最全ARM体系结构知识：汇编、架构、异常级别和安全状态

技术让梦想更伟大 2021-10-19 22:12

来源：智能软件研究中心 | 直接来源：华为开发者社区

作者：罗宇哲

01
ARM汇编指令

操作系统中硬件相关的部分集中体现在汇编指令和对寄存器的操作中，因此我们对ARM体系结构的介绍也围绕**ARMv8-A的汇编指令和寄存器**来展开。

处理器架构是处理器厂商为同一个系列的处理器规定的一个规范。ARM架构是一种精简指令集（RISC）架构，具有以下RISC架构特点：

- 较大的通用寄存器堆。
- load/store体系结构，其中数据处理操作仅对寄存器内容进行操作，而不是直接对内存内容。
- 简单寻址模式，所有load/store地址由寄存器内容和指令确定。该体系结构定义了处理单元与内存（包括缓存）的交互，并包括内存地址翻译系统。它还描述了多个处理单元如何相互作用。面积小、性能强和非常低的功耗是ARM体系结构的关键特性。本小节主要以ARMv8-A架构为例来介绍ARM体系结构的基本特性。ARMv8-A体系结构的一个重要特性是向后兼容，可以支持诸多标准和应用场景下的最优设计。ARMv8-A架构支持64bit的执行模式(AArch64)和32bit的执行模式(AArch32)，这一模式兼容之前的ARM架构。两种执行状态都支持SIMD和浮点指令。

一、AMRv8架构概要

ARM体系结构自推出以来已经有了显著的发展，并且ARM还在继续开发它。到目前为止，已经有八个主要版本，由版本号1到8表示。其中前三个版本现在已经过时了。

通用名称AArch64和AArch32描述了64位和32位执行状态。AArch64是64位执行状态，意味着地址保存在64位寄存器中，并且基本指令集可以使用64位寄存器进行处理。AArch64支持A64指令集。AArch32是32位执行状态，这意味着地址保存在32位寄存器中，并且基本指令集使用32位寄存器进行处理。AArch32支持T32和A32指令集。

ARM支持三种架构配置：

- A系列，面向应用场景的架构(Application Profile)。该系列支持基于内存管理单元（MMU）的虚拟内存系统体系结构（VMSA）。它支持A64、A32和T32指令集。
- R系列，面向实时场景的架构配置。该系列支持基于内存保护单元（MPU）的受保护内存系统体系结构（PMSA）。它支持A32和T32指令集。
- M系列，面向微处理器的架构。该系列实现了一个为低延迟中断处理而设计的程序员模型（programmers’model），该模型具有寄存器硬件堆栈和对中断处理程序的高级语言支持。它支持T32指令集的变种。

  （注：内存保护单元（MPU）是ARM中配备的有效保护系统资源的一种硬件，提供了内存区域保护功能。）

二、ARMv8-A指令集

在ARMv8-A中，可能的指令集取决于执行状态：

1. AArch64：AArch64 state只支持A64指令集。这是一个固定长度的指令集，使用32位指令编码。
2. Arch32：AArch32 state支持以下指令集：

- A32：这是一个固定长度的指令集，使用32位指令编码。它是与ARMv7 ARM指令集兼容。
- T32：这是一个可变长度指令集，它同时使用16位和32位指令编码。它与ARMv7 Thumb®指令集兼容。

ARM指令的基本格式如下[2]：

<Opcode>{<Cond>}<S><Rd>，<Rn>{，<Opcode2>}

其中各个部分的含义为：

- Opcode：操作码，也就是助记符，说明指令需要执行的操作类型；
- Cond：指令执行条件码；
- S：条件码设置项，决定本次指令执行是否影响PSTATE寄存器相应状态位值；

- Rd/Xt：目标寄存器，A32指令可以选择R0-R14，T32指令大部分只能选择RO-R7，A64指令可以选择X0-X30；
- Rn/Xn：第一个操作数的寄存器，和Rd一样，不同指令有不同要求；
- Opcode2：第二个操作数，可以是立即数，寄存器Rm和寄存器移位方式（Rm，#shit）；

ARMv8-A指令集的条件码如下图所示：

cond	Mnemonic	Meaning (integer)	Meaning (floating-point)[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal or unordered	Z == 0
0010	CS or HS	Carry set	Greater than, equal, or unordered	C == 1
0011	CC or LO	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	Unordered	V == 1
0111	VC	No overflow	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[b]	Always	Always	Any

