目 录

第 1 章 ARMv8 简介

1.1 基础认识

ARMv8 的架构继承以往 ARMv7 与之前处理器技术的基础,除了现有的 16/32bit 的 Thumb2 指令支持外,也向前兼容现有的 A32(ARM 32bit)指令集,基于 64bit 的 AArch64 架构,除了新增 A64(ARM 64bit)指令集外，也扩充了现有的 A32(ARM 32bit)和 T32(Thumb2 32bit）指令集，另外还新增加了 CRYPTO(加密)模块支持。

1.2 相关专业名词解释

AArch32	描述 32bit Execution State
AArch64	描述 64bit Execution State
A32、T32	AArch32 ISA （Instruction Architecture）
A64	AArch64 ISA （Instruction Architecture）
Interprocessing	描述 AArch32 和 AArch64 两种执行状态之间的切换
SIMD	Single-Instruction, Multiple-Data （单指令多数据）

(参考文档：ARMv8-A Architecture reference manual-DDI0487A_g_armv8_arm.pdf)

第 2 章 Execution State

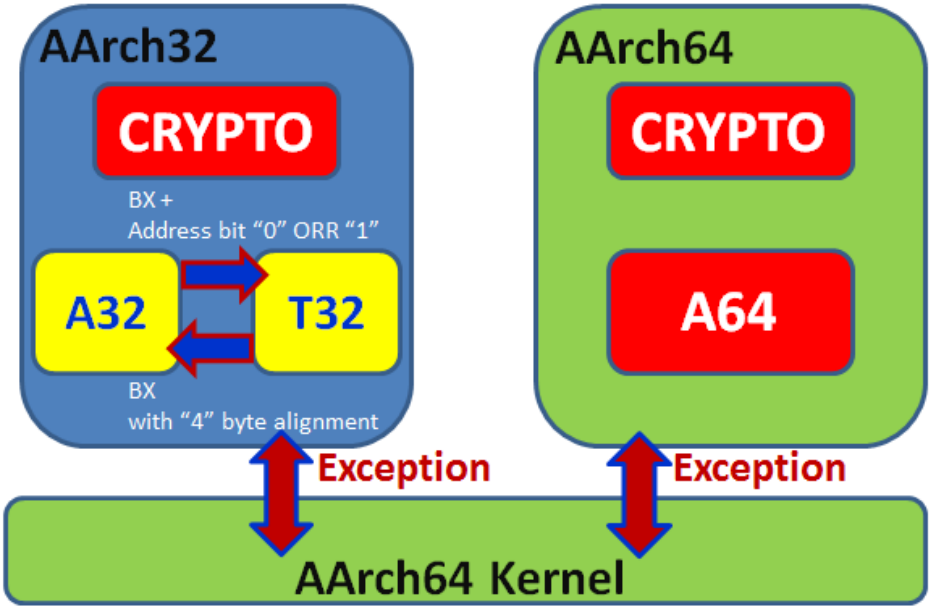
2.1 提供两种 Execution State

• ARMv8 提供 AArch32 state 和 AArch64 state 两种 Execution State，下面是两种 Execution State 对比.

Execution State	Note
AArch32	提供 13 个 32bit 通用寄存器 R0-R12，一个 32bit PC 指针 (R15)、堆栈指针 SP (R13)、链接寄存器 LR (R14)
	提供一个 32bit 异常链接寄存器 ELR，用于 Hyp mode 下的异常返回
	提供 32 个 64bit SIMD 向量和标量 floating-point 支持
	提供两个指令集 A32（32bit）、T32（16/32bit）
	兼容 ARMv7 的异常模型
	协处理器只支持 CP10\CP11\CP14\CP15
AArch64	提供 31 个 64bit 通用寄存器 X0-X30（W0-W30），其中 X30 是程序链接寄存器 LR
	提供一个 64bit PC 指针、堆栈指针 SPx 、异常链接寄存器 ELRx
	提供 32 个 128bit SIMD 向量和标量 floating-point 支持
	定义 ARMv8 异常等级 ELx（x<4），x 越大等级越高，权限越大
	定义一组 PE state 寄存器 PSTATE（NZCV/DAIF/CurrentEL/SPSel 等），用于保存 PE 当前的状态信息
	没有协处理器概念

2.2 决定 Execution State 的条件

SPSR_EL1.M[4] 决定 EL0 的执行状态，为 0 =>64bit ，否则=>32bit
HCR_EL2.RW 决定 EL1 的执行状态，为 1 =>64bit ，否则=>32bit
SCR_EL3.RW 确定 EL2 or EL1 的执行状态，为 1 =>64bit ，否则=>32bit
AArch32 和 AArch64 之间的切换只能通过发生异常或者系统 Reset 来实现.（A32 -> T32 之间是通过 BX 指令切换的）



第 3 章 Exception Level

• ARMv8 定义 EL0-EL3 共 4 个 Exception Level 来控制 PE 的行为.

ELx（x<4），x 越大等级越高，执行特权越高
执行在 EL0 称为非特权执行
EL2 没有 Secure state，只有 Non-secure state
EL3 只有 Secure state，实现 EL0/EL1 的 Secure 和 Non-secure 之间的切换
EL0 & EL1 必须要实现，EL2/EL3 则是可选实现

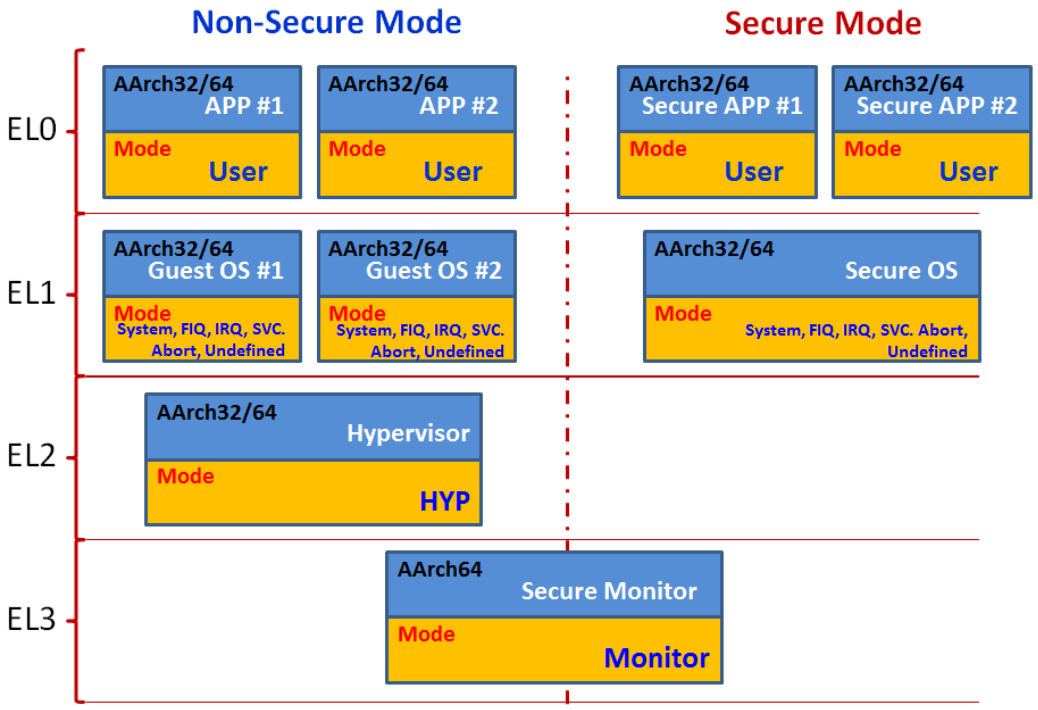
3.1 Exception Level 与 Security

Exception Level	
EL0	Application
EL1	Linux kernel- OS
EL2	Hypervisor（可以理解为上面跑多个虚拟 OS）
EL3	Secure Monitor(ARM Trusted Firmware)
Security	
Non-secure	EL0/EL1/EL2, 只能访问 Non-secure memory
Secure	EL0/EL1/EL3, 可以访问 Non-secure memory & Secure memory,可起到物理屏障安全隔离作用

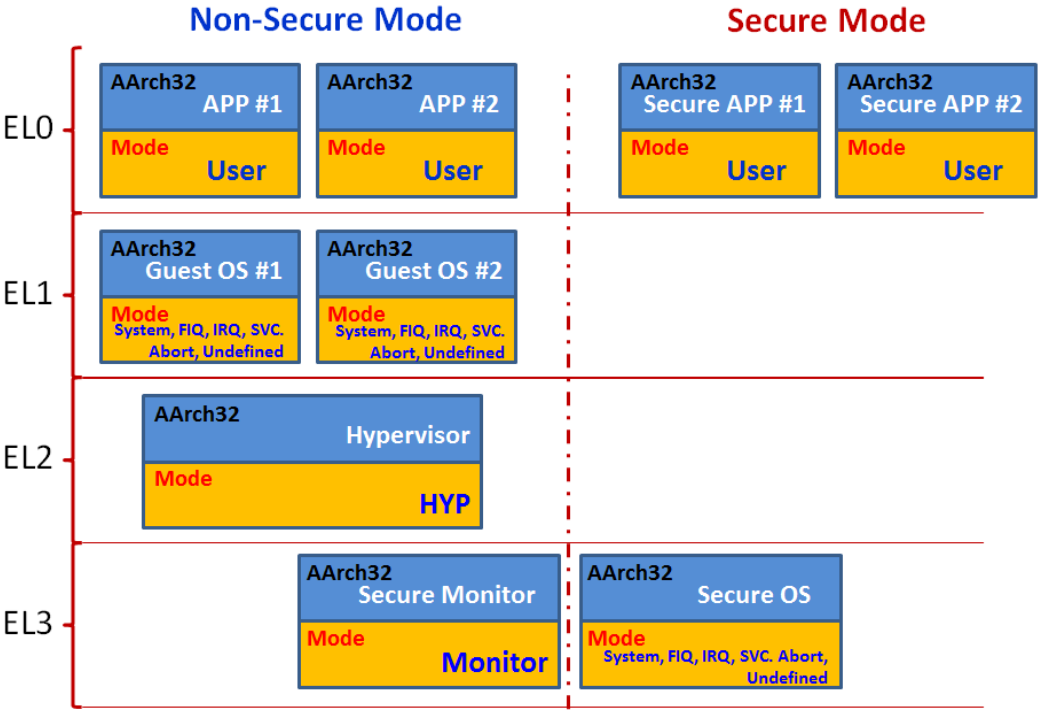
3.1.1 EL3 使用 AArch64、AArch32 的对比

	Note
Common	User mode 只执行在 Non- Secure EL0 or Secure EL0
	SCR_EL3.NS 决定的是 low level EL 的 secure/non-secure 状态，不是绝对自身的
	EL2 只有 Non-secure state
	EL0 既有 Non-secure state 也有 Secure state
EL3 AArch64	若 EL1 使用 AArch32,那么 Non- Secure {SYS/FIQ/IRQ/SVC/ABORT/UND} 模式执行在 Non-secure EL1，Secure {SYS/FIQ/IRQ/SVC/ABORT/UND}模式执行在 Secure EL1
	若 SCR_EL3.NS == 0,则切换到 Secure EL0/EL1 状态，否则切换到 Non-secure EL0/EL1 状态
	Secure state 只有 Secure EL0/EL1/EL3
EL3 AArch32	User mode 只执行在 Non- Secure EL0 or Secure EL0
	若 EL1 使用 AArch32,那么 Non- Secure {SYS/FIQ/IRQ/SVC/ABORT/UND} 模式执行在 Non-secure EL1，Secure {SYS/FIQ/IRQ/SVC/ABORT/UND}模式执行在 EL3
	Secure state 只有 Secure EL0/EL3，没有 Secure EL1，要注意和上面的情况不同

