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：本硕博 202301

学　　号：U202315752

姓　　名：陈宇航

指导教师：李海波

2024 年 10 月 29 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

任务 1 ：C 语言调用汇编代码实现累加求和

进入页面，使用 vi sum.c 后进入编辑 sum 函数的主函数：

```
#include <stdio.h>
extern int add(int num);    //声明外部调用，函数名为 add。
int main()
{
        int i,sum;
        scanf("%d",&i);  //输入初始正整数。
        sum=add(i);  //调用汇编函数 add，返回值赋值给 sum。
        printf("sum=%d\n",sum);  //将累加和输出。
        return 0;
}
```

图 3.1.1.1 sum.c 内容

保存后，使用 vi add.s 命令编辑 add 函数的汇编代码，如下：

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

图 3.1.1.2 add.s 内容

回车，保存后使用 gcc 命令链接之后，使用./sum 进行调用，输入变量 100，输出 5050：

```
"add.s" [New] 9L, 81C written
[root@localhost ~]# gcc sum.c add.s -o sum
[root@localhost ~]# ./sum
100
sum=5050
[root@localhost ~]#
```

图 3.1.1.3 结果

任务 2 ：C 语言内嵌汇编

创建 builtin.c 文件，在里面写代码，然后实现内嵌，方法和结果如下：

```
#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;
}
~
~
~
~
"builtin.c" [New] 19L, 229C written
[root@localhost test]# gcc builtin.c -o builtin
[root@localhost test]# ./builtin
100
sum is 5050
```

图 3.1.1.4 builtin 内容

任务 3：内存拷贝及优化

原始版本：普通循环，采用单步长，字节逐一拷贝。注意三种索引方式：

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

1. 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。
2. 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。
3. 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。

图 3.1.1.5 三种索引

原始版本的速度慢就慢在它完完全全是单线程的，每一个时钟周期只进行一次的循环内容：

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

图 3.1.1.6 原始版本内容

原始版本时间情况如下：

```
[root@localhost test]# ./m1
memorycopy time is  3571058896 ns
```

图 3.1.1.7 原始版本时间

优化 1：采用 2 倍循环展开进行优化。将 copy.s 文件展开两倍，然后通过一次循环复制两个字节的形式，进行两倍速的复制：

```
.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 3.1.1.8 优化 1 内容

结果如下：

```
[root@localhost test]# ./m121
memorycopy time is   236774800 ns
```

图 3.1.1.9 优化 1 时间

优化 2：采用 4 倍循环展开进行优化。将 copy.s 文件展开四倍，然后通过一次循环复制四个字节的形式，进行四倍速的复制：

```
.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 3.1.1.10 优化 2 内容

结果如下：

```
[root@localhost test]# ./m122
memorycopy time is   219335408 ns
```

图 3.1.1.11 优化 2 时间

优化 3：采用内存突发传输方式，使用 ldp、stp 指令，每次循环实现 16 个字节的拷贝：

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

图 3.1.1.12 优化 3 内容

结果如下：

```
[root@localhost test]# ./m21
memorycopy time is    75984512 ns
```

图 3.1.1.13 优化 3 时间

（2）ARM 指令及功能说明

(一)LDR（装载寄存器）：

LDR 用于将数据从内存加载到寄存器。它从指定的内存地址读取数据，并将其存储到通用寄存器中。LDR 指令支持不同的数据类型加载，可以处理不同的内存地址类型，例如字节、半字、字或双字。

LDR 指令的操作模式:

立即数偏移模式：从基地址加上一个立即数偏移量加载数据。

寄存器偏移模式：从基地址加上另一个寄存器的值作为偏移量加载数据。

预索引模式：基地址加上偏移后更新基寄存器，然后加载数据。

后索引模式：先加载数据，然后基地址加上偏移量更新基寄存器。

LDR 指令的常用格式:

LDR Wt, [Xn, #offset]

将内存中从基地址 Xn + offset 处加载 32 位数据到 32 位寄存器 Wt。

LDR Xt, [Xn, #offset]

将内存中从基地址 Xn + offset 处加载 64 位数据到 64 位寄存器 Xt。

LDRB Wt, [Xn, #offset]

