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

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

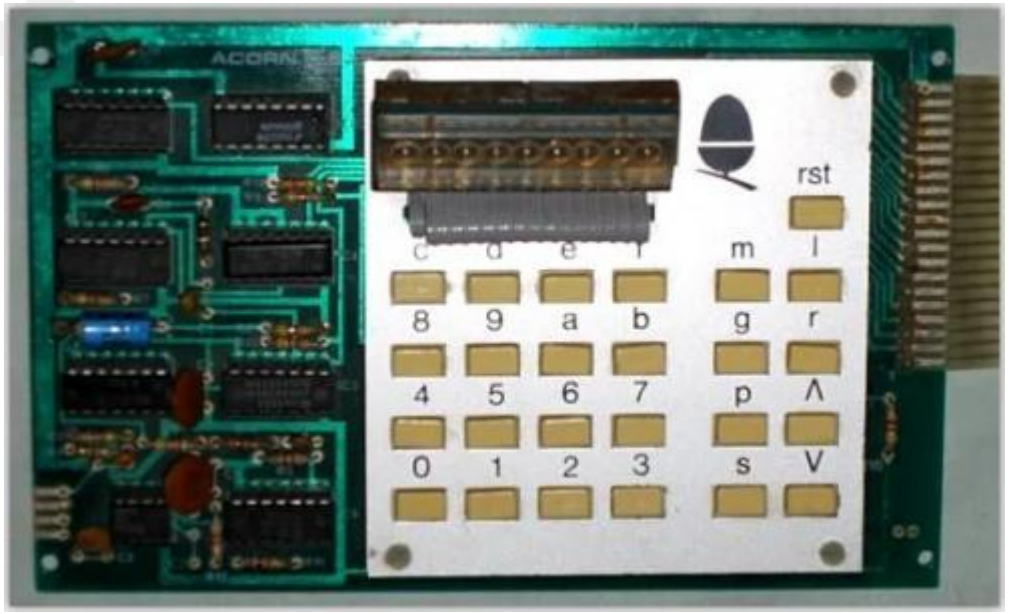
ARM®

02

什么是ARM？

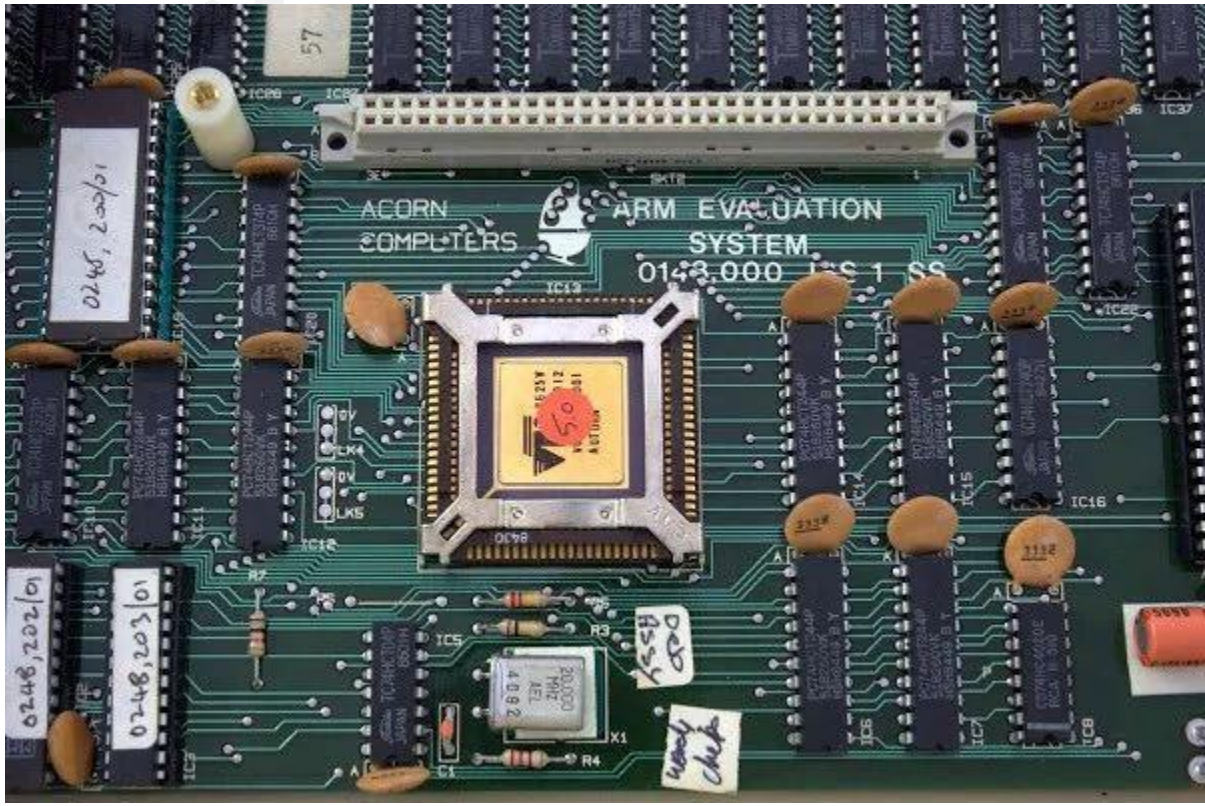
缘起

- 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。

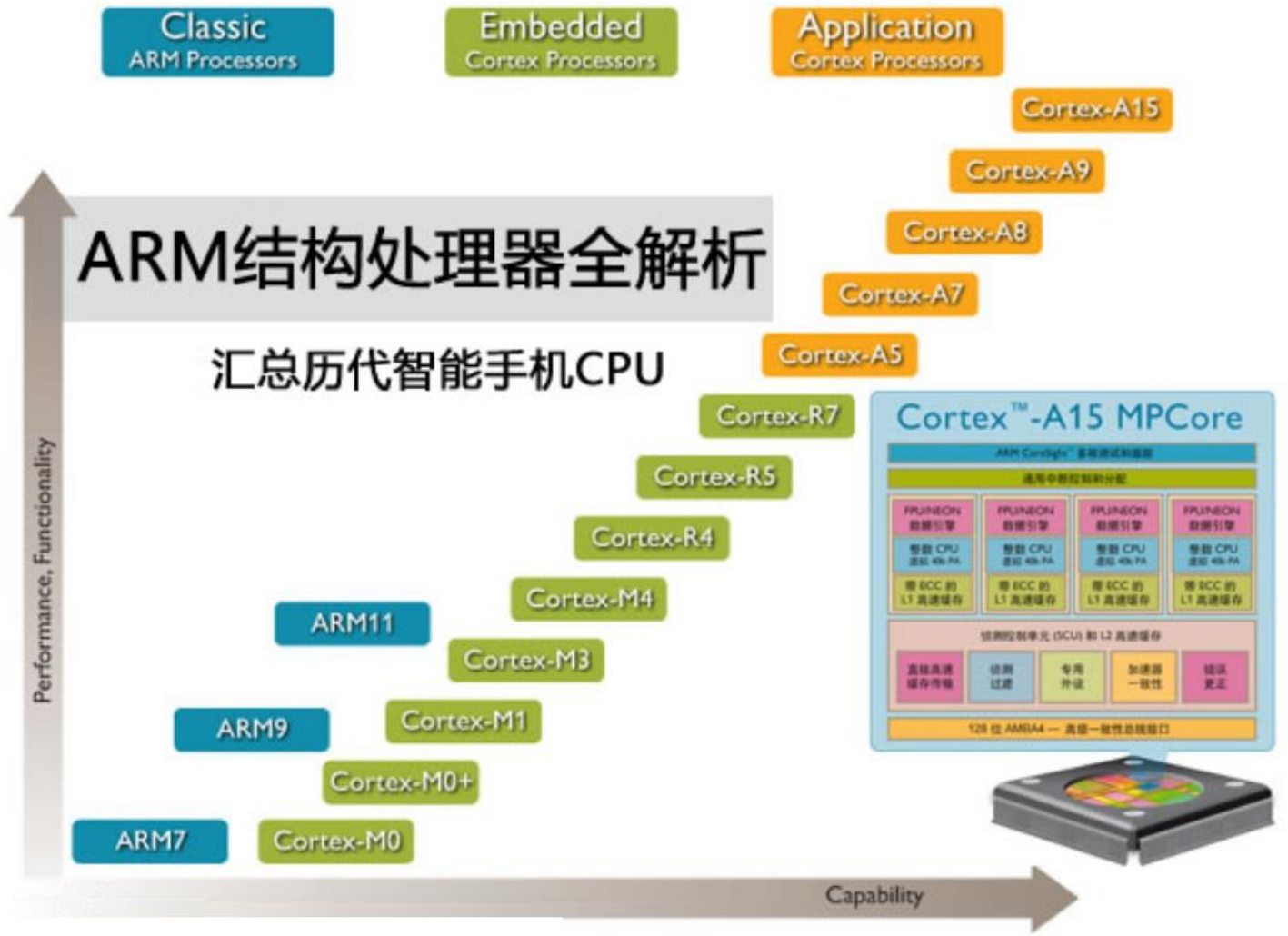
- 目前中国和欧盟要花时间对这笔交易进行评估

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）。

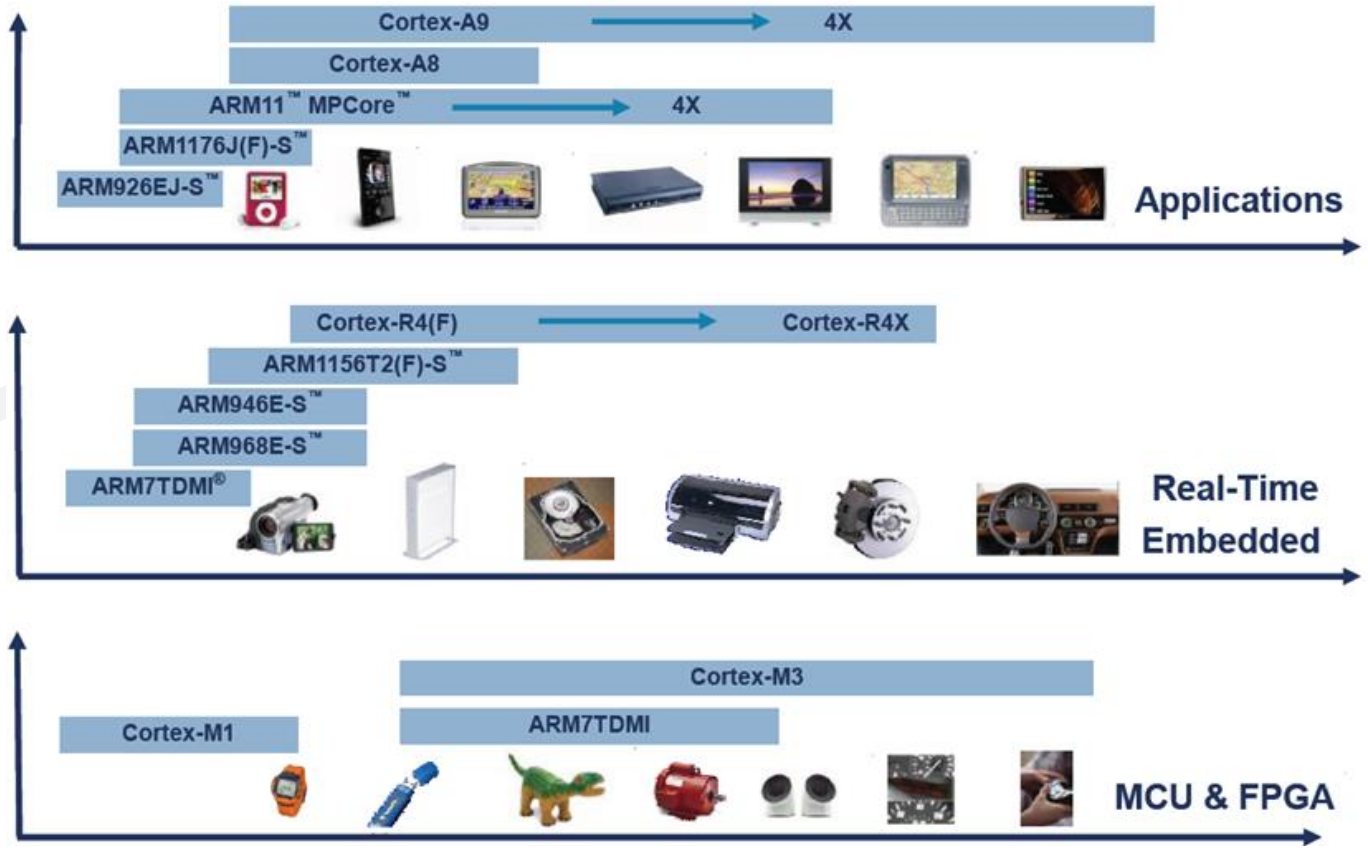
命名的改变

- 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

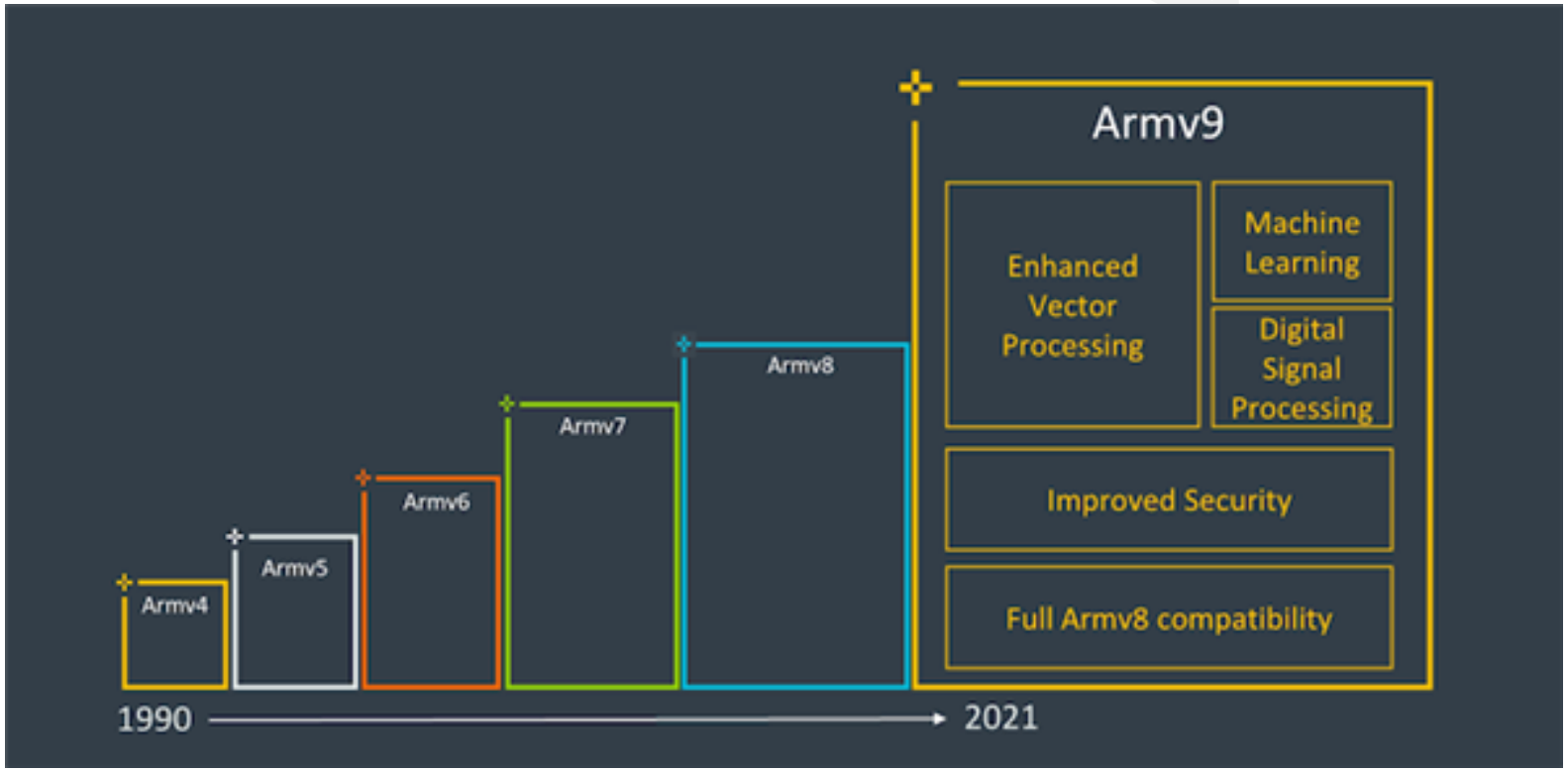
- Cortex-A系列（A：Application）
- 针对日益增长的消费娱乐和无线产品设计，用于具有高计算要求、运行丰富操作系统及提供交互媒体和图形体验的应用领域，如智能手机、平板电脑、汽车娱乐系统、数字电视，智能本、电子阅读器、家用网络、家用网关和其他各种产品。。

