一起学RISC-V汇编第9讲之RISC-V ABI之栈帧

这一节讲解RISC-V中的栈帧。

1 C语言中的{}的秘密

函数执行的底层其实是操作寄存器，CPU的寄存器是有限的，为什么我们进行一系列函数调用后还能正确运行，这些函数之间是怎么协调使用寄存器的?

答案是：**栈**

函数之间能随意调用，还能顺利恢复现场，这个就是栈的功劳。为什么我们在代码中并没有操作栈呢？其实这一切都是编译器帮我们完成的，为程序员屏蔽了比较复杂的栈操作，比如C语言函数中的大括号 `{}` ，我们可以粗暴的认为： `{` 是入栈操作， `}` 是出栈操作，这样为程序员屏蔽了比较复杂的栈操作。

如下伪代码表示A函数调用B函数，B函数调用C函数。

```
function C()
{
    xxxxx;
}

function B()
{
    C();
}

function A()
{
    B();
}
```

其栈帧结构下图，下图来自于《循序渐进，学习开发一个 RISC-V 上的操作系统》相关章节。其过程如下：

调用函数A时，创建函数A的栈帧

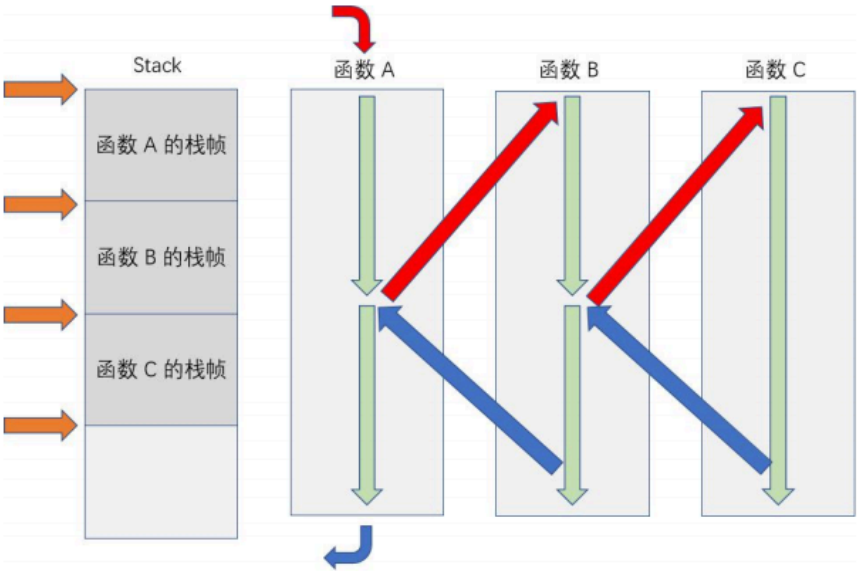
