实验一 ARM 指令

潘盛琪 3170105737

实验目的:

1. 深入理解 ARM 指令和 Thumb 指令的区别和编译选项;
2. 深入理解某些特殊的 ARM 指令，理解如何编写 C 代码来得到这些指令;
3. 深入理解 ARM 的 BL 指令和 C 函数的堆栈保护;
4. 深入理解如何实现 C 和汇编函数的互相调用。

实验器材:

交叉编译软件 CUBEIDE

实验步骤:

1. 生成了 Thumb 指令还是 ARM 指令：如何通过编译参数改变，相同的程序，ARM 和 Thumb 编译的结果有何不同，如指令本身和整体目标代码的大小等;

   编译器设置：

优化选项	-O2

   C 代码：

```c
int main(void)
{
    int a = 1;
}
```

   汇编代码：

```
.Ltext0:
    .cfi_sections    .debug_frame
    .section    .text.startup.main,"ax",%progbits
    .align  1
    .p2align 2,,3
    .global main            //main 为程序入口
    .syntax unified         //用统一汇编语法
    .thumb                  //用 thumb 指令
    .thumb_func
    .fpu softvfp
    .type   main, %function
main:                       //主程序开始
.LFB3:
    .file 1 "../Src/main.c"
    .loc 1 26 0
    .cfi_startproc
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 0, uses_anonymous_args = 0
    @ link register save eliminated.
.LVL0:
    .loc 1 28 0
    movs    r0, #0
    bx  lr
```

反汇编代码：

```
08000170 <main>:
 8000170:   2000        movs    r0, #0
 8000172:   4770        bx  lr
```

结果分析：
由得到的汇编代码中.thumb 可知，得到的是汇编代码，
由反汇编的结果也可以看到，每条指令长度为 16 位，可以判断生成了 Thumb 指令（事实上在后续的实验中发现，printf()等指令能生成 32 位汇编指令，因此实际上汇编指令为 16 位与 32 位混合的 Thumb2 指令集）

2. 对于 ARM 指令,能否产生条件执行的指令;
   C 代码：

```
int a = 0, b = 1, ans = 100;
int main(void)
{
    if(a > b)
    {
        ans++;
    }
    else
    {
        ans--;
    }
}
```

编译器设置：

优化选项	-O2

汇编代码

```
1.  main:
2.  .LFB0:
3.      .file 1 "../Src/main.c"
4.      .loc 1 26 0
5.      .cfi_startproc
6.      @ args = 0, pretend = 0, frame = 0
7.      @ frame_needed = 0, uses_anonymous_args = 0
8.      @ link register save eliminated.
9.      .loc 1 27 0
10.     ldr r3, .L4             //从.L4 载入第一个全局变量 a
11.     ldr r2, [r3]            //r2=r3
12.     ldr r3, .L4+4           //从.L4+4 载入第二个全局变量
13.     ldr r3, [r3]            //r3=r3
14.     cmp r2, r3              //r2 与 r3 比较
15.     .loc 1 29 0
16.     ldr r2, .L4+8           //从.L4+8 载入第三个全局变量
17.     ldr r3, [r2]            //r3=r2
18.     ite gt                  //下面两条指令是条件指令
19.     addgt   r3, r3, #1      //若符号大于则 r3=r3+1
20.     .loc 1 33 0
21.     addle   r3, r3, #-1     //若符号小于则 r3=r3-1
22.     str r3, [r2]            //r2=r3
23.     .loc 1 35 0
24.     movs    r0, #0          //r0=0
25.     bx  lr                  //return
26. .L5:
27.     .align  2
28. .L4:
29.     .word   .LANCHOR0
30.     .word   .LANCHOR1
31.     .word   .LANCHOR2
32.     .cfi_endproc
```

结果分析：

可以看到在第 18-21 行生成了条件执行指令

3. 设计 C 的代码场景,观察是否产生了寄存器移位寻址;

C 代码：

```c
int op(int a, int b)
{
    int ans = a + b * 4;
    return ans;
}
int main(void)
{
    int ans = op(1, 2);
}
```

编译器设置：

优化选项	-O2

汇编代码：

```
op:
.LFB3:
    .file 1 "../Src/main.c"
    .loc 1 27 0
    .cfi_startproc
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 0, uses_anonymous_args = 0
    @ link register save eliminated.
.LVL0:
    .loc 1 30 0
    add r0, r0, r1, lsl #2     // r0 = r0 + r1 << 2;
.LVL1:
    bx  lr                     // return
    .cfi_endproc
```