下面以A64指令集为例简要介绍ARMv8-A的指令体系。A64指令集中的指令主要分为控制指令、访存指令和计算指令。控制指令主要包括有条件分支指令、无条件分支指令、异常产生和返回指令、系统寄存器指令、系统指令、提示指令、同步指令和清除独占访问标志指令。访存指令主要有Load指令和Store指令，这两种指令有许多变种。计算指令包含算数指令、逻辑指令、MOVE指令、移位指令、位扩展指令和SIMD指令等等。以下列出了一些常用的控制指令的名称与用途。

1. 控制指令：

- 条件分支指令：

Mnemonic	Instruction	Branch offset range from the PC
B.cond	Branch conditionally	±1MB
CBNZ	Compare and branch if nonzero	±1MB
CBZ	Compare and branch if zero	±1MB
TBNZ	Test bit and branch if nonzero	±32KB
TBZ	Test bit and branch if zero	±32KB

- 无条件分支指令：

Mnemonic	Instruction	Immediate branch offset range from the PC
B	Branch unconditionally	±128MB
BL	Branch with link	±128MB

- 使用寄存器的无条件分支指令：

Mnemonic	Instruction
BLR	Branch with link to register
BR	Branch to register
RET	Return from subroutine

- 异常产生指令：

Mnemonic	Instruction
BRK	Software breakpoint instruction
HLT	Halting software breakpoint instruction
HVC	Generate exception targeting Exception level 2
SMC	Generate exception targeting Exception level 3
SVC	Generate exception targeting Exception level 1

- 异常返回指令：

Mnemonic	Instruction
ERET	Exception return using current ELR and SPSR

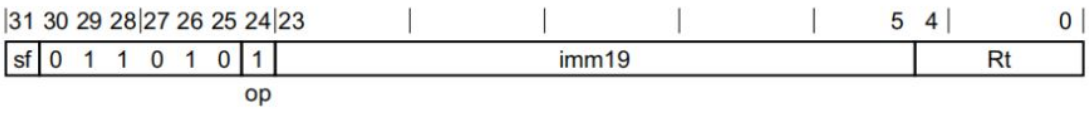
- 系统寄存器指令：

Mnemonic	Instruction
MRS	Move system register to general-purpose register
MSR	• Move general-purpose register to system register • Move immediate to PE state field

- 同步指令和独占状态清除指令：

Mnemonic	Instruction
CLREX	Clear exclusive monitor
DSB	Data synchronization barrier
DMB	Data memory barrier
ISB	Instruction synchronization barrier