• 当 EL3 使用 AArch64 时，有如下结构组合：



• 当 EL3 使用 AArch32 时，有如下结构组合：



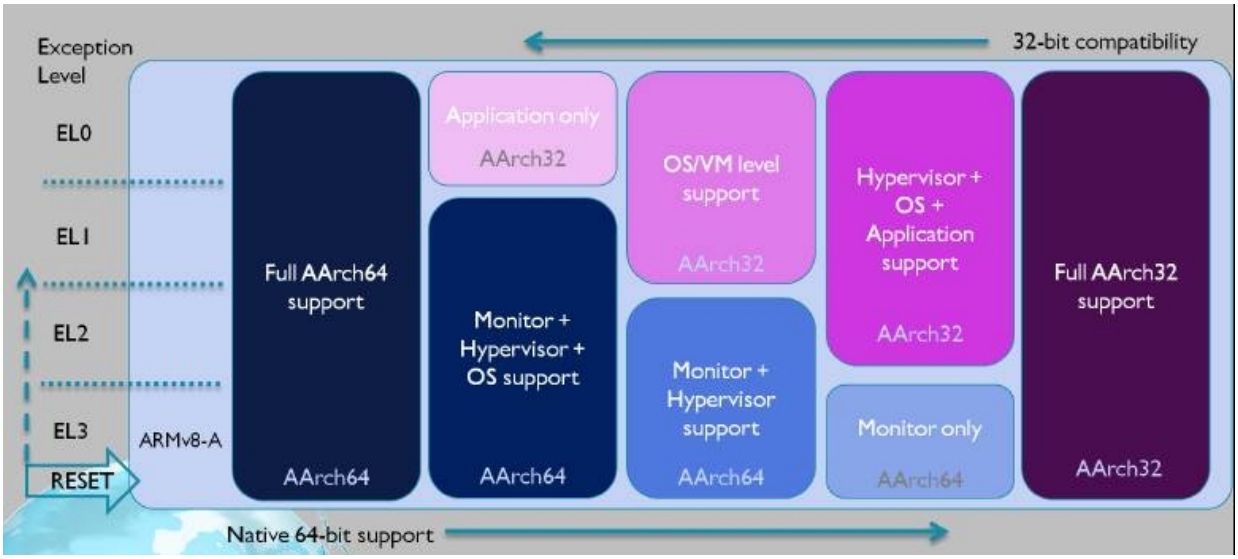
3.2 ELx 和 Execution State 组合

•假设 EL0-EL3 都已经实现，那么将会有如下组合

五类组合	
EL0/EL1/EL2/EL3 => AArch64	此两类组合不存在 64bit -> 32bit 之间的所谓 Interprocessing 切换
EL0/EL1/EL2/EL3 => AArch32	

EL0 => AARCH32, EL1/EL2/EL3 => AArch64	此三类组合存在 64bit -> 32bit 之间的所谓 Interprocessing 切换
EL0/EL1 => AArch32, EL2/EL3 => AArch64	
EL0/EL1/EL2 => AArch32, EL3 => AArch64	
组合规则	
字宽（ELx）<= 字宽（EL(x+1)） { x=0, 1, 2 }	原则：上层字宽不能大于底层字宽

- 五类经典组合图示



3.3 路由控制

- 如果 EL3 使用 AArch64, 则有如下异常路由控制

3.3.1 路由规则

- 路由规则如下图所示（from ARMv8 Datasheet）：

SCR_EL3.EA SCR_EL3.IRQ SCR_EL3.FIQ	SCR_EL3.RW	AMO[a] IMO[a] FMO[a]	Target Exception level when executing in:					
			Non-secure			Secure		
			EL0	EL1	EL2	EL0	EL1	EL3
0	0	0	EL1	EL1	EL2	EL1	EL1	C
	X	1	EL2	EL2	EL2	EL1	EL1	C
	1	0	EL1	EL1	C	EL1	EL1	C
1	X	X	EL3	EL3	EL3	EL3	EL3	EL3

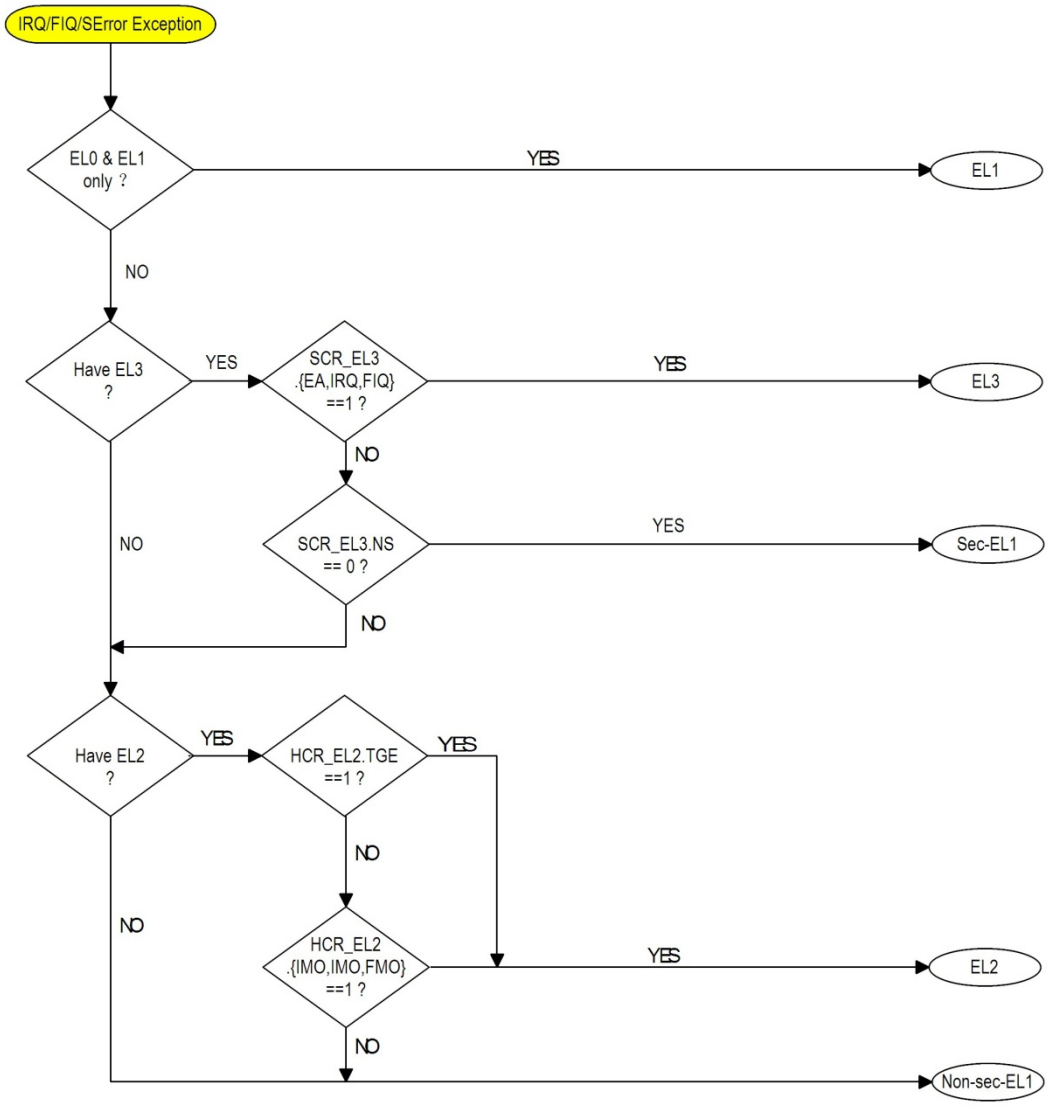
a. If EL2 is using AArch64, these are the HCR_EL2.{AMO, IMO, FMO} control bits. If EL2 is using AArch32, these are the HCR{AMO, IMO, FMO} control bits. If HCR_EL2.TGE or HCR.TGE is 1, these bits are treated as being 1 other than for a direct read.

- 规则小结如下：

若 SPSR_EL1.M[4] == 0，则决定 EL0 使用 AArch64, 否则 AArch32
若 SCR_EL3.RW == 1，则决定 EL2/EL1 是使用 AArch64，否则 AArch32
若 SCR_EL3.{EA, FIQ, IRQ} == 1，则所有相应的 SError\FIQ\IRQ 中断都被路由到 EL3
若 HCR_EL2.RW == 1，则决定 EL1 使用 AArch64，否则使用 AArch32

若 HCR_EL2.{AMO, IMO, FMO} == 1，则 EL1/EL0 所有对应的 SError\FIQ\IRQ 中断都被路由到 EL2,同时使能对应的虚拟中断 VSE,VI,VF
若 HCR_EL2.TGE == 1，那么会忽略 HCR_EL2.{AMO, IMO, FMO}的具体值，直接当成 1 处理，则 EL1/EL0 所有对应的 SError\FIQ\IRQ 中断都被路由到 EL2，同时禁止所有虚拟中断
注意： SCR_EL3.{EA, FIQ, IRQ}bit 的优先级高于 HCR_EL2.{AMO, IMO, FMO} bit 优先级，路由优先考虑 SCR_EL3

3.3.2 IRQ/FIQ/SError 路由流程图



第 4 章 ARMv8 寄存器

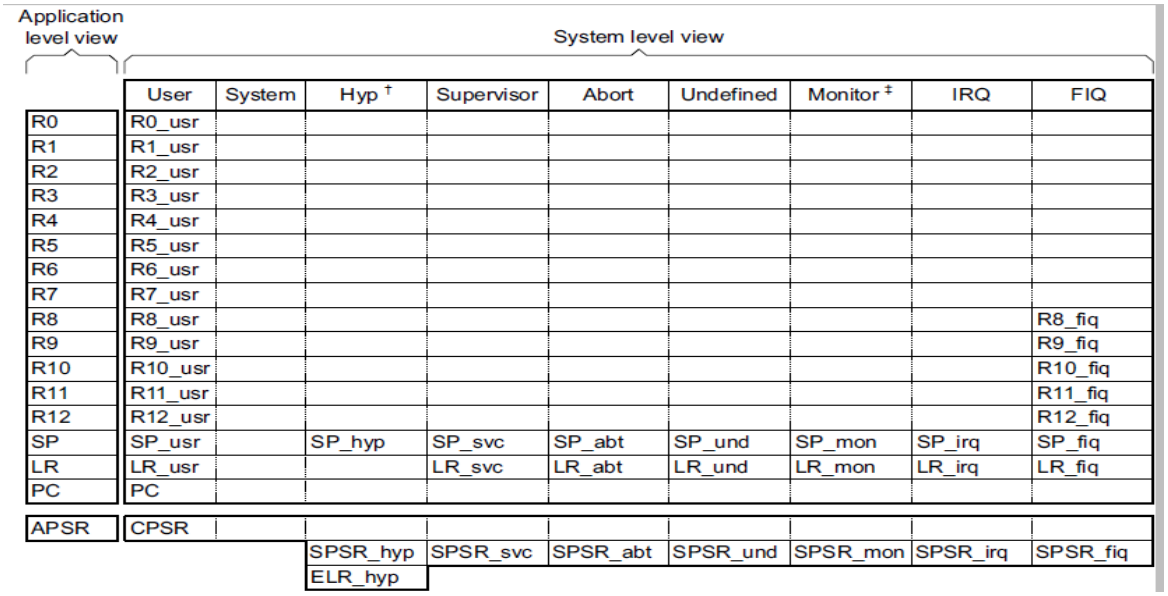
寄存器名称描述

位宽	分类		
32-bit	Wn（通用）	WZR（0 寄存器）	WSP（堆栈指针）
64-bit	Xn（通用）	XZR（0 寄存器）	SP（堆栈指针）

4.1 AArch32 重要寄存器

寄存器类型	Bit	描述
R0-R14	32bit	通用寄存器，但是 ARM 不建议使用有特殊功能的 R13，R14，R15 当做通用寄存器使用.
SP_x	32bit	通常称 R13 为堆栈指针，除了 User 和 Sys 模式外，其他各种模式下都有对应的 SP_x 寄存器：x ={ und/svc/abt/irq/fiq/hyp/mon}
LR_x	32bit	称 R14 为链接寄存器，除了 User 和 Sys 模式外，其他各种模式下都有对应的 SP_x 寄存器：x ={ und/svc/abt/svc/irq/fiq/mon}，用于保存程序返回链接信息地址，AArch32 环境下，也用于保存异常返回地址，也就说 LR 和 ELR 是公用一个，AArch64 下是独立的.
ELR_hyp	32bit	Hyp mode 下特有的异常链接寄存器，保存异常进入 Hyp mode 时的异常地址
PC	32bit	通常称 R15 为程序计算器 PC 指针，AArch32 中 PC 指向取指地址，是执行指令地址+8，AArch64 中 PC 读取时指向当前指令地址.
CPSR	32bit	记录当前 PE 的运行状态数据，CPSR.M[4:0]记录运行模式，AArch64 下使用 PSTATE 代替
APSR	32bit	应用程序状态寄存器，EL0 下可以使用 APSR 访问部分 PSTATE 值
SPSR_x	32bit	是 CPSR 的备份，除了 User 和 Sys 模式外，其他各种模式下都有对应的 SPSR_x 寄存器：x ={ und/svc/abt/irq/fiq/hpy/mon}，注意：这些模式只适用于 32bit 运行环境
HCR	32bit	EL2 特有，HCR.{TEG,AMO,IMO,FMO,RW}控制 EL0/EL1 的异常路由
SCR	32bit	EL3 特有，SCR.{EA,IRQ,FIQ,RW}控制 EL0/EL1/EL2 的异常路由，注意 EL3 始终不会路由
VBAR	32bit	保存任意异常进入非 Hyp mode & 非 Monitor mode 的跳转向量基地址
HVBAR	32bit	保存任意异常进入 Hyp mode 的跳转向量基地址
MVBAR	32bit	保存任意异常进入 Monitor mode 的跳转向量基地址
ESR_ELx	32bit	保存异常进入 ELx 时的异常综合信息，包含异常类型 EC 等，可以通过 EC 值判断异常 class
PSTATE		不是一个寄存器，是保存当前 PE 状态的一组寄存器统称，其中可访问寄存器有：PSTATE.{NZCV,DAIF,CurrentEL,SPSel}，属于 ARMv8 新增内容，主要用于 64bit 环境下

