原文链接：<https://azeria-labs.com/writing-arm-assembly-part-1/>

ARM Exploit Development https://azeria-labs.com/writing-arm-shellcode/

第一篇： ARM汇编简介

第二篇： 数据类型和寄存器

第三篇： ARM指令集

第四篇： 内存读写

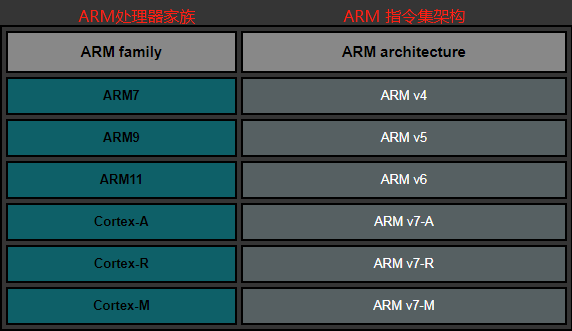
第五篇： 高级内存读写

第六篇： 条件分支

第七篇： 栈和函数

**第一篇： ARM汇编简介**

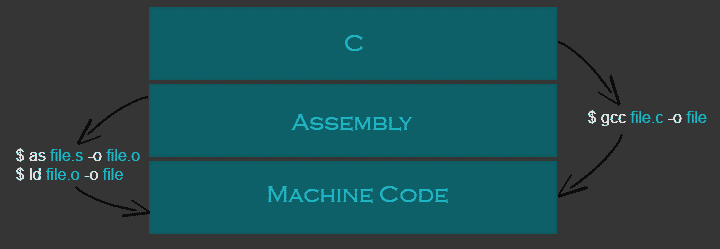
ARM处理器版本和指令集版本之间的映射关系

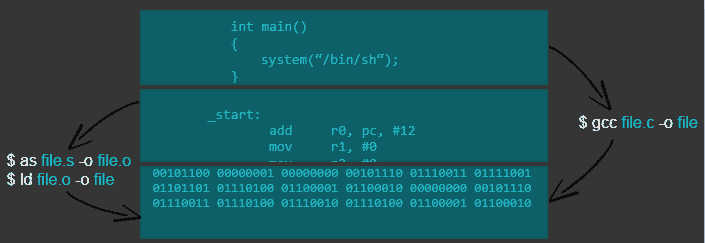


当你编写了后缀为“.s”的汇编文件，你可以使用as将它汇编，最后使用ld链接，如下所示：

$ as program.s –o program.o

$ ld program.o –o program





拓展阅读

1. Whirlwind Tour of ARM Assembly. https://www.coranac.com/tonc/text/asm.htm

2. ARM assembler in Rasberry Pi. http://thinkingeek.com/arm-assembler-raspberry-pi/

3. Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation by Bruce Dang, Alexandre Gazet, Elias Bachaalany and Sebastien Josse.

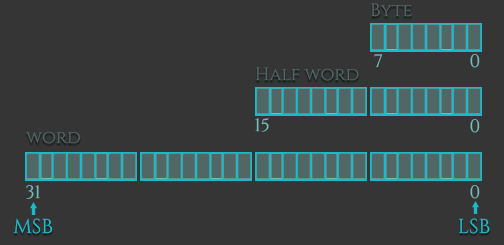
4.ARM Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.dui0068b/index.html

5. Assember User Guide. <http://www.keil.com/support/man/docs/armasm/default.htm>

**第二篇： 数据类型和寄存器**

**数据类型**

和高级语言类似，ARM支持对不同数据类型进行操作。可以供我们载入（load）或者存储（store）的数据类型可以分为有符号和无符号类型的字，半字，或字节。对这些数据类型的扩展是：半字为-h，-sh，字节为-b或者-sb，字没有扩展。有符号类型和无符号类型的区别是：



有符号类型可以包含正数和负数，因此它的取值范围较小

无符号类型可以包含正数（包括0），但是不能包含负数，因此他的取值范围更大

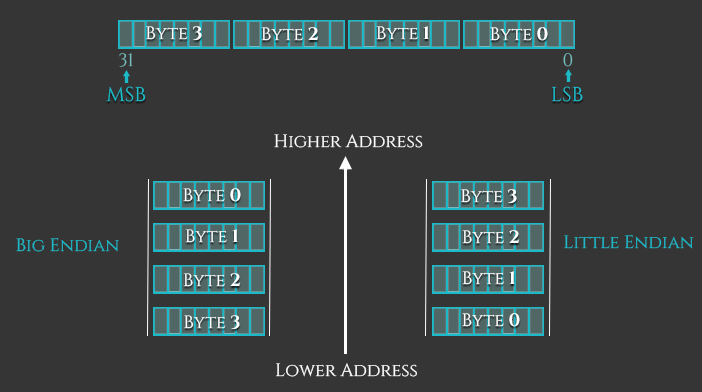
以下列举了几个例子，来说明指令集是如何存储和载入这些数据类型的



**字节顺序**

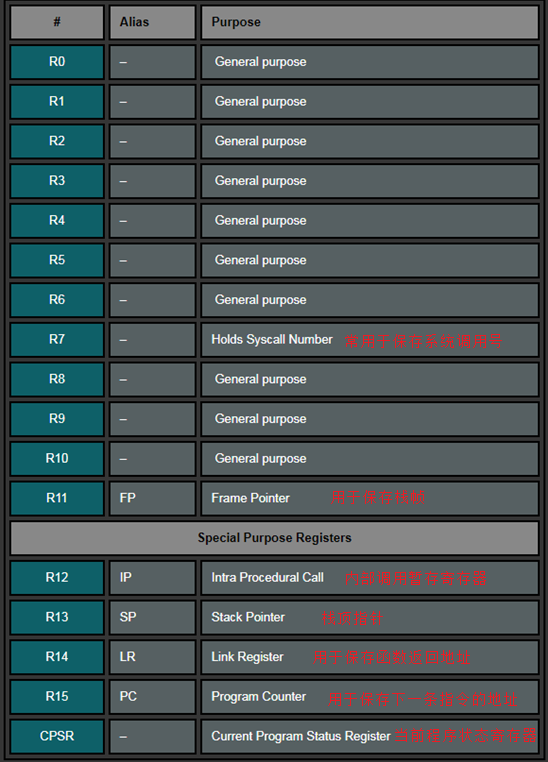
以ARMV6为例，指令的固定小字节和数据访问（fixed little-endian and data accesses）既可以是大字节序，也可以是小字节序，程序状态寄存器（CPSR）的第9位，或者叫E位来控制的。

用大尾端和小尾端来理解



**ARM寄存器**

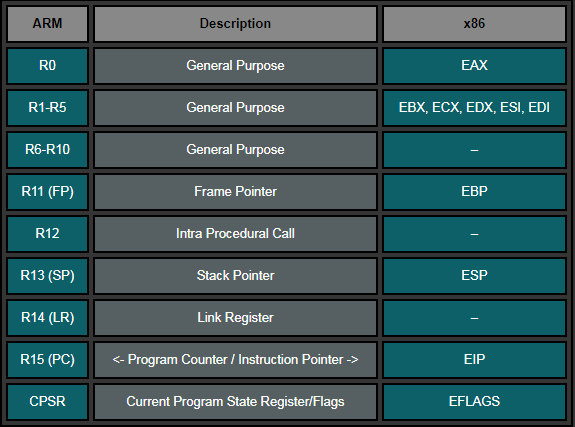
寄存器的数量取决于ARM的版本。根据ARM参考手册可知，有30个32位的通用寄存器（除了 ARMv6-M和ARMv7-M的处理器）。在本基础系列课程中，我们学习的对象是在任意特权模式下都可以访问的寄存器：R0-R15。这16个寄存器可以被分成两组：通用寄存器和特殊功能寄存器。



（ 从r0到r11都是通用寄存器 ）

（特殊功能寄存器，R12是IP寄存器，内部程序调用寄存器。R13，SP，堆栈指针寄存器。R14，LR，连接寄存器。R15，PC，程序计数器。CPSR，当前程序状态寄存器）

下表是ARM寄存器和英特尔处理器的寄存器存在的相关联性的概览



R0-R12可以在通常的运算过程中用来存储临时的数据，指针（定位内存）等。以R0为例，当我们执行算数运算或者存储当前函数的返回值时，可以把R0视为累加器。系统调用发生时，R11开始生效，它存储了系统调用数值。R11作为栈帧指针帮助我们追踪栈帧的边界（稍后会讲到）。此外，ARM专用的函数调用规则规定了函数的前四个参数应该分别存贮与R0到R3中。

R13：SP（堆栈寄存器）堆栈寄存器指向栈的栈顶。栈是内存中一块用于存储特定数据的存储区，它用于回收函数的返回值。因此，栈指针别用来在栈空间中分配出区域，如果我们想获得32位的数据空间，我们就让栈指针减去4。

R14：连接寄存器，当有一处函数调用时，连接寄存器会用一处内存地址来获取更新，该地址是函数初始化的地方的下一行代码的地址。这么做可以允许“子函数”执行完毕后让程序返回到起“父函数”的起始地址。

R15：PC（程序计数器）。程序计数器会根据指令的大小，在指令被执行时自动增加。一条指令的大小在ARM状态下总是4个字节，在THUMB模式下总是2个字节。当一条分支指令正在被执行时，PC保持存贮着目标地址。当执行该指令时，PC存贮了当前指令的地址加8字节（ARM状态下的两条ARM指令）， 或当前指令的地址加4字节（Thumb（V1）模式下的两条Thumb指令的大小）。和x86不同的是，x86下PC永远会指向下一条要被执行的指令。