例如：



32-bit variant (sf = 0)

```
CBNZ <Wt>, <label>
```

64-bit variant (sf = 1)

```
CBNZ <Xt>, <label>

integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
boolean iszero = (op == '0');
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<label>	Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

2. 访存指令：

ARMv8访存指令支持以下寻址模式：

- 基址加上无符号立即数的寻址和基址加上有符号立即数的寻址；
- 基址加上寄存器偏移值；
- 基址加上扩展的寄存器偏移；
- pre-index模式；
- post-index模式；
- PC相对寻址模式。

具体情形见下表：

Addressing Mode	Offset		
	Immediate	Register	Extended Register
Base register only (no offset)	[base{, #0}]	-	-
Base plus offset	[base{, #imm}]	[base, Xm{, LSL #imm}]	[base, Wm, (S|U)XTW {#imm}]
Pre-indexed	[base, #imm]!	-	-
Post-indexed	[base], #imm	[base], Xm[a]	-
Literal (PC-relative)	label	-	-

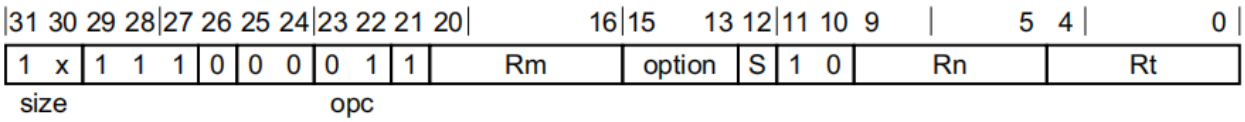
其中对于A64指令集来说，64bit的基址来自通用寄存器X0-X30或来自栈指针SP，立即数或寄存器偏移值则是可选的，对寻址方式的解释如下：

- 寄存器偏移寻址是指来自64bit基址寄存器的地址加上一个偏移值；
- Pre-indexed模式是指寻址地址是64bit基址加上一个偏移值，这个计算和将会写入基址寄存器；
- Post-indexed模式是指寻址地址是64bit的基址，但之后基址和偏移值的和将会写入基址寄存器；由此可见pre-indexed和post-indexed的区别在于使用的地址是先加上偏移值再使用还是先使用再加上偏移值；
- PC相对寻址是指寻址地址是这条指令64bit的PC值加上一个19bit的有符号字偏移，这个地址在当前指令的PC值的 ±1MB范围内并且是4byte对齐的。使用PC相对寻址所load的数据大小至少为32bit并且只能用来预取指令，且PC值不能被其他寻址方式使用。
- 一个立即数偏移可以为有符号的，也可以为无符号的，可以为scaled也可以为unscaled。当一个立即数偏移是scaled的时候，它被编码为传输数据大小的整数倍。虽然汇编程序总是使用byte对齐的偏移，但汇编器或反汇编器会做必要的转换工作，因此可用的byte偏移值取决于load/store指令类型和数据传输的大小。

下面列出了一些load/store指令：

Mnemonic	Instruction	See
LDR	• Load register (register offset) • Load register (immediate offset) • Load register (PC-relative literal)	• *LDR (register)* on page C5-521 • *LDR (immediate)* on page C5-517 • *LDR (literal)* on page C5-520
LDRB	• Load byte (register offset) • Load byte (immediate offset)	• *LDRB (register)* on page C5-527 • *LDRB (immediate)* on page C5-524
LDRSB	• Load signed byte (register offset) • Load signed byte (immediate offset)	• *LDRSB (register)* on page C5-539 • *LDRSB (immediate)* on page C5-536
LDRH	• Load halfword (register offset) • Load halfword (immediate offset)	• *LDRH (register)* on page C5-533 • *LDRH (immediate)* on page C5-530
LDRSH	• Load signed halfword (register offset) • Load signed halfword (immediate offset)	• *LDRSH (register)* on page C5-545 • *LDRSH (immediate)* on page C5-542
LDRSW	• Load signed word (register offset) • Load signed word (immediate offset) • Load signed word (PC-relative literal)	• *LDRSW (register)* on page C5-552 • *LDRSW (immediate)* on page C5-548 • *LDRSW (literal)* on page C5-551
STR	• Store register (register offset) • Store register (immediate offset)	• *STR (register)* on page C5-697 • *STR (immediate)* on page C5-694
STRB	• Store byte (register offset) • Store byte (immediate offset)	• *STRB (register)* on page C5-703 • *STRB (immediate)* on page C5-700
STRH	• Store halfword (register offset) • Store halfword (immediate offset)	• *STRH (register)* on page C5-709 • *STRH (immediate)* on page C5-706

例如Load寄存器指令：



32-bit variant (size = 10)

```
LDR <Wt>, [<Xn|SP>, <R><m>{, <extend> {<amount>}}]
```

64-bit variant (size = 11)

```
LDR <Xt>, [<Xn|SP>, <R><m>{, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<R>	Is the index width specifier, **RESERVED** when option = 00x **W** when option = x10 **X** when option = x11 **RESERVED** when option = 10x
<m>	Is the number [0-30] of the general-purpose index register or the name ZR (31), encoded in the "Rm" field.
<extend>	Is the index extend/shift specifier, defaulting to LSL and **RESERVED** when option = 00x **UXTW** when option = 010 **LSL** when option = 011 **RESERVED** when option = 10x **SXTW** when option = 110 **SXTX** when option = 111
<amount>	For the 32-bit variant: is the index shift amount, optional and defaulting to #0 when <extend> is not LSL, **#0** when S = 0 **#2** when S = 1
<amount>	For the 64-bit variant: is the index shift amount, optional and defaulting to #0 when <extend> is not LSL, **#0** when S = 0

上表中指令的寻址方式有：

- 基址加上12bit无符号scaled立即数偏移寻址；
- 基址加上9bit有符号unscaled立即数偏移寻址；
- 基址加上64bit寄存器偏移，可选为scaled；
- 基址加上32bit可拓展寄存器偏移，可选为scaled；
- 有unscaled9bit有符号立即数偏移的pre-indexed模式；
- 有unscaled9bit有符号立即数偏移的post-indexed模式；
- Load至少32bit数据的PC相对寻址模式。