从 Xn + offset 地址处加载 8 位数据（字节）到寄存器的低位 Wt。

LDRH Wt, [Xn, #offset]

从 Xn + offset 地址处加载 16 位数据（半字）到寄存器 Wt。

（二）STR（存储寄存器）

用于将寄存器中的数据存储到内存中指定的位置。STR 指令将寄存器的值写入特定内存地址，并且可以使用多种寻址模式来决定存储的内存地址位置。此指令常用于将计算结果或临时数据保存到内存，以便在稍后操作中重新访问数据。

STR 指令的操作模式：

立即数偏移模式：基地址加上一个立即数偏移量，用于确定目标内存地址。

寄存器偏移模式：基地址加上一个寄存器值作为偏移量，用于动态计算目标地址。

预索引模式：基地址加上偏移后更新基寄存器，再将数据存储到计算后的地址。

后索引模式：先将数据存储到基地址，然后更新基地址为基地址加偏移量。

STR 指令的常用格式：

STR Wt, [Xn, #offset]

将 32 位寄存器 Wt 中的数据存储到基地址 Xn + offset 指定的内存位置。

STR Xt, [Xn, #offset]

将 64 位寄存器 Xt 中的数据存储到基地址 Xn + offset 指定的内存位置。

STRB Wt, [Xn, #offset]

将 Wt 中的低 8 位字节存储到基地址 Xn + offset 的内存位置。

STRH Wt, [Xn, #offset]

将 Wt 中的低 16 位（半字）存储到基地址 Xn + offset 的内存位置。

ADD（加法运算） 是 ARM A64 架构中的一条算术指令，用于将两个操作数相加，并将结果存储在目标寄存器中。ADD 支持立即数或寄存器值作为操作数，可以进行不同的寻址和数据处理。

（三）ADD（加法运算）

是 ARM A64 架构中的一条基本算术指令，用于对两个数值进行加法运算，并将结果存储在目标寄存器中。操作数可以是一个寄存器和另一个寄存器的值，也可以是寄存器值与立即数的组合。此指令广泛用于累加操作、地址偏移计算和循环控制等场景。

ADD 指令的格式：

ADD (立即数模式):

ADD <目标寄存器>, <源寄存器>, #<立即数>{, <移位>}

目标寄存器：存储结果的寄存器。

源寄存器：提供加数的寄存器。

立即数：要加到源寄存器的固定值。

移位（可选）：立即数可以按指定的位数左移（如 0 或 12 位），以扩大立即数范围。

ADD (寄存器模式):

ADD <目标寄存器>, <源寄存器 1>, <源寄存器 2>{, <移位>}

源寄存器 1 和 源寄存器 2 的值相加并存储到 目标寄存器 中。可以对第二个源寄存器的值进行逻辑左移，以调整加数的大小

（四）SUB（减法运算）

是 ARM A64 架构中的一条算术指令，用于从一个寄存器的值中减去另一个寄存器或立即数的值，并将结果存储在目标寄存器中。与 ADD 指令类似，SUB 支持立即数或寄存器值作为

操作数，常用于差值计算、地址偏移调整和循环控制等场景。

SUB 指令的格式：

SUB（立即数模式）：

SUB <目标寄存器>, <源寄存器>, #<立即数>{, <移位>}

目标寄存器：存储结果的寄存器。

源寄存器：提供被减数的寄存器。

立即数：要从源寄存器中减去的固定值。

移位（可选）：立即数可以按指定的位数左移（如 0 或 12 位），以扩大立即数的范围。

SUB（寄存器模式）：

SUB <目标寄存器>, <源寄存器 1>, <源寄存器 2>{, <移位>}

两个源寄存器的值相减，将结果存储到目标寄存器中。可以对第二个源寄存器的值进行逻辑左移，以调整操作数的大小。

SUB 指令的操作模式

立即数模式：从寄存器中减去一个立即数。

寄存器模式：从一个寄存器中减去另一个寄存器的值，可选地对第二个寄存器的值进行左移。

移位操作：立即数或寄存器值可以进行逻辑左移，以调整操作数的大小。例如，左移 2 位相当于乘以 4。

（五）B（无条件跳转）

是 ARM A64 架构中的一种分支指令，用于执行无条件的程序跳转。该指令可以直接设置程序计数器（PC）为指定的目标地址，从而将程序控制流跳转到该地址处执行。B 指令在需要无条件跳转的场景中非常有用，例如跳过代码段、循环控制、跳转到子程序入口等。

