

# ARM 指令集编程调试实验教程

## 第二课

### 1 makefile 文件

在第一课中我们介绍大家使用 `arm-linux-gcc` 去手工编译 ARM 汇编程序，这个只实用于简单的单个程序文件编译。我们都知道有 `make` 这个命令可以进行一个工程文件的构造管理，`make` 命令可根据文件之间的依赖关系（在 `makefile` 文件中定义）来自动维护目标文件（与手工编译和链接相比，`make` 命令的优点在于它只更新修改过的文件）。

要用 `make` 命令来构造工程，我们必须掌握 `makefile` 文件的建立规则。

`makefile` 定义了一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作。

`makefile` 带来的好处就是“自动化编译”，一旦写好 `makefile`，只需要一个 `make` 命令，整个工程即可实现完全自动编译，从而极大地提高软件开发的效率。

`makefile` 文件如何编写请认真学习我们的课程讲义，这里不做过多描述。总之从现在开始我们每次都会写一个 `makefile` 文件来管理每个实验代码工程。

`Makefile` 我们后面编多文件、复杂项目时还会用的，用到哪我再讲到哪。

### 2 ddd 工具

第一次课大家使用的命令行式的 `gdb` 调试方式，可能大家期望能有个图形化的调试界面，就好像大家编程时候用的 IDE 一样。

本次实验课程我们介绍一个最简单的 `gdb` 调试图形 shell 工具 `ddd`，如果没有安装这个，可以很简单地使用命令 `sudo apt-get install ddd` 进行安装。

### 3 开始 ARM 指令编程调试

大家首先回忆一下 ARM 的编程模式，在教材、网课以及我的讲义里讲的，它有多少个寄存器？一般是这么说的：“ARM 内含 37 个寄存器，其中：31 个通用 32 位寄存器，6 个状态寄存器”。这个是我们采用 ARM7、ARM9 教学时讲的，但是前面我们决定更新实验箱的时候，本来我们打算讲 Cortex-A9，准备的教材也是 Cortex-A9 的，结果 Cortex-A9 的说明书上写的 40 个 32 位寄存器。看图 1，大家看看这个和目前这个网课讲的不同之处在哪里？