我们来看看PC在调试器中的表现形式。我们让接下来的程序，在R0存储PC的地址，并且包含了两条随机的指令。让我们看看究竟会发生什么。

.text

.globl main

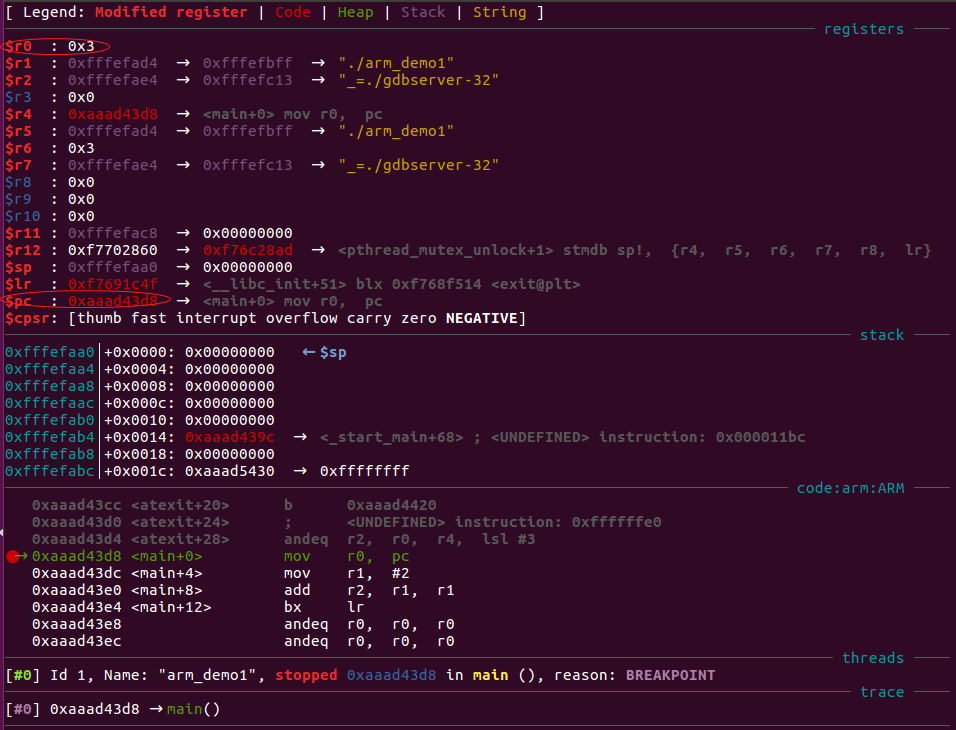
main:

mov r0, pc

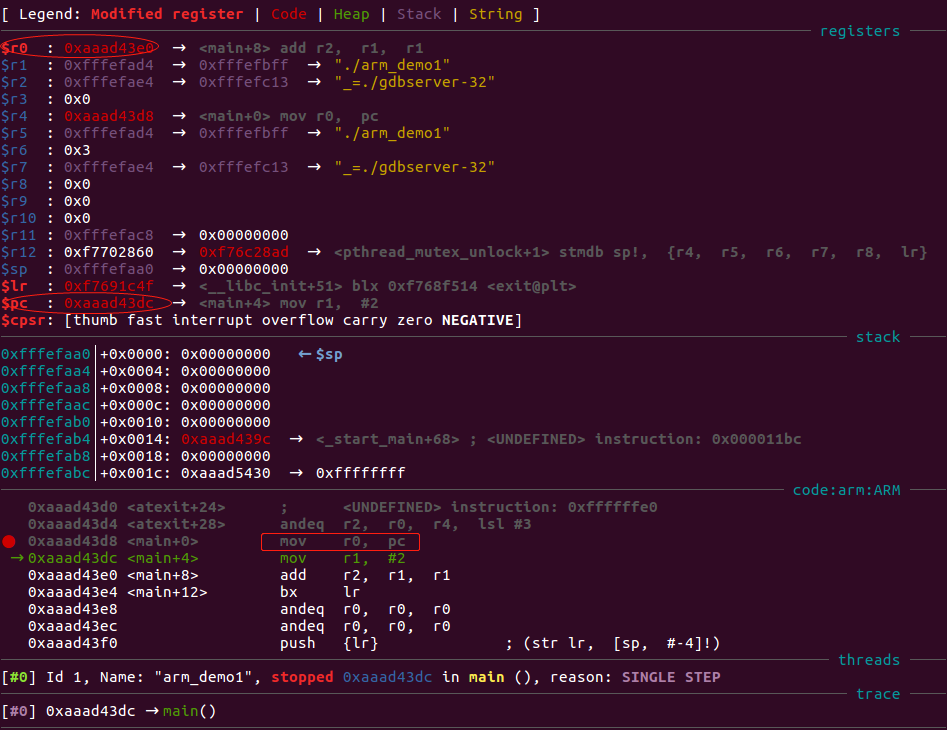
mov r1, #2

add r2, r1, r1

bx lr



我们可以看到，PC里保持着下一条将被执行的指令 (mov r0, pc)的地址0xaaad43d8，现在执行mov r0, pc指令之后R0应该是0xaaad43d8吗？



看看R0中的地址。R0是0xaaad43e0。

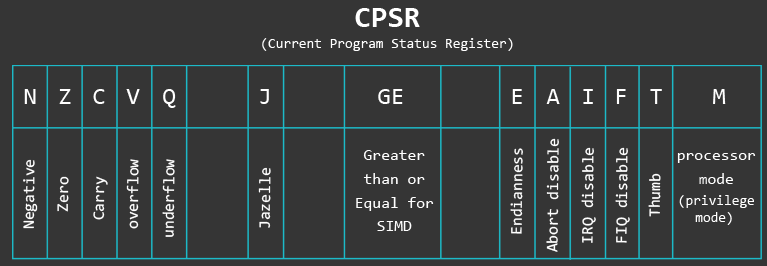
下面有两个解释，感觉第二个更靠谱一点：

1、从这个地址我们可以看出，当我们直接读取PC时，它按照定义，PC指向下一条指令，但是当我们调试程序时，PC却指向当前PC值的下面两条指令的地址处（0xaaad43d8+8=0xaaad43e0）。这是因为，老款的ARM处理器总是获取当前已经执行的指令的后两条指令的地址。ARM保留着这个定义的原因是为了保证和早期处理器的兼容性。

2、产生这种差异的原因其实很简单，调试器显示的PC寄存器的值是经过处理的。当0xaaad43d8处的指令被执行时，PC寄存器已经指向了0xaaad43d8+0x8处的指令，这是由于CPU的流水线机制导致的。CPU取指令，解码指令和执行指令时使用的是不同的硬件部件，因此，这几个操作（实际的CPU可能更复杂，有更多的操作步骤）是可以并行执行的。因为RISC CPU的指令长度一定，所以CPU可以在解码指令之前就知道下一条指令的长度，从而在解码指令时取下一条指令，在执行指令时，对下一条指令进行解码，并取下下一条指令，这称为三级流水线。所以ARM当我们在执行0xaaad43d8处的指令时，PC已经指向0xaaad43d8处进行取指令操作了。这是硬件中真实发生的情况，而调试器为了令展示更有逻辑性，所以PC寄存器显示了当前执行指令的地址，当我们真实调试时不要受此影响。

当前程序状态寄存器

cpsr寄存器显示出CPSR寄存器的当前值，在他的下面一行，你可以看到标志位 thumb, fast, interrupt, overflow, carry, zero, 以及 negative。这些标志位显示了CPSR寄存器中的某些特定的位，根据CPSR的值，当某个位被激活时，对应的字体会变成粗体。其中，N，Z，C和V标志位分别和x86寄存器的SF，ZF，CF和OF标志位表示的含义一一对应。这些标志位用来在汇编级别条件执行指令和循环指令中支配他们执行。



上图显示了32位寄存器CPSR的布局，左侧 (<-) 是最高有效位，右侧 (->) 是最低有效位。每一个单独的单元格（除了GE，M和空白的单元格），其大小都是一个位。这些位表示了针对程序当前状态的不同的属性。



|  |  |
| --- | --- |
| 标记 | 含义 |
| N（Negative） | 指令执行结果为负时置1 |
| Z（Zero） | 指令执行结果为0时置1 |
| C（Carry） | 加法有进位则置1否则置0，减法有借位则置0否则置1 |
| V（overflow） | 指令执行结果超出32位补码存储范围时置1 |
| E（Endian-bit） | 置0时使用小端序，置1时使用大端序 |
| T（Thumb-bit） | 置1时使用Thumb模式，置0时使用ARM模式 |
| M（Mode-bit） | 共5位表示处理器运行模式 |
| J（Jazelle） | 对于有的处理器，置位表示允许以硬件执行java字节码 |

我们假设，我们要使用CMP指令来比较数字1和数字2。结果会是“负的”因为1-2=-1。当我们比较两个相等的数字，比如2和2比，Z（0）标志位会被置位，因为2-2=0。记住，用于CMP指令的寄存器的值不会改变，只有CPSR会基于这些寄存器里的值的比较运算结果的改变而被编辑。

以下是从ARM消息中心的摘录下来的

APSR包含了以下ALU状态标志位：

N– Set when the result of the operation was Negative.当运算结果为负数时置位

Z– Set when the result of the operation was Zero. 当运算结果为0时置位

C– Set when the operation resulted in a Carry. 当运算结果导致进位时置位

V– Set when the operation caused oVerflow. 当运算结果出现溢出时置位

A carry occurs:以下为产生进位的情况：

1.if the result of an addition is greater than or equal to 232 如果相加的结果大于或等于232时

2.if the result of a subtraction is positive or zero 如果相减的结果是正数或是0

3.as the result of an inline barrel shifter operation in a move or logical instruction. 在赋值操作或者逻辑指令中，进行内联桶式移位操作的结果（？？？）

Overflow occurs if the result of an add, subtract, or compare is greater than or equal to 231, or less than 231.

