# 汇编语言与逆向技术

## 第6章 国产鲲鹏CPU汇编语言编程

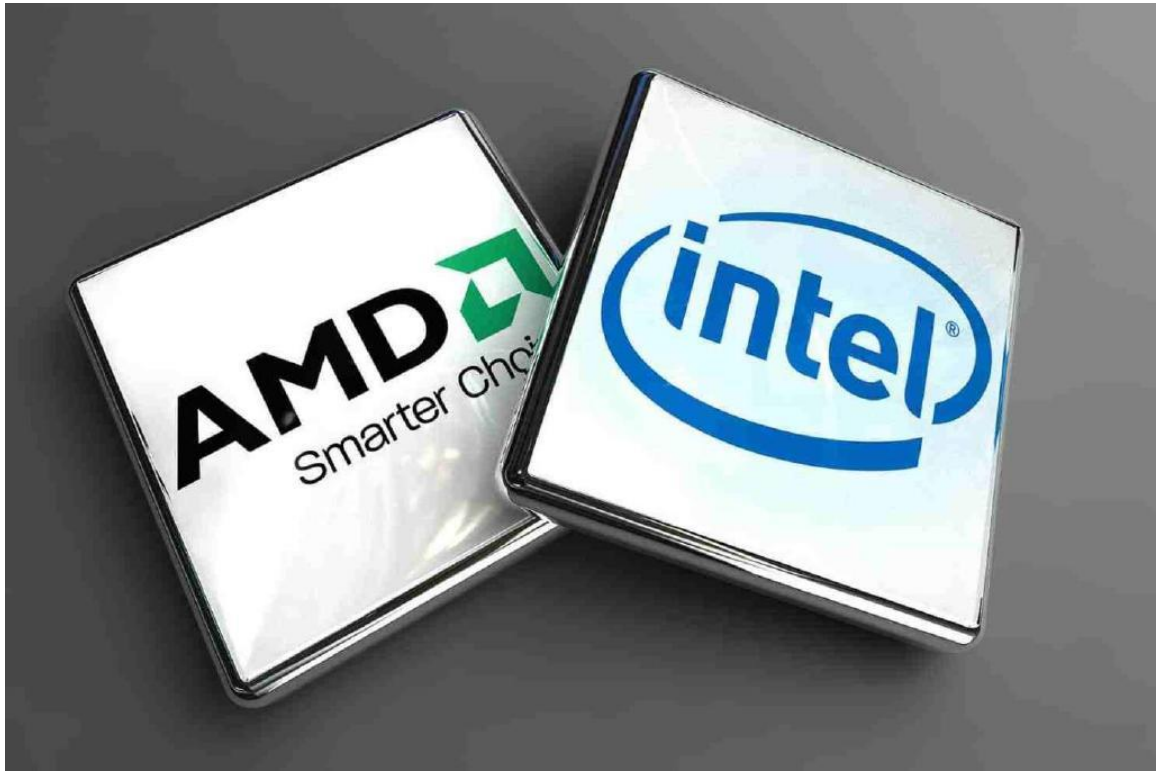
# 主观题 10分

设置

大家知道哪些国产的CPU和操作系统?

作答

# 国外CPU vs. 国产CPU





# 主观题 10分

设置

我们为什么要发展国产CPU、为什么要学习国产CPU的编程?

作答

南開大學 Nankai University

# 美国“对华芯片出口限制”

- 美国商务部宣布将华为及其70家公司列入出口管制实体名单之列
  - 如果没有美国政府的批准，华为将无法向美国企业购买元器件
  - Google、英飞凌、英特尔、高通、 AMD 、赛灵思等宣布停止和华为进行业务往来
  - 无线技术标准的Wi-Fi联盟、蓝牙技术联盟、制定SD存储卡标准的SD协会、PCIe标准组织PCI-SIG等将华为移出了会员名单。