结果分析：
这里可以看到，以函数的形式调用 ans = a + 4 * b，并且选择-02 编译优化是可以得到寄存器移位寻址的汇编代码的。然而若无编译优化则无法得到寄存器移位寻址的汇编代码。

4. 设计 C 的代码场景,观察一个复杂的 32 位数是如何装载到寄存器的;
C 代码：

```
1. int load()
2. {
3.     int a = 65538;
4.     return a;
5. }
6. int main(void)
7. {
8.     load();
9. }
```

编译器设置：

优化选项	-O0

汇编代码：

```
load:
.LFB0:
    .file 1 "../Src/main.c"
    .loc 1 25 0
    .cfi_startproc
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    @ link register save eliminated.
    push    {r7}            //压栈
    .cfi_def_cfa_offset 4
    .cfi_offset 7, -4
    sub sp, sp, #12         //调整栈指针，留出函数的栈空间
    .cfi_def_cfa_offset 16
    add r7, sp, #0          //将栈指针存入 r7
    .cfi_def_cfa_register 7
    .loc 1 26 0
    ldr r3, .L3             //从内存中读取复杂 32 位数
    str r3, [r7, #4]        //将复杂 32 位数存入函数堆栈中
    .loc 1 27 0
    ldr r3, [r7, #4]        //将该 32 位数从函数堆栈中读到 r3
    .loc 1 28 0
    mov r0, r3              //r3 赋给 r0
    adds    r7, r7, #12     //调整栈指针，销毁函数栈空间
    .cfi_def_cfa_offset 4
    mov sp, r7              //恢复栈指针
    .cfi_def_cfa_register 13
    @ sp needed
    pop {r7}                //出栈
    .cfi_restore 7
    .cfi_def_cfa_offset 0
    bx  lr                  //返回
.L4:
    .align  2
.L3:                        //复杂 32 位数存储在这里
    .word   65538
    .cfi_endproc
```

结果分析：
复杂 32 位数和普通数字存到寄存器中的区别就是：简单数字直接 mov {Rx} #读取一个立即数就可以，但复杂 32 位数需要利用.word 在内存中定义这个数，再读入寄存器中。
在测试的时候，我尝试过首先尝试了 65536，但发现和普通数字的存储没有区别，之后尝试 65537，还是一样。直到 65538 开始，才不是直接用立即数存入寄存器中，这一点我感觉非常奇怪。

5. 写一个 C 的多重函数调用的程序,观察和分析:a. 调用时的返回地址在哪里?b. 传入的参数在哪里?c. 本地变量的堆栈分配是如何做的?d. 寄存器是 caller 保存还是 callee 保存?是全体保存还是部分保存?

   C 代码：

```c
int fun2(int num2)
{
    return num2*2;
}
int fun1(int num1, int num2)
{
    return num1+fun2(num2);
}
int main(void)
{
    int ans = fun1(111,22);
    return ans;
}
```

