Lab 2：ARM 指令

编译与反汇编工具

使用 linux 交叉编译工具 arm-linux-gnueabi-gcc 将 C 语言源代码编译为 ARM 指令，利用 arm-linux-gnueabi-objdump 将目标文件反汇编为汇编代码。

ARM/Thumb

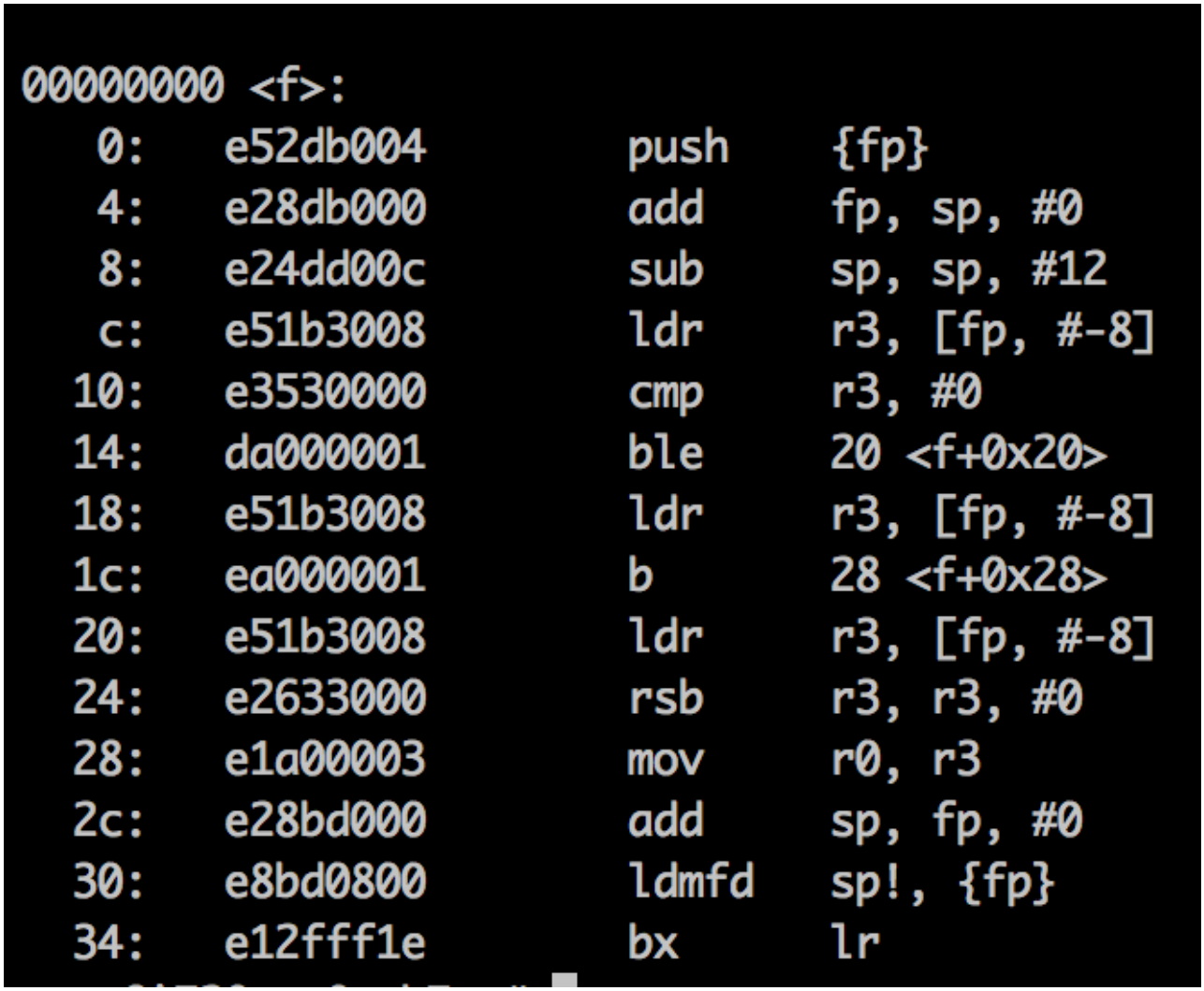
编写一个简单的 C 语言函数。

```
int f(){
        int x;
        if (x > 0)
                return x;
        else
                return -x;

}
```