如果被load或store的指令的寻址模式会修改基址寄存器的内容，且被load/store寄存器恰好的是基址所在的寄存器，那么硬件的行为可能不确定。

3.计算指令：

在操作系统汇编语言中使用的计算指令主要是一些简单的算数计算指令，用于对寄存器的move操作和对地址的计算操作，一般计算指令既可以使用立即数作为操作数，也可以使用寄存器中的数作为操作数。下面

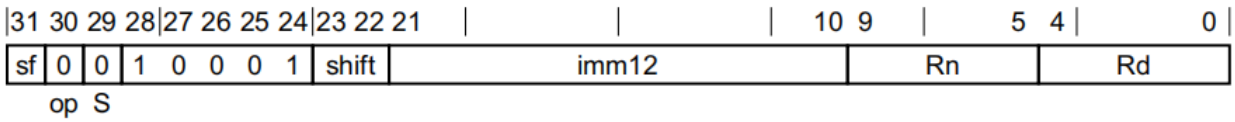
简单列举了一些算数指令：

使用立即数的简单算数指令：

Mnemonic	Instruction	See
ADD	Add	*ADD (immediate)* on page C5-396
ADDS	Add and set flags	*ADDS (immediate)* on page C5-402
SUB	Subtract	*SUB (immediate)* on page C5-738
SUBS	Subtract and set flags	*SUBS (immediate)* on page C5-744
CMP	Compare	*CMP (immediate)* on page C5-451.
CMN	Compare negative	*CMN (immediate)* on page C5-447

例如：

Add (immediate): Rd = Rn + shift(imm)



32-bit variant (sf = 0)

```
ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit variant (sf = 1)

```
ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

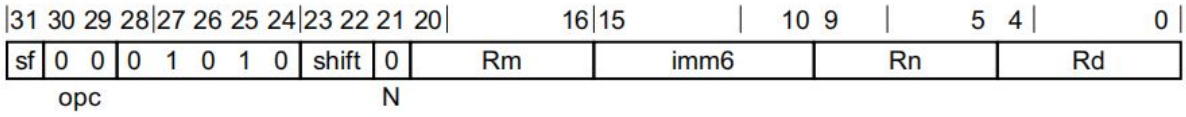
```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
    when '1x' ReservedValue();
```

使用寄存器的逻辑操作指令：

Mnemonic	Instruction	See
AND	Bitwise AND	*AND (shifted register)* on page C5-410
ANDS	Bitwise AND and set flags	*ANDS (shifted register)* on page C5-414
BIC	Bitwise bit clear	*BIC (shifted register)* on page C5-426
BICS	Bitwise bit clear and set flags	*BICS (shifted register)* on page C5-428
EON	Bitwise exclusive OR NOT	*EON (shifted register)* on page C5-474
EOR	Bitwise exclusive OR	*EOR (shifted register)* on page C5-477
ORR	Bitwise inclusive OR	*ORR (shifted register)* on page C5-627
MVN	Bitwise NOT	*MVN* on page C5-617
ORN	Bitwise inclusive OR NOT	*ORN (shifted register)* on page C5-623
TST	Test bits	*TST (shifted register)* on page C5-758

例如：

Bitwise AND (shifted register): Rd = Rn AND shift(Rm, amount)



32-bit variant (sf = 0)

```
AND <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant (sf = 1)

```
AND <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
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;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

其中：

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and

LSL when shift = 00

LSR when shift = 01

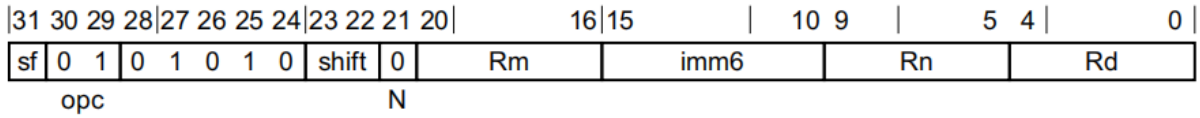
ASR when shift = 10

ROR when shift = 11

寄存器移位指令：

Mnemonic	Instruction	See
MOV	• Move register • Move register to SP or move SP to register	• *MOV (register)* on page C5-604 • *MOV (to/from SP)* on page C5-600

例如：



32-bit variant (sf = 0)

MOV <Wd>, <Wm>

is equivalent to

ORR <Wd>, WZR, <Wm>

and is the preferred disassembly when Rn == '11111' && IsZero(shift:imm6).

64-bit variant (sf = 1)

MOV <Xd>, <Xm>

is equivalent to

ORR <Xd>, XZR, <Xm>

and is the preferred disassembly when Rn == '11111' && IsZero(shift:imm6).

02
ARM架构寄存器

在处理器中，寄存器用于保存需要被快速访问的数据，在操作系统中需要特别注意的寄存器主要有栈指针寄存器（SP）、连接寄存器（LR）、程序计数器（PC）以及当前程序状态寄存器（CPSR）和保存程序状态寄存器（SPSR）。本小节主要以ARMv8-A为例介绍ARM架构的寄存器的基本情况。详情可参见文献[3],D1.6小节。

在这一小节中，我们主要介绍ARMv8架构中AArch64执行状态下的寄存器使用情况。ARM架构中的寄存器主要有两类，一类用于提供系统控制与状态报告；另一类用于指令运行和异常处理。我们主要讨论第二类。

通用寄存器主要用于基本指令集中的指令运行，通用寄存器共有31个，编号为R0-R31。这些通用寄存器可以被当成31个64bit的寄存器，编号为X0-X30；或者被作为31个32bit的寄存器，编号为W0-W30。

在AArch64执行状态下，除了通用寄存器外，每一个异常级别都会有一个栈指针寄存器（StackPointer Register， SP），栈指针寄存器为SPEL0和SPEL1。异常级别用于区分指令的执行权限，我们将在本章的第四期介绍。如果处理器实现中包含EL2，那么还有SPEL2。如果处理器实现中包含EL3，那么还有SPEL3。详情可参考链接[5]。

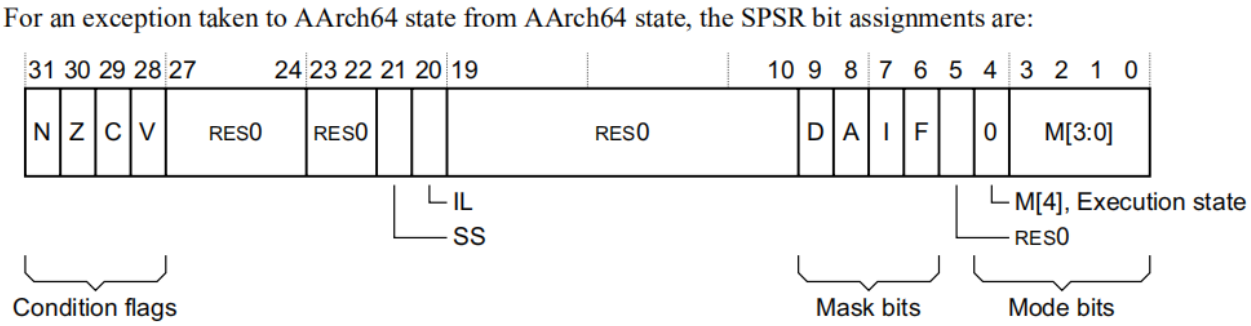
SIMD和浮点寄存器共用一系列寄存器，这些寄存器会用于浮点操作、向量操作和其它SIMD有关的标量操作。SIMD指令是能够复制多个操作数、并把它们打包在大型寄存器的一组指令集[3]。以加法指令为例，单指令单数据（SISD）的CPU对加法指令译码后，执行部件先访问内存，取得第一个操作数；之后再一次访问内存，取得第二个操作数；随后才能进行求和运算。而在SIMD型的CPU中，指令译码后几个执行部件同时访问内存，一次性获得所有操作数进行运算。浮点寄存器和SIMD寄存器共包含32个128bit位宽的寄存器，V0-V31。这些寄存器可以作为：

- 32个双字（64bit）寄存器，D0-D31。
- 32个单字（32bit）寄存器，S0-S31。
- 32个半字（16bit）寄存器，H0-H31。
- 32个单字（8bit）寄存器，B0-B31。

程序状态寄存器(Current Program Status Register，CPSR) 在用户级编程时用于存储条件码。CPSR包含条件码标志，中断禁止位，当前处理器模式以及其他状态和控制信息。

保存程序状态寄存器（SPSR，Saved Program StatusRegister）用于保存CPSR的状态，以便异常返回后恢复异常发生时的工作状态。在A64中，不再使用单一的CPSR寄存器，来保存当前处理器状态，而是用PSTATE来保存处理器状态，而在A32中依然使用CPSR。有关PSTATE和CPSR的详细信息可参考链接[4]。

