计算机组成与结构课程

实验指导手册

版本:2.0

大连理工大学　赖晓晨

华为技术有限公司

2 C 语言与鲲鹏 920 处理器汇编语言混合编程

2.1 实验目的

本实验将通过三个部分介绍 C 调用汇编和 C 内嵌汇编两种混合编程方式以及 ARM 汇编的一些基础指令，ARM 部分指令的详细介绍以及 Linux 常用命令请参考附录中的 ARM 指令以及 Linux 常用命令。

第一部分，介绍 C 语言调用汇编实现累加和求值的方法。

第二部分，介绍 C 语言调用汇编实现更复杂的数组选择排序的方法。

第三部分，介绍 C 语言内嵌汇编的使用方法。

2.2 实验设备

- 华为鲲鹏云服务器；
- 具备网络连接的个人电脑。

2.3 实验原理

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，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。

关于 C 与汇编的混合编程的参数传递，ARM64 提供了 31 个通用寄存器，各自的用途详见表 2-1。参数传递用到的是 x0~x7 这 8 个寄存器，若参数个数大于 8 个则需要使用堆栈来传递参数。

表2-1 ARM64 通用寄存器用途

寄存器	用途
x0~x7	传递参数和返回值，多余的参数用堆栈传递，64位的返回结果保存在x0中。
X8	用于保存子程序的返回地址。
x9~x15	临时寄存器，也叫可变寄存器，无需保存。
x16~x17	子程序内部调用寄存器，使用时不需要保存，尽量不要使用。
x18	平台寄存器，它的使用与平台相关，尽量不要使用。
x19~x28	临时寄存器，子程序使用时必须保存。
x29	帧指针寄存器（FP），用于连接栈帧，使用时必须保存。
x30	链接寄存器（LR），用于保存子程序的返回地址。

2.4 实验任务操作指导

2.4.1 C 语言调用汇编实现累加和求值

首先使用远程登录工具，登录到鲲鹏 ECS 服务器上，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。

按照本实验的《实验环境搭建手册》中的步骤购买和登录鲲鹏云服务器后。（后续几个实验都是在华为鲲鹏云服务器上）

步骤 1 创建文件目录

输入命令 cd /home 进入 home 目录下，注意为了规范文件路径，后续实验文件夹都应在 home 目录下创建。

依次执行命令 mkdir sum、cd sum 创建并进入到 sum 目录。

```
cd /home
```

```
mkdir sum
cd sum
```

步骤 2 创建 sum.c 文件

执行命令 vim sum.c 编写 C 程序，按“A”进入编辑模式后输入代码。

```
vim sum.c
```

编写内容如下：

```
#include <stdio.h>
extern int add(int num);     //声明外部调用，函数名为 add。
int main()
{
        int i,sum;
        printf("请输入一个正整数：");
        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;
}
```

如图 2-1 所示：

```
#include <stdio.h>
extern int add(int num);
int main()
{
        int i,sum;
        printf("请输入一个正整数： ");
        scanf("%d",&i);
        sum=add(i);
        printf("sum=%d\n",sum);
        return 0;
}
```

图2-1 sum.c 代码

编写完成后按“ESC”键进入命令行模式，输入“:wq”后回车保存并退出编辑。

步骤 3 创建 add.s 文件

执行 vim add.s 编写所调用的汇编代码，内容如下：

```
.global add   //定义全局函数，函数名为 add。
        MOV X1,#0
add:                    //lable:add
        ADD x1,x1,x0   //将 x0+x1 的值存入 x1 寄存器。
```

```
    SUB x0,x0,#1   //将 x0-1 的值存入 x0 寄存器。
    CMP x0,#0      //比较 x0 和 0 的大小。
    BNE add        //若 x0 与 0 不相等，跳转到 add；x0=0 则继续执行。
    MOV x0,x1      //将 x1 的值存入 x0（需要 x0 来返回值）。
RET
```

如图 2-2 所示：

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

图2-2 add.s 代码

步骤 4 使用 gcc 编译生成可执行文件

GCC 是由 GNU 开发的编程语言译器。

GCC 最基本的语法是：gcc [filenames] [options]