4.1.1 A32 状态下寄存器组织



Application level view	System level view								
	User	System	Hyp †	Supervisor	Abort	Undefined	Monitor ‡	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

- 所谓的 banked register 是指一个寄存器在不同模式下有对应不同的寄存器，比如 SP，在 abort 模式下是 SP_bat，在 Und 模式是 SP_und，在 iqr 模式下是 SP_irq 等，进入各种模式后会自动切换映射到各个模式下对应的寄存器.
- R0-R7 是所谓的非 banked register，R8-R14 是所谓的 banked register

4.1.1 T32 状态下寄存器组织

A32 使用 Rd/Rn 编码位宽 4 位	T32-32bit 使用 Rd/Rn 编码位宽 4 位	T32-16bit 使用 Rd/Rn 编码位宽 3 位
R0	R0	R0
R1	R1	R1
R2	R2	R2
R3	R3	R3
R4	R4	R4
R5	R5	R5
R6	R6	R6
R7	R7	R7
R8	R8	并不是说 T32-16bit 下没有 R8～R12，而是有限的指令才能访问到，16bit 指令的 Rd/Rn 编码位只有 3 位，所以 Rx 范围是 R0-R7
R9	R9	
R10	R10	
R11	R11	
R12	R12	
SP （R13）	SP （R13）	SP （R13）
LR （R14）	LR （R14） //M	LR （R14） //M
PC （R15）	PC （R15） //P	PC （R15） //P
CPSR	CPSR	CPSR

SPSR	SPSR	SPSR

4.2 AArch64 重要寄存器

寄存器类型	Bit	描述
X0-X30	64bit	通用寄存器，如果有需要可以当做 32bit 使用：W0-W30
LR (X30)	64bit	通常称 X30 为程序链接寄存器，保存跳转返回信息地址
SP_ELx	64bit	若 PSTATE.M[0] ==1，则每个 ELx 选择 SP_ELx，否则选择同一个 SP_EL0
ELR_ELx	64bit	异常链接寄存器，保存异常进入 ELx 的异常地址（x={0,1,2,3}）
PC	64bit	程序计数器，俗称 PC 指针，总是指向即将要执行的下一条指令
SPSR_ELx	32bit	寄存器，保存进入 ELx 的 PSTATE 状态信息
NZCV	32bit	允许访问的符号标志位
DIAF	32bit	中断使能位：D-Debug，I-IRQ，A-SError，F-FIQ ，逻辑 0 允许
CurrentEL	32bit	记录当前处于哪个 Exception level
SPSel	32bit	记录当前使用 SP_EL0 还是 SP_ELx，x= {1,2,3}
HCR_EL2	32bit	HCR_EL2.{TEG,AMO,IMO,FMO,RW}控制 EL0/EL1 的异常路由 逻辑 1 允许
SCR_EL3	32bit	SCR_EL3.{EA,IRQ,FIQ,RW}控制 EL0/EL1/EL2 的异常路由 逻辑 1 允许
ESR_ELx	32bit	保存异常进入 ELx 时的异常综合信息，包含异常类型 EC 等.
VBAR_ELx	64bit	保存任意异常进入 ELx 的跳转向量基地址 x={0,1,2,3}
PSTATE		不是一个寄存器，是保存当前 PE 状态的一组寄存器统称，其中可访问寄存器有：PSTATE.{NZCV,DAIF,CurrentEL,SPSel}，属于 ARMv8 新增内容，64bit 下代替 CPSR

4.3 64、32 位寄存器的映射关系

64-bit	32-bit		64-bit	32-bit
X0	R0		X20	LR_adt
X1	R1		X21	SP_abt
X2	R2		X22	LR_und
X3	R3		X23	SP_und
X4	R4		X24	R8_fiq
X5	R5		X25	R9_fiq
X6	R6		X26	R10_fiq
X7	R7		X27	R11_fiq
X8	R8_usr	64-bit OS Runing AArch32 App	X28	R12_fiq
X9	R9_usr		X29	SP_fiq
X10	R10_usr		X30(LR)	LR_fiq
X11	R11_usr		SCR_EL3	SCR
X12	R12_usr		HCR_EL2	HCR
X13	SP_usr		VBAR_EL1	VBAR
X14	LR_usr		VBAR_EL2	HVBAR
X15	SP_hyp		VBAR_EL3	MVBAR

X16	LR_irq		**ESR_EL1**	DFSR
X17	SP_irq		**ESR_EL2**	HSR
X18	LR_svc			
X19	SP_svc			

第 5 章 异常模型

5.1 异常类型描述

5.1.1 AArch32 异常类型

异常类型	描述	默认捕获模式	向量地址偏移
Undefined Instruction	未定义指令	Und mode	0x04
Supervisor Call	SVC 调用	Svc mode	0x08
Hypervisor Call	HVC 调用	Hyp mode	0x08
Secure Monitor Call	SMC 调用	Mon mode	0x08
Prefetch abort	预取指令终止	Abt mode	0x0c
Data abort	数据终止	Abt mode	0x10
IRQ interrupt	IRQ 中断	IRQ mode	0x18
FIQ interrupt	FIQ 中断	FIQ mode	0x1c
Hyp Trap exception	Hyp 捕获异常	Hyp mode	0x14
Monitor Trap exception	Mon 捕获异常	Mon mode	0x04

5.1.2 AArch64 异常类型

可分为同步异常 & 异步异常两大类,如下表描述：

Synchronous(同步异常)	
异常类型	描述
Undefined Instruction	未定义指令异常
Illegal Execution State	非常执行状态异常
System Call	系统调用指令异常（SVC/HVC/SMC）
Misaligned PC/SP	PC/SP 未对齐异常
Instruction Abort	指令终止异常
Data Abort	数据终止异常
Debug exception	软件断点指令/断点/观察点/向量捕获/软件单步 等 Debug 异常

Asynchronous(异步异常)	
类型	描述

SError or vSError	系统错误类型，包括外部数据终止
IRQ or vIRQ	外部中断 or 虚拟外部中断
FIQ or vFIQ	快速中断 or 虚拟快速中断

异常进入满足以下条件	向量地址偏移表			
	Synchronous （同步异常）	IRQ || vIRQ	FIQ || vFIQ	SError || vSError
SP => SP_EL0 && 从 Current EL 来	0x000	0x080	0x100	0x180
SP => SP_ELx && 从 Current EL 来	0x200	0x280	0x300	0x380
64bit => 64bit && 从 Low level EL 来	0x400	0x480	0x500	0x580
32bit => 64bit && 从 Low level EL 来	0x600	0x680	0x700	0x780

- SP => SP_EL0, 表示使用 SP_EL0 堆栈指针，由 PSTATE.SP == 0 决定, PSTATE.SP == 1 则 SP_ELx；
- 32bit => 64bit 是指发生异常时 PE 从 AArch32 切换到 AArch64 的情况；

5.2 异常处理逻辑

5.2.1 寄存器操作

流程	Note
AArch32 State	
1、 PE 根据异常类型跳转到对应的异常模式 x, x = {und/svc/abt/irq/fiq/hyp/mon}	PE 跳转到哪一种模式通常由路由关系决定
2、保存异常返回地址到 LR_x，用于异常返回用	LR 也是对应模式的 R[14]_x 寄存器，32 位系统下 LR 和 ELR 是同一个寄存器，而 64 位是独立的
3、备份 PSTATE 数据到 SPSR_x	异常返回时需要从 SPSR_x 恢复 PSTATE
4、PSTATE 操作： PSTATE.M[4:0]设置为异常模式 x PSTATE.{A,I,F} = 1 PSTATE.T = 1，强制进入 A32 模式 PSTATE.IT[7：2] = “00000”	PSTATE.M[4]只是对 32 位系统有效，64 为下是保留的，因为 64 位下没有各种 mode 的概念. 异常处理都要切换到 ARM 下进行； 进入异常时需要暂时关闭 A,I,F 中断；
5、据异常模式 x 的向量偏移跳转到进入异常处理	各个 mode 有对应的 Vector base addr + offset
AArch64 state	
1、 保存 PSTATE 数据到 SPSR_ELx,(x = 1,2,3)	异常返回时需要从 SPSR_ELx 中恢复 PSTATE
2、保存异常进入地址到 ELR_ELx，同步异常（und/abt 等）是当前地址，而 异步异常（irq/fiq 等）是下一条指令地址	64 位架构 LR 和 ELR 是独立分开的，这点和 32 位架构有所差别
3、保存异常原因信息到 ESR_ELx	ESR_ELx.EC 代表 Exception Class，关注这个 bit
4、PE 根据目标 EL 的异常向量表中定义的异常地	跳转到哪个 EL 使用哪个向量偏移地址又路由关

址强制跳转到异常处理程序	系决定
5、堆栈指针 SP 的使用由目标 EL 决定	(SPSR_ELx.M[0] == 1) ? h (ELx) : t (EL0)

5.2.2 路由控制

Execution State	异步异常（中断）	路由控制位、优先级排列. 1 允许 0 禁止
AArch32	Asynchronous Data Abort (异步数据终止)	SCR.EA HCR.TGE HCR.AMO
	IRQ or vIRQ	SCR.IRQ HCR.TGE HCR.IMO
	FIQ or vFIQ	SCR.FIQ HCR.TGE HCR.FMO
AArch64	SError or vSError	SCR_EL3.EA HCR_EL2.TGE HCR_EL2.AMO
	IRQ or vIRQ	SCR_EL3.IRQ HCR_EL2.TGE HCR_EL2.IMO
	FIQ or vFIQ	SCR_EL3.FIQ HCR_EL2.TGE HCR_EL2.FMO

