实验一 基于华为鲲鹏云服务器的 ARM 开发

一、 实验目的

1. 掌握 GNU ARM 平台汇编代码的编写以及编译运行方式以及基本的循环分支操作；
2. 掌握在 ARM 平台上通过 C 语言源码来调用汇编源码的方法；
3. 掌握在 ARM 平台上实现 C 语言代码中内嵌汇编代码的方法；
4. 掌握利用 Aarch64 架构下的提高汇编代码执行效率的方式；

二、 实验内容

1. 实现 ARM 平台精简指令集（RISC）编写的 hello-world 程序的编译和运行；
2. 在华为 CloudIDE 开发平台上实现 hello-world 程序的编译和运行；
3. 实现 ARM 平台上通过 C 语言源码来调用汇编源码中的代码；
4. 实现在 ARM 平台上通过 C 语言代码内嵌汇编代码的方式，将一个整数类型值，以字节为单位从小尾端转到大尾端或者相反的功能；
5. 实现 GNU ARM 汇编中如何利用 Aarch64 架构“其访存单元支持每拍 2 条读或写访存指令”的特性，来提升改进代码，提高代码执行效率；
6. 实现通过 GNU 标准的 C 语言和 ARM 汇编代码，在 ARM 平台上用加密指令实现 SHA256 算法；
7. 实现通过构建汇编代码实现基于 ARM 平台的精简指令集（RISC）的字母提升程序的编译和运行。实现将一串连续的字符串输入后，对输入字符串中出现的数字，大写字母，小写字母进行统计。

三、 实验原理

- hello-world 示例程序

在本例子中，两次使用软中断指令 svc 来进行系统调用，系统调用号通过 x8 寄存器传递。在第一次使用 svc 指令来在屏幕上打印一个字符串“Hello”：x0 寄存器用于存放标准屏幕输出 stdout 描述符 0，表明将向屏幕输出一些内容；x1 寄存器用于存放待输出的字符串的首地址 msg；x2 寄存器用于存放待输出字符串的长度 len；x8 寄存器用于存放系统功能调用号 64，即 64 号系统功能即系统写功能 sys_write()，写的目标在 x0 中定义；svc #0 表示是一个系统功能调用。

第二次使用 svc 指令来退出当前程序：x0 寄存器用于存放退出操作码 123，不同的退出操作码将对应不同的退出操作；x8 寄存器用于存放系统功能调用号 93，即 93 号系统功能即系统退出功能 sys_exit()，退出操作码在 x0 中定义；svc #0 表示是一个系统功能调用。

注意：像这种系统功能调用的方式和功能号，都是基于 Arm64 处理器体系结构以及之上所运行的 linux kernel 甚至 BIOS 来共同支持，而不仅仅是 Arm64 架构自身所能完成的。

在.data 部分，加载 msg 和 len 实际上使用的是文字池的方法，即将变量地址放在代码段中不会执行到的位置（因为第二次使用 svc 指令来退出当前程序之后，是不可能将 svc #0 指令之后的内容来当做指令加以执行的），使用时先加载变量的地址，然后通过变量的地址得到变量的值。

本代码是 Aarch64 体系结构的汇编代码，需要在 ArmV8 处理器上运行。寄

存器 Xn 都是 Aarch64 体系结构中的寄存器，svc 是 Aarch64 体系结构中的指令。

- 使用 C 语言代码调用汇编程序

该汇编代码是针对 Aarch64 架构的。在汇编程序中，用.global 定义一个全局函数 strcpy1，然后该函数就可以在 C 代码中用 extern 关键字加以声明，然后直接调用。

- 使用 C 语言代码内嵌汇编程序

通过 C 语言代码内嵌汇编代码，将一个整数类型值，以字节为单位从小尾端转到大尾端或者相反的功能。例如小尾端时 32bit 整数值用 16 进制表示为 0x12345678，将其以字节为单位转换为大尾端存储后，该值为 0x78563412。