不加编译参数，进行编译，再将 .o 文件反汇编。

可以看到每条指令是 32 位的，因为 ARM 指令是 32 位，Thumb 为 16 位，所以编译器默认生成的是 ARM 指令。



编译时加入 -mthumb 参数，将编译为 16 位 Thumb 指令。 可以看出 Thumb 指令所占用的空间更少。

```
00000000 <f>:
   0:   b580            push    {r7, lr}
   2:   b082            sub     sp, #8
   4:   af00            add     r7, sp, #0
   6:   687b            ldr     r3, [r7, #4]
   8:   2b00            cmp     r3, #0
   a:   dd01            ble.n   10 <f+0x10>
   c:   687b            ldr     r3, [r7, #4]
   e:   e001            b.n     14 <f+0x14>
  10:   687b            ldr     r3, [r7, #4]
  12:   425b            negs    r3, r3
  14:   1c18            adds    r0, r3, #0
  16:   46bd            mov     sp, r7
  18:   b002            add     sp, #8
  1a:   bd80            pop     {r7, pc}
```

ARM 条件执行语句

判断 a 与 b 的大小，进入不同分支。

```
int  f(int a, int b){
        int c = 1;
        if (a > b)
                c++;
        else
                c--;
        return c;
}
```

编译时加入 -O2 优化选项。可以到汇编代码中存在 ARM 的分支条转语句。

```
00000000 <f>:
   0:   e1500001        cmp     r0, r1
   4:   c3a00002        movgt   r0, #2
   8:   d3a00000        movle   r0, #0
   c:   e12fff1e        bx      lr
```

寄存器移位寻址

编写乘法操作函数，编译时加入 -O2 优化。

```
00000000 <ff>:
   0:   e0800100        add     r0, r0, r0, lsl #2
   4:   e12fff1e        bx      lr
```

得到的经过优化的汇编指令将乘法转换为加法，第二个操作数为寄存器移位寻址。

```
00000000 <ff>:
   0:   e0800100        add     r0, r0, r0, lsl #2
   4:   e12fff1e        bx      lr
```