当加法，减法或者比较指令的结果大于或者等于231或是小于或者等于-231时会产生溢出。

.text

.globl main

main:

mov r1, #1

mov r2, #2

cmp r1, r2

cmp r2, r1

cmp r2, r2

bx lr

假设我们用cmp指令来比较1和2，结果将为负，Negative标志位被置1。因为cmp指令执行一次隐式的减法操作，1-2=-1。然而，如果我们比较2和1（和刚才相反），减法操作不借位，Carry标志位被置1。如果我们比较两个相同的数，比如2和2，那么2-2=0，在Carry标志位置1的同时，Zero标志位也被置1。

**第三篇： ARM指令集**

ARM和Thumb

ARM处理器有两种工作状态ARM和Thumb（此处先不考虑）。这两种工作状态和运行模式没有任何关系。比如不论是ARM还是Thumb状态的代码都可以运行在用户模式下。这两种工作状态之间最大的差异是指令集，ARM状态的指令长度是32位的，Thumb状态的指令长度是16位的（也可能为32位）。了解如何使用Thumb工作状态对于编写ARM平台的漏洞利用是至关重要的。当我们编写ARM shellcode时，需要使用16 bit的Thumb指令代替32 bit的ARM指令，从而避免在指令中出现’\0’截断。

容易引起混淆的是，不同的ARM版本，支持的Thumb指令集并不相同。在某些版本中，ARM引入了扩展的Thumb指令集（也就是Thumb-2），它支持32 bit指令以及条件执行。这在原本的Thumb指令中都是不受支持的。为了在Thumb状态下支持条件执行，“it”指令被引入。然而，可能是为了简化指令集，这个指令在后来的版本中被删除了。我认为这种设计反而增加了兼容的复杂度。不过，当然我认为没必要知道所有ARM版本的ARM/Thumb指令集变体，我建议你也不必在这上面浪费太多时间。你只需要知道目标设备的版本和该版本对Thumb指令有哪些特殊支持，然后调整你的代码就好了。ARM Infocenter可以帮助你了解各个ARM版本的具体细节（http://infocenter.arm.com/help/index.jsp）。

我们已经知道了Thumb有不同的版本，下面我们对不同的版本做一下简单的介绍，注意不同的命名只是为了区分不同的版本（换句话说，处理器只知道它运行在Thumb状态，其它一概不知）。

Thumb-1（16位指令）：用于ARMv6和更早的版本。

Thumb-2（16位和32位指令）：对Thumb-1的扩展，添加了更多指令并允许它们为16位或32位宽（ARMv6T2，ARMv7）。

ThumbEE：在Thumb-2基础上包含了针对动态代码生成（代码在执行前或执行期间编译代码）的一些变更和补充。

ARM和Thumb的区别：

条件执行：ARM状态下的所有指令都支持条件执行。某些ARM处理器版本允许使用IT指令在Thumb中进行条件执行。条件执行提高了代码密度，因为它减少了要执行的指令数量，并节省了昂贵的分支指令。

32位ARM和Thumb指令：32位Thumb指令具有.w后缀。

桶形移位器是另一种ARM模式特有的功能。它可以将多个指令合并成一个。例如，您可以通过使用如下指令（将移位包含在MOV指令内）左移1位“Mov R1，R0，LSL＃1; R1 = R0 \* 2”从而代替两个乘法指令（只用乘法指令将寄存器的值乘以2，并使用MOV将结果存储到另一个寄存器中）。

要切换处理器的执行状态，必须满足以下两个条件之一：

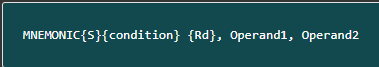
我们可以使用分支指令BX（分支和状态切换）或BLX（分支，返回和状态切换），并将目标寄存器的最低有效位置1。这可以通过将1添加到偏移量（如0x5530+1）来实现。你可能会认为这会导致对齐问题，因为指令总是2或4字节对齐的。然而，这么做不会导致问题，因为处理器在读取指令时是忽略最低有效位的。

如果当前程序状态寄存器中的T位置1，我们知道我们处于Thumb模式。

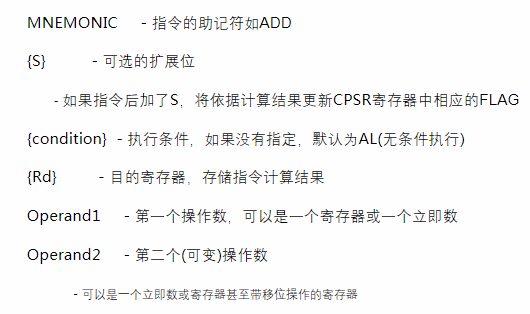
ARM指令简介

这一节的目的是简要介绍ARM的指令集和它的基本用法。作为汇编语言的基本单位，了解指令的用法，指令间的如何关联以及将指令进行组合能实现什么功能对于学习汇编语言是至关重要的。

ARM汇编由ARM指令组成。ARM指令通常跟一到两个操作数，我们使用如下模板描述：



需要指出的是，只有部分指令用到了指令模板中的所有域。模板中各字段的作用如下所示：



助记符、S扩展位、目的寄存器和第一个操作数的作用很好理解，不多做解释，这里补充解释一下执行条件和第二个操作数。设置了执行条件的指令在执行指令前先校验CPSR寄存器中的标志位，只有标志位的组合匹配所设置的执行条件指令才会被执行。第二个操作数被称为可变操作数，因为它可以被设置为多种形式，包括立即数、寄存器、带移位操作的寄存器，如下所示：

#123 - 立即数

Rx - 寄存器比如R1

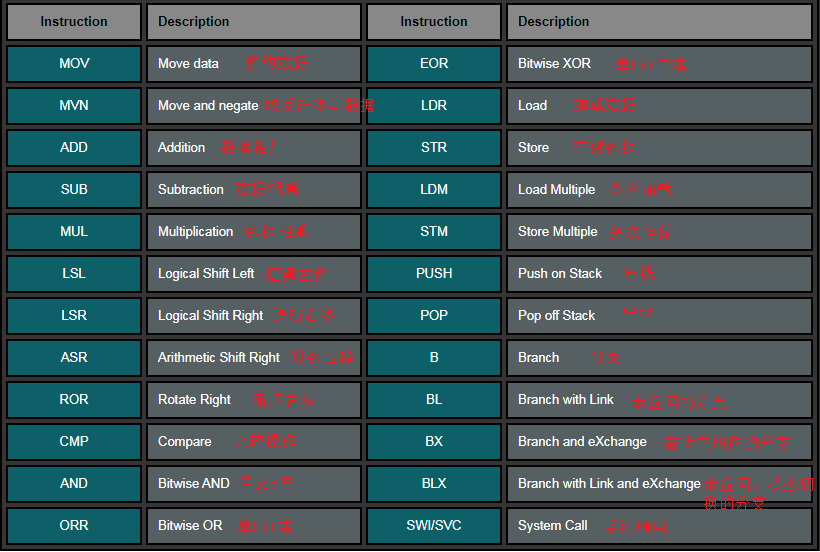
Rx, ASR n - 对寄存器中的值进行算术右移n位后的值

Rx, LSL n - 对寄存器中的值进行逻辑左移n位后的值

Rx, LSR n - 对寄存器中的值进行逻辑右移n位后的值

Rx, ROR n - 对寄存器中的值进行循环右移n位后的值

Rx, RRX - 对寄存器中的值进行带扩展的循环右移1位后的值



栈

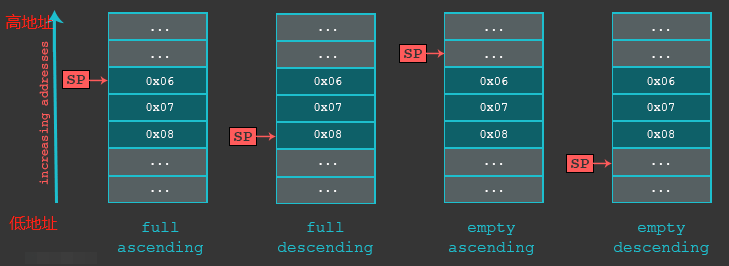
栈用来存储临时数据，使用push，pop指令来和栈交互，push和pop是一系列其他的内存相关指令的别名，而不是实际指令，使用push和pop是为了操作简便。

一条数据（32bit）将被放置的实际地址由栈指针定义，这个地址存储在sp寄存器中。

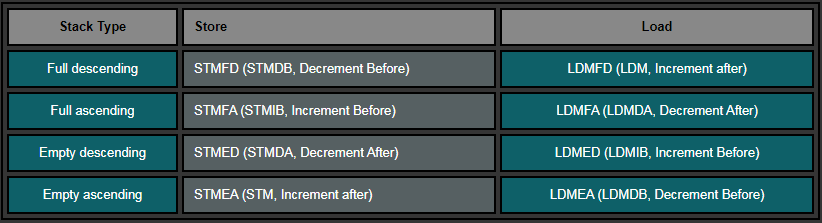
sp可能会指向栈中当前（最新）的的项，或者指向当前项的下一可用的内存槽。

sp当前指向栈中的最后一个项（full实现）

sp指向栈的下一个空白内存槽（empty实现），数据将首先被放置到这里



下面总结了不同栈实现方式，可以使用下表来描述在不同情况下，存储多个指令和载入多个指令在不同情形下是如何使用的



下面是一个使用full descending栈的例子

.text

.global main

main:

mov r0, #2

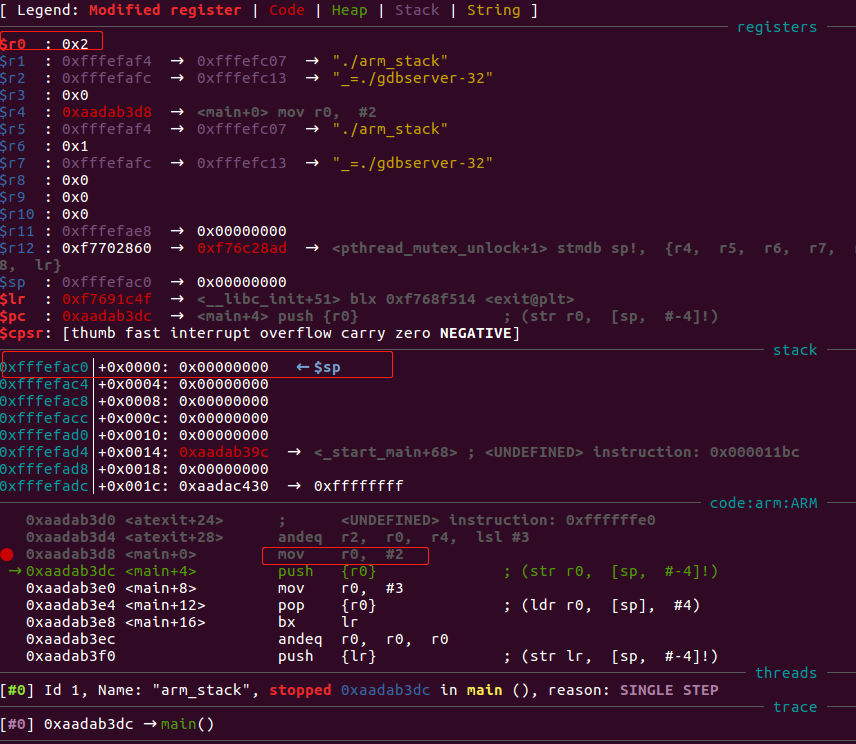
push {r0}

mov r0, #3

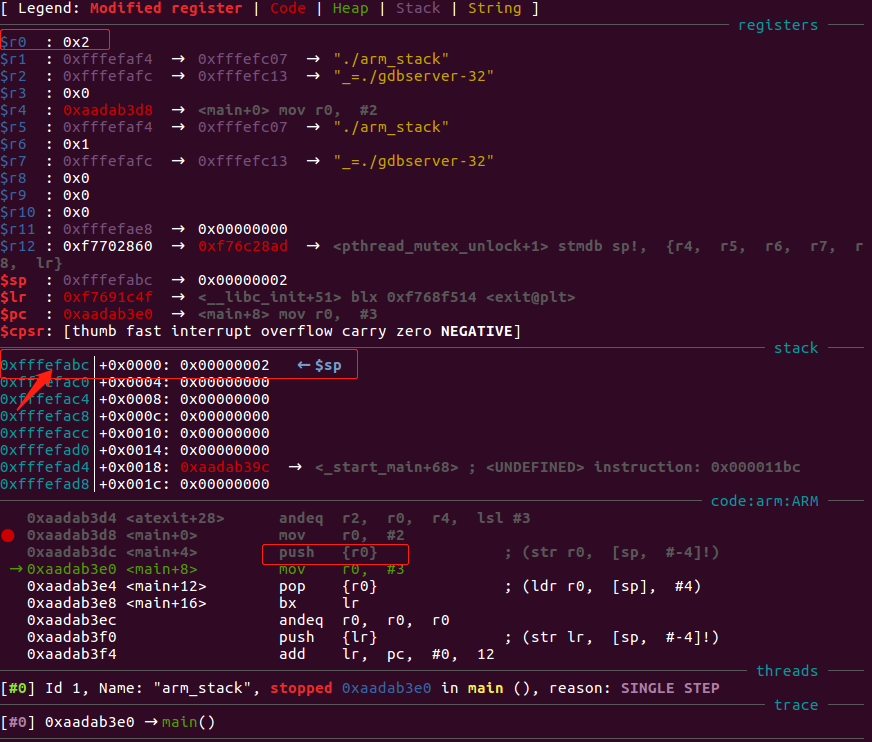
pop {r0}

bx lr

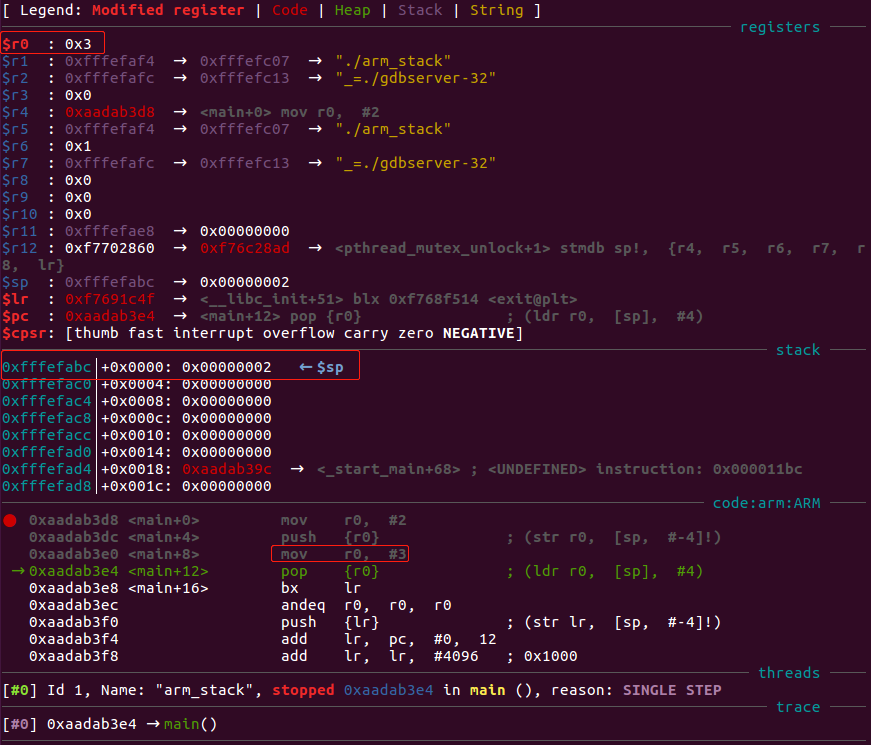
执行mov r0, #2指令后，sp的地址是0xfffefac0，r0是2



执行执行push {r0}指令后，sp的地址是0xfffefac0 – 4 = 0xfffefabc，r0是2

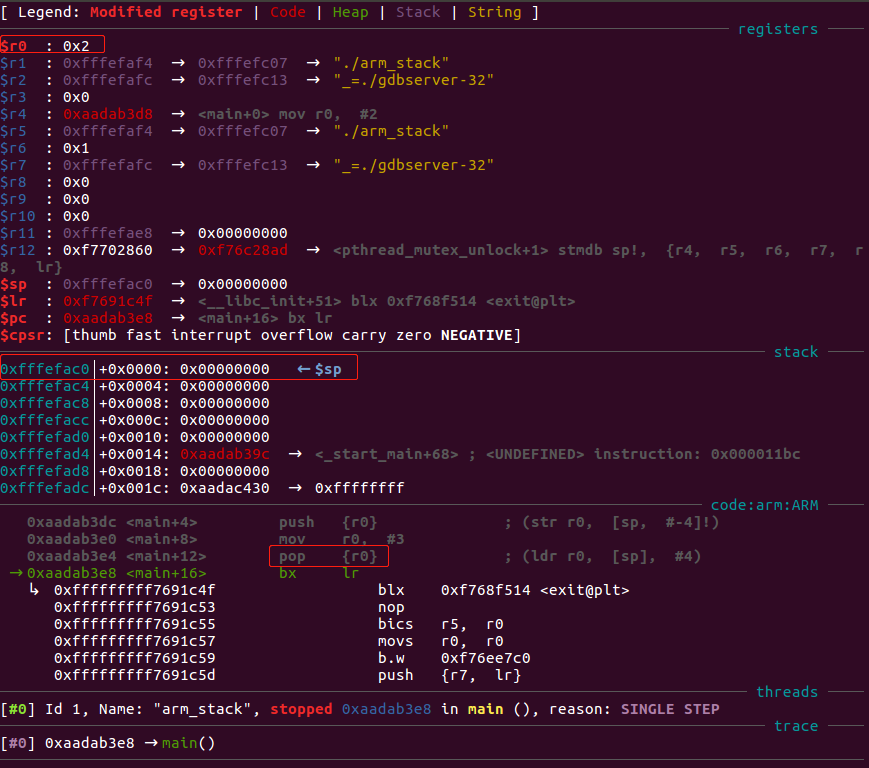


执行mov r0, 3指令后，sp的地址是没改变，r0是3；只是用来改变r0的值

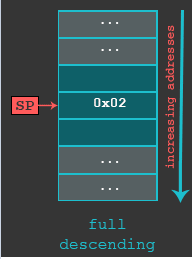


执行pop {r0}指令后，sp的地址是0xfffefabc + 4 = 0xfffefac0，r0是2

pop指令被执行，sp从地址0xfffefabc读取4字节的数据，让后sp寄存器0xfffefabc + 4。



可以看到gdb（cef）中栈的视图是



R11 FP 栈帧寄存器

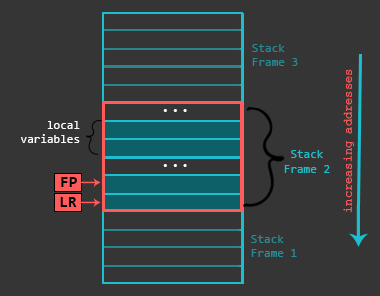
函数利用栈来保存局部变量，保存寄存器状态；为了让所有事物有序运行，函数使用栈帧，即栈中的一片本地化内存区域，专用于特定的函数；栈帧是在函数的prologue中创建的。