- 利用鲲鹏处理器的流水线来优化汇编代码性能实验

关于在 C 代码和汇编代码之间进行参数传递，根据 Arm 公司的 AAPCS64，即 Aarch64 程序调用标准，Aarch64 标准提供了 8 个通用寄存器（x0-x7）用于传递函数参数，依次对应于参数 0、参数 1、参数 2…参数 7。第 8 个参数需要通过 sp 访问，第 9 个参数需要通过 sp + 8 访问，第 n 个参数需要通过 sp + 8*(n-8)访问。一般来说，对于只带有少量参数的函数，仅使用寄存器就足够了；超过 8 个的参数会存放在堆栈中用于传递给子例程。在本例子中，需要传递的参数有三个：第一个参数是目标字符串的首地址，用寄存器 X0 来传递；第二个参数是源字符串的首地址，用寄存器 X1 来传递；第三个参数是传输的字节数目，用寄存器 X2 来传递。

在使用 ldrb/ldp 和 str/stp 等访存指令时，要注意区分这三种形式：

- 前索引方式，形如：ldrb w2,[X1,#1] //将 x1+1 指向的地址处的一个字节放入 w2 中；x1 寄存器的值保持不变。
- 自动索引方式，形如：ldrb w2,[X1,#1]！ //将 x1+1 指向的地址处的一个字节放入 w2 中；然后 x1+1->x1。
- 后索引方式，形如 ldrb w2,[X1],#1 //将 x1 指向的地址处的一个字节放入 w2 中，然后 x1+1->x1。

该程序由两部分组成：第一部分是主函数，采用 Linux C 语言编码，用来测试内存拷贝函数的执行时间；第二部分是内存拷贝函数，采用 GNU Arm64 汇编语言编码。在下面的代码中，用到了上面三种形式的指令，需仔细体会其不同。

为了较为准确的测量内存拷贝函数 memorycopy()的执行时间，调用了 clock_gettime()来分别记录 memorycopy()执行前和执行后的系统时间，以纳秒为计时单位。

- 密码运算实验

本实验会先简单介绍一下密码运算的专用汇编指令集以及 SHA256 的算法，然后通过编写伪代码的方式，更好的理解算法的原理。最后完成在 ARM 平台上用 C 语言和汇编代码加密指令实现 SHA256 算法的实验。

- 字符数字统计实验

在本次实验中，我们需要输入一段字符串，之后通过遍历整个字符串得到字符串中共有多少的整数，多少的大写字母，多少的小写字母。

在本例子中，首先申请一段连续的空间用来存储待读取的字符串用以后续处理，该段空间以 mystring 为首地址，之后使用 ptr 作为遍历该段时所用的地址索引，并初始化为 mystring。然后我们还要定义用来输入和输出操作的静态格式，input 和 output，该格式用于调用 C 语言的接口 scanf 和 printf。除此之外我们还需要定义计数器用来存储，目标类型字符的出现次数。这里定义了 numCnt 表示

数字出现次数，upCnt 表示大写字母出现次数，lowCnt 表示小写字母出现次数。

在进入到代码段之后，我们首先将调用该函数的地址通过 stp 保存起来，这里是一个入栈的操作。__isoc99_scanf 通过寄存器传参，将读取格式 input_format 存入 x0 寄存器，将读取地址 str 存入 x1 寄存器，最后使用 bl __isoc99_scanf 进行函数的调用。在准备工作完成之后进入循环。

首先跳转到.L2 的部分，该部分负责检测字符串是否执行到末尾，将 ptr 内地址指向的内容取出使用 ldrb 存入 w0, 对比 w0 是否等于 0，如果等于则代表结束进行结果输出，将结果分别按数组，大写，小写的顺序存入 w1，w2，w3 并将输出格式存入 x0，之后通过 bl printf

调用输出函数。如果不等于 0 则代表字符串未结束，则跳转到.L6。

.L6 主要的功能在于判断字符是否是数字，也就是说 ASCII 的值是否在[48, 57]的区间内，如果在则读取 numCnt 当前的值，使其增加 1 并存入 numCnt。最后跳到.L4。如果不在则跳到.L3。

.L3 的主要功能是判断当前字符是否是一个小写字符，即判断是否在[97,122]的区间范围内，如果是则将 lowCnt 增加 1，并跳转到.L4 更新索引。如果不在则跳转到.L5。

