一起学RISC-V汇编第9讲之RISC-V ABI之函数调用

目录

1 RISC-V ABI接口

ABI（Application Binary Interface）为应用程序二进制接口，它定义了应用程序之间或应用程序和操作系统之间进行二进制级交互时必须遵循的规则和约定。ABI包括了关于函数调用约定（参数传递，函数返回值等）、数据类型、对齐方式、字节序、 函数栈布局、系统调用等方面的规范。

具体来说，ABI定义了以下内容：

1. 数据类型、对齐方式和字节序；
2. 寄存器使用约定：哪些寄存器用于传递函数参数、返回值和保存临时变量；
3. 函数调用约定：函数调用的具体步骤，包括参数传递、返回值处理等；
4. 栈的使用：栈的增长方向、如何保存和恢复栈指针等；
5. 系统调用：操作系统提供的服务调用方式和参数传递规则；

ABI的存在使得不同的编程语言、编译器和操作系统之间能够进行二进制级的互操作，确保它们能够正确地调用和使用彼此所生成的代码。由于不同的体系结构和操作系统可能有不同的ABI，因此跨平台开发时需要考虑ABI的兼容性。

这里只讲函数调用约定、寄存器使用约定，栈帧这几部分内容，内容放在一起比较长，所以文章分三部分。

2 RISC-V 函数调用约定

函数调用约定主要是约定怎么传递函数参数、怎么返回值。

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.

一般规则如下：

1. **入参的传递：**函数调用时，优先使用 a0-a7 这8个标量寄存器（对于浮点类型是fa0-fa7这8个浮点寄存器）来传递入参
2. **返回值的传递：**一般用a0-a1来传递（浮点数用fa0-fa1来传递）返回值

下面讲述更详细的规则。

2.1 入参的传递

但是如果入参超过8个，只用寄存器不够该怎么办？如果函数入参是结构体类型该如何处理呢？如果入参有标量又有浮点数该如何处理呢？

有如下几种情况：

- 当一个标量位宽不超过XLEN位或者一个浮点实数参数不超过FLEN位时，使用单个参数寄存器传递。若没有可用的参数寄存器，则在栈上传递
- 当一个标量位宽超过XLEN位但是不超过2×XLEN时，则可以在一对参数寄存器中传递，低XLEN位在小编号寄存器中，高XLEN位在大编号寄存器中；若没有可用的参数寄存器，则在栈上传递标量；若只有一个寄存器可用，则低XLEN位在寄存器中传递，高XLEN位在栈上传递
- 若一个标量宽度大于2×XLEN，则通过引用传递（栈传递），并在参数列表中用地址替换。通过引用传递的实参可以由被调用方修改

常见的有如下几个场景：

几种场景	参数列表	参数传递
情形1	n1,n2,n3	a0,a1,a2
情形2	n1,n2,n3,n4,n5,n6,n7,n8,n9,n10	a0,a1,a2,a3,a4,a5,a6,a7,a8,stack
情形3	l1, l2	a0,a1,a2,a3
情形4	s1,s2,s3 d1,d2,d3 s1,d1,d2	fa0,fa1,fa2
情形5	s1,s2,s3,s4,s5,s6,s7,s8,s9,s10	fa0,fa1,fa2,fa3,fa4,fa5,fa6,fa7,a0,a1
情形6	n1,n2,n3,s1,d1	a0,a1,a2,fa0,fa1

表中：n1,n2... 表示标量位宽不超过XLEN的整数，l1,l2...表示标量位宽超过XLEN，不超过2×XLEN的整数，s1,s2...表示单精度浮点，d1,d2...表示双精度浮点。

举例如下：

情形1：标量位宽不超过XLEN，且函数入参个数小于8，使用寄存器传递

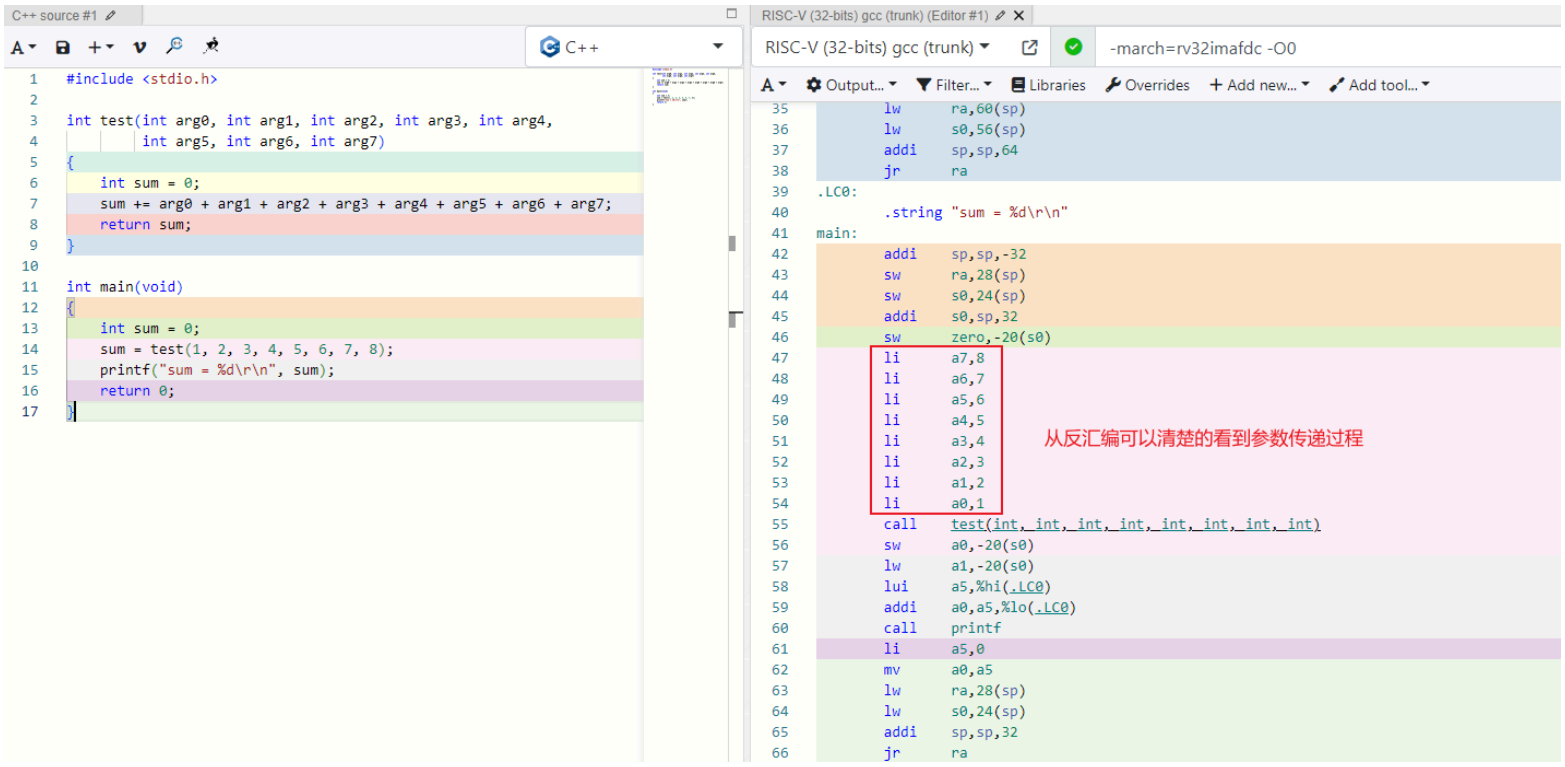
源码：

```
#include <stdio.h>

int test(int arg0, int arg1, int arg2, int arg3, int arg4,
         int arg5, int arg6, int arg7)
{
    int sum = 0;
    sum += arg0 + arg1 + arg2 + arg3 + arg4 + arg5 + arg6 + arg7;
    return sum;
}

int main(void)
{
    int sum = 0;
    sum = test(1, 2, 3, 4, 5, 6, 7, 8);
    printf("sum = %d\r\n", sum);
    return 0;
}
```