将栈帧指针(FP)设置到栈帧的底部，然后为栈帧分配的栈缓存会被开辟。

栈帧（从它的底部开始）通常包含返回地址（LR）、先前的帧指针、需要保存的任何寄存器、函数参数（如果函数接受超过4个参数）、局部变量等。

虽然栈帧的实际存储内容可能有所不同，但是之前概述的那些内容是最常见的。最后，栈帧在函数运行到结尾部分时被破坏。

下图是栈中栈帧的抽象简图



**函数**

ARM函数的结构组成

1、prologue 序言

2、body 函数主体

3、epilogue 尾声

prologue（序言）的目的是保持抽象的先前状态（通过将lr和r11的值存储到栈上），并为函数的局部变量开辟栈空间，例如：

push {r11, lr} 序言开始，将fp和lr保持到栈

add r11, sp, #0 设置栈帧的底部

sub sp, sp, #16 序言的终止，在栈上分配一些缓存区，也为栈帧分配一些内存空间

函数的主体部分通常负责执行某种特定的任务，可以包含多种指令，分支（跳转）到其他函数等，例如：

mov r0, #1

mov r1, #2

bl max

上面的示例代码显示了一个函数的片段，它设置局部变量，然后分支到另一个函数。这段代码还告诉我们，函数的参数（在这种情况下是函数max的参数）如何通过寄存器传递。在某些情况下，当有超过4个参数需要被传递时，我们将使用堆栈来存储剩余的参数。还值得一提的是，函数的结果是通过寄存器R0返回的。因此，无论函数（max）的结果究竟是什么，我们应该能够在函数返回之后，从寄存器R0中把它提取出来。还有一点要指出的是，在某些特定情况下，返回值的长度可能是64位的长度（超过32位寄存器的大小）。在这种情况下，我们可以使用R0与R1组合起来，来返回64位结果。

函数的最后一部分，尾声（epilogue），用来将程序恢复到初始状态（调用函数之前的状态），所以可以接着从函数被调用之前的位置继续往后执行。为了实现该目标，我们需要读取堆栈指针（SP）。这是通过使用栈帧指针寄存器（R11）作为参考并执行加法或者减法操作来完成的。当我们重新调整栈指针时，我们通过将它们从栈中弹出到各自的寄存器中来恢复先前（序言）保存的寄存器值。POP指令可能是结尾部分的最后指令，这取决于函数的类型 。但是，在恢复寄存器值之后，我们可能会使用BX指令来离开函数。尾声（Epilogue）的一个例子是这样的：

sub sp, r11, #0 尾声开始，跳转sp寄存器

pop {r11, pc} 尾声结束，从栈中回复之前的fp，并把之前保存的lr载入pc，跳转到那里继续执行。函数的栈帧至此全部销毁完毕

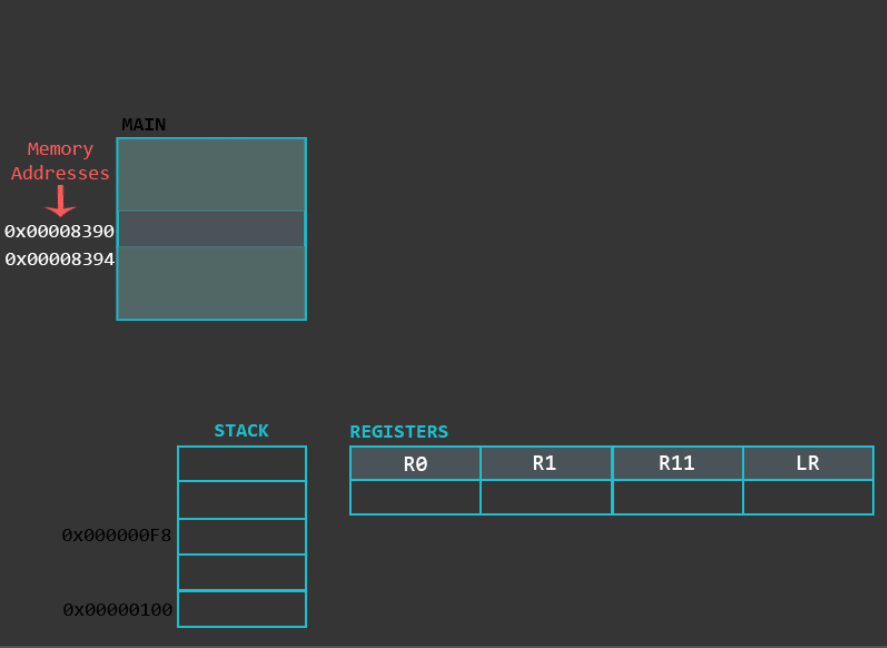
序言部分建立了函数的运行环境

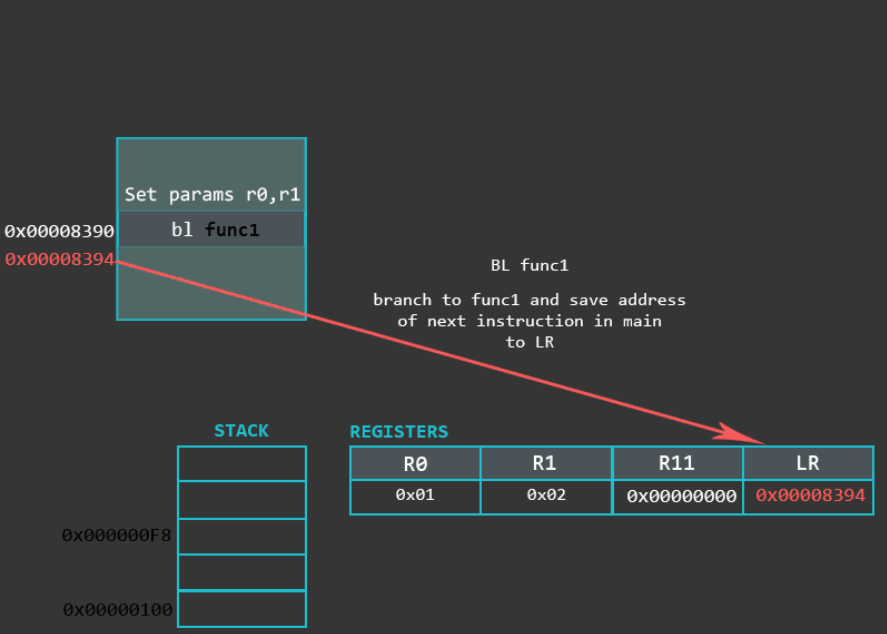
函数体部分实现函数的逻辑并将返回值存储进R0

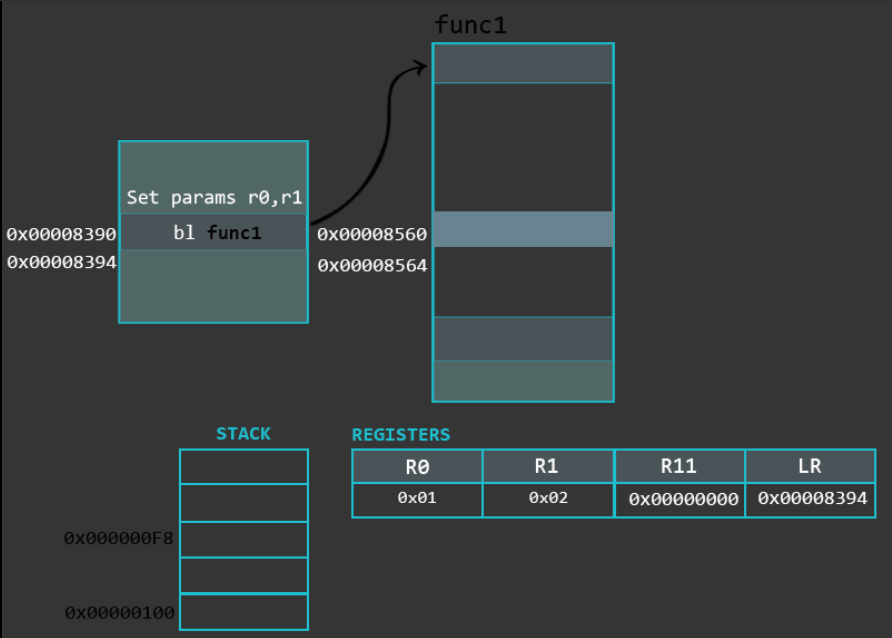
尾声部分恢复了函数被调用之前的状态并继续运行

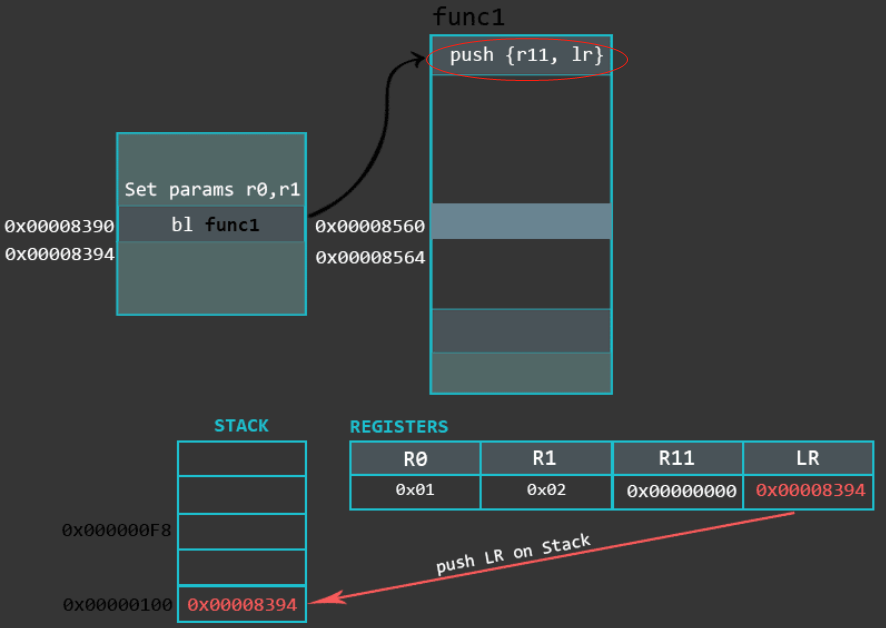
了解函数的另一个关键点是它们的类型：叶和非叶。叶函数是这样一类函数，它本身不调用/分支另一函数。非叶函数是一种函数，除了它自己的逻辑外，还得调用/分支到另一个函数。这两种函数的实现是相似的。然而，它们有一些不同之处。