.L5 的主要功能是判断当前字符是否是一个小写字符，即判断是否在[65,90]的区间范围内，如果是则将 upCnt 增加 1，否则不进行任何操作开始进行.L4。

.L4 的主要功能在于将索引 ptr 进行更新，使其指向新的操作字符。

四、 实验步骤

1. hello-world 示例程序

以下步骤以在华为鲲鹏云服务器上执行为例。

- 创建 hello 目录

执行以下命令，创建 hello 目录，存放该程序的所有文件, 并进入 hello 目录

```
mkdir hello
cd hello
```

- 创建示例程序源码 hello.s

执行以下命令，创建示例程序源码 hello.s

```
vim hello.s
```

代码内容如下：

```
.text
.global tart1
tart1:
    mov x0,#0
    ldr x1,=msg
    mov x2,len
    mov x8,64
    svc #0

    mov x0,123
    mov x8,93
    svc #0

.data
```

```
msg:
    .ascii "Hello World!\n"
len=.-msg
```

- 进行编译运行

保存示例源码文件，然后退出 vim 编辑器。在当前目录中依次执行以下命令，进行代码编译运行。

```
as hello.s –o hello.o
ld hello.o –o hello
./hello
```

2. 使用 C 语言代码调用汇编程序

以下步骤以在华为鲲鹏云服务器上执行为例。

- 创建目录

执行以下命令，创建 called 目录存放该程序的所有文件，并进入 called 目录

```
mkdir called
cd called
```

- 创建 globalCalling.c 源代码

执行以下命令，创建示例调用 C 语言程序源码 globalCalling.c。

```
vim globalCalling.c
```

代码内容如下：

```
/* globalCalling.c*/
#include <stdio.h>
extern void strcpy1(char *d, const char *s);
int main()
{
      const char *srcstring="Source string";
      char dststring[]="Destination string";

      printf("Original Status: %s      %s\n",srcstring,dststring);
     strcpy1(dststring,srcstring);
     printf("Modified Status: %s      %s\n",srcstring,dststring);
      return 0;
}
```

- 创建 globalCalled.S 源代码

执行以下代码命令，创建被调用的汇编语言程序源码 globalCalled.S

```
vim globalCalled.S
```

代码内容如下：

```
/* globalCalled.S */
.global strcpy1

# Start the function: strcpy1
strcpy1:
LDRB w2,[X1],#1
STR w2,[X0],#1
CMP w2,#0      //ascii code "NUL" is the last character of a string;
BNE strcpy1
```

```
RET
```

- 进行编译运行

保存示例源码文件，然后退出 vim 编辑器。在当前目录中依次执行以下命令，进行代码编译运行。

```
gcc globalCalling.c globalCalled.S -o called
./called
```

3. 使用 C 语言代码内嵌汇编程序

以下步骤以在华为鲲鹏云服务器上执行为例。

- 创建目录

执行以下命令，创建 builtin 目录存放该程序的所有文件，并进入 bulitin 目录。

```
mkdir builtin
cd builtin
```

- 创建 C 语言内嵌汇编程序源代码

执行以下命令，创建 C 语言内嵌汇编程序源码 globalBuiltin.c。

```
vim globalBuiltin.c
```

代码内容如下：

```
/* globalBuiltin.c*/
#include <stdio.h>
int main()
{
int val=0x12345678;

  __asm__ __volatile__(
       "mov x3,%1\n"
     "mov w3,w3, ror #8\n"
     "bic w3,w3, #0x00ff00ff\n"

     "mov x4,%1\n"
     "mov w4,w4, ror #24\n"
     "bic w4,w4, #0xff00ff00\n"

     "add w3,w4,w3\n"
     "mov %0,x3\n"

     :"=r"(val)
     :"0"(val)
     :"w3","w4","cc"
     );

printf("out is %x \n",val);
return 0;
}
```

- 进行编译

保存示例源码文件，然后退出 vim 编辑器。在当前目录中依次执行以下命令，进行代码编译。

```
gcc -E globalBuiltin.c -o globalBuiltin.i
gcc -S globalBuiltin.i -o globalBuiltin.s
gcc -c globalBuiltin.s -o globalBuiltin.o
gcc globalBuiltin.o -o globalBuiltin
```

- 进行运行

  运行生成的 globalBuiltin 文件，查看输出结果。

  命令如下：

```
./globalBuiltin
```

4. 利用鲲鹏处理器的流水线来优化汇编代码性能实验

   以下步骤以在华为鲲鹏云服务器上执行为例。

- 执行以下命令，创建 memorycopy 目录存放该程序的所有文件，并进入该目录

```
cd
mkdir memorycopy
cd memorycopy
```