# 鲲鹏CPU


华为鲲鹏920芯片是全球最强的ARM芯片，其性能可与英特尔至强（Xeon）8180相媲美

# CPU架构

- x86、x64架构和ARM架构是最主流的两种CPU架构
  - x86、x64架构高度垄断
  - ARM架构开放授权模式
- 华为和飞腾已获得永久性Arm V8架构指令层级授权，自主研发CPU的设计根基。
  - 提升我国自主创新、安全可控、产业生态、芯片性能等综合实力，逐渐替代Intel等国外CPU芯片

# 本章知识点

1. ARMv8架构的体系结构
2. ARM寻址方式
3. ARM指令集
4. ARM伪指令
5. ARM汇编语言程序结构
6. ARM编译与调试工具
7. 基础程序Hello-World示例

# 1. 基于ARMv8的处理器体系结构

# ARMv8-A的2种执行状态（Execution States）

- AArch32
- AArch64
- 指令集、寄存器不同

# ARMv8-A的执行状态（Execution States）

- AArch32
  - 支持A32（ARM）和T32（Thumb）两种指令集
  - 提供13个32位通用寄存器
  - 提供1个32位程序计数器PC、1个堆栈指针SP、1个链接寄存器LR
  - 提供32个64位寄存器用于增强SIMD向量和标量浮点运算
  - 一组处理状态PSTATE参数用于保存处理单元状态

# ARMv8-A的执行状态

- AArch64
  - 支持单一的A64指令集
    - 固定长度32bits的指令集
  - 提供31个64bit通用寄存器
  - 提供1个64位程序计数器PC、若干堆栈指针SP寄存器、若干异常链接寄存器ELR
  - 提供32个128位寄存器用于增强SIMD向量和标量浮点运算
  - 一组处理状态PSTATE参数用于保存处理单元状态

# 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 |

# ARMv8支持的数据类型

- 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

# ARMv8支持的指令集

- AArch64执行状态下
  - 只能使用A64指令集：32位等长指令字
- AArch32执行状态下
  - 可以使用A32指令集：32位等长指令字
  - 或T32指令集：16位和32位可变长指令字

# ARMv8架构的异常等级与安全模型

- 支持多级执行权限：异常等级的数字越大，软件执行权限越高

| 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

| 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} |
| | | 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： |

# ARMv8架构的异常处理

- 异常：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
- ARMv8-A的异常类型
  - 同步异常（Synchronous exception）：直接由执行指令或者尝试执行指令引起，且异常返回地址指明了引起异常的特定指令细节，例如debugging
  - 异步异常（Asynchronous exception）：由IRQ、FIQ这两个中断请求管脚引起的中断及系统错误引起的异常，例如网卡接收网络数据

