从0学ARM

第一期

一口Linux

想入门和进阶ARM

请加关注一口君的公众号：一口Linux

视频配套资料后台回复：arm

公众号：一口Linux

计划

- 第一期 ARM入门，ARM架构、ARM指令
- 第二期 以一款开发板为例，讲解如何编写对应的驱动：LED、KEY、PWM、ADC、RTC、看门狗、I2C、MPU6050、SPI
- 第三期 uboot 启动、dm9000等

ARM入门第一期视频目标

- 1. 了解常见的ARM相关概念（架构、指令集、soc）
- 2. 掌握ARM指令学习的环境搭建、程序调试
- 3. ARM常用的指令
- 4. 掌握程序跳转
- 5. 掌握汇编程序和C语言程序之间调用问题
- 6. 掌握ARM一些异常处理流程

ARM

01

嵌入式工程师到底要不要学习ARM汇编指令？

配套文章

- 《ARM系列文章合集》
- 《嵌入式工程师到底要不要学习ARM汇编指令？》
- 《到底什么是Cortex、ARMv8、arm架构、ARM指令集、soc？一文帮你梳理基础概念【科普】》
- 《2. 从0开始学ARM-CPU原理，基于ARM的SOC讲解》

- ppt会实时更新

- 公众号：一口Linux　　回复 arm
- 个人VX:　yikoupeng

什么是汇编?

- 机器语言
- **机器语言是机器指令的集合**，机器指令展开来讲就是一台机器可以正确执行的命令。

- **电子计算机的机器指令是一系列二进制数字。**

纸带机

早期的程序设计均使用机器语言。程序员们将用 0、1 数字编程的程序代码打在**纸袋**或**卡片**上，1打孔，0不打孔，再将程序通过**纸带机**或**卡片机**输入计算机，从而进行运算。

机器语言的弊端

- 机器码的晦涩难懂和不易查错

```
10111000000000000000000011
00000101000000000000110000
00101101000000000000000101

10110000000000000000000011
00000101000000000000110000
00010111010000000000000101
```

汇编语言

- 用助记符代替机器指令的操作码，
- 用地址符号或标号代替指令或操作数
- 的地址。

```
1  .text
2  .global _start
3  _start:
4          b       reset
5          ldr     pc,_undefined_instruction
6          ldr     pc,_software_interrupt
7          ldr     pc,_prefetch_abort
8          ldr     pc,_data_abort
9          ldr     pc,_not_used
10         ldr     pc,=irq_handler
11         ldr     pc,_fiq
12
13 _undefined_instruction: .word  _undefined_instruction
14 _software_interrupt:    .word  _software_interrupt
15 _prefetch_abort:        .word  _prefetch_abort
16 _data_abort:            .word  _data_abort
17 _not_used:              .word  _not_used
18 _irq:                   .word  irq_handler
19 _fiq:                   .word  _fiq
20
21
22 reset:
23
24     ldr r0,=0x40008000
25     mcr p15,0,r0,c12,c0,0       @ Vector Base Address Register
26
27
28 init_stack:
29         ldr     r0,stacktop         /*get stack top pointer*/
30
31     /********svc mode stack********/
```

要不要学习汇编?

- 误区1:
- 很多人认为汇编过时了
- 误区2:
- 工作中完全用不到
- 误区3:
- 花太多精力学习所有的汇编指令

一些值得思考问题

- 1. 系统是如何启动的？上电之后做什么工作？
- 2. 链接C语言的函数是如何调用的，参数是如何传递的？
- 3. 中断产生之后，cpu是如何处理的？
- 4. 如何使能、关闭中断？
- 5. 多核处理器是如何分配进程到某个核上运行的？
- 6. MMU是如何实现的？
- 7. 系统调用是如何实现的？
- 8. 编译出来程序为什么能在不同的地址上正确执行？

汇编重要性

- 系统启动、上电代码都是汇编
- 想了解指针的本质、函数名的本质，就要深入汇编级代码，通过反汇编，看底层指令是如何对C语言的高级特性进行处理的
- 要想知道如何真正提高程序运行效率，必须看最终的汇编代码
- 一些不能说的事情，你懂的。。。

如何学习ARM?

- 1. 了解CPU、SOC、总线、寄存器、异常、内存、中断、并发、调度基础知识
- 2. 掌握基本核心汇编指令
- 3. 熟练掌握C语言
- 4. 能看懂基本的电路图：数据线、信号线，常见的元器件
- 5. 掌握常见的硬件控制器原理，并给对应的外设编写驱动程序

最后

- 学习汇编，其实不仅仅是学习了一种语言，而是在学习ARM架构。
- 掌握了该课程，单片机基本不在话下

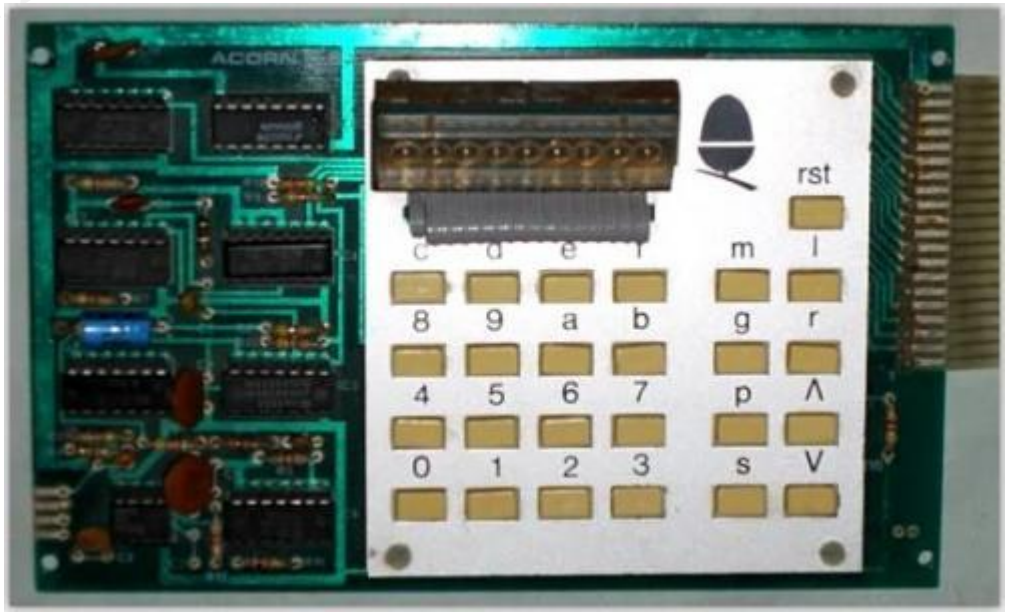
ARM®

02

什么是ARM?

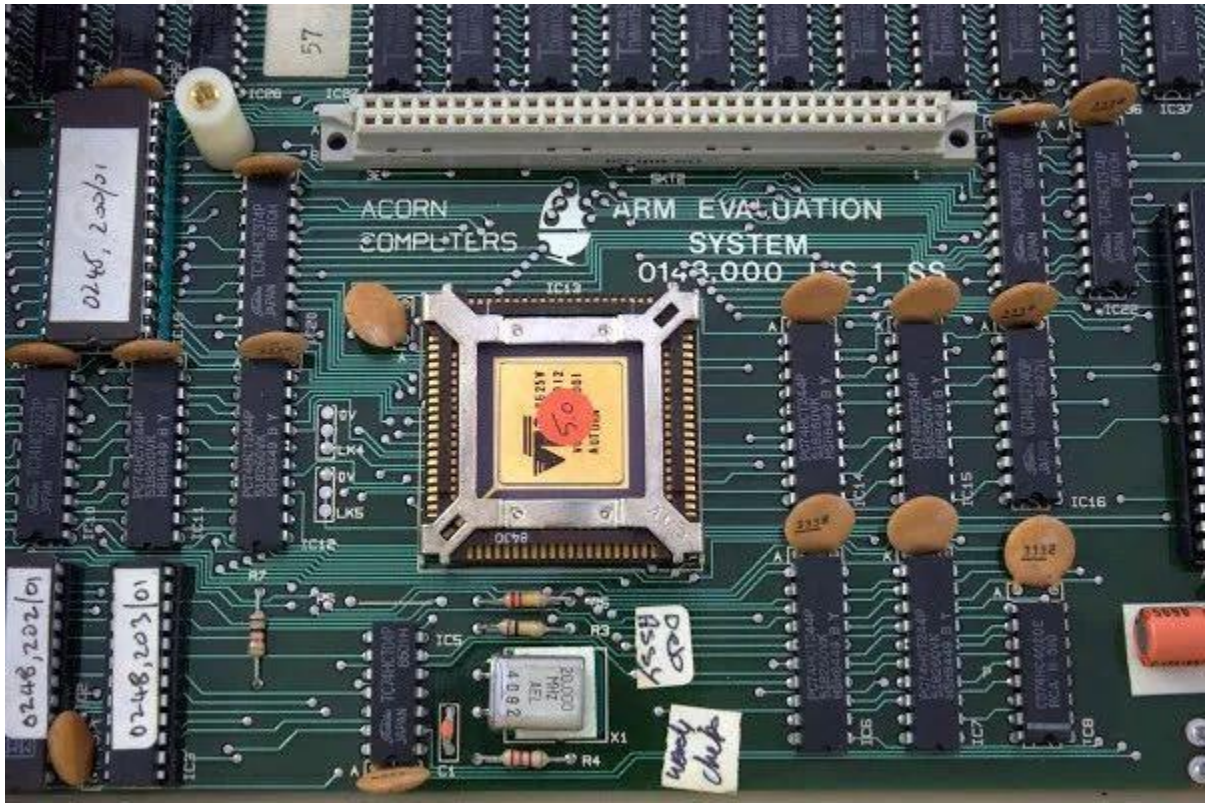
缘起

- 1978年，一个名叫Hermann Hauser的奥地利籍物理学博士，还有他的朋友，一个名叫Chris Curry的英国工程师成立了一家名字叫“CPU”（Cambridge Processor Unit）的公司。
- 他们接的第一份订单：制造赌博机
- 的微控制器系统，
- Acorn System 1



Acorn RISC Machine

- 1981年， Acorn为英国广播公司BBC设计了一款微处理器：**Acorn RISC Machine，用于**在整个英国播放一套提高电脑普及水平的节目



成立ARM公司

- 1990年，Acorn为了和苹果合作，专门成立了一家公司，名叫ARM(Advanced RISC Machines)。

- ARM公司既不出产芯片也不出售芯片，它只出售芯片技能授权。

上市

- 1998年4月17日，业务飞速发展的ARM控股公司，同时在伦敦证交所和纳斯达克上市。

腾飞

- 2007年，划时代产品-iPhone 问世。
  而第一代iPhone，正是使用了ARM设计、三星制造的芯片。

- 2008年，谷歌推出了Android（安卓）系统，基于该系统的的手机CPU也是基于ARM指令集。

辗转反侧

- 2016年7月18日消息，日本软银以234亿英镑（约合310亿美元）的价格收购英国芯片设计公司ARM。

- 2020年9月14日，英伟达正式宣布将以400亿美元的价格从软银手中收购ARM公司。根据协议，英伟达将向软银公司支付价值215亿美元的英伟达股票，以及120亿美元现金。

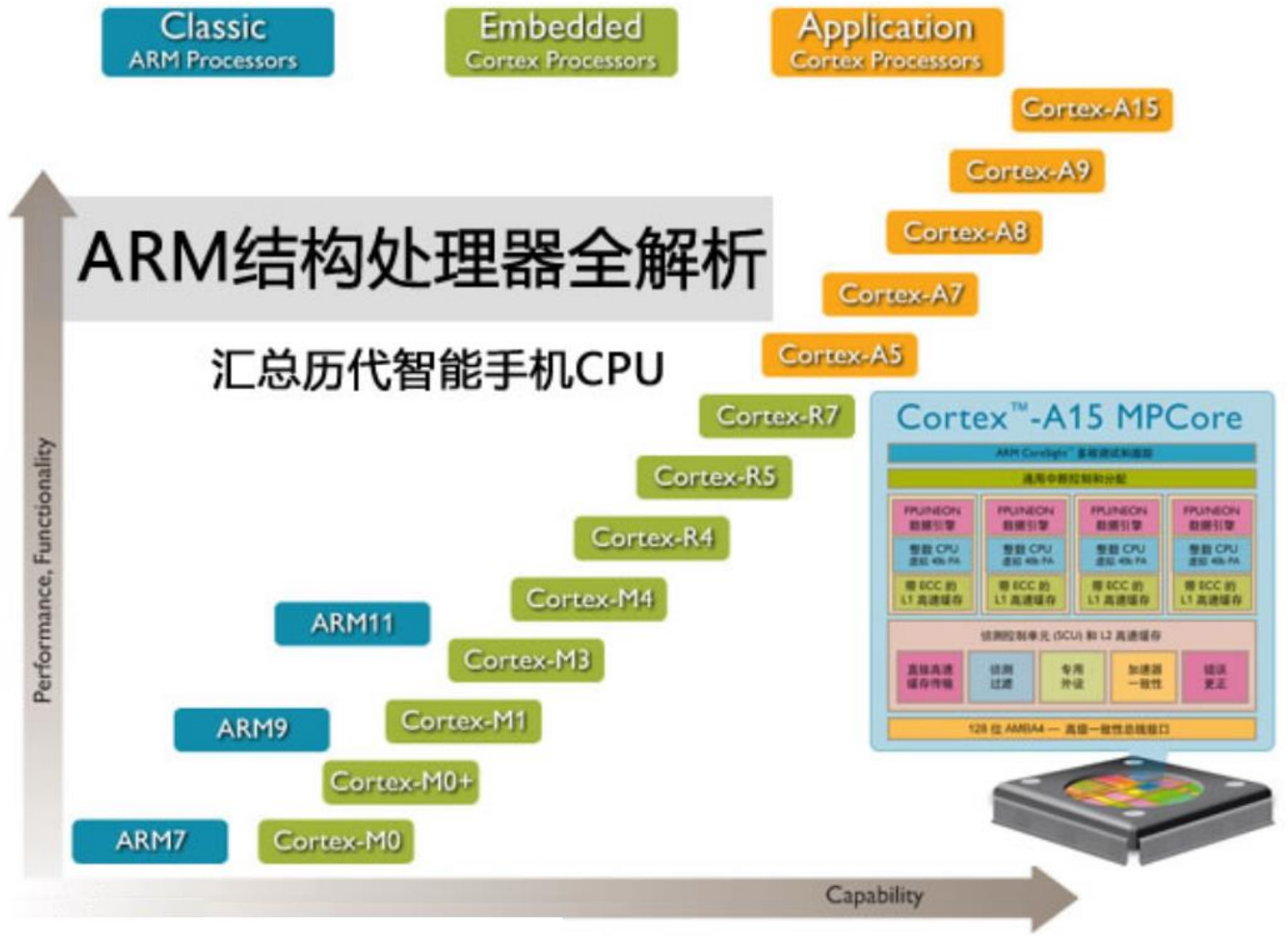
- 目前中国和欧盟要花时间对这笔交易进行评估

ARM内核

- ARM内核：
- 包括了寄存器组、指令集、总线、存储器映射规则、中断逻辑和调试组件等。
- 内核是由ARM公司设计并以销售方式授权给个芯片厂商使用的（ARM公司本身不做芯片）。
- 比如Cortex A8、A9都是ARMv7a 架构;Cortex M3、M4是ARMv7m架构;
- 前者是处理器（就是内核），后者是指令集的架构（也简称架构）。

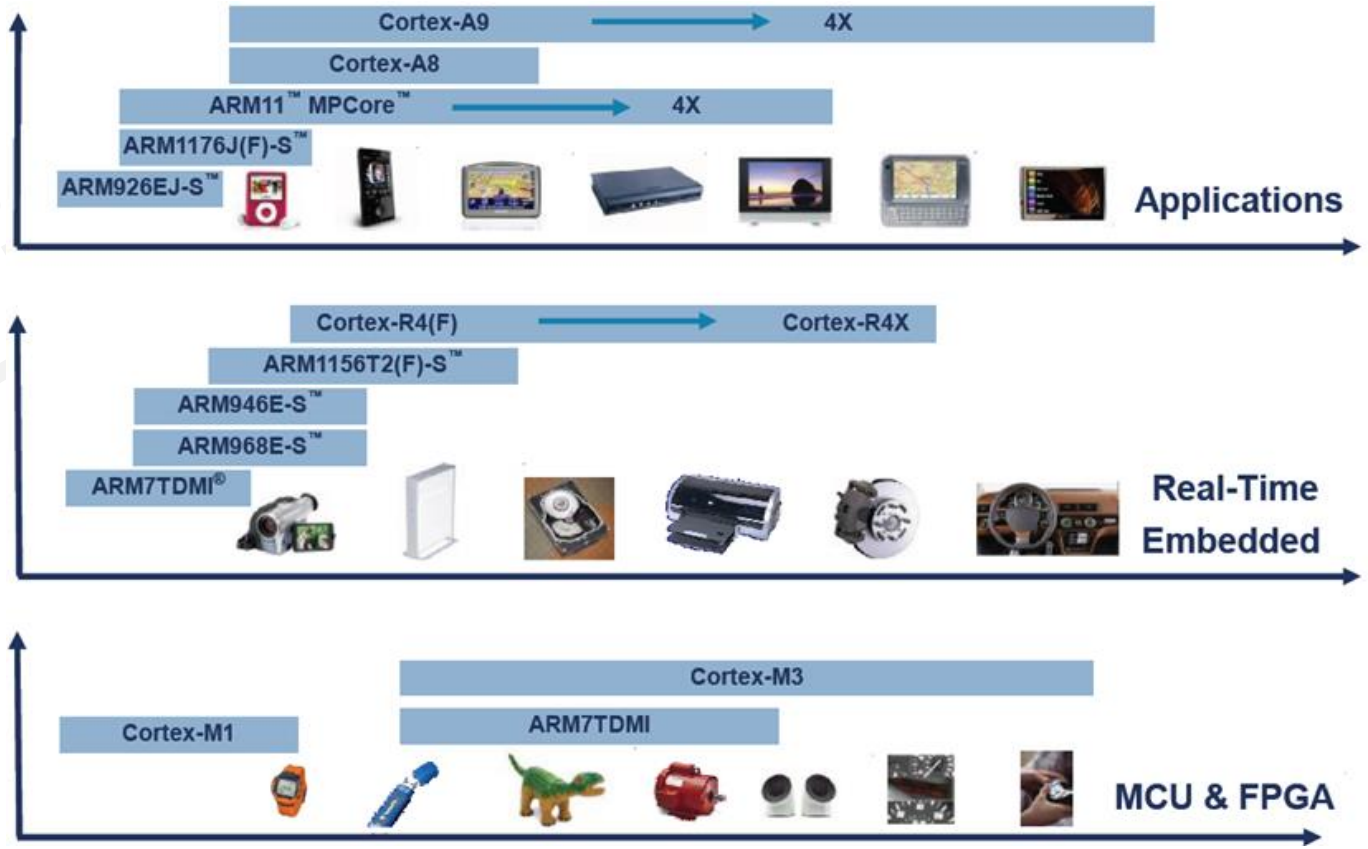
命名的改变

- ARM11芯片之后，也就
- 是从ARMv7架构开始，
- ARM的命名方式有所改变。



Cortex系列

- 新的处理器家族，改以Cortex命名，
- 并分为三个系列，分别是
- Cortex-A，
- Cortex-R，
- Cortex-M



Cortex系列

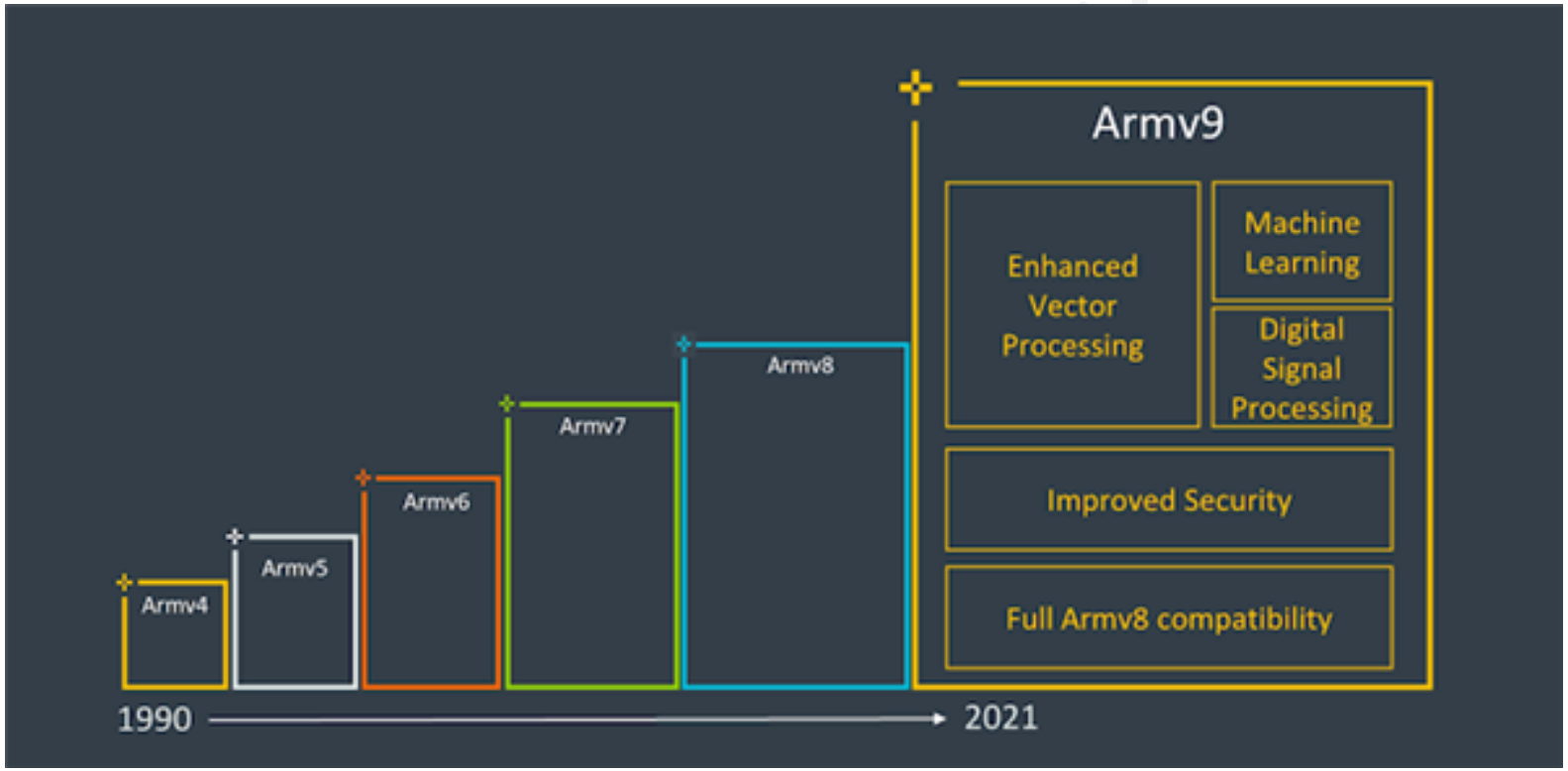
- Cortex-A系列（A：Application）
- 针对日益增长的消费娱乐和无线产品设计，用于具有高计算要求、运行丰富操作系统及提供交互媒体和图形体验的应用领域，如智能手机、平板电脑、汽车娱乐系统、数字电视，智能本、电子阅读器、家用网络、家用网关和其他各种产品。。

- Cortex-R系列 （R：Real-time）
- 针对需要运行实时操作的系统应用，面向如汽车制动系统、动力传动解决方案、大容量存储控制器等深层嵌入式实时应用。

- Cortex-M系列（M：Microcontroller）
- 该系列面向微控制器领域，主要针对成本和功耗敏感的应用，如智能测量、人机接口设备、汽车和工业控制系统、家用电器、消费性产品和医疗器械等。

ARM指令集架构

- 从1985年ARMv1架构诞生起，到2021年，
- ARM架构已经发展到了ARMv9。



Cortex系列

Cortex-A53、Cortex-A57两款处理器属于Cortex-A50系列，首次采用64位ARMv8架构。

2020年ARM发布了一款全新的CPU架构Cortex-A78，是基于ARMv8.2指令集。

2021年Cortex-X2、Cortex-A710和Cortex-A510基于ARMv9架构

总结

1. ARM前身Acorn公司设计的第一款微处理器，叫ARM：Acorn RISC Machine
2. ARM公司的名字，叫ARM：Advanced RISC Machines
3. ARM处理器名字：
   - 以前叫ARM9、ARM11
   - 新的命名规则改以Cortex命名，分别是Cortex-A，Cortex-R，Cortex-M
   - 这三个字母A、R、M合到一起又是ARM
4. ARM指令集，就是ARM架构，比如ARMv8，每个处理器都需要依赖一定的ARM架构来设计

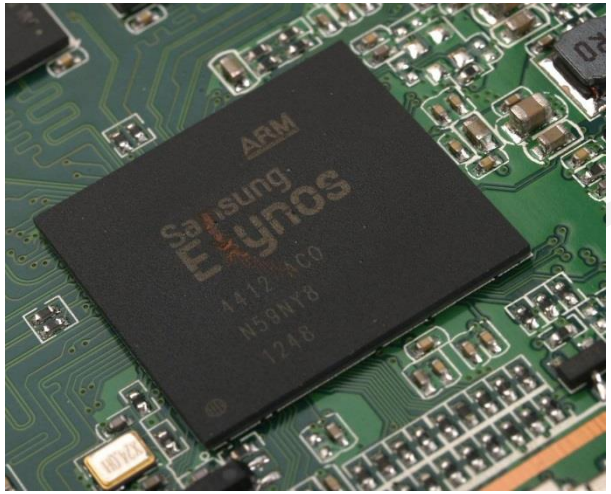
ARM

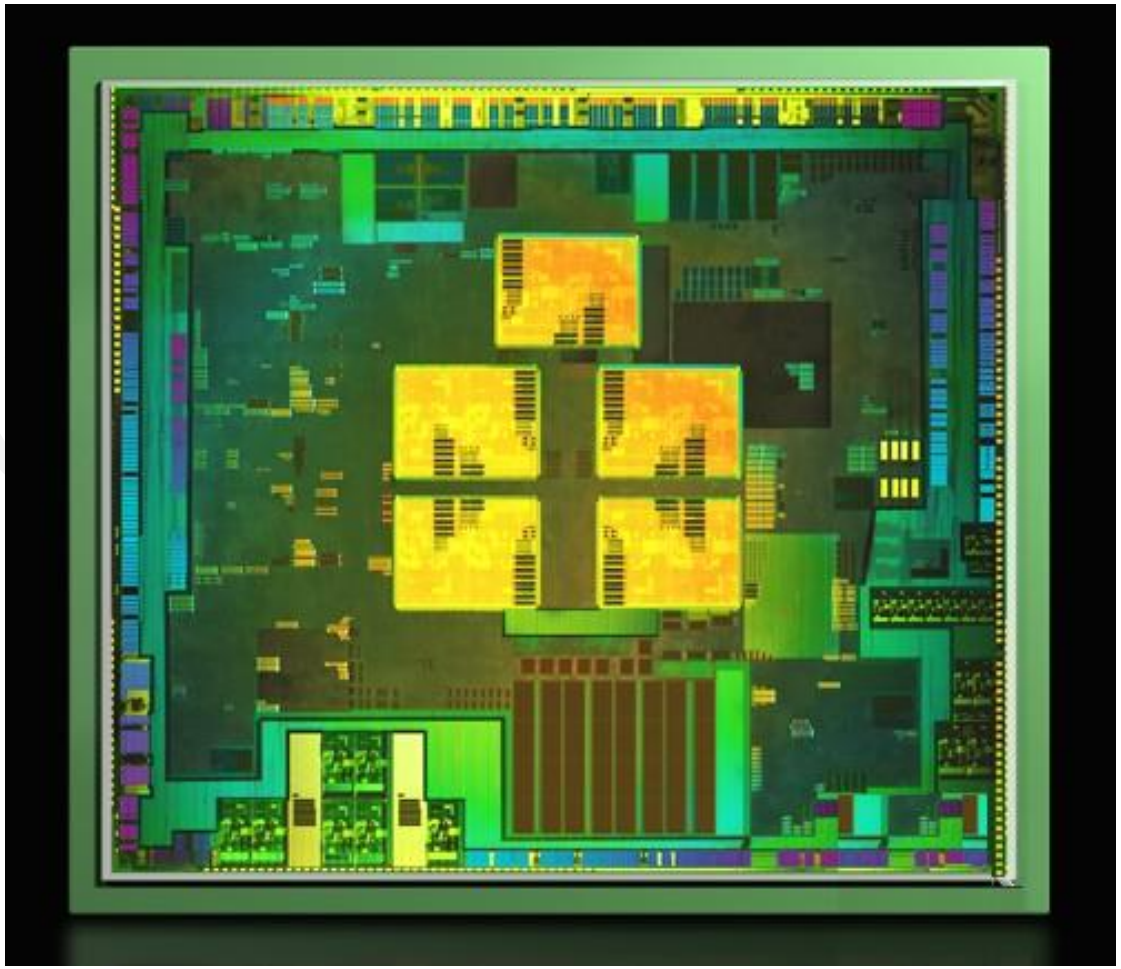
03

什么是SOC、ARM授权?

SOC

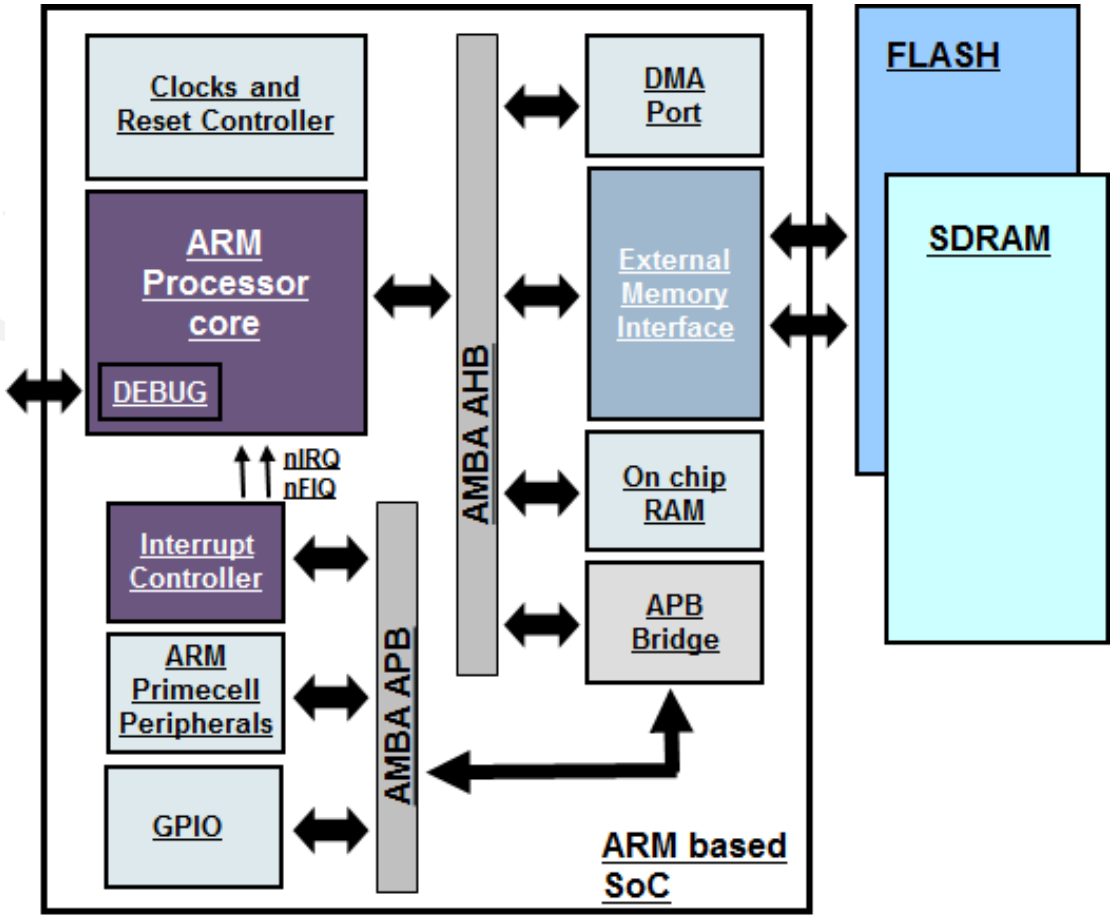
- SoC的全称叫做：
- System-on-a-Chip，
- 把系统都做在一个芯片上





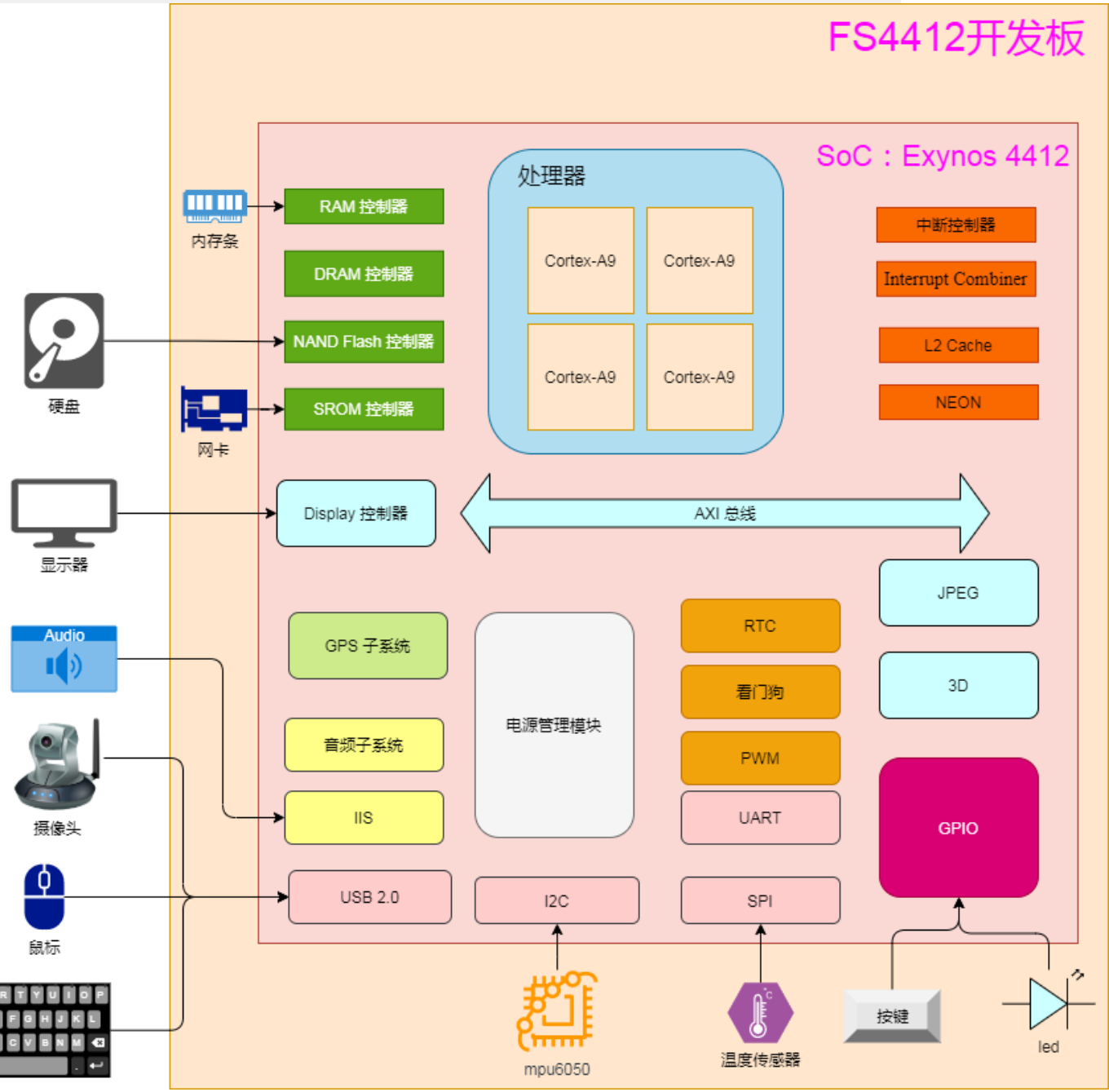
一个典型的基于ARM的Soc架构

- ARM Processor core 处理器核
- Clocks and Reset Controller 时钟和复位电路
- Interrupt Controller 中断控制器
- ARM Propherals 外部设备
- GPIO
- DMA Port
- External Memory Interface 外部内存接口
- On chip RAM 偏上RAM
- AHB、APB总线