- 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

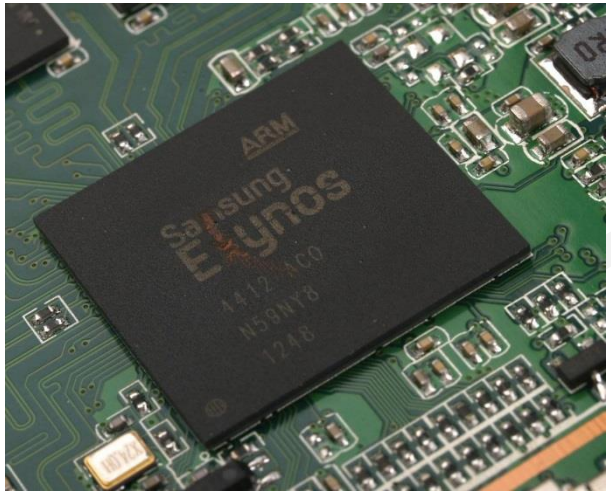
ARM

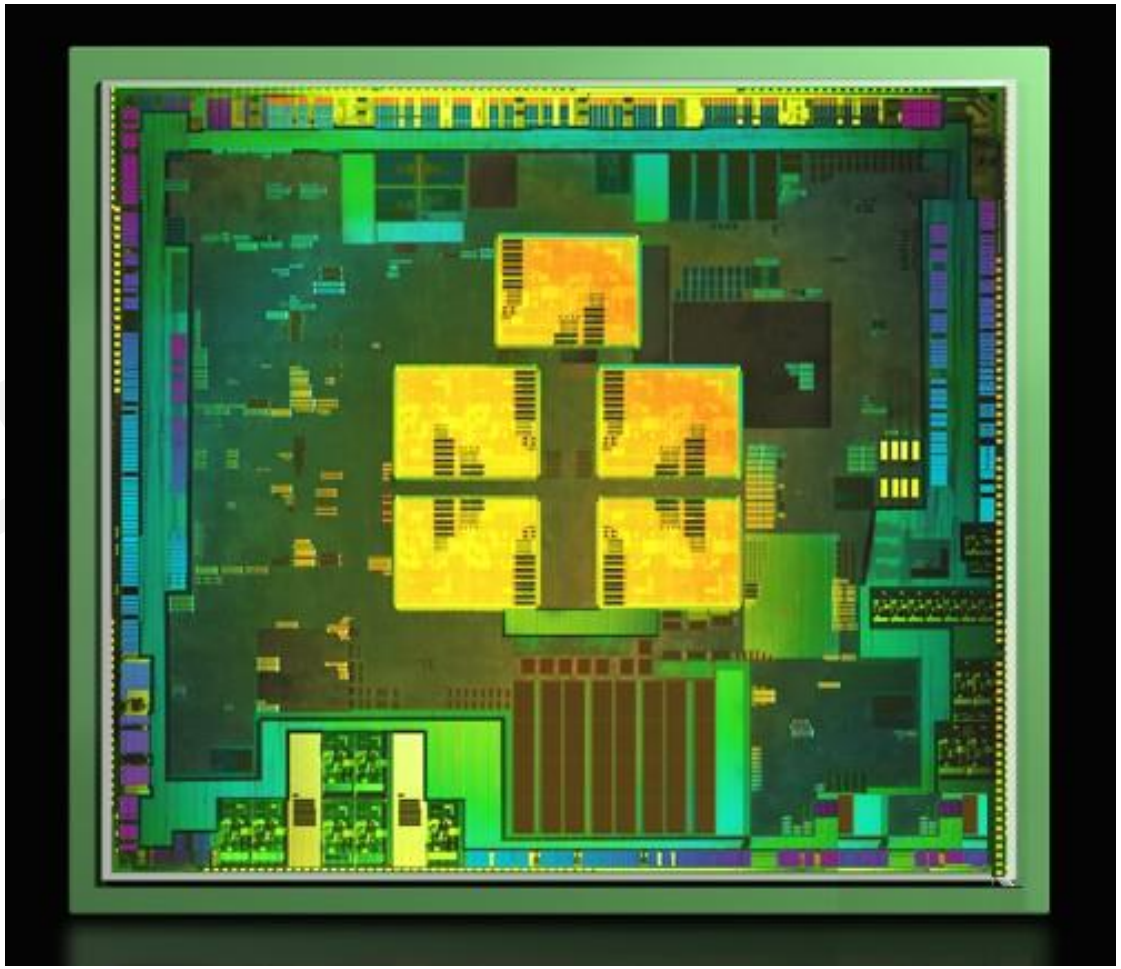
03

什么是SOC、ARM授权？

SOC

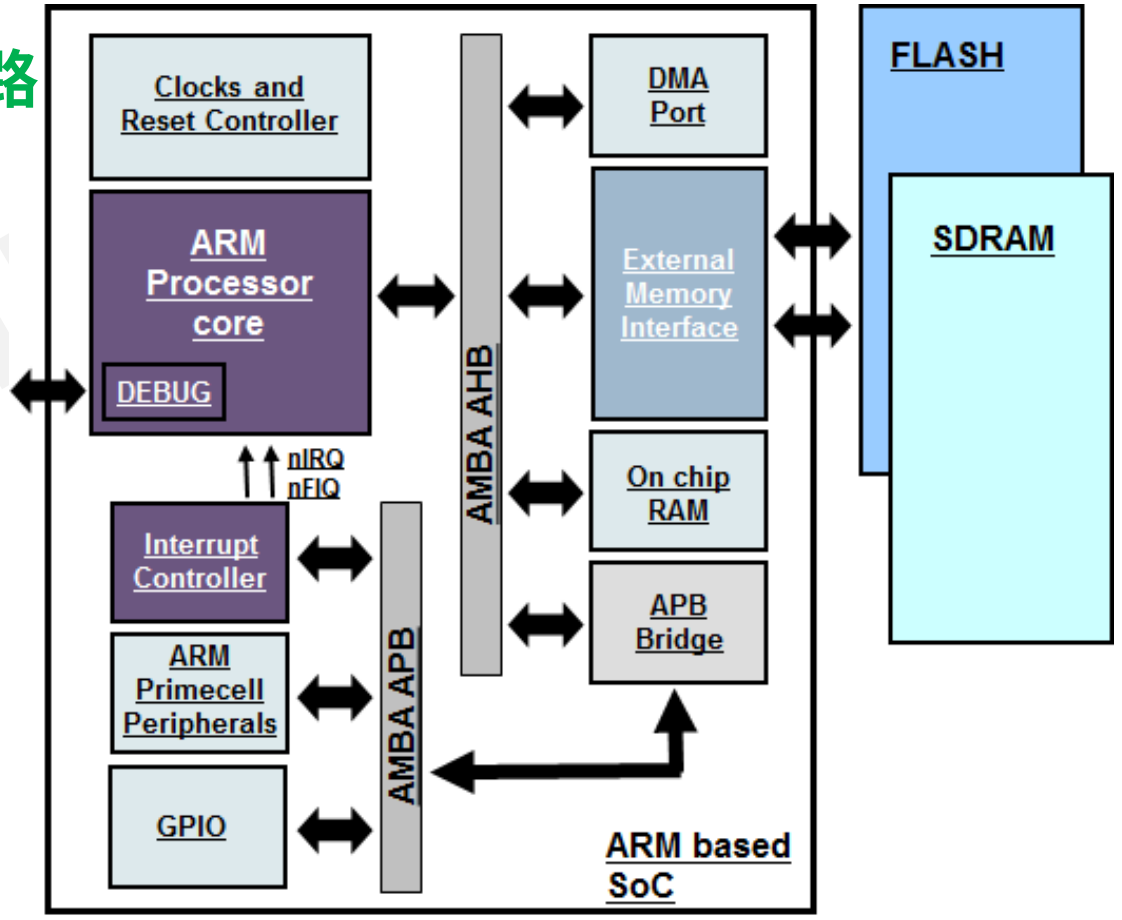
- SoC的全称叫做：
- System-on-a-Chip，
- 把系统都做在一个芯片上





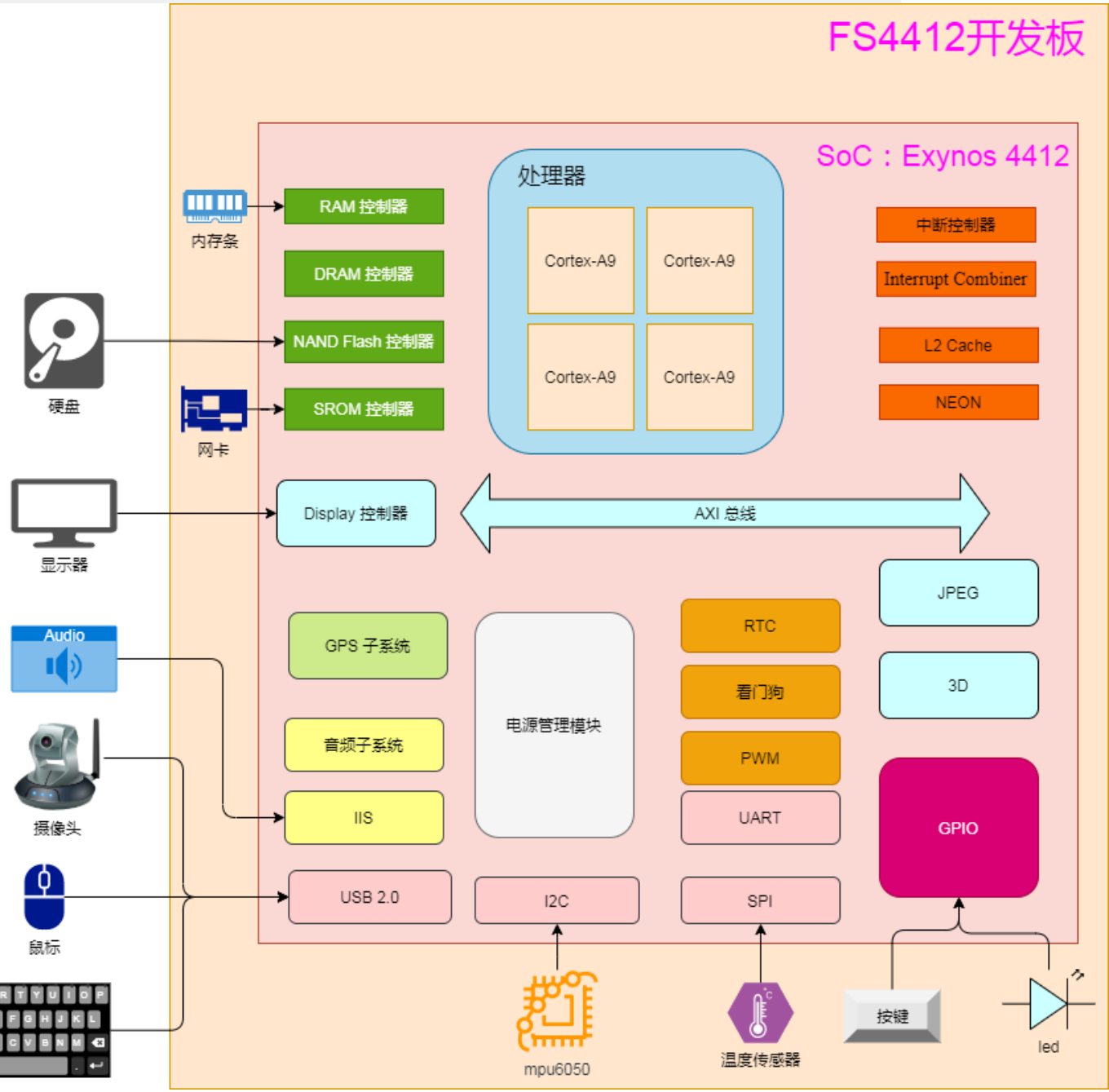
一个典型的基于ARM的Soc架构

- 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

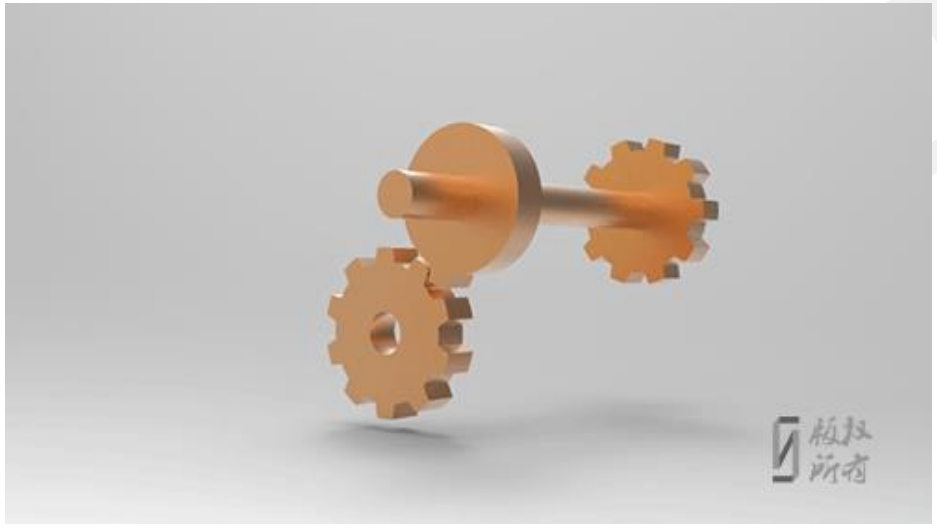
计算机历史

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》

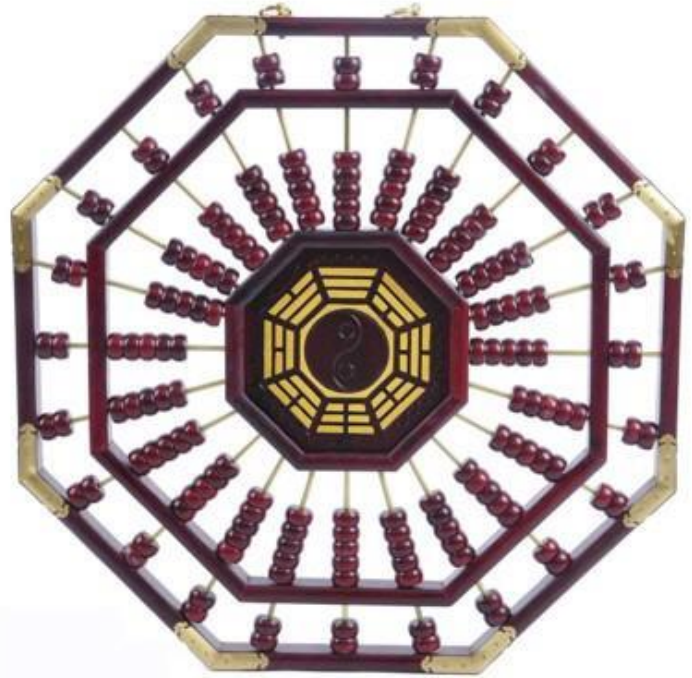
第一台机械计算机

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

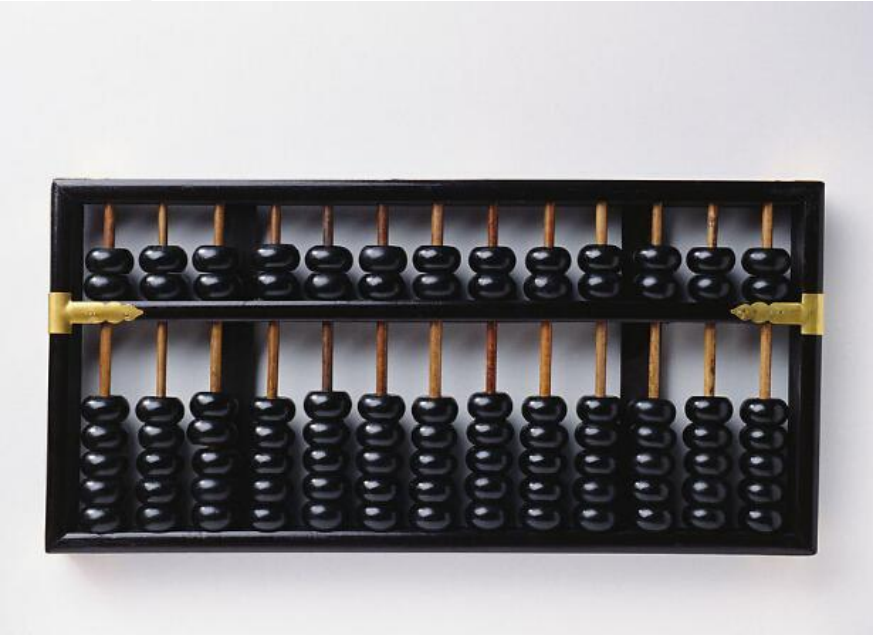


算盘

文王桃木算盘



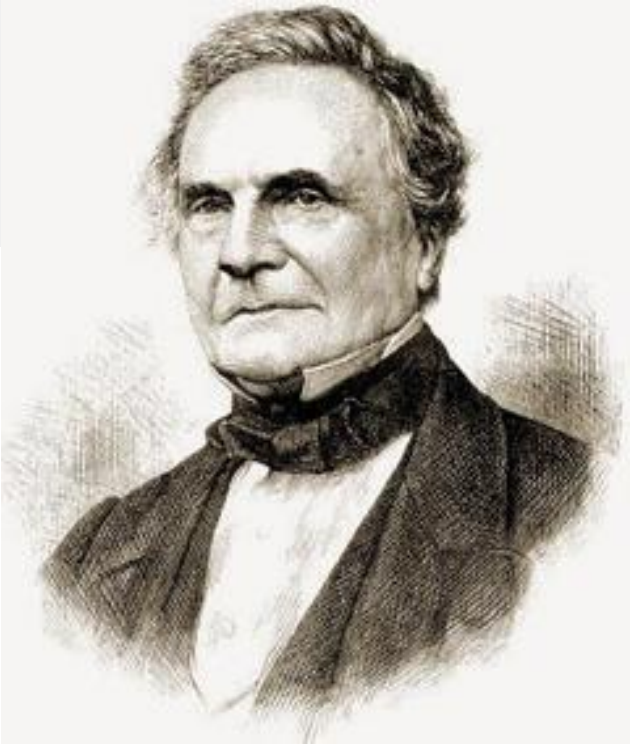
算盘-16进制



四位“计算机之父”

