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

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：CS2209

学　　号：U202210755

姓　　名：章宏宇

指导教师：李专

2024 年 5 月 1 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

2.1.1 调用汇编实现累加求值

首先，用 vi sum.c 来创建文件并编辑。按照任务编写内容如下：

```
1 #include <stdio.h>
2 extern int add(int num);
3 int main(){
4     int i, sum;
5     scanf("%d", &i);
6     sum = add(i);
7     printf("sum=%d\n", sum);
8     return 0;
9 }
~
```

之后，如法炮制，编写 add.s 文件

```
1 .global add
2 add:
3     ADD x1, x1, x0
4     SUB x0, x0, #1
5     CMP x0, #0
6     BNE add
7     MOV x0, x1
8     RET
9
~
```

最后，再用 gcc sum.c add.s -o sum 来编译链接生成可执行文件，最后执行：

```
[root@localhost lab2]# gcc sum.c add.s -o sum
[root@localhost lab2]# ./sum
100
sum=5050
```

2.1.2 C 语言内嵌汇编

这是我第一次在 C 语言内内嵌汇编，长知识了。在 C 语言内内嵌汇编需要遵守一定格

式，大概是

__asm__ [__volatile__](“asm code”:output:input:clobber);

填入需要汇编指令，如下图：

```
#include <stdio.h>
int main()
{
    int val;
    scanf("%d", &val);
    __asm__ __volatile__(
        "add:\n"
        "ADD x1, x0, x1\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;
}
```

这段汇编指令与 add.s 一致，只是省去了链接的过程，最后的效果也相同，如下图：

```
[root@localhost lab2]# gcc builtin.c -o builtin
[root@localhost lab2]# ./builtin
100
sum is 5050
```

2.2.1 内存拷贝基础代码

首先，先按照任务给出的代码完成 time.c 文件：

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

这里值得一提的是，一开始输出时我采用了 %11u 的格式，结果给我符号扩展到 19 位，把我吓一跳。排查了好一会才发现是 %11u，是固定 11 位的意思。。。

接下来编写 copy.s ：

```
1 .global memorycopy
2 memorycopy:
3     ldrb w3, [x1], #1
4     str w3, [x0], #1
5     sub x2, x2, #1
6     cmp x2, #0
7     bne memorycopy
8     ret
9
~
```

大概意思是创建两个索引 x1、x0，用 w3 当作中间寄存器，实现逐字节 copy。最后运行结果如下：

```
[root@localhost lab2]# gcc time.c copy.s -o m1
[root@localhost lab2]# ./m1
memroycopy time is  3697783792 ns
t2: 204438192, t1: 801621696
```

可以发现，单字节拷贝的效率是 3.6s 左右

2.2.2 循环展开优化

思路是把一次循环拷贝多个字节，仿照任务代码编写 copy121.s 如下：

```
1 .global memorycopy
2 memorycopy:
3     sub x1, x1, #1
4     sub x0, x0, #1
5     lp:
6     ldrb w3, [x1, #1]!
7     ldrb w4, [x1, #1]!
8     str w3, [x0, #1]!
9     str w4, [x0, #1]!
10    sub x2, x2, #2
11    cmp x2, #0
12    bne lp
13    ret
~
```

之后编译执行，结果如下：

```
[root@localhost lab2]# gcc time.c copy121.s -o m121
[root@localhost lab2]# ./m121
memroycopy time is   388873200 ns
t2: 743798800, t1: 354925600
```

从输出可得，时间大概在 3.9s 左右，可以看出只展开到两字节并不能带来效率上的显著提升。

之后仿照 2 字节的思路，再编写展开到 4 字节的文件 copy122.s ：

```
 1 .global memorycopy
 2 memorycopy:
 3     sub x1, x1, #1
 4     sub x0, x0, #1
 5     lp:
 6     ldrb w3, [x1, #1]!
 7     ldrb w4, [x1, #1]!
 8     ldrb w5, [x1, #1]!
 9     ldrb w6, [x1, #1]!
10     str w3, [x0, #1]!
11     str w4, [x0, #1]!
12     str w5, [x0, #1]!
13     str w6, [x0, #1]!
14     sub x2, x2, #4
15     cmp x2, #0
16     bne lp
17     ret
~
```

可以发现，本质上原理一样，只不过在一个 loop 中执行了四次单字节的 copy。执行结果如下：

```
[root@localhost lab2]# gcc time.c copy122.s -o m122
[root@localhost lab2]# ./m122
memroycopy time is   329639600 ns
t2: 786577296, t1: 456937696
```

可以发现，展开到四字节后，时间是 3.3s 左右，相对于不展开来说，已经取得长足进步，说明循环展开确实有效。

2.2.3 内存突发传输方式优化

我们知道，cpu 访问内存时为了平衡寄存器和内存的巨大时间差距，会在中间插入多级缓存 cache。而缓存的基本思路就是充分利用数据访问的时间局部性和空间局部性。而如今的计算机机器字长都远远高于一个字节，cache 的块大小也自然超过单字节。所以一个自然的优化思路就是一次 copy 一个块，这样最大程度地利用了缓存机制。

仿照任务代码实现单词 copy16 字节的 copy21.s ：

```
1 .global memorycopy
2 memorycopy:
3     ldp x3, x4, [x1], #16
4     stp x3, x4, [x0], #16
5     sub x2, x2, #16
6     cmp x2, #0
7     bne memorycopy
8     ret
~
```

直接编译链接执行，结果如下：

```
[root@localhost lab2]# gcc time.c copy21.s -o m21
[root@localhost lab2]# ./m21
memroycopy time is    87797904 ns
t2: 627825904, t1: 540028000
```

由图可得，时间效率大大提升，最后只花费 0.8s。

（2）ARM 指令及功能说明

本次实验中，只涉及到一些简单的 arm 指令，如基本运算指令 add、sub，和分支指令如 bne、cmp。这些都和 x86 大差不差。

但还是有些是 arm 所特有的，如 load 指令 ldrb 和 store 指令 str，这些相比于 x86 统一的 mov 指令，更明显地区分了访问寄存器和访问内存空间，这也是必要的，毕竟巨大的时间效率差别摆在那。

四、体会

这次的实验相比前几次的挑战性质的实验来说，实在是太友好了！整个实验说实话最难的部分是配环境。一开始我想用 ssh 远程控制 qemu，但是死活连不上网，最后百度才发现是配置完 eth0 网卡需要手动重启一下，也就是需要 nmcli c reload 一下。。。

但之后我 cmd 可以 ssh 进 qemu，vscode 却死活连不上。一看局域网配置好了，以太网又不行了。配个网络就配了一上午，最后放弃 vscode 还是转回用 vim 了。

之后其实就是按照任务书的教程逐步 copy，期间也遇到过一些小插曲，但无伤大雅，都是易如反掌易如反掌（。

总而言之，任务 5 还是让我学到一些关于内嵌汇编和汇编级优化的知识，也算获益匪浅。