第 3 章

RISC-V 指令集

The devil is in the fine print.

From the Web

魔鬼总是在文件细则中。

网络格言

3.1 RISC-V的历史

RISC-V 最早源自 2010 年夏天美国加州大学伯克利分校 Krste Asanović 教授主持的一个关于开源计算机系统的研究项目。该项目得到了美国国防高级研究计划局（Defense Advanced Research Projects Agency，DARPA）的资助，后来成为 RISC-V 的前身 [这里顺便提一句，国际互联网 Internet 的前身 ARPANET（Advanced Research Projects Agency Network，高级研究计划局网络）也是由 DARPA 资助的]。

RISC-V 中的字母 V 表示第五代的意思，所以发音时应该发作“RISC-Five”，表示它师承于伯克利分校之前开发的一系列 RISC 指令集。根据 RISC-V 的族谱，RISC-V 之前四代指令集都产生于 20 世纪 80 年代。当然，RISC-V 在其形成过程中，也从其他各种流行的指令集（MIPS、SPARC、ARM 等）中吸取了经验教训。

在 RISC-V 问世之际，移动计算主要由 ARM 处理器把持，而 Intel 公司的 x86 处理器则占据了大部分的桌面计算市场，RISC-V 的出现给这两大巨头带来了挑战。与这两大巨头的指令集不同的是，RISC-V 是一个自由和开放的指令集，它的标准化工作由 RISC-V 基金会主持，该组织目前有超过 100 个会员，并在不断扩大之中。对任何想要用 RISC-V 设计实现处理器的公司与个人，他们都不会受到来自 RISC-V 基金会的限制，也无须向 RISC-V 基金会支付授权费用。基金会各会员公司也承诺不会就 RISC-V 的基本议题向其他成员发起诉讼。

由于 RISC-V 没有上面提到的这些限制，因此很快得到了开源社区的大力拥护。面对 RISC-V 的攻城略地，ARM 也开始予以反击。2018 年夏，ARM 上线了一个名为 riscv-basics.com 的网站，对 RISC-V 发起舆论战。但是这种做法很快受到了来自各方的诟病，甚至连 ARM 自己的员工都对此做法表示不满。迫于各方压力，ARM 很快就关闭了该网站。

另外，为了促进 RISC-V 的产业化，RISC-V 的主要开发成员还于 2015 年成立

了一家叫 SiFive 的初创公司，向市场提供各类 RISC-V 的处理器内核，以及相关的软件工具和开发套件。

3.2 8051的CISC指令集与RISC-V的比较

3.2.1 8051 指令集简介

提到 RISC（Reduced Instruction Set Computer，精简指令集计算机），就必然也会提到 CISC（Complex Instruction Set Computer，复杂指令集计算机）。在许多嵌入式系统中得到广泛使用的 8051 单片机，便是 CISC 指令集的典型代表。笔者在开始设计 RISC-V 处理器之前，也曾做过一款 1T（单时钟周期）8051 处理器的设计，所以对这两类不同类型的指令集都有深入了解。这里笔者愿意将这些体会做个总结，并由此来反映 RISC-V 在技术设计上的优势。

可能有许多读者对 8051 单片机早已熟悉，该单片机是由美国 Intel 公司于 20 世纪 80 年代推出的一款 8 位单片机。由于该单片机方便易用，许多公司都推出了第三方的兼容设计。直到今天，8051 单片机依然被许多嵌入式系统所选用。

然而在 20 世纪 80 年代该单片机刚刚问世时，半导体的制造工艺还只能达到 μm 级，处理器所能达到的时钟频率偏低。而且当时硬件设计语言还处于起步阶段，也缺乏自动设计的工具，软件多以手工汇编编程为主。这就导致流水线设计的优势无法得到发挥，并且每条指令需要多个时钟周期才能完成。由于上述原因，当时的指令集设计往往具有以下特点：

（1）尽量在每条指令中实现更多的功能。例如 8051 的 CJNE 指令，就需要在一条指令中依次实现：

① 与累加器做减法。

② 修改进位标示。

③ 将结果做相等比较。

④ 根据比较结果决定是否跳转。

（2）指令集庞大，以实现更多的复杂功能。例如8051虽然是8位单片机，其指令集却包含高达255种不同的指令和格式。

（3）由于以上两点，导致变长指令的出现，以提高内存利用率。8051的指令就有单字节、双字节与三字节三种不同的种类，而且除了对指令解码以外，没有其他的手段帮助判定指令长度。

（4）寻址方式众多。例如在8051指令集中，对数值的操作包括如下方式：

① 立即数寻址。将常数包含在指令中。

② 直接寻址。将内存地址包含在指令中。

③ 间接寻址。将内存地址放入寄存器中，然后将寄存器地址包含在指令中。

④ 寄存器寻址。将操作数放入寄存器中，然后将寄存器地址包含在指令中。

由于众多的寻址方式，同一个功能在指令集中就可能对应多种指令格式。例如在8051指令集中，光是一个加法指令就有12种不同格式。类似地，跳转指令也存在多种的寻址方式和指令格式。

8051指令集的特点，很大程度上也代表了当时众多CISC指令集的共同特点。这种特点是与当时半导体制造水平和软件发展水平相匹配的。随着半导体加工工艺的不断进步和软件开发水平的提高，流水线和高时钟频率的设计开始在处理器设计中流行，汇编语言也开始被C/C++这类高级编程语言所替代。尽管8051是一个非常长寿的指令集，自问世近40年，依然被业界广泛采用，但是今天市面上出现的8051处理器，却早已和它们的祖先大不一样了。

8051的第一代产品，其时钟频率只有12 MHz，每个指令需要12个时钟周期才能完成。而今天我们所使用的8051处理器，都是增强型处理器，除了有更丰富的外围设备外，其增强之处主要表现在：

（1）时钟频率大幅提高。

（2）指令的吞吐率大幅提高，对大部分的指令，都可以做到在单个时钟周期内完成即我们通常说的 1T 8051。

（3）在软件上，支持 C 语言的开发环境。

换句话说，今天的增强型 8051 处理器，虽然其指令集还是 40 年前的那个指令集，但是其内部实现却早已经在原型基础上进行了 RISC 改造（实际上，类似的 RISC 改造也同样发生在 Intel 的 x86 处理器上）。

说明：由于指令集设计的缺陷，这种对 CISC 指令集的 RISC 实现不可避免地要在硬件上付出一定的代价。下面就以笔者主持设计的 PulseRain FP51-1T MCU 为例，对此具体加以说明。

3.2.2 8051 指令集对处理器设计的负面影响

PulseRain FP51-1T MCU 是美国 PulseRain Technology 公司推出的一款针对 FPGA 的 8 位微控制器，其内部的处理器内核是一个增强型 8051，可以对大部分的 8051 指令实现 1T 吞吐率，并且在 FPGA 上可以实现很高的时钟频率（在 Intel MAX10 C8 级器件上主频可以达到 100 MHz）。

8051 的流水线实现如图 3-1 所示，该处理器的内部有一个 5 级流水线，包含指令读取、指令解码（一）、数据内存读取、指令解码（二）和指令执行。尽管该处理器在 FPGA 上有优秀的性能表现，然而由于 8051 指令集本身的缺陷，使得设计者不得不以额外的逻辑资源为代价来换取更高的性能。最后的结果就是与同样时钟频率的 RISC-V 处理器相比，8 位 8051 内核居然比 32 位 RISC-V 内核消耗更多的逻辑资源，占用更大的芯片面积，而更大的芯片面积意味着更加耗电。对 FPGA 器件来说，这些还不是一个太大的问题，但是对专用芯片（ASIC），特别是移动设备的专用芯片来说，更多的耗电往往意味着更短的电池寿命（Battery Life），这可能也是 Intel x86 处理器始终无法在移动设备市场上打开局面的原因之一。

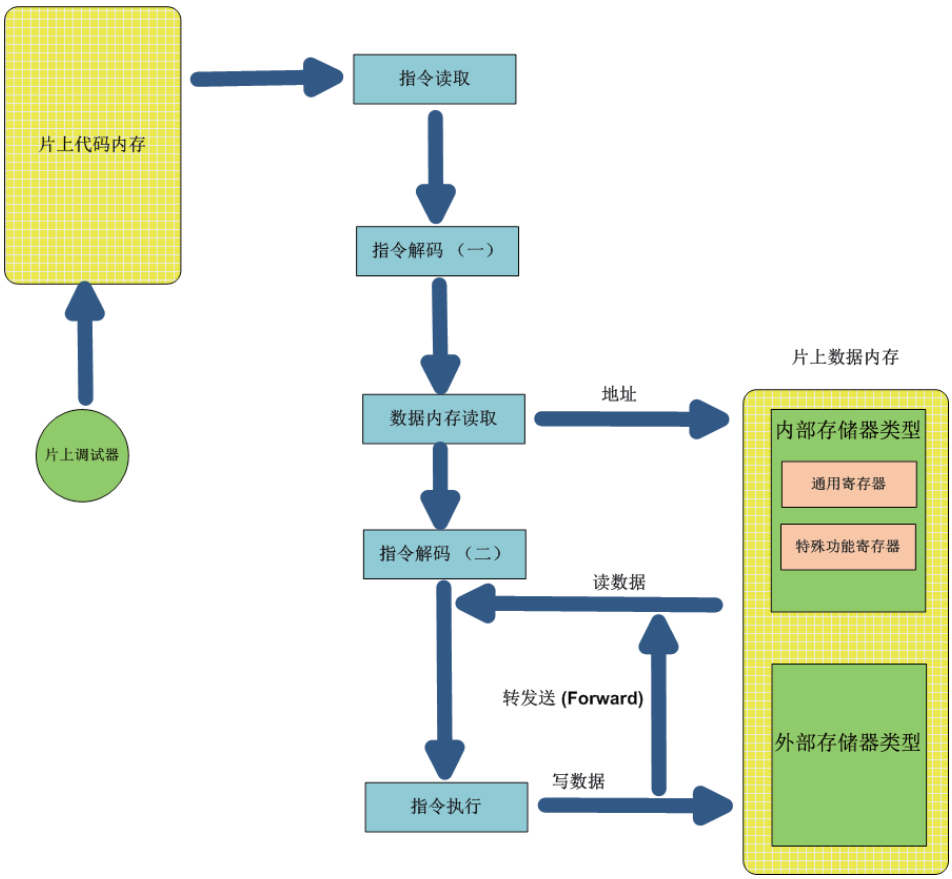


图3-1　8051 的流水线实现

具体来说，8051 指令集的特点会对处理器的 RISC 实现产生如下负面影响：

1）尽量在每条指令中实现更多的功能

为了在实现这些复杂功能的同时保持高吞吐率，流水线的设计者不得不花更多的时间规划流水线的各级。即便如此，有些指令依然无法实现单周期吞吐，例如上文提到的 CJNE 指令，就需要两个时钟周期。

另外，现代的 8051 处理器开发，早已经采用 C 语言代替了早期的汇编语言。而高级语言的编译器往往很难把这类复杂、多功能机器指令的威力全部发挥出来，有违当初指令集的设计初衷。

当然，指令集复杂这个特点也并非一无是处。由于 CISC 指令集的指令复杂，

也使得其代码密度（Code Density）一般要优于同等字宽的 RISC 处理器。

2）庞大的指令集

庞大的指令集必然导致指令的解码阶段变得更为复杂，需要耗费更多的逻辑资源。读者可能已经注意到，在图 3-1 所示的 5 级流水线中，指令集被分为两部分，对它们各自的解码分别占用了流水线的一级。这样设计的原因之一就是为了在庞大指令集下实现高吞吐率、高时钟频率，而不得不做出的妥协。同样时钟频率的 RISC-V 处理器，由于指令集比较精简，就无需做这样的妥协，从而大大节省了逻辑资源，简化了流水线设计。

3）由于以上两点，导致变长指令的出现，以提高内存利用率

8051 的指令有单字节、双字节和三字节三种不同的种类，除解码（Decode）外，没有其他的手段帮助判定指令长度。这种变长的指令结构，导致指令之间的边界很难判定，甚至有可能导致内存的非对齐读取（Unaligned Memory Access），从而对流水线的取指器（Instruction Fetch）设计带来挑战。

幸运的是，8051 的内存架构是哈佛架构，其代码与数据在不同的地址空间中分开存放。这就使得代码存储部分可以单独做一些优化设计。在图 3-1 中左边部分的片上代码内存，实际上被分成 4 个 8 位宽的存储体，这样对代码内存的一次读取就可以得到 4 字节，从而保证至少可以有一条完整的指令。然而即便如此，由于 8051 指令集没有其他辅助手段来帮助判定指令长度，为了确定指令的边界，8051 的取指器不得不为此花费比 RISC-V 更多的逻辑资源。

4）众多的寻址方式

由于 8051 存在众多的寻址方式，使得指令集中的许多指令都可以访问内存。这导致流水线的数据冲突（Data Hazard）很难判断，有时不得不通过硬件自动插入空操作（Null Operation，NOP）来保持数据的正确和完整。这样既消耗了逻辑资源，又降低了流水线的效率，从而对功耗和性能造成双重打击。

说明：虽然 8051 指令集有其历史局限性，但是 8051 处理器却由于其短小精悍、性价比高，一直为笔者所钟爱。其虽历四十载，依然廉颇未老，不乏拥趸。

3.2.3 RISC-V 指令集对处理器设计的正面影响

8051 指令集的缺陷，在 RISC-V 中都得到了避免，具体说明如下。

1. 引入指令长度编码

8051 指令集除了对指令解码以外，没有其他的辅助手段帮助判定指令长度，而 RISC-V 则可以通过指令的低位部分来判断指令的长度，被称为指令长度编码（Instruction Length Encoding）。图 3-2 展示了 16 ～ 64 位指令的编码方式。64 位以上的编码方式，可以在 RISC-V 官方标准中找到。

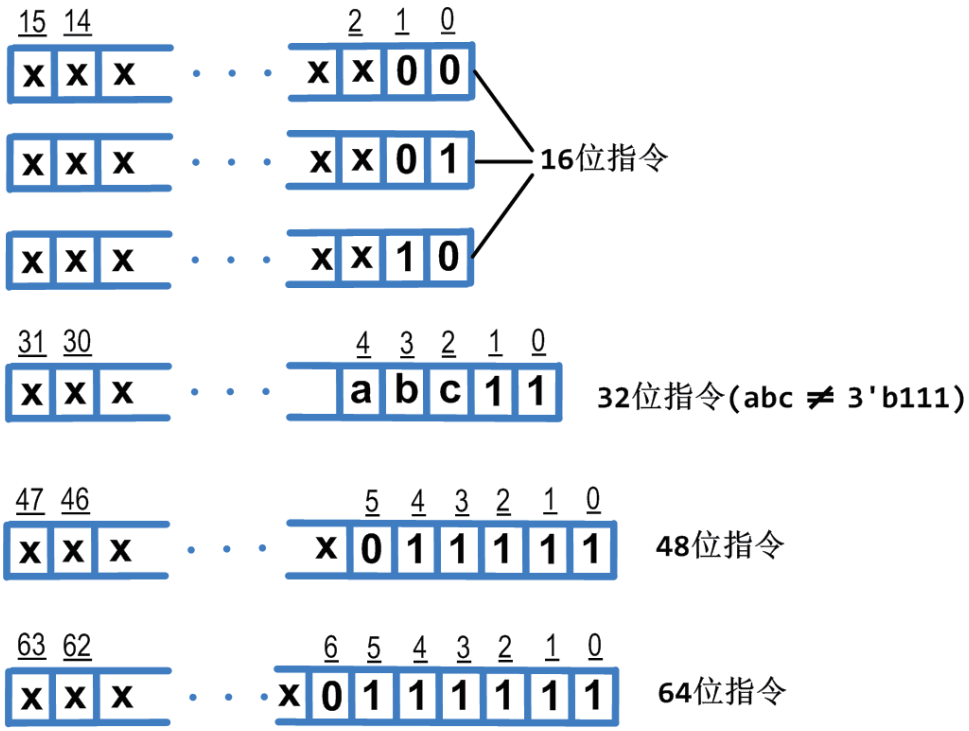


图3-2 RISC-V指令长度编码

指令长度编码的引入，大大简化了流水线取指器的设计，在取指时，硬件只需要集中优化边界对齐的内存读取就可以了。而对非对齐的访问，则可以通过产生异常，让软件处理器来处理。这样既节省了逻辑资源，又不影响处理器的性能。

2. 指令集规模较小，指令格式规整

尽管不是 8 位指令集，RISC-V 的指令集规模却比 8051 这样的 8 位指令集要小许多。RISC-V 的 32 位基础整数指令集只有 47 条指令，即使算上 8 条乘除法扩展指令，其指令总数也不到 8051 指令集规模的 1/4。指令集的小巧使得指令的解码器变得简单，更无须像图 3-1 中那样将指令集分成两部分来分别解码。

同时，RISC-V 的指令格式也非常规整，除了指令长度编码总是处在指令低位以外，在不同指令格式之间，操作码、源寄存器和目标寄存器总是位于相同的位置上。例如在 RISC-V 32 位基础整数指令集中（RV32I），操作码总是占用低 7 位，而源寄存器 1 和 2（rs1，rs2）则分别占据 15 ～ 19 位与 20 ～ 24 位。目标寄存器（rd）则占用 7 ～ 11 位（位索引以 0 为参考起点）。这种规整的指令格式进一步简化了指令解码器和指令执行器的设计。

3. 每条指令实现单个功能

与 CISC 指令集的设计思想截然相反，RISC-V 指令集中的每条指令只集中于优化实现单个的功能，这种将复杂任务通过多个单功能的指令来实现的做法也一直是 RISC 指令集的指导思想。因为这样可以简化流水线的设计，从而能实现更高的时钟主频，最终可以让 RISC 获得比 CISC 更佳的总体性能。

4. 内存访问只能通过 LOAD/STORE

与 8051 指令集中，具有众多的寻址方式不同，在 RISC-V 指令集中，对内存的读写只能通过 LOAD 指令和 STORE 指令实现。而其他的指令，都只能以寄存器为操作对象。没有了复杂的内存寻址方式，使得流水线对数据冲突（Data Hazard）可以及早做出正确的判断，并通过流水线各级之间的转送加以处理，而不需要插入空操作（NOP），极大提高了代码的执行效率。当然，这一特点也是 RISC 指令集的共有特点之一。

至此我们可以看到，CISC 指令集的那些历史局限性，在 RISC-V 指令集中都得到了突破。下面的章节会将 RISC-V 与其他的主流 RISC 指令集做对比，并展示其设计上的考量与取舍。

3.3 RISC-V与其他RISC指令集的比较

根据 Andrew Waterman 的博士论文（Andrew Waterman 是 RISC-V 创始人之一，Krste Asanović 教授的学生），RISC-V 在当初的设计目标中和嵌入式处理器相关的部分如下：

（1）指令集规模小，要求模块化并可扩展。

（2）指令集设计独立于具体的处理器实现。

（3）支持 16 位与 32 位混合编程，以提高代码密度。

（4）对 C/C++ 等编程语言提供硬件支持。

（5）将用户指令集（User-Level ISA）和特权架构（Privileged Architecture）做正交分割（Orthogonalize），即不同特权架构的处理器可以在应用二进制接口（Application Binary Interface，ABI）层面做到代码互相兼容。

基于以上的设计目标，RISC-V 对其他主流指令集的利弊都做了一番深入的研究，并做出了以下改进：

（1）将指令集分为基础指令集与扩展指令集。在处理器实现时，基础指令集是强制要求的，但扩展部分可选。

这样的安排在设计之初就为未来的历史演进留下了余地，避免了其他的指令级随着历史演进而愈来愈臃肿的问题（例如 ARM 指令集，在 ARMv7 中仅整数指令集部分就包含高达 600 条指令）。

（2）去除了对跳转指令延迟槽的支持。

延迟槽（Delay Slot）在许多通用的 RISC 指令集（如 MIPS 和 SPARC）中都包含，甚至在专用数字信号处理器（例如 TI 的 C5x DSP）上也有支持。

延迟槽的目的是提高流水线的利用率，当跳转发生时，硬件不得不清空流水线，重新设置指令取指器。而这时，如果那些紧随在跳转指令之后进入流水线的指令（即延迟槽中的指令）可以继续被完全执行，而不被丢弃，则跳转指令的开销就可以被降低。

延迟槽实际上是把一部分工作量转移给了软件，而且严重限制了处理器的实现方式，所以 RISC-V 对此做了舍弃。不过，RISC-V 的设计者将比较和跳转做了紧密的结合，对跳转指令的效率问题给出了另外一种解决方案。

（3）取消对寄存器窗口的支持。

在函数调用时，编译器往往会插入开场白（Prologue）和收场白（Epilogue）代码来传递参数，并保存寄存器到栈上。当函数嵌套层次比较深时，这种开场白和收场白代码的开销就显得很可观。为了降低函数调用中的这部分开销，在加州大学伯克利分校设计的第一代 RISC 处理器和后来 SUN 公司的 SPARC 处理器当中，都引入了寄存器窗口的设计，也就是在处理器中包含了多套通用寄存器。当函数调用发生时，主调函数（Caller）和被调函数（Callee）共享现有的这套通用寄存器，同时硬件还会给被调函数分配一套新的通用寄存器。这样在函数嵌套调用时，每次调用都无须再保存寄存器到栈上，从而大大降低了开场白和收场白的代码开销。

说明：这种设计的结果就是硬件开销变得很大，而且实际使用起来的效果并不理想，特别是当通用寄存器被耗尽时，其处理会变得非常麻烦和缓慢。因此 RISC-V 对此弃之不用，而代以类似 IBM S/390 中的毫码程序（Millicode Routine）的办法。

（4）支持 16 位指令扩展，并支持 16 位与 32 位混合编程。

与 ARM 等其他指令集不同的是，RISC-V 的 16 位指令只是一个扩展，并不是一个单独的指令集。而且每条 16 位指令都可以翻译成一条对应的 32 位指令，从而简化了指令解码器的设计。

3.4 RISC-V基础指令集（RV32I与RV32E）

RISC-V 的官方标准主要分成两部分：用户指令集（User-Level Instruction Set Architecture）与特权架构（Privileged Architecture）。

RISC-V 用户指令分类如图 3-3 所示，RISC-V 的用户指令集分为基础整数指令集（Base Integer Instruction Set）和扩展指令集（Extension）。根据处理器字长的不同，基础整数指令集又有 32 位、64 位和 128 位之分。而扩展指令集则有 16 位压缩指令（C，Compressed Instructions）、硬件乘除法（M，Integer Multiplication and Division）、取指隔离（Zifencei，Instruction Fetch Fence）等多种不同的扩展。考虑本书的主题主要是针对嵌入式系统开发，所以对 64 位和 128 位的指令将不予讨论。在本章节会主要讨论 RISC-V 32 位整数指令集（RV32I）和 32 位嵌入式指令。

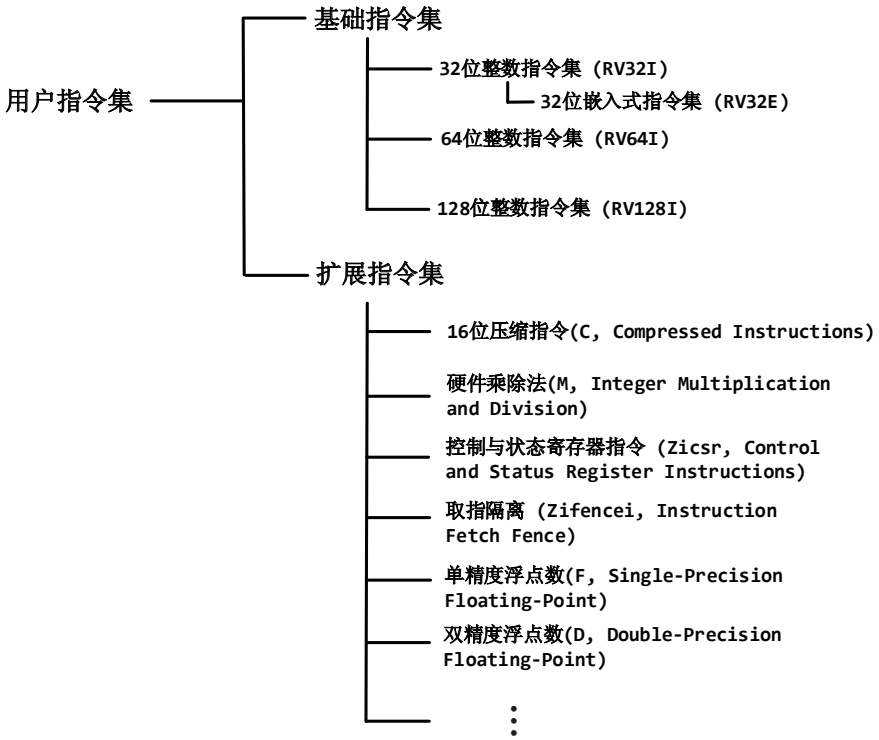


图3-3　RISC-V用户指令分类

3.4.1 RV32I 与 RV32E 基础指令集简介

在 RISC-V 标准刚刚推出时，32 位的基础指令集只有 RV32I，即 32 位整数指令集。后来考虑嵌入式系统资源稀缺的情况，又制定了 RV32E 基础指令集，这里的字母 E 即代表嵌入式（Embedded）。RV32I 和 RV32E 的主要区别是在通用寄存器的数量上，在 RV32I 中，总共有 32 个 32 位宽的通用寄存器，而 RV32E 只支持 16 个 32 位宽的通用寄存器。另外 RV32E 仅支持 M、A、C 三种指令扩展。