# ARMv8架构的异常处理

| 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

- AArch64状态下，引起异常的几类事件：
  - 终止

    ——指令取指错误
  - 复位

    ——最高等级异常
  - 执行异常产生指令

    ——系统调用指令，引起软中断
  - 中断

    ——IRQ和FIQ，后者优先级更高

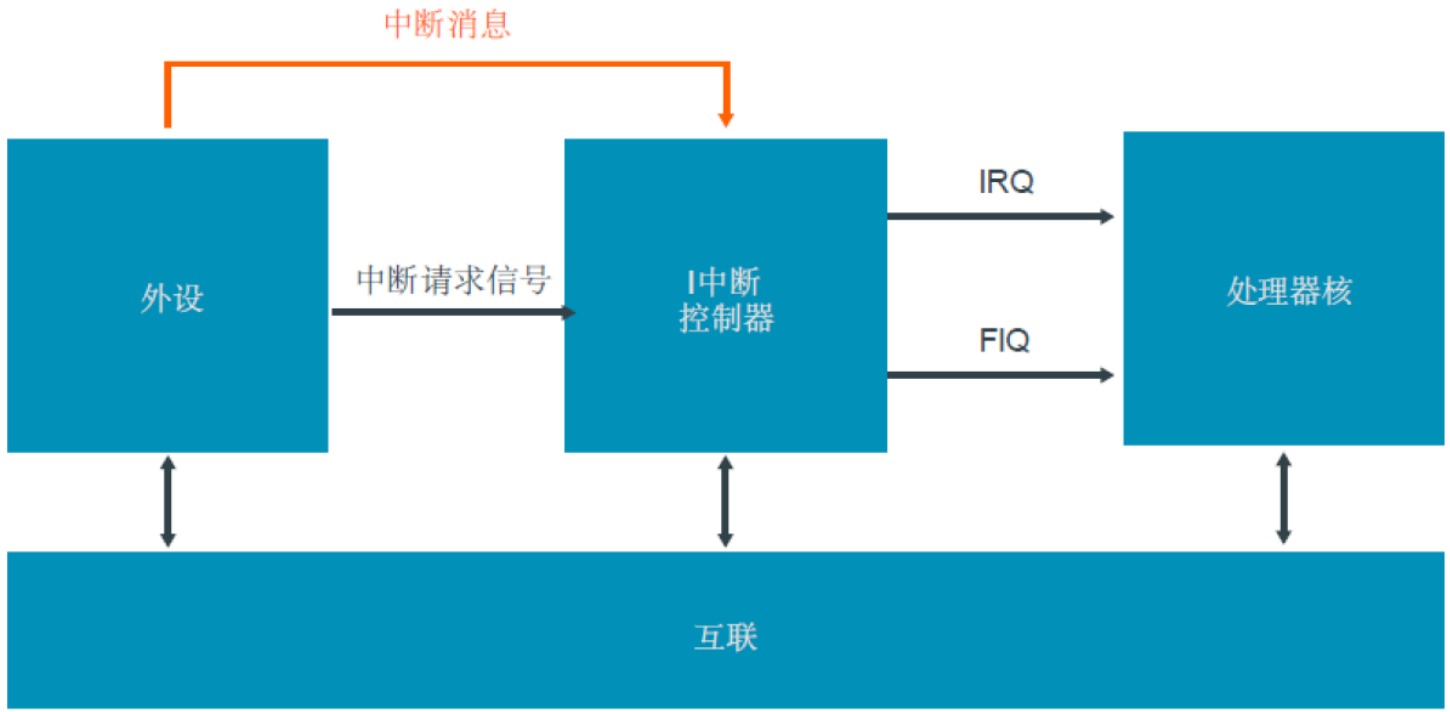
# ARMv8架构的异常处理

- ARMv8-A的异常处理
  1. 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
  2. 用新的处理器状态信息更新程序状态PSTATE
  3. 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架构的中断

- ARMv8架构的通用中断控制器（ GIC ）

  —ARM架构下，系统通过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

- 四种中断源

1. 共享外设中断

——可被连接到任何一个处理器核

2. 私有外设中断

——只属于某个处理器核的外设的中断请求，如通用定时器的中断请求

3. 软件产生的中断

——由软件写入中断控制器内的寄存器引发的中断请求，通常用于处理机间通信

4. 特定位置外设中断

——边沿触发的基于消息的中断

# 2. ARM寻址方式

# ARM寻址方式

- 寻址：找到存储数据或指令的地址
- ARM的寻址方式：

1. 立即数寻址
2. 寄存器寻址
3. 寄存器间接寻址
4. 基址寻址
5. 多寄存器寻址
6. 堆栈寻址

# 立即数寻址

- 立即数寻址指令中的地址码就是操作数本身
- 若操作数为常量，用#表示常量
- 0x或&表示16进制数，否则表示十进制数

例：

MOV R0, #0xFF000　　将立即数0xFF000装入R0寄存器

SUB R0, R0, #64　　R0减去64，结果放入R0

# 主观题 10分

设置

masm32位汇编语言，16进制数如何表示?

