ARM 体系结构数据类型及堆栈操作实验

在第 2 讲的时候，我假设同学们是已经把指令集学完了的，但在 mooc 的课程学习中，我用慕课堂出的讨论题，从很多复制粘贴，基本缺乏自己理解的回答中，我发现很多同学其实对 ARM 体系结构特别是编程模型并没有留下什么映像。所以这次的实验教程文档我用模拟实验来补充讲讲数据在内存中的存放以及堆栈如何使用的基本知识。

1 数据类型的理解

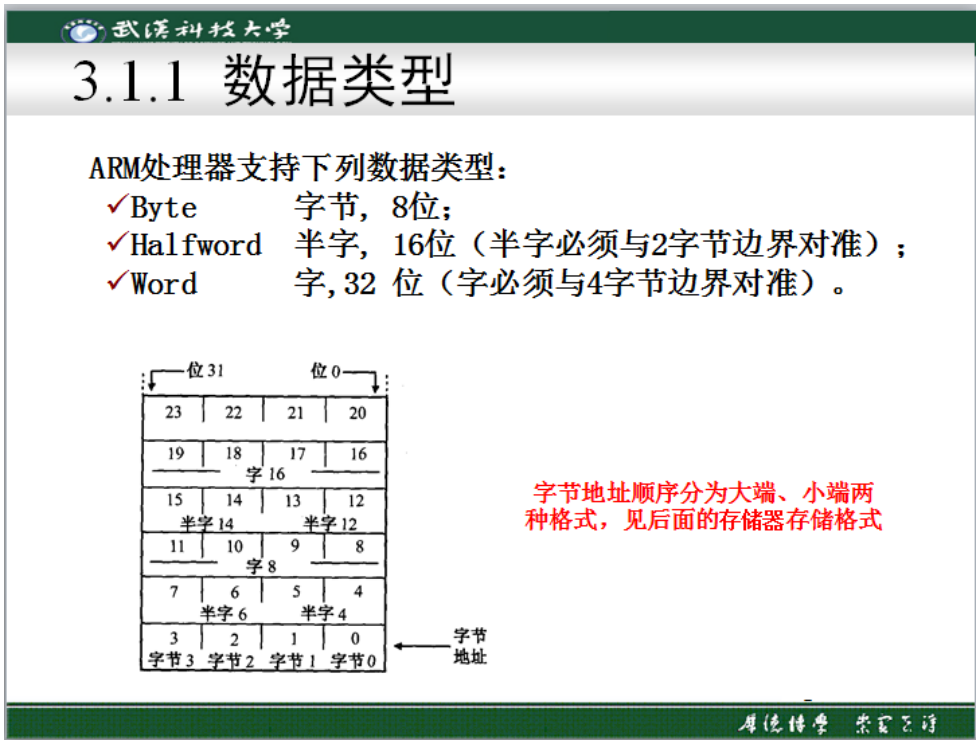

图 1 ARM 支持的数据类型

图 1 是我的讲义的一张 ppt，我们首先看看 ARM 处理器架构的计算机中，支持的数据类型。与高级语言类似，ARM 支持对不同数据类型的操作。我们可以加载（或存储）的数据类型可以是有符号和无符号的字、半字或字节。

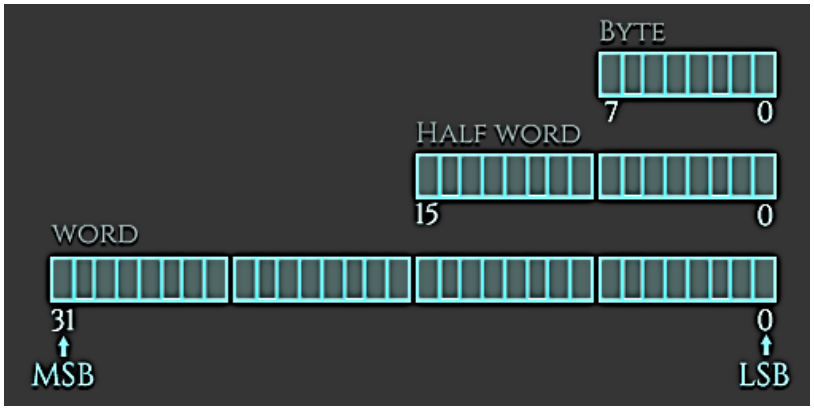

图 2 三种数据类型

见图 2，这里有两个词 LSB 和 MSB, LSB（Least Significant Bit），意思为最低有效位；MSB（Most Significant Bit），即最高有效位，这里提出一个问题，如果这个数据是有符号数据类型，请问 MSB=1 表示的是正数还是负数？

