第5章 ARM处理器的指令系统

目录

ARM指令系统简介

ARM指令分类

ARMv8指令体系包括AArch32和AArch64两种类型指令集合，AArch32指令主要目的是向下兼容ARMv7指令系统的。而每种指令集合中又包括ARM和Thumb两种不同的指令集。当处理器工作在ARM状态时，执行ARM指令集（以下简称ARM指令）。不管是AArch32指令还是AArch64指令，所有的ARM指令均为32位宽度，指令以字对齐的方式保存在存储器中，而所有的Thumb指令都是16位宽度，指令均以半字对应的方式保存在存储器中。

ARM指令编码格式

典型的ARM指令语法格式如下，指令中各部分的含义，如表5-1所示。

{label:*} {opcode{s} {dest{, source1{, source2{, source3}}}}}

注意：花括号是可选的

标识符	Note
Label	标签
Opcode	操作码，也叫助记符，说明指令需要执行的操作类型
S	条件码设置项，决定本次指令执行是否影响PSTATE寄存器响应状态位值
Dest	目标寄存器，用于存放指令的执行结果。
Source1	第一个源操作数
Source2	第二个源操作数
Source3	第三个源操作数

ARM指令条件码域

ARM指令中支持的所有的条件码

操作码	条件助记符	标志位	含义
0000	EQ	Z == 1	相等
0001	NE	Z == 0	不相等
0010	CS	C == 1	无符号的大于或等于
0011	CC	C == 0	无符号小于
0100	MI	N == 1	负数
0101	PL	N == 0	正数或零
0110	VS	V == 1	溢出
0111	VC	V == 0	没有溢出
1000	HI	V == 1 && Z == 0	无符号数大于
1001	LS	!(C == 0 && Z == 1)	无符号数小于或等于
1010	GE	N == V	有符号数大于或等于
1011	LT	N != V	有符号数小于
1100	GT	Z == 0 && N == V	有符号数大于
1101	LE	!(Z == 1 && N != V)	有符号数小于或等于
1110	无（AL）	任意	无条件执行
1111	无（NV）	任意	无条件执行

2

ARM指令的寻址方式

数据处理指令寻址方式

数据处理指令的基本语法格式：<opcode>{S} <Xd>,<Xn>,<shifter_operand>

其中，<shifter_operand>有11种形式，如表所示。

	语　法	寻址方式
1	#<immediate>	立即数寻址
2	<Xm>	寄存器寻址
3	<Xm>, LSL #<shift_imm>	立即数逻辑左移
4	<Xm>, LSL <Rs>	寄存器逻辑左移
5	<Xm>, LSR #<shift_imm>	立即数逻辑右移
6	<Xm>, LSR <Rs>	寄存器逻辑右移
7	<Xm>, ASR #<shift_imm>	立即数算术右移
8	<Xm>, ASR <Rs>	寄存器算术右移
9	<Xm>, ROR #<shift_imm>	立即数循环右移
10	<Xm>, ROR <Rs>	寄存器循环右移
11	<Xm>, RRX	寄存器扩展循环右移

立即数寻址方式

AArch64汇编语言不要求使用“#”符号引入立即数，但汇编程序必须允许这样做。在AArch64汇编语言中在立即数前边可以加“#”符号，也可以不加“#”符号。为了提高代码的可读性，AArch64反汇编程序都会在立即数前边添加一个“#”号。

下面是一些应用立即数的指令：

```
MOV X0, #0xFF          // 将0xFF赋值给X0
ADD X1, X1, #1         // X1 = X1 + 1
CMP X7, #2000          // 将X7寄存器中的值和2000比较
ORR X9, X1, #0xFF     // 将X1中的[7:0]位置1，结果写到X9中
或者
MOV X0, 0xFF                    // 将0xFF赋值给X0
ADD X1, X1, 1                   // X1 = X1 +1
CMP X7, 2000                    // 将X7寄存器中的值和2000比较
ORR X9, X1, 0xFF      // 将X1中的[7:0]位置1，结果写到X9中
```

寄存器寻址方式

寄存器的值可以被直接用于数据操作指令，这种寻址方式是各类处理器经常采用的一种方式，也是一种执行效率较高的寻址方式，举例如下。

```
MOV   X2, X0              // X0的值赋值X2
ADD   X4, X3, X2          // X2加X3，结果赋值X4
CMP   X7, X8              // 比较X7和X8的值
```

