一起学RISC-V汇编第10讲之汇编器语法

目录

了解了RISC-V的基础指令集以及ABI接口，我们就可以动手写汇编程序了，编写汇编程序有两种常用的方式：**汇编源程序**和**内嵌汇编**。

- 汇编源程序：

  即：手写汇编，汇编源程序作为汇编器的输入，一般以.s 或 .S 作为文件扩展名，程序由汇编器指令（Assembler Directive，与架构无关）和汇编指令（Instruction，与指令集相关）两部分构成。

  .S 与 .s 的区别：

  .s文件只包含和CPU架构相关的汇编指令、和汇编相关的汇编指令、注释等，而.S 可以包含预编译，可以简单理解为：**.S = .s + 预编译**，所以.S 中可以使用#include，#ifdef等预编译语句， .S 文件经过cc1预处理器处理后得到.s文件，.s 文件可以由汇编器链接器来汇编链接生成最终的目标文件，

- 内嵌汇编

  即：允许在高级语言（c或c++）中嵌入汇编语言，从而实现汇编语言和高级语言混合编程。

这里只讲汇编器语法，主要参考《汇编语言编程基础-基于LoongArch》，下一讲再讲述内嵌汇编。

1 常用的汇编器指令

汇编器指令与汇编指令不同：

汇编指令：就是前面讲的各种运算，逻辑，移位，比较，跳转等指令，与架构相关；

汇编器指令：是用于指导汇编器怎么工作的指令（如汇编器怎么定义变量和常量，汇编指令在目标文件中如何存放），与架构无关，所以可能ARM，RISCV等架构面向AS汇编器的汇编器指令是相同的。

汇编器语法同样需要常量、变量、函数等，所以就从这里讲起：

1.1 定义字符串变量

定义类似于C语言中的字符串 char str[10] = "hello";

```
.globl str          # 指定符号str为全局变量
.data               # 在data段，所以str并不是常量字符串
.align 3            # 定义2^3 即8字节对齐
.type str,@object   # 类型为对象
.size str,10        # 大小为10字节
```

```
str:                # 字符串符号
.asciz "hello"      # 内容
```

1.2 定义整数变量

定义类似于C语言中的整数 static int var = 20;

```
.local var          # 指定变量为局部类型static，可省略
.data               # 在data段
.align 2            # 定义2^2 即4字节对齐
.type var,@object   # 类型为对象
.size var,4         # 大小为4字节
var:                # 变量符号
.word 20            # 值
```

1.3 定义一个函数

我们可以写一个测试用例，测试汇编是否与C语言等价，见后面例子。

```
/*
add.S

等价于如下C语言：
int add(int a, int b)
{
 return a + b;
}
*/

# 汇编指令

.text               # 代码一般放在text段
.align 2            # 定义2^2 即4字节对齐
.globl add          # 定义作用域
.type add,@function # 类型为函数

add:                # 函数名
add a0, a0, a1
ret
.size add,.-add     # 函数大小
```

以上使用的指令依次说明如下：

1. 全局变量与局部变量

指令	使用示例	描述
.globl （兼容.global）	.globl symbol_name	symbol_name为全局变量或全局函数
.local	.local symbol_name	symbol_name为局部变量或局部函数

在汇编语言中， `.globl` 和 `.local` 是用于定义符号的指令，用于控制符号的作用域。

- `.globl`：用于声明一个全局符号，表示该符号可以在整个程序中被访问和使用，其他文件中可以通过链接器访问这个符号。

- `.local`：用于声明一个局部符号，表示该符号只能在当前模块或函数内部使用，不能被其他文件访问。

简而言之，`.globl` 声明的符号可以被整个程序中的其他文件访问，而 `.local` 声明的符号只能在当前模块或函数内部使用。

2. 段指令

指令	使用示例	描述
.section	.section .text .section .data .section .rodata .section .bss	定义内存段，还可以自定义段
.text	.text	代码段
.data	.data	数据段
.rodata	.rodata	常量区，用来存const修饰的常量或字符串常量
.bss	.bss	未被初始化数据段

3. 对齐方式

指令	使用示例	描述
.align	.align 3	2^3 即8字节对齐
.p2align	.p2align 3	2的n次幂字节对齐，2^3 即8字节对齐
.balign	.balign 3	3字节对齐