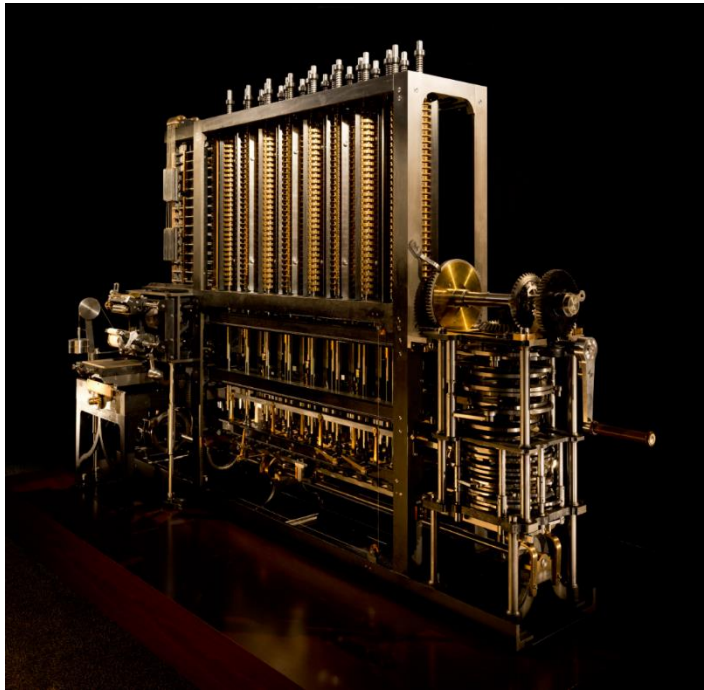
- 巴贝奇　　通用计算机之父
- 图灵　　计算机科学之父
- 约翰·阿坦那索夫　　电子计算机之父
- 冯·诺依曼　　现代计算机之父

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。

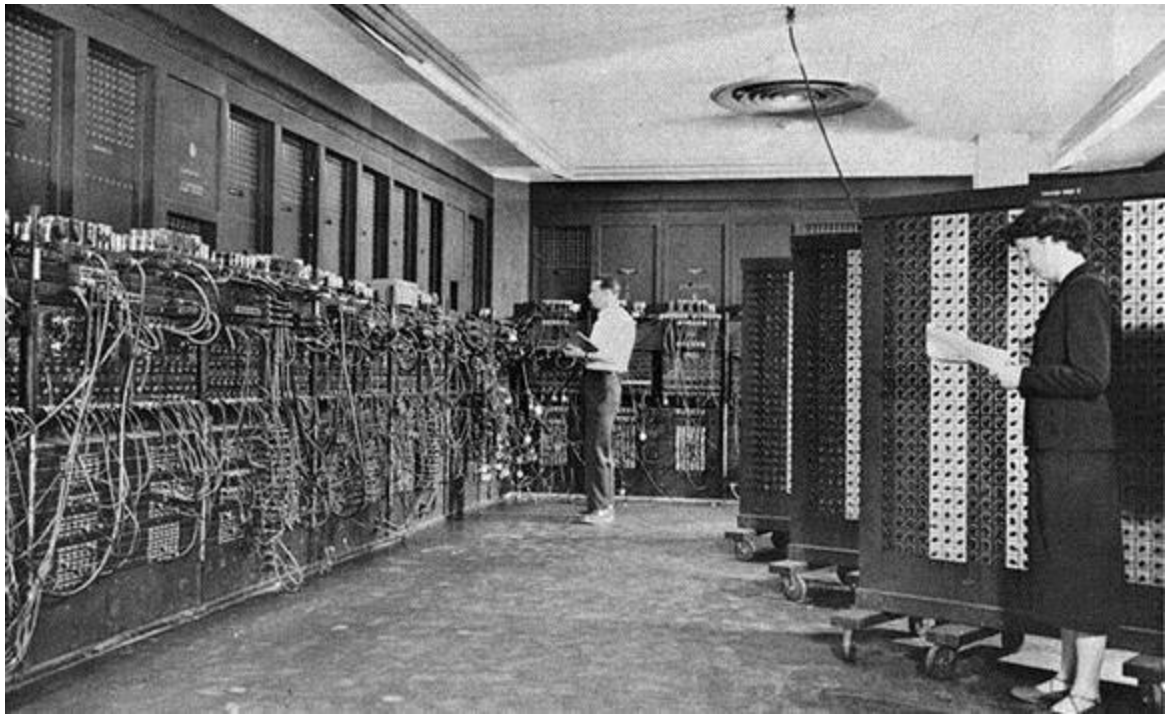
ADA

- 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

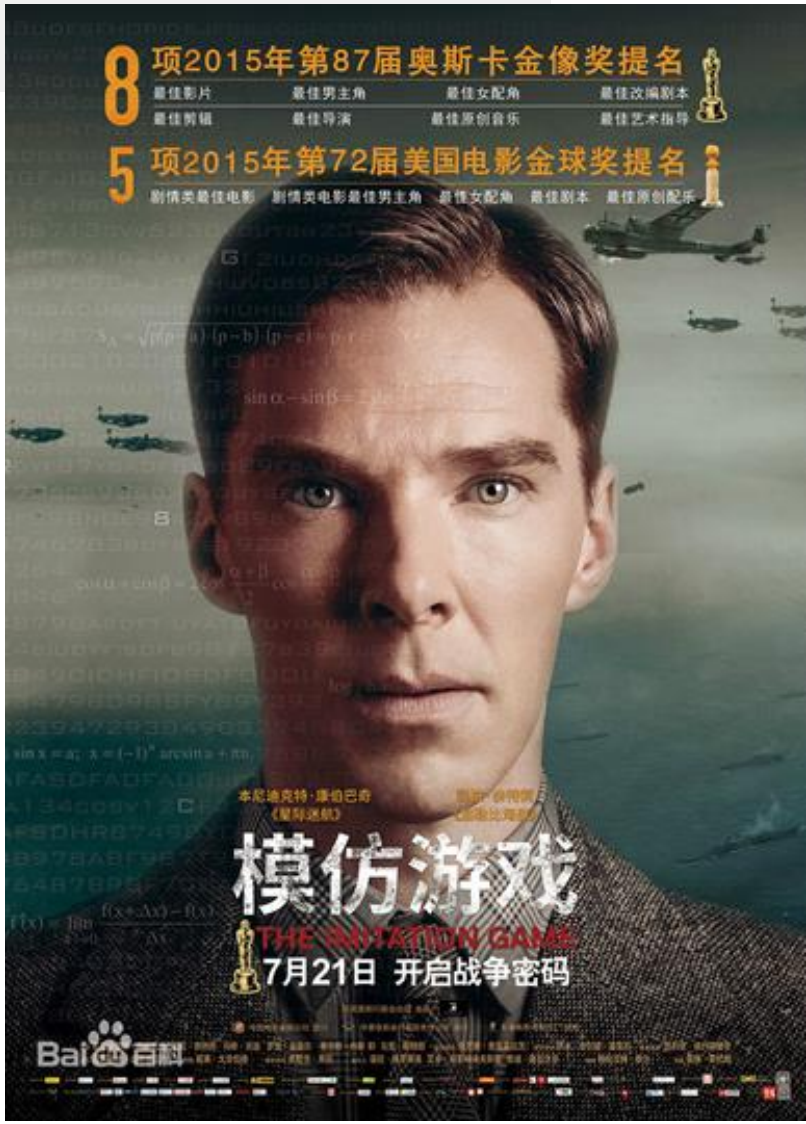
- 阿塔那索夫（J. Atanasoft）是爱阿华大学（University of Iowa）物理学教授

