说说 ARM Cortex-M 内核

Understanding the Core of ARM Cortex-M

邵贝贝 薛涛

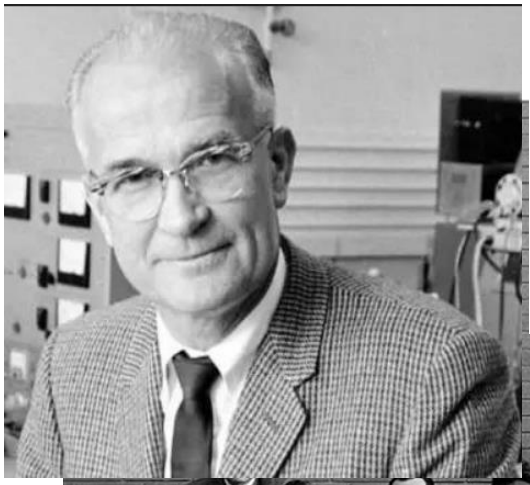
2018.8.17.

主要内容

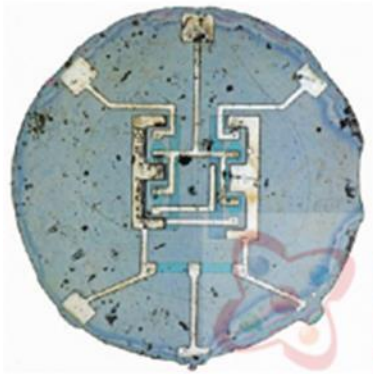
- RISC与CISC 技术
- ARM Cortex的由来与发展
- Cortex-M MCU 中的ARM内核结构
- 对Cortex-M指令系统的分析与鉴赏
- 在Gcc中写汇编的实例
- 用于Cortex-M的两个嵌入式实时操作系统

集成电路简史

- 1948年肖克利发明晶体管
- 1956诺奖成立肖克利实验室
- 1960年 诺伊斯的第一IC
  含4支三极管
- 1965年 摩尔预言
- 1971 年 Intel第一个CPU 4004



Eugene Kleiner尤金-克莱纳
Gordon Moore 戈登-摩尔
Jay Last 杰-拉斯特
Jean Hoerni 金-赫尔尼
Julius Blank 朱利亚斯-布兰科
Robert Noyce 罗伯特-诺伊斯
Sheldon Roberts 谢尔顿-罗伯茨
Victor Grinich维克多-格林尼克



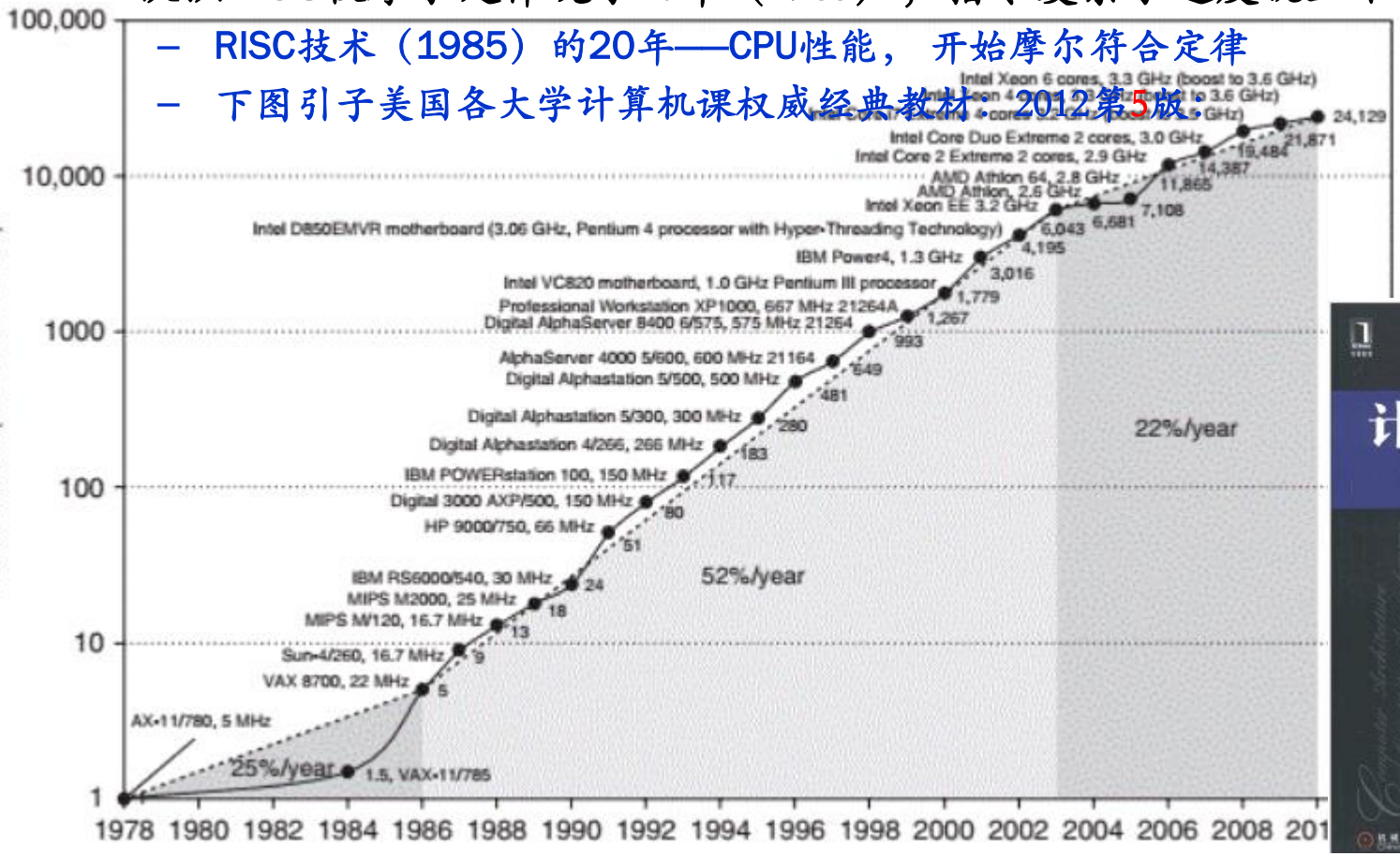
Robert Noyce与其设计的集成电路

Courtesy of Special Collections, Stanford University Libraries

仙童及8人签字的一美元钞票

摩尔定律 与 RISC异步达20年

- 摩尔定律（1965):芯片集成度和性能每隔18~24个月翻倍,价格减半
  - 1965年，片上的晶体管约60个，2010年后达到30亿个硅晶体管
- 认识RISC较摩尔定律晚了20年（1985），指令复杂了速度就上不去了
  - RISC技术（1985）的20年——CPU性能， 开始摩尔符合定律
  - 下图引子美国各大学计算机课权威经典教材：2012第5版：



斯坦 福大学校长翰尼斯(John Hennessy)和加大伯克利分校教授彼特森(David Patterson

CISC 与 RISC 技术

- CISC－复杂指令集，指令多、指令功能强，如8051,Intel x86,MC68K .....
  - 指令长度不同，执行时间不同，难实现流水线作业
  - 设计复杂、用的逻辑门数目多，故功耗高，频率难提高
  - 优点：CPU以多种存储器寻址方式直接同存储器打交道，例如，给存储器加1 (i++;)

- RISC——精简指令流计算机 如 VAX87、PowerPC、MIPs、ARM
  - 可以保证每条指令长度和执行时间相同，流水作业好安排，CPU的设计大为简化
  - 性能、功耗得到革命性改善
  - 缺点：对存储器操作只能是读或写，且只有寄存器间接寻址这一种方式，所有运算只能在内部寄存器间完成

- CISC和RISC各自都有克服其缺点的招法，
  但同样功能下，RISC设计简单，使用的晶体管门数少
  漏电流小，自然功耗就低

比较：ARM Cortex M0+:门数 1.5万门，
性能相当的80486: 门数118万门

John Cocke约翰·科克，46年Duke 机械系学士，56年数学博士，进入IBM，因RISC架构贡献，获87年图灵奖，89年IEEE 计算机先驱奖、94年冯诺依曼奖

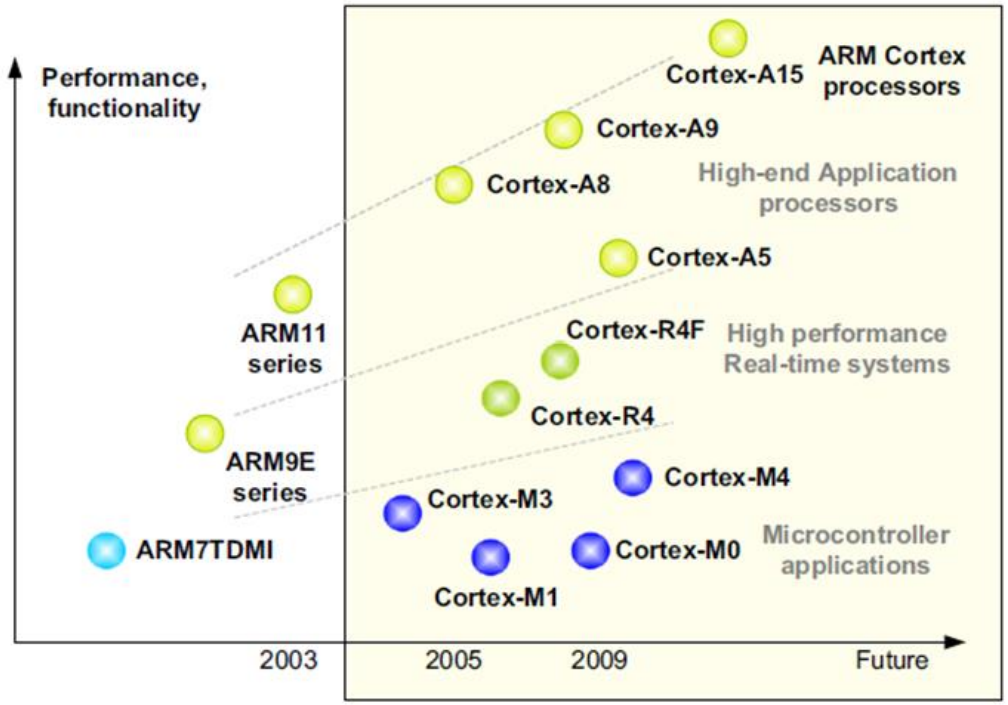
Intel 的CISC 道路

- PowerPC失败说明，Intel优势在于和最早的 8086 完全兼容，在x86及以后，仍不得不继续坚持用CISC。若转到RISC上会使其优势荡然无存

- 英特尔在维护 x86 市场优势同时，也推出过RISC的80860。不成功，市场说明兼容比性能更重要。于是叫停RISC，专心做CISC 并以其雄厚的资金和技术坚持宏大的CISC研发计划，并取得成功

- 90年代后，只有Intel一家坚持开发CISC，以色列人工程师尤里•维塞把超量化 （Superscalar） 概念用到了CISC上，用多个执行单元，在1个周期内并行地执行多条指令。超量化概念是1 968年克雷（Seymour Cray）设计CDC 6600计算机时提出的。1993年Intel完成了32b的超量化CISC指令集设计推出奔腾，2000年以后融入RISC技术，以其领先1～2代的超大规模集成电路技术，奔3、奔4相继达进入最快CPU行列，目前Intel仍占据着云计算用高端服务器阵列CPU市场，我国银河超级计算机也是Intel芯

ARM的由来与发展



ARM公司93年成立，专注设计CPU，自己不生产，出售IP，让各半导体公司设计和生产各自的MCU。大幅提高了产品开发效率，降低了开发成本。

几乎所有半导体厂商都购买了ARM内核,替代原有CPU，进入了后PC时代，开发平台统一了，不再CPU群雄割据，MCU应用大环境得到改善

2004年以后的ARM都叫Cortex

分3个系列:

A 外扩存储器 运行Linux

R Real Time (Reliability)