SOC实例

- 4（quad）个Cortex-A9处理器
- 1MB的 L2 Cache
- Interrupt Controller 中断控制器，管理所有的中断源
- Interrupt Combiner 中断控制器，管理soc内的一些中断源
- NEON ARM 架构处理器扩展结构，旨在通过加速多媒体(video/audio)编解码，用户界面，2D/3D图形及游戏来提高人对多媒体的体验
- DRAM、Internal RAM、NAND Flash、SROM Controller 各种存储设备的控制器
- SDIO、USB、I2C、UART、SPI等总线
- RTC、Watchdog Timer
- Audio Subsystem 声音子系统
- IIS(Integrate Interface of Sound)接口，集成语音接口
- Power Management电源管理
- Multimedia Block 多媒体模块



举例：华为P50

IT技术网	麒麟9000E	麒麟9000L
制作工艺	5nm	
CPU架构	1*A77@3.13GHz 3*A77@2.54GHz 4*A53@2.05GHz	1*A77@2.86GHz 3*A77@2.54GHz 4*A53@2.05GHz
GPU架构	22 核 Mali-G78	18 核 Mali-G78
NPU架构	擦欧双核设计，分别是一大核一小核	仅保留了一颗NPU 大核
芯片基带	巴龙5000 5G基带	

armv8

ARM授权

- 如何来理解ARM授权呢?
- 就比如我们制造汽车，ARM公司相当于拥有最先进的的‘发动机’设计方案，
- 但是他不‘生产发动机’，而是把设计方案授权给各大‘汽车厂商’生产，赚来的钱继续研发更先进的‘发动机’。

ARM授权分类

- 分为架构层级授权、内核层级授权(ip核授权)、使用层级授权

- 一个公司若想使用ARM的内核来做自己的处理器，比如ST、苹果、三星、TI、高通、华为等等，必须向ARM公司购买其架构下的不同层级授权，根据使用需要购买相应的层级授权。

1.架构层级授权

- 是指可以对ARM架构进行大幅度改造，甚至可以对ARM指令集进行扩展或缩减，

- 苹果就是一个很好的例子，在使用ARMv7-A架构基础上，扩展出了自己的苹果swift架构

2.内核层级授权

- 是指可以以一个内核为基础然后在加上自己的外设，比如USART、GPIO、SPI、ADC等等，最后形成了自己的MCU，

- 这种公司很多，比如三星、TI;

3.使用层级授权

- 要想使用一款处理器，得到使用层级的授权是最基本的，这就意味着你只能拿别人提供的定义好的ip来嵌入在你的设计中，不能更改人家的ip，也不能借助人家的ip创造自己的基于该ip的封装产品。

授权举例

- 我写了一篇文章

◆架构层级授权

可以拿去修改后使用

◆内核级授权

我告诉乙，你可以在你的文章中引用我的文章

◆使用层级授权

只能对我的文章进行转发，不能更改，不能添油加醋

ARM®

04

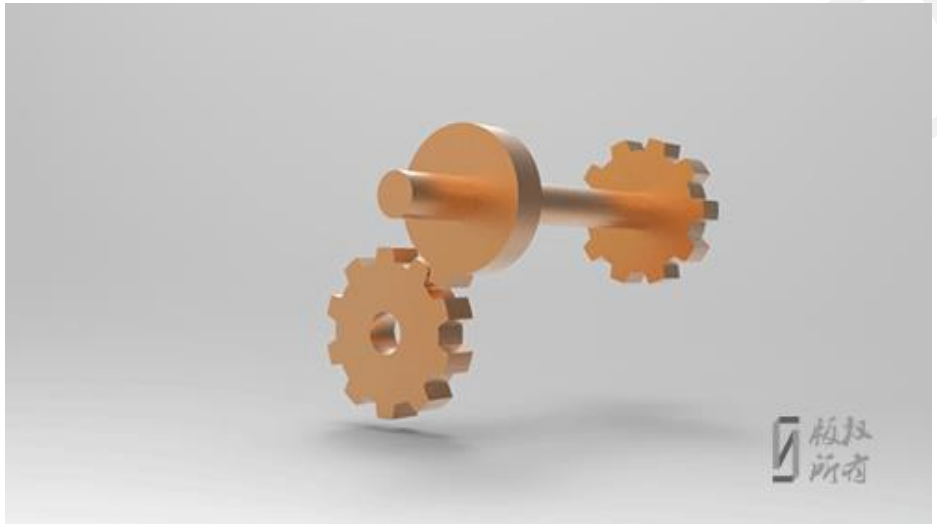
计算机历史

参考文章

- 《图灵、冯诺依曼谁才配得上计算机之父？》
- 《2. 从0开始学ARM-CPU原理，基于ARM的SOC讲解》

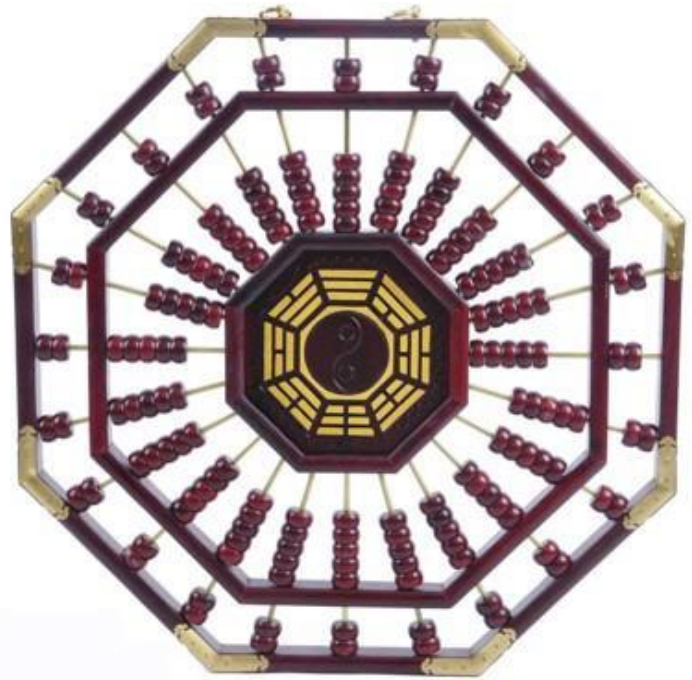
第一台机械计算机

- **契克卡德计算钟（Rechenuhr）**
- 研制时间：1623年~1624年

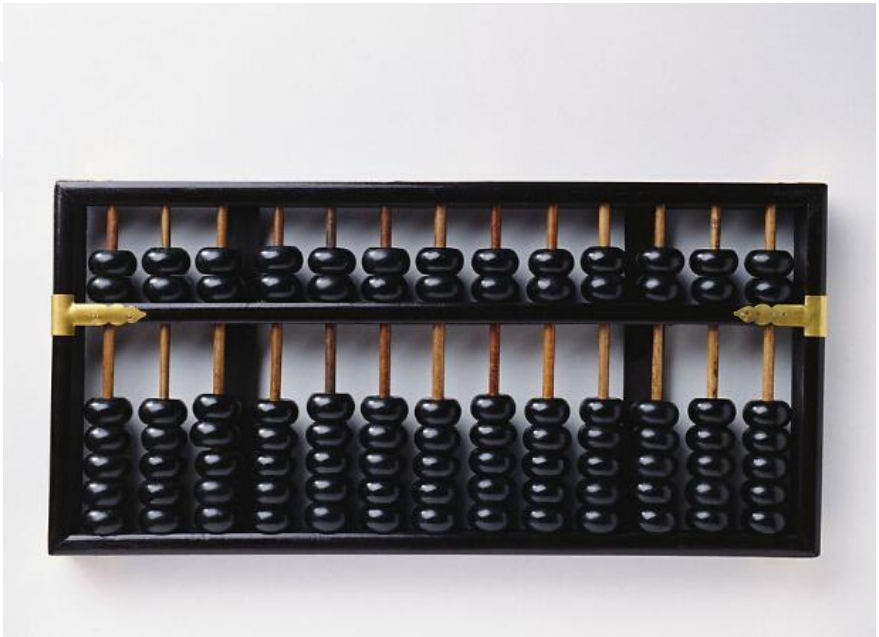


算盘

文王桃木算盘



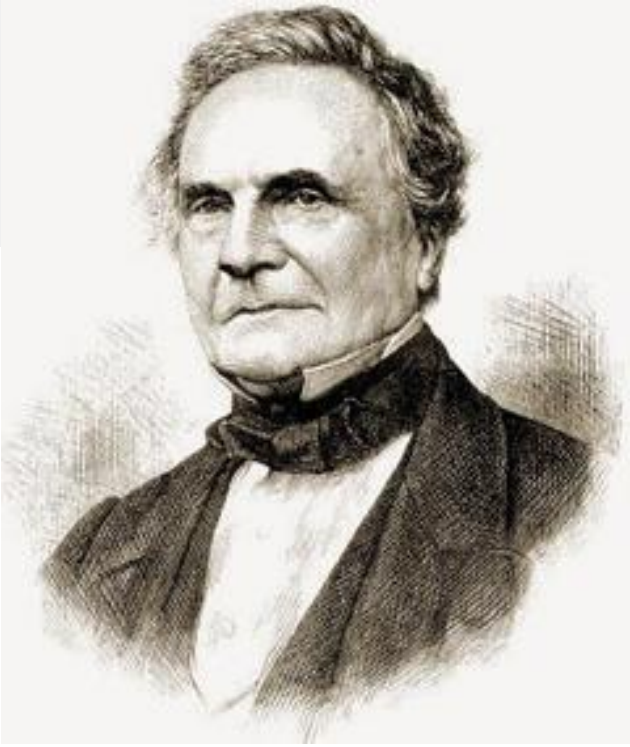
算盘-16进制



四位“计算机之父”

- **巴贝奇** 通用计算机之父
- **图灵** 计算机科学之父
- **约翰·阿坦那索夫** 电子计算机之父
- **冯·诺依曼** 现代计算机之父

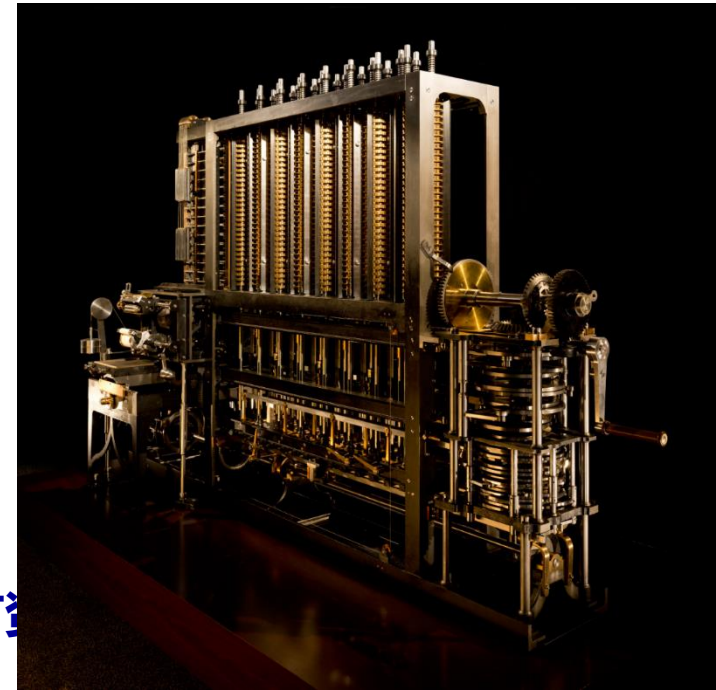
巴贝奇



- 巴贝奇

机械计算机之父，英国贵族，曾孤军奋战下造出的第一台差分机，运算精度达到了6位小数，后来又设计了20位精度的差分机，其设计理念已经达到了机械设计登峰造极的境界。

1985～1991年，伦敦科学博物馆为了纪念巴贝奇诞辰200周年，根据其1849年的设计，用纯19世纪的技术成功造出了差分机2号。



巴贝奇堪称上个世纪最强大脑，他的大脑现保存在英国科学博物馆。

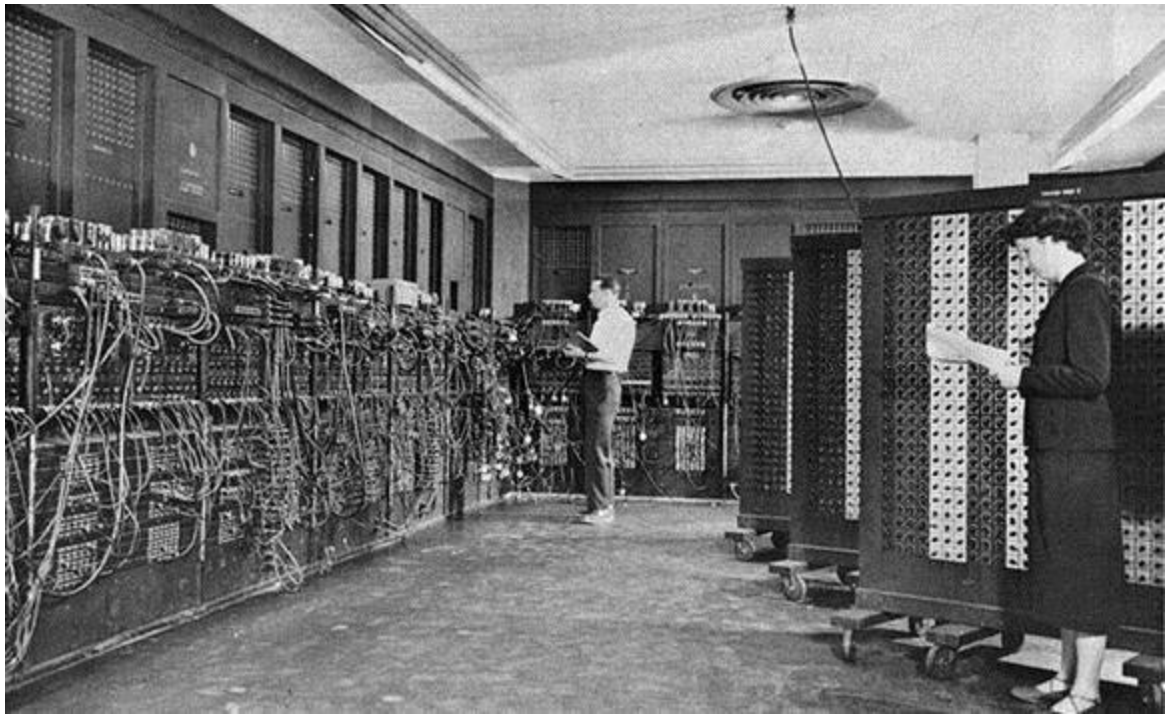
ADA

- 原名奥古斯塔·阿达·拜伦（Augusta Ada Byron），通称阿达·洛芙莱斯（Ada Lovelace），
- 是著名英国诗人拜伦之女，数学家。
- 计算机程序创始人，建立了循环和子程序概念。

阿塔那索夫-ABC

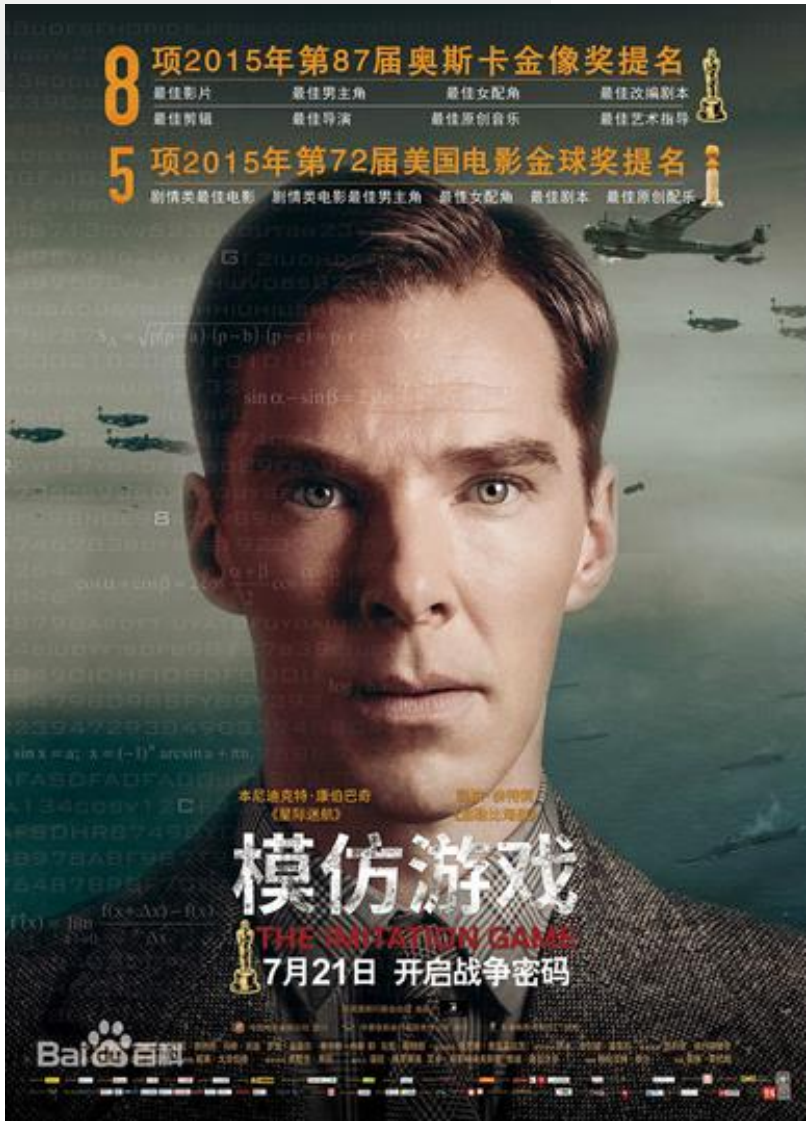
- 阿塔那索夫（J. Atanasoft）是爱阿华大学（University of Iowa）物理学教授

- 第一台电子计算机的试验样机于1939年10月开始运转——这标志着电子计算机的诞生。这台计算机帮助爱阿华大学的教授和研究生们解算了若干复杂的数学方程。阿塔那索夫把这台机器 命名为ABC（Atanasoff- Berry-Computer），其中，A、B分别取俩人姓氏的第一个字母，C即“计算机”的首字母。

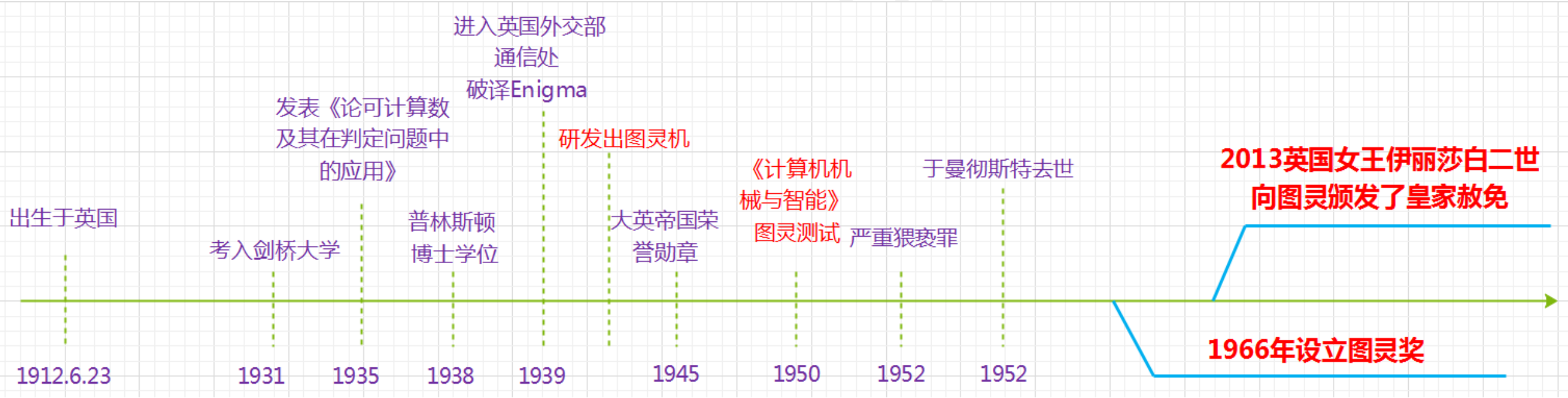


图灵

- 计算机科学之父，人工智能之父
- 第二次世界大战爆发后回到剑桥，曾协助军方破解德国的著名密码系统Enigma，帮助盟军取得了二战的胜利
- 图灵在战时服务的机构于1943年研制成功的CO-LOSSUS(巨人)机，这台机器的设计采用了图灵提出的一些概念。它用了1500个电子管，采用了光电管阅读器；利用穿孔纸带输入；并采用了电子管双稳态线路，执行计数、二进制算术及布尔代数逻辑运算，巨人机共生产了10台，用它们出色地完成了密码破译工作。
- 图灵测试



图灵

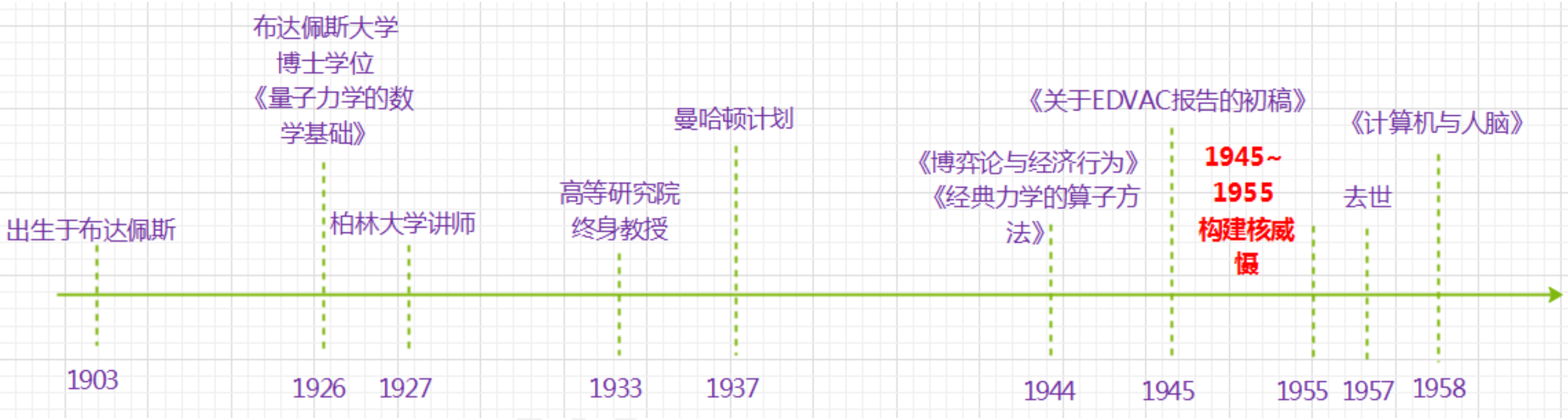


冯诺依曼

- 约翰·冯·诺依曼（John von Neumann，1903年12月28日-1957年2月8日），美籍匈牙利数学家、计算机科学家、物理学家，是20世纪最重要的数学家之一。

- 冯·诺依曼是罗兰大学数学博士，是现代计算机、博弈论、核武器和生化武器等领域内的科学全才之一，被后人称为“现代计算机之父”、“博弈论之父”。

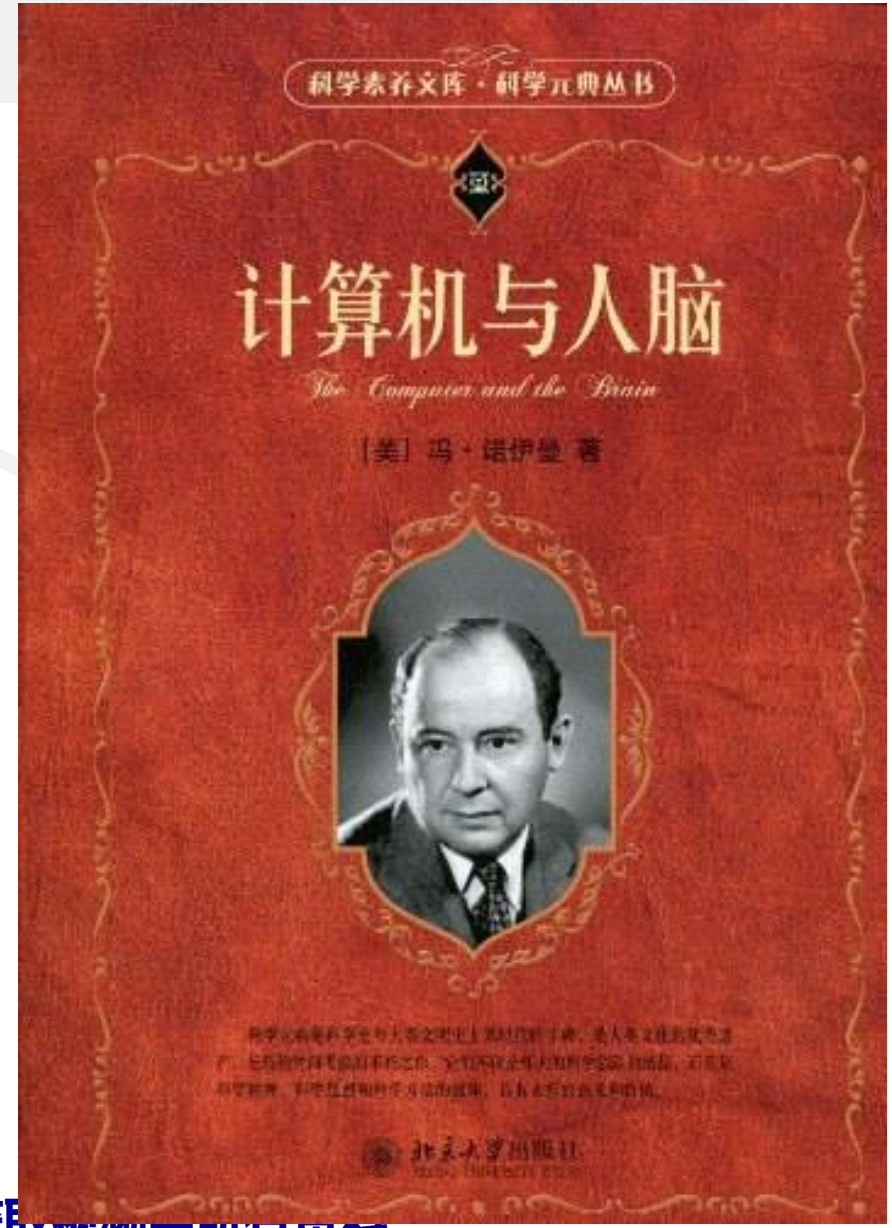
冯诺依曼



《人类武器竞赛史——核武器（下）》

计算机与人脑

- 仿生
- Cpu　大脑
- 总线　神经、血管
- 存储　记忆中枢
- 传感器 温度、气味、咸淡
- 机械臂 四肢



仙童半导体Fairchild

- 晶体管之父肖克利（W.Shockley）博士，离开贝尔实验室返回故乡圣克拉拉，创建“肖克利半导体实验室
- 1957年诺依斯、摩尔共8个人出走创建仙童。

Intel-CPU发展史就是Intel公司的发展历史

- 英特尔公司于1968年由罗伯特·诺伊斯、戈登·摩尔和安迪·格鲁夫创建于美国硅谷。
- 1971年：世界上第一款微处理器4004微处理器
- **1985年：英特386微处理器**
- 1993年：英特尔奔腾（Pentium）处理器
- 2005年：Intel Core处理器
- 2020年1月，发布十代酷睿H系列标压版，i7/i9双双超5GHz。

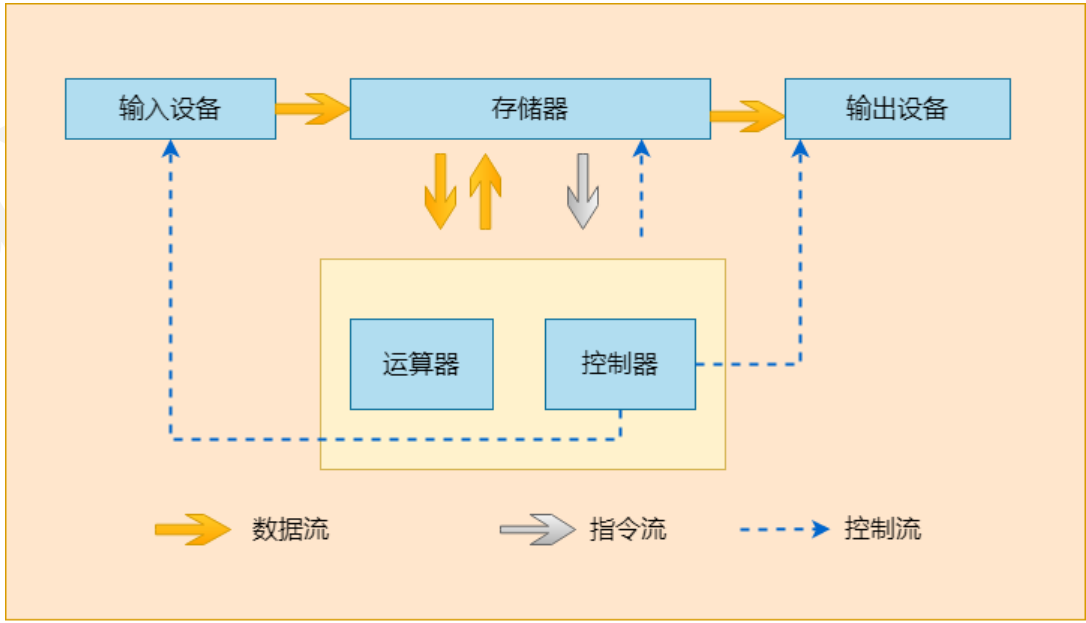
ARM®

05

CPU原理

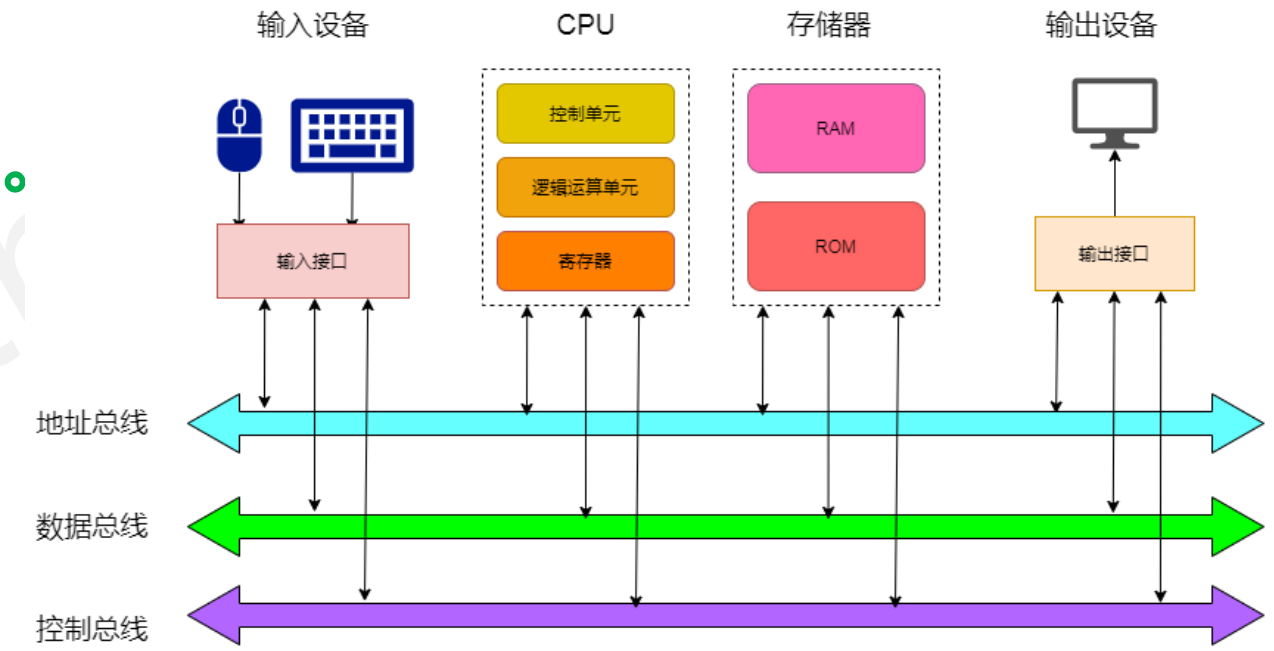
计算机组成

- （1）输入设备
- 输入设备的任务是把人们编好的程序和原始数据送到计算机中去，并且将它们转换成计算机内部所能识别和接受的信息方式。常用的有键盘、鼠标、扫描仪等。
- （2）输出设备
- 输出设备的任务是将计算机的处理结果以人或其他设备所能接受的形式送出计算机。 常用的有显示器、打印机、绘图仪等。
- （3）计算机的总线结构
- 将各大基本部件，按某种方式连接起来就构成了计算机的硬件系统。系统总线包含有三种不同功能的总线，即数据总线DB（Data Bus）、地址总线AB（Address Bus）和控制总线CB（Control Bus）。
- （4）CPU
- CPU中央处理器，类比人脑，作为计算机系统的运算和控制核心，是信息处理、程序运行的最终执行单元。CPU内部主要包括运算器和控制器。



总线

- 数据总线DB（Data Bus）
- 地址总线AB（Address Bus）
- 和控制总线CB（Control Bus）。



存储器

高速缓冲存储器（Cache）：CPU可以直接访问，用来存放当前正在执行的程序中的活跃部分，以便快速地向CPU提供指令和数据。

它分为一级缓存（L1 Cache）、二级缓存（L2 Cache）、三级缓存（L3 Cache）这些数据，它位于内存和 CPU 之间，是一个读写速度比内存更快的存储器。