寄存器移位寻址方式

寄存器的值在被送到ALU之前，可以事先经过桶形移位寄存器的处理。预处理和移位发生在同一周期内，所以有效地使用移位寄存器，可以增加代码的执行效率。

下面是一些在指令中使用了移位操作的例子。

```
ADD  X2, X0, X1, LSR #5    // X1中的值右移5位，X0加X1，结果赋值给X2
MOV  X1, X0, LSL #2        // X0中的值左移2位，结果赋值给X1
SUB  X1, X2, X0, LSR #4    // X0中的值右移4位，X2减X0，结果赋值给X1
```

内存访问指令寻址方式

在AArch64指令集中的加载/存储寻址模式大致遵循T32，使用通用寄存器Xn（n = 0-30）或当前堆栈指针SP的64位基址，具有立即数或寄存器偏移量方式。完整的寻址模式如表5-4所示。某些类型的加载/存储指令可能仅支持其中的一部分。

寻址方式	偏移		
	立即数	寄存器	扩展寄存器
基址寄存器(无偏移)	[base{,#0}]	-	-
基址寄存器(加偏移)	[base{,#imm}]	[base,Xm{,LSL #imm}]	[base,Wm,(S|U)XTW {#imm}]
事先更新寻址	[base,#imm]!	-	-
事后更新寻址	[base],#imm	[base],Xm	-
PC相对寻址	label	-	-

ARM指令集

数据处理指令

数据处理指令是指对存放在寄存器中的数据进行操作的指令。主要包括：

数据传送指令

算术运算指令

逻辑运算指令

移位运算指令

比较与测试指令

乘法指令

数据传送指令

数据传送指令多用于设置初始值或者在寄存器间传送数据。常见的数据传送指令，如表所示。

指令格式	指令含义
MOVZ Wt, #uimm16{, LSL #pos}	将16位立即数搬移到32位寄存器中，其他位为0：Wt = LSL(uimm16, pos)
MOVZ Xt, #uimm16{, LSL #pos}	将16位立即数搬移到64位寄存器中，其他位为0：Xt = LSL(uimm16, pos)
MOVN Wt, #uimm16{, LSL #pos}	将16位立即数取反搬移到32寄存器中，其他位为0：Wt = NOT(LSL(uimm16, pos))
MOVN Xt, #uimm16{, LSL #pos}	将16位立即数取反搬移到64位寄存器中，其他位为0：Xt = NOT(LSL(uimm16, pos))
MOVK Wt, #uimm16{, LSL #pos}	将16位立即数搬移到32位寄存器中，其他位保持不变：Wt<pos+15:pos> = uimm16
MOVK Xt, #uimm16{, LSL #pos}	将16位立即数搬移到64位寄存器中，其他位保持不变：Xt<pos+15:pos> = uimm16
MOV Wd|WZR|WSP, #imm32	将32位立即数搬移到32位寄存器中：Wd|WZR|WSP = #imm32
MOV Xd|XZR|SP, #imm64	将64位立即数搬移到64位寄存器中：Xd|XZR|SP = #imm64
MOV Wd|WSP, Wn|WSP{, LSL #pos}	将Wn|WSP寄存器中的值搬移到Wd|WSP寄存器中：Wd|WSP = LSL(Wn|WSP, pos)
MOV Xd|SP, Xn|SP{, LSL #pos}	将Xn|SP寄存器中的值搬移到Xd|SP寄存器中：Xd|SP = LSL(Xn|XSP, pos)

算数运算指令

算数运算指令主要用于数学运算相关指令。ARM指令集中的算数运算指令主要用于两个数之间的运算。ARMv8支持的加减相关的算数运算指令，如表所示。

指令格式	指令含义
ADD Wd|WSP, Wn|WSP, #aimm	Wd|WSP = Wn|WSP + aimm
ADD Xd|SP, Xn|SP, #aimm	Xd|SP = Xn|SP + aimm
ADDS Wd, Wn|WSP, #aimm	Wd = Wn|WSP + aimm，并设置状态标志位
ADDS Xd, Xn|SP, #aimm	Xd = Xn|SP + aimm，并设置状态标志位
SUB Wd|WSP, Wn|WSP, #aimm	Wd|WSP = Wn|WSP - aimm
SUB Xd|SP, Xn|SP, #aimm	Xd|SP = Xn|SP - aimm
SUBS Wd, Wn|WSP, #aimm	Wd = Wn|WSP - aimm，并设置状态标志位
SUBS Xd, Xn|SP, #aimm	Xd = Xn|SP - aimm，并设置状态标志位
ADD Wd, Wn, Wm{, ashift #imm}	Wd = Wn + ashift(Wm, imm)
ADD Xd, Xn, Xm{, ashift #imm}	Xd = Xn + ashift(Xm, imm)
ADDS Wd, Wn, Wm{, ashift #imm}	Wd = Wn + ashift(Wm, imm) ，并设置状态标志位
ADDS Xd, Xn, Xm{, ashift #imm}	Xd = Xn + ashift(Xm, imm) ，并设置状态标志位
SUB Wd, Wn, Wm{, ashift #imm}	Wd = Wn - ashift(Wm, imm)
SUB Xd, Xn, Xm{, ashift #imm}	Xd = Xn - ashift(Xm, imm)
SUBS Wd, Wn, Wm{, ashift #imm}	Wd = Wn - ashift(Wm, imm)，并设置状态标志位
SUBS Xd, Xn, Xm{, ashift #imm}	Xd = Xn - ashift(Xm, imm)，并设置状态标志位
ADC Wd, Wn, Wm	Wd = Wn + Wm + C
ADC Xd, Xn, Xm	Xd = Xn + Xm + C
ADCS Wd, Wn, Wm	Wd = Wn + Wm + C，并设置状态标志位
ADCS Xd, Xn, Xm	Xd = Xn + Xm + C，并设置状态标志位
SBC Wd, Wn, Wm	Wd = Wn - Wm - 1 + C
SBC Xd, Xn, Xm	Xd = Xn - Xm - 1 + C
SBCS Wd, Wn, Wm	Wd = Wn - Wm - 1 + C，并设置状态标志位
SBCS Xd, Xn, Xm	Xd = Xn - Xm - 1 + C，并设置状态标志位

逻辑运算指令

逻辑运算指令主要用于逻辑运算及逻辑表达式的实现。ARMv8架构支持的逻辑运算相关指令，如表所示。

指令格式	指令含义
AND Wd|WSP, Wn, #bimm32	Wd|WSP = Wn AND bimm32
AND Xd|SP, Xn, #bimm64	Xd|SP = Xn AND bimm64
ANDS Wd, Wn, #bimm32	d = Wn AND bimm32，根据结果设置N和Z条件标志位，并清除C和V条件标志位。
ANDS Xd, Xn, #bimm64	Xd = Xn AND bimm64，根据结果设置N和Z条件标志位，并清除C和V条件标志位
EOR Wd|WSP, Wn, #bimm32	Wd|WSP = Wn EOR bimm32
EOR Xd|SP, Xn, #bimm64	Xd|SP = Xn EOR bimm64
ORR Wd|WSP, Wn, #bimm32	Wd|WSP = Wn OR bimm32
ORR Xd|SP, Xn, #bimm64	Xd|SP = Xn OR bimm64
TST Wn, #bimm32	等价于ANDS WZR,Wn,#bimm32
TST Xn, #bimm64	等价于ANDS XZR,Xn,#bimm64
AND Wd, Wn, Wm{, lshift #imm}	Wd = Wn AND lshift(Wm, imm)
AND Xd, Xn, Xm{, lshift #imm}	Xd = Xn AND lshift(Xm, imm)
ANDS Wd, Wn, Wm{, lshift #imm}	Wd = Wn AND lshift(Wm, imm) ，根据结果设置N和Z条件标志位，并清除C和V条件标志位
ANDS Xd, Xn, Xm{, lshift #imm}	Xd = Xn AND lshift(Xm, imm) ，根据结果设置N和Z条件标志位，并清除C和V条件标志位
BIC Wd, Wn, Wm{, lshift #imm}	Wd = Wn AND NOT(lshift(Wm, imm))
BIC Xd, Xn, Xm{, lshift #imm}	Xd = Xn AND NOT(lshift(Xm, imm))
BICS Wd, Wn, Wm{, lshift #imm}	Wd = Wn AND NOT(lshift(Wm, imm)) ，根据结果设置N和Z条件标志位，并清除C和V条件标志位
BICS Xd, Xn, Xm{, lshift #imm}	Xd = Xn AND NOT(lshift(Xm, imm))，根据结果设置N和Z条件标志位，并清除C和V条件标志位
EON Wd, Wn, Wm{, lshift #imm}	Wd = Wn EOR NOT(lshift(Wm, imm))
EON Xd, Xn, Xm{, lshift #imm}	Xd = Xn EOR NOT(lshift(Xm, imm))
EOR Wd, Wn, Wm{, lshift #imm}	Wd = Wn EOR lshift(Wm, imm)
EOR Xd, Xn, Xm{, lshift #imm}	Xd = Xn EOR lshift(Xm, imm)
ORR Wd, Wn, Wm{, lshift #imm}	Wd = Wn OR lshift(Wm, imm)
ORR Xd, Xn, Xm{, lshift #imm}	Xd = Xn OR lshift(Xm, imm)
ORN Wd, Wn, Wm{, lshift #imm}	Wd = Wn OR NOT(lshift(Wm, imm))

移位操作指令

移位操作指令主要用于对数据进行移位操作的，对于有些时候使用移位操作会让我们的程序可读性变得更强。ARMv8结果支持的移位操作指令，如表所示。

指令格式	指令含义
ASR Wd, Wn, #uimm	算数右移，低位移出高位补符号位
ASR Xd, Xn, #uimm	算数右移，低位移出高位补符号位
LSL Wd, Wn, #uimm	逻辑左移，高位移出低位补0
LSL Xd, Xn, #uimm	逻辑左移，高位移出低位补0
LSR Wd, Wn, #uimm	逻辑右移，低位移出高位补0
LSR Xd, Xn, #uimm	逻辑右移，低位移出高位补0
ROR Wd, Wm, #uimm	循环右移，低位移出补到高位
ROR Xd, Xm, #uimm	循环右移，低位移出补到高位
ASR Wd, Wn, Wm	算数右移，低位移出高位补符号位
ASR Xd, Xn, Xm	算数右移，低位移出高位补符号位
LSL Wd, Wn, Wm	逻辑左移，高位移出低位补0
LSL Xd, Xn, Xm	逻辑左移，高位移出低位补0
LSR Wd, Wn, Wm	逻辑右移，低位移出高位补0
LSR Xd, Xn, Xm	逻辑右移，低位移出高位补0
ROR Wd, Wm, Wm	循环右移，低位移出补到高位
ROR Xd, Xm, Xm	循环右移，低位移出补到高位

整数乘法

ARM乘法指令完成两个数据的乘法。两个64位二进制数相乘的结果是128位的积。在有些ARM的处理器版本中，将乘积的结果保存到两个独立的寄存器中。另外一些版本只将最低有效32位存放到一个寄存器中。无论是哪种版本的处理器，都有乘—累加的变型指令，将乘积连续累加得到总和。而且有符号数和无符号数都能使用。对于有符号数和无符号数，结果的最低有效位是一样的。因此，对于只保留64位结果的乘法指令，不需要区分有符号数和无符号数这两种情况，如表5-10所示为各种形式乘法指令。

指令格式	指令含义
MADD Wd, Wn, Wm, Wa	乘-加(32-bit):Wd = Wa + (Wn × Wm)
MADD Xd, Xn, Xm, Xa	乘-加(64-bit):Xd = Xa + (Xn × Xm)
MSUB Wd, Wn, Wm, Wa	乘-减(32-bit):Wd = Wa – (Wn × Wm)
MSUB Xd, Xn, Xm, Xa	乘-减(64-bit):Xd = Xa – (Xn × Xm)
MNEG Wd, Wn, Wm	乘-反(32-bit):Wd = –(Wn × Wm)等价于MSUB Wd, Wn, Wm, WZR
MNEG Xd, Xn, Xm	乘-反(64-bit):Xd = –(Xn × Xm)等价于MSUB Xd, Xn, Xm, XZR
MUL Wd, Wn, Wm	乘(32-bit):Wd = Wn × Wm等价于MADD Wd, Wn, Wm, WZR
MUL Xd, Xn, Xm	乘(64-bit):Xd = Xn × Xm.等价于 MADD Xd, Xn, Xm, XZR
SMADDL Xd, Wn, Wm, Xa	有符号乘-加(Long):Xd = Xa + (Wn × Wm)源操作数作为有符号数
SMSUBL Xd, Wn, Wm, Xa	有符号乘-减(Long):Xd = Xa - (Wn × Wm)源操作数作为有符号数
SMNEGL Xd, Wn, Wm	有符号乘-反(Long):Xd = Xa - (Wn × Wm)源操作数作为有符号数等价于SMSUBL Xd, Wn, Wm, XZR
SMULL Xd, Wn, Wm	有符号乘(Long):Xd = Wn x Wm源操作数作为有符号数等价于SMADDL Xd, Wn, Wm, XZR
SMULH Xd, Xn, Xm	有符号乘(High):Xd = (Xn × Xm)<127:64>源操作数作为有符号数等价于UMADDL Xd, Wn, Wm, Xa
UMADDL Xd, Wn, Wm, Xa	无符号乘-加(Long): Xd = Xa + (Wn × Wm)源操作数作为无符号数
UMSUBL Xd, Wn, Wm, Xa	无符号乘-减(Long): Xd = Xa - (Wn × Wm)源操作数作为无符号数
UMNEGL Xd, Wn, Wm	无符号乘-反(Long): Xd = -(Wn × Wm)源操作数作为无符号数等价于UMSUBL Xd, Wn, Wm, XZR
UMULL Xd, Wn, Wm	无符号乘(Long): Xd = (Wn × Wm)源操作数作为无符号数等价于UMADDL Xd, Wn, Wm, XZR
UMULH Xd, Xn, Xm	无符号乘-加(High): Xd = (Xn × Xm) <127:64>源操作数作为无符号数

除法指令

整数除法指令用于计算两个数相除，商进行四舍五入取整数。余数可以通过MSUB指令分子-(商*分母)计算得到。如表所示为各种形式除法指令。

注：除数指令在除以0时不会产生错误，而是将0写入目标寄存器。

指令格式	指令含义
SDIV Wd, Wn, Wm	32位有符号除法：Wd = Wn ÷ Wm源操作数位有符号数
SDIV Xd, Xn, Xm	64位有符号除法：Xd = Xn ÷ Xm源操作数位有符号数
UDIV Wd, Wn, Wm	32位无符号除法：Wd = Wn ÷ Wm源操作数位无符号数
UDIV Xd, Xn, Xm	64位有符号除法：Xd = Xn ÷ Xm源操作数位无符号数

单寄存器的Load/Store指令（对齐）

指令格式	指令含义
LDR Wt, addr	从内存的addr地址中加载一个字到Wt寄存器中
LDR Xt, addr	从内存的addr地址中加载一个双字到Xt寄存器中
LDRB Wt, addr	从内存的addr地址中加载一个字节到Wt寄存器中
LDRH Wt, addr	从内存的addr地址中加载一个半字到Wt寄存器中
STR Wt, addr	存储字从Wt到内存的addr地址中
STR Xt, addr	存储双字从Xt到内存的addr地址中
STRB Wt, addr	存储字节从Wt到内存的addr地址中
STRH Wt, addr	存储半字从Wt到内存的addr地址中

单寄存器的Load/Store指令（未对齐偏移）

指令格式	指令含义
LDUR Wt, [base,#simm9]	从内存的base+simm9地址中加载一个字到Wt寄存器中
LDUR Xt, [base,#simm9]	从内存的base+simm9地址中加载一个双字到Xt寄存器中
LDURB Wt, [base,#simm9]	从内存的base+simm9地址中加载一个字节到Wt寄存器中
LDURH Wt, [base,#simm9]	从内存的base+simm9地址中加载一个半字到Wt寄存器中
STUR Wt, [base,#simm9]	存储字从Wt到内存的base+simm9地址中
STUR Xt, [base,#simm9]	存储双字从Wt到内存的base+simm9地址中
STURB Wt, [base,#simm9]	存储字节从Wt到内存的base+simm9地址中
STURH Wt, [base,#simm9]	存储半字从Wt到内存的base+simm9地址中

Load/Store Pair(对)指令（对齐）

指令格式	指令含义
LDP Wt1, Wt2, addr	从内存的addr地址中加载两个字分别到Wt1和Wt2寄存器中
LDP Xt1, Xt2, addr	从内存的addr地址中加载两个双字分别到Xt1和Xt2寄存器中
STP Wt1, Wt2, addr	存储Wt1和Wt2寄存器中的两个字到内存的addr地址中
STP Xt1, Xt2, addr	存储Xt1和Xt2寄存器中的两个双字到内存的addr地址中

Load/Store Non-temporal(非暂存) Pair(对)指令

指令格式	指令含义
LDNP Wt1, Wt2, [base,#imm]	从内存的base+imm地址中加载两个字分别到Wt1和Wt2寄存器中
LDNP Xt1, Xt2, [base,#imm]	从内存的base+imm地址中加载两个双字分别到Xt1和Xt2寄存器中
STNP Wt1, Wt2, [base,#imm]	存储Wt1和Wt2寄存器中的两个字到内存的base+imm地址中
STNP Xt1, Xt2, [base,#imm]	存储Xt1和Xt2寄存器中的两个双字到内存的base+imm地址中

非特权Load/Store 指令

指令格式	指令含义
LDTR Wt, [base,#simm9]	从内存的base+simm9地址中加载字到Wt寄存器中，在EL1时使用EL0特权
LDTR Xt, [base,#simm9]	从内存的base+simm9地址中加载双字到Xt寄存器中，在EL1时使用EL0特权
LDTRB Wt, [base,#simm9]	从内存的base+simm9地址中加载字节到Wt寄存器中，在EL1时使用EL0特权
LDTRH Wt, [base,#simm9]	从内存的base+simm9地址中加载半字到Wt寄存器中，在EL1时使用EL0特权
STTR Wt, [base,#simm9]	存储Xt寄存器中的字到内存的base+simm9地址中，在EL1时使用EL0特权
STTR Xt, [base,#simm9]	存储Xt寄存器中的双字到内存的base+simm9地址中，在EL1时使用EL0特权
STTRB Wt, [base,#simm9]	存储Wt寄存器中的字节到内存的base+simm9地址中，在EL1时使用EL0特权
STTRH Wt, [base,#simm9]	存储Wt寄存器中的半字到内存的base+simm9地址中，在EL1时使用EL0特权

Load-Store Exclusive指令

指令格式	指令含义
LDXR Wt, [base{,#0}]	从内存的base地址中加载字到Wt寄存器中，将物理地址记录为独占访问
LDXR Xt, [base{,#0}]	从内存的base地址中加载双字到Xt寄存器中，将物理地址记录为独占访问
LDXRB Wt, [base{,#0}]	从内存的base地址中加载字节到Wt寄存器中，将物理地址记录为独占访问
LDXRH Wt, [base{,#0}]	从内存的base地址中加载半字到Wt寄存器中，将物理地址记录为独占访问
LDXP Wt1, Wt2, [base{,#0}]	从内存的base地址中加载字到Wt1和Wt2寄存器中，将物理地址记录为独占访问
LDXP Xt1, Xt2, [base{,#0}]	从内存的base地址中加载双字到Xt1和Xt2寄存器中，将物理地址记录为独占访问
STXR Ws, Wt, [base{,#0}]	存储Wt寄存器中的字到内存的base地址中，并将Ws设置为返回的独占访问状态
STXR Ws, Xt, [base{,#0}]	存储Xt寄存器中的字到内存的base地址中，并将Ws设置为返回的独占访问状态
STXRB Ws, Wt, [base{,#0}]	存储Wt寄存器中的字节内存的base地址中，并将Ws设置为返回的独占访问状态
STXRH Ws, Wt, [base{,#0}]	存储Wt寄存器中的半字到内存的base地址中，并将Ws设置为返回的独占访问状态
STXP Ws, Wt1, Wt2, [base{,#0}]	存储Wt1和Wt2寄存器中的字到内存的base地址中，并将Ws设置为返回的独占访问状态。
STXP Ws, Xt1, Xt2, [base{,#0}]	存储Xt1和Xt2寄存器中的双字到内存的base地址中，并将Ws设置为返回的独占访问状态。

条件分支指令

除非特殊说明，否则条件分支相对于程序计数寄存器(PC)位置偏移±1MB。ARMv8架构支持的条件分支指令如表所示。

指令格式	指令含义
B.cond label	分支：如果条件成立有条件的跳转到对应的程序标签处
CBNZ Wn, label	比较和分支不为零(32位):如果Wn不等于零，有条件地跳转到相对应程序标签
CBNZ Xn, label	比较和分支不为零(64位):如果Xn不等于零，有条件地跳转到相对应程序标签
CBZ Wn, label	比较和分支为零(32位):如果Wn等于零，有条件地跳转到相对应程序标签
CBZ Xn, label	比较和分支为零(64位):如果Xn等于零，有条件地跳转到相对应程序标签
TBNZ Xn|Wn, #uimm6, label	测试和分支不为零：如果寄存器Xn中的uimm6位不为零，则有条件地跳转到相对应标签。该位数表示寄存器的宽度，可以写入，并且如果uimm6小于32，则应将其反汇编为Wn。分支偏移范围限制为±32KB
TBZ Xn|Wn, #uimm6, label	测试和分支为零：如果寄存器Xn中的uimm6位为零，则有条件地跳转到相对应标签。 该位数表示寄存器的宽度，可以写入，并且如果uimm6小于32，则应将其反汇编为Wn。分支偏移范围限制为±32KB