A64中SPSR 格式的示意图如下图所示：



其中N、Z、C、V均为条件码标志位。它们的内容可被算术或逻辑运算的结果所改变，并且可以用于决定某条指令是否被执行，其含义如下表所示[8]：

标志位	含义
N	当两个有符号整数运算时：N=1表示运算的结果为负数；N=0表示运算的结果为正数或零。
Z	Z=1表示运算的结果为零，Z=0表示运算的结果非零。
C	可以有4种方法设置C的值：

	• 在加法指令中（包括比较指令CMP），当结果产生了进位,则C=1,表示无符号运算发生上溢出；其他情况C=0。 • 在减法指令中（包括减法指令CMP），当运算中发生借位，则C=0，表示无符号运算数发生下溢出；其他情况下C=1。 • 对于包含移位操作的非加减运算指令，C中包含最后一次溢出的位的数值。 • 对于其他非加减运算指令，C位的值通常不受影响。
V	对于加减运算指令，当操作数和运算结果为二进制的补码表示的带符号数时，V=1表示符号为溢出，通常其他指令不影响V位。

M[3:0]则用来确定异常级别和SP：

Table D1-3 M[3:0] encodings, for exceptions taken from AArch64 state

M[3:0][a]	Exception level and stack pointer
0b1101	EL3h
0b1100	EL3t
0b1001	EL2h
0b1000	EL2t
0b0101	EL1h
0b0100	EL1t
0b0000	EL0t

a. All M[3:0] encodings not shown in the table are reserved.

有关SPSR中各个位的详细信息可以参考文献[1] 1.6.4小节。

连接寄存器LR（R14）的主要作用有两个：

1. 保存子程序返回地址，用MOVE指令或BX指令可以用于实现返回，如MOV PC、LR或BXLR。若子程序中还需要调用子程序，则可以写为：

```
stmfd sp!, {lr}
……
ldmfd sp!, {pc}
```

第一条指令将LR中的内容入栈，最后一条将栈中保存的LR寄存器的内容存入PC中用于返回。

2. 当异常发生时，异常模式的LR用于保存异常返回地址，将LR内容入栈可以处理嵌套中断。

PC是程序计数器，其中保存的是正在被加载的指令，而不是正在被执行的指令。例如，若指令长度为4byte，则PC指向当前正在被执行的指令的地址+8byte的地址。关于LR和PC的详细内容可参考文献[6]和

[7]。

以下是异常级别EL3中使用的寄存器的例图：

ARM架构中处理器有不同的运行模式，因此同一个功能的寄存器在不同的运行模式下可能对应不同的物理寄存器，这些寄存器被称为备份寄存器。如SPSR_svc表示svc模式下使用的SPSR寄存器。ARM架构中常用的运行模式如下表所示[9]：

处理器模式	描述
用户模式(User, usr)	正常程序执行的模式
快速中断模式(FIQ, fiq)	用于高速数据传输和通道处理
外部中断模式(IRQ, irq)	用于通常的中断处理
特权模式(Supervisor, svc)	供操作系统使用的一种保护模式
数据访问中止模式(Abort, abt)	当数据或指令预取中止时进入该模式，用于虚拟存储及存储保护
未定义指令中止模式(Undefined, und)	当执行未定义指令时进入该模式，用于支持通过软件仿真硬件的协处理器
系统模式(System, sys)	用于运行特权级的操作系统任务

ARMv8-A架构还有Monitor（mon）工作模式，用于处理器安全状态与非安全状态的切换，Hypervisor（hyp）模式则用于对虚拟化有关功能的支持。有关安全状态的详细内容在后续的文章中会介绍。

03
ARM架构中的执行状态

ARMv8-A有两种执行模式，一种是AArch64执行模式，另一种是AArch32执行模式。执行状态定义处理单元（Processing Element, PE）的执行环境，包括以下内容：

1. 支持的寄存器宽度
2. 支持的指令集
3. 异常模型
4. 虚拟存储系统(Virtual Memory System Architecture, VMSA)架构
5. 程序员模型