例子中，标量位宽为32bit，XLEN=32bit（-march=rv32imafdc），从反汇编可以看出，入参使用的是a0-a7寄存器传递，返回值使用的是a0



情形2：标量位宽不超过XLEN，且函数入参个数大于8，前8个参数使用寄存器传递，剩下的使用栈stack传递

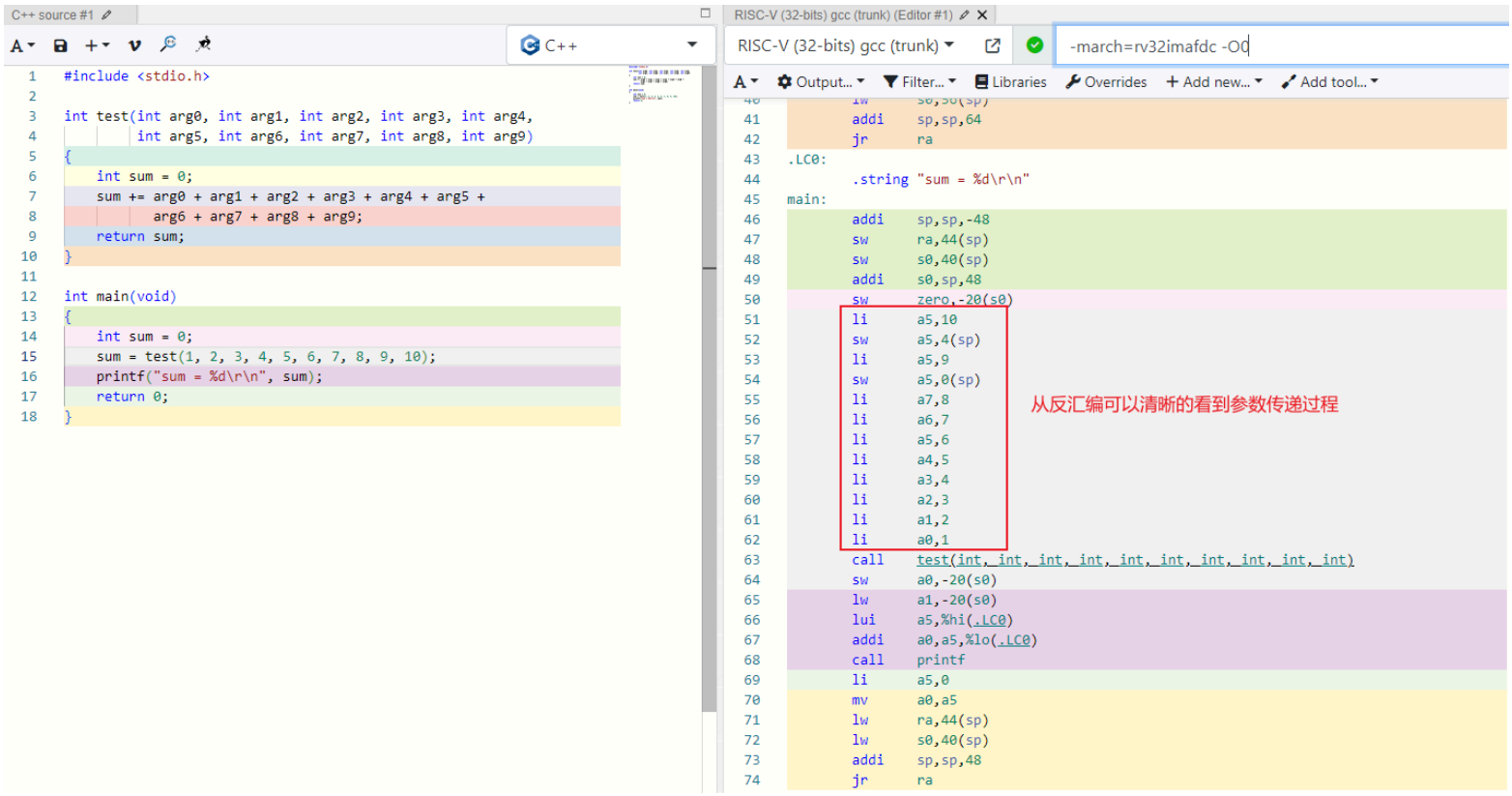
源码：

```c
#include <stdio.h>

int test(int arg0, int arg1, int arg2, int arg3, int arg4,
         int arg5, int arg6, int arg7, int arg8, int arg9)
{
    int sum = 0;
    sum += arg0 + arg1 + arg2 + arg3 + arg4 + arg5 +
           arg6 + arg7 + arg8 + arg9;
    return sum;
}

int main(void)
{
    int sum = 0;
    sum = test(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    printf("sum = %d\r\n", sum);
    return 0;
}
```