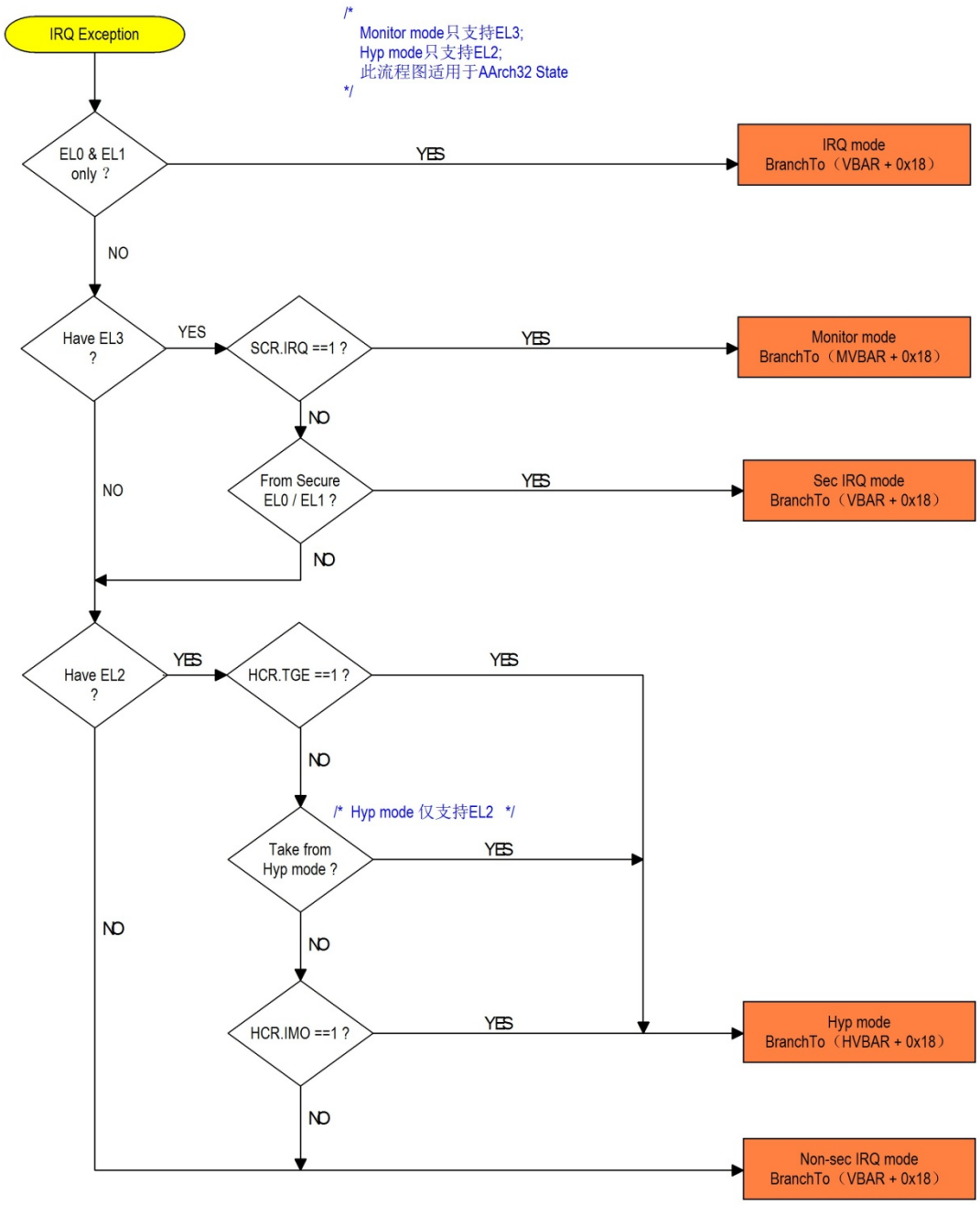
• 若 HCR_EL2.TGE ==1 所有的虚拟中断将被禁止,HCR.{AMO,IMO,FMO} HCR_EL2.{AMO,IMO,FMO}被当成 1 处理.

5.3 流程图对比

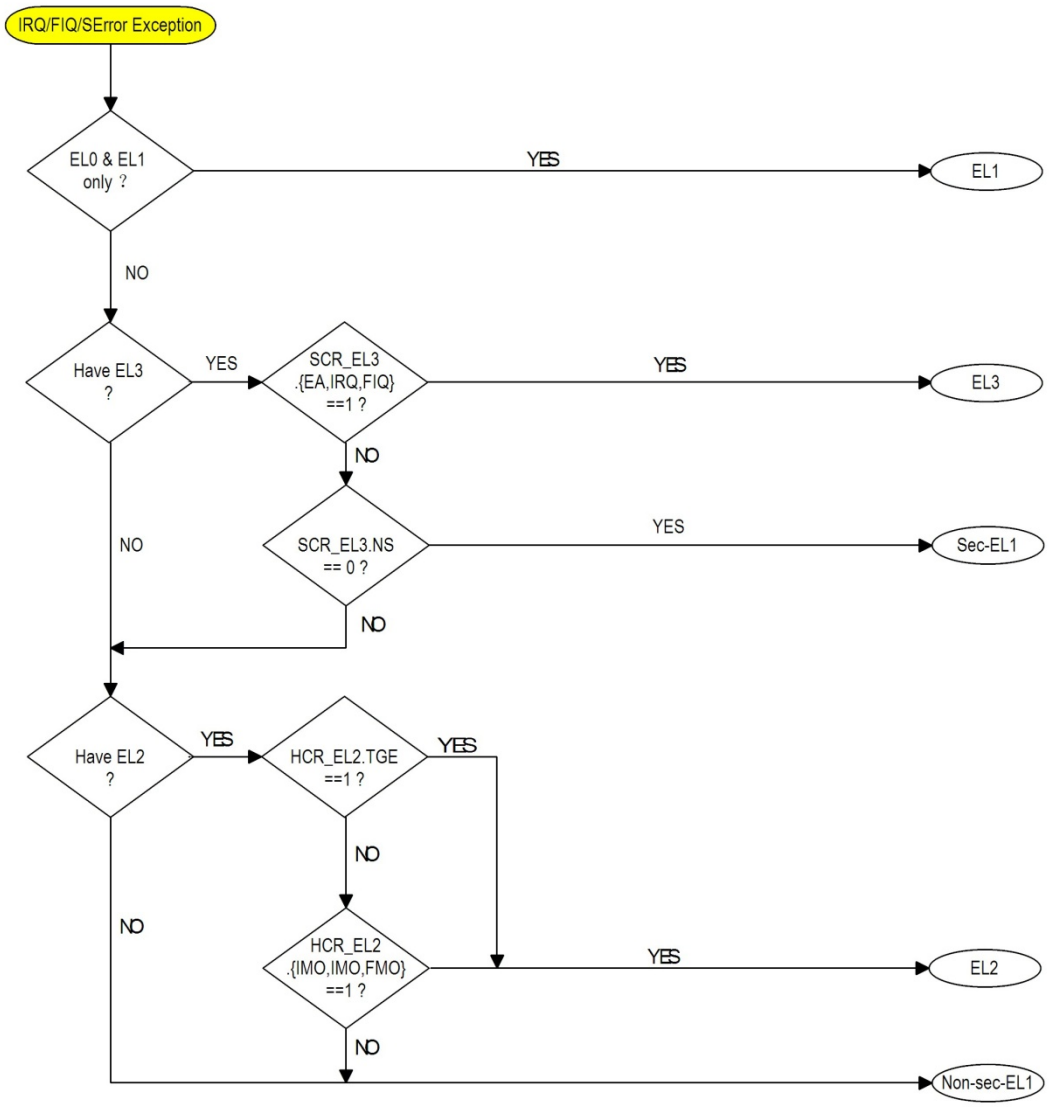
AArch32、AArch64 架构下 IRQ 和 Data Abort 异常处理流程图对比.