上述 RV32I 与 RV32E 的区别对 ASIC 的设计实现是有着实际意义的，在 ASIC 实现中，寄存器通常是通过触发器来实现的。对于面积优化的 RV32I 设计，移除 16 个通用寄存器大约可以节省 25% 的芯片面积；而对于 FPGA 实现来说，移除这 16 个寄存器并不能带来资源上面的节省。因为在 FPGA 当中，通用寄存器可以使用片上内存（Block Memory，BRAM）来实现。以 Intel MAX10 FPGA 为例，其 BRAM 是 M9K 类型，即每片 9Kb。而将 32 个 32 位通用寄存器用内存实现，只需要 1 024 位，会占用一片完整的 M9K BRAM。即使减少通用寄存器数量，其占用的 M9K 数量却依然还是一片，不会减少。基于这一点，本书将重点讨论 RV32I。

3.4.2 RISC-V 地址空间

RISC-V 的地址空间如图 3-4 所示。RISC-V 总共有 3 个独立的地址空间。

1. 内存地址空间

内存地址空间可以用来分配给代码、数据，或者作为寄存器的内存映射（Memory Mapped Registers）。在物理实现时，代码和数据可以共用存储（von Neumann，冯·诺依曼架构），也可以分别存储（Harvard，哈佛架构）。和其他的处理器一样，RISC-V 的处理器也是通过程序计数器（Program Counter，PC）来指示当前正在执行的指令地址的。

在寄存器的内存映射部分，大部分的外围设备寄存器都会被映射到这个空间，其中也包括机器模式的定时器（Mtime）和定时器触发值（Mtimecmp）。

2. 通用寄存器

RV32I 指令集包含 32 个通用寄存器，而 RV32E 只有 16 个这样的寄存器。

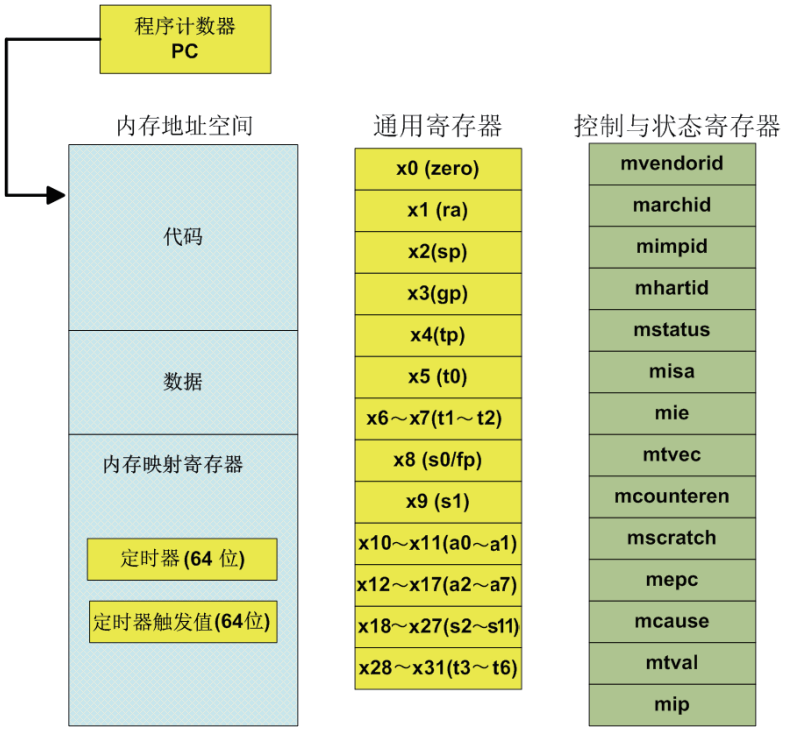


图3-4　RISC-V地址空间

3. 控制与状态寄存器

在 RISC-V 的特权架构部分还对控制与状态寄存器（Control Status Register，CSR）做了定义，并单独分配了 12 位的地址空间。在用户指令集中，则专门定义了 Zicsr 指令集扩展来对 CSR 进行操作。

3.4.3　RV32I 通用寄存器与函数调用约定

RV32I 基础指令集总共定义了 32 个 32 位的通用寄存器。它们分别被标记为 x0 ～ x31。其中零号寄存器 x0 是只读寄存器，其值永远为零。

RISC-V 的设计目标之一就是对 C/C++ 等高级语言提供硬件支持，并保持不同处理器之间在 ABI 层面的相互兼容。RISC-V 的用户指令标准还对函数调用约定（Calling Convention）做了标准化，也就是对函数调用时，哪些寄存器需要保存，还对寄存器具体的职能分配做了规定（因为 RV32E 只有 16 个通用寄存器，所以

RV32E 的 ABI 和 RV32I 的 ABI 不兼容）。在用汇编语言编写时，这 32 个寄存器的名称也根据其在调用约定中的职能而被重新命名。具体如表 3-1 所示。

表 3-1　函数调用约定的寄存器分配

寄存器名称	汇编名称	功能描述	调用返回后其值是否会保证不变
x0	zero	零寄存器	未定义
x1	ra	返回地址	否
x2	sp	栈指针	是
x3	gp	全局指针	未定义
x4	tp	线程指针	未定义
x5	t0	临时寄存器，或者用作替代链接寄存器（见后续章节详述）	否
x6	t1	临时寄存器	否
x7	t2	临时寄存器	否
x8	s0/fp	该寄存器需要被调函数予以保存，或也可用作调用栈的帧指针	是
x9	s1	该寄存器需要被调函数予以保存	是
x10~x11	a0~ a1	函数参数或返回值	否
x12~x17	a2~a7	函数参数	否
x18~x27	s2~s11	该寄存器需要被调函数予以保存	是
x28~x31	t3~t6	临时寄存器	否

注意：需要指出的是，除了硬件指令集会对函数调用约定产生影响外，高级语言的编译器也会对其有影响。

例如，对下面的函数：

```
void dummy(int a, int b, int c, int d, int e);
```

不同的编译器可能对函数参数压栈的顺序有不一致的理解。有的会从左到右，以 a、b、c、d、e 的顺序压栈；有的则反之，从右到左压栈。这种编译器在函数调用约定上的不一致在 C 和 C++ 语言混合编程时经常发生。当 C++ 模块直接调用 C 语言模块时，链接器会给出警告或报错。通常的做法是在调用时，把 C 语言的函数用 C++ 关键字 extern "C" 加以修饰说明，从而给编译器以明确的指示。

3.4.4 RV32I 指令格式

RV32I 基本指令格式如图 3-5 所示，RV32I 的基本指令格式只有 4 种，分别是寄存器类型（R-TYPE）、短立即数类型（I-TYPE）、内存存储类型（S-TYPE）、高位立即数类型（U-TYPE）。

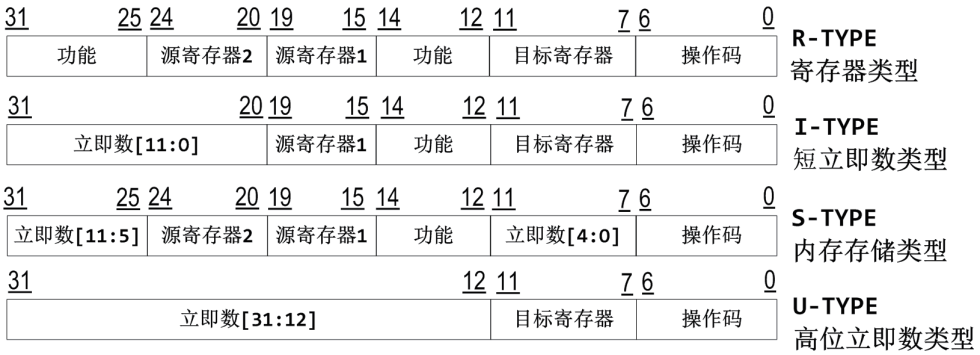


图3-5 RV32I基本指令格式

为了方便跳转指令，RV32I 还包含两种衍生格式 B-TYPE（Branch，条件跳转）与 J-TYPE（Jump，无条件跳转）。B-TYPE 衍生于 S-TYPE，B-TYPE 除了立即数的位排列与 S-TYPE 不一样外，其他的格式都与 S-TYPE 一样。J-TYPE 也是通过类似的方式衍生于 U-TYPE。用这种方式衍生新格式的目的是便于硬件产生目标地址。

上面这些格式，除 R-TYPE 外，其他的格式都需要把最高位（第 31 位）做符号扩展，以产生一个 32 位的立即数，作为指令的操作数。

图 3-5 所示的这些指令格式非常规整，其操作码、源寄存器和目标寄存器总是位于相同的位置上，简化了指令解码器的设计。

3.4.5 RV32I 算术与逻辑指令

1. 立即数指令

1）立即数加法

RV32I 立即数加法的定义如图 3-6 所示。这里特别要指出的是，和许多其他的

指令集不同，在 RV32I 当中并没有专门的状态寄存器和标记位来记录加法溢出。对加法溢出的判断是通过在加法指令之后安排比较和跳转指令来实现的。对符号数加法来说，只有正数加正数，或者负数加负数的情况才有可能发生溢出，所以溢出可以通过符号位（与零比较）来判断。而对无符号数来说，其和应该不小于被加数，所以溢出也可据此判断。

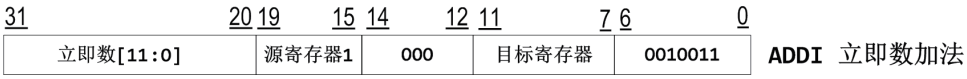


图3-6　立即数加法 ADDI

从 RV32I 对溢出标记的舍弃，也可以看出 RISC-V 非常强调指令集的简洁，极力减少不必要的硬件或指令，秉承了 RISC 指令集将复杂操作通过多条简单指令来实现的原始设计理念，可以说是不忘初心（这里实际上涉及一个更加复杂的话题，即 RISC-V 在设计时对条件编码（Condition Code）的舍弃）。具体的细节会在后续章节加以讨论。

从 ADDI 指令也可以衍生出空操作指令（NOP）。对 RISC-V 指令集，编译器一般会把 ADDI 中的立即数、源寄存器、目标寄存器都置为零，当作空操作指令使用。

2）立即数比较

RV32I 的立即数比较指令如图 3-7 所示。无论是符号数比较还是无符号数比较，图 3-7 中的 12 位立即数都应该通过符号位扩展变为 32 位立即数，然后根据指令 12 ～ 14 位中的功能定义，来决定比较方式符号数（SLTI）/无符号数（SLTIU）。

图3-7　立即数符号数/无符号数比较

比较时，如果源寄存器中的值小于该 32 位立即数，则将目标寄存器置为 1；否则置为零。由此，还可以通过 SLTIU 产生一个衍生指令：

```
SEQZ rd, rs     ⇔   SLTIU rd, rs1, 1
```

该 SEQZ 指令可以很方便地根据源寄存器中的值产生零位标志，而无须添加额外的硬件或指令。

3）立即数逻辑操作

RV32I 的立即数逻辑操作如图 3-8 所示。细心的读者可能已经注意到了，RV32I 中并没有定义逻辑反操作（NOT）。实际上，逻辑反操作可以通过 XORI 来实现（只需将 XORI 指令中的立即数置为全 1 即可）。

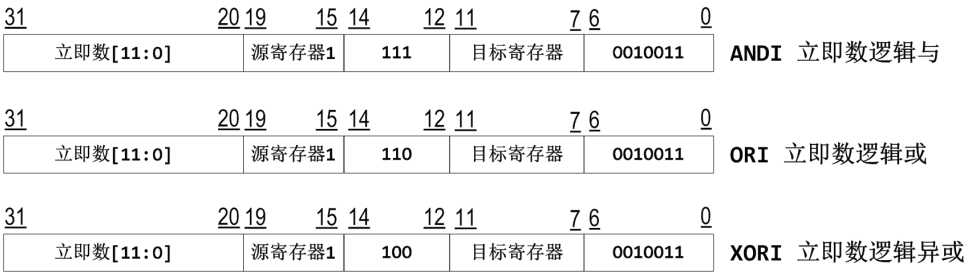


图3-8　立即数逻辑操作

4）立即数移位操作

RV32I 的立即数移位操作如图 3-9 所示。对逻辑左移操作，需要在最低有效位（Last Significant Bit，LSB）补零。对逻辑右移操作，需要在最高有效位（Most Significant Bit，MSB）补零。而对算术右移操作，则需要在高位做符号位扩展。同时，为了指令集的简洁，RV32I 中没有包括循环移位指令，因为循环移位可以通过移位指令和其他指令的组合来实现。

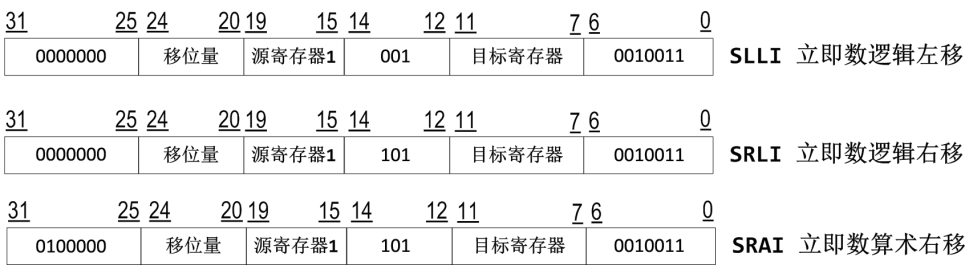


图3-9　立即数移位操作

SRLI 与 SRAI 的唯一编码区别是第 30 位。在处理器硬件实现上述移位指令时，硬件只需判断此位便可加以区分。然而，为了兼容性测试的需要，RISC-V 官方提供了一个非法指令（Illegal Instruction）的软件测试，当硬件遇到非法指令时产生异常。为了通过该测试，硬件设计时需要将图 39 中的高 7 位都考虑进去。

5）32 位立即数构建与地址生成

通过对图 3-5 的观察可以发现，U-TYPE 指令中的立即数有 20 位，而 I-TYPE 指令中的立即数有 12 位。32 位立即数可以通过一条 U-TYPE 指令和一条 I-TYPE 指令来联合构建。图 3-10 中的 LUI（Load Upper Immediate，高位立即数载入）指令即是为此目的而设计的，该指令会将其所携带的 20 位立即数载入目标寄存器的高位，而将目标寄存器的低 12 位置零。如果在 LUI 指令之后紧随一条 ADDI 指令，则可以继续构建目标寄存器的低 12 位，从而拼接出完整的 32 位立即数。该 32 位立即数也可以作为 32 位的地址使用。

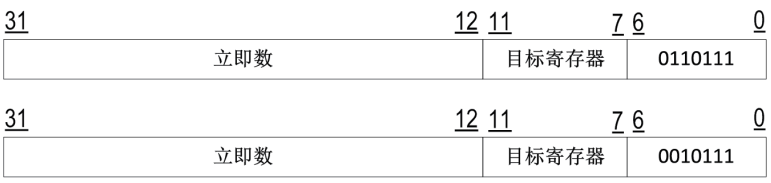


图3-10　立即数和立即地址构建指令

根据 RISC-V 这种“20+12=32”的立即数指令格式，可以把 RISC-V 的内存空间想象成一个分页的结构，其每个页面的大小为 2^{12}=4 096 字节，而页地址则有 20 位。图 3-10 中的 AUIPC（Add Upper Immediate to PC，高位立即数加 PC）指令就是为了移动页地址而设计的，和其他的 U-TYPE 指令一样，AUPIC 也会将其携带的 20 位立即数作为高位，而将低 12 位置零，以生成一个完整的 32 位数。然后该 32 位数会与当前指令计数器（32 位寄存器）的值相加，并将结果存入目标寄存器（RV32I 也用 PC 来存放当前活跃指令的内存地址）。

RISC-V 的设计目标之一就是为高级语言提供硬件支持，而有了 AUIPC 指令，可以很容易构建相对 PC 的寻址方式，从而实现独立于地址的代码（Position Independent Code，PIC）。如果要将相对于当前地址 0x1234 字节的内容载入 x4 寄

存器，则可以通过 AUIPC 指令用如下的代码实现：

```
aupic x4, 0x1        # PC + 0x1000  => x4
lw    x4, 0x234(x4)  # (x4 + 0x234) => x4
```

如果不使用 AUIPC 指令，则需要采用如下的变通办法：

```
jal x4, 0x4          # PC + 4  => x4,  同时也无条件跳转至 PC + 4
lui x5, 0x1          # 0x1000  => x5
add x4, x4, x5       # x4 + x5 => x4,  x4的值变为 PC + 0x1004
lw  x4, 0x230(x4)    # (PC + 0x1004 + 0x230) => x4
```

虽然上面的变通办法也可以达到目的，但是它有以下缺点：

（1）代码晦涩冗长，而且需要借助额外的寄存器 x5。

（2）跳转指令（Jump and Link，JAL）可能会误导流水线的运行，使得流水线执行清空动作。在某些采用 BTB（Branch Target Buffer，分支目标缓冲区）（用来记录之前发生过跳转的指令的 PC 值和目标地址）来做跳转预测的处理器上，上面的跳转指令会在 BTB 中留下记录条目，但对跳转预测却并无帮助，因为目标地址等同于下一条顺序执行的指令地址。

由此可见，AUIPC 的引入极大地减轻了编译器的负担。

注解：跳转预测

为了进一步提高流水线的运行效率，处理器的设计者往往会在取指器中加入跳转预测的模块。跳转预测常用的模块部件有：

① BHT (Branch History Table，跳转历史表)，利用 PC 的末几位地址，来记录之前发生的跳转历史，作为动态跳转预测的依据。BHT 有时也叫 BHB (Branch History Buffer)。

② BTB，当跳转发生时，BTB 会记录下跳转指令的地址和其目标地址（Target Address）。当该 PC 值再次被遇到时，则可以将之前记录的目标地址

作为指令读取的地址。一般来讲，最终真正的目标地址都会在流水线比较靠后的阶段才能确定，如果最后确定的目标地址与之前记录的地址不一致，则宣告跳转预测失败，清空流水线重新取指令。否则，预测成功，流水线不会有停顿。

③ RAS (Return Address Stack，返回地址栈)，RAS 与 BTB 有些类似，RAS 主要是用来对跳转返回指令提供预测地址。当程序遇到函数调用指令时，会把函数的返回地址存入 (push) 到 RAS 中。当取指器认为当前指令是一条跳转返回指令时，就会做退栈动作 (pop)，并把之前存在 RAS 栈顶的地址作为下一条指令的读取地址。之后，在流水线比较靠后的阶段，当最终的目标地址被确定时，如果目标地址与 RAS 提供的预测地址不吻合，则预测失败，清空流水线重新取指令；否则，预测成功，流水线不会有停顿。

2. 寄存器 - 寄存器指令

寄存器 - 寄存器指令包括加减法（见图 3-11）、数值比较（见图 3-12）、逻辑操作（见图 3-13）与移位操作（见图 3-14）。这些指令的功能和前面的立即数指令相似，唯一的区别是立即数指令中的立即数被替换为源寄存器 2（寄存器 - 寄存器指令中包含减法指令，而立即数操作则没有定义减法）。

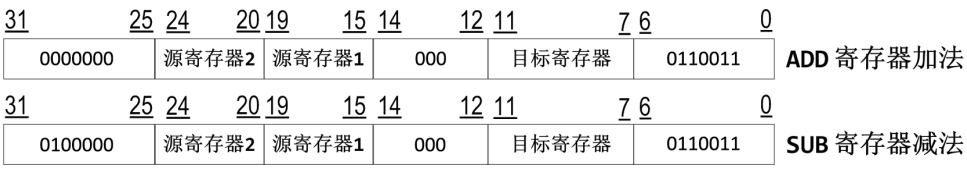


图3-11　寄存器加减法

图3-12　寄存器数值比较

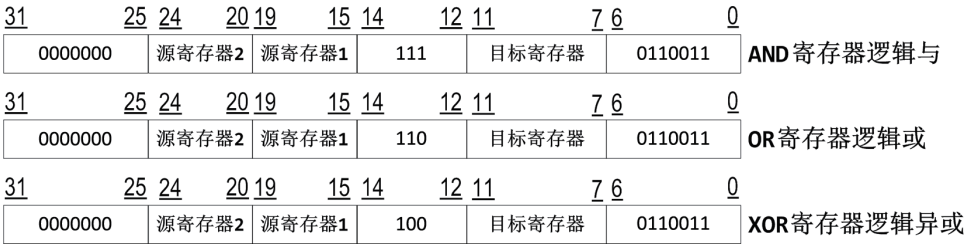


图3-13　寄存器逻辑操作

图3-14　寄存器移位操作

由于这些相似性，本书对寄存器 - 寄存器指令不再赘述，读者如有疑问，可以查阅 RISC-V 官方标准中的相关部分。

3.4.6　控制转移指令

RISC-V 中的转移控制指令（Control Transfer Instructions）主要包括以下两类：

（1）无条件跳转（Unconditional Jump）。

（2）有条件跳转（Conditional Branches）。

不过和其他指令集相比，RISC-V 的跳转指令设计得非常有特色：

（1）RISC-V 中并没有专门的函数调用指令，函数的调用是通过设置跳转指令中的寄存器来实现的，这样减小了指令集的规模。

（2）RISC-V 将比较和跳转相结合，而且利用条件跳转来对“溢出”“零

值”等做出判断。RISC-V 本身并不专设状态寄存器来做溢出位、加法进位、零值标识等，从而简化了处理器结构。

（3）RISC-V 为无条件跳转指令专门定义了一种 J-TYPE 的指令格式，而该格式衍生于 U-TYPE。J-TYPE 只是在 U-TYPE 的基础上，对立即数的某些位做了位置调整。对有条件跳转指令，RISC-V 也做了类似的处理，在 S-TYPE 的基础上衍生出了 B-TYPE。这些细微的调整，可以在一定程度上降低硬件设计的开销。

1. 无条件跳转指令 (Unconditional Jump)

1）直接跳转 JAL（Jump and Link，跳转与链接）

JAL 指令如图 3-15 所示。RISC-V 为 JAL 指令专门定义了 J-TYPE 格式。将图 3-15 和图 3-5 中的 U-TYPE 比较，可以发现除了立即数的某些位做了位置调整以外，其他都没有变化。JAL 指令会把其携带的 20 位立即数做符号位扩展，并左移一位，产生一个 32 位的符号数。然后将该 32 位符号数和 PC 相加来产生目标地址（这样，JAL 可以作为短跳转指令，跳转至 PC ± 1 MB 的地址范围内）。

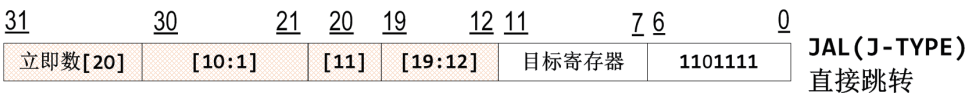


图3-15　JAL 指令

同时，JAL 也会把紧随其后的那条指令的地址存入目标寄存器中。这样，如果目标寄存器是零，则 JAL 等同于 GOTO 指令；否则，JAL 可以实现函数调用的功能。

2）间接跳转 JALR（Jump and Link Register，跳转与链接寄存器）

JALR 指令如图 3-16 所示。JALR 指令会把所携带的 12 位立即数和源寄存器相加，并把相加的结果末位清零，作为新的跳转地址。同时，和 JAL 指令一样，JALR 也会把紧随其后的那条指令的地址存入目标寄存器中。

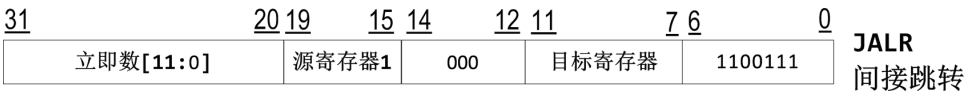


图3-16　JALR指令

JAL 指令受其指令格式所限，只能实现 PC ± 1 MB 的短跳转。而通过如下的指令序列将 JALR 指令和 LUI/AUIPC 指令相结合，可以实现全地址空间的跳转：

```
lui ra, 立即数(20位)
jalr ra, ra, 立即数(12位)
```

或者

```
auipc ra, 立即数(20位)
jalr ra, ra, 立即数(12位)
```

2. 动态预测返回地址

尽管 RISC-V 指令集本身并没有对 JAL 或 JALR 中目标寄存器的取值做出限制，但是根据前面提到的函数调用约定（Calling Convention），JAL/JALR 常用的目标寄存器有 x1（ra，返回地址）和 x5（t0，替代链接寄存器）。对普通的函数调用，x1（ra）会被用来存放返回地址。然而，表 3-1 的调用约定中还定义了 x5（替代链接寄存器），其作用是：

（1）在使用压缩扩展指令集（Compressed Instruction Extension）时，方便将函数调用的开场白和收场白作为公共的函数调用，从而到达提高代码密度（Code Density）的目的。对 x5（替代链接寄存器）的具体用法，会在后续有关“压缩指令扩展”的章节做详细讨论。

（2）对于协程（Coroutine）这种需要实现堆栈切换的情况，利用 x5（替代链接寄存器）可以帮助实现快速的 Coroutine 调用与返回（见表 3-2）。

表 3-2　JALR 指令的 RAS 操作

目标寄存器(rd)	源寄存器(rs1)	rd ?= rs1	RAS操作	备　注
(rd ≠ x1) && (rd ≠ x5)	(rs1 ≠ x1) && (rs1 ≠ x5)	无关	无	非函数调用指令，亦非调用返回指令

续表

目标寄存器(rd)	源寄存器(rs1)	rd ?= rs1	RAS操作	备　注
(rd ≠ x1) && (rd ≠ x5)	(rs1 = x1) || (rs1 = x5)	无关	返回地址出栈	调用返回指令
(rd = x1) || (rd = x5)	(rs1 ≠ x1) && (rs1 ≠ x5)	无关	返回地址入栈	函数调用指令
(rd = x1) || (rd = x5)	(rs1 = x1) || (rs1 = x5)	(rd ≠ rs1)	在同一指令周期中，既出栈又入栈	协程
(rd = x1) || (rd = x5)	(rs1 = x1) || (rs1 = x5)	(rd = rs1)	返回地址入栈	宏操作合并

