|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | 硬件课程设计报告 | | C++实现RISC-V模拟器 |  |  | | --- | | **组长：薛皓天 210340170**  **组员：薛傲翔 210340169**  **杨馥银 210340172** | |

**目录**

[一、 设计目的 4](#_Toc178021192)

[二、 设计过程 4](#_Toc178021193)

[2.1 设计过程说明 4](#_Toc178021194)

[2.1.1 模型机位数选定 4](#_Toc178021195)

[2.1.2 寄存器设计 5](#_Toc178021196)

[2.1.3 设计指令格式 6](#_Toc178021197)

[2.1.4 内存设计 7](#_Toc178021198)

[2.2 指令集设计 7](#_Toc178021199)

[2.2.1 R型指令 7](#_Toc178021200)

[2.2.2 I型指令 9](#_Toc178021201)

[2.2.3 S型指令 10](#_Toc178021202)

[2.2.4 B型指令 12](#_Toc178021203)

[2.2.5 U型指令 13](#_Toc178021204)

[2.2.6 J型指令 14](#_Toc178021205)

[2.3 代码数据处理 15](#_Toc178021206)

[2.3.1 指令文件读取和处理 15](#_Toc178021207)

[2.3.2 模拟器初始化 16](#_Toc178021208)

[2.3.3 运行指令 17](#_Toc178021209)

[2.4 程序测试 18](#_Toc178021210)

[2.4.1 计算类指令测试 18](#_Toc178021211)

**表目录**

[表1 RISC-V模拟器寄存器 5](#_Toc177917069)

**图目录**

[图1 指令的格式图1 指令的格式 1 5](#_Toc177917116)

# 一、 设计目的

RISC-V是一种开源的指令集架构（ISA），具有高度的灵活性和可扩展性。RISC-V规范定义了基本整数指令集（RV32I/RV64I，分别对应32位和64位），并允许通过可选扩展来添加额外的功能，如浮点运算（RV32F/RV64F）、向量运算（RVV）、原子操作（RVA）、压缩指令集（RVC）等。这种模块化设计使得RISC-V能够根据不同的应用需求进行定制，从而优化性能、功耗和成本。

C++作为一种强大的编程语言，提供了丰富的编程特性和库支持，使得我们可以根据需要灵活地实现RISC-V模拟器的各种功能。此外，由于RISC-V架构的模块化设计，模拟器可以很容易地进行定制和扩展，以适应不同的应用场景和需求。

本次硬件课程设计，我们小组使用C++实现RISC-V模拟器目的在于：

1、 通过亲手编写模拟器，可以更深入地理解RISC-V指令集架构的内部工作机制，包括指令的执行流程、寄存器的使用、内存访问等关键概念。这将有助于我们在未来的学习和工作中更好地应用RISC-V架构。

2、 编写RISC-V模拟器是一个将计算机体系结构理论知识付诸实践的好机会。我们将能够应用所学到的关于处理器设计、指令集、流水线、缓存、存储层次结构等概念，通过编程实现这些概念在模拟器中的具体表现。

3、 在编写模拟器的过程中，你们可以尝试实现不同的RISC-V扩展指令集，如浮点运算、向量运算等，以探索这些扩展如何影响处理器的性能和功能。这将有助于你们更深入地理解RISC-V架构的灵活性和可扩展性，并为未来的研究和开发提供经验。

# 二、 设计过程

2.1 设计过程说明

### 2.1.1 模型机位数选定

该RISC-V模型机采用了先进的32位设计，这一核心特性赋予了它卓越的数据处理能力。具体而言，32位的设计意味着该模型机能够直接且高效地处理32位宽的数据单元和操作码，这种能力对于现代计算任务而言至关重要。无论是处理复杂的数学运算、管理庞大的数据集，还是执行精密的逻辑判断，32位架构都能提供足够的位宽来确保操作的准确性和效率。

### 2.1.2 寄存器设计

在设计RISC-V模拟器的过程中，寄存器的配置是至关重要的一环。RISC-V架构精心定义了一套寄存器系统，以满足各种计算任务的需求。如表1所示，RISC-V架构明确规定了32个32位宽（即每个寄存器能够存储4字节的数据）的通用寄存器，这些寄存器构成了处理器内部数据存储与操作的核心部分。