- 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。

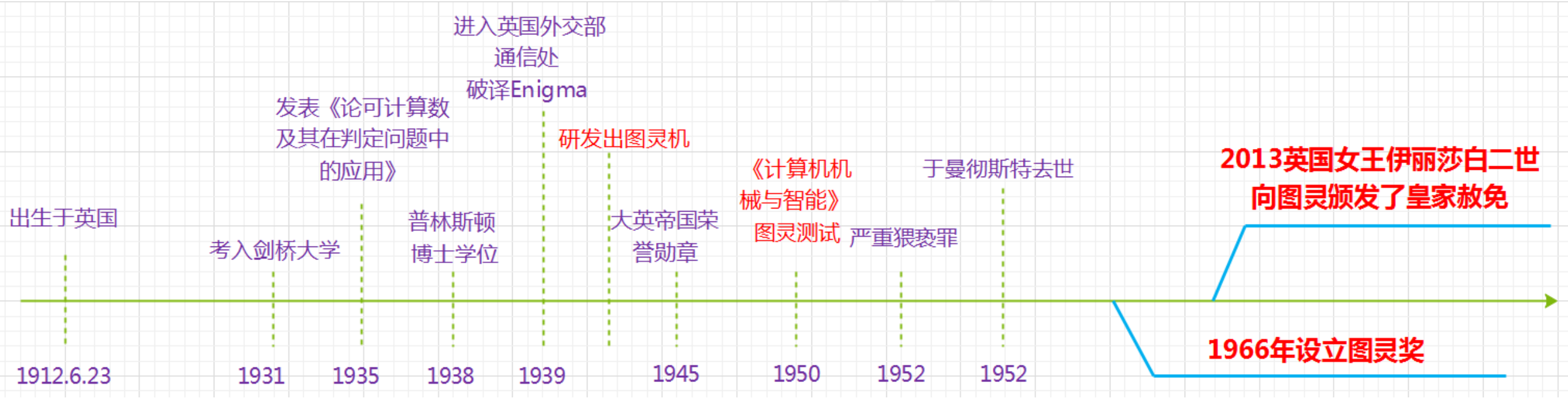


图灵

- 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

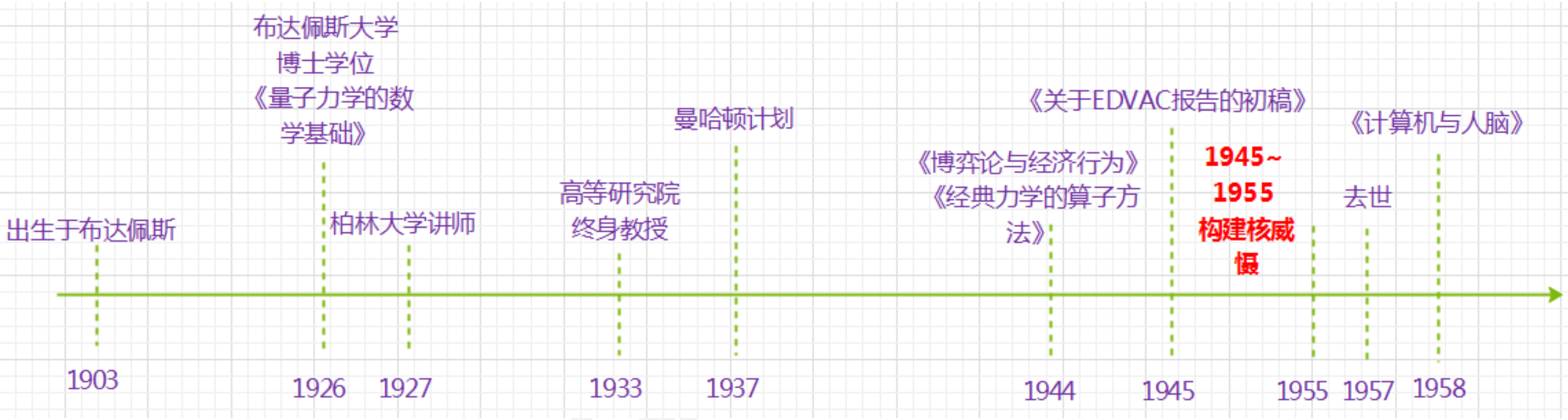


冯诺依曼

- 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”。

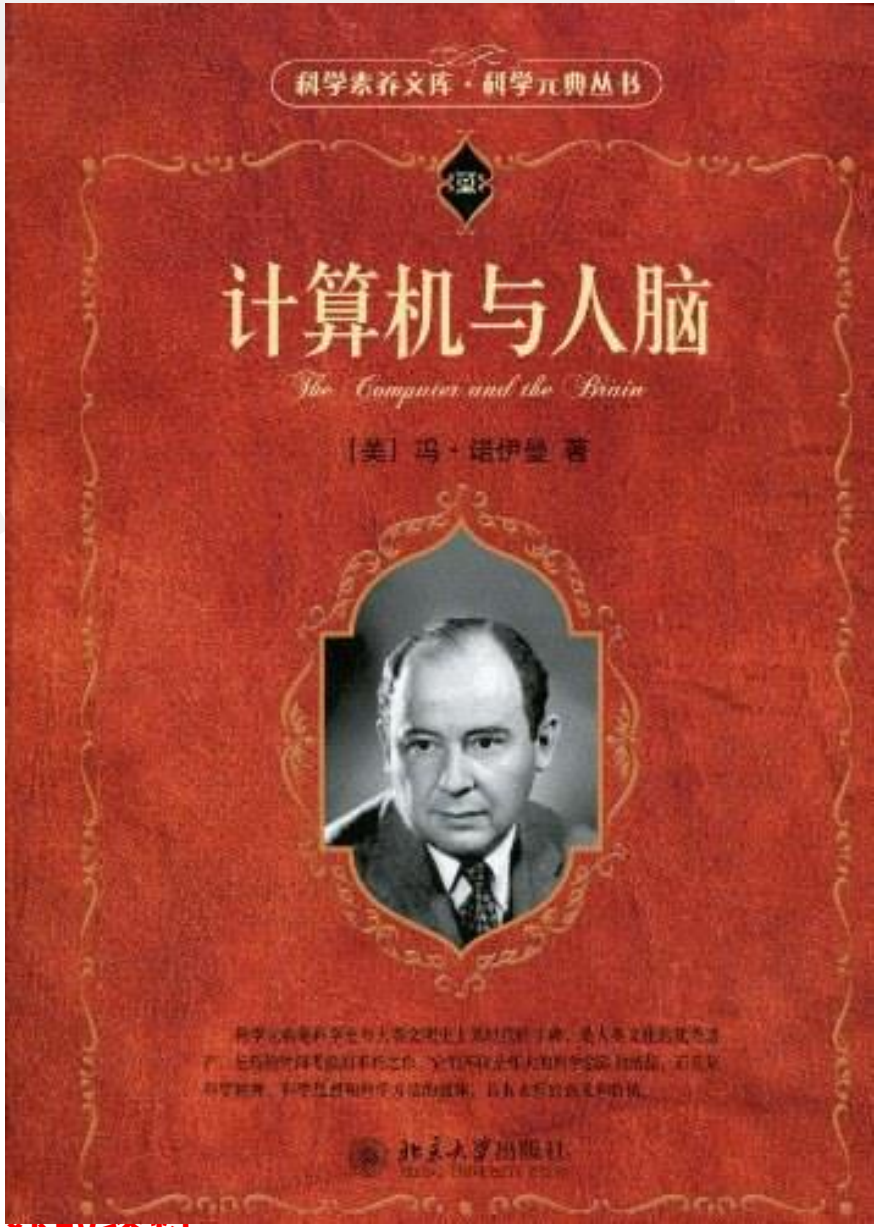
冯诺依曼



《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。

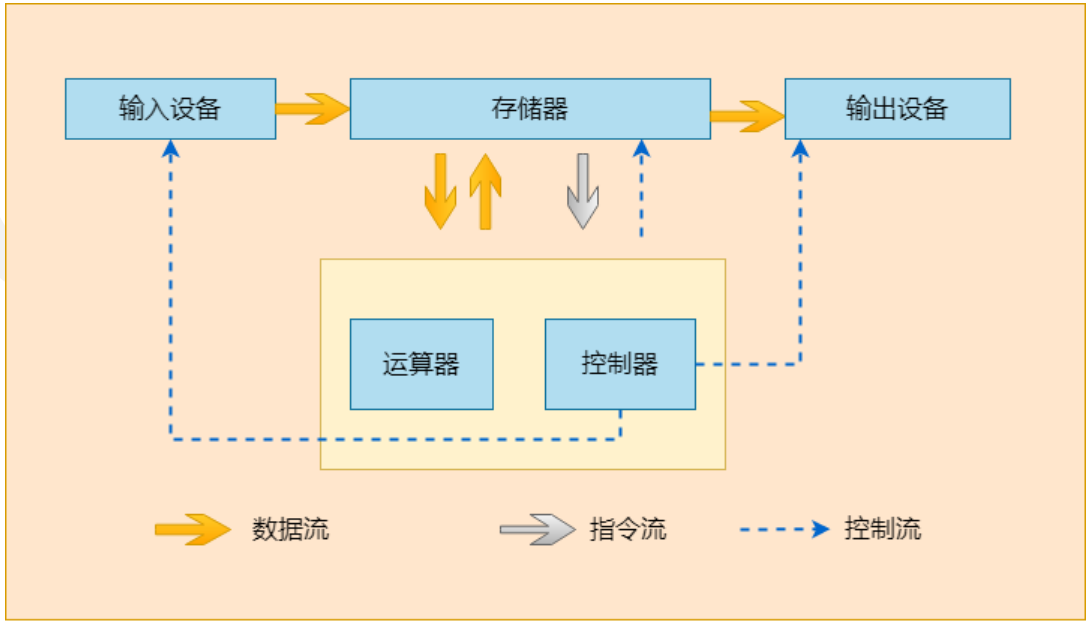
ARM®

05

CPU原理

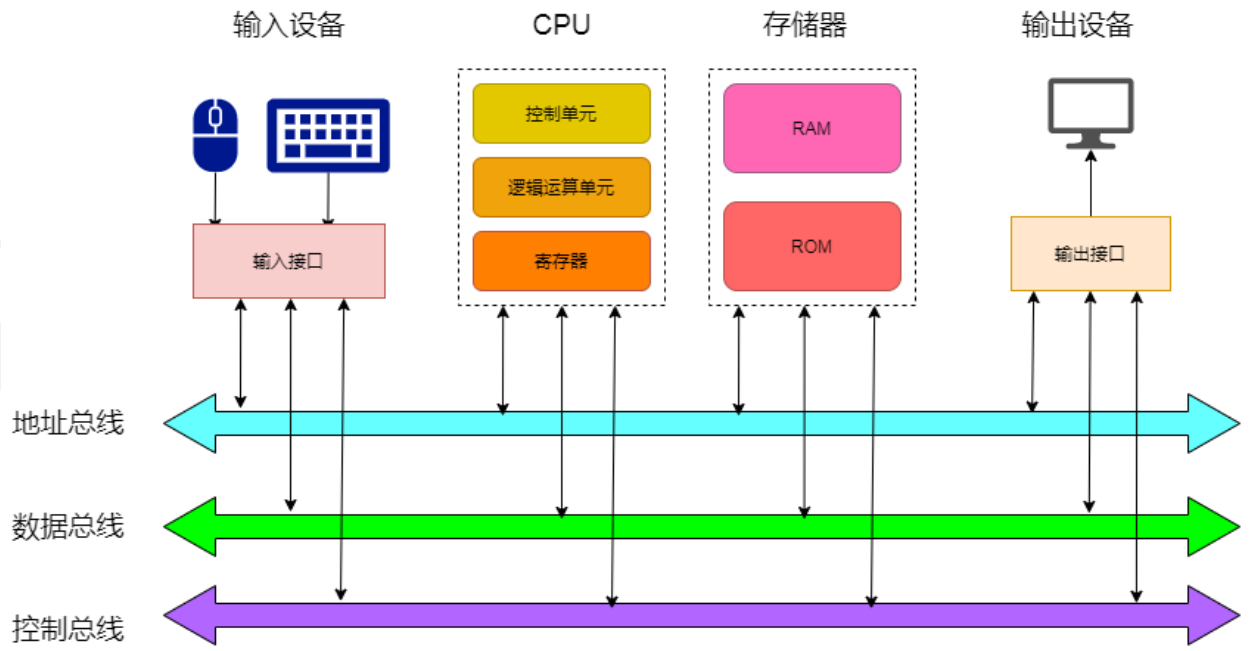
计算机组成

- （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。

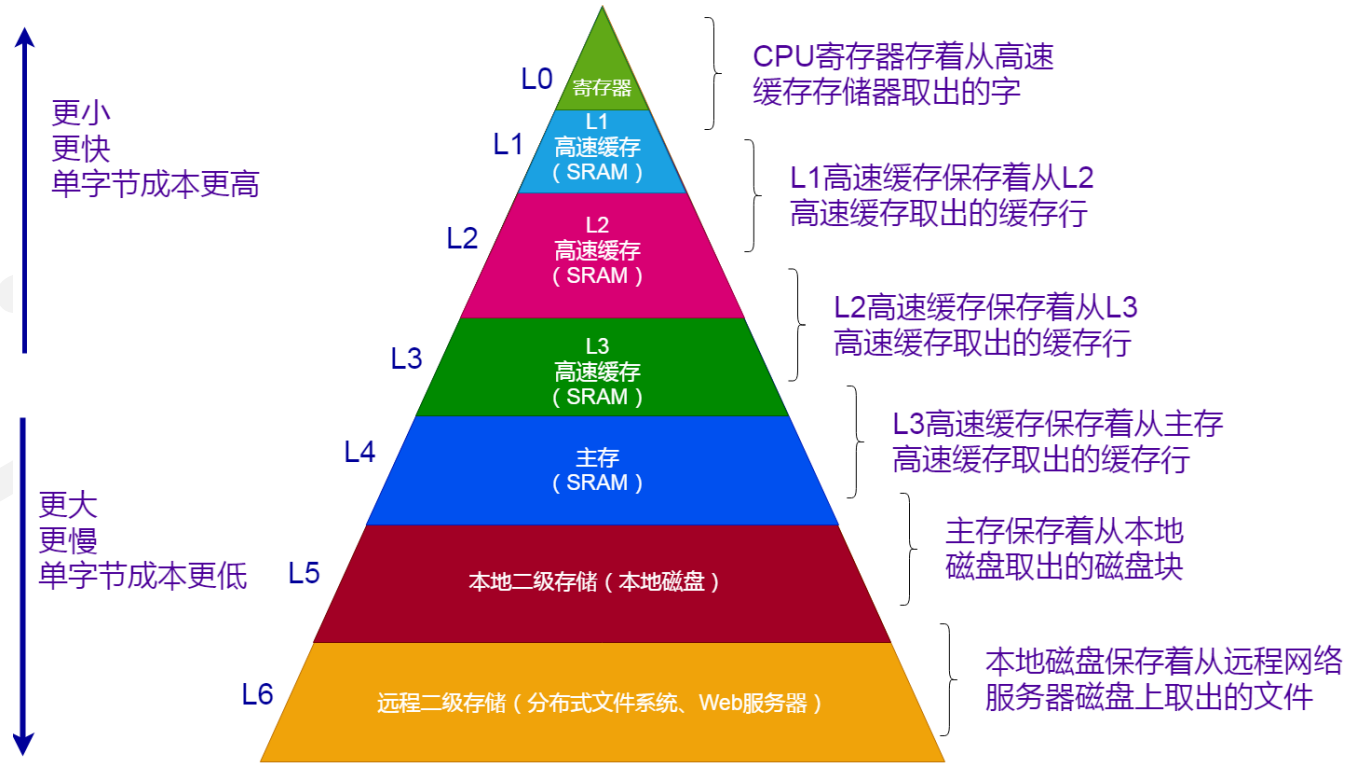
辅助存储器：设置在主机外部，CPU不能直接访问，用来存放暂时不参与运行的程序和数据，需要时再传送到主存。

速度

Cache： 几百到上千**GB/s**

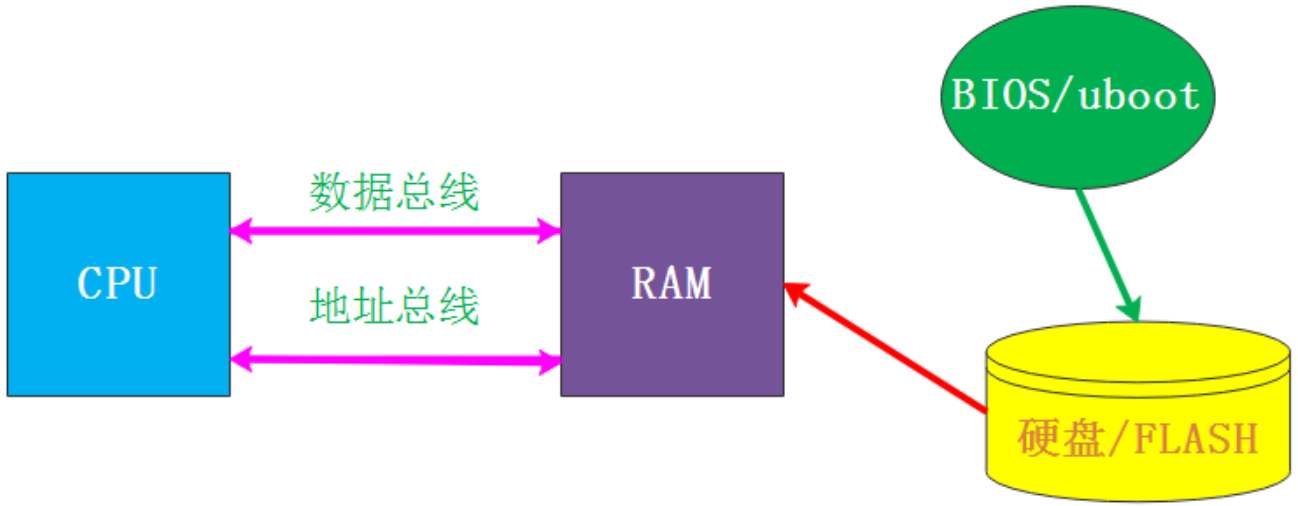
内存： 几十**GB/s**

Disk： 几百**MB/s**



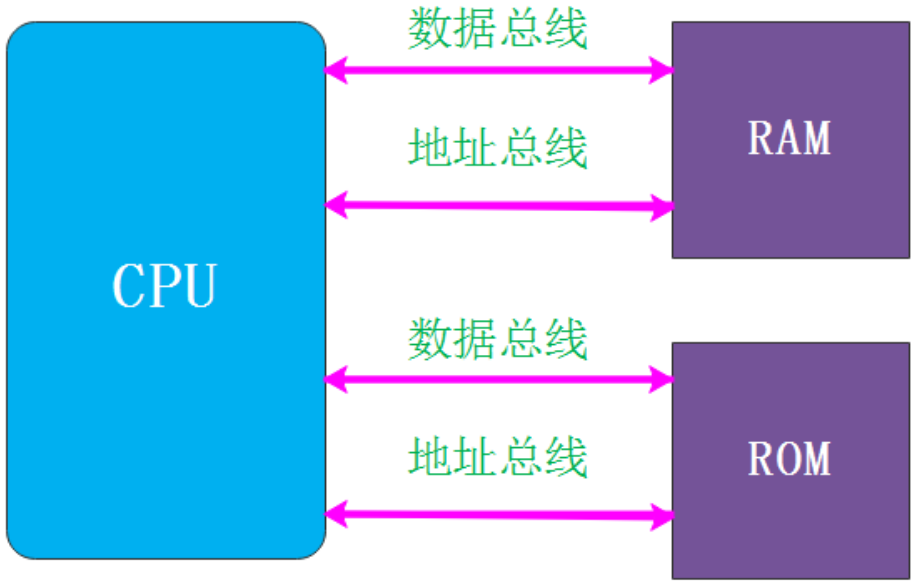
冯诺依曼架构

- 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?

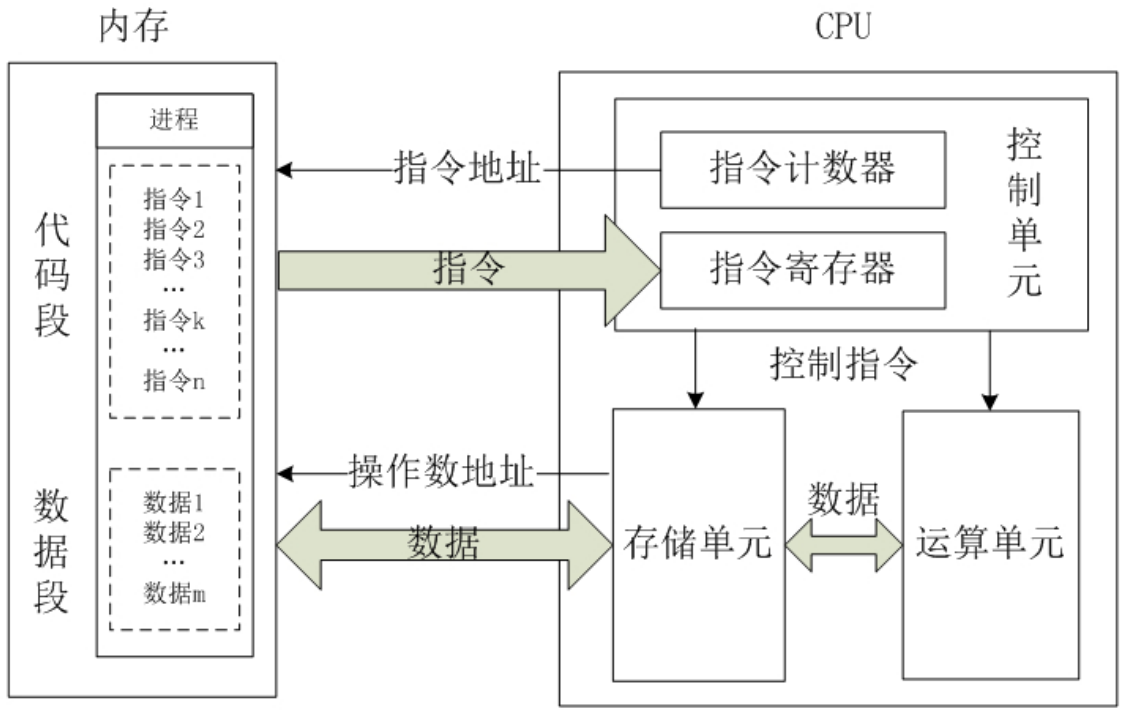
- MCU（单片机）几乎都是用哈佛结构，譬如广泛使用的51单片机、典型的STM32单片机（核心是ARM Cortex-M系列的）都是哈佛结构。
- PC和服务器芯片（譬如Intel AMD），ARM Cortex-A系列嵌入式芯片（譬如三星Exynos 4412，华为的麒麟970等手机芯片）等都是冯诺依曼结构。