- 执行以下命令，创建主函数的 Linux C 代码 memorycopy.c

```
vim memorycopy.c
```

代码内容如下：

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

#define len 60000000
char src[len],dst[len];
long int len1=len;

extern void memorycopy(char *dst,char *src,long int len1);

int main()
{
      struct timespec t1,t2;
      int i,j;

for(i=0;i<len-1;i++)
{
src[i]='a';
}
src[i]=0;

clock_gettime(CLOCK_MONOTONIC,&t1);
memorycopy(dst,src,len1);
clock_gettime(CLOCK_MONOTONIC,&t2);

printf("memorycopy time is %11u ns\n",t2.tv_nsec-t1.tv_nsec);
      return 0;
}
```

- 执行以下命令，创建内存拷贝函数的 GNU Arm64 汇编语言编码 copyfunc.s

```
vim copyfunc.s
```

代码内容如下：

```
.global memorycopy

memorycopy:
   ldrb w3,[x1],#1
   str w3,[x0],#1
   sub x2,x2,#1
   cmp x2,#0
   bne memorycopy

ret
```

说明：内存拷贝函数 memorycopy()的功能是实现将尺寸为 len（这里设为 60000000）的 src 字符数组的内容拷贝到同样尺寸的 dst 字符数组中。memorycopy()函数用 Arm64/Aarch64 汇编代码实现。根据所用访存指令和循环展开粒度的不同，可以有多种实现方式。以上的汇编代码是最原始的方式，不进行循环展开，每次循环只使用 1 个 ldrb 和 1 个 str 指令。

- 进行原始程序的编译运行

保存示例源码文件，然后退出 vim 编辑器。在当前目录中依次执行以下命令，进行代码编译运行。

```
gcc copyfunc.s memorycopy.c -o m1
./m1
```

- 代码的第一阶段改进

采用循环展开的方法，充分利用流水线的多发射机制，对函数 memorycopy()原始汇编代码主体部分的两种改进，方式如下：

循环展开的宽度为 2。将该方法命名为 copyfunc_v2_1.s，汇编代码内容如下：

```
.global memorycopy

memorycopy:
  sub x1,x1,#1
  sub x0,x0,#1
lp:
  ldrb w3,[x1,#1]
  ldrb w4,[x1,#2]!
  str w3,[x0,#1]
  str w4,[x0,#2]!

  sub x2,x2,#2
  cmp x2,#0
  bne lp
ret
```

- 代码的第二阶段改进

第一次改进中中每次读/写内存都是以一个字节为单位进行的，其访存效率较低。可以采用一次读/写 16 个字节的方法，充分利用内存突发传输方式的优势（即内存在连续读/连续写多个数据时，其性能要优于非连续读/写数据的方式），对上一节的代码再次进行改进。Arm64/Aarch64 提供了 ldp 指令和 stp 指令，这两条指令可以一次访问 16 个字节的内存数据，其读/写内存的连续性非常高，可以有效降低访存延时。使用 ldp 和 stp 指令进行改进有如下三种典型的改进方式：

未经循环展开。将该方法命名为 copyfunc_v3_1.s，汇编代码内容如下：

```
.global memorycopy
memorycopy:
 ldp x3,x4,[x1],#16
 stp x3,x4,[x0],#16
 sub x2,x2,#16
 cmp x2,#0
 bne memorycopy

ret
```

5. 密码运算实验

实验说明：本实验通过 GNU 标准的 C 语言和汇编代码，使用加密指令来实现 SHA256 算法。通过对 README.md 文件的加密，验证了加密算法的作用。

以下步骤以在华为鲲鹏云服务器上执行为例。

- 创建目录

执行以下命令，创建 sha256-armv8 目录存放该程序的所有文件, 并进入 sha256-armv8 目录。

```
mkdir sha256-armv8
cd sha256-armv8
```

- 创建 sha256 算法源代码

执行以下命令，创建用加密指令实现 SHA256 算法的 C 源码 sha256.c。

```
vim sha256.c
```

代码内容如下：

```
/* sha256.c*/
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>

void sha256_block_data_order (uint32_t *ctx, const void *in, size_t num);

// SHA-256 initial hash value
const uint32_t H_0[8] = {
    0x6a09e667,
    0xbb67ae85,
```