在32个通用寄存器中，有31个被设计为常规寄存器，它们没有特殊的限制或约束，允许程序自由地对其进行读写操作。这意味着开发者可以根据需要，在这些寄存器中存储临时数据、中间结果或程序状态等信息，以便后续指令的处理和计算。

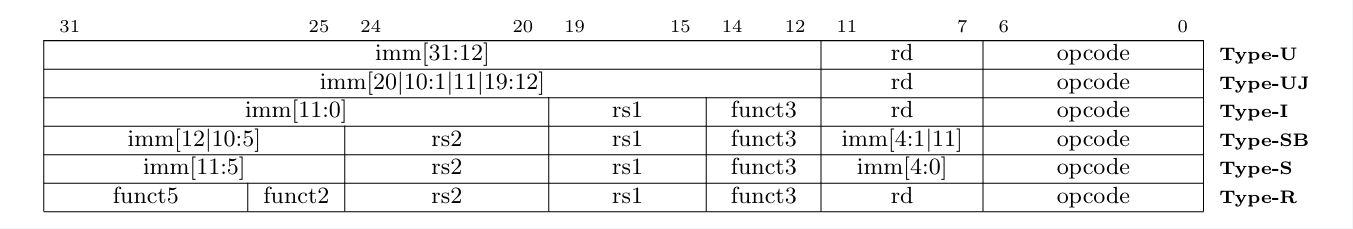
一个特殊的寄存器，即x0寄存器（通常也被称为零寄存器或zero寄存器）。与其他常规寄存器不同，x0寄存器被硬编码为始终包含0值，且其值不可更改。这一设计在RISC-V架构中具有多重用途，例如作为某些指令的默认操作数、简化指令集设计以及提高代码密度等。由于x0寄存器只能读不能写，因此它不会成为程序执行过程中数据修改或错误的源头，从而增强了处理器的稳定性和可靠性。

表1 RISC-V模拟器寄存器

|  |  |  |
| --- | --- | --- |
| 寄存器 | ABI名称 | 说明 |
| x0 | zero | 0值寄存器，读取数据为0 |
| x1 | ra | 用于返回地址(return address) |
| x2 | sp | 用于栈指针（stack pointer） |
| x3 | gp | 用于通用指针 (global pointer) |
| x4 | tp | 用于线程指针 （thread pointer） |
| x5 | t0 | 用于存放临时数据或者备用链接寄存器 |
| x6-x7 | t1-t2 | 用于存放临时数据寄存器 |
| x8 | s0/fp | 需要保存的寄存器或者帧指针寄存器 |
| x9 | s1 | 需要保存的寄存器 |
| x10-x11 | a0-a1 | 函数传递参数寄存器或者函数返回值寄存器 |
| x12-x17 | a2-a7 | 函数传递参数寄存器 |
| x18-x27 | s2-s11 | 需要保存的寄存器 |
| x28-x31 | t3-t6 | 用于存放临时数据寄存器 |

### 2.1.3 设计指令格式

RISC-V架构以其精简的指令集和灵活的指令格式而著称，定义了六种基本指令格式，每种格式都针对不同类型的操作进行了优化。如图1所示，这些指令格式被清晰地呈现出来，每种格式都包含了执行指令所需的关键元素。在RISC-V的指令格式中，opcode（指令操作码）占据了核心地位，它是指令的标识符，决定了指令的类型和将要执行的操作。紧随opcode之后的是imm，这是一个直接嵌入在指令中的数值，用于某些需要立即数作为操作数的指令中，当然在条件跳转指令中，其表示的意义是标号。funct字段则代表了指令对应的具体功能，它进一步细化了opcode所指定的操作类型，使得同一类型的指令能够执行多种不同的操作。这种设计提高了指令集的灵活性和表达能力。RISC-V指令格式还包括了寄存器相关的字段，如rs1（源寄存器1）和rs2（源寄存器2），它们分别指向了指令执行时所需的两个源操作数所在的寄存器。这些寄存器中的值将被用作指令操作的输入。rd（目标寄存器）字段指定了指令执行结果应该存储在哪个寄存器中。

图1 指令的格式

见表2所示为具体的每种的指令实现的功能。

表2 各种指令的功能

|  |  |  |
| --- | --- | --- |
| 序号 | 指令类型 | 功能 |
| 1 | R型指令 | 用于寄存器和寄存器操作 |
| 2 | I型指令 | 用于短立即数和内存载入指令load操作 |
| 3 | S型指令 | 用于内存存储store操作 |
| 4 | B（SB）指令 | 用于有条件跳转操作 |
| 5 | U型指令 | 用于长立即数操作 |
| 6 | J（UJ）型指令 | 用于无条件跳转操作 |

