

特权模式下hello world, 使用rust编译

## 环境搭建

见 "参考资料"<sup>1</sup> 部分内容。

## 新建工程

```
1 $ cargo new aa...
```

汇编代码如下:

```
1  .section .text.entry
2  .globl _start
3  _start:
4      # 加载helloworld地址到a1,编译器会优化为auipc
5      la a1, helloworld
6      # t1 = 13
7      li t1, 13
9  1:
10     # a0 = a1[0], 为sbi接口第一个参数
11     lb a0, 0(a1)
12     # a7 = 1, 1为sbi码, 对应SBI_CONSOLE_PUTCHAR, 打印字符
13     addi a7, x0, 1
14     # 调用sbi
15     ecall
16
17     # a1++
18     addi a1, a1, 1
19     # t1--
20     addi t1, t1, -1
21
22     # if t1 != 0, 往后跳到1标签执行
23
```

```

24     bne t1, x0, 1b
25
26 2:
27     beq x0, x0, 2b
28
29     .data
    helloworld:
        .ascii "Hello Risc-V!"

```

panic代码如下:

```

1 $ cat lang_items.rs
2
3 use core::panic::PanicInfo;
4
5 #[panic_handler]
6 fn panic(_info: &PanicInfo) -> ! {
7     loop {
8
9     }
10 }

```

链接文件如下:

```

1 $ cat linker.ld
2
3 OUTPUT_ARCH(riscv)
4 ENTRY(_start)
5 BASE_ADDRESS = 0x80200000;
6
7
8 SECTIONS
9 {
10     . = BASE_ADDRESS;
11
12     .text : {
13         *(.text.entry)

```

```

14         *(.text .text.*)
15     }
16
17     .data : {
18         *(.data .data.*)
19     }
20
21
22     /DISCARD/ : {
23         *(.eh_frame)
24     }
25 }

```

main代码如下：

```

1 $ cat main.rs
2
3 #![no_std]
4 #![no_main]
5 #![feature(global_asm)]
6
7
8 // use core::arch::asm;
9 use core::arch::global_asm;
10
11 mod lang_items;
12
13 global_asm!(include_str!("entry.asm"));

```

Makefile如下：

```

1 $ cat Makefile
2
3 # Building
4 TARGET := riscv64gc-unknown-none-elf
5 MODE := release
6

```

```
7 KERNEL_ELF := target/${TARGET}/${MODE}/aa
8 KERNEL_BIN := ${KERNEL_ELF}.bin
9 DISASM_TMP := target/${TARGET}/${MODE}/asm
10
11 # Binutils
12 OBJDUMP := rust-objdump --arch-name=riscv64
13 OBJCOPY := rust-objcopy --binary-architecture=riscv64
14
15
16 # Disassembly
17 DISASM ?= -x
18
19 build: env ${KERNEL_BIN}
20
21 env:
22     (rustup target list | grep "riscv64gc-unknown-none-elf (installed)") || rustup
23     cargo install cargo-binutils --vers =0.3.3
24     rustup component add rust-src
25     rustup component add llvm-tools-preview
26
27
28 ${KERNEL_BIN}: kernel
29     @${OBJCOPY} ${KERNEL_ELF} --strip-all -O binary $@
30
31 kernel:
32     @cargo build --release
33
34
35 clean:
36     @cargo clean
37
38 disasm: kernel
39     @${OBJDUMP} ${DISASM} ${KERNEL_ELF} | less
40
41
42 disasm-vim: kernel
43     @${OBJDUMP} ${DISASM} ${KERNEL_ELF} > ${DISASM_TMP}
44     @vim ${DISASM_TMP}
45     @rm ${DISASM_TMP}
```

```
.PHONY: build env kernel clean disasm disasm-vim
```

config文件如下:

```
1 $ cat config
2
3 [build]
4 target = "riscv64gc-unknown-none-elf"
5
6
7 [target.riscv64gc-unknown-none-elf]
8 rustflags = [
9     "-Clink-arg=-Tsrc/linker.ld", "-Cforce-frame-pointers=yes"
10 ]
```

## 注意事项:

如果使用bochs启动的话, 需要进行以下操作:

1. 使用bximage生成 hd 镜像;
2. 使用命令 `$ dd if=aa.bin of=c.img bs=512 count=1 conv=notrunc` 命令把数据写入镜像
3. 使用二进制编辑器 (如 vscode 插件 Hex Editor等) 修改c.img, 找到地址 0x000001F0, 把最后两个字节改为 `55 AA`, 也就是 0x000001FE = 55, 0x000001FF = AA

## 根据RISC-V手册解析二进制

进入 `aa` 目录, 进行编译

```
1 $ cd aa
2 $ make
```

RISC-V指令格式如下:

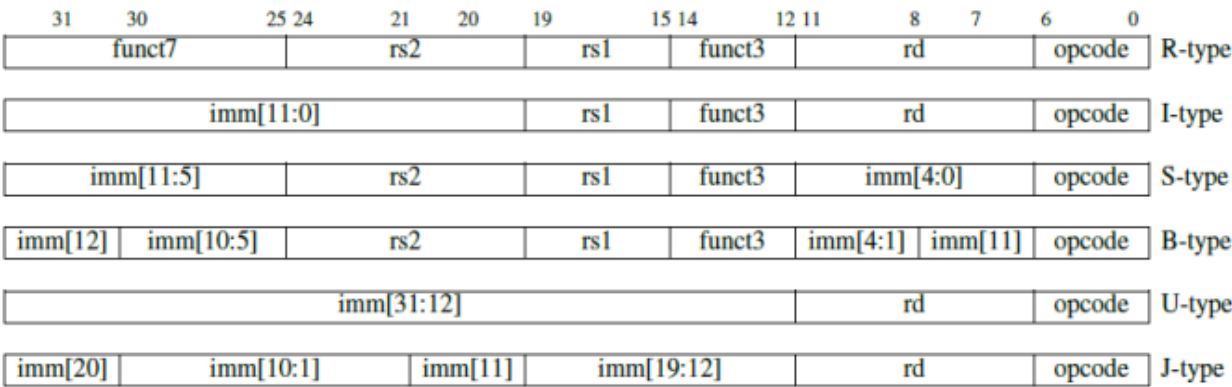


图 2.2: RISC-V 指令格式。我们用生成的立即数值中的位置（而不是通常的指令立即数域中的位置）(imm[x])标记每个立即数字域。第十章解释了控制状态寄存器指令使用 I 型格式的稍微不同的做法。（本图基于 Waterman 和 Asanovi'c 2017 的图 2.2）。

压缩指令格式如下：

Format	Meaning	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CR	Register	funct4				rd/rs1				rs2				op			
CI	Immediate	funct3		imm		rd/rs1				imm				op			
CSS	Stack-relative Store	funct3		imm						rs2				op			
CIW	Wide Immediate	funct3		imm								rd'		op			
CL	Load	funct3		imm		rs1'		imm		rd'		op					
CS	Store	funct3		imm		rs1'		imm		rs2'		op					
CB	Branch	funct3		offset				rs1'		offset				op			
CJ	Jump	funct3		jump target												op	

图 7.8: 16 位 RVC 压缩指令的格式。rd',rs1'和rs2'指的是 10 个常用的寄存器 a0-a5, s0-s1, sp 和 ra。（本图来源于 [Waterman and Asanovi'c 2017] 的表 12.1。）

二进制文件 aa.bin 内容如下：

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded Text
00000000	97	05	00	00	93	85	05	02	35	43	03	85	05	00	85	48	. . . . . 5 C . . . . H
00000010	73	00	00	00	85	05	7D	13	E3	19	03	FE	63	00	00	00	s . . . . } . ä . . þ c . . .
00000020	48	65	6C	6C	6F	20	52	69	73	63	2D	56	21				H e l l o R i s c - V !

参考 [RISC-V手册](#) 对二进制文件进行解析，注意：RISC-V 采用小端表示，指令长度一般为定长，4 个字节

1. 第一条指令 97 05 00 00

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	97	05	00	00	93	85	05	02	35	43	03	85	05	00	85	48
00000010	73	00	00	00	85	05	7D	13	E3	19	03	FE	63	00	00	00
00000020	48	65	6C	6C	6F	20	52	69	73	63	2D	56	21			

此指令按照小端翻译的二进制为

0000 0000 0000 0000 0000 0101 1001 0111

0 0 0 0 0 5 9 7

即从高位到低位为 00 00 05 97

参考RISC-V指令格式，操作码为最低[6:0]位，即 00101111，查找对应的RISC-V手册，此指令对应 `auipc`；

**auipc** rd, immediate  $x[rd] = pc + sext(immediate[31:12] \ll 12)$

PC 加立即数 (Add Upper Immediate to PC). U-type, RV32I and RV64I.

