汇编与接口技术 ——ARM汇编与接口技术

版本：V1.1

HUAWEI

课程说明

- 本课程PPT及配套讲义是对贵校现已开设的汇编语言类课程内容的补充，目的是在现有汇编语言类课程中添加关于ARMv8架构及ARM汇编等知识点；
- 任课讲师可以根据现有课程大纲与授课安排挑选适当内容植入到课程中；
- 任课讲师可以根据现有班级学生对前序知识的掌握程度挑选适当内容植入到课程中；
- 本课程同时提供参考资料文档及资料库链接供任课讲师与学生使用。

前言

- 鉴于目前的国际大环境，以及出于国家信息安全方面的考虑，中国必须建立独立自主的计算机产业，具有完全自主知识产权的处理器及其汇编语言将会非常重要。ARM灵活的授权模式为我国自主研发处理器提供了条件。
- 如今ARM处理器在嵌入式领域如：工业控制、智能家电、智能仪器仪表、机电控制和消费电子领域如：各种移动设备、手机、平板以及高性能服务器领域的应用越来越广泛，市场的需求带动了技术人才的需求，在未来5年中，嵌入式领域将有超过120万的人才缺口，社会急需嵌入式系统相关专业的人才。
- ARM汇编的操作跟硬件密切相关，很多硬件设施的嵌入式编程使用的都是汇编语言，因为汇编语言代码简短，占用内存少，执行速度快，是高效的程序设计语言。现在的数码产品中使用的芯片、主板都包含了嵌入式程序，在这些程序中，汇编语言的使用是相当重要的。除此之外，学习汇编语言还能够进行软件性能优化、跨平台程序移植、设计通用/专用微处理器体系结构...

前言

- 使用汇编语言时，能够更深入地理解计算机的运行过程和原理，能够对计算机硬件和应用程序之间的联系和交互有非常清晰的认识。是最能够锻炼编程者编程思维逻辑的，学习汇编能帮助形成一个软、硬兼备的编程知识体系。
- 精通汇编的程序员，最终将脱离软件开发，成为电子工程师，负责开发设计电路及软件控制，在一些工业公司，核心电子工程师的待遇会是程序员的十倍以上。

目标

- 学完本课程后，您将能够：
  - 了解基于ARMv8架构的处理器/鲲鹏920的体系结构；
    - 了解ARM寄存器
    - ARM汇编寻址方式
    - ARM汇编指令集特色
    - 鲲鹏流水线技术
  - 掌握GNU ARM汇编语法；
  - 编写基本的ARMv8汇编代码并加以调试；
  - 了解ARM的伪指令；
  - 了解ARM汇编语言的程序结构。

目录

ARMv8架构的执行状态（1）

- ARMv8-A的执行状态：
  - AArch32执行状态
    - 支持A32和T32两种指令集，提供13个32位通用寄存器和一个32位程序计数器PC、堆栈指针SP及链接寄存器LR，提供了32个64位寄存器用于增强SIMD向量和标量浮点运算，在AArch32执行状态下，ARM同样定义了一组处理状态PSTATE参数用于保存处理单元的状态。
  - AArch64执行状态
    - 支持单一的A64指令集，定义了全新的ARMv8异常模型，AArch64执行状态的通用寄存器的数量增加到31个，同时提供了32个128位寄存器支持SIMD向量和标量浮点操作，提供一个64位程序计数器PC、若干堆栈指针SP寄存器和若干异常链接寄存器ELR，定义了一组处理状态PSTATE（Process state）参数，用于保存处理单元的状态。

ARMv8架构的执行状态 (2)

- ARMv8-A的执行状态：

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

ARMv8架构支持的数据类型

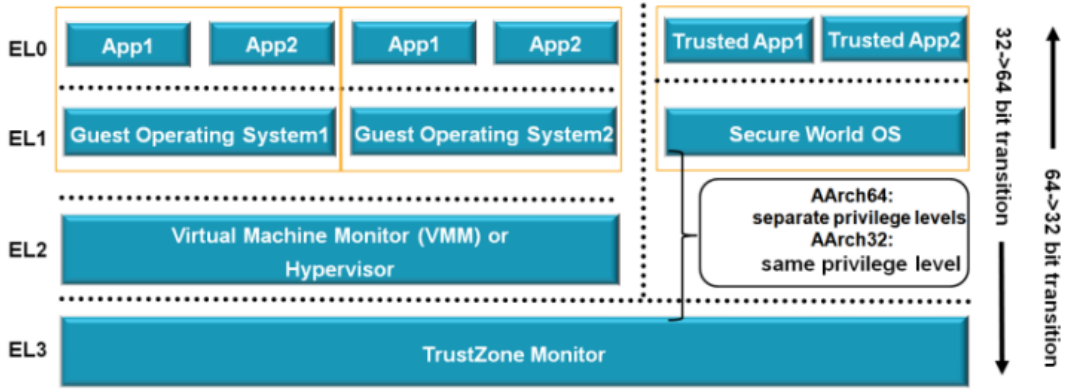
- ARMv8-A支持的数据类型：
  - 四字（Quadword）： 128位
  - 双字节（DoubleWord）：64位
  - 字（Word）：在ARM体系结构中，字的长度为32位
  - 半字（Half-Word）：在ARM体系结构中，半字的长度为16位
  - 字节（Byte）：在ARM体系结构中，字节的长度为8位
- 三种浮点数据类型
  - 半精度（Half-precision）浮点数据
  - 单精度（Single-precision）浮点数据
  - 双精度（Double-precision）浮点数据
  - 两种类型的向量数据处理
    - 其一是增强SIMD（Advanced SIMD），也就是Neon
    - 其二是可伸缩向量扩展（Scalable Vector Extension，SVE）。

ARMv8架构支持的指令集

- ARMv8-A支持的指令集：
  - 在AArch64执行状态下，ARMv8-A架构处理器只能使用A64指令集，该指令集的所有指令均为32位等长指令字。
  - 在AArch32执行状态下，可以使用两种指令集：A32指令集对应ARMv7架构及其之前的ARM指令集，为32位等长指令字结构；T32指令集则对应ARMv7架构及其之前的Thumb/Thumb-2指令集，使用16位和32位可变长指令字结构。

ARMv8架构的异常等级与安全模型（1）

- ARMv8-A的异常等级与安全模型：
  - 支持多级执行权限：程序的执行权限只有在异常处理时才能够改变，不同的执行权限等级由EL0至EL3的四个异常等级标识。异常等级的数字越大，软件的执行权限也越高。程序的权限主要涉及两个方面：一是存储系统的访问权限，二是访问处理器资源的权限。二者都与当前的异常等级密切相关。
  - EL0是最低权限等级，通常也称为非特权（unprivileged）等级；
  - EL1至EL3都属于特权等级，在这些异常等级执行程序被称为特权执行；
  - EL2异常等级提供了虚拟化（virtualization）支持；
  - EL3异常等级支持在“安全状态”和“非安全状态”这两个安全状态之间切换。



