ARM 指令集仿真实验第四课

1. For 循环实验补充

有同学提出老师能否讲一下 ARM 汇编的 for 循环语句，我们这门《嵌入式系统原理与应用》课程并没有太多汇编语言程序设计的内容，主要是让大家了解掌握一般嵌入式系统开发所涉及的一些基本软硬件系统原理，没有篇幅去专门讲程序设计，但大家都学过 C 语言程序设计了，所以嵌入式系统程序对大家来说就不是什么很困难的事了。

下面我对照 C 语言的程序设计方法，举几个例子，相信大家就会对 ARM 汇编的循环结构会掌握得比较快的。

第一个例子：

```
/*下面类似于
For(i=0;i<10;i++)
    Dest[i]=src[i];
*/
.data
src: .int 0,1,2,3,4,5,6,7,8,9
dst: .int 9,8,7,6,5,4,3,2,1,0
.text
.globl _start
_start:
    mov r2, #10
    ldr r0, =src
    ldr r1, =dst
    ;LDR R3, [R0]
    ;STR R3,[R1]
    mov R4, #0

loop:
    LDR R3, [R0,R4]  /*基址加索引寻址：R0<-[R3]; R0=R0+4 */
    STR R3, [R1,R4]
    ADDS R4,R4,#4
    SUBS r2,r2,#1
    BNE loop

    swi 0x00900001

.end
```

这个例子使用基址加索引寻址的方式实现了 10 次循环操作，大家仔细体会，自动化调试的 Makefile 文件编写如下：

```
all:
    arm-linux-gcc loop1.s -o  loop1 -nostdlib -g
```

```
run: all
    sudo qemu-arm -g 1024 loop1  &
    sleep 3
    ddd --debugger arm-linux-gdb loop1


clean:
    rm -rf *.o loop1
```

这个 Makefile 不用再解释了，后面我会专门讲稍微深入一点的 Makefile 文件如何写。

第二个例子：

```
/*下面类似于
For(i=0;i<10;i++)
    *(dest++)=*(src++);
*/
.data
src: .int 0,1,2,3,4,5,6,7,8,9
dst: .int 9,8,7,6,5,4,3,2,1,0
.text
.globl _start
_start:
    mov r2, #10
    ldr r0, =src
    ldr r1, =dst
    ;LDR R3, [R0]
    ;STR R3,[R1]

loop:
    LDR R3, [R0],#4 /*后索引寻址：R3<-[R0]; R0=R0+4 */
    STR R3, [R1],#4
    SUBS r2,r2,#1
    BNE loop


    swi 0x00900001

.end
```

这个例子采用后索引寻址方式实现 for 循环。

第三个例子：

```
/*下面类似于
*dest= *src;
For(i=0;i<10;i++)
```

```
    *(++dest)=*(++src);
*/

.data
src: .int 0,1,2,3,4,5,6,7,8,9
dst: .int 9,8,7,6,5,4,3,2,1,0
.text
.globl _start
_start:
    mov r2, #10
    ldr r0, =src
    ldr r1, =dst
    LDR R3, [R0]
    STR R3,[R1]

loop:
    LDR R3, [R0,#4] ! /*带自动索引的前索引寻址: R3<-[R0+4];R0=R0+4 */
    STR R3, [R1,#4] !
    SUBS r2,r2,#1
    BNE loop

    swi 0x00900001

.end
```

