

# 嵌入式系统 实验二

PB21111733 牛庆源

## 实验要求

- 生成交叉编译器：
  - 使用step-by-step的模式，编译一个你自己的arm-linux-gcc编译器。
  - 修改gcc的代码，使得gcc -v 的输出中包含个人的信息。
  - 使用C代码测试编译器。
- 在linux平台上面安装arm-linux- 工具链：
  - 写一个C语言程序。
  - 使用gcc和arm-gcc编译，比较生成的目标代码的区别。需要使用readelf和objdump等工具。
    - 重点分析ELF文件头部，分段等信息。

## 实验步骤：

### 生成交叉编译器：

1. 按照教程 ([如何构建 GCC 交叉编译器](#)) 步骤，在我的Ubuntu18.04上构建GCC交叉编译器。

#### 1. 准备工作：

软件包下载（下载较新的包，教程中的版本尝试之后发现运行失败）：

```
sudo apt-get install g++ make gawk
wget http://ftpmirror.gnu.org/binutils/binutils-2.28.1.tar.gz
wget http://ftpmirror.gnu.org/gcc/gcc-9.4.0/gcc-9.4.0.tar.gz
wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.15.1.tar.xz
wget http://ftpmirror.gnu.org/glibc/glibc-2.24.tar.xz
wget http://ftpmirror.gnu.org/mpfr/mpfr-4.1.1.tar.xz
wget http://ftpmirror.gnu.org/gmp/gmp-6.2.0.tar.xz
wget http://ftpmirror.gnu.org/mpc/mpc-1.2.1.tar.gz
```

注：由于 *isl* 和 *cloog* 可选，而之前的各种包依赖关系比较复杂，固本实验中没有安装。

提取源包：

```
for f in *.tar*; do tar xf $f; done
```

创建GCC目录到其他一些目录的符号链接（依赖项）：

```
cd gcc-9.4.0
ln -s ../mpfr-4.1.1 mpfr
ln -s ../gmp-6.2.0 gmp
ln -s ../mpc-1.2.1 mpc
cd ..
```

选择安装目录并确定权限以及环境变量：

```
sudo mkdir -p /opt/cross
sudo chown nqy1002_ /opt/cross
export PATH=/opt/cross/bin:$PATH
```

## 2. binutils配置:

```
mkdir build-binutils
cd build-binutils
../binutils-2.28.1/configure --prefix=/opt/cross --target=aarch64-linux --
disable-multilib
make -j4
make install
cd ..
```

## 3. Linux内核头文件:

```
cd linux-4.15.1
make ARCH=arm64 INSTALL_HDR_PATH=/opt/cross/aarch64-linux headers_install
cd ..
```

## 4. C/C++编译器:

```
mkdir -p build-gcc
cd build-gcc
../gcc-9.4.0/configure --prefix=/opt/cross --target=aarch64-linux --enable-
languages=c,c++ --disable-multilib
make -j4 all-gcc
make install-gcc
cd ..
```

## 5. 标准C库头文件和启动文件:

```
mkdir -p build-glibc
cd build-glibc
CFLAGS="-O2 -Wno-error" ../glibc-2.24/configure --prefix=/opt/cross/aarch64-
linux --build=$MACHTYPE --host=aarch64-linux --target=aarch64-linux --with-
headers=/opt/cross/aarch64-linux/include --disable-multilib
libc_cv_forced_unwind=yes
make install-bootstrap-headers=yes install-headers
make -j4 csu/subdir_lib
install csu/crt1.o csu/crti.o csu/crtn.o /opt/cross/aarch64-linux/lib
aarch64-linux-gcc -nostdlib -nostartfiles -shared -x c /dev/null -o
/opt/cross/aarch64-linux/lib/libc.so
touch /opt/cross/aarch64-linux/include/gnu/stubs.h
cd ..
```

注：为确保交叉编译正确进行（保证兼容性），在configure时，由于系统将所有的warning都作error处理，不希望被警告中断编译，加入CFLAGS标识，`-O2`为二级优化，`-Wno-error`将所有警告转化为普通提示。