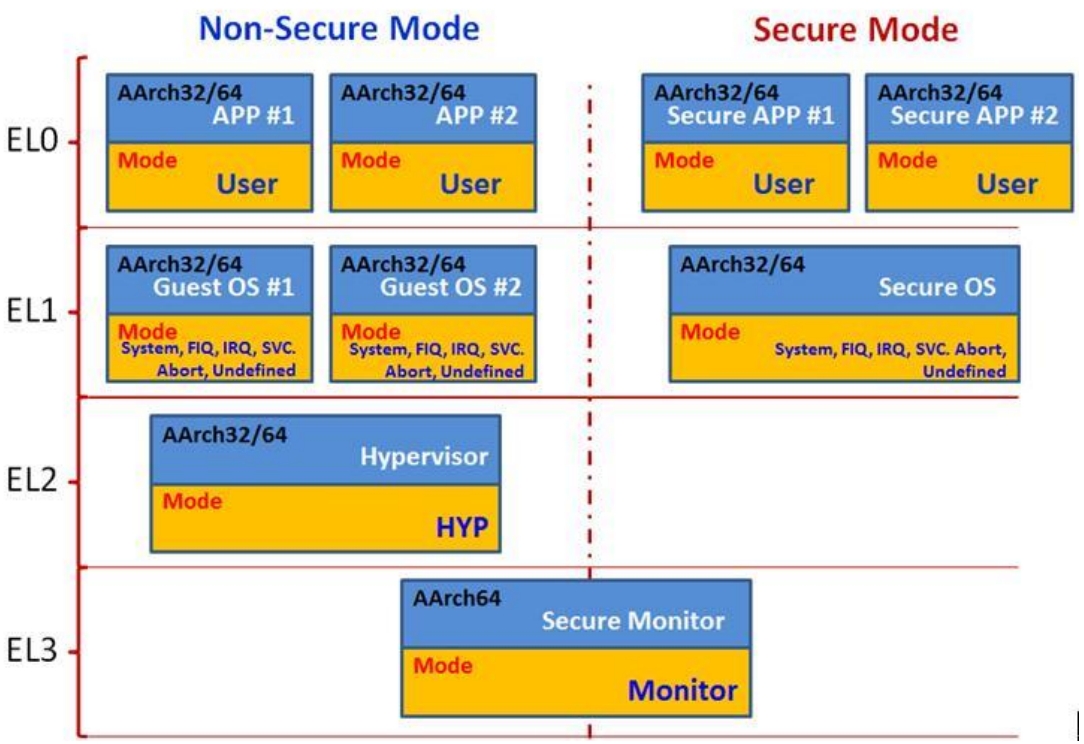
ARMv8架构的异常等级与安全模型 (2)

- ARMv8-A的异常等级与安全模型：

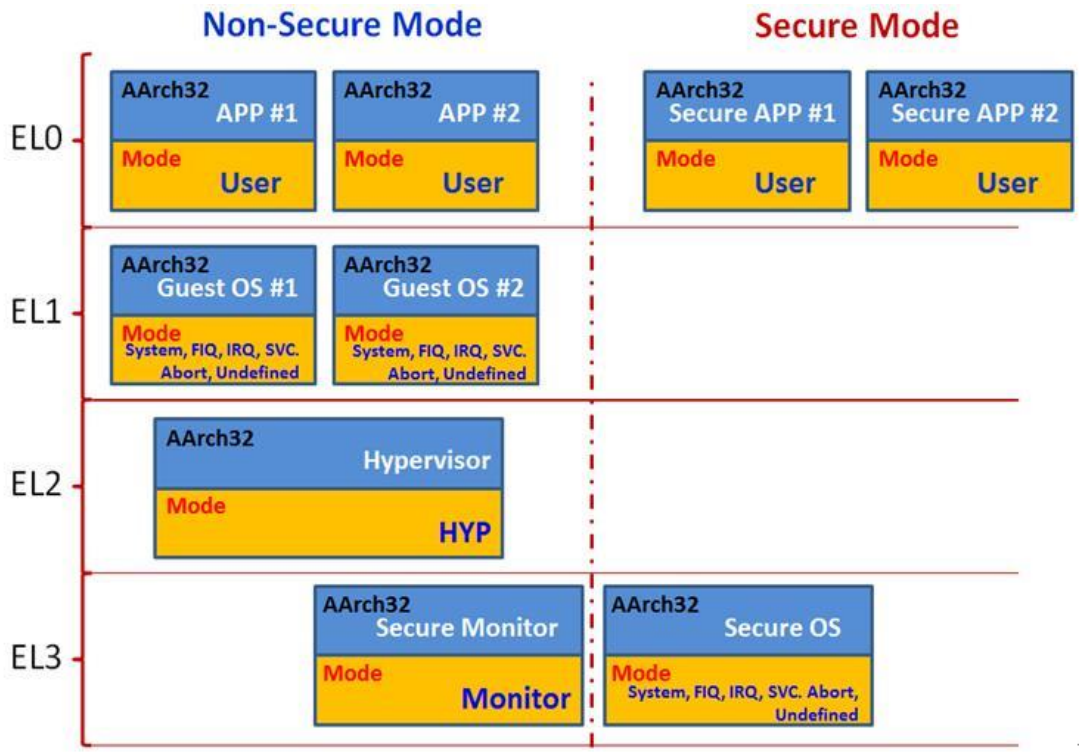
Exception Level	
EL0	Application
EL1	Linux kernel- OS
EL2	Hypervisor (可以理解为上面跑多个虚拟OS)
EL3	Secure Monitor (ARM Trusted Firmware)
Security	
Non-secure	EL0/EL1/EL2, 只能访问Non-secure memory
Secure	EL0/EL1/EL3, 可以访问Non-secure memory & Secure memory,可起到物理屏障安全隔离作用

ARMv8架构的异常等级与安全模型 (3)

- ARMv8-A的异常等级与安全模型：



当EL3使用AArch64时

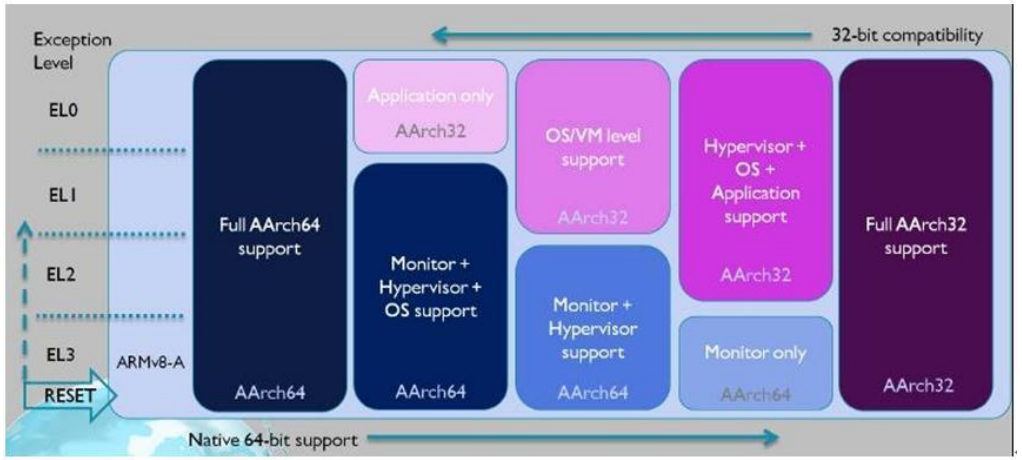


当EL3使用AArch32时

ARMv8架构的异常等级与安全模型 (4)

- ARMv8-A的异常等级与安全模型：

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
五类组合	
EL0/EL1/EL2/EL3 => AArch64	此两类组合不存在64bit –> 32bit之间的所 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	



HUAWEI

ARMv8架构的寄存器（1）

- ARMv8寄存器：

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

ARMv8架构的寄存器 (2)

- 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。
M vBAR	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。

ARMv8架构的寄存器 (3)

- 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：WO-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

ARMv8架构的异常处理（1）

- 异常（Exceptions）是现代处理器必备的程序随机切换机制。最常见的异常是由外部事件引起的中断服务过程。在复杂系统中，异常也用于处理需要特权软件权限才能处理的系统事件。每一个异常类型都有其异常处理程序。
- ARMv8-A的异常类型：
  - 同步异常（Synchronous exception）
  - 异步异常（Asynchronous exception）

  异常直接由执行指令或者尝试执行指令引起，并且异常返回地址指明了引起异常的特定指令的细节，则该异常被定义为同步异常，否则，则称为异步异常。

  异步异常是由IRQ、FIQ这两个中断请求管脚引起的中断以及系统错误引起的异常，相应地被分为三类：IRQ、FIQ和SError（System Error，系统错误）。

ARMv8架构的异常处理 (2)

- ARMv8-A的异常类型：

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

ARMv8架构的异常处理 (3)