例子中，标量位宽为32bit，XLEN=32bit（-march=rv32imafdc），从反汇编可以看出，前8个入参使用的是a0-a7寄存器传递，后面两个入参使用的是栈传递（caller 将多的数据存到栈中，callee在栈对应位置取数），返回值使用的是a0




```
#include <stdio.h>

int test(int arg0, int arg1, int arg2, int arg3, int arg4,
         int arg5, int arg6, int arg7, int arg8, int arg9)
{
    int sum = 0;
    sum += arg0 + arg1 + arg2 + arg3 + arg4 + arg5 +
           arg6 + arg7 + arg8 + arg9;
    return sum;
}

int main(void)
{
    int sum = 0;
    sum = test(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    printf("sum = %d\r\n", sum);
    return 0;
}
```

```
        addi    sp,sp,64
        jr      ra
.LC0:
        .string "sum = %d\r\n"
main:
        addi    sp,sp,-48
        sw      ra,44(sp)
        sw      s0,40(sp)
        addi    s0,sp,48
        sw      zero,-20(s0)
        li      a5,10
        sw      a5,4(sp)
        li      a5,9
        sw      a5,0(sp)
        li      a7,8
        li      a6,7
        li      a5,6
        li      a4,5
        li      a3,4
        li      a2,3
        li      a1,2
        li      a0,1
        call    test(int, int, int, int, int, int, int, int, int, int)
        sw      a0,-20(s0)
        lw      a1,-20(s0)
        lui     a5,%hi(.LC0)
        addi    a0,a5,%lo(.LC0)
        call    printf
        li      a5,0
        mv      a0,a5
        lw      ra,44(sp)
        lw      s0,40(sp)
        addi    sp,sp,48
        jr      ra
```

情形3：标量位宽超过XLEN，但是不超过2×XLEN时，则可以使用一对寄存器来保存一个入参

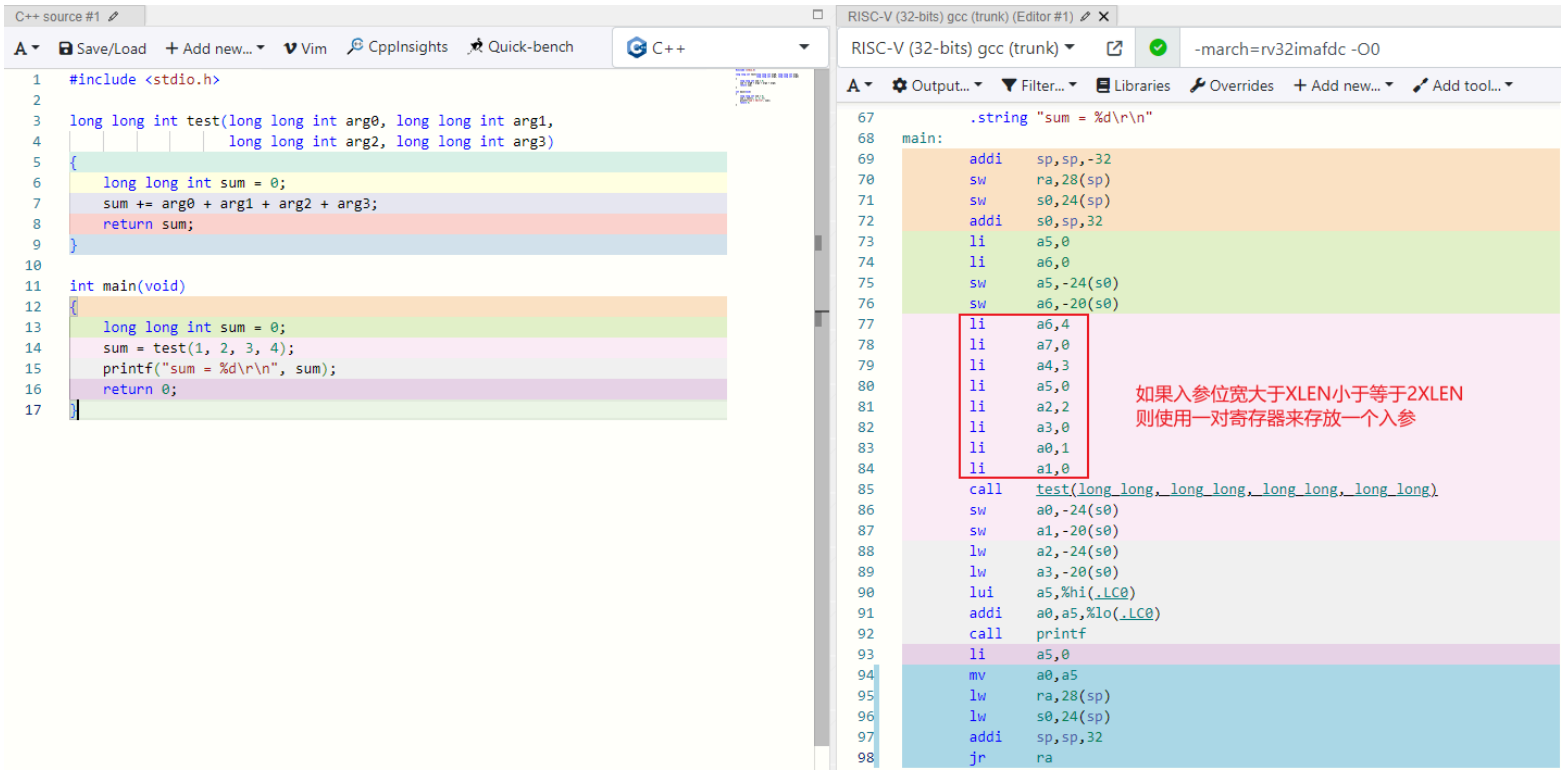
源码：

```
#include <stdio.h>

long long int test(long long int arg0, long long int arg1,
                   long long int arg2, long long int arg3)
{
    long long int sum = 0;
    sum += arg0 + arg1 + arg2 + arg3;
    return sum;
}

int main(void)
{
    long long int sum = 0;
    sum = test(1, 2, 3, 4);
    printf("sum = %ld\r\n", sum);
    return 0;
}
```

