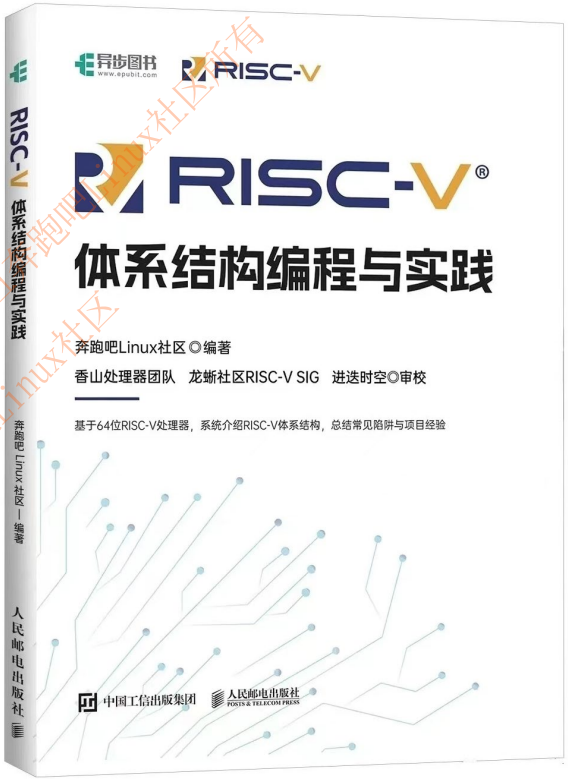
第4季

链接器与链接脚本

本节课主要内容

- 本章主要内容
  - 链接脚本语法介绍
  - 加载重定位
  - 链接重定位
  - 松弛链接优化
  - 实验

技术手册：
1. Using LD, the GNU Linker, v2.34



本节课主要讲解书上第6章内容

扫码订阅RISC-V视频课程

链接器Linker

- 链接器（Linker）是一个程序，将一个或多个由编译器或汇编器生成的目标文件外加库链接为一个可执行文件
- 链接脚本最终会把一大堆编译好的二进制文件（.o文件）综合生成最终二进制可执行文件，也就是把每一个二进制文件整合到一个大文件中。这个大文件有一个总的text/data/bss段。
- GNU Linker采用AT&T链接脚本语言

- 官方文档：v2.34

The GNU linker

ld
(GNU Binutils)
Version 2.34

Compiled by Ben Shushu
Wechat: runninglinuxkernel

```
rlk@master:~$ riscv64-linux-gnu-ld -v
GNU ld (GNU Binutils for Ubuntu) 2.34
```

ld命令

- riscv64-linux-gnu-ld：RISC-V版本的链接器命令
- 命令参数查看： riscv64-linux-gnu-ld –help
- 简单例子：

```
$ ld -o mytest  test1.o test2.o -lc
```

- 查看LD命令默认的链接脚本：riscv64-linux-gnu-ld --verbose
- Benos的例子：

```
39 benos.bin: $(SRC_DIR)/linker.ld $(OBJ_FILES)
40        $(ARMGNU)-ld -T $(SRC_DIR)/linker.ld  -Map benos.map -o $(BUILD_DIR)/benos.elf  $(OBJ_FILES)
41        $(ARMGNU)-objcopy $(BUILD_DIR)/benos.elf -O binary benos.bin
```

常用参数：
-T： 指定链接脚本
-Map：输出一个 符号表 文件
-o: 输出最终可执行二进制文件

表 6.1　　ld 命令的常用选项

选　　项	说　　明
-T	指定链接脚本
-Map	输出一个符号表文件
-o	输出最终可执行二进制文件
-b	指定目标代码输入文件的格式
-e	使用指定的符号作为程序的初始执行点
-l	把指定的库文件添加到要链接的文件清单中
-L	把指定的路径添加到搜索库的目录清单中
-S	忽略来自输出文件的调试器符号信息
-s	忽略来自输出文件的所有符号信息
-t	在处理输入文件时显示它们的名称
-Ttext	使用指定的地址作为代码段的起始点
-Tdata	使用指定的地址作为数据段的起始点
-Tbss	使用指定的地址作为未初始化的数据段的起始点
-Bstatic	只使用静态库
-Bdynamic	只使用动态库
-defsym	在输出文件中定义指定的全局符号

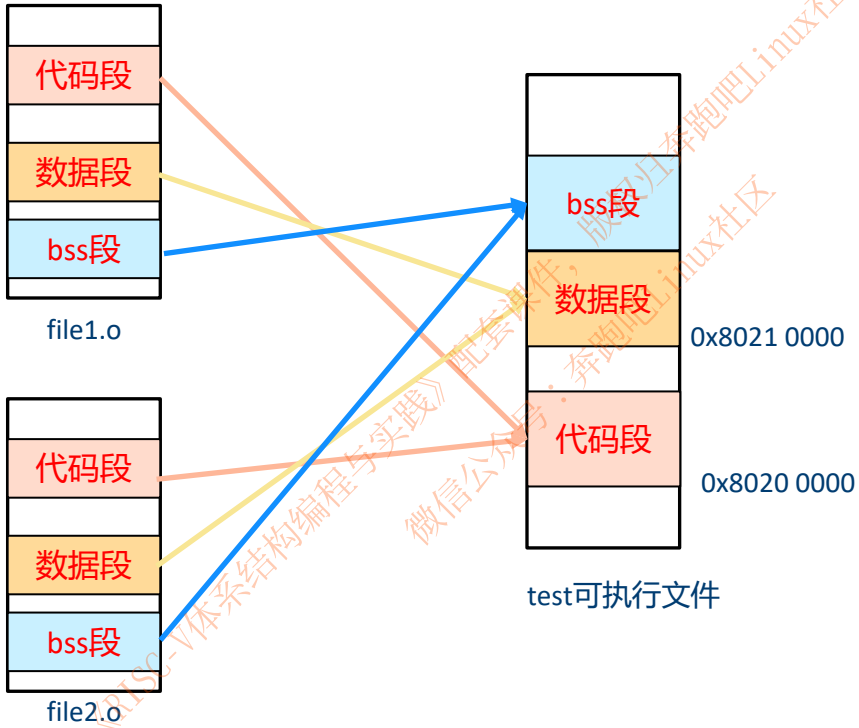
一个简单的例子入门

test.ld 链接脚本

```
1  SECTIONS
2  {
3          . = 0x80200000,
4          .text : { *(.text) }
5          . = 0x80210000;
6          .data : { *(.data) }
7          .bss : { *(.bss) }
8  }
```

file1.o和file2.o 可重定位目标文件 根据链接脚本文件的内存布局链接成一个test可执行文件

file1.o和file2.o的所有段还没有分配地址



Test可执行文件的所有段 根据链接脚本要求已经取得了最终地址

基本概念

- 输入段（input section），输出段（output section）
- 每个段包括 name和大小
- 段的属性：
  - loadable：运行时会加载这些段的内容到内存中
  - allocatable：运行时不会加载段的内容，
- 段的地址：
  - VMA（virtual memory address）：虚拟地址，运行地址
  - LMA （load memory address）：加载地址
  - 通常 ROM的地址为加载地址，而RAM的地址为VMA