- AArch64状态下，引起异常的几类事件：
  - 终止（Aborts）

    指令取指错误时会产生指令终止（Instruction Aborts），而数据访问错误则会引起数据终止（Data Aborts）
  - 复位（Reset）

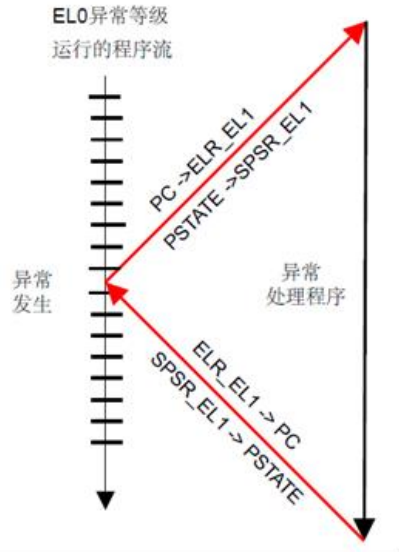
    复位异常是最高等级的异常，并且不能被屏蔽。所有处理单元在系统复位之后总是转至最高异常等级执行复位异常，并初始化系统
  - 执行异常产生指令

    异常产生指令（Exception generating instructions）就是一般所说的系统调用指令，因而执行异常产生指令将引起软中断
  - 中断（Interrupts）

    ARMv8-A架构也支持两种中断：IRQ和FIQ，后者比前者优先级更高。除了某些加载多个数值的指令可以被中断打断外，中断响应一定是发生在开中断状态下当前指令执行结束之后。由于IRQ和FIQ中断的发生都不是直接由软件执行引起的，因而都属于异步异常。

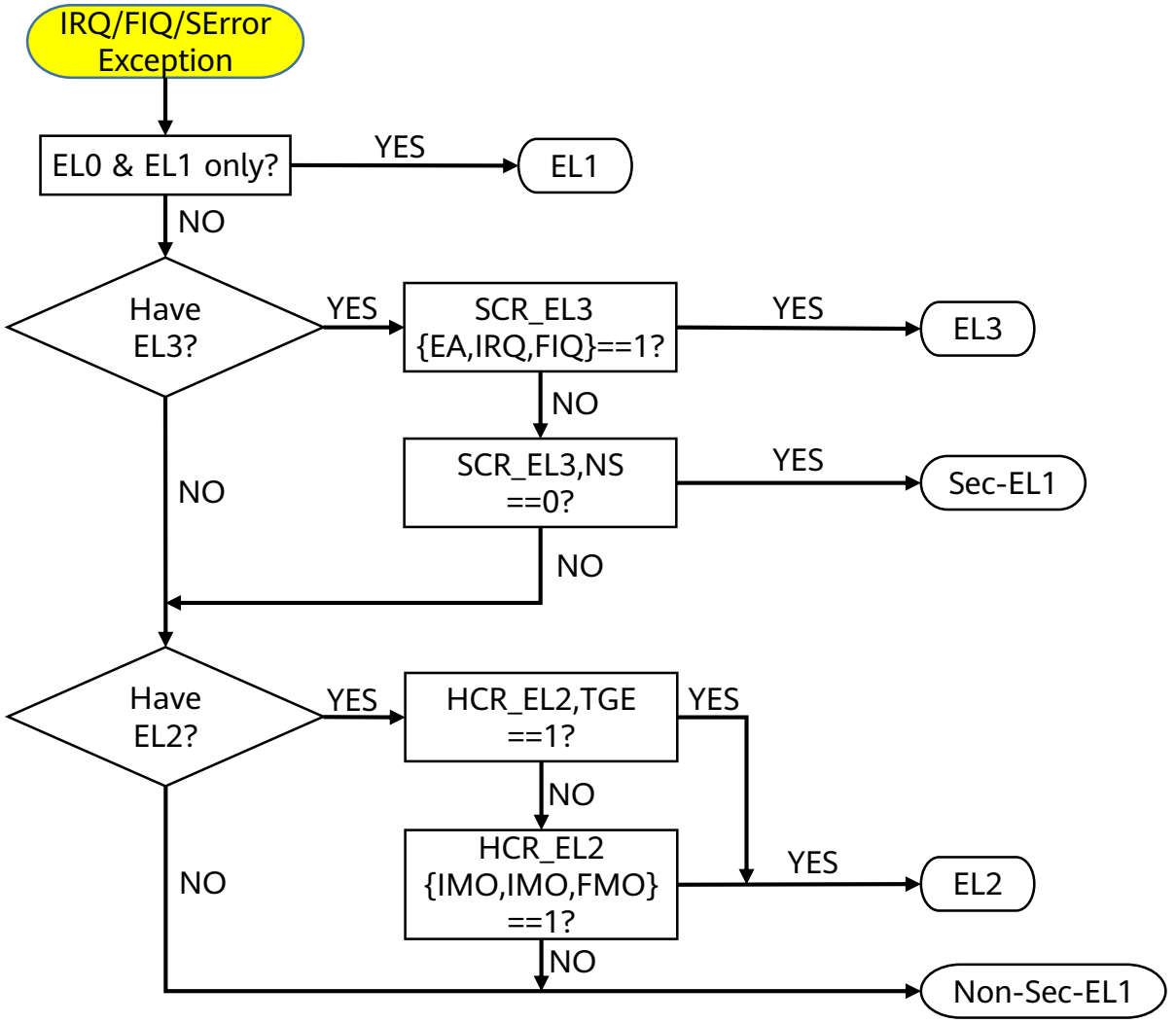
ARMv8架构的异常处理 (4)

- ARMv8-A的异常处理：
  - 首先更新备份程序状态寄存器SPSR_ELn，以保存异常处理结束返回时恢复现场必须的PSTATE信息。
  - 用新的处理器状态信息更新程序状态PSTATE。如果需要，通过此步可以提升异常等级。
  - 将异常处理结束返回的地址保存在异常链接寄存器ELR_ELn中。



ARMv8架构的异常处理 (5)

- IRQ/FIQ/SError路由流程图：



ARMv8架构的异常处理 (6)

- ARMv8-A的异常处理：

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

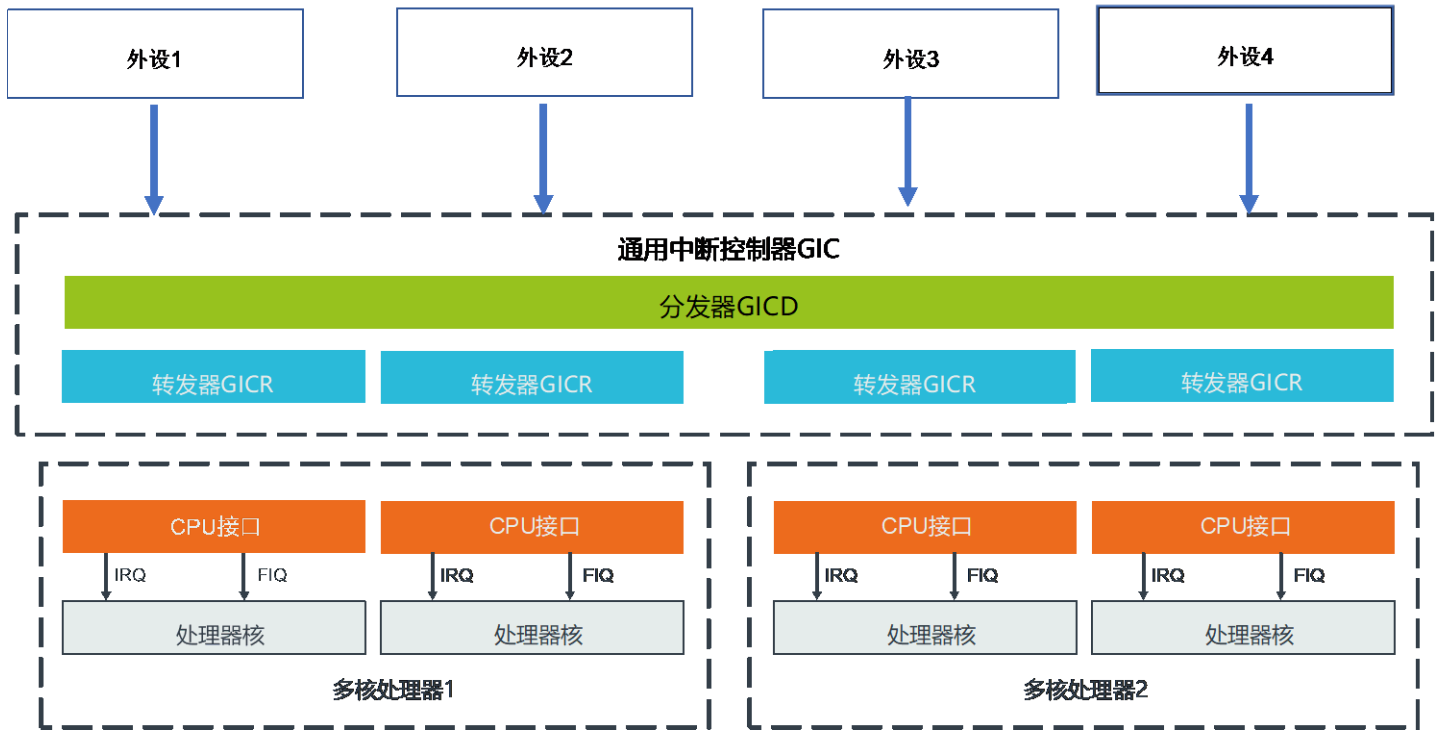
- SP => SP_EL0,表示使用SP_EL0堆栈指针，由PSTATE.SP == 0决定,PSTATE.SP == 1 则SP_ELx；
- 32bit => 64bit 是指发生异常时PE从AArch32切换到AArch64的情况；