M MicroControler

ARM Cortex-M 2004年始于的 Cortex-M3

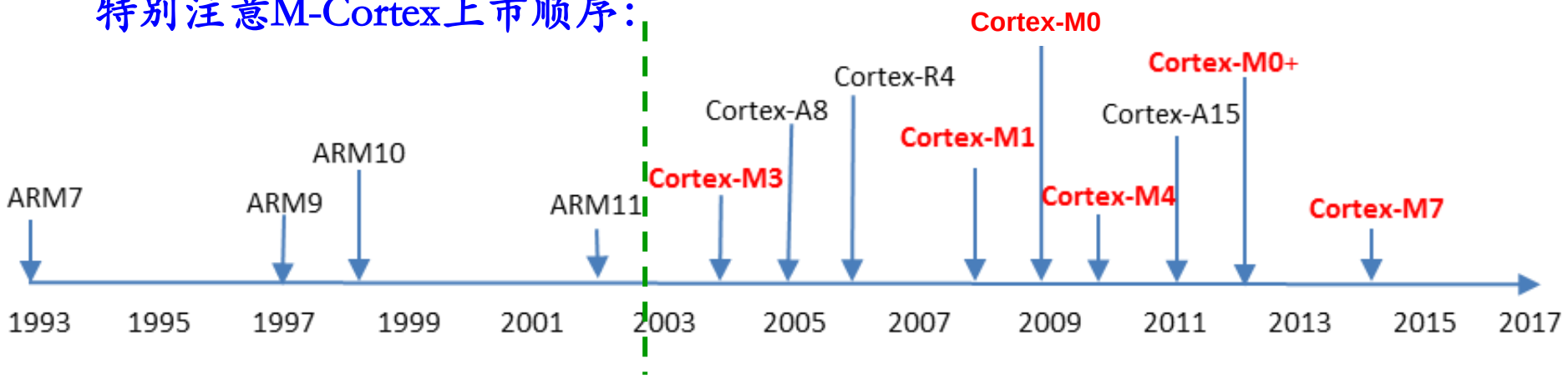
2004年ARM指令集方面有了革命性进展，以后的ARM都叫Cortex

ARM Cortex-M 由ARM11发展而来，但与从前的ARM7/9/11等并不兼容，

ARM-Cortex-M 吸取了Intel x86 兼容方面的教训，做了很好的产品链规划

做到 M0/1/0+/3/4/7……完全向上兼容

特别注意M-Cortex上市顺序：

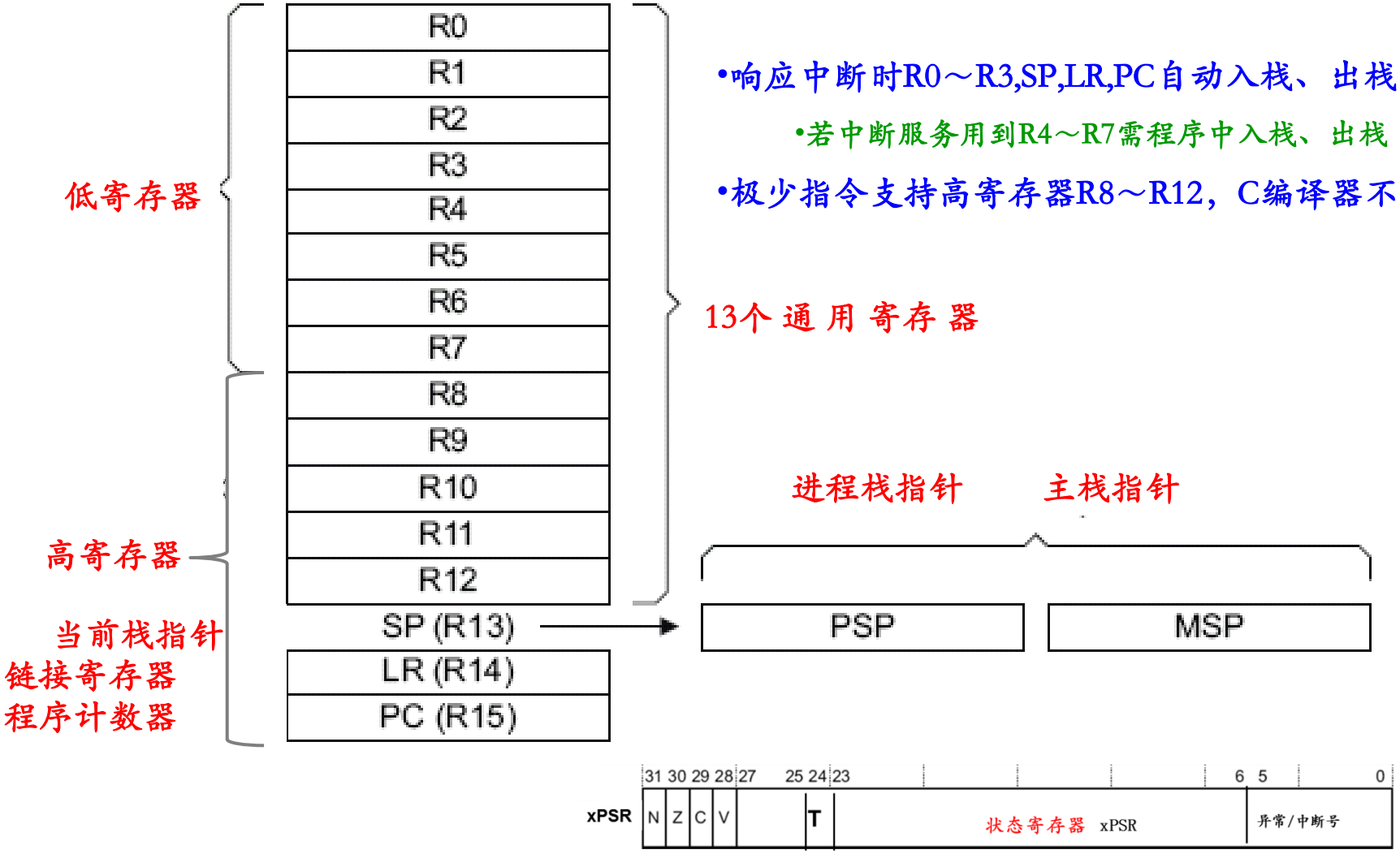


Cortex-M1， 在FPGA 上实现，是为FPGA设计的软核

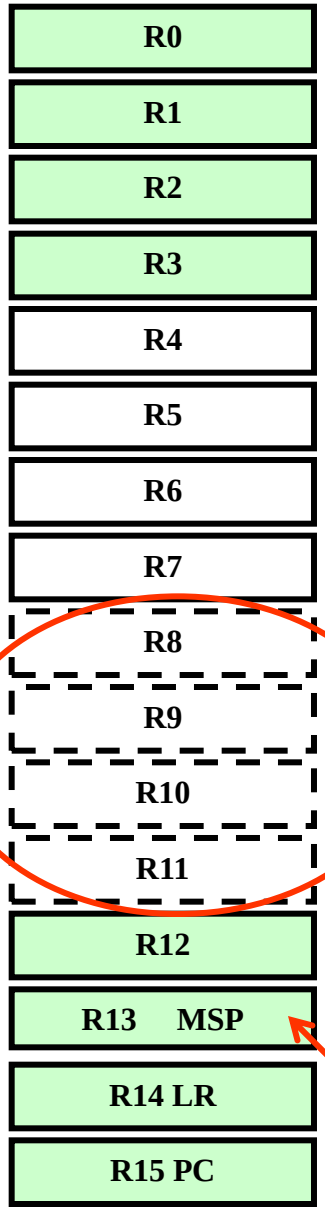
Cortex-M0， 从简化M3得到，仅使用v7的16b指令和个别32b指令，即v6子集，保证兼容

Cortex-M0+，从简化M0而来，3条流水线简化成2条，指令集不变，性能优化，更低功耗

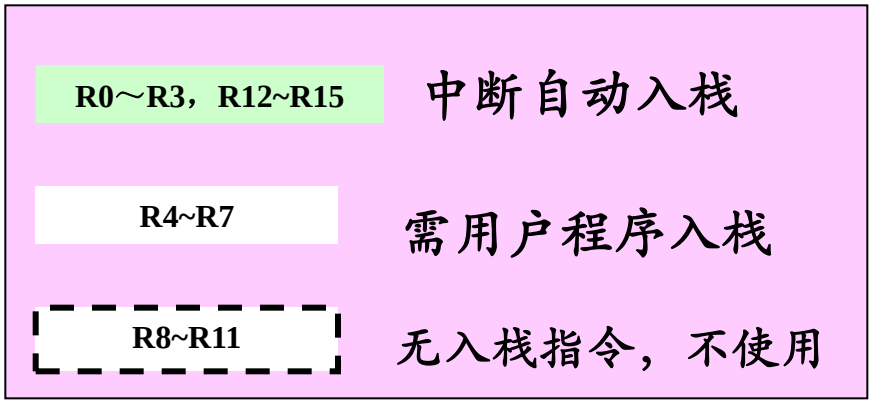
Cortex-M最基本的内部寄存器



Cortex-M 内部寄存器



低组寄存器

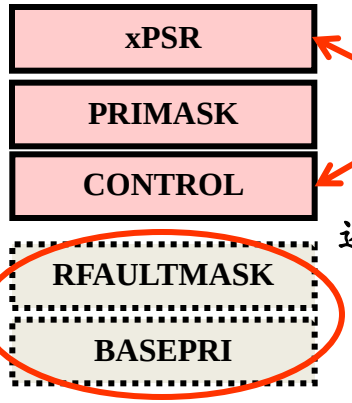


高组寄存器：
无运算、出入栈等功能
C编译器也不用

v6指令基本上只支持R0-R7, SP 或PC

R13 PSP

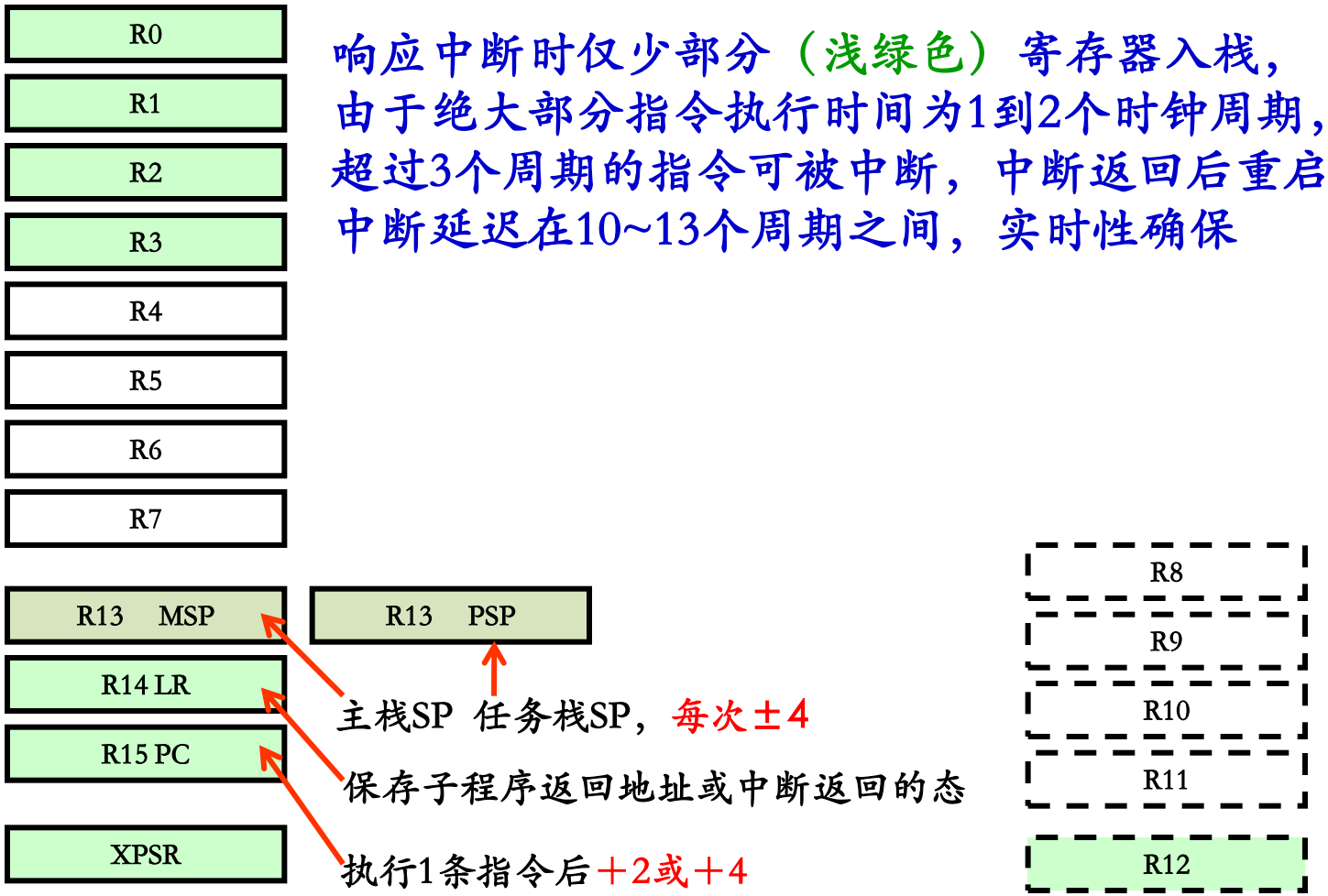
主SP 任务SP



Cortex-M0
精简掉一些有效位

还精简掉两个辅助寄存器

13个通用寄存器可按功能强弱分为3组

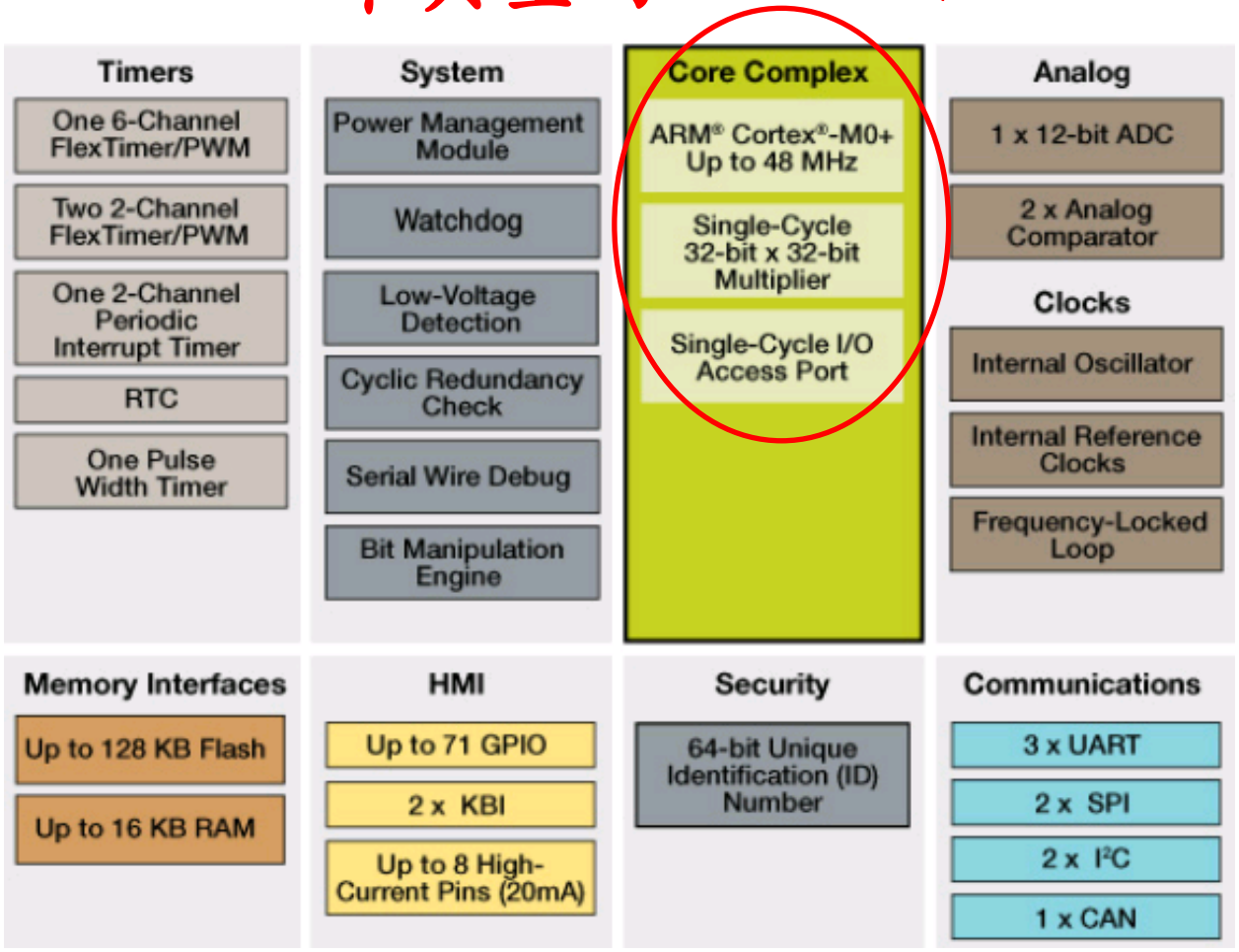


响应中断时仅少部分（浅绿色）寄存器入栈，
由于绝大部分指令执行时间为1到2个时钟周期，
超过3个周期的指令可被中断，中断返回后重启保证
中断延迟在10~13个周期之间，实时性确保