除了数据类型外，大家可以看到我的 ppt 里在这个地方还给大家介绍了字节地址的存放顺序，分为两种：大端格式(big endian)和小端格式(little endian)，见图 3。

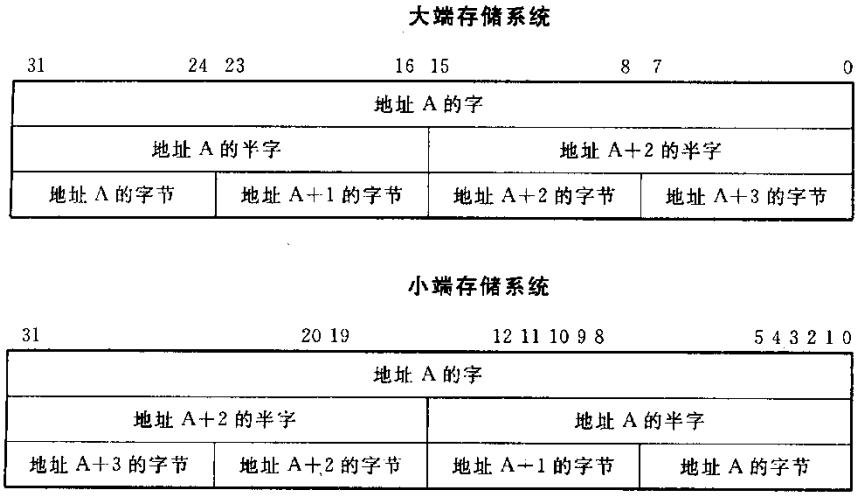

图 3 存储系统字节顺序

从这个图里，大家可以看到 32 位 ARM 存储系统，一个存储单元是 32 位，那么就会涉及到如果这个数据类型少于 32 位的时候，这个数据的高低字节的存放顺序问题。两种模式存放的字节地址顺序是相反的。

说到大端小端的名词由来，有一个有趣的故事，来自于 Jonathan Swift 的《格利佛游记》。Lilliput 和 Blefuscu 这两个强国在过去的 36 个月中一直在苦

战，战争的原因：大家都知道，吃鸡蛋的时候，原始的方法是打破鸡蛋较大的一端，可是那时皇帝的祖父由于小时侯吃鸡蛋，按这种方法把手指弄破了，因此他的父亲就下令，命令所有的子民吃鸡蛋的时候，必须先打破鸡蛋较小的一端，违令者重罚。然后老百姓对此法令极为反感，期间发生了多次叛乱，其中一个皇帝因此送命，另一个丢了王位，产生叛乱的原因就是另一个国家 Blefuscu 的国王大臣煽动起来的，叛乱平息后，就逃到这个帝国避难。据估计，先后几次有 11000 余人情愿死也不肯去打破鸡蛋较小的端去吃鸡蛋。这个其实是乔纳森·斯威夫特讽刺当时英国和法国之间持续的冲突。

有一位网络协议的开创者 Danny Cohen，第一次使用这两个术语指代字节顺序，后来就被大家广泛接受，在计算机工业中指数据储存顺序的模式。

讲了这么多，大家看完我的讲义和 mooc 视频后，还是自己动手做做实验，加深理解吧。

今天开始我们不用汇编，我们直接用 C 语言来讲吧，因为大家 C 语言程序设计都是学过了的，所以我用 C 语言来做这个实验，大家理解可能更快一些。翠花上酸菜，哈哈！

```c
#include <stdio.h>

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

    return 0;
}
```