作答

# 寄存器寻址

- 根据寄存器编码获取寄存器内存储的操作数

例：

MOV R1, R2　　将R2的值存入R1

SUB R0, R1, R2　　将R1的值减去R2的值，结果保存到R0中

# 寄存器间接寻址

- 操作数从寄存器所指向的内存中取出，寄存器存储的是内存地址

例：

LDR R1, [R2]　　将R2指向的存储单元的数据读出，存入R1

STR R1, [R2]　　将R1的值写入到R2所指向的存储单元

SWP R1, R1, [R2]　　将R1中的值和R2所指向的存储单元的内容交换

注：LDR（Load Register）指令用于读取内存数据

STR（Store Register）指令用于写入内存数据

# 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后访问该地址空间

1. 前索引

```
LDR R0, [R1, #4]          // R0 ← mem32[R1 + 4]
```

R1中保存的地址+4，新地址中的值存入R0

# 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后访问该地址空间

2. 自动索引

LDR R0, [R1, #4]! // R0 ← mem32[R1 + 4]；R1 ← R1 + 4

在前索引的基础上，新地址回写进R1

注：！表示回写地址

# 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后访问该地址空间

3. 后索引

LDR R0, [R1], #4 // R0 ← mem32[R1]；R1 ← R1 + 4

R1中保存地址的内容写进R0，R1中保存的地址+4再写进R1

# 多寄存器寻址类型

- 多寄存器寻址也称为块拷贝寻址
- LDM（Load Multiple）多数据加载
  - 将多个地址上的值加载到多个寄存器上
- STM（Store Multiple）多数据存储
  - 将多个寄存器上的值存到多个地址上

# 多寄存器寻址模式

- IA(Increase After)每次传送后地址加4，寄存器从左到右执行
  - STMIA R0, {R1, LR} 先存R1，再存LR
- IB(Increase Before)每次传送前地址加4
- DA（Decrease After）每次传送后地址减4
- DB（Decrease Before）每次传送前地址减4

# 多寄存器寻址格式

- LDM mode Rn{!}, reglist{^}
- STM mode Rn{!}, reglist{^}
  - Rn: 基址寄存器，存储传输数据的起始地址，Rn不能为R15
  - ！：表示最后的地址写回到Rn中
  - reglist：寄存器列表，用“，”隔开，如{R1，R2，R3}
    - 最多16个寄存器
  - ^：不允许在用户模式和系统模式下运行

# 多寄存器寻址实例

- LDR R1, #0x10000000 //数据传输起始地址0x10000000
- LDMIB R1！，{R0, R4-R6} // 从左向右加载
  - R1地址加4，LDR R0, #0x10000004
  - R1地址加4， LDR R4, #0x10000008
  - R1地址加4， LDR R5, #0x1000000C
  - R1地址加4， LDR R6, #0x10000010
  - LDR R1, #0x10000010 // “!”，最后的地址写回R1寄存器

# 主观题 10分

设置

LDR R1, #0x10000000 //数据传输起始地址0x10000000
LDMIA R1！，{R0, R4-R6}
请写出LDMIA指令执行之后R6和R1寄存器的值？

作答

# 多寄存器寻址实例

- LDR R1, #0x10000000 //数据传输起始地址0x10000000
- LDMIA R1！，{R0, R4-R6} // 从左向右加载
  - LDR R0, #0x10000000， R1地址加4
  - LDR R4, #0x10000004， R1地址加4
  - LDR R5, #0x10000008， R1地址加4
  - LDR R6, #0x1000000C， R1地址加4
  - LDR R1, #0x10000010 // “!”，最后的地址写回R1寄存器

# 多寄存器寻址实例

- LDR R1, #0x10000000 //数据传输起始地址0x10000000
- LDR R4, #0x10
- LDR R5, #0x20
- LDR R6, #0x30
- STMIB R1, {R4-R6} //从左向右加载

- 相当于
  - R1地址加4，STR[R4], 0x10000004
  - R1地址加4，STR[R5], 0x10000008
  - R1地址加4，STR[R6], 0x1000000C