B 指令的格式

B <label>

label：目标地址的标签，标识跳转的位置。在汇编语言中，标签是程序中的位置标记，编译器将其转换为相对于当前 PC 的偏移量。

B 指令的操作方式

直接跳转：B 指令跳转到指定的标签位置，标签可以是程序中的任意位置。该指令会将目标地址写入程序计数器 PC 中，使得下一条要执行的指令就是目标地址的指令。

相对偏移：B 指令通过相对偏移量进行跳转，相对偏移量是基于当前 PC 的，因此可以轻松跳转到前后指定范围的代码块中。

B 指令在 ARM 架构中有几个变种，用于在不同条件下跳转：

B.EQ (Equal)：当上一次运算结果等于 0 时跳转。

B.NE (Not Equal)：当上一次运算结果不等于 0 时跳转。

B.GT (Greater Than)：当上一次运算结果大于 0 时跳转。

B.LT (Less Than)：当上一次运算结果小于 0 时跳转。

B.GE (Greater or Equal)：当上一次运算结果大于或等于 0 时跳转。

B.LE (Less or Equal)：当上一次运算结果小于或等于 0 时跳转。

（六）BL（带链接的跳转）

是 ARM A64 架构中的一条跳转指令，常用于函数或子程序的调用。BL 指令通过跳转到指定地址执行子程序，同时将返回地址保存到链接寄存器（LR），从而使子程序结束时能够返回到调用该子程序的地方继续执行。这种特性使 BL 成为实现结构化编程和函数调用的核心指令。

BL 指令的格式

BL <label>

label：目标地址的标签，表示子程序的入口位置。在汇编代码中，标签指向一个特定的指令地址，编译器将其转换为相对于当前 PC（程序计数器）的偏移量。

BL 指令的工作机制

跳转到目标地址：BL 指令将程序计数器（PC）设置为指定的目标地址，使程序跳转到该地址执行子程序。

保存返回地址：在跳转前，BL 指令会将当前 PC 的下一条指令地址（也即返回地址）存入链接寄存器（LR）。这样当子程序完成时，可以通过返回指令（如 RET）从 LR 取回返回地址，将 PC 设置回调用位置，从而返回到调用处继续执行。

BL 指令的应用场景

函数调用：BL 指令最常见的用途是在汇编或低级编程中实现函数或子程序调用。通过 BL 调用子程序，并利用 RET 从子程序返回。

递归调用：在支持递归的系统中，BL 可以多次调用自身或其他函数，且返回地址会依次被保存。

中断和异常处理：BL 指令在某些情况下用于调用中断处理程序或异常处理程序，在完成中断处理后返回到正常执行流程。

（七）CBZ(比较为 0 的时候跳转)

是 ARM A64 架构中的一种条件跳转指令。它将寄存器中的值与零进行比较，如果值为零，则跳转到目标地址；如果不为零，则继续执行下一条指令。CBZ 指令可以帮助实现高效的条件判断和控制流优化，在循环控制、条件分支等场景中非常常用。

CBZ 指令的格式

CBZ <寄存器>, <label>

寄存器：要与零进行比较的寄存器。

label：目标地址标签，指定跳转的位置。当寄存器的值为零时，程序跳转到该标签处继续执行。

CBZ 指令的工作机制

零比较：CBZ 指令检查指定寄存器中的值是否为零。

条件跳转：如果寄存器的值为零，则跳转到指定的 label 地址处继续执行；否则，直接执行下一条指令。

CBZ 指令的应用场景

条件判断：用于在条件满足时跳转，例如判断变量值是否为零来决定是否进入某代码段。

循环控制：在循环中，通过判断计数器值是否为零来决定是否继续循环。

错误处理：CBZ 可以用于检查函数或操作的返回值是否为零，并在值为零时跳转到错误处理部分。

（八）MOV（移动指令）

MOV（移动指令） 是 ARM A64 架构中的一条基本数据传输指令，用于将数据从一个位置复制到另一个位置。MOV 指令支持两种主要操作模式：将一个寄存器的值复制到另一个寄存器，或将一个立即数直接加载到指定寄存器中。由于其简洁高效的特性，MOV 指令广泛应用于数据初始化、寄存器数据传输和常数赋值等场景。

