第1章. 程序设计语言认知实验

1.1程序设计语言

1.1.1. 程序设计语言发展历史

程序设计语言是人机交互的重要接口,程序设计人员使用程序设计语言将要执行的操作编码在一个程序中,经过编译器的翻译后传递给计算机系统进行执行。自从计算机诞生以来，程序设计语言一直在发展变化，即便在几十年后的今天，生命力较旺盛的语言仍然在不断的改进,以期不断提高编程效率和程序执行效率,或者满足其他方面的使用需求。例如C语言经过几十年的发展，已经形成了多个不同的版本。在多核不断发展和普及之后，原有基于库的并行程序设计方法存在诸多问题，人们在考虑是否可以从语言层面对并行程序设计进行更好的支撑，以减轻对编译器和程序设计人员带来的困扰，于是各种语言又进一步改进，也出现了不少新的语言。

目前，已经有上百种程序设计语言在广泛使用，有些语言是传统的语言，生命力较强，已经发展了许多年(例如C); 而有些语言则比较年轻，具有较大的开源社区，正在不断地快速发展和演化（例如 Python 和 GO)。按照不同的分类标准，可以将现有的语言划分为多个不同的类别，例如结构化程序设计语言和面向对象程序设计语言,脚本语言,静态强类型语言和动态强类型语言,函数式语言,命令式和声明式语言等等。这些语言各有特色，例如C语言运行效率较高，在系统领域发挥了重要作用; 而 Java 语言由于较好的跨平台特性，在应用开发上占有重要地位; Python语言编程效率较高，随着云计算、大数据、人工智能和物联网技术的快速发展和应用，越来越受到大家的欢迎; 函数式编程语言具有良好的定义，所以能够支持更好的并行和优化。

受到教育背景、工作环境和个人喜好等各方面的影响，每个程序设计人员所使用的编程语言不尽相同。特别是在学习阶段，受制于教学学时，接触的编程语言比较有限，但是这不应该成为大家了解语言多样性的一个障碍。应该让大家尝试使用更多的程序设计语言，了解每种语言各自的历史和特点，并学会根据需求

和实际情况选择相应的程序设计语言。本实验正是基于上述考虑而设计的。将该实验安排在编译原理课程的引论阶段，主要是基于以下几点考虑:

1. 高级程序设计语言编写的程序是编译器的输入，了解高级程序设计语言的特性，有助于学生更好的了解编译器的设计需求;
2. 语言的特性和语言的实现密切相关，多语言认知实验有助于学生对编译执行、解释执行以及混合执行的理解;
3. 多语言认知实验中包括了多个高级程序设计语言，还包括了一个汇编语言，并要求学生实现同一个功能，借此可以让学生对应的看到一个程序的高级语言版本和汇编语言版本，更清晰的认识到编译器的输入和输出;
4. 了解到语言在编程效率和运行效率之间的差异性，为语言实现和运行时系统理解奠定基础。

1.1.2. C 语言简介

C 语言是一个通用的命令式计算机程序设计语言，属于静态强类型，支持结构化程序设计，是操作系统、数据库和嵌入式等系统级软件开发经常使用的编程语言。该语言最初是由 Dennis Ritchie 和 Ken Thompson 等人于 1969 年到 1973 年在贝尔实验室发明的，当时主要用来实现 Unix 操作系统。PDP-11 计算机上最初的 Unix 使用汇编语言实现，开发者本来打算使用 B 语言的一个简化版本重写 Unix，但是考虑到 B 语言本身的一些限制，于是就产生了新的语言 C，意为 B 的下一代语言。1972 年，Unix 的大部分用 C 改写完成，到 1973 年，随着 struct 类型的出现，C 语言变得更加强大，Unix 的绝大部分代码也被改写为 C 程序。从那以后成为最流行的程序设计语言之一,C 语言被美国国家标准研究所 American National Standards Institute (ANSI)从 1989 年标准化,后来被 International Organization for Standardization (ISO)认可接受。1999 年对 C 语言进行修订，形成 C99 标准，支持 inline 函数，long long int 等新的数据类型，改进 IEEE 754 浮点数支持，单行注释 “//”。接着在 2011 年和 2018 年，分别对 C 有了两次持续的修订和改进，特别是 2011 版将原子操作和多线程加入到语言标准中，能够更好的支持并发程序的设计与实现。

C 语言是一个命令式面向过程的编程语言，对应的高级语言程序很容易被翻

译到机器语言执行，相对其他语言执行效率较高，目前各大厂商和平台都有 C 语言编译器的支持。后续的其他语言都或多或少借鉴了 C 语言的设计思想，例如 C++、C#、Java、Objective-C 等。