链接脚本命令

- ENTRY(_start)：设置程序的入口函数。
- 链接程序有如下几种方式来确定入口点：
  - 使用-e参数
  - 使用ENTRY(symbol)
  - 在.text的最开始的地方
  - 0地址

- INCLUDE filename：引入filename的链接脚本

- OUTPUT filename：输出二进制文件，类似在命令行里使用“-o filename”
- OUTPUT_FORMAT(bfd): 输出BFD格式
- OUTPUT_ARCH(bfdarch) ： 输出处理器体系结构格式

符号赋值

- 符号 也可以像 C语言一样赋值

```
symbol = expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
symbol <<= expression ;
symbol >>= expression ;
symbol &= expression ;
symbol |= expression ;
```

- “.” 表示 location counter，表示当前位置
- 例子：

```
1  floating_point = 0;
2  SECTIONS
3  {
4      .text :
5      {
6          *(.text)
7          _etext = .;
8      }
9
10     _bdata = (. + 3) & ~ 3;
11     .data : { *(.data) }
12 }
```

给符号floating_point赋值为0

_etext 赋值为当前位置（当前位置为代码段结束的地方）

设置bdata的起始地址，当前位置为 ： 与4个字节对齐地方

符号的引用

- 高级语言（C语言） 常常 需要引用 链接脚本定义的符号
- 在C语言里，定义一个变量并初始化变量。 例如　int foo = 100;
  - ✓ 编译器会在符号表中定义了一个符号foo
  - ✓ 编译器会在内存中为符号foo存储100.
- 在链接脚本中定义一个变量：
  - ✓ 链接器仅仅在符号表里定义这个符号，没有分配内存来存储变量的值
- 访问链接脚本定义的变量：访问的是变量的地址，不能访问变量的值

```
start_of_ROM    = .ROM;
end_of_ROM      = .ROM + sizeof (.ROM);
start_of_FLASH = .FLASH;
```

链接脚本

```
extern char start_of_ROM[], end_of_ROM[], start_of_FLASH[];

memcpy (start_of_FLASH, start_of_ROM, end_of_ROM - start_of_ROM);
```

C语言引用

- 我们可以在每个段中设置一些符号，以方便C语言访问每个段的起始地址和结束地址

```
SECTIONS
{
    start_of_text = . ;
    .text: { *(.text) }
    end_of_text = . ;

    start_of_data = . ;
    .data: { *(.data) }
    end_of_data = . ;
}
```

设置location counter的符号，例如start_of_text, end_of_text

```
35 extern char _text[], _stext[], _etext[];
36 extern char _data[], _sdata[], _edata[];
37 extern char __bss_start[], __bss_stop[];
38 extern char __init_begin[], __init_end[];
39 extern char _sinittext[], _einittext[];
40 extern char __start_ro_after_init[], __end_ro_after_init[];
41 extern char _end[];
42 extern char __per_cpu_load[], __per_cpu_start[], __per_cpu_end[];
43 extern char __kprobes_text_start[], __kprobes_text_end[];
44 extern char __entry_text_start[], __entry_text_end[];
45 extern char __start_rodata[], __end_rodata[];
46 extern char __irqentry_text_start[], __irqentry_text_end[];
47 extern char __softirqentry_text_start[], __softirqentry_text_end[];
48 extern char __start_once[], __end_once[];
49
```

Include/asm-generic/sections.h

SECTIONS命令

- SECTIONS命令：告诉链接器如何把输入段（input sections）映射到输出段（output sections），以及如何在内存中摆放这些 输出段。

```
SECTIONS
{
  sections-command
  sections-command
  ...
}
```

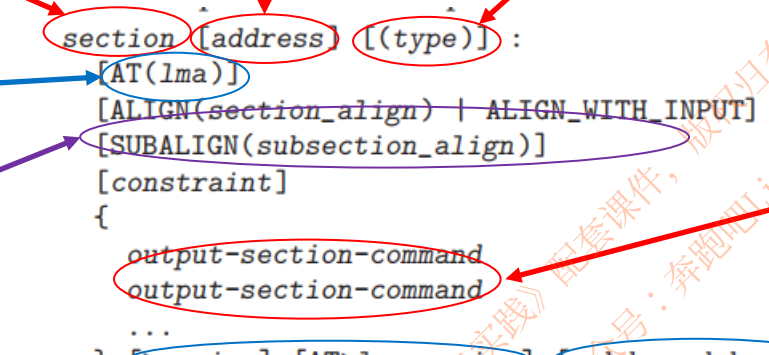
- 输出段的描述符：

```
section [address] [(type)] :
  [AT(lma)]
  [ALIGN(section_align) | ALIGN_WITH_INPUT]
  [SUBALIGN(subsection_align)]
  [constraint]
  {
    output-section-command
    output-section-command
    ...
  } [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp] [,]
```

段名字

VMA（运行地址）

输出段的属性



加载地址

对齐要求

描述输入段如何映射到输出段

指定到特定的内存区域

指定到特定的program segment

LMA加载地址

- 每个段有VMA （虚拟地址，运行地址）以及 LMA（加载地址）
- 在输出段描述符中使用 “AT” 来指定 LMA
- 如果没有通过”AT”来指定 LMA，通常 **LMA = VMA**
- **构建一个基于ROM的映像文件常常会设置输出段的虚拟地址和加载地址不一致**

