刘坤的技术博客

iOS开发同学的arm64汇编入门

在定位某些crash问题的时候，有时候遇到一些问题很诡异。有时候挂在了系统库里面。这个时候定位crash问题往往是比较头疼的。那么这个时候学会一些汇编知识，利用汇编调试技巧进行调试可能会起到意想不到的效果。

学习汇编语言不只是帮助定位crash而已，学习汇编可以帮助你真正的理解计算机。毕竟CPU上跑的就是对应的指令集。

0x1 工具

我们面对的要么是源代码，要么是二进制。因此我们需要一些反汇编的工具来辅助我们进行汇编代码查看。推荐工具有： – Hopper Disassembler 收费应用，看汇编代码非常方便 – MachOView 开源工具，看Mach-o文件结构非常方便。

0x2 基本概念

从高级语言过渡到汇编语言，重要的是基本概念的转换。汇编里面要学习的三个重要概念，我认为是 寄存器、栈、指令。 arm64架构又分为2种执行状态： `AArch64 Application Level` 和 `AArch32 Application Level`, 本文只讲AArch64.

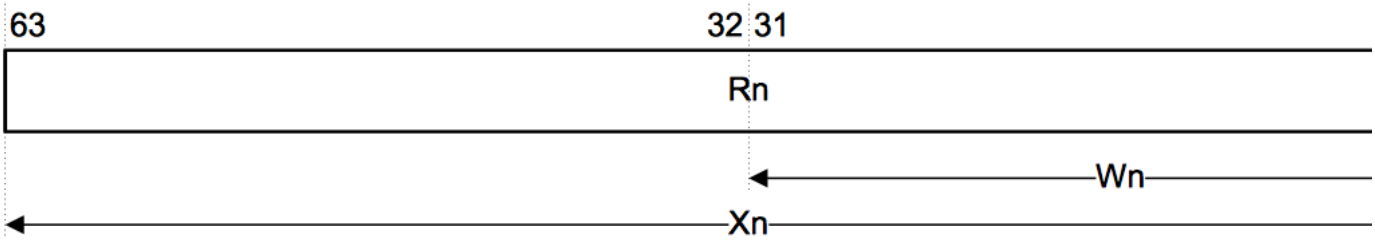
0x21 寄存器

如果你还不知道什么是寄存器，建议先Google一下。 这里不再详细说明，寄存器是CPU中的高速存储单元，要比内存中存取要快的多。

这里说明一下arm64有哪些寄存器：

- **R0 – R30**

`r0 - r30` 是31个通用整形寄存器。每个寄存器可以存取一个64位大小的数。 当使用 `x0 - x30`访问时，它就是一个64位的数。当使用 `w0 - w30`访问时，访问的是这些寄存器的低32位，如图：



其实通用寄存器有32个，第32个寄存器x31，在指令编码中，使用来做 `zero register`, 即`ZR`, `XZR/WZR`分别代表64/32位，`zero register`的作用就是0，写进去代表丢弃结果，拿出来是0.

其中 `r29` 又被叫做 `fp` (frame pointer). `r30` 又被叫做 `lr` (link register)。其用途会在下一节《栈》中讲到。

- **SP**

SP寄存器其实就是 x31，在指令编码中，使用 `SP/WSP`来进行对SP寄存器的访问。

- **PC**

PC寄存器中存的是当前执行的指令的地址。在arm64中，软件是不能改写PC寄存器的。

- **V0 – V31**

V0 - V31 是向量寄存器，也可以说是浮点型寄存器。它的特点是每个寄存器的大小是 128 位的。 分别可以用Bn Hn Sn Dn Qn的方式来访问不同的位数。如图：