多么熟悉的程序啊，嗯，不对啊，老师你那个 checkCPUendian()函数是什么鬼？

哈哈，这里我们来看看这个 checkCPUendian()函数：

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

    c.a = 1;
    return (c.b == 1);
}
```

这里使用了一个技巧，用了一个 union 数据结构。在 C 语言中，有一种和结构体非常类似的语法，叫做联合体（Union），union 和 struct 的区别在于：struct 的各个成员会占用不同的内存，互相之间没有影响；而 union 的所有成员占用同一段内存，修改一个成员会影响其余所有成员。联合体在分配内存时，总是以成员中占内存最大的类型分配，这个函数里面定义的 union 联合体，占据内存是以 `unsigned int a` 来分配空间的，无符号整形占据 32 位也就是四个字节的内存单元，我们用 `c.a = 1` 这一句，将 C 这个联合体赋值为 1，那么它如果去读取 c.b 的值的时候，这个内存里面的数据单元已经被修改为无符号整形数值 1 了，在小端模式下，其中四字节的内存数据，低字节存放 1，高字节存放 0，因为是无符号数，高位 MSB 为零。取 c.b 的值，因为联合体总是从最低字节地址开始取数据，所以 c.b 就等于 1。反之如果内存结构是大端模式，c.a=1 的时候修改的是高字节地址里面的数据为 1，低字节地址里面数据为 0，所以取 c.b 的值的时候，c.b=0。所以这个函数返回比较语句`(c.b == 1)`为真时值为 1，表明机器是小端模式，返回值为零就是大端模式了。

一般来说常用的 Intel X86 结构是小端模式，而 Keil C51 则为大端模式。很多的 ARM、DSP 都为小端模式。ARM 架构在 ARMv3 之前是小端模式，在那之后，

ARM 处理器可以通过硬件配置在大小端之间切换。以 ARMv6 为例，指令是固定的以小端模式存储的，而内存数据的读取方式可以通过控制程序状态寄存器 CPSR 的第 9 位实现在大端和小端之间切换。我们在讲 ARM7/ARM9 处理器的时候，讲过是可以操作 CP15 寄存器，来切换大小端，MMU，缓冲等等的，而大家电赛常用的 cortex-m 核的 stm32 是固定小端模式的。

下面我们写个复杂点的 C 语言结构体程序，看看数据在内存中是如何存放的。

上一个完整的 C 语言代码：

```
#include <stdio.h>

struct T1{char c;short s; int x }t1={'a',0x1111,0x10101010};

typedef struct EXAMPLE{
       char c2[2];
       struct T1  s1[2];
}example;

void pausehere()
{
    printf("pause here\n");
}

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

    c.a = 1;
    return (c.b == 1);
}

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

```
    struct T1 t2;
    //t1.c = 'a';
    //t1.s = 0x1111;
    //t1.x = 0x10101010;
    printf("t1 address: 0x%x\n",&t1);



    t2.c = 'b';
    t2.s = 0x2222;
    t2.x = 0x20202020;
    printf("t2 address: 0x%x\n",&t2);

    example ex;
    ex.c2[0]='c';
    ex.c2[1]='d';
    ex.s1[0] = t1;
    ex.s1[1] = t2;
    printf("ex address: 0x%x\n",&ex);

    pausehere();

//printf("EXAMPLE: %c %c %c %d %d %c %d %d\n",ex.c2[0],ex.c2[1],ex.s1
[0].c,ex.s1[0].s,ex.s1[0].x,ex.s1[1].c,ex.s1[1].s,ex.s1[1].x);

    return 0;
}
```

在这个程序中，我们定义了两个结构体类型 T1 以及 example，申明了一个 global 全局的数据结构 t1，和在 main 函数里定义的 t2，以及 example 类型的 ex。