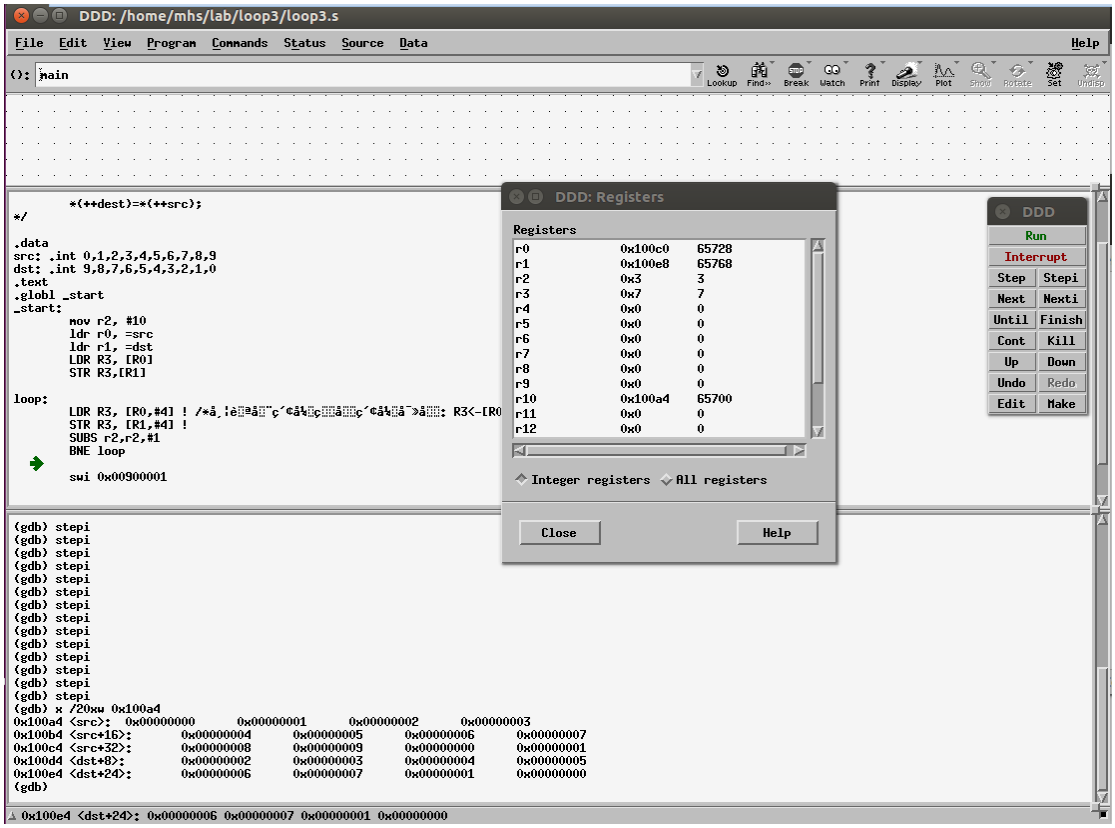

图 1 单步运行调试界面

这个例子采用了带自动索引的前索引寻址实现 for 循环。当然大家也可以写出更多的例子出来，一个嵌入式软件工程师的目标是健壮高效简洁，你们可以发挥自己的风格和想象去创造更好的代码出来。☺

至于用堆栈指针去进行块复制的程序上次补充实验已经讲过了，另外还有方法就是使用 ARM 的 load/store 机制（大家记住 load/store 机制是 ARM 体系结构里面突出的一个优点，这个在讲义里面有讲），对于 LDM 和 STM 执行块复制的程序留给大家自己去编写掌握。

2. 多个源文件需要编译成一个执行文件的 Makefile 文件编写

有时候我们编写程序，会写一些类库便于复用，或者将不同的操作进行分类，这样的项目，通常会有很多源文件，比如按功能、类型、模块把源代码写在不同的目录或者多个程序中。

有关 Makefile 的**目标、依赖、规则**在讲义里有较为详细的讲解，如果没有掌握的同学，也可以找个最简易的入门文档去看，比如知乎的入门文章：一文入门 Makefile。

