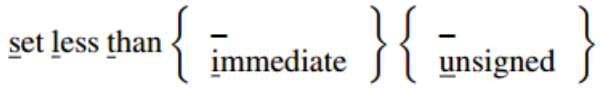
提升计算性能并且让用户能切实享受到性能提升的的唯一方法是同时设计编译器和计算机。这样子，软件用不到的特性将不会被实现在硬件上。
Frances Elizabeth “Fran” Allen, 1981

2.1 介绍

图 2.1 是 RV32I 基础指令集的一页图形表示。
对于每幅图，将有下划线的字母从左到右连接起来，即可组成完整的 RV32I 指令集。
对于每一个图，集合标志{}内列举了指令的所有变体，变体用加下划线的字母或下划线字符_表示。特别的，下划线字符_表示对于此指令变体不需用字符表示。
例如，下图表示了这四个 RV32I 指令：slt，slti，sltu，sltiu。



我们使用这些图（下面几章的第一个图），旨在对本章的指令给出一个进行快速、深入的概述。

2.2 RV32I 指令格式

图 2.2 显示了六种基本指令格式，分别是：用于寄存器-寄存器操作的 R 类型指令，用于短立即数和访存 load 操作的 i 型指令，用于访存 store 操作的 S 型指令，用于条件跳转操作的 B 类型指令，用于长立即数的 U 型指令和用于无条件跳转的 J 型指令。
图 2.3 使用图 2.2 的指令格式列出了图 2.1 中出现的所有 RV32I 指令的操作码。

即使是指令格式也能从一些方面说明 RISC-V 更简洁的 ISA 设计能提高提高性能功耗比。首先，指令只有六种格式，并且所有的指令都是 32 位长，这简化了指令解码。arm 32，特别是 x86-32 有许多不同的指令格式，使得解码部件在低端实现中偏昂贵，在中高端处理器设计中容易带来性能挑战。第二,RISC-V 指令提供三个寄存器操作数，而不是像 x86-32 一样，让源操作数和目的操作数共享一个字段。当一个操作天然就需要有三个不同的操作数，但是 ISA 只提供了两个操作数时，编译器或者汇编程序程序员就需要多使用一条 move（搬运）指令，来保存目的寄存器的值。第三，在 RISC-V 中对于所有指令，要读写的寄存器的标识符总是在同一位置，意味着在解码指令之前，就可以先开始访问寄存器。在许多其他的 ISA 中，某些指令字段在部分指令中被重用作为源目的地，在其他指令中又被作为目的操作数（例如，arm 32 和 MIPS-32）。因此，为了取出正确的指令字段，我们需要时序本就可能紧张的解码路径上添加额外的解码逻辑，使得解码路径的时序更为紧张。第四，这些格式的立即数字段总是符号扩展，符号位总是在指令中最高位。这意味着可能成为关键路径的立即数符号扩展，可以
在指令解码之前进行。

精讲：B 类型和 J 类型指令
如下所述，直接字段为分支指令旋转了 1 位，这是一个变体在 S 格式中，我们重新标记了 B 的格式。跳转指令的直接字段跳跃指令的 12 位，一种 U 格式的转换 J 格式的变体。因此,有
一种真正的四种基本格式，但我们可以保守地估计 RISC-V 有六种格式。

为了帮助程序员，所有位全部是 0 是非法的 RV32I 指令。因此，试图跳转到被清零的内存区域的错误跳转将会立即触发异常，这可以帮助调试。类似地，所有位全部是 1 的指令也是非法指令，它将捕获其他常见的错误，诸如未编程的非易失性内存设备、断开连接的内存总线或者坏掉的内存芯片。
为了给 ISA 扩展留出足够的空间，最基础的 RV32I 指令集只使用了 32 位指令字中的编码空间的不到八分之一。架构师们也仔细挑选了 RV32I 操作码，使拥有共同数据通路的指令的操作码位有尽可能多的位的值是一样的，这简化了控制逻辑。最后,当我们看到，B 和 J 格式的分支和跳转地址必须向左移动 1 位以将地址乘以 2，从而给予分支和跳转指令更大的跳转范围。
RISC-V 将立即数中的位从自然排布进行了一些移位轮换，将指令信号的 fanout 和立即数多路复用的成本降低了近两倍，这也简化了低端实现中的数据通路逻辑。