主存储器：可由CPU直接访问，用来存放当前正在执行的程序和数据。

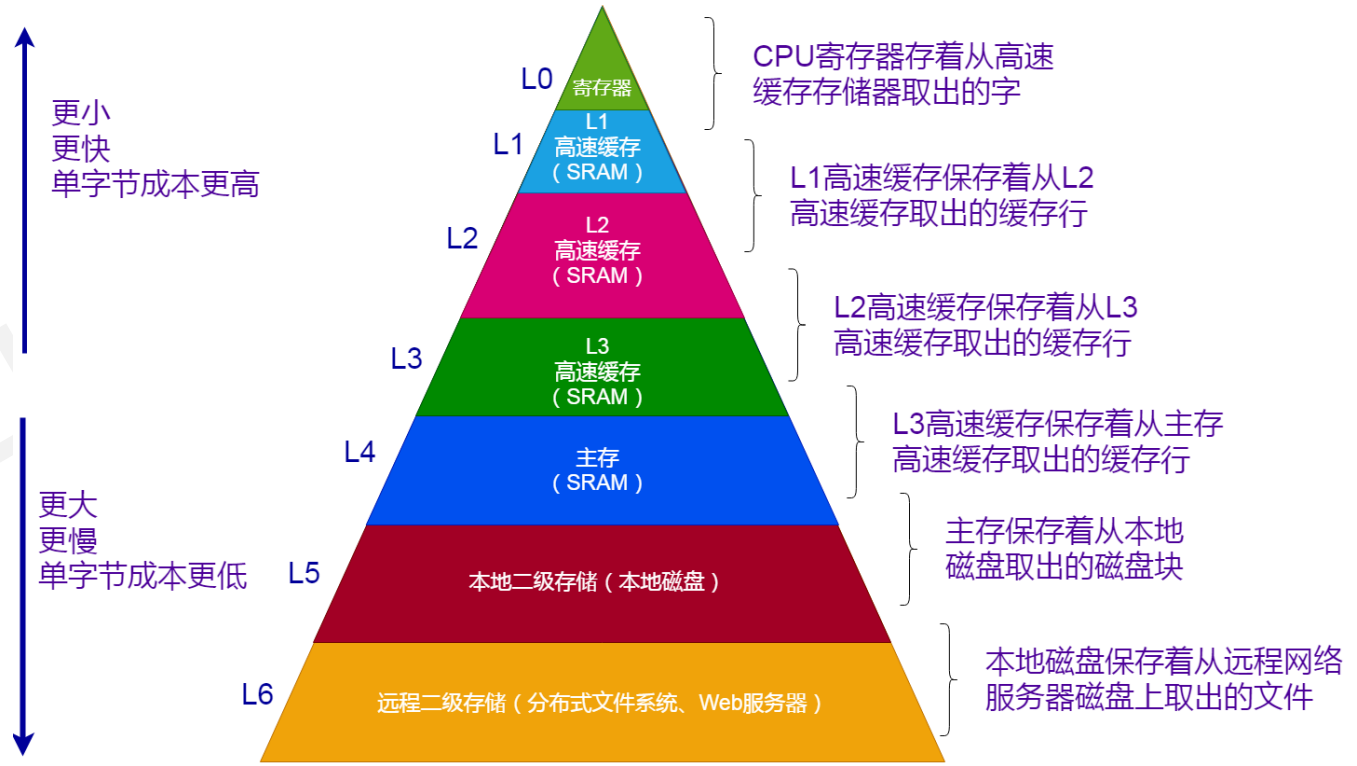
辅助存储器：设置在主机外部，CPU不能直接访问，用来存放暂时不参与运行的程序和数据，需要时再传送到主存。

速度

Cache： 几百到上千**GB/s**

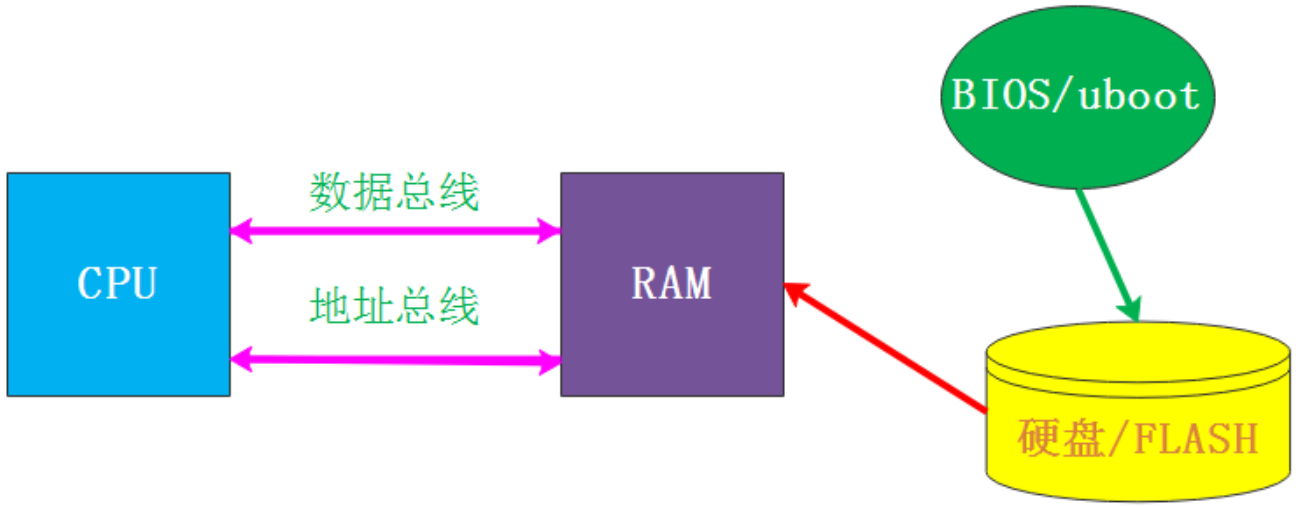
内存： 几十**GB/s**

Disk： 几百**MB/s**



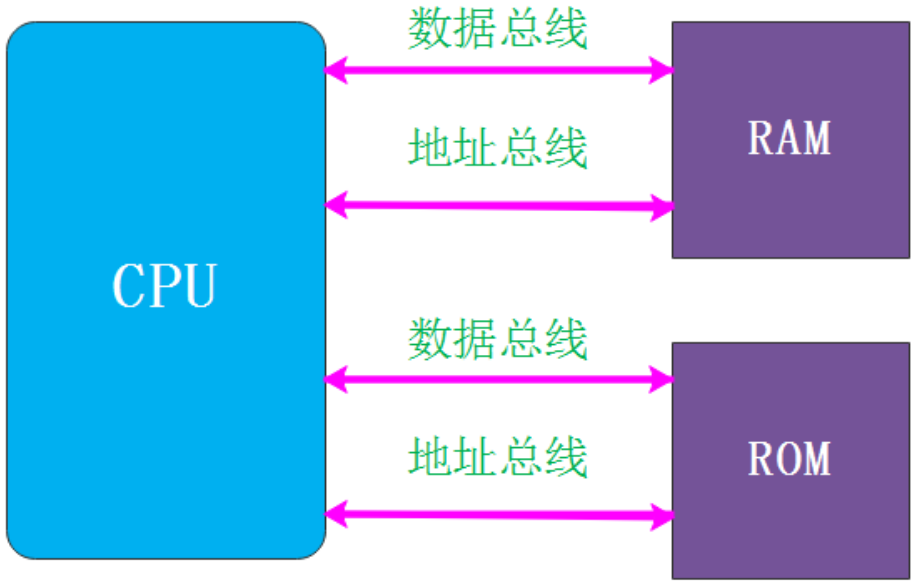
冯诺依曼架构

- 冯诺依曼的核心是：存储程序，顺序执行，并成功将其运用在计算机的设计之中，规定计算机必须具有如下功能：
- （1） 把需要的程序和数据送至计算机中；
- （2） 必须具有长期记忆程序、数据、中间结果和最终运算结果的能力；
- （3） 能够完成各种算术、逻辑运算和数据传送等数据加工处理的能力；
- （4） 能够根据需要控制程序走向，并能根据指令控制机器的各部件协调操作；
- （5） 能够按照要求将处理结果输出给用户。



哈弗结构

- 冯诺依曼结构是程序存储区和数据存储器都是可以放到内存中，统一编码的，而哈弗结构是分开编址的。



哪些处理器是哈佛架构、哪些处理器是冯诺依曼架构?

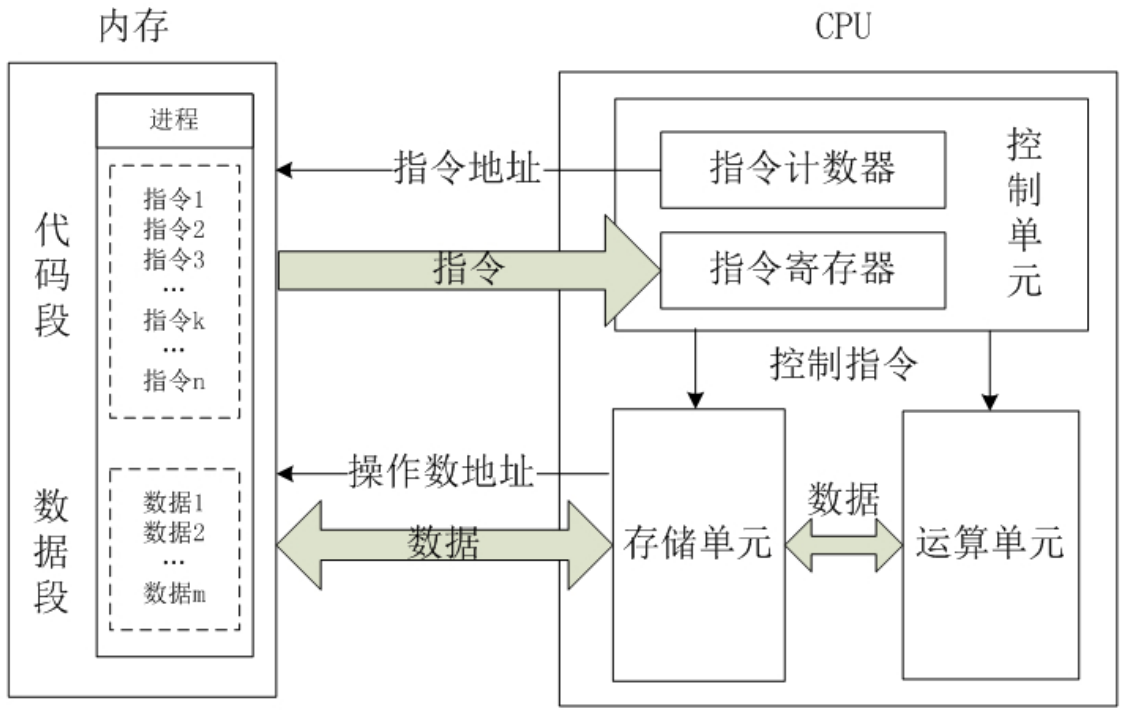
- MCU（单片机）几乎都是用哈佛结构，譬如广泛使用的51单片机、典型的STM32单片机（核心是ARM Cortex-M系列的）都是哈佛结构。
- PC和服务器芯片（譬如Intel AMD），ARM Cortex-A系列嵌入式芯片（譬如三星Exynos 4412，华为的麒麟970等手机芯片）等都是冯诺依曼结构。

混合结构

- 混合结构
- 比如基于Exynos 4412开发板上都配备了1024MB的DDR SDRAM，和8GB的EMMC。
- 正常工作时所有的程序和数据都从EMMC中加载到DDR中，也就是说不管你是指令还是数据，存储都是在EMMC中，运行时都在DDR中，再通过cache和寄存器送给CPU去加工处理。这就是典型的冯诺依曼系统。
- 但是，exynos 4412内部仍然有一定容量的64KB irom和64KB iram，这些irom和iram是用于SoC引导和启动的，芯片上电后首先会执行内部irom中固化的代码，其实执行这些代码时4412就好像一个MCU一样，irom就是他的flash，iram就是他的SRAM，这又是典型的哈佛结构。这其实就是混合式结构设计，而非纯粹设计。

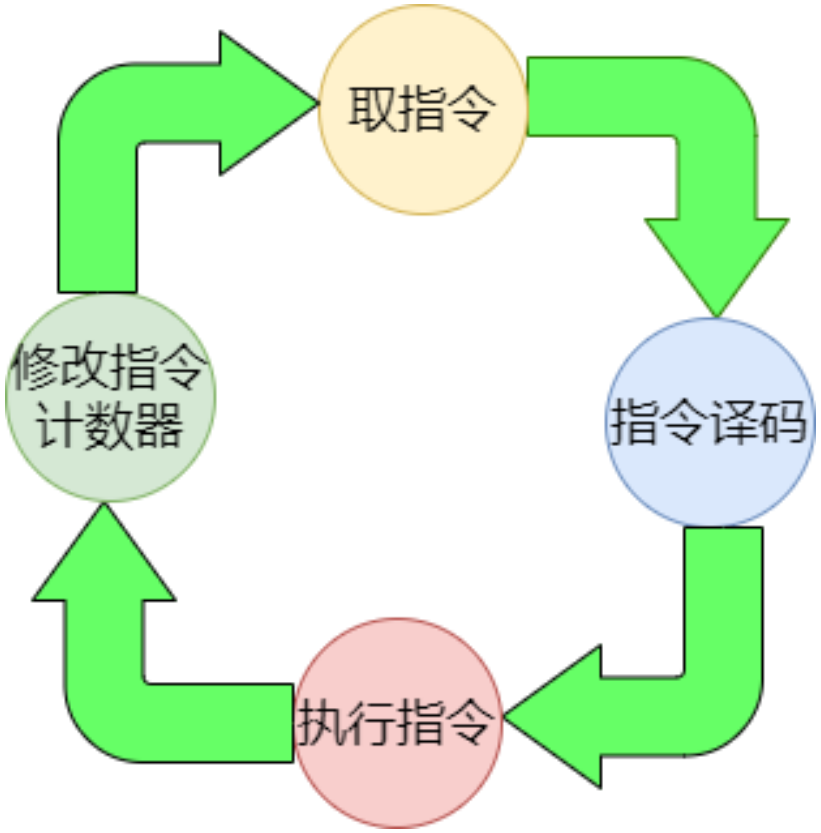
CPU的运行原理

- CPU的控制单元在时序脉冲的作用下，将指令计数器里所指向的指令地址(这个地址是在内存里的)送到地址总线上去，然后CPU将这个地址里的指令读到指令寄存器进行译码。
- 有运算器执行对应的机器指令，并将结果通过地址总线写回数据段



指令执行步骤

- 取指令：
- CPU的控制器从内存读取一条指令并放入指令寄存器。
- 指令译码：
- 指令寄存器中的指令经过译码，决定该指令应进行何种操作(就是指令里的操作码)、操作数在哪里(操作数的地址)。
- 执行指令:
- 分两个阶段“取操作数”和“进行运算”。
- 修改指令计数器:
- 决定下一条指令的地址。



CISC和RISC

- CISC（Complex Instruction Set Computers，复杂指令集计算机）
  - CISC以Intel，AMD的X86 CPU为代表
- RISC（Reduced Instruction Set Computers，精简指令集计算机）
  - RISC以ARM，IBM Power为代表

ARM指令集

- ARM指令是RISC（Reduced Instruction Set Computing），即精简指令运算集

- MIPS
  - 龙芯
- X86
  - intel

ARM指令格式

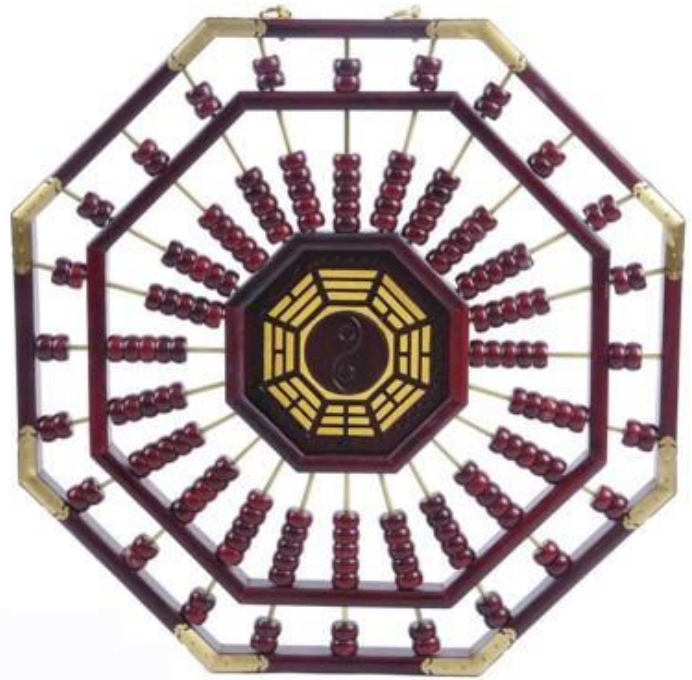
操作码	操作数的地址

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	
Cond	0	0	I	Opcode				S	Rn	Rd	Operand2						数据处理/PSR 传送
Cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm	乘法
Cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rn	1	0	0	1	Rm	长乘法
Cond	0	0	0	1	0	B	0	0	Rn	Rd	0 0 0 0	1	0	0	1	Rm	单数据交换
Cond	0	0	0	1	0	0	1	0	1 1 1 1	1 1 1 1	1 1 1 1	0	0	0	1	Rm	分支和状态切换跳转
Cond	0	0	0	P	U	0	W	L	Rn	Rd	0 0 0 0	1	S	H	1	Rm	半字数据传送：寄存器偏移
Cond	0	0	0	P	U	1	W	L	Rn	Rd	Offset	1	S	H	1	Offset	半字数据传送：立即数偏移
Cond	0	1	0	P	U	B	W	L	Rn	Rd	Offset						单数据传送
Cond	0	1	I											1			未定义（*Undefined*）
Cond	1	0	I	P	U	B	W	L	Rn	寄存器列表							块数据传送
Cond	1	0	1	L	Offset												分支跳转
Cond	1	1	0	P	U	B	W	L	Rn	CRd	CP#	Offset					协处理器数据传送
Cond	1	1	1	0	CP Opc				CRn	CRd	CP#	CP		0	CRm		协处理器数据操作
Cond	1	1	1	0	CP Opc			L	CRn	Rd	CP#	CP		1	CRm		协处理器寄存器传送
Cond	1	1	1	1	处理器忽略												软件中断

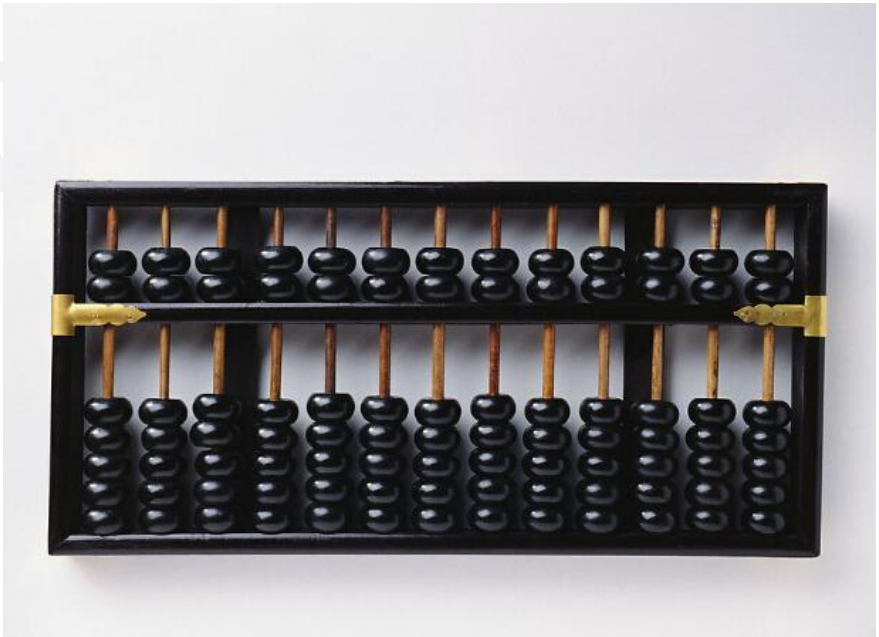
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

算盘

文王桃木算盘



算盘-16进制



ARM

06

学习ARM指令开发

环境-KEIL MDK uVision

参考文章

- 《1. 从0开始学ARM-安装Keil MDK uVision集成开发环境》

- 工具下载地址：https://download.csdn.net/download/daocaokafei/13029121

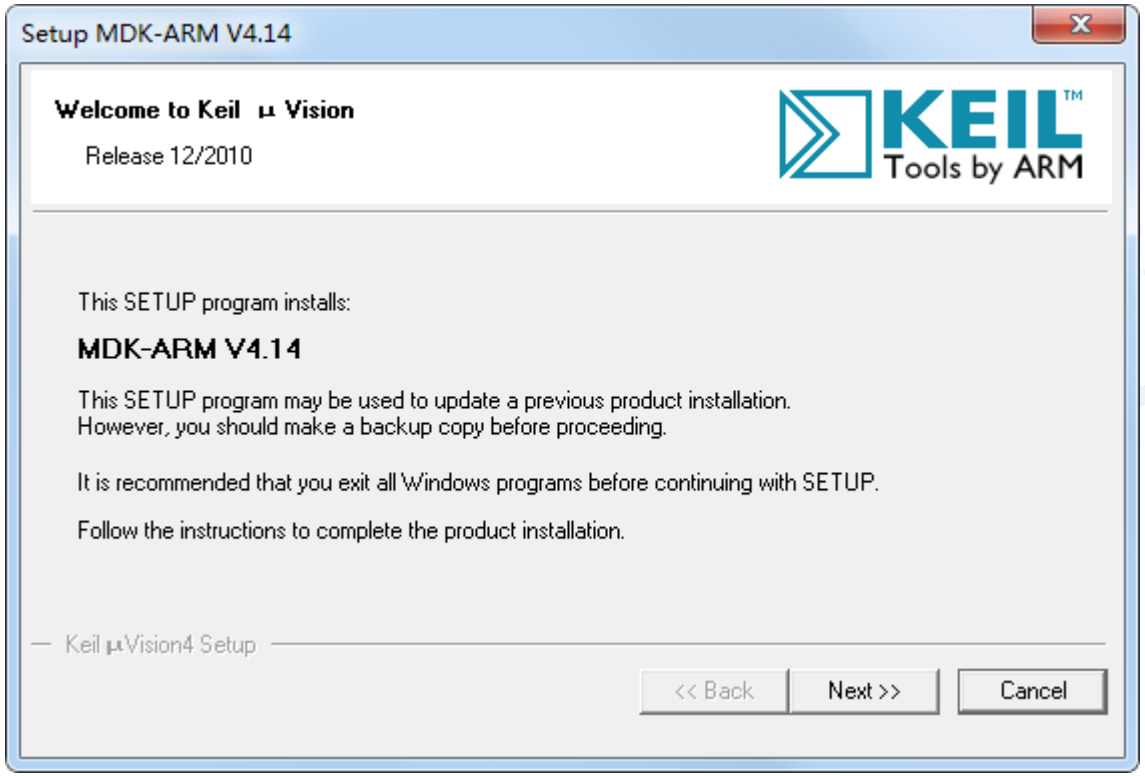
KEIL、MDK、uVision、ARM之间的关系

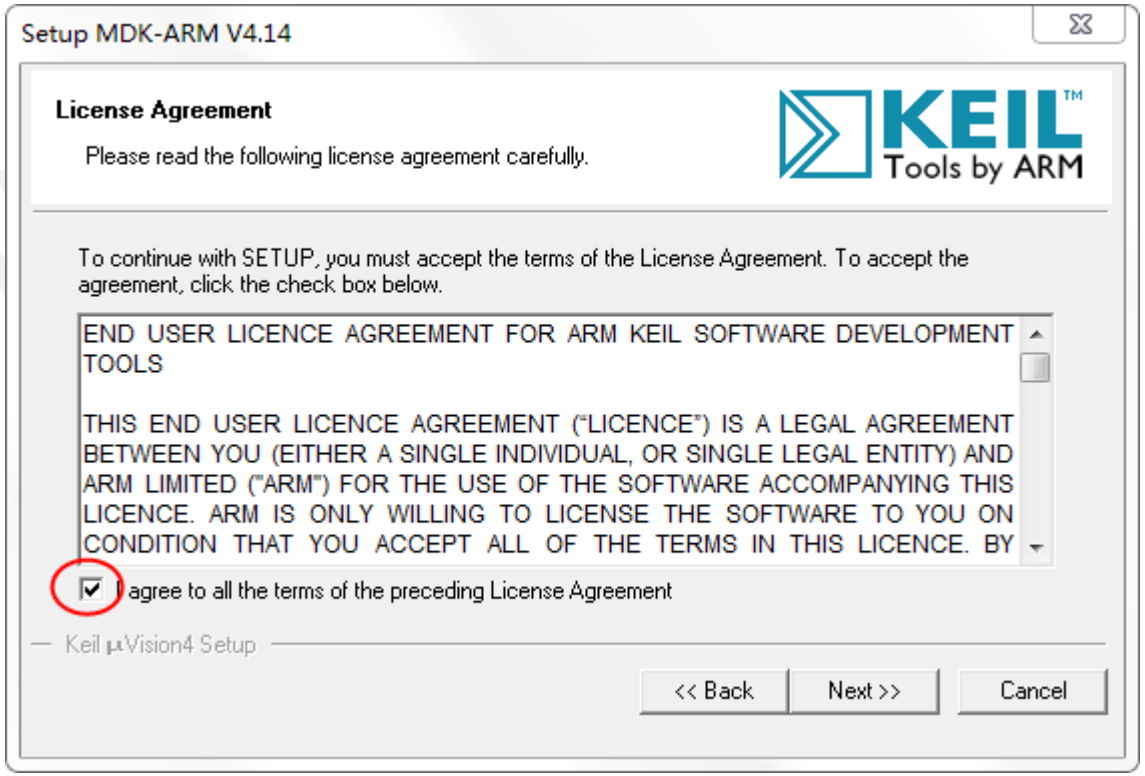
- 1） KEIL
- 1) 既是公司名称， 同时也是KEIL公司所有的开发工具。
- .2) 2005年被ARM收购。
- 2） uVision
- 1) KEIL公司开发的集成开发环境（IDE）。
- 2) 共有4个版本：uVision2、uVision3、uVision4、uVision5。
- 3） MDK
- 1) 英文全称：Microcontroller Development Kit。
- 2) MDK-ARM = KEIL MDK = RealView MDK = KEIL For ARM，统一用 MDK-ARM 称呼。

MDK-ARM

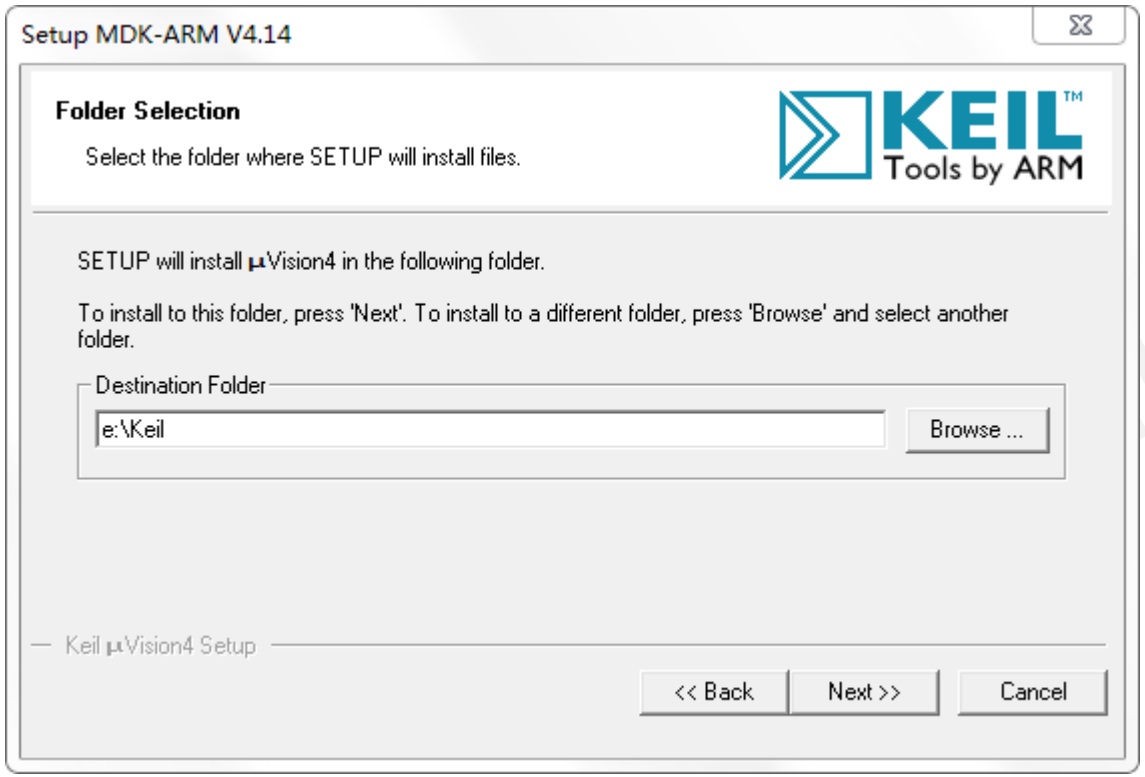
- MDK-ARM 为基于Cortex-M、Cortex-R4、ARM7、ARM9处理器设备提供了一个完整的开发环境。 MDK-ARM专为微控制器应用而设计，不仅易学易用，而且功能强大，能够满足大多数苛刻的嵌入式应用。

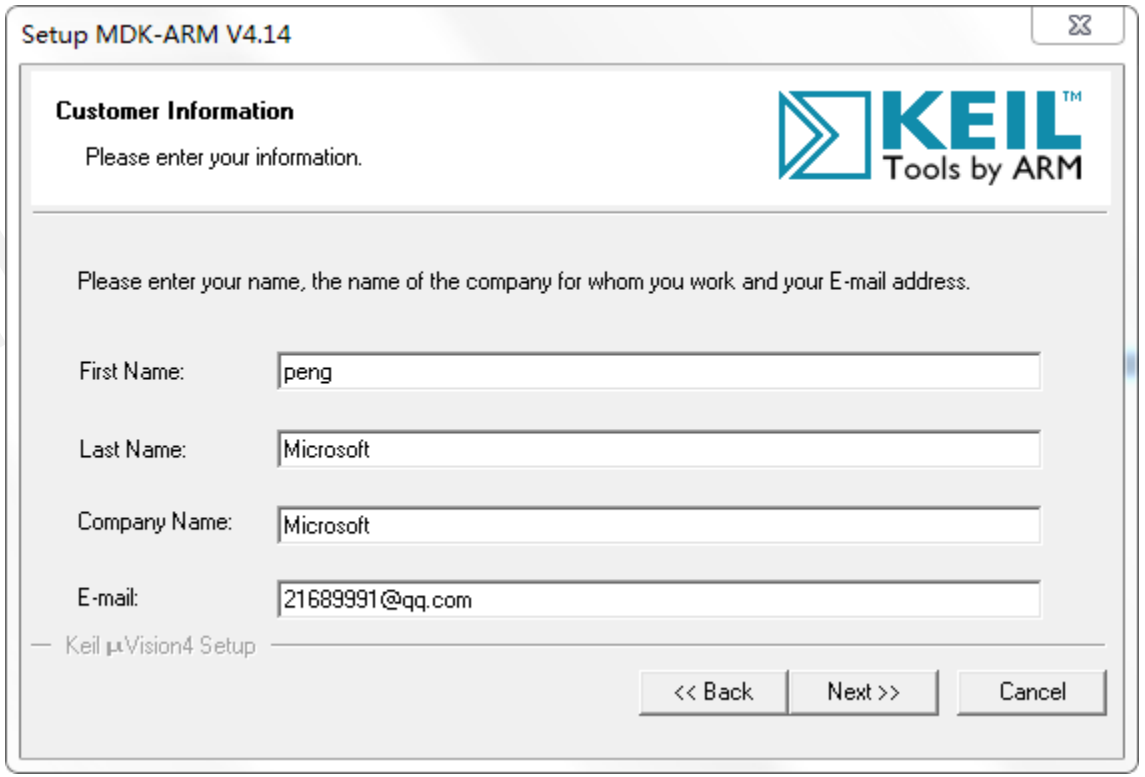
- MDK-ARM有四个可用版本，分别是MDK-Lite、MDK-Basic、MDK-Standard、MDK-Professional。所有版本均提供一个完善的C / C++开发环境，其中MDK-Professional还包含大量的中间库。

