Assignment 003: Lab 3：ARM指令

葛现隆 3120102146

一、 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。

二、 实验器材

- 树莓派板一块；
- 5V/1A电源一个；
- microUSB线一根；

三、 实验步骤

使用交叉编译工具或本机编译工具，通过 C 代码和反汇编工具研究:
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;

程序代码

```
#include <stdio.h>
int main(){
        int i=0;
        i++;
        return 0;
}
```

gcc默认编译

```
pi@raspberrypi ~ $ vim arm_1.c
pi@raspberrypi ~ $ gcc -c arm_1.c
pi@raspberrypi ~ $ objdump -d arm_1.o
```

查看反汇编结果，默认为ARM

```
arm_1.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd00c        sub     sp, sp, #12
   c:   e3a03000        mov     r3, #0
  10:   e50b3008        str     r3, [fp, #-8]
  14:   e51b3008        ldr     r3, [fp, #-8]
  18:   e2833001        add     r3, r3, #1
  1c:   e50b3008        str     r3, [fp, #-8]
  20:   e3a03000        mov     r3, #0
  24:   e1a00003        mov     r0, r3
  28:   e28bd000        add     sp, fp, #0
  2c:   e8bd0800        ldmfd   sp!, {fp}
  30:   e12fff1e        bx      lr
```

gcc编译时添加-c mthumb， 即可使用Thumb编译（指令长度为16位）。

```
pi@raspberrypi ~ $ gcc -c -mthumb -msoft-float arm_1.c
pi@raspberrypi ~ $ objdump -d arm_1.o

arm_1.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   b580            push    {r7, lr}
   2:   b082            sub     sp, #8
   4:   af00            add     r7, sp, #0
   6:   2300            movs    r3, #0
   8:   607b            str     r3, [r7, #4]
   a:   687b            ldr     r3, [r7, #4]
   c:   3301            adds    r3, #1
   e:   607b            str     r3, [r7, #4]
  10:   2300            movs    r3, #0
  12:   1c18            adds    r0, r3, #0
  14:   46bd            mov     sp, r7
  16:   b002            add     sp, #8
  18:   bd80            pop     {r7, pc}
  1a:   46c0            nop                     ; (mov r8, r8)
```

2 对于 ARM 指令,能否产生条件执行的指令;
代码如下:

```
#include <stdio.h>

int main(){
        int i=10;
        if(i>0)
                i=9;
        if(i<0)
                i=8;
        return 0;
}
```

ARM编译结果如下，出现ble分支指令，ARM支持条件执行指令。

```
pi@raspberrypi ~/arm $ objdump -d arm_branch.o

arm_branch.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd00c        sub     sp, sp, #12
   c:   e3a0300a        mov     r3, #10
  10:   e50b3008        str     r3, [fp, #-8]
  14:   e51b3008        ldr     r3, [fp, #-8]
  18:   e3530000        cmp     r3, #0
  1c:   da000001        ble     28 <main+0x28>
  20:   e3a03009        mov     r3, #9
  24:   e50b3008        str     r3, [fp, #-8]
  28:   e51b3008        ldr     r3, [fp, #-8]
  2c:   e3530000        cmp     r3, #0
  30:   aa000001        bge     3c <main+0x3c>
  34:   e3a03008        mov     r3, #8
  38:   e50b3008        str     r3, [fp, #-8]
  3c:   e3a03000        mov     r3, #0
  40:   e1a00003        mov     r0, r3
  44:   e28bd000        add     sp, fp, #0
  48:   e8bd0800        ldmfd   sp!, {fp}
  4c:   e12fff1e        bx      lr
```

3 设计 C 的代码场景,观察是否产生了寄存器移位寻址;
C代码如下

```
#include <stdio.h>
int main(){
        int i=1;
        int j;
        j=i<<2;
        return 0;
}
```

ARM汇编指令如下，对于j=i<<2，对应但指令是lsl r3, r3, #2，使用了lsl移位指令。

```
pi@raspberrypi ~/arm $ vim arm_shift.c
pi@raspberrypi ~/arm $ gcc -c arm_shift.c
pi@raspberrypi ~/arm $ objdump -d arm_shift.o

arm_shift.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd00c        sub     sp, sp, #12
   c:   e3a03001        mov     r3, #1
  10:   e50b3008        str     r3, [fp, #-8]
  14:   e51b3008        ldr     r3, [fp, #-8]
  18:   e1a03103        lsl     r3, r3, #2
  1c:   e50b300c        str     r3, [fp, #-12]
  20:   e3a03000        mov     r3, #0
  24:   e1a00003        mov     r0, r3
  28:   e28bd000        add     sp, fp, #0
  2c:   e8bd0800        ldmfd   sp!, {fp}
  30:   e12fff1e        bx      lr
```