RISC-V 的函数调用约定将 JAL/JALR 在调用和返回时可使用的寄存器限制在 x1 或 x5 上，为动态预测返回地址创造了条件。在 3.4.5 节的注解部分，提到了跳转预测常用的三种模块：BHT、BTB 与 RAS。将 RAS 与无条件跳转指令相结合，则可以很好地实现返回地址的动态预测，具体做法如下：

（1）JAL 指令的 RAS 操作。

根据函数调用约定，当 JAL 的目标寄存器值是 x1 或者 x5 时，可以判定其是在做函数调用。这时可以把返回地址（紧随其后的那条指令的地址）压入 RAS。

（2）JALR 指令的 RAS 操作。

JALR 指令既可以作为函数调用指令，又可以作为调用返回指令，其 RAS 的操作方式如表 3-2 所示。

注解：协程

在多任务处理中（Multi-task），协程（Coroutine）可以被看作一种合作式（Collaborative），非抢断式（Non-preemptive）的线程（Thread）。图 3-17 展示了两个协程互相调用的情形，其中每一个节点都是一个退让（Yield）操作。这样的话，协程 A 和 B 会轮流按照 1、2、3、4、5 的顺序执行。

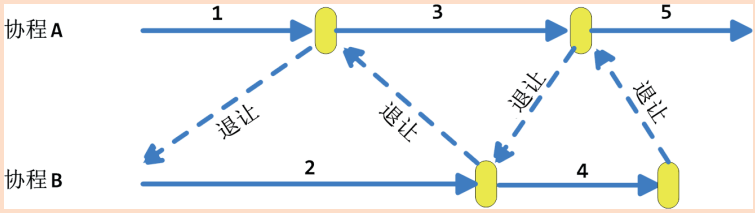


图3-17　协程

为了实现协程，往往需要做堆栈的切换。把 alternative link register和表 3-2 相结合，把 x1(ra) 与 x5（替代链接寄存器）分别分配给图 3-17 中的协程 A 和 B，则可以提高跳转预测的准确率，以实现协程的快速调用和返回。在图 3-17 中除第一个退让外，其他的退让操作既可以看作从一个协程返回另外一个协程，又可以看作从一个协程调用另外一个协程。这就是为什么表 3-2 中的第 4 行既有出栈操作，又有入栈操作。

注解：宏操作合并

在处理器设计中，可以设计一个硬件模块，在指令解码阶段之前，通过对已取指令序列的观察，将其中某些前后相邻的简单指令合并为一条复杂的指令（可以是处理器内部定义的专门指令），以提高指令执行效率，这种做法称为宏操作合并（Macro-Op Fusion）。3.4.6 节最后所提到的 LUI+JALR 或 AUIPC+JALR 指令序列，就是被宏操作合并的典型例子。表 3-2 中的最后一行，就是为了提高这两个指令序列被宏操作合并之后的跳转预测准确率而设计的。

这里要注意的是，宏操作合并是处理器设计实现的方法，与指令集本身无关。另外，Intel 公司是宏操作合并技术的发明者，对该项技术拥有专利。

3. 条件编码

在讨论有条件跳转指令（Conditional Branches）之前，有必要先介绍一下条件编码（Condition Code）的概念。实际上，以笔者多年从事嵌入式设计的经验来看，由于各种技术指标之间的相互抵触（例如数字设计的 AT^2 边界限制），设计的过程更多的是在各个技术方案之间寻求妥协和取舍，以求达到总体最优的过程。RISC-V 的设计者也是基于各种综合的考量，最终决定舍弃条件编码，而代之以将

条件判断与跳转指令直接相结合的方案，具体如下。

在传统的 RISC 设计中，设计者往往会安排一个状态寄存器，在其中放置各类的标志位 [例如溢出标志(Overflow)、零标志(Zero Flag)、进位标志(Carry)等]。在某条指令更改了这些标志位之后，后续的指令会根据更改后的标志位来决定是否需要被执行。为此，这些标志位会被编码（即条件编码），并成为指令编码的一部分。

在 ARMv8 指令集中，就在指令编码中专门分配了 4 位，用来做条件编码，以表示比较相等（结果为零）、溢出等状态。对“ if (a == 10) {…}”这样的高级代码，编译器的常用处理方式是在一个计算指令之后跟随一个条件执行指令，如下面的伪代码所示：

```
subtraction (register - 10)        # 减法，结果可以被丢弃
branch if zero flag is not set     # 如果不相等则跳转
```

注意：上面的代码序列实际上包含了状态寄存器的使用。第一条的减法指令会影响状态寄存器中的零标志位，而第二条的跳转指令中的条件编码包含有零标志判断，但是不包含对通用寄存器的读取。

使用条件编码的优点是可以让条件跳转指令变得相对比较简单（不涉及对通用寄存器的读取，只涉及标志位），这样跳转条件就可以在流水线比较靠前的阶段被判断出来。但是这样做的缺点是条件编码需要在指令编码中占用一定的位，而且需要在条件跳转指令之前安排另外一条指令用来设置标志位，降低了代码密度。同时，硬件也需要有专门的状态寄存器，记录各种标志位。

而 RISC-V 的设计者则另辟蹊径，将上面标志位设置指令合并到条件跳转指令中去，在条件跳转指令中直接读取通用寄存器做判断，这种做法的优点是：

- 没有条件编码，节省下的位可以被指令编码派作其他用途，从而减小指令集规模。
- 只要一条指令就可以实现上面需要两条指令来实现的功能，提高了代码密度。
- 不需要专门的状态寄存器来记录各种标志位，降低了硬件的开销。

这种做法的缺点是：由于需要在条件跳转指令中直接读取通用寄存器，跳转条件要在流水线中比较靠后的阶段才能判定。RISC-V 的设计者认为，目前跳转预测的准确度（预测跳转是否发生）和精确度（预测跳转目标地址）都已经大幅提高，将跳转条件的判定在流水线中后移并不会给性能带来太大的负面影响。权衡利弊，RISC-V 最终还是舍弃了条件编码。

4. 有条件跳转指令

RV32I 的有条件跳转指令总共有 6 条，其定义如图 3-18 所示（图中网格标记的深色部分为立即数）。RISC-V 为有条件跳转指令专门定义了新的指令格式 B-TYPE，其衍生于图 3-5 中的 S-TYPE。通过将图 3-18 与图 3-5 中的 S-TYPE 作比较，可以发现 B-TYPE 只是在 S-TYPE 的基础上对立即数的某些位做了顺序调整，其原因会在后续章节讨论。

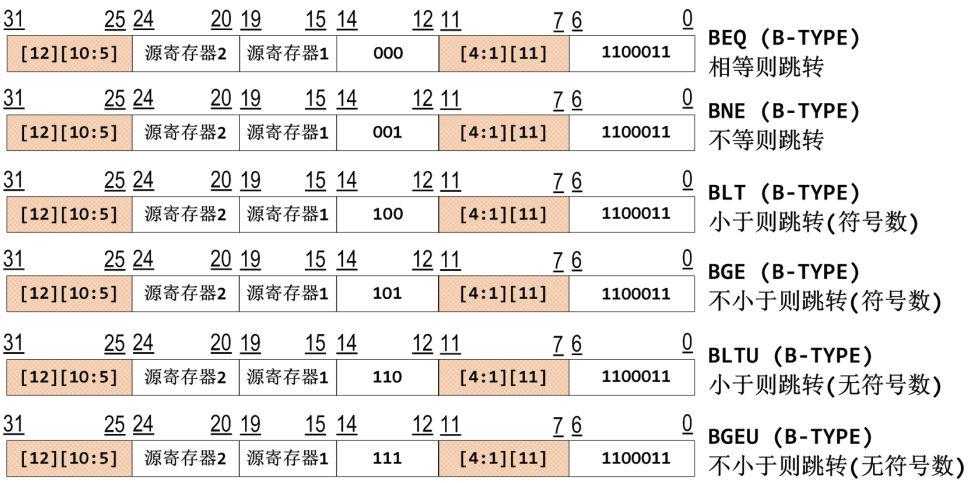


图3-18　有条件跳转指令

有条件跳转指令会将源寄存器 1 中的值和源寄存器 2 中的值做比较，如果满足表 3-3 中的比较条件则跳转，其目标地址产生的办法如下：

有条件跳转指令会把其携带的 12 位立即数做符号位扩展，并左移 1 位，来产生一个 32 位的立即数。该立即数会和当前的程序计数器值相加，来产生最终的目标地址。这样的话，有条件跳转指令能跳转至 PC ± 4 KB 的地址范围内。

表 3-3　有条件比较跳转指令

有条件跳转指令	跳 转 条 件	备　　注
BEQ	(源寄存器 1) = (源寄存器 2)	—
BNE	(源寄存器 1) ≠ (源寄存器 2)	—
BLT	(源寄存器 1) < (源寄存器 2)	有符号数比较
BLTU	(源寄存器 1) < (源寄存器 2)	无符号数比较
BGE	(源寄存器 1) ≥ (源寄存器 2)	有符号数比较
BGEU	(源寄存器 1) ≥ (源寄存器 2)	无符号数比较

另外，将源寄存器 1 和源寄存器 2 对换，则可以从表 3-3 里的最后 4 条指令产生另外 4 条衍生指令，即 BGT（符号数比较，大于则跳转）、BGTU（无符号数比较，大于则跳转）、BLE（符号数比较，不大于则跳转）、BLEU（无符号数比较，不大于则跳转）。

5. 静态跳转预测

RISC-V 的设计者没有使用条件编码，而是选择让有条件跳转指令（Conditional Branches）直接对寄存器进行比较，导致跳转条件要在流水线中比较靠后的阶段才能判定。为了减少这种设计策略对处理器性能的负面影响，RISC-V 对跳转预测非常重视。除了动态跳转之外，针对有条件跳转指令，RISC-V 还对软硬件设计者做出了如下的建议，以帮助提高静态跳转的准确率：

（1）在软件设计中不要用条件永远为真的有条件跳转指令（例如 BEQ x0，x0）来代替无条件跳转指令，以减少对分支预测不必要的干扰。

（2）对于有条件跳转指令，其顺序分支应存放比较常用的代码，而其跳转发展应存放不太常用的代码。对高级语言的循环指令来说，这意味着循环体应放在顺序分支上。

（3）和高级语言的循环指令相关的另外一条对处理器硬件设计的建议就是：对于向前跳转的指令（目标地址大于 PC），应静态预测该跳转不发生。对向后跳转的指令（目标地址小于 PC），应静态预测该跳转会发生。如果和上一条建议相结合使用的话，则这种静态预测的策略非常符合大多数循环体

的实际状况（重复 N 次，然后退出循环）。软件设计者在做优化设计时，也应该将此条考虑在内。对有条件跳转指令，由于目标地址的产生非常简单直接（PC+ 立即数），所以不需要很大的硬件开销就可以实施这条指令。

6. B-TYPE 和 J-TYPE 中的立即数

RV32I 中为 JAL 指令和有条件跳转指令分别定义了 J-TYPE 与 B-TYPE 格式。这两种格式实际上各自衍生于 U-TYPE 与 S-TYPE。J-TYPE 除了立即数的位排列与 U-TYPE 不一样外，其他的格式都与 U-TYPE 一样。B-TYPE 也是通过类似的方式衍生于 S-TYPE。RISC-V 的设计者做出这种安排的主要原因是为了减小硬件的开销，现说明如下。

为了实现 16 位地址的边界对齐，JAL 指令和有条件跳转指令都需要将其所携带的立即数做左移一位的操作。如果不在指令格式上做些处理的话，那么最后生成的 32 位立即数，其位的位置和 U-TYPE 与 S-TYPE 相比，会因位置的偏差而没有对齐，这就会增加指令解码器的硬件开销。而衍生定义 J-TYPE 和 B-TYPE 的目的，就是为了将移位后的大部分位保留在原来的位置上。

将图 3-5 中的 I-TYPE、S-TYPE 和 U-TYPE 与图 3-15 及图 3-18 做比较，可以将处理器中有关 32 位立即数生成的各位组成归纳如表 3-4 所示。

表 3-4　立即数位组成

立 即 数 位	对应的指令位
[31]	位 31
[30:20]	I-TYPE, S-TYPE, B-TYPE, J-TYPE：位 31 的符号位扩展
	U-TYPE:[30:20]
[19:12]	I-TYPE, S-TYPE, B-TYPE：位 31 的符号位扩展
	U-TYPE，J-TYPE:[19:12]
[11]	I-TYPE, S-TYPE：位 31
	B-TYPE：位 7
	J-TYPE：位 20
	U-TYPE：数值零

续表

立 即 数 位	对应的指令位
[10:5]	I-TYPE，S-TYPE，B-TYPE，J-TYPE:[19:12]
	U-TYPE: 数值零
[4:1]	I-TYPE，J-TYPE:[24:21]
	S-TYPE，B-TYPE:[11:8]
	U-TYPE: 数值零
[0]	I-TYPE: 位 20
	S-TYPE: 位 7
	B-TYPE, U-TYPE，S-TYPE: 数值零

从表 3-4 可以看出，在最终需要产生的 32 位立即数中，有多于 80% 的位（26/32）其来源最多只有两个，从而极大降低了立即数生成的硬件开销。而这一优势，都是拜衍生格式 B-TYPE 与 J-TYPE 所赐。

3.4.7 内存载入与存储指令

1. 小端模式

对于字长超过 8 位的系统，其数据在内存中的存放方式主要有两种：一种是小端模式（Little Endian，小字节序或低字节序），即对每个字长的数据，将低位的字节存放在内存地址的低位部分；另外一种存放方式是大端模式（Big Endian，大字节序或高字节序），即对每个字长的数据，将高位的字节存放在内存地址的低位部分。虽然从技术角度来说，这两种存放方式各有利弊，但是考虑到目前大部分的商用系统都采用小端模式，所以 RISC-V 的设计者也决定将小端模式定为 RISC-V 的标准模式。

2. 载入指令

和传统的 RISC 指令集一样，RISC-V 避免了 CISC 指令集中那种通过多种寻址方式访问内存的做法，而将内存的访问仅限于载入（Load）和存储（Store）指令。RISC-V 中的内存载入指令包括单字节、双字节和 32 位三种字长。同时对单字节和双字节指令，根据其符号位处理方式的不同（符号位扩展或零位填充），又可分为符号数载入与无符号数载入。为此，RISC-V 中定义了 5 条不同的载入指令，如图 3-19 所示。

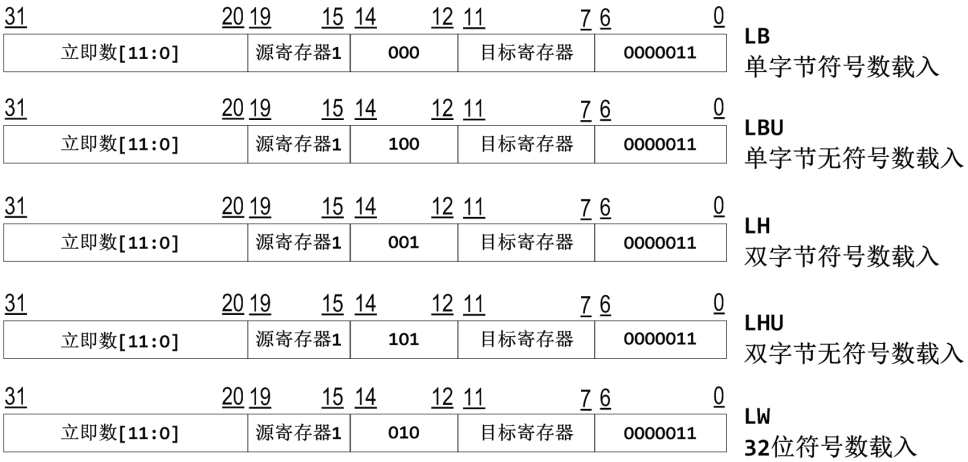


图3-19　内存载入指令

在图 3-19 所示的内存载入指令中，需要将其所携带的 12 位立即数作符号扩展，变为一个 32 位的符号数，然后将该 32 位符号数同源寄存器相加，以产生内存的读取地址。内存读取完成后，从内存读取的数据最终会被存入目标寄存器。

内存的读取可能会导致硬件异常，具体的细节会在后续章节讨论。

3. 存储指令

和载入指令相类似，RISC-V 中还根据字长的不同，分别定义了三种内存存储指令，如图 3-20 所示。其内存写入地址的产生也和载入指令类似，即将其所携带的 12 位立即数作符号扩展并同源寄存器 1 相加。而需要写入内存的数据则存放于源寄存器 2 当中。

图3-20　内存存储指令

同内存读取一样，内存的写入操作也可以导致硬件异常。

4. 非对齐的内存读写 (Misaligned Memory Access)

和许多其他的系统一样，内存读写地址都是字节地址，如果软件操作不慎，就可能引起非对齐的内存读写，导致内存操作无法在一个读写周期内完成。对于这种情况，RISC-V 中并没有强制规定应对之道，而是交由处理器的设计者来自行决定。通常来说，处理器的设计者可以有硬件处理和软件处理两种解决策略。

（1）处理器的设计者可以选择以硬件方式来处理非对齐的内存读写，除了增加硬件开销之外，还需要考虑读写操作的原子化问题。由于非对齐的内存读写往往需要两个读写周期才能完成，如果处理器硬件在设计时不加以特殊处理，则可能会由于外部中断等原因，导致在这两个读写周期之间被插入其他的操作，使内存读写不再是一个原子化的操作。RISC-V 本身对这种情况并没有做强制规定，而是由软硬件开发者自行决定。

（2）处理器的设计者也可以选择不在硬件层面去处理，而代之让硬件产生异常，然后交由软件处理。

3.4.8 RV32I 内存同步指令

RISC-V 的设计者在设计之初就考虑到了并行处理的问题，在 RISC-V 的术语中，每个处理器核可以包含多个硬件线程，称作硬件线程（Hardware Thread，HART）。每个 HART 都有自己的程序计数器和寄存器空间，独立顺序运行指令。不同的 HART 会共享同一个内存地址空间，从这一点上来说，HART 和 Intel 处理器中的超线程（Hyper Thread）非常类似。

在 RISC-V 中，当这些不同的 HART 需要做内存访问同步时，需要显式地（Explicitly）使用 FENCE 指令来同步数据。由于多 HART 的问题不在本书讨论的范围之内，所以本书对 FENCE 指令不再做进一步阐述。然而，即使是顺序执行指令的单核单硬件线程处理器，其也会有内存同步的问题，具体主要有以下两种情形：

（1）自修改代码（Self-modifying Code），即程序在执行过程中，会动态修改自己的指令存储内存。无论在工业界还是学术界，这种做法都是颇有争议的。甚至有些极端的看法认为只有病毒软件才需要自修改代码，而且自修改代码往往会带来安全上的隐患，所以不建议使用。

（2）软件载入过程。软件载入的一般情况是处理器上电后会先运行引导加载程序（Bootloader），然后引导加载程序会把其他软件当作数据载入到内存中，接着跳转至载入地址，运行新载入的软件。在这个过程当中，处理器可能存在指令内存预读取、指令缓存、流水线等一系列对内存同步有复杂影响的活动，在新软件被运行之前，需要采取措施，以保证内存同步。

不论是上面哪一种情况，为了实现内存同步，处理器都先要运行某个固定的指令序列来清空缓存、刷新流水线等。这个固定的指令序列被称作内存屏障（Memory Barrier）。内存屏障通常是由处理器设计者提供给软件设计者的。在 RISC-V 中，定义了指令同步命令 FENCE.I（该指令属于 Zifencei 扩展），用来发挥内存屏障的作用。

由于本书只讨论单处理核的情况，所以在本书所涉及的范围内，FENCE 与 FENCE.I 的实现并没有太大的区别，其定义如图 3-21 所示。

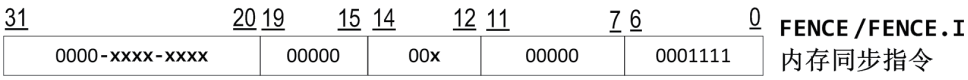


图3-21　内存同步指令

3.4.9　控制与状态寄存器指令

RISC-V 中除了内存地址空间和通用寄存器地址空间外，还定义了一个独立的控制与状态寄存器（Control Status Register，CSR）地址空间，其地址宽度是 12 位。随着设计目标的不同，每个处理器实际实现的 CSR 可能会有所不同，因此 RISC-V 将 CSR 的定义放在了特权架构部分，不过这些 CSR 的访问方式却是一致的，RISC-V 将访问这些 CSR 的指令定义在了用户指令集中（Zicsr 指令集扩展）。

如图 3-22 所示，RISC-V 中一共定义了 6 条访问 CSR 的命令，其具体功能

如表 3-5 所示。

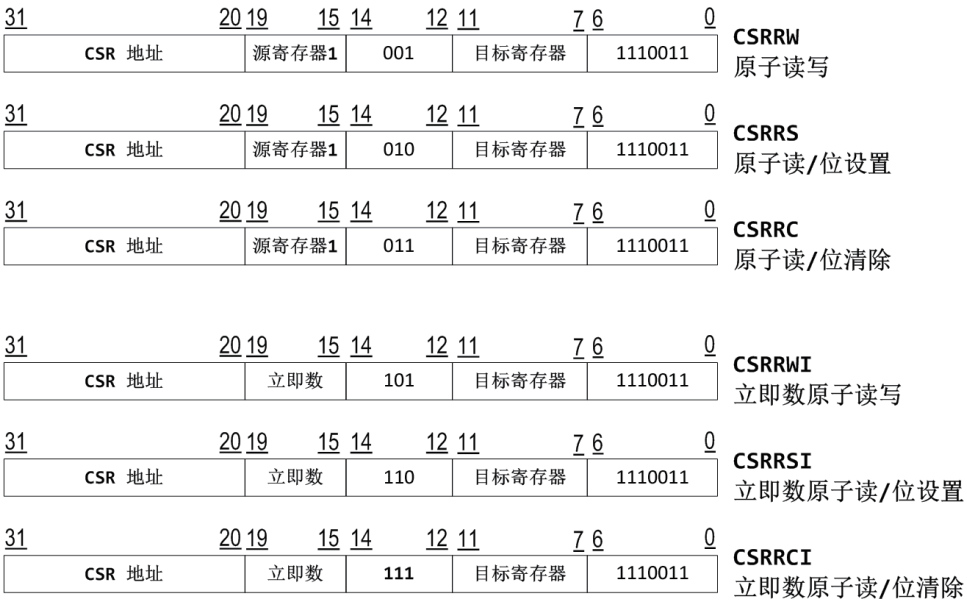


图3-22 控制与状态寄存器(CSR)操作指令

表 3-5 CSR 命令的功能定义

指 令	功 能
CSRRW	(CSR)→(目标寄存器)；(源寄存器 1)→(CSR)
CSRRS	(CSR)→(目标寄存器)；(CSR) | (源寄存器 1)→(CSR)
CSRRC	(CSR)→(目标寄存器)；(~(CSR)) & (源寄存器 1)→(CSR)
CSRRWI	(CSR)→(目标寄存器)；立即数→(CSR)
CSRRSI	(CSR)→(目标寄存器)；(CSR) | 立即数→(CSR)
CSRRCI	(CSR)→(目标寄存器)；(~(CSR)) & 立即数→(CSR)

在表 3-5 中定义的这些命令还包含如下规定：

(1) 表 3-5 中定义的操作都是原子操作。

(2) 对从 CSR 中读取的值都应该做零位扩展(将高位未使用部分置零)。

(3) 对立即数，也应将未使用的高 27 位置零。

（4）如果目标寄存器为 x0 的话，则 CSRRW 和 CSRRWI 应避免读取 CSR，避免产生不必要的副作用。

（5）如果源寄存器为 x0，则 CSRRS、CSRRC 应避免对 CSR 写入操作。

（6）如果立即数为零，则 CSRRSI、CSRRCI 应避免对 CSR 写入操作。

在具体实现时，上述的这 6 条指令可以由硬件实现，也可以为了减小硬件开销，而选择让硬件产生异常，转而由软件来处理。

3.4.10 环境调用与软件断点

如图 3-23 所示，RISC-V 中还定义了两条指令（ECALL 和 EBREAK），以实现操作系统的系统调用与软件断点。

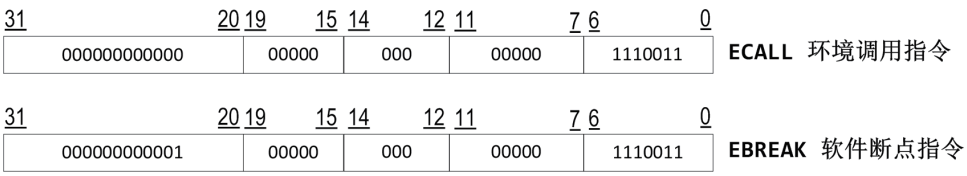


图3-23 环境调用与软件断点指令

3.4.11 基础指令集的面积优化方案

RV32I 中共包含 47 条指令，分为 6 类，其各类包含的指令条目数如表 3-6 所示。

表 3-6 RV32I 指令条目数

指　令	指 令 数
算术逻辑指令	立即数指令 (11) + 寄存器-寄存器指令 (10) = 21
跳转指令	无条件跳转指令 (2) + 有条件跳转指令 (6) = 8
内存存储指令	Load (5) + Store (3) = 8
FENCE 指令	2
CSR 指令	6
环境调用与断点	2