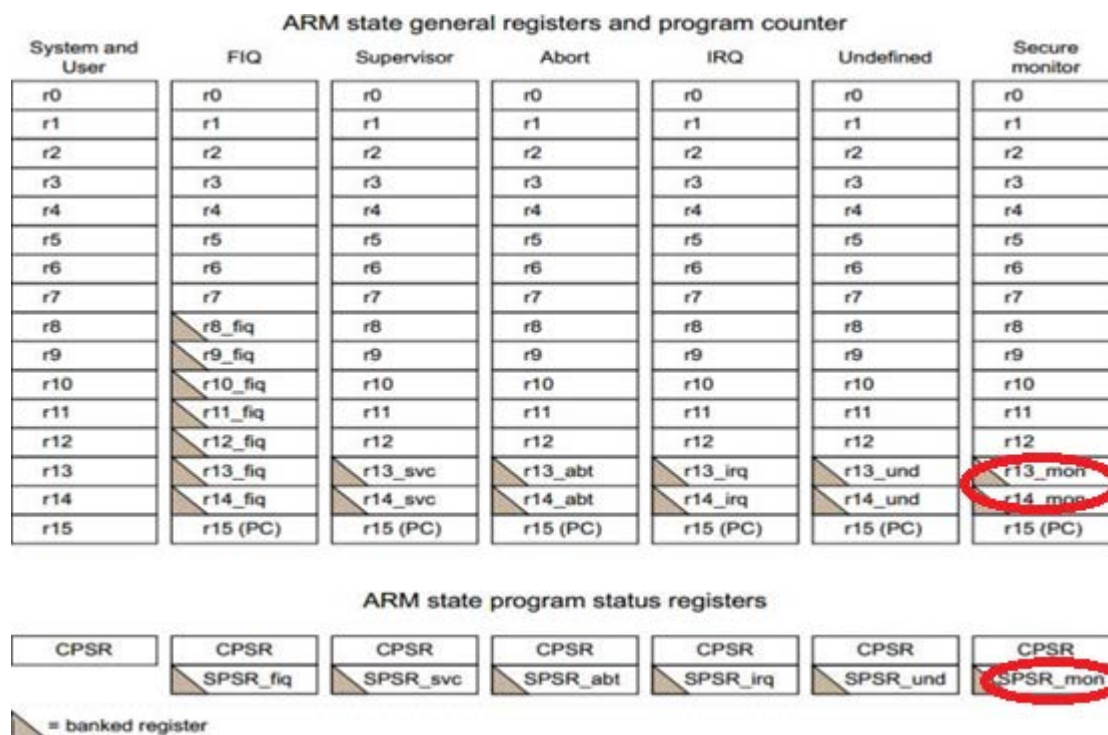


图 1 Cortex-A9 寄存器

Cortex-A9 多出来一个 Secure monitor 模式（不是以前的 7 个工作模式，是 8 个了），所以 R13、R14 以及 CPSR 各多了一套物理寄存器，37+3 就是 40 个了。

但不幸的是去年学院给大家最终配置的是超级先进的 Cortex-A53，教材没有，我也是临时去啃 A53 的 datasheet，发觉 A53 的寄存器更多，属于 ARMv8 指令集，支持 32/64 位操作，AArch 拥有 31 个通用寄存器，系统运行在 64 位状态下的时候名字叫 Xn，运行在 32 位的时候就叫 Wn，5 类特殊寄存器，64 位还有 NEON 和浮点寄存器，名字都不一样了，我认为实在不适合做教材给初学者讲。

所以你们还是以 37 个为准，37 个以外我们目前都还用不上。实在不行，就说根据 ARM 型号的不同，寄存器数量有所不同吧：):):)，包括工作模式也是以 7 个为准哈。

很多教材上讲编程模式的时候，可能很多同学就会问这是在讲硬件呀，怎么是编程模式呢，这是因为我们一般教材都是从软件编程的角度去理解硬件的原因。我们如果使用 ARM 汇编，就会直接是和寄存器打交道了。

今天这堂实验课，我们主要帮助大家理解 ARM 指令集的寻址方式，至于后面的指令类型，希望大家自行去调试理解、使用。

**寻址方式**是根据指令中给出的地址码字段来寻找真实操作数地址的方式。ARM

处理器支持的基本寻址方式有：

- 寄存器寻址
- 立即寻址
- 寄存器移位寻址
- 寄存器间接寻址
- 变址寻址
- 多寄存器寻址
- 堆栈寻址
- 块复制寻址
- 相对寻址

我们先来看第一个例子：

```
.globl _start
_start:
    MOV R0, #0x10
    MOV R2, #10
    MOV R1, R2
    MOV R0, R0
    SUB R0,R1,R2
    ADD R0, R1, R2
    ADD R3,R2,R1,LSL #2
    MOV R0, #0
    swi 0x00900001

.end
```

寄存器寻址、立即寻址指令我想大家不可能不会理解的，所以我们不会介绍，我们主要关注寄存器移位寻址：

**ADD R3, R2, R1, LSL #2 ;  $R3 \leftarrow R2 + 4 \times R1$**

这一条指令。

写好代码存盘为 ex1.s，编写一个自动化的 makefile 文件如下：

```
all:
    arm-linux-gcc ex1.s -o ex1 -nostdlib -g

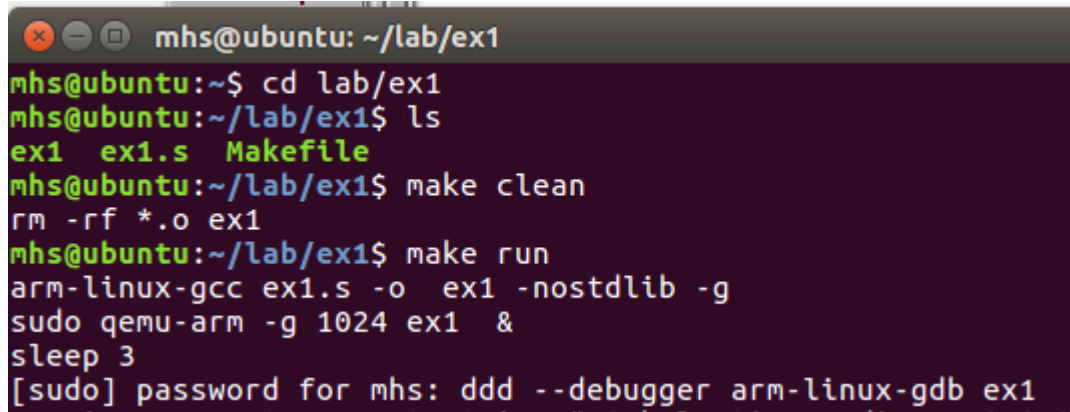
run: all
    sudo qemu-arm -g 1024 ex1 &
    sleep 3
    ddd --debugger arm-linux-gdb ex1
```

clean:

```
rm -rf *.o ex1
```

在这个 makefile 文件里我们有三个目标 all、run、clean。对 make 和 makefile 不熟悉的请参考我的讲义第 6 章 ppt 有详细讲解。注：目标 run 的第一句最后有个&,请大家注意这个 linux 命令后缀符是让命令在后台执行的意思，然后等待 3 微妙，再继续往下执行。