有什么不同吗?在这一章和后面的章节的结束部分，我们将描述 RISC-V 与其他指令集的不同之处。这种对比通常是描述相比于其他指令集，RISC-V 少了什么。省略什么特性和包括什么特性一样，都能体现架构师的良好的品味。
arm 32 指令集 12 位的立即字段不仅仅是一个常量，而是一个函数的输入，此函数根据 12 位立即数的输入来产生一个常量：8 位被零扩展到全宽度，然后被循环右移。右移的位数是 12 位立即数中剩余 4 位的值乘 2。设计者希望在 12 位中编码更多有用的常数来减少执行指令的数量。在大多数指令格式中，arm 32 也将十分宝贵的四位编码空间拿出来专门用于条件执行。这些条件执行指令不仅使用频率低而且增加了乱序处理器的复杂性。

精讲：乱序处理器是高速的、流水化的处理器，它们一有机会就执行指令，而不是在按照程序顺序。这种处理器的一个关键特性是寄存器重命名，把程序中的寄存器名称映射到大量的内部物理寄存器。条件执行的问题是不管条件是否成立，都必须给这些指令中的寄存器分配相应的物理寄存器。但内部物理寄存器的可用性是影响乱序处理器的关键性能资源。

2.3 RV32I 寄存器

图 4 列出了 RV32I 寄存器以及由 RISC-V 应用程序二进制接口（ABI）所定义的寄存器名称在我们的示例代码中，我们将使用 ABI 名称，使它们更容易阅读。为了让汇编语言程序员和编译器编写者感到高兴，RV32I 有 31 寄存器加上一个值恒为 0 的 x0 寄存器，总是值为 0。与之相比，arm 32 只有 16 个寄存器，x86-32 甚至只有 8 个寄存器。

有什么不同吗?为常量 0 单独分配一个寄存器是 RISC-V ISA 能如此简单的一个很大的因素。第 3 章的第 36 页的图 3 给出了许多 arm 32 和 x86-32 的原生指令操作，这两个指令集中没有零寄存器。我们可以用 RV32I 指令完成功能相同的操作，只需使用零寄存器作为操作数。

程序计数器（PC）是 arm 32 的 16 个寄存器之一，这意味着任何改变寄存器的指令都有可能是一个分支指令？？？？。PC 作为一个寄存器使硬件分支预测变得复杂，因为在典型的 ISA 中，仅 10%-20%的指令为分支指令，而在 arm32 中，任何指令都有可能是分支指令。而分支预测的准确性对于良好的流水线性能至关重要。另外将 PC 作为一个寄存器也意味着可用的通用寄存器少了一个。

2.4 RV32I 整数计算

附录 A 给出了所有 RISC-V 指令的细节信息，包括格式和操作码。在本节中，以及接下来的章节的类似小节中，我们将给出 ISA 的一些概述。这能够让有基础的汇编语言程序员了解 RISC-V，同时也顺便说明 RISC-V 的特性如何满足第一章中阐述的七个 ISA 指标。

简单的算术指令（加、减）、逻辑指令（与，或，异或），以及图 2.1 中的移位指令（算术左移、逻辑右移、逻辑左移）和其他 IS 差不多。他们从寄存器读取两个 32 位的值，并将 32 位结果写入目标寄存器。RV32I 还提供了这些指令的立即数版本。和 ARM-32 不同,立即数总是进行符号扩展，这样子如果需要，我们可以用立即数表示负数，正因为如此，我们并不需要一个立即数版本的 sub。

程序可以根据比较结果生成布尔值。为应对这种使用场景下，RV32I 提供一个当小于时置位的指令。如果第一个操作数小于第二个操作数，它将目标寄存器设置为 1，否则为 0。不出所料，对这个指令，有一个有符号版本（slt）和无符号版本（sltu），分别用于处理有符号和无符号整数比较。相应的，上述两条指令也有立即数版本的（slti，sltiu）。正如我们将要看到的，虽然 RV32I 分支指令可以检查两个寄存器之间的所有关系，但一些条件表达式涉及多对寄存器之间的关系。对于这些表达式，编译器或汇编语言程序员可以将 slt 以及与或异或等逻辑指令组合使用来解决更复杂的条件表达式。

Page 18，语句不通顺。

图 2.1 剩下的两条整数计算指令主要用于构造大的常量数值和链接。加载立即数到高位（lui）将 20 位常量加载到寄存器的高 20 位。接着便可以使用标准的立即指令来创建 32 位常量。这样子，仅使用 2 条 32 位 RV32I 指令，便可构造一个 32 位常量。向 PC 高位加上立即数（auipc）让我们仅用两条指令，便可以基于当前 PC 以任意偏移量转移控制流或者访问数据。将 auipc 中的 20 位立即数与 Jalr（参见下面）中 12 位立即数的组合，我们可

以将执行流转移到任何 32 位 PC 相对地址。而 auipc 加上普通加载或存储指令中的 12 位立即数偏移量，使我们可以访问任何 32 位 PC 相对地址的数据。