如果处理器设计者想要极力减小硬件逻辑开销，可以选择性地将表 3-6 中最后的 3 类共 10 条指令做简化实现，即将 FENCE 指令用 NOP 来代替，将 CSR 指令和 ECALL、EBREAK 合并解码（这些指令的操作码部分都是一样的）并产生异常，然后交由软件处理。这样的话，指令集实际需要实现的总指令数变为 47–2–6–2+1=38 条。对面积（Area）优化的处理器设计，可以采取这种简化的方案。

3.5 RISC-V扩展指令集

RISC-V 将指令集分成基础指令集和扩展指令集（Extension）。基础指令集已经在前文中得到阐述，而扩展指令集则有 M（乘除法扩展）、C（压缩指令扩展）、A（原子操作扩展）、F（单精度浮点扩展）、D（双精度浮点扩展）等众多的标准扩展。除此之外，RISC-V 还允许处理器设计者添加非标准的扩展。

由于扩展众多，在编译器编译代码时，需要把目标处理器具体支持的指令扩展告诉编译器。以 GCC（GNU C Compiler）为例，其在编译代码时，往往需要软件工程师提供以下两个选项：-march 和 -mabi。

（1）-march 选项被用来告知 GCC 目标处理器的基础指令集和扩展，对 32 位基础指令集 RV32I，常用的选项有：

① -march=rv32i，仅支持基础 32 位整数指令集（RV32I）。

② -march=rv32im，支持 RV32I + 乘除法扩展。

③ -march=rv32imc，支持 RV32I + 乘除法扩展 + 16 位压缩指令扩展。

（2）-mabi 选项被用来告知 GCC 其应该生成的 ABI，对 32 位基础指令集，常用的选项有：

① -mabi=ilp32，C 语言中的 int、long 和指针类型都是 32 位，浮点参数通过整数寄存器传递。

② -mabi=ilp32f，-mabi=ilp32d，浮点参数通过浮点寄存器传递。

出于实际考虑，本书只讨论标准扩展中的 32 位乘除法扩展和 16 位压缩指令扩展部分。前者对数值计算非常重要，后者则能大大提高代码密度。

3.5.1 乘除法扩展（M Extension）

如图 3-24 所示，RISC-V 中根据乘数和被乘数的类型（有符号数 / 无符号数）和结果的截取范围（高 32 位 / 低 32 位），分别定义了 4 条 32 位的乘法指令，其结果也是 32 位。

图3-24 乘法指令

同样，根据除数和被除数的类型（符号数 / 无符号数），RISC-V 中定义了 4 条 32 位除法和余数指令，如图 3-25 所示。

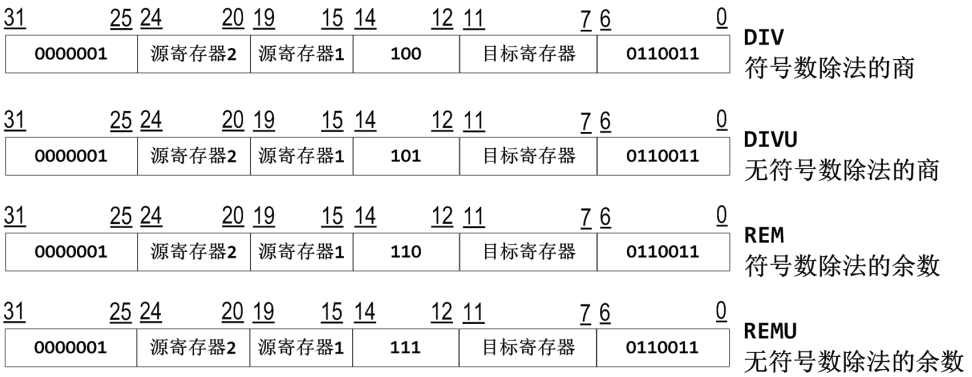


图3-25 除法和余数指令

RISC-V 舍弃了条件编码和状态标志位，而采用有条件跳转指令来帮助判断溢出等状态，除法指令也沿袭了这一设计思路。在 RISC-V 的除法中，无论被除数和除数的值是什么，都不会让硬件产生异常。不过，下面两种情况是需要特别判断的。

（1）除数为零。

在这种情况下，商应为 32 位全 1（32'hFFFFFFFF），而余数应等于被除数。

（2）符号数除法溢出。

这种情况只会发生在被除数为 -2^{31} 而除数为 −1 的情况。由于补码（Two's Complement）中正数和负数的不对称性，2^{31} 无法用 32 位符号数表示，导致溢出。在这种情况下，商应为 32'h80000000，而余数则为 0。

对 FPGA 实现来说，由于大部分的 FPGA 器件中都带有硬件乘法器，乘法指令可以直接用硬件乘法器来实现。而对除法和余数指令，既可以采用传统的移位相减的办法，又可以采用 SRT（Sweeney Robertson and Tocher）除法等实现快速除法。

3.5.2 压缩指令集扩展

为了提高代码密度，RISC-V 中引入了 16 位的压缩指令扩展（C Extension）。和 32 位指令集 RV32I 相比，C Extension 的引入可以将代码密度提高 40% 左右。

RISC-V 的 C Extension 对 32 位、64 位和 128 位的指令集都做了扩展，所以被统称为 RVC。本书将只讨论其中对 32 位指令集的扩展部分，即 RV32C，并且讨论也将集中于整数指令集部分。

1. C Extension 的格式

和许多 RISC 指令集不同，RISC-V 的 16 位压缩指令只是一个扩展（Extension），而不是一个独立的指令集。在 RV32C 中的每一条指令，实际上都可以被转化为一条完整的 32 位指令。RV32C 只是对部分 32 位指令的一种简写方式，从而将纯 32 位代码转化为 16 位和 32 位的混合方式。这样做的好处是处理器如果需要支持 C Extension，只需要修改指令取指器和指令解码器就可以了，大大简化了处理器的设计。

C Extension 中总共定义了 8 种指令格式，如图 3-26 所示。作为一种压缩指令扩展，C Extension 的指令格式中对立即数和寄存器都做了一些限制。

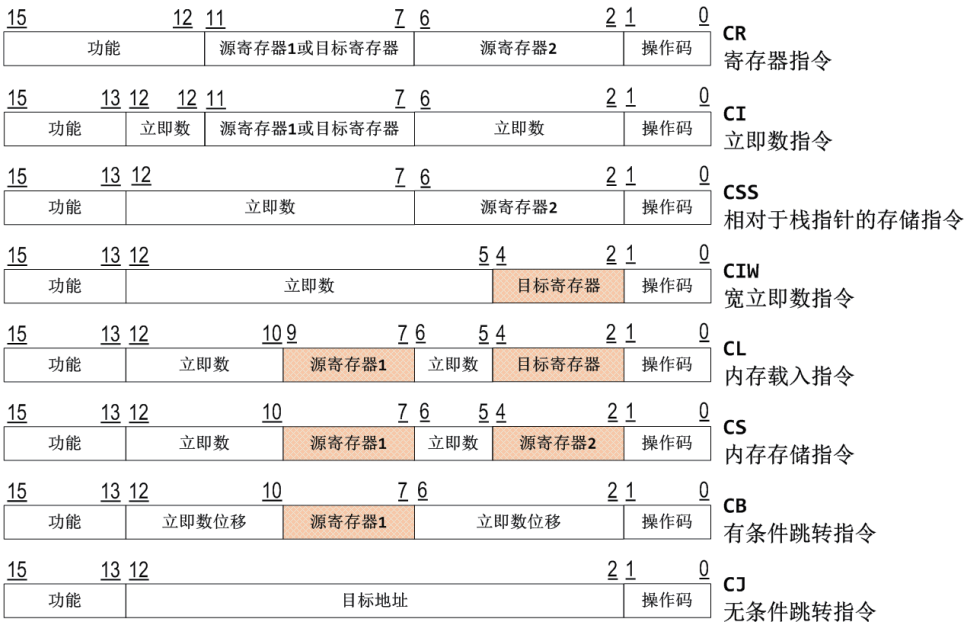


图3-26　C Extension指令格式

（1）立即数的位数被缩减。

（2）寄存器的寻址有 3 位和 5 位两种方式。对 3 位的寄存器寻址（图 3-26 中深色标记部分），其仅限于部分通用寄存器（x8 ～ x15，3'b000 对应 x8，3'b111 对应 x15）。由表 3-1 可以看到，x8 ～ x15 都是函数调用时必须用到的寄存器，临时寄存器没有被包含其中。

（3）如果指令同时涉及源寄存器和目标寄存器，则二者必须相等（如图 3-26 中所示的 CB 格式，尽管在图 3-26 中只标注了源寄存器，但是实际上在某些具体的指令中，也包含源寄存器和目标寄存器相同的情形，图 3-32 和表 3-12 中采用的是 CB 格式的 C.ANDI 指令）。

2. 16 位载入与存储压缩指令

C Extension 中定义了两种 LOAD 指令，如图 3-27 所示。一种是基于栈指针（x2）

的 C.LWSP 指令，另一种是基于寄存器的 C.LW 指令。它们对应的 32 位指令可以在表 3-7 中找到。

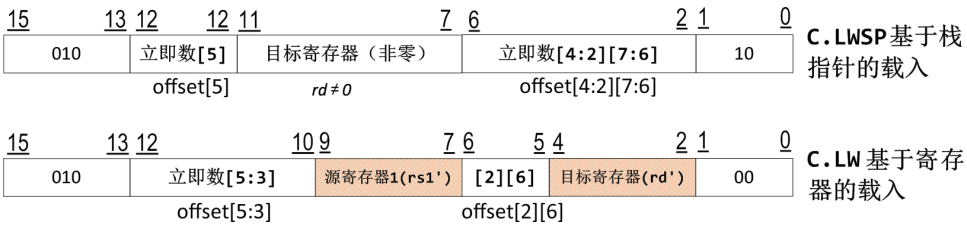


图3-27　C Extension中的LOAD指令

表 3-7　压缩 LOAD 指令对应的 32 位指令

16位压缩指令	对应的32位指令
C.LWSP	lw rd, offset[7:2](x2)　　(rd ≠ 0)
C.LW	lw rd', offset[6:2](rs1')

图 3-27 中的立即数（无符号数）都被左移 2 位，然后才被当作位移量。为此，图 3-27 中的立即数都做了一些位的重新安排，其原因和衍生出 B-TYPE 和 J-TYPE 的原因是一样的，都是为了减小处理器实现的硬件开销，此处不再赘述。

另外，图 3-27 中的 C.LW 指令，其寄存器只有 3 位表示。为了和 5 位的寄存器加以区分，在其名称后加了单引号（rs1' 与 rd'，而不是 rs1 与 rd）。

和 16 位载入压缩指令相类似，C Extension 中也分别基于栈指针（x2）和寄存器定义了两种存储指令，即 C.SWSP 和 C.SW。它们的定义如图 3-28 所示，对应的 32 位指令可以在表 3-8 中找到。由于 STORE 指令格式和前文的 LOAD 指令非常类似，这里就不再进一步展开。

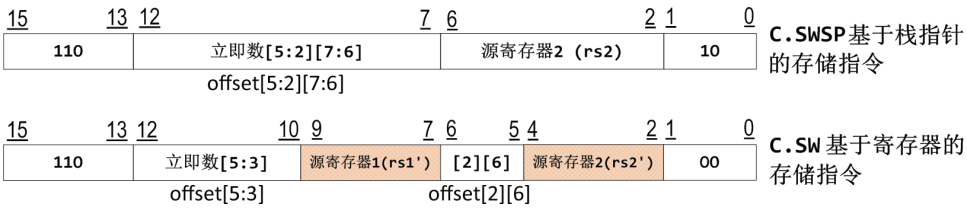


图3-28　C Extension中的STORE指令

表 3-8　压缩 STORE 指令对应的 32 位指令

16位压缩指令	对应的32位指令
C.SWSP	sw rs2, offset[7:2](x2)
C.SW	sw rs2', offset[6:2](rs1')

3. 16 位跳转压缩指令

C Extension 中定义了 4 条无条件跳转压缩指令，其定义如图 3-29 所示，它们对应的 32 位指令可以在表 3-9 中找到。在这 4 条指令中，C.J 和 C.JR 都不会保存返回地址（默认目标寄存器为零），而 C.JAL 和 C.JALR 则默认目标寄存器为 x1（ra）。同时，C.JAL 和 C.JALR 的返回地址是 PC+2，而不是之前 32 位指令中的 PC+4。

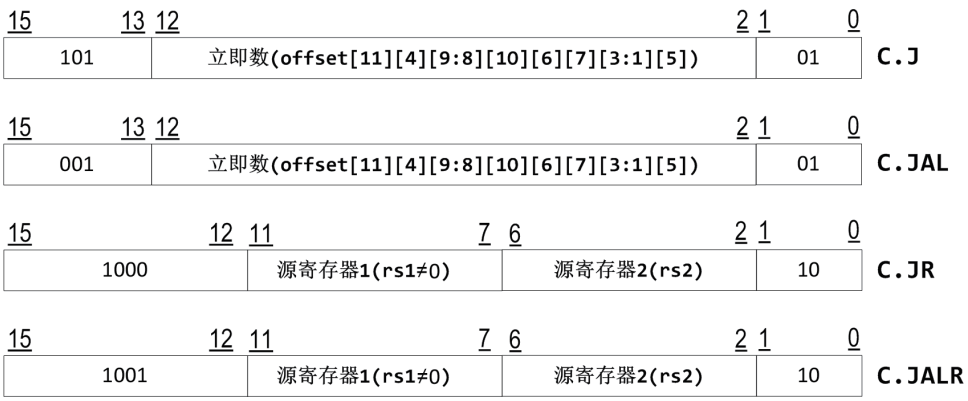
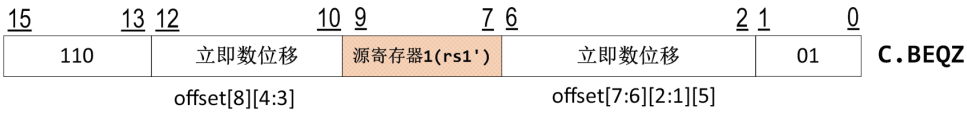


图3-29　C Extension中的无条件跳转指令

表 3-9　无条件跳转压缩指令对应的 32 位指令

16位压缩指令	对应的32位指令
C.J	jal x0, offset[11:1]
C.JAL	jal x1, offset[11:1]
C.JR	jalr x0, rs1, 0
C.JALR	jalr x1, rs1, 0

另外，C Extension 中还定义了 2 条有条件跳转压缩指令，其定义如图 3-30 所示，它们对应的 32 位指令可以在表 3-10 中找到。这两条指令都默认源寄存器 2 为 x0。



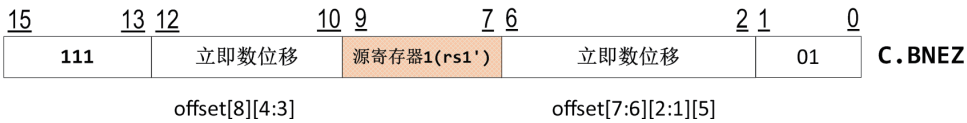


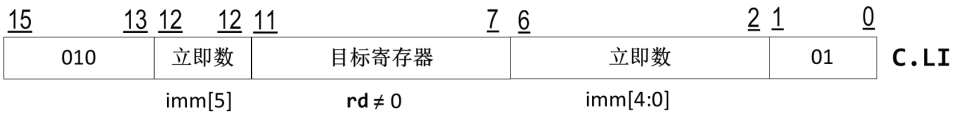
图3-30　C Extension中的有条件跳转指令

表 3-10　有条件跳转压缩指令对应的 32 位指令

16位压缩指令	对应的32位指令
C.BEQZ	beq rs1', x0, offset[8:1]
C.BNEZ	bne rs1', x0, offset[8:1]

4. 16 位整数计算压缩指令

C Extension 中制定了 2 条压缩指令，来生成整数常量（Integer Constant-Generation Instruction）。它们的定义如图 3-31 所示，它们对应的 32 位指令可以在表 3-11 中找到。其中，C.LI 指令中的立即数需要做符号扩展，而 C.LUI 中的立即数则是非零的无符号数。



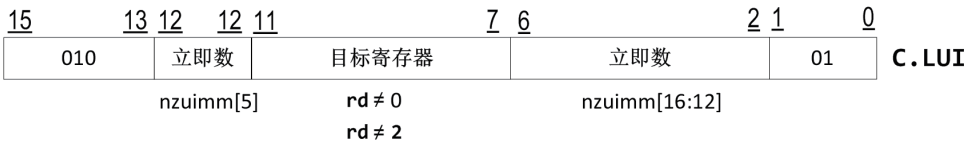


图3-31　C Extension中的常数生成指令

表 3-11　常数生成压缩指令对应的 32 位指令

16位压缩指令	对应的32位指令
C.LI	addi rd, x0, imm[5:0]　(rd ≠ 0)
C.LUI	lui rd, nzuimm[17:12]　(rd ≠ 0,rd ≠ 2)

另外，C Extension 中还定义了 2 条立即数加法指令，3 条立即数移位指令和 1 条立即数逻辑指令，其定义如图 3-32 所示。它们对应的 32 位指令可以在表 3-12 中找到，其中 C.ADDI4SPN 指令默认的源寄存器是 x2（sp 栈指针），以便基于栈指针的计算。

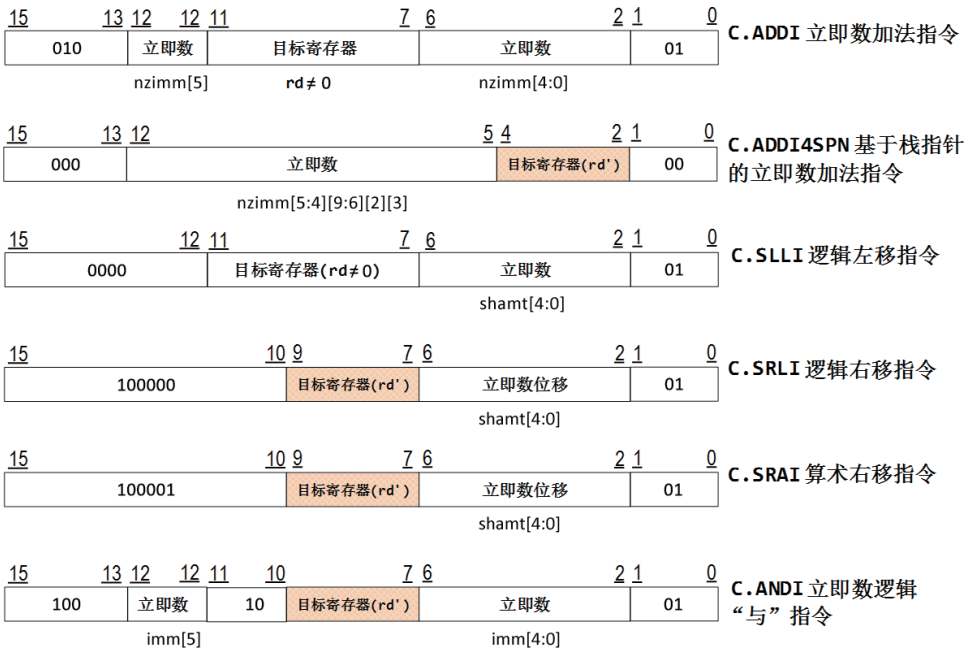


图3-32　C Extension中的寄存器-立即数指令

表 3-12　寄存器 - 立即数压缩指令对应的 32 位指令

16位压缩指令	对应的32位指令
C.ADDI	addi rd, rd, nzimm[5:0]　(rd ≠ 0)
C.ADDI4SPN	addi rd', x2, nzuimm[9:2]
C.SLLI	slli rd, rd, shamt[4:0] (shamt[4:0] ≠ 0)
C.SRLI	srli rd', rd', shamt[4:0] (shamt[4:0] ≠ 0)
C.SRAI	srai rd', rd', shamt[4:0] (shamt[4:0] ≠ 0)
C.ANDI	andi rd', rd', imm[5:0]

同时，和立即数指令相对应，C Extension 中也定义了寄存器 - 寄存器操作的压缩指令，其定义如图 3-33 所示。它们对应的 32 位指令可以在表 3-13 中找到，其中的寄存器复制指令 C.MV 实际上是一条将源寄存器 1 默认为 x0 的加法指令。

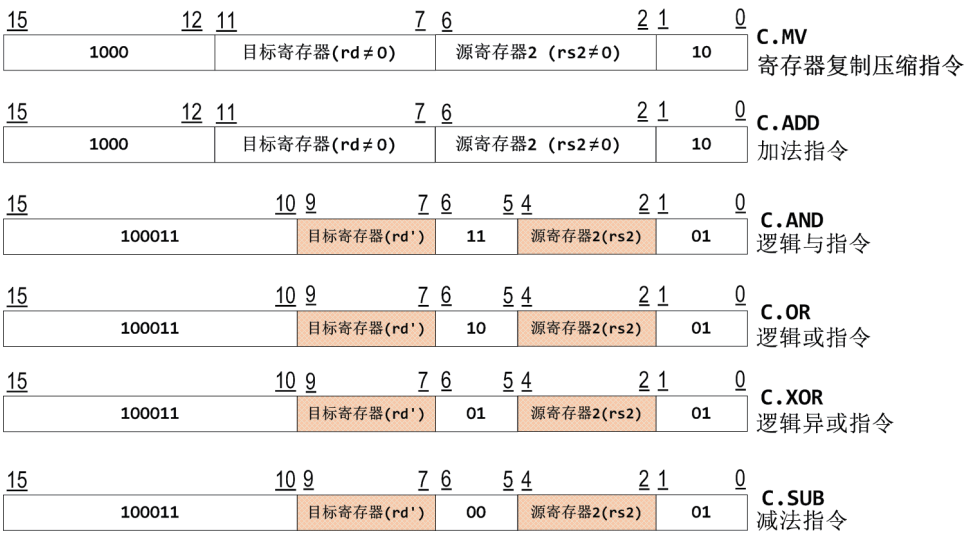


图3-33　C Extension中的寄存器-寄存器指令

表 3-13　寄存器 - 寄存器压缩指令对应的 32 位指令

16位压缩指令	对应的32位指令
C.MV	add rd, x0, rs2
C.ADD	add rd, rd, rs2
C.AND	and rd', rd', rs2'
C.OR	or rd', rd', rs2'
C.XOR	xor rd', rd', rs2'
C.SUB	sub rd', rd', rs2'

5. 其他的 16 位压缩指令 (Miscellaneous)

1）C.NOP，16 位空操作指令

和 32 位的空操作指令类似，C Extension 中也利用目标寄存器为零的加法指令来衍生出空操作指令，即 c.nop = c.addi x0，0 = addi x0，x0，0

2）16 位非法操作指令（Illegal Instruction）

和 32 位指令集不同的是，C Extension 专门将全零的编码定义为非法操作指令，以方便利用硬件异常来处理被零初始化的代码内存。

3）16 位软件断点指令（C.EBREAK）

C Extension 中也为 16 位压缩指令集定义了对应的软件断点指令，其机器代码为 16'h9002。

6. 函数调用的开场白和收场白

在讨论函数调用约定时，曾经提到通用寄存器 x5 既可以作为临时寄存器（t0），又可以作为替代链接寄存器（见表 3-1）。

之所以在 RISC-V 中引入 16 位压缩扩展指令集，其初衷就是为了降低代码量，提高代码密度。而在代码的每个函数调用的开始，往往都需要编写代码，将当前寄存器值保存到堆栈上。当函数返回时，也需要编写代码，将之前保存的寄存器值从堆栈恢复到寄存器。这两部分代码，分别称为开场白与收场白。由于函数的调用和返回在大部分代码中都会频繁出现，压缩指令集的设计者自然会希望将这部分代码的代码量降至最低。

由此，各种 RISC 指令集和处理器的设计者们给出了不同的解决方案。在加州大学伯克利分校设计的第一代 RISC 处理器（RISC I）和后来 SUN 公司的 SPARC 处理器当中，都采用了寄存器窗口的解决方案。但是寄存器窗口使得硬件开销变得非常大，而实际使用效果却并不理想，特别是当通用寄存器被耗尽时，其处理非常麻烦和缓慢，最终为后来的设计者所放弃。

后来的设计者如 ARM 公司，则在压缩指令集中引入了 Load-Multiple 和 Store-Multiple 指令。这些指令可以在一个指令周期内，将内存中多个地址连续的存储字载入多个寄存器，或反之将多个寄存器中的内容在一个指令周期内写入内存的连续地址。使用这些指令，自然可以极大降低开场白与收场白的代码量。根据 Andrew Waterman 在其博士论文中提供的统计数据，使用 Load-Multiple 和 Store-Multiple 以后，可以将 Linux 内核的代码量降低 8% 左右。

然而，RISC-V 的设计者再三斟酌后，决定忍痛割爱，不在 C Extension 中支持 Load-Multiple 与 Store-Multiple，其原因主要如下：

（1）在前文提到的所有 16 位压缩指令，都可以在 32 位指令集中找到对应的指令。也就是说，每一条 16 位压缩指令，都是其对应的 32 位指令的简写版。如果引入 Load-Multiple 与 Store-Multiple 指令，则会打破这一原则。

（2）在 3.3 节中提到，RISC-V 的设计目标之一就是希望指令集设计独立于具体的处理器实现。而引入 Load-Multiple 与 Store-Multiple，会在一定程度上束缚处理器设计者的手脚，有违 RISC-V 的设计初衷。

（3）对那些有 MMU（Memory Management Unit，内存管理单元）的处理器，会有虚拟地址（Virtual Address）和物理地址（Physical Address）两种不同的地址空间。程序在虚拟地址上运行，需要访问内存时，再通过 MMU 将虚拟地址转换为物理地址。这就导致在虚拟地址空间里连续的地址，转换为物理地址后可能不再连续。如果要在这种系统上实现 Load-Multiple 与 Store-Multiple，会大大增加硬件异常处理的难度。