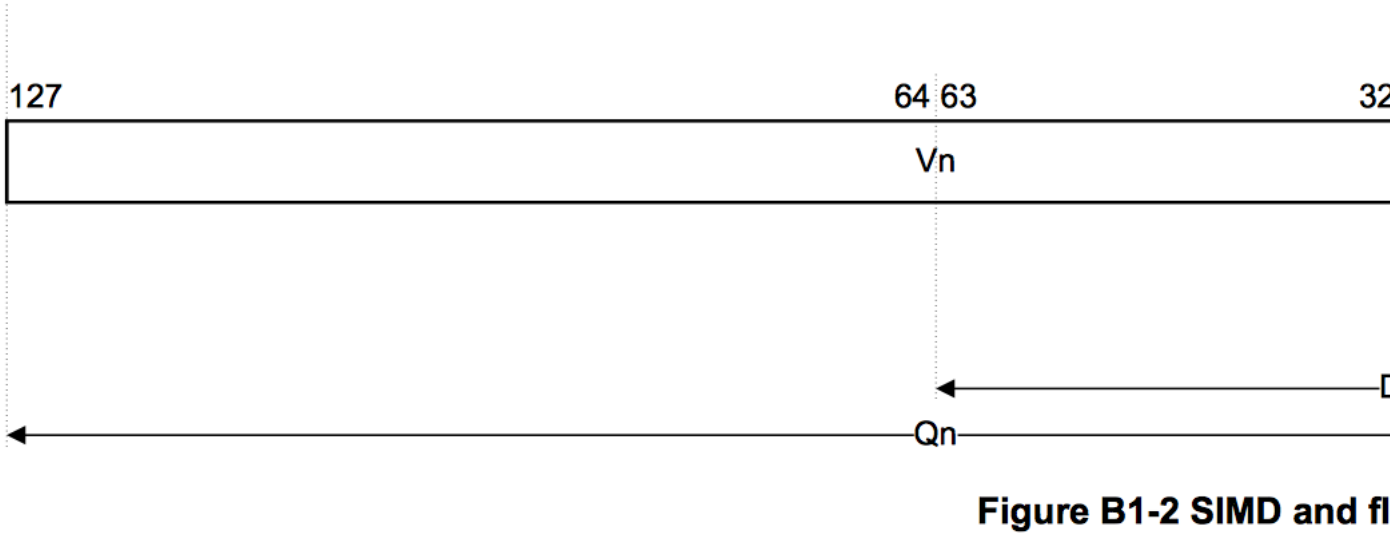


Figure B1-2 SIMD and fl

Bn Hn Sn Dn Qn可以这样理解记忆, 基于一个word是32位，也就是4Byte大小：

Bn: 一个Byte的大小
Hn: half word. 就是16位
Sn: single word. 32位
Dn: double word. 64位
Qn: quad word. 128位

- **SPRs**

SPRs是状态寄存器，用于存放程序运行中一些状态标识。不同于编程语言里面的if else.在汇编中就需要根据状态寄存器中的一些状态来控制分支的执行。状态寄存器又分为 The Current Program Status Register (CPSR) 和 The Saved Program Status Registers (SPSRs)。 一般都是使用CPSR， 当发生异常时， CPSR会存入SPSR。当异常恢复，再拷贝回CPSR。