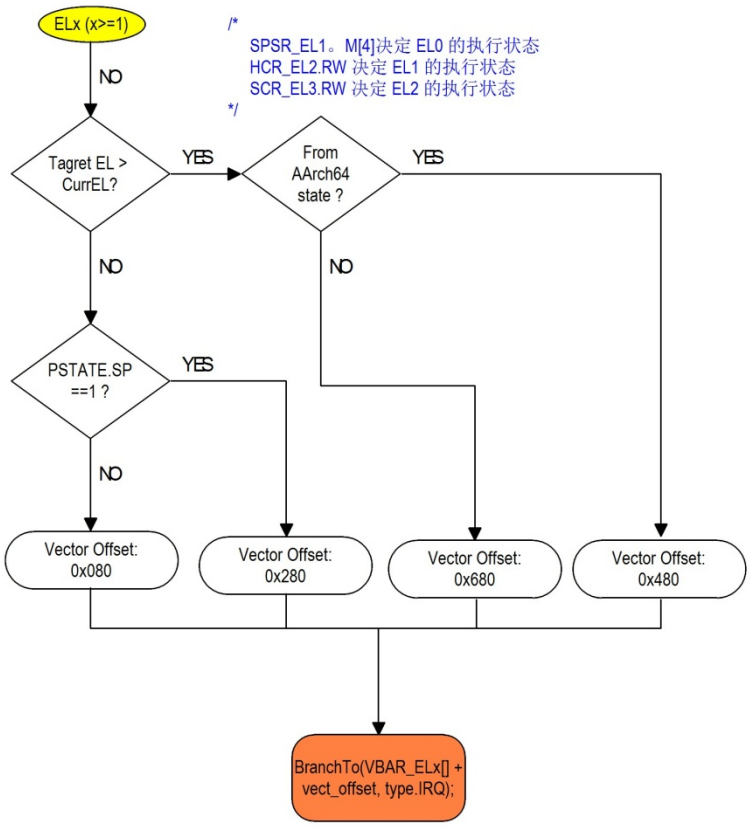
5.3.1 IRQ 流程图

5.3.1.1 AArch32



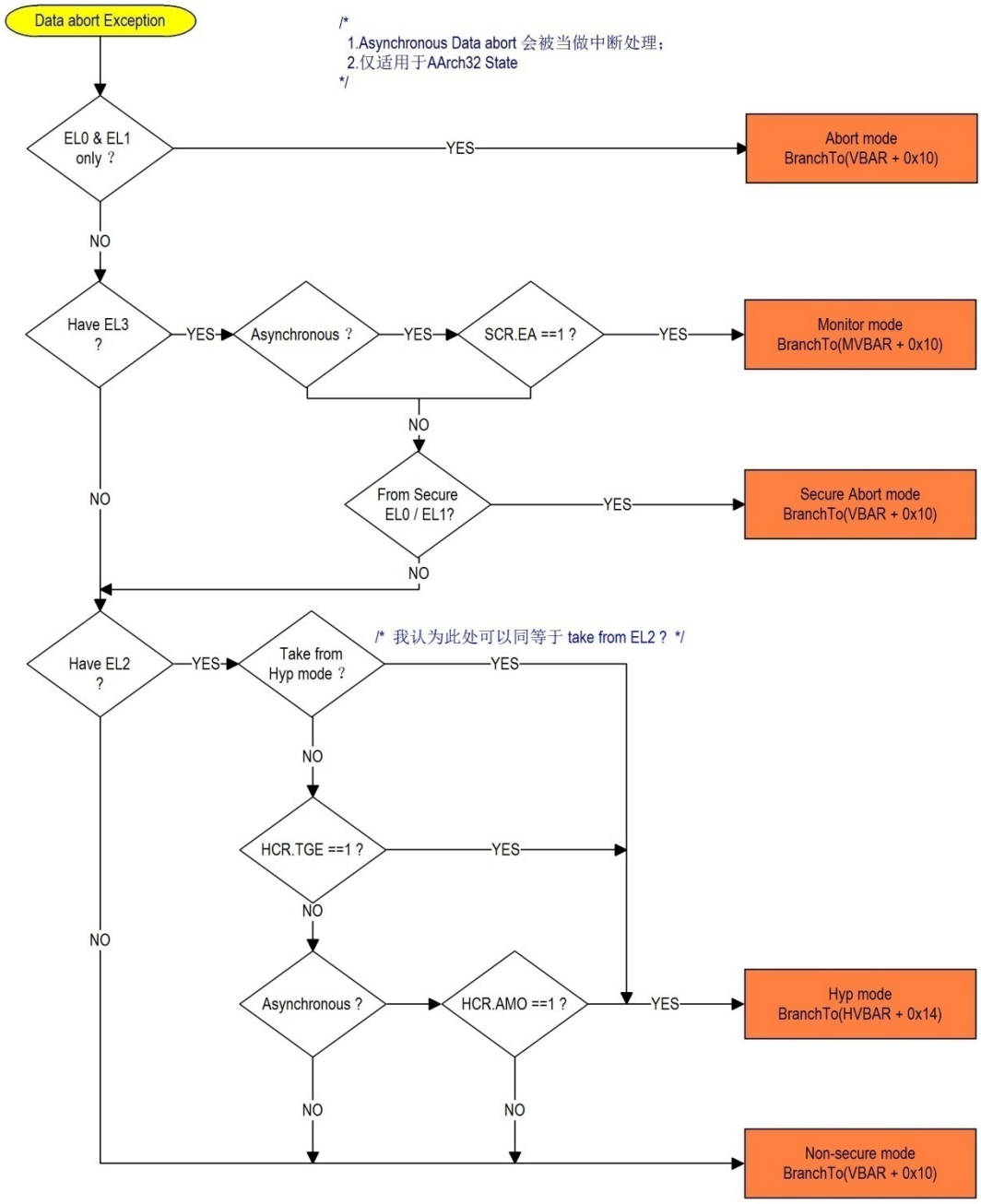
5.3.1.2 AArch64



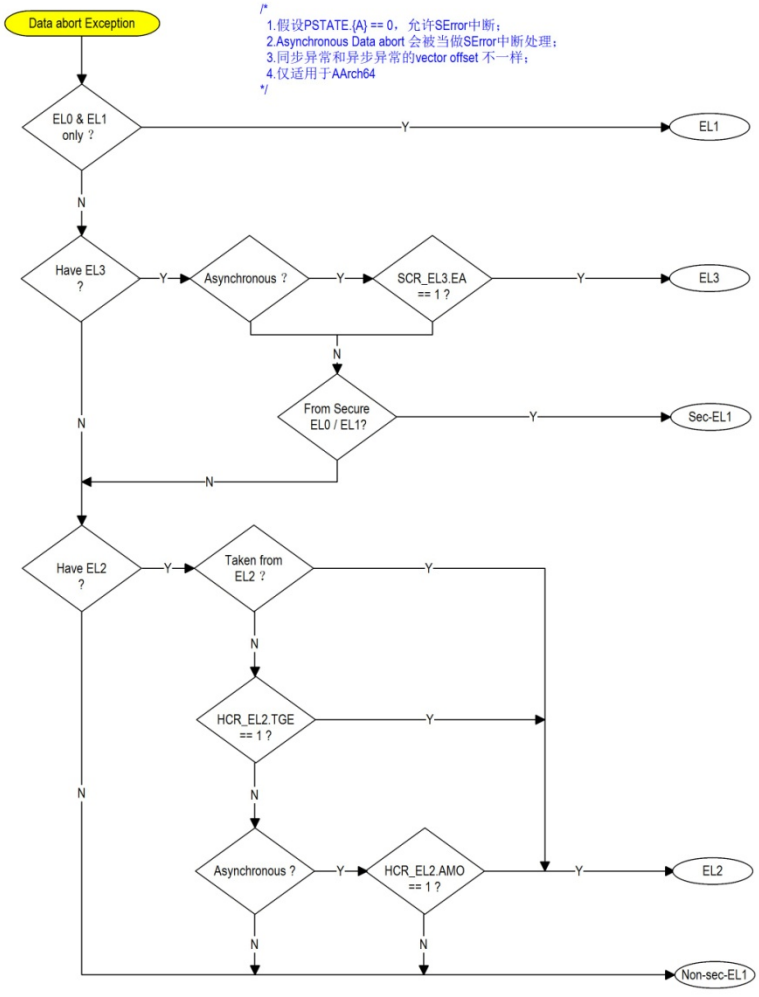

ELx (x>=1)
/*
SPSR_EL1。M[4]决定 EL0 的执行状态
HCR_EL2.RW 决定 EL1 的执行状态
SCR_EL3.RW 决定 EL2 的执行状态
*/
NO
Tagret EL > CurrEL?
YES
From AArch64 state ?
YES
NO
NO
PSTATE.SP ==1 ?
YES
NO
Vector Offset: 0x080
Vector Offset: 0x280
Vector Offset: 0x680
Vector Offset: 0x480
BranchTo(VBAR_ELx[] + vect_offset, type.IRQ);

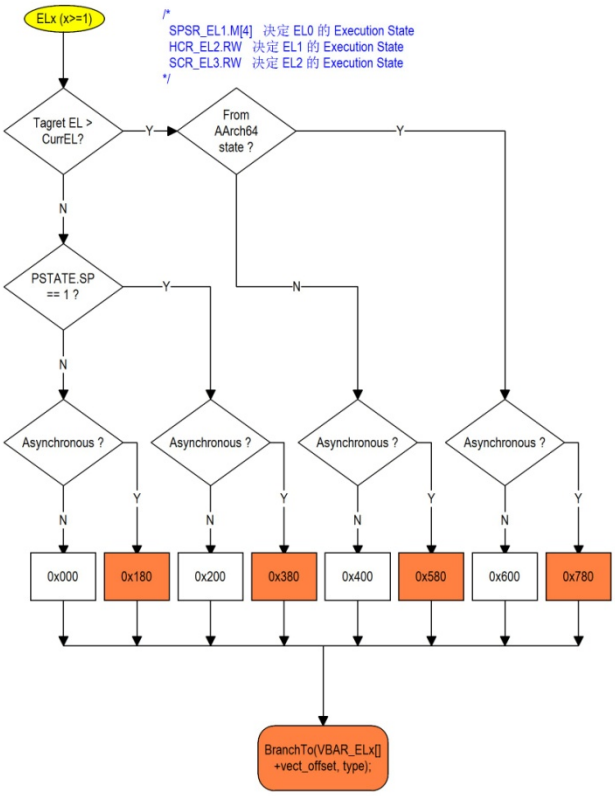
5.3.2 Data Abort 流程图

5.3.2.1 AArch32



5.3.2.2 AArch64


Data abort Exception
/*
1.假设PSTATE.{A} == 0，允许SError中断；
2.Asynchronous Data abort 会被当做SError中断处理；
3.同步异常和异步异常的vector offset 不一样；
4.仅适用于AArch64
*/
EL0 & EL1 only ?
Y
EL1
N
Have EL3 ?
Y
Asynchronous ?
Y
SCR_EL3.EA == 1 ?
Y
EL3
N
From Secure EL0 / EL1?
Y
Sec-EL1
N
N
Have EL2 ?
Y
Taken from EL2 ?
Y
N
HCR_EL2.TGE == 1 ?
Y
N
N
Asynchronous ?
Y
HCR_EL2.AMO == 1 ?
Y
EL2
N
N
Non-sec-EL1


ELx (x>=1)
/*
SPSR_EL1.M[4] 决定 EL0 的 Execution State
HCR_EL2.RW 决定 EL1 的 Execution State
SCR_EL3.RW 决定 EL2 的 Execution State
*/
Tagret EL > CurrEL?
Y
From AArch64 state ?
Y
N
PSTATE.SP == 1 ?
Y
N
N
Asynchronous ?
Asynchronous ?
Asynchronous ?
Asynchronous ?
N
Y
N
Y
N
Y
N
Y
0x000
0x180
0x200
0x380
0x400
0x580
0x600
0x780
BranchTo(VBAR_ELx[] +vect_offset, type);

5.4 源代码异常入口

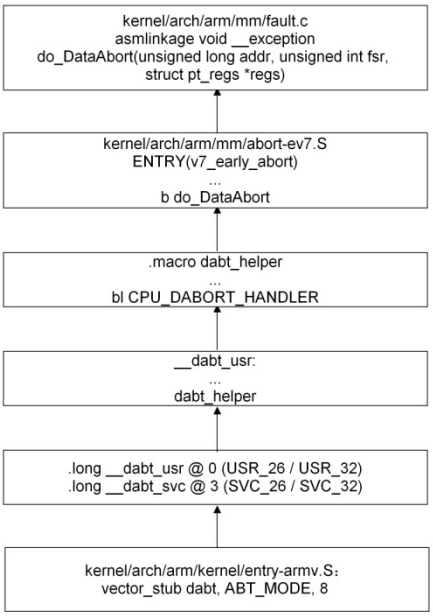
5.4.1 C 函数入口

异常类型	AArch32 State		AArch64 State	
	所在文件	C 函数	所在文件	C 函数
Und	arm/kernel/traps.c	do_undefinstr	Arm64/kernel/traps.c	do_undefinstr
Data Abort	arm/mm/fault.c	do_DataAbort	arm64/mm/fault.c	do_mem_abort
IRQ	arm/kernel/irq.c	asm_do_IRQ	arm64/kernel/irq.c	handle_IRQ
FIQ				
System Call				

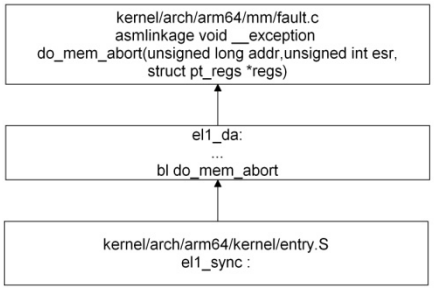
5.4.2 上报流程图

例举 Data Abort 和 IRQ 中断的入口流程图

5.4.2.1 Data Abort 上报

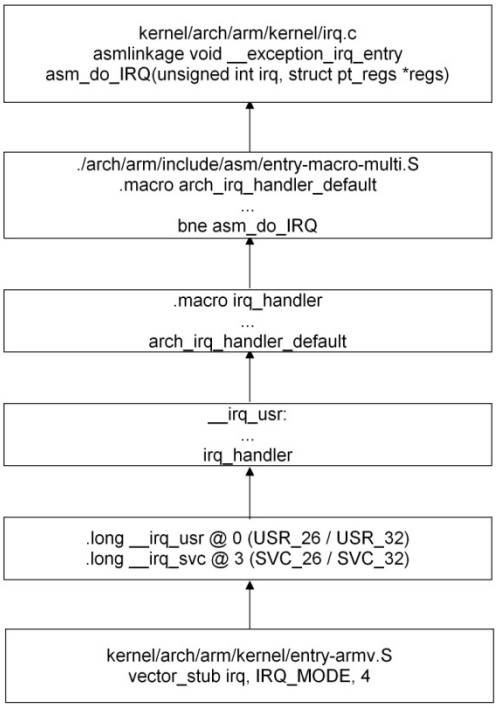


AArch32 Data Abort 函数入口

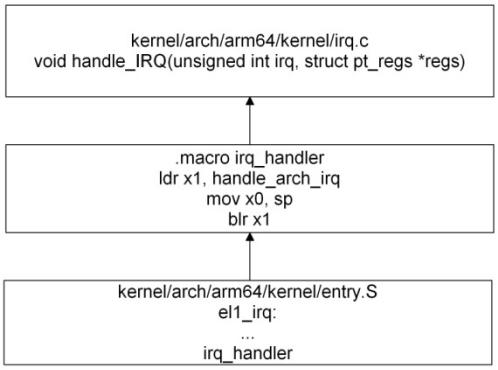


AArch64 Data Abort 函数入口

5.4.2.2 IRQ 上报



AArch32 IRQ 函数入口



AArch64 IRQ 函数入口

5.4.3 异常进入压栈准备

分析 64 位 kernel_entry 压栈代码逻辑（代码路径：kernel/arch/arm64/kernel/entry.S）

• sp 指向 #S_LR - #S_FRAME_SIZE 位置	#S_FRAME_SIZE 是 pt_regs 结构图的 size
• 依次把 x28-x29 … x0-x1 成对压入栈内	每压入一对寄存器,sp 指针就移动 -16 =((64/8)*2）字节长度，栈是向地址减少方向增长的.
• 保存 sp+#S_FRAME_SIZE 数据到 x21	add x21, sp, #SP_FRAME_SIZE
• 保存 elr_el1 到 x22	mrs x22, elr_el1
• 保存 spsr_el1 到 x23	mrs x23, spsr_el1
• 把 lr、x21 写入 sp+#S_LR 地址内存	保存 lr 和 x21 的数据到指定栈内存位置
• 把 x22、x23 写入 sp+#S_PC 地址内存	保存 elr,spsr 数据到指定栈内存位置

5.4.4 栈布局

第 6 章 ARMv8 指令集

6.1 概况

- A64 指令集
- A32 & T32 指令集
- 指令编码

6.1.1 指令基本格式

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

- 其中尖括号是必须的，花括号是可选的
- □ A32： Rd => {R0 - R14}
- □ A64： Rd =>Xt => {X0 - X30}

标识符	Note
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，在编码中占 4bit，0b0000 -0b1110
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
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 R0-R7，A64 指令可以选择 X0-X30 or W0-W30
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）

6.1.2 指令分类

类型	Note
□ 跳转指令	条件跳转、无条件跳转（#imm、register）指令
□ 异常产生指令	系统调用类指令（SVC、HVC、SMC）
□ 系统寄存器指令	读写系统寄存器，如 ：MRS、MSR 指令 可操作 PSTATE 的位段寄存器
□ 数据处理指令	包括各种算数运算、逻辑运算、位操作、移位(shift)指令
□ load/store 内存访问指令	load/store {批量寄存器、单个寄存器、一对寄存器、非-暂存、非特权、独占}以及 load-Acquire、store-Release 指令 （A64 没有 LDM/STM 指令）
□ 协处理指令	A64 没有协处理器指令