因此，RISC-V 的设计者别出心裁，在借鉴了 IBM S/390 计算机毫码程序之后，提出了如下的方案：

（1）由于开场白和收场白的工作仅仅只是将数据在堆栈和寄存器之间移动的，这些工作完全可以用共享的代码来实现。开场白和收场白本身也可以用类似函数调用的方式来实现。

（2）然而，这种函数调用是一种特殊的函数调用，因为：

① 这种调用本身不再需要开场白和收场白。

② 由于原先普通函数调用的返回地址需要使用 x1（ra）来存放，调用开场白和收场白时则不能再使用 x1（ra），而需要在函数调用约定中另外再分配一个寄存器。这个寄存器就是 x5（t0/ 替代链接寄存器）。

（3）在软件实现时，可以做如下操作：

① 把原先的开场白代码，用 jal t0，shared_prologue 代替，以调用共享的开场白代码，并将返回地址存入 x5（t0）。

② 在共享的开场白代码中，用一连串的 c.swsp 指令将需要保存的寄存器值（其中也包括 x1（ra））推入堆栈中，最后用 c.jr t0 指令返回。

③ 把原先的收场白代码，用 jal x0，shared_epilogue 代替。

④ 在共享的收场白代码中，用一连串的 c.lwsp 指令，将之前保存在堆栈上的寄存器值恢复到对应的寄存器中（其中也包括 x1（ra）），最后用 c.jr ra 结束整个函数调用。

根据 Andrew Waterman 提供的统计数据，使用以上类似毫码程序的方案以后，可以将 Linux 内核的代码量降低 7.5% 左右，其表现基本上和之前提到的 Load-Multiple/Store-Multiple 方案相当，但是却避免了 Load-Multiple/Store-Multiple 方案带来的缺点。

3.6 RISC-V特权架构

如前所述，RISC-V 的设计者将其官方标准分成了两部分：用户指令集与特权架构，其目的是希望不同特权架构的处理器可以在 ABI 互相兼容。换句话说，支持同一用户指令集的处理器可以根据实际需求而在特权架构的设计上采取不同的策略。

本书将在后文介绍的软核处理器属于 MCU（Microcontroller Unit，微处理器单元）的范畴，本章将会重点讨论以下的内容：

- 特权层级，特别是机器模式（Machine Mode, M-Mode）。
- 控制状态寄存器。

- 机器层级指令集。
- 异常和中断。
- 调试。

3.6.1 特权层级

RISC-V 处理器中的软件代码都是在硬件线程上运行的。为了加强对操作系统和信息安全的支持，RISC-V 替 HART 定义了 3 种工作模式（见图 3-34）：机器模式、超级用户模式（Supervisor Mode，S-Mode）和普通用户模式（User Mode，U-Mode）。每种模式分别对应一个特权层级（Privilege Levels）。其中机器模式的特权层级最高，而普通用户模式的特权层级最低。在高特权层级运行的代码比在低特权层级的代码拥有更多的权限，受到的约束也比低特权层级的代码要少。

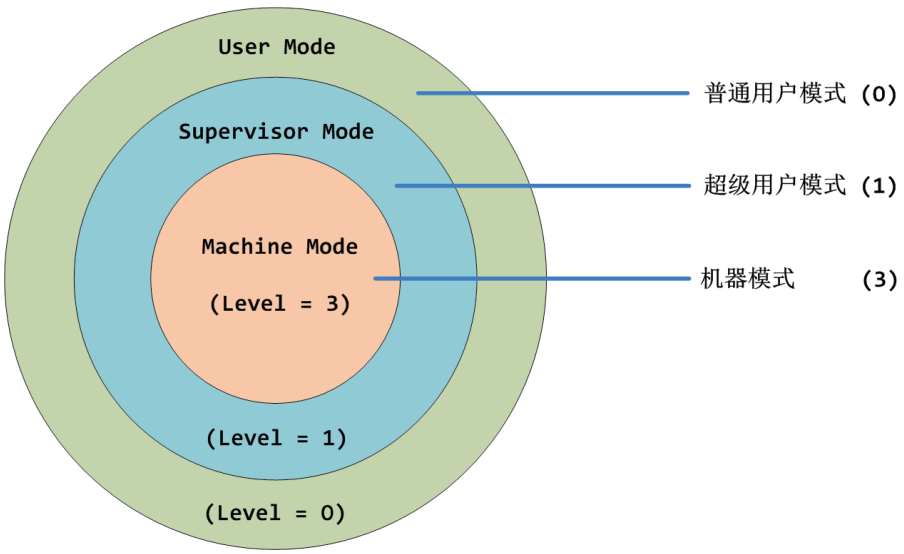


图3-34 特权层级

在处理器设计时，机器模式是强迫要求实现的。其他的两个模式，处理器设计者则可以选择性地加以实现。一般来说，小规模的嵌入式系统只需要机器模式就可以了，而对信息安全有特殊要求的系统，则可能需要机器模式加普通用户模式。运行类似 UNIX 这样大型操作系统的处理器，则需要实现以上所有的模式。

UNIX 和信息安全不在本书的讨论范围之内，本书的剩余部分将会集中于机器模式的讨论。

3.6.2 控制状态寄存器

RISC-V 在特权架构部分单独定义了一个控制状态寄存器的地址空间，并分配了 12 位地址来做索引。在这 12 位地址当中，最高的两位 [11:10] 被用来指示寄存器的读写权限。如果这两位是 2'b11 的话，则表示该寄存器是只读寄存器；否则，该寄存器既可以被读取，又可以被写入。地址位 [9:8] 表示有权访问该寄存器的最低特权层级。对要讨论的机器模式 CSR，这两位都是 2'b11。在表 3-14 中列出了本书会涉及的所有 CSR 寄存器。

表 3-14　本书所涉及的 CSR 寄存器列表

地　址	读写权限	特权层级	寄存器名称	寄存器描述
0xF11	只读	机器模式	mvendorid	厂商标识 (Vendor ID)
0xF12	只读	机器模式	marchid	架构标识 (Architecture ID)
0xF13	只读	机器模式	mimpid	实现标识 (Implementation ID)
0xF14	只读	机器模式	mhartid	硬件线程标识 (HART ID)
0x300	读写	机器模式	mstatus	状态寄存器
0x301	读写	机器模式	misa	处理器所支持的指令集与扩展
0x304	读写	机器模式	mie	中断允许
0x305	读写	机器模式	mtvec	异常相量寄存器，保存异常发生时的向量基准地址
0x340	读写	机器模式	mscratch	草稿寄存器
0x341	读写	机器模式	mepc	保存异常发生时的程序寄存器值
0x342	读写	机器模式	mcause	异常原因寄存器
0x343	读写	机器模式	mtval	参见后文有关异常处理的章节
0x344	读写	机器模式	mip	待定的中断（参见后文有关中断的章节）
0xB00	读写	机器模式	mcycle	机器周期计数器（低 32 位）
0xB02	读写	机器模式	minstret	机器指令计数器（低 32 位）
0xB80	读写	机器模式	mcycleh	机器周期计数器（高 32 位）
0xB82	读写	机器模式	minstreth	机器指令计数器（高 32 位）

表 3-14 看起来有点长，但这只是众多 CSR 寄存器中的一小部分。读者如果想了解 RISC-V 完整的 CSR 寄存器列表，则可以查找 RISC-V 的官方标准。本章就对表 3-14 中的寄存器做仔细讨论。

1. mvendorid 寄存器

为了对不同厂商设计生产的 RISC-V 处理器加以区分，RISC-V 在其特权架构标准部分制定了 mvendorid 寄存器，用来存储厂商标识代码。对 RV32 来说，这是一个 32 位的只读寄存器，它的取值实际上是衍生于 JEDEC 厂商标识代码。

JEDEC 是联合电子设备工程委员会（Joint Electron Device Engineering Council）的英文缩写，目前的名称为 JEDEC 固态技术协会（Solid State Technology Association）。它是一个在 1958 年成立的行业协会组织，其总部位于美国弗吉尼亚州。所有的电子产品生产厂商都可以向 JEDEC 付费申请得到一个 JEDEC 的厂商标识代码（即 JEDEC 厂商标识代码）。

JEDEC 的厂商标识代码分为两部分：第一部分是 Bank 域（Bank Field）；第二部分是只有一字节的 Offset 域（Offset Field）。对于这个 8 位的 Offset 域，其最高位是奇数校验码，其余 7 位对应 Bank 域里面的厂商标识。如果 Bank 域的值是 n，Offset 域的值是 m，那么其对应的完整的 JEDEC 厂商标识代码应该是将 0x7F 重复 $n-1$ 遍，然后再在后面接上 m。

例如 JEDEC 给美国 PulseRain Technology 公司分配厂商标识的文件中有如下文字：

the following JEDEC Manufacturer ID number has been assigned to your company：

94 decimal（bank 11）

0101 1110 binary

5E hex

由此，n=11，m=0x5E 其完整的 JEDEC 厂商标识代码就是：

0x7F-0x7F-0x7F-0x7F-0x7F-0x7F-0x7F-0x7F-0x7F-0x7F-0x7F-0x5E

由于 JEDEC 提供的厂商标识远超出了 32 位寄存器可以表示的范围，RISC-V 在特权架构标准中定义了一个方法，用来在 JEDEC 厂商标识代码的基础上衍生出一个 32 位的数值，然后固化于 mvendorid 寄存器中。根据 JEDEC 提供的 *n* 与 *m* 数值，RISC-V32 位厂商标识代码可以用如下公式产生：

$$\text{Vendor ID}=((n-1)<<7)+(m\&0x7F) \qquad (3\text{-}1)$$

根据上文 *n*=11，*m* ＝ 0x5E ，可以得出 PulseRain Technology 的 RISC-V 厂商标识代码为 0x55E。

2. marchid（体系架构标识代码）

根据 RISC-V 官方标准，marchid 在 RV32 下是一个 32 位的只读寄存器，用来存放 HART 所对应的体系架构的标识代码。对于开源的架构来说，这个寄存器的值由 RISC-V 基金会负责在全球分配，其最高位必须是 0。对于商业公司所研发的架构来说，其值由具体的商业公司来分配，但是其最高位必须为 1，其余的位不能全为零。这样，如果将该寄存器和 mvendorid 寄存器一起使用，则可以唯一地标识 HART 的体系架构。

如果处理器设计者选择不支持这个寄存器，则应该返回零值。

3. mimpid（实现标识代码）

根据 RISC-V 官方标准，mimpid 寄存器在 RV32 下也是一个 32 位的只读寄存器，其主要的目的是标明处理器的版本号。该寄存器的格式完全由处理器设计者自行决定，如果处理器设计者选择不支持该寄存器，则应该返回零值。

4. mhartid（硬件线程标识）

在 RISC-V 的术语中，每个处理器核可以包含有多个硬件线程，称作 HART（Hardware Thread）。每个 HART 都有自己的程序计数器和寄存器空间，独立顺序运行指令。mhartid 寄存器用来给这些 HART 编号索引。在多处理器系统中，HART 的编号无须连续，但是必须至少有一个 HART 必须被编号为零。

5. misa（指令集寄存器）

由于 RISC-V 指令集标准涵盖多种字长（32 位 /64 位 /128 位），并包含多种

指令集扩展（如 16 位压缩指令集扩展 C，乘除法扩展 M 等）。misa 寄存器的目的是为了向软件告知处理器具体支持的字长和扩展，以方便软件的可移植性。本寄存器各位完整的定义可以在 RISC-V 官方标准中找到。对于只支持单个 HART 和机器模式的 RV32 处理器来说，如表 3-15 所示的这些位值得关注。

表 3-15　指令集寄存器

位　索　引	注　　释
[31:30]	MXL（Machine XLEN，字长）（X Register Length，XLEN） 对 RV32 来说，这两位应该被置为 2'b01
[12]	M（乘除法扩展支持）
[8]	I（基础整数指令集支持） 该位总是为 1
[4]	E（RV32E 嵌入式指令集支持）
[3]	C（16 位压缩指令集支持）

6. mstatus（硬件线程状态寄存器）

mstatus 寄存器用来标识和控制 HART 的操作状态，其各位完整的定义可以在 RISC-V 官方标准中找到。对于只支持单个 HART 和机器模式的处理器来说，需要注意表 3-16 中的两个位。

表 3-16　硬件线程状态寄存器

位　索　引	注　　释
[7]	mpie（machine previous interrupt enable，全局中断使能保持 / 恢复） 当中断或异常发生时，该位会记录下当前 mie 的值。当用 MRET 从中断或异常中返回时，该位的值将被复制到 mie 中
[3]	mie（machine interrupt enable，全局中断使能） 当该位为零时，外部中断和时钟中断都将被禁止

7. mscratch（草稿寄存器）

在 RV32 下，这是一个 32 位的读写寄存器。除了被用来作为 CSR 寄存器操作的读写测试以外，它还可以被操作系统作为暂存空间。

8. 与中断和异常有关的 CSR 寄存器

RISC-V 中还定义了多个 CSR 寄存器用来处理中断与异常，其中与机器模式相关的部分主要如下：

- mtvec（machine trap vector base-address register，机器模式异常向量基地址寄存器）。
- mip（machine interrupt register, pending interrupt，机器模式中断等待寄存器）。
- mie（machine interrupt register, interrupt enable，机器模式中断使能寄存器）。
- mcause（machine cause register，机器模式异常原因寄存器）。
- mepc（machine exception program counter，机器模式异常 PC 寄存器）。
- mtval（machine trap value register，机器模式异常值寄存器）。

9. 计数器

作为一种硬件性能监测的手段，RISC-V 在其特权架构部分定义了一系列计数器，用来记录从某一时间点开始后处理器已运行的时钟周期数和已执行的指令数。具体来说，其主要包括如下的 CSR 寄存器。

1）mcycle 与 mcycleh

RISC-V 中为机器模式定义了一个 64 位的 cycle 寄存器，用来记录机器已经运行的时钟周期数。这个寄存器的低 32 位和高 32 位分别存放在 mcycle 和 mcycleh 中。

对 RV32 来说，由于无法一次性地将 mcycle 和 mcycleh 同时读取出来，为了保证 64 位数据的完整性，需要在寄存器的读取方式上做一些处理。一种解法是要求软件总是先读取 mcycle，紧接着再读取 mcycleh。软件读取 mcycle 时，硬件同时将当时的 mcycleh 值保存下来，并在下次读取时提供该值。而另外一种解法则如代码 3-1 所示（参见 RISC-V 官方标准，将 cycle 寄存器的值读入到 x3∶x2 中）。

代码3-1　64位cycle寄存器的读取

```
again:
    rdcycleh x3
    rdcycle  x2
    rdcycleh x4
    bne x3, x4, again
```

2）minstret 与 minstreth

RISC-V 还为机器模式定义了一个 64 位的 instret 寄存器，用来记录机器已经完成的指令数（The Number of Instructions Retired）。该寄存器的低 32 位和高 32 位被分别存放在 minstret 与 minstreth 中。

同 cycle 寄存器一样，当在 RV32 中读取该寄存器时，也会面临保持 64 位数据完整性的问题。其解法也与上述读取 cycle 寄存器的解法相同。

3.6.3　定时器

RISC-V 在设计时也对 RTC（Real Time Clock，实时时钟）的实现做了考虑。实际上，要在真正意义上实现一个实时时钟，需要以下几部分的硬件支持：

（1）需要一个时钟定时器（Timer），运行在固定的频率。

（2）需要能有办法获取时间基准，以用来计算实时时间（Wall Clock）。简单地说，就是要能有办法获取当前精确的日期与时间。在桌面系统中，这个时间基准可以通过网络从专门的时间服务器获取。在许多嵌入式系统里，这个时间基准可以通过 GPS 获取。

（3）需要有办法能保持时钟定时器的不间断运行。这意味着 RTC 需要有自己的电源域，这样即使处理器的其他部分进入深度休眠状态，RTC 可以依然保持运行。而且 RTC 的电源域一般还需要有替代电源，如电池等。

为此，RISC-V 在其特权架构部分为机器模式定义了两个 64 位的寄存器：mtime 与 mtimecmp。同时，为了方便 RTC 的独立运行，减小实现 RTC 的硬件开销，

让多个 HART 能共享 RTC，RISC-V 中将这两个寄存器定义为内存映射寄存器（见图 3-4），以映射到内存空间中（而不是像 mcycle 这样定义为 CSR）。而这两个 64 位寄存器在内存空间中的地址，则由具体的实现决定，RISC-V 标准中并没有对它们的地址做硬性规定。

1. mtime 寄存器

mtime 是时钟定时器。一般来说，它应该以比较精确的石英晶体振荡器为时钟源，并以固定的频率做计数。然而，这个固定的频率具体是多少，RISC-V 中并没有作出明确规定。许多系统将该频率设置为 32.768 kHz，因为 32.768 kHz 的晶振非常容易获得，而且 32.768 kHz 频率较低，适合做休眠时钟。另外，32 768 又是 2 的整数次幂，很容易由 32.768 kHz 产生周期为 1s 的时钟。

提示：笔者在 RV32 的设计实践中发现，如果 mtime 的运行频率不能被处理器的主时钟频率整除的话，则可能会给软件，特别是嵌入式操作系统的运行带来额外的开销。因为操作系统在做任务调度时，需要对时间片有精确的计算。如果处理器的主时钟频率不是 mtime 运行频率的整数倍（如 mtime 运行于 32.768 kHz，而处理器主时钟频率为 100 MHz），则操作系统可能需要做非常复杂的 64 位除法运算。出于性能的考虑，这时操作系统往往要求处理器能支持硬件乘除法扩展（即支持 M Extension）。而对同样的嵌入式操作系统，如果处理器的主时钟频率是 mtime 运行频率的整数倍（如 mtime 运行于 1 MHz，而处理器主时钟频率为 100 MHz），则处理器只要支持基础整数指令集即可。

2. mtimecmp 寄存器

mtimecmp 的主要作用便是将其与 mtime 的值做比较。当 mtime 的值大于或等于 mtimecmp 时，便可触发产生时钟中断。

由于 mtimecmp 是一个 64 位的寄存器，在 RV32 系统中至少需要两条写指令才能完成对其的更新。而部分更新的 mtimecmp 寄存器值可能会误触发产生时钟中断。对此，通常的处理方法有两种：

（1）在更新 mtimecmp 之前禁止时钟中断（Disable Timer Interrupt）。在 mtimecmp 更新完毕后再重置并允许时钟中断。

（2）采用 RISC-V 官方标准中建议的汇编代码序列（代码 3-2）。

假设需要写入 mtimecmp 的低 32 位存放于寄存器 a0 中，而高 32 位存放于寄存器 a1 中，如代码 3-2 所示。

代码3-2　mtimecmp的写入

```
li t0, -1            # 将0xFFFFFFFF写入寄存器t0
sw t0, mtimecmp      # 将mtimecmp的低32位置为0xFFFFFFFF
sw a1, mtimecmp + 4  # 设置mtimecmp的高32位
sw a0, mtimecmp      # 设置mtimecmp的低32位
```

注意：上面的汇编代码需要被完整并严格地顺序执行。编译器的优化，中断服务程序（Interrupt Service Routine，ISR）的插入，或者高端处理器的乱序执行都可能对上面代码的正确性产生影响。

3.6.4　中断与异常

1. 中断与异常的比较

我们知道，软件并不总是按照其原先计划好的步骤运行，在多数情况下，软件在执行过程中总会发生一些意外，使得处理器不得不暂停现有的软件执行步骤，转而去做其他的额外处理。这种意外事件主要分为两种情况。

（1）这种意外事件是由软件执行本身引发的。常见的情形包括：

- 软件在执行过程中访问了一个不存在的 CSR 寄存器。
- 软件在访问内存时没有按照字长对齐。
- 遭遇断点或者操作系统调用。

这种由软件本身引起的意外事件通常被称作异常（许多处理器会将被零除也作为一种异常，不过 RISC-V 的除法指令是不会产生异常的）。

（2）这种意外事件是由独立于软件运行的外部事件引发的。

这种由外部事件导致的意外通常被称作中断。在单个 HART 的机器模式下，中断主要来源于两个地方：

① 定时器中断。

② 来自处理器核外部的中断，主要由外围设备产生。

在实际的硬件处理中，中断和异常的处理非常相近。

2. RISC-V 的中断控制器结构

在中小规模的嵌入式系统中，一般都会对中断信号的电气特性做直接处理。具体地说，中断信号的电气特性一般有两种：电平触发（Level Trigger）和边沿触发（Edge Trigger），而且一般以电平触发居多。对于多个中断源的情况，可以简单地将它们线或（Wired-OR）在一起，作为共享中断，如图 3-35 所示。对于中断延迟要求比较高的情形，也可以用专门的中断向量控制器（Vectored Interrupt Controller，VIC）来处理，如图 3-36 所示。

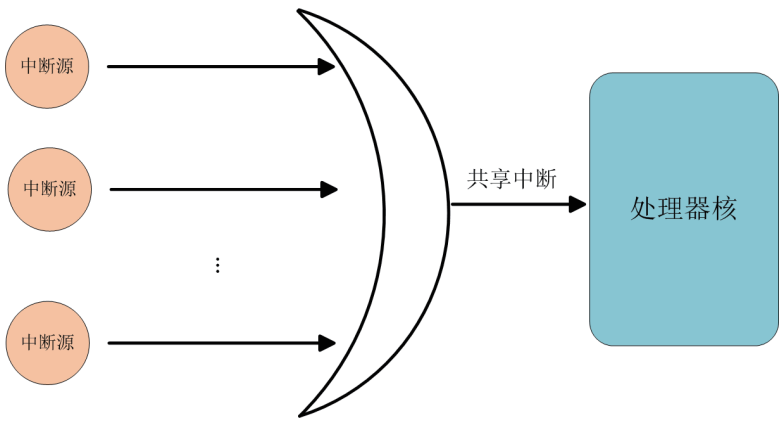


图3-35　共享中断

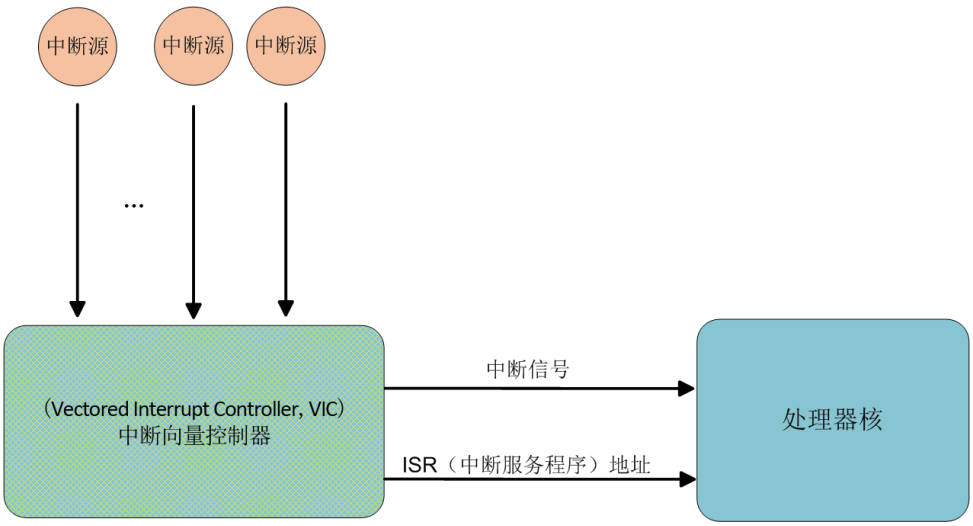


图3-36　中断向量控制器

在共享中断的情况下，处理器核在收到中断信号后，需要在中断处理程序中逐个查询外围设备，以确定中断源，因此其中断相应的延迟较大，其优点是硬件设计比较简单。对于像中断优先级、中断嵌套等问题，则大多都交由软件来处理。

为了减小中断延迟，许多中高端的嵌入式处理器都会在处理器核之外放置一个中断向量控制器，如图 3-36 所示。中断向量控制器在向处理器核提供中断信号的同时，还会提供与中断源相对应的 ISR（Interrupt Service Routine，中断服务程序）的入口地址。这些 ISR 的地址便组成了中断向量表（Interrupt Vector Table，IVT）。同时，中断向量控制器一般还会支持中断优先级、中断屏蔽等设置。与共享中断相比，中断向量控制器的硬件开销较大，但是软件处理则相对简单直接。

以上说的这两种方式都称为带外中断，其中断信号和数据是分开独立的。然而，随着系统规模的日益增大和高速串行数据传输的不断发展，点对点的拓扑结构变得流行起来。PCI-Express 总线便是其典型代表之一，它采用的中断机制被称为消息告知中断（Message Signaled Interrupt，MSI），这是一种带内中断的中断机制。在这种中断方式下，设备通过向某个指定的地址写入特殊的消息来发送中断信号。而外围设备也通过交换矩阵（Switch）和处理器核相连。MSI 中断机制的优点是其可扩展性比较好，缺点是其软硬件都比较复杂。

这种 MSI 中断机制和交换矩阵的思路显然也影响了 RISC-V 的设计者。在 RISC-V 标准中，对 RISC-V 的外部中断控制定义为 PLIC（Platform-Level Interrupt Controller，平台级中断控制器），其结构如图 3-37 所示。