有什么不同？首先，RISC-V 中没有字节或半字宽度的整数计算操作。操作始终是以完整的寄存器宽度。内存访问需要的能量比算术运算高几个数量级。因此低宽度的数据访问可以节省大量的能量，但低宽度的运算不会。 ARM-32 具有一个不寻常的功能，对于大多数算术逻辑运算中的一个操作数，你可以选择对它进行移位。尽管这些指令的使用频率很低，但它使数据路径和数据通路更加复杂。与此相对的是，RV32I 提供了单独的移位指令。

RV32I 也不包含乘法和除法，它们包含在可选的 RV32M 扩展中（参见第 4 章）。与 ARM-32 和 x86-32 不同，即使处理器没有添加乘除法扩展，完整的 RISC-V 软件栈也可以运行，这可以缩小嵌入式芯片的面积。MIPS-32 汇编程序可能用一系列移位以及加法指令来替换乘法，以提高性能，这可能会使程序员看到处理器执行了汇编程序中没有的指令，进而造成混淆。
虽然不是硬件问题？？？RV32I 可以忽略了这些特性：循环移位指令和整数算术溢出检测，这两个特性都可以用若干条 RV32I 指令来实现（参见第 2.6 节）

2.5 RV32I 加载和存储

除了提供 32 位字（lw，sw）的加载和存储外，图 2.1 中说明，RV32I 支持加载有符号和无符号字节和半字（lb，lbu，lh，lhu）和存储 字节和半字（sb，sh）。有符号字节和半字符号扩展为 32 位再写入目的地寄存器。即使是自然数据类型更窄，低位宽数据也是被扩展后再处理，这使得后续的整数计算指令能正确处理所有的 32 位。在文本和无符号整数中常用的无符号字节和半字，在写入目标寄存器之前都被无符号扩展到 32 位。
加载和存储的支持的唯一寻址模式是符号扩展 12 位立即数到基地址寄存器，这在 x86-32 中被称为位偏移寻址模式[Irvine 2014]。
有什么不同？ RV32I 省略了 ARM-32 和 X86-32 的复杂寻址模式。另外，ARM-32 提供的寻址模式并非适用于所有数据类型，但 RV32I 寻址不会歧视任何数据类型。 RISC-V 可以模仿某些 x86 寻址模式。例如，将立即数字段设置为 0 与即与 X86 中的寄存器间接寻址效果相同。与 x86-32 不同，RISC-V 没有特殊的堆栈指令。将 31 个寄存器中的某一个作为堆栈指针（见图 2.4），标准寻址模式使用起来和压栈（push）和出栈（pop）类似，并且不增加 ISA 的复杂性。与 MIPS-32 不同，RISC-V 不支持延迟加载（delayed load）。与延迟分支的设计相似，为了更好的适应五级流水线，MIPS-32 重新定义了 load 指令的语义，load 上来的数据在 load 指令两个指令后才可用。但是对于后来出现的更长的流水线，delayed load 带来的收益逐渐消失，因此 RISC-V 不支持延迟加载。

虽然 ARM-32 和 MIPS-32 要求存储在内存中的数据，要按照数据的自然大小进行边界对齐，但是 RISC-V 没有这个要求。移植旧的代码有时需要未对齐的访问。对于不对齐访问，一种选择是在基础 ISA 中禁止不对齐访问，然后提供 一些单独的指令用于不对齐访问，例如 MIPS-32 中的 Load Word Left 和 Load Word Right。然而，这会使寄存器访问变得复杂，因为 lwl 并且 lwr 需要对寄存器进行部分写，而不是简单地对寄存器进行完整的写。支持

不对齐访问的，另一种方法就是让普通的加载和存储指令支持不对齐访问，这简化了整体设计。

详细说明：字节序问题
RISC-V 选择了低字节优先的字节排序，因为它在商业上占主导地位：所有 x86-32 系统，Apple iOS，谷歌 Android 操作系统和微软 Windows for ARM 都是低字节优先序。由于字节顺序仅在同时以按字访问和按字节访问同一份数据时才会有影响，字节序只会影响很少一部分的程序员。

2.6 RV32I 条件分支
RV32I 可以比较两个寄存器并根据比较结果上进行分支跳转。比较可以是：相等（beq），不相等 （bne），大于等于（bge），或小于（blt）。最后两种比较有符号比较，RV32I 也提供相应的无符号版本比较的：bgeu 和 bltu。剩下的两个比较关系（大于和小于等于）可以通过简单地交换两个操作数，即可完成比较。因为 x 〈y 表示 y〉 x 且 x≥y 表示 y≤x。
由于 RISC-V 指令长度必须是两个字节的倍数 - 关于可选的双字节指令 ，请参考第七章-分支指令的寻址方式是 12 位的立即数乘以 2，符号扩展它，然后将得到值加到 PC 上作为分支的跳转地址。PC 相对寻址可用于位置无关的代码，简化了链接器和加载器的工作（第 3 章）。