输入段

1. 输入段用来告诉链接器如何将输入文件映射到内存布局。

```
*(.text)
```

'*'是一个通配符，可以匹配任何文件名的代码段。

2. 例子：

```
*(.text .rodata)
*(.text) *(.rodata)
```

第一条语句按照加入输入文件的顺序把相应的代码段和只读数据段加入；而第二句条语句先加入到所有输入文件的代码段，再加入到所有输入文件的只读数据段

3. EXCLUDE_FILE命令

```
EXCLUDE_FILE (*crtend.o *otherfile.o) *(.ctors)

*(EXCLUDE_FILE (*somefile.o) .text .rodata)

EXCLUDE_FILE (*somefile.o) *(.text .rodata)
```

输入段的例子

```
1    SECTIONS {
2        outputa 0x10000 :
3        {
4            all.o
5            foo.o (.input1)
6        }
7        outputb :
8        {
9            foo.o (.input2)
10           foo1.o (.input1)
11       }
12       outputc :
13       {
14           *(.input1)
15           *(.input2)
16       }
17   }
```

书上例6-6

案例分析

- 下面的链接文件会创建3个段，其中
  - ✓ text段的虚拟地址和加载地址为0x1000
  - ✓ mdata段的虚拟地址设置为0x2000，但是通过AT符号指定了加载地址是在text段的结束地址，_data指定了data段的虚拟地址为0x2000。
  - ✓ bss段的虚拟地址是在0x3000。

```
SECTIONS
 {
        .text 0x1000 : { *(.text) _etext = . ; }

        .mdata 0x2000 :
                AT ( ADDR (.text) + SIZEOF (.text) )
        {
                _data = . ;
                *(.data);
                _edata = . ;  }

        .bss 0x3000 :
        { _bstart = . ;  *(.bss) *(COMMON) ; _bend = . ;}
}
```

代码段

mdata段

_data 和_edata只是一个符号，用来记录mdata段的VMA地址

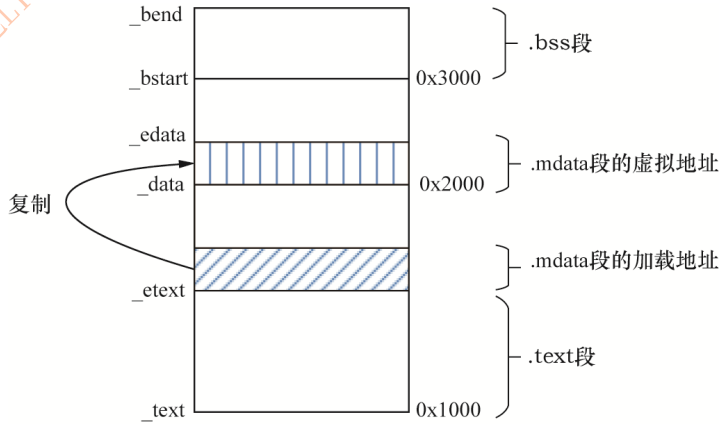
bss段

- mdata段的加载地址和链接地址（虚拟地址）不一样，因此程序的初始化代码需要把mdata段从ROM的加载地址复制到SDRAM中的虚拟地址中。
- 数据段加载地址在_etext起始的地方，mdata段的运行地址是在_data起始的地方，mdata段的大小为“_edata - _data”，下面这段代码，把mdata段从_etext起始的地方复制到_data起始的地方。

```
<程序初始化>
extern char _etext, _data, _edata, _bstart, _bend;
char *src = &_etext;
char *dst = &_data;

/* ROM中包含了data段内容，位于test段结束地址。把它们复制到data的链接地址 */
while (dst < &_edata)
  *dst++ = *src++;

/* 清bss.  */
for (dst = &_bstart; dst< &_bend; dst++)
  *dst = 0;
```



常见的内建函数（builtin functions）

- ABSOLUTE(exp)返回表达式（exp）的绝对值。它主要用于在段定义中给符号赋绝对值。

```
SECTIONS
{
      . = 0xb0000,
      .my_offset : {
            my_offset1 = ABSOLUTE(0x100);
            my_offset2 = (0x100);
      }
}
```

```
my_offset       0x00000000000b0000              0x0
                0x0000000000000100              my_offset1 = ABSOLUTE (0x100)
                0x00000000000b0100              my_offset2 = 0x100
```

- ADDR(section)： 返回前面已经定义过的段的VMA地址

```
SECTIONS { ...
  .output1 :
    {
    start_of_output_1 = ABSOLUTE(.);
    ...
    }
  .output :
    {
    symbol_1 = ADDR(.output1);
    symbol_2 = start_of_output_1;
    }
... }
```

- ALIGN(n)： 返回 下一个与n字节对齐的地址，它是基于当前的位置（location counter）来计算对齐地址的。
- 注意这里是n个字节，而不是2^n个字节。

```
SECTIONS { ...
  .data ALIGN(0x2000): {
    *(.data)
    variable = ALIGN(0x8000);
  }
... }
```

data段 会在下一个与0x2000对齐的地址上，

定义一个variable的变量，这个变量是在下一个与0x8000对齐的地址上

- SIZEOF(section）：返回一个段的大小

```
SECTIONS{ ...
  .output {
    .start = . ;
    ...
    .end = . ;
    }
  symbol_1 = .end - .start ;
  symbol_2 = SIZEOF(.output);
... }
```

- MAX(exp1， exp2）/ MIN（exp1，exp2）：返回两个表达式的最大值或者最小值
- LOADADDR(section)：返回段的加载地址。
- INCLUDE引入另外的链接脚本文件

```
SECTIONS
{
  INCLUDE "sbi/sbi_base.ld"
}
```

- PROVIDE：从链接脚本中导出一个符号，不过只有当这个符号没有定义在任何链接目标时才会被使用。

```
SECTIONS
{

  . = 0x80500000,
  PROVIDE (my_label = .);

  . = 0x80200000,
  .text : { *(.text.boot) }

  ...
}
```

```
1   .global my_label
2   my_label:
3       .byte 8
4
5   test_provide:
6           la a0, my_label
7           ret
```

Memory 命令

MEMORY命令描述目标平台上内存块的位置与长度

```
MEMORY
  {
    name [(attr)] : ORIGIN = origin, LENGTH = len
    ...
  }
```