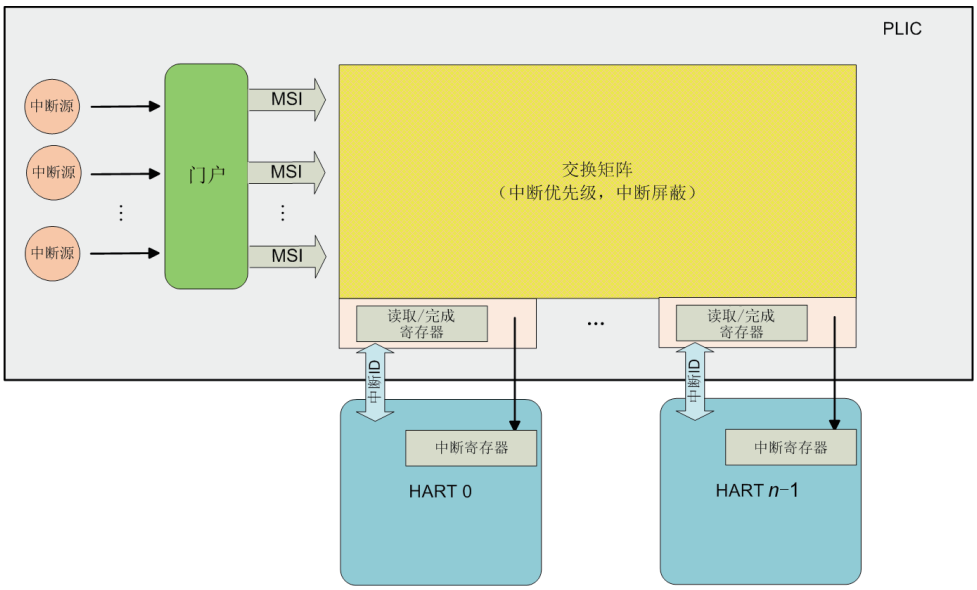


图3-37　PLIC的结构

从图 3-37 可以看出，PLIC 的设计考虑到了多个 HART 的情况。图 3-37 中门户的作用主要是将中断源来的中断电气信号转换为 MSI，然后交由交换矩阵来处理。交换矩阵可以被软件配置，以对中断优先级和中断屏蔽等做出设定。门户的另外一个作用是当来自某个中断源的中断正在被处理时，阻止接收同一中断源的后续中断。

对某个 HART 来说，如果中断发生，交换矩阵会通知 HART，而这种通知的方式可以有多种实现方式。对于复杂的系统，这种通知本身就可以是 MSI；对于相对简单的系统，这种通知可以是简单的硬连线，直接连接到 HART 内部中断寄存器的等待中断位上。

HART 在收到来自交换矩阵的中断通知后，需要读取对应的读取 / 完成寄存器来确定中断源。读取 / 完成寄存器是一个内存映射寄存器，当其被读取时，会返回中断源的 ID（Indentifier）。同时，读取寄存器的动作也会被 PLIC 认定为对中断

的读取，从而修正 PLIC 中中断等待的状态。

当 HART 结束对中断的处理后，需要将刚才处理完成的中断源 ID 再写入读取 / 完成寄存器。PLIC 在收到这个写入动作后，会修改门户的状态，以允许接收对应中断源的后续中断。

注意：这里要提醒读者注意的是，PLIC 只包括对外部中断的处理。为方便 RTC 的实现，RISC-V 标准中还专门定义了时钟定时器。而时钟定时器的中断属于局部中断（Local Interrupt），其在 HART 中有专门的寄存器位对应。其他的局部中断还包括软件中断和处理器设计者的自定义局部中断。这些 Local Interrupt 会在后文讨论中断相关的寄存器时做详细讨论。在 SiFive 公司的 Freedom E31 处理器中，将这些同局部中断相关的寄存器（时钟定时器寄存器、软件中断寄存器等）统称为核局部中断寄存器（Core Local Interruptor，CLINT）。所以对 HART 来说，其完整的中断拓扑结构如图 3-38 所示。

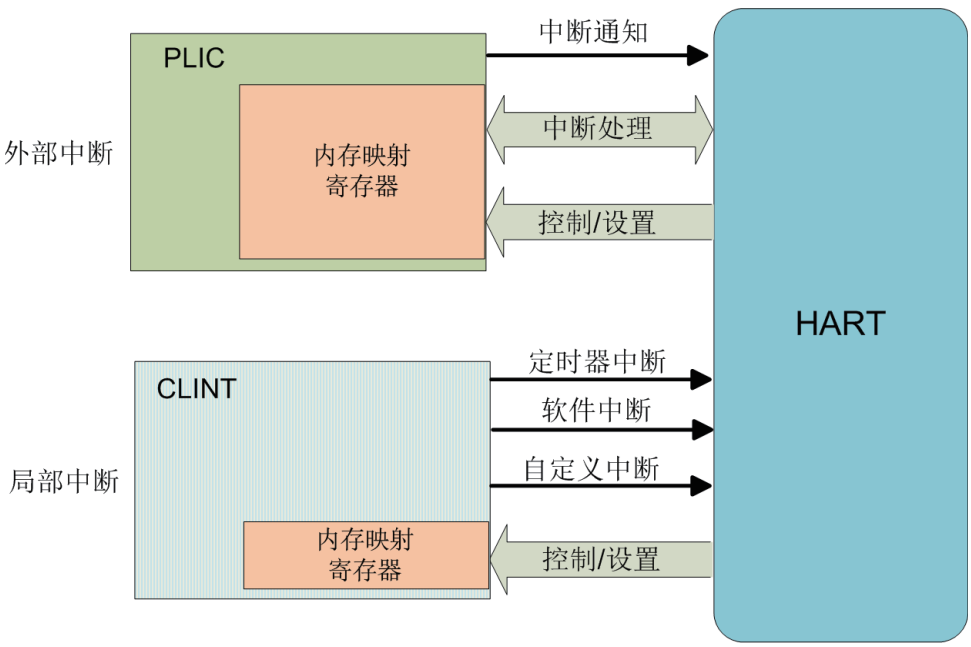


图3-38　外部中断和局部中断

这里笔者想借此机会对 PLIC/CLINT 的设计发表一些个人见解，供读者参考：

（1）RISC-V 的设计者对大规模的多处理器系统做了很多考量。从图 3-37 可以看出，RISC-V 的外部中断控制器有很好的可扩展性。然而，对于单个 HART 的仅支持机器模式的处理器核来说，这种结构显得比较复杂。同时和图 3-36 相比，这种结构并不是纯粹的中断向量控制器结构。软件依然需要通过读取内存映射寄存器来确定中断源，而不是由硬件支持的中断向量表来直接调用 ISR。所以 PLIC 依然会有比较大的中断延迟。

（2）在图 3-38 的架构下，如果需要进一步减少中断延迟，则可以通过 CLINT 中的自定义局部中断来实现。然而，在讨论相关的 CSR 寄存器时可以发现，RISC-V 对自定义局部中断的向量化处理并没有很完整的定义，向量编码可扩展的空间也非常有限，从而给 VIC 的实现造成了障碍。这就导致在目前的架构下，RISC-V 处理器的设计者不得不做一些额外的非标准设计来适应对中断延迟有严格要求的场合。

（3）在本书撰写之际，RISC-V 基金会发布的最新官方标准包括用户指令集 20190608 版和特权架构 20190608 版。上述对外部中断和局部中断的处理架构的讨论，都是基于这两个官方标准和之前的较早版本。然而，随着 RISC-V 在嵌入式系统中的应用和普及，RISC-V 的设计者可能也意识到了目前中断处理架构的不足。所以 SiFive 公司又提议了一个新的中断处理的架构标准，叫作 CLIC（Core-Local Interrupt Controller，核局部中断控制器），并将其用在了 SiFive 公司的 E20 处理器上。CLIC 的结构如图 3-39 所示。

CLIC 可以被看作是 PLIC 和 CLINT 的合并与简化。图 3-39 的外部中断主要是用来在大规模系统与更高层级的 PLIC 相连的。实际上大部分的外设都可以直接被连接到 CLIC 上。同时 CLIC 架构标准中还定义了一些新的 CSR 寄存器，例如 mtvt（machine trap vector table，机器模式异常向量表）等，用来加强对中断向量的支持。

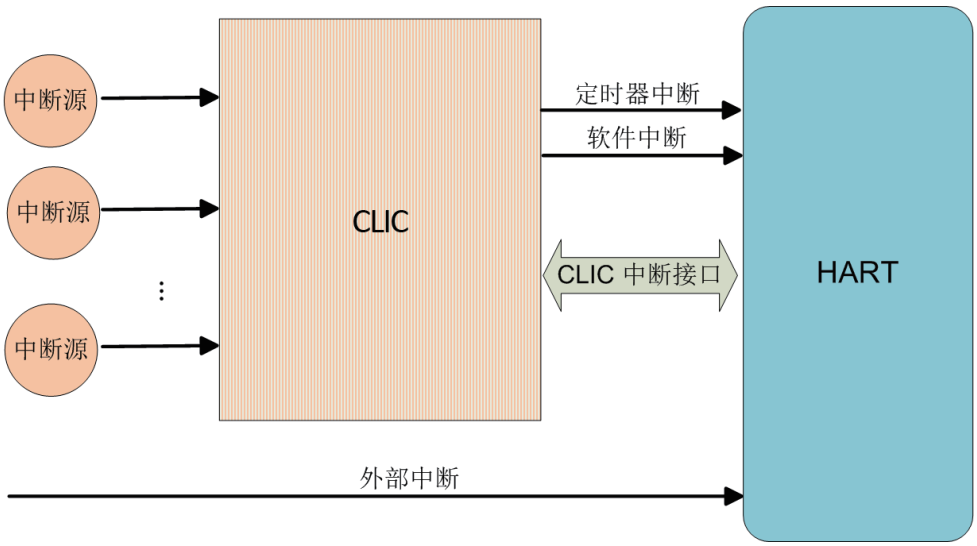


图3-39　CLIC结构图

提示：在本书撰写之际，CLIC 架构标准还只是处在提议和草稿阶段。由于其目前还不是 RISC-V 的官方标准，本书在后续章节将不会对其再做进一步的深入讨论。有兴趣的读者可以在 GitHub 上找到更多的相关内容。

3. RISC-V 中断和异常的触发

在 RISC-V 中，对中断和异常的处理方式非常相近。二者一般都可以被称作异常情况。对于单个 HART 的机器模式，当异常情况发生时，硬件一般要经历以下的处理步骤：

（1）确定中断是否被屏蔽。

对于单个 HART 的机器模式，下面两个 CSR 寄存器会影响中断的屏蔽。

① mstatus 寄存器中的 mie 位（见表 3-16），这是全局的中断使能位。但是该位不会屏蔽异常处理。

② mie（machine interrupt register，interrupt enable，机器模式中断使能寄存器）寄存器中的相关位。在 RV32 下，mie 寄存器是一个 32 位的可读写寄存器，其与机器模式相关的位如表 3-17 所示。

表 3-17　mie 寄存器的位定义

位　索　引	注　　释
3	机器模式软件中断使能 (msie)
7	机器模式定时器中断使能 (mtie)
11	机器模式外部中断使能 (meie)
[31 : 12]	供用户自定义扩展

这里对 RISC-V 特权架构中定义的软件中断做一下讨论。在 RISC-V 中，机器模式软件中断的主要目的是提供一种手段，用来在多 HART 系统中中断其他的 HART。为此，处理器的设计者需要在 CLINT 部分提供一个内存映射寄存器（或寄存器位），称为 msip（machine software interrupt pending，机器模式软件中断等待寄存器）。对 msip 的写操作会触发软件中断。

（2）确定异常情况发生的原因。

当中断或异常发生时，处理器需要正确填写 CSR 寄存器 mcause 中的相关内容。对于 RV32 来说，机器模式异常原因寄存器 mcause 是一个 32 位的可读写寄存器（这意味着软件也可以修改其内容）。mcause 的 MSB，即位 31 被用来标识这个异常情况是中断还是异常。如果是中断，则该位应该被置为 1；如果是异常，则该位应被置为 0。mcause 剩下的位被用来作为异常编码。虽然在标准中称其为异常编码，但其也包括中断的情况（在中断情况下，异常编码实际上是中断源编号）。在目前的标准中，只用到了其中的低 4 位。对于单个 HART 的机器模式，如果异常情况是中断，则相关中断源编号如表 3-18 所示。如果异常情况是异常，则对应的异常种类编码如表 3-19 所示（细心的读者也许会发现，表 3-17 与表 3-18 的位定义是一样的）。

表 3-18　mcause 的中断源编号

异 常 编 码	中断源编号
3	机器模式软件中断
7	机器模式定时器中断
11	机器模式外部中断

表 3-19　mcause 的异常种类编号

异常编码	异常种类
0	指令地址没有对齐
1	取指失败
2	非法指令
3	断点
4	内存数据读取地址没有对齐
5	内存数据读取失败
6	内存数据写入地址没有对齐
7	内存数据写入失败
11	机器模式下的环境调用

（3）确定异常情况发生的地址。

对于机器模式，RISC-V 在其特权架构标准中定义了 mepc（machine exception program counter，机器模式异常程序计数器）寄存器，用来存放异常情况发生时的程序计数器的值。对于异常来说，当前触发异常的指令的 PC 值是一个重要参数，所以 mepc = PC。而对中断来说，mepc 值则会被中断处理程序末尾的 MRET（M-Return）指令用来作为中断返回地址。所以，mepc 需要存放下一条指令的地址。

（4）确定与异常情况相关的参数。

为了帮助异常情况的处理，RISC-V 还在其特权架构标准的机器模式中定义了 mtval 寄存器，以提供与异常情况相关的参数。在 RV32 下，mtval 是一个 32 位的可读写寄存器。当内存访问出现异常时，对应的内存读写地址应该被保存在这个寄存器里。

（5）改变 PC 值，调用中断 / 异常处理程序，并设置相应的中断比特状态位。

对于机器模式，RISC-V 在其特权架构标准中定义了 mtvec 寄存器，用来确定异常情况处理程序的地址，在 RV32 下，这是一个 32 位的可读写寄存器。其中的低两位用来确定中断模式，其余高 30 位被用来作为基地址（BASE）。中断模式的定义如表 3-20 所示。

表 3-20　mtvec 的中断模式定义

中断模式	中断方式描述
0	直接模式。在该模式下，新的 PC 值被直接设置为 mtvec 中的基地址值。即 PC = BASE
1	向量模式。在该模式下，新的 PC 值被设置为 PC = BASE + 4 × Exception_Code 这里的异常编码即为 mcause 寄存器中的异常编码，如表 3-18 和表 3-19 所示
2，3	保留供未来扩展

由于 mcause 中的异常编码目前只有 4 位，其大部分已被占用，而 RISC-V 官方标准中也没有定义专门的 CSR 来支持中断向量表（Interrupt Vetor Table，IVT），所以表 3-20 中的向量模式并不能很好地对来自外设的中断进行类似 VIC 这样的向量化支持。目前还在草案和提议阶段的 CLIC 标准将改变这一状况，在目前的 CLIC 标准提议草案中，已经对表 3-20 中的模式 2 与模式 3 做了扩充。

为对应中断的情况，硬件还需要将 mip 寄存器中的相应位设置为等待。mip 寄存器中的位定义如表 3-21 所示。很显然，表 3-21 中的位定义与表 3-17 中的位定义是一一对应的。

表 3-21　mip 寄存器的位定义

位　索　引	注　　释
3	机器模式软件中断状态位 (MSIP)
7	机器模式定时器中断状态位 (MTIP)
11	机器模式外部中断状态位 (MEIP)
[31 : 12]	供用户自定义扩展

4. RISC-V 中断和异常的返回

在机器模式下，当异常情况处理程序完成所有操作后，需要调用 MRET（M-Return，机器返回指令）指令返回。MRET 指令的定义如图 3-40 所示。当处理器遇到 MRET 指令时，应将 PC 值置为 mepc 寄存器中的值，这样指令从之前被异常情况打断的地方继续执行。

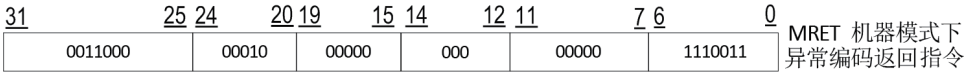


图3-40 MRET指令

5. WFI（中断等待指令，Wait for Interrupt）

为了给操作系统多提供一个调度的方法，RISC-V 在其特权架构标准中还定义了中断等待指令，如图 3-41 所示，当处理器遇到该指令时，则进入停顿状态，直到中断的发生。

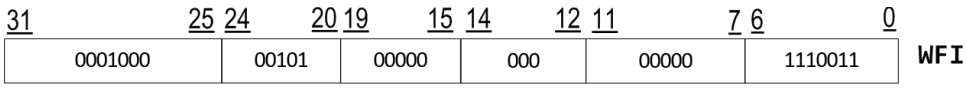


图3-41 WFI指令

当中断发生时，处理器会设置 mepc = PC + 4（即 WFI 之后的那条指令的地址）。在机器模式下，当中断处理结束，MRET 返回时，则会将 PC 设置为 mepc 的值，从而使得处理器会执行 WFI 之后的那条指令。

6. 环境调用与断点

为了给操作系统和软件调试提供更多调度和中断的方式，RISC-V 标准中还定义了环境调用指令 ECALL（Environment Call）和断点指令 EBREAK（Environment Breakpoint），它们的定义如图 3-23 所示。当处理器遇到 ECALL 或 EBREAK 指令时，都会产生异常。其中 ECALL 在机器模式下的异常编码是 11，而 EBREAK 的异常编码是 3（参见表 3-19）。

RISC-V 的特权架构标准中特别强调，当遇到 ECALL 和 EBREAK 指令时，应该将 mepc 寄存器（此处仅讨论机器模式）的值设置为当前指令的地址，而不是下一条指令的地址。细心的读者也许会问：“如果是这样，当异常处理结束时，调用 MRET 指令时，岂不是又回到了原来的 ECALL/EBREAK 指令，陷入重复执行的死循环？”对此，笔者就以 GDB 中的软件断点的操作为例来具体解释。

相信许多读者都对 GDB（GNU Debugger）不陌生。当我们需要在 GDB 中设置软件断点时，一般的做法是在 GDB 命令行中键入“break 断点地址”。当处理器执行到该断点地址时，软件中断被触发后，我们可以检查寄存器的值或读取内存

中的内容，然后用 continue（继续执行）命令来继续程序的执行。在这些调试操作的背后，GDB 到底做了些什么?

如图 3-42 所示，当在 GDB 命令行中键入“break 断点地址”后，调试器会将内存对应内存地址中的指令换作 EBREAK 指令。

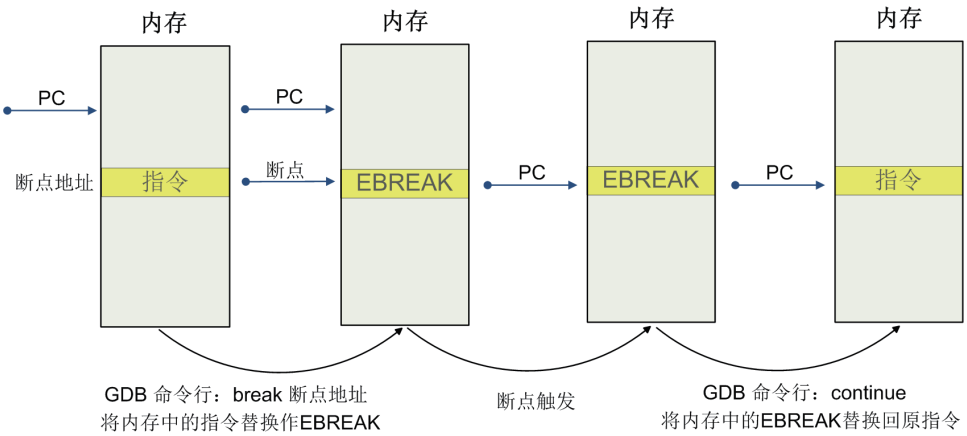


图3-42　软件中断

随后，当处理器运行到对应的断点地址时，会触发 EBREAK 断点异常，进入调试器事先准备好的断点异常处理程序中。在这里，用户可以查看寄存器和内存中的内容，以帮助调试。

当用户完成寄存器和内存内容查看后，可以在 GDB 命令行中键入“continue”以继续运行程序。但是在继续运行之前，GDB 会将内存中的 EBREAK 再替换回原先的指令，以避免调试器可能带来的副作用。这就是为什么 mepc 应该被设置为断点地址，而不是指向断点地址之后的那条指令。

3.6.5　程序的调试

提到了断点和 GDB，在 RISC-V 已经发布的官方标准中，除了用户指令集与特权架构外，还包括了一个“外部调试器支持”标准（External Debugger Support）。

提示：不过与前两者不同的是，笔者始终无法找到外部调试器支持标准在 1.0 以上的版本。在本书被撰写之际，该标准的最新官方版本是 0.13.2。鉴于这种情况，本书对调试器仅做一般性的讨论，且仅限于嵌入式系统的程序调试。

对于嵌入式系统来说，调试器主要有两种实现方式。

1. 软件方式：ROM Monitor（只读存储器监视器）/GDB Stub（存根）

如图 3-43 所示，在这种软件实现的调试器下，调试软件（例如 GDB、GNU Debugger）运行在主机电脑上，而目标系统中会首先运行一个叫 ROM Monitor 的程序。对于 GDB 的情况，这个 ROM Monitor 也被称作 GDB Stub。GDB Stub 会通过网络或者串行口接收用户通过 GDB 发来的调试指令，将被调试的应用程序从主机载入到目标系统中，并设置软件断点等。当应用程序运行并触发了软件断点后，处理器的控制又回到 GDB Stub 手中，然后由用户做进一步调试。GDB Stub 支持的常用功能包括应用软件的载入、软件断点的设置、寄存器和内存的读取等。

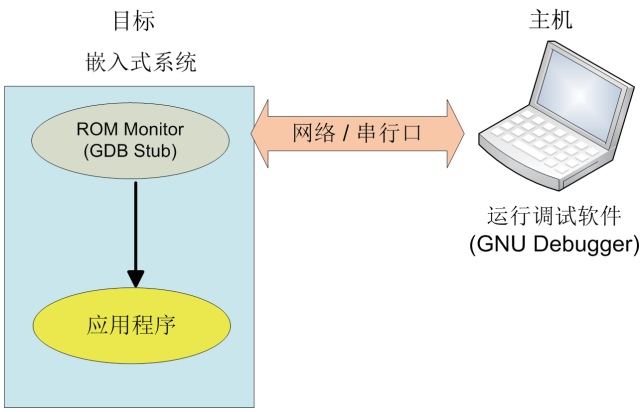


图3-43　ROM Monitor / GDB Stub

对于这种用 ROM Monitor/GDB Stub 来实现调试器的方法，其优点是不需要额外的硬件支持，其硬件开销比较小。但是其缺点是：

（1）其支持的功能比较有限。

（2）由于 GDB Stub 需要占用一定的内存，并与应用程序共存于系统中，

其调试的方式有侵入性。这种 ROM Monitor / GDB Stub 的方式对裸金属系统还能胜任，而对于比较复杂的嵌入式操作系统则会显得力不从心，甚至还会发生资源冲突。

2. 硬件方式：JTAG（Joint Fest Action Group，联合测试工作组）

这也是 RISC-V 和其他的嵌入式处理器所采取的方式。这种方法除了需要处理器本身的硬件支持外，还需要借助一个外部的调试控制器（见图 3-44）或调试适配器（见图 3-45）。和前一种方式相比，这种调试方式很大程度上增加了硬件的开销，而且其功能和稳定性也大为提高。

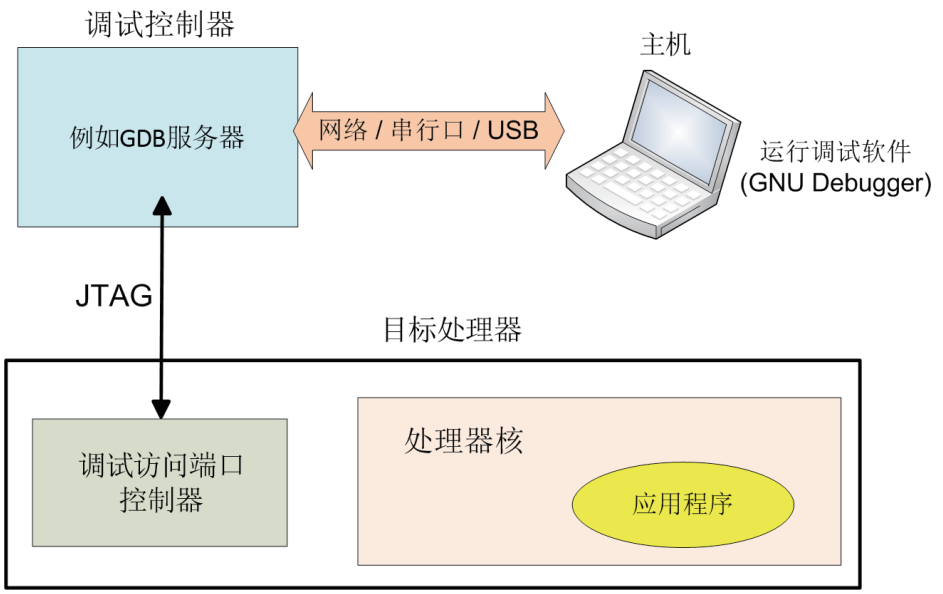


图3-44　硬件调试控制器

在这种调试方式下，处理器一般需要支持 JTAG 接口。同时，在主机上运行的 GDB 不再直接与处理器交换信息。取而代之的是，GDB 会同另外一个叫作 GDB 服务器的软件进行对话。GDB 服务器可以运行在主机之外的外部硬件上（见图 3-44），此时这个外部硬件被称作调试控制器。当然，GDB 服务器也可以和 GDB 运行在同一台主机上，而用一个相对简单的外部硬件来收发 JTAG 信号。此时这个外部硬件被称作调试适配器。