程序写好了，我们写个自动化的 makefile 文件吧，如何写上节课我们已经讲过了，这次直接上代码：（不懂的可以在 qq 群问）

```
all:
    arm-linux-gcc -o demo datademo.c -g -static

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


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

存盘后我们直接 make run 就可以自动编译、调试了，调试界面出来了，见图 4。

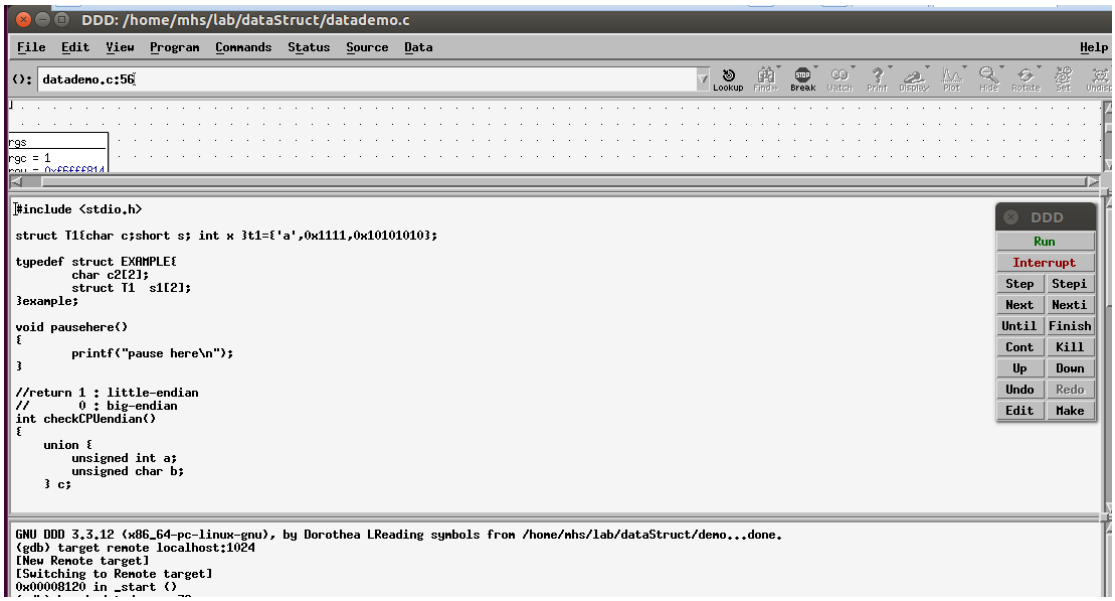

图 4 调试界面

大家可以看到我键入 target remote localhost:1024 后，DDD 连上了 gdb。

接下来我们添加一个 breakpoint 断点，放在主程序的“pausehere();”这一句。

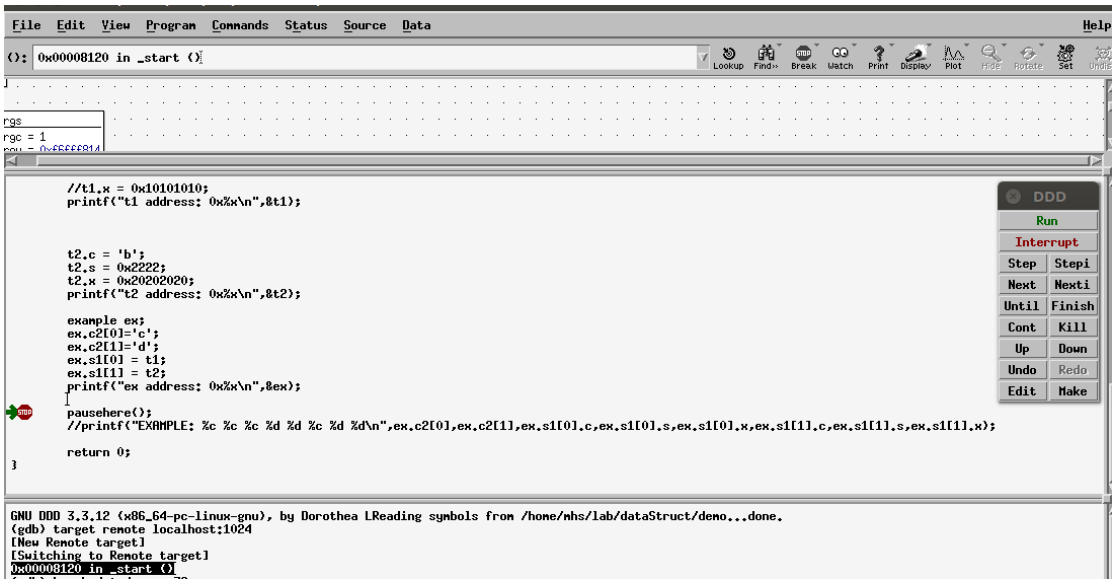

图 5 添加一个断点

然后我们一路点击 step 单步运行，一直到运行指示箭头停留在断点这一句，返回到运行命令的终端窗口去看看，见图 6。

```
Warning: Could not load font "-*-lucidatypewriter-medium-r-*-*-*-120-*-*-*-*-iso
8859-*", using font "fixed" instead
Warning: Could not load font "-*-lucidatypewriter-bold-*-*-*-*-120-*-*-*-*-iso88
59-*", using font "fixed" instead
little-endian
t1 address: 0x8eec8
t2 address: 0xf6fff6b8
ex address: 0xf6fff6a4
Warning: Could not load font "-*-helvetica-medium-r-*-*-*-80-*-*-*-*-iso8859-*",
 using font "fixed" instead
```

图 6 单步运行到断点处出来的结果

大家可以看到在程序中系统默认是小端格式，我们用取址符&获取了 t1、t2 以及 ex 这三个结构体变量的内存地址，使用 printf 语句打印出来了。从图 5 我们看到程序仿真运行的时候入口地址是 0x00008120(系统默认的_start()入口)，由于 t1 是在外面定义的结构体，所以它初始化在全局的数据区里面，地址是 0x8eec8，我们用 gdb 调试命令（x /8xw 0x0008eec8 ）查看以下内存。