ARMv8架构的异常向量

- AArch64的异常向量与异常向量表：

  每个异常等级都有其自身的异常向量表，因而共有EL3、EL2和EL1三个异常向量表。每个异常等级都有其相应的向量基址寄存器VBAR（Vector Base Address Register），指明在该异常等级的异常向量表的基地址。

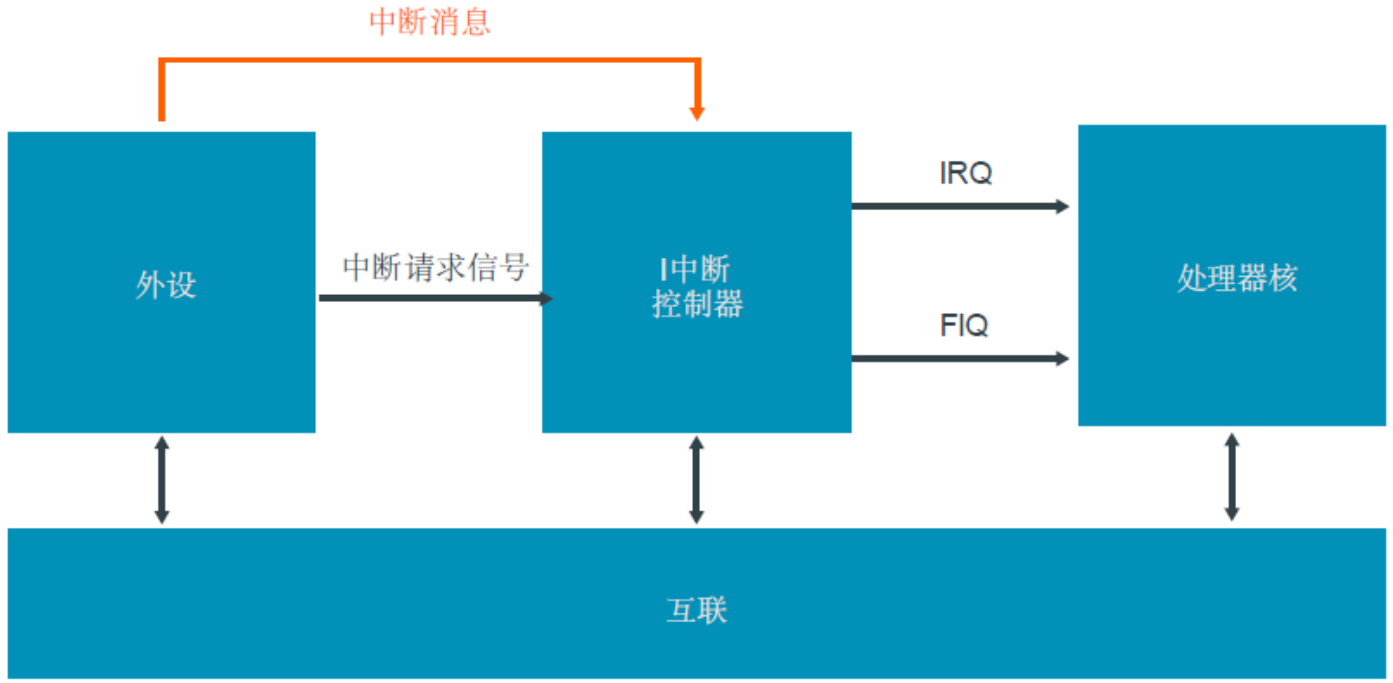
  每个异常等级的异常向量表实际上有4组，每组给出的四个异常入口分别对应同步异常、IRQ、FIQ和系统错误这四种异常类别。至于应该选择哪一组的异常向量，则取决于异常是发生于当前异常等级还是更低的异常等级、异常将使用哪一个堆栈指针（SP0还是SPn）以及异常状态所处的执行状态（AArch64或AArch32）等因素。



HUAWEI

ARMv8架构的通用中断控制器（GIC）

- ARMv8-A的通用中断控制器架构：
  - 对于高性能、复杂的处理器而言，外部中断源应该是相当多的，而ARM处理器核本身只能支持FIQ和IRQ两级外部中断请求输入，因而在ARM架构下，系统通过**通用中断控制器**（Generic Interrupt Controller，**GIC**）实现中断请求的仲裁、优先级排队和向处理器中断申请等操作。



ARMv8架构的中断类型（1）

- 中断控制器处理的中断源分为四种类型：
  - 共享外设中断（ Shared Peripheral Interrupt，SPI ）：外设的这类中断请求可以被连接到任何一个处理器核。
  - 私有外设中断（ Private Peripheral Interrupt，PPI ）：只属于某一个处理器核的外设的中断请求，例如通用定时器的中断请求。
  - 软件产生的中断（ Software Generated Interrupt，SGI ）：由软件写入中断控制器内的SGI寄存器引发的中断请求，通常用于处理机间通信；
  - 特定位置外设中断（ Locality-specific Peripheral Interrupt，LPI ）：边沿触发的基于消息的中断，其编程模式与其他类中断源完全不同。