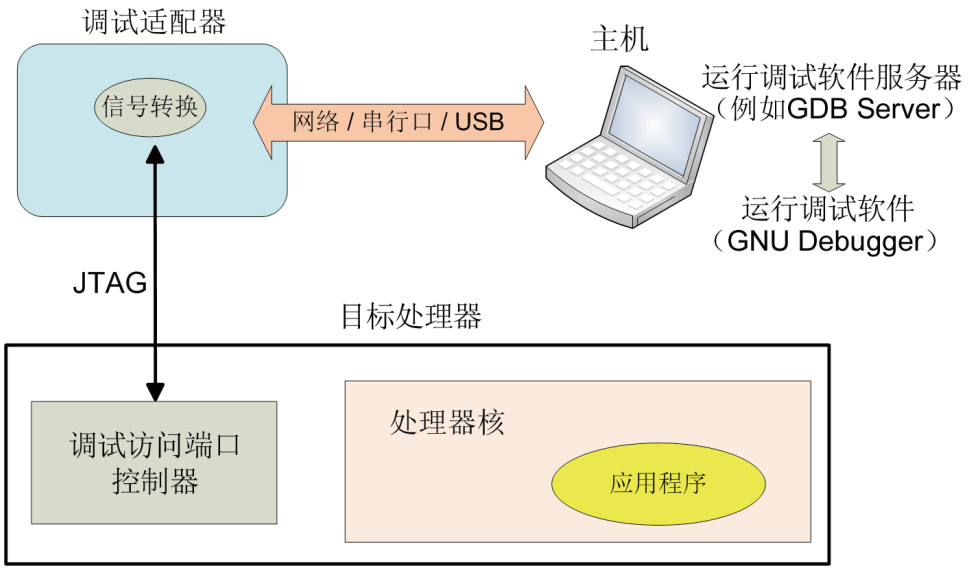
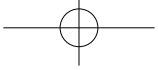
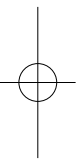
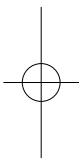


图3-45　硬件调试适配器

不论是调试控制器还是调试适配器，整个调试系统的本质都是通过 GDB 服务器将 GDB 的命令转换为相应的 JTAG 操作，并通过独立于处理器的外部硬件来实现这些 JTAG 操作。在实践中，许多 RISC-V 处理器都会采用 Open OCD（Open On-Chip Debugger，开源片上调试器）作为调试软件，而 Open OCD 实际上起到了 GDB 服务器的作用。

第 4 章

设计基于 RISC-V 指令集的 Soft CPU

口言之，身必行之。

《墨子·公孟》

4.1 2018 RISC-V Soft CPU Contest 获奖作品：PulseRain Reindeer

说明：在英语中有句俚语“If you’re going to talk the talk，you’ve got to walk the walk.”。本章将讲解一个由笔者主持设计的 RISC-V RV32IM MCU Core，以展示如何在 FPGA 中实现 RISC-V 软核处理器。

这个 RISC-V 软核处理器名叫 PulseRain Reindeer，其源自于美国 PulseRain Technology 公司内部产品线的一个简化版本。在 2018 年由 RISC-V 官方组织的 RISC-V Soft CPU 竞赛中，该软核处理器位列季军（https://riscv.org/2018/10/risc-v-contest）。在本书创作之际，笔者有幸能与国内的小脚丫团队合作，将软核处理器做了进一步的改进与提升，并顺利移植到了小脚丫综合实验平台上。读者可以从 PulseRain Technology 在 GitHub 的官方账号上找到其完整的源代码，这些源代码会在本书的代码资源中提供，其文件名是 Reindeer_Step-1.1.2.zip。

PulseRain Reindeer 的处理器核心采用 Verilog 2001 编写，其余的外设等部分采用 System Verilog 编写，并部分引用了 PulseRain Technology 的 PulseRain RTL 库。这个处理器在开发过程中遵循了本书提出的 FARM 开发模式，在设计之初就对软件的配套做了考量，并在软硬件设计上做了安排。由此，当在 FPGA 中加入 PulseRain Reindeer 软核后，与本书配套的小脚丫综合实验平台就可以作为第三方开发板，直接整合到 Arduino IDE 集成开发环境中。与此对应的 Arduino 支持包（板级支持包），也已经在 GitHub 上公开发布。在本书的代码资源中，其对应的文件名是 Arduino_RISCV_IDE-master.zip。

PulseRain Reindeer 软核还有非常好的通用性和可移植性。除了在 Intel 等主流的 FPGA 架构使用，该软核也在一些高性价比的 FPGA 新架构（例如 EFINIX 公司的 Quantum 架构 FPGA）上得到了成功的移植和验证。在 EFINIX Trion T20/C4 FPGA 上，该软核可以达到超过 110 MHz 的时钟主频。对此感兴趣的读者，可以

在 PulseRain Technology 的官方 GitHub 上找到与此相关的移植代码。在 Arduino 支持包里，也已经有了对 EFINIX Trion T20 开发板的支持。

笔者将会结合 FPGA 的器件特点，对 PulseRain Reindeer 的内核设计进行详细讨论。

4.2 适合于FPGA的设计目标

正如在讨论FPGA时提到的AT^2定律一样，数字设计在很多时候是在“鱼”与“熊掌”之间寻找一个合适的平衡点。在设计之初，有必要明确定义设计所追求的目标，特别是要在多个互相冲突的设计指标之间作出取舍。

就基于 FPGA 的 RISC-V 软核处理器来说，笔者认为其设计重点应该集中在以下几方面：

（1）软核处理器应该以 MCU（Microcontroller）设计为主。

在讨论“数字逻辑与处理器各自适用的领域”（2.15 节）时曾经提到，所有的任务都可以被归为两类：控制密集型和处理（计算）密集型。FPGA 含有大量的数字逻辑资源，比处理器更适合于处理（计算）密集型的任务。但是对控制密集型的工作，普通的数字逻辑却往往力不从心，而在 FPGA 中嵌入软核 MCU，则可以很好地弥补这一缺点（见图 2-33）。

（2）软核处理器应该能达到比较高的主时钟频率。

由于控制密集型的任务和处理（计算）密集型的任务不可避免地会有交互，如果嵌入 FPGA 中的软核 MCU 能和其他的电路工作在同一频率下，则可以避免时钟域跨越，并简化数据交换的方式。

对处理器设计来说，除了时钟主频以外，还有一个重要的指标就是 CPI（Clocks Per Instruction，指令的平均周期数）。追求高主频，在某种程度上会对 CPI 造成负面影响。这是因为高主频往往意味着更长的流水线设计。当

跳转预测失败时，往往需要清空流水线（Flush the Pipeline），并重新取指，长流水线在此时会需要更多的时钟周期来重新装载。

在 FPGA 中嵌入软核处理器的主要目的是为了做控制，而不是处理，即该软核处理器并不需要追求非常强大的计算能力和计算效率。因此，和时钟主频相比，CPI 在这里可以看作一个次要的设计指标。

（3）软核处理器应该尽量降低对 FPGA 资源的消耗。

根据 AT^2 定律，这个设计指标在某种程度上是与“追求高主频”相冲突的。随着 FPGA 器件容量的不断提升，笔者倾向于把该指标的优先级放在“追求高主频”之后。

有一点要指出的是，FPGA 的资源除了逻辑资源以外，还包括片上内存。而由于软核处理器需要存储程序和数据，会消耗比较多的内存，而片上内存往往是 FPGA 的紧缺资源。以与本书配套的小脚丫综合实验平台为例，其采用的 FPGA（Intel Cyclone 10 LP 10CL016YU256C8G）包含多达 15 000 个逻辑单元，其片上内存却只有 56 KB。如果需要在软核处理器运行嵌入式操作系统，则这些内存会显得捉襟见肘。

为了节约宝贵的 FPGA 片上内存，可以有两种解决方法：

① 在软核处理器中支持 C Extension，以提高代码密度。

根据 Andrew Waterman 的博士论文，C Extension 可以将代码密度提高约 40%。但其代价是处理器的设计变得复杂，并占用更多的逻辑资源。考虑到 FPGA 的其余部分也需要消耗片上内存，这种方法所能带来的改变非常有限。

② 在软核处理器中支持片外内存的访问。

由于 DRAM 的存储密度要比 SRAM（Synchronous Dynamic Random Access Memory，同步动态随机存取存储器）高出许多，如果 FPGA 中的软核处理器能访问片外的 DRAM，则可以将大部分的片上内存用于除了软核处理器之外的其他功能。这个方法的代价就是需要额外的逻辑资源来实现 DRAM 控制器。

考虑到大部分的软核 MCU 的时钟主频都低于 200 MHz，一个比较实用的方案就是在 FPGA 之外放置一个 SDRAM（常用的规格有 PC-100、PC-133 等）。和 DDR 相比，SDRAM 的控制器相对比较简单，其逻辑开销也较小，而且很多 FPGA 厂商都会提供现成的 IP。与本书配套的小脚丫综合实验平台便采用了这一方案，在 FPGA 之外配置了 8 MB 的 SDRAM，用来作为 PulseRain Reindeer 的代码和数据内存。

（4）软核处理器应采用（冯·诺依曼架构）。

从处理器内存架构的角度来说，目前主要有两种选择：冯·诺依曼架构（von Neumann Architecture）与哈佛架构（Harvard Architecture）。

如图 4-1 所示，冯·诺依曼架构的核心思想是“存储程序”。在冯·诺依曼架构下，程序代码和数据被不加区分地存放在同一个物理内存中。其优点是由于只有一条内存总线，控制相对简单，内存控制器的开销比较小；其缺点是这条唯一的内存总线会成为提升系统性能的瓶颈。

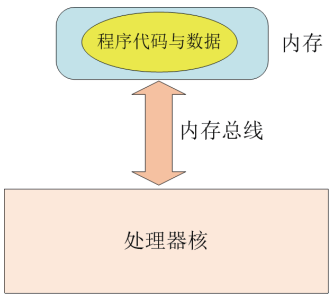


图4-1 冯·诺依曼架构

而哈佛架构则将程序代码和数据在物理内存中分开存储。如图 4-2 所示，在哈佛架构中有两条内存总线，分别用来访问代码内存与数据内存。这种架构的优点是指令取指和数据读写有各自的专用总线，在物理内存中不会发生冲突，有利于系统性能特别是 CPI（Clock Per Instruction，执行某个程序的指令平均时钟周期数）的提升，其缺点是内存控制器的开销较大。特别是由于代码内存和数据内存在物理内存中分为两块，缺乏总体调度的灵活性，给内存使用效率和软件开发带来负面影响。

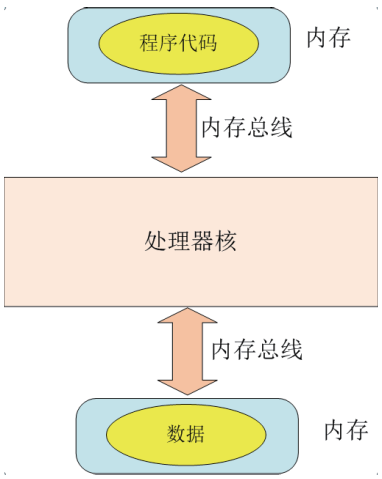


图4-2　哈佛架构

在 FPGA 中嵌入软核处理器的主要目的是为了完成控制任务，而不是要追求非常强大的计算能力和计算效率，因此哈佛结构所带来的 CPI 性能提升只是一个次要的设计指标。而同时，如果软核处理器采用 FPGA 片上内存来存储程序代码和数据，则哈佛架构这种双内存设计会让有限的片上内存变得更加左支右绌；如果采用片外内存，则只支持单总线结构。

由此，笔者建议该软核处理器应采用冯·诺依曼架构，而不是哈佛架构。

（5）软核处理器应该要方便软件的开发设计。

与 FARM 开发模式相呼应，软核处理器中还带有一些额外的功能模块，以方便软件的开发设计，特别是对于 Arduino IDE 集成开发环境的支持。

4.3 PulseRain Reindeer的设计策略

基于以上对设计目标的讨论，PulseRain Reindeer 处理器采用了如下的设计策略。

1. 采用了 2×2 的流水线设计，内存布局采用冯·诺依曼架构

为了追求较高的时钟主频，PulseRain Reindeer 处理器中包含有 4 级流水线。

- 取指（Instruction Fetch）。
- 指令译码（Instruction Decode）。
- 指令执行（Execution）。
- 数据访问（Data Access）包括寄存器的更新与内存的读写。

与普通的 4 级流水不同的是，PulseRain Reindeer 对这 4 个流水线阶段采用了 2×2 的布局，如图 4-3 所示。在这种布局下，在双数时钟周期下，只有“取指”和“指令执行”这两个阶段是活跃的。而在单数时钟周期，只有“指令译码”和“数据访问”这两个阶段是活跃的。

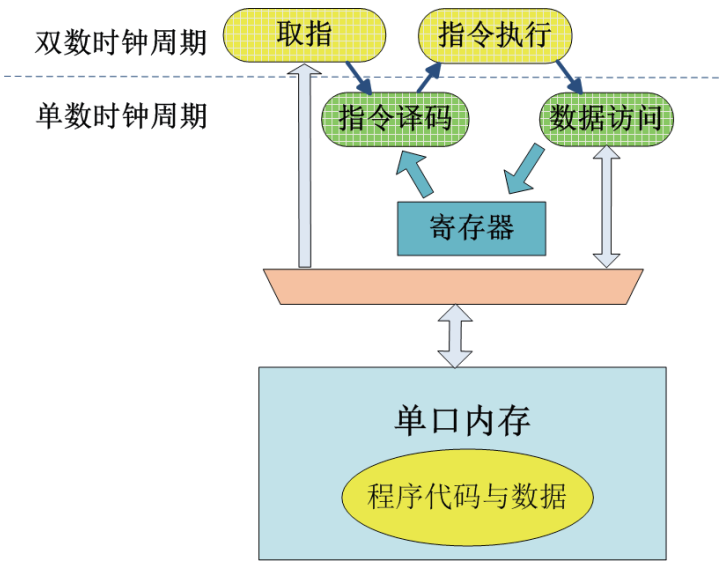


图4-3　2×2 流水线设计

采取这种布局主要是出于以下考虑：

（1）FPGA 的内部结构不同于普通的数字芯片。在 FPGA 中，寄存器并不是最稀缺的资源，而减少大块的组合逻辑则往往是降低走线资源消耗、提高时钟频率的关键。PulseRain Reindeer 采用多级流水线结构便是出于此目的。

（2）而采用 2×2 的流水线布局，则以牺牲 CPI 为代价，减小对逻辑资源的消耗。同时，在这种布局中，指令取指和内存数据访问被安排在不同的时钟周期，从而避免了冯·诺依曼架构的单内存总线带来的内存访问难题。

（3）作为比较，PulseRain Reindeer 在初始设计阶段也曾经考虑过两级流水，以简化控制，并提高 CPI。通过将设计原型在 Intel Cyclone 10 C8 级别上的布线测试后发现，这种两级流水设计的时钟主频在 70 MHz 左右就发生了时序收敛的困难，而 4 级流水则可以在同样的 FPGA 器件上运行超过 100 MHz 的时钟频率。

2. 支持 FPGA 片上内存与片外内存的混合使用

与访问 FPGA 片上内存不同的是，片外内存往往都有比较大的访问延迟。当片上内存与片外内存混合使用时，情况就变得比较复杂。在内存控制单元和流水线的设计上，PulseRain Reindeer 为此做了调校，以支持片上内存与片外内存的混合使用。

说明：在与本书配套的小脚丫综合实验平台上，PulseRain Reindeer 可以被灵活配置，以同时支持片上内存与片外的 SDRAM 访问。由于大容量片外内存的存在，使得 PulseRain Reindeer 无须再支持 C Extension，从而减少了对 FPGA 逻辑资源的消耗。

3. 支持基于硬件的引导加载程序（加载器）

在本书开头，提到了 FARM 开发模式，其中谈到了对 Arduino IDE 集成开发环境的支持。这里只想提及一下其中的程序 Image 下载部分，这个问题实际上并非 Arduino 独有，而是一般性的问题。

传统的下载办法（实际上也是 Arduino 采用的办法），便是在处理器上预先运行一个称为 Bootloader 的软件，通过这个软件同主机上的上传工具通信，来下载程序 Image，如图 4-4 所示（实际上图 4-4 可以看作是图 3-43 的简化版）。

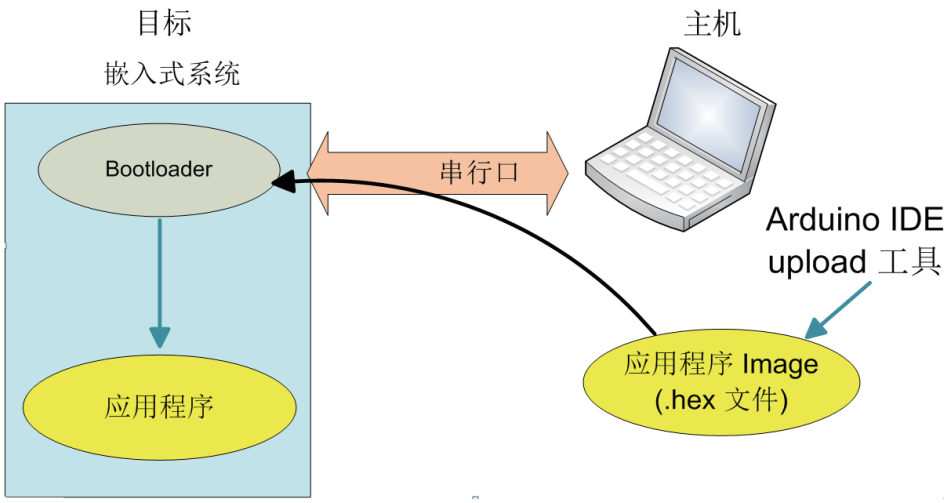


图4-4　传统程序Image下载方式

图 4-4 这种方法的缺点是需要在上电以后将 Bootloader 载入到处理器的内存中。一般的做法是将 Bootloader 放入 ROM，并映射到处理器的地址空间中。对于 FPGA 的软核处理器，则可以将 Bootloader 代码直接作为比特流的一部分，用来初始化片上内存。但是这种做法除了要占用相当可观的片上内存外，还存在可移植性的问题，并非所有厂商的 FPGA 都会支持将内存初始化数据存放于比特流中。

为了更好地支持 FARM 开发模式，PulseRain Reindeer 除了软核处理器本身，还为之配套设计了一个基于硬件的引导加载程序（Hardware Based Bootloader）。如图 4-5 所示，这个基于硬件的 Bootloader 会与处理器核共享同一个串口，并且它还会与处理器核中的内存控制器协调工作，以将程序 Image 载入 FPGA 片上内存或片外内存中。和传统的下载方法相比，这种基于硬件的 Bootloader 不需要任何 ROM 来存储代码，并且它本身可以被用来复位和启动处理器核，以及提供复位后的初始地址，从而比传统方法更稳定与灵活，在不同 FPGA 器件之间的可移植性也比较好。

与之相对应的是（见图 4-6），Arduino IDE 在主控端会运行一个 Python 脚本（reindeer_config.py）作为上传工具。这个 Python 脚本还可以独立于 Arduino IDE 单独运行。对 elf 文件，这个 Python 脚本会调用工具链，将 elf 文件中的相关部分截取出来，并通过基于硬件的引导加载程序载入到内存中。

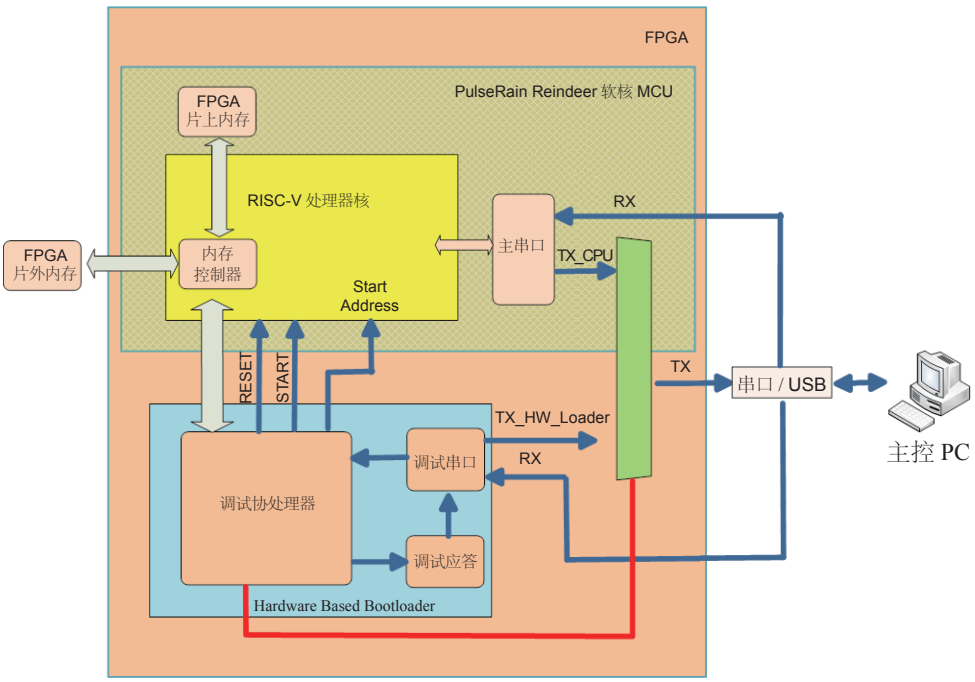


图4-5　基于硬件的引导加载程序

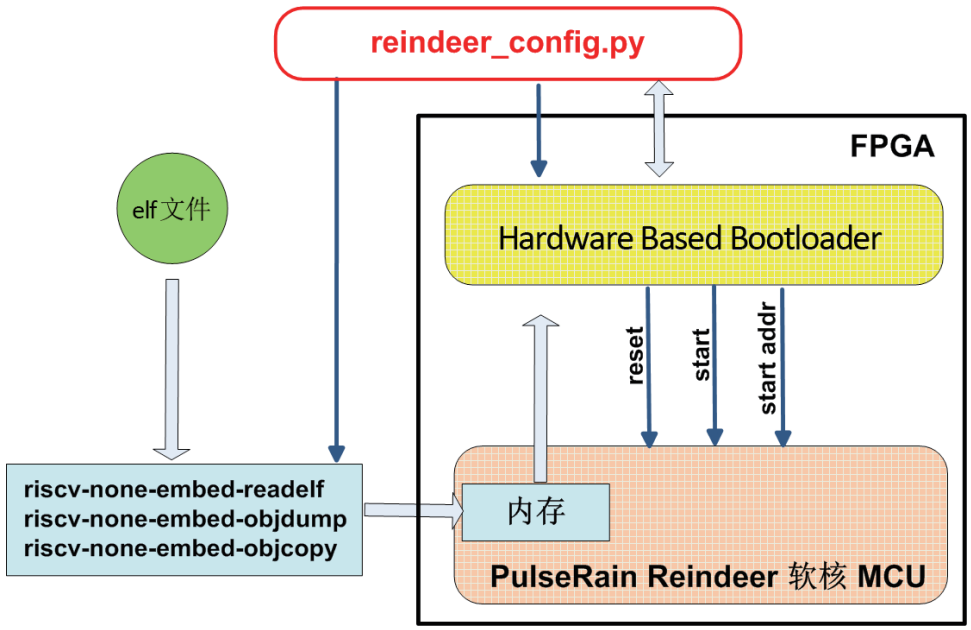


图4-6　用Python Script载入软件

4.4 PulseRain Reindeer的RTL设计

4.4.1 与 FPGA 平台相关部分

从软核 MCU 移植性的角度来说，可以将整个 FPGA 划分为两部分：①与 FPGA 平台相关部分；②独立于 FPGA 平台部分。

对于具有 PulseRain Reindeer 软核 MCU 的 FPGA 来说，整个 FPGA 的顶层架构如图 4-7 所示。将 PulseRain Reindeer 软核 MCU 移植到不同的 FPGA 平台上时，需要对应的平台提供以下模块。

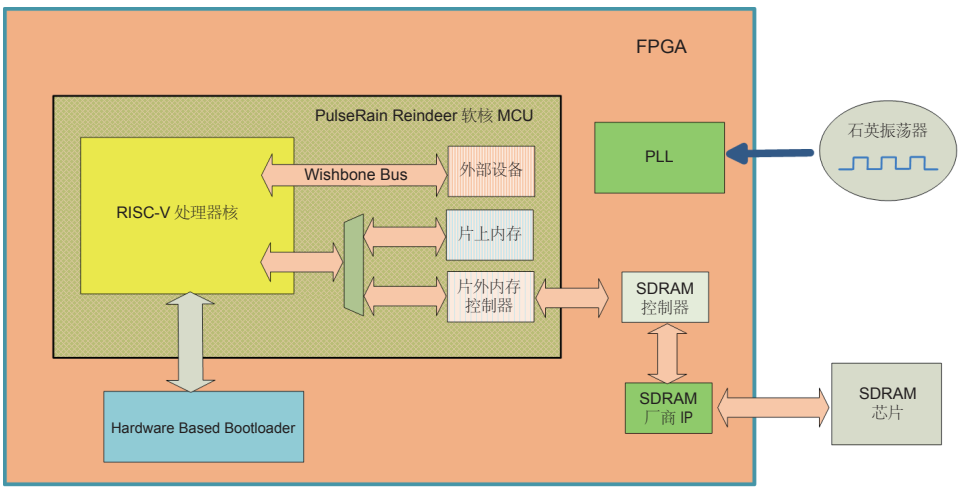


图4-7 FPGA顶层架构

1. PLL（Phase Locked Loop，锁相环）

PLL 一般由 FPGA 片外的石英振荡器提供时钟参考。在有些 FPGA 平台上，时钟参考也可以由片上的 RC 振荡器来提供。

2. SDRAM 厂商 IP

如果 PulseRain Reindeer 被设置成需要使用 FPGA 片外 SDRAM，则需要使用 FPGA 厂商提供的 SDRAM IP（对于 SDRAM，网上也可以找到一些开源的 IP）。