```
(gdb) x /8xw 0x0008eec8
0x8eec8 <t1>:   0x11110061      0x10101010      0x00000830      0x0008fc30
```

图 7 内存查看 t1 数据存放图

t1={'a',0x1111,0x10101010};看看这是否符合小端存放规律？

再看一下 t2 和 ex 在内存中的存放吧，见图 8。

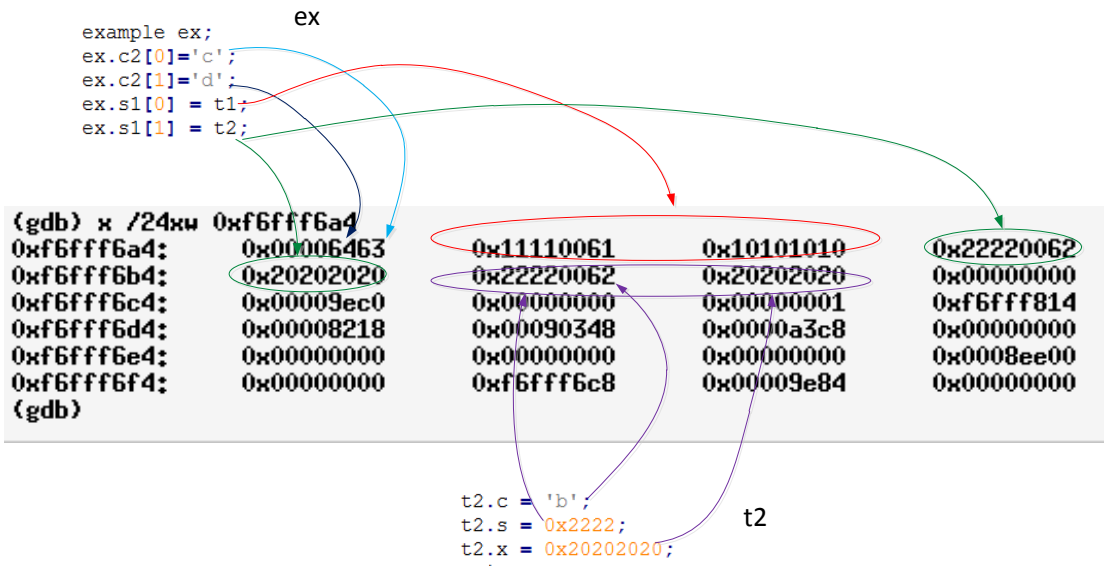

图 8 变量 ex、t2 在内存中的存放图

2 ARM 堆栈操作

本来不想再讲 ARM 寄存器与堆栈操作的，但有同学报告 mooc 课程视频讲义里老师把链接寄存器讲成了 R13,这是个错误，可能这个 mooc 网的课程上线不太久，他们还没发现吧。

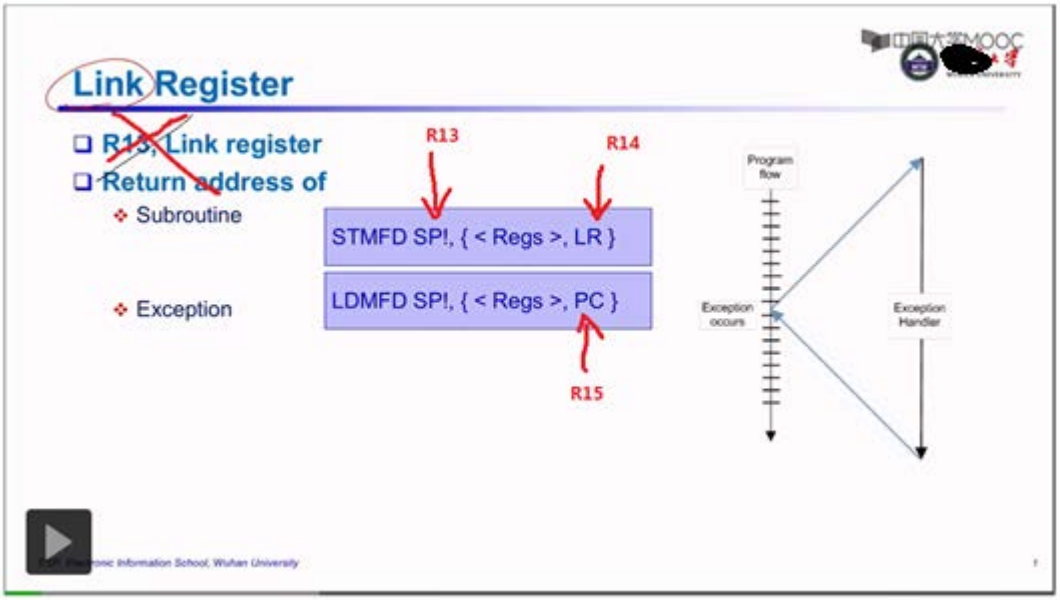

图 9 mooc 课程错误

这里 R13 是堆栈寄存器，R14 才是链接寄存器，R15 是 PC 指针。同学反应这块比较难，所以我还是建议大家和我一起来做实验加深理解。

我的课程讲义里讲的有堆栈如何初始化以及如何实现堆栈的块拷贝操作，这里示范一下，代码在上次上传的 ex6 实验里面。

我还是直接上代码：