接下来我们进行图 2 的操作。



```
mhs@ubuntu: ~/lab/ex1
mhs@ubuntu:~$ cd lab/ex1
mhs@ubuntu:~/lab/ex1$ ls
ex1  ex1.s  Makefile
mhs@ubuntu:~/lab/ex1$ make clean
rm -rf *.o ex1
mhs@ubuntu:~/lab/ex1$ make run
arm-linux-gcc ex1.s -o ex1 -nostdlib -g
sudo qemu-arm -g 1024 ex1 &
sleep 3
[sudo] password for mhs: ddd --debugger arm-linux-gdb ex1
```

图 2 直接 make run 进行编译、运行、调试

命令执行后会另外出现 gdb 的图形调试界面，见图 3。

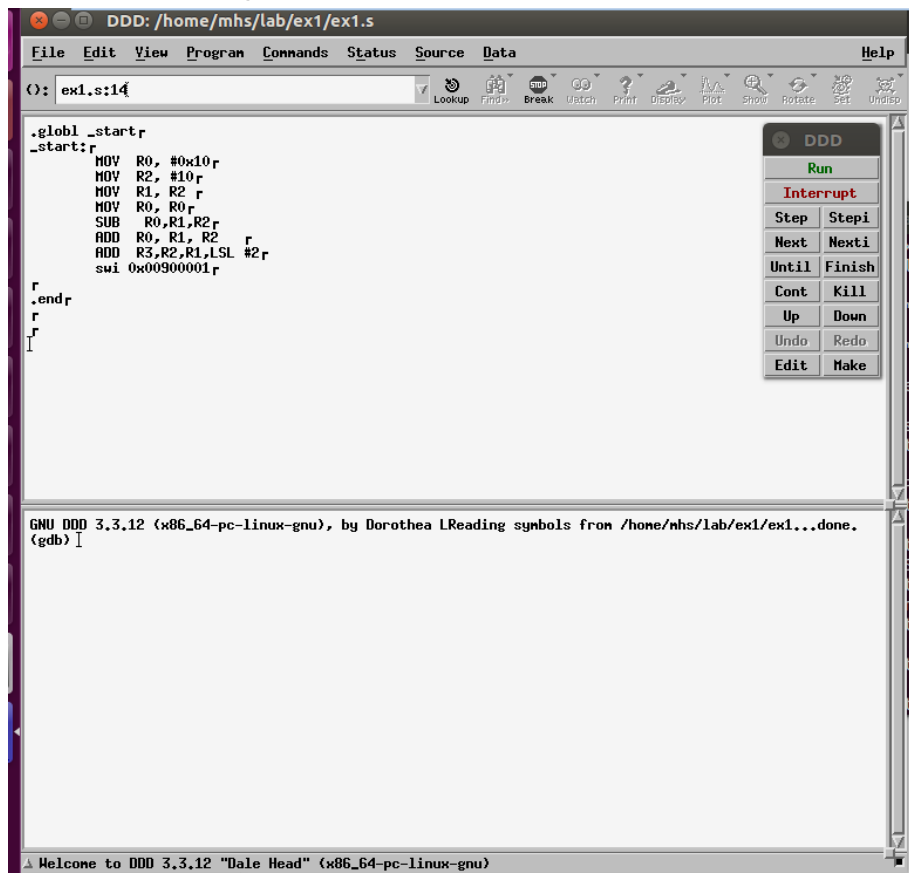


图 3 ddd 图形 shell gdb 调试

同样，我们在下方(gdb)后面键入：

```
target remote localhost:1024
```

回车后就会看到 gdb 连接成功，然后我们就可以用菜单点击命令进行调试了，图 4 我们设置一个断点，然后单步或者 run 调试，可以随时查看寄存器的内容随程序运行的改变。