1.1.3. Java 简介

Java 语言是由 Sun Microsystems 公司的 James Gosling 及其同事在 1990 年左右设计开发的。不像其他语言要么编译为机器码执行，要么直接从源码开始解释执行，Java 程序首先被翻译为字节码，然后在 Java 虚拟机 JVM 中解释执行，目前通常使用 JIT 技术提高程序运行效率[1]。

Java 是一个面向对象程序设计语言，是由 Sun Microsystems 公司的 James Gosling 和他的项目组（Green Team）在 1995 年发布的[5]，后来 Sun Microsystems 被 Oracle 收购，Java 及其相关平台和工具也转由 Oracle 开发维护。Java 继承了 C 和 C++的很多语法，但是抛弃了较多的低层次的内容，例如指针和内存操作等。Java 中提供了垃圾收集器，编写程序时不需要再手动删除创建的对象。

图 1 Green Team 合影 [4]

1991 年 James Gosling 启动了一个名为“Oak”的项目，这个项目的目标是实现一个虚拟机并设计一个与 C 类似但比 C/C++简单的语言，最初是为了用于可交互电视的嵌入式应用。由于 Oak 商标已经被占用，所以改名为 Java，并于 1995 年发布了 1.0 版本。在讨论名字的时候，项目组成员提出了许多候选项，Java 是印度尼西亚的一个岛屿，那里出产了第一种咖啡（称为 java coffee），是一种浓缩咖啡豆。James Gosling 在办公室附近喝咖啡时选定了 Java 这个名字[3]。

由于其虚拟机免费，支持大多数主流平台，安全且可配置，并做到了“Write

Once， Run Anywhere”的承诺，很快得到了 Netscape、Oracle 和 Microsoft 等公司的支持。随后发布的 Java 2 中包括了 J2EE 和 J2SE，分别对应企业版和标准版。Sun 一直没有对 Java 标准版收费，但是在企业版取得了相应的回报。Sun 公司将 JDK（Software Development Kit）和 JRE（Runtime Environment）分开，二者的区别主要在于，JRE 中并没有包含 Java 编译器。

之后 Java 一直平稳发展[4]，直到 2006 年在 Javaone 大会上，Sun 公司宣布会将 Java 开源，同年 10 月再 Oracle OpenWorld 会议上，Jonathan Schwartz 称 Java 核心平台将在 30 天到 60 天内开源。2006 年 12 月，Sun 终于将 Java HotSpot 虚拟机和编译器开源,并使用了 GNU GPL 协议,并承诺除了个别组件外，JDK 的其他部分（主要是 Java 运行时）将于 2007 年 3 月同样按照 GPL 协议开源，少数组件不能开源主要是因为 Sun 没有权限将这些构件以 GPL 的形式开源。按照开源鼻祖 Richard Stallman 的说法，这将终止 Java 的开源之路[2]。

遵照之前的承诺,2007 年上半年 Sun 公司以 GPL 发布了 Java Class Library 的绝大部分代码，其中一些受限的部分是由第三方授权给 Sun 公司的，而 Sun 不能修改开源协议以 GPL 协议开源，其中就包括了 Java 图形用户接口 GUI。Sun 声称他们计划用其他实现替换这些专属模块，以实现类库的完全开源。

2007 年 5 月发布的时候，OpenJDK 的类库中有 4%是没有开源的，到 2008 年 5 月 OpenJDK 6 出现的时候，只有不到 1%的部分没有开源，但是这 1%涉及的部分（SNMP）并没有在 Java 规范中，因此基本上不需要其他的任何的二进制插件就可以构建出 OpenJDK。阻碍开源的每个组件要不自己变成了免费的开源软件，要不就被替换掉了。2010 年 12 月，JDK 终于实现了完全开源，这也是 Sun Microsystems 和 OpenJDK 社区长期共同努力的结果。

1.1.4. Python 简介

Python 是由 Guido van Rossum 设计并在 1991 年发布的一个高级语言，与其他语言的显著区别在于空格的作用不同，空格以及相应的缩进在 Python 中起到了非常重要的格式化作用，提高了代码的可读性。Python 是动态强类型语言，和 Java 一样支持自动内存管理。另外，该语言支持面向过程和面向对象编程模式，并且有非常丰富的库程序可以供大家使用。

Python 的解释器最早是使用 C 语言实现的，称之为 CPython，后来又有人基于 Java 实现了 Jython，微软在.NET 的基础上实现了 IronPython。CPython 能够调用 C 语言的各种库程序，Jython 可以调用 Java 语言库程序，IronPython 则能够调用 C#的各种库程序。

1.1.5. Haskell 简介

Haskell 是一个标准化的纯函数式编程语言，最新的 Haskell 标准是 2010 年发布的，从 2016 年 5 月份起，一个工作组在起草新的版本标准，预计 2020 年发布。Haskell 的主要特点是类型推断和延迟求值。