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。

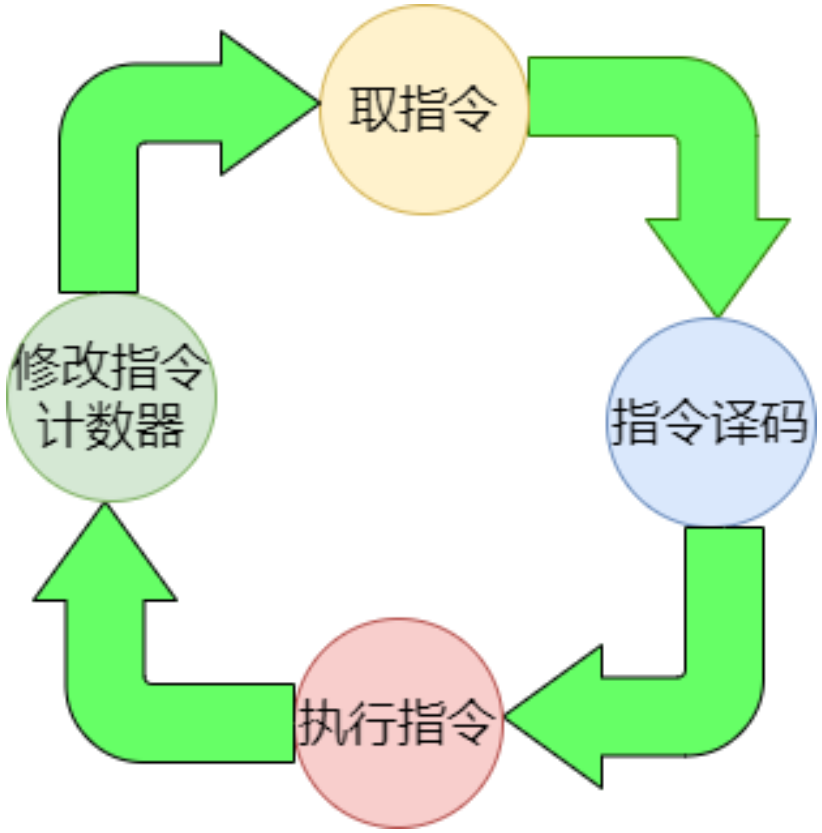
CPU的运行原理

- 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

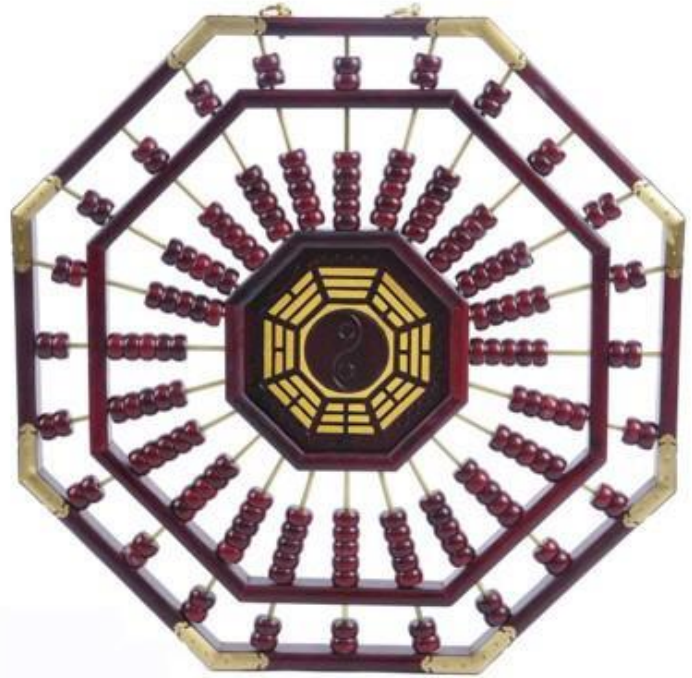
- MIPS
  - 龙芯
- X86
  - intel

ARM指令格式

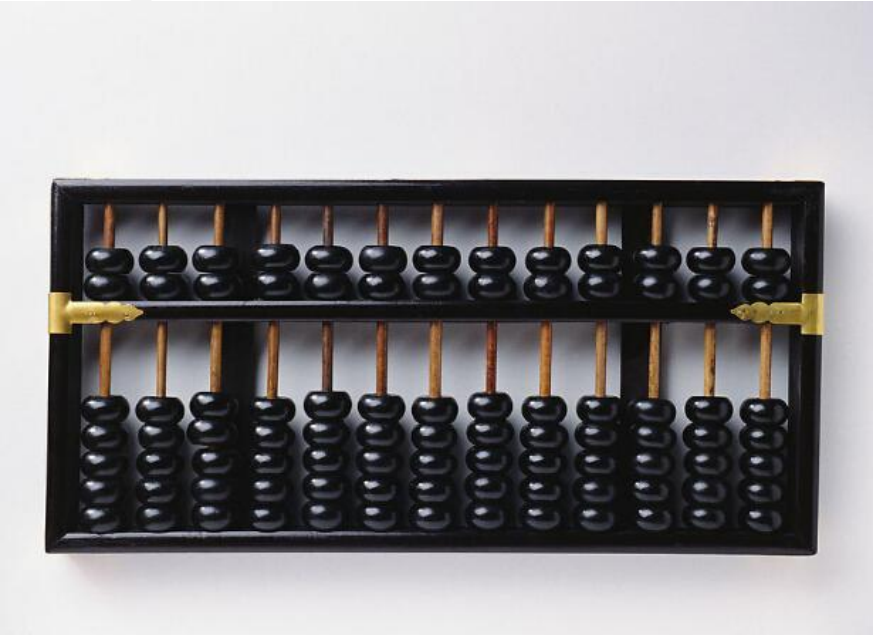
31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	
Cond	0 0	I	Opcode	S	Rn	Rd	Operand2			数据处理/PSR 传送
Cond	0 0	0	0 0 0 A	S	Rd	Rn	Rs	1 0 0 1	Rm	乘法
Cond	0 0	0	0 1 U A	S	RdHi	RdLo	Rn	1 0 0 1	Rm	长乘法
Cond	0 0	0	1 0 B 0	0	Rn	Rd	0 0 0 0	1 0 0 1	Rm	单数据交换
Cond	0 0	0	1 0 0 1	0	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

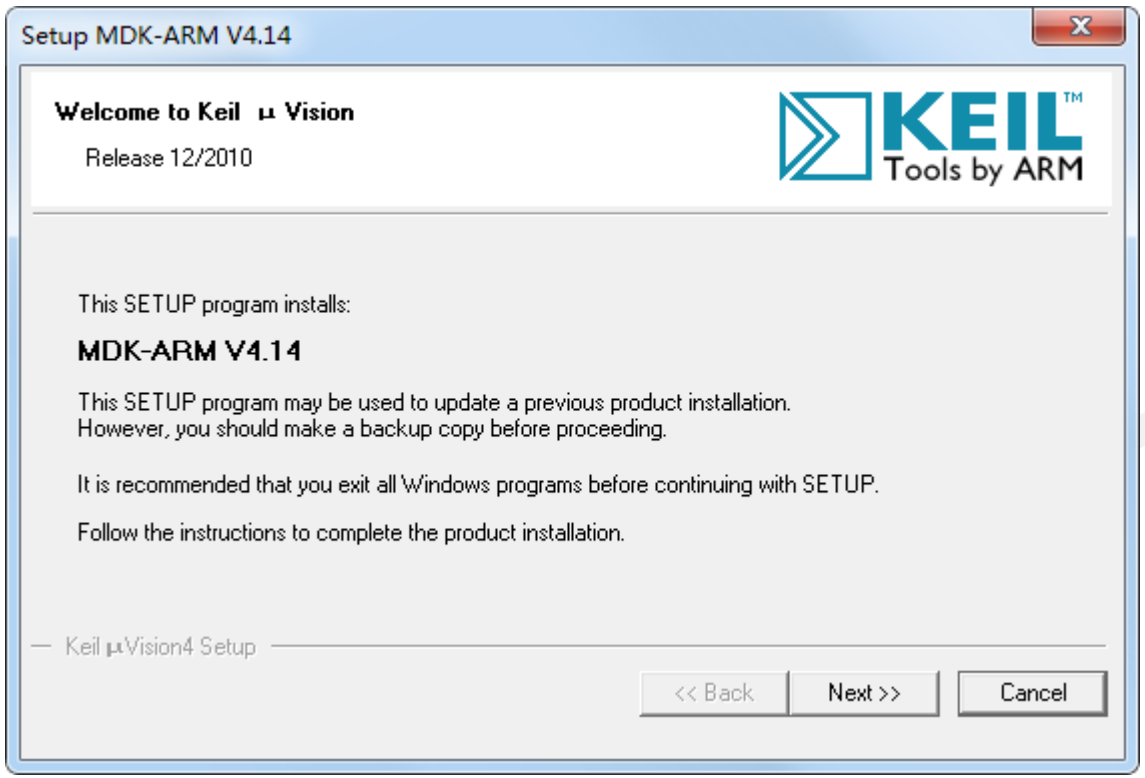
KEIL、MDK、uVision、ARM之间的关系

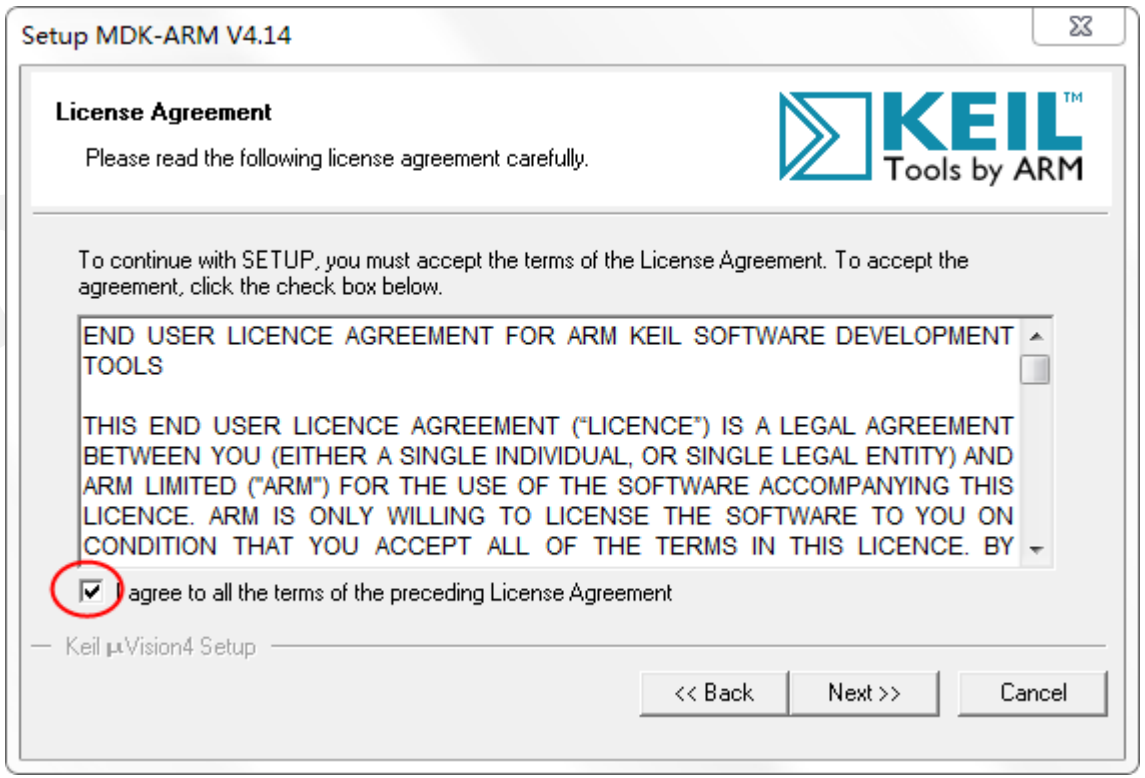
- 1）KEIL
- 1) 既是公司名称 ， 同时也是KEIL公司所有的开发工具。
- .2) 2005年被ARM收购。
- 2）uVision
- 1) KEIL公司开发的集成开发环境（IDE）。
- 2) 共有4个版本：uVision2、uVision3、uVision4、uVision5。
- 3）MDK
- 1) 英文全称：Microcontroller Development Kit。
- 2) MDK-ARM = KEIL MDK = RealView MDK = KEIL For ARM，统一用 MDK-ARM 称呼。

MDK-ARM

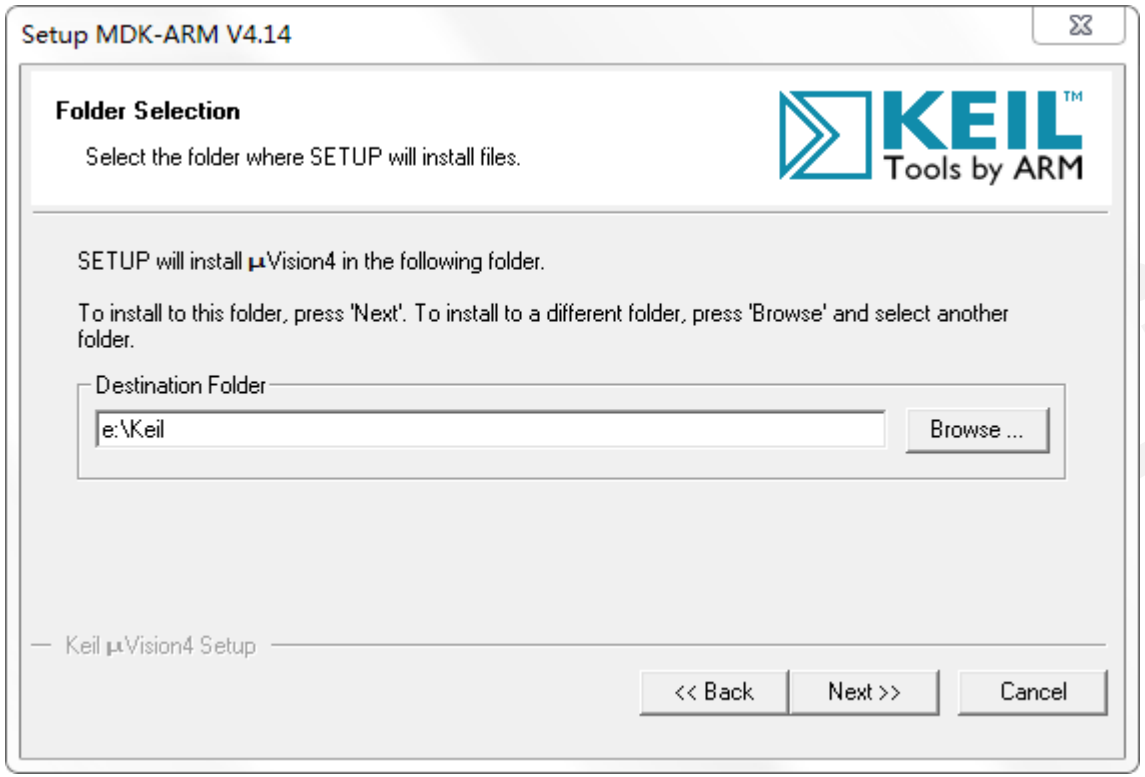
- MDK-ARM 为基于Cortex-M、Cortex-R4、ARM7、ARM9处理器设备提供了一个完整的开发环境。 MDK-ARM专为微控制器应用而设计，不仅易学易用，而且功能强大，能够满足大多数苛刻的嵌入式应用。

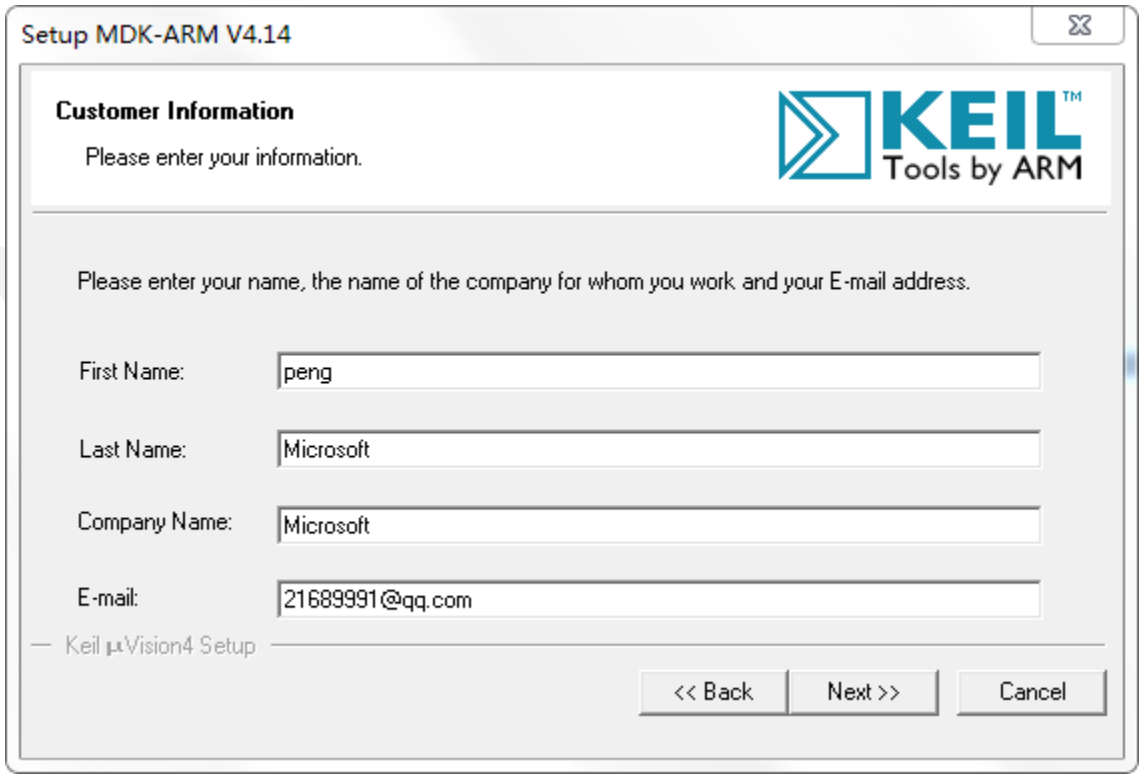
- MDK-ARM有四个可用版本，分别是MDK-Lite、MDK-Basic、MDK-Standard、MDK-Professional。所有版本均提供一个完善的C / C++开发环境，其中MDK-Professional还包含大量的中间库。