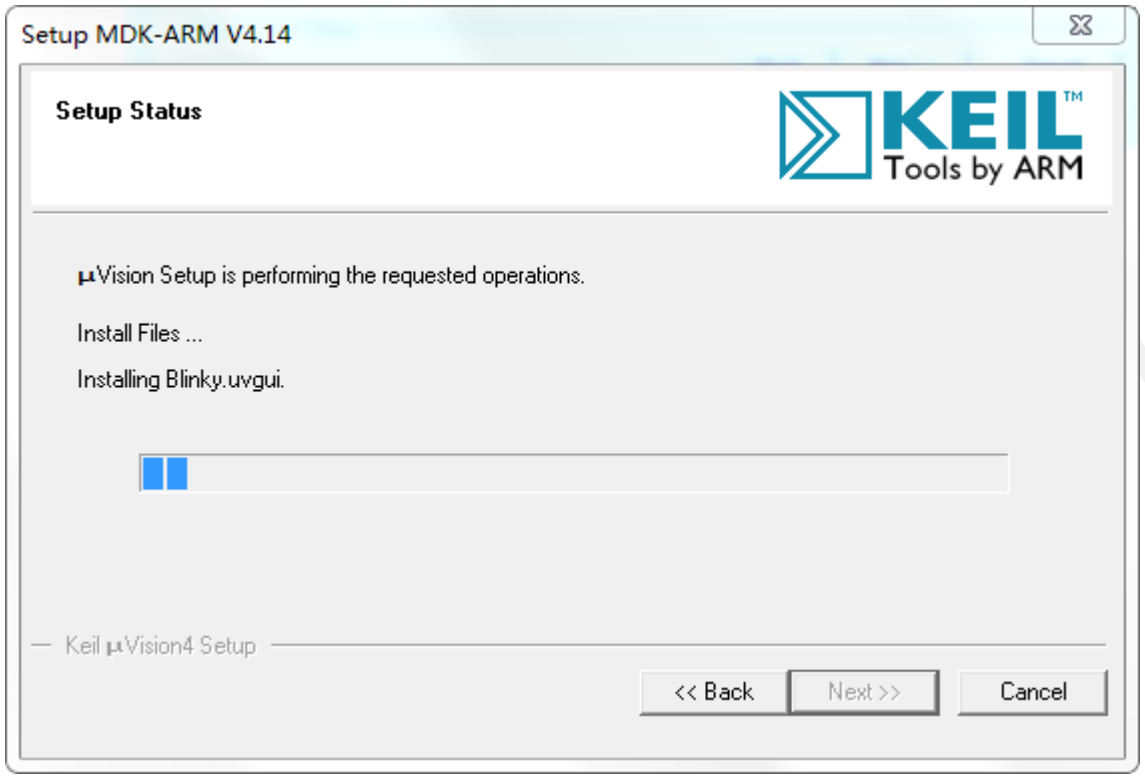


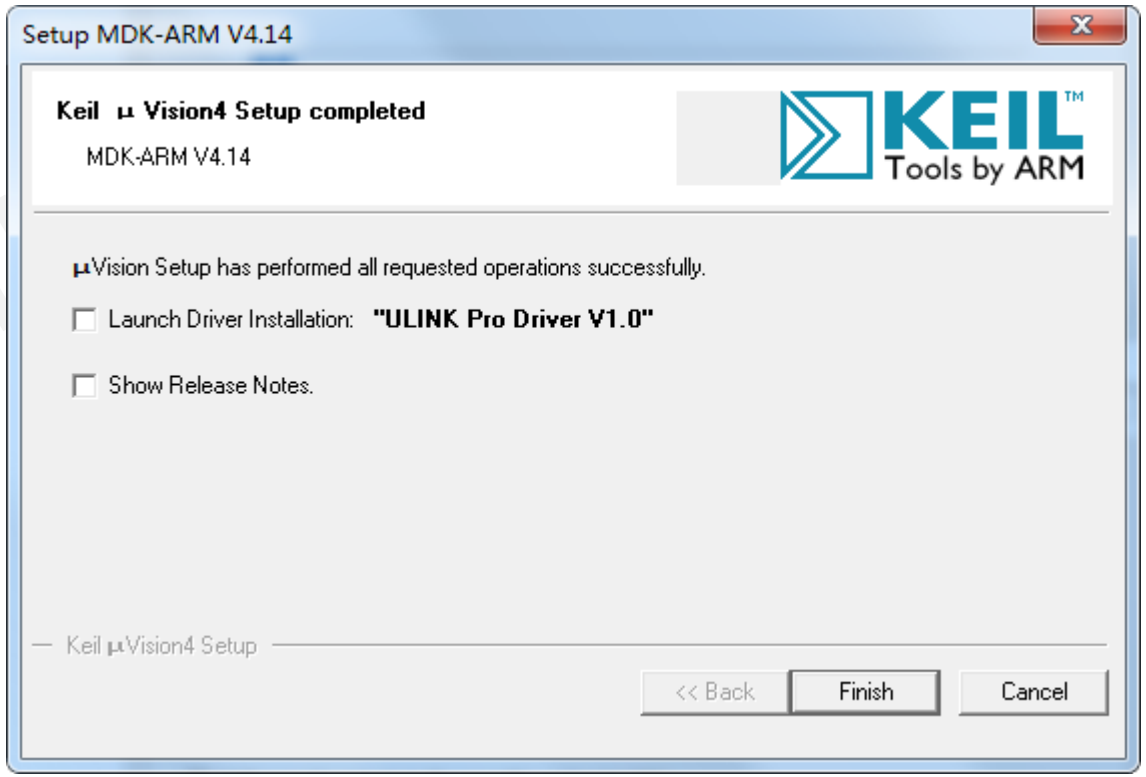


选择安装目录，尽量不要有中文目录：

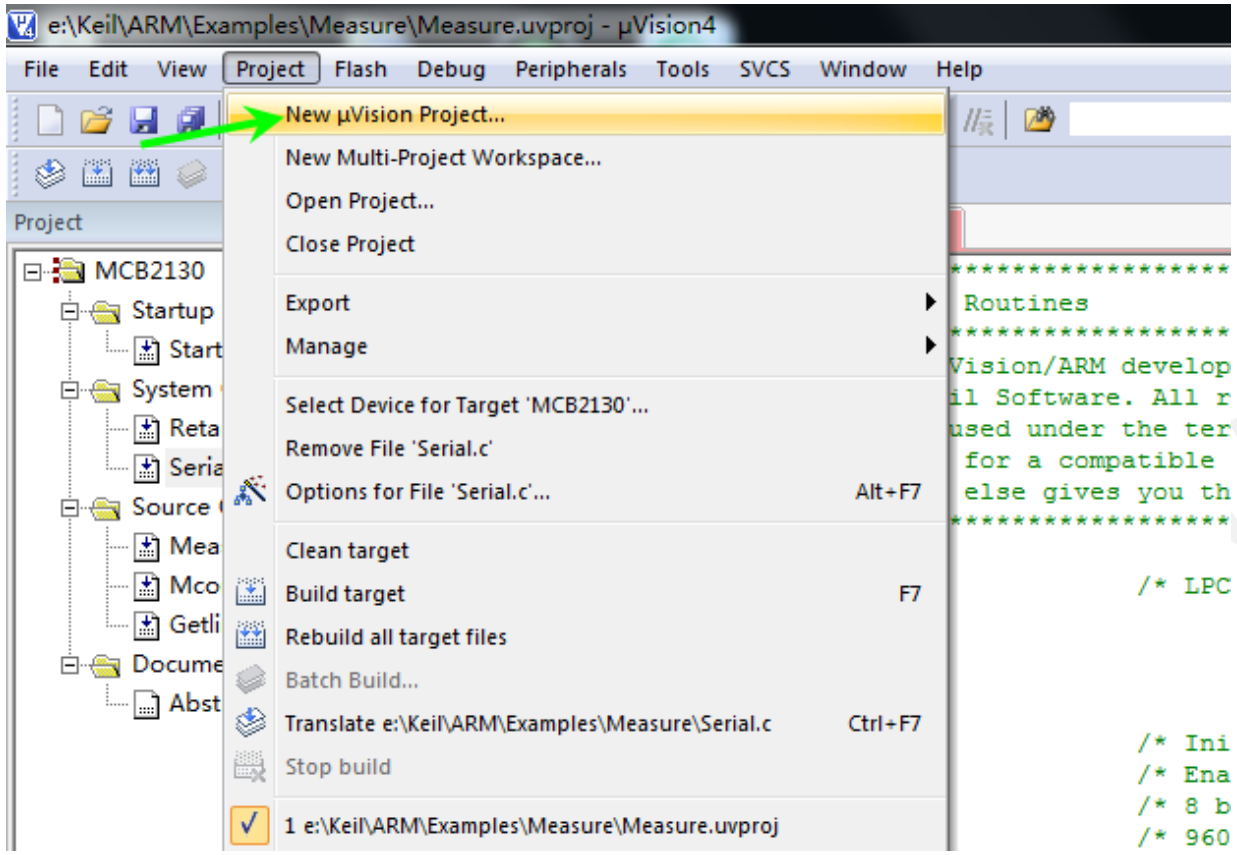


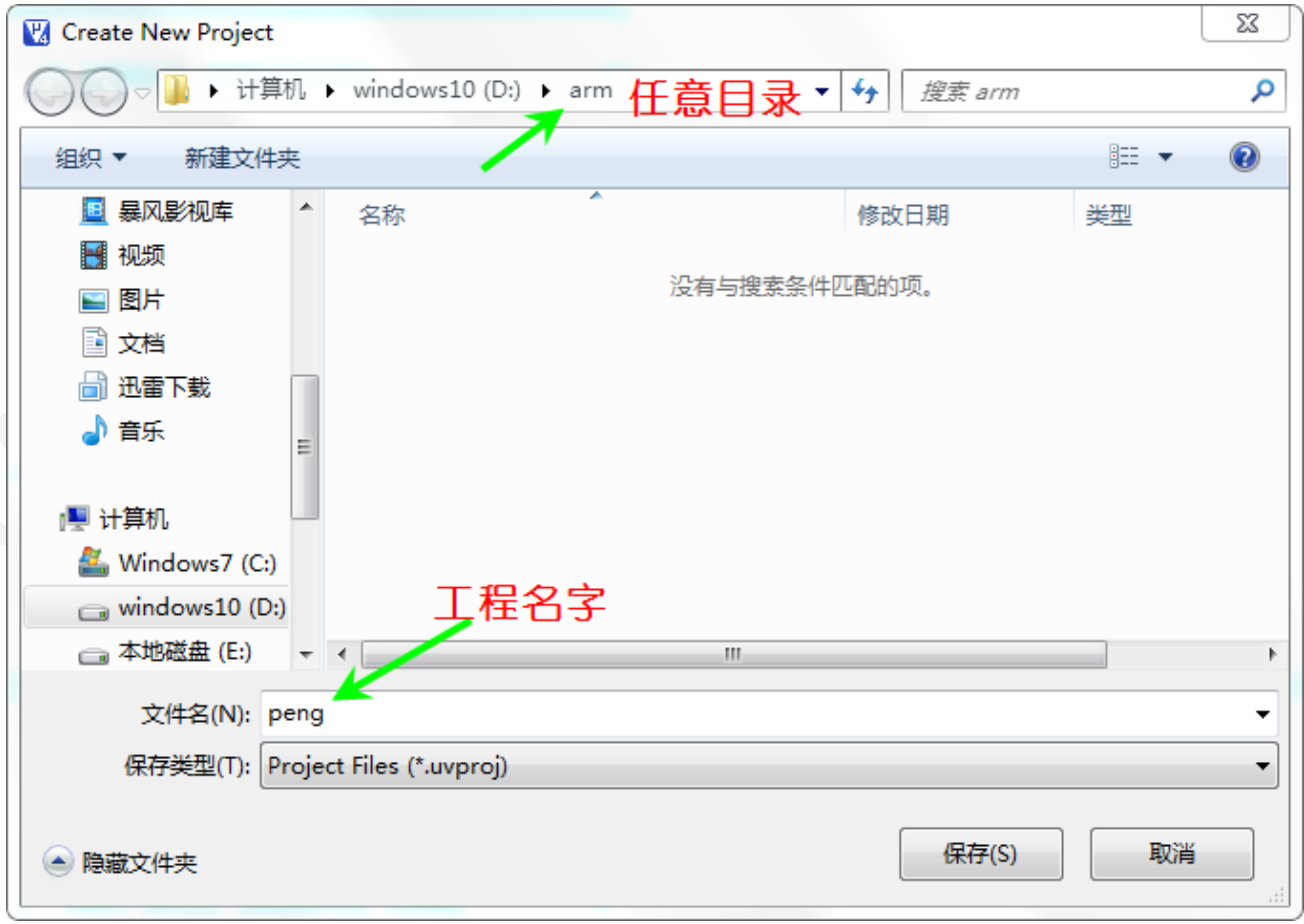






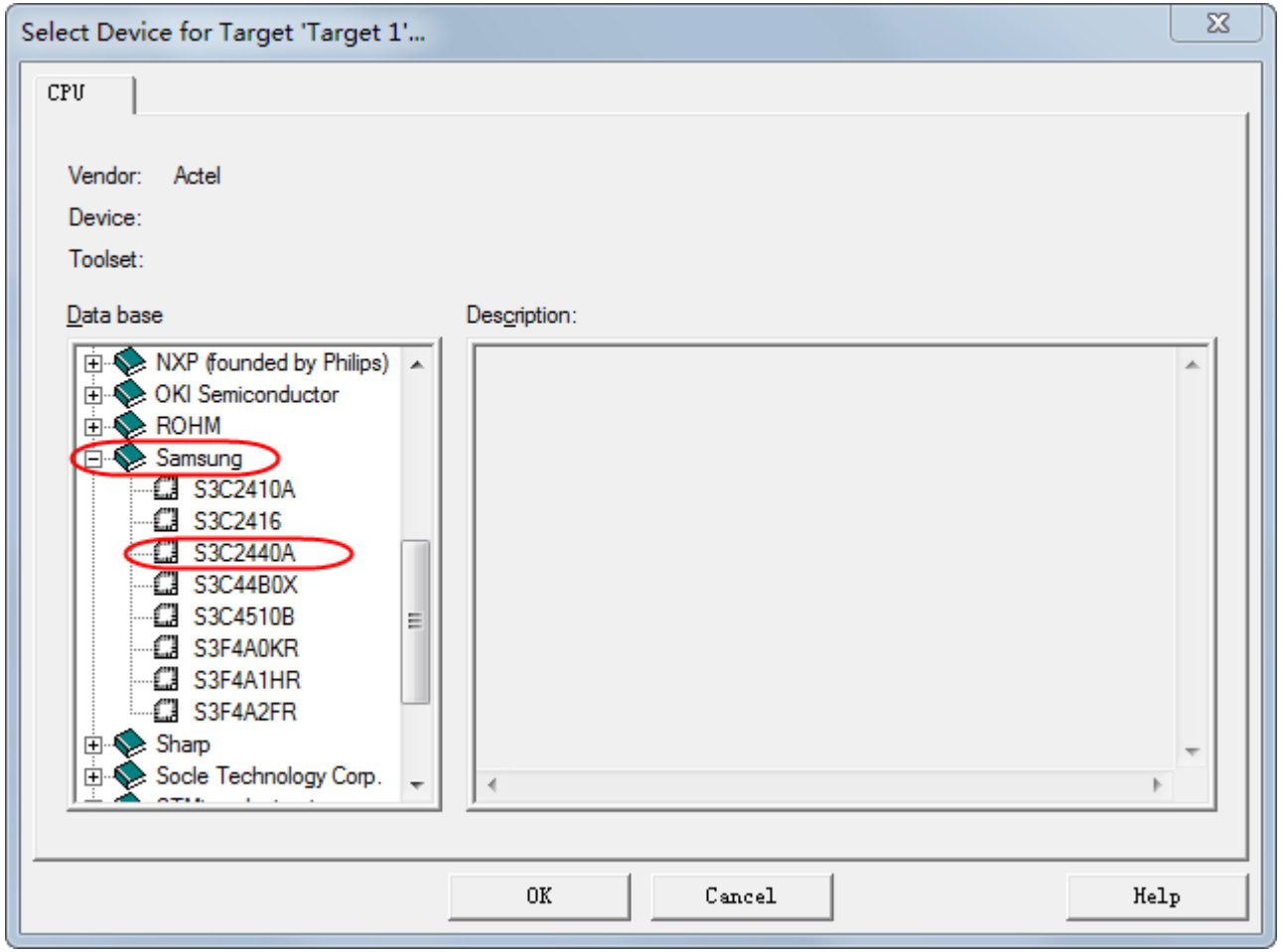
创建第一个工程

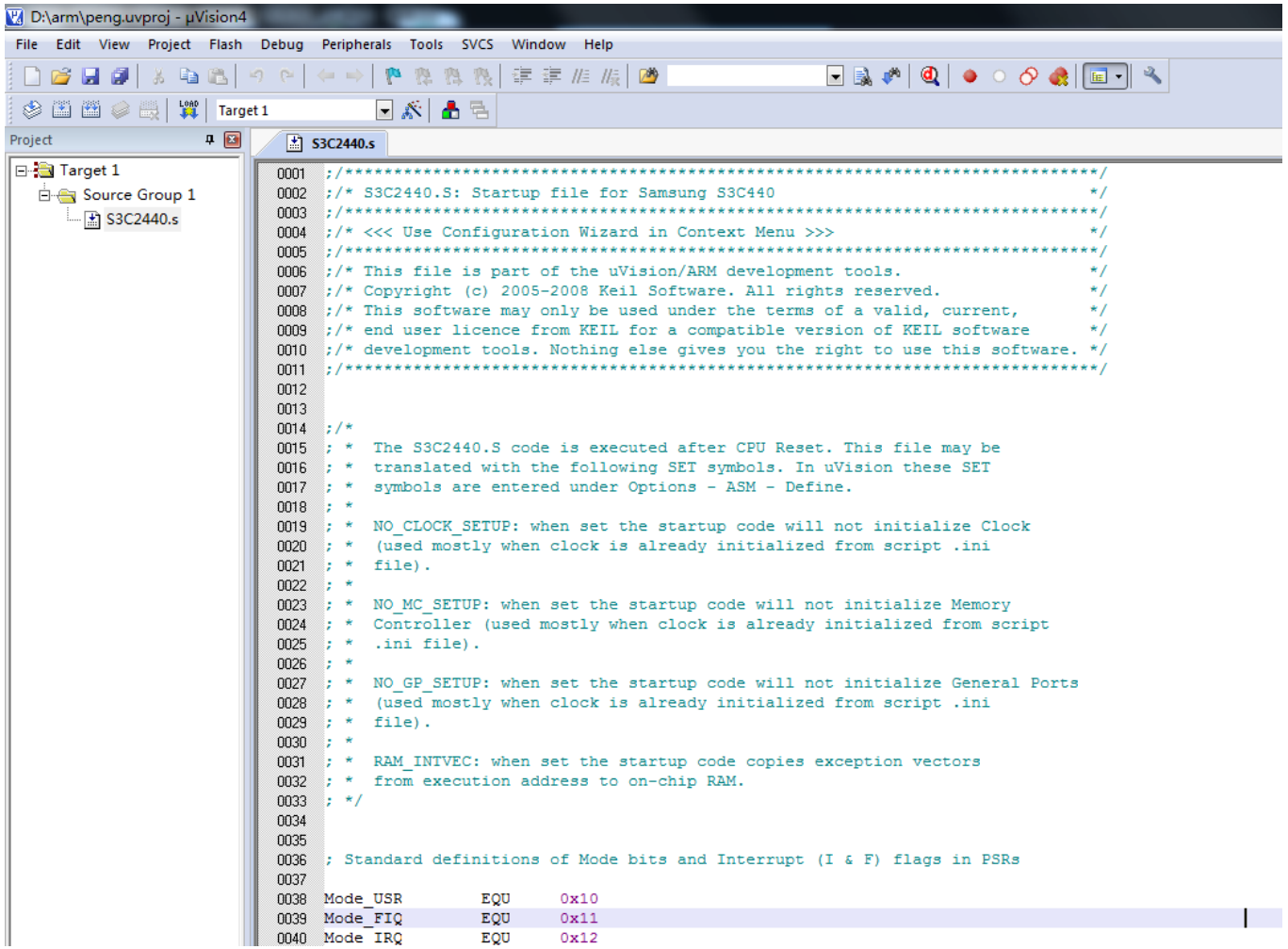




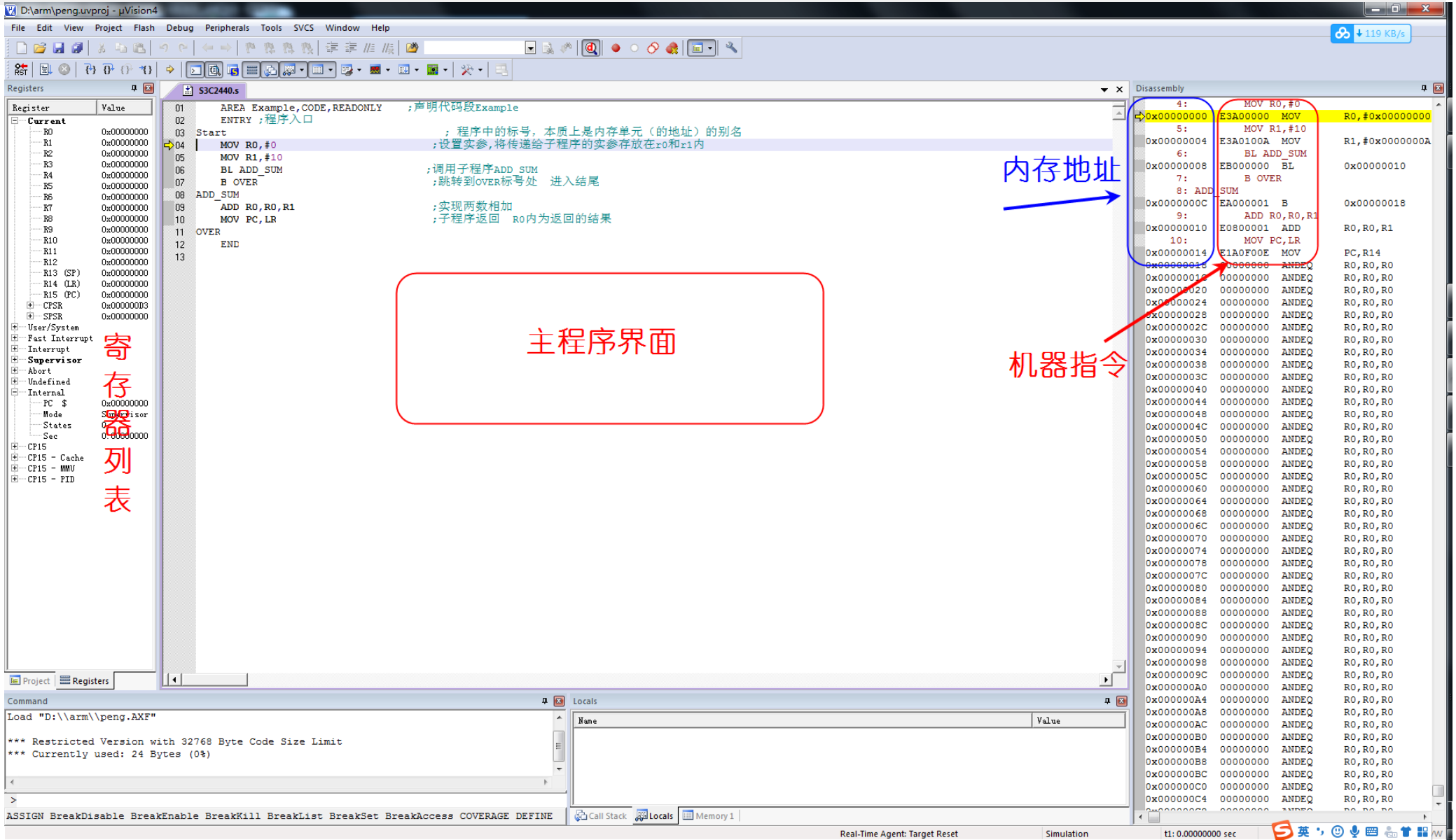
关注公众号：一口Linux 回复：arm 获取视频中所有资料

选择CPU->Samsung->S3C2440A





界面介绍


D:\arm\peng.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Registers
Register Value
Current
R0 0x00000000
R1 0x00000000
R2 0x00000000
R3 0x00000000
R4 0x00000000
R5 0x00000000
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x00000000
R14 (LR) 0x00000000
R15 (PC) 0x00000000
CPSR 0x000000D3
SPSR 0x00000000
User/System
Fast Interrupt
Interrupt
Supervisor
Abort
Undefined
Internal
PC $ 0x00000000
Mode Supervisor
States 0
Sec 0.00000000
CP15
CP15 - Cache
CP15 - MMU
CP15 - PID
寄存器列表
Project
Registers
S3C2440.s
01 AREA Example,CODE,READONLY ;声明代码段Example
02 ENTRY ;程序入口
03 Start ; 程序中的标号，本质上是内存单元（的地址）的别名
04 MOV R0,#0 ;设置实参,将传递给子程序的实参存放在r0和r1内
05 MOV R1,#10
06 BL ADD_SUM ;调用子程序ADD_SUM
07 B OVER ;跳转到OVER标号处 进入结尾
08 ADD_SUM
09 ADD R0,R0,R1 ;实现两数相加
10 MOV PC,LR ;子程序返回 R0内为返回的结果
11 OVER
12 END
13
主程序界面
Disassembly
4: MOV R0,#0
0x00000000 E3A00000 MOV R0,#0x00000000
5: MOV R1,#10
0x00000004 E3A0100A MOV R1,#0x0000000A
6: BL ADD_SUM
0x00000008 EB000000 BL 0x00000010
7: B OVER
8: ADD_SUM
0x0000000C EA000001 B 0x00000018
9: ADD R0,R0,R1
0x00000010 E0800001 ADD R0,R0,R1
10: MOV PC,LR
0x00000014 E1A0F00E MOV PC,R14
内存地址
机器指令
Command
Load "D:\\arm\\peng.AXF"
*** Restricted Version with 32768 Byte Code Size Limit
*** Currently used: 24 Bytes (0%)
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE DEFINE
Locals
Name Value
Call Stack Locals Memory 1
Real-Time Agent: Target Reset
Simulation
t1: 0.00000000 sec

实例代码-注意缩进

```
        AREA Example,CODE,READONLY    ;声明代码段Example
        ENTRY ;程序入口
Start                ; 程序中的标号，本质上是内存单元（的地址）的别名
        MOV R0,#0     ;设置实参,将传递给子程序的实参存放在r0和r1内
        MOV R1,#10
        BL ADD_SUM    ;调用子程序ADD_SUM
        B OVER        ;跳转到OVER标号处 进入结尾
ADD_SUM
        ADD R0,R0,R1  ;实现两数相加
        MOV PC,LR     ;子程序返回 R0内为返回的结果
OVER
        END
```

ARM

07

ARM模式及其切换、异常

ARM技术特征

- ARM处理器有如下特点
- 体积小、低功耗、低成本、高性能。
- 支持Thumb(16位）/ARM(32位）双指令集，能很好地兼容8位/16位器件。
- 大量使用寄存器，指令执行速度更快。
- 大多数数据操作都在寄存器中完成。
- 寻址方式灵活简单，执行效率高。
- 指令长度固定。

ARM的基本数据类型

- ARM采用的是32位架构，ARM的基本数据类型有以下3种。
- Byte:
  - 字节，8bit。
- Halfword:
  - 半字，16bit(半字必须与2字节边界对齐）。
- Word:
  - 字，32bit(字必须与4字节边界对齐）。
- 存储器可以看做是序号为0~2^32-1的线性字节阵列。每一个字节都有唯一的地址。

ARM处理器工作模式

- Cortex-A系列的ARM处理器工作模式有8种

模式分类	处理器工作模式	异常模式	说明
非特权模式	用户（user）		用户程序运行模式
特权模式 该模式下可以访问系统资源	系统（system）		运行特权级的操作系统任务
	一般中断（IRQ）	异常模式 通常由系统异常状态切换进该组模式	普通中断模式
	快速中断（FIR）		快速中断模式
	管理（supervisor）		提供操作系统使用的一种保护模式，swi命令状态
	中止（abort）		虚拟内存管理和内存数据访问保护
	未定义指令终止（undefined）		支持通过软件仿真硬件的协处理
	monitor		用于执行安全监控代码的模式

1. 用户模式：

- 用户模式是用户程序的工作模式，
- 它运行在操作系统的用户态，
- 它没有权限去操作其它硬件资源，只能执行处理自己的数据，
- 也不能切换到其它模式下，要想访问硬件资源或切换到其它模式只能通过软中断（SWI）或产生异常。

2.系统模式

- 系统模式是特权模式，不受用户模式的限制。
- 用户模式和系统模式共用一套寄存器，
- 操作系统在该模式下可以方便的访问用户模式的寄存器，而且操作系统的一些特权任务可以使用这个模式访问一些受控的资源。

3. 一般中断模式

- 一般中断模式也叫普通中断模式，用于处理一般的中断请求
- 通常在硬件产生中断信号之后自动进入该模式
- 该模式为特权模式，可以自由访问系统硬件资源

4.快速中断模式

- 快速中断模式是相对一般中断模式而言的，它是用来处理对时间要求比较紧急的中断请求
- 主要用于高速数据传输及通道处理中。

5.管理模式(SVC)

- 管理模式是**CPU上电后默认模式**
- 因此在该模式下主要用来做系统的初始化
- 软中断处理也在该模式下，当用户模式下的用户程序请求使用硬件资源时通过软件中断进入该模式。

6.中止模式

- 中止模式用于支持虚拟内存或存储器保护，
- 当用户程序访问非法地址，没有权限读取的内存地址时，会进入该模式，

- linux下编程时经常出现的segment fault通常都是在该模式下抛出返回的。

7.未定义模式

- 未定义模式用于支持硬件协处理器的软件仿真，
- CPU在指令的译码阶段不能识别该指令操作时，会进入未定义模式。

8. Monitor

- 是为了安全而扩展出的用于执行安全监控代码的模式；也是一种特权模式

模式切换

- ARM微处理器的运行模式可以通过软件改变，也可以通过外部中断或异常处理改变。

- 应用程序运行在用户模式下，当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。

异常（Exception）

- 指由处理器执行指令导致原来运行程序的中止，
- 异常与指令运行相关，是CPU执行程序产生的，是同步的，可分为精确异常和非精确异常。
- 异常处理遵守严格的程序顺序，不能嵌套，只有当第一个异常处理完并返回后才能处理后续的异常。

异常源

异常源	描述
Reset	上电时执行
Undef	当流水线中的某个非法指令到达执行状态时执行
SWI	当一个软中断指令被执行完的时候执行
Prefetch	当一个指令被从内存中预取时，由于某种原因而失败，如果它能到达执行状态这个异常才会产生
Data	如果一个预取指令试图存取一个非法的内存单元，这时异常产生
IRQ	通常的中断
FIQ	快速中断

异常源与模式关系

- 快速中断请求异常进入快速中断模式，支持高速数传输及通道处理（FIQ异常响应时进入此模式）；
- 中断请求异常进入中断模式，用于通用中断处理
- 预取指中止，数据中止异常进入中止模式，用于支持虚拟内存和/或存储器保护；
- 未定义指令异常进入未定义模式，
- 支持硬件协处理器的软件仿真软件中断(swi)，复位异常(reset)进入管理模式，操作系统保护代码

ARM

08

ARM寄存器

ARM寄存器

- Cortex A系列ARM处理器共有40个32位寄存器,其中33个为通用寄存器,7个为状态寄存器。
- usr模式和sys模式共用同一组寄存器。

ARM state general registers and program counter

System and User	FIQ	Supervisor	Abort	IRQ	Undefined	Secure monitor
r0	r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12	r12
r13	r13_fiq	r13_svc	r13_abt	r13_irq	r13_und	r13_mon
r14	r14_fiq	r14_svc	r14_abt	r14_irq	r14_und	r14_mon
r15	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

ARM state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und	SPSR_mon

= banked register

通用寄存器包括R0~R15,可以分为3类:

- 1. 未分组寄存器R0~R7
- 2. 分组寄存器R8~R14、R13(SP) 、 R14(LR)
- 3. 程序计数器PC(R15)、

R8_fiq-R12_fir
为快中断独有

ARM state general registers and program counter

System and User	FIQ	Supervisor	Abort	IRQ	Undefined	Secure monitor
r0	r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12	r12
r13	r13_fiq	r13_svc	r13_abt	r13_irq	r13_und	r13_mon
r14	r14_fiq	r14_svc	r14_abt	r14_irq	r14_und	r14_mon
r15	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

ARM state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und	SPSR_mon

= banked register

1. 未分组寄存器R0~R7

- 在所有运行模式下,未分组寄存器都指向同一个物理寄存器,它们未被系统用作特殊的用途
- 因此在中断或异常处理进行运行模式转换时由于不同的处理器运行模式均使用相同的物理寄存器,
- 所以可能造成寄存器中数据的破坏

2. 分组寄存器R8~R14

- 对于分组寄存器,它们每一次所访问的物理寄存器都与当前处理器的运行模式有关。
- R8~R12
- 对于R8~R12来说,每个寄存器对应2个不同的物理寄存器,当使用FIQ(快速中断模式)时,访问寄存器 R8_fiqR12_fiq;当使用除FIQ模式以外的其他模式时,访问寄存器R8_usr~R12_usr。
- R13,R14
- 对于R13,R14来说,每个寄存器对应7个不同的物理寄存器,其中一个是用户模式与系统模式共用,
- 另外6个物理寄存器对应其他6种不同的运行模式,并采用以下记号来区分不同的物理寄存器:
- R13_mode　R14_mode
- (其中mode可为:usr,fiq,irq,svc,abt,und，mon)

3. 寄存器R13（sp）

- 在ARM指令中常用作**堆栈指针**,用户也可使用其他的寄存器作为堆栈指针,而在Thumb指令集中,某些指令强制性的要求使用R13作为堆栈指针。

- 由于处理器的每种运行模式均有自己独立的物理寄存器R13，在用户应用程序的初始化部分，一般都要初始化每种模式下的R13，使其指向该运行模式的栈空间。

- 这样，当程序的运行进入异常模式时，可以将需要保护的寄存器放入R13所指向的堆栈，而当程序从异常模式返回时，则从对应的堆栈中恢复，采用这种方式可以保证异常发生后程序的正常执行。

4. R14（LR）链接寄存器(Link Register)

- 当执行子程序调用指令(BL)时,R14可得到R15(程序计数器PC)的备份
- 在每一种运行模式下，都可用R14保存子程序的返回地址

- 1.当用BL或BLX指令调用子程序时，将PC的当前值复制给R14，
- 2. 执行完子程序后，又将R14的值复制回PC，即可完成子程序的调用返回。
- 以上的描述可用指令完成。
- **MOV PC, LR 或者 BX LR**
- **STMFD SP！,{,LR}　　LDMFD SP！,{,PC}**

5. R15(PC)程序计数器

- 寄存器R15用作程序计数器(PC)

- 在ARM状态下,位[1:0]为0,位[31:2]用于保存PC,在Thumb状态下,位[0]为0,位[31:1]用于保存PC。
- 比如如果pc的值是0x40008001,那么在寻址的时候其实会查找地址0x40008000，低2位会自动忽略掉。

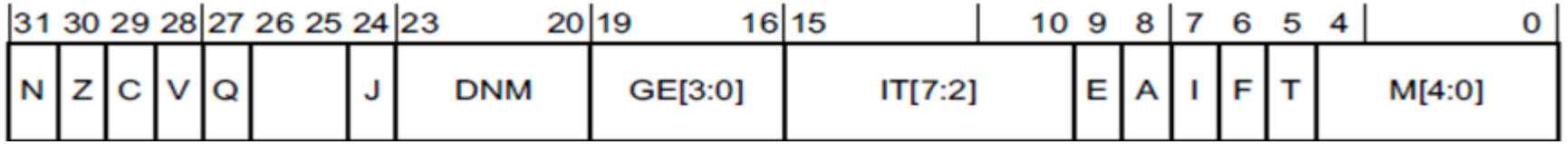
- 由于ARM体系结构采用了多级流水线技术，对于ARM指令集而言，PC总是指向当前指令的下两条指令的地址,即PC的值为当前指令的地址值加8个字节。
- 即：PC值=当前程序执行位置+8

6. CPSR、SPSR

- CPSR(Current Program Status Register，当前程序状态寄存器)，CPSR可在任何运行模式下被访问
- 每一种运行模式下又都有一个专用的物理状态寄存器，称为SPSR(Saved Program Status Register，备份的程序状态寄存器)，
- 当异常发生时，SPSR用于保存CPSR的当前值，从异常退出时则可由SPSR来恢复CPSR