有什么不同？如上所述，RISC-V 去掉了 MIPS-32，Oracle SPARC 等指令集中被广为诟病的延迟分支特性等。对于条件分支，它还没有像 ARM-32 和 x86-32 那样使用条件码。条件码的存在使得大多数指令都需要隐式设置一些额外状态，这使乱序执行的依赖计算复杂化。最后，它省略了 x86-32 中的循环指令：loop，loope，loopz，loopne，loopnz。

精讲：不使用条件码实现大位宽数据的加法
在 RV32I 中是通过 sltu 计算进位来实现的：
add a0，a2，a4 ＃加低 32 位：a0 = a2 + a4
sltu a2，a0，a2 ＃如果（a2 + a4）〈a2，a2'= 1 ，否则 a2'= 0
add a5，a3，a5 ＃加高 32 位：a5 = a3 + a5
add a1，a2，a5 ＃加上低 32 位的进位

精讲：获取 PC
当前的 PC 可以通过将 auipc 的 U 立即数字段设置为 0 来获得。对于 x86-32，要想读取 PC，你需要先进行函数调用，（这样子可以将 PC 推入堆栈）;然后被调用的函数可以从堆栈中读取刚被压栈的 PC，最后将 PC 值返回给调用者（需要再弹出堆栈）。因此，或许当前的 PC 至少需要 1 个 store，2 个 load 和 2 个跳转！

精讲：软件检查溢出
大部分（但不是所有）程序都忽略整数算术溢出，因此 RISC-V 依赖于软件溢出检查。检查无符号加法的溢出只需要在指令后添加一个额外的分支指令 addu t0，t1，t2; bltu t0，t1，overflow。
对于带符号的加法，如果已知一个操作数的符号，则溢出检查只需要在加法后添加一条分支指令：addi t0，t1，+ imm; blt t0，t1，overflow。

这覆盖了常见的加立即数的情况。对于一般的带符号加法，我们需要在加法指令后添加三个附加指令，当且仅当一个操作数为负数时，结果才能小于另一个操作数，否则就是溢出。

```
add t0, t1, t2
slti t3, t2, 0          # t3 = (t2<0)
slt t4, t0, t1          # t4 = (t1+t2<t1)
bne t3, t4, overflow  # 若 (t2<0) && (t1+t2>=t1)
```

或者(t2>=0) && (t1+t2<t1)则为溢出

2.7 RV32I 无条件跳转

图 2.1 中的跳转并链接指令（jal）具有双重功能。若将下一条指令 PC + 4 的地址保存到目标寄存器中，通常是返回地址寄存器 ra（见图 2.4），便可以用它来实现过程调用。如果使用零寄存器（x0）作为目标寄存器，则可以实现无条件跳转，因为 x0 不能更改。像分支一样，jal 将其 20 位分支地址乘以 2，进行符号扩展后再添加到 PC 上，便得到了跳转地址。

跳转和链接（jalr）指令的寄存器版本同样是多用途的。它可以调用地址是动态计算出来的函数，或者也可以实现调用返回（只需 ra 作为源寄存器，零寄存器（x0）作为目的寄存器）。Switch 和 case 语句的地址跳转，也可以使用 jalr 零寄存器？？？？？？？

有什么不同？ RV32I 避开了错综复杂的程序调用指令，例如 x86-32 的进入和离开指令，或 Intel Itanium，Oracle SPARC 和 Cadence Tensilica 中的寄存器窗口。

2.8 RV32I 杂项

图 2.1 中的控制状态寄存器指令（csrrc、csrrs、csrrw、csrrci、csrrsi、csrrwi），使我们可以轻松地访问一些程序性能计数器。对于这些 64 位计数器，我们一次可以读取 32 位。这些计数器包括了系统时间，时钟周期以及执行的指令数目。

```
void insertion_sort(long a[], size_t n)
{
for (size_t i = 1, j; i < n; i++) {
long x = a[i];
for (j = i; j > 0 && a[j-1] > x; j--) {
a[j] = a[j-1];
}
a[j] = x;
}
}
```

图 2.5：C 语言版的插入排序。虽然简单，插入排序比复杂的排序算法有许多优势：对于小数据集来说，它是内存使用效率高、速度快，同时还有适应性强、稳定、能在线处理的特点。Gcc 编译器生成了以下四个数字的代码。我们设置优化标志以减少代码大小,因为这产生了最容易理解的代码。