Attr属性：

- ✓ R：只读
- ✓ W：可读可写
- ✓ X：可执行
- ✓ A：可分配
- ✓ I/L：已经初始化的段
- ✓ !：对属性进行反转

ORIGIN 是一个关于内存区域地始地址的表达式，可以简写成org或者o

LENGTH是一个关于内存区域长充，可以简写成len或者l

```
MEMORY
  {
    rom (rx)  : ORIGIN = 0, LENGTH = 256K
    ram (!rx) : org = 0x40000000, l = 4M
  }
```

例子

定义rom，只读可执行属性，起始地址为0，大小为32KB

定义ram，可读可写可执行，起始地址0x20000000，大小64KB

text.boot段的加载地址放在rom里面

text段的加载地址放在rom里面

数据段的加载地址放在rom，运行地址放在ram

```
/*
 * 定义两个内存区域rom和ram。rom只有读和执行权限，ram需要可读可写可执行权限
 */
MEMORY
{
  rom (rxa) :   ORIGIN = 0x00000000, LENGTH = 32K
  ram (rwxa) :   ORIGIN = 0x20000000, LENGTH = 64K
}

SECTIONS
{
        . = 0x0,
        .text.boot : {
                *(.text.boot)
                *(.rela.text.boot)
        } AT>rom

        /* 代码段放在ROM里 */
        _text = .;
        .text : {
                *(.text)
                *(.rela.text)
                . = ALIGN(8);
        } AT>rom
        _etext = .;

        /* 数据段，加载地址(LMA) 通过“AT>rom”
         * 放在rom中，运行地址(VMA)放在ram里*/
        .data :
        {
                _data = .;
                *(.rodata .rodata*)
                *(.data .data*)
                *(.sdata .sdata*)
                *(.srodata .srodata*)
                . = ALIGN(8);
                _edata = .;

                . = ALIGN(4096);
                *(.stack)
        } >ram AT>rom

        /* bss放在ram里 */
```

实验1：分析benos中的链接脚本文件

目的：熟悉链接脚本语言

1. 分析 benos中的链接脚本文件linker.ld, 详细分析每一条语句的含义。

s/linker.ld

```
SECTIONS
{
        . = 0x80200000,

        .text : { *(.text.boot) }
        .text : { *(.text) }
        .rodata : { *(.rodata) }
        .data : { *(.data) }
        . = ALIGN(0x8);
        bss_begin = .;
        .bss : { *(.bss*) }
        bss_end = .;
}
```

实验2：打印每个段的内存布局

目的：熟悉链接脚本语言

1. 在C语言中打印出benos映像文件的内存布局图，即每个段的起始地址和结束地址，以及段的大小。
2. 查看benos.map文件进行对比，打印的内存布局是否正确。

```
BenOS image layout:
  .text.boot: 0x80200000 - 0x80200032 (    50 B)
       .text: 0x80200038 - 0x80201188 (  4432 B)
     .rodata: 0x80201188 - 0x80201478 (   752 B)
       .data: 0x80201478 - 0x80203000 (  7048 B)
        .bss: 0x80203010 - 0x80223420 (132112 B)
```

3. 修改链接文件，把data段的VMA修改成0x80209000，然后在打印出内存布局看是否有什么变化。

Part 2 重定位

加载重定位

- 加载重定位（Load Relocation）：链接器在做加载时的重定位

- 加载地址：存储代码的物理地址，在GNU链接脚本里称为LMA。
- 运行地址：程序运行时的地址，在GNU链接脚本里称为VMA。
- 链接地址：在编译、链接时指定的地址，编程人员设想将来程序要运行的地址。

当一个程序上述3个地址不一致的时候，
需要做加载重定位

情况1：链接地址等于运行地址与加载地址的情况

```
1     SECTIONS
2     {
3         . = 0x80200000;
4         .text.boot : { *(.text.boot) }
5         .text : { *(.text) }
6         .rodata : { *(.rodata) }
7         .data : { *(.data) }
8         . = ALIGN(0x8);
9         bss_begin = .;
10        .bss : { *(.bss*) }
11        bss_end = .;
12    }
```

```
rlk@master:benos$ cat benos.map

Memory Configuration

Name             Origin             Length             Attributes
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map

                 0x0000000080200000                . = 0x80200000

.text            0x0000000080200000      0x4f6
 *(.text.boot)
 .text.boot      0x0000000080200000       0x2c build/boot_s.o
                 0x0000000080200000                _start
 *(.text)
 .text           0x000000008020002c      0x112 build/uart_c.o
                 0x000000008020002c                uart_send
                 0x000000008020006c                uart_send_string
                 0x00000000802000a2                uart_init
```

加载地址 = 运行地址 = 链接地址 = 0x80200000

情况2：加载地址不等于链接地址的情况

```
1     TEXT_ROM = 0x80300000;
2
3     SECTIONS
4     {
5
6         . = 0x80200000,
7
8         _text_boot = .;
9         .text.boot : { *(.text.boot) }
10        _etext_boot = .;
11
12        _text = .;
13        .text : AT(TEXT_ROM)
14        {
15            *(.text)
16        }
17        _etext = .;
18
19        ...
20    }
```

```
rlk@master:benos$ cat benos.map

Memory Configuration

Name             Origin             Length             Attributes
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map

                 0x0000000080300000                TEXT_ROM = 0x80300000
                 0x0000000080200000                . = 0x80200000
                 0x0000000080200000                _text_boot = .

.text.boot       0x0000000080200000       0x5a
 *(.text.boot)
 .text.boot      0x0000000080200000       0x5a build/boot_s.o
                 0x0000000080200000                _start
                 0x000000008020005a                _etext_boot = .
                 0x0000000080200060                . = ALIGN (0x8)
                 0x0000000080200060                _text = .

.text            0x0000000080200060     0x114a load address 0x0000000080300000
 *(.text)
 .text           0x0000000080200060      0x94c build/printk_c.o
                 0x0000000080200362                myprintf
```

代码段：

加载地址 = 0x80300000
运行地址 = 链接地址 = 0x80200000
加载地址！ = 链接地址

```
.globl _start
_start:
      /*
         假设代码段存储在ROM中(LMA)，而ROM的地址在0x80300000

         我们需要把代码段 从加载地址(LMA)拷贝到 运行地址(VMA)
       */
      la t0, TEXT_ROM
      la t1, _text
      la t2, _etext
.L0:
     ld  a5, (t0)
     sd a5, (t1)
     addi t1, t1, 8
     addi t0, t0, 8
     bltu t1, t2, .L0
```

