实验h1 华为云实验

目录

[基础代码示例程序 1](#_Toc137543628)

[1.实验1：hello-world示例程序 1](#_Toc137543629)

[1.1实验目的 1](#_Toc137543630)

[1.2实验原理 1](#_Toc137543631)

[1.3程序代码 1](#_Toc137543632)

[1.4实验结果 2](#_Toc137543633)

[1.5结果讨论 3](#_Toc137543634)

[1.6思考题 4](#_Toc137543635)

[2.实验2：使用C语言代码调用汇编程序 4](#_Toc137543636)

[2.1实验目的 4](#_Toc137543637)

[2.2实验原理 4](#_Toc137543638)

[2.3程序代码 4](#_Toc137543639)

[2.4实验结果 5](#_Toc137543640)

[2.5结果讨论 6](#_Toc137543641)

[2.6思考题 7](#_Toc137543642)

[3.实验3：使用C语言代码内嵌汇编程序 7](#_Toc137543643)

[3.1实验目的 7](#_Toc137543644)

[3.2程序代码 7](#_Toc137543645)

[3.3实验结果 8](#_Toc137543646)

[3.4结果讨论 8](#_Toc137543647)

[3.5思考题 9](#_Toc137543648)

[实验：利用鲲鹏处理器的流水线来优化汇编代码性能实验 10](#_Toc137543649)

[1.实验目的 10](#_Toc137543650)

[2.实验原理 10](#_Toc137543651)

[3.实验代码和结果展示 10](#_Toc137543652)

[3.1原始示例程序源码 10](#_Toc137543653)

[3.2代码的第一阶段改进 13](#_Toc137543654)

[3.3代码的第二阶段改进 15](#_Toc137543655)

[4.结果讨论 18](#_Toc137543656)

[18](#_Toc137543657)

[5思考题 20](#_Toc137543658)

# 基础代码示例程序

## 1.实验1：hello-world示例程序

### 1.1实验目的

实现ARM平台精简指令集（RISC）编写的hello-world程序的编译和运行。

### 1.2实验原理

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。

### 1.3程序代码

**示例程序源码hello.s**

.text

.global tart1 ; 声明 tart1 符号为全局符号

tart1: ; tart1 函数的开始

mov x0, #0 ; 将值 0 移动到寄存器 x0

ldr x1, =msg ; 将字符串 msg 的地址加载到寄存器 x1

mov x2, len ; 将 len 的值移动到寄存器 x2

mov x8, 64 ; 将值 64 移动到寄存器 x8

svc #0 ; 执行软中断指令，进入操作系统

mov x0, 123 ; 将值 123 移动到寄存器 x0

mov x8, 93 ; 将值 93 移动到寄存器 x8

svc #0 ; 执行软中断指令，进入操作系统

.data

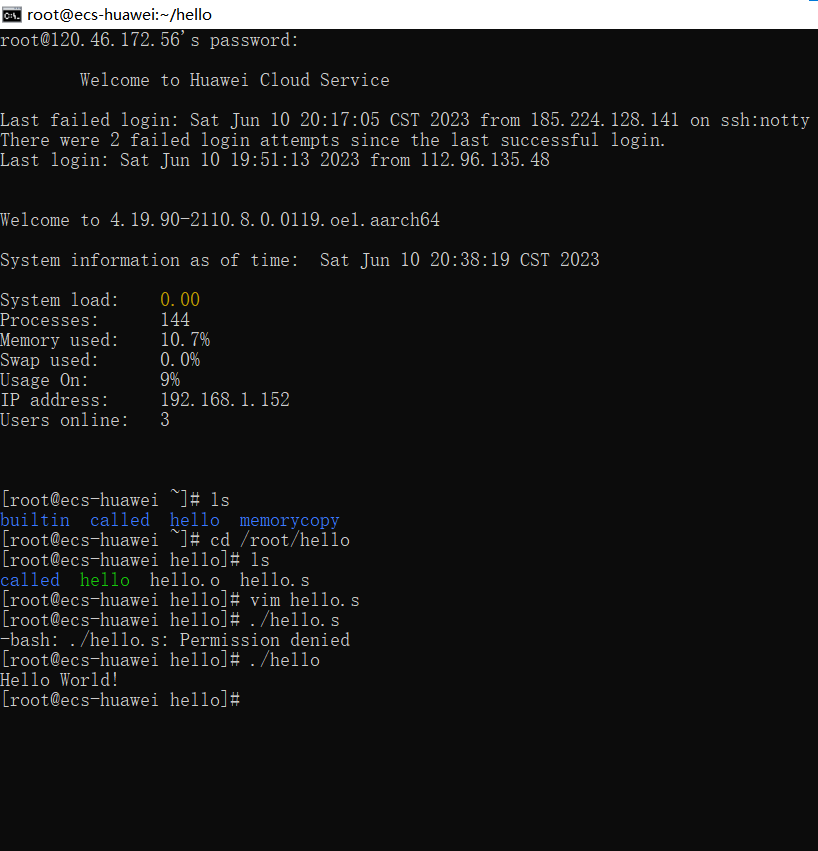
msg:

.ascii "Hello World!\n" ; 字符串 "Hello World!\n"

len = . - msg ; 计算 len 的值，即 msg 字符串的长度

### 1.4实验结果





### 1.5结果讨论

Helloworld这个程序使用汇编语言写的，这段代码定义了一个名为 **tart1** 的函数，实现了一些功能。在 **tart1** 函数中，首先将寄存器 **x0** 设置为 0，然后加载字符串 **msg** 的地址到寄存器 **x1**，将 **len** 的值移动到寄存器 **x2**，接着通过执行软中断指令进入操作系统。然后，将寄存器 **x0** 设置为 123，将寄存器 **x8** 设置为 93，再次通过软中断指令进入操作系统。

在数据段（**.data**）中定义了一个字符串 **msg**，内容为 "Hello World!\n"，并通过 **. - msg** 计算出了字符串的长度，将结果存储在 **len** 中。

可以看到编写的hello-wolrd示例程序已经在华为鲲鹏云服务器上通过编译和运行，并成功输出结果。

### 1.6思考题

 思考下同样的代码在X86平台能否运行，为什么？

答案：不能，因为X86平台使用的是复杂指令集(CISC)，而我们实验中使用到的华为鲲鹏云服务器是基于ARM平台的，使用的是精简指令集(RISC)，二者的汇编指令差异较大。

## 2.实验2：使用C语言代码调用汇编程序

### 2.1实验目的

实现ARM平台上通过C语言源码来调用汇编源码中的代码。

### 2.2实验原理

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。

### 2.3程序代码

**globalCalling.c源代码**

/\* globalCalling.c \*/

#include <stdio.h>

extern void strcpy1(char \*d, const char \*s); // 声明 strcpy1 函数，该函数在另一个文件中定义

int main()

{

const char \*srcstring = "Source string"; // 声明一个指向常量字符串的指针

char dststring[] = "Destination string"; // 声明一个字符数组

printf("Original Status: %s %s\n", srcstring, dststring); // 打印原始状态：源字符串和目标字符串

strcpy1(dststring, srcstring); // 调用 strcpy1 函数，将源字符串复制到目标字符串

printf("Modified Status: %s %s\n", srcstring, dststring); // 打印修改后的状态：源字符串和目标字符串

return 0;

}

**globalCalled.S源代码**

/\* globalCalled.S \*/

.global strcpy1 // 声明 strcpy1 符号为全局符号

# Start the function: strcpy1

strcpy1: // strcpy1 函数的开始

LDRB w2, [X1], #1 // 从内存中加载一个字节到寄存器 w2，并递增 X1 寄存器的值

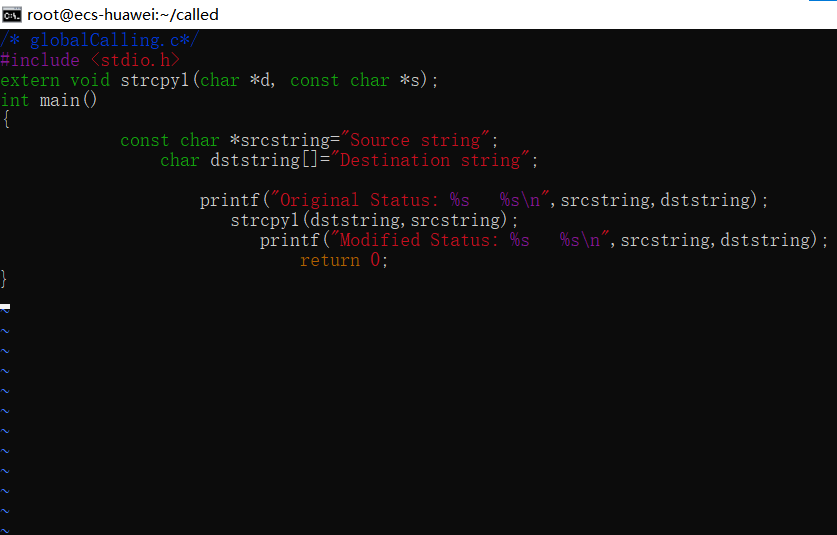
STR w2, [X0], #1 // 将寄存器 w2 的值存储到内存中，并递增 X0 寄存器的值

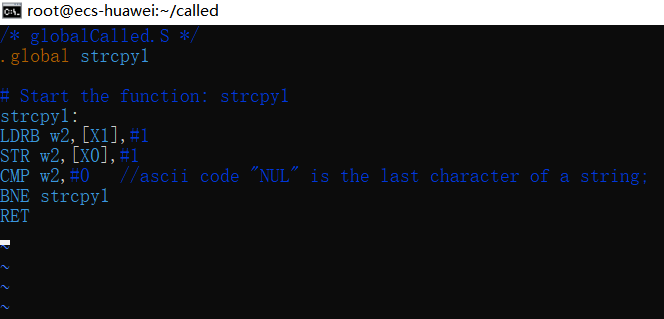
CMP w2, #0 // 将寄存器 w2 的值与 0 进行比较

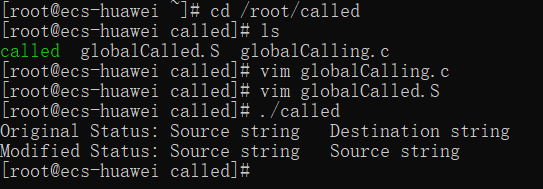
BNE strcpy1 // 如果 w2 不等于 0，跳转到 strcpy1 标签处继续执行

RET // 函数返回

### 2.4实验结果







### 2.5结果讨论

**globalCalled.S**：

* 定义了一个全局的汇编函数 **strcpy1**，该函数实现了字符串的复制操作。
* 函数通过字节拷贝的方式，从源字符串的地址 **X1** 读取一个字节数据，并将其存储到目标字符串的地址 **X0**，然后递增源字符串和目标字符串的指针。
* 重复以上步骤，直到遇到字符串的结尾（ASCII 码为 "NUL"，值为 0）。
* 最后，函数返回。

**globalCalling.c**：

* 在 **stdio.h** 头文件中声明了 **extern void strcpy1(char \*d, const char \*s);**，表示 **strcpy1** 函数的声明。
* 在 **main** 函数中，定义了一个指向源字符串的指针 **srcstring** 和一个目标字符串数组 **dststring**。
* 使用 **printf** 函数输出原始的源字符串和目标字符串。
* 调用 **strcpy1(dststring, srcstring)**，将源字符串复制到目标字符串中。
* 再次使用 **printf** 函数输出修改后的源字符串和目标字符串。
* 返回 0，表示程序正常结束。

**globalCalling.c** 文件中的 **main** 函数通过调用 **globalCalled.S** 文件中的 **strcpy1** 函数，实现了字符串的复制操作。在输出结果中，可以看到目标字符串已经被修改为与源字符串相同的内容。

### 2.6思考题

 除了使用调用汇编脚本的方式，还有哪些方法可以使用C语言调用汇编代码？

可以使用C语言中的asm关键字来在C语言源码中嵌入汇编代码。

## 3.实验3：使用C语言代码内嵌汇编程序

### 3.1实验目的

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。

### 3.2程序代码

**/\* globalBuiltin.c\*/**

/\* 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"

);

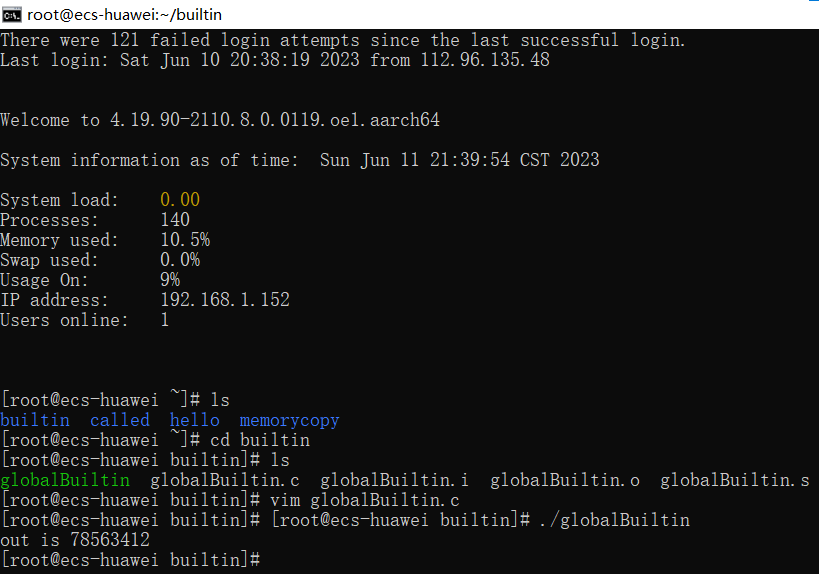
printf("out is %x \n",val);

return 0;

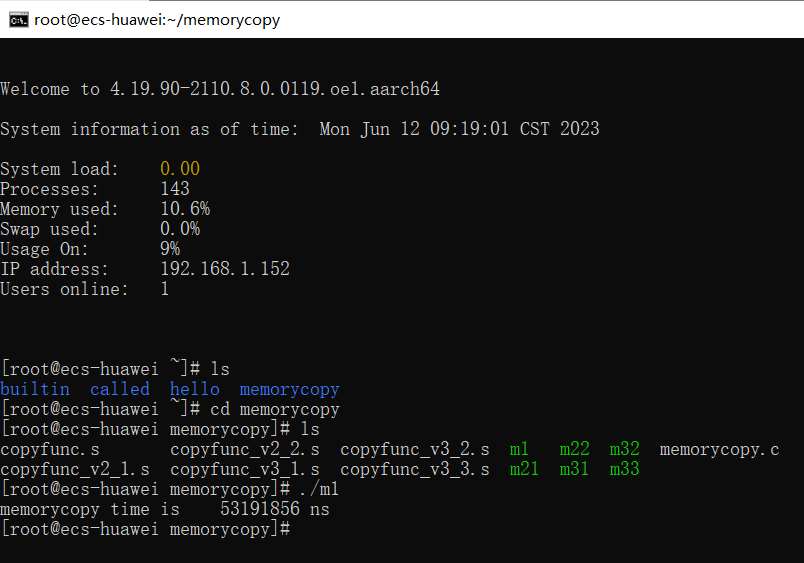
}

### 3.3实验结果





### 3.4结果讨论

这段代码使用了内嵌汇编的方式，实现了以下操作：

初始化一个整型变量 val，赋值为 0x12345678。

使用 ARM 汇编指令对变量 val 进行处理。

首先，将变量 val 的值移动到寄存器 x3 中。

接着，将寄存器 w3 中的值循环右移 8 位，相当于对 val 进行了字节交换。

再使用位与操作 bic，将寄存器 w3 的低字节和高字节分别置零，即保留了中间两个字节。

然后，将变量 val 的值再次移动到寄存器 x4 中。

将寄存器 w4 中的值循环右移 24 位，相当于再次对 val 进行了字节交换。

使用位与操作 bic，将寄存器 w4 的高字节和低字节分别置零，即保留了中间两个字节。

使用指令 add 对寄存器 w3 和寄存器 w4 进行求和操作，并将结果保存在寄存器 w3 中。

最后，将寄存器 x3 的值移动回变量 val，并输出最终的结果。

简而言之，这段代码通过汇编指令对变量 val 进行了字节交换和位操作，最终将交换后的结果存回到 val 中。

### 3.5思考题

在C代码中内嵌的汇编语句的基本格式为是什么？

\_\_asm\_\_ \_\_volatile\_\_ (“asm code”

：输出操作数列表

：输入操作数列表

：clobber列表

)

# 实验：利用鲲鹏处理器的流水线来优化汇编代码性能实验

## 1.实验目的

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，来提升改进字符串复制代码，提高代码执行效率。

## 2.实验原理

鲲鹏920处理器的流水线支持2个访存指令的发射队列，如下图中的红色框内所示，其访存单元WB支持每拍2条读或写访存指令。本实验指导书通过在华为鲲鹏云服务器上，利用该特性来不断改进代码的实现方式提高代码的运行效率。完成实验操作后，读者会了解到双发射队列流水线指令的适用范围，并掌握如何利用该特性编写和改进代码程序。



鲲鹏920处理器流水线结构图

## 3.实验代码和结果展示

### 3.1原始示例程序源码

**memorycopy.c**

#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;

}

**copyfunc.s**

.global memorycopy

memorycopy:

ldrb w3,[x1],#1

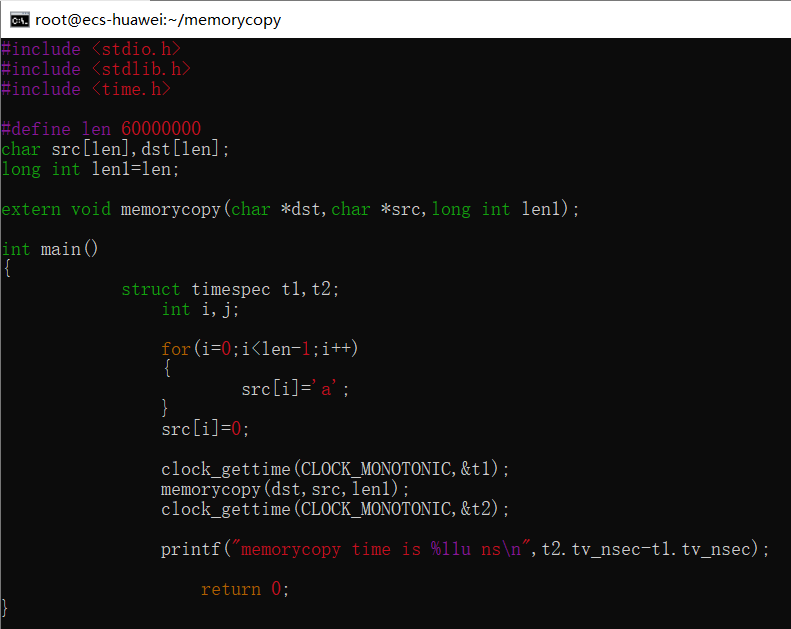
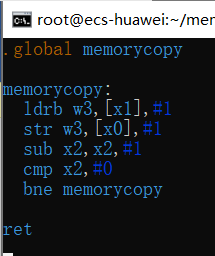
str w3,[x0],#1

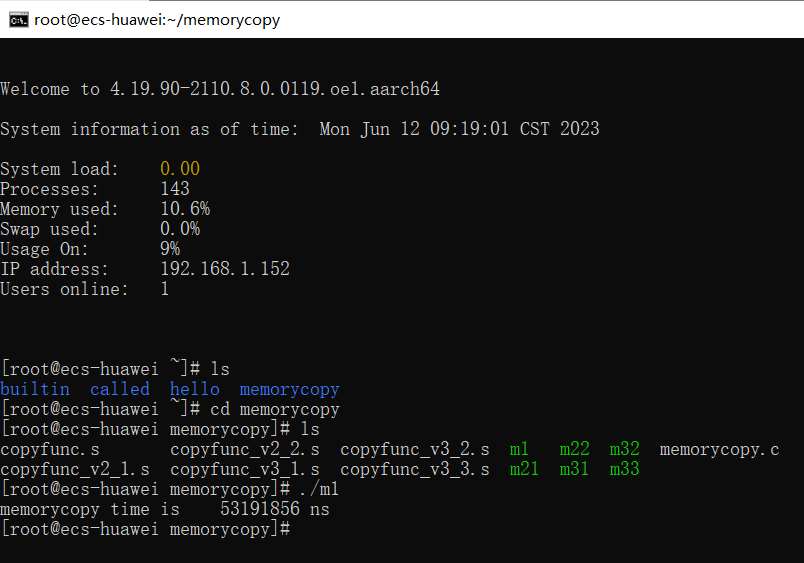
sub x2,x2,#1

cmp x2,#0

bne memorycopy

ret

**** ****

****

### 3.2代码的第一阶段改进

**copyfunc\_v2\_1.s**

.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

**copyfunc\_v2\_2.s**

.global memorycopy

memorycopy:

sub x1,x1,#1

sub x0,x0,#1

lp:

ldrb w3,[x1,#1]

ldrb w4,[x1,#2]

ldrb w5,[x1,#3]

ldrb w6,[x1,#4]!

str w3,[x0,#1]

str w4,[x0,#2]

str w5,[x0,#3]

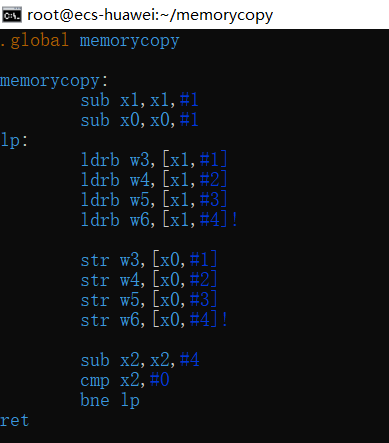
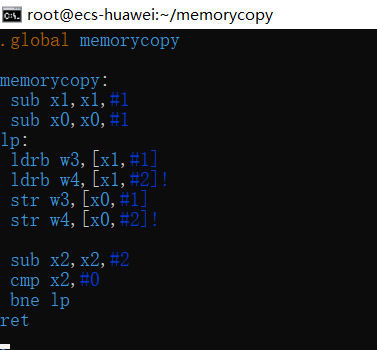
str w6,[x0,#4]!

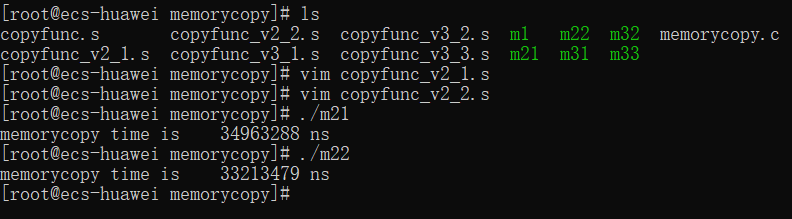
sub x2,x2,#4

cmp x2,#0

bne lp

ret

****

****

### 3.3代码的第二阶段改进

**copyfunc\_v3\_1.s**

.global memorycopy

memorycopy:

ldp x3,x4,[x1],#16

stp x3,x4,[x0],#16

sub x2,x2,#16

cmp x2,#0

bne memorycopy

ret

copyfunc\_v3\_2.s

.global memorycopy

memorycopy:

sub x1,x1,#16

sub x0,x0,#16

lp:

ldp x3,x4,[x1,#16]

ldp x5,x6,[x1,#32]!

stp x3,x4,[x0,#16]

stp x5,x6,[x0,#32]!

sub x2,x2,#32

cmp x2,#0

bne lp

ret

copyfunc\_v3\_3.s

.global memorycopy

memorycopy:

sub x1,x1,#16

sub x0,x0,#16

lp:

ldp x3,x4,[x1,#16]

ldp x5,x6,[x1,#32]

ldp x7,x8,[x1,#48]

ldp x9,x10,[x1,#64]!

stp x3,x4,[x0,#16]

stp x5,x6,[x0,#32]

stp x7,x8,[x0,#48]

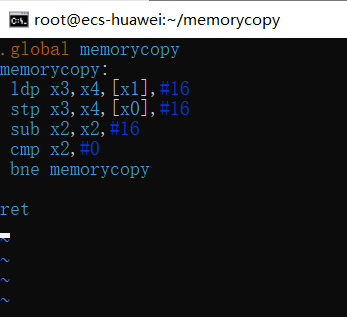
stp x9,x10,[x0,#64]!

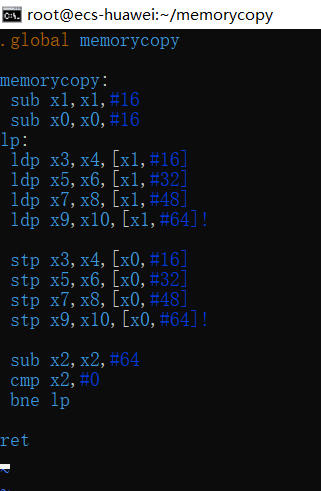
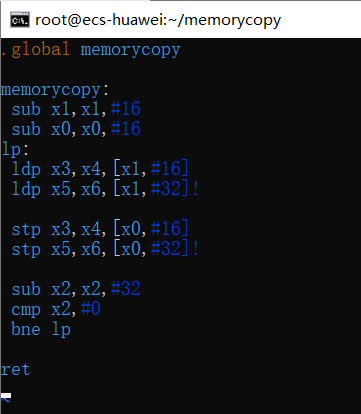
sub x2,x2,#64

cmp x2,#0

bne lp

ret



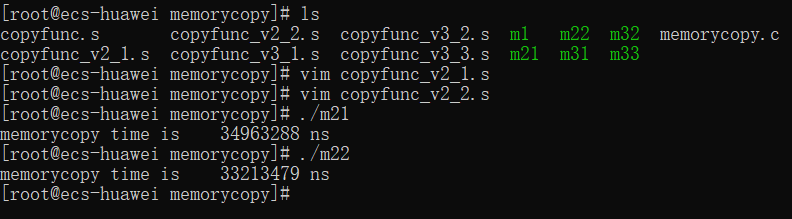


## 4.结果讨论

## 

可以看到原始方法编写的memorycopy的执行时间为53191856纳秒,該方法有著明顯的缺點，因爲原始代码每一次循环（即指令执行周期内）只进行了一次访存，并没有充分利用鲲鹏920处理器其访存单元WB支持每拍2条读或写访存指令的特性。

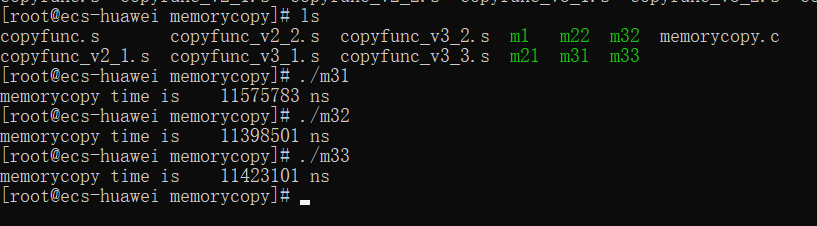
**第一阶段改进**



能明显看到通过利用鲲鹏920处理器特性而专门改进过的汇编代码与原来相比，缩减了接近一倍的时间。

原因分析：鲲鹏920有两个load/store流水线，其访存单元支持每拍2条读或写访存操作。原始代码下由于源字符串的地址和目标的字符串的地址并不连续，而且这种不连续地址的一读一写交替进行，导致内存访问的连续性很差，cache命中率较低。另一方面，由于循环分支指令较多，会经常刷新流水线，这也大大降低了两个访存指令发射队列中访存指令的充满速度。比起原始代码，改进代码则有效改进了这些缺点，使得其访存延迟大大降低。

**第二阶段改进**



通过改进上一节的代码，我们可以提高内存访问的效率。在第一次改进中，每次读取或写入内存都是以一个字节为单位进行的，这种方式效率较低。为了充分利用内存突发传输的优势，我们可以采用一次读取或写入16个字节的方法。Arm64/Aarch64架构提供了ldp指令和stp指令，它们可以一次性访问16个字节的内存数据。这些指令具有较高的内存连续性，可以有效降低访存延迟。因此，通过使用ldp和stp指令，我们可以进一步改进代码，提升内存访问效率。于是，我们在前面的三段汇编代码上都采用一次读取或写入16个字节的方法，得到三个新的汇编代码文件。从图中数据可以明显看到后三种改进代码都远远好于第一阶段的代码。

在以上改进的代码中，同一类型的访存指令（例如ldrb指令或stp指令）之间是独立的，它们之间没有任何依赖关系，包括寄存器资源和数据依赖。这意味着在循环展开后，我们可以充分利用两条load/store流水线并行执行这些访存指令，以提高访存性能。

这种并行执行的优势在于，每个访存指令可以独立地进行加载或存储操作，而不会受到其他指令的干扰。因此，当代码中的访存指令之间存在依赖关系时，比如下面的情况：

ldrb w3,[x1],#1

ldrb w4,[x1],#1

后一条ldrb指令中的x1取值依赖于前一条ldrb指令执行完毕之后x1取值的更新。在这种情况下，改进的效果就会大打折扣，无法充分利用并行执行来提升访存性能。

因此，通过确保改进代码中的访存指令之间不存在依赖关系，我们可以最大程度地发挥并行执行的优势，从而有效提升访存性能。

## 5思考题

采用以2byte、4byte、8byte为基本传输单位的访存指令，其各自性能如何？

使用较大的传输单位可以在一次访存操作中读取/写入更多的数据，从而减少访存操作的次数，提高访存性能。

如下两种代码写法，哪一种性能更优？为什么？

代码1：

memorycopy:

sub x1,x1,#1

sub x0,x0,#1

lp:

ldrb w3,[x1,#1]!