例子中，标量位宽为64bit，XLEN=32bit（-march=rv32imafdc），从反汇编可以看出，使用2个标量寄存器保存一个入参，且低XLEN位在小编号寄存器中，高XLEN位在大编号寄存器中，若入参没有足够可用的参数寄存器，则在栈上传递标量；若只有一个寄存器可用，则低XLEN位在寄存器中传递，高XLEN位在栈上传递。



情形4：如果数据类型是浮点，位宽不超过FLEN，且函数入参个数小于8，使用寄存器传递

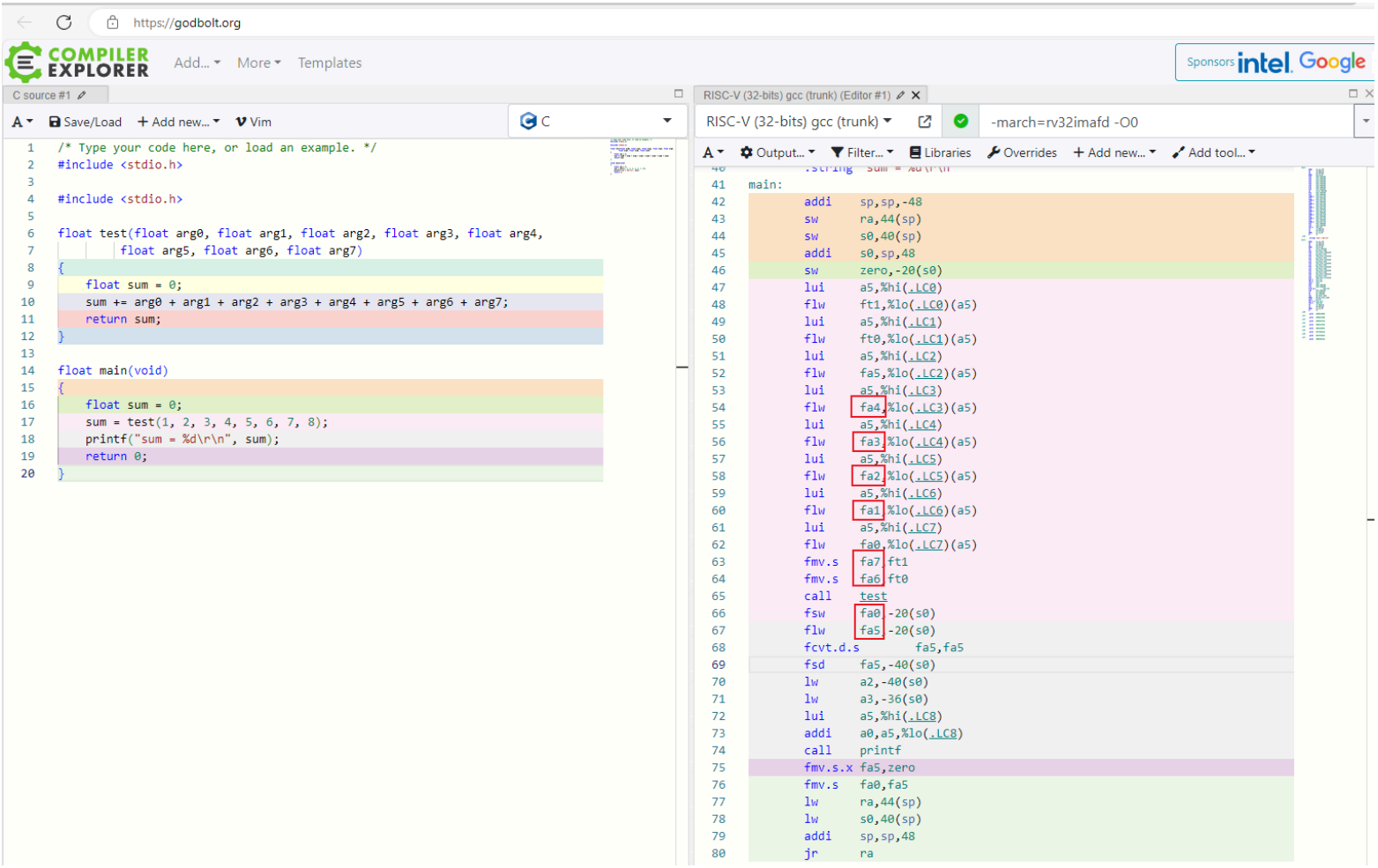
源码：

```
#include <stdio.h>

float test(float arg0, float arg1, float arg2, float arg3, float arg4,
          float arg5, float arg6, float arg7)
{
    float sum = 0;
    sum += arg0 + arg1 + arg2 + arg3 + arg4 + arg5 + arg6 + arg7;
    return sum;
}

float main(void)
{
    float sum = 0;
    sum = test(1, 2, 3, 4, 5, 6, 7, 8);
    printf("sum = %d\r\n", sum);
    return 0;
}
```

例子中，标量位宽为32bit，FLEN=64bit（-march=rv32imafdc），从反汇编可以看出，入参使用的是fa0-fa7寄存器传递，返回值使用的是fa0