1987 年左右，大概有 10 多个函数式编程语言，广泛使用的是 Miranda，但是 Miranda 是一个商业公司支持的语言。在 1987 年召开的 FPCA 会议上，大家一致认为应该成立一个委员会定义一种开放标准的函数式编程语言，以供大家研究函数式编程语言使用。在该委员会的努力之下，Haskell 1.0 于 1990 年发布，此后又出现了 1.1 到 1.4 版本。1998 年和 2010 年 Haskell 有了重大的节点发布。

Haskell 最主要的特点是延迟求值，lambda 表达式，模式匹配，列表理解，类型类和类型多态等特征。Haskell 是一个纯函数式编程语言，意味着所有的函数都没有副作用，比较适合于并行执行。

Haskell 是静态强类型系统，有一个不断壮大的语言生态社区，已有 5400 多个第三方开源库和工具。Haskell 目前存在多个版本实现，其主流实现 Glasgow Haskell Compiler (GHC)即支持解释执行也支持翻译执行。

1.1.6. MIPS 汇编简介

指令集架构（Instruction-Set Architecture，ISA），一般指的是一个处理器支持的指令、指令的字节集编码及其他相关细节。

指令集是一组指令的集合，规定了一个处理器可以完成的操作及其执行结果。指令集架构则根据指令集的规定，向外延伸规定了更多的细节。这些细节包括但不限于:

- 指令类型与编码

- 系统状态
- 中断与异常处理
- IO 接口

有了指令集架构，硬件设计人员可以在规则下设计出不同的硬件实现；软件设计人员也可以设计出符合规则的软件系统。此时的指令集架构可以理解为一个抽象层：它规定了硬件的模式但不干涉具体实现，也规定了软件如何调用硬件并说明了会有什么结果。这样，一个软件若遵循了某一指令集架构，则可以在任何遵循同一指令集架构的计算机上运行，实现了软件的可移植性。

一般认为，指令集架构分为复杂指令集（Complex Instruction Set Computer，CISC）和精简指令集（Reduced Instruction Set Computer，RISC）。二者的区别如下为：复杂指令集包含了许多复杂、不常用操作，指令条数较多，长度不固定；精简指令集包含基本、常用操作，指令条数较少，长度固定。

处理器发展早期，复杂指令集被大范围应用。此时，计算机性能的提高往往是通过增加硬件的复杂性获得的。长期的发展导致常见的复杂指令集的指令系统十分庞大、复杂，如 x86，这对处理器设计者有着很大的挑战。

另一方面，1979 年以帕特逊教授为首的一批科学家研究发现管理学中的“80/20”定律十分适合描述复杂指令集：80%的重要任务由 20%的人完成，剩下的 80%只完成 20%的工作。用在复杂指令集上其意为：80%的工作只使用了 20%的指令，剩下的 80%指令只会在 20%的情况中出现。而这些不常用指令一般都可使用简单指令实现。实现这些指令对设计和芯片面积来时都是极大的浪费。

这样，精简指令集应运而生。它只提供了少量的、使用频率很高的指令，在功能完备的前提下力图简化处理器的设计复杂度。在此之后，大部分指令集架构都采取了精简指令集。

x86 是由美国英特尔公司推出的一种复杂指令集，于 1978 年推出的 Intel8086 中央处理器中首度出现，后被 IBM PC 选用，还与微软公司结成了 Wintel 联盟。现在，x86 在 PC 领域达成了几乎无法撼动的垄断。X86 的两大制造商为 Intel 和 AMD。两家公司在不断地竞争中挤垮了数家生产 x86 处理器的其他公司，形成了现在的格局。

ARM（Advanced RISC Machine）是由英国 Acorn 公司设计的一种 RISC 指令

集，于 1985 年在同名处理器上第一次出现。后 Acorn 公司剥离出 ARM 公司。ARM 公司的盈利模式与 Intel 公司有很大不同：ARM 公司现在不生产处理器，而是将自己设计的 ARM 处理器核已 IP 的形式授权给其他公司；其他公司使用 ARM 公司设计的 IP 核，根据应用领域添加适当的外围电路，在进行生产售卖。而 ARM 公司的利润则来源于 IP 核授权时的前期授权费和制造商贩卖 ARM 处理器所得利润按照百分比支付给 ARM 的版税。而 ARM 作为移动和嵌入式等低功耗领域的处理器巨头，无形的胁迫其他的厂商来购买 ARM 的 IP 核。一般的厂商除去购买 ARM 的 IP 核来生产处理器外，还有购买其他厂商 IP 核与自己设计 IP 核两个选择。但是由于 ARM 在低功耗领域的巨大体量，出于资金、设计周期和兼容性等方面的考虑，其他厂商不得不继续向 ARM 购买 IP 核。因此，ARM 虽然并不像 Intel 一样知名并且少数公司垄断行业，但其凭借整个 ARM 阵营（高通、苹果、华为、三星、TI 等）仍实现了低功耗应用领域的垄断。