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代码如下:

```
#include <stdio.h>
int main(){
        unsigned int i =0x87654321;
        return 0;
}
```

ARM汇编指令如下，通过ldr指令，将32位大整数赋值到r3寄存器。

```
pi@raspberrypi ~/arm $ gcc -c arm_32_bit.c
pi@raspberrypi ~/arm $ objdump -d arm_32_bit.o

arm_32_bit.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd00c        sub     sp, sp, #12
   c:   e59f3014        ldr     r3, [pc, #20]   ; 28 <main+0x28>
  10:   e50b3008        str     r3, [fp, #-8]
  14:   e3a03000        mov     r3, #0
  18:   e1a00003        mov     r0, r3
  1c:   e28bd000        add     sp, fp, #0
  20:   e8bd0800        ldmfd   sp!, {fp}
  24:   e12fff1e        bx      lr
  28:   87654321        .word   0x87654321
```

5 写一个 C 的多重函数调用的程序,观察和分析:
C代码如下:

```
#include <stdio.h>
int f1(int a, int b){
        int i=0;
        i=f2(a,b);
        return i;
}

int f2(int a, int b){
        int i=1;
        i=f2(a, b);
        return i;
}

int f3(int a, int b){
        int i=2;
        i=a+b;
        return i;
}

int main(){
        f1(4, 5);
        return 0;
}
```

ARM汇编指令代码如下；

```
pi@raspberrypi ~/arm $ objdump -d arm_multi_call.o

arm_multi_call.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <f1>:
   0:	e92d4800 	push	{fp, lr}
   4:	e28db004 	add	fp, sp, #4
   8:	e24dd010 	sub	sp, sp, #16
   c:	e50b0010 	str	r0, [fp, #-16]
  10:	e50b1014 	str	r1, [fp, #-20]
  14:	e3a03000 	mov	r3, #0
  18:	e50b3008 	str	r3, [fp, #-8]
  1c:	e51b0010 	ldr	r0, [fp, #-16]
  20:	e51b1014 	ldr	r1, [fp, #-20]
  24:	ebfffffe 	bl	3c <f2>
  28:	e50b0008 	str	r0, [fp, #-8]
  2c:	e51b3008 	ldr	r3, [fp, #-8]
  30:	e1a00003 	mov	r0, r3
  34:	e24bd004 	sub	sp, fp, #4
  38:	e8bd8800 	pop	{fp, pc}

0000003c <f2>:
  3c:	e92d4800 	push	{fp, lr}
  40:	e28db004 	add	fp, sp, #4
  44:	e24dd010 	sub	sp, sp, #16
  48:	e50b0010 	str	r0, [fp, #-16]
  4c:	e50b1014 	str	r1, [fp, #-20]
  50:	e3a03001 	mov	r3, #1
  54:	e50b3008 	str	r3, [fp, #-8]
  58:	e51b0010 	ldr	r0, [fp, #-16]
  5c:	e51b1014 	ldr	r1, [fp, #-20]
  60:	ebfffffe 	bl	3c <f2>
  64:	e50b0008 	str	r0, [fp, #-8]
  68:	e51b3008 	ldr	r3, [fp, #-8]
  6c:	e1a00003 	mov	r0, r3
  70:	e24bd004 	sub	sp, fp, #4
  74:	e8bd8800 	pop	{fp, pc}
00000078 <f3>:
  78:	e52db004 	push	{fp}		; (str fp, [sp, #-4]!)
  7c:	e28db000 	add	fp, sp, #0
  80:	e24dd014 	sub	sp, sp, #20
  84:	e50b0010 	str	r0, [fp, #-16]
  88:	e50b1014 	str	r1, [fp, #-20]
  8c:	e3a03002 	mov	r3, #2
  90:	e50b3008 	str	r3, [fp, #-8]
  94:	e51b2010 	ldr	r2, [fp, #-16]
  98:	e51b3014 	ldr	r3, [fp, #-20]
  9c:	e0823003 	add	r3, r2, r3
  a0:	e50b3008 	str	r3, [fp, #-8]
  a4:	e51b3008 	ldr	r3, [fp, #-8]
  a8:	e1a00003 	mov	r0, r3
  ac:	e28bd000 	add	sp, fp, #0
  b0:	e8bd0800 	ldmfd	sp!, {fp}
  b4:	e12fff1e 	bx	lr

000000b8 <main>:
  b8:	e92d4800 	push	{fp, lr}
  bc:	e28db004 	add	fp, sp, #4
  c0:	e3a00004 	mov	r0, #4
  c4:	e3a01005 	mov	r1, #5
  c8:	ebfffffe 	bl	0 <f1>
  cc:	e3a03000 	mov	r3, #0
  d0:	e1a00003 	mov	r0, r3
  d4:	e8bd8800 	pop	{fp, pc}
```