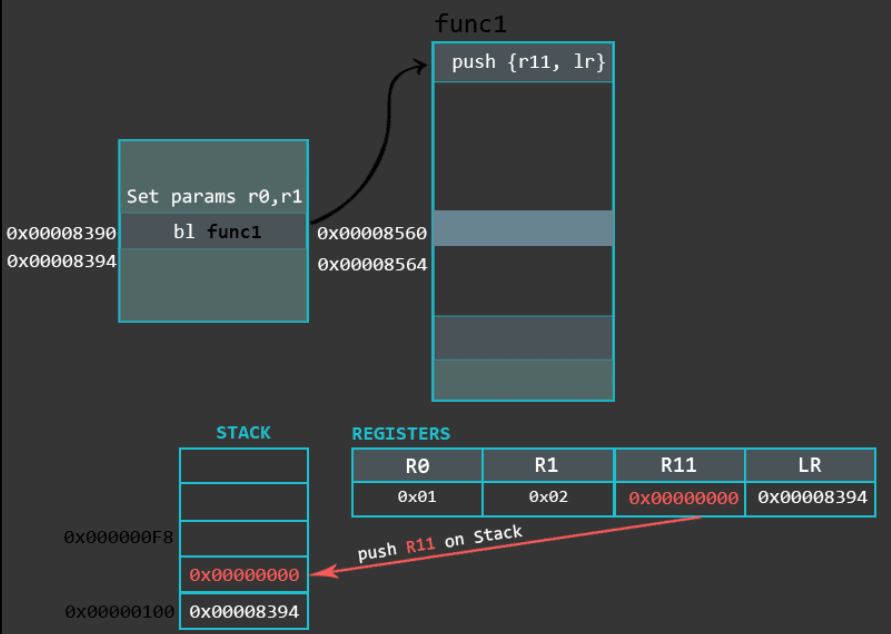
我们可以看看下面的动画，它说明了非叶和叶函数的内部工作状况

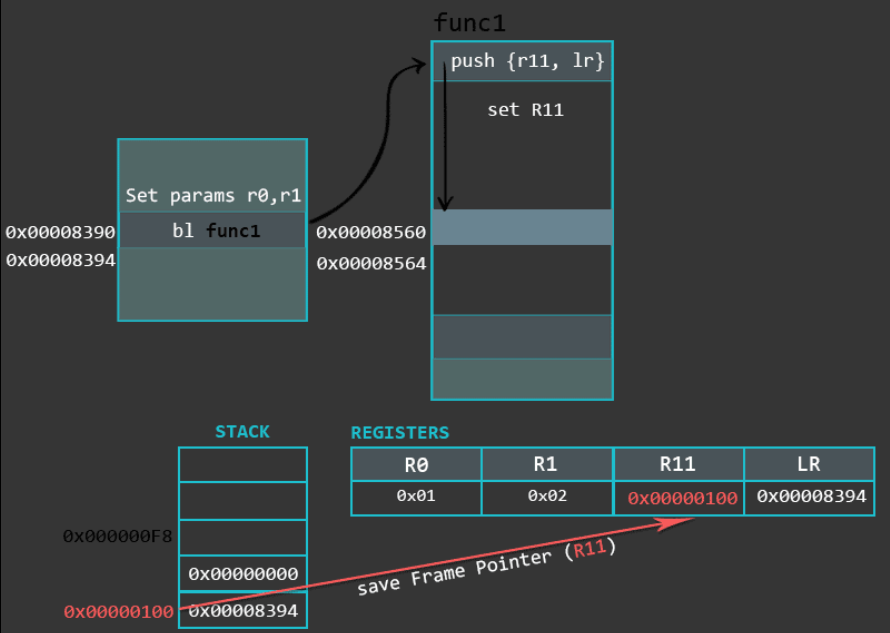


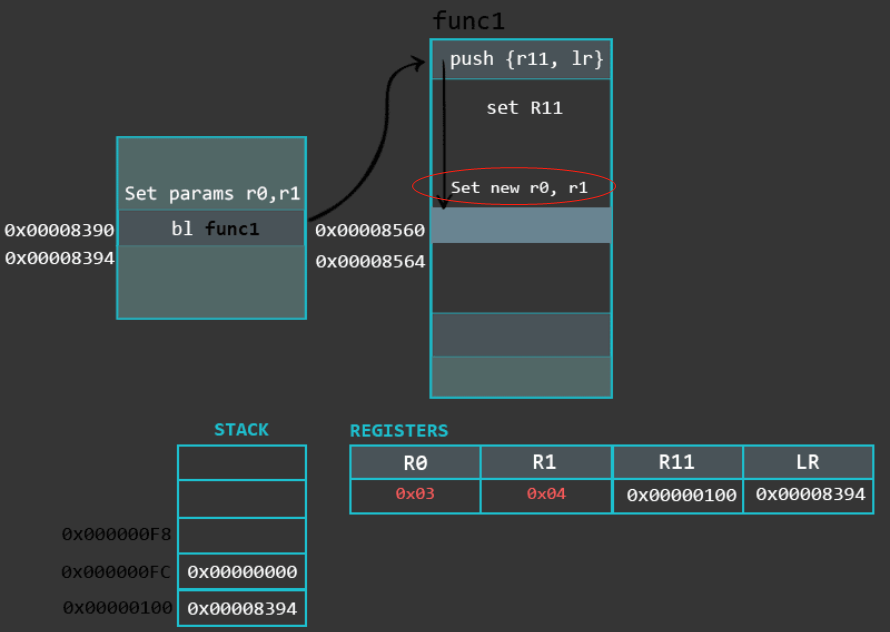


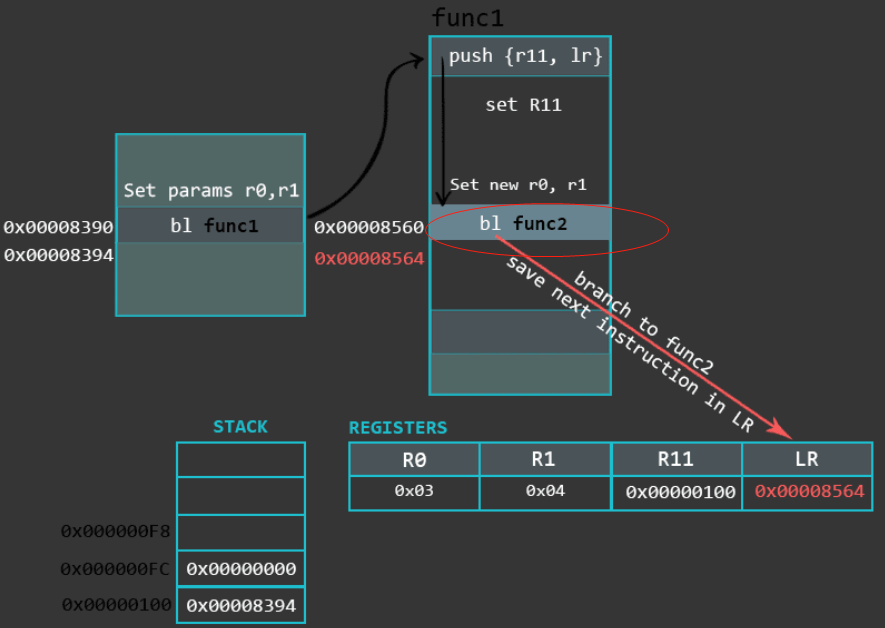


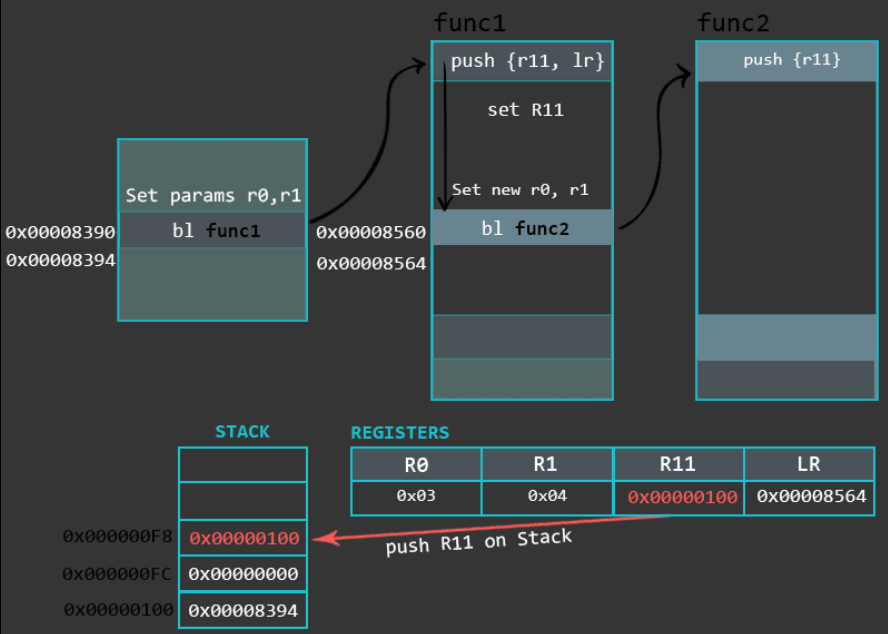


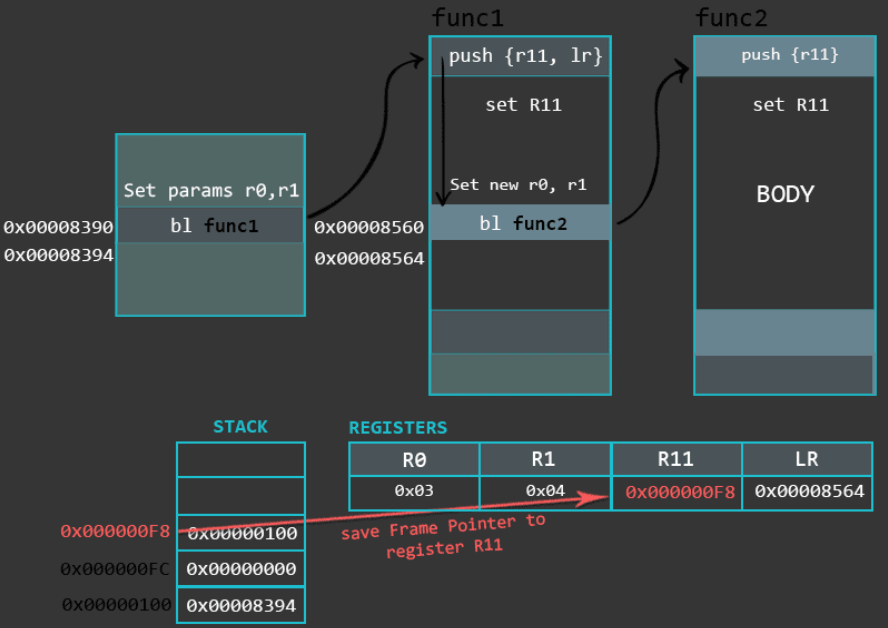


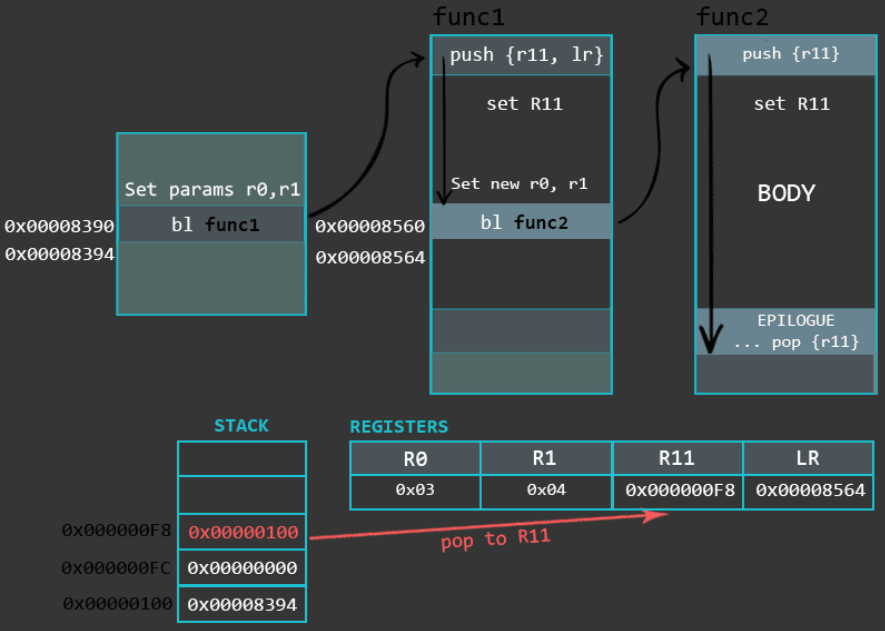