## 6. 编译器支持库：

```
cd build-gcc
make -j4 all-target-libgcc
make install-target-libgcc
cd ..
```

注：报错 `PATH_MAX` 未定义，找到发生error的位置 `gcc-`

`9.4.0/libsanitizer/asan/asan_linux.cc` 并为 `PATH_MAX` 赋值 `2048`

## 7. 标注C库：

```
cd build-glibc
make -j4
make install
cd ..
```

## 8. 标准C++库：

```
cd build-gcc
make -j4
make install
cd ..
```

## 2. 修改gcc代码，使得gcc -v 的输出中包含个人的信息。

1. 路径为：`gcc-9.4.0/gcc/gcc.c`。

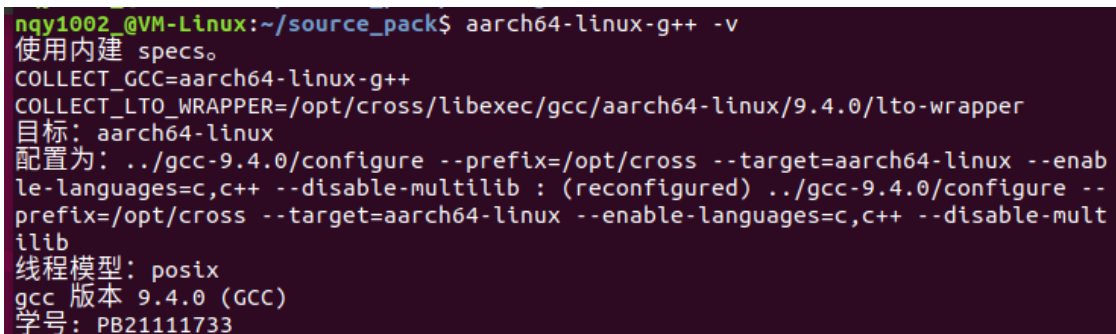
2. 其中函数 `print_configuration` 用于输出gcc的配置信息，在最后一行加入：

```
fnotice(file, "学号：PB21111733\n");
```

并保存文件。

3. `cd` 到 `build-gcc` 目录，重新执行一遍 1.4 中的内容。

4. 输出结果如图：



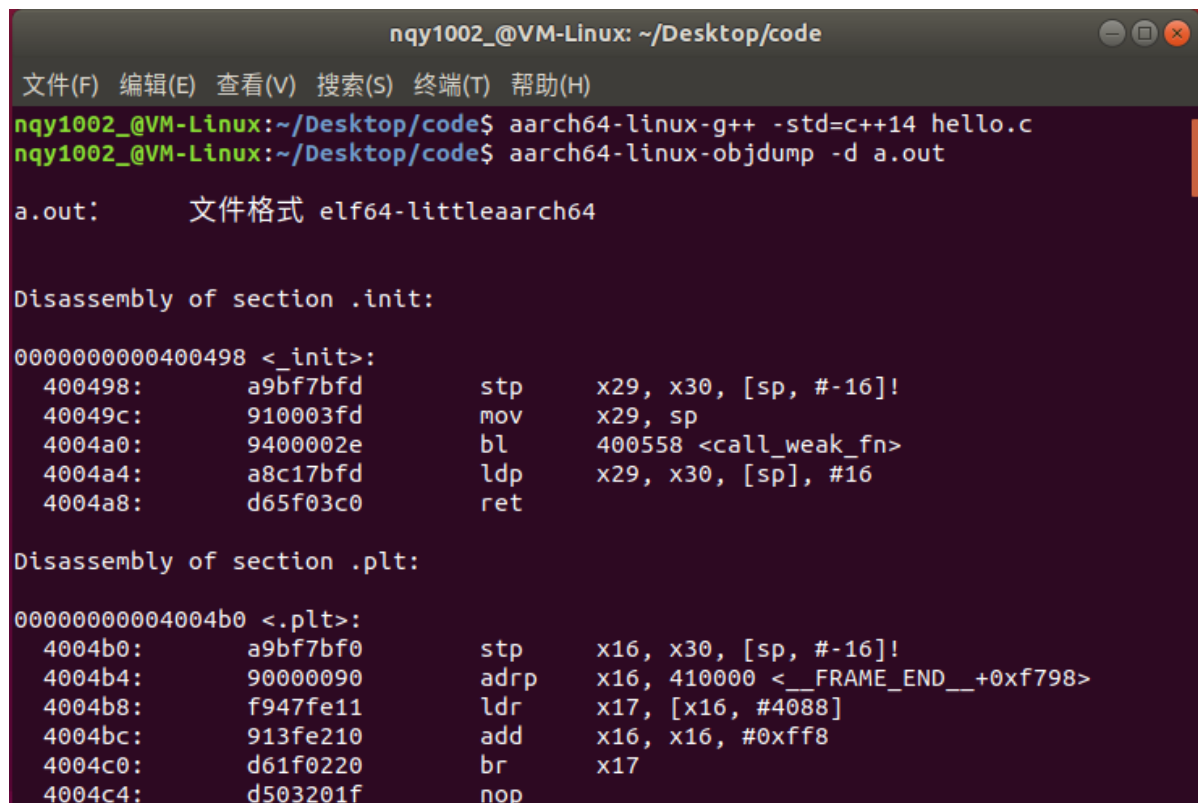
```
ngy1002 @VM-Linux:~/source_pack$ aarch64-linux-g++ -v
使用内建 specs。
COLLECT_GCC=aarch64-linux-g++
COLLECT_LTO_WRAPPER=/opt/cross/libexec/gcc/aarch64-linux/9.4.0/lto-wrapper
目标：aarch64-linux
配置为：../gcc-9.4.0/configure --prefix=/opt/cross --target=aarch64-linux --enable-languages=c,c++ --disable-multilib : (reconfigured) ../gcc-9.4.0/configure --prefix=/opt/cross --target=aarch64-linux --enable-languages=c,c++ --disable-multilib
线程模型：posix
gcc 版本 9.4.0 (GCC)
学号：PB21111733
```

