一起学RISC-V汇编第9讲之RISC-V ABI之寄存器使用约定

目录

寄存器使用约定告诉我们函数调用时通过哪些寄存器传递参数、通过哪些寄存器保存返回值、哪些寄存器可以任意使用而不用保存旧值等。

1 RISC-V 寄存器使用约定

第2讲寄存器这一章，列出了32个通用寄存器以及32个浮点寄存器：

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 25.1: Assembler mnemonics for RISC-V integer and floating-point registers, and their role in the first standard calling convention.

整理如下：

Preserved（callee-saved）	**NonPreserved（caller-saved）**
Saved registers：`s0–s11`	Return address: `ra`
Stack pointer: `sp`	Argument registers: `a0–a7`
	Return values: `a0–a1`

Preserved（callee-saved）	NonPreserved（caller-saved）
	Temporary registers: `t0-t6`

2 Caller-saved 与 Callee-saved

由RISC-V寄存器的个数是有限的，而函数是非常多的，调用路径可能非常长，这么多函数共用有限的寄存器，怎么样才能安全的访问寄存器呢？最安全的做法是，每次调用其它的函数前把寄存器值保存到栈中，等从子函数返回后，再将寄存器的值出栈，恢复函数调用前的状态，通过这个办法，各个函数就都可以随意使用所有寄存器了。

我们将调用函数称为caller，被调用函数称为callee。

2.1 对比几种不同的寄存器保存方式

对比如下几种方式：

- 方式1：寄存器全部由caller 保存
- 方式2：寄存器全部由callee保存
- 方式3：按照程序调用约定，一部分寄存器由调用者保存（caller-saved）， 一部分由被调用者保存（callee-saved）

方式1 寄存器全部由caller保存-可行但效率很低

这种做法是最安全的，但效率非常低，访问栈相当于访问内存，比直接访问寄存器速度明显下降，所以有必要想办法减少栈的使用。

方式2 寄存器全部由callee保存-不可行

寄存器全部由callee保存行吗？答案是不可行，有些寄存器我们必须在改变之前保存（即caller中保存）。

参数寄存器（`a0-a7`）只能是 caller-saved 。为了传递参数，caller 必须将参数寄存器中原本的值覆盖成传给 callee 的参数，如果参数寄存器内之前有值，那么就需要 caller 进行保存。这个工作没法由 callee 进行，因为提交给 callee 的参数寄存器已经被污染（改变）了。

返回值寄存器(`a0-a1`)只能是 caller-saved 。如果让 callee 保存 caller 的返回值，那么 callee 的自己的返回值该怎么传递给 caller 呢?

返回地址寄存器(`ra`）只能是 caller-saved，如果 callee 在进入函数时保存了 caller 的 `ra` ，而退出函数前恢复了 caller 的 `ra` ,那么返回到一个错误的地方，应该保存自己的 `ra` ，而非caller的 `ra`

方式3 一部分寄存器由调用者保存（caller-saved）， 一部分由被调用者保存（callee-saved）

我们将寄存器分为两组：

- caller-saved （调用者保存），这是说调用者可以按照自己的需要来保存这些寄存器，如果自己使用了，那么就保存，如果没有使用，就不需要保存。
- callee-saved （被调用者保存），被调用者在执行正式的功能前前需要将这些寄存器压栈，在返回前需要将这些寄存器弹栈恢复，从caller的视角看，这些寄存器在调用前后值肯定是一样的。从而叫做保存寄存器。

下面一段话引用自计算机组成-寄存器保存

> 不过这种标准的定义其实并不容易让人理解其中的原因。我们可以换一个角度去看这件事。
>
> callee-saved 寄存器都会本着"谁污染，谁治理"的策略被保护起来，所以callee-saved register是 **safe** 的，也就是在call后寄存器的值不会改变。。

而 caller-saved 的寄存器有可能被污染（在 caller 并不保存的情况下），其实核心特性是 **unsafe** ，也就是"调用前后并能保证寄存器中的值不发生变化"，所以这种寄存器也被称为临时寄存器，如果有需求才会由caller保存。

2.2 为什么要分caller-saved与callee-saved?

原因归根到底是因为编译器缺少全局优化的能力。

如果在全局视角下分析，在调用过程中，我们真正需要保存的寄存器，具有两个特性：

1. 被 caller 使用且生命周期必须跨越调用过程。
2. 被 callee 使用。

那无论是 callee 还是 caller 都没有办法很好达到这个理想状态，因为作为 caller 虽然可以正确的分析出具有第 1 点特性的寄存器，但是对于第 2 点无能为力。同样的 callee 可以清楚地知道第 2 点，但是对于第 1 点一无所知。这种困境是由于"函数"这个概念本就是为了代码段间的解耦和隔离，如果想要一个个解耦的函数，就必须承担无法全局优化的代价。

如果这样看，那么将寄存器分成差不多大小的两组的行为是有一定道理的，我们虽然不能做到全局优化，但是可以做到基于 caller 和 callee 各自一定的权力，使得双方的关于寄存器使用的"洞见"都可以得到利用。总好过只利用一方的信息。

2.3 caller-saved与callee-saved寄存器的灵活使用

从使用上来说，如果寄存器中是一个生命周期很长的变量，那么应当使用 callee-saved register ，它可以保证在其生命周期的多次函数调用中，值始终不发生变化；而如果寄存器中是一个用完就扔的临时变量，那么应当使用 caller-saved-register ，使用它并不需要保存，心理负担小。

caller-saved的使用场景：

1. **临时数据的存放**：在进行一系列计算时，编译器可能会选择使用caller saved registers来存储临时结果，因为这些结果在当前函数调用结束后不再需要。
2. **非关键性数据的处理**：对于那些不会被随后的函数调用所需的数据，使用caller saved registers可以在调用后立即丢弃。

callee-saved的使用场景：

1. **重要数据的保护**：当函数需要调用另一个函数时，它依赖callee saved registers来保护那些必须跨函数调用保存的关键数据。
2. **保持状态的连续性**：对于递归调用和多层嵌套调用，callee saved registers可以帮助保持调用链上每个函数的状态。

参考：

1. Releases · riscv-non-isa/riscv-elf-psabi-doc (github.com)
2. RISC-V函数调用规范 - 知乎 (zhihu.com)
3. 为什么CPU寄存器要分为两组，被调用者保存寄存器和调用者保存寄存器?
4. (15 封私信 / 80 条消息) risc-v 返回地址寄存器是由caller保存还是callee保存? - 知乎 (zhihu.com)
5. Caller-saved register and Callee-saved register-CSDN博客
6. 计算机组成-寄存器保存 | 钟鼓楼 (thysrael.github.io)
7. RISC-V基础之函数调用（三）保留寄存器（包含实例）_riscv中如何保存和恢复更新寄存器的值-CSDN博客
8. 为什么要区分caller saved和callee saved registers – PingCode