把符号位扩展的 20 位（左移 12 位）立即数加到 *pc* 上，结果写入 *x[rd]*。

31	12 11	7 6	0
immediate[31:12]		rd	0010111

rd为 [11:7]位，即01011，对应寄存器为 `x11`，即 `a1`；

立即数为 [12:31]位，即 0；

所以解析出的汇编指令为 `auipc a1, 0`，跟 `objdump` 反汇编的结果一样。

此指令结果为 `a1 = pc + 0 = 0`。（pc 这里假定为偏移地址 0，实际运行中为入口地址，如x86 实模式为 0x7c00）

## 2. 第二条指令 93 85 5 02

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded Text
00000000 97 05 00 00 93 85 05 02 35 43 03 85 05 00 85 48	. . . . . 5 C . . . . . H
00000010 73 00 00 00 85 05 7D 13 E3 19 03 FE 63 00 00 00	s . . . . . } . ä . . . p c . . .
00000020 48 65 6C 6C 6F 20 52 69 73 63 2D 56 21	H e l l o R i s c - V !

此指令按照小端翻译的二进制为

0000 0010 000 0101 1000 0101 1001 0011

0 2 0 5 8 5 9 3

参考RISC-V指令格式，操作码为最低[6:0]位，即 0010011，查找对应的RISC-V手册，此指令对应 `addi`；

**addi** rd, rs1, immediate  $x[rd] = x[rs1] + sext(immediate)$

加立即数 (Add Immediate). I-type, RV32I and RV64I.

把符号位扩展的立即数加到寄存器 *x[rs1]* 上，结果写入 *x[rd]*。忽略算术溢出。

压缩形式: `c.li rd, imm`; `c.addi rd, imm`; `c.addi16sp imm`; `c.addi4spn rd, imm`

31	20 19	15 14	12 11	7 6	0
immediate[11:0]		rs1	000	rd	0010011

rd为 [11:7]位，即 01011，对应寄存器为 `x11`，即 `a1`；


rs1为 [19:15]位，即 01011，对应寄存器为 `x11`，即 `a1`；

立即数[11:0]低12位对应 [31:20]位, 即 `0000 0010 0000` , 值为` ; (此值为偏移值, 即 `a1` 存放的地址 $a1 = a1 + 36$ , 也就是`a1`指向字符串地址, 可以验证一下)

`a1`之前的地址 `0x00000000` + 32 = `0x00000020` , 此地址正好对应字符串 `Hello Risc-V!` 的首地址。

此指令解析完为 `addi a1, a1, 32` 。

### 3. 第三条指令 `35 43 03 85` ? ? ? ? 需要注意

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	97	05	00	00	93	85	05	02	35	43	03	85	05	00	85	48
00000010	73	00	00	00	85	05	7D	13	E3	19	03	FE	63	00	00	00
00000020	48	65	6C	6C	6F	20	52	69	73	63	2D	56	21			

此指令比较特殊, 我们先按照4个自己取值, 对应二进制为

`1000 0101 0000 0011 0100 0011 0011 0101`

8    5    0    3    4    3    3    5

参考RISC-V指令格式, 操作码为最低[6:0]位, 即 `0110101` , 查找对应的RISC-V手册, 找不到对应的指令, 尴尬了😓

RISC-V也有压缩指令, 即RV32C, 主要是编译器为了优化指令长度, 减小代码体积。一般压缩指令长度为2个字节, 也就是16位。每条短指令必须和一条标准的 32 位 RISC-V 指令一一对应。

所以, 此时, 我们应该取指2个字节, 即 `35 43`

对应的二进制

`0100 0011 0011 0101`

4    3    3    5

从精简指令编码的一般形式[15:13][1:0]可以查到 `010 ... 01` 对应的精简指令为

`c.li`

**c.li** `rd, imm`

`x[rd] = sext(imm)`

立即数加载 (*Load Immediate*). RV32IC and RV64IC.

扩展形式为 `addi rd, x0, imm`.

