7.7 AArch64汇编指令集

arm64-v8a对应着两套架构的指令集：分别是AArch32（简称A32）的ARM、Thumb、Thumb2指令集，与AArch64（简称A64）的64位指令集。

7.7.1 AArch64指令编码

AArch64指令集是根据指令不同的位域分布，将指令集分成了几大编码组（Encoding Group），每个编码组下，再细分指令所属的类别。

28	27	26	25	24	Encoding Group
0	0	-	-	-	不可预料
1	0	0	-	-	数据处理（立即数）指令
1	0	1	-	-	分支、异常生成与系统指令
-	1	-	0	-	分支、异常生成与系统指令
-	1	0	1	-	数据处理（寄存器）指令
0	1	1	1	-	数据处理 SIMD与浮点指令
1	1	1	1	-	数据处理 SIMD与浮点指令

可以看到，只有bit[28:24]的几个位才会影响到指令的编码组，最后的0b0111与0b1111都同属于SIMD与浮点指令编码组。

AArch64虽然属于64位的指令系统，但它仍然是32位的指令长度，指令在使用时，与AArch32指令最大的不同，体现在寄存器的使用上。比如AArch32中，将R1寄存器的值传入R0寄存器，对应的指令为“MOV R0, R1”，在AArch64上，对应的有32位的“MOV W0, W1”，与64位的“MOV X0, X1”。具体是使用32位还是64位的寄存器，主要看指令最高位bit[31]的`sf`域的值是否为1，如果为1就使用64位，反之，使用32位。通常在具体指令的伪代码描述部分会有如下一行，用来说明寄存器使用位数的判断方法：

```
integer datasize = if sf == '1' then 64 else 32;
```

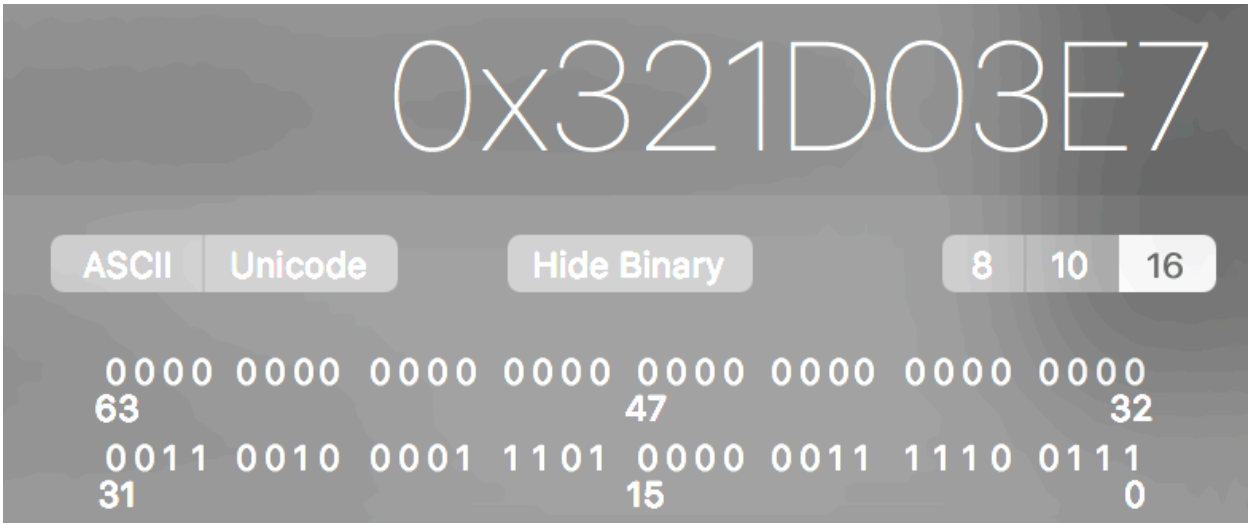
将app2.c程序编译成AArch64的汇编代码，可以执行如下命令：

```
$ export NDK=/usr/local/opt/android-sdk/ndk-bundle
$ export SYSROOT=$NDK/platforms/android-21/arch-arm64
$ export CC="$NDK/toolchains/llvm/prebuilt/darwin-x86_64/bin/clang --
sysroot=$SYSROOT -target aarch64-linux-androideabi
-gcc-toolchain $NDK/toolchains/aarch64-linux-android-4.9/prebuilt/darwin-
x86_64"
$ $CC app2.c -fPIE -pie  -march=armv8-a -mtune=cortex-a53 -O3 -o app2
```