Power（Performance Optimization With Enhanced RISC）是由美国 IBM 公司设计的一种 RISC 指令集，于 1990 年与名为“RISC System/6000”的计算机一起正式面世。IBM 制作的基于 Power 的服务器证明了 Risc 在高性能领域也有明显的优势。现在，Power 指令集的主要应用领域为服务器领域。Power9 已经被应用，并且 Power 仍在不断地发展之中：在可见的未来，IBM 计划推出 Power9 和 Power10。另，微软公司推出的 Xbox360 游戏机采用的是一个三核的，基于 Power 的处理器。

MIPS（Microprocessor without Interlocked Pipeline Stages）是由斯坦福大学的 Hennessy 教授领导的小组开发，其设计理念为透过指令管线化来增加 CPU 运算的速度，于 1981 年面世。后 Hennessy 离开斯坦福大学并成立 MIPS 公司。由于经营不善，MIPS 公司被收购。索尼的数字电视与娱乐设备、Linksys 的宽带设备都使用了基于 MIPS 的处理器。

1.1.7. RISC-V 汇编简介

上述的四个仅是众多 ISA 中的佼佼者，但即便有了如此多的 ISA，有一个问题仍然没有解决：我们仍然没有一个开源的 ISA。

大多数完善的 ISA 几乎都由一个公司掌控，单一公司一时的浮沉将影响整个

ISA 生态; 另一方面，RISC-V 的设计者最开始希望找到一个完善的指令集用于研究，但却由于授权等各种问题遇到了阻挠，RISC-V 的设计计划便被提上了日程。

RISC-V 主要有以下两个特性: 开源和精简。开源，即意味着任何人可以拿来使用。现在 RISC-V 属于 RISC-V 基金会，一个开放的、非盈利性质的基金会。基金会的目的是缓慢、谨慎的发展 RISC-V，“力图让它之于硬件如同 Linux 之于操作系统一样受欢迎”。精简，则是彻头彻尾的简单，最大程度上方便硬件设计者。首先是指令分为模块，只需实现需要使用的模块即可; 其次，在基本的模块 RV32I 中，只有仅仅 47 条指令，最大限度的减少设计工作量与芯片面积。

RISC-V 的指令集并不固定，而采用模块化进行组织。每个模块使用一个英文字母进行表示，实现部分可选的功能。注意: 字母 I 代表的基本整数指令子集为必选模块，可以实现一个编译器的全部功能。其指令子集和功能对应关系如下表。另外除去 A 子集外所有的指令子集均可由 I 子集仿真实现。

表 1-1RISC-V 指令子集

基本指令集	指令数	描　　述
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 位指令
拓展指令集	指令数	描　　述
M	8	整数乘法与除法指令
A	11	储存器原子操作指令和 Load-Reserved/Store-Condition 指令
F	26	单精度(32bit)浮点指令
D	26	双精度(64bit)浮点指令
G(IMAFD)		IMAFD 的通用组合
C	46	压缩指令，指令长度为 16 位

在处理器设计中，只需要跟据使用的环境挑选合适的指令子集加以实现即可。比如，可以选择 RV32I 与 M 组成 RV32IM; 在大型计算机中可以选择 RV64G，而在嵌入式中选择 RV32EC。还有很多其他子集，如 V（向量计算指令）、B（位操作指令）、L（十进制浮点指令）等。这些子集或与初步学习关系不大或正处于开发之中，故不再赘述。

接下来，将主要介绍 RV32I。RV32I 中共有 32 个通用整数寄存器，其中 x0 是恒零计数器，x1 到 x31 为通用寄存器。额外有一个 PC（Program Counter）寄

存器，用来储存当前指令地址。

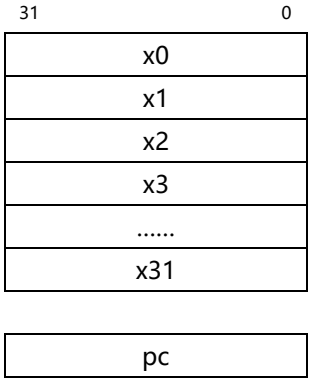

图 1-2 通用寄存器

x0 从硬件实现上连接为 0，所有对于 x0 的写入都被视为无效，所有从 x0 的读取结果都是 0。这个设计允许了许多伪指令的简单实现，如一个空指令（nop）对应基本指令为:

```
addi x0,x0,0      # x0 = x0 + 0
```

使用立即数加法，把 x0 置为 x0 与 0 的和。X1 到 x31 是通用寄存器，实现上他们是等价的，拥有完全相同的功能，但在后续章节将介绍调用规范，每一个寄存器将分配一个特定的用途。在使用中最好遵循这些规范。