装载 32 位立即数

```
int f(){
        return 0x80000000;
}
~
```

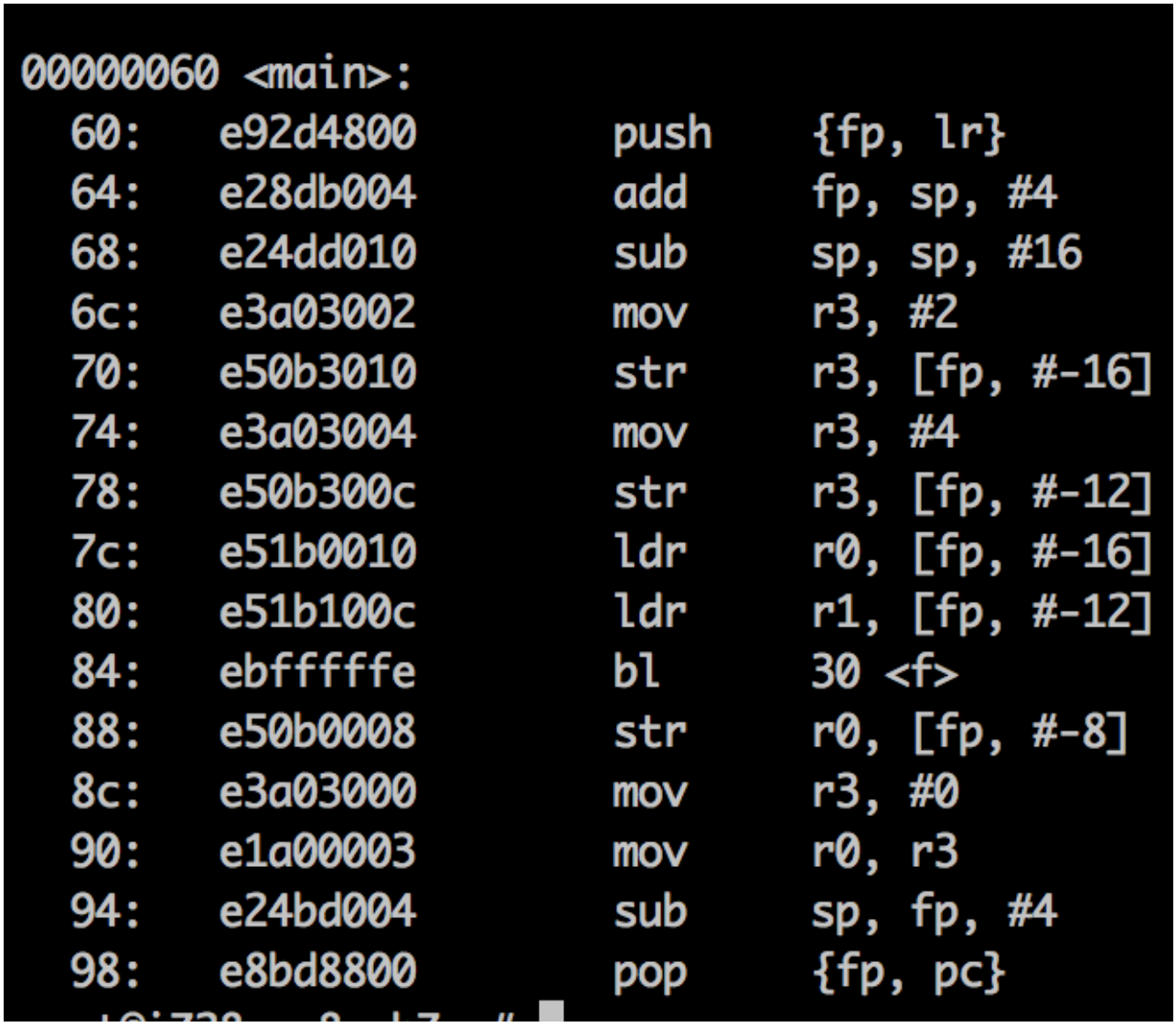
```
00000000 <f>:
   0:   e3a00102        mov     r0, #-2147483648        ; 0x80000000
   4:   e12fff1e        bx      lr
```

多重函数调用

```
int sub(int a, int b){
        return a - b;
}

int f(int a, int b){
        return sub(a, b);
}

int main(){
        int a = 2;
        int b = 4;
        int c = f(a, b);
        return 0;
}
```