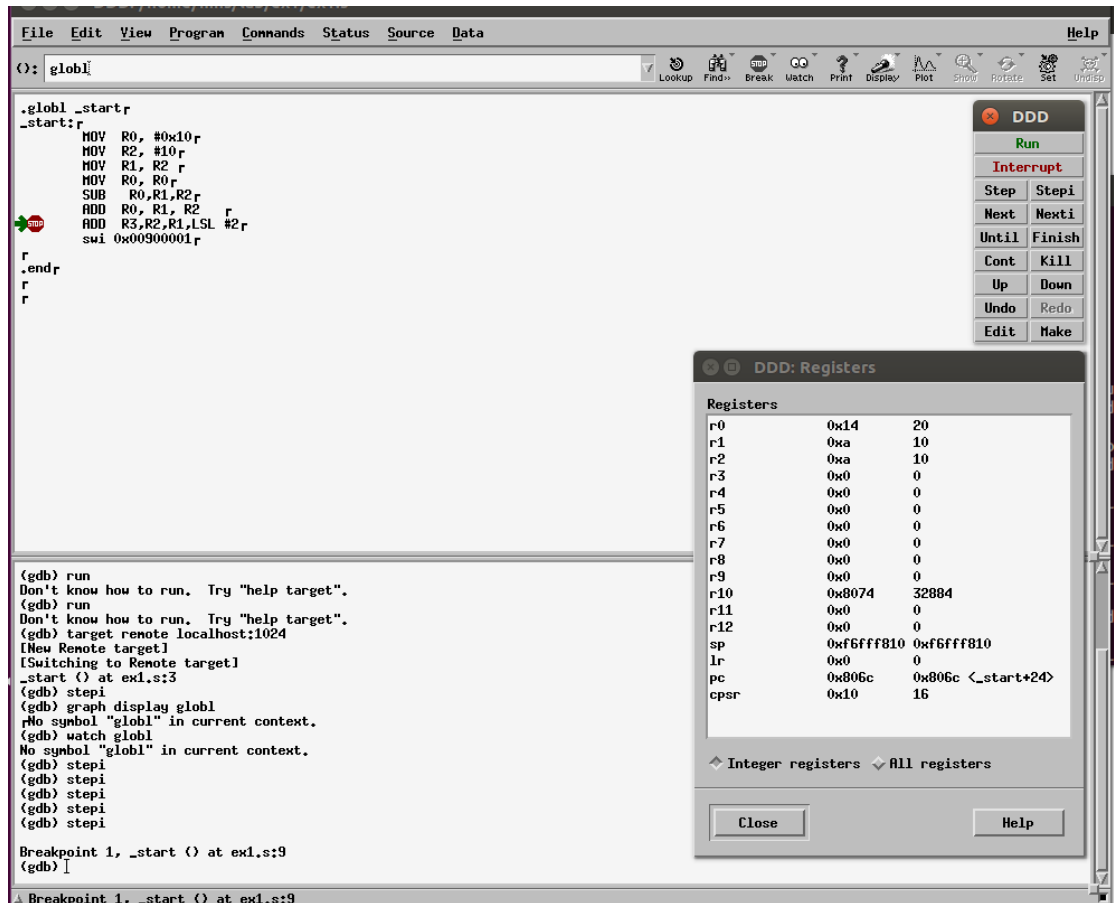


图 4 添加断点，查看寄存器值

如图 5 所示，我们可以看到运行 `ADD R3, R2, R1, LSL #2` 这一句后，R3 寄存器的值变成了  $50 = R2 + R1 * 2^2 = 10 + 10 * 4$ 。