情形5：如果数据类型是浮点，位宽不超过FLEN，且函数入参个数大于8，如果标量寄存器有剩余，则可以使用标量寄存器来传递浮点参数

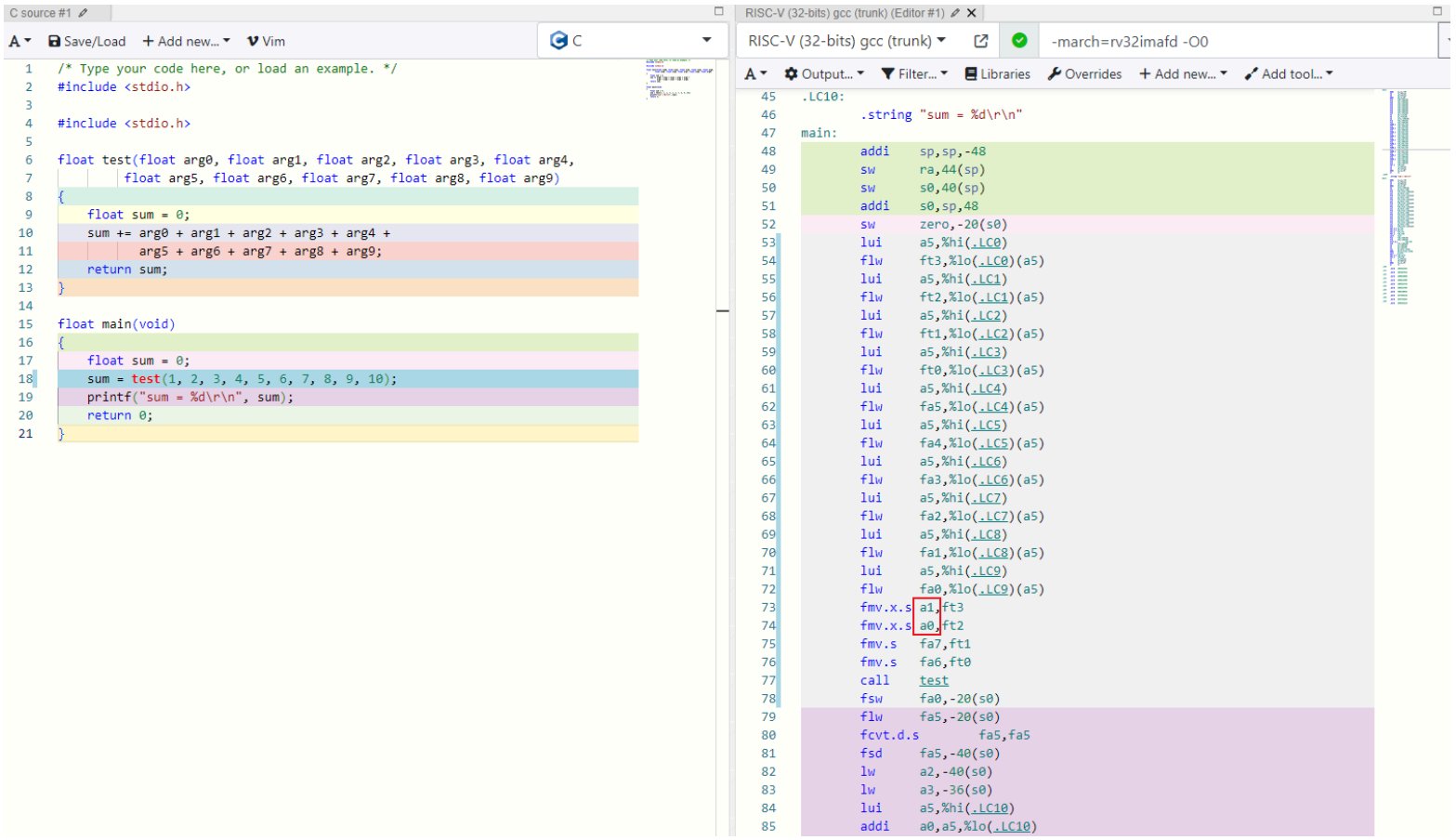
源码：

```
#include <stdio.h>

float test(float arg0, float arg1, float arg2, float arg3, float arg4,
         float arg5, float arg6, float arg7, float arg8, float arg9)
{
    float sum = 0;
    sum += arg0 + arg1 + arg2 + arg3 + arg4 +
           arg5 + arg6 + arg7 + arg8 + arg9;
    return sum;
}

float main(void)
{
    float sum = 0;
    sum = test(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    printf("sum = %d\r\n", sum);
    return 0;
}
```

例子中，FLEN=64bit（-march=rv32imafdc），从反汇编可以看出，前8个入参使用的是fa0-fa7寄存器传递，后面两个入参使用的标量寄存器a0, a1传递