# 寄存器堆栈寻址

- 存储空间中的栈与寄存器组之间的批量数据传输
  - 使用R13（SP）作为栈指针，指向栈顶
  - 先进后出FILO方式访问
- 基本指令：LDM/STM

# 寄存器堆栈寻址

- 栈指针位置
  - FULL：栈指针指向栈顶元素（即最后一个入栈的数据元素）时称为FULL栈
  - EMPTY：栈指针指向与栈底元素相邻的一个可用单元时称为EMPTY栈
- 栈增长方向
  - Descending：当数据栈向内存地址减小的方向增长时，称为Descending栈
  - Ascending：当数据栈向内存地址增加的方向增长时，称为 Ascending栈

# 寄存器堆栈寻址

- 栈的4种管理方式
  - FD （Full Descending Stack）：满递减栈，入栈前栈指针减4
  - FA（Full Ascending Stack）：满递增栈，入栈后栈指针减4
  - ED （Empty Descending Stack）：空递减栈，入栈前栈指针加4
  - EA （Empty Ascending Stack）：空递增栈，入栈后栈指针加4

多选题 1分

设置

ARMv8支持4种栈的生长（管理）方式：FA、FD、EA、ED。大家回忆一下，x86 CPU支持哪种栈的生长方式？

A FA

B FD

C EA

D ED

提交

地址模式名称

| | | Ascending | | Descending | |
|---|---|---|---|---|---|
| | | Full | Empty | Full | Empty |
| Increment | Before | STMIB<br>STMFA | | | LDMIB<br>LDMED |
| | After | | STMIA<br>STMEA | LDMIA<br>LDMFD | |
| Decrement | Before | | LDMDB<br>LDMEA | STMDB<br>STMFD | |
| | After | LDMDA<br>LDMFA | | | STMDA<br>STMED |

# 寄存器堆栈寻址

- STMFD SP!, {R2-R4} （相当于入栈操作，寄存器的值从右往左存储）
  - [SP-4] ← R4
  - [SP-8] ← R3
  - [SP-12] ← R2
  - 最后的地址再写回SP
  - SP的初始值0x1000000C，执行之后SP的值是0x10000000

# 寄存器堆栈寻址

- LDMFD SP!, {R2-R4} （相当于出栈操作，注意对寄存器从左到右加载）
  - R2← [SP] ;
  - R3 ← [SP+4] ;
  - R4 ← [SP+8] ;
  - 因为有！，最后的地址写回SP，修改了它的值

# 主观题 10分

设置

SP寄存器的值是0x10000000，指令LDMFD SP!, {R2-R4}执行之后，SP寄存器的值是多少？

作答

# 3. ARM指令集

# ARM指令集

- GNU ARM汇编语言语法格式：

**[<label>:][<instruction or direction or pseudo-instruction>]@comment**

instruction 指令

direction 操作数

pseudo-instruction 伪指令

<label>:为标号，GRM ARM汇编中，任何以冒号结尾的标识符都认为是一个标号

comment为语句的注释

# ARM指令集

- GNU ARM汇编语言语法格式：

**[<label>:][<instruction or direction or pseudo-instruction>]@comment**

注：

- ARM指令、伪指令、伪操作、寄存器名不可大小写混用，可全部大写或全部小写
- 可以在行末用“\”表示换行

# GNU ARM汇编语法格式

- 局部变量定义的语法格式：N{routname}
  - N为0~99之间的数字
  - routname为当前局部范围的名称，通常为该变量作用范围的名称（通过ROUTE伪操作定义）

# GNU ARM汇编语法格式

- 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（由ROUT伪操作定义）
  - 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

# GNU ARM汇编语法格式

- 特殊字符和语法
  - 代码中的注释符号：‘@’
  - 整行注释符号：‘#’
  - 语句分离符号：‘;’
  - 立即数前缀：‘#’或‘$’

# 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}<br>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 |

# A64指令特点

- A64指令编码宽度固定32bit
- 31个（X0-X30）个64bit通用寄存器（用作32bit时是W0-W30）
- 支持48bit虚拟寻址空间
- PC指针不能作为数据处理指令或load指令的目的寄存器，X30通常用作LR