### 3. 使用C代码测试编译器。

编写C代码如下：

```
//hello.c
#include <stdio.h>
int main()
{
    int i = 0, j = 1;
    for(i = 0; i < 10; i++)
    {
        j += i;
    }
    printf("%d", j);
    return 0;
}
```

测试汇编器：



The terminal window shows the following commands and output:

```
nqy1002_@VM-Linux: ~/Desktop/code
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
nqy1002_@VM-Linux:~/Desktop/code$ aarch64-linux-g++ -std=c++14 hello.c
nqy1002_@VM-Linux:~/Desktop/code$ aarch64-linux-objdump -d a.out

a.out:          文件格式 elf64-littleaarch64

Disassembly of section .init:

0000000000400498 <.init>:
400498:      a9bf7bfd      stp     x29, x30, [sp, #-16]!
40049c:      910003fd      mov     x29, sp
4004a0:      9400002e      bl      400558 <call_weak_fn>
4004a4:      a8c17bfd      ldp     x29, x30, [sp], #16
4004a8:      d65f03c0      ret

Disassembly of section .plt:

00000000004004b0 <.plt>:
4004b0:      a9bf7bf0      stp     x16, x30, [sp, #-16]!
4004b4:      90000090      adrp    x16, 410000 <__FRAME_END__+0xf798>
4004b8:      f947fe11      ldr     x17, [x16, #4088]
4004bc:      913fe210      add     x16, x16, #0xff8
4004c0:      d61f0220      br      x17
4004c4:      d503201f      nop
```

可以看到顺利反汇编。

### 其他事项：

在终端进行测试时，会出现 `aarch64-linux-g++: 未找到命令` 的问题，是环境变量没有持久化导致的。

解决方法为：

- 编辑 `~/.bashrc` 文件：

```
nano ~/.bashrc
```

- 添加 `PATH` 到文件末尾：

```
export PATH=/opt/cross/bin:$PATH
```

- 保存, 退出:

```
source ~/.bashrc
```

- 重启虚拟机后问题解决。

## 在linux平台上面安装arm-linux- 工具链:

注: 安装了arm-linux-gcc-4.6.4

### 编写C语言程序如下 (同上) :

```
//hello.c
#include <stdio.h>
int main()
{
    int i = 0, j = 1;
    for(i = 0; i < 10; i++)
    {
        j += i;
    }
    printf("%d", j);
    return 0;
}
```

使用gcc和arm-gcc编译, 比较生成的目标代码的区别。需要使用readelf和objdump等工具。

编译:

```
arm-linux-gcc hello.c -o pp
arm-linux-gcc -c hello.c -o pp.o
gcc hello.c -o gccpp
```

#### 1. 查看ELF文件头信息:

```
readelf -h gccpp > gccpp_header.txt
readelf -h pp > pp_header.txt
diff gccpp_header.txt pp_header.txt
```