还有一些系统寄存器，还有 FPSR FPCR是浮点型运算时的状态寄存器等。基本了解上面这些寄存器就可以了。

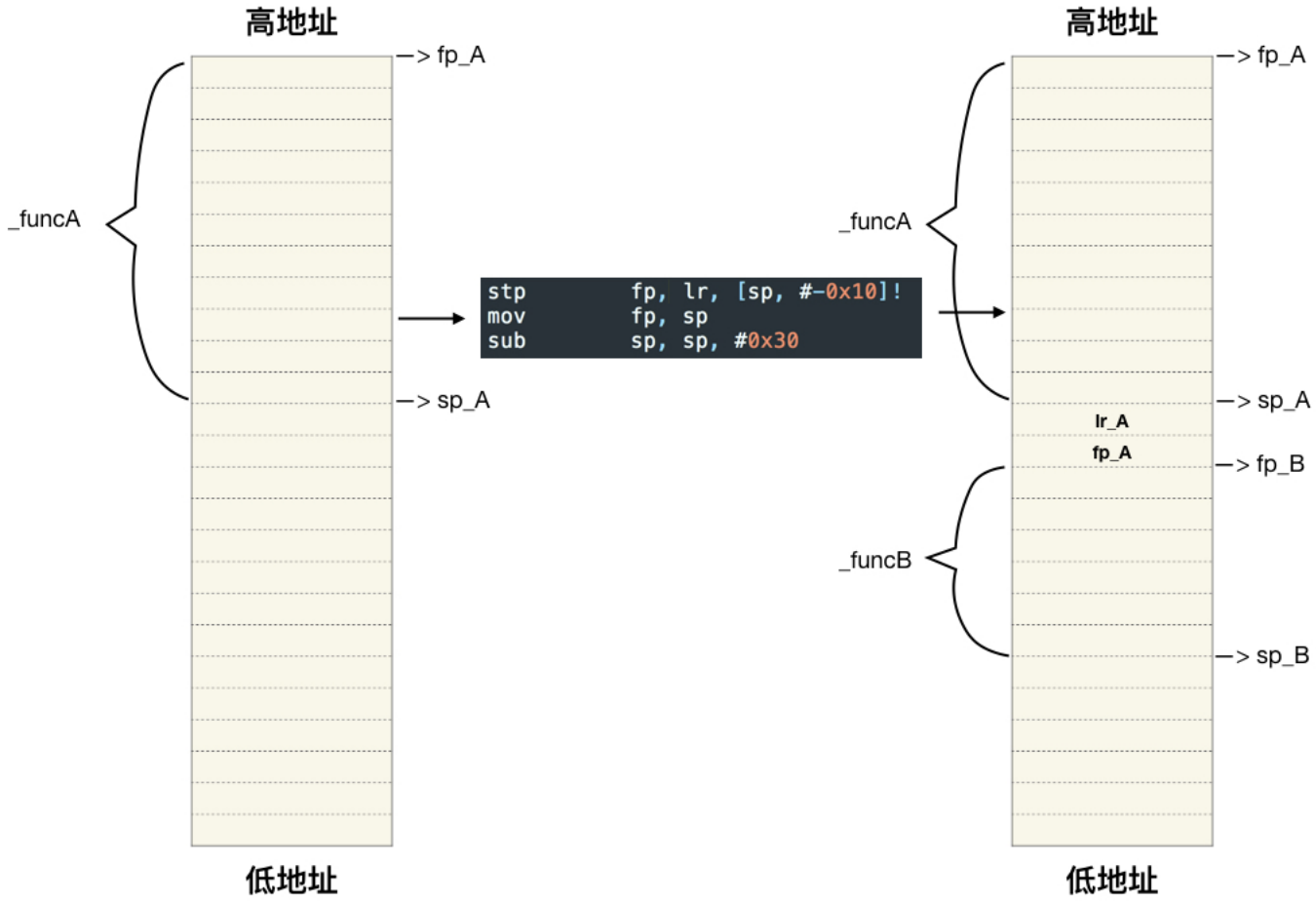
0x22 栈

栈就是指令执行时存放临时变量的内存空间。在学习汇编代码的执行过程中，了解栈的结构非常重要。

先列出一些栈的特性：

- 栈是从高地址到低地址的， 栈低是高地址，栈顶是低地址。
- fp指向当前frame的栈底，也就是高地址。
- sp指向栈顶，也就是地地址。

下面的图简单的描述了从方法A调用方法B时 栈是如何划分的：



其中3行汇编代码就是方法B的前三行汇编指令。它们做的事情就是图中描述的事情 (x29就是fp, x30就是lr)：

- 将fp, lr保存到 sp - 0x10的地方. 也就是图中 --> fp_B的位置。然后将sp设置为 sp-0x10
- 将 fp 设置为当前 sp。也就是 --> fp_B的位置。 这一步就设置了_funcB的 fp了
- 将 sp 设置为 sp - 0x30。 也就是将sp指向了图中 --> sp_B 的位置

  注： lr 是link register中的值，它存的是方法_funcA的执行的最后一行指令的下一行。它的作用也很好理解：当_funcB执行完了之后要返回_funcA继续执行，但是计算机要如何知道返回到哪执行呢？ 就是靠lr记录了返回的地址，方法才能得以正常返回。

说道这里，那么当 _funcB执行完毕后，是如何把栈恢复到_funcA的过程的呢？ 我们直接分析 _funcB的最后3条指令：

```
1 mov        sp, fp;              //  sp 设置为fp, 就是图中 -->fp_B 的位置
2 ldp           fp, lr, [sp], #0x10; //  从sp指向的地址中读取 2个64位，分别存入fp,lr。 然后将sp += 0x10
3 // 这一步执行完之后，fp就执行了图中 -->fp_A. lr恢复成 _funcA的返回地址。 sp指向了 -->sp_A.
4 // 这个时候状态已经完全恢复到了 _funcA 的环境
5 ret;    // 返回指令，这一步直接执行lr的指令。
```