### 2.1.4 内存设计

采用C++自带的库函数map实现，在C++中，std::map是一个非常有用的容器，它存储元素形成键值对（key-value pairs），其中每个键都是唯一的，并且根据键的排序准则自动排序元素。虽然std::map的键（key）在内部实现中并不直接对应内存的物理地址，而是作为一个抽象的概念存在，用于唯一标识和访问与之关联的值（value），但我们可以将这个概念理解为在逻辑上形成了一个类似于“地址-数据”的映射关系，其中“地址”指的是通过键来访问数据的唯一标识符，而“数据”则是存储在std::map中的值（value）。进一步扩展这个描述，std::map通过其高效的内部数据结构（通常是红黑树）来管理这些键值对，确保了对数时间复杂度的查找、插入和删除操作。这意味着，无论std::map中存储了多少元素，这些操作的时间开销都相对可控，与元素总数的对数成正比。此外，std::map还提供了丰富的成员函数来支持各种操作，比如find用于查找键对应的值，insert用于插入新的键值对，erase用于删除指定的键值对，以及begin、end等迭代器相关的成员函数，使得遍历std::map中的所有元素变得简单直接。

2.2 指令集设计

### 2.2.1 R型指令

如表3所详尽展示的，是R型指令（Register-type Instruction）的基本格式，这种指令类型在处理器架构中占据重要地位，一共实现了10条具体指令。这些基本操作主要聚焦于寄存器和寄存器之间的交互，包括但不限于算术逻辑运算（如加法、减法、位运算等）以及复杂的数据计算任务。为了提高代码的可读性和复用性，这些R型指令被封装在一个名为Rtype的类中。

如图2所示，Rtype类不仅反映了R型指令的内在逻辑和特性，还通过其精心设计的成员函数为外部提供了便捷的调用接口。这些接口允许开发者以面向对象的方式，直接利用Rtype类的实例来执行相应的R型指令，无需深入了解指令背后的复杂实现细节。这种封装方式极大地简化了程序设计，使得处理器指令的调用变得更加直观和高效。同时，Rtype类的设计也充分考虑了可扩展性和可维护性，为未来可能新增的R型指令或指令集的升级预留了足够的空间。

表3 R型指令的格式

|  |  |  |
| --- | --- | --- |
| 指令名称 | 格式 | 功能 |
| ADD指令 | ADD rd,rs1,rs2 | rs1 + rs2 -> rd |
| SUB指令 | SUB rd,rs1,rs2 | rs1 - rs2 -> rd |
| XOR指令 | XOR rd,rs1,rs2 | rs1 ^ rs2 -> rd |
| SRL指令 | SRL rd,rs1,rs2 | (逻辑右移)rs1 >> rs2 -> rd |
| OR指令 | OR rd,rs1,rs2 | rs1 | rs2 -> rd |
| AND指令 | AND rd,rs1,rs2 | rs1 & rs2 -> rd |
| SLL指令 | SLL rd,rs1,rs2 | (逻辑左移)rs1 << rs2 -> rd |
| SLT指令 | SLT rd,rs1,rs2 | （有符号）rs1<rs2时，rd=1 |
| SRA指令 | SRA rd,rs1,rs2 | (算术右移)rs1 >> rs2 -> rd |
| SLTU指令 | SLTU rd,rs1,rs2 | （无符号）rs1<rs2时，rd=1 |



图2 R型指令类

### 2.2.2 I型指令

如表4所详细展示的是I型指令（Immediate-type Instruction）的基本格式，这一格式针对需要处理短立即数以及执行内存载入（load）操作等特定任务进行了优化。在I型指令集中，一共实现了15条具体指令，每一条都承担着执行特定基本功能的重要角色。这些功能包括但不限于从内存中高效地读取数据、对短立即数执行算术逻辑运算或数据计算等，为程序提供了灵活且强大的数据处理能力。

为了提升代码的可读性、可维护性以及复用性，这些I型指令被精心封装在了一个名为Itype的类中。Itype类不仅准确地反映了I型指令的内在逻辑和特性，还通过其内部定义的成员函数和属性，为外部提供了简洁而高效的调用接口。这一设计使得开发者在编写涉及I型指令的程序时，能够更加方便地调用这些指令，而无需深入了解它们背后的复杂实现细节。