首先明确，RV32I 的所有指令的长度均为 32，处理器解码时无需判断长度，简化了解码逻辑。指令共分为六种格式，R 类型为寄存器指令，其余的 I、S、B（SB）、U 和 J（UJ）类型均为立即数指令。指令编码格式和生成立即数格式如下图:

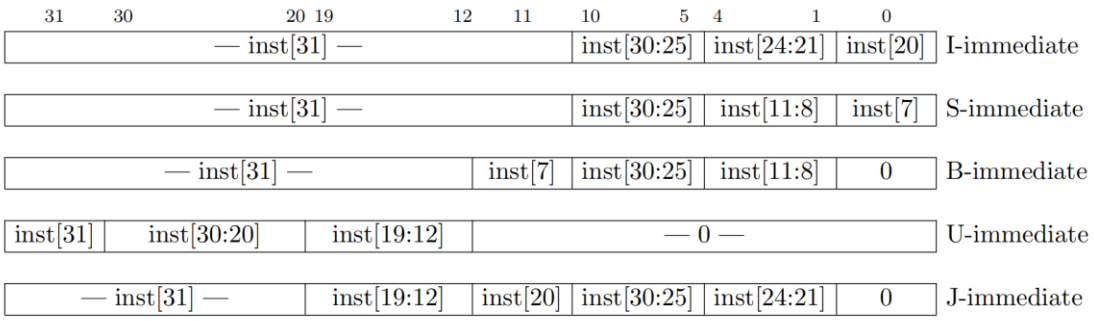

图 1-3 指令编码格式和生成立即数格式

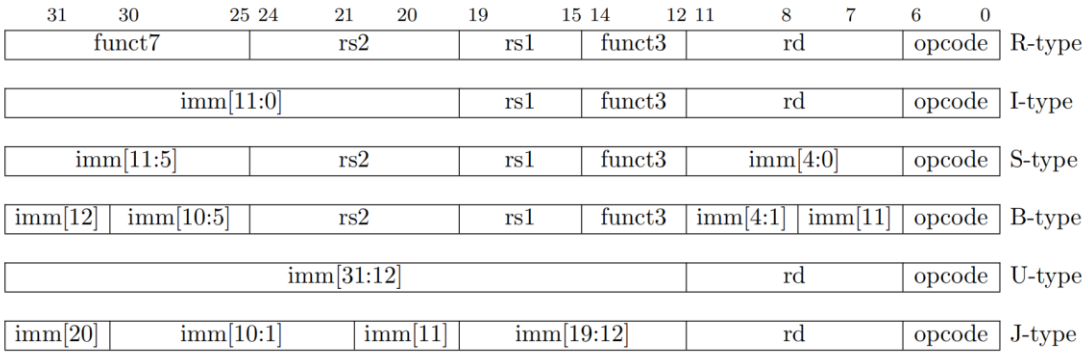

图 1-4RV32I 基本指令格式

可以看出，除去 U 和 J 类型，每一个指令类型的操作数都为三个，而不是像 x86-32 一样让目标操作数与源操作数共用一个字段。举一个简单的例子说明这样做的好处。当我们想完成如下一个操作，把 x7 置为 x5 与 x6 的和，只需要:

```
add x7,x5,x6      # x7 = x5 + x6
```

仿照 x86-32 的格式则需要:

```
mov x7,x5     # x7 = x5
add x7,x6         # x7 = x7 + x6
```

这样即可节省一条指令。

RV32I 中可以使用的通用寄存器数量为 32，故所有指令类型中对于寄存器的标志符占据了五位。在设计方面，所有指令类型的寄存器标志符都在同一个位置，这样设计的好处是在指令解码前便可以开始对寄存器的访问，使处理器解码路径的时序更容易实现。

能产生立即数的指令格式有五个，对应了五种立即数。所有立即数都是符号扩展的，其符号位永远在指令的最高位，这同样意味着符号扩展操作可以和指令译码同时进行。B（分支）和 J（跳转）指令格式对应的立即数左移一位成为 2 的倍数，使分支跳转拥有更大的范围。

我们拿 1.2.1 中给出的指令作为例子:

```
add x7,x5,x6      # x7 = x5 + x6
```

其中 add 称作操作码（Operation code）; x7 称作目标寄存器（Destination register）; x5 和 x6 分别称作第一和第二操作寄存器（First/Second operand register）。

RV32I 所有指令如图 1.4（指令编码和指令详细说明在附录中给出）。

RV32I 的整数运算指令包括：算数指令、逻辑指令、位移指令、构造指令和置位指令。除去装载的其他指令还有一个对应的立即数版本。以 add rd,sr1,sr2 为例，其表示从两个寄存器读数求和并将结果放置到 rd 寄存器中；addi rd,sr1,imm 表示把一个寄存器的值，与符号扩展的十二位立即数求和，结果放置到 rd 寄存器中。sub 没有立即数版本，只需将 addi 的立即数变为负数即可。