AArch64为64位执行状态。对应上述内容，此执行状态：

1. 提供31个64位通用寄存器，其中X30用作过程链接寄存器（ProcedureLink Register）。
2. 提供64位程序计数器（PC）、堆栈指针（SP）和异常链接寄存器(ELRs)。
3. 提供32个128位寄存器以支持SIMD矢量和标量浮点运算。
4. 提供单一指令集A64。
5. 定义ARMv8异常模型，该模型最多有四个异常级别EL0-EL3，它们提供执行权限层次结构。
6. 支持64位虚拟寻址。
7. 定义一系列与PSTATE相关的寄存器。A64指令集包括能直接操作各种PSTATE寄存器的指令。
8. 使用后缀命名每个系统寄存器，该后缀指示可以访问寄存器的最低异常级别。

AArch32为32位执行状态。对应上述内容，此执行状态：

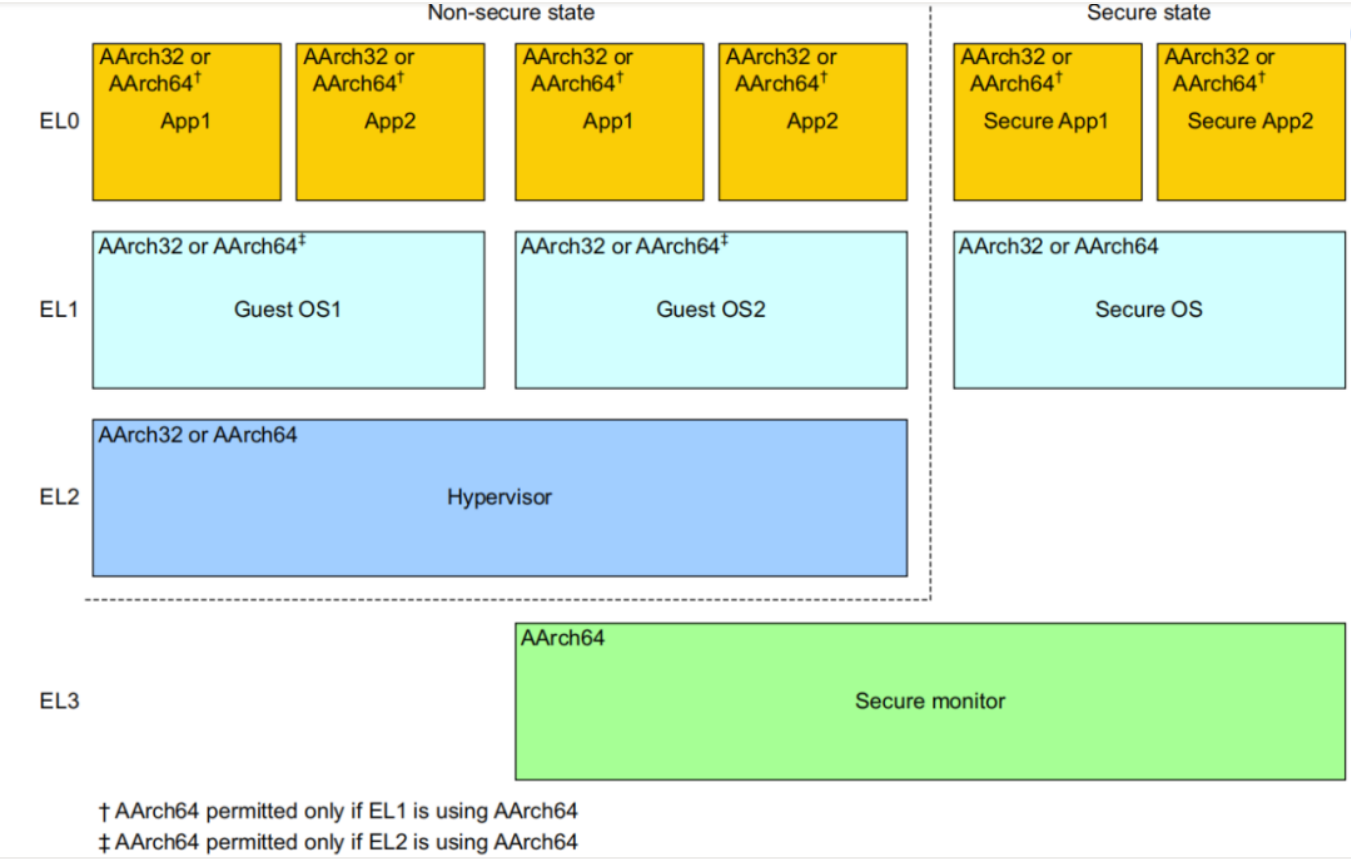
1. 提供13个32位通用寄存器和一个32位PC、一个32位SP寄存器和一个32位链接寄存器（Link Register，LR）。链接寄存器用作异常链接寄存器和过程链接寄存器。其中一些寄存器有多个备份寄存器，用于不同的处理器工作模式。我们在上一期提到过，同一个功能的寄存器在不同的处理器运行模式下可能对应不同的物理寄存器，这些寄存器被称为备份寄存器。
2. 为从Hyp(hypervisor)模式返回的异常提供一个异常链接寄存器。
3. 提供32个64位寄存器，用于对高级SIMD矢量和标量浮点计算的支持。
4. 提供两个指令集，A32和T32。
5. 支持基于处理器工作模式的ARMv7-A异常模型，并将其映射到基于异常级别的ARMv8异常模型。
6. 使用32位虚拟地址。
7. 使用单个当前程序状态寄存器（CPSR）保存处理器状态。
8. 在AArch64和AArch32执行状态之间进行转换称为内部处理(interprocessing)。