7.7.2 AArch64指令格式解析

接下来，以十六进制0x321D03E7为例，通过分析它的指令编码，找出它对应的AArch64汇编指令。

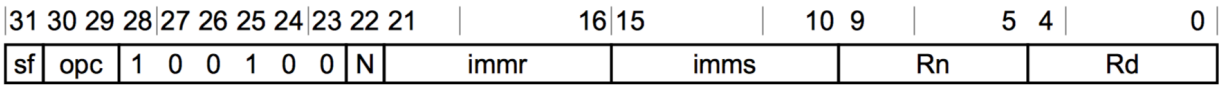
打开计算器，查看0x321D03E7的二进制编码，如图所示：



bits[28:26]为0b100，对应的是数据处理（立即数）指令编码组。数据处理（立即数）指令编码组中所有的指令类别如下表所示：

28	27	26	25	24	23	指令类别
1	0	0	0	0	-	相对PC寄存器，取址指令
1	0	0	0	1	-	加/减（立即数）指令
1	0	0	1	0	0	逻辑（立即数）指令
1	0	0	1	0	1	宽移动（立即数）指令
1	0	0	1	1	0	位域（Bitfield）指令
1	0	0	1	1	1	提取（Extract）指令

bits[28:23]的值为0b100100，对应的是逻辑（立即数）指令。逻辑（立即数）指令位域表示的格式如图所示：



所有可选的指令如表所示：

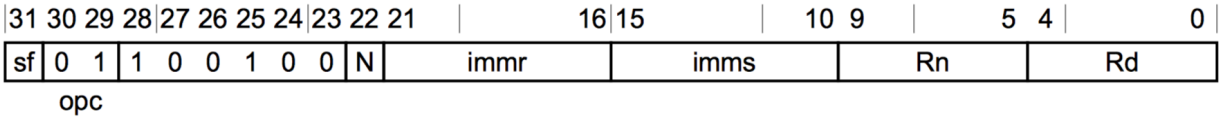
sf	opc	N	指令	寄存器位数
0	00	0	AND（立即数）	32位
0	01	0	ORR（立即数）	32位
0	10	0	EOR（立即数）	32位
0	11	0	ANDS（立即数）	32位
1	00	-	AND（立即数）	64位
1	01	-	AND（立即数）	64位
1	10	-	EOR（立即数）	64位
1	11	-	ANDS（立即数）	64位

这里的bit[31]的`sf`域为0，表示是使用32位寄存器的指令，bits[30:29]的`opc`域值为01，表示是这是ORR（立即数）指令。

查看ORR（立即数）指令格式，使用32位寄存器的指令格式的声明如下所示：

```
ORR <Wd|WSP>, <Wn>, #<imm>
```

指令的位域分布如图所示：



查看ORR指令的伪代码描述，如下所示：

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
        when '00' op = LogicalOp_AND; setflags = FALSE;
        when '01' op = LogicalOp_ORR; setflags = FALSE;
        when '10' op = LogicalOp_EOR; setflags = FALSE;
        when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0'
        then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

从上面的伪代码中可以计算出，d的值为`Rd`域对应的整型数值，它位于bits[4:0]，值为0b00111，即数值为7，可知指令格式的`Wd`值为W7。n的值为`Rn`域对应的整型数据，它位于bits[9:5]，值为0b11111，这种全1的情况下，表示的是0寄存器，即`Wn`对应的是WZR寄存器。接下来只要知道指令格式中`imm`的计算方法，指令的完整格式就清楚了。

`imm`通过`DecodeBitMasks()`方法计算得到，它的伪代码如下：

```
// DecodeBitMasks()
// ================

// Decode AArch64 bitfield and logical immediate masks which use a similar
encoding structure

(bits(M), bits(M)) DecodeBitMasks (bit immN, bits(6) imms, bits(6) immr,
boolean immediate)
        bits(M) tmask, wmask;
        bits(6) levels;

        // Compute log2 of element size
        // 2^len must be in range [2, M]
        len = HighestSetBit(immN:NOT(imms));
        if len < 1 then ReservedValue();
        assert M >= (1 << len);

        // Determine S, R and S - R parameters
        levels = ZeroExtend(Ones(len), 6);

        // For logical immediates an all-ones value of S is reserved
        // since it would generate a useless all-ones result (many times)
        if immediate && (imms AND levels) == levels
                then ReservedValue();

        S = UInt(imms AND levels);
        R = UInt(immr AND levels);
        diff = S - R; // 6-bit subtract with borrow

        esize = 1 << len;
        d = UInt(diff<len-1:0>);
        welem = ZeroExtend(Ones(S + 1), esize);
        telem = ZeroExtend(Ones(d + 1), esize);
        wmask = Replicate(ROR(welem, R));
        tmask = Replicate(telem);
        return (wmask, tmask);
```

可以看到，整个`imm`的计算过程比较复杂了，这里就不展开推算了。直接给出计算后的结果为8。

综合上面的分析，总结可得，十六进制0x321D03E7对应的AArch64汇编指令为“ORR W7, WZR, #8”。同样可以使用`rasm2`进行验证结果（注意是小端字节序）：

```
$ rasm2 -a arm -b64 -d E7031D32
orr w7, wzr, 8
```

如果使用IDA Pro分析这段指令，你会发现，它生成的十六进制0x321D03E7的指令为“MOV W7, #8”，关于这点，可以在ARM指令手册中看到如下描述：

```
Bitwise inclusive OR (immediate): Rd = Rn OR imm
This instruction is used by the alias MOV (bitmask immediate).
```

将`Rn`与`imm`异或后的结果传给`Rd`，本质上它的行为与“MOV Rd, Rn, #imm”是一样的。只是IDA Pro觉得使用MOV指令代替ORR指令，汇编代码更易读罢了。