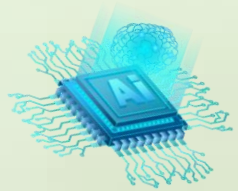
第3章 指令系统与汇编语言基本语法



本章给出RISC-V架构的基本指令系统及汇编语言基本语法，通过汇编环境了解指令对应的机器码，直观的基本理解助记符与机器指令的对应关系。基本掌握任何一种CPU的指令系统，当遇到新的CPU时就不会感到陌生，其本质不变。学习指令系统的基本方法是：理解寻址方式、记住几个简单指令、利用汇编语言编程练习。

3.1 RISC-V架构概述（了解）

RISC-V是一个基于精简指令集计算机原则而开源的指令集架构，随着RISC-V生态系统的发展，它将在微型计算机领域占有重要地位。要了解RISC-V架构，需要先了解一下芯片架构的基本含义。

3.1.1 RISC与ISA名词解释

1. 精简指令集计算机RISC

精简指令集计算机（Reduced Instruction Set Computer，RISC）的特点是指令数目少、格式一致、执行周期一致、执行时间短，采用流水线技术等。

这种设计模式的技术背景是：CPU实现复杂指令功能的目的是让用户代码更加便捷，但复杂指令通常需要几个指令周期才能实现，且实际使用较少；此外，处理器和主存之间运行速度的差别也变得越来越大。

RISC是对比于复杂指令计算机（Complex Instruction Set Computer，CISC）而言的，可以粗略地认为，RISC只保留了CISC常用的指令，并进行了设计优化，更适合设计嵌入式处理器。

2. 指令集架构ISA

所谓**指令集架构（Instruction Set Architecture，ISA）**是与程序设计相关的计算机架构部分，包括数据类型、指令、寄存器、地址模式、内存架构、外部I/O、中断和异常处理等。常见的指令集架构主要有x86架构、ARM架构、RISC-V架构等。

3.1.2 RISC-V简介

1. RISC-V的由来

RISC-V的读音为risk-five，它是由美国加州大学伯克利分校于2010年推出。其初衷是为了打破ARM、英特尔等公司在指令集架构领域内的垄断，经过多年的发展，RISC-V已经得到了业界的高度重视。

2. RISC-V在中国的发展

2017年进入中国，目前国产RISC-V架构芯片型号众多，是微处理器与微控制器芯片产业的重要发展方向之一。

3. RISC-V与X86、ARM架构的简明比较

表2-1 RISC-V与X86、ARM架构的简明比较

比较指标	X86	ARM	RISC-V
指令集类型	CISC	RISC	RISC
寄存器宽度	32、64	32、64	32、64
源码	不开源	不开源	开源
用户可控性	难以满足需求	现阶段满足需求，未来存在变数	可望满足需求
生态系统	比较成熟	比较成熟	逐步发展
授权费用	缺乏成熟的授权模式	架构授权费用高	无

3.2 寄存器与寻址方式 （重点）

CPU内部必须有一些存放数据的地方，犹如家庭中具有客厅、厨房、餐厅、卧室、书房、卫生间等不同房间，CPU内部也必须有一些特殊用途的寄存器，以便满足不同功能要求。不同架构的CPU，其内部的寄存器名称、功能稍有差异，但具有一定的共性。

CPU内部的寄存器是其内部数据暂存的地方，数量一般不会很多，每个寄存器都有自己的名字，有的具备特殊功能。寻址方式是指汇编程序的一条指令中操作数在哪里。

本节给出寄存器通用基础知识、RISC-V架构主要寄存器、指令保留字简表与寻址方式，还给出如何能知道一条汇编指令的机器码。

3.2.1 寄存器基础知识及相关基本概念（共性）

以程序员视角，从底层学习一个CPU，理解其内部寄存器用途是重要一环。计算机所有指令运行均由CPU完成，CPU内部寄存器负责信息暂存，其数量与处理能力直接影响CPU的性能。对CPU内部寄存器的操作与对内存的操作不同之处在于，使用汇编语言编程时，对CPU内部寄存器的访问直接使用寄存器的名字，访问不需要经过地址、数据、控制三总线，而对内存的访问涉及地址单元，需要经过三总线，因此对寄存器的访问比对内存的访问速度快。

