练习 1

1. 笔记见文件“AISRG 阅读笔记”。
2. ARM 和 x86-64 的差别：

	RISC (AArch64)	CISC (x86-64)
指令长度	定长	变长
寻址模式	寻址方式单一	多种寻址方式
内存操作	load/store	mov
实现	增加通用寄存器数量	微码
指令复杂度	简单	复杂
汇编复杂度	复杂	简单
中断响应	快	慢
功耗	低	高
处理器结构	简单	复杂

练习 2

使用 GDB 的 where 命令后得到如下输出：

```
#0  0x0000000000080000 in _start ()
Backtrace stopped: not enough registers or memory available to unwind further
```

所以入口的地址是 0x0000000000080000。

练习 3

（1）首先通过以下命令看到 build/kernel.img 文件完整的 ELF 头部信息：

```
readelf -h build/kernel.img
```

其中有这样一行：

```
Entry point address:                0x80000
```

入口地址就是函数_start ()的地址，运行以下命令在当前目录下搜索文件内容：

```
grep -rn "_start" *
```

最终发现这个函数定义在文件 boot/start.S 中。

（2）_start 函数的汇编代码如下：

```
BEGIN_FUNC(_start)
    mrs  x8, mpidr_el1
    and  x8, x8,    #0xFF
    cbz  x8, primary
secondary_hang:
    bl secondary_hang
primary: …
END_FUNC(_start)
```

阅读汇编代码，_start 函数首先将 mpidr_el1 这个系统寄存器的值放到 x8 寄存器中，再将 x8 的值和 0xFF 进行按位与运算。这些操作实际上是判断当前处理器的 id 是否是 0，如果是 0，则跳到 primary 标签正常执行，否则跳到 secondary_hang 标签来挂起该控制流。

练习 4

运行下面的命令：

```
readelf -h build/kernel.img
```

得到如下结果：

```
build/kernel.img:     file format elf64-little
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 init          0000b5b0  0000000000080000  0000000000080000  00010000  2**12
                  CONTENTS, ALLOC, LOAD, CODE
  1 .text         000011dc  ffffff000008c000  000000000008c000  0001c000  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .rodata       000000f8  ffffff0000090000  0000000000090000  00020000  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .bss          00008000  ffffff0000090100  0000000000090100  000200f8  2**4
                  ALLOC
  4 .comment      00000032  0000000000000000  0000000000000000  000200f8  2**0
                  CONTENTS, READONLY
```

从结果中可以发现，init 段的 VMA 和 LMA 相同，其余段的 VMA 和 LMA 不同，差值都为 0xffffff0000000000，在这里我们不考虑.comment 段，因为它是不需要执行的无用段。Bootloader 位于.init 段，它开始运行时页表尚未初始化，不支持虚拟内存，所以 VMA 就是 LMA。Kernel 位于.text 段，在 bootloader 运行过程中，页表被初始化并开启了 MMU，kernel 代码被映射到低地址段（LMA）和高地址段（VMA）两份，所以 VMA 和 LMA 就不同了。

练习 5

简单的进制转换算法，不再赘述。

练习 6

注意到在_start 函数的 primary 标签下有这样的汇编代码：

```
/* Prepare stack pointer and jump to C. */
      adr   x0, boot_cpu_stack
      add  x0, x0, #0x1000
      mov sp, x0
```

这几行汇编代码的目标是把 boot_cpu_stack 的地址加上 4096 存进 sp 寄存器中，这样就完成了内核栈的初始化。所以**内核栈初始化的代码位于_start 函数**。内核栈实际上就是 boot_cpu_stack 所定义的二维数组：

```
char boot_cpu_stack[PLAT_CPU_NUMBER][INIT_STACK_SIZE] ALIGN(16);
```