汇编指令 `.align expr` 用于指定符号的对齐方式，不同架构下 `.align expr` 意思不同，比如x86中， .align 3表示3字节对齐，而在其它一些平台可能表示8（2^3）字节对齐，为了避免歧义，所以引入了.p2align与.balign两条汇编指令，它们的行为是确定的，.p2align 3 表示8字节(2^3)对齐，而 .balign 4 在任何架构中都表示4字节对齐。

4. 类型对象

指令	使用示例	描述
.type	.type my_function, @function .type var, @object	定义my_function为函数，定义var为变量

5. 设置符号大小

指令	使用示例	描述
.size	.size var, 10 .size my_function, .-my_function	用于设置符号（包括变量和函数）的大小： 设置变量大小时接一个正整数，如：.size var, 10； 当设置函数大小时，常为“. - my_function ”动态计算大小， 因为手动计算函数大小不方便，伪指令可能会被展开。

6. 定义字符串

指令	使用示例	描述
.string	.string “hello”	定义字符串
.ascii	.string “hello\0”	定义字符串，.ascii在字符串末尾不会自动追加'\0'
.asciz	.asciz “hello”	定义字符串，.asciz在字符串末尾自动追加'\0'

以上三种方式都可以用于定义一个字符串，且可不带参数或者多个字符串（逗点分开），用于把汇编好的字符串存入连续的地址。

7. 定义数据

汇编指令中，用于定义数据类型的汇编指令有：

指令	使用示例	描述
.byte	.byte 10	定义字节8bit数 10
.half	.half 100	定义半字16bit
.word	.word 10000	定义字32bit
.long	.long 10000	与 `.word` 相同
.8byte	.8byte 100000000	用于分配8字节的内存空间
.dword	.dword 100000000	与 `.long` 相同，用于分配4字节的内存空间并初始化
.quad	.quad 10000000000	用于分配8字节的内存空间
.float	.float 3.14	分配4字节的内存空间，使用单精度浮点来初始化
.double	.double 3.14	分配8字节的内存空间，使用双精度浮点来初始化

2 其它汇编器指令

2.1 条件编译与文件引用

GNU汇编器提供如：.set/.if/.else/.endif/.ifdef/.ifndef/.ifeq等条件编译指令，但我们也可以直接在.S 中使用C语言的#ifdef、#else、#endif、#define等预处理指令。

同样，文件引用可以.include "file"来实现，也可以直接在.S 中使用C语言的#include等预处理指令，常用的指令如下表：

GNU汇编	等价C语言预处理指令（需使用.S）	功能
.set FLAG, 0	#define FLAG 0	设置常量
.if FLAG==1 .else .endif	#if FLAG==1 #else #endif	条件编译
.ifdef symbol .ifndef symbol	#ifdef symbol #ifndef symbol	是否定义符号symbol

GNU汇编	等价C语言预处理指令（需使用.S）	功能
.include "file"	#include "file"	引用文件

2.2 宏定义

汇编器指令.macro name args .endm 功能类似c语言中的宏定义功能，其中name为宏名称，args为参数，可以为多个参数，参数之间可以使用空格或者逗号分隔，也可以为0个参数，以.endm结尾。如下例表示插入a条nop指令的宏：

```
.text
.macro inset_nop a   # 宏名称为inset_nop，参数为a
    .rept \a         # 宏内使用参数加上反斜杠\
     nop
    .endr
.endm
```

其中：宏定义体中使用参数时格式为\参数，.rept \a 和 .endr 用于将内部的语句展开a次。

如汇编某处插入10条nop指令，可以这么调用：

```
inset_nop 10
```

2.3 循环展开

上例已经展示了循环展开的使用，汇编器指令“.rept count”和“.endr”可用于将其内部的语句循环展开count次，

```
.globl table           # 指定数组为全局变量
.data                  # 在data段，所以str并不是常量字符串
.align 2               # 定义2^2 即4字节对齐
.type table,@object    # 类型为对象
.size table,10         # 大小为4字节
table:                 # 数组
.word
.rept 10
.word 200
.endr
```