```
    0x3c6ef372,
    ........
    sha256_block_data_order(H, buffer,1);

    // print hash
    sha256_print_hash(H, "Final Hash");

    // convert hash to char array (in correct order)
    for (i = 0; i < 8; i++) {
        buffer[i*4 + 0] = H[i] >> 24;
        buffer[i*4 + 1] = H[i] >> 16;
        buffer[i*4 + 2] = H[i] >>  8;
        buffer[i*4 + 3] = H[i];
    }

    // print hash
    printf("Hash:\t");
    for (i = 0; i < 32; i++) {
        printf("%02x", buffer[i]);
    }
    printf("\n");

    return 0;
}
```

- 创建 sha256-armv8-aarch32.S 源代码

  执行以下命令，创建 sha256-armv8-aarch32.S 源代码。

```
vim sha256-armv8-aarch32.S
```

  代码内容如下：

```
/* sha256-armv8-aarch32.S */
.text
.code   32

# SHA256 assembly implementation for ARMv8 AArch32

.global  sha256_block_data_order
.type    sha256_block_data_order,%function
.align   2
sha256_block_data_order:
........
```

- 创建 sha256-armv8-aarch64.S 源代码

  执行以下命令，创建 sha256-armv8-aarch64.S 源代码。

```
vim sha256-armv8-aarch64.S
```

  代码内容如下：

```
/* sha256-armv8-aarch64.S */
.text
.arch    armv8-a+crypto
```

```
# SHA256 assembly implementation for ARMv8 AArch64

.global  sha256_block_data_order
.type     sha256_block_data_order,%function
.align    2
sha256_block_data_order:

.Lsha256prolog:

      stp          x29, x30, [sp,#-64]!
      mov          x29, sp
      adr          x3, .LKConstant256
      str          q8, [sp, #16]
      ld1          {v16.4s-v19.4s}, [x3], #64
      ld1          {v0.4s}, [x0], #16
      ld1          {v20.4s-v23.4s}, [x3], #64
      add          x2, x1, x2, lsl #6
      ld1          {v1.4s}, [x0]
      ld1          {v24.4s-v27.4s}, [x3], #64
      sub          x0, x0, #16
      str          q9, [sp, #32]
      str          q10, [sp, #48]
      ld1          {v28.4s-v31.4s}, [x3], #64
```

- 创建 makefile

  执行以下命令，创建 Makefile。

```
vim Makefile
```

  代码内容如下：

```
CC = gcc
CFLAGS = -O3 -mcpu=generic+crypto
sha256:sha256.c
    $(CC) $(CFLAGS) sha256.c sha256-armv8-aarch64.S -o sha256
```

- 创建 README.md

  执行以下命令，创建 README.md。

```
vim README.md
```

  代码内容如下：

```
# SHA-256

This is a very basic implementation of a SHA-256 hash according to the [FIPS
180-4 standard](http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf)
in C. I did it for educational purposes, the code is not optimized at all, but
(almost) corresponds line by line to the standard.

The algorithm to process a small block of data is quite simple and very well
described in the standard. I found that correctly implementing the padding is
much harder.
```

```
It does not have any dependencies (except for the C standard library of course)
and can be compiled with `make`. When `sha256sum` is installed, a short test can
be run with `make test`.

Usage:
./main <input file>
```
```

- 编译

  执行以下命令，进行编译。

```
make
```

- 运行程序

  执行以下命令，运行可执行文件 sha256，并查看结果。

```
./sha256 README.md
```

6. 字符数字统计实验

以下步骤以在华为鲲鹏云服务器上执行为例。

- 创建汇编源代码文件 char_count.s。

执行以下命令进行汇编源码文件的创建：

```
touch char_count.s
```

- 编辑汇编代码

使用 vim 命令打开并编辑 char_count.s 文件。

```
vim char_count.s
```

按 i 进入插入模式，并输入以下 ARM 汇编代码：

```
.text
    .data
numCnt:
    .xword    0
upCnt:
    .zero 4
lowCnt:
    .zero 4
mystring:
    .zero 1001
ptr:
    .xword    mystring
    .section    .rodata
input:
    .string    "%s"
output:
    .string    "%d %d %d\n"

    .text
    .global    main
    .type main, %function
main:
    stp   x29,  x30,  [sp, #-16]!
    adr x1 , mystring
```
```