情形6：入参包括整数，单精度浮点，双精度浮点

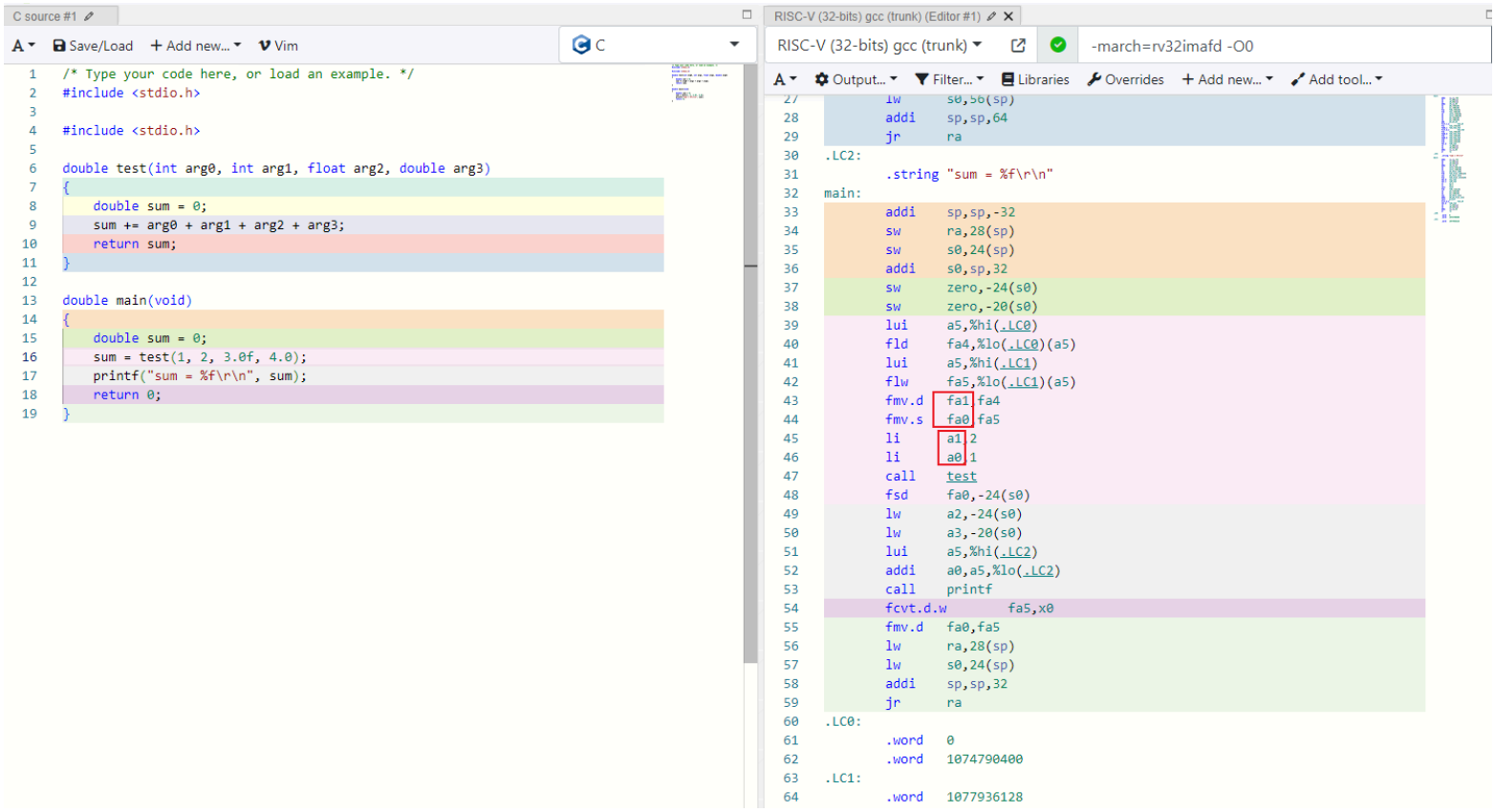
源码：

```c
#include <stdio.h>

double test(int arg0, int arg1, float arg2, double arg3)
{
    double sum = 0;
    sum += arg0 + arg1 + arg2 + arg3;
    return sum;
}

double main(void)
{
    double sum = 0;
    sum = test(1, 2, 3.0f, 4.0);
    printf("sum = %f\r\n", sum);
    return 0;
}
```

例子中，XLEN=32bit，FLEN=64bit（-march=rv32imafdc），从反汇编可以看出，标量用a0-a7传递，浮点数用fa0-fa7传递。



情形7：结构体作为入参

又分如下几种情况：

1. 若结构体的宽度不超过XLEN位，则这个结构体可以在寄存器中传递，若没有可用的寄存器，则在栈上传递。
2. 若结构体的宽度超过XLEN位，不超过2×XLEN位，则可以在一对寄存器中传递。若只有一个寄存器可用，则结构体的前半部分在寄存器中传递，后半部分在栈上传递；若没有可用的寄存器，则在栈上传递结构体。
3. 若一个结构体的宽度大于2×XLEN位，则通过引用传递，并在参数列表中被替换为地址。传递到栈上的结构体会对齐到类型对齐和XLEN中的较大者，但不会超过栈对齐要求。

2.2 返回值的传递

几种场景	返回值类型	返回值存放
情形1	void	当函数没有返回值时，不需要考虑返回寄存器的处理
情形2	标量（整形或指针类型）位宽不超过XLEN	a0
情形3	标量（整形或指针类型）位宽超过XLEN，不超过2xXLEN	a0,a1
情形4	单精度浮点（float）或 双精度浮点（double）	fa0
情形5	struct	需要细分，如果寄存器能放下使用寄存器，否则用栈