如图3所示，Itype类的实现图展示了该类的主要结构和关键组件。通过这个图，我们可以直观地看到Itype类是如何组织起来的，包括它是如何封装I型指令的操作码、立即数、目标寄存器等关键信息的。

表4 I型指令的格式

|  |  |  |
| --- | --- | --- |
| 指令名称 | 格式 | 功能 |
| LB 指令 | LB rd,rs1,imm | M[rs1 + imm] -> rd（读出1个字节） |
| LH 指令 | LH rd,rs1,imm | M[rs1 + imm] -> rd（读出2个字节） |
| LW 指令 | LW rd,rs1,imm | M[rs1 + imm] -> rd（读出4个字节） |
| LBU 指令 | LBU rd,rs1,imm | 取无符号1字节 |
| LHU 指令 | LHU rd,rs1,imm | 取无符号2字节 |
| ADDI 指令 | ADDI rd,rs1,imm | rs1 + imm -> rd |
| XORI 指令 | XORI rd,rs1,imm | rs1 ^ imm -> rd |
| ANDI 指令 | ANDI rd,rs1,imm | rs1 & imm -> rd |
| ORI 指令 | ORI rd,rs1,imm | rs1 | imm -> rd |
| SLTI指令 | SLTI rd,rs1,imm | (有符号)rs1<imm时，rd=1 |
| SLTIU指令 | SLTIU rd,rs1,imm | (无符号)rs1<imm时，rd=1 |
| SLLI指令 | SLLI rd,rs1,imm | rs1 << imm -> rd (有符号) |
| SRLT指令 | SRLI rd,rs1,imm | rs1 >> imm -> rd (无符号) |
| SRAI指令 | SRAI rd,rs1,imm | rs1 >> imm -> rd (有符号) |
| JALR指令 | JALR rd,rs1,imm | rs1 + imm -> PC |



图3 I型指令类

### 2.2.3 S型指令

如表5所详尽描述的，是S型指令（Store-type Instruction）的基本格式，这一特定格式的指令集专注于内存存储（store）操作，是实现数据持久化和数据交换的重要机制之一。在S型指令集中，总共实现了3条精心设计的指令，每一条都专注于执行内存存储的基本功能，如将寄存器中的数据安全地写入到内存中的指定位置。

为了提升代码的可读性、可维护性和复用性，同时便于开发者在程序中灵活地运用这些S型指令，这些指令被封装在了一个名为Stype的类中。Stype类不仅准确地反映了S型指令的内在逻辑和特性，还通过其内部定义的成员变量（如指令的操作码、源寄存器rs1和rs2、立即数imm等）和成员函数（如执行指令的方法），为外部提供了直观且便捷的调用接口。

如图4所示，Stype类的实现图直观地展示了该类的结构设计和关键组件。通过这个图，我们可以清晰地看到Stype类是如何将S型指令的核心功能——即将寄存器rs1的值存储到内存地址M[rs2 + imm]所指向的单元中——封装在内的。这一过程涉及到了对寄存器值的读取、地址计算的执行以及内存写入操作的完成，所有这些步骤都被巧妙地隐藏在Stype类的调用接口之后，使得开发者在调用S型指令时无需关心其背后的复杂实现细节。

表5 S型指令的格式

|  |  |  |
| --- | --- | --- |
| 指令名称 | 格式 | 功能 |
| SB 指令 | SB rs1,rs2,imm | 将[7:0]位，存储到特定的地址单元 |
| SH 指令 | SH rs1,rs2,imm | 将低[15:0]位，存储到特定的地址单元 |
| SW 指令 | SW rs1,rs2,imm | 将低[31:0]位，存储到特定的地址单元 |



图4 S型指令类

### 2.2.4 B型指令

如表6所清晰展示的是B型指令（Branch-type Instruction）的基本格式，这一指令类型专门设计用于实现程序中的有条件跳转操作，是控制程序流程的关键组成部分。B型指令集共包含了6条指令，每一条都承载着特定的跳转逻辑，旨在根据指令执行前的条件判断结果，决定是否改变程序的执行路径。

为了增强代码的可读性、可维护性和复用性，同时方便开发者在程序中灵活地使用这些B型指令，这些指令被封装在了一个名为Btype的类中。Btype类不仅准确捕捉了B型指令的核心功能特性，即根据条件进行跳转，还通过内部定义的成员变量（如跳转地址、条件码等）和成员函数（如执行跳转的方法、设置跳转条件的方法等），为外部提供了直观且易于使用的调用接口。