ISA	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32I	RV32I+RVC

Instructions	19	18	24	24	20	19	19
Bytes	76	46	96	56	45	76	52

图 2.6: 插入排序在不同指令集下生成的指令数目以及指令大小。7 章介绍了 ARM Thumb-2, microMIPS 以及 RV32C 指令集。

在 RISC-V 指令集中，ecall 指令用于向运行时环境发出请求，例如系统调用。调试器使用 ebreak 指令将控制转移到调试环境。
fence 指令对外部可见的访存请求，如设备 I / O 和内存访问等进行串行化。外部可见指对处理器的其他核心、线程，外部设备或协处理器可见。 fence.i 指令同步指令和数据流。在执行 fence.i 指令之前，对于同一个硬件线程，RISC-V 不保证用存储指令写到内存指令区的数据可以被取指令取到。
第 10 章介绍 RISC-V 系统指令。

有什么不同？ RISC-V 使用内存映射 I / O 而不是像 x86-32 一样，使用 in，ins，insb，insw 和 out，out，outsb 等指令来进行 I/O。为支持字符串处理，RISC-V 实现了字节存取，而不是像 x86-32 那样实现了 rep，movs，coms，scas，lods 等 16 条特殊的字符串处理指令。

2.9 使用插入排序比较 RV32I，ARM-32，MIPS-32 和 x86-32 指令集
我们已经介绍了 RISC-V 基本指令集，并说明了与 ARM-32，MIPS-32 和 x86-32 相比，它做了哪些取舍。我们现在通过真实程序来进行一场直接的较量。图 2.5 显示了我们的基准测试——用 C 实现的插入排序。图 2.6 是一个表，它总结了在编译到不同 ISA 后，插入排序的指令数和字节数。
图 2.8 至 2.11 显示了插入排序编译生成的 RV32I，ARM-32，MIPS-32 和 x86-32 的汇编代码。尽管强调简单性，RISC-V 版本使用相同数目或更少的指令， 并且不同架构的代码大小非常接近。在此示例中，RISC-V 的比较、执行分支指令和 ARM-32 和 X86-32 中花式繁多的寻址模式以及入栈出栈指令一样，能够节省大量指令。

过去的错误 ARM-32 (1986) MIPS-32 (1986) x86-32 (1978)	RV32I (2011) 吸取的经验教训			
成本	必须支持整数乘除法	必须支持整数乘除法	8 位以及 16 位操作 必须支持整数乘除法	无 8 位、16 位操作 可选的整数乘除法支持(RV32M)
间接性	无零寄存器 条件指令执行 复杂的寻址模式栈操作执行 算术/逻辑指令中存在移位	立即数支持符号扩展及符号扩展 一些算术指令会抛溢出异常	无零寄存器 复杂的过程调用指令(enter/leave). 栈指令(push/pop). 复杂寻址模式 循环指令	寄存器 x0 专门用于存放常数 0 立即数只进行符号扩展 一种数据寻址模式 没有条件执行 没有复杂的函数调用指令以及栈指令 算术指令不抛异常 使用单独的移位指令来处理移位操作
性能	分支指令使用条件码 在不同格式的指令中，源和目的寄存器的位置不同 加载多个计算得到的立即数 PC 是一个通用寄存器	在不同格式的指令中，源和目的寄存器的位置不同	分支指令使用条件码 每个指令中最多只能使用两个寄存器	使用同一条指令实现比较及跳转（不使用条件码） 每条指令三个寄存器 不能一次 load 多个数据 不同指令格式中，源及目的寄存器字段位置固定 立即数是常数（不是由计算得出的） PC 不是通用寄存器
指令集结构与实现分离	？？？？ Exposes the pipe line length when writing the PC as a general	分支指令延迟槽 Load 指令延迟槽 乘除法使用单独的 HI，LO 寄存器	寄存器不是通用的(AX,CX, DX, DI, SI 有特殊用途)	分支指令没有延迟槽 Load 指令无延迟槽 通用寄存器

	purpose register? ?			
ISA 可扩展性	有限的指令码空间	有限的指令码空间		大量可用的指令码空间
程序大小	仅有 32bit 指令 (Thumb-2 是作为一个独立的 ISA)	仅 32bit 指令 (microMIPS 是作为一个独立的 ISA)	指令长度可用是不同字节，但这是一个很不好的选择	32 位指令+16 位 RV32C 扩展
编程、编译、链接的复杂性	仅 15 个寄存器内存数据必须对齐 不规则的数据寻址模式 不一致的性能计数器	内存数据必须对齐 不规则的数据寻址模式 不一致的性能计数器	仅 15 个寄存器 内存数据必须对齐 不规则的数据寻址模式 不一致的性能计数器	31 个寄存器 数据可用不对齐 PC 相对的数据寻址模式. 对称的数据寻址模式 定义在架构中的性能计数器