00000060 <main>:
60: e92d4800 push {fp, lr}
64: e28db004 add fp, sp, #4
68: e24dd010 sub sp, sp, #16
6c: e3a03002 mov r3, #2
70: e50b3010 str r3, [fp, #-16]
74: e3a03004 mov r3, #4
78: e50b300c str r3, [fp, #-12]
7c: e51b0010 ldr r0, [fp, #-16]
80: e51b100c ldr r1, [fp, #-12]
84: ebfffffe bl 30 <f>
88: e50b0008 str r0, [fp, #-8]
8c: e3a03000 mov r3, #0
90: e1a00003 mov r0, r3
94: e24bd004 sub sp, fp, #4
98: e8bd8800 pop {fp, pc}

```
00000030 <f>:
  30:   e92d4800        push    {fp, lr}
  34:   e28db004        add     fp, sp, #4
  38:   e24dd008        sub     sp, sp, #8
  3c:   e50b0008        str     r0, [fp, #-8]
  40:   e50b100c        str     r1, [fp, #-12]
  44:   e51b0008        ldr     r0, [fp, #-8]
  48:   e51b100c        ldr     r1, [fp, #-12]
  4c:   ebfffffe        bl      0 <sub>
  50:   e1a03000        mov     r3, r0
  54:   e1a00003        mov     r0, r3
  58:   e24bd004        sub     sp, fp, #4
  5c:   e8bd8800        pop     {fp, pc}
```

```
00000000 <sub>:
   0:   e52db004        push    {fp}
   4:   e28db000        add     fp, sp, #0
   8:   e24dd00c        sub     sp, sp, #12
   c:   e50b0008        str     r0, [fp, #-8]
  10:   e50b100c        str     r1, [fp, #-12]
  14:   e51b2008        ldr     r2, [fp, #-8]
  18:   e51b300c        ldr     r3, [fp, #-12]
  1c:   e0633002        rsb     r3, r3, r2
  20:   e1a00003        mov     r0, r3
  24:   e28bd000        add     sp, fp, #0
  28:   e8bd0800        ldmfd   sp!, {fp}
  2c:   e12fff1e        bx      lr
```

- 函数调用时的返回地址在 lr 中。
- 传递的参数在 r0, r1, r2 ... 寄存器中。
- 先将参数按顺序压入堆栈高地址，再使用本地变量。
- 寄存器是 callee 保存，部分保存

MLA 累加的乘法

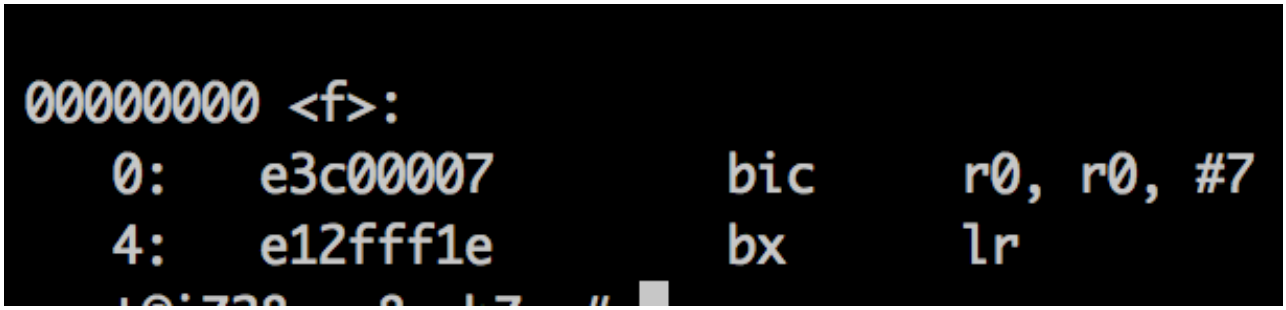
```
int f(int a, int b, int c) {
        return a * b + c;
}
```

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

BIC 指令

BIC 指令对 Rn 进行位清除，将 Rn 与 Operand2 的反码按位与。

```
int f(unsigned int a) {
        int b = a & 0xfffffff8;
        return b;
}
```



汇编函数

利用循环，每次输出一个字符，并判断是否超出规定范围。

在函数中要保存 lr 寄存器的值，以便返回 caller 继续执行。

```c
#include <stdio.h>
extern void * f(char* s, int n);
int main(){
        char* s = "Hello World";
        f(s, 5);
}


asm(
        "f: "
        "push    {r2, r3, fp, lr}\n"
        "mov     r2, r0\n"
        "LOOP:"
        "add     r3, r0, r1\n"
        "cmp     r2, r3\n"
        "beq     EXIT\n"
        "ldr     r0, [r2, #1]!\n"
        "bl      putchar\n"
        "b       LOOP\n"
        "EXIT:"
        "pop     {r2, r3, fp, pc}\n"
);
```