想要BenOS正常运行，我们需要把代码段从加载地址复制到链接地址

情况3：运行地址不等于链接地址

```
SECTIONS
{
    /*
    * 设置benos链接地址0xffff000000000000
    */
    . = 0xffff000000000000,

    _text_boot = .;
    .text.boot : { *(.text.boot) }
    _etext_boot = .;

    . = ALIGN(8);
    _text = .;
    .text :
    {
            *(.text)
    }
    . = ALIGN(8);
    _etext = .;

        ...
    }
```

```
rlk@master:benos$ cat benos.map

Memory Configuration

Name             Origin             Length             Attributes
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map

                 0xffff000000000000                . = 0xffff000000000000
                 0xffff000000000000                _text_boot = .

.text.boot       0xffff000000000000       0x32
 *(.text.boot)
 .text.boot      0xffff000000000000       0x32 build/boot_s.o
                 0xffff000000000000                _start
                 0xffff000000000032                _etext_boot = .
                 0xffff000000000038                . = ALIGN (0x8)
                 0xffff000000000038                _text = .
```

QEMU上电复位后把benos.bin加载并跳转到0x80200000地址处，而运行时地址和加载地址都为0x80200000，而链接地址为0xffff000000000000

解决办法：
初始化MMU，并且把物理内存DDR映射到内核空间里，然后做一次重定位的操作，让CPU的运行地址重定位到链接地址处

案例分析1：OpenSBI重定位

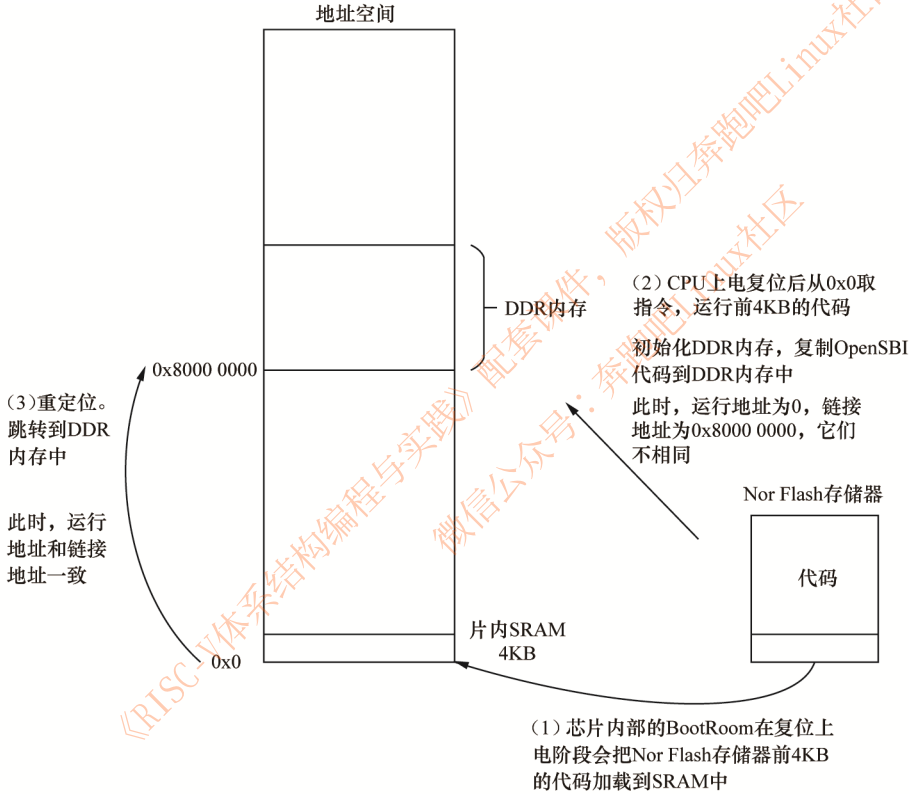
- 芯片启动的一般流程：
  1. 芯片上电后，先运行Boot ROM的程序。
  2. BootROM程序会初始化Nor Flash等外部存储介质，然后把OpenSBI加载到DDR中，并跳转到DDR中运行。
  3. OpenSBI切换到S模式，并跳转到S模式下的U-boot运行。
  4. U-boot初始化一下硬件和启动环境，跳转到Linux内核运行。

- 位置无关代码：指令的执行是与内存地址无关的，无论运行地址和链接地址相等或者不相等，该指令都能正常运行
- 位置有关代码：指令的执行是与内存地址有关的，和当前PC值无关。

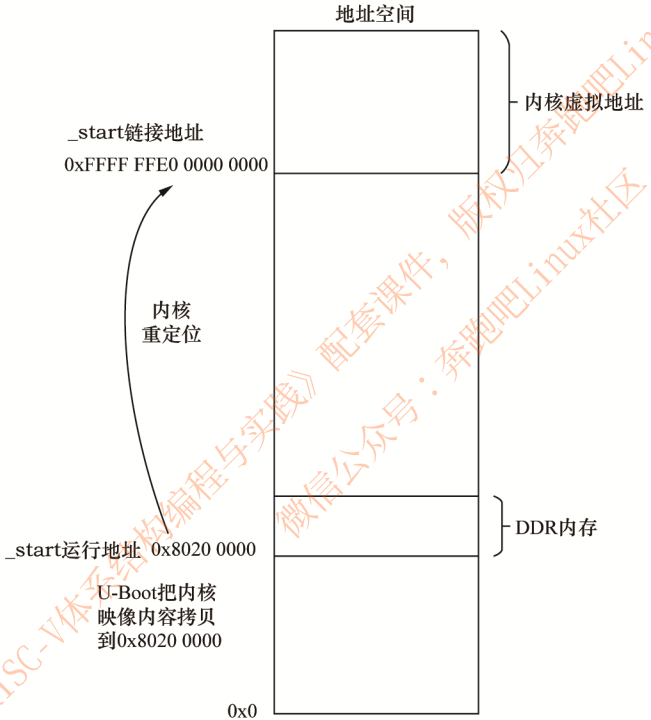
- RISC-V汇编里面通过修改ra返回地址为当前链接地址的值，函数返回时就会跳转到链接地址处运行，从而来实现重定位。

```
li a1, PAGE_OFFSET
add ra, ra, a1
```