6.2 A64 指令集

□ A64 指令编码宽度固定 32bit
□ 31 个（X0-X30）个 64bit 通用用途寄存器（用作 32bit 时是 W0-W30），寄存器名使用 5bit 编码
□ PC 指针不能作为数据处理指或 load 指令的目的寄存器，X30 通常用作 LR
□ 移除了批量加载寄存器指令 LDM/STM, PUSH/POP, 使用 STP/LDP 一对加载寄存器指令代替

□ 增加支持未对齐的 load/store 指令立即数偏移寻址，提供非-暂存 LDNP/STNP 指令，不需要 hold 数据到 cache 中
□ 没有提供访问 CPSR 的单一寄存器，但是提供访问 PSTATE 的状态域寄存器
□ 相比 A32 少了很多条件执行指令，只有条件跳转和少数数据处理这类指令才有条件执行.
□ 支持 48bit 虚拟寻址空间
□ 大部分 A64 指令都有 32/64 位两种形式
□ A64 没有协处理器的概念

6.2.1 指令助记符

整型	
W/R	32bit 整数
X	64bit 整数
加载/存储、符号-0 扩展	
B	无符号 8bit 字节
SB	带符号 8bit 字节
H	无符号 16bit 半字
SH	带符号 16bit 半字
W	无符号 32bit 字
SW	带符号 32bit 字
P	Pair（一对）
寄存器宽度改变	
H	高位（dst gets top half）
N	有限位（dst < src）
L	Long （dst > src）
W	Wide（dst==src1,src1>src2）？

6.2.2 指令条件码

编码	助记符	描述	标记
0000	EQ	结果相等为 1	Z==1
0001	NE	结果不等为 0	Z==0
0010	HS/CS	无符号高或者相同进位，发生进位为 1	C==1
0011	LO/CC	无符号低清零，发生借位为 0	C==0
0100	MI	负数为 1	N==1
0101	PL	非负数 0	N==0
0110	VS	有符号溢出为 1	V==1
0111	VC	没用溢出为 0	V==0
1000	HI	无符号 >	C==1 && Z==0
1001	LS	无符号 <=	!(C==1 && Z==0)
1010	GE	带符号 >=	N==V

1011	LT	带符号 <	N!=V
1100	GT	带符号 >	Z==0 && N==V
1101	LE	带符号 <=	!(Z==0 && N==V)
1110	AL	无条件执行	Any
1111	NV		

6.2.3 跳转指令

6.2.3.1 条件跳转

B.cond	cond 为真跳转
CBNZ	CBNZ X1, label //如果 X1!= 0 则跳转到 label
CBZ	CBZ X1, label //如果 X1== 0 则跳转到 label
TBNZ	TBNZ X1, #3 label //若 X1[3]!=0,则跳转到 label
TBZ	TBZ X1, #3 label //若 X1[3]==0,则跳转到 label

6.2.3.2 绝对跳转

B	绝对跳转
BL	绝对跳转 #imm，返回地址保存到 LR（X30）
BLR	绝对跳转 reg，返回地址保存到 LR（X30）
BR	跳转到 reg 内容地址,
RET	子程序返回指令，返回地址默认保存在 LR（X30）

6.2.4 异常产生和返回指令

SVC	SVC 系统调用，目标异常等级为 EL1
HVC	HVC 系统调用，目标异常等级为 EL2
SMC	SMC 系统调用，目标异常等级为 EL3
ERET	异常返回，使用当前的 SPSR_ELx 和 ELR_ELx

6.2.5 系统寄存器指令

MRS	R <- S: 通用寄存器 <= 系统寄存器
MSR	S <- R: 系统寄存器 <= 通用寄存器

6.2.6 数据处理指令

数据处理指令类型					
算数运算	逻辑运算	数据传输	地址生成	位段移动	移位运算
ADDS	ANDS	MOV	ADRP	BFM	ASR
SUBS	EOR	MOVZ	ADR	SBFM	LSL
CMP	ORR	MOVK		UBFM	LSR
SBC	MOVI			BFI	ROR
RSB	TST			BFXIL	
RSC				SBFIZ	
CMN				SBFX	
MADD				UBFIZ	
MSUB					
MUL					
SMADDL					
SDIV					
UDIV					

6.2.6.1 算术运算指令

ADDS	加法指令，若 S 存在，则更新条件位 flag
ADCS	带进位的加法，若 S 存在，则更新条件位 flag
SUBS	减法指令，若 S 存在，则更新条件位 flag
SBC	将操作数1减去操作数2，再减去标志位 C 的取反值，结果送到目的寄存器 Xt/Wt
RSB	逆向减法，操作数2 －操作数1，结果Rd
RSC	带借位的逆向减法指令，将操作数2减去操作数1，再减去标志位 C 的取反值，结果送目标寄存器 Xt/Wt
CMP	比较相等指令
CMN	比较不等指令
NEG	取负数运算，NEG X1，X2 // X1 = X2 按位取反+1（负数=正数补码+1）
MADD	乘加运算
MSUB	乘减运算
MUL	乘法运算
SMADDL	有符号乘加运算
SDIV	有符号除法运算
UDIV	无符号除法运算

6.2.6.2 逻辑运算指令

ANDS	按位与运算，如果 S 存在，则更新条件位标记
EOR	按位异或运算
ORR	按位或运算
TST	例如：TST W0, #0X40 ; 指令用来测试 W0[3]是否为 1,相当于：AND WZR,W0, #0X40

6.2.6.3 数据传输指令

MOV	赋值运算指令
MOVZ	赋值#uimm16 到目标寄存器 Xd
MOVN	赋值#uimm16 到目标寄存器 Xd，再取反
MOVK	赋值#uimm16 到目标寄存器 Xd，保存其它 bit 不变

6.2.6.4 地址生成指令

ADRP	base = PC[11:0]=ZERO(12); Xd = base + label;
ADR	Xd = PC + label

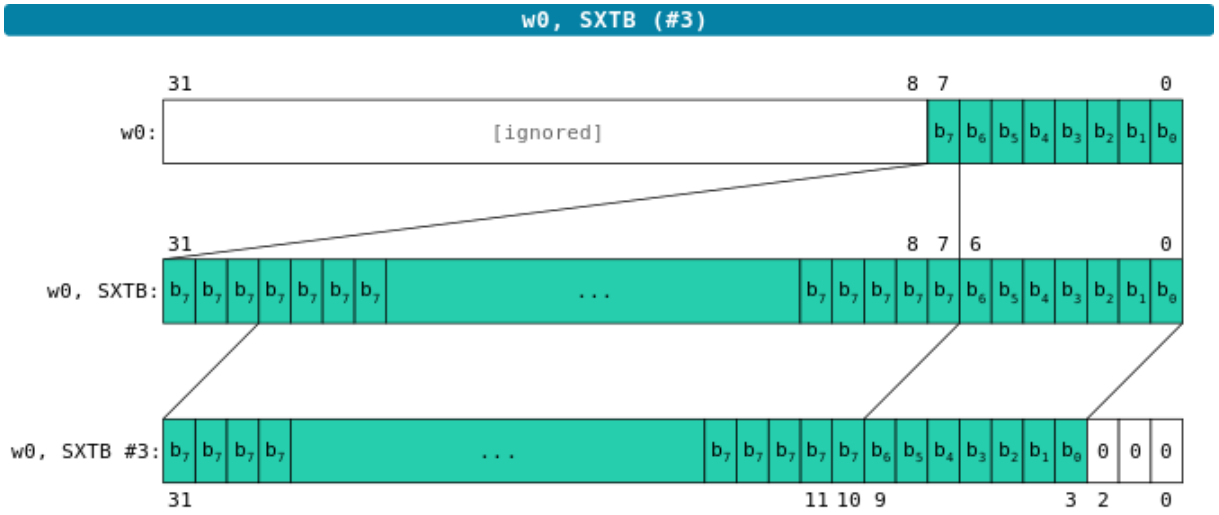
6.2.6.5 位段移动指令

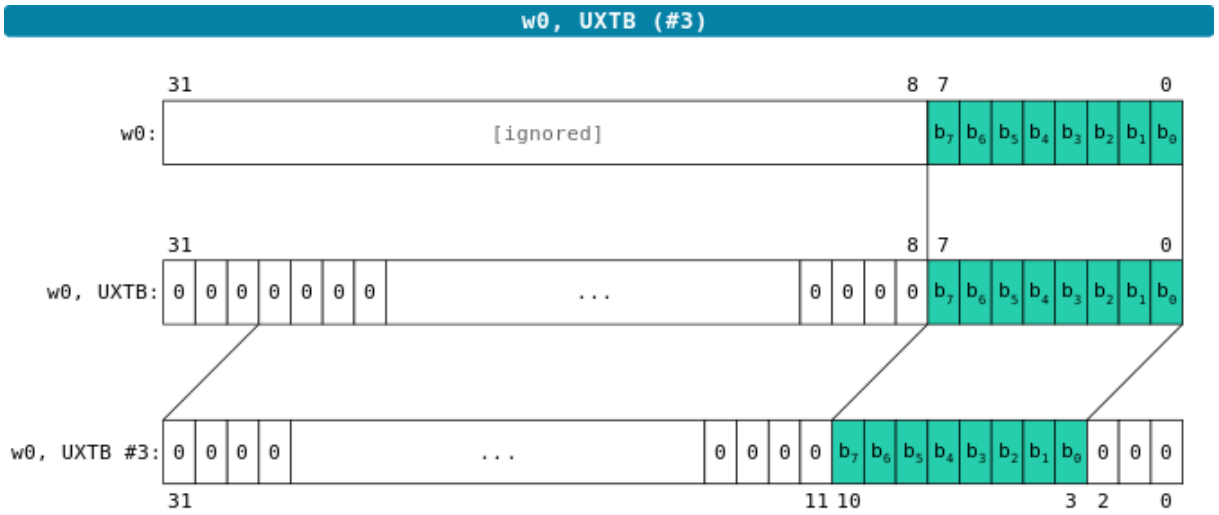
BFM	BFM Wd, Wn, #r, #s if s>=r then Wd<s-r:0> = Wn<s:r>, else Wd<32+s-r,32-r> = Wn<s:0>.
SBFM	
UBFM	
BFI	
BFXIL	
SBFIZ	
SBFX	
UBFX	
UBFZ	

6.2.6.6 移位运算指令

ASR	算术右移 >> （结果带符号）
LSL	逻辑左移 <<
LSR	逻辑右移 >>
ROR	循环右移：头尾相连

SXTB	字节、半字、字符号/0 扩展移位运算 关于 SXTB #imm 和 UXTB #imm 的用法可以使用以下图解描述：
SXTH	
SXTW	
UXTB	
UXTH	





6.2.7 Load/Store 指令

对齐偏移	非对齐偏移	PC-相对寻址	访问一对	非暂存	非特权	独占	Acquire Release
LDR	LDUR	LDR	LDP	LDNP	LDTR	LDXR	LDAR
LDRB	LDURB	LDRSW	LDRSW	STNP	LDTRB	LDXRB	LDARB
LDRSB	LDURSB		STP		LDTRSB	LDXRH	LDARH
LDRH	LDURH				LDTRH	LDXP	STLR
LDRSH	LDURSH				LDTRSH	STXR	STLRB
LDRSW	LDURSW				LDTRSW	STXRB	STLRH

STR	STUR				STTR	STXRH	LDAXR
STRB	STURB				STTRB	STXP	LDAXRB
STRH	STURH				STTRH		LDAXRH
							LDAXP
							STLXR
							STLXRB
							STLXRH
							STLXP

6.2.7.1 寻址方式

类型	立即数偏移	寄存器偏移	扩展寄存器偏移
基址寄存器（无偏移）	{ base{, #0 } }		
基址寄存器（+ 偏移）	{ base{, #imm } }	{ base, Xm{, LSL #imm } }	[base, Wm, (S|U)XTW {#imm}]
Pre-indexed（事先更新）	[base, #imm]!		
Post-indexed（事后更新）	[base, #imm]	{ base }, Xm	
PC-相对寻址	label		

6.2.7.2 Load/Store (Scaled Offset)

支持的寻址方式
对齐的，无符号#imm12 偏移，不支持 pre-/post-index 操作
非对齐，带符号#imm9 偏移，支持 pre-/post-index 操作
对齐 or 非对齐的 64bit 寄存器偏移
对齐 or 非对齐的 32bit 寄存器偏移