函数A调用函数B时，创建函数B的栈帧

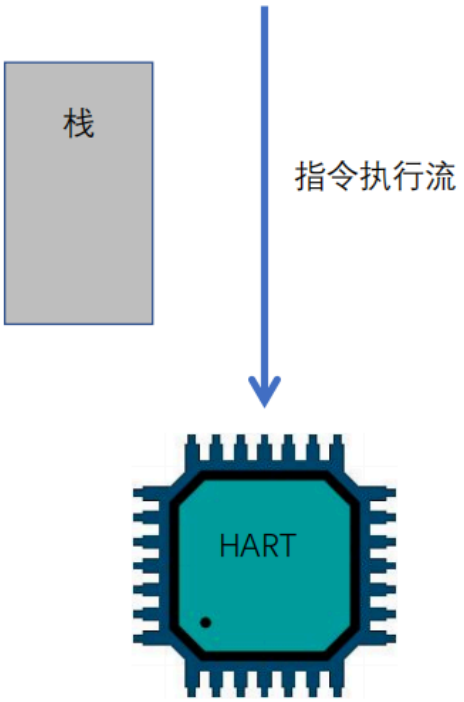
函数B调用函数C时，创建函数C的栈帧

从函数C返回B时，函数C的栈帧就注销了

从函数B返回A时，函数B的栈帧就注销了

函数A执行完成后，其栈帧也注销了





上述仅粗略的描述了栈帧的创建与注销，下面以一个实例来展示过程。

2 RISC-V函数栈帧

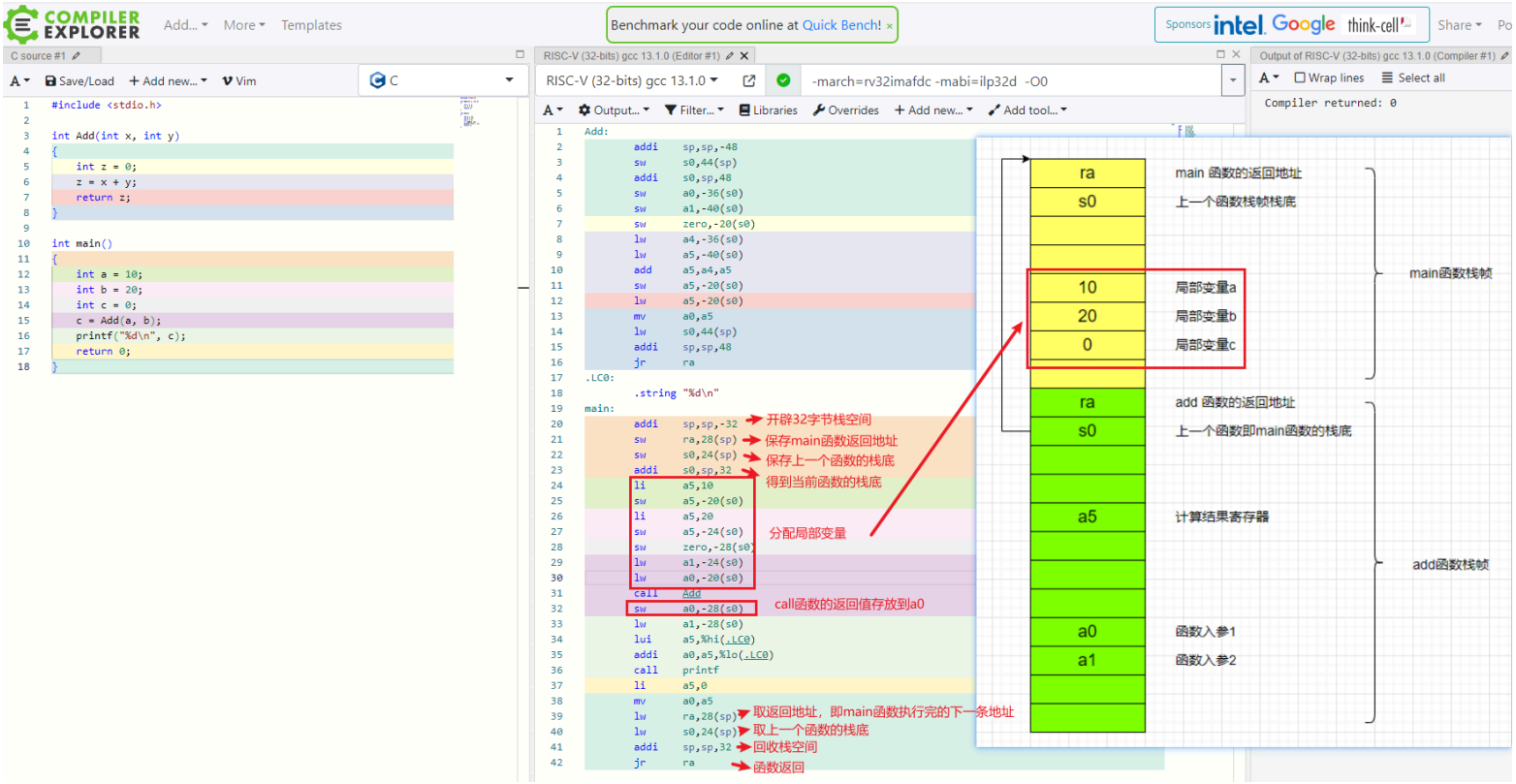
以一个示例展示栈帧。

```c
#include <stdio.h>

int Add(int x, int y)
{
        int z = 0;
        z = x + y;
        return z;
}

int main()
{
        int a = 10;
        int b = 20;
        int c = 0;
        c = Add(a, b);
        printf("%d\n", c);
        return 0;
}
```