- 由于用户模式和系统模式不属于异常模式，它们没有SPSR，当在这两种模式下访问SPSR，结果是未知的。

CPSR



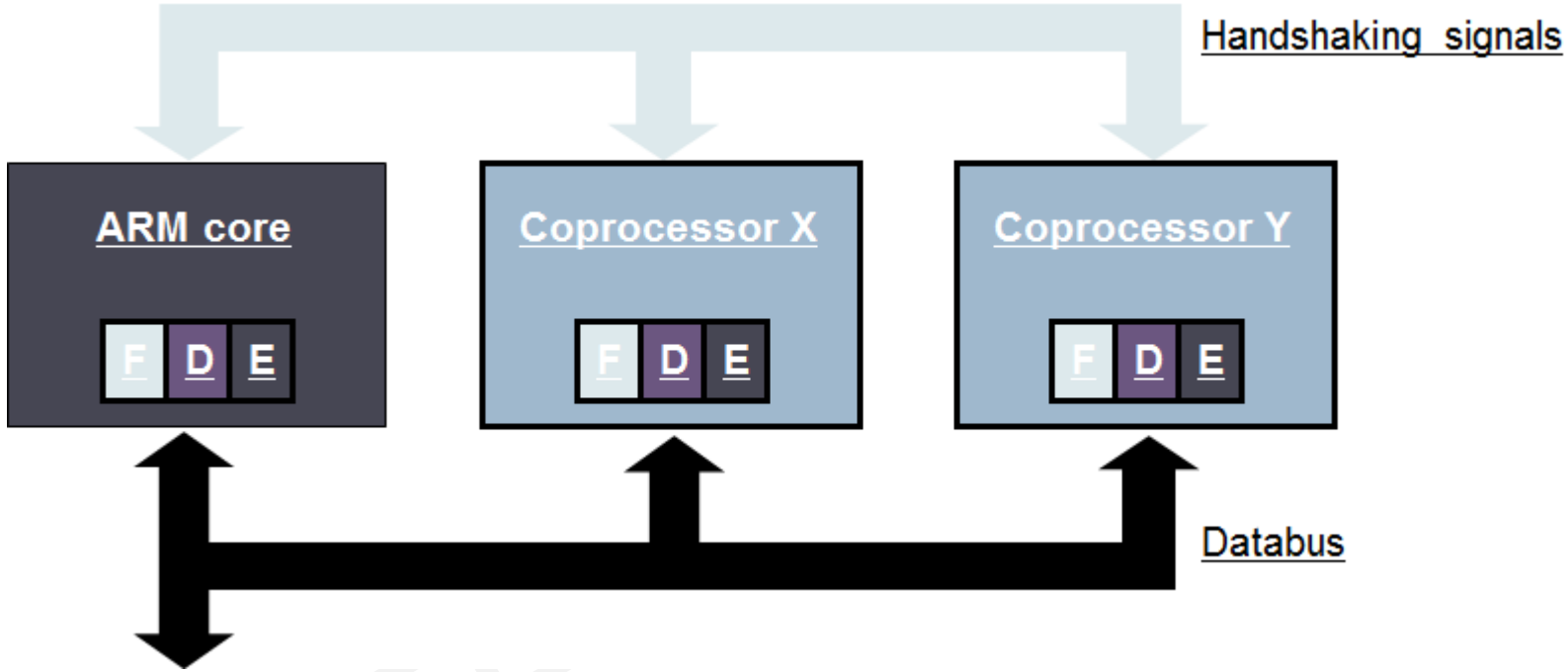
- 条件位：
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out or borrow
  - V = ALU operation o**V**erflowed
- Q 位：
  - 仅ARM v5TE-J架构支持
  - 指示饱和状态
- J 位
  - 仅ARM v5TE-J架构支持
  - T=0;J = 1 处理器处于Jazelle状态
  - 也可以和其他位组合
- DNM位：Do Not Modify
- GE[3:0] 大于或等于(当执行SIMD指令时有效)
- IT[7:2] IF....THEN....指令执行状态位
- E位：大小端控制位
- A位：A=1 禁止不精确的数据异常
- 中断禁止位：
  - I = 1: 禁止 IRQ.
  - F = 1: 禁止 FIQ
- T Bit
  - **T = 0;J=0; 处理器处于 ARM 状态**
  - T = 1;J=0 处理器处于 Thumb 状态
  - T = 1;J=1 处理器处于 ThumbEE 状态
- Mode位：
  - 处理器模式位
  - 10000 User mode; 10001 FIQ mode; 10011 SVC mode;
    10111 Abort mode; 11011 Undfined mode; 11111 System mode;
    10110 Monitor mode; 10010 IRQ

ARM

09

协处理器、指令流水线

协处理器



ARM体系结构允许通过增加协处理器来扩展指令集。最常用的协处理器是用于控制片上功能的系统协处理器。

例如，控制Cache和存储管理单元MMU的CP15协处理器、设置异常向量表地址的mcr指令。

协处理器指令

- ARM支持16个协处理器，在程序执行过程中，每个协处理器忽略属于ARM处理器和其他协处理器指令，
- 当一个协处理器硬件不能执行属于她的协处理器指令时，就会产生一个未定义的异常中断，在异常中断处理程序中，可以通过软件模拟该硬件的操作，
- 比如，如果系统不包含向量浮点运算器，则可以选择浮点运算软件模拟包来支持向量浮点运算。

- ARM协处理器指令包括如下三类：
- 用于ARM处理器初始化ARM协处理器的数据操作
- 用于ARM处理器的寄存器和ARM协处理器的寄存器间的数据传送操作
- 用于在ARM协处理器的寄存器和内存单元之间传送数据
- 这些指令包括如下5条:
- CDP协处理器数据操作指令
- LDC协处理器数据读入指令
- STC协处理器数据写入指令
- MCR ARM寄存器到协处理器寄存器的数据传送指令
- MRC 协处理器寄存器到ARM寄存器的数据传送指令

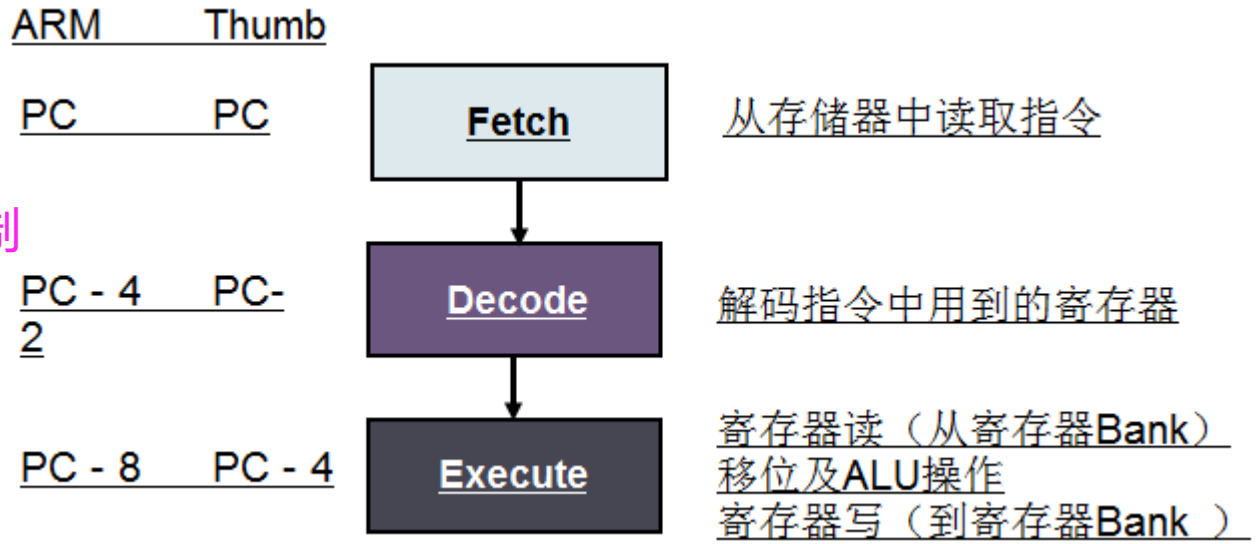
流水线

- 为什么引入流水线?

- 手术台的医生

3级流水线

（1）取指令
从寄存器装载一条指令。
（2）译码（decode）
识别被执行的指令，并为下一个周期准备数据通路的控制信号。在这一级，指令占有译码逻辑，不占用数据通路。
（3）执行
处理指令并将结果写回寄存器。



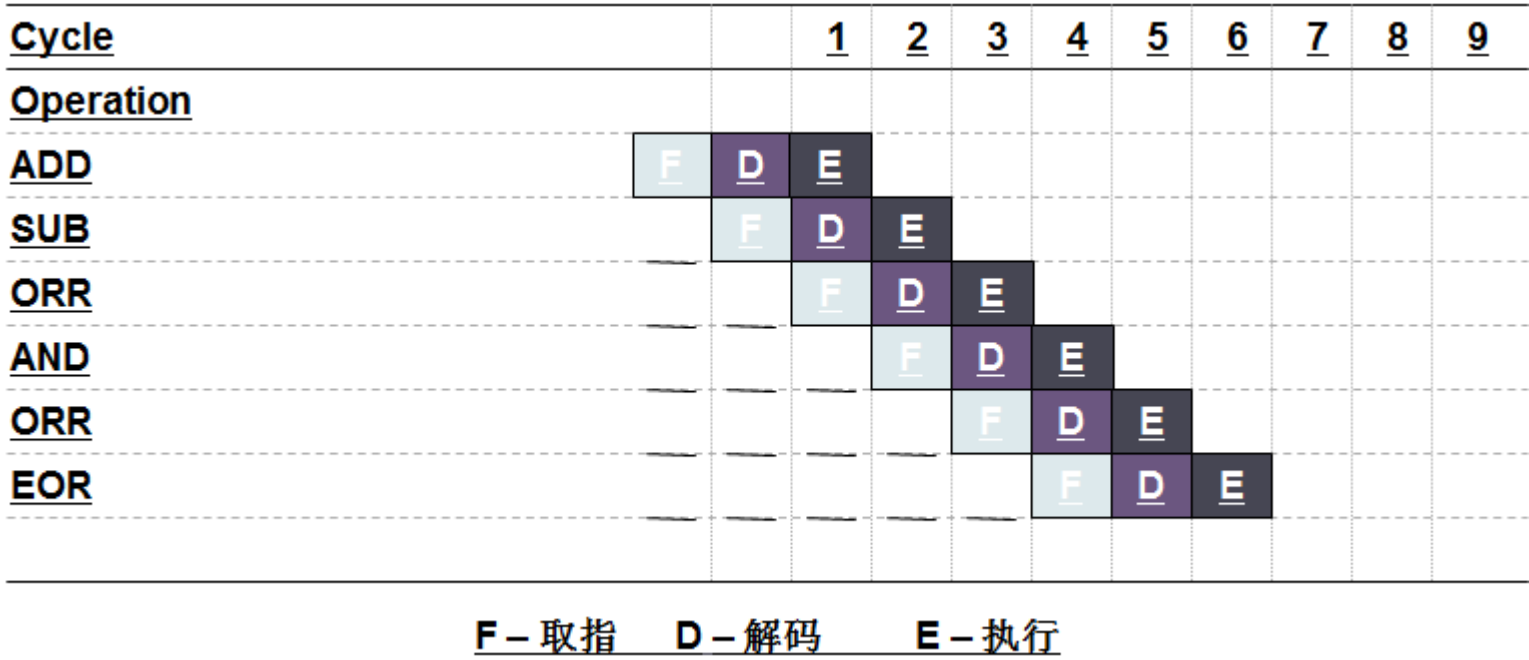
当处理器执行简单的数据处理指令时，流水线使得平均每个时钟周期能完成1条指令。但一条指令需要3个时钟周期来完成，因此有3个时钟周期的延时，但吞吐率是每个周期一条指令。
对于3级流水线，PC寄存器里的值并不是正在执行的指令的地址，而是预取指令的地址

从经典ARM系列到现在Cortex系列，ARM处理器的结构在向复杂的阶段发展，但没改变的是CPU的取址指令和地址关系，**不管是几级流水线，都可以按照最初的3级流水线的操作特性来判断其当前的PC位置。**

最佳流水线

这是一个理想的实例，所有的指令都在寄存器中执行，且处理器完全不必离开芯片本身。
每个周期，都有一条指令被执行，流水线的容量得到了充分的发挥。

指令周期数 (CPI) = 1



LDR流水线

与最佳流水线不同，装载(LDR) 操作将数据移进片内导致了指令/数据总线被占用，因此随后紧跟了内部的写周期（ writeback）以完成将数据写回寄存器。

参考

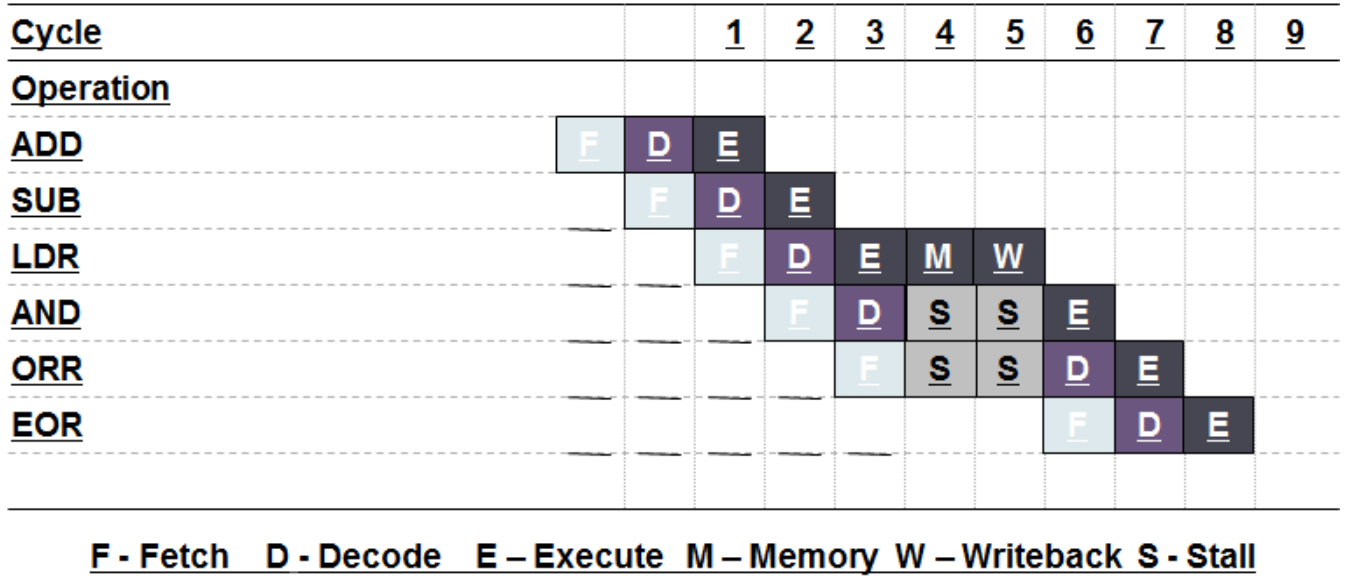
1、数据总线在周期1, 2, 3 被使用，周期6是取指，周期4用于数据装载，而周期5是一个内部周期用来完成载入的数据写回到寄存器中。

2、周期3为执行周期：产生地址

3、周器4为数据周期：从存储器中取数据（数据只有在周期4的末尾出现在内核中）

4、 周期5写回周期：通过数据通道中的B总线和ALU将数据写回到寄存器bank 中

5、 周期6的执行被推迟了，直到周期5写回完成（使用ALU ）。同样内部周期是不需要等待状体的，但读写存储器时可能需要。



分支流水线

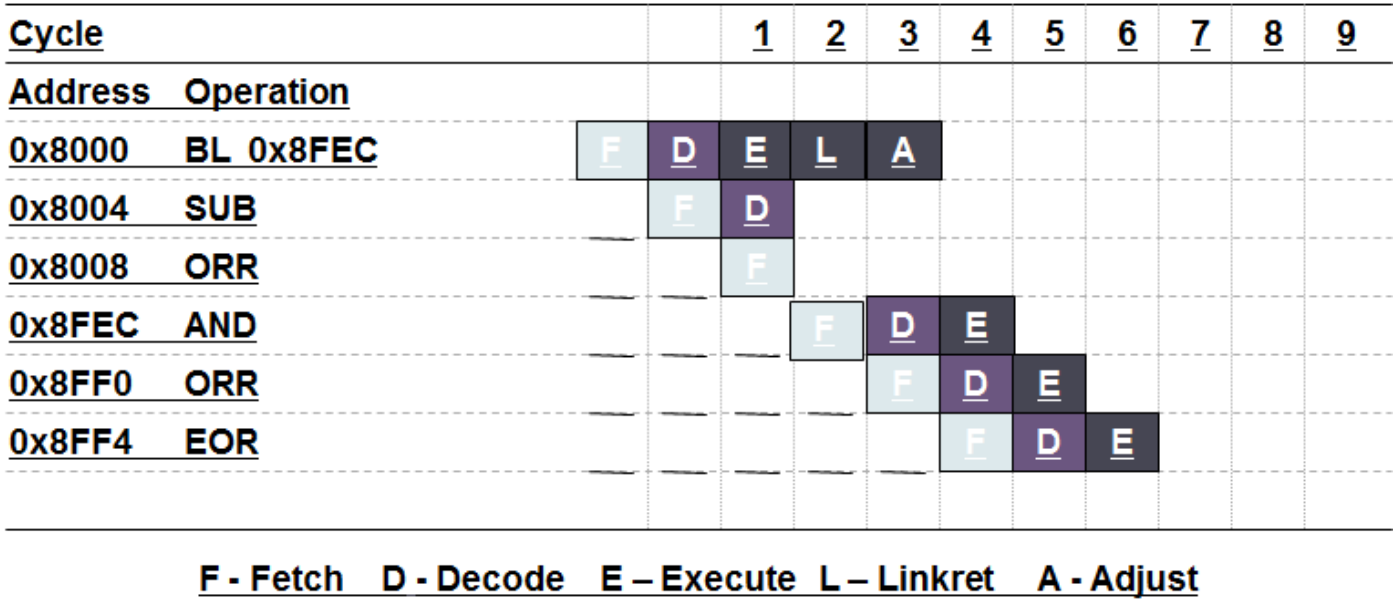
BL指令用于实现指令流的跳转，并存储返回地址到寄存器R14（LR）中

1. 分支指令在其第一周期计算分支的目的地，同时在现行PC处完成一次指令预取，流水线被阻断。这种预取在任何情况下都要做的，因为当判决地址产生时已来不及停止预取。

2. 第二个周期在分支的目标地址完成取指，而返回地址则存于R14如果link位已设置。

3. 第三周期完成目标地址+4的取指，重新填满流水线，并且如果跳转是带链接的还要修改R14（减去4）以便简单地返回。

4. 分支需要三个时钟周期来执行BL，随后会涉及调整阶段。



中断流水线

1. 周期1: 内核被告知有中断
IRQ在现行指令执行完之前不会被响应（ MUL and LDM/STM 指令会有长的延迟)
解码阶段：中断被解码（中断已使能，设置了相应标志位…）。如果中断被使能和服务，正常的指令将不会被解码。

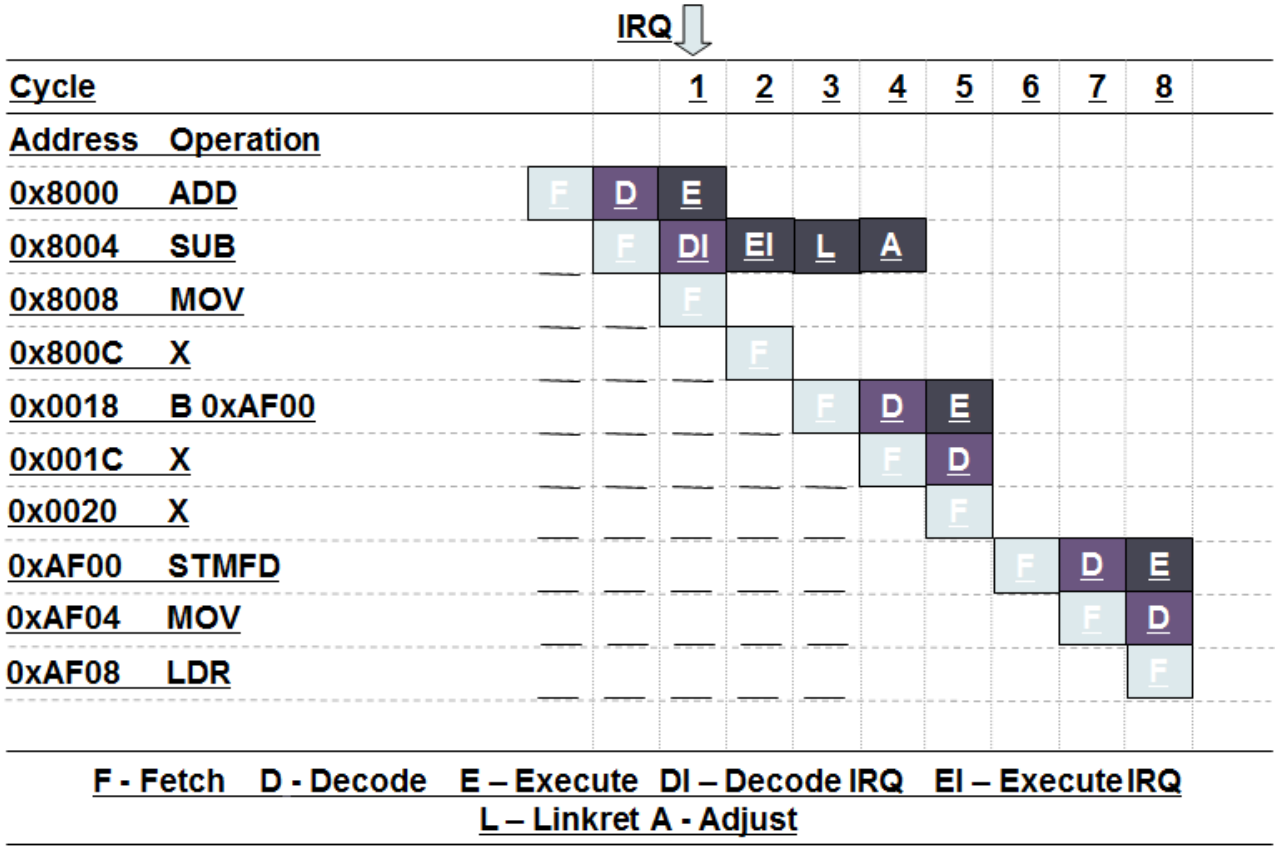
2. 周期 2: 此时总是进入ARM状态.
执行中断 (获取IR向量的地址), 保存 CPSR 于 SPSR, 改变CPSR模式为 IRQ 模式并禁止进一步的 IRQ 中断输入。

3. 周期 3: 保存 PC (0x800C) 于 r14_irq, 从IRQ异常处理向量处取指

4. 周期 4: 解码向量表中的指令; 调整r14irq 为0x8008

5. 周期 4和 5: 无有用的指令取指， 由于周期 6的跳转

6. 周期 6: 取异常处理子程序的第一条指令;
从子程序返回: SUBS pc,lr,#4



IRQ 中断的反应时间最小=7周期

ARM

10

ARM指令1
MOV、立即数

指令处理指令

- 数据处理指令
  数据处理指令可分为数据传送指令、算术逻辑运算指令和比较指令等。
- 1. 数据传送指令
- 数据传送指令用于在寄存器和存储器之间进行数据的双向传输。
- 2. 算术逻辑运算指令
- 算术逻辑运算指令完成常用的算术与逻辑的运算，该类指令不但将运算结果保存在目的寄存器中，同时更新CPSR中的相应条件标志位。

MOV

- 语法
  - MOV{条件}{S}　目的寄存器，源操作数
- 功能
  - MOV指令完成从另一个寄存器、被移位的寄存器或将一个立即数加载到目的寄存器。其中S选项决定指令的操作是否影响CPSR中条件标志位的值，当没有S时指令不更新CPSR中条件标志位的值

指令示例

- MOV r0, #0x1 ;将立即数0x1传送到寄存器R0
- MOV R1，R0 ;将寄存器R0的值传送到寄存器R1
- MOV PC，R14 ;将寄存器R14的值传送到PC，常用于子程序返回
- MOV R1，R0，LSL ＃3 ;将寄存器R0的值左移3位后传送到R1

- 举例

什么是立即数?

- 立即数是由 0-255之间的数据循环右移偶数位生成。
- 判断规则如下：
- 1. 把数据转换成二进制形式，从低位到高位写成4位1组的形式，最高位一组不够4位的，在最高位前面补0。
- 2. 数1的个数，如果大于8个肯定不是立即数，如果小于等于8进行下面步骤。
- 3. 如果数据中间有连续的大于等于24个0,循环左移2的倍数，使高位全为0。
- 4. 找到最高位的1，去掉前面最大偶数个0。
- 5. 找到最低位的1，去掉后面最大偶数个0。
- 6. 数剩下的位数，如果小于等于8位，那么这个数就是立即数，反之就不是立即数。

举例

- MOV R0，#0xfff
- 0000 0000 0000 0000 0000 1111 1111 1111
- 1111 1111 1111

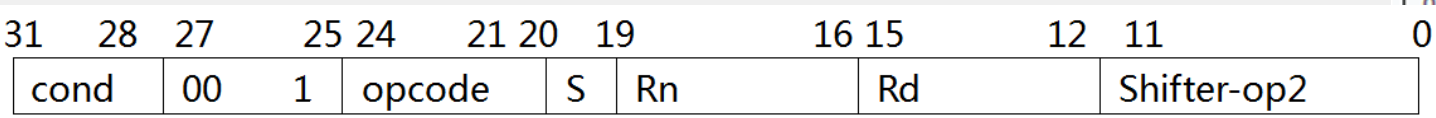
MOV机器码

```
    AREA Example,CODE,READONLY     ;声明代码段Example
    ENTRY ;程序入口
Start
//测试代码，添加在以下位置即可，后面不再贴完整代码
    mov r1,#0x80000001
OVER
    END
```

Disassembly

```
0x00000000   E3A01106   MOV      R1,#0x80000001
0x00000004   00000000   ANDEQ    R0,R0,R0
0x00000008   00000000   ANDEQ    R0,R0,R0
```

ARM指令助记符



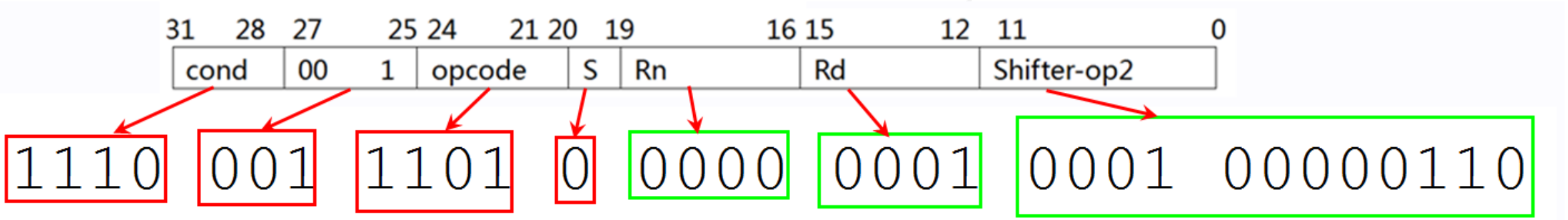
条件码	助记符后缀	标 志	含 义
0000	EQ	Z置位	相等
0001	NE	Z清零	不相等
0010	CS	C置位	无符号数大于或等于
0011	CC	C清零	无符号数小于
0100	MI	N置位	负数
0101	PL	N清零	正数或零
0110	VS	V置位	溢出
0111	VC	V清零	未溢出
1000	HI	C置位Z清零	无符号数大于
1001	LS	C清零Z置位	无符号数小于或等于
1010	GE	N等于V	带符号数大于或等于
1011	LT	N不等于V	带符号数小于
1100	GT	Z清零且（N等于V）	带符号数大于
1101	LE	Z置位或（N不等于V）	带符号数小于或等于
1110	AL	忽略	无条件执行

- **{<cond>}：条件码域**
- **<opcode>：操作码域**
- **{S}：条件码设置域：**这是一个可选项，当在指令中设置{S}域时，指令执行的结果将会影响程序状态寄存器CPSR中相应的状态标志
- **<Rd>：目的操作数：**总是一个寄存器
- **<Rn>：第一操作数：**也必须是个寄存器
- **<shift_op2>：第二操作数：**在第二操作数中可以是寄存器、内存存储单元或者立即数

MOV指令解析

mov r1, #0x80000001

E3A01106



各个位域

bite	含义
1110	Cond忽略
00 1	
1101	opcode
0	s 命令不含S
0000	rn，没有源寄存器为0
0001	rd 目的结存器R0
0001	shifter
0000 0110	操作数

第二操作数如果是立即数：

bit:[11-8]表示操作数向左移动的位数/2,

bit:[7-0]表示最终的操作数

立即数0x80000001二进制为：

1000 0000 0000 0000 0000 0000 0000 0001

循环左移2位后得到以下结果：

00 0000 0000 0000 0000 0000 0000 0001 10

ARM

11

ARM指令2
移位操作

移位操作

- ARM微处理器支持数据的移位操作，移位操作在ARM指令集中不作为单独的指令使用，它只能作为指令格式中是一个字段，在汇编语言中表示为指令中的选项。

MOV R0, R1, LSL#2

移位操作

- 移位操作包括如下6种类型
- 1. LSL（或ASL）逻辑（算术）左移 （ASL与LSL等价）
- 2. LSR逻辑右移
- 3. ASR算术右移
- 4. ROR循环右移
- 5. RRX带扩展的循环右移

1) LSL（或ASL）逻辑（算术）左移

寻址格式：

通用寄存器，LSL（或ASL） 操作数

完成对通用寄存器中的内容进行逻辑（或算术）的左移操作，按操作数所指定的数量向左移位，**低位用零来填充**。

其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

MOV R0, R1, LSL #2； 将R1中的内容左移两位后传送到R0中。

2) LSR逻辑右移

- 寻址格式：
  - 通用寄存器，LSR 操作数
- 完成对通用寄存器中的内容进行右移的操作，按操作数所指定的数量向右移位，左端用零来填充。
- 其中，操作数可以是通用寄存器，也可以是立即数（0 ~ 31）。
- 如：

- MOV　R0, R1, LSR #2　；将R1中的内容右移两位后传送到R0中，左端用零来填充。

3) ASR算术右移

- 寻址格式：
  - 通用寄存器，ASR 操作数
- 完成对通用寄存器中的内容进行右移的操作，按操作数所指定的数量向右移位，左端用第31位的值来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。
- 如：
- MOV　R0, R1, ASR #2　　；将R1中的内容右移两位后传送到R0中，左端用第31位的值来填充。

4) ROR循环右移

- 寻址格式：
  - 通用寄存器，ROR 操作数
- 完成对通用寄存器中的内容进行循环右移的操作，按操作数所指定的数量向右循环移位，左端用右端移出的位来填充。
- 其中，操作数可以是通用寄存器，也可以是立即数（0～31）。
- 显然，当进行32位的循环右移操作时，通用寄存器中的值不改变。
- 如：
- MOV　R0, R1, ROR #2；将R1中的内容循环右移两位后传送到R0中。

5）RRX带扩展的循环右移

- 寻址格式：
  - 通用寄存器，RRX 操作数
- 完成对通用寄存器中的内容进行带扩展的循环右移的操作，按操作数所指定的数量向右循环移位，左端用进位标志位C来填充。
- 其中，操作数可以是通用寄存器，也可以是立即数（0～31）。
- 如：
- MOV　R0, R1, RRX #2　　；将R1中的内容进行带扩展的循环右移两位后传送到R0中。

- 举例

ARM

12

ARM指令3
CMP、TST

CMP比较指令

- 语法
  - CMP{条件} 操作数1，操作数2
- CMP指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行比较，同时更新CPSR中条件标志位的值。该指令进行一次减法运算，但不存储结果，只更改条件标志位。cmp是做一次减法，并不保存结果，仅仅用来产生一个逻辑，体现在改变cpsr相应的condition位。

- 标志位表示的是操作数1与操作数2的关系(大、小、相等)，
- 指令示例：
  - CMP R1，R0 ；将寄存器R1的值与寄存器R0的值相减，并根据结果设置CPSR的标志位
  - CMP R1，#100；将寄存器R1的值与立即数100相减，并根据结果设置CPSR的标志位

TST条件指令

- 语法
  - TST{条件} 操作数1，操作数2
- TST指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行按位的与运算，并根据运算结果更新CPSR中条件标志位的值。操作数1是要测试的数据，而操作数2是一个位掩码，根据测试结果设置相应标志位。当位与结果为0时，EQ位被设置。
- 指令示例
- TST R1，#%1 ；用于测试在寄存器R1中是否设置了最低位（%表示二进制数）。

条件码	助记符后缀	标 志	含 义
0000	EQ	Z置位	相等
0001	NE	Z清零	不相等
0010	CS	C置位	无符号数大于或等于
0011	CC	C清零	无符号数小于
0100	MI	N置位	负数
0101	PL	N清零	正数或零
0110	VS	V置位	溢出
0111	VC	V清零	未溢出
1000	HI	C置位Z清零	无符号数大于
1001	LS	C清零Z置位	无符号数小于或等于
1010	GE	N等于V	带符号数大于或等于
1011	LT	N不等于V	带符号数小于
1100	GT	Z清零且（N等于V）	带符号数大于
1101	LE	Z置位或（N不等于V）	带符号数小于或等于
1110	AL	忽略	无条件执行

例1：找出三个寄存器中数据最大的数

- mov r0, #3
- mov r1, #4
- mov r2, #5
- cmp r1,r0
- movgt r0,r1
- cmp r2,r0
- movgt r0,r2

例2：求两个数的差的绝对值

- mov r0,#9
- mov r1,#15
- cmp r0,r1
- beq stop
- subgt r0,r0,r1
- sublt r1,r1,r0

ARM

13

ARM指令4
数据处理指令

数据处理指令

- ADD
- ADC
- SUB
- SBC
- AND
- ORR
- BIC

ADD

- ADD{条件}{S} 目的寄存器，操作数1，操作数2

- ADD指令用于把两个操作数相加，并将结果存放到目的寄存器中。
  操作数1应是一个寄存器，操作数2可以是一个寄存器，被移位的寄存器，或一个立即数。
- 指令示例：
  - ADD　R0，R1，R2　　；　R0 = R1 + R2
  - ADD　R0，R1，#256　　；　R0 = R1 + 256
  - ADD　R0，R2，R3，LSL#1　　；　R0 = R2 + (R3 << 1)

举例

- 1.加法运算
- mov r0, #1
- mov r1, #2
-
- add r2, r0, r1 ; r2 = r0 + r1
- add r2, r0, #4
- add r2, r0, r1, lsl #2 ; r2 = r0 + R1 < < 2 ; (R0 + R1*4)

ADC

- 除了正常做加法运算之外，还要加上CPSR中的C条件标志位，如果要影响CPSR中对应位，加后缀S。

SUB

- SUB指令的格式为：
- SUB{条件}{S} 目的寄存器，操作数1，操作数2
- SUB指令用于把操作数1减去操作数2，并将结果存放到目的寄存器中。
- 操作数1应是一个寄存器，操作数2可以是一个寄存器，被移位的寄存器，或一个立即数。
- 该指令可用于有符号数或无符号数的减法运算。
- 指令示例：
  - SUB R0，R1，R2 ； R0 = R1 - R2
  - SUB R0，R1，#256 ； R0 = R1 - 256
  - SUB R0，R2，R3，LSL#1 ； R0 = R2 - (R3 << 1)

SBC

- 除了正常做加法运算之外，还要再减去CPSR中C条件标志位的反码
- 根据执行结果设置CPSR对应的标志位

AND

- AND指令的格式为：
  - AND{条件}{S} 目的寄存器，操作数1，操作数2
- AND指令用于在两个操作数上进行与运算（按位与），并把结果放置到目的寄存器中。
- 操作数1应是一个寄存器，操作数2可以是一个寄存器，被移位的寄存器，或一个立即数。
- 该指令常用于屏蔽操作数1的某些位。

- 举例
  - AND　R0，R0，＃3　　；该指令保持R0的0、1位，其余位清零。

ORR

- ORR指令的格式为：
  - ORR{条件}{S} 目的寄存器，操作数1，操作数2
- ORR指令用于在两个操作数上进行逻辑或运算，并把结果放置到目的寄存器中。操作数1应是一个寄存器，操作数2可以是一个寄存器，被移位的寄存器，或一个立即数。该指令常用于设置操作数1的某些位。
- 举例：
  - ORR　R0，R0，＃3　　；该指令设置R0的0、1位，其余位保持不变。

BIC

- BIC指令用于清除操作数1的某些位，并把结果放置到目的寄存器中。
- BIC指令的格式为：
  - BIC{条件}{S} 目的寄存器，操作数1，操作数2
- 操作数1应是一个寄存器，操作数2可以是一个寄存器，被移位的寄存器，或一个立即数。
- 操作数2为32位的掩码，如果在掩码中设置了某一位，则清除这一位。未设置的掩码位保持不变。
- 举例
  - BIC　R0，R0，＃%1011　；该指令清除 R0 中的位 0、1、和 3，其余的位保持不变。

2. 64位加法运算的实现

- adc 64位加法 （r0, r1） = （r0, r1） + （r2, r3）
- mov r0, #0
- mov r1, #0xffffffff
- mov r2, #0
- mov r3, #0x1
- adds r1, r1, r3 ; r1 = r1 + r3 必须加S后缀
- adc r0, r0, r2 ; r0 = r0 + r2 + c;add 带 扩展的加法

3. 减法

- ; 3. sub rd = rn - op2
- mov r0, #1
- sub r0, r0, #1 ; r0 = r0 - 1

4. 64位减法

```
; 4. sbc  64位减法 r0, r1 =  r0, r1 - r2, r3
    ; cpsr c 对于加法运算 C = 1 则代表有进位，  C = 0 无进位
        ;                  对于减法运算 C = 1 则代表无借位，  C = 0 有借位
        mov r0, #0
        mov r1, #0x0
        mov r2, #0
        mov r3, #0x1
        subs r1, r1, r3
        sbc r0, r0, r2          ;sbc  带扩展的减法
```

5.位清除

- ; 5. bic 位清除
- mov r0, #0xffffffff
- bic r0, r0, #0xff ; and r0, r0, #0xffffff00

ARM

14

ARM指令5
跳转指令B、BL

跳转指令

- 跳转指令用于实现程序流程的跳转，在ARM程序中有两种方法可以实现程序流程的跳转：
  - 使用专门的跳转指令；
    - B　　跳转指令
    - BL　　带返回的跳转指令
    - BLX　　带返回和状态切换的跳转指令thumb指令
    - BX　　带状态切换的跳转指令thumb指令
  - 直接向程序计数器PC写入跳转地址值，通过向程序计数器PC写入跳转地址值，可以实现在4GB的地址空间中的任意跳转，在跳转之前结合使用。
  - 　MOV LR,PC

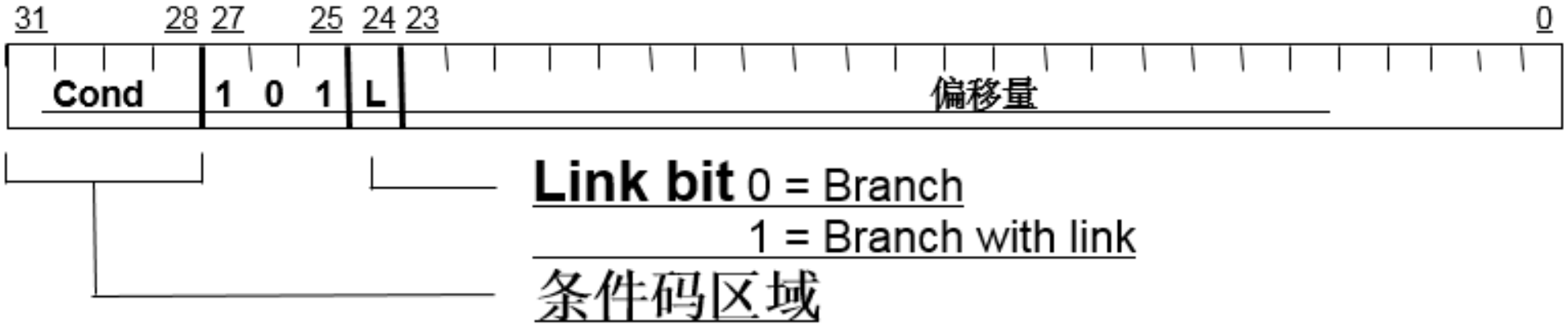
B 指令

- 指令的格式为：
  - B{条件} 目标地址
- B指令是最简单的跳转指令。一旦遇到一个 B 指令，ARM 处理器将立即跳转到给定的目标地址，从那里继续执行。
  - B label 程序无条件跳转到标号label处执行
  - CMP R1 ， #0
  - BEQ label 当CPSR寄存器中的Z条件码置位时，程序跳转到标号Label处执行。

BL 指令

- BL 指令的格式为：
  - BL{条件} 目标地址
- BL是另一个跳转指令，但跳转之前，会在寄存器R14中保存PC当前值，因此，可以通过将R14 的内容重新加载到PC中，来返回到跳转指令之后的那个指令处执行。该指令是实现子程序调用的一个基本但常用的手段。
  - BL label 当程序无条件跳转到标号Label处执行时，同时将当前的PC值保存到R14中
- 子函数要返回执行以下指令即可：
  - MOV PC,LR

BL指令机器码



域	含义
cond	条件码
101	操作码
L	命令是否包含L
offset	指令跳转偏移量

B指令寻址空间

- 机器指令中offset是24个bite，最高位包含一个符号位，1个单位表示偏移一条指令，所以可以寻址$\pm 2^{23}$条指令，即±8M条指令。
- 而一条指令是4个字节，所以最大寻址空间为**±32MB的地址空间。**

举例-查看offset值

```
        AREA Example,CODE,READONLY
        ENTRY ;程序入口
Start
        MOV R0,#0
        MOV R1,#10
        BL ADD_SUM
        B OVER
ADD_SUM
        ADD R0,R0,R1
        MOV PC,LR
OVER
        END
```

如何访问全部32-bit地址空间?