其中[options]就是编译器所需要的参数，[filenames]给出相关的文件名称。

-c：只编译，不链接成为可执行文件，编译器只是由输入的.c 等源代码文件生成.o 为后缀的目标文件。

-o output_filename：确定输出文件的名称为 output_filename，同时这个名称不能和源文件同名。如果不给出这个选项，gcc 就给出预设的可执行文件 a.out。

-g：产生符号调试工具（GNU 的 gdb）所必要的信息，要想对源代码进行调试，我们就必须加入这个选项。

执行 gcc sum.c add.s -o sum 进行编译，然后执行./sum 运行，输入 100，返回累加和 5050，如图 2-3 所示：

```
gcc sum.c add.s -o sum
./sum
```

```
[root@ecs-01 sum]# gcc sum.c add.s -o sum
[root@ecs-01 sum]# ./sum
100
sum=5050
[root@ecs-01 sum]#
```

图2-3 编译运行

编译成功，程序执行结果正确，说明 C 程序成功调用了汇编程序。

2.4.2 C 语言调用汇编实现选择排序

本示例程序实现选择排序功能。C 代码中定义初始数组（排序前），调用汇编函数 sort 对初始数组进行排序，最后输出排序之后的数组。

步骤 1 创建文件目录

输入命令 cd /home 进入 home 目录。

依次执行命令 mkdir sort、cd sort 创建并进入 sort 文件夹。

```
cd /home
mkdir sort
cd sort
```

步骤 2 创建 sort.c 文件

执行命令 vim sort.c 编写 c 程序，

```
vim sort.c
```

编写内容如下：

```
#include<stdio.h>
extern void sort(int *a);   //声明外部函数 sort。
int main()
{
        int a[6]={66,11,44,33,55,22};   //初始数组（排序前）。
        printf("before: ");
        for(int i=0;i<6;i++)
        {
                printf("%d ",a[i]);
        }  //输出排序前数组。
        sort(a);   //调用 sort 进行选择排序。
        printf("\nsort:     ");
        for(int i=0;i<6;i++)
        {
                printf("%d ",a[i]);
        }  //输出排序后的数组。
        printf("\n");
        return 0;
}
```

如图 2-4 所示：

```c
#include<stdio.h>
extern void sort(int *a);
int main()
{
        int a[6]={66,11,44,33,55,22};
        printf("before: ");
        for(int i=0;i<6;i++)
        {
                printf("%d ",a[i]);
        }
        sort(a);
        printf("\nsort: ");
        for(int i=0;i<6;i++)
        {
                printf("%d ",a[i]);
        }
        printf("\n");
        return 0;
}
```

图2-4 sort.c

输入完成后保存并退出。

步骤 3 创建 call.s 文件

执行命令 vim call.s 编写汇编代码。

```
vim call.s
```

在汇编代码中传入的参数为数组的初始地址，存放在 x0 中。x1、x2 分别存放比较两数的地址，w3 和 w4 分别存放比较的两个数的值，x6 和 x7 为内外两层循环的计数器。

对于 ARM64 寄存器来说，x 开头代表其是一个 64 位寄存器，w 开头代表它是一个 32 位寄存器。本示例中数组定义为 int 数组，int 类型数组占四字节，32 位，因此改用 w 寄存器来存放比较值。

编写代码如下：

```
.global sort    //声明全局函数 sort
sort:
                mov x6,#6    //初始化外层循环数
loop1:
                mov x7,x6     //初始化内层循环数
                mov x2,x0
                mov x1,x0     //将初始地址存入 x1 和 x2 中
loop2:
                sub x7,x7,#1    //内层循环自减计数
                add x2,x2,#4    //x2 中的地址增加四位，即指向下一个数
                ldr w3,[x1]
                ldr w4,[x2]    //将 x1 和 x2 存放的地址指向的值分别存入 w3 和 w4 中
                cmp w3,w4    //比较 w3 和 w4 的值，判断是否需要交换
```

```
                bls next    //前者小于后者无需交换，跳转到 next
                str w3,[x2]
                str w4,[x1]    //前者大于后者数值互换
next:           cmp x7,#1    //内层循环结束判断
                bne loop2
                add x0,x0,#4    //选择排序判断下一个数应该为多少
                sub x6,x6,#1
                cmp x6,#1
                bne loop1    //外层循环跳转
ret
```