以上寄存器指令多为16位

高寄存器: v6的C编译器不用
v7可实现同等功能，需32b指令

一个典型的 MCU （Micro Controller Unit）

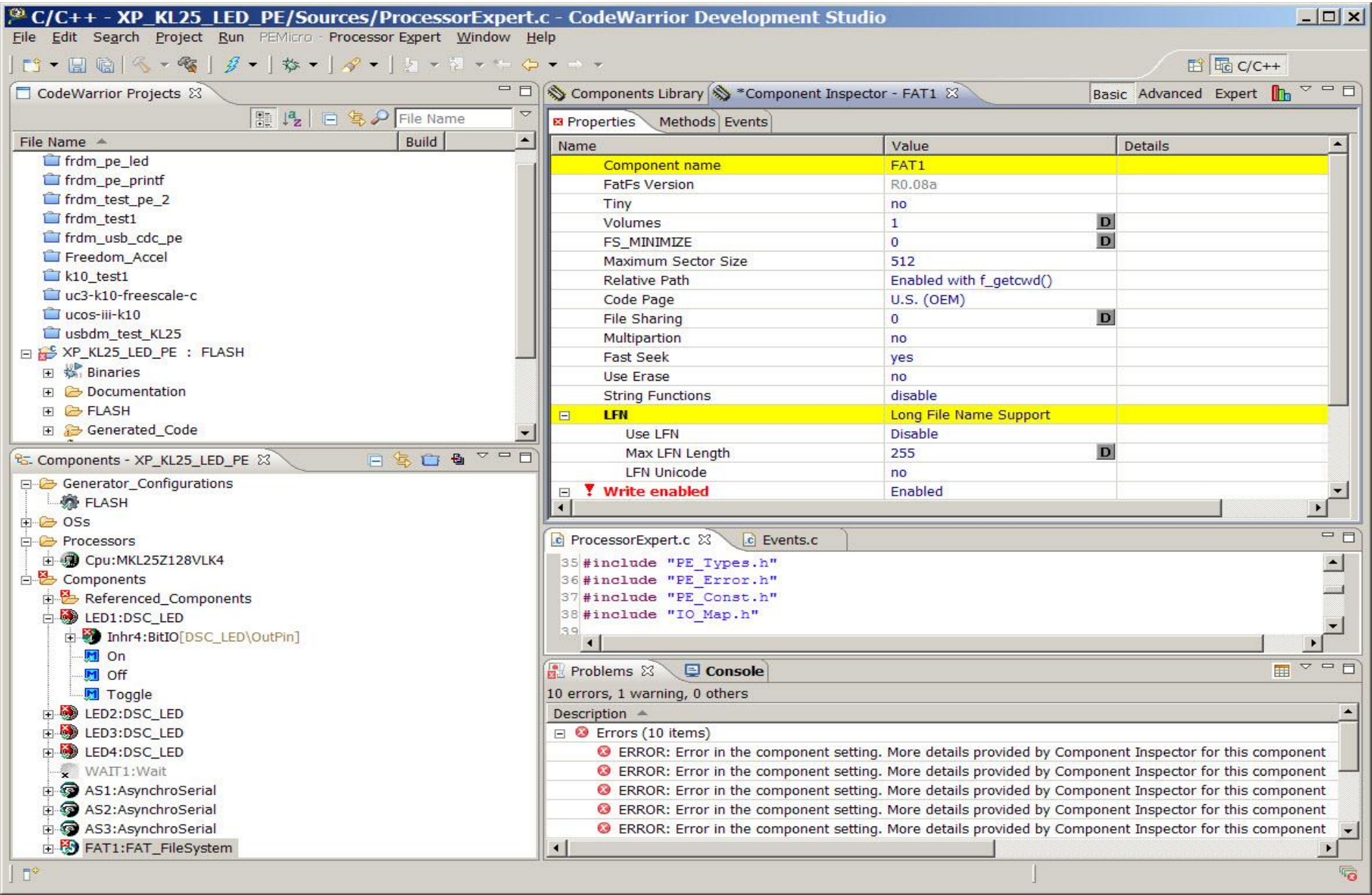


ARM 只提供内核 存储器、I/O等其它模块需MCU厂商提供

GPIO
KBI
UART
ADC
DAC
PWM
RTC
I^2C
CAN
USB
Ethernet
DMA

Cortex-M 的集成开发环境IDE

CodeWarrior(Gcc)， IAR,， MDK-ARM



大学MCU课程转向ARM后出现的问题

- MCU课程给学生毕业后的自学能力打基础，基础知识很重要
- 大学不做些硬件实验，工作后不会再有了
- 过去课程讲8051（CISC），书多，易懂，换ARM出现问题
- Cortex-M，从复杂的M3/4开始的，来势猛,较8051难
- IDE界面友好，C编程，大学课程只讲IDE用法，不涉及ARM内核及指令，成了软件课
- 教师须深入到ARM内核，以真正理解RISC，才能讲好这门课
- 写好的设备驱动程序，定会用到汇编，也应该弄懂ARM
- ARM 手册是写给芯片设计者的，不是写给用户的，难看懂，错误多。乃至对芯片设计者，仍需咨询ARM的工程师
- MCU厂家手册中不讲ARM内核，只讲I/O模块，存储器
- 至今未见一本能把ARM Cortex-M 内核说清楚的教材，可以看出，多数写ARM教材的作者也没弄懂

清华的“口袋”实验室

教学软件：CodeWarrior（Gcc）

存问题：几乎不涉及ARM内核

Kinetis K20 (ARM Cortex M4)

Kinetis KL25 (ARM Cortex M0+)

Freescale 9S12XS128

写的一个监控程序为的是弄懂ARM Cortex-M

- 监控程序是驻留在单片机最小系统中的软件部分，通过1个串行口与PC通信，用于的现场调试。
- 教学：看到寄存器、机器码

利用 IDE调试程序生成

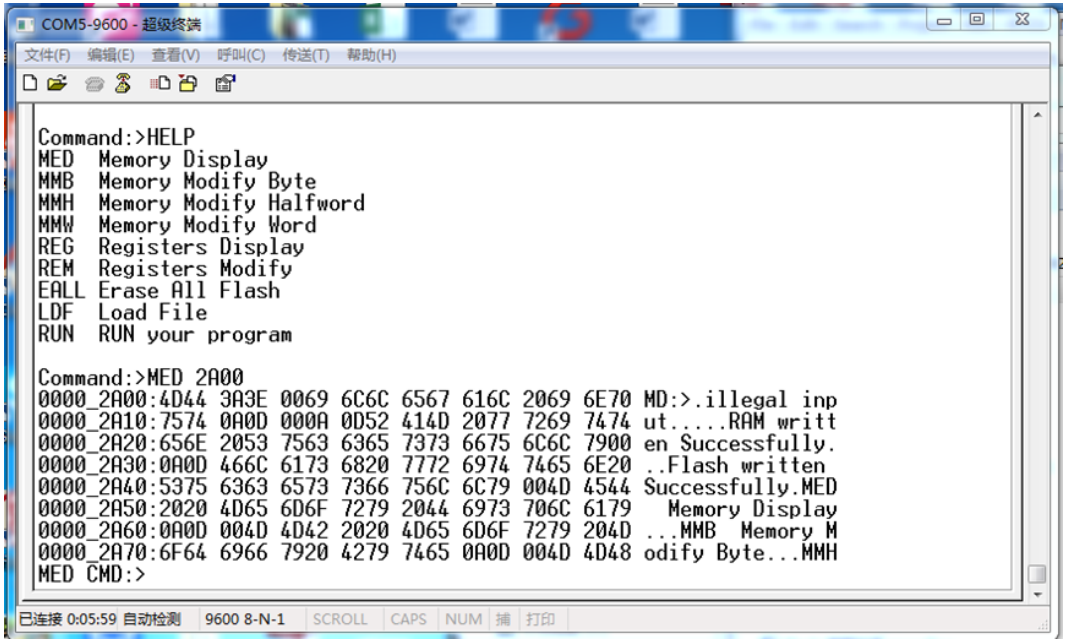
可下载代码

然后让MCU最小系统脱离调试器

独立运行

在线调试程序,处理中断，出错显示

为的是深入了解ARM 和RISC



- Boot-Loader：仅有应用程序下载、运行，Flash擦除等简单功能；
- Monitor：增加了寄存器显示/修改，中断处理，反汇编等操作命令
- Debugger：再增加设断点、单步执行、跟踪等高级调试功能

我们写的Corte-M0+监控程序

一种古老而传统的用串口调试MCU应用的方法，不使用D-Link工具

M0+ Monitor/Debug version v1.8

Tsinghua University all right reserved

Type HELP for help

Command:>HELP

MED Memory Display	显示Flash、RAM、I/O寄存器内容
DSA Disassemble Program	反汇编程序代码
MMB Memory Modify Byte	按字节(8b)修改RAM 、I/O寄存器内容
MMH Memory Modify Half word	按半字（16b)修改RAM 、I/O寄存器内容
MMW Memory Modify Word	按字（32b）修改RAM 、I/O寄存器内容
REG Registers Display	显示CPU内部寄存器的内容用于调试
REM Registers Modify	修改CPU内部寄存器的内容
EALL Erase All Flash	擦除监控程序以外的全部Flash
LDF Load File	向Flash/RAM下载程序
RUN Run your program	运行用户程序

- 涉及内部寄存器的程序只能用汇编写，通过写汇编深入理解ARM指令集
- 程序不优化: ～25KB，优化后:～12KB ，优化后的程序不再符合ANSI 规范

Gcc涉及内部寄存器的操作——插入.S汇编文件

```
/* This   .S  file uses GNU syntax */
                .text
                .align          2
                .global         myadd
                .type myadd function
/* Function myadd
  Returns the sum of two integers:
  Inputs:
                r0 - First operand
                r1 - Seconds operand
  Return value:
                r0 - sum of the two input operands
 */
myadd:
                add r0, r1, r0

                bx lr
```

```
/* Function register_copy  description:no LR  */

register_to_mirror:
                PUSH {R0-R7,LR}          /*PUSH R0,R1*/
                bl re_address      /*get the address of mirror_Rx*/

                MRS R7, MSP
                LDR R1,[R7]    /*copy R0 to r1*/
                STR R1, [R0]   /*store R0 to mirror_R0*/

                ADD R7,#4
                ADD R0,#4
                LDR R1,[R7]    /*copy R1 to r1*/
                STR R1, [R0]   /*store R1 to mirror_R1*/

                ADD R7,#4
                ADD R0,#4
                LDR R1,[R7]    /*copy R2 to r1*/
                STR R1, [R0]   /*store R2 to mirror_R2*/

                ..................
```

