取出跳转的目标地址对应的指令，此时instr1即将进入执行阶段，instr2即将进入译码阶段。

此时，instr1已经完成译码，并且控制信号已经输入至ID/EX寄存器，instr1即将进入执行阶段，所以只需要将此时ID/EX寄存器中的控制信号置0，不采集instr1的控制信号即可实现指令冲刷，如图 39所示。

此时，instr2已经完成取指，并且将指令输入到了IF/ID寄存器，instr2即将进入译码阶段，所以只需要将此时IF/ID寄存器中的指令置0，即可实现指令冲刷，如图 40所示。

目前CPU将无条件跳转也考虑入控制相关，所以只要发生跳转，就需要进行冲刷流水线，所以可以把branch\_judge模块的输出信号jump\_flag当作发生控制冒险的信号。

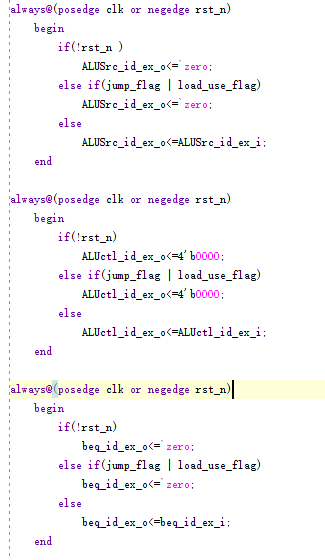


图 39 ID/EX寄存器

文本, 信件

描述已自动生成

图 40 IF/ID寄存器

1. 仿真测试
   * 1. 数据前推测试

（1）第一组指令如图 41所示，仿真结果如图 42所示。

文本

描述已自动生成

图 41 第一组指令

电脑萤幕画面

中度可信度描述已自动生成

图 42 第一组仿真

第一组指令出现了数据相关，通过仿真可以看到均能正确执行。

（2）第二组指令如图 43所示，仿真结果如图 44所示。

文本

描述已自动生成

图 43 第二组指令

图片包含 示意图

描述已自动生成

图 44 第二组仿真

由于sw指令的特殊之处，在访存阶段才会使用源寄存器的读出的数据，也就是传入EX/MEM流水线寄存器的Rd\_data2应该是经过多选器选择之后的数据（也就是前递之后的数据）而不是ID/EX流水线寄存器的Rd\_data2（前递之前的数据）。

第二指令通过仿真可以看到均能正确执行。

* + 1. 停顿流水线测试

文本

描述已自动生成

图 45 加载-使用型指令

图形用户界面

描述已自动生成

图 46 加载-使用型仿真

加载-使用型相关指令如图 45所示，仿真结果如图 46所示。

在load\_use\_flag信号拉高后，处理器进行了一个周期的停顿，执行结果正确。

* + 1. 冲刷流水线测试

一些文字和图案

描述已自动生成文本

描述已自动生成

图 47 控制冒险指令

图片包含 图形用户界面

描述已自动生成

图 48 控制冒险仿真1

图形用户界面

描述已自动生成

图 49 控制冒险仿真2

控制冒险指令如图 47所示，仿真如图 48图 49所示。

最终运行结果正确，能够正常冲刷指令。

1. 总结

本次课程设计后，我对CPU设计有了更深刻的理解。设计基于RISC-V指令架构的多周期流水线CPU对我来说初显陌生，处处碰壁。但通过回归基础，深入理解CPU设计的各个环节，我逐渐掌握了数据通路的绘制和代码实现。这个过程虽复杂，但从简单逻辑运算指令到整体框架的构建，再到处理相关和冒险问题，都是必经之路。

CPU的设计并不是像我之前想象的那么遥不可及，但也并非简单的任务。在编写完整个代码后回头看，我发现我设计的CPU并不是太复杂，但勉强实现了预期的功能。其中最关键的是要真正弄清楚每个CPU组成部分的工作原理，这样才能正确地将各个部件的逻辑联系起来。此外，还需要明确各个部件的前后时序关系，这是设计流水线CPU的关键。我设计的这个流水线CPU相对简单，因为真实环境中EX阶段的运算并不像我设计的那样简单，它涉及到补码、移位等一系列变换来实现各种复杂的运算。

通过这次课程设计，我不仅提升了在数字电路和计算机体系结构领域的综合应用能力，还深刻理解了RISC-V指令集的设计理念及其在现代计算机体系结构发展中的重要性。我也学会了如何更好地管理复杂性，使设计更清晰可维护。此外，我对CPU流水线设计原理有了更直观的认识。最后，我坚信随着国内外对硬件领域的持续投入和研究，我国的系统硬件发展将逐步走向世界前列。

1. 参考文献

[1]不学无术的小胖子. 从零开始设计RISC-V处理器——指令系统. <https://blog.csdn.net/qq_45677520/article/details/122309083>, 2022.01.04.

[2] 不学无术的小胖子. 从零开始设计RISC-V处理器——单周期处理器的设计. <https://blog.csdn.net/qq_45677520/article/details/122386632>, 2022.01.10.

[3] 不学无术的小胖子. 从零开始设计RISC-V处理器——单周期处理器的仿真. <https://blog.csdn.net/qq_45677520/article/details/122419435>, 2022.01.13.

[4] 不学无术的小胖子. 从零开始设计RISC-V处理器——ALU的优化. <https://blog.csdn.net/qq_45677520/article/details/122489700>, 2022.01.15.

[5] 不学无术的小胖子. 从零开始设计RISC-V处理器——五级流水线之数据通路的设计. <https://blog.csdn.net/qq_45677520/article/details/122549842>, 2022.02.03.

[6] 不学无术的小胖子. 从零开始设计RISC-V处理器——五级流水线之控制器的设计. <https://blog.csdn.net/qq_45677520/article/details/122796817>, 2022.02.07.

[7] 不学无术的小胖子. 从零开始设计RISC-V处理器——五级流水线之数据冒险. <https://blog.csdn.net/qq_45677520/article/details/122806845>, 2022.02.14.

[8] 不学无术的小胖子. 从零开始设计RISC-V处理器——五级流水线之控制冒险. <https://blog.csdn.net/qq_45677520/article/details/124675730>, 2022.05.10.

|  |
| --- |
| 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:** |
| |  |  |  |  | | --- | --- | --- | --- | |  | 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）： 2023 年6 月 30 日 | | | | |