如图 2-5 所示：

```
.global sort
sort:
        mov x6,#6
loop1:
        mov x7,x6
        mov x2,x0
        mov x1,x0
loop2:
        sub x7,x7,#1
        add x2,x2,#4
        ldr w3,[x1]
        ldr w4,[x2]
        cmp w3,w4
        bls next
        str w3,[x2]
        str w4,[x1]
next:   cmp x7,#1
        bne loop2

        add x0,x0,#4
        sub x6,x6,#1
        cmp x6,#1
        bne loop1

ret
~
```

图2-5 call.s

编写完成后，保存退出。

步骤 4 编译并运行可执行文件

执行以下命令进行编译：

```
gcc sort.c call.s -o sort
```

输入命令./sort 运行程序。

```
./sort
```

如图 2-6 所示：

```
[root@ecs-01 sort]# gcc sort.c call.s -o sort
[root@ecs-01 sort]# ./sort
before: 66 11 44 33 55 22
sort:   11 22 33 44 55 66
[root@ecs-01 sort]#
```

图2-6 运行排序程序

编译成功，程序执行结果正确。

2.4.3 C 语言内嵌汇编

C 语言是无法完全代替汇编语言的，一方面是汇编语言效率比 C 语言要高，另一方面是某些特殊的指令在 C 语言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。

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
);
```

说明：

1. __asm__前后各两个下划线，并且两个下划线之间没有空格，用于声明这行代码是一个内嵌汇编表达式，是内嵌汇编代码时必不可少的关键字。
2. 关键字 volatile 前后各两个下划线，并且两个下划线之间没有空格。该关键字告诉编译器不要优化内嵌的汇编语句，如果想优化可以不加 volatile； 在很多时候，如果不使用该关键字的话，汇编语句有可能被编译器修改而无法达到预期的执行效果。
3. 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，如果不使用 clobber 列表部分，则格式可以简化为：

```
__asm__ __volatile__ (“asm code”:output：input);
```

如果不使用输入部分，则格式可以简化为：

```
__asm__ __volatile__ (“asm code”:output::clobber);
```

此时，即使输入部分为空，输出部分之后的“：”也是不能省略的。另外，输入部分和 clobber 列表部分是可选的，如果都为空，则格式可以简化为：

```
__asm__ __volatile__ (“asm code”:output);
```

4. 括号之后要以“;”结尾。

以下示例程序实现的是计算累加和功能，与 2.4.1 中示例程序的功能相同，用到了 C 语言内嵌汇编的方法，汇编指令部分与 2.4.1 相同。

步骤 1 创建文件目录

输入命令 cd /home 进入 home 目录。

执行命令 mkdir builtin 创建文件夹，输入 cd builtin 进入文件夹。

```
cd /home
mkdir builtin
cd builtin
```

步骤 2 创建 builtin.c 文件

执行命令 vim builtin.c 编写 c 程序。

```
vim builtin.c
```

编写内容如下：

```
#include <stdio.h>
int main()
{
int val;
printf("请输入一个正整数： ");
scanf("%d",&val);
__asm__ __volatile__(
                                    "MOV x1,#0 \n"
                                    "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)
                                        //=代表只写，即在汇编代码里只能改变 C 语言变量的值，而不能取它的值。
                                        // 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 语言变量即 val；
                                    :"0"(val)   //0 代表与第一个输出参数共用同一个寄存器。
                                    :
                                );
printf("sum is %d \n",val);
return 0;
    }
```

如图 2-7 所示：

```
#include <stdio.h>
int main()
{
        int val;
        printf("请输入一个正整数: ");
        scanf("%d",&val);
__asm__ __volatile__(
         "MOV X1,#0\n"
         "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2-7 builtin.c

输入完成后保存并退出。

步骤 3 编译并运行可执行文件

输入命令 gcc builtin.c -o builtin 进行编译，编译成功后输入./builtin 执行程序，输入 100，返回累加和 5050。

```
gcc builtin.c -o builtin
./builtin
```

如图 2-8 所示：

```
[root@ecs-01 builtin]# gcc builtin.c -o builtin
[root@ecs-01 builtin]# ./builtin
100
sum is 5050
[root@ecs-01 builtin]#
```