```
    adr x0 , input
    bl    __isoc99_scanf
    b     .L2
```

- 保存汇编代码

  先按 ecs 键退出 vim 的插入模式，接着按 shift 和：键进入命令模式，然后输入 wq，保存源码并退出 vim 编辑器。

- 进行数字判定

  本小节主要实现了.L6 的功能，即当出现数字的时候 numCnt 进行加一操作。

```
.L6:
    adr x0 , ptr
    ldr   x0, [x0]
    ldrb  w0, [x0] //将数据加载到 w0
    cmp  w0, 47 //对比 w0 和 47
    bls   .L3 //对比跳转指令，如果对比结果是小于等于则跳转到.L3 进行小写字符判定 如果大于则不进行跳转
    adr x0 , ptr
    ldr   x0, [x0]
    ldrb  w0, [x0]
    cmp  w0, 57 //对比 w0 和 57
    bhi   .L3 //如果上述比较结果是大于则跳转
    adr x0 , numCnt //如果运行到这一步则代表，该字符是一个数字
    ldr   w0, [x0]
    add  w1, w0, 1
    adr x0 , numCnt
    str   w1, [x0] //将数字加载 加一之后进行写回。
    b     .L4
```

- 小写字母判定和大写字母判定

  本小节主要实现了.L3 和.L5 的功能的功能，即当出现小写时的时候 lowCnt 进行加一操作。出现大写时，upCnt 进行加一。这里直接给出两个模块的实现，实现方法和前者类似。

```
.L5:
    adr x0 , ptr
    ldr   x0, [x0]
    ldrb  w0, [x0]
    cmp  w0, 64
    bls   .L4
    adr x0 , ptr
    ldr   x0, [x0]
    ldrb  w0, [x0]
    cmp  w0, 90
    bhi   .L4
    adr x0, upCnt
    ldr   w0, [x0]
    add  w1, w0, 1
    adr x0, upCnt
    str   w1, [x0]
.L4:
```

```
    adr x0 , ptr
    ldr   x0, [x0]
    add  x1, x0, 1
    adr x0 , ptr
    str   x1, [x0]
```

五、 实验结果

1. hello-world 示例程序实验

```
[root@embeddedsystem ~]# cd projects/
[root@embeddedsystem projects]# mkdir hello
[root@embeddedsystem projects]# cd hello
[root@embeddedsystem hello]# vim hello.c
[root@embeddedsystem hello]# as hello.c -o hello.o
[root@embeddedsystem hello]# ls
hello.c  hello.o
[root@embeddedsystem hello]# ld hello.o -o hello
ld: warning: cannot find entry symbol _start; defaulting to 000000000040
00b0
[root@embeddedsystem hello]# ls
hello  hello.c  hello.o
[root@embeddedsystem hello]# ./hello
Hello World!
[root@embeddedsystem hello]#
```

2. 使用 C 语言代码调用汇编程序实验

```
[root@embeddedsystem called]# gcc globalCalling.c globalCalled.S -o call
ed
[root@embeddedsystem called]# ls
called  globalCalled.S  globalCalling.c
[root@embeddedsystem called]# ./called
Original Status: Source string   Destination string
Modified Status: Source string   Source string
[root@embeddedsystem called]#
```

3. 使用 C 语言代码内嵌汇编程序实验

```
[root@embeddedsystem builtin]# vim globalBuiltin.c
[root@embeddedsystem builtin]# gcc -E globalBuiltin.c -o globalBuiltin.i
[root@embeddedsystem builtin]# gcc -S globalBuiltin.i -o globalBuiltin.s
[root@embeddedsystem builtin]# gcc -c globalBuiltin.s -o globalBuiltin.o
[root@embeddedsystem builtin]# gcc globalBuiltin.o -o globalBuiltin
[root@embeddedsystem builtin]# ls
globalBuiltin    globalBuiltin.i  globalBuiltin.s
globalBuiltin.c  globalBuiltin.o
[root@embeddedsystem builtin]# ./globalBuiltin
out is 78563412
[root@embeddedsystem builtin]#
```

4. 利用鲲鹏处理器的流水线来优化汇编代码性能实验

```
[root@embeddedsystem memorycopy]# gcc copyfunc.s memorycopy.c -o m1
[root@embeddedsystem memorycopy]# ls
copyfunc.s  m1  memorycopy.c
[root@embeddedsystem memorycopy]# ./m1
memorycopy time is    53120660 ns
[root@embeddedsystem memorycopy]#
```

- 代码的第一阶段改进