上面描述了方法如何调用的。我们知道在编程语言里面方法都有入参，有返回值的。在汇编里面如何体现呢？

- 一般来说 arm64上 x0 – x7 分别会存放方法的前 8 个参数
- 如果参数个数超过了8个，多余的参数会存在栈上，新方法会通过栈来读取。
- 方法的返回值一般都在 x0 上。
- 如果方法返回值是一个较大的数据结构时，结果会存在 x8 执行的地址上。

0x23 指令

在上一级的内容中我们已经看到了一些指令。 汇编指令除了数量较多，其基本原理都是比较简单的，单拎出来一条指令就是很simple的操作。 比如mov就是一个赋值。ldr就是一个取值。

那汇编指令大概可以分为哪几种呢？我认为了解以下几种基本指令就可以正常阅读汇编代码了。

0x231 运算

- 算术运算

算术运算就是像 ADD SUB MUL ... 等加减乘除运算，也是很好理解的指令
如：

```
1 add x0, x1, x2; // 把 x1 + x2 = x0 这样一个操作。
2 sub sp, sp, 0x30; // 把 sp - 30 存入sp.
```

```
3 cmp x11, #4;  // 相当于 subs xzr, x11, #4.
4               // 如果 x11 - 4 == 0, 那么状态寄存器NZCV.Z = 1
5               // 如果 x11 - 4 < 0, 那么 NZCV.N = 1
```

NZCV是状态寄存器中存的几个状态值，分别代表运算过程中产生的状态，其中：
* N, negative condition flag，一般代表运算结果是负数
* Z, zero condition flag, 运算结果为0
* C, carry condition flag, 无符号运算有溢出时，C=1。
* V, oVerflow condition flag 有符号运算有溢出时，V=1。

- 逻辑运算指令

有 LSL(逻辑左移) LSR(逻辑右移) ASR(算术右移) ROR(循环右移)。
有 AND(与) ORR(或) EOR(异或)

逻辑位移运算通常也可以与算术运算一起用，如：

```
1  add  x14, x4, x27, lsl #1; // 意思是把  (x27 << 1) + x4 = x14;
```

- 拓展位数运算

有 zero extend(高位补0) 和 sign extend(高位填充和符号位一致，一般有符号数用这个)。 一般用来补齐位数。常和算术运算配合一起.
如：

```
1 add        w20, w30, w20, uxth  // 取 w20的低16位，无符号补齐到32位后再进行  w30 + w20的运算。
```

- Mov

0x232 寻址

既然是和内存相关的，那就是两种，一种存，一种取。一般来说
L打头的基本都是取值指令，如 LDR LDP;
S打头的基本都是存值指令，如 STR STP;

例：

```
1 ldr x0, [x1]; // 从`x1`指向的地址里面取出一个 64 位大小的数存入 `x0`
2 ldp x1, x2, [x10, #0x10]; // 从 x10 + 0x10 指向的地址里面取出 2个 64位的数，分别存入x1, x2
3 str x5, [sp, #24]; // 把x5的值（64位数值）存到 sp+24 指向的内存地址上
4 stp x29, x30, [sp, #-16]!; // 把 x29, x30的值存到 sp-16的地址上，并且把 sp-=16.
5 ldp x29, x30, [sp], #16;  // 从sp地址取出 16 byte数据，分别存入x29, x30. 然后 sp+=16;
```

其中寻址的格式由分为下面这3种类型：

```
1 [x10, #0x10]      // signed offset。 意思是从 x10 + 0x10的地址取值
2 [sp, #-16]!       // pre-index。  意思是从 sp-16地址取值，取值完后在把 sp-16  writeback 回 sp
3 [sp], #16         // post-index。 意思是从 sp 地址取值，取值完后在把 sp+16 writeback 回 sp
```