图2-8 执行 builtin

编译成功，程序执行结果正确。

继续比较，完成的话就将总比较趟数减一在跳回到 loop1 进行下一次比较。本例介绍指令迁移报告如图 8-25 所示。

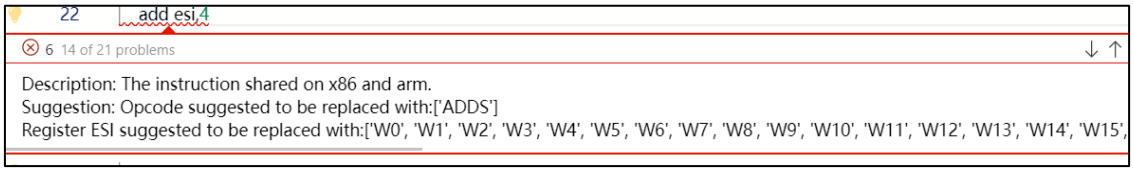

图8-25 迁移建议

最终完成所有排序后 ret 返回，根据 8.4.2 结果可以看到所有排序都成功实现，实验完成。

9 附录 1：Linux 常用命令

9.1 基本命令

9.1.1 关机和重启

命令：

```
shutdown -h now                    #立刻关机
shutdown -h 5                      #5 分钟后关机
poweroff                           #立刻关机
```

重启

```
shutdown -r now                    #立刻重启
shutdown -r 5                      #5 分钟后重启
reboot                             #立刻重启
```

9.1.2 帮助命令

命令： --help

```
shutdown –help                    #查看关机命令帮助信息
ifconfig  --help                  #查看网卡信息
man                               #（命令说明书）
man shutdown
```

注意：man shutdown 打开命令说明书之后，使用按键 q 退出

9.2 2 目录操作命令

9.2.1 目录切换命令

命令：cd 目录

```
cd /                              #切换到根目录
cd /usr                           #切换到根目录下的 usr 目录
cd ../                            #切换到上一级目录 或者  cd ..
cd ~                              #切换到 home 目录
cd -                              #切换到上次访问的目录
```

9.2.2 目录查看命令

命令：ls [-al]

```
Ls                                #查看当前目录下的所有目录和文件
ls -a                             #查看当前目录下的所有目录和文件（包括隐藏的文件）
ls -l 或 ll                       #列表查看当前目录下的所有目录和文件（显示更多信息）
ls /dir                           #查看指定目录下的所有目录和文件   如：ls /usr
```

9.2.3 目录操作命令

9.2.3.1 创建目录

命令：mkdir 目录

```
mkdir    aaa          # 在当前目录下创建一个名为 aaa 的目录
mkdir    /usr/aaa     # 在指定目录下创建一个名为 aaa 的目录
```

9.2.3.2 删除目录或文件

命令：rm [-rf] 目录

删除文件：

```
rm  文件                    #删除当前目录下的文件
rm -f  文件                     #删除当前目录的的文件（不询问）
#删除目录：
rm -r aaa                       #递归删除当前目录下的 aaa 目录
rm -rf aaa                      #递归删除当前目录下的 aaa 目录（不询问）
#全部删除：
rm -rf *                        #将当前目录下的所有目录和文件全部删除
rm -rf /*                   #【慎用！】将根目录下的所有文件全部删除
```