2.4 本地标签和程序跳转

为了方便程序的编写，汇编器指令中提供一种本地标签(Local Label)用于逻辑跳转。本地标签有如下两种方式：

- 可采用编号（可以为数字、字母、特殊字符或其组合）加冒号“:”的格式
- 标签“Nb”和“Nf”(其中“N”是数字)，这是 GNU 程序集的智能扩展。它可以“向前”搜索“f”，“向后”搜索“b”，找到标签N。

  ```
  1: j 1f  # 向后跳转到第三条（即1:j 2f）位置
  2: j 1b  # 向前跳转到第一条（即1:j 1f）位置
  1: j 2f  # 向后跳转到第四条（即2:j 1b）位置
  2: j 1b  # 向前跳转到第三条（即1:j 2f）位置
  ```

  等价于：

  ```
  label1: j label3
  label2: j label1
  ```

```
label3: j label4
label4: j label3
```

2 汇编源程序例子

举一个汇编源程序的例子：

```
## add.s

# 函数add_asm
.text
.align 2
.globl add_asm
.type add_asm,@function
add_asm:
add a0, a0, a1
ret
.size add_asm,.-add_asm


# 字符串变量，有FLAG控制，FLAG=1时str="hello flag1"，否则str="hello flag0"
.set FLAG,1
.globl str
.data
.align 2
.type str,@object
.size str,12
.if FLAG == 1
str:
.asciz "hello flag1"
.else
str:
.asciz "hello flag0"
.endif

# 数组table，相当于int table[10];
.globl table          # 指定数组为全局变量
.data                 # 在data段，所以str并不是常量字符串
.align 2              # 定义2^2 即4字节对齐
.type table,@object   # 类型为对象
.size table,40        # 大小为40字节
table:                # 数组
.word
.rept 10
.word 200
.endr
```

测试代码：

```
// main.c

#include <stdio.h>
#include <stdlib.h>

extern char *str;
extern int table[10];
extern int add_asm(int a, int b);
```

```
int add_c(int a, int b)
{
    return a + b;
}

int main(void)
{
    int a = 10;
    int b = 21;

    int res_c, res_asm;

    res_c = add_c(a, b);
    res_asm = add_asm(a, b);

    printf("res_c = %d res_asm = %d\r\n", res_c, res_asm);

    printf("str is %s\r\n", &str);

    for (int i = 0; i < 10; i++) {
        printf("table[%d] = %d\r\n", i, table[i]);
    }
    return 0;
}
```

编译：

```
riscv64-unknown-linux-gnu-gcc -O2 -static add.S main.c -o demo_asm
```

拷贝到risc-v linux环境，执行， log如下：

```
res_c = 31 res_asm = 31
str is hello flag1
table[0] = 200
table[1] = 200
table[2] = 200
table[3] = 200
table[4] = 200
table[5] = 200
table[6] = 200
table[7] = 200
table[8] = 200
table[9] = 200
```

示例2：rv32上的打印helloword代码

```
.text
.align 2                # 指示符：将代码按 2^2 字节对齐
.globl main             # 指示符：声明全局符号 main
main:                   # main 的开始符号
addi sp,sp,-16          # 分配栈帧
sw ra,12(sp)            # 保存返回地址
lui a0,%hi(string1)     # 计算 string1
addi a0,a0,%lo(string1) # 的地址
lui a1,%hi(string2)     # 计算 string2
addi a1,a1,%lo(string2) # 的地址
call printf             # 调用 printf 函数
```

```
lw ra,12(sp)              # 恢复返回地址
addi sp,sp,16             # 释放栈帧
li a0,0                   # 装入返回值 0
ret                       # 返回
.size main,.-main

.section .rodata          # 指示符：进入只读数据节
.balign 4                 # 指示符：将数据按 4 字节对齐
string1:                  # 第一个字符串符号
.string "Hello, %s!\n"    # 指示符：以空字符结尾的字符串
string2:                  # 第二个字符串符号
.string "world"           # 指示符：以空字符结尾的字符串
```

参考：

1. 汇编（一）：risc-v汇编语法 - 知乎 (zhihu.com)
2. riscv-asm-manual/riscv-asm.md at master · riscv-non-isa/riscv-asm-manual · GitHub
3. linux/include/linux/linkage.h