这是一个二维数组，每一维对应一个 CPU。内核为每个栈保留 4096 个字节的空间。

练习 7

首先在 stack_test 处设置断点：

```
b stack_test
```

持续 continue 下去，每次打印栈的情况：

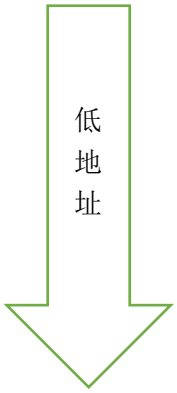
```
(gdb) x/10ag $x29
0xffffff00000920c0 <kernel_stack+8128>:    0xffffff00000920e0                    <kernel_stack+8160>
    0xffffff000008c0c0 <main+76>
0xffffff00000920d0 <kernel_stack+8144>:    0x0    0xffffffc0
0xffffff00000920e0 <kernel_stack+8160>:    0x887e0 <boot_cpu_stack+4080> 0xffffff000008c018
0xffffff00000920f0 <kernel_stack+8176>:    0x0    0x873c8 <secondary_boot_flag>
0xffffff0000092100 <kernel_stack+8192>:    0x0    0x0
(gdb) x/10ag $x29
0xffffff00000920a0 <kernel_stack+8096>:    0xffffff00000920c0                    <kernel_stack+8128>
    0xffffff000008c070 <stack_test+80>
0xffffff00000920b0 <kernel_stack+8112>:    0x5    0xffffffc0
0xffffff00000920c0 <kernel_stack+8128>:    0xffffff00000920e0                    <kernel_stack+8160>
    0xffffff000008c0c0 <main+76>
0xffffff00000920d0 <kernel_stack+8144>:    0x0    0xffffffc0
0xffffff00000920e0 <kernel_stack+8160>:    0x887e0 <boot_cpu_stack+4080> 0xffffff000008c018
(gdb) x/10ag $x29
0xffffff0000092080 <kernel_stack+8064>:    0xffffff00000920a0                    <kernel_stack+8096>
    0xffffff000008c070 <stack_test+80>
0xffffff0000092090 <kernel_stack+8080>:    0x4    0xffffffc0
0xffffff00000920a0 <kernel_stack+8096>:    0xffffff00000920c0                    <kernel_stack+8128>
    0xffffff000008c070 <stack_test+80>
0xffffff00000920b0 <kernel_stack+8112>:    0x5    0xffffffc0
0xffffff00000920c0 <kernel_stack+8128>:    0xffffff00000920e0                    <kernel_stack+8160>
    0xffffff000008c0c0 <main+76>
(gdb) x/10ag $x29
0xffffff0000092060 <kernel_stack+8032>:    0xffffff0000092080                    <kernel_stack+8064>
    0xffffff000008c070 <stack_test+80>
0xffffff0000092070 <kernel_stack+8048>:    0x3    0xffffffc0
0xffffff0000092080 <kernel_stack+8064>:    0xffffff00000920a0                    <kernel_stack+8096>
    0xffffff000008c070 <stack_test+80>
0xffffff0000092090 <kernel_stack+8080>:    0x4    0xffffffc0
0xffffff00000920a0 <kernel_stack+8096>:    0xffffff00000920c0                    <kernel_stack+8128>
    0xffffff000008c070 <stack_test+80>
(gdb) x/10ag $x29
0xffffff0000092040 <kernel_stack+8000>:    0xffffff0000092060                    <kernel_stack+8032>
    0xffffff000008c070 <stack_test+80>
0xffffff0000092050 <kernel_stack+8016>:    0x2    0xffffffc0
0xffffff0000092060 <kernel_stack+8032>:    0xffffff0000092080                    <kernel_stack+8064>
    0xffffff000008c070 <stack_test+80>
0xffffff0000092070 <kernel_stack+8048>:    0x3    0xffffffc0
0xffffff0000092080 <kernel_stack+8064>:    0xffffff00000920a0                    <kernel_stack+8096>
    0xffffff000008c070 <stack_test+80>
(gdb) x/10ag $x29
0xffffff0000092020 <kernel_stack+7968>:    0xffffff0000092040                    <kernel_stack+8000>
    0xffffff000008c070 <stack_test+80>
0xffffff0000092030 <kernel_stack+7984>:    0x1    0xffffffc0
0xffffff0000092040 <kernel_stack+8000>:    0xffffff0000092060                    <kernel_stack+8032>
    0xffffff000008c070 <stack_test+80>
0xffffff0000092050 <kernel_stack+8016>:    0x2    0xffffffc0
0xffffff0000092060 <kernel_stack+8032>:    0xffffff0000092080                    <kernel_stack+8064>
    0xffffff000008c070 <stack_test+80>
```

从六次栈的状态可以发现，每个 stack_test 递归嵌套级别将 4 个 64 位值压入堆栈，FP 处存的是调用者函数的 FP 寄存器的值，FP+8 处存的是该函数的 LR 值，FP-8 处存的值都是 0xffffffc0，FP-16 处存的是该函数的参数。

练习 8

根据练习 8 中的分析，栈结构如下：

……
参数
其他数据
调用者函数的 FP 寄存器的值
该函数的 LR 值
……
参数
其他数据
调用者函数的 FP 寄存器的值
该函数的 LR 值
……



练习 9

我的代码如下：

```
long long* fp = (long long*)(*(long long*)read_fp());
while(*(fp) != 0){
        printk("LR %lx FP %lx Args ", *(fp+1), fp);
        long long* ap = fp - 2;
        int i;
        for(i = 0; i < 5; i++){
                printk("%d ", *ap);
                ap--;
        }
        printk("\n");
        fp = (long long*) *fp;
}
```

read_fp()函数返回的是调用它的函数的 FP 的值，而题目要求打印的是调用调用 read_fp()函数的函数的 LR，FP 和 Args，所以要首先定位到调用者的 FP 的值。上面已经分析过，FP+8 处存的是该函数的 LR 值，FP-16 处存的是该函数的参数，题目只要求列出前五个参数，所以只需要循环 5 次。如此递归直到调用者不存在为止。