注意：rm 不仅可以删除目录，也可以删除其他文件或压缩包，为了方便大家的记忆，无论删除任何目录或文件，都直接使用 rm -rf 目录/文件/压缩包

9.2.3.3 目录修改

重命名目录

```
命令：mv 当前目录  新目录
```

示例：mv aaa bbb　　#将目录 aaa 改为 bbb

注意：mv 的语法不仅可以对目录进行重命名而且也可以对各种文件，压缩包等进行重命名的操作。

剪切目录

```
命令：mv 目录名称 目录的新位置
```

示例：mv /usr/tmp/aaa /usr　　#将/usr/tmp 目录下的 aaa 目录剪切到 /usr 目录下面

注意：mv 语法不仅可以对目录进行剪切操作，对文件和压缩包等都可执行剪切操作。

拷贝目录

```
命令：cp -r 目录名称 目录拷贝的目标位置   -r 代表递归
```

示例：　cp /usr/tmp/aaa　/usr　　#将/usr/tmp 目录下的 aaa 目录复制到 /usr 目录下面

注意：cp 命令不仅可以拷贝目录还可以拷贝文件，压缩包等，拷贝文件和压缩包时不用写-r 递归。

9.2.3.4 目录搜索

```
命令：find 目录 参数 文件名称
```

示例：find /usr/tmp -name 'a*'　　#查找/usr/tmp 目录下的所有以 a 开头的目录或文件

9.3 文件操作命令

9.3.1 新建文件

```
命令：touch  文件名
```

示例：touch　aa.txt　　#在当前目录创建一个名为 aa.txt 的文件

9.3.2 删除文件

```
命令：rm -rf  文件名
```

9.3.3 修改文件

打开文件

```
vi  文件名
```

示例：vi aa.txt 或者 vim aa.txt　　#打开当前目录下的 aa.txt 文件

若文件不存在则新建文件并打开

注意：使用 vi 编辑器打开文件后，并不能编辑，因为此时处于命令模式，点击键盘 i/a/o 进入编辑模式。

- 编辑文件

使用 vi 编辑器打开文件后点击按键：i ， a 或者 o 即可进入编辑模式。

i：在光标所在字符前开始插入

a：在光标所在字符后开始插入

o：在光标所在行的下面另起一新行插入

- 保存文件：

第一步：ESC　进入命令行模式

第二步：进入底行模式

第三步：wq　　#保存并退出编辑

- 取消编辑：

第一步：ESC　进入命令行模式

第二步：：进入底行模式

第三步：q!　　#撤销本次修改并退出编辑

9.3.4 查看文件

文件的查看命令：cat/more/less/tail

cat：看最后一屏

示例：使用 cat 查看/etc/sudo.conf 文件，只能显示最后一屏内容。

cat sudo.conf

more：百分比显示

示例：使用 more 查看/etc/sudo.conf 文件，可以显示百分比，回车可以向下一行，空格可以向下一页，q 可以退出查看

more sudo.conf

less：翻页查看

示例：使用 less 查看/etc/sudo.conf 文件，可以使用键盘上的 PgUp 和 PgDn 向上和向下翻页，q 结束查看

less sudo.conf

tail：指定行数或者动态查看

示例：使用 tail -10 查看/etc/sudo.conf 文件的后 10 行，Ctrl+C 结束

tail -10 sudo.conf

10 附录 2：ARM 指令

10.1 LDR 字数据加载指令

LDR 指令的格式为：

LDR{条件} 目的寄存器，<存储器地址>

LDR 指令用于从存储器中将一个 32 位的字数据传送到目的寄存器中。该指令通常用于从存储器中读取 32 位的字数据到通用寄存器，然后对数据进行处理。当程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。

指令示例：

```
LDR   R0，[R1]                    ；将存储器地址为 R1 的字数据读入寄存器 R0。
LDR   R0，[R1，R2]                ；将存储器地址为 R1+R2 的字数据读入寄存器 R0。
LDR   R0，[R1，＃8]               ；将存储器地址为 R1+8 的字数据读入寄存器 R0。
LDR   R0，[R1，R2]！              ；将存储器地址为 R1+R2 的字数据读入寄存器 R0，并将新地址 R1 + R2 写入 R1。
LDR   R0，[R1，＃8]！             ；将存储器地址为 R1+8 的字数据读入寄存器 R0，并将新地址 R1 + 8 写入 R1。
LDR   R0，[R1]，R2                ；将存储器地址为 R1 的字数据读入寄存器 R0，并将新地址 R1 + R2 写入 R1。
LDR   R0，[R1，R2，LSL＃2]!       ；将存储器地址为 R1 + R2×4 的字数据读入寄存器 R0，并将新地址 R1 + R2×4 写入 R1。
LDR    R0，[R1]，R2，LSL＃2       ；将存储器地址为 R1 的字数据读入寄存器 R0，并将新地址 R1 + R2×4 写入 R1。
```

10.2 LDRB 字节数据加载指令

LDRB 指令的格式为：

LDR{条件}B 目的寄存器，<存储器地址>

LDRB 指令用于从存储器中将一个 8 位的字节数据传送到目的寄存器中，同时将寄存器的高 24 位清零。该指令通常用于从存储器中读取 8 位的字节数据到通用寄存器，然后对数据进行处理。当程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。