   汇编代码：

```
fun2:
.LFB0:
    .file 1 "../Src/main.c"
    .loc 1 26 0
    .cfi_startproc
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    @ link register save eliminated.
    push    {r7}                    //保护现场
    .cfi_def_cfa_offset 4
    .cfi_offset 7, -4
    sub sp, sp, #12                 //为当前函数局部变量预留空间
    .cfi_def_cfa_offset 16
    add r7, sp, #0                  //r7 指向栈顶
    .cfi_def_cfa_register 7
    str r0, [r7, #4]                //读取参数，存入栈中
    .loc 1 27 0
    ldr r3, [r7, #4]                //将 num2 读入 r3
    lsls    r3, r3, #1              //r3=r3*2
    .loc 1 28 0
    mov r0, r3                      //结果传入 r0
    adds    r7, r7, #12             //r7 指向 fun1 栈顶
    .cfi_def_cfa_offset 4
    mov sp, r7                      //sp=r7
    .cfi_def_cfa_register 13
    @ sp needed
    pop {r7}                        //r7 出栈
    .cfi_restore 7
    .cfi_def_cfa_offset 0
    bx  lr                          //返回
    .cfi_endproc
```

```
fun1:
.LFB1:
    .loc 1 30 0
    .cfi_startproc
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    push    {r7, lr}            //保护现场
    .cfi_def_cfa_offset 8
    .cfi_offset 7, -8
    .cfi_offset 14, -4
    sub sp, sp, #16             //为当前函数局部变量预留空间
    .cfi_def_cfa_offset 24
    add r7, sp, #0              //r7 指向栈顶
    .cfi_def_cfa_register 7
    str r0, [r7, #4]            //读取参数 num1
    str r1, [r7]                //读取参数 num2
    .loc 1 31 0
    movs    r3, #33             //定义本地变量
    str r3, [r7, #12]           //存入栈中
    .loc 1 32 0
    movs    r3, #44             //定义本地变量
    str r3, [r7, #8]            //存入栈中
    .loc 1 33 0
    ldr r0, [r7]                //利用 r0 传递参数 num2
    bl  fun2                    //调用 fun2
    mov r2, r0                  //将返回结果 r0 存入 r2
    ldr r3, [r7, #4]            //将参数 num1 存入 r3
    add r3, r3, r2              //r3=r3+r2
    .loc 1 34 0
    mov r0, r3                  //r3 存入 r0 作为返回值
    adds    r7, r7, #16         //r7 指向 main 栈顶
    .cfi_def_cfa_offset 8
    mov sp, r7                  //sp=r7
    .cfi_def_cfa_register 13
    @ sp needed
    pop {r7, pc}                //r7，pc 出栈，返回
    .cfi_endproc
```

```
main:
.LFB2:
    .loc 1 34 0
    .cfi_startproc
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    push    {r7, lr}
    .cfi_def_cfa_offset 8
    .cfi_offset 7, -8
    .cfi_offset 14, -4
    sub sp, sp, #8              //为当前函数的局部变量分配栈空间
    .cfi_def_cfa_offset 16
    add r7, sp, #0
    .cfi_def_cfa_register 7
    .loc 1 35 0
    movs    r1, #22             //r1 传第二个参数
    movs    r0, #111            //r0 传第一个参数
    bl  fun1                    //调用 fun1
    str r0, [r7, #4]            //将返回值 r0 存入栈中
    .loc 1 36 0
    ldr r3, [r7, #4]            //从栈中读取数据到 r3
    .loc 1 37 0
    mov r0, r3                  //r0=r3 进行返回
    adds    r7, r7, #8          //r7=r7+8
    .cfi_def_cfa_offset 8
    mov sp, r7                  //sp=r7
    .cfi_def_cfa_register 13
    @ sp needed
    pop {r7, pc}                //return
    .cfi_endproc
```

结果分析：

（1） 调用时的返回地址存储在 lr 中，在调用函数时压栈，在函数返回时出栈存入 pc。在 fun2 中由于不再调用函数，因此无需将 lr 压栈，在返回时利用 bx 即可

（2） 传入的参数在 r0 和 r1 中，若有更多参数则会存入 r2，r3，若超过四个则存入栈中

（3） 本地变量的堆栈分配由小地址到大地址分别为传入参数和本地变量，且越早定义地址越大

（4） r7 是 callee 保存

（5） 部分保存

6. MLA 是带累加的乘法,尝试要如何写 C 的表达式能编译得到 MLA 指令。

C 代码：

```c
int fun(int a, int b, int c)
{
    int ans = a * b + c;
    return ans;
}
int main(void)
{
    int ans = fun(11, 22, 33);
    return ans;
}
```

编译器设置：

优化选项	-O1

汇编代码：

```
fun:
.LFB0:
    .file 1 "../Src/main.c"
    .loc 1 25 0
    .cfi_startproc
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 0, uses_anonymous_args = 0
    @ link register save eliminated.
.LVL0:
    .loc 1 28 0
    mla r0, r1, r0, r2
.LVL1:
    bx  lr
    .cfi_endproc
```

结果分析：

在第 11 行生成了 MLA 指令

7. BIC 是对某一个比特清零的指令，尝试要如何写 C 的表达式能编译得到 BIC 指令。

C 代码:

```c
int fun(int a, int b)
{
    int ans = a & (~b);
    return ans;
}
int main(void)
{
    int ans = fun(0xff,0xff);
    return ans;
}
```

汇编代码:

```
fun:
.LFB0:
    .file 1 "../Src/main.c"
    .loc 1 25 0
    .cfi_startproc
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 0, uses_anonymous_args = 0
    @ link register save eliminated.
.LVL0:
    .loc 1 28 0
    bic r0, r0, r1     //将 r0 按位清零
.LVL1:
    bx  lr
    .cfi_endproc
```

结果分析:
在第 11 行生成了 BIC 指令