实现反汇编功能 (Command_DSM_Sub_Function.h文件)

```
//                                                  5b操作码
'LSLS',0x00000000 + 0xf800 + FORMAT5_Rd_Rm_imm5, //0 0 0 0 0 imm5 Rm Rd  LSLS  <Rd>,<Rm>,#<imm5>   b10:b0 = imm5<<6 | (Rm-R0)<<3 | (Rd-R0)(#0=MOVS <Rd>,<Rm>)
'LSRS',0x08000000 + 0xf800 + FORMAT5_Rd_Rm_imm5,//0 0 0 0 1 imm5 Rm Rd  LSRS  <Rd>,<Rm>,#<imm5>  b10:b0 = imm5<<6 | (Rm-R0)<<3 | (Rd-R0)(#0=MOVS <Rd>,<Rm>)
'ASRS',0x10000000 + 0xf800 + FORMAT5_Rd_Rm_imm5,//0 0 0 1 0 imm5 Rm Rd  ASRS  <Rd>,<Rm>,#<imm5>  b10:b0 = imm5<<6 | (Rm-R0)<<3 | (Rd-R0)(#0=MOVS <Rd>,<Rm>)
'STR ',0x60000000 + 0xf800 + FORMAT5_Rt_Rn_imm5_point,              //0 1 1 0 0 imm5,Rn,Rt   STR     <Rt>,[<Rn>,#<imm5>]  b10:b0 = imm5<<6 | (Rn-R0)<<3 | (Rt-R0)
'LDR ',0x68000000 + 0xf800 + FORMAT5_Rt_Rn_imm5_point, //0 1 1 0 1 imm5 Rn Rt    LDR     <Rt>,[<Rn>,#<imm5>]  b10:b0 = imm5<<6 | (Rn-R0)<<3 | (Rt-R0)
'STRB',0x70000000 + 0xf800 + FORMAT5_Rt_Rn_imm5_point, //0 1 1 1 0 imm5,Rn,Rt  STRB    <Rt>,[<Rn>,#<imm5>]  b10:b0 = imm5<<6 | (Rn-R0)<<3 | (Rt-R0)
'LDRH',0x88000000 + 0xf800 + FORMAT5_Rt_Rn_imm5_point, //1 0 0 0 1 imm5 Rn Rt  LDRH    <Rt>,[<Rn>,#<imm5>] b10:b0 = imm5<<6 | (Rn-R0)<<3 | (Rt-R0)
'MOVS',0x20000000 + 0xf800 + FORMAT5_Rd_imm8,    //0 0 1 0 0 Rd imm8       MOVS  <Rd>,#<imm8>  b10:b0 = Rd-R0<<8 | imm8
'CMP ',0x28000000 + 0xf800 + FORMAT5_Rd_imm8,    //0 0 1 0 1 Rn imm8       CMP    <Rn>,#<imm8>  b10:b0 = Rd-R0<<8 | imm8
'ADDS',0x30000000 + 0xf800 + FORMAT5_Rd_imm8,    //0 0 1 1 0 Rn imm8       ADD     <Rdn>,#<imm8>  b10:b0 = Rd-R0<<8 | imm8
'SUBS',0x38000000 + 0xf800 + FORMAT5_Rd_imm8,    //0 0 1 1 1 Rdn imm8     SUBS    <Rdn>,#<imm8>  b10:b0 = Rd-R0<<8 | imm8
'ADD ',0xa8000000 + 0xf800 + FORMAT5_Rd_SP_imm8,//1 0 1 0 1 Rd imm8       ADD     <Rd>,SP,#imm8   b10:b8 = Rd-R0<<8 | imm8
'STR ',0x90000000 + 0xf800 + FORMAT5_Rd_SP_imm8_point,//1 0 0 1 0 Rt imm8  STR     <Rt>,[SP,#<imm8>] b10:b0 = Rt-R0<<8 | imm8
'LDR ',0x98000000 + 0xf800 + FORMAT5_Rd_SP_imm8_point, //1 0 0 1 1 Rt imm8 LDR    <Rt>,[SP,#<imm8>]  b10:b0 = Rt-R0<<8 | imm8
'LDR ',0x48000000 + 0xf800 + FORMAT5_Rd_PC_imm8_point, //0 1 0 0 1 Rt imm8 LDR    <Rt>,[PC,#<imm8>]  注意，显示的IMM8是实际数值乘以4显示，比如1显示 #4
'ADR ',0xa0000000 + 0xf800 + FORMAT5_Rd_PC_imm8_point, //1 0 1 0 0 Rd imm8 ADR    <Rd>,[PC,#<imm8>]  b10:b0 = Rd-R0<<8 | imm8  =ADD <Rd>, PC, #<const>
'STM ',0xc0000000 + 0xF800 + FORMAT5_RdN_registers,       //1 1 0 0 0 Rn Red_list  STM  <Rn>!,{registers}
'LDM ',0xc8000000 + 0xf800 + FORMAT5_RdN_registers,       //1 1 0 0 1 Rn,Reg_list  LDM   <Rn>!,{registers}   无！ 时，Rn不加
'B   ',0xe0000000 + 0xf800 + FORMAT5_imm11,         //1 1 1 0 0 imm11               B      <label>     For DSM, if 1 1 1 1 0...., goto 32b operand

//                                                  7b操作码
'ADDS',0x18000000 + 0xfe00 + FORMAT7_Rm_Rn_Rd, //0 0 0 1 1 0 0 Rm Rn Rd  ADDS  <Rd>,<Rn>,<Rm>  Rm-R0<<6 | Rn-R0<<3 | Rd-R0
'SUBS',0x1a000000 + 0xfe00 + FORMAT7_Rm_Rn_Rd,  //0 0 0 1 1 0 1 Rm Rn Rd  SUBS  <Rd>,<Rn>,<Rm>  Rm-R0<<6 | Rn-R0<<3 | Rd-R0
'ADDS',0x1c000000 + 0xfe00 + FORMAT7_Rm_Rn_imm3,                 //0 0 0 1 1 1 0 imm3 Rn Rd ADDS  <Rd>,<Rn>,#<imm3>   imm3 <<6 | Rn-R0<<3 | Rd-R0
'SUBS',0x1d000000 + 0xfe00 + FORMAT7_Rm_Rn_imm3,                 //0 0 0 1 1 1 1 imm3 Rn Rd SUBS  <Rd>,<Rn>,#<imm3>   imm3 <<6 | Rn-R0<<3 | Rd-R0
.....
//                                                  9b 操作码
'ADD ',0x42000000 + 0xff80 + FORMAT9_Rm_SP_Rn,   //0 1 0 0 0 1 0 0 0DnRm Rdn  ADD   <Rdm>,SP,<Rm>
'ADD ',0x44800000 + 0xff80 + FORMAT9_SP_Rn,      //0 1 0 0 0 1 0 0 1DnRm Rdn  ADD   SP,<Rm>
'ADD ',0xb0000000 + 0xff80 + FORMAT9_SP_imm7,    //1 0 1 1 0 0 0 0 0 imm7 ADD        SP,#<imm7>
'SUB ',0xb0800000 + 0xff80 + FORMAT9_SP_imm7,    //1 0 1 1 0 0 0 0 1 imm7 SUB        SP,#<imm7>
'BX  ',0x47000000 + 0xff80 + FORMAT9_Rm,         //0 1 0 0 0 1 1 1 0 Rm (0)(0)(0)    BX  <Rm>
'BLX ',0x47800000 + 0xff80 + FORMAT9_Rm,         //0 1 0 0 0 1 1 1 1 Rm (0)(0)(0)    BLX                <Rm>
.....
```

表中每条指令占8B
看看v6到底有多少条指令?
手册上有什么错误?

列表看出，ARM v6 共有98个指令码，考虑到有借助指令代码相同，实际96条指令

以Gcc 的 C编译生成的汇编验证（.lst文件）

C程序：

```
if (counter > 10)
   Counter=0
      else  Counter=counter + 1
```

设Counter 在R0中，生成汇编：

```
;  对C编译器的说明
;若Counter是局部变量，该变量应该在栈中
;  R7=SP,是栈标志，用作局部变量指针
;  注解、语句标号是看懂后加上去的
```

```
•            24     if (counter > 10)
•     0000096a:   ldr r3,[r7,#4]          ;加载counter的数值到r3
•     0000096c:   cmp r3,#10              ;将r3中counter的数值与10进行比较
•     0000096e:   ble main+0x16 (0x976)   ;如果小于或者等于，执行
      0x00000976的程序
•            25     counter=0;           ;如果大于，则执行此程序
•     00000970:   movs r3,#0              ;counter赋值为0
•     00000972:   str r3,[r7,#4]          ;将counter数值保存到r3寄存器中
•            26     else  counter=counter + 1;
•     00000976:   ldr r3,[r7,#4]          ;将counter的数值加载到r3当中
•     00000978:   adds r3,#1              ;counter数值加1
•     0000097a:   str r3,[r7,#4]          ;将counter数存储到内存中
```

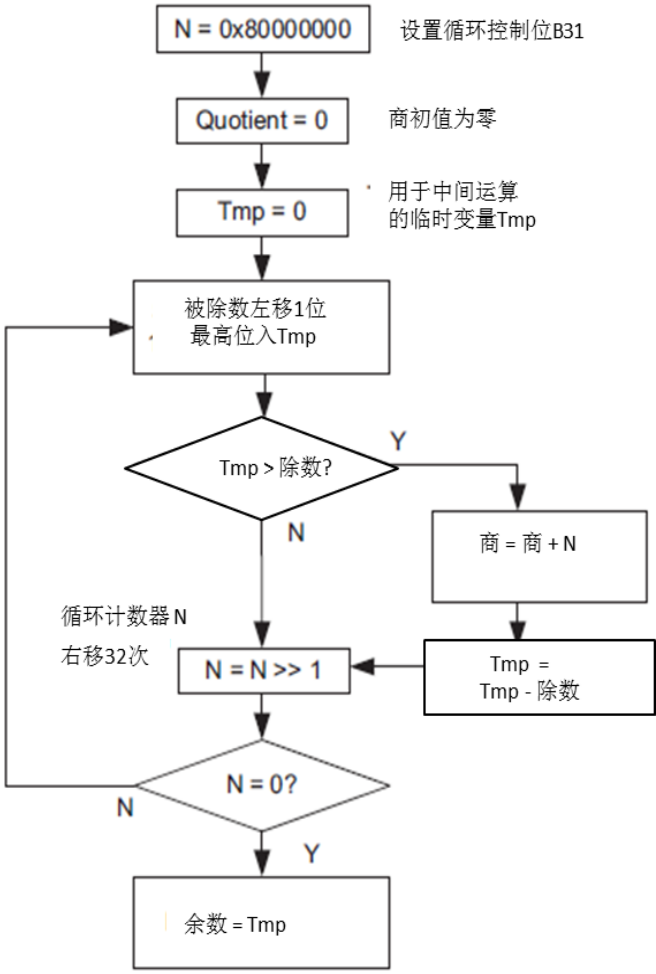
C生成的汇编程序 .lst 文件

```
Total = 0;
for (i = 0; i < 5; i = i + 1)
    Total = Total + i;
```

```
29                      Total = 0;
00000966:  movs r3,#0                       ;将r3寄存器置为0
00000968:  str r3,[r7,#4]                   ; Total=0
31                      for (i = 0; i < 5; i = i + 1)
0000096a:  movs r3,#0                       ;r3=0
0000096c:  str r3,[r7,#0]                   ;将局部变量i初始化为0
0000096e:  b main+0x18 (0x97e)              ;跳转到 0x0000097e 对应的程序中
33                      Total = Total + i;
00000970:  ldr r2,[r7,#4]                   ;取局部变量total到r2中
00000972:  ldr r3,[r7,#0]                   ;将取局部变量i到r3中
00000974:  adds r3,r2,r3                    ;将两数据相加，结果在r3中
00000976:  str r3,[r7,#4]                   ;Total=r3
31                      for (i = 0; i < 5; i = i + 1)
00000978:  ldr r3,[r7,#0]                   ;r3= i
0000097a:  adds r3,#1                       ;将r3的数值加1
0000097c:  str r3,[r7,#0]                   ;将r3数值存储到i对应的存储空间当中
0000097e:  ldr r3,[r7,#0]                   ;将i的数值加载到r3当中
00000980:  cmp r3,#4                        ;将r3的数值与4进行对比
00000982:  ble main+0x10 (0x970)            ;如果小于或者等于，跳转到 0x00000970内存
```

除法子程序

```
; Inputs:  R0:被除数 dividend,  R1: 除数 divider;
; Outputs:  R0 商quotient,  R1 余数remainder;
simple_divide:
PUSH    {R2-R4}          ; 入栈保护要用到的寄存器
MOV     R2, R0           ;保存R0中的被除数, R0将用于得到商
MOVS    R3, #0x1         ;循环次数控制
LSLS    R3, #31          ; b31=1，N = 0x80000000
MOVS    R0, #0           ;初始化成零
MOVS    R4, #0           ;中间变量寄存器Tmp初始化为零
simple_divide_loop:
LSLS    R2, #1           ;从被除数最高位做起，最高位入C标
ADCS    R4,R4            ;左移1位、最低位=进位标志C
CMP     R4, R1           ;可否做一次减法?
BCC     simple_divide_lessthan
ADDS    R0, R3           ; 得到32位商值的其中1位
SUBS    R4, R1           ; 做一次减法
simple_divide_lessthan:
LSRS    R3, #1           ; N = N >> 1，循环1次
BNE     simple_divide_loop
MOV     R1, R4           ; 保存余数到R1中
POP     {R2-R4}          ; 恢复曾经入栈的寄存器
BX      LR               ; 子程序返回
```



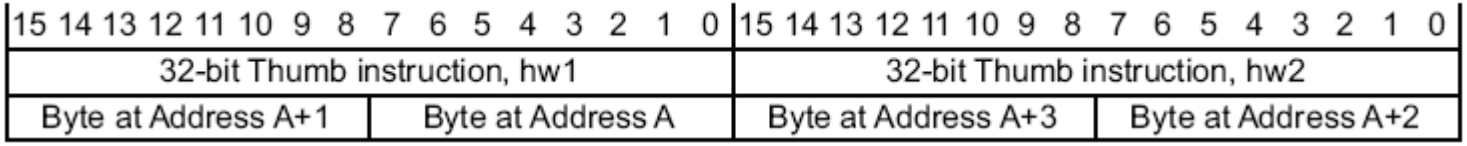
汇编写的开方程序

```
; Input : R0  Output : R0 (square root result)
        PUSH {R1-R3}            ; Save registers to stack
        MOVS R1, #0x1           ; Set loop control register
        LSLS R1, R1, #15        ; R1 = 0x00008000
        MOVS R2, #0             ; Initialize result
Loop    ADDS R2, R2, R1         ; M = (M t N)
        MOVS R3, R2             ; Copy (M + N) to R3
        MULS R3, R3, R3         ; R3 = (M + N)^2
        CMP R3, R0
        BLS lesseq
        SUBS R2, R2, R1         ; M = (M  - N)
Lesseq  LSRS R1, R1, #1         ; N = N >> 1
        BNE Loop
        MOV R0, R2              ; Copy to R0 and return
        POP {R1-R3} ;
        BX LR                   ; Return
```

汇编管理指令（伪指令）

```
                .text
                .align   2
                .global  myadd
                .type    myadd function
Word            DCD      0x12345678
Half word       DCW      0x1234
Byte            DCB      0x12
String          DCB      “Hello\n”, 0
Instrc          DCI      0xBE00 ; Breakpoint-BKPT 0
```

指令对齐与字节顺序（32b 的高低位问题）



写成的监控程序

```
Command:>
ARM Cortex-M Debug v1.0
Tsinghua University all right reser
Type HELP for help

Command:> HELP
MED   Memory Display
MMB   Memory Modify Byte
MMH   Memory Modify Halfword
MMW   Memory Modify Word
REG   Registers Display
DSM   Disassemble Program
EALL  Erase All Flash
LDF   Load File
RUN   RUN your program

Command:>
```