15	13	12	11	7	6	2	1	0
010	imm[5]	rd	imm[4:0]	01				

`rd`为 [11:7]位, 即 `00110` , 对应寄存器为 `x6` , 即 `t1` ;


立即数[5:0]对应为 [12][6:2]位, 即 `001101` , 值为 `13` ;



对应的压缩指令为 `c.li t1, 13` , 扩展形式为 `addi t1, x0, 13` , 即 `t1 = x0 + 13 = 13` ;

13 代表着字符串的长度, 也就是 t1中存的是字符串长度。

#### 4. 第四条指令 `03 85 05 00`

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	97	05	00	00	93	85	05	02	35	43	03	85	05	00	85	48
00000010	73	00	00	00	85	05	7D	13	E3	19	03	FE	63	00	00	00
00000020	48	65	6C	6C	6F	20	52	69	73	63	2D	56	21			

对应的二进制

0000 0000 0000 0101 1000 0101 0000 0011

参考RISC-V指令格式, 操作码为最低[6:0]位, 即 `0000011` , 查找对应的RISC-V手册, 对应的指令为 `lb` ; (lb有多个变种, 对比[14:12]即可查出是哪个形式, 在此不再赘)

**lb** *rd, offset(rs1)*  $x[rd] = sext(M[x[rs1] + sext(offset)][7:0])$   
字节加载 (*Load Byte*). I-type, RV32I and RV64I.  
从地址  $x[rs1] + sign-extend(offset)$  读取一个字节, 经符号位扩展后写入  $x[rd]$ 。

31	20 19	15 14	12 11	7 6	0
offset[11:0]	rs1	000	rd	0000011	


rd为 [11:7]位, 即 `01010` , 对应寄存器为 `x10` , 即 `a0` ;

rs1为 [19:15]位, 即 `01011` , 对应寄存器为 `x11` , 即 `a1` ;

偏移[11:0]低12位对应 [31:20]位, 即 `0000 0000 0000` , 值为 `0` ; (此值为偏移值)

所以此指令为 `lb a0, 0(a1)` , 也就是 `a0 = a1 + 0` ; (即 a0 值为 a1 指向的地址加偏移0处的1个字节。如第一次值为 `H` )

#### 5. 第五条指令 `85 48`

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	97	05	00	00	93	85	05	02	35	43	03	85	05	00	85	48
00000010	73	00	00	00	85	05	7D	13	E3	19	03	FE	63	00	00	00
00000020	48	65	6C	6C	6F	20	52	69	73	63	2D	56	21			

为什么只取2个字节, 请参考第三条指令说明。

对应的二进制为

0100 1000 1000 0101

从精简指令编码的一般形式[15:13][1:0]可以查到 `010 ... 01` 对应的精简指令为

`c.li` ;

**c.li** rd, imm

$x[rd] = sext(imm)$

立即数加载 (*Load Immediate*). RV32IC and RV64IC.

扩展形式为 **addi** rd, x0, imm.

15	13	12	11	7 6	2 1	0
010	imm[5]	rd	imm[4:0]	01		

rd为 [11:7]位, 即 **10001**, 对应寄存器为 **x17**, 即 **a7**;

立即数[5:0]对应为 [12][6:2]位, 即 **000001**, 值为 **1**;

对应的压缩指令为 **c.li a7, 1**, 扩展形式为 **addi a7, x0, 1**, 即  $a7 = x0 + 1 = 1$ ; (a7的值为)

## 6. 第六条指令 **73 00 00 00**

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 97 05 00 00 93 85 05 02 35 43 03 85 05 00 85 48
00000010 <b>73 00 00 00</b> 85 05 7D 13 E3 19 03 FE 63 00 00 00
00000020 48 65 6C 6C 6F 20 52 69 73 63 2D 56 21

二进制为

0000 0000 0000 0000 0000 0000 0111 0011

参考RISC-V指令格式, 操作码为最低[6:0]位, 即 **1110011**, 查找对应的RISC-V手册, 对应的指令为 **ecall**; (此操作码对应多条指令, 需根据其余高位来进行判断)

**ecall**

**RaiseException(EnvironmentCall)**

环境调用 (*Environment Call*). I-type, RV32I and RV64I.