```
.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:

    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*/
```

```
brkpoint:
    swi 0x00900001

.end
```

同样大家在仿照上面的例子自己写个 makefile 文件。

运行，调试，先弄个断点在 brkpoint 处，然后单步调试。

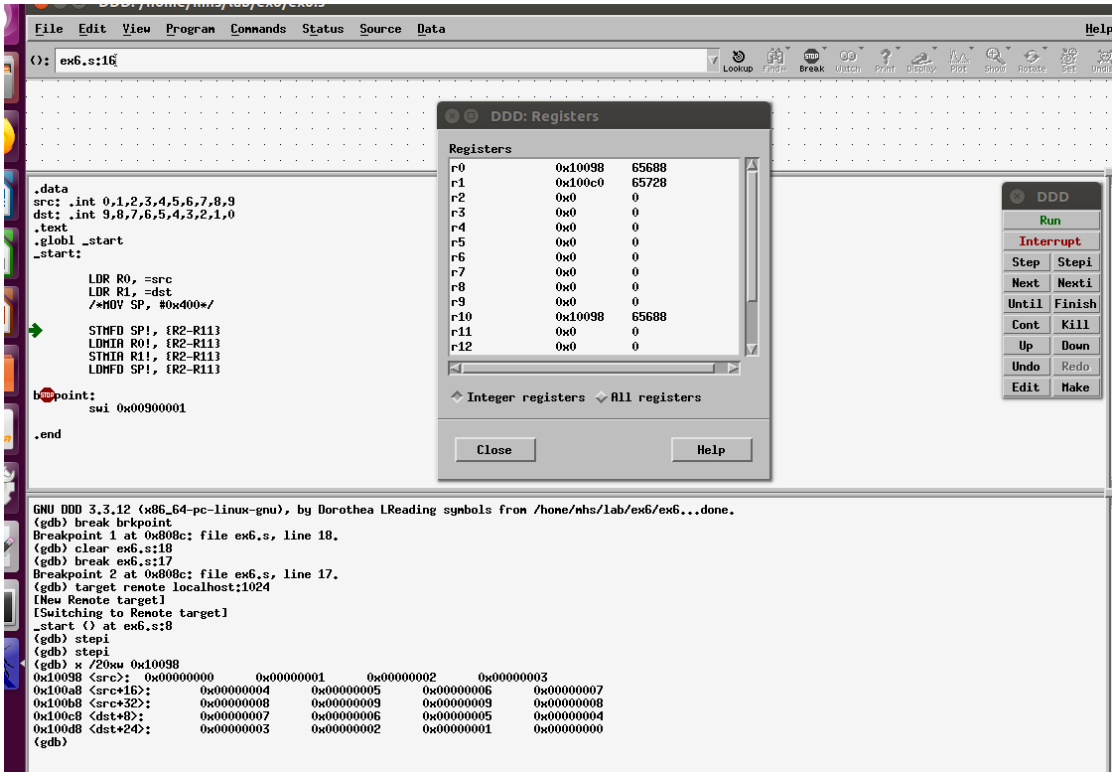

图 10 堆栈操作进行块数据搬运

大家从图 10 可以看到，在进行操作之前内存中 src 和 dst 的数据存放。然后你可以单步运行，一步步看数据是如何变化的。

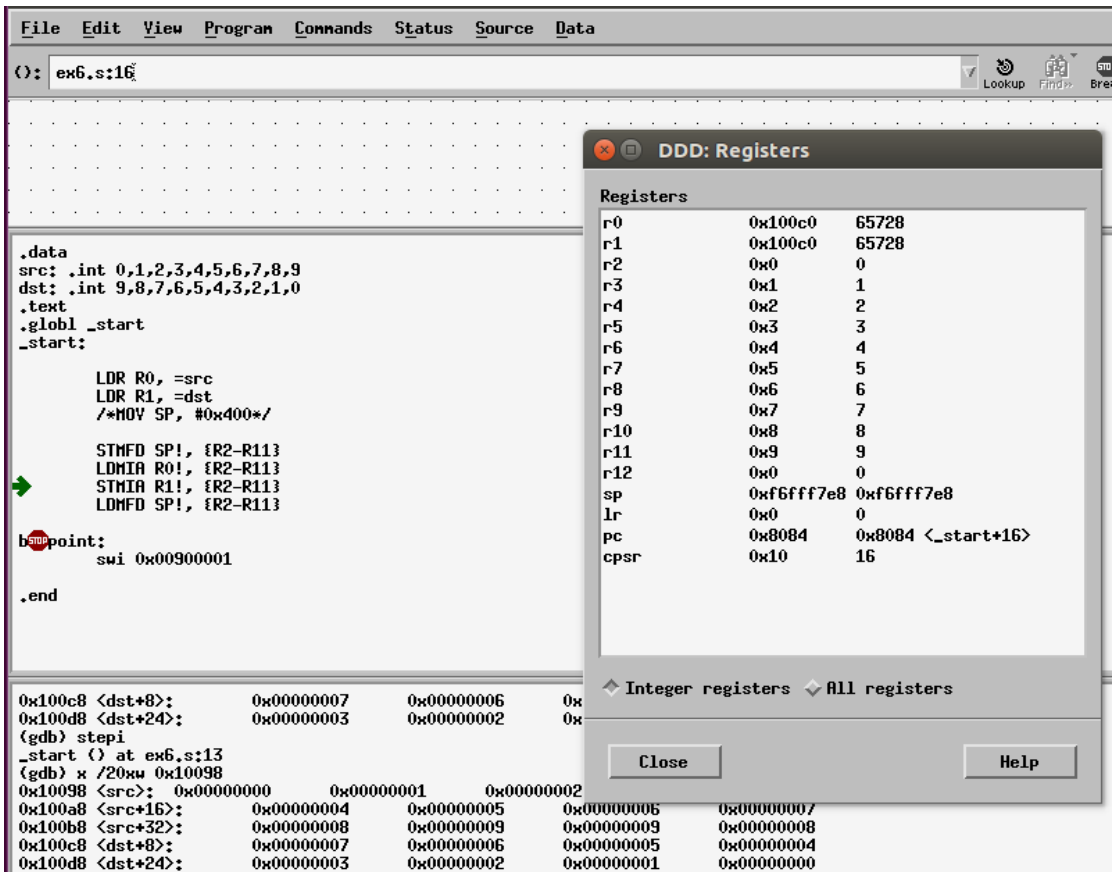

图 11 将 R2-R11 放到堆栈保护起来，再将 src 的数据搬到了 R2-R11 中

看随着单步运行，我们一次性将内存中 src 的数据搬到 r2-r11 中，然后我们再单步运行一下。