如图5所示，Btype类的实现图直观地揭示了该类的内部结构设计和关键组件。通过这个图，我们可以清晰地看到Btype类是如何将B型指令的跳转逻辑封装在内的，以及它是如何提供简洁的调用接口来触发跳转操作的。

表6 B型指令的格式

|  |  |  |
| --- | --- | --- |
| 指令名称 | 格式 | 功能 |
| BEQ 指令 | BEQ rs1, rs2, imm | rs1和rs2相等，则跳转PC+imm |
| BNE 指令 | BNE rs1, rs2, imm | rs1和rs2不相等，则跳转PC+imm |
| BGE 指令 | BGE rs1, rs2, imm | rs1大于等于rs2，则跳转PC+imm |
| BLT 指令 | BLT rs1, rs2, imm | rs1小于rs2，则跳转PC+imm |
| BLTU 指令 | BLTU rs1, rs2, imm | 无符号rs1小于无符号rs2，则跳转PC+imm |
| BGEU 指令 | BGEU rs1, rs2, imm | 无符号rs1大于等于无符号rs2，则跳转PC+imm |



图5 B型指令类

### 2.2.5 U型指令

如表7所详尽列出的是U型指令（U-type Instruction）的基本格式，这一指令类型专注于实现长立即数操作，为程序提供了处理较大数值或地址的直接手段。在U型指令集中，总共实现了两条指令，这两条指令各自承载着特定的功能，旨在满足程序中对于长立即数处理的需求。

如图6所示，Utype类的实现图清晰地展示了该类的结构设计和关键组件。通过这个图，我们可以直观地看到Utype类是如何将U型指令的长立即数操作逻辑封装在内的，以及它是如何提供便捷的调用接口来执行这些操作的。这种封装方式使得开发者在需要处理长立即数时，能够直接通过Utype类来完成，而无需深入了解底层指令的复杂性和实现细节。

表7 U型指令的格式

|  |  |  |
| --- | --- | --- |
| 指令名称 | 格式 | 功能 |
| LUI 指令 | LUI rd, imm | 一个20位的立即数加载到寄存器rd的高20位，低12位为0 |
| AUIPC 指令 | AUIPC rd, imm | 一个20位立即数加到PC的高20位，生成一个32位地址 |