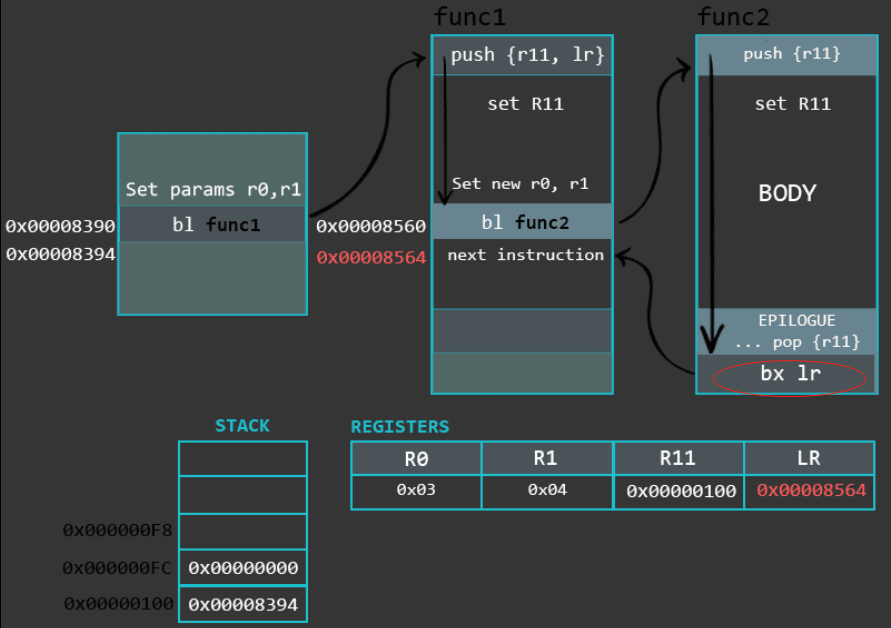


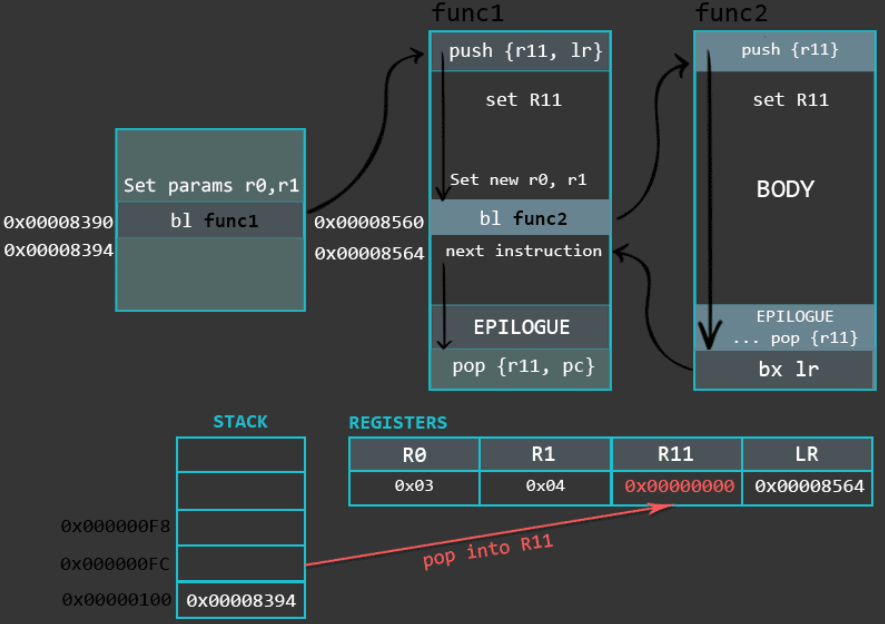


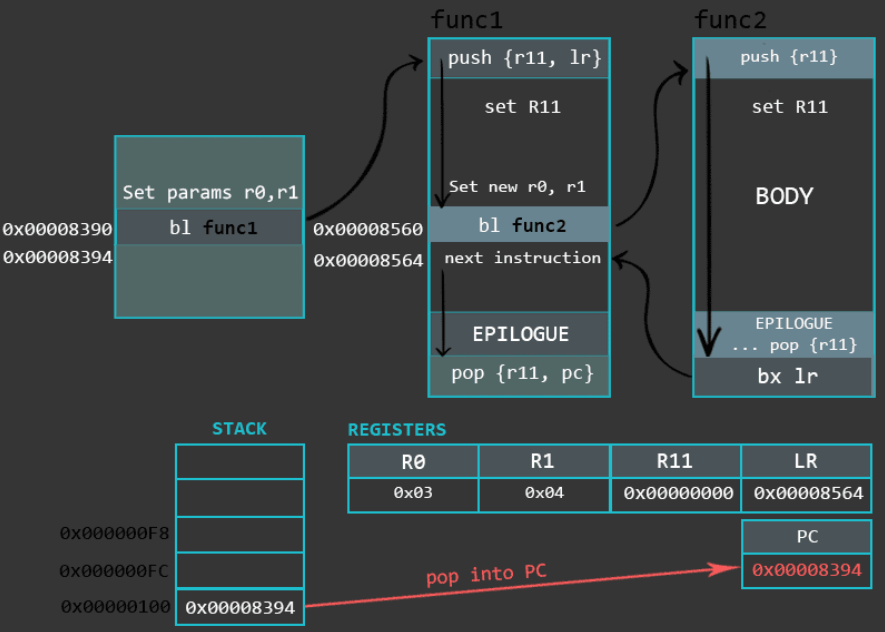


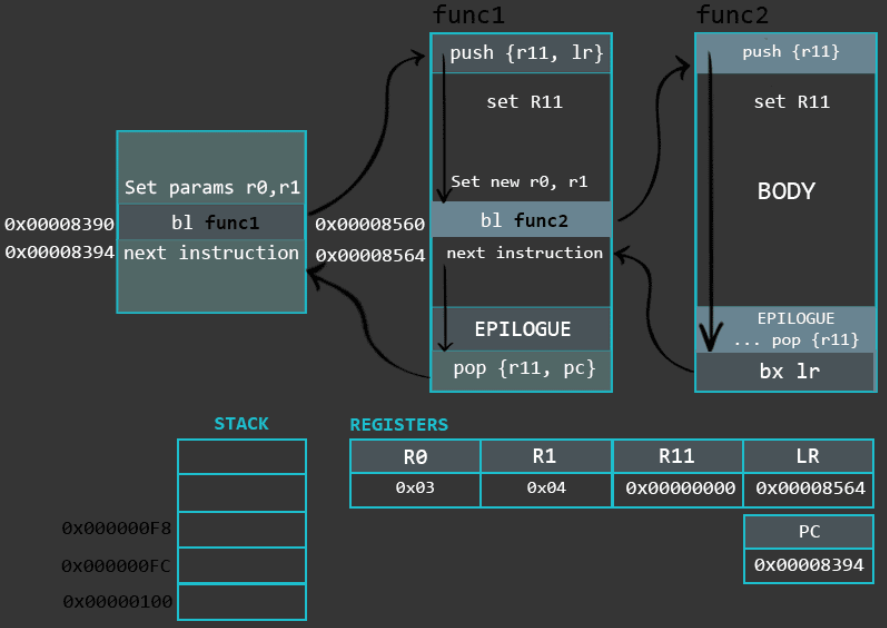












在google搜索引擎搜索

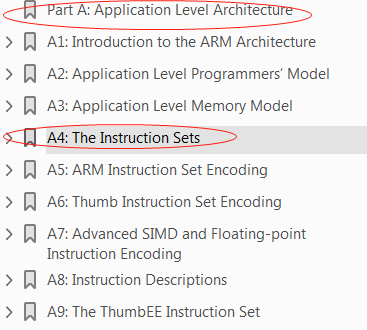
ARM Architecture Reference Manual ARMv7-A site:arm.com