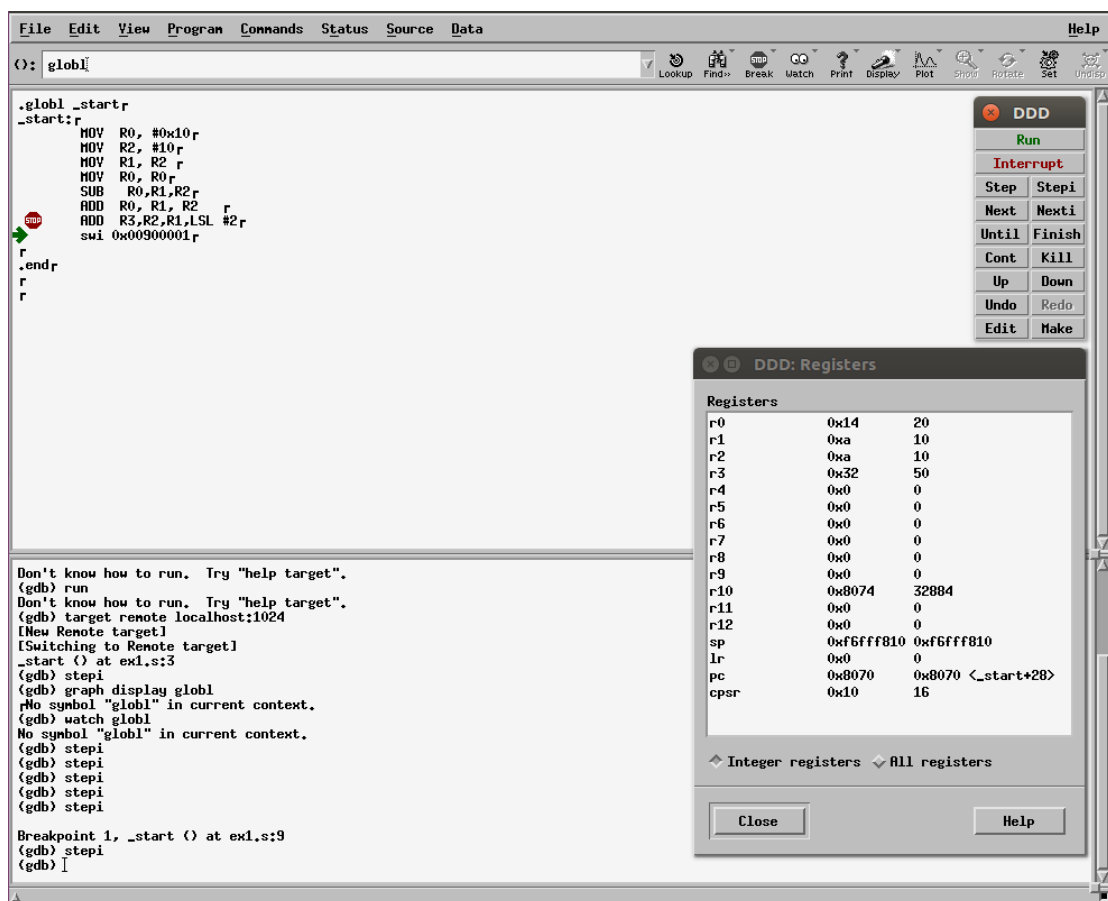


图 5 寄存器移位寻址调试结果

#### 4 作业布置：

请大家掌握后，对我 ppt 里面所讲所有示范 ARM 指令寻址方式程序示例逐一进行运行调试，加深理解。

重点理解语句：

LDR R3, [R0,#4] ! /\*带自动索引的前索引寻址：R3<-[R0+4];R0=R0+4 \*/

LDR R3, [R0],#4 /\*后索引寻址：R3<-[R0]; R0=R0+4 \*/

LDR R3, [R0,R4] /\*基址加索引寻址：R3<-[R3+R4] \*/

LDR R0, =src /\*R0 指向源数据区起始地址\*/

LDR R1, =dst /\*R1 指向目的数据区起始地址\*/

/\*MOV SP, #0x400 堆栈指针指向 0x400，堆栈增长模式由装载指令的类型域确定\*/

STMFD SP!, {R2-R11} /\*保存 R2-R11 的内容到堆栈，并更新栈指针，FD：满递减堆栈，由此可知堆栈长向\*/

LDMIA R0!, {R2-R11} /\*从 R0 所指的源数据区装载 10 个字数据到 R2-R11 中，每次装载 1 个字后 R0 中地址加 1，最后更新 R0 中地址\*/

STMIA R1!, {R2-R11} /\*将 R2-R11 的 10 个字数据存入 R1 所指的目的地数据区，每次装载 1 个字后 R1 中地址加 1，最后更新 R1 中地址\*/

LDMFD SP!, {R2-R11} /\*将堆栈内容恢复到 R2-R11 中，并更新堆栈指针，此时整 10 字单元数据已经复制完成，且出栈模式应和入栈模式一样\*/

我上传了讲义里面的 6 个实例程序，请大家逐一编写、调试、加深理解，有问题可在 qq 群给我留言讨论。

注：本实验教程为武汉科技大学机器人与智能系统研究院闵华松老师的网络课程教学文档，可以复制，不做商业用途。