2.10 结束语

那些不记得过去的人，注定要重复过去。

图 2.7 使用第 1 章中的七个 ISA 设计指标来组织前面提到的一些过去的指令集中学习到的经验教训，并说明了这些经验教训对 RV32I 设计的积极影响。我们并不是手 RISC-V 是第一个拥有这些积极结果的 ISA。事实上，RV32I 从 RISC-I，它的曾祖父母[Patterson 2017]那里，继承了如下这些特性：

32 位字节可寻址的地址空间

•所有指令均为 32 位长

•31 个寄存器，全部 32 位宽，寄存器 0 硬连线为零

•所有操作都在寄存器之间（没有寄存器到内存的操作）

•加载/存储字加上有符号和无符号加载/存储字节和半字

•所有算术，逻辑和移位指令都有立即数版本的指令

•Immediates 总是符号扩展

•仅提供一种数据寻址模式（寄存器+立即数）和 PC 相对分支

•无乘法或除法指令

•一个指令，用于将大立即数加载到寄存器的高位，这样加载 32 位常量到寄存器只需要两条指令

RISC-V 的出现比过去的 ISA 晚了四分之一到三分之一个世纪后开始，这使它的设计者得以实践 Santayana 的建议，即借用之前指令集中好的设计，但不重复它们不好的瑕疵 - 包括 RISC-I 指令集中的瑕疵。另外 RISC-V 基金会将通过可选的指令集扩展的方式缓慢扩展着指令集，以避免出现困扰过去的成功指令集的疯狂地增量发展。

详述：RV32I 是否与众不同？

早期的微处理器有单独的浮点运算芯片，所以那些浮点运算指令是可选的。摩尔定律使得我们很快就将所有功能（包括浮点运算）都实现了在同一块芯片上，而且模块化在指令集中逐渐消失。
在更简单的处理器中只实现完整的指令集的子集，并利用软件异常来模拟未实现的指令已经有数十年历史了，在 IBM 360 的 44 型号和 Digital Equipment microVAX 中就已经实践了这种思路。 RV32I 的不同之处在于完整的软件堆栈只需要 RV32I 中的基本指令，因此，对于 RV32G 中未实现的指令，RV32I 处理器无需通过软件异常来进行模拟。在这方面，最接近 RISC-V 的 ISA 可能是 Tensilica Xtensa，它是专为嵌入式应用设计的。它的指令集包含有 80 条基础指令。并且它的指令集旨在被用户根据自己的需求扩展一些加速指令，以加速其应用程序。 与 Tensilica Xtensa 相比，RV32I 具有更简单的基础 ISA，具有 64 位地址版本，并且对超级计算机和微控制器都提供了针对性的指令集扩展。

图 2.7：RISC-V 架构师从过去的指令集错误中吸取的教训。通常的教训是避免过去的 ISA“优化”。经验和教训按照第一章中提出的七个 ISA 指标进行分类。？？？在成本，简单性和性能下列出的许多指令集特性可以互换？？？，因为这只是一个设计品味问题，但不管它们出现在哪里，它们都很重要。

```
# RV32I (19 instructions, 76 bytes, or 52 bytes with RVC)
# a1 is n, a3 points to a[0], a4 is i, a5 is j, a6 is x
0: 00450693 addi a3,a0,4 # a3 is pointer to a[i]
4: 00100713 addi a4,x0,1 # i = 1
Outer Loop:
8: 00b76463 bltu a4,a1,10 # if i < n, jump to Continue Outer loop
Exit Outer Loop:
c: 00008067 jalr x0,x1,0 # return from function
Continue Outer Loop:
10: 0006a803 lw a6,0(a3) # x = a[i]
14: 00068613 addi a2,a3,0 # a2 is pointer to a[j]
18: 00070793 addi a5,a4,0 # j = i
Inner Loop:
1c: ffc62883 lw a7,-4(a2) # a7 = a[j-1]
20: 01185a63 bge a6,a7,34 # if a[j-1] <= a[i], jump to Exit Inner Loop
24: 01162023 sw a7,0(a2) # a[j] = a[j-1]
28: fff78793 addi a5,a5,-1 # j--
2c: ffc60613 addi a2,a2,-4 # decrement a2 to point to a[j]
30: fe0796e3 bne a5,x0,1c # if j != 0, jump to Inner Loop
Exit Inner Loop:
34: 00279793 slli a5,a5,0x2 # multiply a5 by 4
38: 00f507b3 add a5,a0,a5 # a5 is now byte address oi a[j]
3c: 0107a023 sw a6,0(a5) # a[j] = x
```

```
40: 00170713 addi a4,a4,1 # i++
44: 00468693 addi a3,a3,4 # increment a3 to point to a[i]
48: fc1ff06f jal x0,8 # jump to Outer Loop
```