结果如下:

```

nqy1002_@VM-Linux:~/Desktop/code$ diff gccpp_header.txt pp_header.txt
2,3c2,3
<  Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
<  类别:                                     ELF64
---
>  Magic:      7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
>  类别:                                     ELF32
8,9c8,9
<  类型:                                     DYN (共享目标文件)
<  系统架构:                                   Advanced Micro Devices X86-64
---
>  类型:                                     EXEC (可执行文件)
>  系统架构:                                   ARM
11,18c11,18
<  入口点地址:                                0x540
<  程序头起点:                                64 (bytes into file)
<  Start of section headers:                  6448 (bytes into file)
<  标志:                                       0x0
<  本头的大小:                                64 (字节)
<  程序头大小:                                56 (字节)
<  Number of program headers:                  9
<  节头大小:                                  64 (字节)
---
>  入口点地址:                                0x82c8
>  程序头起点:                                52 (bytes into file)
>  Start of section headers:                  1768 (bytes into file)
>  标志:                                       0x50000002, Version5 EABI, <unknown>
>  本头的大小:                                52 (字节)
>  程序头大小:                                32 (字节)
>  Number of program headers:                  8
>  节头大小:                                  40 (字节)
20c20
<  字符串表索引节头:  28
---
>  字符串表索引节头:  26

```

发现两者的ELF文件头信息表明在**类别**，**类型**，**系统架构**，**入口点地址**等方面都有很大差异。

	类别	类型	系统架构	入口点地址
gcc	ELF64	DYN	X86-64	0x540
arm-gcc	ELF32	EXEC	ARM	0x82c8

## 2. 查看程序段信息：

```

readelf -l gccpp > gccpp_segments.txt
readelf -l pp > pp_segments.txt
diff gccpp_segments.txt pp_segments.txt

```

结果如下：

```

may1002_@VM-Linux:~/Desktop/code$ diff gccpp_sections.txt pp_sections.txt
2,4c2,4
< elf 文件类型为 DYN (共享目标文件)
< Entry point 0x540
< There are 9 program headers, starting at offset 64
--
> elf 文件类型为 EXEC (可执行文件)
> Entry point 0x82c8
> There are 8 program headers, starting at offset 52
7,27c7,10
< Type Offset VirtAddr PhysAddr
< FileSiz MemSiz
< PHDR 0x0000000000000040 0x0000000000000040 0x0000000000000040
< 0x00000000000001f8 0x00000000000001f8 R 0x8
< INTERP 0x0000000000000238 0x0000000000000238 0x0000000000000238
< 0x000000000000001c 0x000000000000001c R 0x1
< [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
< LOAD 0x0000000000000000 0x0000000000000000 0x0000000000000000
< 0x0000000000000870 0x0000000000000870 R E 0x200000
< LOAD 0x0000000000000db8 0x0000000000000db8 0x0000000000000db8
< 0x000000000000258 0x000000000000260 RW 0x200000
< DYNAMIC 0x0000000000000dc8 0x00000000000020dc8 0x00000000000020dc8
< 0x00000000000001f0 0x00000000000001f0 RW 0x8
< NOTE 0x0000000000000254 0x0000000000000254 0x0000000000000254
< 0x0000000000000044 0x0000000000000044 R 0x4
< GNU_EH_FRAME 0x0000000000000728 0x0000000000000728 0x0000000000000728
< 0x000000000000003c 0x000000000000003c R 0x4
< GNU_STACK 0x0000000000000000 0x0000000000000000 0x0000000000000000
< 0x0000000000000000 0x0000000000000000 RW 0x10
< GNU_RELRO 0x0000000000000db8 0x00000000000020db8 0x00000000000020db8
< 0x0000000000000248 0x0000000000000248 R 0x1
--
< Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
< EXIDX 0x00045c 0x000845c 0x000845c 0x0008 0x0008 R 0x4
< PHDR 0x000034 0x0008034 0x0008034 0x00100 0x00100 R E 0x4
< INTERP 0x000134 0x0008134 0x00013 0x00013 R 0x1
< [Requesting program interpreter: /lib/ld-linux.so.3]
< LOAD 0x000000 0x0008000 0x0008000 0x00468 0x00468 R E 0x8000
< LOAD 0x000468 0x0010468 0x0010468 0x0011c 0x00120 RW 0x8000
< DYNAMIC 0x000474 0x0010474 0x00048 0x00048 RW 0x4
< NOTE 0x000148 0x0008148 0x0008148 0x00020 0x00020 R 0x4
< GNU_STACK 0x000000 0x0008000 0x0008000 0x00000 0x00000 RW 0x4
--
31,37c20,20
< 00
< 01 .interp
< 02 .interp.note.ABI-tag.note.gnu.build-id.gnu.hash.dynsym.dynstr.gnu.version.gnu.version_r.rela.dyn.rela.plt.init.plt.plt.got.text.fini.rodata.eh_frame_hdr.eh_frame
< 03 .init_array.fini_array.dynamic.got.data.bss
< 04 .dynamic
< 05 .note.ABI-tag.note.gnu.build-id
< 06 .eh_frame_hdr
--
00 .ARM.exidx
01
02 .interp
03 .interp.note.ABI-tag.hash.dynsym.dynstr.gnu.version.gnu.version_r.rel.dyn.rel.plt.init.plt.text.fini.rodata.ARM.exidx.eh_frame
04 .init_array.fini_array.jcr.dynamic.got.data.bss
05 .dynamic
06 .note.ABI-tag
39d27
08 .init_array.fini_array.dynamic.got