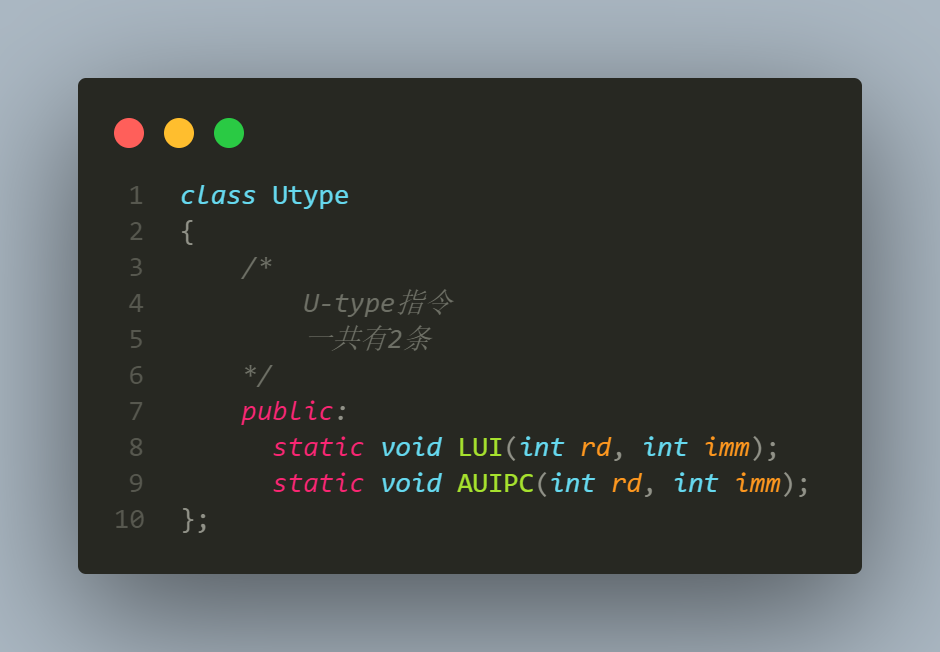


图6 U型指令类

该指令作用在于存取全局变量的地址，如图7所示，由于是32位机，若PC为20位，那么可以通过使用AUIPC指令进行数据左移，将该20位数据放入高20位。

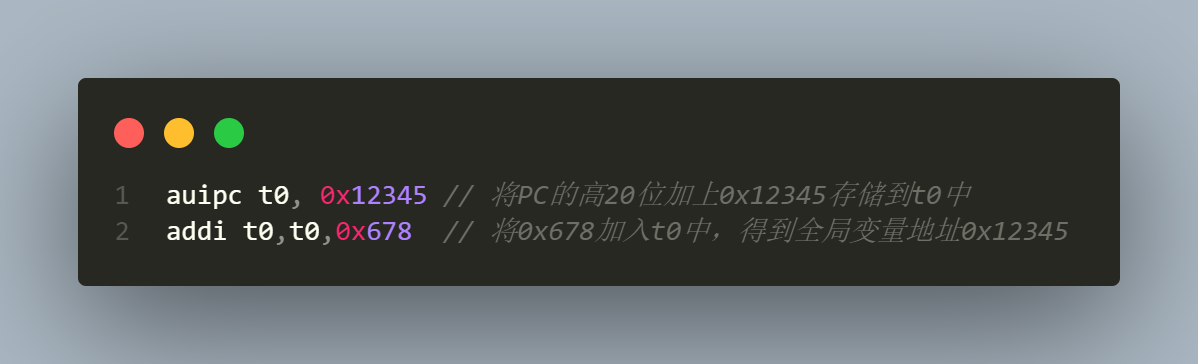


图7 AUIPC举例

### 2.2.6 J型指令

如表8所详尽描述的是J型指令（Jump-type Instruction）的基本格式，这一指令类型专注于实现无条件跳转操作，是控制程序流程中至关重要的一环。在J型指令集中，尽管只实现了一条指令，但这条指令却承载着将程序执行流无条件转移到指定地址的重要功能，对于实现函数调用、循环控制以及异常处理等编程结构具有不可替代的作用。

如图8所示，Jtype类的实现图直观地展示了该类的结构设计和关键组件。通过这个图，我们可以清晰地看到Jtype类是如何将J型指令的无条件跳转逻辑封装在内的，以及它是如何为开发者提供简便的调用接口来触发跳转操作的。

表8 J型指令的格式

|  |  |  |
| --- | --- | --- |
| 指令名称 | 格式 | 功能 |
| JAL指令 | JAL rd, imm | 将PC的值紧随其后的那条指令地址，存入rd寄存器中，然后将PC=PC+imm |

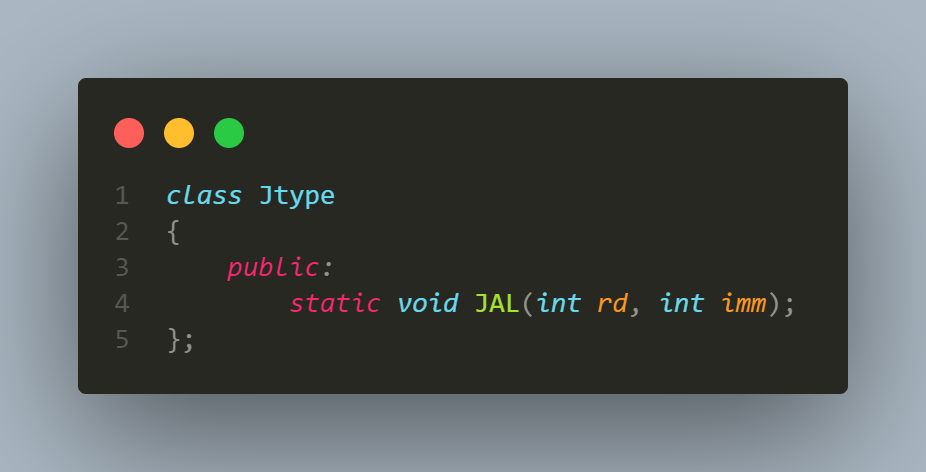


图8 J型指令类

## **2.3** 代码数据处理

### 2.3.1 指令文件读取和处理

首先，文件打开与准备。用户需要提供一个文件路径，该路径指向包含RISC-V指令集的文件。使用C++的ifstream（输入文件流）对象来打开并读取该文件。检查文件是否成功打开，如果未成功，则输出错误信息并退出程序。