图 2.8：插入排序的 RV32I 代码如图 2.5 所示。十六进制的地址在左边， 接下来是十六进制的机器语言代码，然后是汇编语言指令，最后是评论以及注释。RV32I 分配两个寄存器用以指向 a[j]和 a[j-1]。RV32I 有很多寄存器，其中一些被 ABI 预留用于函数调用。与其他 ISA 不同，它会跳过保存和恢复寄存器值到内存的过程。虽然代码大小大于 x86-32，但使用可选的 RV32C 指令（请参阅 第 7 章）缩小了指令大小的差距。注意 RV32I 中的一条比较和分支指令顶得上 ARM-32 和 x86-32 比较所需的三条指令。

```
# ARM-32 (19 instructions, 76 bytes; or 18 insns/46 bytes with Thumb-2)
# r0 points to a[0], r1 is n, r2 is j, r3 is i, r4 is x
0: e3a03001 mov r3, #1 # i = 1
4: e1530001 cmp r3, r1 # i vs. n (unnecessary?)
8: e1a0c000 mov ip, r0 # ip = a[0]
c: 212fff1e bxcs lr # don't let return address change ISAs
10: e92d4030 push {r4, r5, lr} # save r4, r5, return address
Outer Loop:
14: e5bc4004 ldr r4, [ip, #4]! # x = a[i] ; increment ip
18: e1a02003 mov r2, r3 # j = i
1c: e1a0e00c mov lr, ip # lr = a[0] (using lr as scratch reg)
Inner Loop:
20: e51e5004 ldr r5, [lr, #-4] # r5 = a[j-1]
24: e1550004 cmp r5, r4 # compare a[j-1] vs. x
28: da000002 ble 38 # if a[j-1]<=a[i], jump to Exit Inner Loop
2c: e2522001 subs r2, r2, #1 # j--
30: e40e5004 str r5, [lr], #-4 # a[j] = a[j-1]
34: 1afffff9 bne 20 # if j != 0, jump to Inner Loop
Exit Inner Loop:
38: e2833001 add r3, r3, #1 # i++
3c: e1530001 cmp r3, r1 # i vs. n
40: e7804102 str r4, [r0, r2, lsl #2] # a[j] = x
44: 3afffff2 bcc 14 # if i < n, jump to Outer Loop
48: e8bd8030 pop {r4, r5, pc} # restore r4, r5, and return address
```

图 2.9：图 2.5 中插入排序的 ARM-32 代码。十六进制的地址在左边，接下来是十六进制的机器语言代码，然后是汇编语言指令，最后是注释、评论。由于寄存器不足，为了腾出两个空寄存器，以便之后重用，ARM-32 将两个寄存器的值保存到堆栈中（和返回地址放在一起）。它使用了一种将 i 和 j 缩放为字节地址的寻址方式。？？？鉴于一个分支有有可能在 ARM-32 和 Thumb-2 之间改变 ISA，bxcs 首先设置返回的最低有效位保存前地址为 0？？？。条件码使得我们在递减 j 后在检查它时可以少用一条比较指令，但在其他地方比较仍然需要三条指令。

```
# MIPS-32 (24 instructions, 96 bytes, or 56 bytes with microMIPS)
# a1 is n, a3 is pointer to a[0], v0 is j, v1 is i, t0 is x
0: 24860004 addiu a2,a0,4 # a2 is pointer to a[i]
4: 24030001 li v1,1 # i = 1
Outer Loop:
8: 0065102b sltu v0,v1,a1 # set on i < n
c: 14400003 bnez v0,1c # if i<n, jump to Continue Outer Loop
10: 00c03825 move a3,a2 # a3 is pointer to a[j] (slot filled)
14: 03e00008 jr ra # return from function
18: 00000000 nop # branch delay slot unfilled
Continue Outer Loop:
1c: 8cc80000 lw t0,0(a2) # x = a[i]
20: 00601025 move v0,v1 # j = i
Inner Loop:
24: 8ce9fffc lw t1,-4(a3) # t1 = a[j-1]
28: 00000000 nop # load delay slot unfilled
2c: 0109502a slt t2,t0,t1 # set a[i] < a[j-1]
30: 11400005 beqz t2,48 # if a[j-1]<=a[i], jump to Exit Inner Loop
34: 00000000 nop # branch delay slot unfilled
38: 2442ffff addiu v0,v0,-1 # j--
3c: ace90000 sw t1,0(a3) # a[j] = a[j-1]
40: 1440fff8 bnez v0,24 # if j != 0, jump to Inner Loop
44: 24e7fffc addiu a3,a3,-4 # decr. a2 to point to a[j] (slot filled)
Exit Inner Loop:
48: 00021080 sll v0,v0,0x2 #
4c: 00821021 addu v0,a0,v0 # v0 now byte address oi a[j]
50: ac480000 sw t0,0(v0) # a[j] = x
54: 24630001 addiu v1,v1,1 # i++
58: 1000ffeb b 8 # jump to Outer Loop
5c: 24c60004 addiu a2,a2,4 # incr. a2 to point to a[i] (slot filled)
```