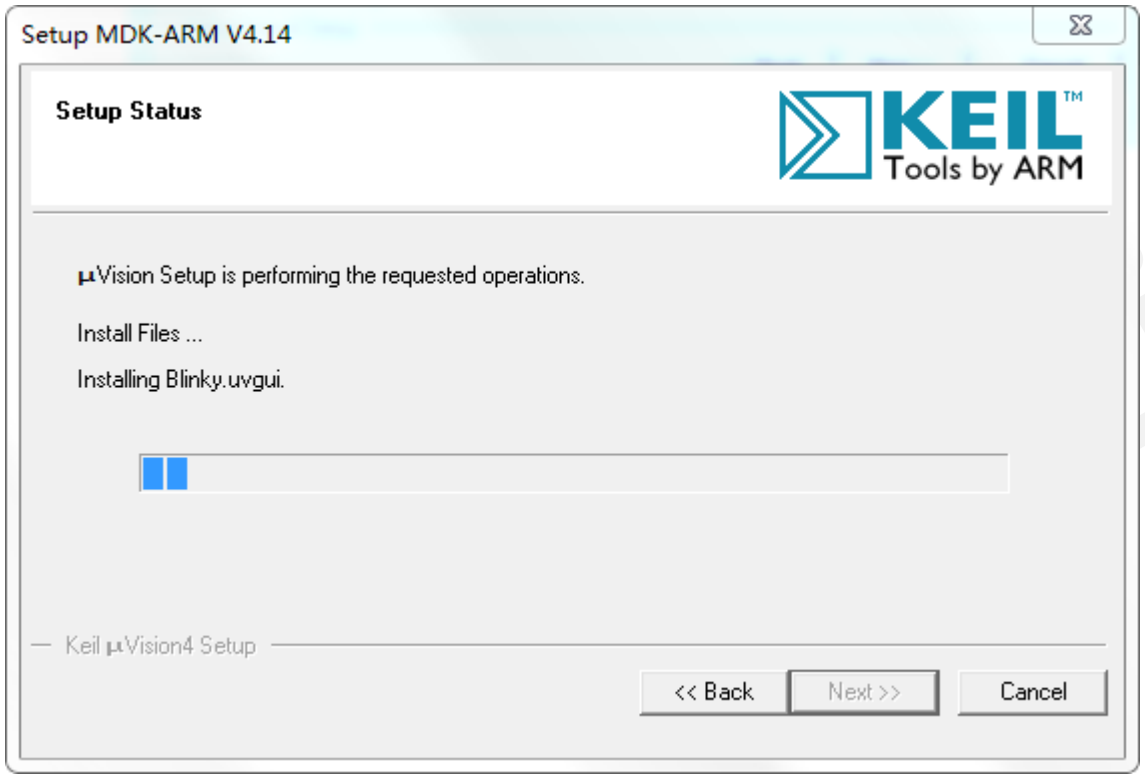


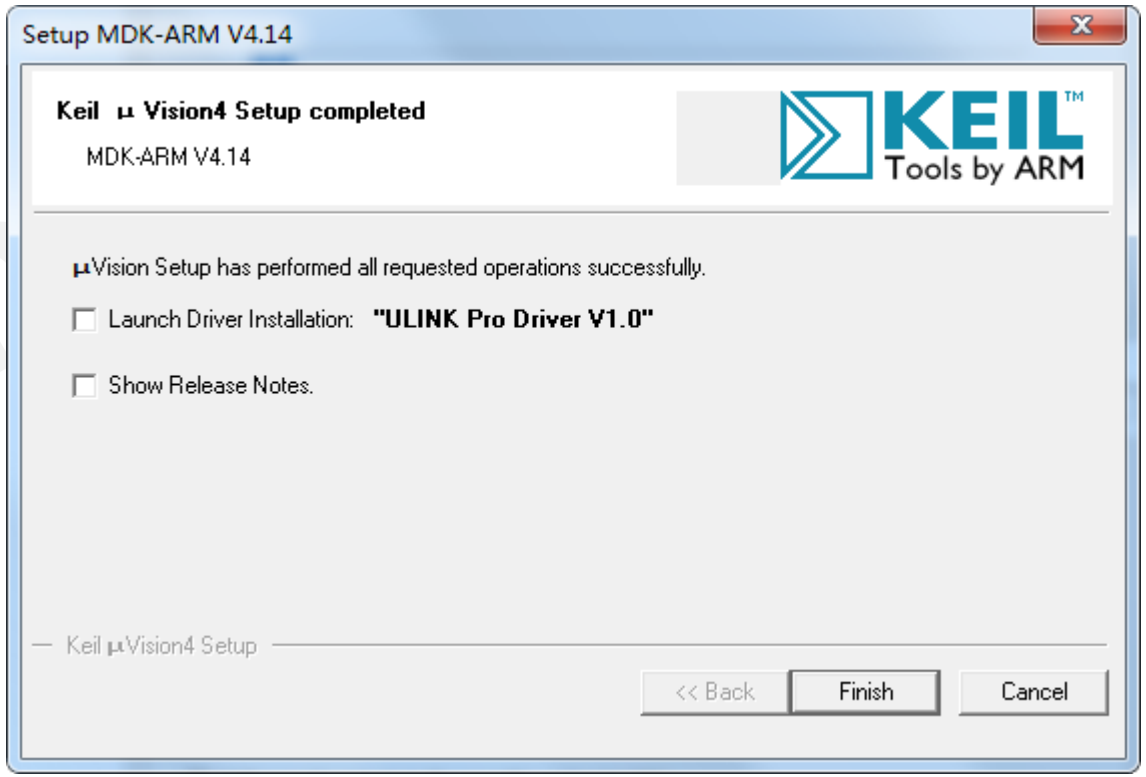


选择安装目录，尽量不要有中文目录：

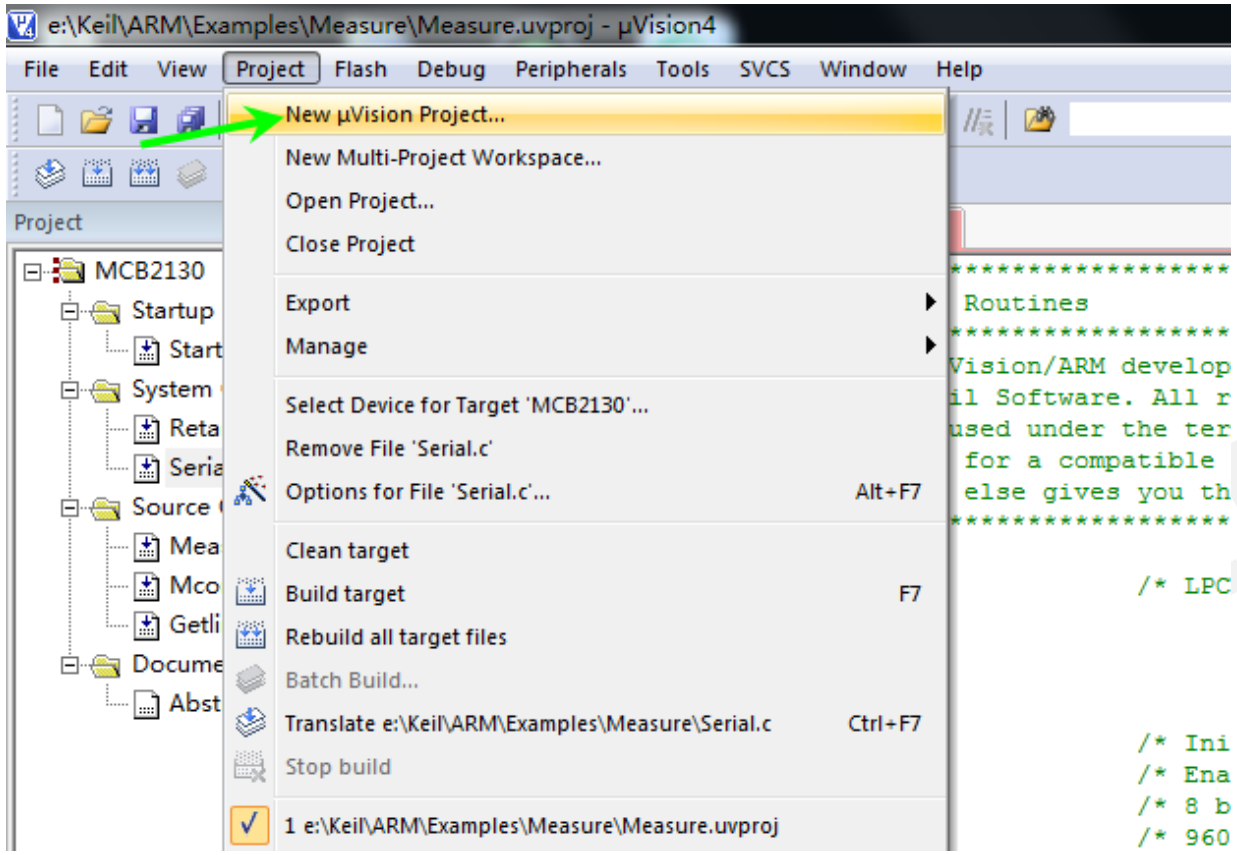


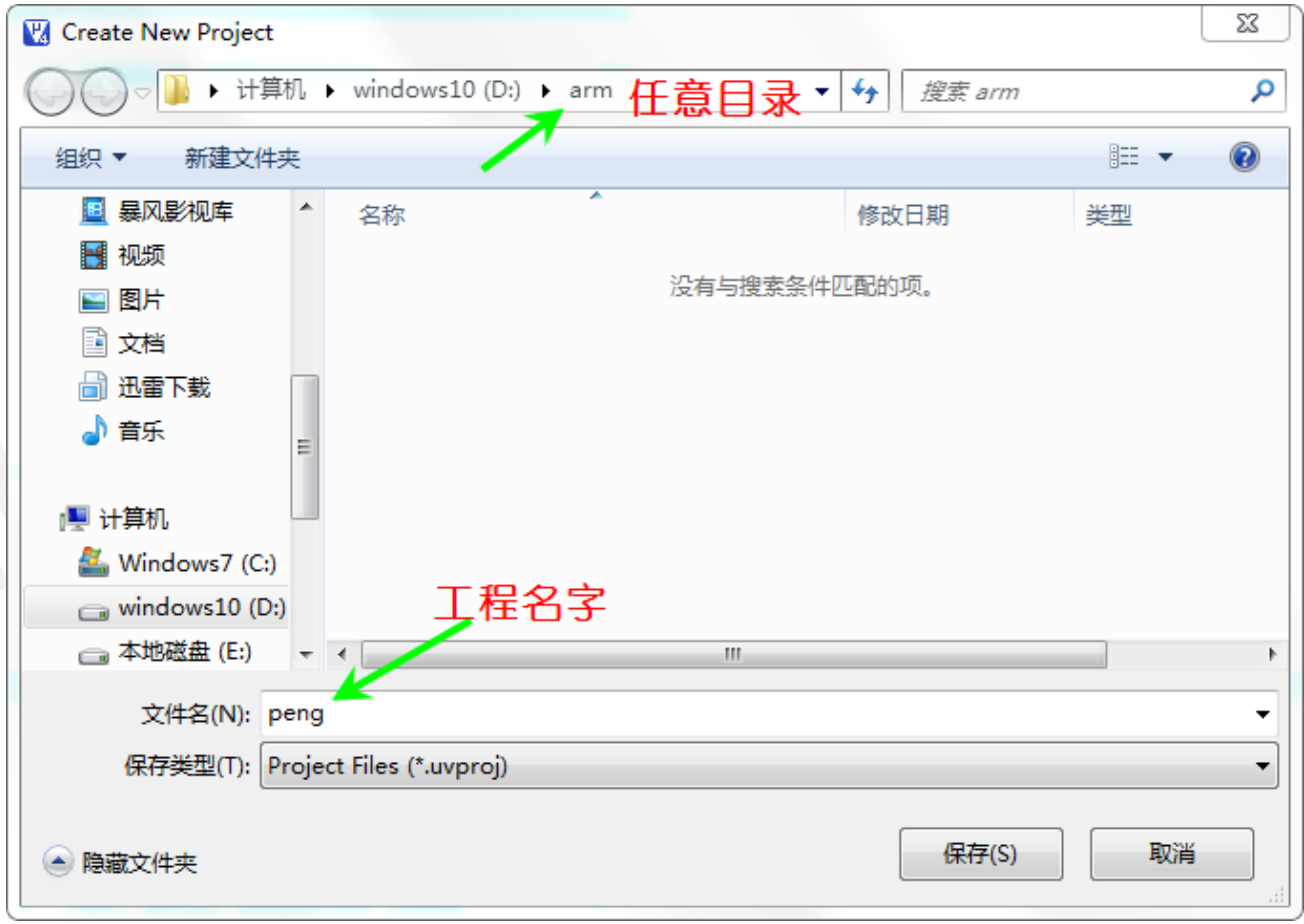




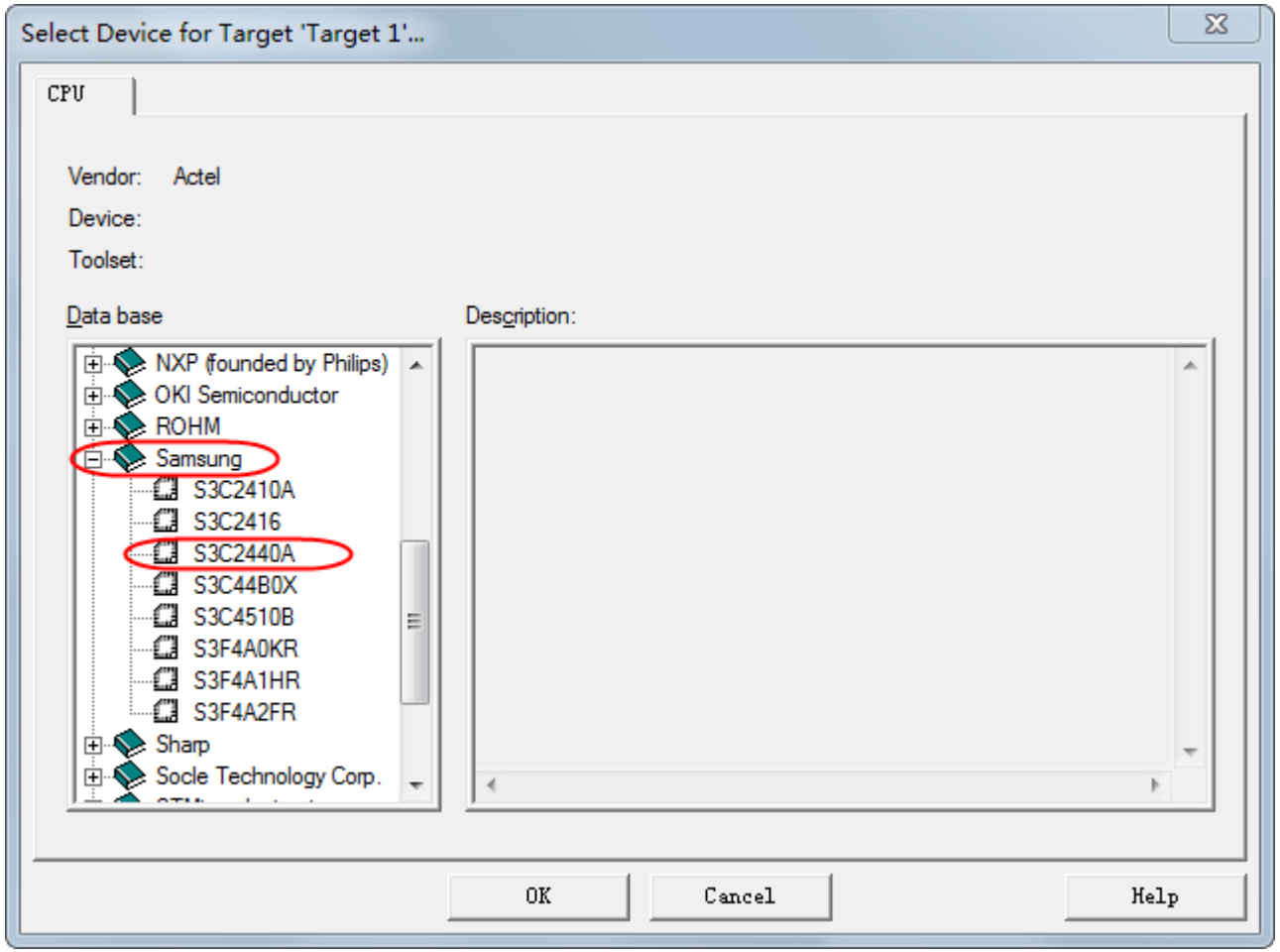


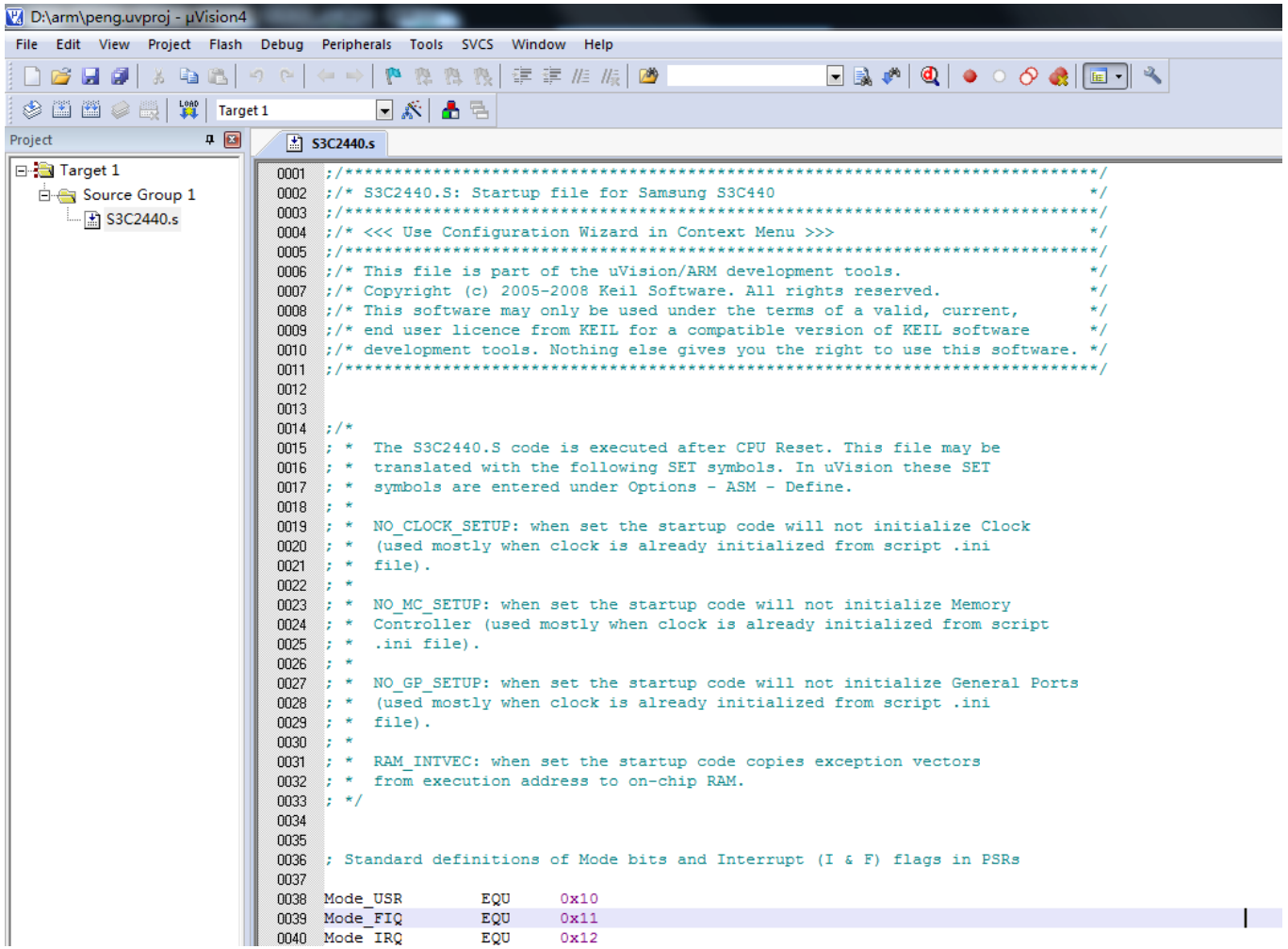
创建第一个工程



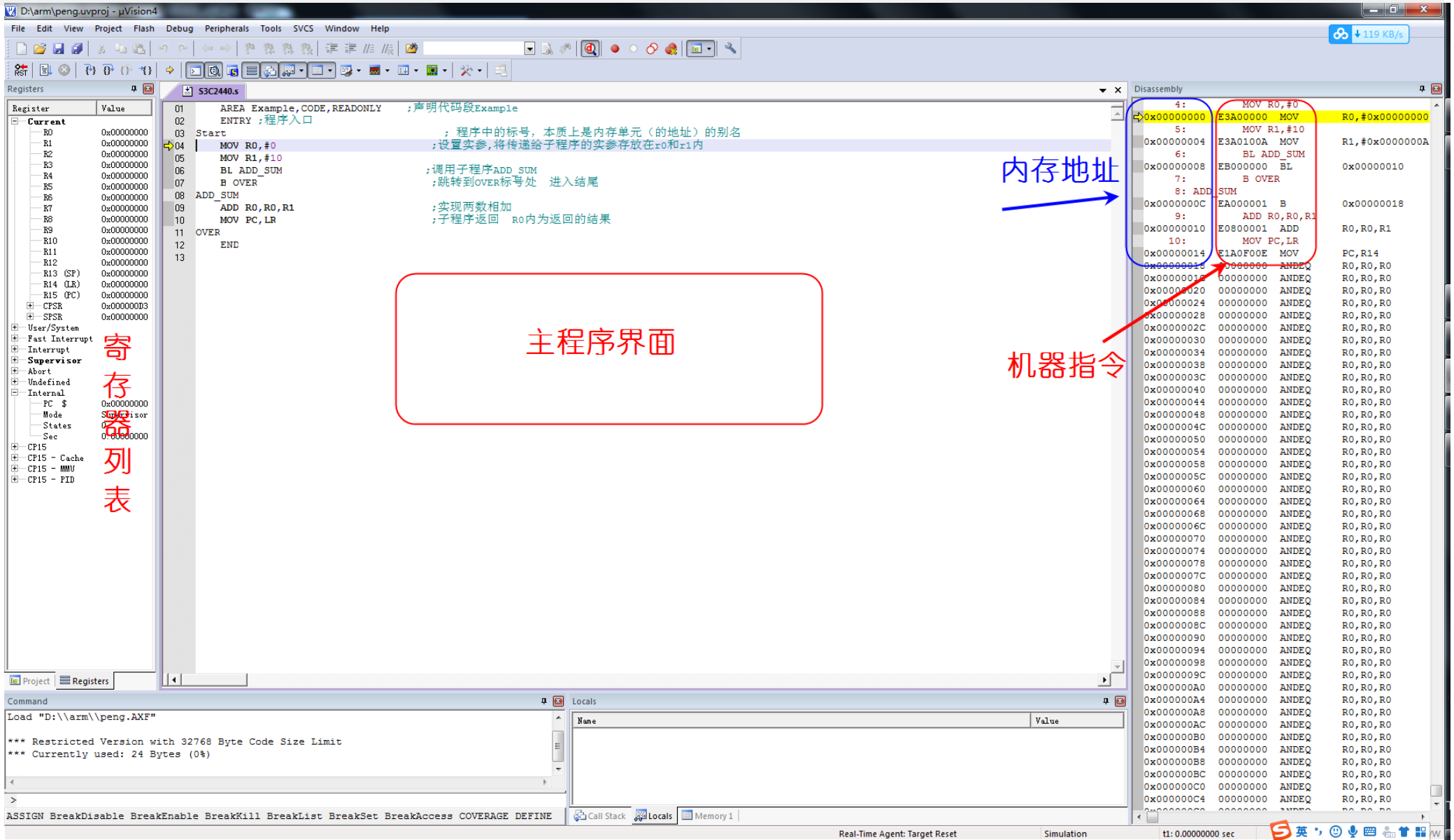


选择CPU->Samsung->S3C2440A





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
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

关

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。

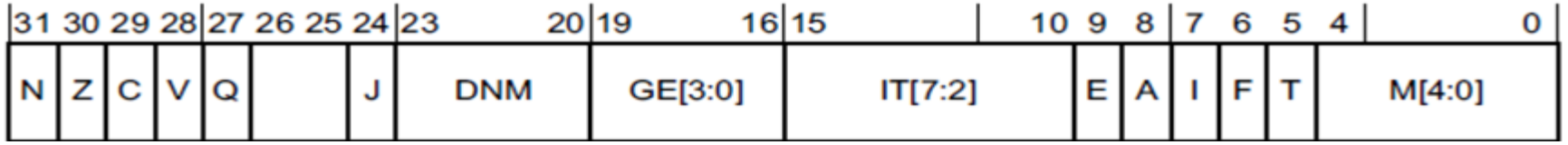
- 由于ARM体系结构采用了多级流水线技术，对于ARM指令集而言，PC总是指向当前指令的下两条指令的地址,即PC的值为当前指令的地址值加8个字节。
- 即：PC值=当前程序执行位置+8