有两点需要注意：所有的立即数都是符号扩展至 32 位再进行运算；所有的运算指令均没有溢出检测机制。

符号扩展的作用比较容易理解，即没有字节或半字宽度的运算，所有都是寄存器的完整宽度运算；没有溢出检测是因为仅需要添加几条额外指令，以较小的代价便可实现，但从硬件设计上极大的减轻了设计复杂度。

两条构造指令用于构造 32 位立即数。可以注意到，由于指令长度固定为 32，操作码和寄存器也占用了一定长度。这使得一条指令无法直接构造 32 位立即数，故设计了 load 和 auipc 两条指令，用于构造立即数的高 20 位到目标寄存器并将其低十二位置 0，再使用其他指令构造低 12 位。前者用于构造常数，后者用于构造 pc 相对地址。

Integer Computation

add { - / immediate }

subtract

{ and / or / exclusive or } { - / immediate }

{ shift left logical / shift right arithmetic / shift right logical } { - / immediate }

load upper immediate

add upper immediate to pc

set less than { - / immediate } { - / unsigned }

Loads and Stores

{ load / store } { byte / halfword / word }

load { byte / halfword } unsigned

Control Transfer

branch { equal / not equal }

branch { greater than or equal / less than } { - / unsigned }

jump and link { - / register }

Miscellaneous instructions

fence loads & stores

fence.instruction & data

environment { break / call }

control status register { read & clear bit / read & set bit / read & write } { - / immediate }

置位指令可以用于生成布尔值。但 RV32I 只提供了小于时置位的指令及其立即数、无符号、立即数且无符号版本，其他的比较只需要调整寄存器的顺序，有需要时添加异或指令即可完成。同时，对于复杂的逻辑表达式，也可以用该与异或指令的组合完成。

RV32I 的控制转移指令包括无条件跳转指令和条件分支指令。

无条件跳转指令用于实现程序跳转，包括 jal 和 jalr 两条指令。前者的跳转流程为将返回地址（即下一条指令的地址 pc+4）储存到目标寄存器（一般为 x1），在将立即数加到 pc 寄存器上完成跳转；后者同样先保存返回地址，而随后将 pc 设置为立即数与指定的另一寄存器的值相加且最低值置 0 的结果。后者的跳转地址为动态计算得到的，可以用于实现动态函数，调用返回等功能。当返回地址不需要时，可将 x0 作为目标寄存器。

条件分支指令用于实现条件判断并跳转，包括 beq（相等），bne（不相等），bge（大于等于）和 blt（小于）五条指令。大于和小于等于关系的比较跳转可通过伪指令实现，均需要调换两源寄存器的顺序即可，即：$x>y \Leftrightarrow y<x$。跳转方式为将立即数左移一位（乘 2）符号扩展并加到 pc 上。另，RISC-V 没有分支延迟。

读取存储指令对应 load 和 store 两组指令。load 中包括加载有符号指令: lw（32 位字），lh（16 位半字），lb（8 位字节）; 加载无符号数指令: lhu（无符号 16 位半字），lbu（无符号 8 位字节）。Store 中包括: sw（32 位字），sh（16 位半字），sb（8 位字节）。load 和 store 指令的内存地址均由基地址与符号扩展的 12 位立即数相加得到。另: 数据并不要求按照其自然大小进行边界对其。

剩下的指令包括控制状态寄存器指令、环境调用与断点指令和 fence 指令。控制状态寄存器指令包括 csrrc、csrrs、csrrw、csrrci、csrrsi、csrrwi 六条指令，用于完成对于计数器的访问。计数器为 64 位，一次可以读取 32 位。环境调用和环境断点指令为 ecall 和 ebreak，用于对系统运行环境发送请求。fence 指令包括 fence 和 fence.i 两条指令。fence 用于顺序化其他线程、外部设备等对 I/O 和存储器的访问。fence.i 指令用于同步指令和数据流。下表将给出寄存器及其二进制接口（ABI）名称与解释。

表 1-2RV32I 寄存器与二进制接口名称

Register	ABI name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Caller
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary /alternate link register	Caller
x6-7	t1-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
X18-27	s2-11	Saved registers	Callee
X28-31	t3-6	Temporaries	Caller

上表的最后一列给出了 RV32I 各个寄存器在调用过程中的保留情况。Caller 表示该寄存器由函数调用者控制，在函数调用过程前后不改变; Callee 表示在函数调用过程中，寄存器的值将不予保留。

RV32I 将寄存器 x2 定义为 ‘Stack pointer ‘。该寄存器将一直储存栈顶元素的地址。

RV32I 没有如 x86-32 中的 push 与 pop 指令，使用堆栈需要多条指令完成。

下面将分别给出 push 和 pop 的一种实现。注意，下例中入栈出栈以 4 字节为一元素，指针加减应为 4 的倍数。

```
addi sp, sp, -8      # 改变 sp，将入栈两个元素
sw t1, 4(sp)      # 将 t1 入栈
sw t2, 0(sp)      # 将 t2 入栈
…
lw t2, 0(sp)      # 将 t2 入栈
lw t1, 4(sp)      # 将 t1 入栈
addi sp, sp, -8      # 改变 sp，出栈两个元素
```

在函数调用过程中，如需要使用一个调用者保留的寄存器，可在无影响的情况下将该寄存器的值进行入栈操作，在返回前完成出栈操作。一般的，在函数返回时，堆栈的状态应与调用函数时的堆栈状态相同。

按照的调用惯例，RV32I 的堆栈从高地址开始，入栈时向下增长，通常对齐到 16 字节（上例对齐到 4 字节）。下图将展示 RV32 的内存分配惯例，顶部为高地址，底部为低地址。

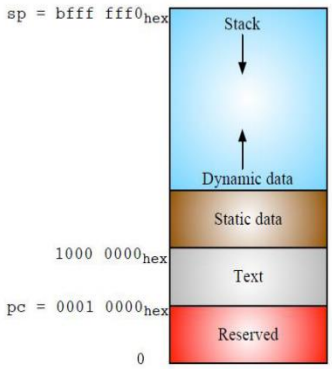

图 2.1　RV32 内存分

如图所示：堆栈从 0xbfff fff0 开始向下增长；程序代码从 0x0001 0000 开始，向上延伸储存；规定寄存器 gp（global pointer）指向静态数据区开始地址 0x1000 0000；继续向上为动态数据区，由 C 语言中的 malloc 函数分配使用。

函数调用分为以下六个主要阶段：

将参数储存到函数能够访问到的位置；

跳转到函数开始的位置；

获取函数需要的局部存储资源，按需保存寄存器；

执行函数中的指令;

将返回值储存到调用者能够访问到的位置,恢复寄存器,释放局部储存资源;

返回调用函数位置。

RV32I 规定了 x10-17 为 a0-7（argument registers，参数寄存器），用于调用时传递参数给函数。跳转时一般使用 jal 指令，进行返回地址的保存与跳转;若跳转地址超出了 12 位有符号立即数的表示范围,可以使用伪指令 call 进行跳转。规定 x1 为 ra（return address，返回地址寄存器）用于储存函数返回时的地址，即调用时跳转指令的地址+4。

函数中的指令要符合寄存器的使用规范。RV32I 提供了足够多的寄存器，应尽量将操作数放在寄存器中，减少保存和恢复寄存器的次数; 同时规定了 t0-6（temporary register，临时寄存器）给函数使用，在调用过程中不保留其储存的值; 相对的，规定了 s0-11（saved register，保存寄存器）在调用过程中应保留其储存的值。对于其他的寄存器，若按照表 2.1 在调用过程中也需要保留储存的值，但仍需要使用该寄存器，如需要对堆栈进行操作时的 sp 寄存器和继续调用其他函数时的 ra 寄存器，在函数调用前后应该保持不变。

执行完函数体后，a0-1 两个寄存器同时规定为 return value（返回值寄存器），用于在函数返回时储存返回值。最后使用指令 jalr 或伪指令 ret / jr 进行返回。

此部分将给出两个 C 语言函数及其对应 RV32I 汇编代码。C 语言函数代码如下:

```
Int Leaf (int g, int h, int i, int j)
{
   Int f ;
   f = (g + h) - (i + j);
   return f;
}
```

为进行展示，作如下说明与规定: 参数 g，h，i，j 由 a0-3 进行储存、传递; f 对应 s0，且额外需要一个临时寄存器 s1; s0 与 s1 储存的值在调用中需要保留，故需进行入栈、出栈操作。

```
Leaf:
addi sp, sp, -8      # 入栈两个元素
sw s1, 4(sp)      # 保存 s1 储存值
sw s0, 0(sp)      # 保存 s0 储存值
```

```
add s0, a0, a1        # f = g + h
add s1, a2, a3        # s1 = i + j
sub a0, s0, s1        # 返回值 a0 = (g + h) - (i + j)

lw s0, 0(sp)      # 恢复 s1 储存值
lw s1, 4(sp)      # 恢复 s0 储存值
addi sp, sp, 8        # 出栈两个元素
ret               # 返回，等价于伪指令 jr ra 和指令 jalr x0, ra, 0
```

RISC-V 的汇编程序由数据段和代码段构成，由汇编指示符说明接下来代码属于那一段。几个常用的汇编指示符如下（详细的汇编指示符见附录）:

```
.text             # 进入代码段
.align 2          # 后续代码按 22 字节对其
.globl main       # 声明全局符号 main
.section data     # 进入数据段
.section rodata       # 进入只读数据段
.balign 4         # 后续代码按 4 字节对其
.string "Hello\n"# 创建空字符串结尾的字符串
```