```

通过对比也能发现两者在Type, Offset和Vaddr, Size和Align等方面的不同 (详见上图)。

### 3. 查看段表信息:

```

readelf -S gccpp > gccpp_sections.txt
readelf -S pp > pp_sections.txt
diff gccpp_sections.txt pp_sections.txt

```

结果如下:

gcc:

[号]	名称	类型	地址	偏移量
	大小	全体大小	旗标 链接 信息	对齐
[ 0]	0000000000000000	NULL	0000000000000000	00000000
[ 1]	.interp	PROGBITS	0000000000000238	00000238
	000000000000001c	0000000000000000	A 0 0	1
[ 2]	.note.ABI-tag	NOTE	0000000000000254	00000254
	0000000000000020	0000000000000000	A 0 0	4
[ 3]	.note.gnu.build-id	NOTE	0000000000000274	00000274
	0000000000000024	0000000000000000	A 0 0	4
[ 4]	.gnu.hash	GNU_HASH	0000000000000298	00000298
	000000000000001c	0000000000000000	A 5 0	8
[ 5]	.dynsym	DYSYM	00000000000002b8	000002b8
	00000000000000a8	0000000000000018	A 6 1	8
[ 6]	.dynstr	STRTAB	0000000000000360	00000360
	0000000000000084	0000000000000000	A 0 0	1
[ 7]	.gnu.version	VERSYM	00000000000003e4	000003e4
	000000000000000e	0000000000000002	A 5 0	2
[ 8]	.gnu.version_r	VERNEED	00000000000003f8	000003f8
	0000000000000020	0000000000000000	A 6 1	8
[ 9]	.rela.dyn	RELA	0000000000000418	00000418
	00000000000000c0	0000000000000010	A 5 0	8

arm-gcc:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	00000000	000000	000000	00		0	0	0
[ 1]	.interp	PROGBITS	00008134	000134	000013	00	A	0	0	1
[ 2]	.note.ABI-tag	NOTE	00008148	000148	000020	00	A	0	0	4
[ 3]	.hash	HASH	00008168	000168	000028	04	A	4	0	4
[ 4]	.dynsym	DYNSYM	00008190	000190	000050	10	A	5	1	4
[ 5]	.dynstr	STRTAB	000081e0	0001e0	000043	00	A	0	0	1
[ 6]	.gnu.version	VERSYM	00008224	000224	00000a	02	A	4	0	2
[ 7]	.gnu.version_r	VERNEED	00008230	000230	000020	00	A	5	1	4
[ 8]	.rel.dyn	REL	00008250	000250	000008	08	A	4	0	4
[ 9]	.rel.plt	REL	00008258	000258	000020	08	A	4	11	4
[10]	.init	PROGBITS	00008278	000278	00000c	00	AX	0	0	4
[11]	.plt	PROGBITS	00008284	000284	000044	04	AX	0	0	4
[12]	.text	PROGBITS	000082c8	0002c8	000184	00	AX	0	0	4
[13]	.fini	PROGBITS	0000844c	00044c	000008	00	AX	0	0	4
[14]	.rodata	PROGBITS	00008454	000454	000008	00	A	0	0	4
[15]	.ARM.exidx	ARM_EXIDX	0000845c	00045c	000008	00	AI	12	0	4