从共性知识角度及功能来看，CPU内至少应该有数据缓冲类寄存器、栈指针类寄存器、程序指针类寄存器、程序状态类寄存器及其他功能寄存器。

本小节先从一般意义上阐述寄存器基础知识，下一小节给出RISC-V架构的主要寄存器。

1．数据缓冲类寄存器

CPU内数量最多的寄存器是用于数据缓冲的寄存器，一些芯片的寄存器名称用Register的首字母加数字组成，如R0、R1、R2等等，不同CPU其种类不同。例如Intel X86系列的通用寄存器主要有EAX，EBX，ECX，EDX，ESP，EBP，ESI，EDI等，ARM系列的通用寄存器主要有R0～R12。RISC-V系列的通用寄存器主要有X0～X31。有时这些通用寄存器采用统一编号，可以用作特殊功能。

2．栈指针类寄存器 （难点）

编程中，有全局变量与局部变量的概念。从存储器角度看，全局变量具有固定的地址，每次读写都是那个地址。而在一个子程序中开辟的局部变量不是，用RAM中的哪个地址不是固定的，采用“后进先出（Last In First Out，LIFO）”原则使用一段RAM区域，这段RAM区域被称为栈区 。它有个栈底的地址，是一开始就确定的，此时栈顶与栈底是重合的，当有数据进栈时，地址自动向一个方向变动，栈顶与栈底就分开了 ，不然数据就放到同一个存储地址中了，CPU中需要有个地方保存这个不断变化的地址，这就是栈指针（Stack Pointer）寄存器，简称SP。至于数据进栈地址增加还是减小，取决于芯片的设计，但无论如何，出栈与进栈的地址变动方向总是相反。SP的内容是下一个进栈数据的存储地址，或是下一个出栈数据的访问地址。

这里的栈，其英文单词为Stack，在单片微型计算机中基本含义是RAM中存放临时变量的一段区域。现实生活中，Stack的原意是指临时叠放货物的地方，但是叠放的方法是一个一个码起来的，最后放好的货物，必须先取下来，先放的货物才能取，否则无法取。在计算机科学的数据结构学科中，栈是允许在同一端进行插入和删除操作的特殊线性表。允许进行插入和删除操作的一端称为栈顶（top），另一端为栈底（bottom）；栈底固定，而栈顶浮动；栈中元素个数为零时称为空栈。插入一般称为进栈（PUSH），删除则称为出栈（POP）。栈也称为后进先出表。

3. 程序指针类寄存器

计算机的程序存储在存储器中，CPU中有个寄存器指示将要执行的指令在存储器中位置，这就是程序指针类寄存器。在许多CPU中，它的名字叫做程序计数寄存器（Program Counter，PC）。

4. 程序运行状态类寄存器

CPU在进行计算过程中，会出现诸如进位、借位、结果为0、溢出等等情况，CPU内需要有个地方把它们保存下来，以便下一条指令结合这些情况进行处理，这类寄存器就是程序运行状态类寄存器。常用单个英文字母表示其含义，例如，N表示有符号运算中结果为负（Negative）、Z表示结果为零（Zero）、C表示有进位（Carry）、V表示溢出（Overflow）等。

5. 其他功能寄存器

不同CPU中，除了具有数据缓冲、栈指针、程序指针、程序运行状态类等寄存器之外，还有表示浮点数运算、中断屏蔽 等寄存器。

所谓中断屏蔽，就是中断进来也不理它。中断是暂停当前正在执行的程序，先去执行一段更加紧急程序的一种技术，它是计算机中的一个重要概念，将在第8章较为详细的阐述。中断屏蔽标志，就是表示是否允许某种中断进来的标志。

3.2.2 RISC-V架构主要寄存器 （个性）（难点）

这是一个具体的CPU架构，按照从一般到个别的哲学原理，我们来认识一个具体CPU的内部寄存器，了解其功能，随后学习指令系统将与它们打交道，第4章开始的汇编语言编程也是直接与它们打交道。

RISC-V 处理器架构中，寄存器主要有通用寄存器（General Purpose Register，GPR）、控制和状态寄存器（Control and Status Register，CSR）、特殊功能寄存器等。这些寄存器为程序执行、硬件状态管理、异常处理等提供支持。

表3-2　RISC-V架构的主要通用整数寄存器

寄存器名	ABI 接口名称	英文描述	中文描述
x0	zero	Hardwired zero	常数 0
x1	ra	Return address	返回地址
x2	sp	Stack pointer	堆栈指针
x3	gp	Global pointer	全局指针
x4	tp	Thread pointer	线程指针
x5～x7	t0～t2	Temporary	临时寄存器
x8	s0/fp	Saved register，frame pointer	保存寄存器或帧指针
x9	s1	Saved register	保存寄存器
x10～x11	a0～a1	Function argument，return value	函数参数或返回值
x12～x17	a2～a7	Function argument	函数参数
x18～x27	s2～s11	Saved register	保存寄存器
x28～x31	t3～t6	Temporary	临时寄存器

RISC-V架构包含32个通用整数寄存器（x0～x31），有的芯片x0被预留为常数0，有的芯片x0作为程序计数器PC，其他为普通的通用整数寄存器，如表2-2所示。其中有七个临时寄存器（t0～t6），用于存放函数参数；十二个保存寄存器（s0～s11），四个指针类寄存器（堆栈指针sp、全局指针gp、线程指针tp、帧指针fp），两个函数参数或返回值寄存器（a0～a1），六个函数参数寄存器（a2～a7）及返回地址寄存器ra等。

临时寄存器是指在函数调用过程中不保留这部分寄存器存储的值，与之对应的是保存寄存器，则在函数调用过程中保留这部分寄存器存储的值，这样可以减少保存和恢复寄存器的次数。全局指针gp优化对±2KB内全局变量的访问，线程指针tp优化对±2KB内线程局部变量的访问，主要用于操作系统中。fp和sp可以确定当前函数使用的栈空间。

表中ABI是Application Binary Interface（应用程序二进制接口）的缩写，表示汇编编程时使用的名称。

3.2.3 指令保留字简表与寻址方式 （重点）

CPU的功能是从外部设备获得数据，通过加工、处理，再把处理结果送到CPU的外部世界。设计一个CPU，首先需要设计一套可以执行特定功能的操作命令，这种操作命令称为指令。CPU所能执行的各种指令的集合，称为该CPU的指令系统。

RISC-V的指令集使用模块化的方式进行组织，每一个模块使用一个英文字母来表示。RISC-V最基本也是唯一强制要求实现的指令集部分是由I字母表示的基本整数指令子集。

表3-3　RISC-V的模块化指令集

基本/扩展	类　型	指令数	描　述
基本指令集	RV32I	47	32 位地址空间与整数指令，支持 32 个通用整数寄存器
	RV32E	47	RV32I 的子集，仅支持 16 个通用整数寄存器
	RV64I	59	64 位地址空间与整数指令及一部分 32 位的整数指令
	RV128I	71	128 位地址空间与整数指令及一部分 64 位和 32 位的指令
扩展指令集	M	8	整数乘法与除法指令
	A	11	存储器原子操作指令，Load/Store 指令
	F	26	单精度（32 比特）浮点指令
	D	26	双精度（64 比特）浮点指令，必须支持 F 扩展指令
	C	46	压缩指令，指令长度为 16 位

1. 指令保留字简表 （理解+记忆）

表3-4 RISC-V基本保留字

类 型		保 留 字	含 义
数据传送类		auipc	生成与 PC 指针相关的地址
		la、lb、lh、li、lw、lhu、lbu	将存储器中的内容加载到寄存器中
		sb、sw、sh、mv	将寄存器中的内容存储到存储器中
		lui	将立即数存储到寄存器中
数据操作类	算术运算类	add、addi、sub、mul、div	加、减、乘、除指令
		slt、slti、sltu、sltui	比较指令
	逻辑运算类	and、andi、or、ori、xor、xori	按位与、或、异或
	移位类	sra、srai、sll、sll、srl、srli	算术右移、逻辑左移、逻辑右移
	csr 类	csrrw、csrrs、csrrc、csrrwi、csrrsi、csrrci	用于读写 CSR 寄存器
跳转类	无条件类	jal、jalr	无条件跳转指令
	有条件类	beq、bne、blt、bltu、bge、bgeu	有条件跳转指令
其他指令		call、ret、fence、fengei、ecall、ebreak	调用指令、返回指令、存储器屏障指令、特殊指令

要求：

（1）记忆如取数、存数、算数运算、跳转等基本保留字；（2）在RISC-V手册中查找各个保留字（助记符）的英文来源

2. 寻址方式 （重点）

指令是对数据的操作，通常把指令中所要操作的数据称为操作数，CPU所需的操作数可能来自寄存器、指令代码、存储单元。而确定指令中所需操作数的各种方法称为寻址方式（addressing mode）。

1）立即数寻址：操作数直接通过指令给出

/*将立即数的低12位与rs1中整数相加，结果写会rd寄存器*/

addi rd，rs1，imm[11:0]

2）寄存器寻址：操作数来自寄存器

/*将寄存器rs1中整数值与rs2中整数值相加结果写回rd寄存器*/

add rd，rs1，rs2

3）偏移寻址及寄存器间接寻址：操作数来自存储单元

/*从地址rs1+offset[11:0]处读取32位数据写入rd*/

lw rd，offset[11:0](rs1)

3.2.4 机器码的获取方法 （实际操作）

1. 运行源文件

运行电子资源【03-Software\CH03\LST-ASM】，看看 li a0，#0xDE 语句生成的机器码是什么？

2. 执行程序获得的信息

表3-5 指令“li a0，0xde”的存储细节

地　址	00013552	00013553	00013554	00013555
内　容	13	05	E0	0D

3. 列表文件.lst中的信息（打开程序查看）

小端模式（little-endian）是指将两字节以上的一个数据的低字节放在存储器的低地址单元

4. 十六进制机器码文件.hex中的信息（打开程序查看）

可在其中找到任意一个语句机器码所在地址

3.3 RISC-V基本指令分类解析（按照科学方法记忆部分）

本节在前面给出指令简表与寻址方式的基础上，按照数据传送类、数据操作类、跳转类、CSR类、其他指令5个方面，简要阐述RV32I的29组基本指令的功能。

友情提示：按照助记符的来源及寻址方式记住几组常用指令，然后在此基础上，编写2～3个汇编工程，是学习指令系统的基本方法。

3.3.1 数据传送类指令

数据传送类指令的功能有两种情况，一是取存储器地址空间中的数传送到寄存器中，二是将寄存器中的数传送到另一寄存器或存储器地址空间中。

1. 取数指令

（1）la rd, symbol　　//l=load（取）， a=address（地址）
（实例运行）

（2）li rd, imm　　//l=load（取）， i= immediate （立即数）
（课堂练习）

指令（助记符）的英文来源参见：RISC-V手册

（3）取（偏移+寄存器值）为地址的内容（有符号数）到rd中

lb rd，offset[11:0](rs1)　　//b=byte（字节）

lh rd，offset[11:0](rs1)　　//h=halfword（半字）

lw rd，offset[11:0](rs1)　　//w=word（字）

offset[11:0]：12位，地址偏移范围-2048～2047

（4）取（偏移+寄存器值）为地址的内容（无符号数）到rd中

lbu rd，offset[11:0](rs1)　　//u=unsigned（无符号）

lhu rd，offset[11:0](rs1)

至此，取数指令学习完成，共4组，共7个具体指令，后续指令举一反三地学习

2. 存数指令

（5）寄存器中的内容存储（store）至存储器中

sb rs2，offset[11:0](rs1)　　//s=store

sh rs2，offset[11:0](rs1)

sw rs2，offset[11:0](rs1)

（6）//把寄存器rs1复制到寄存器rd中，实际被扩展为addi rd, rs1, 0

mv rd，rs1　　//mv=move （伪指令）

3. 生成与指针PC相关地址指令　（了解）（难点）

PC加立即数（Add Upper Immediate to PC）指令

（7）auipc rd，imm　//a=add（加），u=upper（高位），i=immediate，pc即程序计数器，把符号位扩展的20位（左移12位）立即数加到PC上，结果写入寄存器rd中。