然后，逐行读取与处理。通过文件流逐行读取文件内容，直到文件末尾。对于每一行，首先使用std::stringstream或std::istringstream结合std::getline（配合适当的条件判断）来去除前导空格和忽略完全由空格或制表符组成的空行。

紧接着，指令解析与分类处理。

1. 正常指令：如果当前行不包含注释或标号，直接将其视为一条正常指令，并存储到指令数组中。同时，更新PC值，每次增加4（因为假定每条指令占用4字节）。
2. 带注释的指令：如果当前行包含“#”标识的注释，则首先定位到“#”字符，然后检查“#”之后是否还有有效的指令部分。如果有，则截取并存储这部分指令到数组中，并更新PC值。如果“#”之后无有效指令，则忽略该行。
3. 标号与指令不在同一行：若当前行仅包含标号（假设标号格式明确，如以特定字符或模式开始），则读取该标号，并将当前的PC值（在读取下一行之前）作为该标号的地址存入标号数组中。PC值继续递增以准备下一条指令的读取。
4. 标号与指令在同一行：如果当前行同时包含标号和指令，则通过定位“:”字符来分隔标号和指令。首先读取并存储标号及其对应的PC值，然后截取并存储“:”之后的指令到数组中。之后，同样更新PC值。

最后，PC值的管理。在整个处理过程中，PC值（程序计数器）应持续跟踪当前指令的内存地址。由于是32位计算机，每条指令占用4字节，因此每次成功处理一条指令（无论是否带注释或标号）后，PC值应增加4。

### 2.3.2 模拟器初始化

1. RISC-V 32个通用寄存器管理：RISC-V架构支持32个通用寄存器，这些寄存器通过顺序表这一高效的数据结构进行管理，确保了它们在内存中的连续存储，便于快速访问。在程序初始化阶段，所有寄存器均被赋予初始值0，这一策略简化了寄存器的初始化流程，为后续操作提供了统一的起点。
2. 寄存器名到编号的映射机制：为了提升代码的可读性和维护性，我们采用了哈希表（具体为unordered\_map）来建立寄存器名称（遵循ABI规范）与其内部编号之间的映射关系。这种设计允许开发者在编写代码时直接使用寄存器名称进行访问，而无需记忆复杂的编号，极大地提高了开发效率。
3. 编号到寄存器名的反向映射：与上述映射机制相对应，我们还实现了从寄存器编号到其名称的反向映射，同样采用unordered\_map数据结构实现。这一功能在调试和结果输出阶段尤为重要，它允许我们根据寄存器的编号快速查找到对应的名称，便于问题定位和结果展示。
4. 存储器管理机制：为了高效地管理内存中的数据，我们选择了哈希表（具体为map）作为存储结构。这种选择基于其支持按键（即内存地址）排序的特性，使得我们可以实现从低地址到高地址的顺序存储，便于数据的连续访问和管理。同时，我们采用小端模式进行数据存储，即数据的低位字节存放在内存的低地址端，这一模式与当前多数计算机系统的硬件实现相符，有助于减少数据转换的开销。
5. 标号管理机制：在处理条件跳转等程序控制流时，标号的快速定位变得尤为重要。因此，我们采用unordered\_map来管理标号，将标号名称映射到其在程序中的具体位置（如指令地址）。这种设计使得在执行条件跳转时，程序能够迅速根据标号名称找到目标位置，极大地提高了程序执行的效率。同时，unordered\_map提供的快速查找能力，也确保了即便在大型程序中，标号的定位也能保持高效。

### 2.3.3 运行指令

通过使用PC指向下一条需要执行的指令，六种不同的类型采用六种不同的分析方法，由于不同类型对应的格式不同，因此需要分类分析。每一条指令，无论什么类型都是操作码和操作数组成，对于操作码，本模拟器对操作码的大小写不敏感，即大小写都可以作为操作码。如图9所示即为R型指令的的分析过程，取出每一条指令的操作数，调用类的函数最终获得想要的结果。