例如，我们的项目代码有 main.c，mytool1.c，mytool2.c，helloworld.c，mytool1.h，mytool2.h 等源文件，下面是示范的简略代码。

main.c:

```
#include<stdio.h>
#include"mytool1.h"
#include"mytool2.h"

int main(int argc, char **argv)
{
    printf("hello world!\n");
    mytool1();
    mytool2();

    return 0;
}
```

mytool1.c:

```
#include<stdio.h>
#include"mytool1.h"

void mytool1(){
    printf("mytool1!\n");
}
```

mytool2.c:

```
#include<stdio.h>
#include"mytool2.h"
```

```c
void mytool2(){
    printf("mytool2!\n");
}
```

helloworld.c:

```c
#include<stdio.h>

int main(int argc, char **argv)
{
    printf("hello!\n");

    return 0;
}
```

mytool1.h:

```c
#ifndef MYTOOL1
#define MYTOOL1
void mytool1();
#endif
```

mytool2.h:

```c
#ifndef MYTOOL2
#define MYTOOL2
void mytool2();
#endif
```

从这个项目的源代码编写中，我们可以得出图 2 的依赖关系。

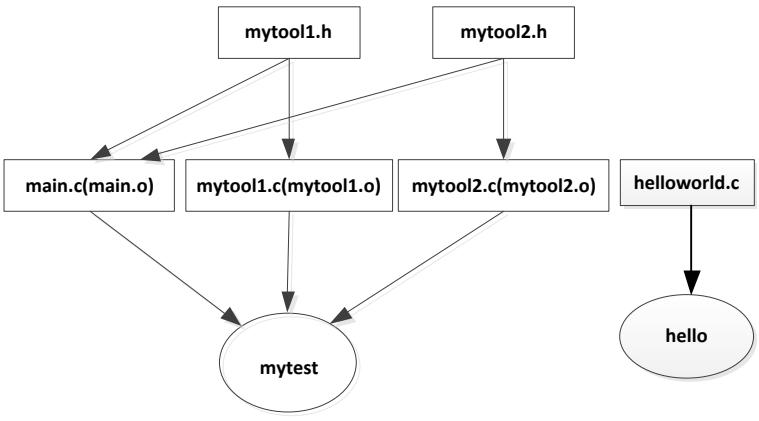

图 2 项目源文件依赖关系图

依据这个依赖关系我们写出的 Makefile 文件如下：

```
all:mytest hello
CC = gcc
INSTDIR = $(HOME)/lab/install
mytest:main.o mytool1.o mytool2.o
    $(CC) -o mytest main.o mytool1.o mytool2.o
main.o:main.c mytool1.h mytool2.h
    $(CC) -c main.c
mytool1.o:mytool1.c mytool1.h
    $(CC) -c mytool1.c
mytool2.o:mytool2.c mytool2.h
    $(CC) -c mytool2.c
hello:helloworld.o
    $(CC) -o hello helloworld.c
clean:
    rm -rf *.o mytest hello
install: mytest
    @if [ -d $(INSTDIR) ]; \
    then \
    cp mytest $(INSTDIR) &&\
    chmod a+x $(INSTDIR)/mytest &&\
    chmod og-w $(INSTDIR)/mytest &&\
    echo "Installed in $(INSTDIR) success!";\
    else \
    echo "Sorry, $(INSTDIR) does not exist";false;\
    fi
```