可以发现他们在段的名称、大小、对齐方式等方面有所不同。

#### 4. 反汇编二进制代码：

```
objdump -d gccpp > gccpp_disasm.txt
arm-linux-objdump -d pp.o > pp_disasm.txt
diff gccpp_disasm.txt pp_disasm.txt
```

对比得出两者指令集、指令顺序以及处理方式等方面的区别（比较显然的是，gcc汇编了例如 `<_init>` 在内的很多内容，而arm-gcc只有 `<main>`），如图：

gcc：

```

gccpp:      文件格式 elf64-x86-64

Disassembly of section .init:

00000000000004f0 <_init>:
4f0:  48 83 ec 08      sub    $0x8,%rsp
4f4:  48 8b 05 ed 0a 20 00  mov    0x200aed(%rip),%rax      # 200fe8 <__gmon_start__>
4fb:  48 85 c0          test   %rax,%rax
4fe:  74 02            je     502 <_init+0x12>
500:  ff d0            callq  *%rax
502:  48 83 c4 08      add    $0x8,%rsp
506:  c3              retq

Disassembly of section .plt:

0000000000000510 <.plt>:
510:  ff 35 aa 0a 20 00  pushq  0x200aaa(%rip)           # 200fc0 <_GLOBAL_OFFSET_TABLE_+0x8>
516:  ff 25 ac 0a 20 00  jmpq   *0x200aac(%rip)         # 200fc8 <_GLOBAL_OFFSET_TABLE_+0x10>
51c:  0f 1f 40 00      nopl   0x0(%rax)

0000000000000520 <printf@plt>:
520:  ff 25 aa 0a 20 00  jmpq   *0x200aaa(%rip)         # 200fd0 <printf@GLIBC_2.2.5>
526:  68 00 00 00 00    pushq  $0x0
52b:  e9 e0 ff ff ff   jmpq   510 <.plt>

Disassembly of section .plt.got:

0000000000000530 <__cxa_finalize@plt>:
530:  ff 25 c2 0a 20 00  jmpq   *0x200ac2(%rip)         # 200ff8 <__cxa_finalize@GLIBC_2.2.5>
536:  66 90            xchg   %ax,%ax

Disassembly of section .text:

0000000000000540 <_start>:
540:  31 ed            xor     %ebp,%ebp

```

arm-gcc：



打开(O) pp\_disasm.txt 保存(S)

```
pp.o:      file format elf32-littlearm

Disassembly of section .text:

00000000 <main>:
 0: e92d4800      push    {fp, lr}
 4: e28db004      add     fp, sp, #4
 8: e24dd008      sub     sp, sp, #8
 c: e3a03000      mov     r3, #0
10: e50b3008      str     r3, [fp, #-8]
14: e3a03001      mov     r3, #1
18: e50b300c      str     r3, [fp, #-12]
1c: e3a03000      mov     r3, #0
20: e50b3008      str     r3, [fp, #-8]
24: ea000006      b       44 <main+0x44>
28: e51b200c      ldr     r2, [fp, #-12]
2c: e51b3008      ldr     r3, [fp, #-8]
30: e0823003      add     r3, r2, r3
34: e50b300c      str     r3, [fp, #-12]
38: e51b3008      ldr     r3, [fp, #-8]
3c: e2833001      add     r3, r3, #1
40: e50b3008      str     r3, [fp, #-8]
44: e51b3008      ldr     r3, [fp, #-8]
48: e3530009      cmp     r3, #9
4c: daffffff5     ble     28 <main+0x28>
50: e59f3018      ldr     r3, [pc, #24] ; 70 <main+0x70>
54: e1a00003      mov     r0, r3
58: e51b100c      ldr     r1, [fp, #-12]
5c: ebffffffe     bl      0 <printf>
60: e3a03000      mov     r3, #0
64: e1a00003      mov     r0, r3
68: e24bd004      sub     sp, fp, #4
6c: e8bd8800      pop     {fp, pc}
70: 00000000      .word   0x00000000
```

纯文本 制表符宽度: 8 第 1 行, 第 1 列 插入

## 其他事项:

实验后的文件目录:

