華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础

实验名称： ARM 指令系统的理解

院　　系 ： 计算机科学与技术

专业班级 ： CS2304

学　　号 ： U202315653

姓　　名 ： 郝依凝

指导教师 ： 金良海

2025 年 4 月 23 日

一、实验目的与要求

通过在 ARM 虚拟环境下调试执行程序，了解 ARM 的指令系统。

实验环境：ARM 虚拟实验环境 QEMU

工具：gcc, gdb 等

二、实验内容

任务 1、C 与汇编的混合编程

任务 2、内存拷贝及优化实验

程序及操作方法 见〈ARM 实验任务.pdf〉

三、实验记录及问题回答

（1）实验任务的实验结果记录

1）C 与汇编的混合编程

```
[root@localhost ~]# gcc sum.c add.s -o sum
[root@localhost ~]# ./sum
100
sum=5050
[root@localhost ~]#
```

图 1 C 语言调用汇编实现累加和求值

```
[root@localhost ~]# gcc builtin.c -o builtin
[root@localhost ~]# ./builtin
100
sum is 5050
[root@localhost ~]#
```

图 2 C 语言内嵌汇编

2）内存拷贝及优化

```
[root@localhost ~]# gcc time.c copy.s -o m1
[root@localhost ~]# ./m1
momorycopy time is   165711200 ns
[root@localhost ~]#
```

图 3 内存拷贝基础代码

```
[root@localhost ~]# gcc time.c copy121.s -o m121
[root@localhost ~]# ./m121
momorycopy time is   152669008 ns
[root@localhost ~]#
```

图 4 内存拷贝 2 倍展开优化

```
[root@localhost ~]# gcc time.c copy122.s -o m122
[root@localhost ~]# ./m122
momorycopy time is   134057408 ns
[root@localhost ~]#
```

图 5 内存拷贝 4 倍展开优化

```
[root@localhost ~]# gcc time.c copy21.s -o m21
[root@localhost ~]# ./m21
momorycopy time is    42263563 ns
[root@localhost ~]#
```

图 6 内存突发传输方式优化

（2）ARM 指令及功能说明

1）C 与汇编的混合编程

a）C 语言调用汇编实现累加和求值

本示例实现的功能是:输入一个正整数,输出从 0 到该正整数的所有正整数的累加和,输入输出功能在 C 代码中实现,计算功能通过调用汇编函数实现。需要传入的参数是输入的正整数,汇编传出的参数为累加和,因此只用到一个 x0 寄存器即可实现参数传递功能。

```
#include<stdio.h>
extern int add(int num);
int main()
{
int i,sum;
scanf("%d",&i);
sum=add(i);
printf("sum=%d\n",sum);
return 0;
}
```

图 7 sum.c 文件

sum.c 文件：声明函数 add，调用外部函数 add，实现接收参数输入、结果输出功能。

```
.global add
add:
ADD x1,x1,x0
SUB x0,x0,#1
CMP x0,#0
BNE add
MOV x0,x1
RET
```

图 8 add.s 文件

add.s 文件：C 语言调用汇编有两个关键点——调用与传参。对于调用,我们需要在汇编程序中通过.global 定义一个全局函数 add，然后该函数就可以在 C 代码中通过 extend 关键字加以声明,使其能够在 C 代码中直接调用。将参数保存在 x0 寄存器，将 x0+x1 的值存入 x1 寄存器，使用 x1 寄存器保存 sum 临时值，将 x0-1 的值存入 x0 寄存器，比较 x0 和 0 的大小判断循环是否结束，若 x0 与 0 不相等及循环未结束,跳转到 add 继续累加;若 x0=0 则循环结束，将 x1 的值存入 x0，通过 x0 来返回值。实现输出从 0 到该正整数的所有正整数的累加和。

```
[root@localhost ~]# gcc sum.c add.s -o sum
[root@localhost ~]# ./sum
100
sum=5050
[root@localhost ~]#
```

图 9 C 语言调用汇编实现累加和求值测试结果

测试结果：输入 100 得到累加和为 5050，测试结果正确。

b）C 语言内嵌汇编

在 C 语言代码中内嵌汇编语句的基本格式为:

__asm__ __volatile__("asm code"

:输出操作数列表

:输入操作数列表

:clobber 列表

);

asm 用于声明这行代码是一个内嵌汇编表达式，关键字 volatile 告诉编译器不要优化内嵌的汇编语。括号里面包含四个部分:汇编代码(asm code)、输出操作数列表(output)、输入操作数列表(input)和 clobber 列表(破坏描述符)。这四个部分之间用":"隔开。其中, 输入操作数列表部分和 clobber 列表部分是可选的。括号之后要以";"结尾。

```
#include<stdio.h>
int main()
{
int val;
scanf("%d",&val);
__asm__ __volatile__(
"add:\n"
"ADD x1,x1,x0\n"
"SUB x0,x0,#1\n"
"CMP x0,#0\n"
"BNE add\n"
"MOV x0,x1\n"
:"=r"(val)
:"0"(val)
:
);
printf("sum is %d\n",val);
return 0;
}
```

图 10 builtin.c 文件

builtin.c 文件：变量 val 接收一个整型输入，通过内嵌汇编，其中汇编代码部分逻辑与上述 add.s 文件一致；r 代表存放在某个通用寄存器中，即在汇编代码里用一个寄存器代替()部分中定义的 C 变量即 val，=代表只写，即在汇编代码里只能改变 C 变量的值，而不能取它的值；"0"(val)中 0 代表与第一个输出参数共用同一个寄存器 x0，即将 val 的初始值作为寄存器 x0 的输入。

```
[root@localhost ~]# gcc builtin.c -o builtin
[root@localhost ~]# ./builtin
100
sum is 5050
[root@localhost ~]#
```

图 11 C 语言内嵌汇编测试结果

测试结果：输入 100 得到累加和为 5050，测试结果正确。

2）内存拷贝及优化

优化效果通过计算对应代码段的执行时间来判断。具体方案是通过 C 语言调用汇编，在 C 代码中计算时间，在汇编代码中设计不同的方案，对比每种方案的执行时间，判断优化效果。本示例的优化针对内存读写,示例程序功能是内存拷贝,拷贝功能在汇编函数中实现。

a）基础代码

本程序由两部分组成:第一部分是主函数,采用 LinuxC 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。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。

```
#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("momorycopy time is %11u ns\n",t2.tv_nsec-t1.tv_nsec);
return 0;
}
"time.c" 22L, 441C
```

图 12 time.c 文件

time.c 文件：拷贝内容长度为 60000000，源地址为 src，目标地址为 dst，通过外部函数 memorycopy 实现将字符串由 src 拷贝至 dst，通过 clock_gettime 函数计算拷贝用时。

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

图 13 copy.s 文件

copy.s 文件：在汇编程序中通过.global 定义一个全局函数 memorycopy，将 src 字符串地址传入 x1，将 dst 字符串地址传入 x0。使用后索引方式，将 x1 指向的地址处的一个字节放入 w3 中,然后 x1+1->x1，将 w3 中一个字节放入 x0 指向的地址处中，然后 x0+1->x0，通过寄存器 x2 传入字符串长度，判断 x2 是否递减为 0 判定循环是否结束。实现将字符串由 src 拷贝至 dst 功能。

```
[root@localhost ~]# gcc time.c copy.s -o m1
[root@localhost ~]# ./m1
momorycopy time is   165711200 ns
[root@localhost ~]#
```

图 14 内存拷贝基础代码测试结果

测试结果：用时 165711200ns

b）循环展开优化

循环展开是最常见的代码优化思路,通过减少指令总数来实现代码优化。

```
.global memorycopy
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
ret
```

图 15 copy121.s 文件

copy121.s 文件：在汇编程序中通过.global 定义一个全局函数 memorycopy，将 src 字符串地址传入 x1，将 dst 字符串地址传入 x0。采用自动索引方式，需先将寄存器 x1、寄存器 x0 中保存地址前移 1；循环过程中，执行将 x1+1 指向的地址处的一个字节放入 w3 后 x1+1->x1，将 x1+1 指向的地址处的一个字节放入 w4 后 x1+1->x1，执行将 w3 中一个字节放入 x0+1 指向的地址处中后 x0+1->x0，将 w4 中一个字节放入 x0+1 指向的地址处中后 x0+1->x0，实现每次循环读取 2 个字节，写入 2 个字节；通过寄存器 x2 传入字符串长度,判断 x2 是否递减 2 直至为 0 判定循环是否结束。实现将字符串由 src 拷贝至 dst 功能。

```
[root@localhost ~]# gcc time.c copy121.s -o m121
[root@localhost ~]# ./m121
momorycopy time is   152669008 ns
[root@localhost ~]#
```

图 16 内存拷贝 2 倍展开优化测试结果

测试结果：用时 152669008ns，函数执行时间得到优化。

```
.global memorycopy
memorycopy:
sub x1,x1,#1
sub x0,x0,#1
lp:
ldrb w3,[x1,#1]!
ldrb w4,[x1,#1]!
ldrb w5,[x1,#1]!
ldrb w6,[x1,#1]!
str w3,[x0,#1]!
str w4,[x0,#1]!
str w5,[x0,#1]!
str w6,[x0,#1]!
sub x2,x2,#4
cmp x2,#0
bne lp
ret
```

图 17 copy122.s 文件

copy122.s 文件：代码逻辑与 copy121.s 中逻辑一致；采用自动索引方式，每次循环中改为从源地址依次读取 4 个字节分别存入寄存器 w3、w4、w5、w6，将寄存器 w3、w4、w5、w6 内容依次向目标地址写入 4 个字节；通过寄存器 x2 传入字符串长度，判断 x2 是否递减 4 直至为 0 判定循环是否结束。实现将字符串由 src 拷贝至 dst 功能。

```
[root@localhost ~]# gcc time.c copy122.s -o m122
[root@localhost ~]# ./m122
momorycopy time is   134057408 ns
[root@localhost ~]#
```

图 18 内存拷贝 4 倍展开优化测试结果

测试结果：用时 134057408ns，函数执行时间基于二倍的基础上也得到了优化，更多次数的循环展开可能会让程序的执行时间得到更好的优化。

c）内存突发传输方式优化

之前两次优化每次内存读写都是以一个字节为单位进行的,这样效率很低。由于内存在连续读/写多个数据时,其性能要优于非连续读/写数据的方式,此次优化思路是一次对多个字节进行读写。这就用到了 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,大大提高了内存读写效率。

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

图 19 copy21.s 文件内容

copy21.s 文件：在汇编程序中通过.global 定义一个全局函数 memorycopy，将 src 字符串地址传入 x1，将 dst 字符串地址传入 x0。ldp 指令从源地址一次读取 16 个字节存入寄存器 x3、x4 中，x1+16->x1，stp 指令将寄存器 x3、x4 中内容向目标地址一次写入 16 个字节，x0+16->x0，实现一次循环同时读取和写入 16 字节；通过寄存器 x2 传入字符串长度，判断 x2 是否递减 16 直至为 0 判定循环是否结束。实现将字符串由 src 拷贝至 dst 功能。

```
[root@localhost ~]# gcc time.c copy21.s -o m21
[root@localhost ~]# ./m21
momorycopy time is    42263563 ns
[root@localhost ~]#
```

图 20 内存突发传输方式优化测试结果

测试结果：用时 42263563ns，一次对 16 字节读写程序执行效率明显优于单字节读写。

四、体会

通过本次实验，我深入理解了 ARM 指令系统的基本原理及其在程序优化中的应用。

在 C 与汇编的混合编程任务中，我掌握了 C 语言调用汇编函数的关键技术，例如通过寄存器传递参数和返回值，以及全局函数的声明与调用。

在 C 语言内嵌汇编任务中，我熟悉了`__asm__ __volatile__`的语法规则，理解了输入/输出操作数列表和 clobber 列表的作用，进一步体会到汇编代码在细粒度控制硬件资源时的灵活性。

在内存拷贝及优化任务中，我对比了不同优化策略的性能差异。基础的单字节拷贝效率较低，而通过循环展开（2 倍、4 倍）减少了循环次数，降低了指令开销，使执行时间逐步缩短。最显著的优化来自内存突发传输方式，使用 ldp 和 stp 指令一次性读写 16 字节，充分利用了内存的连续访问特性，执行时间从基础代码的 165711200ns 优化至 42263563ns，性能提升近 4 倍。这一过程让我认识到硬件特性对程序性能的影响，以及合理选择指令的重要性。

C 语言是无法完全代替汇编语言的,一方面是其效率比 C 要高,另一方面是某些特殊的指令在 C 语法中是没有等价的语法的。例如:操作某些特殊的 CPU 寄存器如状态寄存器、操作主板上的某些 I/O 端口或者对性能要求极其苛刻的场景等,我们都可以通过在 C 语言中内嵌汇编代码来满足要求。

本次实验让我深刻体会到汇编语言在底层优化中的不可替代性，也让我意识到理论知识与实践结合的重要性。未来在开发高性能程序时，我将更加关注内存操作和指令级优化，充分利用硬件特性提升效率。同时，ARM 指令系统的学习为我后续探索嵌入式系统和体系结构打下了坚实基础。