当通过“.section .data“进入数据段或“.section .rodata“进入只读数据段后，可以使用“.byte“(创建连续的 8 位变量)、“.half“(创建连续的 16 位变量)、“.string“(创建字符串）等指示符在内存数据区创建数据。

通过“.text”进入代码段后，既是需要执行的代码部分。包括汇编指令与伪指令。

程序段与数据端均允许有标签（lable)。标签是提供给汇编器的信息，它在数据段代表下一个声明的数据的存储地址,在代码段代表下一个指令的存储地址。它可以由纯字母、纯数字或字母和数字组成，之后跟随一个“:”表示该符号位标签。注意，纯数字的标签在使用时需要在之后添加一个“b”，如:

```
1:
addi t1, t1, 1    # t1 += 1
j 1b          # 循环条指令
```

一般程序需要指明入口地址，即全局符号 main。首先按照上文提到的方法进行全局变量声明，随后在数据段定义 main 符号即可。当程序执行时，从 main 指示的指令处开始执行。

1.2实验目的

了解程序设计语言的发展历史，了解不同程序设计语言的各自特点；感受编译执行和解释执行两种不同的执行方式，初步体验语言对编译器设计的影响，为后续编译程序的设计和开发奠定良好的基础。

1.3实验内容

给定一个特定的功能，分别使用 C/C++、Java、Python 和 Haskell 实现该功能，对采用这几种语言实现的编程效率，程序的规模，程序的运行效率进行对比分析。例如分别使用上述几种语言实现一个简单的矩阵乘法程序，输入两个矩阵，输出一个矩阵，并分析相应的执行效果。

1.4实验过程与方法

可以在任意文本编辑器中编辑代码，保存为相关文件后，对于 C/C++和 Java，分别调用相应的编译器将其编译为可执行程序和字节码文件，直接执行 C 对应的可执行程序，使用 Java 虚拟机运行相应的 Java 程序，Python 和 Haskell 则需要相应的解释程序。

一般机器上自带 C/C++语言的编译器，可以分别从 Java、Python 和 Haskell 的官方网站下载相关的解释器和运行时环境，对应的官方网站地址如下表所示：

表 1 各个语言相关信息

语言	网址	推荐版本
Python	https://www.python.org	Python 2.7
Haskell	https://www.haskell.org/	GHC 7.0
Java	http://www.oracle.com/technetwork/java/javase/downloads/index.html	Java 1.6 以上
MIPS	http://courses.missouristate.edu/KenVollmar/mars/	
RISC-V	https://github.com/jiweixing/bit-rars	

需要说明的是，Python 和 Haskell 的语法比较特殊，从初步接触到上手编写程序需要一段时间。Python 官方网站提供了学习视频和在线教程，可以根据需要进行学习。

MIPS 和 RISC-V 可以从网上搜索相应的汇编器，也可以使用表 1 中的两个模

拟器。使用模拟器只能计算程序运行的 Cycle，没有时间测量。

1.5实验提交内容

本实验要求提交实验源码，C/C++需提供对应的可执行程序（不需要编译的中间文件），Java 提供编译后的 class 文件或者 jar 包，每个人提交一份实验报告。提交目录如下所示:

C/C++/	*bin*	*Python/*	*src*
	src		
Java/	*bin*	*Haskell/*	*src*
	src		*src*
MIPS/x86/RISC-V/ARM assembly	*src*	*Doc/*	*report*

每个教学班级应按照语言进行分类，分别打包，以方便验收。实验报告应包括如下内容:

- 实验目的和内容
- 实现的具体过程和步骤
- 运行效果截图
- 语言易用性和程序规模对比分析
- 程序运行性能对比分析
- 实验心得体会

其中语言易用性主要从学习难度和语言的编程效率等方面对四种语言进行对比分析，程序规模主要考察分析使用不同语言实现同一种功能所得到的程序规模大小，可以以代码行数为度量单位进行对比分析。

程序运行性能以程序运行时间为度量单位，每个程序应运行 5~10 次并取其平均值作为度量结果。在实验中应说明实验进行的计算机系统硬件配置情况，例如 CPU 核数，CPU 主频，内存和 Cache 大小等。如果程序的运行时间较短，则测试的结果的误差会比较大，建议增加输入数据规模，再进行对比分析。

1.6建议的题目列表

（1）使用上述各种语言分别实现矩阵相乘;

（2）使用上述各种语言分别实现归并排序;

（3）使用上述各种语言分别实现快速排序;

（4）使用上述各种语言分别实现矩阵和向量相乘。

1.7参考文献

[1]. https://www.freejavaguide.com/history.html

[2]. https://en.wikipedia.org/wiki/OpenJDK

[3]. https://www.javatpoint.com/history-of-java

[4]. https://javapapers.com/core-java/java-history/

[5]. https://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html