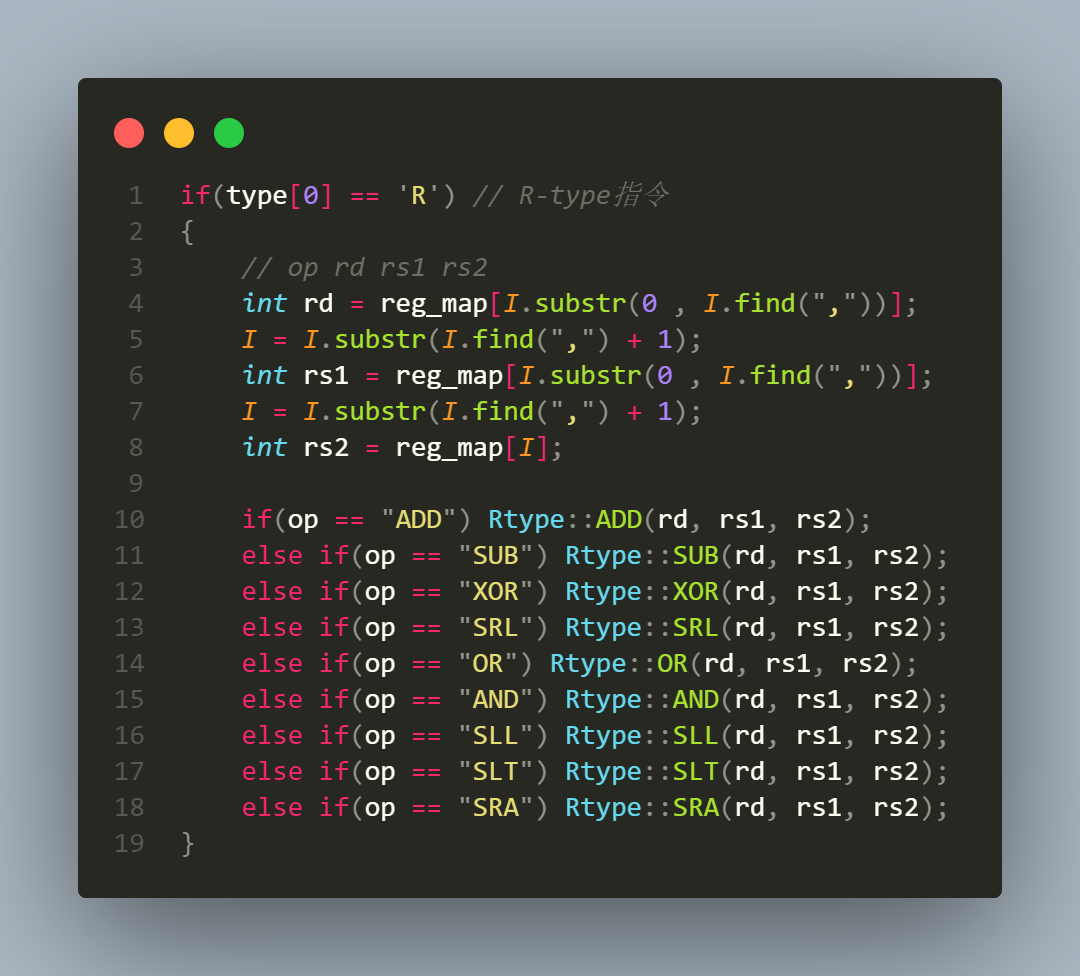


图9 R型指令

## **2.4** 程序测试

### 2.4.1 计算类指令测试

三、实验总结及心得体会

学生评价-问卷调查（学生填写）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | 完全同意 | 同意 | 基本同意 | 不同意 | 完全不同意 |
| 课程目标1 | 培养学生系统设计能力，使学生掌握基本计算机的设计能力，重点掌握CPU的设计。培养学生独立完成计算机各个模块的设计，掌握各模块间的关系及作用。培养学生对数据流、地址流、控制流的设计能力，总线形式的选择及设计能力。 |  |  |  |  |  |
| 课程目标2 | 培养学生各类工程设计能力，包括逻辑电路设计能力、硬件描述语言的程序设计能力，以及电路图与程序语言配合的设计方法等。在QuartusⅡ环境下，完成模型机的逻辑设计，该模型机需符合RISC机型的特点并能够正常工作。 |  |  |  |  |  |
| 课程目标3 | 培养学生掌握多种工程测试手段，包括仿真指令测试、仿真程序测试、以及实验仪上脱机程序测试的方法，进行整机调试，以验证模型机设计的完整性。培养学生独立思考、查找资料、设计、修改、调试的能力；不设上限的开放性设计目标，促使学生不断完善和提高。 |  |  |  |  |  |