可以找到最新的ARMv7-A文档，需要下载pdf

<https://developer.arm.com/documentation/ddi0406/latest/>

或者在<https://developer.arm.com/documentation>找

主要看PARA的A4章节



**this指针**

class Number{

public:

void setNum1(int n){

num1 = n;

}

int getNum1(){

return num1;

}

public:

int num2;

int num1;

//int num2;

};

int main(int argc, char\*\* argv){

Number cnumber;

cnumber.num1 = 1;

cnumber.num2 = 2;

cnumber.setNum1(3);

return cnumber.getNum1();

}

$r0 : 0xfffefab4 → 0x00000002

$r1 : 0x3

str r1, [r0, #4]

这里r0其实是this指针，this指针偏移4个字节，就是int num1

如果类中

int num1;

int num2;

那么setNum函数中汇编代码是

STR R1, [R0]

//--------- int num1; int num2;---------------

函数setNum1的汇编代码，查看伪代码

\_DWORD \*\_\_fastcall sub\_1430(\_DWORD \*result, int a2)

{

\*result = a2;

return result;

}

第一个参数result是Number cnumber;的this指针，恰好也是num1的地址

函数getNum1的汇编代码，查看伪代码

int \_\_fastcall sub\_1450(int a1)

{

return \*(\_DWORD \*)a1;

}

第一个参数a1是Number cnumber;的this指针，恰好也是num1的地址

//--------- int num2; int num1;---------------

函数setNum1的汇编代码，查看伪代码

\_DWORD \*\_\_fastcall sub\_1430(\_DWORD \*result, int a2)

{

\*(\_DWORD \*)(result + 4) = a2;

return result;

}

第一个参数result是Number cnumber;的this指针，恰好也是num1的地址

函数getNum1的汇编代码，查看伪代码

int \_\_fastcall sub\_1450(int a1)

{

return \*((\_DWORD \*)this + 1);

}

第一个参数a1是Number cnumber;的this指针，恰好也是num1的地址

如果程序strip过后，是看不到符号的提示的，发布的程序都是stip过的，没有strip的时候，IDA中的提示非常清晰

.text:000013E8 ; int \_\_cdecl main(int argc, const char \*\*argv, const char \*\*envp)

.text:000013E8 STMFD SP!, {R11,LR}

.text:000013EC MOV R11, SP

.text:000013F0 SUB SP, SP, #0x18

.text:000013F4 MOV R2, #0

.text:000013F8 STR R2, [R11,#var\_4]

.text:000013FC STR R0, [R11,#var\_8]

.text:00001400 STR R1, [SP,#0x18+var\_C]

.text:00001404 MOV R0, #1

.text:00001408 STR R0, [SP,#0x18+var\_10]

.text:0000140C MOV R0, #2

.text:00001410 STR R0, [SP,#0x18+var\_14]

.text:00001414 ADD R0, SP, #0x18+var\_14 ; this

由于Number cnumber是局部变量，所以在栈上面

.text:00001418 MOV R1, #3 ; int

.text:0000141C BL \_ZN6Number7setNum1Ei ; Number::setNum1(int)

.text:00001420 ADD R0, SP, #0x18+var\_14 ; this

.text:00001424 BL \_ZN6Number7getNum1Ev ; Number::getNum1(void)

.text:00001428 MOV SP, R11

.text:0000142C LDMFD SP!, {R11,PC}

.text:0000142C ; } // starts at 13E8

.text:0000142C ; End of function main

.text:00001430 ; \_DWORD \_\_fastcall Number::setNum1(Number \*\_\_hidden this, int)

.text:00001430 SUB SP, SP, #8

.text:00001434 STR R0, [SP,#8+var\_4]

.text:00001438 STR R1, [SP,#8+var\_8]

.text:0000143C LDR R0, [SP,#8+var\_4]

.text:00001440 LDR R1, [SP,#8+var\_8]

.text:00001444 STR R1, [R0,#4]

.text:00001448 ADD SP, SP, #8

.text:0000144C BX LR

.text:0000144C ; End of function Number::setNum1(int)

函数是不占对象的内存的，只是占用了代码段

构造函数 参数有this指针，返回的r0 this指针

.text:00001440 ; \_DWORD Number::Number(Number \*\_\_hidden this)

.text:00001440 var\_4 = -4

.text:00001440

.text:00001440 SUB SP, SP, #4

.text:00001444 STR R0, [SP,#4+var\_4]

.text:00001448 LDR R0, [SP,#4+var\_4]

.text:0000144C MOV R1, #3

.text:00001450 STR R1, [R0]

.text:00001454 ADD SP, SP, #4

.text:00001458 BX LR

.text:00001458 ; End of function Number::Number(void)

R0是this指针，在BX LR之前，R0都是this指针的存在，没有改变

int \_\_fastcall Number::Number(int result)

{

\*(\_DWORD \*)result = 3;

return result;

}

看函数 getNum1

.text:0000147C ; \_DWORD Number::getNum1(Number \*\_\_hidden this)

.text:0000147C WEAK \_ZN6Number7getNum1Ev

.text:0000147C \_ZN6Number7getNum1Ev ; CODE XREF: main+4C↑p

.text:0000147C

.text:0000147C var\_4 = -4

.text:0000147C

.text:0000147C SUB SP, SP, #4

.text:00001480 STR R0, [SP,#4+var\_4]

.text:00001484 LDR R0, [SP,#4+var\_4]

.text:00001488 LDR R0, [R0,#4]

R0已经被改成地址this指针偏移4字节后的数据

.text:0000148C ADD SP, SP, #4

.text:00001490 BX LR 返回是DWORD

.text:00001490 ; End of function Number::getNum1(void)

全局变量的初始化 .init\_array 比main函数的调用还要早

\_\_INIT\_ARRAY\_\_ 🡪 \_GLOBAL\_\_sub\_I\_main.cpp 🡪 \_\_cxx\_global\_var\_init

静态对象在函数中初始化一次

if ( !(v3 & 1) && \_cxa\_guard\_acquire(&`guard variable for'main::s\_cnumber) )

{

Number::Number((Number \*)&main::s\_cnumber);

\_cxa\_guard\_release(&`guard variable for'main::s\_cnumber);

}

虚表在构造函数和析构函数中

.text:00001E68 sub\_1E68 ; CODE XREF:

.text:00001E68 ; \_\_unwind {

.text:00001E68 STMFD SP!, {R11,LR}

.text:00001E6C MOV R11, SP

.text:00001E70 SUB SP, SP, #0x10

.text:00001E74 LDR R1, =(aNumber - 0x1E80)

.text:00001E78 ADD R1, PC, R1 ; "Number"

.text:00001E7C LDR R2, =(\_ZTV6Number - 0x1E88)

.text:00001E80 ADD R2, PC, R2 ; `vtable for'Number

.text:00001E84 ADD R2, R2, #8

.text:00001E88 STR R0, [R11,#var\_4]

.text:00001E8C LDR R0, [R11,#var\_4]

.text:00001E90 STR R2, [R0] 把虚表地址存在对象0地址上

.text:00001E94 STR R0, [SP,#0x10+var\_8]

.text:00001E98 MOV R0, R1 ; format

.text:00001E9C BL printf

.text:00001EA0 LDR R1, [SP,#0x10+var\_8]

.text:00001EA4 STR R0, [SP,#0x10+var\_C]

.text:00001EA8 MOV R0, R1

.text:00001EAC MOV SP, R11

.text:00001EB0 LDMFD SP!, {R11,PC}

.text:00001EB0 ; End of function sub\_1E68

.data.rel.ro:000044B8 ; `vtable for'Number

.data.rel.ro:000044B8 \_ZTV6Number DCD 0 ; DATA XREF: sub\_1E68+18↑o

.data.rel.ro:000044B8 ; .text:off\_1EB8↑o ...

.data.rel.ro:000044B8 ; offset to this

.data.rel.ro:000044BC DCD \_ZTI6Number ; `typeinfo for'Number

.data.rel.ro:000044C0 DCD sub\_1EBC 虚函数

.data.rel.ro:000044C4 DCD sub\_1EDC 虚函数

虚表中有虚函数的地址

所以逆向的时候，找到虚表就可以找到构造函数 析构函数 虚函数

在IDA静态分析添加虚表的结构体，按T将偏移转成结构体，来标记有虚函数的类，用指针调用虚函数时，方便静态分析代码：

IDA窗口，键盘insert快捷键或者Edit/Add struct type添加一结构；键盘快捷键D或者Edit/Data添加函数

;加注释