案例分析1：OpenSBI重定位



案例分析2：Linux内核重定位



<linux5.15/arch/riscv/kernel/head.S>

```
1     relocate:
2           /* 重定位返回地址 */
3           li a1, PAGE_OFFSET
4           la a0, _start
5           sub a1, a1, a0
6           add ra, ra, a1
7
8           ...
9           ret
```

链接器实验3：加载地址不等于运行地址

目的：熟悉链接脚本的运行地址(VMA)和加载地址(LMA)

1. 我们假设：
   代码段（text段）是存储在ROM中，ROM的地址为0x80300000
   而运行地址为0x80200000，这是RAM的地址。即LMA=0x80300000，VMA=0x80200000

   而其他段例如.text.boot, .data, .rodata以及.bss段的加载地址和运行地址都是在RAM中。

请修改benos的链接脚本以及汇编源码，让benos可以正确运行。

当VMA！ = LMA时，benos.map文件能显示出“load address 0x80300000”

```
.text           0x0000000080200068     0x1c3c load address 0x0000000080300000
 *(.text)
 .text          0x0000000080200068      0xe3c build/benos/printk_c.o
                0x000000008020050c              myprintf
                0x0000000080200cb8              init_printk_done
                0x0000000080200d50              printk
 .text          0x0000000080200ea4      0x1e4 build/benos/uart_c.o
                0x0000000080200ea4              uart_send
                0x0000000080200f04              uart_send_string
                0x0000000080200f70              putchar
                0x0000000080200fc0              uart_init
 .text          0x0000000080201088       0xc8 build/benos/string_c.o
                0x0000000080201088              strlen
                0x00000000802010dc              memcpy
```

链接器实验4：设置链接地址

目的：熟悉链接脚本

1. 修改BenOS的链接脚本，让其链接地址为0xffff000000000000。查看benos.map文件，对比运行地址和链接地址的区别。

```
rlk@master:benos$ cat benos.map

Memory Configuration

Name             Origin             Length             Attributes
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map

                0xffff000000000000                . = 0xffff000000000000
                0xffff000000000000                _text_boot = .

.text.boot      0xffff000000000000       0x3c
 *(.text.boot)
 .text.boot     0xffff000000000000       0x3c build/benos/boot_s.o
                0xffff000000000000                _start
                0xffff00000000003c                _etext_boot = .
                0xffff000000000040                . = ALIGN (0x8)
                0xffff000000000040                _text = .
```

链接重定位与松弛链接优化

- 链接器在链接成最终可执行二进制文件时才具有全局内存地址布局图，这些符号的最终地址是由链接器来分配和确定，这个过程称为**链接重定位（Link Relocation）**。
- **松弛链接优化（linker relaxation），**它的目的是为了减少不必要的指令从而达到优化目的。
- 在RISC-V架构中，需要两条指令来实现对一个符号地址的访问，一条是访问高地址部分，另外一条是访问低地址部分。
- 在链接阶段，通常我们可以使用一条指令来完成上述操作从而达到优化目的
- 松弛链接优化：
  - ✓ 函数跳转优化。
  - ✓ 符号地址访问优化。

链接重定位的过程

<test.c>

```
int a = 5;

int foo(void)
{
   return a;
}

int main(void)
{
    foo();
}
```

例子

<test.s>

```
1     a:
2           .word    5
3           .text
4
5     foo:
6           ...
7           lla  a5,a
8           lw   a5,0(a5)
9           ...
10
11    main:
12          ...
13          call foo
14          ...
```

riscv64-linux-gnu-gcc test.c -o test --save-temps

```
rlk@master:riscv_example$ riscv64-linux-gnu-objdump -d  -r test.o

test.o:     file format elf64-littleriscv


Disassembly of section .text:

0000000000000000 <foo>:
   0: 1141                addi sp,sp,-16
   2: e422                sd   s0,8(sp)
   4: 0800                addi s0,sp,16
   6: 00000797            auipc    a5,0x0    //此处需要做重定位
          6: R_RISCV_PCREL_HI20 a
   a: 00078793            mv   a5,a5
          a: R_RISCV_PCREL_LO12_I    .L0
   e: 439c                lw   a5,0(a5)
  10: 853e                mv   a0,a5
  12: 6422                ld   s0,8(sp)
  14: 0141                addi sp,sp,16
  16: 8082                ret

0000000000000018 <main>:
  18: 1141                addi sp,sp,-16
  1a: e406                sd   ra,8(sp)
  1c: e022                sd   s0,0(sp)
  1e: 0800                addi s0,sp,16
  20: 00000097            auipc    ra,0x0    //此处需要做重定位
         20: R_RISCV_CALL  foo
  24: 000080e7            jalr ra # 20 <main+0x8>
  28: 4781                li   a5,0
  2a: 853e                mv   a0,a5
  2c: 60a2                ld   ra,8(sp)
  2e: 6402                ld   s0,0(sp)
  30: 0141                addi sp,sp,16
  32: 8082                ret
```

```
rlk@master:riscv_example$ riscv64-linux-gnu-objdump -r test.o

test.o:     file format elf64-littleriscv

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000006 R_RISCV_PCREL_HI20  a
000000000000000a R_RISCV_PCREL_LO12_I  .L0
0000000000000020 R_RISCV_CALL     foo
```

表 6.2 常用 RISC-V 重定位类型

编号	重定位类型	说明	计算公式
18	R_RISCV_CALL	函数调用，用于 CALL 和 TAIL 伪指令	S + A - P
23	R_RISCV_PCREL_HI20	PC 相对寻址（高 20 位部分）	S + A - P
24	R_RISCV_PCREL_LO12_I	PC 相对寻址（低 12 位部分），用于加载指令	S - P
25	R_RISCV_PCREL_LO12_S	PC 相对寻址（低 12 位部分），用于存储指令	S - P
26	R_RISCV_HI20	绝对地址寻址（高 20 位部分）	S + A
27	R_RISCV_LO12_I	绝对地址寻址（低 12 位部分），用于加载指令	S + A
28	R_RISCV_LO12_S	绝对地址寻址（低 12 位部分），用于存储指令	S + A
51	R_RISCV_RELAX	表示这条指令会被松弛链接优化	-

变量a的重定位过程

全局变量a的重定位是如何修正的?