通过引发环境调用异常来请求执行环境。

31	20 19	15 14	12 11	7 6	0
0000000000000000	00000	000	00000	1110011	

此时进行 **SBI** 调用, 根据前面的 a7 保存的调用号为 1, 调用接口为

**SBI\_CONSOLE\_PUTCHAR**, 即打印1个字符;

a0 保存的为第一个参数 **H**, 即要打印的字符为 **H**。

## 7. 第七条指令 **85 05**

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 97 05 00 00 93 85 05 02 35 43 03 85 05 00 85 48
00000010 73 00 00 00 <b>85 05</b> 7D 13 E3 19 03 FE 63 00 00 00
00000020 48 65 6C 6C 6F 20 52 69 73 63 2D 56 21

对应二进制为

0000 0101 1000 0101

从精简指令编码的一般形式[15:13][1:0]可以查到 **000 ... 01** 对应的精简指令为

**c.addi**;

## c.addi rd, imm

$$x[rd] = x[rd] + \text{sext}(imm)$$

加立即数 (Add Immediate). RV32IC and RV64IC.

扩展形式为 **addi** rd, rd, imm.

15	13	12	11	7 6	2 1	0
000	imm[5]	rd	imm[4:0]	01		

rd为 [11:7]位, 即 **01011**, 对应寄存器为 **x11**, 即 **a1** ;

立即数[5:0]对应为 [12][6:2]位, 即 **000001**, 值为 **1** ;

对应的压缩指令为 **c.addi a1, 1**, 扩展形式为 **addi a1, a1, 1**, 即 **a1 = a1 + 1** ;

从第二条指令, 得知 a1 存的是字符串首地址, 此时条指令结束后, a1往后移了1位, 也就是指向第二个字符的地址 **0x21** ;

### 8. 第八条指令 **7d 13**

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 97 05 00 00 93 85 05 02 35 43 03 85 05 00 85 48
00000010 73 00 00 00 85 05 <b>7D 13</b> E3 19 03 FE 63 00 00 00
00000020 48 65 6C 6C 6F 20 52 69 73 63 2D 56 21

对应二进制

0001 0011 0111 1101

从精简指令编码的一般形式[15:13][1:0]可以查到 **000 ... 01** 对应的精简指令为

**c.addi** ;

## c.addi rd, imm

$$x[rd] = x[rd] + \text{sext}(imm)$$

加立即数 (Add Immediate). RV32IC and RV64IC.

扩展形式为 **addi** rd, rd, imm.

15	13	12	11	7 6	2 1	0
000	imm[5]	rd	imm[4:0]	01		

rd为 [11:7]位, 即 **00110**, 对应寄存器为 **x6**, 即 **t1** ;

立即数[5:0]对应为 [12][6:2]位, 即 **111111**, 值为 **-1** ;

对应的压缩指令为 **c.addi t1, 1**, 扩展形式为 **addi t1, t1, -1**, 即 **t1 = t1 - 1** ;

从第三条指令得知, t1 保存的是字符串长度, 此指令执行完后, 字符串长度减一即 12, 因为第一个字符已经打印完成, 所以剩余12个字符。

### 9. 第九条指令 **e3 19 03 fe**

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	97	05	00	00	93	85	05	02	35	43	03	85	05	00	85 48
00000010	73	00	00	00	85	05	7D	13	E3	19	03	FE	63	00	00 00
00000020	48	65	6C	6C	6F	20	52	69	73	63	2D	56	21		

对应二进制

1111 1110 0000 0011 0001 1001 1110 0011

参考RISC-V指令格式，操作码为最低[6:0]位，即 **1100011**，funct3 [14:12]值为 **001**，对应的指令为 **bnez**；（因为rs2 为0，所以是bnez，而不是 bne）

**bne** *rs1, rs2, offset* if (*rs1*  $\neq$  *rs2*) pc += sext(offset)  
 不相等时分支 (*Branch if Not Equal*). B-type, RV32I and RV64I.  
 若寄存器 *x[rs1]*和寄存器 *x[rs2]*的值不相等，把 *pc* 的值设为当前值加上符号位扩展的偏移 *offset*。  
 压缩形式: **c.bnez** *rs1, offset*

31	25	24	20	19	15	14	12	11	7	6	0
offset[12 10:5]		rs2		rs1		001		offset[4:1 11]			1100011