目的是可以访问任何32 位PC 相对地址的数据。

4. lui指令 （了解）（难点）

高位立即数加载（Load Upper Immediate）指令

（8）lui rd，imm //l=load（取），u=upper（高位），i=immediate，将 20 位无符号数加载到寄存器的高 20 位

示例：如何将32为立即数0x12345678赋给寄存器？

```
lui  t1,0x12345          //将0x12345左移12位，低12位补0，赋给t1
addi  t1,t1,0x678        //t1 = t1 + 低12位立即数
```

思考：如何将2050（超出2047）赋给t1？

```
lui  t1, 0x1             //t1 = 0x1 << 12 = 0x1000（即 4096）
addi t1, t1, -2046       //t1 = 4096 - 2046 = 2050
```

3.3.2 数据操作类指令

数据操作主要指算术运算、逻辑运算、移位等。

1. 算术运算类指令

（9）加

```
        add  rd，rs1，rs2
        addi rd，rs1，imm[11:0]
（10）减  sub  rd，rs1，rs2
（11）乘  mul  rd，rs1，rs2
（12）除  div  rd，rs1，rs2
```

（13）比较

```
slt  rd，rs1，rs2              //小于则置位（Set if Less Than）
slti  rd，rs1，imm[11:0]       //小于立即数则置位（Set if Less Than Immediate）
sltu  rd，rs1，rs2             //无符号小于则置位(Set if Less Than, Unsigned)
sltiu  rd，rs1，imm[11:0]
```

2. 逻辑运算类指令

（14）与

```
and  rd，rs1，rs2              //rd = rs1 & rs2
andi  rd，rs1，imm[11:0]       //rd = rs1 & 有符号位的12位数
```

（15）或

```
or  rd，rs1，rs2               //rd = rs1 | rs2
ori  rd，rs1，imm[11:0]        //rd = rs1 | 有符号位的12位数
```

（16）异或

```
xor  rd，rs1，rs2                  //rd = rs1 ^ rs2
xori  rd，rs1，imm[11:0]           // rd = rs1 ^ 12位有符号的立即数
```

3．移位运算

（17）算法右移

```
sra  rd，rs1，rs2                  //rd = rs1 >> rs2（Shift Right Arithmetic）
sra  rd，rs1，shamt[4:0]           //rd = rs1 >> 移动长度（ Shift Amount）
```

（18）逻辑左移

```
sll  rd，rs1，rs2                  //rd = rs1 << rs2（Shift Left Logical）
sll  rd，rs1，shamt[4:0]           // rd = rs1 << 移动长度
```

（19）逻辑右移

```
srl  rd，rs1，rs2                  //rd = rs1 >> rs2 （Shift Right Logical）
srl  rd，rs1，shamt[4:0]           //rd = rs1 >> 移动长度
```

3.3.3 跳转类指令

跳转指令主要指无条件跳转与有条件跳转指令。

1. 无条件跳转指令

（20）（j也是跳转指令）

jal rd，offset //跳转到offset处，返回地址保存到rd处（Jump and Link）

例：jal ra, my_function //调用 my_function，返回地址保存到 ra

jalr rd，offset(rs1) //（Jump and Link Register）

例：jalr ra, 8(t0) // 跳转到 t0 + 8 的地址，返回地址保存到 ra

2. 有条件跳转指令

（21）

//相等和不相等的跳转

beq rs1，rs2，offset　　//相等跳转（Branch if Equal）

bne rs1，rs2，offset　　//不等跳转 （Branch if Not Equal）

//小于跳转

blt rs1，rs2，offset　　//小于跳转（Branch if Less Than），有符号

bltu rs1，rs2，offset　　//小于跳转（Branch if Less Than, Unsigned）

//大于等于跳转，英文全称自行查手册

bge rs1，rs2，offset　　//大于等于跳转 (Branch if Greater Than or Equal)

bgeu rs1，rs2，offset

注意：这里的offset范围是-4KB ~ +4KB

3.3.4 控制及状态寄存器类指令

控制及状态寄存器类指令访问的是专用的控制及状态寄存器

（22）（自行查手册）

```
csrrw  rd，csr，rs1
csrrs  rd，csr，rs1
csrrc  rd，csr，rs1
csrrwi  rd，csr，imm[4:0]
csrrsi  x0，csr，imm[4:0]
csrrci  rd，csr，imm[4:0]
```