已连接 0:02:05 自动检测 | 9600 8-N-1 | SCROLL | CAPS | N

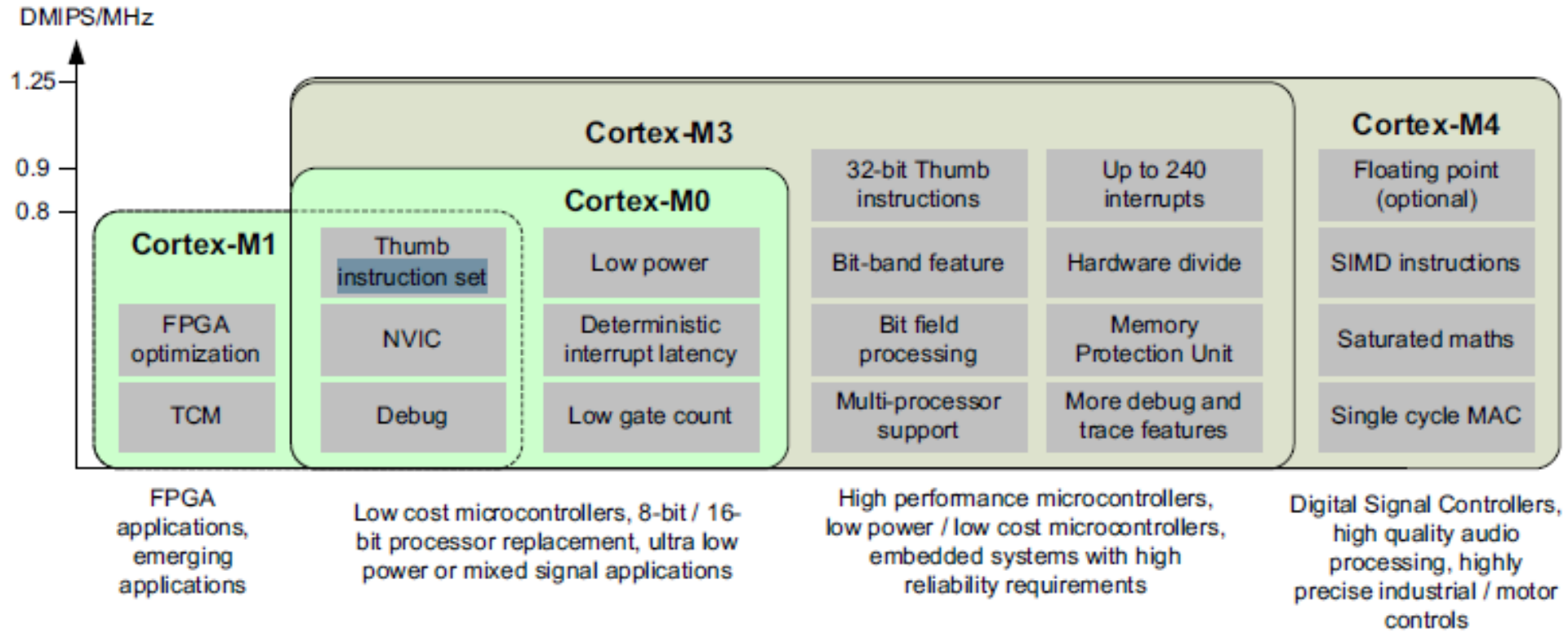
```
Command:>
ARM Cortex-M Debug v1.0
Tsinghua University all right reserved
Type HELP for help

Command:>DSM 00000800
00000800 0x1808  ADDS R0,R1,R0
00000802 0x4770  BX   Rmd
00000804 0xB5FF  PUSH R0x00FFR
00000806 0xF000  BL 0x00000DD8
00000808 0xFAE7
0000080A 0xF3EF  MRS  R3,MSP
0000080C 0x8708
0000080E 0x6839  LDR  R1,[R7,#0x00]
00000810 0x6001  STR  R1,[R0,#0x00]
00000812 0x3704  ADDS R7,#0x0004
00000814 0x3004  ADDS R0,#0x0004
00000816 0x6839  LDR  R1,[R7,#0x00]

Command:>DSM 00003400
00003400 0xFFD4  DCW  0xFFD4
00003402 0xB280  UXTH R0,R0
00003404 0x5620  LDSB R0,[R4,R0]
00003406 0x5245  DCW  0x5245
00003408 0xFFD4  DCW  0xFFD4
0000340A 0xB2C0  REV  R0,R0
0000340C 0x5620  LDSB R0,[R4,R0]
0000340E 0x5245  DCW  0x5245
```

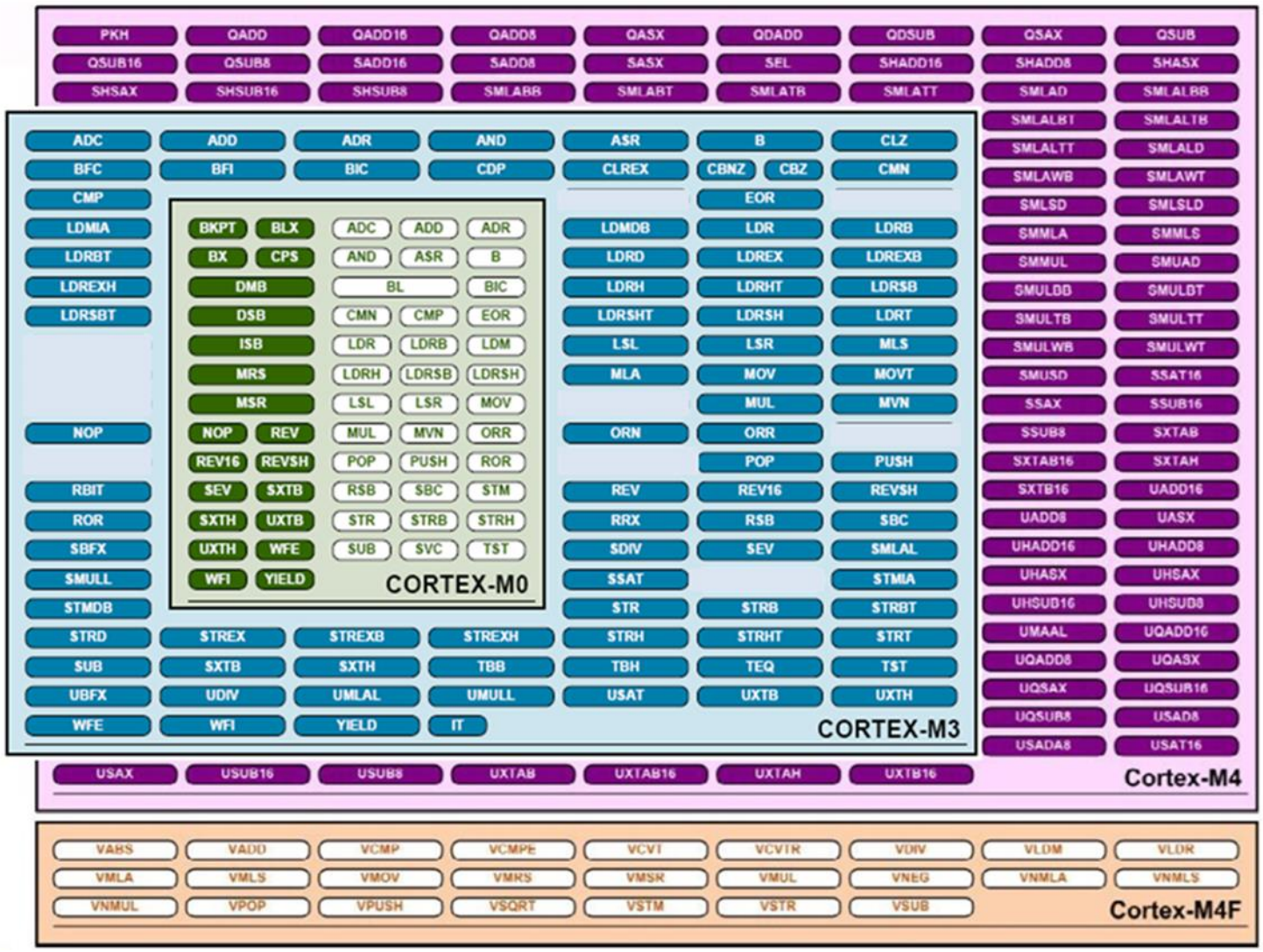
9600 8-N-1

Cortex-M1/0/0+/3/4/7之间的兼容关系



M0+和M0指令相同，主要是将3级流水线简化为2级，降低了功耗，还进一步简化了中断，性能功耗比提高到1.77倍，戏称M0-,中断处理过程更容易讲清楚

Cortex-M0/M3/M4指令兼容



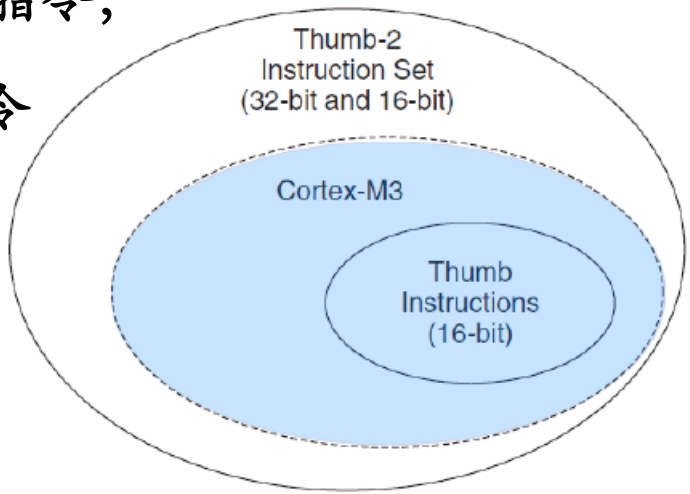
RISC运算在内部寄存器间完成，快，多数指令在1个时钟周期内完成。读写存储器慢，需要2个周期，但2个周期可读入2条指令16b指令，平均每周期读入1条指令，由于有2~3条流水线，平均1周期1指令

ARM和Thumb 指令集的区别

- ARM有2种运行态，分别支持ARM32b 和Thumb16b 2个指令集
  - 靠xPSR中T位切换运行状态
- ARM和Thumb指令的区别：
  - ARM指令是 条件执行指令,条件在高4b中
  - Thumb指令是无条件执行指令，更改指令集得改写T位来切换
- ARM指令均为32位，高4位是执行条件：
  - 前14种　有转移条件
  - 1110　　BAL　　Branch Allways　无条件转移
  - 1111　　BNV　　Branch Never　　不转移，保留将来作它用
- Cortex-M使用的Thumb2 指令集
  - Thumb2 在Thumb态下可执行32b指令
  - 16/32b混用，无需状态切换
  - Cortex-M3使用ARM v7指令集
  - Cortex-M1/0+使用精简掉大部分32b指令后的ARMv6指令集(v7的子集)

ARM/Thumb 指令集与Thumb2指令集的区别

- Cortex-M 使用Thmb-2指令集
  - 16/32b指令混行，都是无条件执行指令
  - 将条件执行指令集中高4b条件码用做16位指令码得到humb-2用于Cortex-M3。后来的M4扩展了M3。M7兼容M4
  - 提取ARMv7的16b指令和几条32b指令，生成ARMv6指令子集，用于Cotex-M1和 后来的M0，M0+，兼容M0，v6是v7的一个子集
  - 16/32b指令混合执行按16位对齐，32b指令可拼接
  - M4/7增加了选自ARMv7-A/-R系列指令集的指令，
  - 含饱和运算、DSP的乘加指令、协处理器指令



v6 子集的 56 条指令是v7中的指令精华

一些指令不会用到，用到的汇编指令不足40条，易讲清楚

算术逻辑移位：	ASR	LSL	LSR	ROR			
逻辑运算：	AND	ORR	EOR	BIC	TST	MVN	RSB$_{NEG}$
算术运算：	ADD	ADC	ADR	SUB	SBC	MUL	
比较，转移：	CMP	CMN	B	BL			
1/2/4B数据类型转换：		REV	XT				
多寄存器读/写：	LDM	STM	PUSH	POP			
1/2/4B存储器读/写：		LDR	STR				
子程序调用返回：	BLX	BX					
中断相关：	CPS	SVC	WFI	BKPT			
特殊寄存器读写：	MRS	MSR					

Thumb						Thumb-2	
ADC	ADD	ADR	AND	ASR	B		
BIC	BL		BX	CMN	CMP	SEV	WFE
EOR	LDM	LDR	LDRB	LDRH	LDRSB	WFI	YIELD
SDRSH	LSL	LSR	MOV	MUL	MVN	DMB	
ORR	POP	PUSH	ROR	RSB	SBC	DSB	
STM	STR	STRB	STRH	SUB	SVC	ISB	
TST	BKPT	BLX	CPS	REV	REV16	MRS	
REVSH	SXTB	SXTH	UXTB	UXTH		MSR	

借用操作码指令：	MOV	NOP				
不用的指令：	(SEV)	(WFE)	(YIELD)	(ISB)	(DMB)	(DSB)

Cortex-M0/M0+用到的指令仅37条，含3条32b 指令

Thumb2-v6 指令按执行速度分类

- 数据传送指令： MOV ADR
- 算数运算指令: LADD ADC SUB SBC RSB CMP CMN MUL
- 逻辑运算指令： AND ORR EOR BIC MVN TST 寄存器间操作——单周期类
- 左右移位指令 : ASR LSL LSR ROR REV
- 位扩展、字节顺序交换指令: SXTB SXTH UXT UXTH
- 栈操作指令: PUSH POP 存储器读、写——双周期类
- 存储器的读写指令 LDR STR LDM STM
- 程序流控制（跳转、子程序调用）指令 B BL BX BLX
- 异常处理相关指令CPS SVC 3周期类
- 特殊寄存器读写指令：MRS MSR与内存隔离指令DSB DMB ISB
- 示意指令NOP WFI BKPT（WFE）(SEV) （YIELD）

多周期指令，如乘法，批处理指令等，可被中断，返回后重新执行，可视为3周期

借来的指令：NOP 借自 MOV R8,R8 ， MOV是LSL的特例

时钟周期——指令读入与执行

- 读取指令和执行指令分别由3（2）级流水线完成
- 2个时钟周期可从存储器读取1个32位编码，2条16位指令
- 2个周期亦可执行2条单16b指令
- 平均1个时钟周期执行1条指令，16b指令优势明显

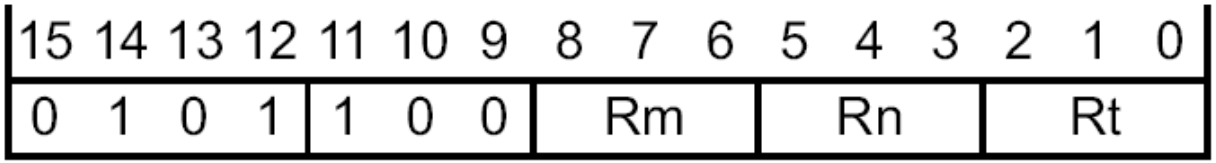
指令由操作码和操作数组成

语句标号	操作码	操作数1、2、3	；注解
Subtract	SUBS	<Rd>, <Rn>, <Rm>	； Rd =Rn-Rm

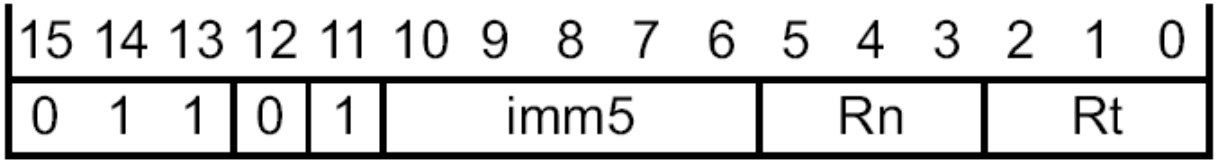
- 指令（操作码）、目标寄存器（Rd）、源寄存器Rn、Rm:
- 在16位宽的限制下，安排开56条指令和2～3个操作数需精心设计
  - 如果Rd、Rn、Rm都对16个寄存器有效，指令就只能有16条了
  - 即便3个操作数只面向低寄存器，也需要9b编码
  - 涉及PC、SP寄存器的，也需要1b予以区分
  - Thumb2-V6看似56条指令，实际上有96个指令码，平均指令码长度6.5b

16b 指令构成

- RISC要求指令对齐，操作数必须包含在指令中。即便3个操作数都仅限于8个低寄存器，最多可安排128条指令。这还没有包括非常重要的PC、SP寄存器



- 对于极为重要的移位指令、位操作指令，指令得夹带5b(0~31)操作数，此时就只能有5b操作码（仅能安排32条指令了）



资源紧张，指令集必须精心设计!