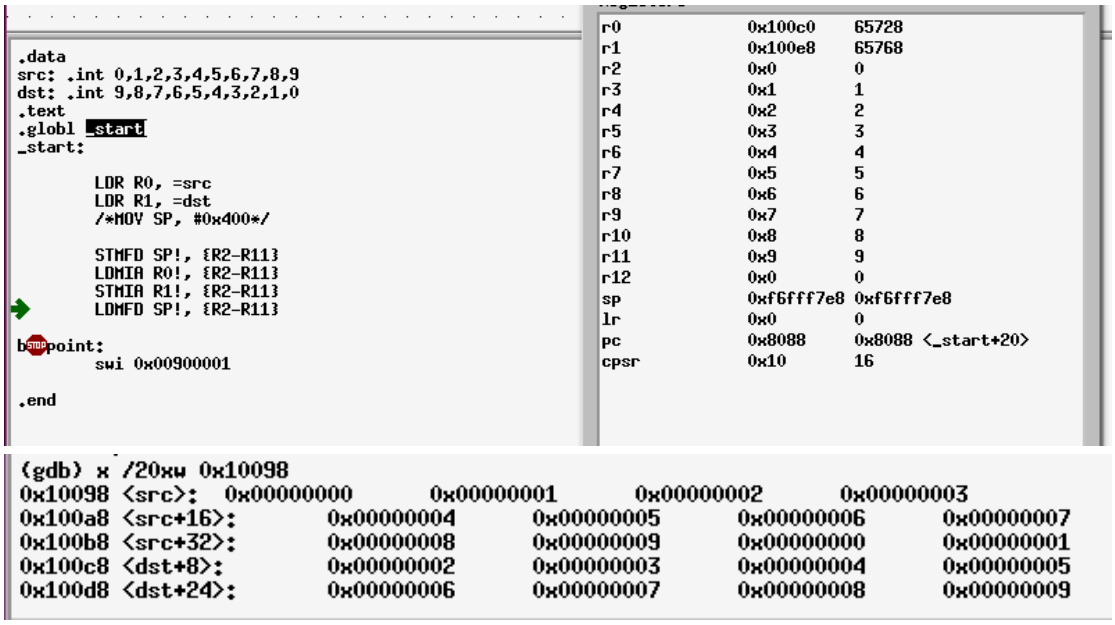

图 12 修改 R1 指向的 dst 数据块

可以看到内存中 dst 区已经被改写为 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 了。

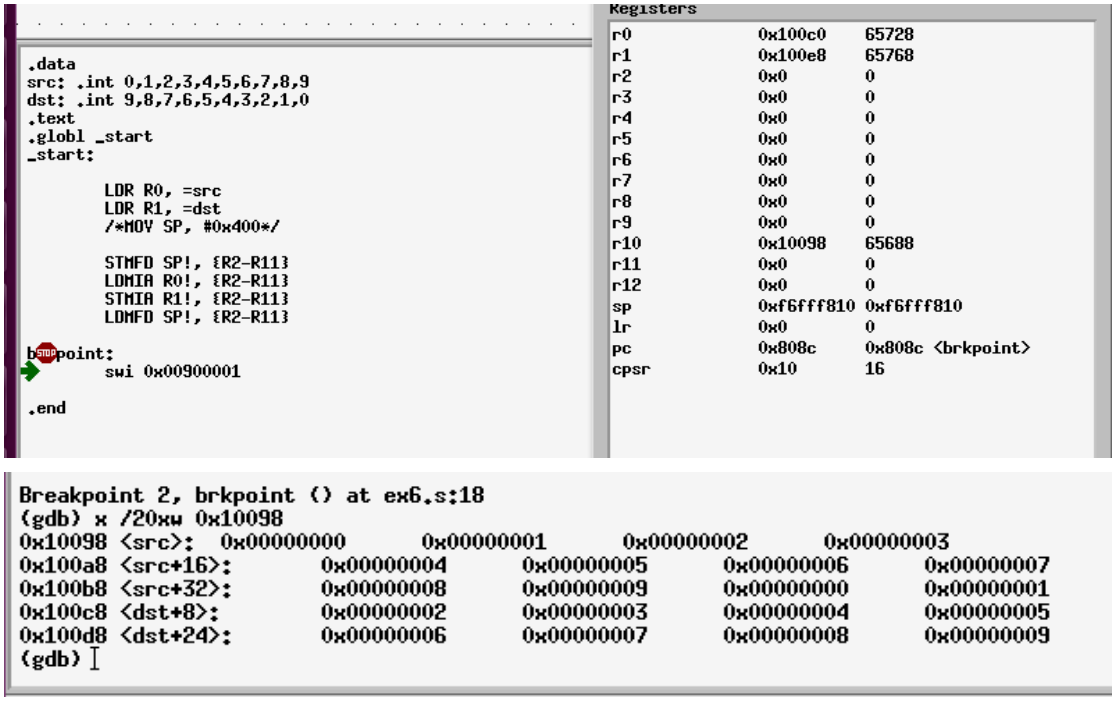

图 13 恢复寄存器内容

最后我们从堆栈中又恢复了 r2 到 r11 的初始值。可以看看和图 10 的时候寄存器里面的值是一样的了。