# 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 | CBX X1, label //如果X1==0则跳转到label |
| TBNZ | TBNZ X1, #3 label //如果X1[3]!=0则跳转到label |
| TBZ | TBZ X1, #3 label //如果X1[3]==0则跳转到label |

# 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） |

# 跳转指令

- 函数调用的产生和返回指令

| 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 |

# 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：

MRS R0, CPSR　　状态寄存器CPSR的值存入寄存器R0

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

# 算术运算指令

| 指令 | 说明 | 指令 | 说明 |
|---|---|---|---|
| ADDS | 加法指令，若S存在，则更新调键位flag | NEG | 取负数运算 |
| SUBS | 带进位的加法，若S存在，则更新条件位flag | MADD | 乘加运算 |
| CMP | 减法指令，若S存在，则更新条件位flag | MSUB | 乘减运算 |
| 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 | MUL | 乘法运算 |
| RSB | 逆向减法，操作数2减去操作数1，结果送到目的寄存器Xt/Wt | SMADDL | 有符号乘加运算 |
| RSC | 带借位的逆向减法指令，操作数2减去操作数1，再减去标志位C的取反值，结果送目的寄存器Xt/Wt | SDIV | 有符号除法运算 |
| CMP | 比较相等指令 | UDIV | 无符号除法运算 |
| CMN | 比较不等指令 | | |

# 算数运算指令

例：

## 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 |

# 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 |

# 逻辑运算指令

例：

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

# 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，保存其他bit不变 |

# 数据传输指令

例：

## MOV 数据传送指令

MOV R0, #0xFF000　　立即寻址，将立即数0xFF000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

## MOVN 数据取反传送指令

MOVN R0, #0　　R0=-1

# 地址生成指令

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

# 位段移动指令

BFM、SBFM、UBFM、BFI、BFXIL、SBFIZ、SBFX、UBFX、UBFZ等

# 移位运算指令

| 指令 | 说明 |
| --- | --- |
| ASR | 算数右移 >> （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、SXTH、<br>UXTB、UXTH | 字节、半字、字符号/0扩展移位运算 |

# 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 |

# SIMD指令简介

- 单指令多数据流，能够复制多个操作数并打包在大型寄存器
- SIMD可以以同步的方式，同一时间内执行同一条指令
- 非常适合多媒体应用等数据密集型运算

# 4. ARM伪指令

# ARM伪指令

- 伪指令是编译器支持的指令，而非硬件芯片支持的指令
  - 编译器把伪指令转化成对应芯片支持的指令
  - 伪操作：

    数据定义伪操作、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……

# 数据定义伪操作

| | | |
|---|---|---|
| .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” |

# 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伪指令可以嵌套使用。

## .macro . mend

——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，在程序中通过宏指令可以多次调用该段代码。宏操作可以使用一个或多个参数。

# 汇编控制伪操作

## .macro .mend示例

```
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
```

# 杂项伪操作

| 伪操作 | 语法 | 说明 |
| --- | --- | --- |
| .arm | .arm | 定义以下代码使用ARM指令集编译 |
| .thumb | .thumb | 定义以下代码使用Thumb指令集编译 |
| .section | .section expr | 定义一个段。expr可以是.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 |

# ADR伪指令

- 小范围地址读取伪指令

——地址值是字节对齐（8位）时，使用相对偏移取值范围为-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, label*