指令的前6位编码——操作码

操作码	指令或指令子集
00xxxx	立即数相关运算指令 如 Shift , add, sub, compare
01000x	寄存器相关运算指令
01001x	PC寻址的存储器读写
0101xx	存储器读写（LoaD:读, STore:写）
011xxx	存储器读写
100xxx	存储器读写
10100x	PC-相对寻址, 如ADR
10101x	SP-相对寻址 (SP +偏移量) , 如 ADD
1011xx	其他16位指令
11000x	多寄存器写，如STM, STMIA, STMEA
11001x	多寄存器读，如LDM, LDMIA, LDMFD
1101xx	条件转移和系统调用Conditional branch, and Supervisor Call
11100x	无条件转移
11101x	32位Thumb2 指令 （ v6 用到的仅1条指令：BL）
1111xx	32位Thumb2 指令

指令共16位，操作码占去6位，操作数还有10位，源寄存器、目的寄存器各占用3～4位，留给立即数偏移量的位数自然十分有限

（ARM指令的高4位是条件位，111x表示无条件，故Thumb/Thumb2都是无条件执行指令）

汇编指令助记符的读法

ADDS	Add with Signal
ADC	Add with Carry
SUB	Subtract
SBC	Subtract with Carry
RSBS	Reverse Subtract (NEG)
CMP	Compare
CMN	Compare negative
MUL	Multiply
BIC	Bitwise Clear
MVNS	Logical Bitwise NOT
TST	Test
ASR	Arithmetic Shift Right
LSL	Logical Shift Left
LSR	Logical Shift Right
ROR	Rotate Right
REV	Reverse Byte Order

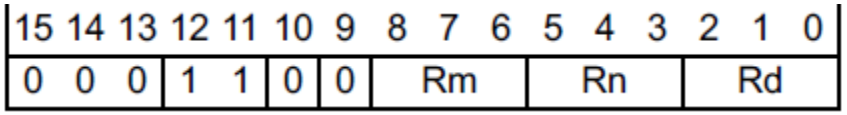
LDR	Load word from memory
LDRH	Load Half word
LDRB	Load Byte
LDM	load Multiple
LDMIA	load Multiple Increment Addr
STR	Store word to memory
MSR	Move to Special Register
MRS	Move R from Special Register
B	Branch
BL	Branch & Link
BX	Branch with exchange
SVC	Supervisor Call
CPS	Change Processor State
CPSIE	Clear PRIMASK (IRQ E)
CPSID	Set PRIMASK (disable IRQ)
WFI	Wait for Interrupt
ISB	Instruction Synchron. Barrier

分析一条ADD 指令

ADDS <Rd>,<Rn>,#<imm3>

ADDS <Rdn>,#<imm8>

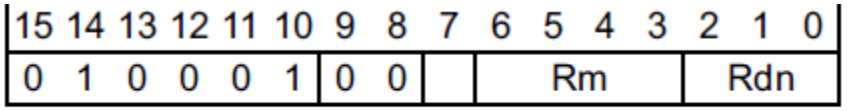
ADDS <Rd>,<Rn>,<Rm>　　影响标志，用于低寄存器加



if (DN:Rdn) == 1101(R13) 或

Rm == '1101' then ADD (SP +Rm)

不影响标志的解码T2:



If Rm≠PC, (DN:Rdn) =15, Rdn=PC+Rm

ADD SP, <Rm>　　**SP =SP + Rm**

ADD <Rd>, SP　　**Rd = Rd + SP**

ADD <Rd>, SP, #immed8　　**Rd =SP+ZeroExtend(#immed8<<2)**

ADD SP, #immed7　　**SP=SP +ZeroExtend(#immed7 <<2).**

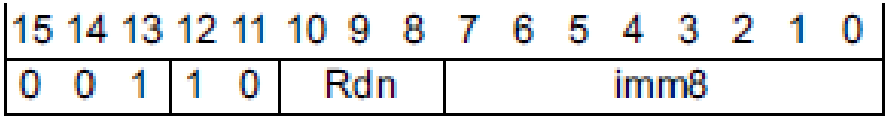
ADD <Rd>, PC, #immed8　可写成：**ADR <Rd>, <label>**

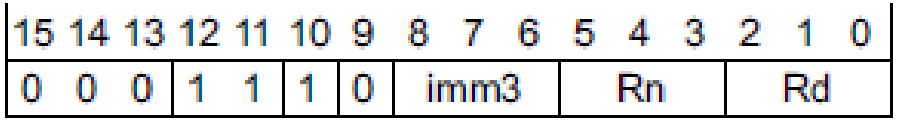
Rd=(PC[31:2]<<2)=ZeroExtend(#immed8 <<2)

第2个编码方式Rdn和Rm都是4位，Rdn中d=n，含高寄存器,不影响标志

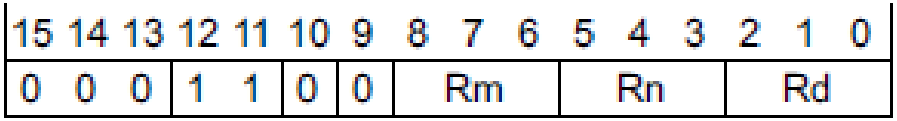
ADD的16b指令有4个编码

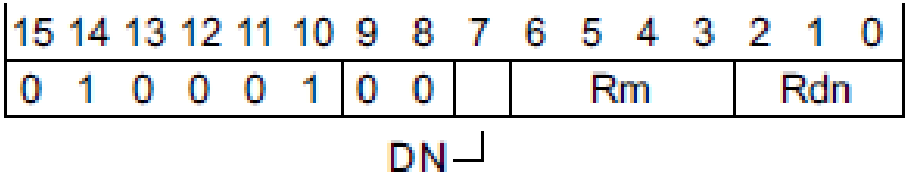
- **ADD** (立即数)
  - **ADDS <Rdn>,#<imm8>**
  - **Rdn=Rdn + #immed8**
  - 指令中只能代入一个<256的数
  - **ADDS <Rd>,<Rn>,#<imm3>**
  - **Rd=Rn + #immed3**
  - 最大7，常用4，指向下一个地址





- **ADD** (寄存器)
  - **ADDS <Rd>,<Rn>,<Rm>**
  - **Rd = Rm + Rn**





  - **ADD <Rdn>,<Rm>**
  - **Rdn = Rdn + Rm**

DN=1,则Rdn含高寄存器，当然也包括PC、SP

  - **16b 指令可夹带1个8b立即数，v7中的32 b 指令可夹带1个最多16b立即数**
  - **ADD指令占用了1/32+1/128+1/128+1/256＝13/256=1/19.7的16b指令资源!**

v7的 32b ADD指令还有3个指令码（不含SP）

```
ADD{S}<c>.W <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	Rn				0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
```

夹带12b立即数，影响标志位

```
ADDW<c> <Rd>,<Rn>,#<imm12>
```

夹带12b立即数，不影响标志位

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	Rn				0	imm3			Rd				imm8							

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
ADD{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	0	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

ADD{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
Rd= Rn + Rm;
做完加法后再将Rd左右移位
相当于2条指令

type	imm3:imm2	Instruction	See
00	00000	Move	*MOV*
	not 00000	Logical Shift Left	*LSL* (
01	-	Logical Shift Right	*LSR* (
10	-	Arithmetic Shift Right	*ASR* (
11	00000	Rotate Right with Extend	*RRX*
	not 00000	Rotate Right	*ROR*

v6 支持 R8~R11的16b指令仅3条

Instruction	MOV	;Move register into register
Syntax	MOV <Rd>, <Rm>	;Copy Register
	CPY <Rd>, <Rm>	;同上，另一种写法
Syntax	ADD <Rdn>, <Rm>	; Add two registers
	Rd = Rd + Rm	;without updating APSR
Syntax	CMP <Rdn>,<Rm>	;Compere

Note : Rdn, Rm can be high or low registers.

注意：没有R8~R12的入栈出战指令， R12在响应中断时自动入栈、出栈，R8~R11无法入栈、出栈

可不使用R8~R11这4个高寄存器。但高寄存器包括PC和SP。ADD用于地址计算无须影响标识位，只能用于基地址+寄存器偏移量类的运算。故高寄存器可用作存储器分页寄存器，将R8～R11中的一个设为最常用的一个常数，如RAM、I/O等的基地址；或置0，用于存储器或寄存器快速清零。以便快速给低寄存器赋值

如果MOV 指令面向低寄存器，要求影响标志写成MOVS，实际是循环移位LSL指令的另一种写法。

算数与逻辑运算指令（寄存器间）

加法	3位立即数加	ADDS Rd, Rn, #<imm3>	N Z C V	Rd := Rn + imm3
	低寄存器间加	ADDS Rd, Rn, Rm	N Z C V	Rd := Rn + Rm
	8位立即数加	ADDS Rd, Rd, #<imm8>	N Z C V	Rd := Rd + imm 8
	低寄存器带进位加	ADCS Rd, Rm	N Z C V	Rd:=Rd + Rm + C bit
减法	寄存器低到低	SUBS Rd, Rn, Rm	N Z C V	Rd := Rn – Rm
	偏移量3位 立即数	SUBS Rd, Rn, #<imm3>	N Z C V	Rd := Rn – imm3
	偏移量8位 立即数	SUBS Rd, #<imm8>	N Z C V	Rd := Rd – imm8
	带借位减	SBCS Rd, Rm	N Z C V	Rd := Rd–Rm–借位位C
	求负	RSBS Rd, Rn, #0	N Z C V	Rd := – Rn
乘	16位数乘法	MULS Rd, Rm, Rd	N Z	Rd := Rm * Rd
比较	比较Rn, Rm	CMP Rn, Rm	N Z C V	更新Rn-Rm 的标志位
	与 –Rm 比较	CMN Rn, Rm	N Z C V	更新Rn + Rm 的标志位
	与8位立即数比较	CMP Rn, #<imm8>	N Z C V	更新Rn–imm8 的标志位
逻辑运算	与	ANDS Rd, Rm	N Z	Rd &= Rm
	异或	EORS Rd, Rm	N Z	Rd ^= Rm
	或	ORRS Rd, Rm	N Z	Rd |= Rm
	位清零	BICS Rd, Rm	N Z	Rd &=!Rm
	取反	MVNS Rd, Rm	N Z	R d = ! Rm
	测试位	TST Rn, Rm	N Z	更新Rn&Rm标志位

MOV指令是移位指令的特殊形式（不移位）

单条指令可完成2个操作，给寄存器赋值和移0~31位：

- MOVS <Rd>,<Rm>,ASR #<n> 实际上是 ASRS <Rd>,<Rm>,#imm5
- MOVS <Rd>,<Rm>,LSL #<n> 实际上是 LSLS <Rd>,<Rm>,#imm5
- MOVS <Rd>,<Rm>,LSR #<n> 实际上是 LSRS <Rd>,<Rm>,#imm5

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	imm5					Rm			Rd		

- MOVS Rd, Rm *is a pseudonym for LSLS Rd, Rm, #0.*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

v6手册上错了，这些是32b的v7指令：

- MOVS <Rd>,<Rm>,ASR <Rs> → ASRS <Rd>,<Rm>,<Rs>
- MOVS <Rd>,<Rm>,LSL <Rs> → LSLS <Rd>,<Rm>,<Rs>
- MOVS <Rd>,<Rm>,LSR <Rs> → LSRS <Rd>,<Rm>,<Rs>
- MOVS <Rd>,<Rm>,ROR <Rs> → RORS <Rd>,<Rm>,<Rs>