可以手动设置LR寄存器，然后装载到PC中。

```
MOV lr, pc
LDR pc, =dest
```

在编译项目过程中，ARM连接器（linker）会自动为长跳转（超过32Mb范围）。

举例-子函数跳转

```
 area first, code, readonly
        code32
        entry
main
        ; bl 指令, 子函数调用
        mov r0,#1
        bl child_func
        mov r0,#2
stop
        b stop
child_func
        mov r1,r0
        mov r2,lr
        mov r0, #3          ; <===  pc
        bl child_func_2
        mov r0,#4
        mov r0,r1
        mov lr,r2
        mov pc, lr
child_func_2 ;叶子函数
        mov r3,r0
        mov r4,lr  ; 保存直接父函数用到的所有寄存器
        mov r0, #5
        mov r0,r3
        mov lr,r4 ;返回到直接父函数之前，把它用到的所有寄存器内容恢复
        mov pc, lr
        end
```

- 由上述例子所示，每调用一级子函数，我们都把返回地址存入到未分组寄存器中，但是未分组寄存器毕竟是有限的，像Linux内核函数的调用层次往往很深，通用寄存器根本不够用，要想保存返回地址，就需要对数据进行压栈，那我们就要为每个模式的栈设置空间，那如何设置栈空间呢？下一篇我们继续讨论。

ARM

15

ARM指令6
MRS、MSR

程序状态寄存器访问指令MRS、MSR

- ARM微处理器支持CPSR(程序状态寄存器)访问指令，
  - MRS、MSR
- 用于在程序状态寄存器和通用寄存器之间传送数据。

MRS

- MRS{条件}　　通用寄存器，程序状态寄存器（CPSR或SPSR）

- MRS指令用于将程序状态寄存器的内容传送到通用寄存器中。
- 该指令一般用在以下几种情况：

- 当需要改变程序状态寄存器的内容时，可用MRS将程序状态寄存器的内容读入通用寄存器，修改后再写回程序状态寄存器。
- 当在异常处理或进程切换时，需要保存程序状态寄存器的值，可先用该指令读出程序状态寄存器的值，然后保存。
- 如：
  - MRS　　R0，CPSR　　；传送CPSR的内容到R0
  - MRS　　R0，SPSR　　；传送SPSR的内容到R0

MSR

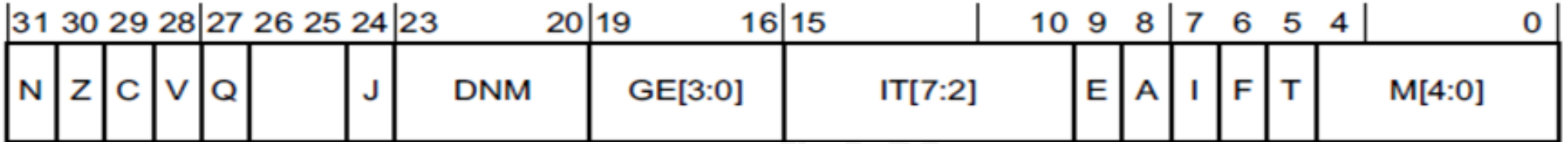
- MSR{条件}　　程序状态寄存器（CPSR或SPSR）_<域>，操作数
- MSR指令用于将操作数的内容传送到程序状态寄存器的特定域中。
- 其中，操作数可以为通用寄存器或立即数。
- <域>用于设置程序状态寄存器中需要操作的位，32位的程序状态寄存器可分为4个域：
  - 位[31：24]为条件标志位域，用f表示；
  - 位[23：16]为状态位域，用s表示；
  - 位[15：8]为扩展位域，用x表示；
  - 位[7：0]为控制位域，用c表示；

MSR

- 该指令通常用于恢复或改变程序状态寄存器的内容，在使用时，一般要在MSR指令中指明将要操作的域。
  如：
  - MSR　CPSR，R0　　；传送R0的内容到CPSR
  - MSR　SPSR，R0　　；传送R0的内容到SPSR
  - MSR　CPSR_c，R0　　；传送R0的内容到CPSR，但仅仅修改CPSR中的控制位域

应用举例-1. 使能中断

- 要是能中断，必须将寄存器CPSR的bit[7]设置为0



- 要将寄存器CPSR的bit[7]设置为0，但是不能影响其他位，
- 所以必须先用msr读取出cpsr的值到通用寄存器Rn（n取值0~8）
- 然后修改bit[7]设置为0，再将该寄存器的值设置到CPSR中

代码

```
        area reset,code
        code32
        entry
start
        bl enale_irq
enale_irq
        mrs r0,cpsr
        bic r0,r0,#0x80
        msr cpsr_c,r0
        mov pc,lr
```

2. 禁止中断

```
        area reset,code
        code32
        entry
start
        bl diable_irq
diable_irq
        mrs r0,cpsr
        orr r0,r0,#0x80
        msr cpsr_c,r0
        mov pc,lr
        end
```

ARM

16

ARM模式栈初始化

设置各模式的栈地址

- 要想初始化各个模式的栈地址，必须首先切换到对应的模式，然后再将栈地址设置到寄存器sp即可。

模式分类	处理器工作模式	异常模式	说明
非特权模式	用户（user）		用户程序运行模式
特权模式 该模式下可以访问系统资源	系统（system）		运行特权级的操作系统任务
	一般中断（IRQ）	异常模式 通常由系统异常状态切换进该组模式	普通中断模式
	快速中断（FIR）		快速中断模式
	管理（supervisor）		提供操作系统使用的一种保护模式，swi命令状态
	中止（abort）		虚拟内存管理和内存数据访问保护
	未定义指令终止（undefined）		支持通过软件仿真硬件的协处理
	monitor		用于执行安全监控代码的模式

如何设置栈指针

- SP(R13)在ARM指令中常用作**堆栈指针**,用户也可使用其他的寄存器作为堆栈指针。

- 处理器的每种运行模式均有自己独立的物理寄存器R13，在用户应用程序的初始化部分，一般都要初始化每种模式下的R13，使其指向该运行模式的栈空间。

- 这样，当程序的运行进入异常模式时，可以将需要保护的寄存器放入R13所指向的堆栈，而当程序从异常模式返回时，则从对应的堆栈中恢复，采用这种方式可以保证异常发生后程序的正常执行。

代码

```
1     area reset,code
2     code32
3     entry
4  start
5     bl stack_init
6  stack_init                    ; 栈指针初始化函数
7  ;    @undefine_stack
8     msr cpsr_c,#0xdb           ; 切换到未定义异常
9     ldr sp,=0x34000000      ; 栈指针为内存最高地址，栈为倒生的栈
10                               ; 栈空间的最后1M 0x34000000~0x33f00000
11 ;    @abort_stack
12    msr cpsr_c,#0xd7              ; 切换到终止异常模式
13    ldr        sp,=0x33f00000     ; 栈空间为1M，0x33f00000~0x33e00000
14 ;   @irq_stack
15    msr      cpsr_c,#0xd2         ; 切换到中断模式
16    ldr        sp,=0x33e00000     ; 栈空间为1M，0x33e00000~0x33d00000
17 ;   @ sys_stack
18    msr  cpsr_c,#0xdf            ; 切换到系统模式
19    ldr  sp,=0x33d00000          ; 栈空间为1M，0x33d00000~0x33c00000
20    msr  cpsr_c,#0xd3         ; 切换回管理模式
21    mov pc,lr
22    end
```

bite	模式	ARM模式可访问的寄存器
0b10000	用户模式user	PC,CPSR,R0~R14
0b10001	FIQ模式	PC,CPSR,SPSR_fiq,R14_fiq~R8_fiq,R0~R7
0b10010	IRQ模式	PC,CPSR,SPSR_irq,R14_irq~R13_irq,R0~R12
0b10011	管理模式	PC,CPSR,SPSR_svc,R14_svc~R13_svc,R0~R12
0b10111	中止模式Abort	PC,CPSR,SPSR_abt,R14_abt~R13_abt,R0~R12
0b11011	未定义模式	C,CPSR,SPSR_und,R14_und~R13_und,R0~R12
0b11111	系统模式	PC,CPSR,R0~R14

异常向量表

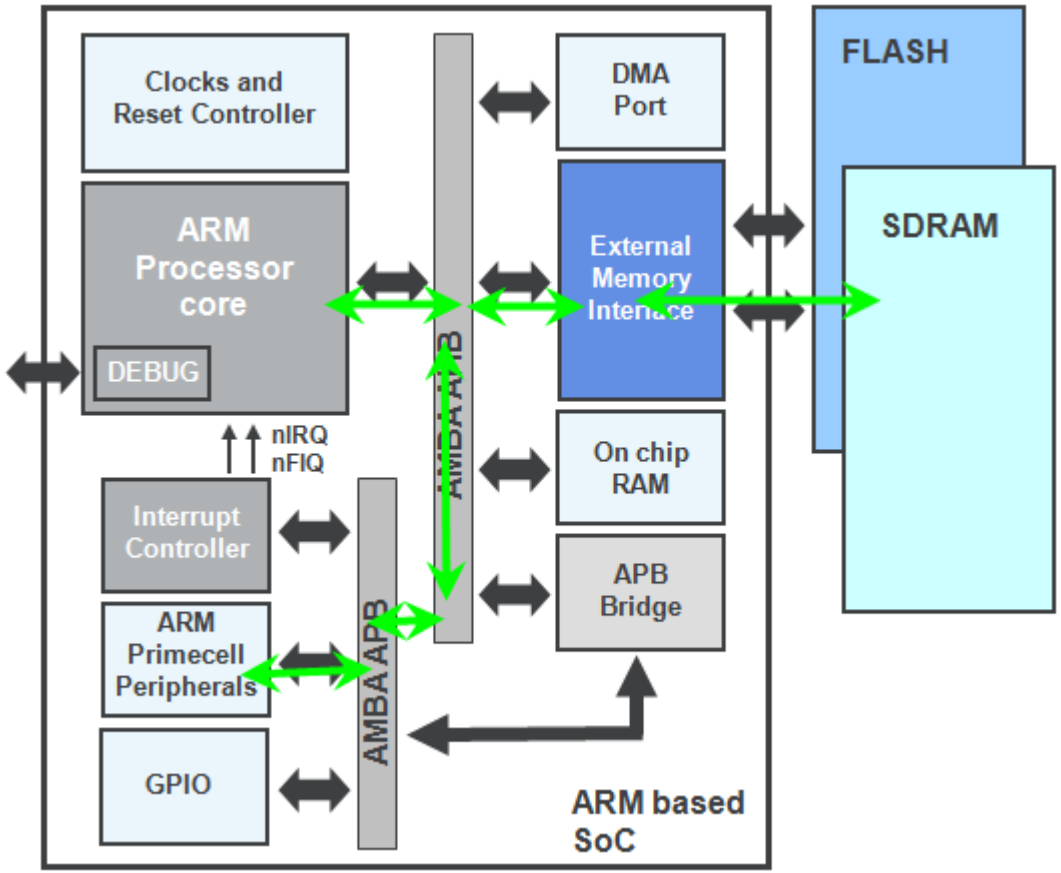
```
38 .globl _start
39 _start: b   reset
40     ldr pc, _undefined_instruction
41     ldr pc, _software_interrupt
42     ldr pc, _prefetch_abort
43     ldr pc, _data_abort
44     ldr pc, _not_used
45     ldr pc, _irq
46     ldr pc, _fiq
```

```
126 reset:
127     bl  save_boot_params
128     /*
129      * set the cpu to SVC32 mode
130      */
131     mrs r0, cpsr
132     bic r0, r0, #0x1f
133     orr r0, r0, #0xd3
134     msr cpsr,r0
135
136 #if 1
137         ldr r0, =0x11000c40 @GPK2_7 led2
138         ldr r1, [r0]
139         bic r1, r1, #0xf0000000
140         orr r1, r1, #0x10000000
141         str r1, [r0]
142
143         ldr r0, =0x11000c44
144         mov r1,#0xff
145         str r1, [r0]
146 #endif
147
148 /*
149  * Setup vector:
150  * (OMAP4 spl TEXT_BASE is not 32 byte aligned.
151  * Continue to use ROM code vector only in OMAP4 spl)
152  */
153 #if !(defined(CONFIG_OMAP44XX) && defined(CONFIG_SPL_BUILD))
154     /* Set V=0 in CP15 SCTRL register - for VBAR to point to vector */
155     mrc p15, 0, r0, c1, c0, 0   @ Read CP15 SCTRL Register
156     bic r0, #CR_V       @ V = 0
157     mcr p15, 0, r0, c1, c0, 0   @ Write CP15 SCTRL Register
158
159     /* Set vector address in CP15 VBAR register */
160     ldr r0, =_start
161     mcr p15, 0, r0, c12, c0, 0  @Set VBAR
162 #endif
```

ARM®

17 LDR、STR详解

如何访问外存?



LDR指令

- LDR指令的格式为：
  - LDR{条件} 目的寄存器，<存储器地址>

- 功能：
  - LDR指令用于从存储器中将一个32位的字数据传送到目的寄存器中。
  - 当程序计数器PC作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。

STR指令

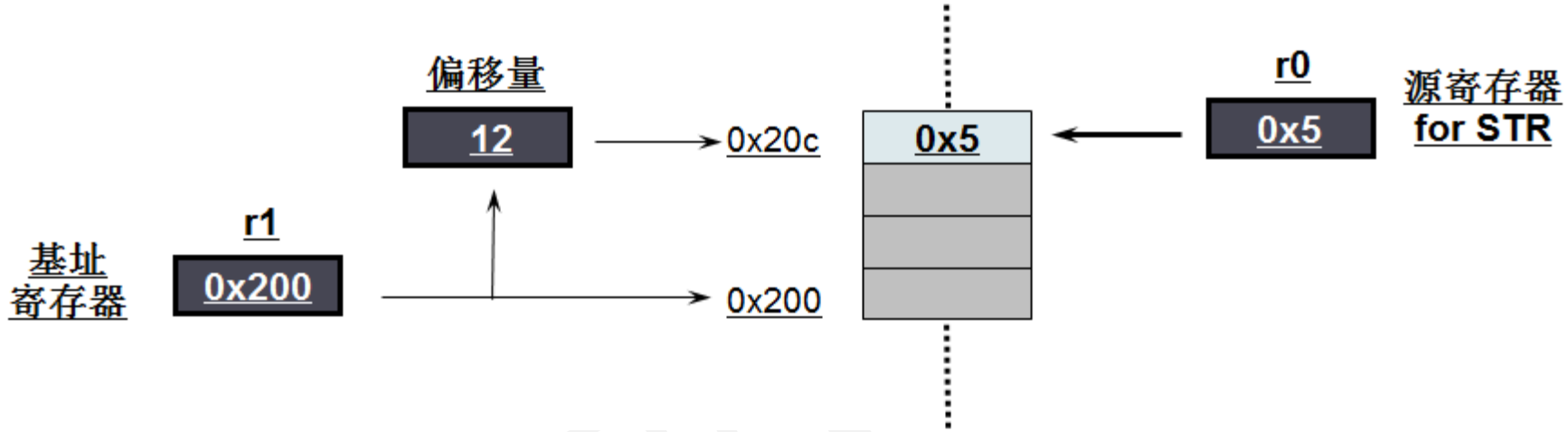
- STR指令的格式为：
  STR{条件} 源寄存器，<存储器地址>

- 功能：
  - STR指令用于从源寄存器中将一个32位的字数据传送到存储器中。
-

STR举例

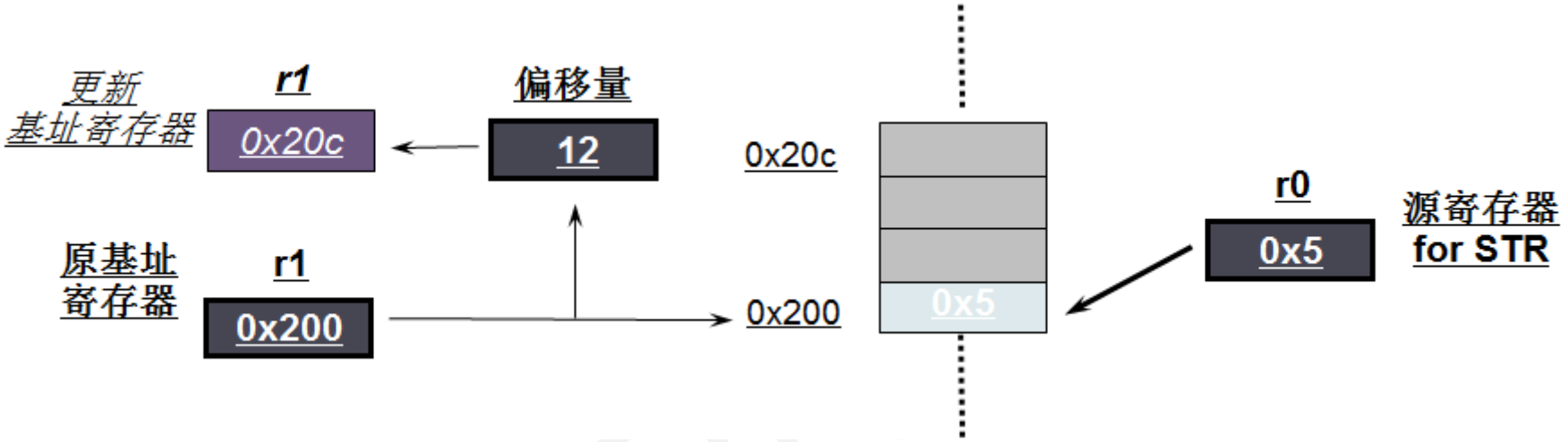
- STRR0，[R1]，＃12　　；将R0中的字数据写入以R1为地址的存储器中，并将新地址R1 + 12写入R1。
- STRR0，[R1，＃12]　　；将R0中的字数据写入以R1 + 12为地址的存储器中。

STR r0,[r1,#12]



1.寄存器r0中的值是0x5,r1中的值是0x200
2.将r1的值加上#12，得到地址0x20c
3.将r0寄存器里的值发送给该地址对应的内存，即向地址0x20c中赋值0x5

STR r0,[r1],#12



1.寄存器r0的值是0x5,r1中的值是0x200
2.将r0寄存器里的值发送给该r1中的值对应的内存，即向地址0x200中赋值0x5
3.将r1的值加上#12并赋值给r1，r1的值就变成了0x20c

LDR举例

- LDR　R0，[R1]　　；将存储器地址为R1的字数据读入寄存器R0。
- LDR　R0，[R1，R2]　　；将存储器地址为R1+R2的字数据读入寄存器R0。
- LDR　R0，[R1，＃8]　　；将存储器地址为R1+8的字数据读入寄存器R0。
- LDR　R0，[R1，R2]！　　；将存储器地址为R1+R2的字数据读入寄存器R0，并将
- ；新地址R1＋R2写入R1。
- LDR　R0，[R1，＃8]！　　；将存储器地址为R1+8的字数据读入寄存器R0，并将新
- ；地址R1＋8写入R1。
- LDR　R0，[R1]，R2　　；将存储器地址为R1的字数据读入寄存器R0，并将新地
- ；址R1＋R2写入R1。
- LDR　R0，[R1，R2，LSL＃2]!　　；将存储器地址为R1＋R2×4的字数据读入寄存器R0，
- ；并将新地址R1＋R2×4写入R1。
- LDR　R0，[R1]，R2，LSL＃2　；将存储器地址为R1的字数据读入寄存器R0，并将新地
- ；址R1＋R2×4写入R1。

后缀

- LDR/STR指令都可以加B、H、SB、SH的后缀，
- 分别表示加载/存储字节、半字、带符号的字节、带符号的半字。

- 如LDRB指令表示从存储器加载一个字节进寄存器。当使用这些后缀时，要注意所使用的存储器要支持访问的数据宽度。

为什么设计这么多操作?

- 有以下C代码
  - int *ptr;
  - x = *ptr++;

- 可以翻译成:
  - LDR r0, [r1], #4

ARM

18

ARM寻址方式

LDR、STR举例

```
    area first, code, readonly
    code32
    entry
start
    mov r0, #0x10000003
    mov r1, #0x40000000   ; SAMSUNG 2410 , 2410 = > sram 0x40000000  0x3fffff00
    str r0, [r1]              ;内存单元的地址r1寄存器的内容指示
    ldr r2,[r1]
stop
    b stop
    end
```

S3C2440A内存映射

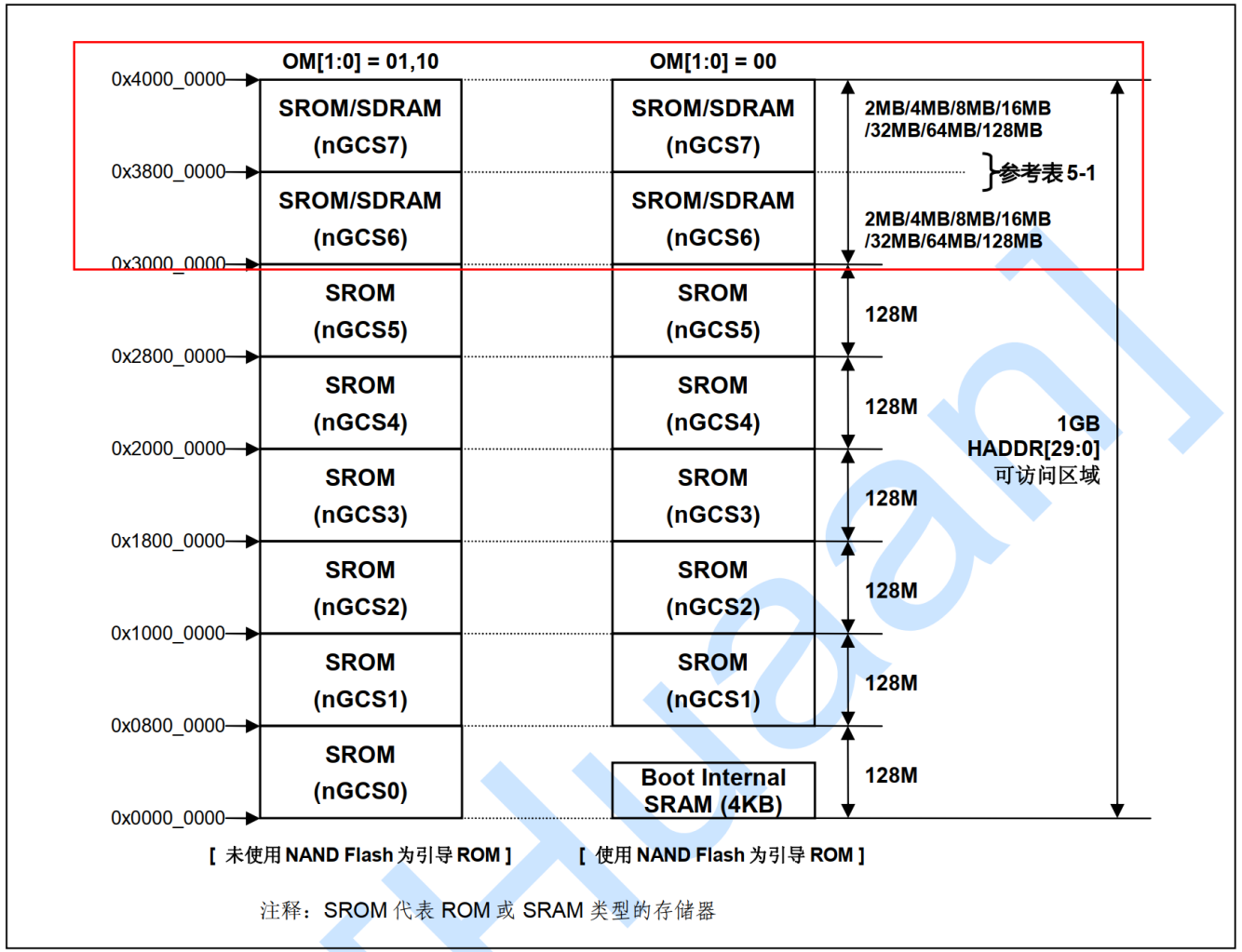
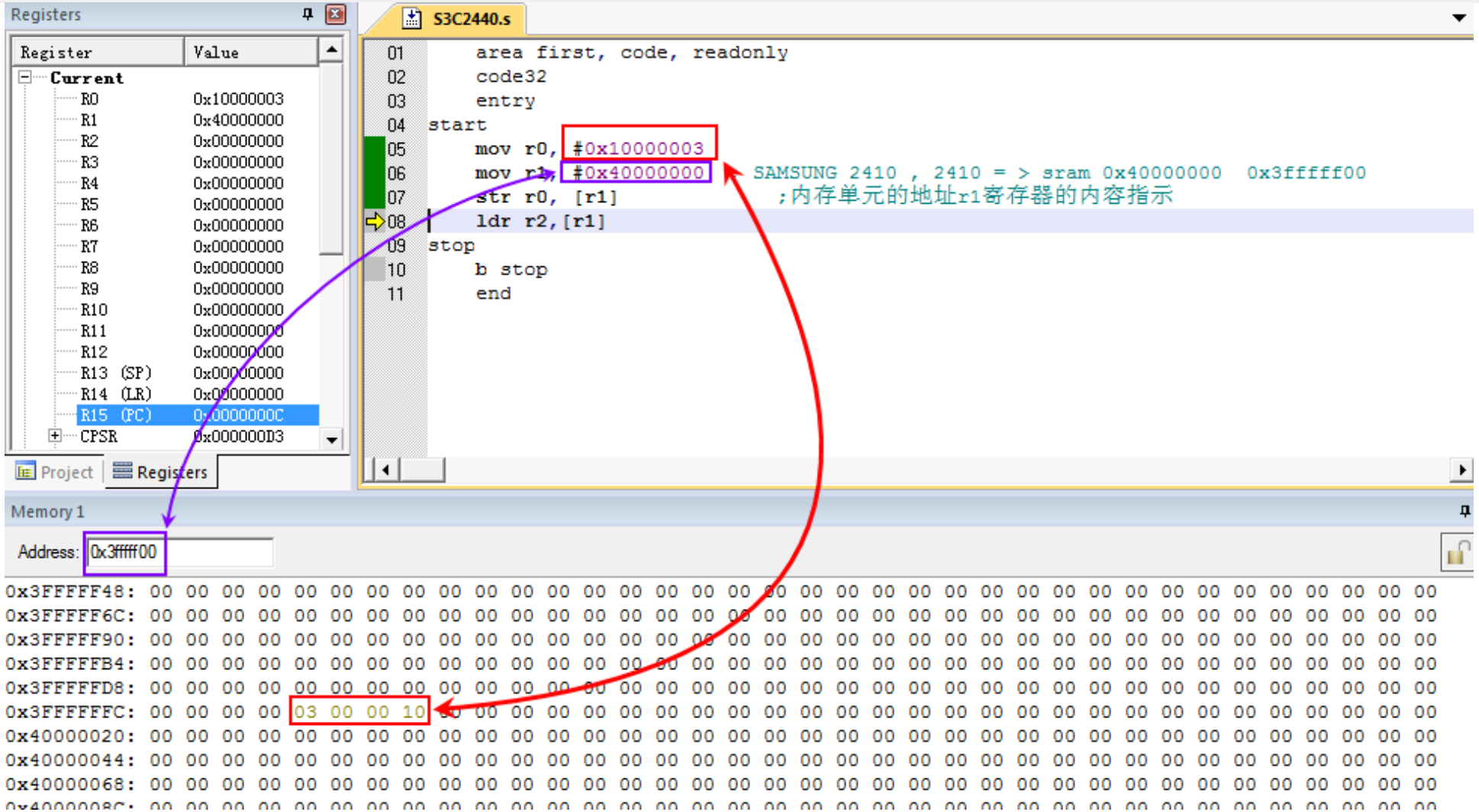


图 5-1. 复位后 S3C2440A 的存储器映射

内存信息



立即寻址

- 立即寻址也叫立即数寻址，这是一种特殊的寻址方式，操作数本身就在指令中给出，只要取出指令也就取到了操作数。
- 这个操作数被称为立即数，对应的寻址方式也就叫做立即寻址。
  - Add r0,r0,#1　　;R0=R0+1
- 在以上两条指令中，第二个源操作数即为立即数，
- 要求以“#”为前缀，对于以十六进制表示的立即数，还要求在“#”后加上“0x”或“&”。

寄存器寻址

- 利用寄存器中的数值作为操作数，这种寻址方式是各类微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。
  - Add R0 , R1,R2　　;R0=R1+R2
- 该指令的执行效果是将寄存器R1和R2的内容相加，其结果存放在寄存器R0中。

寄存器间接寻址

- 以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。
  - LDR R0,[R1]　　; R0=[R1]
- 第二条指令将以R1的值为地址的存储器中的数据传送到R0中。

基址变址寻址

- 将寄存器（该寄存器一般称作基址寄存器）的内容与指令中给出的地址偏移量相加，从而得到一个操作数的有效地址：
  - LDR R0，[R1，#4]　　;R0=[R1+4]
  - LDR R0，[R1，#4]！　　;R0=[R1+4]、R1=R1+4
  - LDR R0，[R1]，#4　　;R0=[R1]　、R1=R1+4
  - LDR R0，[R1，R2]　　;R0=[R1+R2]
-

相对寻址

- 与基址变址寻址方式相类似，相对寻址以程序计数器PC的当前值为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。
- 以下程序段完成子程序的调用和返回，跳转指令BL采用了相对寻址方式：
  - BL NEXT　　;跳转到子程序NEXT处执行
  - ……
  - NEXT
  - ……
  - MOV PC,LR　；从子程序返回

多寄存器寻址

- 采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。这寻址方式可以用一条指令完成传送最多16个通用寄存器的值。
  - LDMIA R0，{R1，R2，R3，R4}　;R1=[R0]　R2=[R0+4]　R3=[R0+8]　R4=[R0+12]
- 该指令的后缀IA表示在每次执行完加载/存储操作后，R0按字长度增加，因此，指令可将连续存储单元的值传送到R1～R4。

堆栈寻址、批量加载/存储指令

- 堆栈是一种数据结构，按先进后出（First In Last Out，FILO）的方式工作，使用一个称作堆栈指针的专用寄存器指示当前的操作位置，堆栈指针总是指向栈顶。
- 批量数据加载/存储指令可以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。常用的加载存储指令如下：
  - LDM批量数据加载指令
  - STM批量数据存储指令
- LDM（或STM）指令的格式为：
  - LDM（或STM）{条件}{类型} 基址寄存器{！}，寄存器列表{^}

类型

- LDM（或STM）指令用于从由基址寄存器所指示的一片连续存储器到寄存器列表所指示的多个寄存器之间传送数据，该指令的常见用途是将多个寄存器的内容入栈或出栈。
- 其中，{类型}为以下几种情况：
  - IA　　每次传送后地址加1；
  - IB　　每次传送前地址加1；
  - DA　　每次传送后地址减1；
  - DB　　每次传送前地址减1；
  - FD　　满递减堆栈；　　向低地址方向生长
  - ED　　空递减堆栈；
  - FA　　满递增堆栈；　　向高地址方向生长
  - EA　　空递增堆栈；
  - 【满堆栈】：堆栈指针SP指向最后压入堆栈的有效数据项
  - 【空堆栈】：堆栈指针指向下一个要放入数据的空位置