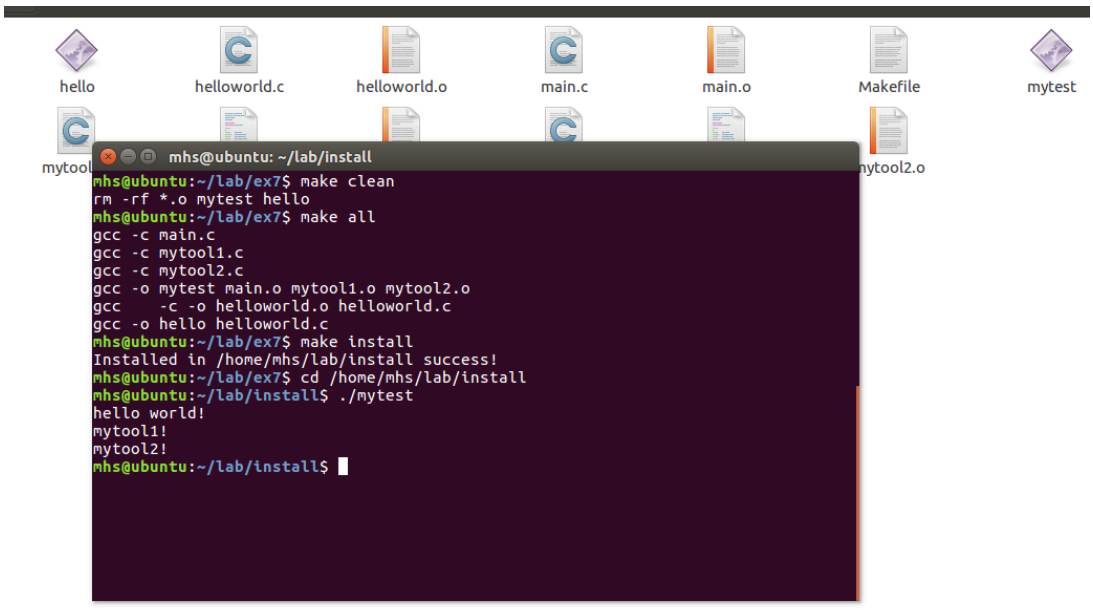

图 3 编译、安装、运行示范

大家可以看到我们写了一个 all 目标，它包含 mytest 和 hello 两个文件目标；

第二行和第三行是两个宏，一个是编译器指定宏，一个是用于安装软件的路径宏。后面描述的是 mytest 和 hello 这两个目标进一步的依赖关系以及生成规则。需要一再强调的是每条规则前面不是空格，千万不要用空格代替，那是一个 tab 键。Install 命令写了一段脚本，将生成的执行文件拷贝到安装目录，并赋予可执行权限。

示范编译与安装、运行结果如图 3 所示，请大家自行编写项目深入体会。

3 汇编与 C 混合编程

大家可以看到我在课程中并不提倡大家用太多的汇编去编程，甚至是一会儿汇编，一会儿又用 C 语言。实际上在嵌入式系统领域，它体现出来的产品高附加值，除了目前强大的硬件 IP 外，更体现在软件附加值上，嵌入式系统计算机和 PC 机在计算的强大性方面越来越没有了任何界限。

所以我们讲嵌入式系统，讲完指令集和初始化代码（bootloader 的第一阶段和第二阶段部分）是用汇编，后面讲操作系统、驱动代码、应用程序设计基本都是高级语言了（C/C++、java 等）。

在讲启动代码之前，我们先给大家讲一讲汇编和 C 语言的混合编程实验。

首先是 C 语言中如何使用汇编语言，在 C 语言中使用汇编语言，一般的考虑是提高程序的效率和直接使用硬件资源才会考虑。

比如上一次的实验中，我们测试过系统的大小端问题，qemu 目前仿真 ARM 好像默认是小端，无法用软件模拟大端模式，官方网站说有支持，但我一直没看到纯软件模拟的支持是在哪里，如果你们有人发现了解决方法，请告诉我。

上次我们说 ARM7/ARM9 可以使用 CP15 协处理器的 C1 寄存器设置大小端模式，这里我们把原来那个 C 语言程序拿过来测试一下。

```
#include <stdio.h>

//return 1 : little-endian
//      0 : big-endian
int checkCPUendian()
{
    union {
        unsigned int a;
        unsigned char b;
    } c;

    c.a = 1;
    return (c.b == 1);
}
void init()
{
    asm(
    /*
     * set the cpu to SVC32 mode
     */
    "mrs    r0,cpsr\n"
```

```
    "bic    r0,r0,#0x1f\n"
    "orr    r0,r0,#0xd3\n"
    "msr    cpsr,r0\n"
    );
    asm volatile(
    "mov r1, #0x80\n"
    "mcr p15, 0, r1, c1, c0, 0\n"
    :::"r1"  //:::"r1" 向 GCC 声明：我对 r1 作了改动

    );
}

int main(int argc, char **argv)
{
    init();
    if(checkCPUendian()==1)
        printf("little-endian\n");
    else
        printf("big-endian\n");

    return 0;
}
```