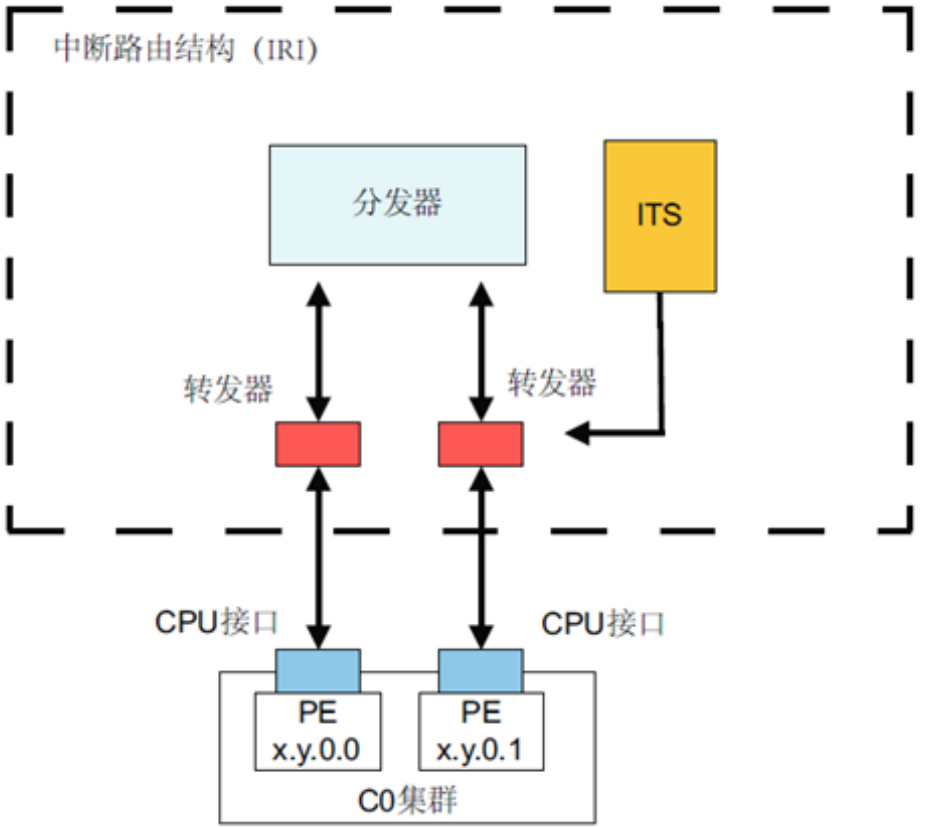
ARMv8架构的中断类型（2）

- GIC中断标识与中断类型对照：

中断标识（INTID）	中断类型	中断类型缩写
0 - 15	软件产生的中断	SGI
16 – 31 1056 – 1119（GICv3.1）	私有外设中断	PPI
32 – 1019 4096 – 5119（GICv3.1）	共享外设中断	SPI
1020 -1023	特殊中断号	
1024 -8191	保留	
8192及以上	特定位置外设中断	LPI

ARMv8架构的通用中断控制器组成结构

- GIC由四种逻辑部件组成：
  - 分发器（Distributor）
  - 转发器（Redistributor）
  - CPU接口
  - 中断翻译服务部件ITS
  - （Interrupt Translation Service components）



目录

基于ARMv8架构的鲲鹏处理器

- 鲲鹏处理器架构特性
  - ARMv8.0架构，支持EL0~EL3， 支持Trust-Zone, AArch64 only
  - 兼容ARMv8.1
    - 支持Atomic指令
    - 支持PSTATE.PAN
    - 支持Virtualization Host Extension
  - 兼容ARMv8.2
    - 支持半精度浮点指令
    - 支持Dot Product指令
    - 支持RAS Extension
    - 支持Statistical Profiling Extension
  - 支持ARMv8.3部分指令：
    - 支持SIMD complex number
    - 支持JavaScript conversion instruction
  - 支持ARMv8.4 Memory Partitioning and Monitoring Extension (MPAM)
  - 支持ARMv8.5 Restrictions on the effects of speculation
    - Support Mitigation for 变种1/2/3/3a/4

基于ARMv8架构的鲲鹏处理器——流水线技术 (1)

Taishan coreV110 Pipeline Architecture

基于ARMv8架构的鲲鹏处理器——流水线技术 (2)

- Branch预测和取指流水线解耦设计，取指流水线每拍最多可提供32Bytes指令供译码，分支预测流水线可以不受取指流水停顿影响，超前进行预测处理；
- 定浮点流水线分开设计，解除定浮点相互反压，每拍可为后端执行部件提供4条整型微指令及3条浮点微指令；
- 整型运算单元支持每拍4条ALU运算（含2条跳转）及1条乘除运算；
- 浮点及SIMD运算单元支持每拍2条ARM Neon 128bits 浮点及SIMD运算；
- 访存单元支持每拍2条读或写访存操作，读操作最快4拍完成，每拍访存带宽为2x128bits读及1x128bits写。

鲲鹏920系列芯片——流水线技术 (1)

- 原理

C/C++代码在编译时，GCC编译器将源码翻译成CPU可识别的指令序列，写入可执行程序的二进制文件中。CPU在执行指令时，通常采用流水线的方式并行执行指令，以提高性能，因此指令执行顺序的编排将对流水线执行效率有很大影响。

在指令流水线中要考虑：执行指令计算的硬件资源数量、不同指令的执行周期、指令间的数据依赖等等因素。

我们可以通知编译器，程序所运行的目标平台(CPU)指令集、流水线，来获取更好的指令序列编排。在GCC 9.1.0版本，支持了鲲鹏处理器所兼容的Arm-v8指令集、tsv110流水线。

如果在编译时增加编译选项指定使用tsv110流水线 ，使编译器按照鲲鹏处理器的流水线编排指令执行顺序，就可以充分利用流水线的指令级并行，以提升性能。

鲲鹏920系列芯片——流水线技术 (2)

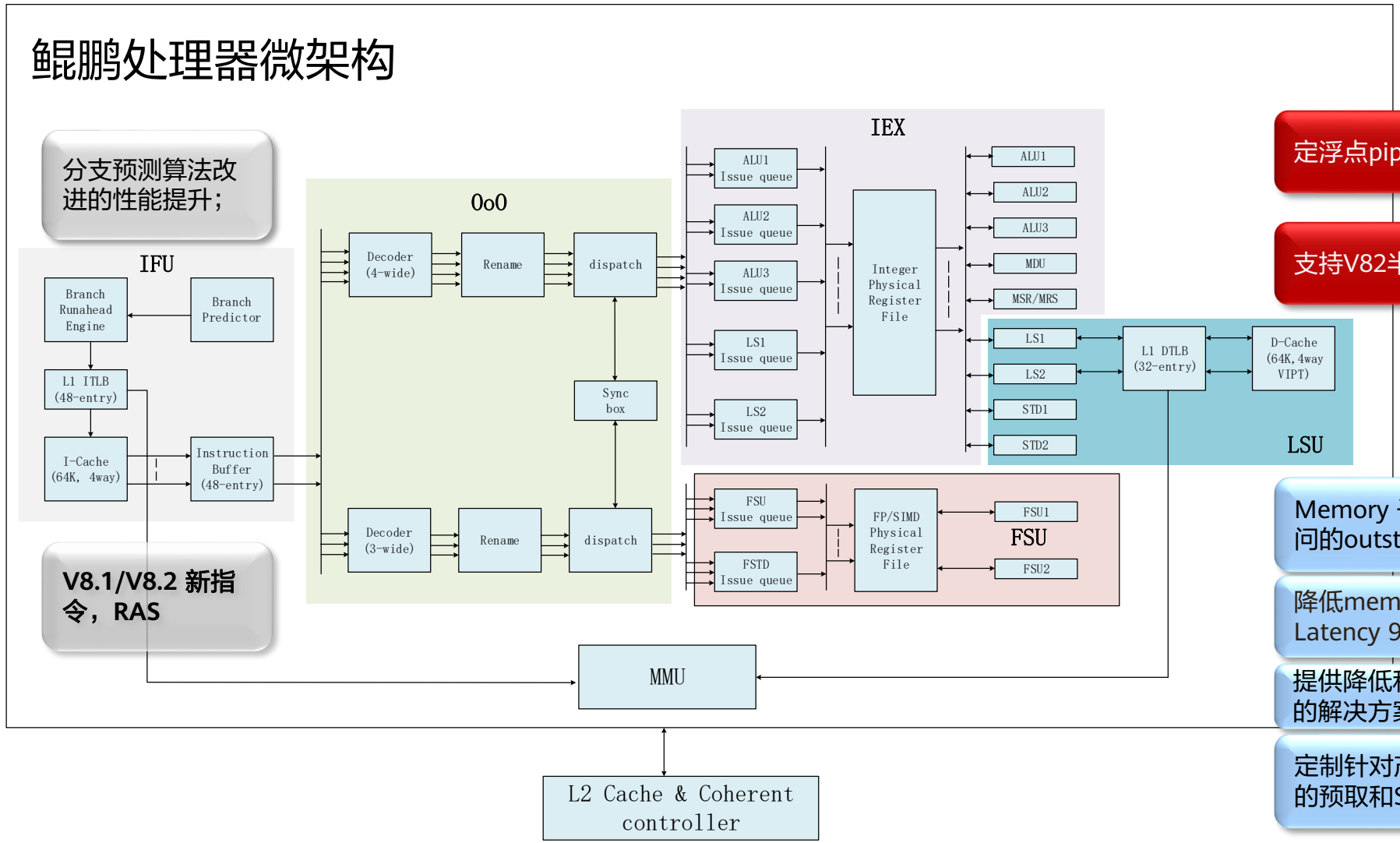
- 修改方式
  - 在Euler系统中使用HCC编译器，可以在CFLAGS和CPPFLAGS里面增加编译选项：

```
-mtune=tsv110 -march=ARMv8-a
```