MOV是寄存器间的传值指令，是借助传值并移位指令移位的指令实现的

乘法指令 MUL

只能用于16b乘法，否则结果会溢出：

乘法　　Multiply　　MULS Rd, Rm

注意可能的溢出：

Rd=Rd*Rm的低32位 只影响N、Z标志

执行周期数，取决于乘数中1的数目

ARM 手册: 说执行周期1 or 32， 有误，1指有硬件乘法器的MCU，32指乘数中1的数目。因结果仅保存低32b，实际上是 16b*16b 乘法

- ARMv6去掉了除法指令,需要靠软件实现
- M3/4/7中，可实现32b乘法，结果为64b，在2个寄存器中
- 有些M0+中有硬件乘法器

移位指令（参考：桶式移位寄存器）

ASR <Rd>, <Rm>, #immed5 ; Arithmetic Shift Right

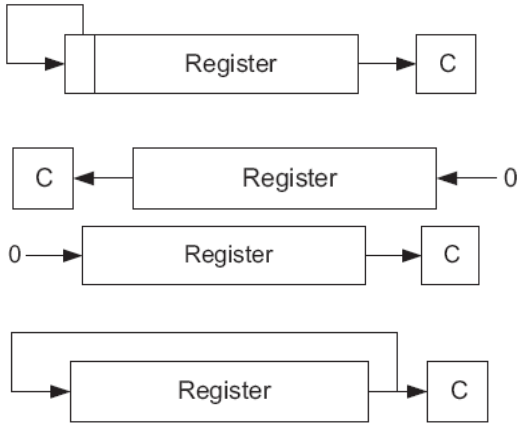
ASR <Rd>, <Rm> ; Arithmetic Shift Right

LSL <Rd>, <Rm> ; Logical Shift Left

LSR <Rd>, <Rm>, #immed5 ; Logical Shift right

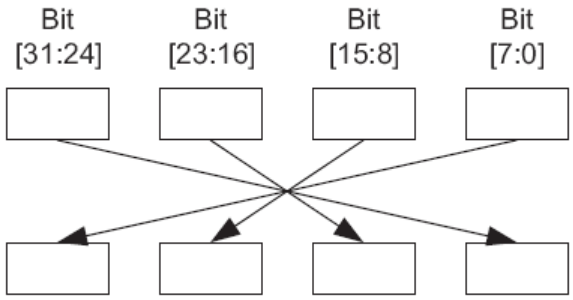
ROR <Rd>, <Rm> ; Rotate Right

ROR <Rd>, <Rm>, #immed5 ; Rotate Right

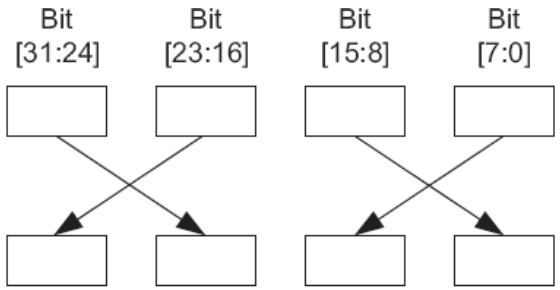


换位指令

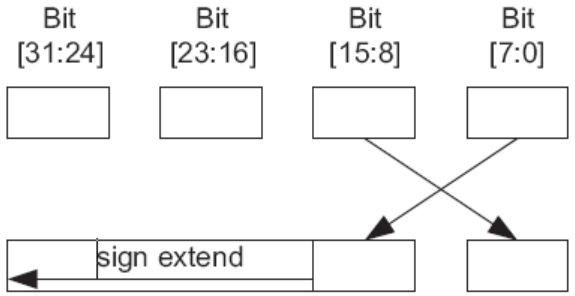
REV <Rd>, <Rm>



REV16 <Rd>, <Rm>



REVSH <Rd>, <Rm>



数据类型转换指令

SXTB, SXTH, UXT, UXTH 用于有、无符号半字、字节的扩展

RISC的存储器读/写

寻址方式仅1种：寄存器+偏移量，16b指令偏移量小于32，或在寄存器中：

Syntax	LDR <Rt>, [<Rn>, #immed5]	; Word read	偏移量最多5b
	LDRH <Rt>, [<Rn>, #immed5]	; Half Word read	
	LDRB <Rt>, [<Rn>, #immed5]	; Byte read	
Syntax	LDR <Rt>, [<Rn>, <Rm>]	; Word read	偏移量在Rm中
	LDRH <Rt>, [<Rn>, <Rm>]	; Half Word read	
	LDRB <Rt>, [<Rn>, <Rm>]	; Byte read	
Syntax	STR <Rt>, [<Rn>, <Rm>]	; Word Write	偏移量在Rm中
Syntax	STR <Rt>, [<Rn>, <Rm>]	; Word Write	偏移量最多5b

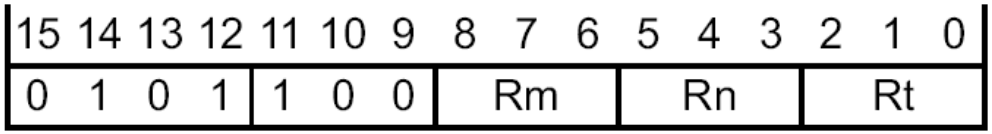
- 实质上只有寄存器+偏移量这1种寻址方式
  - 用寄存器表示偏移量可达32b，
  - 使用指令自带的立即数则只有5b，0~124
- 寄存器间接寻址是RISC 唯一的寻址方式，实现 i++ 得先让指针指向该变量，再读入寄存器中，再＋＋，再写回指针指向的变量，显然不如CISC方便。有时读写指令前后还得关、开中断予以保护，C难以实现。MCU设计者有办法解决

存储器读的指令编码

需要2个周期完成

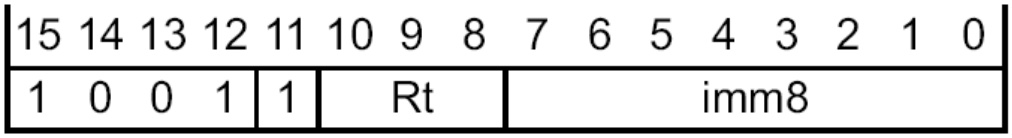
读存储器地址Rm +偏移量Rn 到目的寄存器 Rt:

LDR <Rt>,[<Rn>,<Rm>] ; **Rt =[Rm + Rn]** [ ] 表示地址



各为8个低寄存器

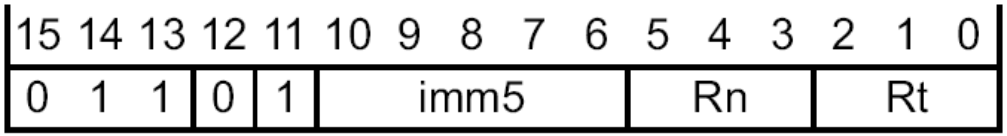
LDR <Rt>, [PC, #<imm8>] （**LDR <Rt>,[SP{,#<imm8>}]**）



0 ～1020

LDR R0,=0x12345678 ;汇编器提供的复合指令，在PC下方定义一个常数

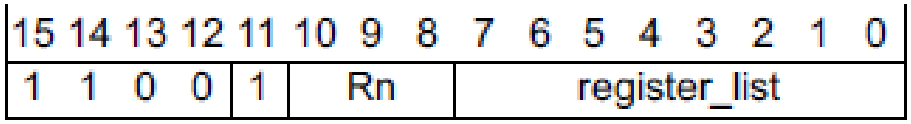
LDR <Rt>, [<Rn>{,#<imm5>}] ;偏移量为5位、字



多寄存器读/写 LDM/STM

给多个寄存器赋值

Syntax LDM <Rn>, {<Ra>, <Rb> ,..}



;Ra=[Rn], Rb=[Rn+4],…

例如： LDM R2, {R1, R2, R5 e R7} **; Read R1,R2…R6 and R7 from memory.**

由LDM合成DMIA指令,PLDMFD同POP:

Syntax LDMIA R0!, {<Ra>, <Rb> ,..} **; 同上，然后R0 再+4**

LDMFD R0!, {<Ra>, <Rb> ,..} **; Ra=memory[R0], Rb=memory[R0-4]…**

;同上，然后R0再减4

写多个存储器 (STM 及由其合成的STMIA和STM, STMEA同PUSH)

Syntax STM<Rn>, {<Ra>, <Rb> ,..} **; Ra=memory[Rn], Rb=memory[Rn+4]…**

STMIA R0!, {R1, R2, R5 e R7} **; Store R1, R2, R5, R6, and R7 to memory**

; and update R0 to address after where R7 stored

- 多寄存器读写指令执行周期数为1+N 含PC的加一个周期
- LDM, STM, PUSH ,POP ， MUL等多周期指令可随时被中断，中断返回后借助PC值重新执行这条指令

栈操作指令

```
PUSH  {R1, R2, R5-R7, LR}        ; Store R1, R2, R5, R6, R7, and LR to stack
POP   {R1, R2, R5-R7,PC}         ; Restore R1, R2, R5, R6, R7 from stack
```

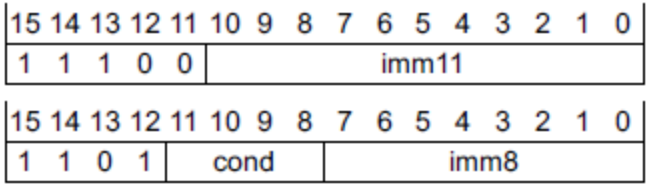
无条件转移指令 (B <label>)

- B <label> Branch ;16b指令，转移范围±2046B
- BX <Rm> Branch and Exchange ;可转移到任何Rm所含32位 地址
  - 因为只支持T指令集，Rm的b0须置1，即不改变指令集，范围有±2GB
  - BX LR 是子程序返回指令
- BL <label> ； Branch and Link32b指令,子程序调用,返回地址在LR, 范围±16MB
- BLX <Rm> ； Branch and Link with Exch 子程序调用，返回地址在LR范围±2GB
  - Rm是指向函数的指针，b0=1,16b指令，用于例如调用RAM中的程序

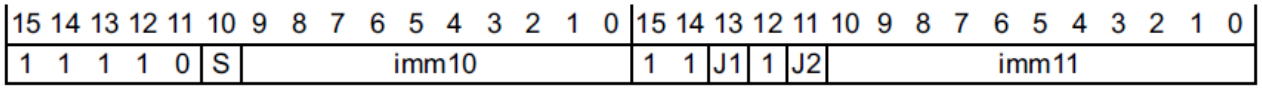
跳转指令(其中BL 是32位指令)

B <label> ; Branch 范围为PC ± 2KB

B <cond> <label> ;范围为PC ± 254B



BL <label> ; 函数调用，范围±16MB

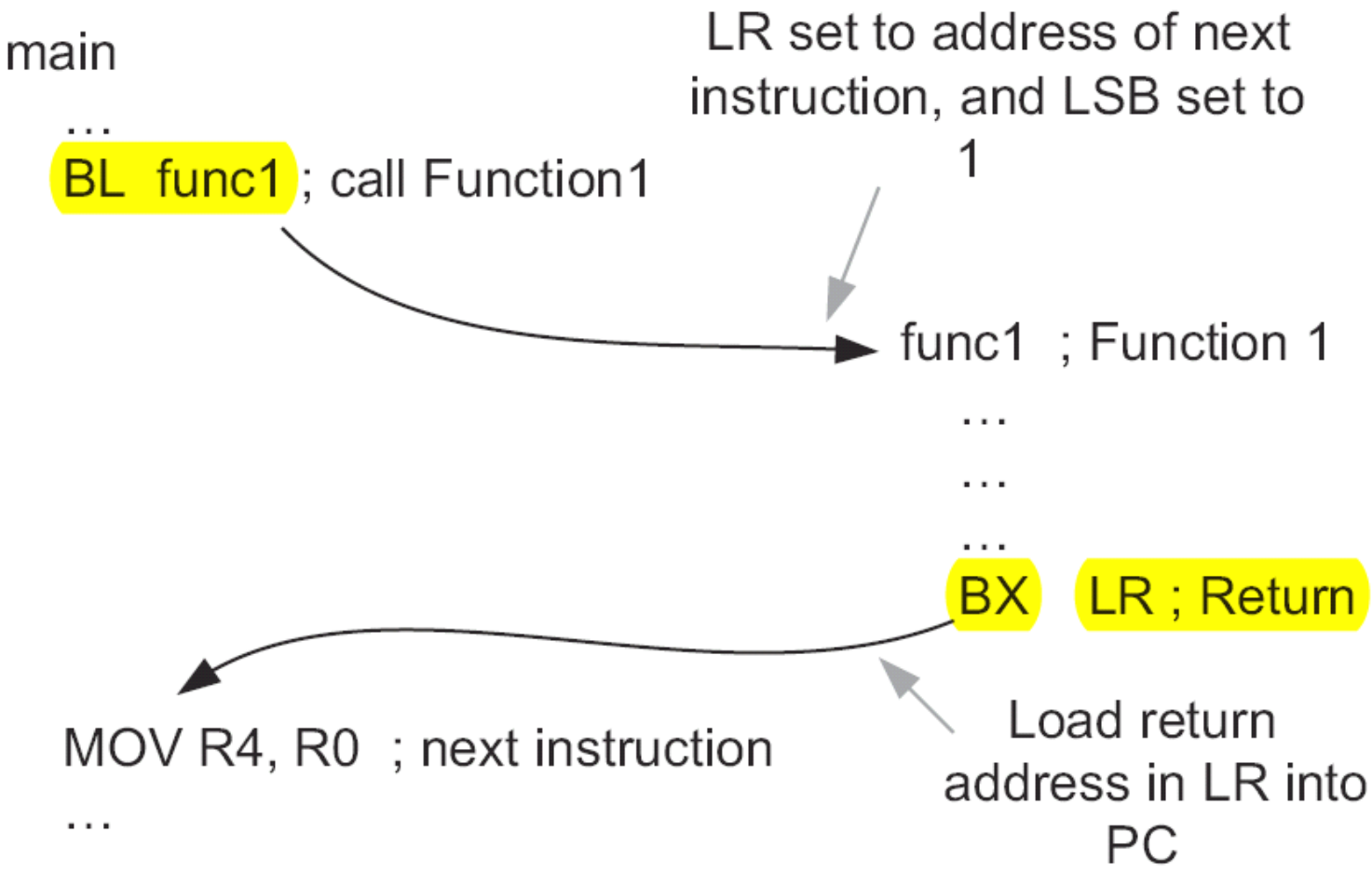


BLX <Rm> ; 函数调用，范围±2GB

跳转条件：

BCC	Branch if Carry Clear C = 0	BEQ	Branch if Equal Z = 1
BCS	Branch if Carry Set C = 1	BHI	Branch if Higher C+ Z = 0
BMI	Branch if Minus N = 1	BLS	Branch if Lower or Same C+ Z = 1
BNE	Branch if Not Equal Z = 0	BGE	Branch if Greater than or Equal N ⊕ V = 0
BPL	Branch if Plus N = 0	BGT	Branch if Greater Than Z+ (N ⊕ V) = 0
BVC	Branch if overflow V Clear V = 0	BLE	Branch if Less than or Equal Z+ (N ⊕ V) = 1
BVS	Branch if V Set V = 1	BLT	Branch if Less Than N⊕ V = 1