a 调用时的返回地址在哪里?
可见，返回地址存放在lr寄存器中。
b 传入的参数在哪里?
两个参数时，参数传入r0,r1中，当参数个数大于4个时，多余参数放在堆栈中。
c 本地变量的堆栈分配是如何做的?
本地变量放在了堆栈高地址。
d 寄存器是 caller 保存还是 callee 保存?是全体保存还是部分保存?
R0, R1, R2, R3由caller保存，其余由callee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语言代码:

```
#include <stdio.h>
int f(int a, int b, int c){
        return a*b+c;
}

int main()
{
        f(1,2,3);
        return 0;
}
```

ARM汇编指令，直接编译可知，并不能得到mla指令；

```
pi@raspberrypi ~/arm $ vim arm_mla.c
pi@raspberrypi ~/arm $ gcc -c arm_mla.c
pi@raspberrypi ~/arm $ objdump -d arm_mla.o

arm_mla.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <f>:
   0:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd014        sub     sp, sp, #20
   c:   e50b0008        str     r0, [fp, #-8]
  10:   e50b100c        str     r1, [fp, #-12]
  14:   e50b2010        str     r2, [fp, #-16]
  18:   e51b3008        ldr     r3, [fp, #-8]
  1c:   e51b200c        ldr     r2, [fp, #-12]
  20:   e0020392        mul     r2, r2, r3
  24:   e51b3010        ldr     r3, [fp, #-16]
  28:   e0823003        add     r3, r2, r3
  2c:   e1a00003        mov     r0, r3
  30:   e28bd000        add     sp, fp, #0
  34:   e8bd0800        ldmfd   sp!, {fp}
  38:   e12fff1e        bx      lr

0000003c <main>:
  3c:   e92d4800        push    {fp, lr}
  40:   e28db004        add     fp, sp, #4
  44:   e3a00001        mov     r0, #1
  48:   e3a01002        mov     r1, #2
  4c:   e3a02003        mov     r2, #3
  50:   ebfffffe        bl      0 <f>
  54:   e3a03000        mov     r3, #0
  58:   e1a00003        mov     r0, r3
  5c:   e8bd8800        pop     {fp, pc}
```

使用-O1优化编译，可获得指令mla。

```
pi@raspberrypi ~/arm $ gcc -c arm_mla.c -O1
pi@raspberrypi ~/arm $ objdump -d arm_mla.o

arm_mla.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <f>:
   0:   e0202091        mla     r0, r1, r0, r2
   4:   e12fff1e        bx      lr

00000008 <main>:
   8:   e3a00000        mov     r0, #0
   c:   e12fff1e        bx      lr
```

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语言命令：

```
#include <stdio.h>
int main(){
        int i;
        i=0x12345678;
        i=i&(~0x0f);
        return 0;
}
```

ARM汇编指令如下，可知，其实bic指令为操作数1的值与操作数2的值的反码按位与的操作。

```
pi@raspberrypi ~/arm $ vim arm_bic.c
pi@raspberrypi ~/arm $ gcc -c arm_bic.c
pi@raspberrypi ~/arm $ objdump -d arm_bic.o

arm_bic.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd00c        sub     sp, sp, #12
   c:   e59f3020        ldr     r3, [pc, #32]   ; 34 <main+0x34>
  10:   e50b3008        str     r3, [fp, #-8]
  14:   e51b3008        ldr     r3, [fp, #-8]
  18:   e3c3300f        bic     r3, r3, #15
  1c:   e50b3008        str     r3, [fp, #-8]
  20:   e3a03000        mov     r3, #0
  24:   e1a00003        mov     r0, r3
  28:   e28bd000        add     sp, fp, #0
  2c:   e8bd0800        ldmfd   sp!, {fp}
  30:   e12fff1e        bx      lr
  34:   12345678        .word   0x12345678
```