# ADRL伪指令

- 中等范围地址读取伪指令

——地址值是字节对齐（8位）时，使用相对偏移取值范围为-64~64KB，当地址值是字对齐（32位）时，取值范围为-256~256KB。

- 语法格式：

ADRL{cond} register, label

*ADRL R0, label*

# 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*

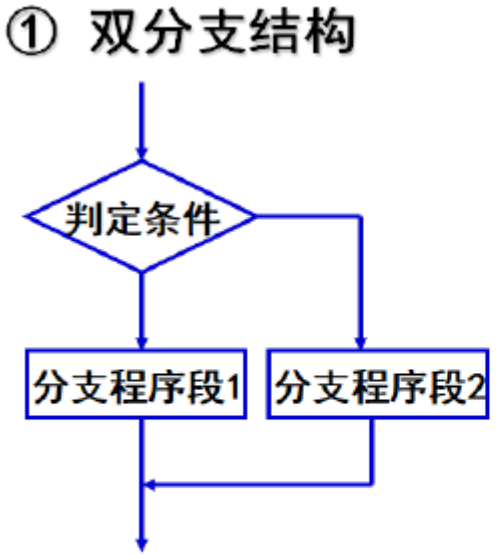
# 5. ARM汇编语言程序结构

# 顺序结构

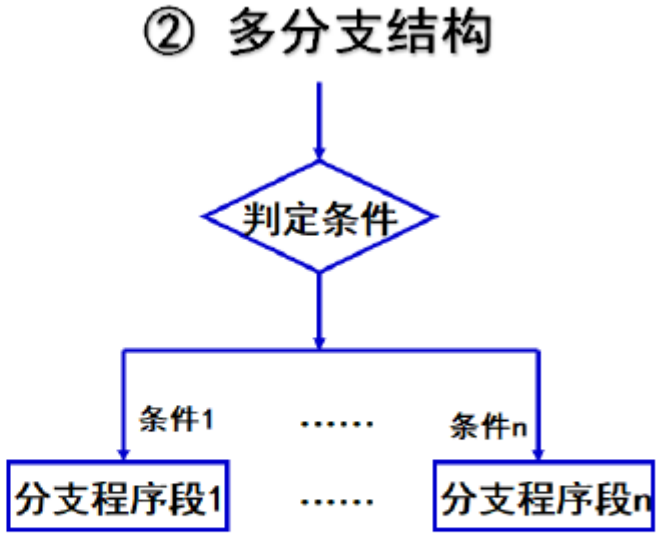
- 最简单、最基本的一种程序结构形式
- 由程序的开头顺序执行直到程序结束为止，执行过程中没有任何分支

# 分支结构

- 事先把各种可能出现的情况和处理方法写在程序里，由计算机做出判断并跳转或调用相应的程序处理
- 运行方向是向前的，在确定的条件下，只能执行多个分支中的一个分支




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

# 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
```

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

# 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
```

# 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：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，调用子程序时需要保留内容

# 6. ARM编译与调试工具

# gcc编译器

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

# 编译与调试流程

- 使用SSH工具，输入用户名和密码登录

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

C:\Users\1wx941162>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 ~]#
```

# 编译与调试流程

- 更新编译环境

```
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 “souce /opt/rh/devtoolset-7/enable” >>/etc/profile
```

# 编译与调试流程

- 使用“gcc -v”命令可以输出gcc版本

```
[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)
```

- 使用gcc来编译c语言程序

# 7. 基础程序Hello-World示例

# hello-world示例程序

- 实验目的：实现ARM平台精简指令集（RISC）编写的hello-world程序的编译和运行。
- 实验内容：两次使用软中断指令svc来进行系统调用。第一次使用svc指令在屏幕上打印出一个字符串“Hello”；第二次使用svc指令来退出当前程序。

# hello-world示例程序

```
.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
```

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

```
.globl _start
_start:
        mov x0, 0     // stdout has file
descriptor 0
        ldr x1, =msg  // buffer to write
        mov x2, len   // size of buffer
        mov x8, 64    // sys_write() is at
index 64 in kernel functions table
        svc #0        // generate kernel call
sys_write(stdout, msg, len);

        mov x0, 123 // exit code
        mov x8, 93  // sys_exit() is at index
93 in kernel functions table
        svc #0      // generate kernel call
sys_exit(123);

.data //data section
msg:
    .ascii      "Hello World!
"
len = . - msg
```