注意

- {！}为可选后缀，若选用该后缀，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，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R15，选用该后缀时表示：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。
  - STMFD R13!，{R0，R4-R12，LR}　　；将寄存器列表中的寄存器（R0，R4到R12，LR）存入堆栈，向低地址方向生长。
  - LDMFD R13!，{R0，R4-R12，PC}　　；将堆栈内容恢复到寄存器（R0，R4到R12，LR）。

ARM

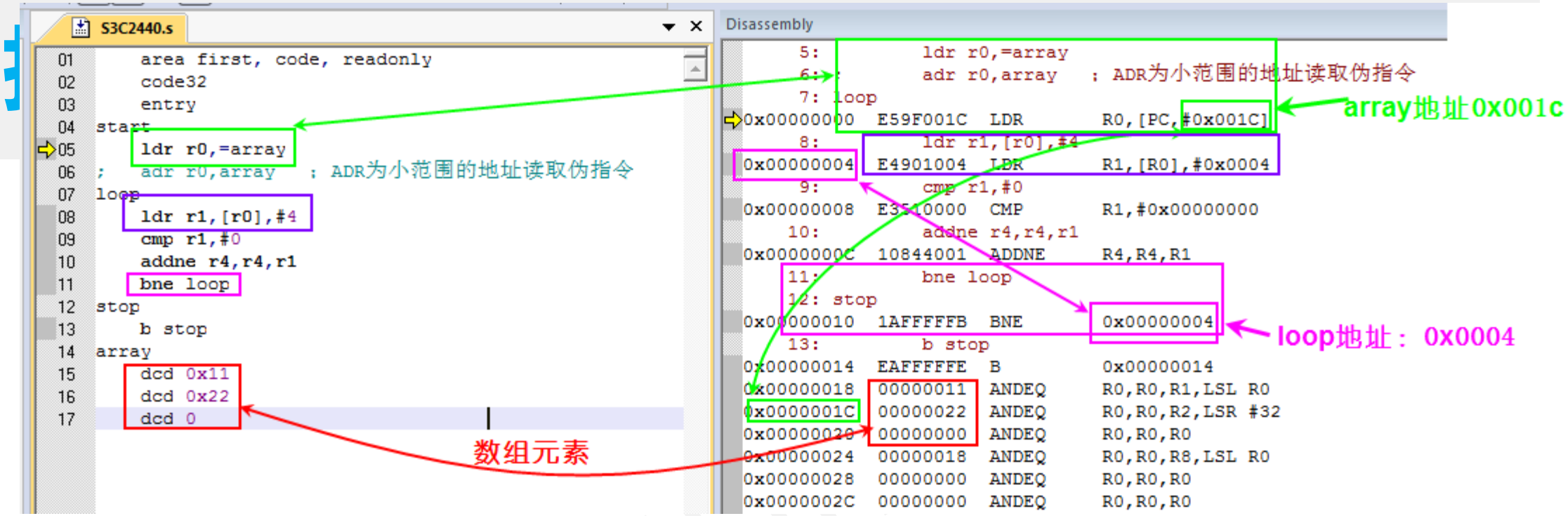
19

ARM寻址举例

例1 数组求和

- 编写一个ARM汇编程序，累加一个“数组”的所有元素，碰上0时停止。结果放入 r4。

```
1     area first, code, readonly
2     code32
3     entry
4  start
5     ldr r0,=array
6  ;  adr r0,array   ; ADR为小范围的地址读取伪指令
7  loop
8     ldr r1,[r0],#4
9     cmp r1,#0
10    addne r4,r4,r1
11    bne loop
12 stop
13    b stop
14    ; DCD 伪操作  数据缓冲池技术
15    ; dcd  机器码
16 array
17    dcd 0x11
18    dcd 0x22
19    dcd 0
```



1. ldr r0,=array，编译器会计算出array标号的地址0x0018,注意该值是偏移当前指令所在内存位置的偏移量，所以该指令最终被翻译成

ldr r0,[pc,#0x001c]

为什么是0x001c?

pc的值是指向指令的前一条指令

2. 数组元素的3个值依次存放在0x0018、0x001c、0x0020这3个地址中
3. ldr r1,[r0],#4每次取出r0指向的内存的值并写入到r1，同时将r0值自加4
4. bne loop 的loop被编译器计算为地址0x0004

例2 数据压栈退栈

- 先将栈地址设置为将要压栈的数据存入寄存器r1-r5中，然后压栈

```
1     area first, code, readonly
2     code32
3     entry
4  Start
5     ; mov r0, #0x40000000
6     ldr sp, =0x40001000   ;注意地址
7
8     mov r1, #0x11
9     mov r2, #0x22
10    mov r3, #0x33
11    mov r5, #0x55
12    ; 压栈
13    stmfd sp!, {r1-r3, r5}
14    ;stmia r0!, {r1-r3, r5} ; 加感叹号是自动修改基地址
15    mov r1, #0
16    mov r2, #0
17    mov r3, #0
18    mov r5, #0
19
20    ldmfd sp!, {r1-r3, r5}
21    ;ldmdb r0!, {r2,r1,r3, r5} ; 寄存列表书写顺序无所谓，  低地址内容对应低编号寄存器
22 stop
23    b stop
24    end
```

例3 函数嵌套调用

- 当有多级函数嵌套，函数返回值我们不可能都存储在通用寄存器中，必须利用ldm将程序跳转前的寄存器值以及函数的返回地址压栈。

```
    area first, code, readonly
    code32
    entry
start
    ldr sp, =0x40002000
    mov r1, #0x11
    mov r2, #0x22
    mov r3, #0x33
    mov r5, #0x55
    bl child_func1          ;  【先写跳转到 child_func1，再写跳转到child_func】
    add  r0, r1,r2
stop
    b stop
; 非叶子函数
child_func
    stmfd sp!, {r1-r3,r5, lr} ;;;在子函数里首先将所有寄存器值压栈保存，
                          ;;防止在子函数里篡改原本在主函数里运算需要的值，
                          ;;通常需要把r0-r12全都保存，为了安全和程序通用性应该这么做
    mov r1, #10         ;;在这里子函数想怎么做自己的事情就可以做自己的事情
    bl child_func1
    ldmfd sp!, {r1-r3,r5,lr};;;;; 放在主函数bl之后的第一句行吗？
    mov pc, lr
child_func1
    stmfd sp!, {r1-r3,r5};;;不论嵌套多少层子函数，都是先压栈，
    mov r1, #11
    ldmfd sp!, {r1-r3,r5};;对应的，在返回到自己的父函数之前将自己出栈
    mov pc, lr
    end
```

ARM

20

ARM异常

异常（Exception）

- 异常是理解CPU运转最重要的一个知识点，几乎每种处理器都支持特定异常处理，中断是异常中的一种。
- 有时候我们衡量一个操作系统的时候实时性就是看os最短响应中断时间以及单位时间内响应中断次数。

异常源

异常源	描述
Reset	上电时执行
Undef	当流水线中的某个非法指令到达执行状态时执行
SWI	当一个软中断指令被执行完的时候执行
Prefetch	当一个指令被从内存中预取时，由于某种原因而失败，如果它能到达执行状态这个异常才会产生
Data	如果一个预取指令试图存取一个非法的内存单元，这时异常产生
IRQ	通常的中断
FIQ	快速中断

1. reset复位异常

- 当CPU刚上电时或按下reset重启键(复位电路)之后进入该异常，该异常在管理模式下处理。

2.irq/fiq一般/快速中断请求

- CPU和外部设备是分别独立的硬件执行单元，CPU对全部设备进行管理和资源调度处理，
- CPU要想知道外部设备的运行状态，要么CPU定时的去查看外部设备特定寄存器，要么让外部设备在出现需要CPU干涉处理时“打断”CPU，让它来处理外部设备的请求
- 这里的“打断”操作就叫做中断请求

- 根据请求的紧急情况，中断请求分一般中断和快速中断，
  - 快速中断具有最高中断优先级和最小的中断延迟，通常用于处理高速数据传输及通道的中数据恢复处理，如DMA等，绝大部分外设使用一般中断请求。

3.预取指令中止异常

- 该异常发生在CPU流水线取指阶段，如果目标指令地址是非法地址进入该异常

- 该异常在中止异常模式下处理。

4.未定义指令异常

- 该异常发生在流水线技术里的译码阶段，如果当前指令不能被识别为有效指令，产生未定义指令异常，

- 该异常在未定义异常模式下处理。

5.软件中断指令（swi）异常

- 该异常是应用程序自己调用时产生的，用户程序申请访问硬件资源时需要调用该指令
- 例如：printf()打印函数，
  - 用户程序要想实现打印必须申请使用显示器，而用户程序又没有外设硬件的使用权，只能通过使用软件中断指令切换到内核态，
  - 通过操作系统内核代码来访问外设硬件，内核态是工作在特权模式下，操作系统在特权模式下完成将用户数据打印到显示器上。
- 这样做的目的无非是为了保护操作系统的安全和硬件资源的合理使用，该异常在管理模式（SVC）下处理。

6.数据中止访问异常

- 该异常发生在要访问数据地址不存在或者为非法地址时，该异常在中止异常模式下处理。（segment error）

ARM的异常优先级

- 依次递减
  - Reset→
  - Data abort→
  - FIQ→
  - IRQ→
  - Prefetch abort→
  - Undefined instruction/SWI。

异常与模式关系

- reset异常-----》SVC模式
- fiq -----》快中断模式，
  - 支持高速数传输及通道处理（FIQ异常响应时进入此模式）
- irq -----》中断模式
  - 用于通用中断处理
- prefetch预取指中止，数据中止异常-----》中止模式，
  - 用于支持虚拟内存和/或存储器保护
- undef未定义指令异常-----》未定义模式
  - 支持硬件协处理器的软件仿真（未定义指令异常响应时进入此模式）
- swi软件中断，复位异常-----》管理模式
  - 操作系统保护代码（系统复位和软件中断响应时进入此模式）

FIQ 比 IRQ快的原因

- fiq 比 irq 的优先级高
- FIQ 向量位于向量表的最末端，异常处理不需要跳转
- FIQ 比 IRQ 多5个私有的寄存器（r8-r12），在中断操作时，压栈出栈操作的少

ARM

21

ARM异常处理流程

ARM异常处理完整过程

- 一、 硬件阶段：
  - 4大步3小步
- 二、 异常处理
  - 保存现场、异常处理
- 三、 异常返回
  - 回复现场

一、硬件阶段- 4大步3小步

- 1. 保存执行状态：将CPSR复制到发生的异常模式下SPSR中；
- 2. 模式切换：
  - CPSR模式位强制设置为与异常类型相对应的值，
  - 处理器进入到ARM执行模式，
  - 禁止所有IRQ中断，当进入FIQ快速中断模式时禁止FIQ中断；
- 3. 保存返回地址：将下一条指令的地址（被打断程序）保存在LR(异常模式下LR_excep)中。
- 4. 跳入异常向量表：强制设置PC的值为相应异常向量地址，跳转到异常处理程序中。

1.保存执行状态

- 当前程序的执行状态是保存在CPSR里面的，
- 异常发生时，要保存当前的CPSR里的执行状态到异常模式里的SPSR里，
- 将来异常返回时，恢复回CPSR，恢复执行状态。

2.模式切换

- 1.硬件自动根据当前的异常类型，将异常码写入CPSR里的M[4:0]模式位，这样CPU就进入了对应异常模式下。
- 2.不管是在ARM状态下还是在THUMB状态下发生异常，都会自动切换到ARM状态下进行异常的处理，这是由硬件自动完成的，将CPSR[5] 设置为 0。
- 3. CPU会关闭中断IRQ（设置CPSR 寄存器I位），防止中断进入，如果当前是快速中断FIQ异常，关闭快速中断（设置CPSR寄存器F位）。

3.保存返回地址

- 当前程序被异常打断，切换到异常处理程序里，异常处理完之后，需要返回当前被打断模式继续执行，因此必须要保存当前执行指令的下一条指令的地址到LR_excep

- 由于异常模式不同以及ARM内核采用流水线技术，异常处理程序里要根据异常模式计算返回地址。

4.跳入异常向量表

- 该操作是CPU硬件自动完成的，当异常发生时，CPU强制将PC的值修改为一个固定内存地址，这个固定地址叫做异常向量。

异常向量表

- 异常向量表是一段特定内存地址空间，每种ARM异常对应一个字长空间（4Bytes），正好是一条32位指令长度，当异常发生时，CPU强制将PC的值设置为当前异常对应的固定内存地址。

地址	异常	进入模式
0x00000000	复位	管理模式
0x00000004	未定义指令	未定义模式
0x00000008	软件中断	管理模式
0x0000000C	中止（预取）	中止模式
0x00000010	中止（数据）	中止模式
0x00000014	保留	保留
0x00000018	中断 IRQ	中断模式
0x0000001C	快中断 FIQ	快中断模式

异常向量表地址

- 异常向量表0x00000000地址处是reset复位异常，之所以它为0地址，是因为CPU在上电时自动从0地址处加载指令
- 存储器映射地址0x00000000是为向量表保留的。在有些处理器中，向量表可以选择定位在高地址0xFFFF0000处【可以通过协处理器指令配置】

异常向量表

- 跳入异常向量表操作是异常发生时，硬件自动完成的，剩下的异常处理任务完全交给了程序员。

- 由上表可知，异常向量是一个固定的内存地址，我们可以通过向该地址处写一条跳转指令，让它跳向我们自己定义的异常处理程序的入口，就可以完成异常处理了。

安装异常向量表

```
1 | b reset    ;跳入reset处理程序
2 | b HandleUndef   ;跳入未定义处理程序
3 | b HandSWI       ;跳入软中断处理程序
4 | b HandPrefetchAbt     ;跳入预取指令处理程序
5 | b HandDataAbt       ;跳入数据访问中止处理程序
6 | b HandNoUsed   ;跳入未使用程序
7 | b HandleIRQ     ;跳入中断处理程序
8 | b HandleFIQ     ;跳入快速中断处理程序
```

二、异常处理

- 异常处理程序最开始，要保存被打断程序的执行现场，程序的执行现场无非就是保存当前操作寄存器里的数据，可以通过下面的栈操作指令实现保存现场：
  - STMFD SP_excep!, {R0 – R12, LR_excep}
- 在跳转到异常处理程序入口时，已经切换到对应异常模式下了，因此这里的SP是异常模式下的SP_excep了，所以被打断程序现场（寄存器数据）是保存在异常模式下的栈里
- 上述指令将R0~R12全部都保存到了异常模式栈
- 最后将修改完的被打断程序返回地址入栈保存，后面可以通过类似：MOV PC, LR的指令，返回用户程序继续执行。

三、异常处理的返回

- 异常处理完成之后，返回被打断程序继续执行，具体操作如下：
  1. 恢复被打断程序运行时寄存器数据
  2. 恢复程序运行时状态CPSR
  3. 通过进入异常时保存的返回地址，返回到被打断程序继续执行

返回地址修正

- 一条指令的执行分为：取指，译码，执行三个主要阶段，CPU由于使用流水线技术，造成当前执行指令的地址应该是PC – 8（32位机一条指令四个字节），那么执行指令的下条指令应该是PC – 4。在异常发生时，CPU自动会将将PC – 4 的值保存到LR里，但是该值是否正确还要看异常类型才能决定。
-

一般/快速中断请求返回

- 当FIQ/IRQ异常中断产生时，程序计数器pc的值已经更新，它指向当前指令后面第3条指令（对于ARM指令，它指向当前指令地址加12字节的位置；
- 当FIQ/IRQ异常中断产生时，处理器将值（pc-4）保存到FIQ/IRQ异常模式下的寄存器lr_irq/lr_irq中，它指向当前指令之后的第2条指令，因此正确返回地址可以通过下面指令算出：
  - SUBS　PC，LR_irq，#4 ；一般中断
  - SUBS　PC，LR_fiq，#4 ；快速中断

软中断指令（SWI）异常返回

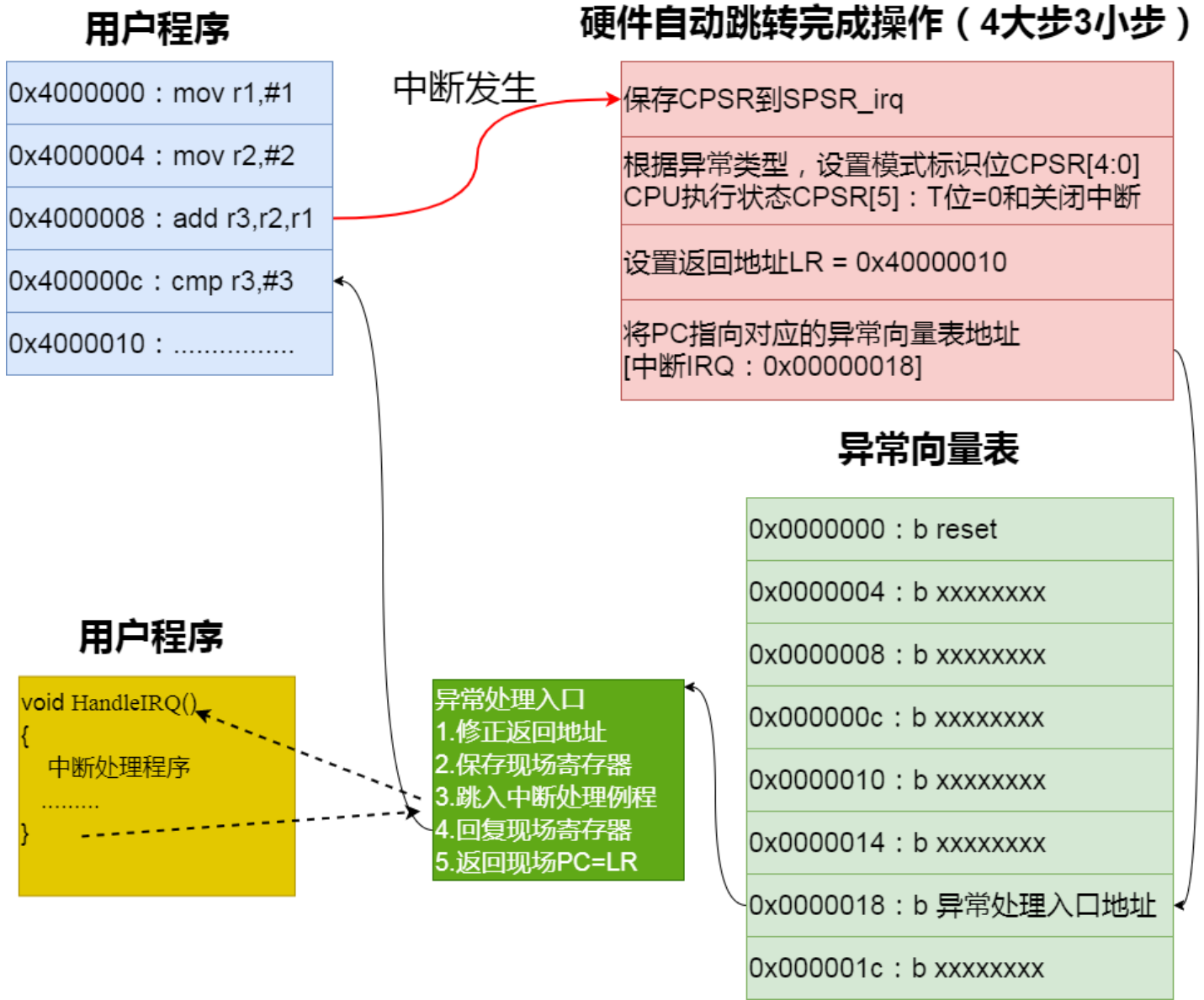
- 当SWI指令执行时，pc的值还未更新，它指向当前指令后面第2条指令（对于ARM指令，它指向当前指令地址加8字节的位置；对于Thumb指令，它指向当前指令地址加4字节的位置），当未定义指令异常中断发生时，处理器将值（pc-4）保存到lr_svc中，此时（pc-4）指向当前指令的下一条指令，所以从SWI异常中断处理返回的实现方法与从未定义指令异常中断处理返回一样：
  - MOV PC，LR_svc

ARM®

22

中断异常处理举例

中断处理流程举例



一、 硬件阶段：
4大步3小步
二、 异常处理
保存现场、异常处理
三、 异常返回
回复现场

取视频中所有资料

ARM®

23

软中断SWI

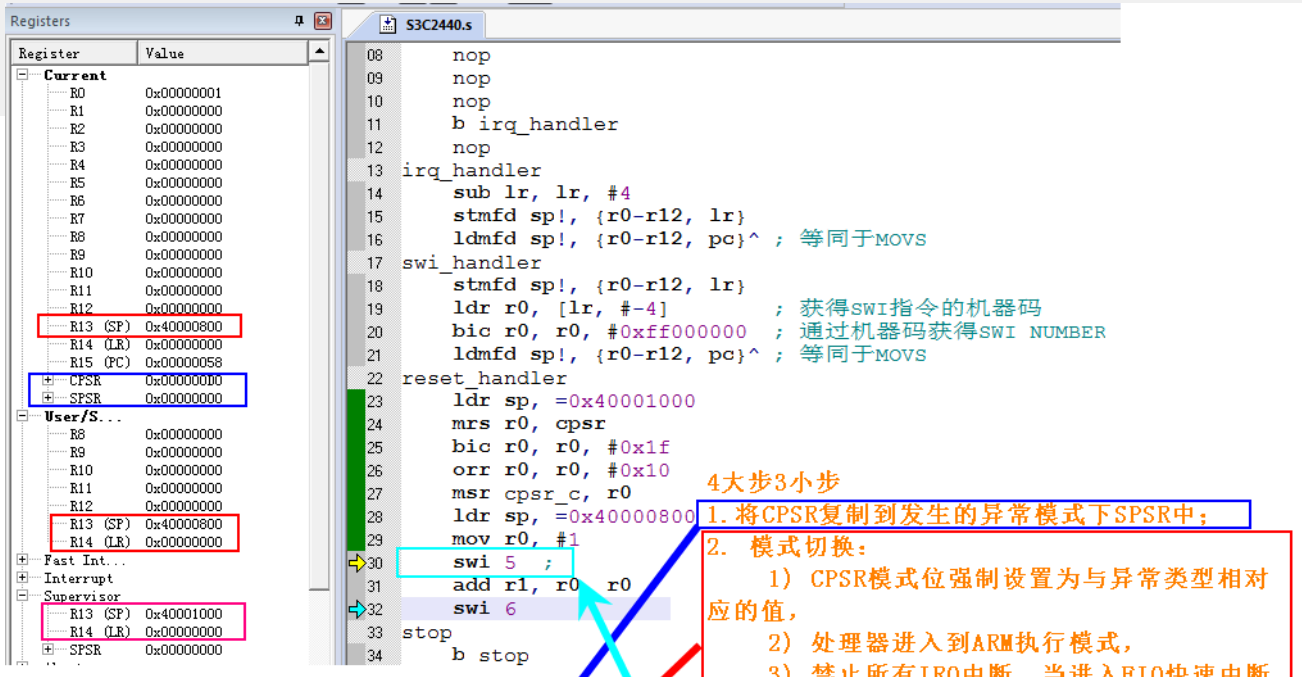
SWI指令

- SWI指令的格式为：
  - SWI{条件} 24位的立即数
- SWI指令用于产生软件中断，以便用户程序能调用操作系统的系统例程。
- 操作系统在SWI的异常处理程序中提供相应的系统服务
- 指令中24位的立即数指定用户程序调用系统例程的类型
  - 相关参数通过通用寄存器传递，当指令中24位的立即数被忽略时，用户程序调用系统例程的类型由通用寄存器R0的内容决定，同时，参数通过其他通用寄存器传递。

举例：分析这个例子有什么问题?

```
    area first, code, readonly
    code32
    entry
; 定义的异常向量表
vector
    b reset_handler  ; 跳转到 reset_handler
    nop
    b swi_handler    ; SWI 指令异常跳转的地址
    nop
    nop
    nop
    nop
    nop
swi_handler
    ; swi handler code
    ; 异常处理首先要压栈保存处理器现场
    mrs r0, cpsr
    bic r0, r0, #0x1f
    orr r0, r0, #0x10
    msr cpsr_c, r0

    ;ldr r0, [lr, #-4]  ; 获得SWI指令的机器码，lr前面那个指令是swi指令，下标在该指令中
    ;bic r0, r0, #0xff000000  ; 通过机器码获得SWI NUMBER
    movs pc, lr     ; lr > pc 且 spsr -> cpsr返回 SVC -> USER
reset_handler
    ; 初始化 SVC 模式堆栈
    ldr sp, =0x40001000
    ; 修改当前的模式从SVC模式改变为USER模式
    mrs r0, cpsr
    bic r0, r0, #0x1f
    orr r0, r0, #0x10
    msr cpsr_c, r0
    ; 初始化 USER 模式堆栈
    ldr sp, =0x40000800
    mov r0, #1
    ; USER SWI
    swi 5  ; open  APP USER 这条语句由用户程序自己出发异常
           ; 观察并记录对比指令执行前后的 PC LR CPSR SPSR SP的变化
           ;并思考异常产生后处理器硬件自动发生了那些变化
    add r1, r0, r0
stop
    b stop
    end
```

如何同时跳转并切换模式?



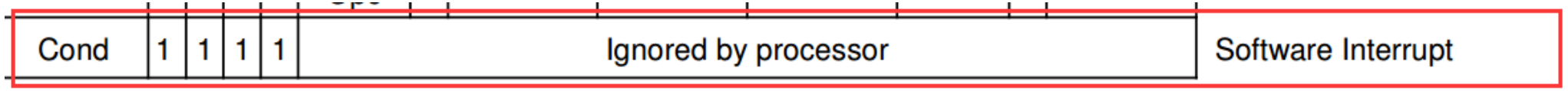
频中所有资料

如何同时跳转并切换模式?

- 从swi异常返回时，我们需要执行两个动作：
  - 将spsr拷贝会cpsr，
  - pc = lr 跳转回原来的位置
- 这两个动作都必须要执行，但是如果分步执行的话，spsr拷贝回去后，当前模式就变回了usr模式，那么对应的lr的值就变成了lr_usr,此时的值0x0【之前没有执行过bl指令】，那怎么跳转会去呢?
- 我们可以用以下命令
  - movs pc, lr
- 此命令同时执行两个动作：
  - pc = lr
  - cpsr = spsr 返回 SVC -> USER
- 从而实现了同时跳转并切换模式。
  如果入口已经使用ldm压栈可以用一下指令回复：
  - LDMFD SP_excp!, {r0-r12, pc}^

如何获取软中断号?

- 要获取swi指令的中断号，我们只能从swi的机器码中得到对应的值，



- 而要想得到swi这条指令的内容，就要先找到这条指令的地址，
  而lr的值是swi这条指令的下一条指令的地址，所以我们可以通过以下代码得到软中断号。
  - ldr r0, [lr, #-4] ；获得SWI指令的机器码，lr前面那个指令是swi指令，下标在该指令中
  - bic r0, r0, #0xff000000 ；通过机器码获得SWI NUMBER

BKPT指令

- BKPT指令的格式为：
  - BKPT 16位的立即数
- BKPT指令产生软件断点中断，可用于程序的调试。

ARM

24

汇编伪指令
GNU编写风格

MDK和GNU

- GNU风格

```
.global _start
_start:         @汇编入口
    ldr sp,=0x41000000
.end            @汇编程序结束
```

- MDK风格

```
    AREA Example,CODE,READONLY     ;声明代码段Example
    ENTRY ;程序入口
Start
    MOV R0,#0
OVER
    END
```

MDK概念参考第六节

什么是伪指令?

- 伪指令告诉汇编程序进行什么操作
  - 仅仅在汇编时有效，
    - 比如变量的定义，内存空间的分配

- 而指令是机器运行对应机器的一个动作，只有汇编通过，机器才能运行。

初学者应该学习哪种风格?

- 从事Linux驱动开发应该学习GNU风格的汇编代码
  - 因为做Linux驱动开发必须掌握的linux内核、uboot，而这两个软件就是GNU风格的

- GNU计划，是由**理查德·斯托曼**在1983年9月27日公开发起的。它的目标是创建一套完全自由的操作系统。
- 早期GNU组织为了编译Linux源码而开发了一款C语言编译器，后期逐渐支持了各种平台的各种编程语言。

标号symbol（或label）

- 标号只能由a～z，A～Z，0～9，“.”，_等（由点、字母、数字、下划线等组成，除局部标号外，不能以数字开头）字符组成。
- Symbol的分类
  - 基于PC的标号
    - 基于PC的标号是位于目标指令前的标号或者程序中数据定义伪操作前的标号。这种标号在汇编时将被处理成PC值加上（或减去）一个数字常量，常用于表示跳转指令“b”等的目标地址，或者代码段中所嵌入的少量数据。
  - 基于寄存器的标号
    - 基于寄存器的标号常用MAP和FIELD来定义，也可以用EQU来定义。这种标号在汇编时将被处理成寄存器的值加上（或减去）一个数字常量，常用于访问数据段中的数据。
  - 绝对地址：32位常量
    - 绝对地址是一个32位数据。它可以寻址的范围为[0，2^{32}-1]即可以直接寻址整个内存空间

局部标号

- 局部标号主要在局部范围内使用，而且局部标号可以重复出现。
- 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
- 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

局部标号举例

```
1:
subs r0, r0, #1    @每次循环使r0=r0-1
bne 1F                    @跳转到1标号去执行
```

- 如果F和B都没有指定，编译器先向前搜索，再向后搜索；
- 如果A和T都没有指定，编译器搜索所有从当前层次到宏的最高层次，比当前层次低的层次不再搜索；
- 如果指定了routname，编译器向前搜索最近的ROUT伪操作，若routname与该ROUT伪操作定义的名称不匹配，编译器报告错误，汇编失败。

常数

- 1. 十进制数以非0数字开头,如:123和9876；
- 2. 二进制数以0b开头,其中字母也可以为大写；
- 3. 八进制数以0开始,如:0456,0123；
- 4. 十六进制数以0x开头,如:0xabcd,0X123f；
- 5. 字符串常量需要用引号括起来,中间也可以使用转义字符,如: “You are welcome!\n”；
- 6. 当前地址以“.”表示,在GNU汇编程序中可以使用这个符号代表当前指令的地址；
- 7. 表达式：在汇编程序中的表达式可以使用常数或者数值, 其他的符号如:+、 -、 *、 /、%、<、<<、>、>>、|、&、^、!、==、>=、<=、&&、|| 跟C语言中的用法相似。

特殊字符和语法

- 1. 代码行中的注释符号: '@'
  2. 整行注释符号: '#'
  3. 语句分离符号: ';'
  4. 立即数前缀: '#' 或 '$'

标号语句格式

- 任何Linux汇编行都是如下结构：
  - [<label>:][<instruction or directive or pseudo-instruction>}@comment
    - <label>：标号, GNU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。
    - instruction：指令
    - directive：伪操作
    - pseudo-instruction：伪指令
    - comment：语句的注释

其他注意事项

- 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)。

分段

- .section伪操作

  .section <section_name> {,"<flags>"}

- section_name
  - 代码段名称为.text
  - 初始化的数据段为.data
  - 未初始化的数据段为.bss
  - rodata段：字符串和#define定义的常量
  - heap堆、stack栈、常量段
- flags

a	**可分配**
w	可写段
x	执行段

例1：定义一个代码段

```
.section .text, “x”  @用.section伪操作定义了代码段
.global add                  @声明符号add给编译器用户链接程序
add:
    ADD r0, r0, r1           @ add 输入参数
    MOV pc, lr                @ 从子函数返回
@ end of program
```

例2：定义一个段

```
.section .mysection    @自定义数据段，段名为 “.mysection”
.align  2                          @4字节对齐
strtemp:
.ascii  "Temp string \n\0" @将"Temp string \n\0"这个字符串存储在以标号strtemp:为起始地址的一段内存空间里
```

例3：定义入口点

- 汇编程序的缺省入口是_start标号，用户也可以在连接脚本文件中用ENTRY标志指明其它入口点。

```
.section .data                    @初始化数据段
    < initialized data here>
.section .bss                     @初始化bss段
    < uninitialized data here>
.section .text
.globl  _start
_start:
    <instruction code goes here>
```

ARM®

25

GNU其他伪操作、宏定义

数据定义伪操作

标号	含义
.byte	单字节定义 0x12,'a',23
.short	定义2字节数据 0x1234,65535
.long /.word	定义4字节数据 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"，注意：.ascii 伪操作定义的字符串需要每行添加结尾字符 '\0'，其他不需要
.space/.skip	用于分配一块连续的存储区域并初始化为指定的值，如果后面的填充值省略不写则在后面填充为0；

数据定义举例

.word

val:

.word 0x11223344

mov r1,#val @将值0x11223344设置到寄存器r1中

.space

label:

```
__v7_setup_stack:
    .space  4 * 11                  @ 11 registers
```

. Ascii

```
cpu_arm1020_name:
    .ascii  "ARM1020"
```

@ ascii伪操作定义的字符串需要自行添加结尾字符'\0'

- 注意：
  - 变量的定义放在stop后，.end前，标号是地址的助记符，标号不占存储空间，位置在.end前就可以，相对随意。

杂项伪操作标识符

标号	含义
.global/	用来声明一个全局的符号
.arm	定义一下代码使用ARM指令集编译
.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
.align	.align{alignment}{,fill}{,max} 通过用零或指定的数据进行填充来使当前位置与指定边界对齐 **.align 4 --- 16字节对齐 2的4次方** **.align (4) --- 4字节对齐**
.org	.org offset{,expr} 指定从当前地址加上offset开始存放代码，并且从当前地址到当前地址加上offset之间的内存单元，用零或指定的数据进行填充
.extern	用于声明一个外部符号，用于兼容性其他汇编
.code 32	同.arm
.code 16	同.thumb
.weak	用于声明一个弱符号，如果这个符号没有定义，编译就忽略，而不会报错
.end	文件结束
.include	.include “filename” 包含指定的头文件, 可以把一个汇编常量定义放在头文件中
.equ	格式：.equ symbol, expression把某一个符号(symbol)定义成某一个值(expression).该指令并不分配空间，类似于c语言的#define
.set	给一个全局变量或局部变量赋值，和.equ的功能一样

.global伪操作

- .global/ .globl：

用来定义一个全局的符号，

- 格式如下：

  .global symbol

  或者 .globl symbol

管理员: cmd - vim entry-armv.S (Admin)

```
        .section .vectors, "ax", %progbits
__vectors_start:
        W(b)    vector_rst
        W(b)    vector_rst
        W(b)    vector_und
        W(ldr)  pc, __vectors_start + 0x1000
        W(b)    vector_pabt
        W(b)    vector_dabt
        W(b)    vector_addrexcptn
        W(b)    vector_irq
        W(b)    vector_fiq

        .data

        .globl  cr_alignment
        .globl  cr_no_alignment
cr_alignment:
        .space  4
cr_no_alignment:
        .space  4

#ifdef CONFIG_MULTI_IRQ_HANDLER
        .globl  handle_arch_irq
handle_arch_irq:
        .space  4
```

entry-armv.S [unix] (11:40 31/03/2014) 1146,2-9 99%

vim.exe 查找

.align .end .include .incbin伪操作

- .align:用来指定数据的对齐方式,格式如下:
  .align [absexpr1, absexpr2]

- .include:可以将指定的文件在使用.include 的地方展开,一般是头文件,
  - 例如:
  - .include "uart.h"

- .incbin伪操作可以将原封不动的一个二进制文件编译到当前文件中

.rept伪操作

- .rept重复执行接下来的指令，以.rept开始，以.endr结束。
- .rept语法结构如下：

```
.rept cnt   @cnt是重复次数
    数据定义
.endr     @结束重复定义
```

.if伪操作

- 根据一个表达式的值来决定是否要编译下面的代码

```
.if  logical-expressing
     ......
.else
     ......
.endif
```

- 举例：

```
.if  val2==1
    mov r1,#val2
.endif
```

举例

- 例1：.set

```
.set start, 0x40
mov r1, #start     ；r1里面是0x40
```

- 例2：.equ

```
.equ   start,  0x40
mov r1, #start     ；r1里面是0x40
```

- 例3：.equ

```
.equ   PI, 31415
```

- 等价于c语言中的：

```
#define PI 31415
```

macro宏定义

- 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。macro宏定义类似c语言里的宏函数。
- 语法格式：

```
.macro   {$label} 名字{$parameter{,$parameter}...}
      ......code
.endm
```

- $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。
- 注意：先定义后使用。

例1：没有参数的宏实现子函数返回

```
.macro MOV_PC_LR
    MOV PC,LR
.endm
```

- 调用方式如下：
- MOV_PC_LR

例2：带参数宏实现子函数返回

```
.macro MOV_PC_LR ,param
    MOV r1,\param
    MOV PC,LR
.endm
```

- 调用方法如下：
  - MOV_PC_LR #12

举例:Linux内核宏定义举例

- linux-3.14-fs4412\arch\arm\kernel\entry-armv.S

```
        .macro  pabt_helper
        @ PABORT handler takes pt_regs in r2, fault address in r4 and psr in r5
#ifdef MULTI_PABORT
        ldr     ip, .LCprocfns
        mov     lr, pc
        ldr     pc, [ip, #PROCESSOR_PABT_FUNC]
#else
        bl      CPU_PABORT_HANDLER
#endif
        .endm
```

```
        pabt_helper
```

```
        .macro  usr_ret, reg
#ifdef CONFIG_ARM_THUMB
        bx      \reg
#else
        mov     pc, \reg
#endif
        .endm
```

```
        usr_ret lr
```

ARM®

26

伪指令

伪指令

- 即它不是标准的ARM指令，伪指令在编译时会转化为对应的其他ARM指令（可能对应多条ARM指令）。
  - 1. ADR小范围地址读取伪指令；
  - 2. ADRL中等范围的地址读取伪指令；
  - 3. LDR大范围地址读取伪指令；
  - 4. NOP空操作伪指令；

1）ldr伪指令和ldr指令区分

- 下面是ldr伪指令：
  ldr r1,=val @ r1 = val
  - 将val标号地址赋给r1，类似于以下C代码

  int p =10;
  r1 = &p;

- 下面是ldr指令
  ldr r2,val @ r1 = *val
- 是arm指令,将标号val地址里的内容给r2，类似于下面C代码
  - int *p =10;
  - r2 = *p;

2）如何利用ldr伪指令实现长跳转?