情形1：当函数没有返回值时，不需要考虑返回寄存器的处理

情形2：当函数返回类型是标量（整形或指针类型）位宽不超过XLEN，返回值存放在整型寄存器a0上

情形3：当函数返回类型是标量（整形或指针类型）位宽超过XLEN，不超过2xXLEN，返回值存放在整型寄存器a0，a1上

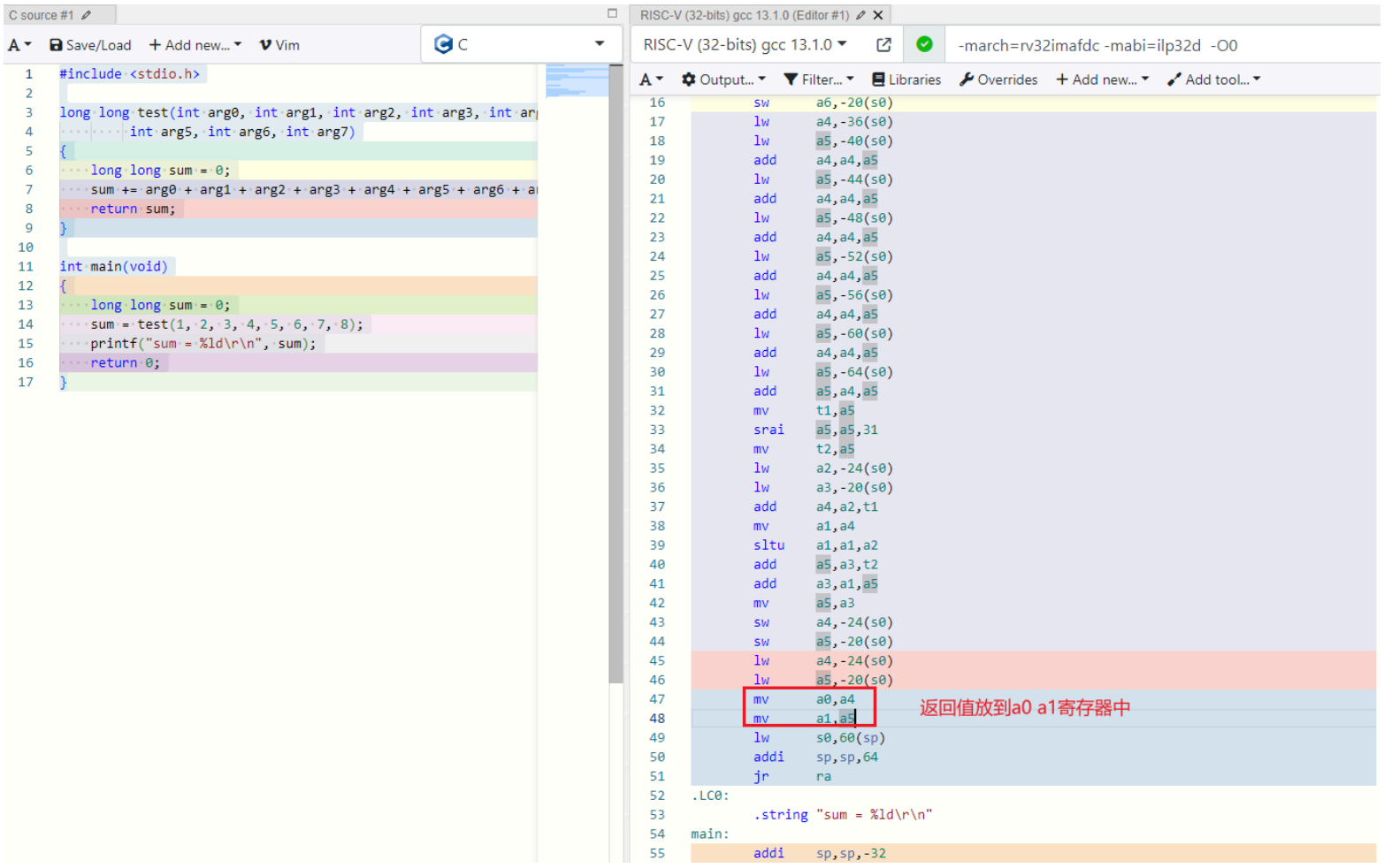
同样举例：

```c
#include <stdio.h>

long long test(int arg0, int arg1, int arg2, int arg3, int arg4,
         int arg5, int arg6, int arg7)
{
    long long sum = 0;
    sum += arg0 + arg1 + arg2 + arg3 + arg4 + arg5 + arg6 + arg7;
    return sum;
}

int main(void)
{
    long long sum = 0;
    sum = test(1, 2, 3, 4, 5, 6, 7, 8);
    printf("sum = %ld\r\n", sum);
    return 0;
}
```

例子中，标量位宽为64bit，XLEN=32bit（-march=rv32imafdc），即返回值位宽超过XLEN，但是不超过2×XLEN，从反汇编可以看出，使用的是a0、a1来存放返回值。



情形4：当函数返回类型是单精度浮点或双精度浮点（float 或double），返回值存放在整型寄存器fa0上

情形5：返回值为结构体（struct），返回值需要细分，如果寄存器能放下则使用寄存器（标量使用a0/a1, 浮点使用fa0/fa1），否则用栈传递

寄存器能放下的情况，结构体放到a0 a1上返回：

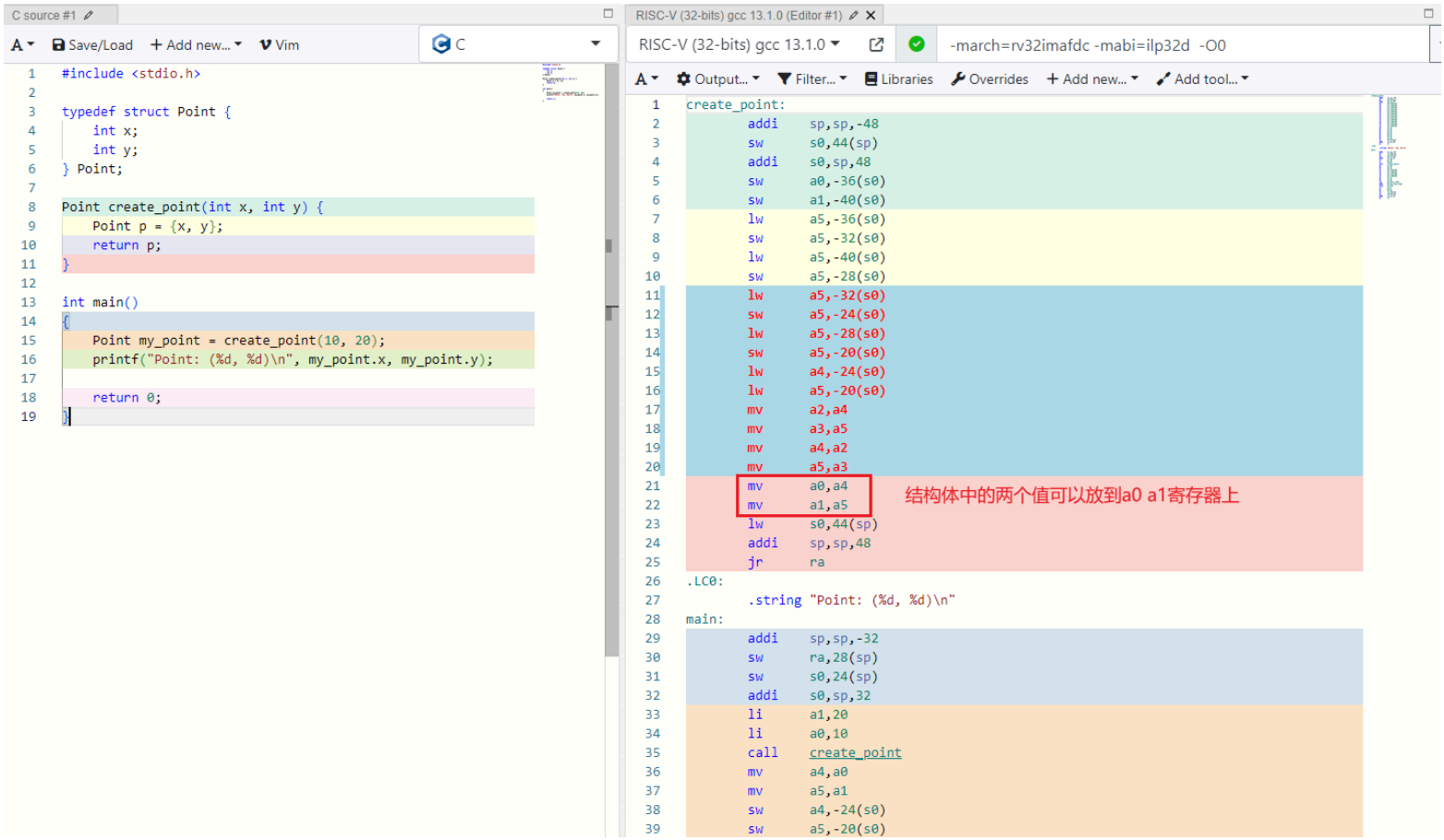
```c
#include <stdio.h>

typedef struct Point {
    int x;
    int y;
} Point;

Point create_point(int x, int y) {
    Point p = {x, y};
    return p;
}

int main()
{
    Point my_point = create_point(10, 20);
    printf("Point: (%d, %d)\n", my_point.x, my_point.y);

    return 0;
}
```