  - 在其它操作系统中，可以升级GCC版本到9.10，并在CFLAGS和CPPFLAGS里面增加编译选项：

```
-mtune=tsv110 -march=ARMv8-a
```

基于ARMv8架构的鲲鹏处理器



定浮点pipe独立，提升dispatch带宽

支持V82半精度浮点计算

Memory 子系统深度优化，提升内存访问的outstanding深度

降低memory访问时延，每个核私有L2，Latency 9 cycles

提供降低和规避Device访问对性能影响的解决方案

定制针对产品应用的指令和数据Cache的预取和Streaming算法

目录

ARM寻址方式 (1)

- 寻址就是找到存储数据或指令的地址，寻址方式的方便与快捷是衡量CPU性能的一个重要方面，ARM处理器共有八种寻址方式：
  - 立即数寻址
  - 寄存器寻址
  - 寄存器间接寻址
  - 基址寻址
  - 多寄存器寻址
  - 堆栈寻址
  - 相对寻址
  - 寄存器移位寻址

ARM寻址方式 (2)

- 立即数寻址：
  - 立即数寻址指令中的地址码就是操作数本身，可以立即使用的操作数。其中，#0xFF000和#64都是立即数。如操作数是常量，用#表示常量；0x或&表示16进制数，否则表示十进制数。

例如：

```
MOV  R0,#0xFF000
```

@指令省略了第1个操作数寄存器。将立即数0xFF000(第2操作数)装入R0寄存器

```
SUB  R0,R0,#64
```

@R0减64，结果放入R0

ARM寻址方式 (3)

- 寄存器寻址：
  - 操作数的值在寄存器中，指令执行时直接取出寄存器值来操作，寄存器寻址是根据寄存器编码获取寄存器内存储的操作数

例如：

```
MOV R1,R2
```

@将R2的值存入R1　　在第1个操作数寄存器的位置存放R2编码

```
SUB R0,R1,R2
```

@将R1的值减去R2的值，结果保存到R0在第2操作数位置，存放的是寄存器R2的编码

ARM寻址方式 (4)

- 寄存器间接寻址：
  - 操作数从寄存器所指向的内存中取出，寄存地存储的是内存地址

例如：

```
LDR R1,[R2]
```

@将R2指向的存储单元的数据读出，保存在R1中 R2相当于指针变量

```
STR R1,[R2]
```

@将R1的值写入到R2所指向的内存

```
SWP R1,R1,[R2]
```

@将寄存器R1的值和R2指定的存储单元的内容交换

[R2]表示寄存器所指向的内存

LDR 指令用于读取内存数据

STR 指令用于写入内存数据

ARM寻址方式 (5)

- 基址变址寻址：
  - 基址寄存器的内容与指令中的偏移量相加，得到有效操作数的地址，然后访问该地址空间，基址变址寻址分为三种：
    - 前索引，例如：

```
LDR  R0, [R1,#4]
```

@R1存的地址+4，访问新地址里面的值，放到R0；

    - 自动索引，例如：

```
LDR  R0, [R1,#4]!
```

@在前索引的基础上，新地址回写进R1；(注：!表示回写地址)

    - 后索引，例如：

```
LDR  R0 [R1],#4
```

@R1存的地址的内容写进R0，R1存的地址+4再写进R1；

ARM寻址方式 (6)

- 多寄存器寻址：
  - 一条指令完成多个寄存器的传送，最多16个寄存器，也称为块拷贝寻址

例如：

```
LDMIA R1!,{R2-R7,R12}
```

@将R1指向的存储单元中的数据读写到R2～R7、R12中，然后R1自加1

```
STMIA R1!,{R2-R7,R12}
```

@将寄存器R2～R7、R12的值保存到R1指向的存储单元中，然后R1自加1

注：基址寄存器不允许为R15，寄存器列表可以为R0～R15 的任意组合。这里R1没有写成[R1]!，是因为这个位不是操作数位，而是寄存器位。LDMIA 和 STMIA 是块拷贝指令，LDMIA是从R1所指向的内存中读数据，STMIA是向R1所指向的内存写入数据

  - 执行这类指令要考虑如下几个问题：
    - 基址寄存器指向原始地址有没有放一个有效值
    - 寄存器列表哪个寄存器被最先传送
    - 存储器地址增长方向
    - 指令执行完成后，基址寄存器有没有指向一个有效值

ARM寻址方式 (7)

- 寄存器堆栈寻址：
  - 是按特定顺序存取存储区，按后进先出原则，使用专门的寄存器SP（堆栈指针)指向一块存储区

例如：

```
LDMIA SP!,{R2-R7,R12}
```

@将栈内的数据，读写到R2～R7、R12中，然后下一个地址成为栈顶

```
STMIA SP!,{R2-R7,R12}
```

@将寄存器R2～R7、R12的值保存到SP指向的栈中，SP指向的是栈顶

ARM寻址方式 (8)

- 相对寻址：
  - 即读取指令本身在内存中的地址。是相对于PC内指令地址偏移后的地址。由程序计数器PC提供基准地址，指令中的地址码字段作为偏移量，两者相加后得到的地址即为操作数的有效地址

例如：

```
BL    ROUTE1                    @调用到 ROUTE1 子程序
BEQ LOOP                        @条件跳转到 LOOP 标号处
…
LOOP MOV R2,#2
…
ROUTE1
…
```

目录

ARM指令集 (1)

- GNU ARM汇编语言语法格式：

[<label>:][<instruction or directive or pseudo-instruction>} @comment

  - instruction伪指令
  - directive伪操作
  - pseudo-instruction伪指令
  - <label>: 为标号, GNU ARM汇编中，任何以冒号结尾的标识符都被认为是一个标号，而不一定非要在一行的开始
  - comment为语句的注释

注意：

- ARM指令，伪指令，伪操作，寄存器名可以全部为大写字母，也可全部为小写字母，但不可大小写混用。
- 如果语句太长，可以将一条语句分几行来书写，在行末用“\”表示换行 (即下一行与本行为同一语句)，“\”后不能有任何字符，包含空格和制表符 (Tab)。

ARM指令集 (2)

- GNU ARM汇编语言语法格式：
  - 局部变量定义的语法格式：**N{routname}**
    - N：为0~99之间的数字。
    - routname：当前局部范围的名称（为符号），通常为该变量作用范围的名称 (用ROUT伪操作定义的)
  - 局部变量引用的语法格式：**%{F|B}{A|T}N{routname}**
    - %：表示引用操作
    - N：为局部变量的数字号
    - routname：为当前作用范围的名称（用ROUT伪操作定义的）
    - F：指示编译器只向前搜索
    - B：指示编译器只向后搜索
    - A：指示编译器搜索宏的所有嵌套层次
    - T：指示编译器搜索宏的当前层次

ARM指令集 (3)

- GNU ARM汇编语言语法格式：
  - GNU ARM汇编特殊字符和语法
    - 代码行中的注释符号: ‘@’
    - 整行注释符号: ‘#’
    - 语句分离符号: ‘;’
    - 立即数前缀: ‘#’ 或 ‘$’

ARM指令集 (4)

- 指令分类：

指令类型	说明
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

ARM指令集 (5)

- A64指令特点：
  - 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

ARM指令集 (6)

- 跳转指令：
  - 条件跳转

指令	说明
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

ARM指令集 (7)

- 跳转指令：
  - 绝对跳转

指令	说明
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）

例如：

```
BL  func            @调用子程序func
    ...
    func
    ...
    MOV R15,R14     @子程序返回
```

ARM指令集 (8)

- 异常产生和返回指令：

指令	说明
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

ARM指令集 (9)