重定位类型为R_RISCV_PCREL_HI20，表示PC相对寻址。重定位需要修正的值： **offset = S + A – P**

```
$ riscv64-linux-gnu-readelf -s test
Symbol table '.symtab' contains 66 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    ...
    56: 00000000000005fc    24 FUNC    GLOBAL DEFAULT   12 foo
    60: 0000000000002008     4 OBJECT  GLOBAL DEFAULT   19 a
```

全局变量a的地址为0x2008，计算公式中的S为0x2008。

```
rlk@master:riscv_example$ riscv64-linux-gnu-objdump -r test.o

test.o:     file format elf64-littleriscv

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000006 R_RISCV_PCREL_HI20  a
000000000000000a R_RISCV_PCREL_LO12_I  .L0
0000000000000020 R_RISCV_CALL     foo
```

变量a需要重定位的偏移量为0x6

Foo函数需要重定位的偏移量为0x20

```
$ riscv64-linux-gnu-objdump -d test

00000000000005fc <foo>:
...
 602: 00002797          auipc   a5,0x2
 606: a0678793          addi a5,a5,-1530 # 2008 <a>
 60a: 439c              lw   a5,0(a5)
...

0000000000000614 <main>:
...
 61c: 00000097          auipc   ra,0x0
 620: fe0080e7          jalr -32(ra) # 5fc <foo>
...
```

foo()函数的地址为0x5fc，需要做重定位的偏移地址为0x6，因此需要做重定位的位置是0x5fc + 0x6 = 0x602，即计算公式中的P为0x602。

```
$ riscv64-linux-gnu-readelf -r test.o

Relocation section '.rela.text' at offset 0x1c8 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000006  000800000017 R_RISCV_PCREL_HI2 0000000000000000 a + 0
```

重定位需要附加的字节数为0，计算公式中的A为0。

把上述值代入计算公式： offset = S + A – P = 0x2008 + 0 – 0x602 = 0x1a06。

```
hi20 = (offset >> 12) + offset[11] = (0x1a06 >> 12) + 1 = 2
lo12 = offset & 0xfff = 0xa06
```

```
auipc a5, 0x2
addi  a5, a5, -1530
```

函数foo的重定位过程

```
$ riscv64-linux-gnu-objdump -d test

00000000000005fc <foo>:
...
 602: 00002797          auipc    a5,0x2
 606: a0678793          addi a5,a5,-1530 # 2008 <a>
 60a: 439c              lw   a5,0(a5)
...

0000000000000614 <main>:
...
 61c: 00000097          auipc    ra,0x0
 620: fe0080e7          jalr -32(ra) # 5fc <foo>
...
```

它需要重定位的地方是偏移地址为0x20的位置，重定位的类型为R_RISCV_CALL，其偏移量计算公式为：offset = S – P。

已知S为foo()函数的地址，即0x5fc。P为需要重定位的位置，即0x61c。

代入计算公式offset = S – P = 0x5fc - 0x61c = -32。

```
auipc    ra,0
jalr     ra, -32(ra)
```

松弛链接 之 函数跳转优化

CALL指令是一条伪指令，它主要由如下两条指令组成。

```
call fun

=>
auipc ra, 0
jalr  ra, ra, 0
```

在链接阶段，链接器分配和确定了fun函数的地址，从而可以根据偏移量来确定是否可以选择使用短跳转指令来实现。

```
jal offset
```

例子

```
<test.c>

int foo(void )
{
}

int main(void)
{
    foo();
}
```

```
$ riscv64-linux-gnu-objdump -d -t -r test.o
test.o:     file format elf64-littleriscv

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 g     F .text  0000000000000010 foo
0000000000000010 g     F .text  000000000000001c main

Disassembly of section .text:

0000000000000000 <foo>:
   0:	1141                	addi	sp,sp,-16
   …
   e:	8082                	ret

0000000000000010 <main>:
  10:	1141                	addi	sp,sp,-16
  12:	e406                	sd	ra,8(sp)
  14:	e022                	sd	s0,0(sp)
  16:	0800                	addi	s0,sp,16
  18:	00000097          	auipc	ra,0x0
                18: R_RISCV_CALL	foo
                18: R_RISCV_RELAX	*ABS*
  1c:	000080e7          	jalr	ra # 18 <main+0x8>
  …
  2a:	8082                	ret
```

R_RISCV_CALL：重定位类型
R_RISCV_RELAX告诉链接器，在链接阶段可以进行松弛链接优化。

-mno-relax：关闭松弛链接优化。

```
$ riscv64-linux-gnu-gcc test.c -o test -mno-relax
$ riscv64-linux-gnu-objdump -d -r test
00000000000005fc <foo>:
 5fc: 1141                addi sp,sp,-16
 ...
 60a: 8082                ret

000000000000060c <main>:
 60c: 1141                addi sp,sp,-16
 60e: e406                sd   ra,8(sp)
 610: e022                sd   s0,0(sp)
 612: 0800                addi s0,sp,16
 614: 00000097            auipc    ra,0x0
 618: fe8080e7            jalr -24(ra) # 5fc <foo>
 ...
 626: 8082                ret
```

-mrelax：使能松弛链接优化，GCC默认

```
$ riscv64-linux-gnu-gcc test.c -o test
$ riscv64-linux-gnu-objdump -d -r test

00000000000005ea <foo>:
 5ea: 1141                addi sp,sp,-16
 ...
 5f8: 8082                ret

00000000000005fa <main>:
 5fa: 1141                addi sp,sp,-16
 5fc: e406                sd   ra,8(sp)
 5fe: e022                sd   s0,0(sp)
 600: 0800                addi s0,sp,16
 602: fe9ff0ef            jal  ra,5ea <foo>
 ...
```