MOV 指令的格式

将寄存器值复制到另一个寄存器：

MOV <目标寄存器>, <源寄存器>

目标寄存器：用于存储数据的寄存器。

源寄存器：提供要复制的值的寄存器。

将立即数加载到寄存器：

MOV <目标寄存器>, #<立即数>

目标寄存器：用于存储立即数的寄存器。

立即数：直接赋给目标寄存器的数值。

MOV 指令的操作模式

寄存器到寄存器的传输：将一个寄存器中的数据直接复制到另一个寄存器中。这种操作不会改变源寄存器中的值，只是在目标寄存器中创建一个相同的数据副本。

立即数赋值：将一个立即数（常数值）直接加载到指定寄存器中，用于初始化或设置寄存器的初始值。

（九）ROR（右旋转）

是 ARM A64 架构中的一条位操作指令，用于将寄存器中的位向右旋转指定的位数。旋转操作会将寄存器的低位“旋转”到高位，因此它是一种“循环”移位，不会丢失数据。这种操作广泛应用于数据处理、加密算法和数据压缩等需要位操作的场景。

ROR 指令的格式

ROR <目标寄存器>, <源寄存器>, #<位数>

目标寄存器：存储旋转结果的寄存器。

源寄存器：需要右旋转的寄存器。

位数：指定右旋转的位数（0 到寄存器位宽 - 1，例如 64 位寄存器最大右旋转 63 位）。

ROR 指令的工作原理

右旋转操作的基本原理是将源寄存器中的位向右循环移动到目标寄存器的指定位置，并将低位部分“绕回”到高位。例如，右旋转 1 位时，寄存器中的最低位会被移动到最高位位置，

其余各位向右移动一位，从而实现无损数据的“循环”位移。

（十）RET（返回）

是 ARM A64 架构中的一条控制流指令，用于将程序执行流程从当前子程序返回到调用该子程序的主程序中。RET 指令会跳转到链接寄存器（LR）中保存的地址，这个地址通常是调用指令后的下一条指令的位置，因此可以使程序在子程序完成操作后，准确返回到调用处继续执行。

RET 指令的格式

RET

RET：无条件返回到链接寄存器（LR）中存储的地址。该地址是调用子程序的指令（如 BL 指令）自动保存的，RET 会将程序计数器（PC）设置为该地址。

RET 指令的工作机制

返回地址存储：当调用子程序时（通常用 BL 指令），ARM 架构会将返回地址自动保存到链接寄存器（LR）中，LR 中的值即为返回地址。

返回操作：RET 指令将 PC 设置为 LR 中的地址，跳转到调用子程序的原始位置继续执行。

四、体会

在这次 ARM 指令系统的实验中，通过在 ARM 虚拟环境下编写和调试程序，我对 ARM 架构的底层操作有了更深入的理解。具体来说，实验分为几个任务，包含了 C 与汇编的混合编程、内存拷贝的优化等操作。在这些过程中，我不仅熟悉了常用的 ARM 指令（如 LDR、STR、ADD、SUB、B、BL 等）的实际应用，还对这些指令如何优化程序性能有了直观的认识。

实验过程中，我深刻体会到 ARM 指令系统的高效性，特别是在内存拷贝的优化部分，通过循环展开和内存突发传输等方法，实验显示了不同优化策略对程序执行效率的显著提升。例如，采用四倍循环展开后，内存拷贝速度显著提高，而使用 ldp、stp 指令的突发传输方式，则进一步加速了数据传输。这让我认识到，在底层编程中，通过合理利用指令和优化策略，可以在硬件限制内充分提升程序性能。

此外，汇编语言与 C 语言的混合编程也让我更清晰地理解了高层语言与底层指令的转换过程。在任务中使用的 CBZ、MOV、ROR 等指令，让我体会到在控制流和数据处理方面，汇编语言提供了灵活而强大的控制能力。在未来的学习中，这些 ARM 架构的底层知识将对我理解和优化高层编程提供很大帮助，也为我进一步研究嵌入式系统和优化算法奠定了基础。