3.3.5 其他指令

未列入数据传输类、数据操作类、跳转类、csr类四大类的指令，归为其他指令，大部分了解，要求完全理解call、ret、nop指令。

调用指令	（23）	call rd, symbol
		call
返回指令	（24）	ret
空操作指令	（25）	nop
存储器屏障指令	（26）	fence pred，succ
	（27）	fence.i
特殊指令	（28）	ecall
	（29）	ebreak

3.4 汇编语言的基本语法

能够在MCU内直接执行的指令序列是机器语言，用助记符号来表示机器指令便于记忆，这就形成了汇编语言。因此，用汇编语言写成的程序不能直接放入MCU的程序存储器中去执行，必须先转为机器语言。把用汇编语言写成的源程序“翻译”成机器语言的工具叫汇编程序或编译器（assembler），以下统一称为汇编器。

3.4.1 汇编语言格式

汇编语言源程序可以用通用的文本编辑软件编辑，以ASCII码形式保存。编译器除了识别MCU的指令系统外，为了能够正确地产生目标代码以及方便汇编语言的编写，编译器还提供了一些在汇编时使用的命令、操作符号。由于编译器提供的指令仅是为了更好地做好“翻译”工作，并不产生具体的机器指令，因此这些指令被称为伪指令（pseudo instruction）。

以行为单位的汇编语言格式：

标号： 操作码 操作数 注释

1. 标号

被编译后，标号就是一个地址

2. 操作码

操作码包括指令码和伪指令和用户自定义宏。 编译器不区分操作码中字母的大小写。

3. 操作数

操作数可以是地址、标号或指令码定义的常数，也可以是由伪运算符构成的表达式。（1）常数标识。 十进制（默认不需要前缀)、十六进制（0x前缀）、二进制（0b前缀）。（2）圆点“.”代表当前程序计数器的值。

4. 注释

注释是说明文字，汇编语言注释有多种，根据不同编译器有#、//、/**/等。

3.4.2 常用伪指令简介

不同集成开发环境下的伪指令稍有不同，伪指令书写格式与所使用的开发环境有关。伪指令主要有用于常量以及宏的定义、条件判断、文件包含等。

1. 系统预定义的段

通常划分为如下3个段：.text、.data和.bss，其中，.text是只读的代码区；.data是可读可写的数据区，.bss是可读可写且没有初始化的数据区。

```
.text      /*表明以下代码在.text段*/
.data      /*表明以下代码在.data段*/
.bss       /*表明以下代码在.bss段*/
```

2. 常量的定义

使用常量定义，能够提高程序代码的可读性，并且使代码维护更加简单。常量的定义可以使用.equ汇编指令，下面是GNU编译器的一个常量定义的例子：

```
.equ MAINLOOP_COUNT,600000      //类似于C语言define定义
 li t2,MAINLOOP_COUNT           //t2 = MAINLOOP_COUNT表示的值
```

3. 程序中插入常量

表3-17　用于程序中插入不同类型常量的常用伪指令

插入数据的类型	GNU 编译器
字	.word（如.word 0x12345678）
半字	.half（如.word 0x1234）
字节	.byte（如.byte 0x12）
字符串	.string（如.string “hello\n”，只是生成的字符串以‘\0’结尾）

4. 条件伪指令

.if条件伪指令后面紧跟一个恒定的表达式（即该表达式的值为真），并且最后要以.endif结尾。中间如果有其他条件，可以用.else填写汇编语句。

.ifdef标号表示如果标号被定义，则执行下面的代码

5. 文件包含伪指令

.include “filename”

利用它可以把另一个源文件插入当前的源文件一起汇编，成为一个完整的源程序

6. 其他常用伪指令 （实际编程时参考例子）

本章作业：2、3、6、7、8