```
[root@embeddedsystem memorycopy]# gcc copyfunc_v2_1.s memorycopy.c -o m21
[root@embeddedsystem memorycopy]# ls
copyfunc.s  copyfunc_v2_1.s  copyfunc_v2_2.s  m1  m21  memorycopy.c
[root@embeddedsystem memorycopy]# ./m21
memorycopy time is    33475320 ns
[root@embeddedsystem memorycopy]# gcc copyfunc_v2_2.s memorycopy.c -o m22
[root@embeddedsystem memorycopy]# ls
copyfunc.s       copyfunc_v2_2.s  m21  memorycopy.c
copyfunc_v2_1.s  m1               m22
[root@embeddedsystem memorycopy]# ./m22
memorycopy time is  3328546696 ns
[root@embeddedsystem memorycopy]#
```

- 代码的第二阶段改进

```
[root@embeddedsystem memorycopy]# gcc copyfunc_v3_1.s memorycopy.c -o m31
[root@embeddedsystem memorycopy]# ls
copyfunc.s       copyfunc_v2_2.s  copyfunc_v3_2.s  m1   m22  memorycopy.c
copyfunc_v2_1.s  copyfunc_v3_1.s  copyfunc_v3_3.s  m21  m31
[root@embeddedsystem memorycopy]# ./m31
memorycopy time is    12133420 ns
[root@embeddedsystem memorycopy]# gcc copyfunc_v3_2.s memorycopy.c -o m32
[root@embeddedsystem memorycopy]# ls
copyfunc.s       copyfunc_v2_2.s  copyfunc_v3_2.s  m1   m22  m32
copyfunc_v2_1.s  copyfunc_v3_1.s  copyfunc_v3_3.s  m21  m31  memorycopy.c
[root@embeddedsystem memorycopy]# ./m32
memorycopy time is    12037570 ns
[root@embeddedsystem memorycopy]# gcc copyfunc_v3_3.s memorycopy.c -o m33
[root@embeddedsystem memorycopy]# ls
copyfunc.s       copyfunc_v2_2.s  copyfunc_v3_2.s  m1   m22  m32  memorycopy.c
copyfunc_v2_1.s  copyfunc_v3_1.s  copyfunc_v3_3.s  m21  m31  m33
[root@embeddedsystem memorycopy]# ./m33
memorycopy time is    12069230 ns
[root@embeddedsystem memorycopy]#
```

5. 密码运算实验

```
[root@embeddedsystem sha256-armv8]# make
gcc -O3 -mcpu=generic+crypto sha256.c sha256-armv8-aarch64.S -o sha256
[root@embeddedsystem sha256-armv8]# ls
Makefile   sha256                 sha256-armv8-aarch64.S
README.md  sha256-armv8-aarch32.S  sha256.c
[root@embeddedsystem sha256-armv8]# ./sha256 README.md
-------------------------------------------------------------------------------
| Final Hash                                                                  |
-------------------------------------------------------------------------------
|   H[0]   |   H[1]   |   H[2]   |   H[3]   |   H[4]   |   H[5]   |   H[6]   |   H[7]   |
|  6a77139a | c35bcdd6 | 8dc64244 | f6f30751 | d4d33c2e | 3c7c350e | a8f38be2 | 9348d6e0 |
-------------------------------------------------------------------------------
[root@embeddedsystem sha256-armv8]#
```

6. 字符数字统计实验

```
[root@embeddedsystem charCount]# gcc char_count.s -o char_count
[root@embeddedsystem charCount]# ls
char_count  char_count.s
[root@embeddedsystem charCount]# ./char_count
aBCafgahog12334Afa8
6 3 10
[root@embeddedsystem charCount]#
```

六、 心得体会

通过华为云线上开发实验，我掌握了在鲲鹏云服务器上从搭建资源环境到代码输入并且运行的一系列命令流程，并且第一次接触到了云编程的使用，通过六个编程代码的编写，我能够熟练地使用 C 语言源码来调用汇编源码、C 语言代码中内嵌汇编代码，并且通过后面三个实验的实际应用加深了这种理解，对于自己的工程实践能力，也有了较大的提升。