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



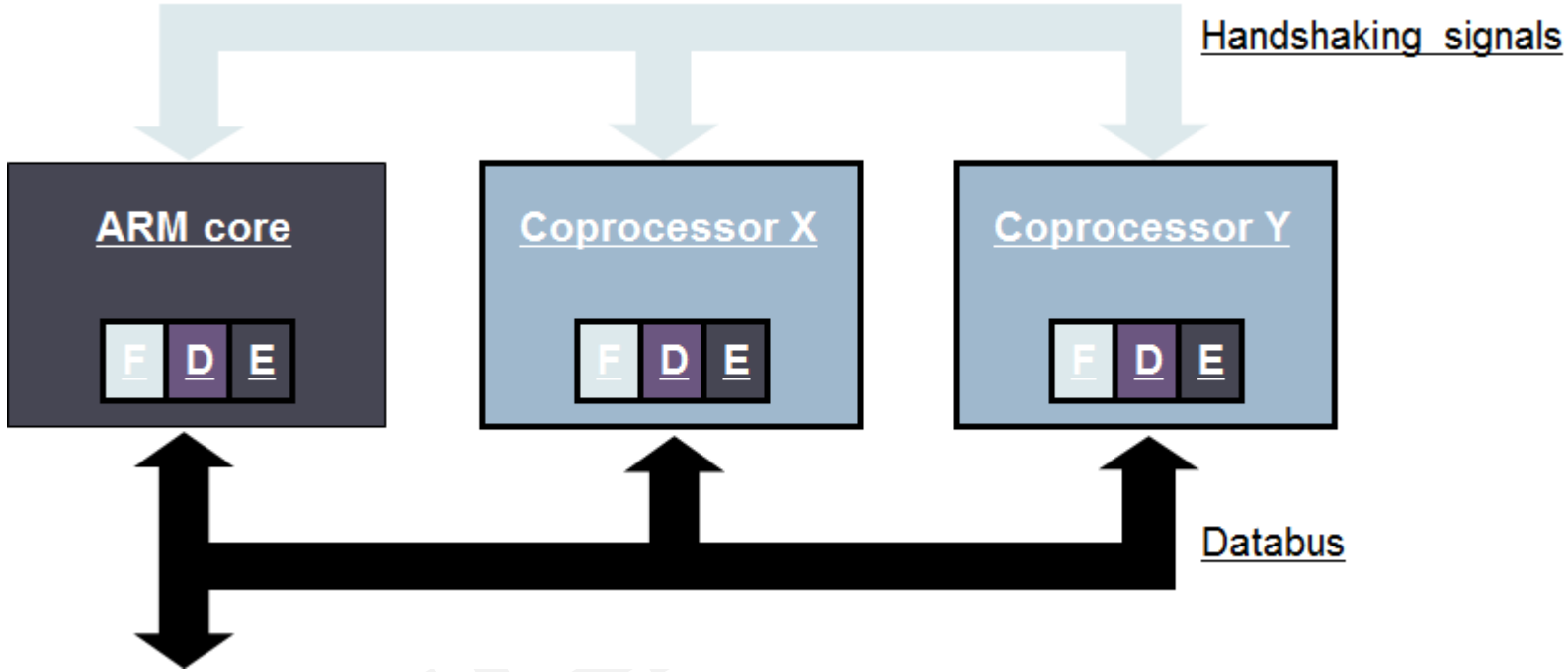
- **条件位：**
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out or borrow
  - V = ALU operation o**V**erflowed
- Q **位：**
  - **仅**ARM v5TE-J**架构支持**
  - **指示饱和状态**
- J **位**
  - **仅**ARM v5TE-J**架构支持**
  - T=0;J = 1 **处理器处于**Jazelle**状态**
  - **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

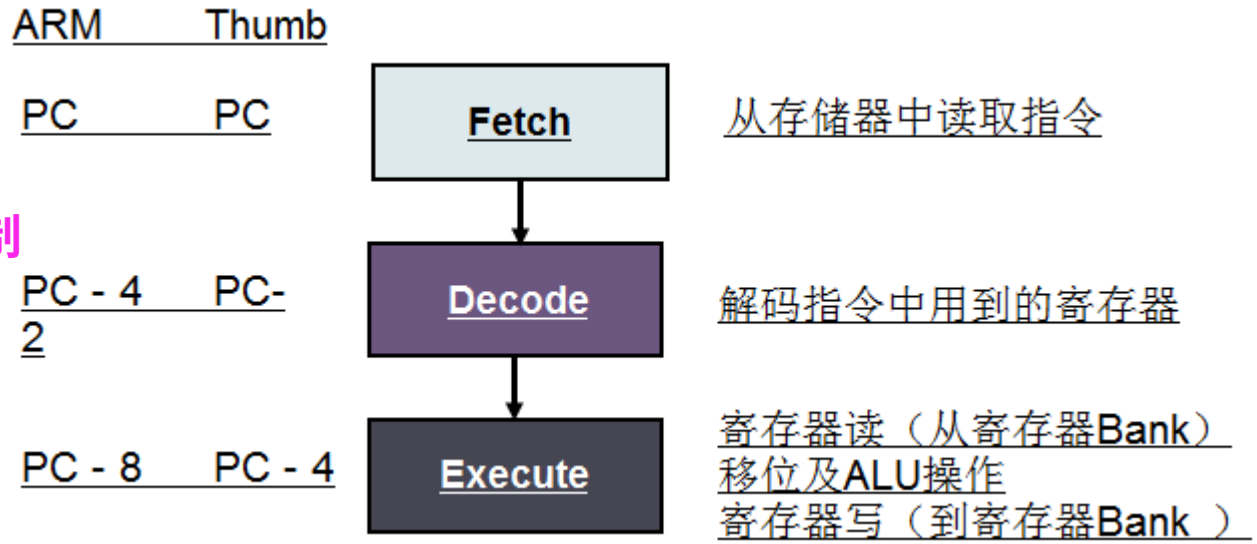
流水线

- 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。



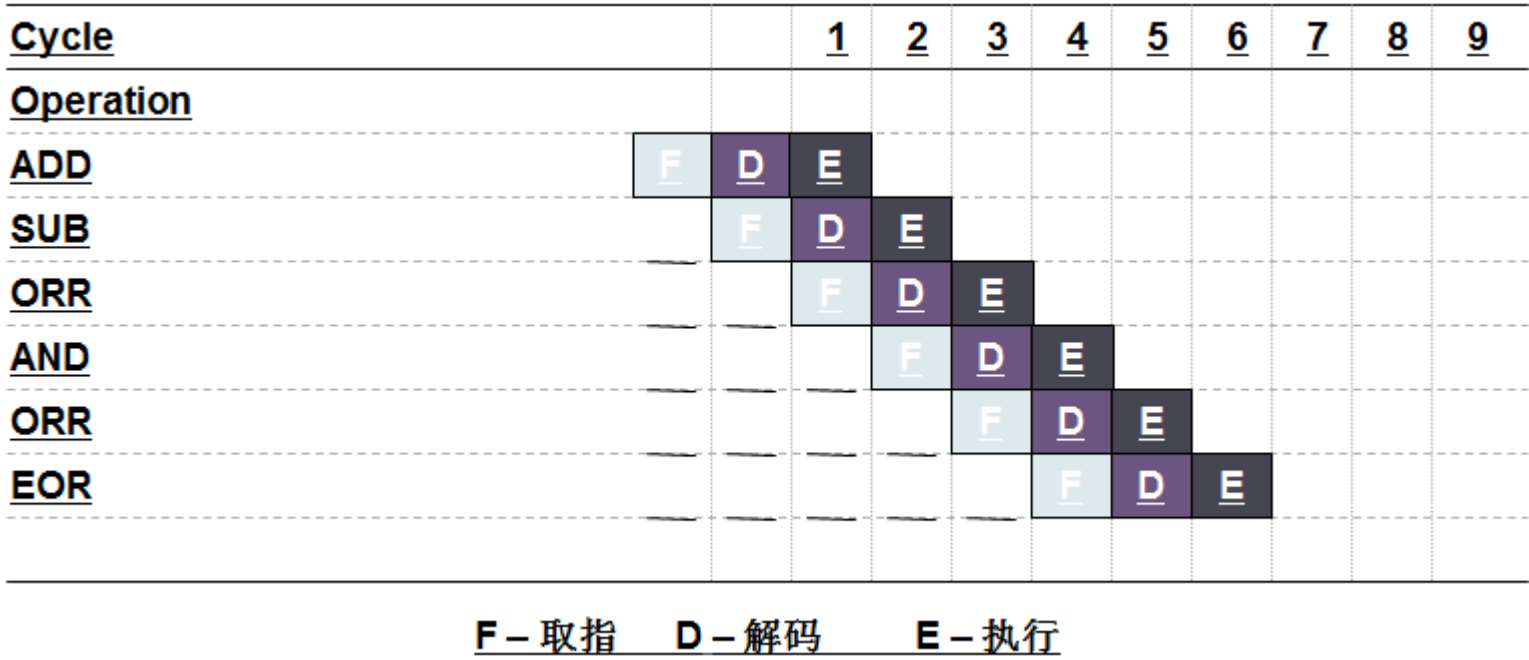
当处理器执行简单的数据处理指令时，流水线使得平均每个时钟周期能完成1条指令。但一条指令需要3个时钟周期来完成，因此有3个时钟周期的延时，但吞吐率是每个周期一条指令。
对于3级流水线，PC寄存器里的值并不是正在执行的指令的地址，而是预取指令的地址