- ldr pc，=32位地址

3）编码中解决非立即数的问题

- ldr r0,=0x999 ； 0x999 不是立即数，

/home/peng/test/basic

```
1 .text
2 .global _start
3 _start:       @汇编入口
4
5     ldr r0,=0x999
6     b main
7 .end          @汇编程序结束
```

```
5 Disassembly of section .text:
6
7 40008000 <_start>:
8 40008000:   e51f0000    ldr r0, [pc, #-0]   ; 40008008 <_start+0x8>
9 40008004:   ea000000    b   4000800c <main>
10 40008008:  00000999    muleq   r0, r9, r9
11
12 4000800c <main>:
13 4000800c:   eafffffe    b   4000800c <main>
```

ARM

27

GNU风格代码编译

1. 不含lds文件的编译

```
peng@ubuntu:~/test/nolds$ tree ./
./
├── main.c
├── Makefile
└── start.s

0 directories, 3 files
```

```
1  TARGET=start
2  TARGETC=main
3  all:
4      arm-none-linux-gnueabi-gcc -O0 -g -c -o $(TARGETC).o  $(TARGETC).c
5      arm-none-linux-gnueabi-gcc -O0 -g -c -o $(TARGET).o $(TARGET).s
6      #arm-none-linux-gnueabi-gcc -O0 -g -S -o $(TARGETC).s  $(TARGETC).c
7      arm-none-linux-gnueabi-ld   $(TARGETC).o    $(TARGET).o -Ttext 0x40008000 -o $(TARGET).elf
8      arm-none-linux-gnueabi-objcopy   -O binary -S  $(TARGET).elf  $(TARGET).bin
9  clean:
10     rm -rf *.o *.elf *.dis *.bin
```

O0 – O3 数字越大，优化程度越高

1. 定义环境变量TARGET=start，**start**为汇编文件的文件名
2. 定义环境变量TARGETC=main，**main**为c语言文件
3. 目标：**all**，4~8行是该指令的指令语句
4. 将main.c编译生成**main.o,$(TARGETC)**会被替换成main
5. 将**start.s**编译生成**start.o,$(TARGET)**会被替换成start
6. 4-5也可以用该行1条指令实现
7. 通过**ld**命令将main.o、start.o链接生成**start.elf,-Ttext 0x40008000**表示设置代码段起始地址为**0x40008000**
8. 通过objcopy将**start.elf**转换成**start.bin**文件，**-O binary** (或–out-target=binary) 输出为原始的二进制文件,-**S** (或 --strip-all)输出文件中不要重定位信息和符号信息，缩小了文件尺寸，
9. **clean**目标
10. clean目标的执行语句，删除编译产生的临时文件

2. 依赖lds文件编译

- 实际的工程文件，尤其Linux内核有几万个文件，段的分布及其复杂，所以这就需要我们借助lds文件来定义内存的分布。

```
root@ubuntu:/home/peng# tree arm/
arm/
├── main.c
├── Makefile
├── map.lds
└── start.s

0 directories, 4 files
```

lds文件

- 1. OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
  - 指定输出object档案预设的binary 文件格式。可以使用objdump -i列出支持的binary 文件格式；
- 2. OUTPUT_ARCH(arm)
  - 指定输出的平台为arm，可以透过objdump -i查询支持平台；
- 3. ENTRY(_start) ：
  - 将符号_start的值设置成入口地址；
- 7. . = 0x40008000:
  - 把定位器符号置为0x40008000(若不指定, 则该符号的初始值为0)；
- 9. .text : { .start.o(.text) *(.text) } :
  - 前者表示将start.o放到text段的第一个位置，后者表示将所有(*符号代表任意输入文件)输入文件的.text section合并成一个.text section；
- 15. .rodata : { *(.data) } :
  - 将所有输入文件的.rodata section合并成一个.rodata section；
- 18. .data : { *(.data) } :
  - 将所有输入文件的.data section合并成一个.data section；
- 21. .bss : { *(.bss) } :
  - 将所有输入文件的.bss section合并成一个.bss section；该段通常存放全局未初始化变量
- 20. . = ALIGN(4);表示下面的段4字节对齐

```
1  OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
2  /*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
3  OUTPUT_ARCH(arm)
4  ENTRY(_start)
5  SECTIONS
6  {
7      . = 0x40008000;
8      . = ALIGN(4);
9      .text      :
10     {
11         gcd.o(.text)
12         *(.text)
13     }
14     . = ALIGN(4);
15     .rodata :
16     { *(.rodata) }
17     . = ALIGN(4);
18     .data :
19     { *(.data) }
20     . = ALIGN(4);
21     .bss :
22      { *(.bss) }
23 }
```

包含lds文件的Makefile

```
CROSS_COMPILE = arm-none-linux-gnueabi-
NAME =start
CFLAGS=-mfloat-abi=softfp -mfpu=vfpv3 -mabi=apcs-gnu -fno-builtin  -fno-builtin-function -g -O0 -c
LD  = $(CROSS_COMPILE)ld
CC  = $(CROSS_COMPILE)gcc
OBJCOPY = $(CROSS_COMPILE)objcopy
OBJDUMP = $(CROSS_COMPILE)objdump
OBJS=start.o  main.o
#=============================================#
all:  $(OBJS)
    $(LD)  $(OBJS) -T map.lds -o $(NAME).elf
    $(OBJCOPY)  -O binary  $(NAME).elf $(NAME).bin
    $(OBJDUMP) -D $(NAME).elf > $(NAME).dis
%.o: %.S
    $(CC) $(CFLAGS) -c -o  $@ $<
%.o: %.s
    $(CC) $(CFLAGS) -c -o  $@ $<
%.o: %.c
    $(CC) $(CFLAGS) -c -o  $@ $<
clean:
    rm -rf $(OBJS) *.elf *.bin *.dis *.o
```

编译

- 最终生成start.bin,该文件可以烧录到开发板测试

```
peng@ubuntu:~/test/lds$ make
arm-none-linux-gnueabi-gcc -mfloat-abi=softfp -mfpu=vfpv3 -mabi=apcs-gnu -fno-builtin  -fno-builtin-function -g -O0 -c
rt.o start.s
arm-none-linux-gnueabi-gcc -mfloat-abi=softfp -mfpu=vfpv3 -mabi=apcs-gnu -fno-builtin  -fno-builtin-function -g -O0 -c
n.o main.c
arm-none-linux-gnueabi-ld  start.o  main.o -T map.lds -o start.elf
arm-none-linux-gnueabi-objcopy  -O binary  start.elf start.bin
arm-none-linux-gnueabi-objdump -D start.elf > start.dis
peng@ubuntu:~/test/lds$
peng@ubuntu:~/test/lds$
peng@ubuntu:~/test/lds$ ls
main.c  main.o  Makefile  map.lds  start.bin  start.dis  start.elf  start.o  start.s
```

补充

- 1. 其中交叉编译工具链**arm-none-linux-gnueabi-** 要根据自己实际的平台来选择，本例是基于三星的**exynos-4412**工具链实现的。
- 2. 地址0x40008000也不是随便选择的，

3.1 Overview

This section describes the base address of region.

Base Address	Limit Address	Size	Description
0x0000_0000	0x0001_0000	64 KB	iROM
0x0200_0000	0x0201_0000	64 KB	iROM (mirror of 0x0 to 0x10000)
0x0202_0000	0x0206_0000	256 KB	iRAM
0x0300_0000	0x0302_0000	128 KB	Data memory or general purpose of Samsung Reconfigurable Processor SRP.
0x0302_0000	0x0303_0000	64 KB	I-cache or general purpose of SRP.
0x0303_0000	0x0303_9000	36 KB	Configuration memory (write only) of SRP
0x0381_0000	0x0383_0000	–	AudioSS's SFR region
0x0400_0000	0x0500_0000	16 MB	Bank0 of Static Read Only Memory Controller (SMC) (16-bit only)
0x0500_0000	0x0600_0000	16 MB	Bank1 of SMC
0x0600_0000	0x0700_0000	16 MB	Bank2 of SMC
0x0700_0000	0x0800_0000	16 MB	Bank3 of SMC
0x0800_0000	0x0C00_0000	64 MB	Reserved
0x0C00_0000	0x0CD0_0000	–	Reserved
0x0CE0_0000	0x0D00_0000	–	SFR region of Nand Flash Controller (NFCON)
0x1000_0000	0x1400_0000	–	SFR region
0x4000_0000	0xA000_0000	1.5 GB	Memory of Dynamic Memory Controller (DMC)-0
0xA000_0000	0x0000_0000	1.5 GB	Memory of DMC-1

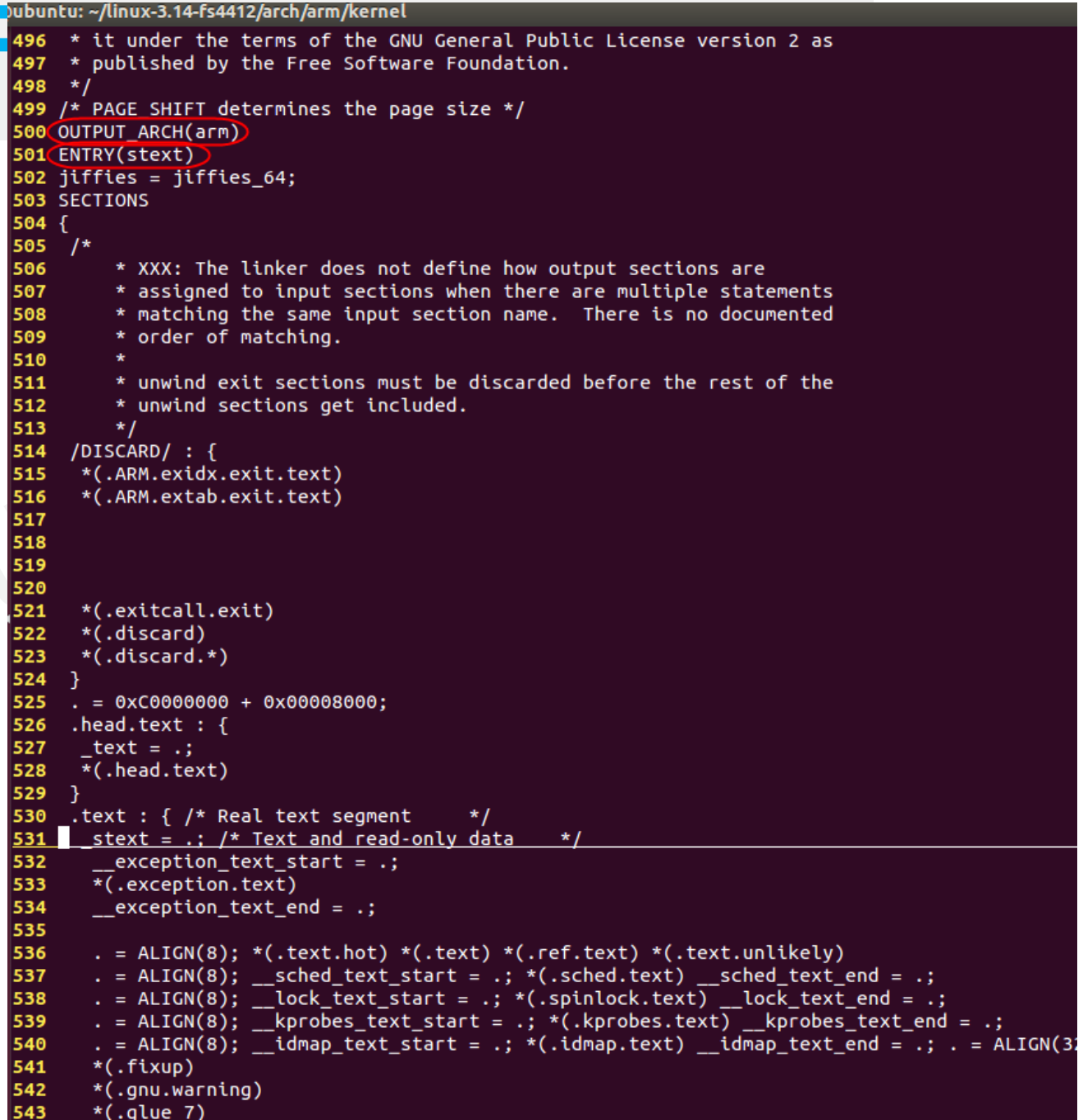
频中所有资料

U-boot的lds文件

- H:\u-boot-origen\arch\arm\cpu\u-boot.lds

Linux内核的lds文件

- D:\linux-3.14-fs4412\arch\arm\kernel\vmlinux.lds



ARM®

28 内联汇编

1、内联汇编

- 内联汇编即在C中直接使用汇编语句进行编程，使程序可以在C程序中实现C语言不能完成的一些工作
- 例如，在下面几种情况中必须使用内联汇编或嵌入型汇编。
  - 1. 程序中使用饱和算术运算(Saturating Arithmetic)
  - 2. 程序需要对协处理器进行操作
  - 3. 在C程序中完成对程序状态寄存器的操作

内联汇编格式

```
__asm__ __volatile__("asm code"
    :output
    :input
    :changed registers);
```

- asm或__asm__开头，小括号+分号，括号内容写汇编指令。

asm code

- 1. **asm code**主要填写汇编代码，
- 2. 指令+\n\t 用双引号引上。

```
"mov r0, r0\n\t"
"mov r1,r1\n\t"
"mov r2,r2"
```

output(asm->C)

- **output(asm->C)用于定义输出的参数，通常只能是变量：**

```
1  :"constraint" (variable)
2  "constraint"用于定义variable的存放位置：
3   r 表示使用任何可用的寄存器
4   m 表示使用变量的内存地址
5   + 可读可写
6   = 只写
7   & 表示该输出操作数不能使用输入部分使用过的寄存器，只能用"+&"或"=&"的方式使用
```

input(C->asm)

- 用于定义输入的参数，可以使变量也可以是立即数：

```
:"constraint" (variable/immediate)
"constraint"用于定义variable的存放位置：
 r 表示使用任何可用的寄存器(立即数和变量都可以)
 m 表示使用变量的内存地址
 i 表示使用立即数
```

注意

- 1. 使用__asm__和__volatile__表示编译器将不检查后面的内容，而是直接交给汇编器。
- 2. 如果希望编译器为你优化，__volatile__可以不加
- 3. 没有asm code也不能省略""
- 4. 没有前面的和中间的部分，不可以相应的省略:
- 5. 没有changed 部分，必须相应的省略:
- 6. 最后的;不能省略，对于C语言来说这是一条语句
- 7. 汇编代码必须**放在一个字符串**内，且字符串中间**不能直接按回车换行**，可以写成多个字符串，注意中间不能有任何符号，这样就会将两个字符串合并为一个
- 8. 指令之间必须要换行，还可以使用\t使指令在汇编中保持整齐

实例1：无参数，无返回值

- 这种情况，output和input可以省略：

```
asm
( //汇编指令
    "mrs r0,cpsr     \n\t"
    "bic r0,r0,#0x80 \n\t"
    "msr cpsr,r0     \n\t"
);
```

实例2：有参数，有返回值

- 让内联汇编做加法运算，求a+b，结果存在c中

```
int a =100,  b =200,  c =0;
asm
(
    "add %0,%1,%2\n\t"
    : "=r"(c)
    : "r"(a),"r"(b)
    : "memory"
);
```

例3：有参数 ，有返回值

- 求a+b，结果存在sum中，把a-b的存在d中

```
asm volatile
(
    "add %[op1],%[op2],%[op3]\n\t"
    "sub %[op4],%[op2],%[op3]\n\t"
    :[op1]"=r"(sum),[op4]"=r"(d)
    :[op2]"r"(a),[op3]"r"(b)
    :"memory"
);
```

ARM®

29

汇合汇编

ATPCS和AAPCS

- 为了使单独编译的C语言程序和汇编程序之间能够相互调用,必须为子程序之间的调用规定一定的规则

- **ATPCS**
  - 即ARM-Thumb过程调用标准(ARM-THUMB procedure call standard)

- **AAPCS**
  - 即ARM结构过程调用规范（**ARM Architecture Procedure Call Standard**）。

规则

- 下面3方面的内容
- 1. 各寄存器的使用规则及其相应的名称。
- 2. 数据栈的使用规则。
- 3. 参数传递的规则。

1. 寄存器的使用规则：

- 寄存器的使用必须满足下面的规则：
- 1）子程序间通过寄存器**R0—R3**来传递参数，这时，寄存器R0～R3可以记作A1-A4。被调用的子程序在返回前无需恢复寄存器R0～R3的内容。
- 2）在子程序中，使用寄存器**R4～R11**来保存局部变量．这时，寄存器 R4 ～ R11可以记作V1 ～ V8。
  - **如果在子程序中使用到了寄存器V1～V8中的某些寄存器，子程序进入时必须保存这些寄存器的值，在返回前必须恢复这些寄存器的值；对于子程序中没有用到的寄存器则不必进行这些操作。在Thumb程序中，通常只能使用寄存器R4～R7来保存局部变量。**
- 3）**寄存器R12**用作过程调用时的临时寄存器（用于保存SP，在函数返回时使用该寄存器出**栈**），记作ip。
- 4）**寄存器R13**用作数据栈指针，记作**sp**。在子程序中寄存器R13不能用作其他用途。寄存器sp在进入子程序时的值和退出子程序时的值必须相等。
- 5）**寄存器R14**称为连接寄存器，记作**lr**。它用于保存子程序的返回地址。如果在子程序中保存了返回地址，寄存器R14则可以用作其他用途。
- 6）**寄存器R15**是程序计数器，记作**pc**。它不能用作其他用途。
-

2、堆栈使用规则

- AAPCS规定堆栈为**FD类型**，即**满递减堆栈**。
- 并且堆栈的操作是**8字节对齐**。

3、参数的传递规则

- 在参数传递时,将所有参数看做是存放在连续的内存单元中的字数据。
- 然后,依次将各名字数据传送到寄存器R0,R1,R2,R3;
- 如果参数多于4个,将剩余的字数据传送到数据栈中,入栈的顺序与参数顺序相反,即最后一个字数据先入栈.

```
void func(a,b,c,d,e)
   a -- r0
   b -- r1
   c -- r2
   d -- r3
   e -- 栈
```

子程序结果返回规则

- 1. 结果为一个32位的整数时,可以通过寄存器R0返回.
- 2. 结果为一个64位整数时,可以通过R0和R1返回，依此类推.
- 3. 对于位数更多的结果,需要通过调用内存来传递.

```
int func1(int m, int n)
    m  -- r0
    n  -- r1
    返回值给 r0
```

例1：c调用汇编文件中函数带返回值

```c
extern int add(int a,int b);

printf("%d \n",add(2,3));
```

```
;.asm
add:
    add r2,r0,r1
    mov r0,r2
    MOV pc, lr
```

1.a->r0,b->r1
2.返回值通过r0返回计算结果给c代码

例2，用汇编实现一个strcopy函数

```c
//.c
#include <stdio.h>
extern void strcopy(char* des, const char* src);
int main(){
    const char* srcstr = "yikoulinux";
    char desstr[]="test";
    strcopy(desstr, srcstr);
    return 0;
}
```

```asm
;.asm
.global strcopy
strcopy:        ;R0指向目的字符串 ;R1指向源字符串
 LDRB R2, [R1], #1    ;加载字字符并更新源字符串指针地址
 STRB R2, [R0], #1    ;存储字符并更新目的字符串指针地址
 CMP R2, #0    ;判断是否为字符串结尾
 BNE strcopy    ;如果不是，程序跳转到strcopy继续循环
 MOV pc, lr    ;程序返回
```

例3. 汇编调用C

```
;.asm ;
.text .global _start
_start:
    STR lr, [sp, #-4]! ;保存返回地址lr
    ADD R1, R0, R0 ;计算2*i(第2个参数)
    ADD R2, R1, R0 ;计算3*i(第3个参数)
    ADD R3, R1, R2 ;计算5*i
    STR R3, [SP, #-4]! ;第5个参数通过堆栈传递
    ADD R3, R1, R1 ;计算4*i(第4个参数)
    BL fcn ;调用C程序
    ADD sp, sp, #4 ;从堆栈中删除第五个参数
 .end
```

```
//.c
int fcn(int a, int b , int c, int d, int e)
{
    return a+b+c+d+e;
}
```

假设程序进入fcn时，R0中的值为i；

```
fcn(i, 2*i, 3*i, 4*i, 5*i);
```

ARM®

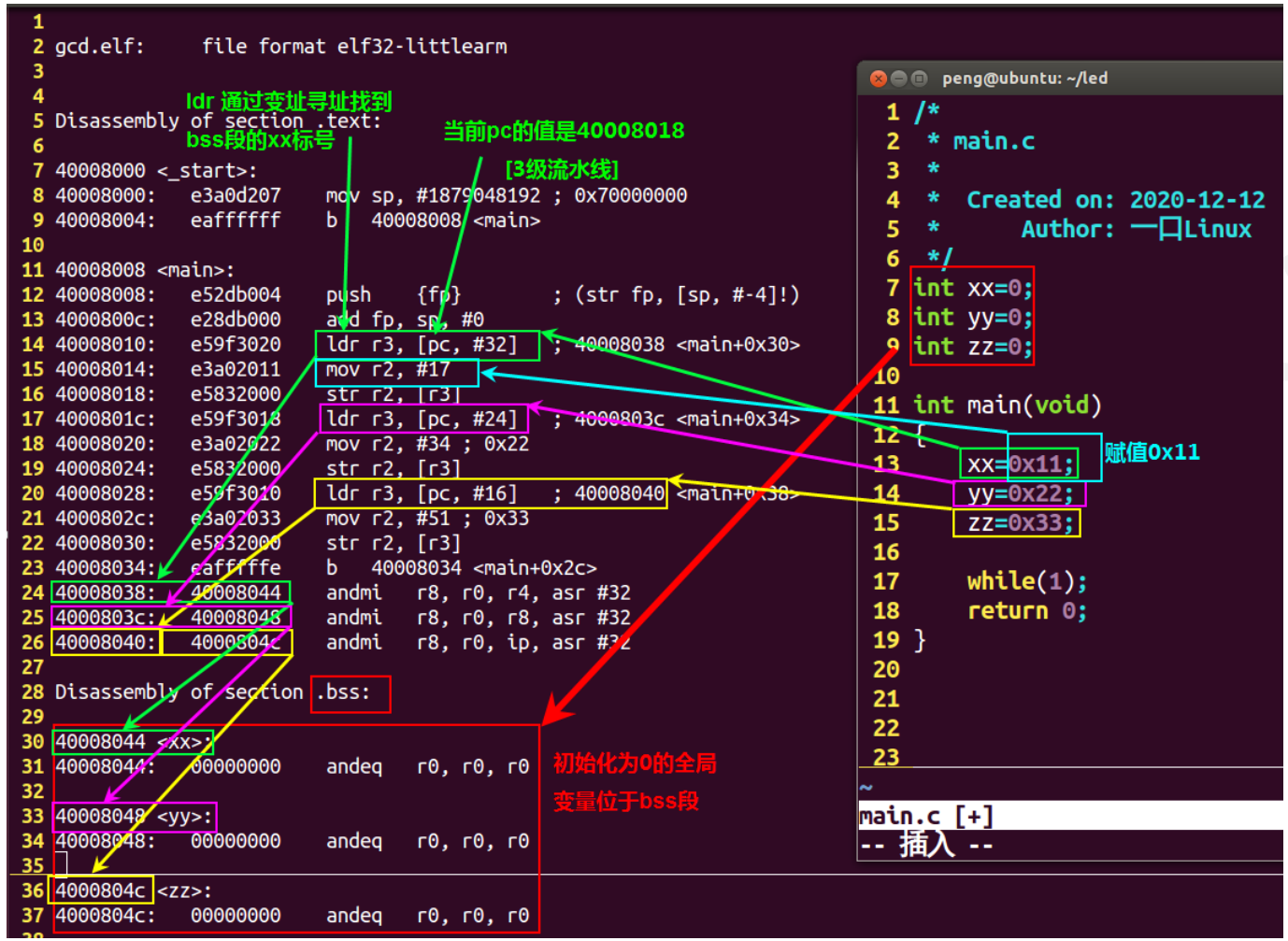
30

为什么建议使用结构体？

- 很多新手，在编写代码的时候，特别喜欢定义全局变量，而不是把这些变量封装到一个结构体中

- **Cortex所有的寻址模式都是间接寻址**——换句话说**一定依赖一个寄存器作为基地址。**

实例1：没有结构体




```
gcd.elf:     file format elf32-littlearm

Disassembly of section .text:

40008000 <_start>:
40008000:   e3a0d207    mov sp, #1879048192 ; 0x70000000
40008004:   eafffffd    b   40008008 <main>

40008008 <main>:
40008008:   e52db004    push    {fp}        ; (str fp, [sp, #-4]!)
4000800c:   e28db000    add fp, sp, #0
40008010:   e59f3020    ldr r3, [pc, #32]   ; 40008038 <main+0x30>
40008014:   e3a02011    mov r2, #17
40008018:   e5832000    str r2, [r3]
4000801c:   e59f3018    ldr r3, [pc, #24]   ; 4000803c <main+0x34>
40008020:   e3a02022    mov r2, #34 ; 0x22
40008024:   e5832000    str r2, [r3]
40008028:   e59f3010    ldr r3, [pc, #16]   ; 40008040 <main+0x38>
4000802c:   e3a02033    mov r2, #51 ; 0x33
40008030:   e5832000    str r2, [r3]
40008034:   eafffffe    b   40008034 <main+0x2c>
40008038:   40008044    andmi   r8, r0, r4, asr #32
4000803c:   40008048    andmi   r8, r0, r8, asr #32
40008040:   4000804c    andmi   r8, r0, ip, asr #32

Disassembly of section .bss:

40008044 <xx>:
40008044:   00000000    andeq   r0, r0, r0

40008048 <yy>:
40008048:   00000000    andeq   r0, r0, r0

4000804c <zz>:
4000804c:   00000000    andeq   r0, r0, r0
```

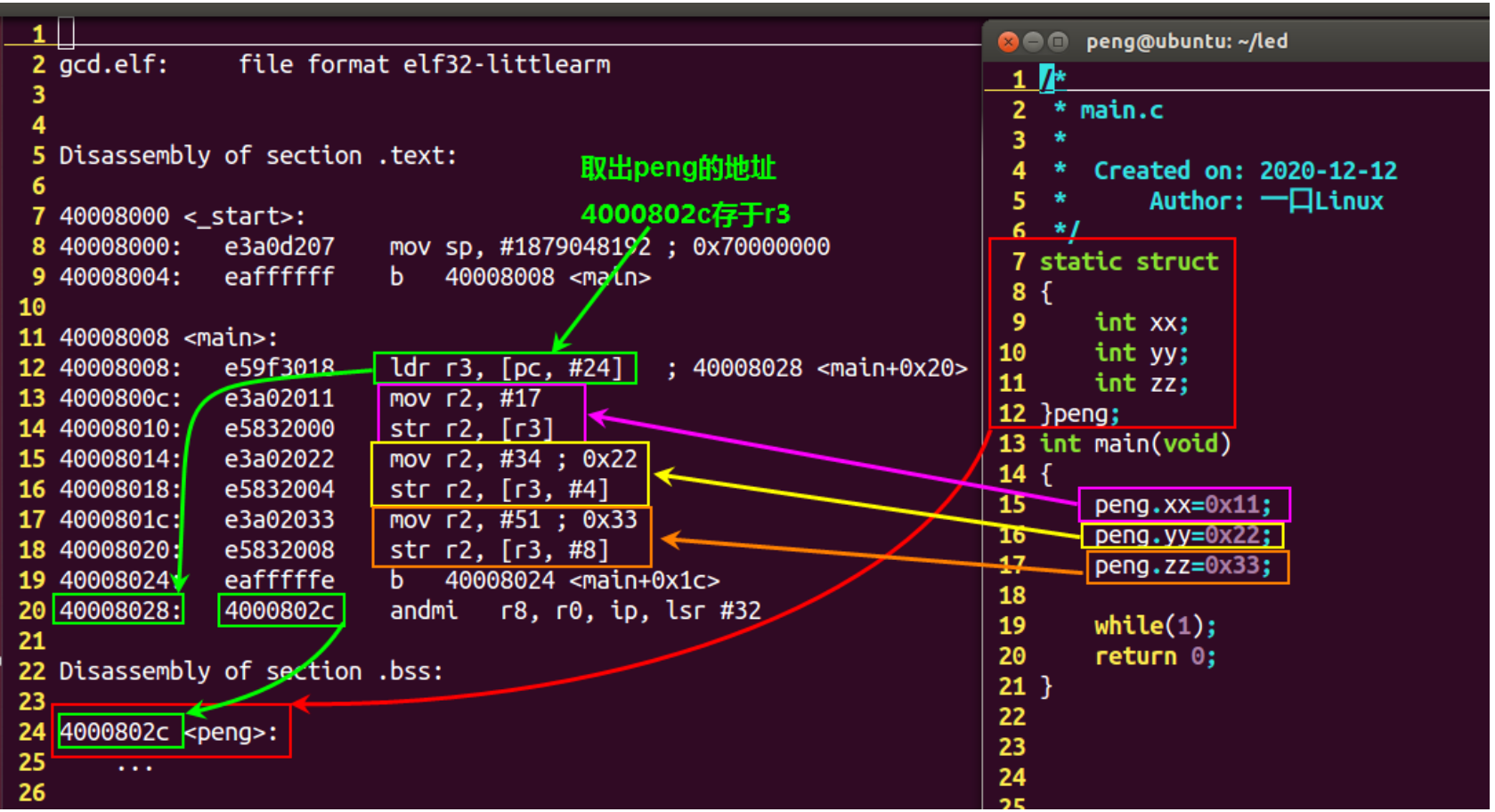
peng@ubuntu: ~/led

```
/*
 * main.c
 *
 *  Created on: 2020-12-12
 *      Author: 一口Linux
 */
int xx=0;
int yy=0;
int zz=0;

int main(void)
{
    xx=0x11;
    yy=0x22;
    zz=0x33;

    while(1);
    return 0;
}
```

main.c [+]
-- 插入 --

14. 通过当前pc值40008018偏移32个字节，找到xx变量的链接地址40008038，然后取出其内容40008044存放在r3中，该值就是xx在bss段的地址
15. 通过将立即数0x11即#17赋值给r2
16. 将r2的内让那个写入到r3对应的指向的内存，即xx标号对应的内存中

实例2：使用结构体



实例3：提高优化等级

```
1  TARGET=gcd
2  TARGETC=main
3  all:
4       arm-none-linux-gnueabi-gcc -Os   -lto -g -c -o $(TARGETC).o  $(TARGETC).c
5       arm-none-linux-gnueabi-gcc -Os  -lto -g -c -o $(TARGET).o $(TARGET).s
6       arm-none-linux-gnueabi-gcc -Os  -lto -g -S -o $(TARGETC).s  $(TARGETC).c
7       arm-none-linux-gnueabi-ld   $(TARGETC).o    $(TARGET).o -Tmap.lds  -o  $(TARGET).elf
8       arm-none-linux-gnueabi-objcopy -O binary -S $(TARGET).elf $(TARGET).bin
9       arm-none-linux-gnueabi-objdump -D $(TARGET).elf > $(TARGET).dis
10 clean:
11      rm -rf *.o *.elf *.dis *.bin
```

14. 把peng的地址40008024装载到r3中
15. r0写入立即数0x11
16. r1写入立即数0x22
17. r0写入立即数0x33
18. 通过stm指令将r0、r1、r2的值顺序写入到40008024内存中

```
 7 40008000 <_start>:
 8 40008000:   e3a0d207    mov sp, #1879048192 ; 0x70000000
 9 40008004:   eaffffff    b   40008008 <main>
10
11 Disassembly of section .text.startup:
12
13 40008008 <main>:
14 40008008:   e59f3010    ldr r3, [pc, #16]   ; 40008020 <main+0x18>
15 4000800c:   e3a00011    mov r0, #17
16 40008010:   e3a01022    mov r1, #34 ; 0x22
17 40008014:   e3a02033    mov r2, #51 ; 0x33
18 40008018:   e8830007    stm r3, {r0, r1, r2}
19 4000801c:   eafffffe    b   4000801c <main+0x14>
20 40008020:   40008024    andmi   r8, r0, r4, lsr #32
21
22 Disassembly of section .bss:
23
24 40008024 <peng>:
25     ...
```

- 《arm第一期录制完毕》
- 欢迎大家加我好友：yikoupeng
-

想入门和进阶ARM，
请加关注一口君的公众号：一口Linux