在 init()函数里面我们写了两段 ARM 汇编，第一段直接操作寄存器，第二段我们想操作 CP15 协处理器，修改协处理器 CP15 的 C1 寄存器里面第八位相关大端/小端的设置位。

注意这里由于我们用的 qemu-arm，它是不能仿真系统的，只能模拟程序软件，如果要模拟硬件，必须使用 qemu-system-arm。qemu-arm 和 qemu-system-arm 的区别在于：qemu-arm 是用户模式的模拟器(更精确的表述应该是系统调用模拟器)，而 qemu-system-arm 则是系统模拟器，它可以模拟出整个机器并运行操作系统。所以在 qemu-arm 仿真环境下，我们修改 CPSR 实际上它没起到任何作用，同样第二段的 mcr 指令也没起作用，内存模拟还是小端模式，当然这些寄存器里面的值确实是被修改成功了。注意 qemu-arm 没法用 mrc 指令，因为实际上根本就没有协处理器 CP15 存在，读就是非法的。后面我们讲汇编混合跳转到 C 语言程序以及 bootloader 仿真实验就会开始用到 qemu-system-arm 去仿真。

Makefile 文件的编写和前面没有区别，当然编译参数是选择动态链接库还是静态链接，都是根据项目选择，qemu-am 模拟一般没法用库文件，所以一般选择静态链接参数，而后面讲 bootloader 的时候，我们的系统会用到库，会涉及是静态还是动态链接。

再来看如何在汇编程序里如何用 C 语言混合编程。

先看汇编代码 startup.s：

```
.global _Start
.global cmsg
.global clen
```

```
.data
msg:
       .ascii  "Hello, ARM!\n"
len = . - msg

cmsg:
       .ascii  "Test jump into c entry!\r\n"
clen = . - cmsg


.text

_Start:
       /*Syscall write(int fd, const void *buf, size_t count)*/
       mov %r0, $1     /*输出到 stdout，stdout 的设备描述符为 1*/
       ldr %r1, =msg   /*buf=msg*/
       ldr %r2, =len   /*count=len*/
       mov %r7, $4     /*syscall 4*/
       swi $0         /*invoke syscall*/

   BL c_entry

       /*Syscall exit*/
       mov %r0, $0
       mov %r7, $1
       swi $0
.end
```

在这里 startup.s 文件里，我定义了三个全局入口，_Start, cmsg, clen,其中 _Start 代表的是全局的程序入口地址，cmsg 和 clen 是定义的一个字符串，准备拿到 C 语言里面去用的。

_Start 开始我们用系统软中断调用将字符串打印输出到 stdout 设备。接下来大家看到了一个跳转指令：

BL c_entry

这条指令表示无条件转到 c_entry，一般来说这个 c_entry 应该全局，以便链接器能找到它。但 arm-linux-ld 链接器好像可以找到，所以我没有申明成 global。

c_entry 在哪呢，在 init.c 文件里面：

```
//#include <stdio.h>

extern unsigned char* cmsg;
extern unsigned int clen;

extern int sum(int a,int b);

void display()
```

```
{
        asm(
        "mov %r0, $1\n"     /*输出到 stdout，stdout 的设备描述符为 1*/
        "ldr %r1, =cmsg\n"   /*buf=msg*/
        "ldr %r2, =clen\n"   /*count=len*/
        "mov %r7, $4\n"     /*syscall 4*/
        "swi $0\n"          /*invoke syscall*/
    );

}


int c_entry()
{

    if(3==sum(1,2))
        display();

    return 0;
}
```

在这个 C 语言程序里面我故意又引用了一个 sum 函数，这个函数是 extern 外部的，在哪呢，在 sum.s 汇编文件里面，定义成一个 global 标签了。它是一个函数，使用 r0 和 r1 传递两个参数，然后将和返回到 r0 作为函数返回值。

```
/*sum.s
*/

.text

.global sum

sum:

mov r3,r0
add r3,r1
mov r0,r3
mov pc,lr   ;#  lr 即 r14

.end
```

sum 函数在最后一句是：mov pc,lr，将链接寄存器直接赋值给 PC，表示程序到原来调用函数的断点处返回，指向了下一条代码。

这个示范工程我设计的故意有点绕，三个源程序文件：startup.s,init.c,sum.s，从汇编跳到 C 语言，又从 C 跳到汇编程序，返回又在 C 语言里引用了嵌入汇编，大家可以认真体会一下。