- 系统寄存器指令：

指令	说明
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

例如：

```
MRS R0,CPSR                 @状态寄存器CPSR的值存入寄存器R0中

MSR CPSR_f,R0               @用R0的值修改CPSR的条件标志域

MSR CPSR_fsxc,#5            @CPSR的值修改为5
```

ARM指令集 (10)

- 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					

ARM指令集 (11)

- 算术运算指令：

指令	说明
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

ARM指令集 (12)

- 算术运算指令：

  例如：

  ADD 加法指令

```
ADD R0,R1,#5                @R0=R1+5
ADD R0,R1,R2                @R0=R1+R2
ADD R0,R1,R2,LSL #5         @R0=R1+R2左移5位
```

  SUB 减法指令

```
SUB R0,R1,R2                @R0=R1-R2
SUB R0,R1,R2,LSL #5         @R0=R1-R2左移5位
```

ARM指令集 (13)

- 逻辑运算指令：

指令	说明
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：ANDS WZR,W0, #0X40

ARM指令集 (14)

- 逻辑运算指令：

  例如：

  AND逻辑与指令

AND R0,R0, #5	*@保持R0的第0位和第2位，其余位清0*

  ORR逻辑或指令

ORR R0,R0, #5	*@R0的第0位和第2位设置为1，其余位不变*

  EOR逻辑异或指令

EOR R0,R0, #5	*@R0的第0位和第2位取反，其余位不变*

ARM指令集 (15)

- 数据传输指令：

指令	说明
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

ARM指令集 (16)

- 数据传输指令：

  例如：

  MOV 数据传送指令

```
MOV  R0,#0xFF000        @立即寻址，将立即数0xFF000(第2操作数)装入R0寄存器
MOV R1,R2               @寄存器寻址，将R2的值存入R1
MOV R0,R2,LSL #3        @移位寻址，R2的值左移3位，结果放入R0
```

  M vN 数据取反传送指令

```
M vN R0, # 0 ;R0=-1
```

ARM指令集 (17)

- 地址生成指令：

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

ARM指令集 (18)

- 位段移动指令：

指令	说明
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	

ARM指令集 (19)

- 移位运算指令：

指令	说明
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	

ARM指令集 (20)

- Load/Store指令：

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

ARM指令集 – SIMD指令简介

- SIMD (Single Instruction Multiple Data)，单指令多数据流，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
- SIMD可以以同步的方式，在同一时间内执行同一条指令
- 以加法指令为例：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
- 这个特点使SIMD非常适合于多媒体应用等数据密集型运算

目录

ARM伪指令（1）

- 伪指令是编译器支持的指令，不是硬件芯片支持的指令。
  - 编译器在编译时，会把伪指令转化对应的芯片支持的指令。伪指令集包括：伪操作和伪指令
  - 伪操作：
    - 数据定义（Data Definition）伪操作
    - 汇编控制伪操作
    - 杂项伪操作
  - 伪指令：
    - ADR伪指令
    - ADRL伪指令
    - LDR伪指令

ARM伪指令 (2)

- 数据定义（ Data Definition ）伪操作

.byte	单字节定义	.byte	0x12,’a’,23
.short	定义2字节数据	.short	0x1234,65535
.long /.word	定义4字节数据	.word	0x12345678
.quad	定义8字节	.quad	0x1234567812345678
.float	定义浮点数	.float	0f3.2
.string/.asciz/.ascii	定义字符串	.ascii	“abcd\0”,

ARM伪指令 (3)

- 汇编控制伪操作
  - .if .else .endif ---- 类似c语言里的条件编译，汇编控制伪操作用于控制汇编程序的执行流程 .if、.else、.endif伪操作能根据条件的成立与否决定是否执行某个指令序列。当.if后面的逻辑表达式为真，则执行.if后的指令序列，否则执行.else后的指令序列；.if、.else、.endif伪指令可以嵌套使用。
  - .macro，.endm --- 类似c语言里的宏函数 .macro伪操作可以将一段代码定义为一个整体，称为宏指令。然后就可以在程序中通过宏指令多次调用该段代码。其中，$标号在宏指令被展开时，标号会被替换为用户定义的符号。宏操作可以使用一个或多个参数，当宏操作被展开时，这些参数被相应的值替换。

ARM伪指令 (4)

- MACRO 、 MEND 示例

  例如：

```
MACRO                                   @宏定义
CALL $Function,$dat1,$dat2              @宏名称为CALL,带3 个参数
IMPORT $Function                        @声明外部子程序 宏开始
MOV R0,$dat1                            @设置子程序参数,R0=$dat1
MOV R1,$dat2
BL $Function                            @调用子程序 宏最后一句
MEND                                    @宏定义结束
CALL FADD1,#3,#2                        @宏调用，后面是三个参数
```

ARM伪指令 (5)

- 杂项伪操作：

伪操作	语法	说明
.arm	.arm	定义一下代码使用ARM指令集编译
.thumb	.thumb	定义一下代码使用Thumb指令集编译
.section	.section expr	定义一个段。expr可以使.text　.data.　.bss
.text	.text {subsection}	将定义符开始的代码编译到代码段
.data	.data {subsection}	将定义符开始的代码编译到数据段,初始化数据段
.bss	.bss {subsection}	将变量存放到.bss段,未初始化数据段

ARM伪指令 (6)

- ADR伪指令
  - ADR伪指令为小范围地址读取伪指令，使用的相对偏移范围：当地址值是字节对齐 (8位) 时，取值范围为-255～255，当地址值是字对齐 (32位) 时，取值范围为-1020～1020。
  - 语法格式：

```
ADR{cond}   register,label

ADR  R0,  lable
```

ARM伪指令 (7)

- ADRL伪指令
  - ADRL伪指令为中等范围地址读取伪指令。使用相对偏移范围：当地址值是字节对齐时，取值范围为-64～64KB；当地址值是字对齐时，取值范围为-256～256KB
  - 语法格式：

```
ADRL{cond}  register,label

ADRL  R0，lable
```

ARM伪指令 (8)

- LDR伪指令
  - LDR伪指令装载一个32位的常数和一个地址到寄存器
  - 语法格式：

```
LDR{cond}  register,=[expr|label-expr]

LDR    R0, =0XFFFF0000
```

目录

汇编语言程序结构 (1)

- 顺序结构
- 分支结构
  - 双分支结构
  - 多分支结构
- 循环结构
- 子程序

汇编语言程序结构 (2)

- 顺序结构

  顺序结构程序是最简单的也是最基本的一种程序结构形式。这种结构的程序有程序的开头顺序的执行直到程序结束为止，执行过程中没有任何分支。

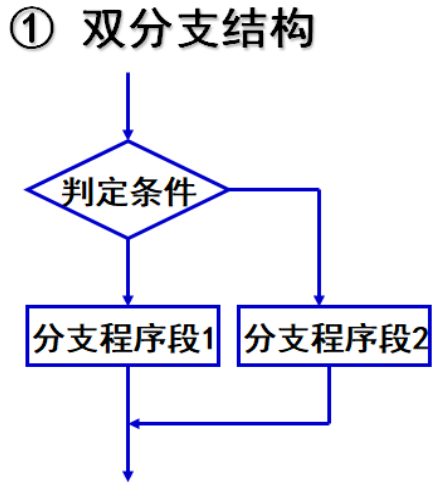
汇编语言程序结构 (3)

- 分支结构

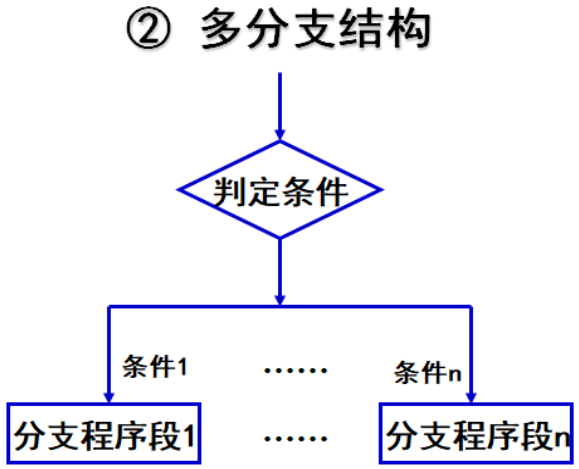
  可以事先把各种可能出现的情况和处理的方法写在程序里，然后由计算机自动做出判断，并跳转或调用相应的程序处理。特点是：其运行方向是向前的，再确定的条件下，只能执行多个分支中的一个分支。