0x233 跳转

跳转氛围有返回跳转BL和无返回跳转B。 有返回的意思就是会存lr,因此 BL的L也可以理解为LR的意思。

1.存了LR也就意味着可以返回到本方法继续执行。一般用于不同方法直接的调用
2.B相关的跳转没有LR，一般是本方法内的跳转，如while循环，if else等。

跳转相关的指令还会有种逻辑运算，就是condition code。配合状态寄存器中的状态标示，就是代码分支if else实现的关键。
condition code有以下这些，表格中还标注除了分别是比NZCV的哪个值：

Encoding	Name (& alias)	Meaning (integer)	Meaning (floating point)	Flags
0000	EQ	Equal	Equal	Z==1
0001	NE	Not equal	Not equal, or unordered	Z==0
0010	HS (CS)	Unsigned higher or same (Carry set)	Greater than, equal, or unordered	C==1
0011	LO (CC)	Unsigned lower (Carry clear)	Less than	C==0
0100	MI	Minus (negative)	Less than	N==1
0101	PL	Plus (positive or zero)	Greater than, equal, or unordered	N==0
0110	VS	Overflow set	Unordered	V==1
0111	VC	Overflow clear	Ordered	V==0
1000	HI	Unsigned higher	Greater than, or unordered	C==1 && Z==0
1001	LS	Unsigned lower or same	Less than or equal	!(C==1 && Z==0)
1010	GE	Signed greater than or equal	Greater than or equal	N==V
1011	LT	Signed less than	Less than or unordered	N!=V
1100	GT	Signed greater than	Greater than	Z==0 && N==V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z==0 && N==V)
1110	AL	Always	Always	Any
1111	NV[1]			

如：

```
cmp x2, #0;         // x2 - 0 = 0。  状态寄存器标识zero: PSTATE.NZCV.Z = 1
b.ne  0x1000d48f0;  // ne就是个condition code, 这句的意思是，当判断状态寄存器 NZCV.Z != 1才跳转，因此这句不会跳转

0x1000d4ab0 bl testFuncA;  // 跳转方法，这个时候 lr 设置为 0x1000d4ab4
0x1000d4ab4 orr x8, xzr, #0x1f00000000 // testFuncA执行完之后跳回lr就周到了这一行
```

0x4 小结

本文简单介绍了一些arm64的汇编知识，arm64汇编的学习对于理解iOS代码的执行，计算机的运行都有着不少的好处。我们在日常中利用汇编知识可以定位一些疑难杂症的crash问题。可以从汇编原理出手开一个个脑洞，玩一些黑科技。比如包瘦身，静态扫描等。

汇编指令的执行是简单确定的，不会像我们调试其他代码一眼，有些诡异问题，而汇编每条指令的结果都是确定的，从这一角度来定位问题往往可以定位到根本原因。

在汇编指令执行的世界，你可以对代码执行有更深刻的理解，原来一行代码会被分解成这么多的指令！因此，如果你在看完本文后对于学习汇编有了兴趣，但是有很多细节还不太懂，建议你自己用hopper反编译一些代码，自己尝试一行一行理解每一个指令的意义，基本看透几个方法就可以融汇贯通了。

0x5 参考

- ARMv8-A Architecture – ARM

Share

Comments

28 • 回复 • 分享 ›

Vsir • 1年前
文中代码注释：“其中！代表writeback,就是改变sp的值”这句话是错的，“!”表示pre-index。
比如：
ldp x24, x23, [sp], #0x10
没有“!”，但也会writeback。
• 回复 • 分享 ›

张昊 ➚ Vsir • 1个月前
An <address> can take multiple forms:

An address expression:
<expression>
A pre-indexed address – where the address generated is used immediately:
[Rn, <expression>]{!}
[Rn, {-}Rm]{!}
[Rn, {-}Rm <shift> count]{!}
A post-indexed address – where the address generated later replaces the base register:
[Rn], <expression>
[Rn], {-}Rm
[Rn], {-}Rm <shift> count
Where is any of LSL, LSR, ASR, ROR or RRX as described earlier.

Pre-indexed writeback denoted by {!} causes the final address generated to be written back into Rn.
看文档，感觉楼主之前的翻译没问题
• 回复 • 分享 ›

刘坤 管理员 ➚ Vsir • 1年前
多谢指正！
7 • 回复 • 分享 ›

Shimin Pan • 2年前
好久没更新文章了，一更新全是干货！
• 回复 • 分享 ›