寄存器不能放下的情况，使用栈返回：

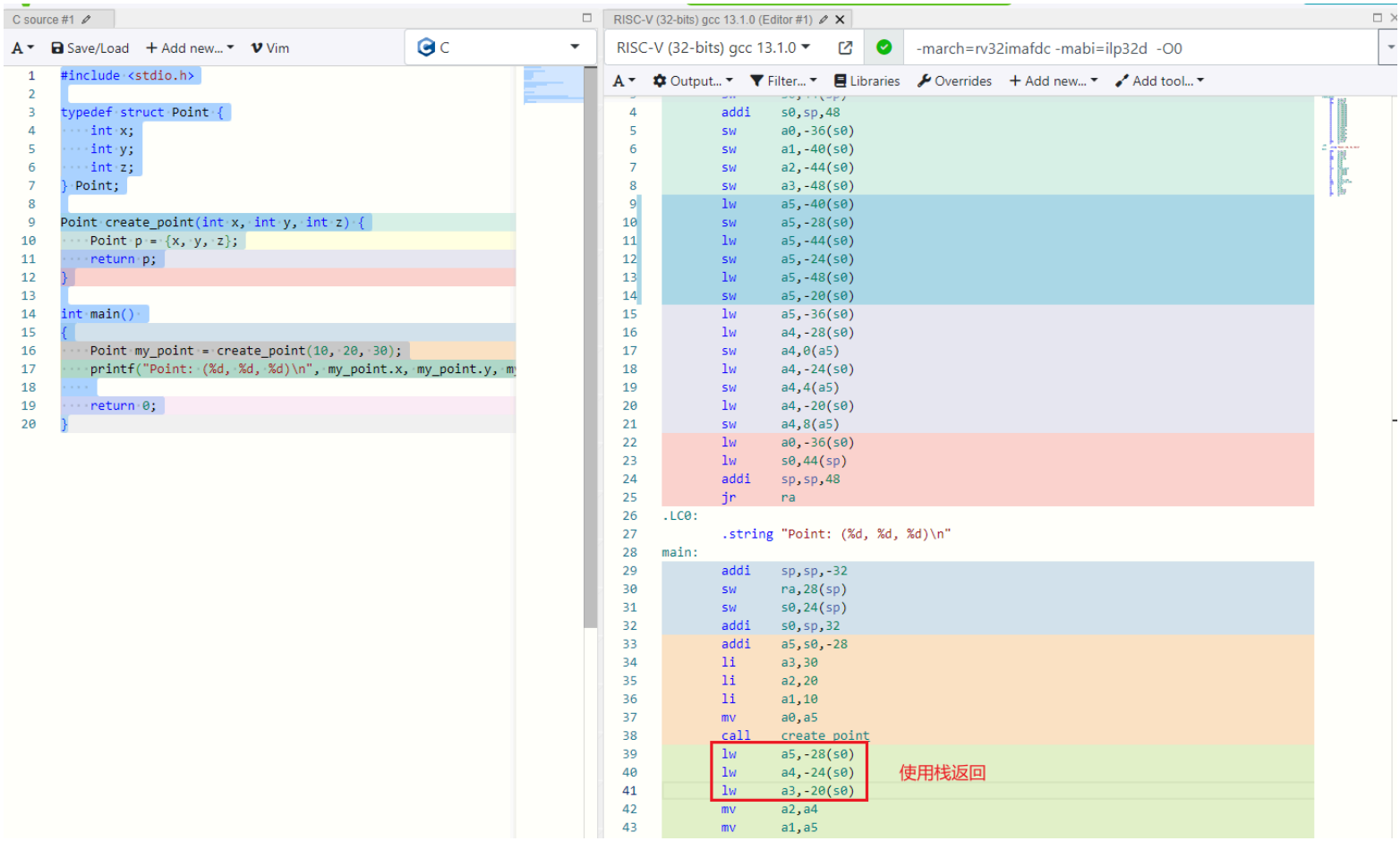
```c
#include <stdio.h>

typedef struct Point {
    int x;
    int y;
    int z;
} Point;

Point create_point(int x, int y, int z) {
    Point p = {x, y, z};
    return p;
}
```

```c
int main()
{
    Point my_point = create_point(10, 20, 30);
    printf("Point: (%d, %d, %d)\n", my_point.x, my_point.y, my_point.z);

    return 0;
}
```



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)