指令示例：

```
LDRB R0，[R1]                     ；将存储器地址为 R1 的字节数据读入寄存器 R0，并将 R0 的高 24 位清零。
LDRB R0，[R1，＃8]!               ；将存储器地址为 R1 + 8 的字节数据读入寄存器 R0，并将新地址 R1 + 8 写入 R1。
```

10.3 LDRH 半字数据加载指令

LDRH 指令的格式为：

LDR{条件}H 目的寄存器，<存储器地址>

LDRH 指令用于从存储器中将一个 16 位的半字数据传送到目的寄存器中，同时将寄存器的高 16 位清零。该指令通常用于从存储器中读取 16 位的半字数据到通用寄存器，然后对数据进行处理。当程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。

指令示例：

LDRH R0，[R1] ；将存储器地址为 R1 的半字数据读入寄存器 R0，并将 R0 的高 16 位清零。

LDRH R0，[R1，＃8] ；将存储器地址为 R1 + 8 的半字数据读入寄存器 R0，并将 R0 的高 16 位清零。

LDRH R0，[R1，R2] ；将存储器地址为 R1 + R2 的半字数据读入寄存器 R0，并将 R0 的高 16 位清零。

10.4 STR 字数据存储指令

STR 指令的格式为：

STR{条件} 源寄存器，<存储器地址>

STR 指令用于从源寄存器中将一个 32 位的字数据传送到存储器中。该指令在程序设计中比较常用，且寻址方式灵活多样，使用方式可参考指令 LDR。

指令示例：

STR R0，[R1]，＃8 ；将 R0 中的字数据写入以 R1 为地址的存储器中，并将新地址 R1 + 8 写入 R1。

STR R0，[R1，＃8] ；将 R0 中的字数据写入以 R1 + 8 为地址的存储器中。

STR R0，[R1，＃8]! ；将 R0 中的字数据写入以 R1 为地址的存储器中，并将新地址 R1 + 8 写入 R1。

10.5 STRB 字节数据存储指令

STRB 指令的格式为：

STR{条件}B 源寄存器，<存储器地址>

STRB 指令用于从源寄存器中将一个 8 位的字节数据传送到存储器中。该字节数据为源寄存器中的低 8 位。

指令示例：

STRB R0，[R1] ；将寄存器 R0 中的字节数据写入以 R1 为地址的存储器中。

STRB R0，[R1，＃8] ；将寄存器 R0 中的字节数据写入以 R1 + 8 为地址的存储器中。

10.6 STRH 半字数据存储指令

STRH 指令的格式为：

STR{条件}H 源寄存器，<存储器地址>

STRH 指令用于从源寄存器中将一个 16 位的半字数据传送到存储器中。该半字数据为源寄存器中的低 16 位。

指令示例：

STRH R0，[R1] ；将寄存器 R0 中的半字数据写入以 R1 为地址的存储器中。

STRH R0，[R1，＃8] ；将寄存器 R0 中的半字数据写入以 R1 + 8 为地址的存储器中。

10.7 LDP/STP 指令

是 LDP/STP 的衍生，可以同时读/写两个寄存器，并访问 16 的字节的内存数据，

指令示例：

LDP x3,x4,[x1,#16] ；读取 x1+16 地址后的 16 个字节的数据写入 x3、x4 寄存器中。

LDP x3,x4,[x1],#16 ；读取 x1 地址后的 16 个字节的数据写入 x3、x4 寄存器中，并更新 x1=x1+16。

LDP x9,x10,[x1,#64]! ；读取 x1+64 地址后的 16 个字节的数据写入 x9、x10 寄存器中。并将新地址 x1 + 64 写入 x1。

STP x3,x4,[x0,#16] ；将 x3、x4 中的数据写入以 x0+16 地址后的 16 个字节地址中

STP x3,x4,[x0],#16 ；将 x3、x4 中的数据写入以 x0 地址后的 16 个字节地址中并更新 x0=x0+16。

STP x9,x10,[x0,#64]! ；将 x9、x10 中的数据写入以 x0+64 地址后的 16 个字节地址 中，并将新地址 x0 + 64 写入 x0。

说明：《计算机组成与结构》课程配套实验手册中的实验内容由大连理工的赖晓晨老师提供，华为公司负责实验手册文档的编写。