04
ARMv8-A架构的异常级别和安全状态

ARMv8-A有四个异常级别，从EL0到EL3。对于异常级别ELn，整数n增加表示软件执行的特权权限变大了。EL0级别下的执行叫非特权执行（unprivileged execution）。EL1主要用于运行操作系统内核。EL2可以支持非安全操作的虚拟化。EL3则支持安全状态和非安全状态之间的转换。安全状态与ARM TrustZone技术有关[2]。安全状态可以运行可信执行环境（TEE， Trusted Execution Environment）及安全应用，用于保障隐私数据和程序运行环境的安全性。

ARMv8-A架构并未直接指定哪些软件应该运行在哪些异常级别，但是在通常情况下，有如异常级别的使用模型：
1.应用程序运行在EL0；
2.操作系统内核和相关功能运行在EL1；
3.Hypervisor[3]运行在EL2；
4.安全世界状态和正常世界状态的切换在EL3完成。

下图反映了ARM-v8A架构中的执行状态、安全状态和异常级别之间关系[1]：



从图中我们可以看出，Hypervisor相关的支持特性主要是在EL2的非安全状态实现的。Hypervisor可以支持虚拟机之间的切换，而虚拟机主要被包含在EL1的非安全状态和EL0的非安全状态中。一些Guest OS可以运行在EL1状态里，每一个Guest OS可以运行在一个虚拟机上。而应用则运行在EL0的非安全状态中，同时也运行在Guest OS上。

引用

[1]ARM® Architecture Reference ManualARMv8， for ARMv8-A architecture profile**

[2]https://blog.csdn.net/tanli20090506/article/details/71487570

[3]ARM® Architecture Reference ManualARMv8, for ARMv8-A architectureprofile

[4]https://www.cnblogs.com/smartjourneys/p/6845078.html

[5]https://baike.baidu.com/item/SIMD/3412835?fr=aladdin

[6]http://www.lujun.org.cn/?p=1676

[7]https://www.cnblogs.com/pengdonglin137/p/10259971.html

[8]https://blog.csdn.net/layuetian2011/article/details/52039328

[9]https://blog.csdn.net/allan0508/article/details/52624618

[10]https://blog.csdn.net/myarrow/article/details/9701499

[11]《ARM体系结构与编程（第二版）》，杜春雷主编。

[12]ARM® Architecture Reference ManualARMv8, for ARMv8-A architecture profile

[13]ARM® Architecture Reference Manual ARMv8, forARMv8-A architecture profile

[13]https://baike.baidu.com/item/trustzone/15953889?fr=aladdin

[14]https://blog.csdn.net/baidu_23959681/article/details/82732488

END

关注我的微信公众号，回复“ 加群 ”按规则加入技术交流群。

技术让梦想更伟大

一个在深圳认真做技术的职场老鸟，分享嵌入式行业技术经验感悟

304篇原创内容

公众号

欢迎关注我的视频号：

技术让梦想更伟大

扫描二维码，关注我的视频号

点击“**阅读原文**”查看更多分享，欢迎**点分享、收藏、点赞、在看。**

阅读原文

喜欢此内容的人还喜欢

我用来提高编码效率的 6 个免费工具

技术的游戏

【论文分享】国内伪基站垃圾短信行为特征分析，定位犯罪团伙！

安全女巫

APP黑灰产上下游分析整理

编码安全