Zero-Extend / Sign-Extend	
0 扩展	从 Memory 读取一个无符号 32 位 Wn 数据写到一个 64 位 Xt 寄存器中，Wn 数据被存储到 Xt[31:0]，Xt[63:32] 使用 0 代替
符号扩展	从 Memory 读取一个有符号 32 位 Wn 数据写到一个 64 位 Xt 寄存器中，Wn 数据被存储到 Xt[31:0]，Xt[63:32] 使用 Wn 的符号位值（Wn[31]）代替

LDR	从 Memory 地址 addr 中读取双字/字节/半字/字数据到目标寄存器 Xt/Wt 中 带”S”表示需要符号扩展.
LDRB	
LDRSB	
LDRH	
LDRSH	

LDRSW	
STR	把 Xn/Wn 中的双字/字节/半字数据写入到 Memory 地址 addr 中
STRB	
STRH	

6.2.7.3 Load/Store (Unscaled Offset)

□所谓 Scaled 和 Unscaled 其实就是可以见到理解为对齐和非对齐,本质就是是否乘以一个常量,因为 scaled 的总是可以乘以一个常量来达到对齐，而 Unscaled 就不需要，是多少就多少，更符合人类自然的理解

支持的寻址方式
□ 非对齐的，有符号#simm9 偏移，不支持 pre-/post-index 操作

LDUR	从 Memory 地址 addr 中读取双字/字节/半字/字数据到目标寄存器 Xt/Wt 中 带”S”表示需要符号扩展. 立即数偏移 #simm9 = { -256 ~ +256 } 的任意整数，不需要对齐规则.
LDURB	
LDURSB	
LDURH	
LDURSH	
LDURSW	
STUR	把 Xn/Wn 中的双字/字节/半字数据写入到 Memory 地址 addr 中 立即数偏移 #simm9 = { -256 ~ +256 } 的任意整数，不需要对齐规则.
STURB	
STURH	

6.2.7.4 Load/Store PC-relative（PC 相对寻址）

支持的寻址方式
• 不支持 pre-/post-index 操作

LDR	从 Memory 地址 addr 中读取双字/字数据到目标寄存器 Xt/Wt 中 带”S”表示需要符号扩展.
LDRSW	

6.2.7.5 Load/Store Pair(一对)

支持的寻址方式
• 对齐的，有符号#simm7 偏移，支持 pre-/post-index 操作

LDP	从 Memory 地址 addr 处读取两个双字/字数据到目标寄存器 Xt1，Xt2 带”S”表示需要符号扩展.
LDRSW	
STP	把 Xt1，Xt2 两个双字/字数据写到 Memory 地址 addr 中

6.2.7.6 Load/Store Non-temporal(非暂存) Pair

□ 所谓 Non-temporal 就是就是用于你确定知道该地址只加载一次，不需要触发缓存，避免数据被刷新，优化性能，其它指令都默认会写 Cache

支持的寻址方式
• 对齐的，有符号#simm7 偏移，不支持 pre-/post-index 操作

LDNP	从 Memory 地址 addr 处读取两个双字/字数据到目标寄存器 Xt1,Xt2, 标注非暂存访问,不更新 cache 带”S”表示需要符号扩展.
STNP	把 Xt1，Xt2 两个双字/字数据写到 Memory 地址 addr 中，标注非暂存访问，不更新 cache

6.2.7.7 Load/Store Unprivileged（非特权）

□ 所谓 Unprivileged 就是说 EL0/EL1 的内存有不同的权限控制，这条指令以 EL0 的权限存取，用于模拟 EL0 的行为，该指令应用于 EL1 和 EL0 之间的交互.

支持的寻址方式
□ 非对齐的，有符号#simm9 偏移，不支持 pre-/post-index 操作

LDTR LDTRB LDTRSB LDTRH LDTRSH LDTRSW	从 Memory 地址 addr 中读取双字/字节/半字/字数据到目标寄存器 Xt/Wt 中， 当执行在 EL1 的时候使用 EL0 的权限 带”S”表示需要符号扩展
STTR STTRB STTRH	把 Xn/Wn 中的双字/字节/半字数据写入到 Memory 地址 addr 中， 当执行在 EL1 的时候使用 EL0 的权限

6.2.7.8 Load/Store Exclusive（独占）

□ 在多核 CPU 下，对一个地址的访问可能引起冲突，这个指令解决了冲突，保证原子性(所谓原子操作简单理解就是不能被中断的操作),是解决多个 CPU 访问同一内存地址导致冲突的一种机制。比如 2 个 CPU 同时写，其中一条的 Ws 就会返回失败值。通常用于锁，比如 spinlock，可以参考代码：arch/arm64/include/asm/spinlock.h

支持的寻址方式
• 无偏移基址寄存器，不支持 pre-/post-index 操作

LDXR LDXRB	从 Memory 地址 addr 中读取双字/字节/半字数据到目标寄存器 Xt/Wt 中， 标记物理地址是独占访问的

LDXRH	
LDXP	从 Memory 地址 addr 中读取一对双字数据到目标寄存器 Xt1，Xt2 中，标记物理地址是独占访问的
STXR	把 Xn/Wn 中的双字/字节/半字数据写入到 Memory 地址 addr 中， 返回是否独占访问成功状态（Ws）
STXRB	
STXRH	
STXP	把 Xt1，Xt2 一对双字字数据写入到 Memory 地址 addr 中，返回是否独占访问成功状态

6.2.7.9 Load-Acquire/Store-Release

Load-Acquire Acquire 的语义是读操作	相当于半个 DMB 指令，只管读内存操作
Store-Release Release 的语义是写操作	相当于半个 DMB 指令，只管写内存操作

支持的寻址方式
□ 无偏移基址寄存器，不支持 pre-/post-index 操作

Non-exclusive(非独占)	
LDAR	从 Memory 地址 addr 中读取一个双字/字节/半字数据到目标寄存器 Xt/Wt 中， 标记物理地址为非独占访问
LDARB	
LDARH	
STLR	把一个双字/字节/半字数据 Xt/Wt 写到 Memory 地址 addr 中， 返回是否独占访问成功状态
STLRB	
STLRH	
Exclusive(独占)	
LDAXR	从 Memory 地址 addr 中读取一个双字/字节/半字数据到目标寄存器 Xt/Wt 中， 标记物理地址为独占访问
LDAXRB	
LDAXRH	
LDAXP	LDAXP 是 Pair 访问
STLXR	把一个双字/字节/半字数据 Xt/Wt 写到 Memory 地址 addr 中， 返回是否独占访问成功状态
STLXRB	
STLXRH	
STLXP	STLXP 是 Pair 访问

6.2.8 屏障指令

DMB	数据内存屏障指令	保证该指令前的所有内存访问结束，而该指令之后引起的内存访问只能在该指令执行结束后开始，其它数据处理指令等可以越过 DMB 屏障乱序执行
DSB	数据同步屏障指令	DSB 比 DMB 管得更宽，DSB 屏障之后的所有得指令不可越过屏障乱序执行
ISB	指令同步屏障指令	ISB 比 DSB 管的更宽，ISB 屏障之前的指令保证执行完，屏障之后的指令直接 flush 掉再重新从 Memroy 中取指

- 以 DMB 指令为例介绍屏障指令原理.

ADD X1, X2, X3 ------(A) LDR X4, addr ------(B) STR X6, addr2 DMB <option> -----(DMB) LDR X5, addr3 ------(C) STR X7, addr4 SUB X8, X9, #2 ------(D)	左边程序中，因为有（DMB）的屏障作用，（C）必须要等（B）执行完成后才可以执行，保证执行顺序。而（A）、(D)不属于 Memory access 指令，可以越过 DMB 屏障 乱序执行; 而结合到 Load-Acquire/Store-Release，可以分别理解为半个 DMB 指令，Load-Acquire 只管 Memory read，而 Store-Release 只管 Memroy write，组合使用可以增加代码乱序执行的灵活性和执行效率.

6.3 A32 & T32 指令集

6.3.1 跳转指令

B	条件跳转
BL	跳转前会把当前指令的下一条指令保存到R14 （lr）
BX	只能用于寄存器寻址,寄存器最低位值用于切换ARM/Thumb工作状态,ARM/Thumb的切换只能通过跳转实现，不能通过直接write register方式实现.
BLX	BL & BX的并集
CBNZ	比较非0跳转
CBZ	比较为0跳转
TBNZ	测试位比较非0跳转
TBZ	测试位比较 0 跳转
BLR	带返回的寄存器跳转
BR	跳转到寄存器
RET	返回到子程序

6.3.2 异常产生、返回指令

- 参考 A64 指令集.

6.3.3 系统寄存器指令

- 参考 A64 指令集.

6.3.4 系统寄存器指令

- 参考 A64 指令集.

6.3.5 数据处理指令

- 参考 A64 指令集.

6.3.6 Load/Store 指令

6.3.6.1 寻址方式

Offset addressing	偏移寻址(reg or #imm)	[<Rn>, <offset>]
Pre-indexed addressing	事先更新寻址，先变化后操作	[<Rn>, <offset>]!
Post-indexed addressing	事后更新寻址，先操作后变化	[<Rn>], <offset>

6.3.6.2 Load /Store

Normal		非特权		独占		Load Acquire	Store Release	独占	
								Acquire	Release
LDR	STR	LDRT	STRT	LDREX	STREX	LDA	STL	LDAEX	STLEX
LDRH	STRH	LDRHT	STRHT	LDREXH	STREXH	LDAH	STLH	LDAEXH	STLEXH
LDRSH		LDRSHT							
LDRB	STRB	LDRBT	STRBT	LDREXB	STREXB	LDAB	STLB	LDAEXB	STLEXB
LDRSB		LDRSBT							
LDRD	STRD			LDREXD	SETEXD			LDAEXD	STLEXD

- LDRD/ STRD 和 A64 的 LDP/STP 用法类似，表中的 D(Dua)关键字和 A64 的 P(Pair)关键字是一个意思，都是指操作一对寄存器.
- 以上指令用法和 A64 类似.

6.3.6.3 Load /Store(批量)

LDM	LDM {Cond} {类型} 基址寄存器{!}, 寄存器列表{^} 从指定内存中加载批量数据到寄存器堆
STM	STM {Cond} {类型} 基址寄存器{!}, 寄存器列表{^} 把寄存器堆中批量数据存储到指定内存地址
PUSH	批量压入栈
POP	批量弹出栈

类型	助记符	指令	Note
地址	IA	LDMIA/STMIA	先操作，后递增 4 字节

变化方式	IB	LDMIB/STMIA	先递增 4 字节，后操作
	DA	LDMDA/STMDA	先操作，后递减 4 字节
	DB	LDMDB/STMDB	先递减 4 字节，后操作
栈操作	FD	LDMFD/STMFD	满递减堆栈，SP 指向最后一个元素
	FA	LDMFA/STMFA	满递增堆栈，SP 指向最后一个元素
	ED	LDMED/STMED	空递减堆栈，SP 指向将要压入数据的空地址
	EA	LDMEA/STMEA	空递增堆栈，SP 指向将要压入数据的空地址

•关于数据栈类型

满递减	堆栈首部是高地址，堆栈向低地址增长。SP 总是指向堆栈最后一个元素（最后一个元素是最后压入的数据）
满递增	堆栈首部是低地址，堆栈向高地址增长。SP 总是指向堆栈最后一个元素（最后一个元素是最后压入的数据）
空递减	堆栈首部是低地址，堆栈向高地址增长。SP 总是指向下一个将要放入数据的空位置
空递增	堆栈首部是高地址，堆栈向低地址增长。SP 总是指向下一个将要放入数据的空位置

- LDM/STM 可以实现一次在一片连续的存储器单元和多个寄存器之间传送数据，批量加载指令用于将一片连续的存储器中的数据传送到多个寄存器，批量存储指令完成相反的操作
- {!}为可选后缀，若选用，则当数据传送完毕之后，将最后的地址写入基址寄存器，否则基址寄存器的内容不改变，基址寄存器不允许为 R15(PC)，寄存器列表可以为 R0 ~ R15 的任意组合
- {^}为可选后缀，当指令为 LDM 且寄存器列表中包含有 R15，选用该后缀表示：除了正常的数据传送之外，还将 SPSR 复制到 CPSR，同时，该后缀还表示传入或传出的是用户模式下的寄存器，而不是当前模式下的寄存器

```
LDMIA R0!, {R1-R4}      // R1<----[R0]
                        // R2<----[R0 + 4]
                        // R3<----[R0 + 8]
                        // R4<----[R0 + 12]
```

```
LDMIA R0!, {R1-R4}      // R1<----[R0]
                        // R2<----[R0 + 4]
                        // R3<----[R0 + 8]
                        // R4<----[R0 + 12]
```

```
STMFD  SP!, {R0-R3}     //[R0]<----[SP]
                        //[R1]<----[SP + 4]
                        //[R2]<----[SP + 8]
                        //[R3]<----[SP + 12]
```

```
LDMFD SP!, {R6-R8}      // R6<----[SP]
                        // R7<----[SP + 4]
                        // R8<----[SP + 8]
```

6.3.7 IT(if then)指令

- 基本格式：IT{<x>{<y>{<z>}}}{<q>} <cond>
- T32 中的 IT 指令用于根据特定条件来执行紧跟其后的 1-4 条指令，其中 X, Y, Z 分别是执行第二、三、四条指令的条件，可取的值为 T（then）或 E(else)，<cond>条件的值控制指令的执行逻辑.