图 2.10：图 2.5 中插入排序的 MIPS-32 代码。十六进制的地址在左边，接下来是十六进制的机器语言代码，然后是汇编语言指令，最后是注释。MIPS-32 代码中有三条 nop 指令，这增加了它的长度。两个是由于延迟分支，另一个是由于延迟加载。编译器无法找到有用的指令来填充延迟槽。延迟的分支也使代码更难理解，因为不管分支会不会跳转，延迟槽中的指令都会被执行。例如，地址 5c 处的最后一条指令（addiu）是 循环的一部分，尽管它是在分支指令之后。

```
# x86-32 (20 instructions, 45 bytes)
# eax is j, ecx is x, edx is i
# pointer to a[0] is in memory at address esp+0xc, n is in memory at esp+0x10
0: 56 push esi # save esi on stack (esi needed below)
1: 53 push ebx # save ebx on stack (ebx needed below)
```

```
2: ba 01 00 00 00 mov edx,0x1 # i = 1
7: 8b 4c 24 0c mov ecx,[esp+0xc] # ecx is pointer to a[0]
Outer Loop:
b: 3b 54 24 10 cmp edx,[esp+0x10] # compare i vs. n
f: 73 19 jae 2a <Exit Loop> # if i >= n, jump to Exit Outer Loop
11: 8b 1c 91 mov ebx,[ecx+edx*4] # x = a[i]
14: 89 d0 mov eax,edx # j = i
Inner Loop:
16: 8b 74 81 fc mov esi,[ecx+eax*4-0x4] # esi = a[j-1]
1a: 39 de cmp esi,ebx # compare a[j-1] vs. x
1c: 7e 06 jle 24 <Exit Loop> # if a[j-1]<=a[i],jump Exit Inner Loop
1e: 89 34 81 mov [ecx+eax*4],esi # a[j] = a[j-1]
21: 48 dec eax # j--
22: 75 f2 jne 16 <Inner Loop> # if j != 0, jump to Inner Loop
Exit Inner Loop:
24: 89 1c 81 mov [ecx+eax*4],ebx # a[j] = x
27: 42 inc edx # i++
28: eb e1 jmp b <Outer Loop> # jump to Outer Loop
Exit Outer Loop:
2a: 5b pop ebx # restore old value of ebx from stack
2b: 5e pop esi # restore old value of esi from stack
2c: c3 ret # return from function
```

图 2.11：图 2.5 中插入排序的 x86-32 代码。十六进制的地址在左边，接下来是十六进制的机器语言代码，然后是汇编语言指令，最后是注释。由于缺少寄存器，x86-32 将将两个寄存器保存在堆栈中，以便腾出这两个寄存器供后续使用。而且，本来在 RV32I 中可以分配到寄存器的两个变量（n 和指向[0]的指针），现在是保存在内存中的。它使用？？？Scaled？？？下标索引寻址模式，这对于访问[i]和[j]具有良好效果。 这里的 20 条 x32-86 指令中有 7 个是只有一个字节那么长，这使得对于这个简单的程序，x86-32 的代码规模很小。x86 有两个流行的汇编语言版本：Intel / Microsoft 和 AT&T / Linux。我们使用英特尔语法，部分原因是它将目的地放在左边，而源操作数放在右边，与 RISC-V，ARM-32 和 MIPS-32 的操作数顺序一致。而 AT&T 的操作数顺序则与之相反（并且对于寄存器操作数，需要在名字前加上%）。对于一些程序员来说，这看似微不足道的事情几乎是一个宗教问题。我们这里做出这样的选择，纯粹是因为教学方便，而非因为所谓“正统的信仰”。