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

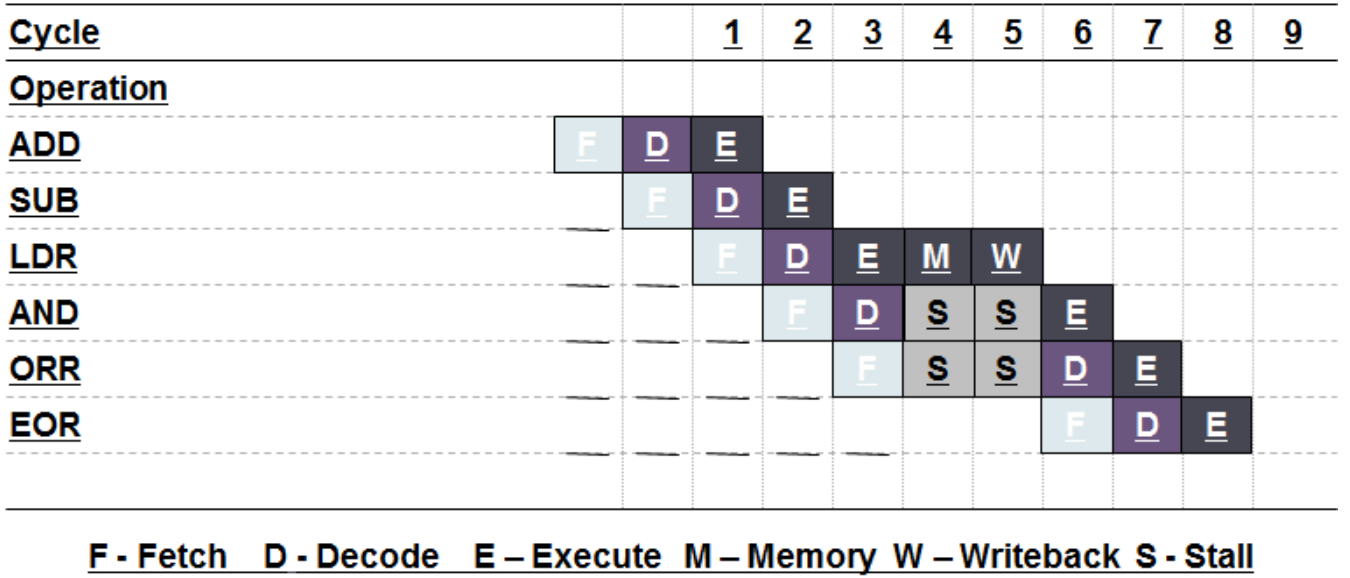
1、数据总线在周期1, 2, 3 被使用，周期6是取指，周期4用于数据装载，而周期5是一个内部周期用来完成载入的数据写回到寄存器中。

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

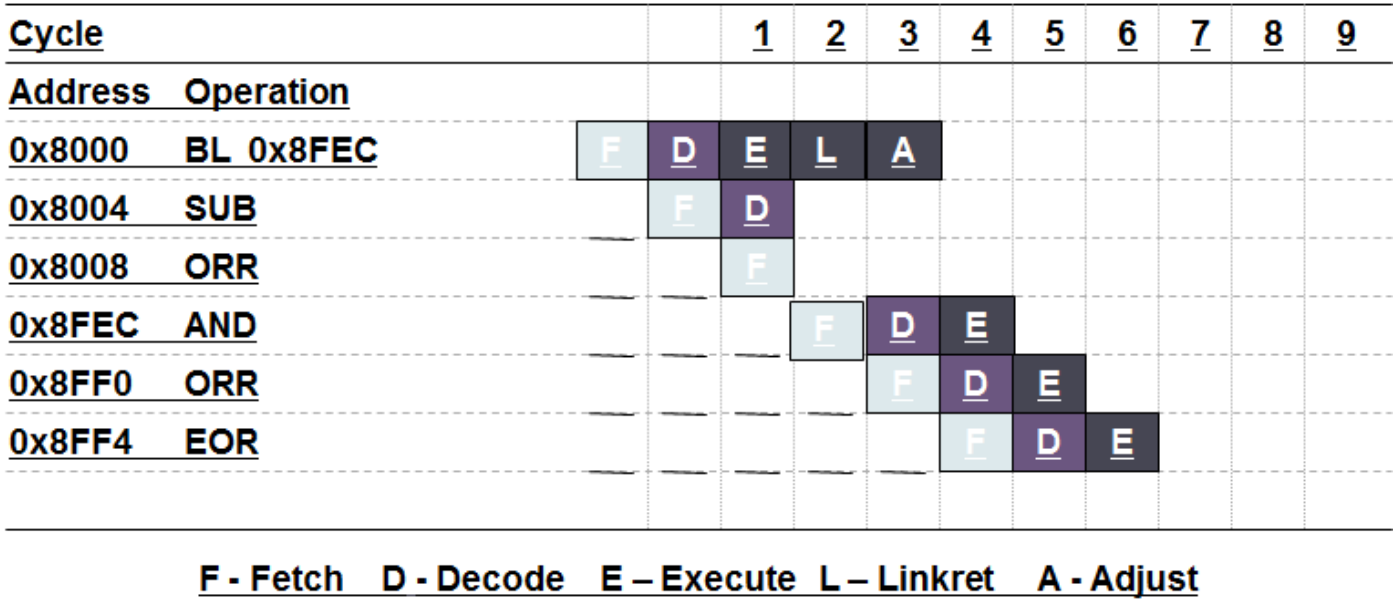
BL指令用于实现指令流的跳转，并存储返回地址到寄存器R14（LR）中

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。

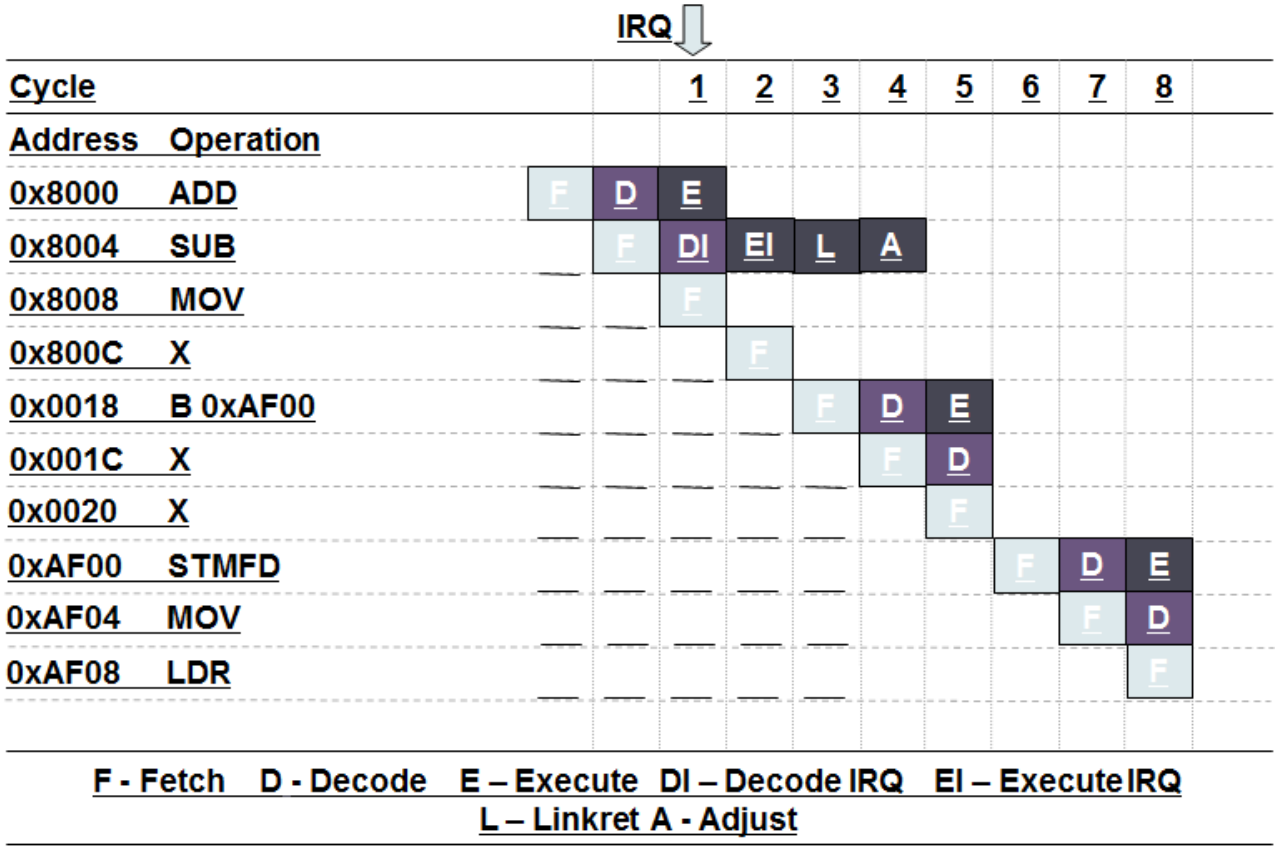
2. 周期 2: 此时总是进入ARM状态.
执行中断 (获取IR向量的地址), 保存 CPSR 于 SPSR, 改变CPSR 模式为 IRQ 模式并禁止进一步的 IRQ 中断输入。

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

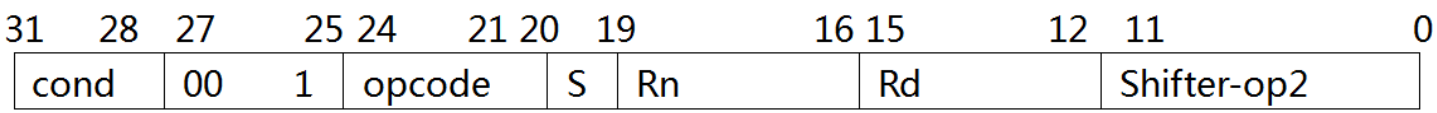
MOV机器码

```
    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



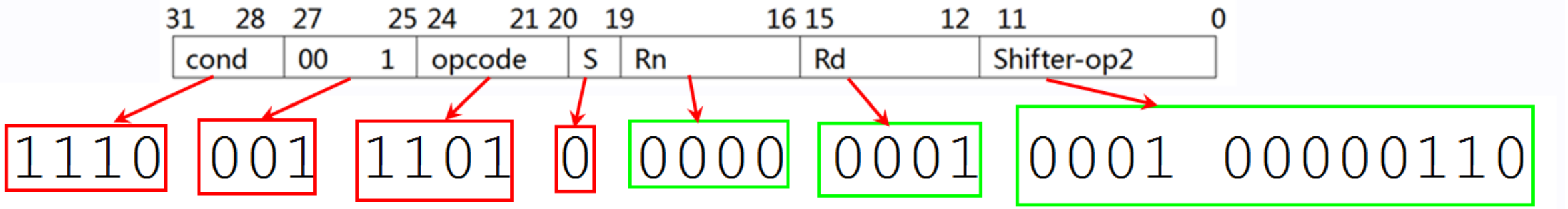
条件码	助记符后缀	标 志	含 义
0000	EQ	Z置位	相等
0001	NE	Z清零	不相等
0010	CS	C置位	无符号数大于或等于
0011	CC	C清零	无符号数小于
0100	MI	N置位	负数
0101	PL	N清零	正数或零
0110	VS	V置位	溢出
0111	VC	V清零	未溢出
1000	HI	C置位Z清零	无符号数大于
1001	LS	C清零Z置位	无符号数小于或等于
1010	GE	N等于V	带符号数大于或等于
1011	LT	N不等于V	带符号数小于
1100	GT	Z清零且（N等于V）	带符号数大于
1101	LE	Z置位或（N不等于V）	带符号数小于或等于
1110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

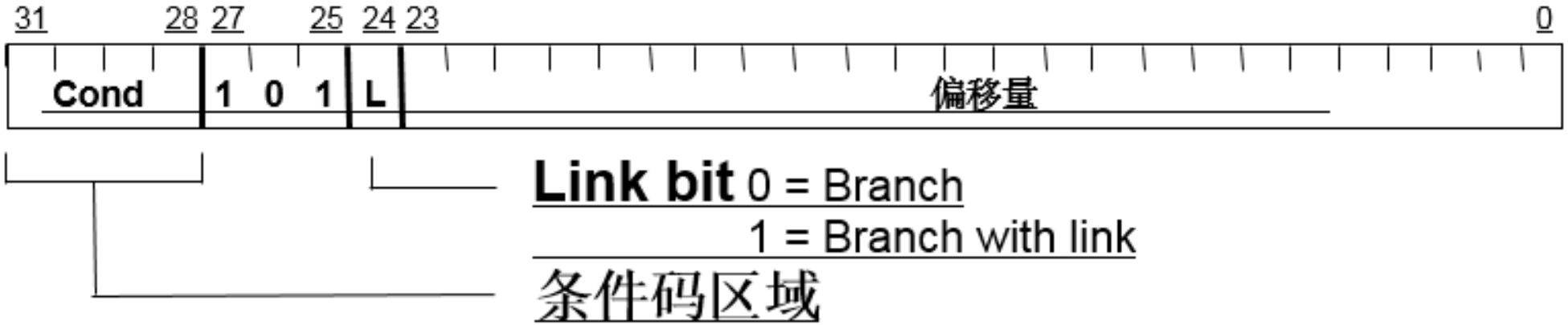
B 指令

- 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
```

ARM

15

ARM指令6
MRS、MSR

- 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

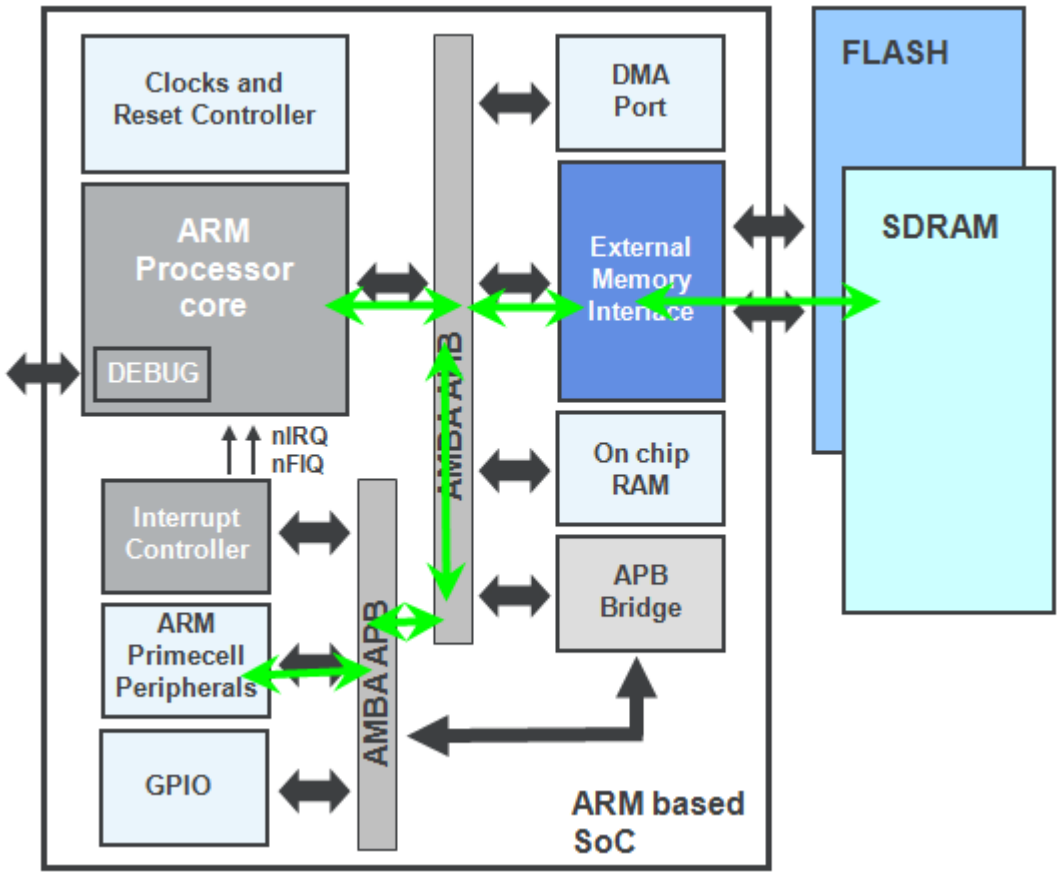
16

ARM指令7
MRS、MSR实例

ARM

17

ARM指令8
寻址方式



ARM

18

ARM指令9
寻址举例1

例1 数组求和

例2 内存数据读写

ARM

19

ARM指令10
寻址举例2

例3 数据压栈退栈

例4 函数嵌套调用

ARM

20

ARM指令11
ldrex 和 strex

想入门和进阶ARM，
请加关注一口君的公众号：一口Linux