T 表示<cond>条件为 TRUE 则执行对应指令，E 表示<cond>为 FALSE 执行对应指令，如下例子描述.

```
ITETT   EQ
MOVEQ   R0, #1   // 1
MOVNE   R0, #0   // 2
MOVEQ   R1, #0   // 3
```

据 EQ（N==1）的条件是否成立判断，2、3、4 执行逻辑分别是 E、T、T

若 EQ 为真(N==1)，则执行 1、3、4（T）的 MOV 操作，否则执行 2(E)的 MOV 操作

E

T

MOVEQ R2, #0 // 4	T

6.3.8 协处理器指令

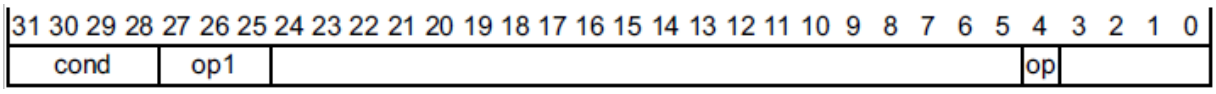
CDP	数据操作指令，用于 ARM 通知协处理器执行特定操作
LDC	数据加载指令，用于源寄存器所指向的 Mem 数据传送到目标寄存器
STC	数据存储指令，用于源寄存器所指向的数据传送到目标寄存器所指向的 Mem 中
MCR	数据传送指令，ARM 寄存器 => 协处理器寄存器
MRC	数据传送指令，ARM 寄存器 <= 协处理器寄存器

6.4 指令编码

- A32
- T32-16bit
- T32-32bit
- A64

6.4.1 A32 编码

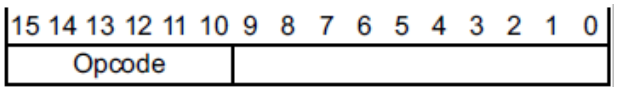
- 基本格式



固定 32bit 编码，要求字对齐
位于[31:28] 的 4bit 宽条件码
op1 位段控制指令类型：数据处理、load/store、跳转、协处理器指令…
Rd/Rn 宽度为 4bit，寄存器可访问范围 R0-R15 ，R15(PC)通常不做通用寄出去用途.

6.4.2 T32-16bit 编码

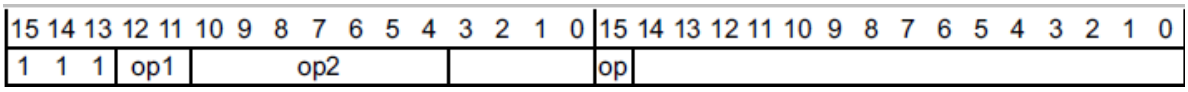
- 基本格式



固定 13bit 编码，要求半字对齐
位于[15:10] 的 5bit 决定指令类型，详见 Datasheet F3.4/P2475.
没用 cond 条件码位.
Rd/Rn 宽度为 3bit，寄存器可访问范围 R0-R7

6.4.3 T32-32bit 编码

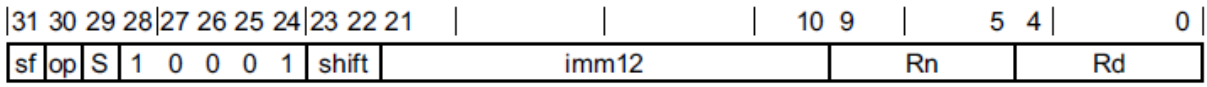
- 基本格式



固定 32bit 宽编码，由两个连续 16bit 半字组合而成，要求半字对齐
第一个半字的高三位固定为 111，Op2 位段决定指令类型.
如果 op1 == 00，那么表示会被编码位 16bit 指令，否则是 32bit 指令
Rd/Rn 宽度为 4bit，寄存器可访问范围 R0-R14

6.4.4 A64 编码

- 以 ADD 指令为例



固定 32bit 宽编码，若 sf == 0 则表示 32bit 指令，否则表示 64bit 指令
Rd/Rn 宽度为 5bit，寄存器可访问范围 X0-X30
对比 A32 指令很少 cond 位.
详细参考 Datasheet C4 章节.

6.4 汇编代码分析

- 以 memcpy.S 为例，分析笔记如下：

http://note.youdao.com/share/?id=f7976e6571ceae443da4e36d28842dcb&type=note

第 7 章 流水线

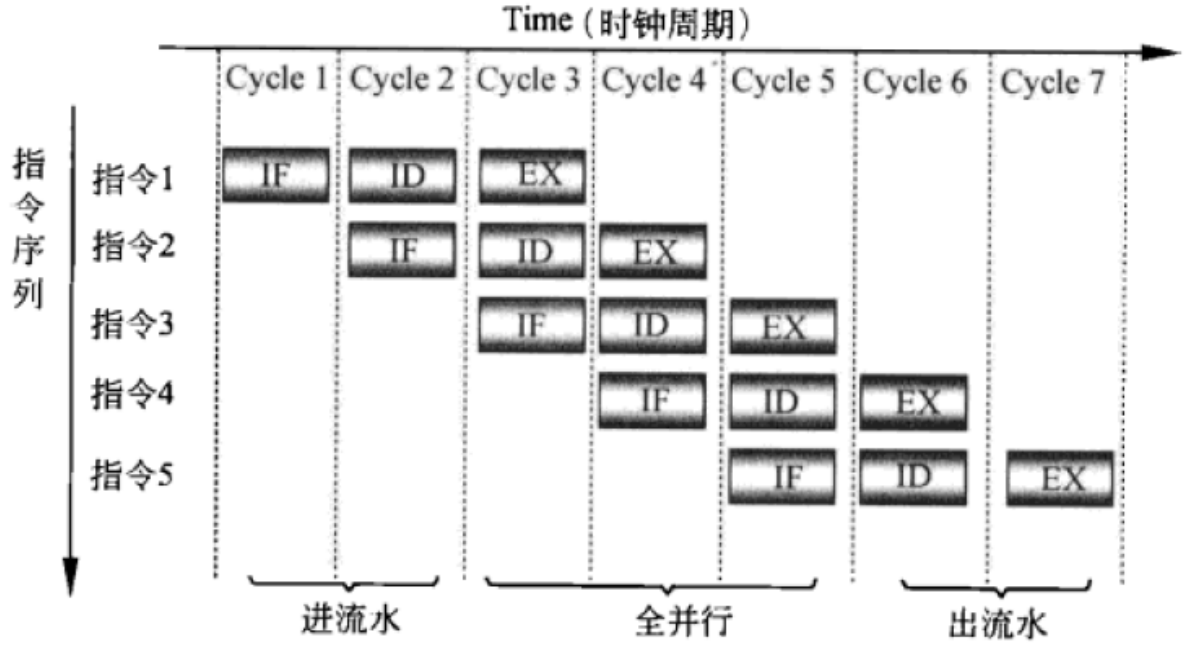
7.1 简介

1、不能减少单指令的响应时间，和 single-cycle 指令的响应时间是相同的
2、多指令同时使用不同资源，可提升整体单 cycle 内的指令吞吐量，极大提高指令执行效率
3、指令执行速率被最慢的流水线级所限制，执行效率被依赖关系限制影响

7.1.1 简单三级流水线

IF	Instruction fetch	取指
ID	Instruction decode & register file read	译码 & 读取寄存器堆数据
EX	Execution or address calculation	执行 or 地址计算

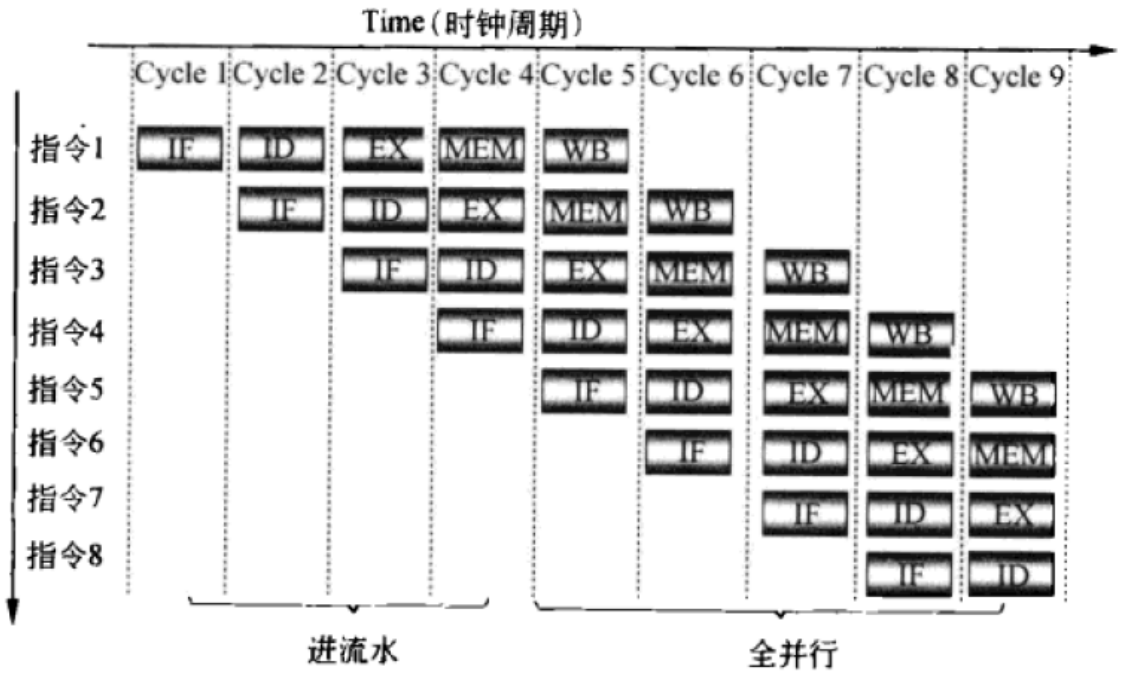
图示：



7.1.2 经典五级流水线

IF	Instruction fetch	取指
ID	Instruction decode & register file read	译码 & 读取寄存器堆数据
EX	Execution or address calculation	执行 or 地址计算
MEM	Data memeory access	读写内存数据
WB	Write back	数据写回到寄存器堆

图示：



7.2 流水线冲突

类型	Note	解决方法
结构冲突	不同指令同时占用问储器资源冲突，早期处理器程序、数据存储器混合设计产生的问题。	分离程序、数据存储器，现代处理器已不存在这种冲突
数据冲突	不同指令同时访问同一寄存器导致，通常发生在寄存器 RAW（read after write）的情况下，WAR(write after read) & WAW(write after write) 的情况再 ARM 不会发生.	• SW 插入 NOP，增加足够的 cycle 等待，但是对 CPU 性能有大影响 • HW 使用 forwarding（直通）解决，对性能影响小
控制冲突	B 指令跳转，导致其后面的指令的 fetch 等操作变成无用功，因此跳转指令会极大影响 CPU 性能.	• SW 插入 NOP，增加足够的 cycle 等待，同样对 CPU 性能有大影响 • 使用分支预测算法来减少跳转带来的性能损失

7.3 指令并行

• 指令并行提升方法

1、增加单条流水线深度，若是 N 级流水线，那么在 single-cycle 内有 N 条指令被执行.
2、Pipeline 并行，若有 M 条流水线，每条流水线深度为 N，那么 single-cycle 内有 M*N 条指令被执行，极大提升指令执行效率.