

# 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

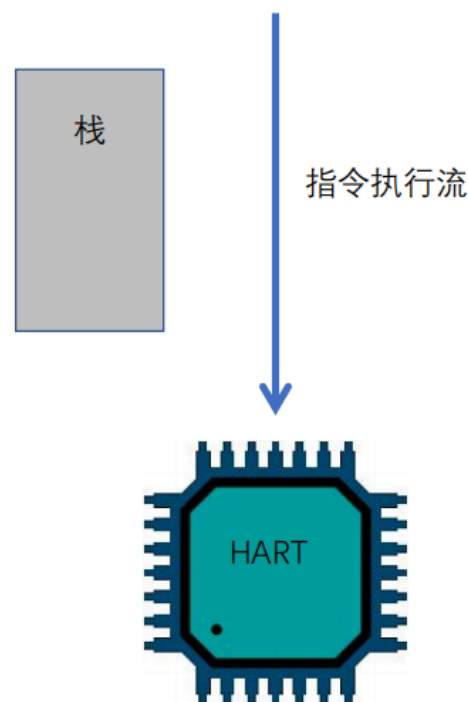
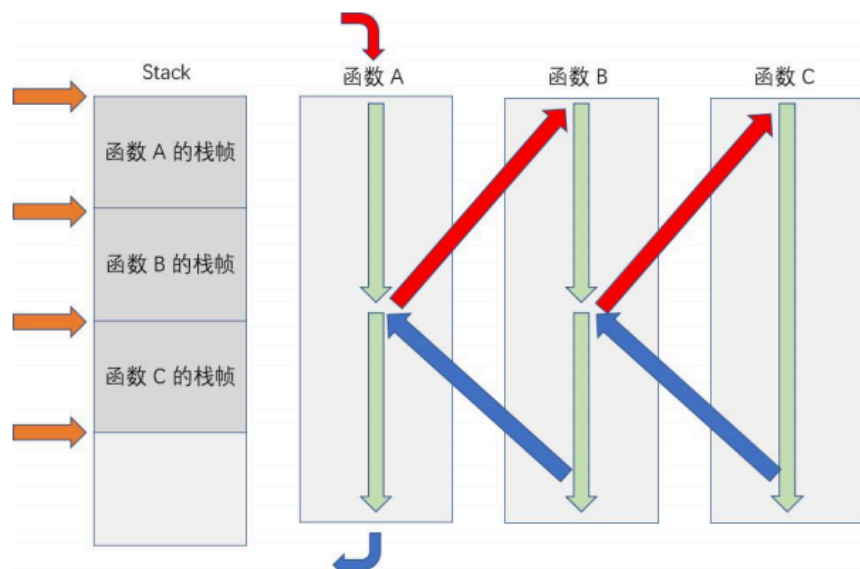
函数A调用函数B时，创建函数B的栈帧

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。

```
#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：

COMPILER EXPLORER

Add... More Templates

Benchmark your code online at Quick Bench! x

SponsorsintelGooglethink-cell

Share Po

C source #1

Save/Load

+ Add new...

Vim

C

```
1 #include <stdio.h>
2
3 int Add(int x, int y)
4 {
5     int z = 0;
6     z = x + y;
7     return z;
8 }
9
10 int main()
11 {
12     int a = 10;
13     int b = 20;
14     int c = 0;
15     c = Add(a, b);
16     printf("%d\n", c);
17     return 0;
18 }
```

RISC-V (32-bits) gcc 13.1.0 (Editor #1)

RISC-V (32-bits) gcc 13.1.0

-march=rv32imafdc -mabi=ilp32d -O0

Output...

Filter...

Libraries

Overrides

+ Add new...

+ Add tool...

Compiler returned: 0

1 Add:
2 addi sp,sp,-48
3 sw s0,44(sp)
4 addi s0,sp,48
5 sw a0,-36(s0)
6 sw a1,-40(s0)
7 sw zero,-20(s0)
8 lw a4,-36(s0)
9 lw a5,-40(s0)
10 add a5,a4,a5
11 sw a5,-20(s0)
12 lw a5,-20(s0)
13 mv a0,a5
14 lw s0,44(sp)
15 addi sp,sp,48
16 jr ra
17
18 .LC0:
19 .string "%d\n"
20
21 main:
22 addi sp,sp,-32
23 sw ra,28(sp)
24 sw s0,24(sp)
25 addi s0,sp,32
26 li a5,10
27 sw a5,-20(s0)
28 li a5,20
29 sw a5,-24(s0)
30 sw zero,-28(s0)
31 lw a1,-24(s0)
32 lw a0,-20(s0)
33 call Add
34 sw a0,-28(s0)
35 lw a1,-28(s0)
36 lui a5,%hi(.LC0)
37 addi a0,a5,%lo(.LC0)
38 call printf
39 li a5,0
40 mv a0,a5
41 lw ra,28(sp)
42 lw s0,24(sp)
43 addi sp,sp,32
44 jr ra

main 函数的返回地址

上一个函数栈帧底

10 局部变量a

20 局部变量b

0 局部变量c

add 函数的返回地址

上一个函数即main函数的栈底

a5 计算结果寄存器

a0 函数入参1

a1 函数入参2

开辟32字节栈空间

保存main函数返回地址

保存上一个函数的栈底

得到当前函数的栈底

分配局部变量

call函数的返回值存放到a0

取返回地址,即main函数执行完的下一条地址

取上一个函数的栈底

回收栈空间

函数返回

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。

gcc.godbolt.org

COMPILER EXPLORER

Add... More Templates

C source #1

A Save/Load + Add new... Vim

C

```
1 #include <stdio.h>
2
3 int __attribute__((optimize("omit-frame-pointer"))) Add(int x, int y)
4 {
5     int z = 0;
6     z = x + y;
7     return z;
8 }
9
10 int __attribute__((optimize("omit-frame-pointer"))) main()
11 {
12     int a = 10;
13     int b = 20;
14     int c = 0;
15     c = Add(a, b);
16     printf("%d\n", c);
17     return 0;
18 }
```

Benchmark your code online at [Quick Bench!](#)

Sponsors

RISC-V (32-bits) gcc 13.1.0 (Editor #1)

RISC-V (32-bits) gcc 13.1.0 -march=rv32imafdc -mabi=ilp32d -O0

A Output... Filter... Libraries Overrides + Add new... Add tool...

```
1 Add:
2     addi sp,sp,-32
3     sw a0,12(sp)
4     sw a1,8(sp)
5     sw zero,28(sp)
6     lw a4,12(sp)
7     lw a5,8(sp)
8     add a5,a4,a5
9     sw a5,28(sp)
10    lw a5,28(sp)
11    mv a0,a5
12    addi sp,sp,32
13    jr ra
14
15 .LC0:
16     .string "%d\n"
17
18 main:
19     addi sp,sp,-32
20     sw ra,28(sp)
21     li a5,10
22     sw a5,12(sp)
23     li a5,20
24     sw a5,8(sp)
25     sw zero,4(sp)
26     lw a1,8(sp)
27     lw a0,12(sp)
28     call Add
29     sw a0,4(sp)
30     lw a1,4(sp)
31     lui a5,%hi(.LC0)
32     addi a0,a5,%lo(.LC0)
33     call printf
34     li a5,0
35     mv a0,a5
36     lw ra,28(sp)
37     addi sp,sp,32
38     jr ra
```

不使用栈指针后就不需要保存s0/fp 寄存器了  
都是通过栈指针sp来寻址入参位置

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](#)