3. FPGA 片上内存

对于不同的 FPGA 平台，其片上内存的配置方式会略有不同。例如在 Intel 公司提供的 FPGA 中，就有 M9K（每块内存 9 Kb），M10K（每块内存 10 Kb），M20K（每块内存 20 Kb）等多种不同的片上内存种类。在目前的 PulseRain Reindeer 的 RTL 代码中，对片上内存采用了由综合软件根据代码推断（Infer）的方式。如果这种 Infer 的方式不能被厂商的 FPGA 综合软件完全认可，则需要根据 FPGA 厂商的指引加以重新配置。

4.4.2 独立于 FPGA 平台部分

在图 4-7 中左下角的 Hardware Based Bootloader 已经在图 4-5 中有详细描述，此处不再赘述。而图 4-7 中的 PulseRain Reindeer 软核 MCU 则在很大程度上与具体的 FPGA 平台无关，其内部细节如图 4-8 所示。

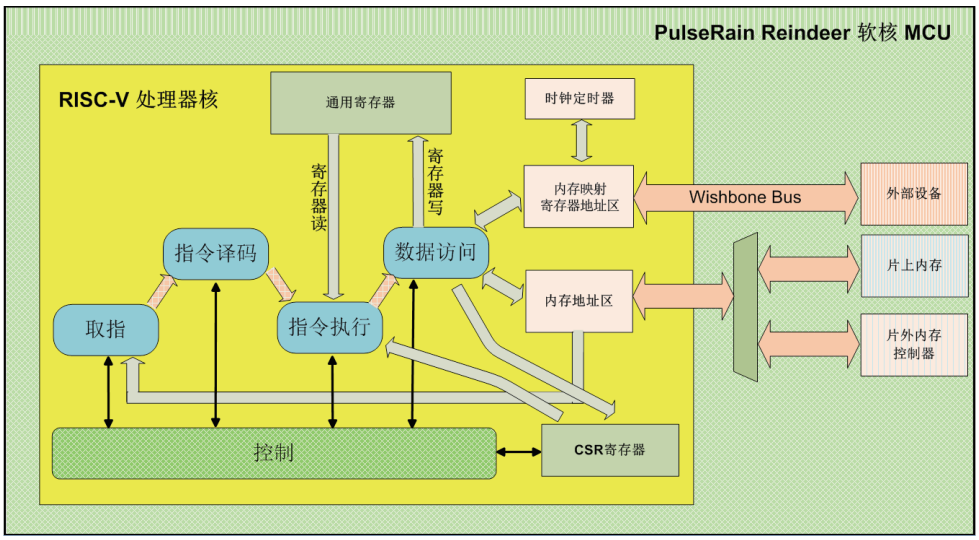


图4-8　PulseRain Reindeer 软核MCU

从图 4-8 可以看出，PulseRain Reindeer 软核 MCU 主要由三部分组成。

1. 外围设备

在 PulseRain Reindeer 软核 MCU 中，外围设备通过 Wishbone 总线和处理器核

相连。根据具体应用的不同，这些外围设备可以被灵活地定制化，这也是 FPGA 相对于普通数字芯片的优势之一。

一般常用的外围设备有：

- 串行口 UART（Universal Asynchronous Receiver/Transmitter）。
- I^2C 总线接口。
- SPI 接口。
- GPIO（General Purpose Input Output）。
- PWM（Pulse Width Modulation）。
- PS2。
- MicroSD。
- 旋转编码器（Rotary Encoder）。

2. 内存接口

PulseRain Reindeer 采用了冯·诺依曼架构，将程序代码和数据不加区分地存放于内存中。而对 FPGA 来说，内存又分为片上内存（Block RAM）和片外内存。片外内存控制器则需要通过一个在 MCU 之外的中间模块（例如图 4-7 中的 SDRAM 控制器）和具体的内存 IP 做数据交换。在与本书配套的小脚丫综合实验平台上采用了 SDRAM 作为片外内存，具体的做法将在后续章节讨论实验平台时做详细描述。

3. RISC-V 处理器核

处理器核部分包括通用寄存器、CSR 寄存器、内存地址分配、流水线的数据通路和控制等。

4.4.3 通用寄存器的设计

在 RISC-V 用户指令集标准（User-Level ISA）中提到，RV32 定义了 32 个 32

位的通用寄存器（其中 x0 恒为零值）。在 FPGA 中，如果直接用触发器来实现这些通用寄存器，则需要 32 × 32=1 024 个触发器。对于小脚丫平台上的 Intel Cyclone 10 LP（10CL016YU256C8G）FPGA，则根据图 2-1 中的逻辑单元结构，至少需要消耗相同数量的逻辑单元才能实现所有的通用寄存器（大约占该 FPGA 总逻辑容量的 7%）。

同时，通过观察 RISC-V 指令格式，可以发现许多 RISC-V 指令都包含两个源寄存器（标记为 rs1 和 rs2），即在同一指令中，需要读取两个通用寄存器。如果用触发器来实现通用寄存器，则同时还需要两个 32∶1 的多路复用器，每个多路复用器的数据宽度都是 32 位。

综合以上考虑，PulseRain Reindeer 中采用了两块简单双口 Block RAM 来实现通用寄存器，如图 4-9 所示。

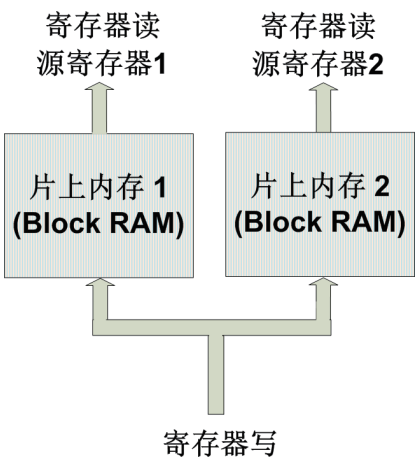


图4-9　用Block RAM来实现通用寄存器

在图 4-9 中，当寄存器被写入时，同样的数据会被同时写入这两块 Block RAM 中。而在寄存器读取时，这两块 Block RAM 分别对应源寄存器 1 与 2。

在图 4-8 所示的 4 个流水线阶段中，寄存器的读地址在“取指”阶段就可以确定。而寄存器的写地址和写数据会在“数据访问”阶段被确定。因为 PulseRain Reindeer 采用的 2 × 2 流水线设计，“取指”和“数据访问”发生在不同的时钟周期，所以不会产生由于对内存同时读写而造成的数据模糊（但是由于数据相关性而引

起的流水线阶段之间的转发问题依然会发生）。

Block RAM 的输出驱动能力一般都弱于触发器，如果让这些 Block RAM 的输出直接参与很多组合逻辑，则会对时序收敛产生负面影响。同时，Block RAM 会有一个时钟周期的读延迟。如果在“取指”阶段给出寄存器读地址，则数据会在“指令译码”阶段变得有效，而这些寄存器读数据会在“指令执行”阶段被用到。采用 2×2 的流水线布局后，可以在“指令执行”阶段将 Block RAM 的输出寄存后再操作，无须再做太多的数据相关性处理。Block RAM 的输出驱动能力弱，将 Block RAM 的输出寄存后再操作则有利于提高时钟主频。

4.4.4 CSR 寄存器的实现

表 3-14 所列出的 CSR 寄存器在 PulseRain Reindeer 中都得到了实现。与通用寄存器的数据读操作不同的是，对 CSR 这类控制寄存器的额外读操作可能会产生不必要的副作用，因此 CSR 寄存器的读地址和读使能要在“指令译码”阶段才能确定。CSR 寄存器都是用触发器实现的，不存在 Block RAM 这样的读延迟，所以读数据依然可以及时在“指令执行”阶段得到使用。

当中断或异常发生时，流水线会被暂停，而某些 CSR 寄存器，如 mtvec、mepc、mtval、mcause 等会在此时被读取或更新。

4.4.5 时钟定时器的实现

由图 4-8 可以看出，PulseRain Reindeer 的地址空间主要被分为两部分：代码 / 数据内存和内存映射寄存器。 内存映射寄存器主要被用来作为外围设备寄存器的地址空间映射。理论上时钟定时器也是一种外围设备，然而，考虑到 RISC-V 标准对时钟定时器已经做了明确的定义，所以在 PulseRain Reindeer 中直接将其包含在了处理器核当中。

在介绍时钟定时器的 mtime 寄存器时曾经提到，时钟定时器应该运行在固定的计数频率，并建议处理器主频能被该计数频率所整除（3.6.3 节）。所以在 PulseRain Reindeer 中，将该计数频率设置为了 1 MHz，即时钟定时器的分辨率为 1μs。

4.4.6 流水线的设计

1. 取指器

PulseRain Reindeer 是一个 RV32IM 处理器。通过对外部大容量 DRAM 内存的支持，PulseRain Reindeer 避免了压缩指令集（C Extension）的实现。由于只需要支持 32 位的读取，取指器也不用考虑太多指令地址边界对齐的问题。

FPGA 片上内存一般只有一个时钟周期的读延迟，而外部内存的读延迟则往往要大得多。通过对指令读地址的判断，内存控制器可以很快地确定是否需要读延迟，从而设立相应的握手信号来反馈给取指器，以实现 FPGA 片上内存和片外内存的混合使用。

由图 3-5 可以看出，如果指令需要读取通用寄存器，则源寄存器 1 的地址总是在位 [19:15]，而源寄存器 2 的地址总是在位 [24:20]。考虑到对通用寄存器的读取不会有其他作用，不论指令的类型是什么，PulseRain Reindeer 都会以这两个位置上的数值为地址，对通用寄存器进行读取。

2. 指令译码器

从图 3-5 还可以看出，指令位 [6:0] 是操作码。而根据图 3-2，在 RV32IM 下，位 [1:0] 总是 3，所以在 PulseRain Reindeer 中只需要对位 [6:2] 译码便可确定指令操作类型，并产生相应的控制信号。这些控制信号会在接下来的指令执行器中被用到。

3. 指令执行器

指令执行器需要执行以下的几类指令：

- ALU（Arithmetic Logic Unit，算术逻辑单元）。

如图 4-10 所示，算术逻辑指令包括“加”“减”“移位”“与”“或”“异或”等。在参与算术逻辑的两个操作数中（图 4-10 中的寄存器 X 与 Y），操作数 X 总是来自于通用寄存器，而操作数 Y 则可以来自通用寄存器或者指令自带的立即数。对 ALU 的操作选择和数据源选择都来自于指令译码器产生的控制信号。

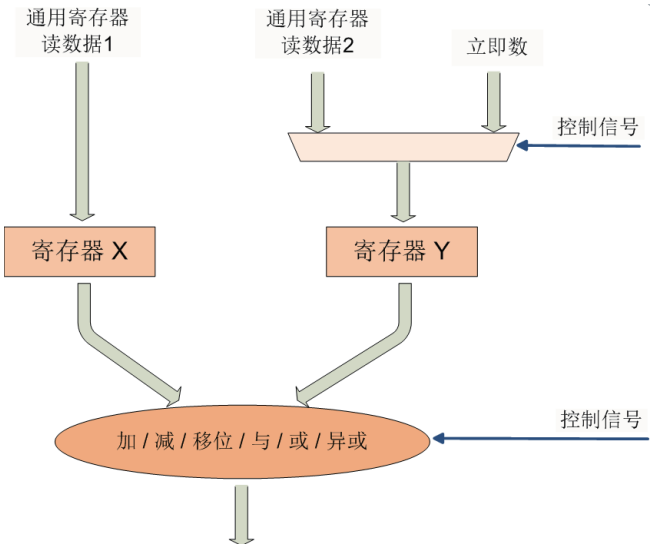


图4-10　算术逻辑单元

- 乘除法（M Extension）。

PulseRain Reindeer 支持 RV32IM 指令集。其中 M Extension（硬件乘除法）可以被选择性地配置。

- 无条件跳转指令（JAL / JALR）。

对于无条件跳转，其后一条指令的地址需要被存入目标寄存器中。

- LUI / AUIPC（Load Upper Immediate / Add Upper Immediate to PC）。

这两条立即地址构建指令（见图 3-10）的结果也会被写入目标寄存器。

以上这些指令都会更新目标寄存器，其具体的写入值如表 4-1 所示。

表 4-1　目标寄存器的写入值

指　令	目标寄存器写入值
算术逻辑操作	算术逻辑单元 (Arithmetic Logic Unit，ALU) 的输出
乘除法操作	乘除法结果
无条件跳转	PC + 4
LUI	高 20 位立即数左移 12 位
AUIPC	高 20 位立即数左移 12 位后加 PC

除了以上这些指令外，执行器还需要对下面的这些指令做出处理：

- CSR 操作指令。

指令执行器与 CSR 寄存器有专用总线相连，以做数据更新。

- BRANCH 指令，ECALL / EBREAK，LOAD / STORE。

这些指令无须更新目标寄存器，会产生相应的内部控制标记，供流水线控制器做参考。

4. 数据访问

在数据访问阶段，通用寄存器会被更新，由 LOAD / STORE 指令产生的内存访问也会在这个阶段产生。由于 2×2 的流水线布局，“内存访问”阶段和“取指”阶段被安排在了不同的时钟周期，以尽量降低内存访问冲突发生的可能性。

5. 流水线控制

流水线控制的主要目的就是对跳转指令和异常 / 中断的处理，如图 4-11 所示。因为流水线控制比较烦琐和复杂，所以图 4-11 只列出了其中的主要部分。

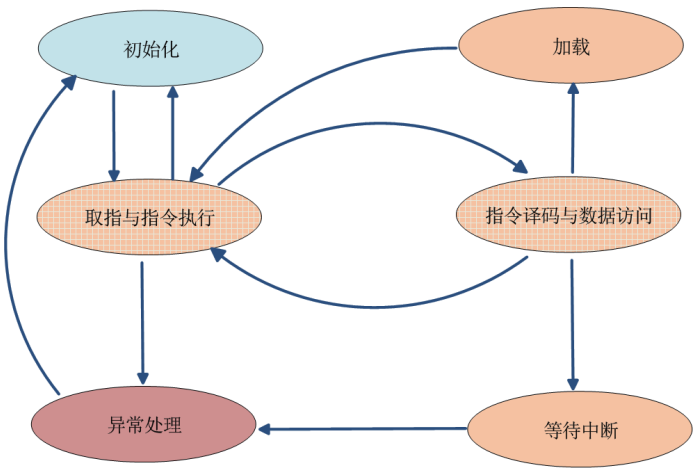


图4-11　流水线控制的主要状态

对于 2×2 的流水线布局，取指与指令执行状态和指令译码与数据访问状态分别对应于图 4-3 提到的双数时钟周期（“取指”与“指令执行”）和单数时钟周期

（“指令译码”与“数据访问”）。其中跳转指令则会将流水线控制转入初始化状态，以重新加载流水线。

而异常 / 中断的处理则需要一个额外的异常处理状态，以根据异常 / 中断的具体类别，设置异常编码（见表 3-18 与表 3-19），并确定异常 / 中断处理的返回地址（即 mepc 寄存器）。

4.5 处理器验证的方式

4.5.1 黑盒（Black Box）测试与白盒（White Box）测试

作为软件运行的最终平台，处理器的准确性是至关重要的。对此，RISC-V 官方在 GitHub 上公布了一套 RISC-V 指令集的标准测试程序，以作为处理器兼容性和正确性认证的标准。

在本书撰写之际，RISC-V 的合规测试在 RV32I 下共有 55 个标准测试程序，而在 M Extension 下则有 8 个测试程序。

对于 RISC-V 合规测试中包含的这些测试程序，其做法都是采用 Signature 检测验证法，即测试程序在运行过程中会向内存中写入某些标记。在程序运行结束后，再将内存中的这些数据读取出来，并与标准结果做比对。这个方法不对处理器运行的中间状态做监测，可以看作是一种黑盒验证法。

然而，在商用开发中，这种黑盒验证法还不充分。实际上，当黑盒测试失败时，确定出问题的具体位置是一件非常困难的事情，在许多商用处理器的开发中，往往采用白盒验证法。其具体做法如下：

（1）在处理器硬件开发之前，先用软件开发一个处理器的模拟器，用来确定处理器的行为模式。

（2）将测试程序作为这个模拟器的输入，在其上运行，并产生测试向量。测试向量包含处理器在每个时钟周期下应有的内部状态，如程序计数器（PC）

的值、取指器提供的指令、所有通用寄存器的值、内存读写的地址与数据等。

（3）将同样的程序作为处理器 RTL 仿真的输入，将仿真所得到的处理器内部状态与测试向量做比较。

（4）修改 RTL 设计，直至仿真通过所有的测试向量为止。

PulseRain Reindeer 对这两种验证方法都有采用。在后边会介绍用 Verilator 进行黑盒法验证，以及用 Modelsim 进行白盒法验证。

4.5.2 用 Verilator 做处理器内核的黑盒验证

Verilator 是一款非常出色的开源仿真软件。和其他商用仿真软件不同的是，Verilator 会将 Verilog 文件编译成 C++ 语言，然后再用 C++ 编译器编译并执行。与之相对应的是，Verilator 中的测试平台也可以用 C++ 编写。C++ 具有作为高级编程语言的强大功能，使得这种用 C++ 编写的测试平台可以直接与 RISC-V 的工具链相交互，对处理器的自动化仿真非常有帮助。

然而和商用仿真软件相比，Verilator 存在其固有的缺点。Verilator 缺点如下。

- 只支持 Verilog，对 System Verilog 的支持不完整。
- 对模块内部信号的检测还没有足够的支持。
- 以命令行为主，对波形的显示非常不方便。

由此，PulseRain Reindeer 仅将 Verilator 用作处理器内核的黑盒法验证，来运行回归测试。

Verilator 仿真如图 4-12 所示，对于 RISC-V 提供的测试程序，PulseRain Reindeer 的 Verilator 测试平台采用了和图 4-6 非常类似的结构，并且用 C++ 代替了图 4-6 中的 Python Script 和 Hardware Based Bootloader。这样，PulseRain Reindeer 的 Verilator 测试平台可以直接将测试程序（elf 文件）载入到内存中进行仿真，还可以在仿真结束后再读取内存并作 Signature 的比对。

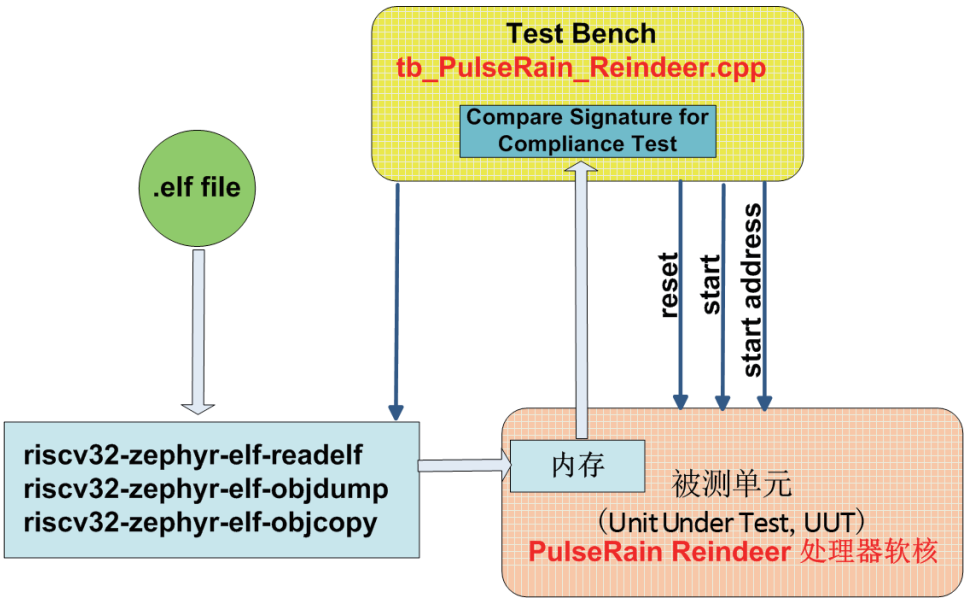


图4-12 Verilator仿真

在介绍小脚丫综合实验平台时，还会对 Verilator 的具体操作做进一步介绍。

4.5.3 用 Modelsim 做处理器的白盒验证

前边章节提到了如何用 Verilator 来对 RISC-V 处理器内核做验证检验。在 Verilator 进行的仿真中并不包含外部内存控制器，所有的代码都是通过测试平台写入到片上的 Block RAM 里面。这个做法虽然可以很快地对 RISC-V 处理器内核进行合规测试验证，但是存在以下问题：

（1）Verilator 不能很方便地检测模块内部的信号。所以验证的主要方法就是仿真运行合规测试的各个程序（elf 文件）。在仿真结束后，通过读取内存中的数据，并与标准的 Signature 做比较来进行判断。在 4.5.1 节中介绍白盒验证法时提到，更稳妥和精确的验证方法应该在每个时钟周期都对处理器的状态（PC 程序计数器、IR 指令寄存器，以及通用寄存器的值等）进行侦测，并与测试向量比较，以便及时发现并定位问题。

（2）上文提到的 Verilator 仿真只包含了 RISC-V 处理器内核部分，但是在实际的系统中，FPGA 除了包含处理器内核之外，还包含了由 FPGA 厂商提供的各类 IP。例如在小脚丫综合实验平台的 FPGA 里就包含了 PLL 和 SDRAM 控制器等。为了保证系统的正确运行，更好的做法应该是将所有这些 IP 也包含在内，从 FPGA 上电复位开始仿真。

因此，PulseRain Reindeer 在处理器级别（包括处理器内核、外设与 DRAM 控制器等）的验证采取了以下的白盒验证法。实际上，许多商用系统也采用了类似的验证方式：

（1）用 Modelsim 代替 Verilator，以方便对模块内部信号的检测，并且对所有的 IP 都建立仿真库。作为一款优秀的商用仿真软件，Modelsim 一直是主流 FPGA 厂商青睐的仿真软件。在 Intel Prime Quartus Lite Edition 中带有一个 Modelsim 初学者版本，可以被用来仿真本书所涉及的所有样例。

（2）为外部内存芯片也建立相对应的仿真模型。例如，小脚丫平台上采用的外部内存芯片是 SDRAM（IS45S16400J），在用 Intel Prime Quartus 产生 SDRAM 控制器时，软件也会提供一个相对应的仿真模型。用户还可以为这个仿真模型提供一个 dat 文件，作为 SDRAM 的内存初始值。在验证检验仿真时，这个 dat 文件正好可以被用来存放验证检验的程序代码（见图 4-13）。

（3）将合规测试的各个程序在 RISC-V 的模拟器上运行，产生测试向量。在本书所采用的测试向量中，测试向量的每一行包括了 PC 值（程序计数器）、IR 值（指令）、各通用寄存器的值等。为简化起见，本书提供的测试平台将只比较测试向量中的前两列（PC 值与 IR 值）。

本书使用的测试向量，来自于 PulseRain Technology 公司内部开发的 RISC-V 模拟器。类似的 RISC-V 模拟器在 GitHub 上有很多，读者也可以将这些模拟器稍做修改后产生自己所需要的测试向量。

（4）使用测试平台将 UUT（Unit Under Test，在这里即为 PulseRain Reindeer Step RISC-V 微控制器）、SDRAM Simulation Model 整合在一起，如图 4-13 所示。

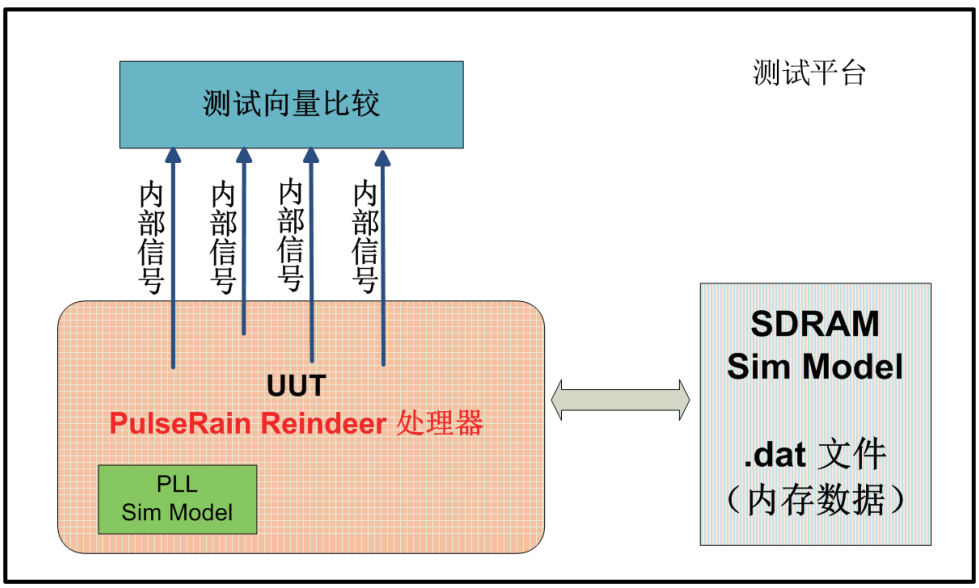


图4-13　Modelsim 仿真

（5）运行 Modelsim 仿真，并提取 UUT 内部信号与测试向量做比较。

在介绍小脚丫综合实验平台时，还会对相关的具体操作做进一步讨论。