好了，这样的一个项目我们如何编写自动化编译测试 Makefile 文件呢？

代码如下，我简单地进行了 C 语言、汇编语言的编译，生成带调试符号的目标文件，最后用 arm-linux-ld 链接器将三个 o 文件链接成一个 elf 文件（Elf 文件、BIN 文件是什么，后面我们再解释，这里先不讲这个，大家只需要记住现在我们将三个目标文件链接在一起了生成了 test.elf 文件）。

```
all:
    arm-linux-gcc -c  init.c -o init.o  -g -nostdlib
    arm-linux-gcc -c  startup.s -o startup.o -g
    arm-linux-gcc -c  sum.s -o sum.o -g
    #arm-linux-gcc -o  ctest init.o startup.o sum.o  -g
    arm-linux-ld -T map.lds init.o startup.o sum.o -o test.elf

run: all
    sudo qemu-arm  -g 1234 test.elf   &
    sleep 3
    ddd --debugger arm-linux-gdb test.elf

clean:
    rm -rf *.o test.elf
```

我们还是用 DDD 去调试，直接 make run 就会出来图 4 的画面，键入 target remote localhost:1234 进行连接 gdb 后，我们可以一路点击 stepi 单步调试下去。

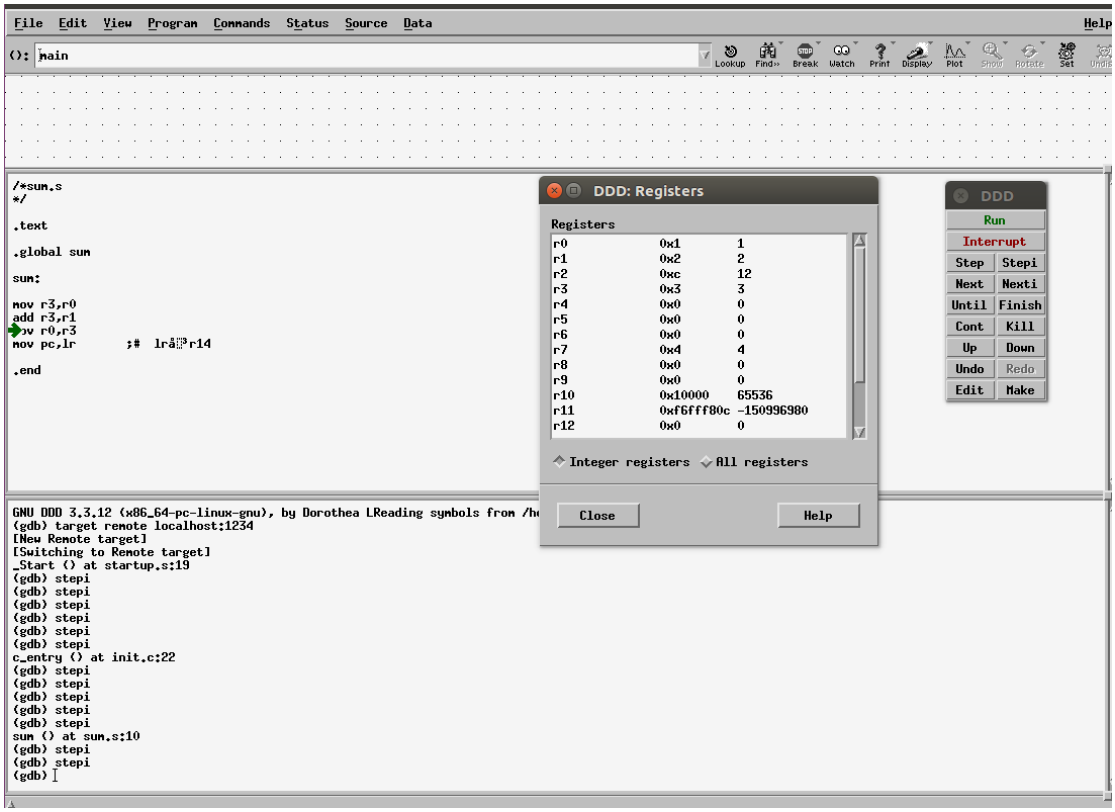

图 4 单步调试界面

我们可以看到调试界面会在汇编文件和 C 文件之间穿梭，大家可以认真体会，这里面的寄存器操作以及内存的变化。

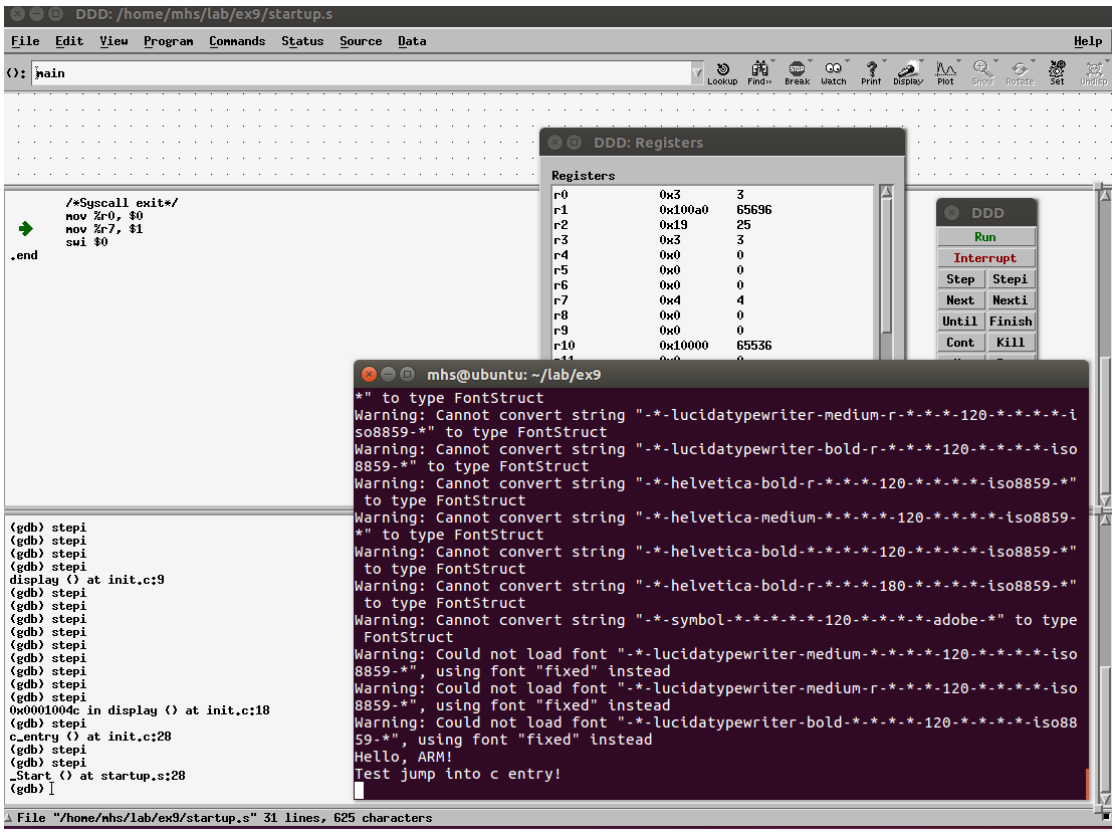

图 5 终端窗口成功打印出两行字符串

最后大家可以看到在终端窗口，程序运行退出之前，成功打印出两行字符串。至此，我们对第三章 ARM 指令集的程序仿真调试实验讲解完毕，请大家前后对照讲义，进行深入学习，融会贯通。