其栈帧示意图如下：



不使用栈基址寄存器：

从上述执行过程也可以看出，其实不使用栈基址寄存器（s0/fp），仅使用栈指针寄存器(sp)，函数也可以正常运行

gcc中可以使用如下方法开启关闭帧指针。

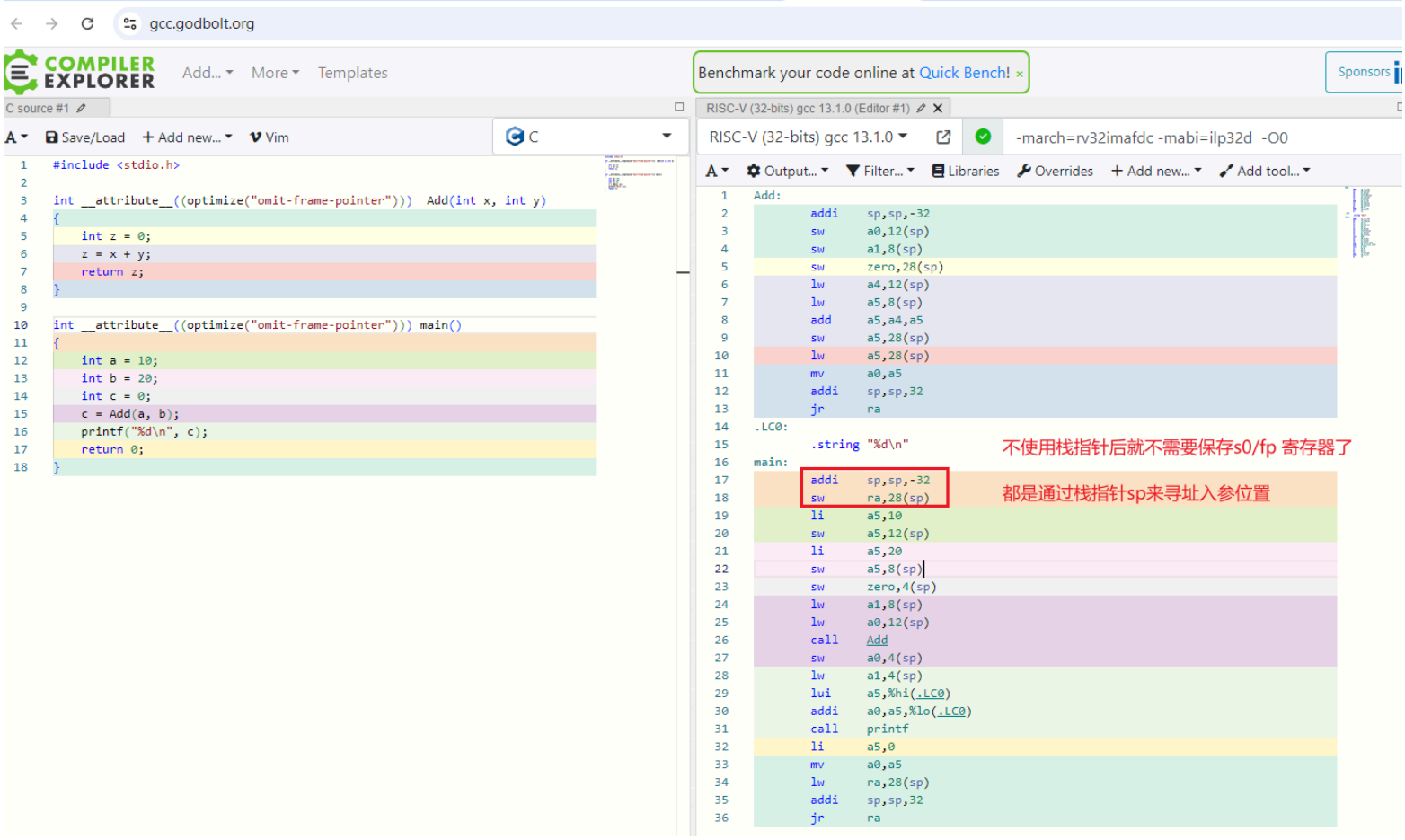
- __attribute__((optimize("no-omit-frame-pointer"))) 修饰函数，开启帧指针
- __attribute__((optimize("omit-frame-pointer"))) 修饰函数，关闭帧指针

在 RISC-V 架构中使用 `s0` / `fp` 来指向当前函数调用的栈帧顶部的寄存器。它主要用于：

1. **栈帧导航**：帮助确定当前栈帧的位置。
2. **异常处理**：在某些情况下，用于异常处理和堆栈回溯。

省略帧指针可以减少每次函数调用时所需的寄存器操作次数，从而提高程序的执行速度。但是，这样做也有一定的缺点，例如：

- **调试困难**：没有帧指针，调试时难以准确回溯堆栈。
- **异常处理**：某些依赖于准确堆栈信息的异常处理机制可能无法正常工作。



参考：

1. Releases · riscv-non-isa/riscv-elf-psabi-doc (github.com)
2. RISC-V架构的函数调用规范和栈布局_riscv函数调用规范-CSDN博客