无条件分支指令

无条件分支指令支持的立即数分支偏移范围为±128MB。ARMv8架构支持的无条件分支指令（立即数）如表所示。

指令格式	指令含义
B label	分支：
BL label	分支和链接：无条件跳转到PC相对标签，跳转指令的下一条指令的地址到X30寄存器中
BR Xm	分支寄存器:无条件跳转到Xm寄存器地址处，并提示CPU这不是子例程返回
BLR Xm	分支和链接寄存器:无条件地跳转到Xm寄存器地址处，写跳转指令的下一条指令的地址到X30寄存器中
RET {Xm}	返回:跳转到Xm寄存器地址处，并提示CPU这是一个子例程返回。如果省略Xm，则汇编程序默认返回X30寄存器对应的地址中

程序状态寄存器访问指令

指令格式	指令含义
MRS Xt, <system_register>	将< system_register >移动到xt，其中< system_register >是如上所述的系统寄存器名
MSR <system_register>, Xt	将xt移动到< system_register >，其中< system_register >是如上所述的系统寄存器名
MSR DAIFClr, #uimm4	使用uimm4作为位掩码选择清除一个或多个DAIF异常掩码:位3选择D掩码，位2选择a掩码，位1选择I掩码，位0选择F掩码
MSR DAIFSet, #uimm4	使用uimm4作为位掩码来选择一个或多个DAIF异常掩码的设置:位3选择D掩码，位2选择a掩码，位1选择I掩码，位O选择F掩码
MSR SPSel, #uimm4	使用uimm4作为控制值来选择当前栈指针:如果第0位被设置，则选择当前异常级别的栈指针;如果第0位被清除，则选择共享的EL0栈指针。uimm4的1到3位是保留的，应该是零

异常产生指令

ARM指令集中提供异常产生的指令，通过这两条指令可以用软件的方法实现异常，如表所示为ARM异常产生指令。

指令格式	指令含义
SVC #uimm16	生成针对异常级别1(system)的异常，在uimm16中使用16位有效载荷
HVC #uimm16	生成针对异常级别2 (hypervisor)的异常，在uimm16中使用16位有效负载
SMC #uimm16	生成针对异常级别3(secure monitor)的异常，在uimm16中使用16位有效负载
ERET	异常返回:从当前异常级别的SPSR_ELn寄存器重新构造处理器状态，并分支到ELR_ELn中的地址

ARM汇编语言伪指令

ADR伪指令

ADR伪指令的功能是把标签所在的地址加载到寄存器中。这个指令将基于PC相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中。当地址值是字节对齐时，取值范围为-255～255Byte；当地址值是字对齐时，取值范围为-1020～1020Byte。当地址值是16字节对齐时，其取值范围更大。这条指令等价于add <register>，pc，offset。offset（是当前指令和标号的偏移量）。

（1）指令的语法格式

```
ADR 〈register〉〈label〉
register: 要装载的寄存器编号。
abel: 基于 PC 或具体寄存器的表达式。
```

（2）指令举例

```
Start:
MOV X0, #10          @ PC的值是当前指令的地址加8
ADR X4, Start
```

LDR伪指令

LDR伪指令用于装载一个64位的常数或一个地址到寄存器中。

（1）指令的语法格式

LDR register, =expr

register：目标寄存器。

expr是64位常量表达式。汇编器根据expr的取值情况，对LDR伪指令做如下处理。

当expr表示的值可以作为MOV和MOVN指令中的立即数时，汇编器用MOV和MVN指令代替LDR指令。当expr表示的值不能作为MOV和MOVN指令中的立即数时，汇编器将expr表示的值放到内存空间当中，然后再用ldr内存读取指令读取该常数到寄存器中。

（2）指令举例

LDR X3, =0xff0

谢谢聆听！