在使能 松弛链接优化下，减少了一条指令

松弛链接 之符号地址访问优化

访问32位PC相对地址或者符号

```
auipc a0, %pcrel_hi(sym)
addi a0, a0, %pcrel_lo (sym)
```

例子

```
<benos/src/kernel.c >
long a = 5;
long b = 10;
long data(void) {
      return a | b;
}
void kernel_main(void)
{
      …
      data();
      while(1)
            ;
}
```

```
$ riscv64-linux-gnu-objdump -d -r build_src/kernel_c.o

build_src/kernel_c.o:     file format elf64-littleriscv


Disassembly of section .text:

0000000000000000 <data>:
   0:  00000797            auipc   a5,0x0
                0: R_RISCV_PCREL_HI20 a
                0: R_RISCV_RELAX  *ABS*
   4:  00078793            mv   a5,a5
                4: R_RISCV_PCREL_LO12_I    .L0
                4: R_RISCV_RELAX  *ABS*
   8:  6398                ld   a4,0(a5)
   a:  00000797            auipc   a5,0x0
                a: R_RISCV_PCREL_HI20 b
                a: R_RISCV_RELAX  *ABS*
   e:  00078793            mv   a5,a5
                e: R_RISCV_PCREL_LO12_I    .L0
                e: R_RISCV_RELAX  *ABS*
  12:  639c                ld   a5,0(a5)
  14:  8fd9                or   a5,a5,a4
  16:  853e                mv   a0,a5
  18:  8082                ret
```

R_RISCV_PCREL_HI20表示符号地址的高20位，R_RISCV_PCREL_LO12_I表示符号地址的低12位。

R_RISCV_RELAX用来告诉链接器这里需要做松弛链接优化。

GP优化

RISC-V提供了全局指针寄存器GP，它指向数据段（sdata段）中的一个地址。

如果全局变量存储在以这个GP地址为基地址的±2KB范围内，那么链接器把AUIPC与ADDI指令的组合可以替换成对GP基地址的相对寻址，只需要一条LW或者ADDI指令即可，从而达到优化目的。

```
auipc rd, symbol[31:12]
addi rd, rd, symbol[11:0]

优化=>
addi rd, gp, offset
```

```
$ riscv64-linux-gnu-objdump -d -r benos.elf
…
0000000080200d74 <data>:
    80200d74:   00002797            auipc   a5,0x2
    80200d78:   29c78793            addi    a5,a5,668 # 80203010 <a>
    80200d7c:   6398                ld      a4,0(a5)
    80200d7e:   00002797            auipc   a5,0x2
    80200d82:   29a78793            addi    a5,a5,666 # 80203018 <b>
    80200d86:   639c                ld      a5,0(a5)
    80200d88:   8fd9                or      a5,a5,a4
    80200d8a:   853e                mv      a0,a5
    80200d8c:   8082                ret
```

GP优化前的效果

```
$ riscv64-linux-gnu-objdump -d -r benos.elf
…
0000000080200d7c <data>:
80200d7c:   81018793            addi    a5,gp,-2032 # 80203010 <a>
80200d80:   6398                ld      a4,0(a5)
80200d82:   81818793            addi    a5,gp,-2024 # 80203018 <b>
80200d86:   639c                ld      a5,0(a5)
80200d88:   8fd9                or      a5,a5,a4
80200d8a:   853e                mv      a0,a5
80200d8c:   8082                ret
```

GP优化后的效果

在BenOS中使能GP优化

1. 修改linker.ld链接脚本，新增sdata段和__global_pointer$。
2. 在启动汇编代码中初始化GP寄存器。
3. 重新编译BenOS，并反汇编查看。
4. 查看benos.map

<benos/src/linker.ld>

```
/* 新增 sdata 段和设置_global_pointer$ */
.sdata : {
     __global_pointer$ = . + 0x800;
     *(.sdata)
     *(.sbss*)
 }
```

<benos/src/boot.S>

```
_start:

     ...

.option push
.option norelax
  la gp, __global_pointer$
.option pop
```

```
$ riscv64-linux-gnu-objdump -d -r benos.elf
…
    0000000080200d7c <data>:
    80200d7c:   81018793          addi a5,gp,-2032 # 80203010 <a>
    80200d80:   6398              ld   a4,0(a5)
    80200d82:   81818793          addi a5,gp,-2024 # 80203018 <b>
    80200d86:   639c              ld   a5,0(a5)
    80200d88:   8fd9              or   a5,a5,a4
    80200d8a:   853e              mv   a0,a5
    80200d8c:   8082              ret
```

<benos/benos.map>

```
…
.sdata          0x0000000080203000       0x20
                0x0000000080203800                __global_pointer$ = (. + 0x800)
 *(.sdata)
 .sdata         0x0000000080203000        0x8 build_src/printk_c.o
 .sdata         0x0000000080203008        0x4 build_src/uart_c.o
.sdata          0x0000000080203010       0x10 build_src/kernel_c.o
                0x0000000080203010                a
                0x0000000080203018                b
…
```

链接器实验5：松弛链接优化1

目的：熟悉链RISC-V中的松弛链接优化技术。

在BenOS中，构造一个无法使用函数跳转优化（松弛链接优化）的场景。

链接器实验6：使能GP优化

目的：熟悉链RISC-V中的松弛链接优化技术。

在BenOS中，通过使能GP来 测试符号地址访问优化。

文字不如声音，声音不如视频

扫描订阅RISC-V视频课程

第4季 奔跑吧Linux社区 视频课程

RISC-V体系结构编程与实践

主讲：笨叔

课程名称	进度	时长（分钟）
第1课：课程介绍（免费）	完成	20
第2课：RISC-V体系结构介绍（免费）	完成	47
第3课：RISC-V处理器微架构（免费）	完成	48
第4课：搭建RISC-V开发环境（免费）	完成	30
第5课：RISC-V指令集（免费）	完成	128
第6课：RISC-V函数调用规范	完成	40
第7课：RISC-V GNU AS汇编器	完成	42
第8课：RISC-V GNU 链接脚本	完成	90
第9课：RISC-V GNU 内嵌汇编	完成	52
第10课：RISC-V异常处理	完成	80
第11课：RISC-V中断处理	完成	52
第12课：RISC-V内存管理	完成	116
第13课：内存管理实验讲解	完成	36
第14课：cache基础知识	完成	78
第15课：缓存一致性	完成	96
第16课：RISC-V TLB管理	完成	54
第17课：RISC-V原子操作	未录制	
第18课：RISC-V内存屏障	未录制	
第19课：BenOS操作系统相关知识	未录制	
第20课：RVV可伸缩矢量计算	未录制	
第21课：RISC-V压缩指令	未录制	
第22课：RISC-V虚拟化	未录制	
		总计17小时

更多精彩内容马上献上....
微信公众号：奔跑吧Linux社区

视频课程持续更新中...