  分支结构的分类：

  - 双分支结构
  - 多分支结构



用条件转移指令来实现



联用跳转表和无条件转移指令实现

汇编语言程序结构 (4)

- 条件跳转示例

```
AREA Example,CODE,READONLY          @声明代码段Example
ENTRY                               @程序入口
Start
MOV  R0,#2                          @将R0赋初值2
MOV  R1,#5                          @将R1赋初值5
ADD  R5,R0,R1                       @将R0和R1内的值相加并存入R5
CMP  R5,#10
BEQ  DOEQUAL                        @若R5为10，则跳转到DOEQUAL标签处
WAIT
CMP  R0,R1
ADDHI R2,R0,#10                     @若R0 >R1 则R2=R0+10
ADDLS R2,R1,#5                      @若R1<=R2 则R2=R1+5
DOEQUAL
ANDS R1,R1,#0x80                    @R1=R1& 0x80,并设置相应标志位
BNE  WAIT                           @若R1 的d7位为1则跳转到WAIT标签
OVER
END
```

汇编语言程序结构 (5)

- 循环结构

  需要多次重复执行相同的或相似的功能时可以使用循环结构。

- 循环程序结构：
  - 初始化部分：设置循环执行的初始化状态。
  - 循环体部分：需要多次重复执行的部分。
  - 循环控制部分：用于控制循环体的执行的次数。循环体每次执行后，应该修改循环条件，是循环能够在适当的时候终止执行。

- 循环控制方法
  - 计数控制法
  - 条件控制法
  - 混合控制法

汇编语言程序结构 (6)

- 循环结构示例

```
AREA Example,CODE,READONLY          @声明代码段Example
ENTRY                               @程序入口
Start
MOV  R1,#0                          @将R1赋初值0
LOOP
ADD  R1,R1,#1
CMP  R1,#10
BCC  LOOP          @R1小于10则执行跳转到LOOP处执行循环，即R1从0到10后退出循环
END
```

汇编语言程序结构 (7)

- 子程序
  - 主程序：往往要调用子程序或处理中断, 暂停主程序，执行子程序或中断服务程序。
  - 子程序：子程序又称为过程。在一个实际程序中，有些操作要执行多次，把要重复执行（ subroutine ）操作编为子程序。也常把一些常用的操作标准化、通用化成子程序。
  - 子程序结构是模块化程序设计的基础，调用子程序时需保留内容。

目录

ARM编译与调试工具 (1)

- 使用gcc编译器编译c文件流程
  - 预编译处理，生成main.i文件
  - 编译处理，生成main.s文件
  - 汇编处理，生成main.o文件
  - 链接处理，生成main.exe文件

ARM编译与调试工具 (2)

- 使用SSH工具，输入用户名和密码，登陆

```
Microsoft Windows [版本 10.0.18363.720]
(c) 2019 Microsoft Corporation。保留所有权利。

C:\Users\lwx941162>ssh root@121.36.77.157
The authenticity of host '121.36.77.157 (121.36.77.157)' can't be established.
ECDSA key fingerprint is SHA256:BsuJXyOz4oTDLfid745+rUwrnvG/DRdM+SSMzjBAo6Q.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '121.36.77.157' (ECDSA) to the list of known hosts.
root@121.36.77.157's password:
Permission denied, please try again.
root@121.36.77.157's password:
Last failed login: Wed Aug 12 10:21:12 CST 2020 from 119.3.119.20 on ssh:notty
There was 1 failed login attempt since the last successful login.
Last failed login: Wed Aug 12 10:21:12 CST 2020 from 119.3.119.20 on ssh:notty
There was 1 failed login attempt since the last successful login.

       Welcome to Huawei Cloud Service

[root@esc-huawei ~]# uname -a
Linux esc-huawei 4.18.0-80.7.2.e17.aarch64 #1 SMP Thu Sep 12 16:13:20 UTC 2019 aarch64 aarch64 aarch64 GNU/Linux
[root@esc-huawei ~]#
```

ARM编译与调试工具 (3)

- 更新编译环境，命令如下：

```
yum groupinstall "Development tools“
```

- 升级gcc版本，依次如下命令：

```
yum -y install centos-release-scl
yum -y install devtoolset-7-gcc devtoolset-7-gcc-c++ devtoolset-7-binutils
scl enable devtoolset-7 bash
echo "source /opt/rh/devtoolset-7/enable" >>/etc/profile
```

ARM编译与调试工具 (4)

- 使用“gcc -v”命令可以输出gcc版本，过程截图如下：

```
[root@ecs-huawei2 ~]# scl enable devtoolset-7 bash
[root@ecs-huawei2 ~]# echo "source /opt/rh/devtoolset-7/enable" >>/etc/profile
[root@ecs-huawei2 ~]# gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/opt/rh/devtoolset-7/root/usr/libexec/gcc/aarch64-redhat-linux/7/lto-wrapper
Target: aarch64-redhat-linux
Configured with: ../configure --enable-bootstrap --enable-languages=c,c++,fortran,lto --prefix=/opt/rh/devtoolset-7/root/usr --mandir=/opt/rh/devtoolset-7/root/usr/
share/man --infodir=/opt/rh/devtoolset-7/root/usr/share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --enable-shared --enable-threads=posix --enable-check
ing=release --enable-multilib --with-system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --enable-gnu-unique-object --enable-linker-build-id --with-gcc
-major-version-only --enable-plugin --with-linker-hash-style=gnu --enable-initfini-array --with-default-libstdcxx-abi=gcc4-compatible --with-isl=/builddir/build/BUI
LD/gcc-7.3.1-20180303/obj-aarch64-redhat-linux/isl-install --disable-libmpx --enable-gnu-indirect-function --build=aarch64-redhat-linux
Thread model: posix
gcc version 7.3.1 20180303 (Red Hat 7.3.1-5) (GCC)
```

ARM编译与调试工具 (5)

- 使用gcc来编译c语言程序
  - 新建汇编源文件，命令如下：

```
vim hello.s
```

  - 在hello.s中输入以下代码：

```
.text
.global tart1
tart1:
        mov x0,#0
        ldr x1,=msg
        mov x2,len
        mov x8,64
        svc #0

        mov x0,123
        mov x8,93
        svc #0
.data
msg:
        .ascii "Hello World!\n"
len=.-msg
```

ARM编译与调试工具 (6)

- 保存hello.s文件，然后通过运行以下命令将其编译为二进制文件

```
as hello.s –o hello.o
```

使用以下命令进行链接，输出可执行文件

```
ld hello.o –o hello
```

使用以下命令执行hello程序

```
./hello
```

ARM编译与调试工具 (7)

- 程序运行结果如下图所示：

```
[root@ecs-huawei2 ~]# vim hello.s
[root@ecs-huawei2 ~]# as hello.s -o hello.o
[root@ecs-huawei2 ~]# ld hello.o -o hello
ld: warning: cannot find entry symbol _start; defaulting to 00000000004000b0
[root@ecs-huawei2 ~]# ls
hello  hello.o  hello.s
[root@ecs-huawei2 ~]# ./hello
Hello World!
```

更多信息

- 更多信息请参考如下二维码

华为培训APP　企业技术支持APP　华为企业业务APP

学习推荐

- 华为官方网站
  - 在线学习：https://e.huawei.com/cn/talent/#/home
  - 企业业务：http://enterprise.huawei.com/cn/
  - 技术支持：http://support.huawei.com/enterprise/
- 热门工具
  - HedEx Lite
  - 网络资料工具中心
  - 信息查询助手

技术支持

热门工具

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and organization for a fully connected, intelligent world.

HUAWEI