ldrb w4,[x1,#1]!

str w3,[x0,#1]!

str w4,[x0,#1]!

sub x2,x2,#2

cmp x2,#0

bne lp

代码2：

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

前者更优。在第一个代码片段中，使用了前一节提到的ldrb指令和stp指令，可以一次读取/写入两个字节。这样的内存访问方式利用了内存的突发传输特性，可以提高访存性能。此外，循环展开后的代码可以充分利用两条load/store流水线并行执行指令，进一步提高性能。

而第二个代码片段中，依然使用了ldrb指令，但在加载和存储过程中仅处理一个字节。这种逐字节的访存方式可能导致访存延迟的增加，并且无法充分利用并行执行的优势。

* 是否可以采用8路/16路/32路…循环展开的方式？如果可以，会遇到什么问题？性能会怎样变化？

可以采用循环展开的方式来提高访存性能。通过将循环体中的访存操作重复展开多次，可以增加指令级并行性和访存操作的重叠，从而提高整体的访存吞吐量。

当使用8路、16路、32路等较大的循环展开时，会增加指令的数量和并行度。这可以在某些情况下提高性能，特别是当处理器具有多个执行单元或乱序执行能力时。

然而，采用较大的循环展开也会带来一些问题和限制：

指令缓存大小限制：较大的循环展开会增加指令的数量，可能超过处理器的指令缓存大小，导致缓存不命中和指令流水线的停顿。

数据依赖性：在循环展开中，如果存在数据依赖关系，例如后一次迭代需要使用前一次迭代的计算结果，较大的展开度可能导致这些依赖无法满足，从而影响正确性和性能。

循环展开带来的代码膨胀：较大的循环展开会增加代码的长度，可能导致指令缓存命中率下降，从而影响性能。

性能的具体变化取决于多个因素，包括处理器架构、缓存大小、数据依赖性、访存模式等。在某些情况下，较大的循环展开可以显著提高性能，但在其他情况下，可能会带来负面影响。因此，在进行循环展开时，需要综合考虑以上因素，并进行实际的性能测试和评估，以确

以确定最佳的循环展开策略。

* 查阅一下Linux Arm64环境下，C函数memcpy的汇编源代码的实现过程（例如：<https://blog.csdn.net/forever_2015/article/details/50286009>），并测试其性能。

根据代码的结构，可以大致分为以下几个阶段：

非对齐内存块处理：当数据块的首地址非16字节对齐时，需要处理尾巴部分的字节。这个阶段的开销相对较小，主要包括加载和存储操作。

数据块长度小于等于63字节的处理：这部分代码主要通过条件判断和加载/存储操作来处理小于等于63字节的数据块。

数据块长度大于63字节的处理：对于大于63字节的数据块，代码会进行循环处理，每次处理64字节的数据，直到剩余数据块长度小于64字节。

基于以上分析，对于从1字节到1KB之间大小的数据块，我们可以有以下观察和推测：

对于非对齐内存块的处理，在数据块较小时（如几十字节以内），这部分开销可能是可以忽略不计的。

当数据块长度小于等于63字节时，代码通过条件判断逐个处理字节、半字和字大小的数据。由于代码逻辑简单且没有循环，处理时间应该是线性增长的。

当数据块长度大于63字节时，代码会进行循环处理，每次处理64字节的数据。这部分代码的性能可能较好，因为它使用了一次加载和存储操作来处理64字节的数据。

综上所述，对于从1字节到1KB之间的数据块，代码的性能可能会在不同的阶段有所差异。对于较小的数据块，性能可能较低，而对于较大的数据块，性能可能会有所提升。但具体的性能表现还需要在实际环境中进行测试才能得出准确结论。

* 有一段C语言代码，如下所示：

for(i=0;i<10000;i++,out++)

\*out=array[\*i];

是否可以基于相同的原理来对其进行改写，以优化其性能？

可以代码如下：

const int\* end = array + 10000;

int\* out\_end = out + 10000;

while (out < out\_end) {

\*out++ = \*array++;

}

通过使用指针递增和一次加载指令，减少了循环迭代和访存操作次数，从而提高了性能。