**bnez** *rs1, offset* if (*rs1*  $\neq$  0) pc += sext(offset)  
 不等于零时分支 (*Branch if Not Equal to Zero*). 伪指令(Pseudoinstruction), RV32I and RV64I.  
 可视为 **bne** *rs1, x0, offset*.

rs1 对应 [19:15]位，即 **00110**，对应寄存器 **x6**，即 **t1**；

rs2 对应 [24:20]位，即 **00000**，对应寄存器 **x0**；

offset [12|10:5][4:1|11] 对应 [31:25][11:7]位，即 **1111111110010**，最后补个0，为第0位。此值为负数 **-14**，即向前偏移14个字节，

当前地址为 **0x18**，向前偏移14个字节为 **0xa**，对应的第四条指令。

此指令的意思：判断 x6 寄存器值如果不等于 0，则跳转到 **0xa** 地址继续执行。

#### 10. 第十条指令 **63 00 00 00**

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	97	05	00	00	93	85	05	02	35	43	03	85	05	00	85 48
00000010	73	00	00	00	85	05	7D	13	E3	19	03	FE	63	00	00 00
00000020	48	65	6C	6C	6F	20	52	69	73	63	2D	56	21		

对应二进制

0000 0000 0000 0000 0000 0000 0110 0011

参考RISC-V指令格式，操作码为最低[6:0]位，即 **1100011**，funct3 [14:12]值为 **000**，查找对应的RISC-V手册，对应的指令为 **beq**；

**beq**  $rs1, rs2, offset$  if ( $rs1 == rs2$ )  $pc += sext(offset)$   
 相等时分支 (*Branch if Equal*). B-type, RV32I and RV64I.  
 若寄存器  $x[rs1]$  和寄存器  $x[rs2]$  的值相等, 把  $pc$  的值设为当前值加上符号位扩展的偏移  $offset$ 。  
 压缩形式: **c.beqz**  $rs1, offset$

31	25 24	20 19	15 14	12 11	7 6	0
offset[12 10:5]	rs2	rs1	000	offset[4:1 11]	1100011	

**beqz**  $rs1, offset$  if ( $rs1 == 0$ )  $pc += sext(offset)$   
 等于零时分支 (*Branch if Equal to Zero*). 伪指令(Pseudoinstruction), RV32I and RV64I.  
 可视为 **beq**  $rs1, x0, offset$ .

11.  $rs1$  对应 [19:15]位, 即 `00000`, 对应寄存器 `x0` ;  
 $rs2$  对应 [24:20]位, 即 `00000`, 对应寄存器 `x0` ;  
 $offset$  [12|10:5][4:1|11] 对应 [31:25][11:7]位, 即 `00000000000000`, 最后补个0, 为第0位。此值为 `0`, 即偏移1个字节,  
 当前地址为 `0x1c`, 偏移0个字节为 `0x1c + 0 = 0x1c`。  
 此指令的意思: 在当前指令进行无限循环。
12. 二进制最后存储内容为字符串。

## 进行反汇编

```

1 $ rust-objdump -S target/riscv64gc-unknown-none-elf/release/aa
2
3 target/riscv64gc-unknown-none-elf/release/aa: file format elf64-littleriscv
4
5 Disassembly of section .text:
6
7
8 0000000080200000 <_start>:
9 80200000: 97 05 00 00    auipc    a1, 0
10 80200004: 93 85 05 02    addi     a1, a1, 32
11 80200008: 35 43                addi     t1, zero, 13
12 8020000a: 03 85 05 00    lb       a0, 0(a1)
13 8020000e: 85 48                addi     a7, zero, 1
14 80200010: 73 00 00 00    ecall
15 80200014: 85 05                addi     a1, a1, 1
16 80200016: 7d 13                addi     t1, t1, -1
17
```

```
80200018: e3 19 03 fe    bnez    t1, 0x8020000a <_start+0xa>
8020001c: 63 00 00 00    beqz    zero, 0x8020001c <_start+0x1c>
```

可以看到，跟参考手册解析内容一致。

## 参考资料：

1. [实验环境配置 — rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 \(rcore-os.github.io\)](#).
  2. [RISC-V手册](#)
- 

## 1. 参考资料：

1. [实验环境配置 — rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 \(rcore-os.github.io\)](#).
2. [RISC-V手册](#)