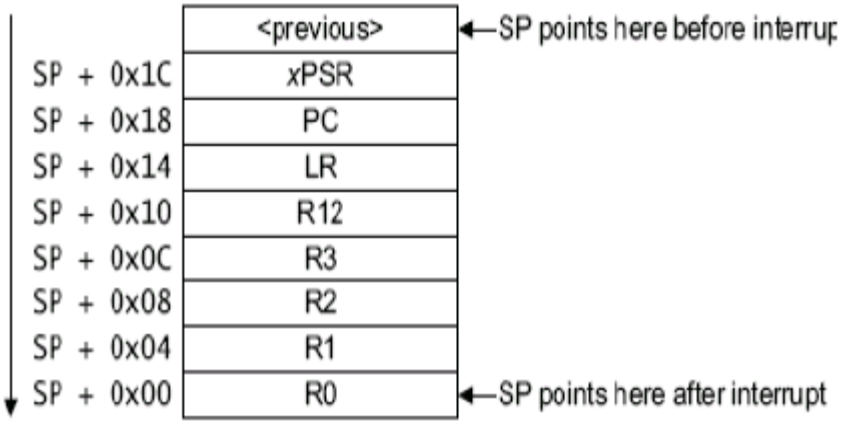
子程序调用、返回指令的用法



RISC 特点：子程序只能调用1次，嵌套调用得先将LR寄存器入栈

进入中断与中断返回

- 进入中断时使用当前栈保护现场，然后给LR赋上3个可能的值并进入中断处理模式（Handler ）获取向量，进入ISR;
- 中断服务子程序遇到下列指令返回（EXC_RETURN）：
  - POP PC
  - BX
- 中断处理程序ISR遇到BX LR时，即可恢复所有入栈寄存器使中断返回，LR中原值不必保留了，新值有3种情况：
  - 0xFFFFFFF1 ：使用MPS返回，退出了一层嵌套的中断，但仍处于中断处理模式中
  - 0xFFFFFFF9：使用MSP返回，从中断处理模式中返回到主模式（Thread Mode）
  - 0xFFFFFFFD：使用PSP，脱离中断处理模式，返回后仍使用PSP
- 如果ISR中需要调用子程序且需要子程序返回时，则须将LR入栈、出栈以保护LR的上述值不被破坏。



同步与屏障指令清理流水线，v6 中一般不用

ISB **Instruction Synchronization Barrier**

DMB **Data Memory Barrier**

DSB **Data Synchronization Barrier**

即便用到也以CMSIS函数形式出现

软中断指令SVC和设硬件断点指令BKPT

- **System Sevice Call:** 参数0～255要通过堆栈中的IR值提取PC值，并从指令低8位提取出来，传递给系统调用跳转表
- 断点设置行为和指令格式类似软中断，但会进入Debug模式。

指令格式及汇编程序

语句标号	操作码	操作数1、2、3	; 注解
Subtract	SUBS	<Rd>, <Rn>, <Rm>	; Rd =Rn-Rm

程序实例： 两个变量相加

汇编管理指令

```
* Add two arguments
*
            .AREA subrout, CODE, READONLY       ; Name this block of code
            .ENTRY                              ; Mark first instruction to execute
start       MOV     r0, #10                     ; Set up parameters
            MOV     r1, #3
            BL      Do_add                      ; Call subroutine
stop        MOV     r0, #0x18                   ; angel_SWIreason_ReportException
            LDR     r1, =0x20026                ; ADP_Stopped_ApplicationExit
            SVC     #0x12                       ; ARM semihosting (formerly SWI)
Do_add      ADD     r0, r0, r1                  ; Subroutine code
            BX      lr                          ; Return from subroutine
            .END                                ; Mark end of file
```

RISC 如何给寄存器赋值?

RISC类型CPU中立即数的概念很不一样，立即数仅指偏移量。要改变过去我们习惯的CISC中的立即数概念。要把一个数读入寄存器，得使用下面的方法：

程序（Flish）中用汇编管理指令（伪指令）定义该数据（CISC概念中的立即数），然后以PC到定义立即数的地址为偏移量，将“立即数”从存储器写入寄存器：

LDR <Rt>, [PC, #immed8]　　; Word read　8b偏移量，4b对齐，4~1020

要把1个立即数写入存储器，还得再把存储器地址用上述方法读到寄存器，再将上述寄存器的值写进存储器：

STR <Rt>, [Rm] ;

可借助移位指令给寄存器赋值一个5b精度的值，同时左移若干位实现

编译器提供的合成指令（所谓伪指令）LDR

```
        LDR R0,=0x12345678      ;用于给寄存器赋值的伪指令（literal load）
```

得用如下指令实现：

```
        LDR  R0, [PC, #<imm8>] ;  偏移量范围：0～1020B
```

可写成

```
        LDR R0, Const           ;还要在 其下方不远于1KB处定义这个常数:
        …… ……
        BX  LR                  ;子程序返回
Const   DCD 0x12345678          ;汇编管理指令定义一个常数或字符串
```

汇编器要计算Const 到当前PC的差并右移2位，填入上述语句，以及把语句：

CONST DCD 0x12345678 插入到下方1KB以内的合适的地方，并使之不影响程序流(将Thumb 指令成双对齐)。通常是返回指令BX LR后面。

然后用LTORG 汇编管理指令留出“文字池”Literal Pool，按偶数地址对齐

也可使用批处理语句LDM、LDMIA初始化时做此事

将数值写回存储器也只能用寄存器指针间接寻址

合成指令LDR的存储器读写汇编程序举例

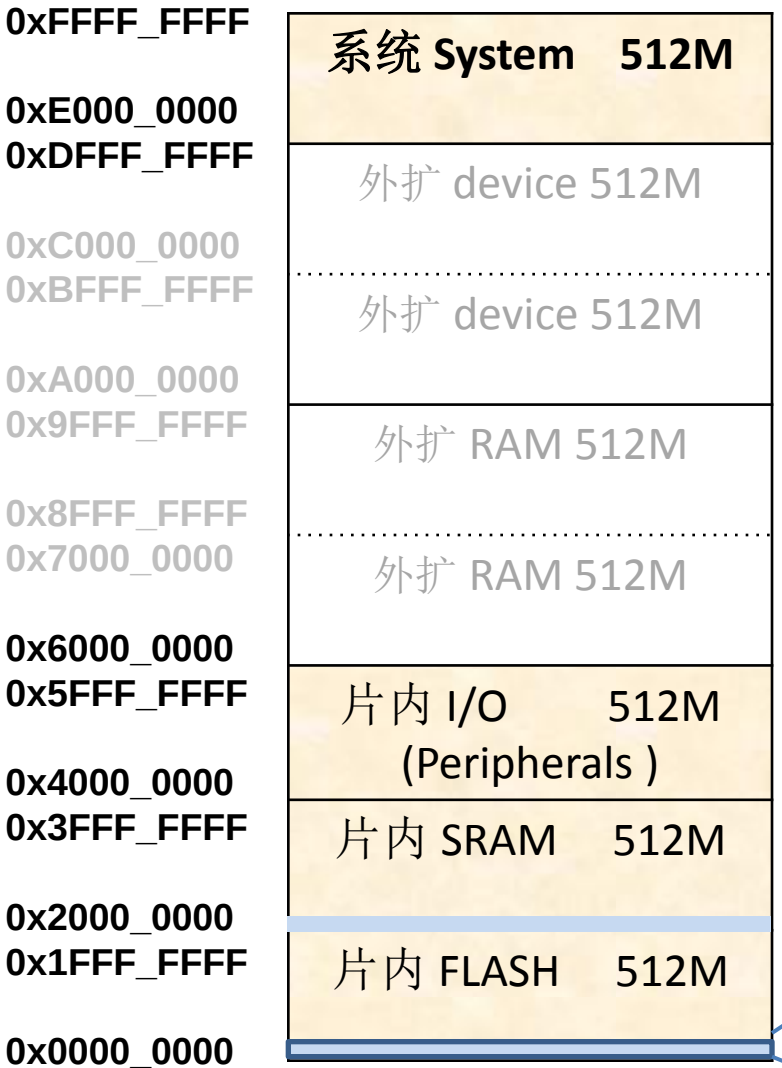
把存储器中的向量表复制到RAM中，把0x00000000的48*4Bytes数据复制到0x20000020:

```
    LDR r0,=0x00000000      ; Source address源地址,可使用LDR r0, #0优化
    LDR r1,=0x20000020      ; Destination address，得使用组合指令
    LDR r2,=48              ; number of bytes to copy，>31,也得使用组合指令
copy_loop                   ; acts as loop counter 循环
    SUBS r2, r2, #4         ; decrement offset and loop counter 一次4个字节
    LDR  r4,[r0, r2]        ; read 1 word(4Bytes) 读一个字
    STR   r4,[r1, r2]       ; write 1 word          写一个字
    BNE copy_loop           ; loop until all data copied 循环48次
```

初始化变量时指令多，但执行100次，一次6周期，还是比CISC快、指令效率高，其实所有应用程序可以100%用C写，懂汇编有利调试和优化程序关键部位

MCU设计者做些了什么？

ARM 定义的寻址空间被b31,b30,b29分成8个512M　Cortex-M只用其中4个

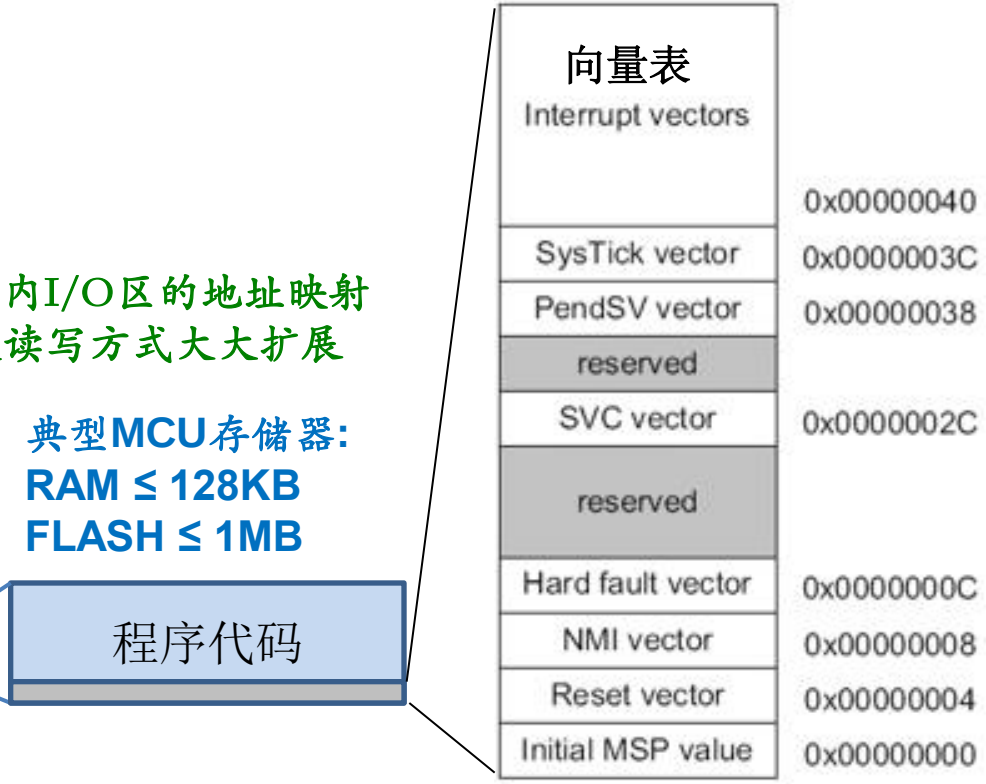


System 由ARM和MCU制造商共同定义使用

单片方式下的MCU不使用外扩区

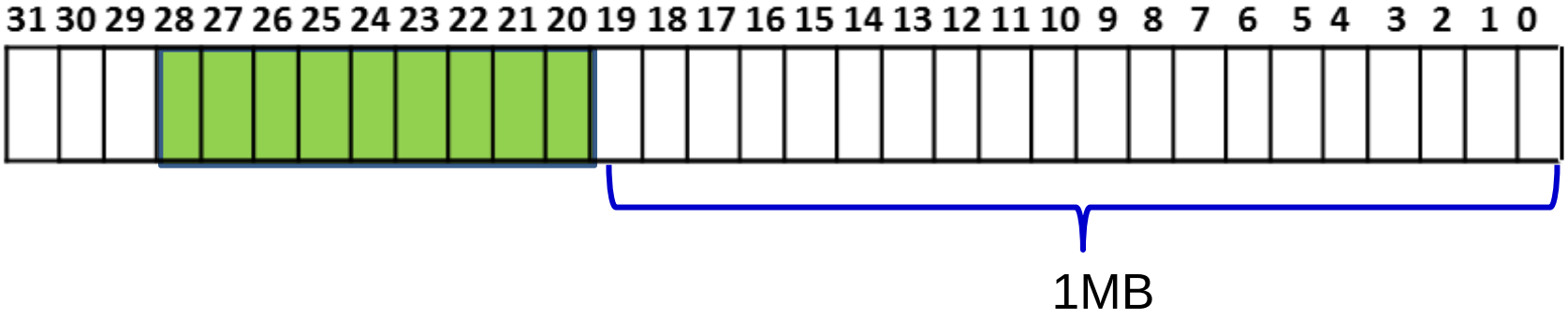
片内I/O区的地址映射
使读写方式大大扩展

典型MCU存储器:
RAM ≤ 128KB
FLASH ≤ 1MB



MCU厂商用地址线中的空闲位为读写指令添加新功能

b19~b28 这10b用于指令编码：



Cortex M MCU工作在单片方式下，RAM/Flash 不会大于1MB，至少有9b地址线不用。MCU厂家在把ARM内核、存储器、I/O设计成芯片时会利用这些位，给存储器读写指令增加新功能，特别重要的功能是：

读-改-写一体化的“原子操作”功能，包括：

某1位的位操作：置1、清零、与、或、非、异或；

多个连续位的植入、提取等，弥补了RISC的不足

只是尚无法实现CISC的 i＋＋操作

以存储器读/写指令实现位清零,置1，位段提取,写入

b=位置　　w＝宽度

	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
bfib	0	*	*	1	-	-	b	b	b	-	w	w	w	mem_addr																		
bfih	0	*	*	1	-	b	b	b	b	w	w	w	w	mem_addr																		0
bfiw	0	*	*	1	b	b	b	b	b	w	w	w	w	mem_addr																	0	0

例如读存储器指令Load，可实现可实现读后把某一位置1

	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
iolas1b	0	1	0	0	1	1	-	-	b	b	b	-	mem_addr																			
iolas1h	0	1	0	0	1	1	-	b	b	b	b	-	mem_addr																			0
iolas1w	0	1	0	0	1	1	b	b	b	b	b	-	mem_addr																		0	0

例：

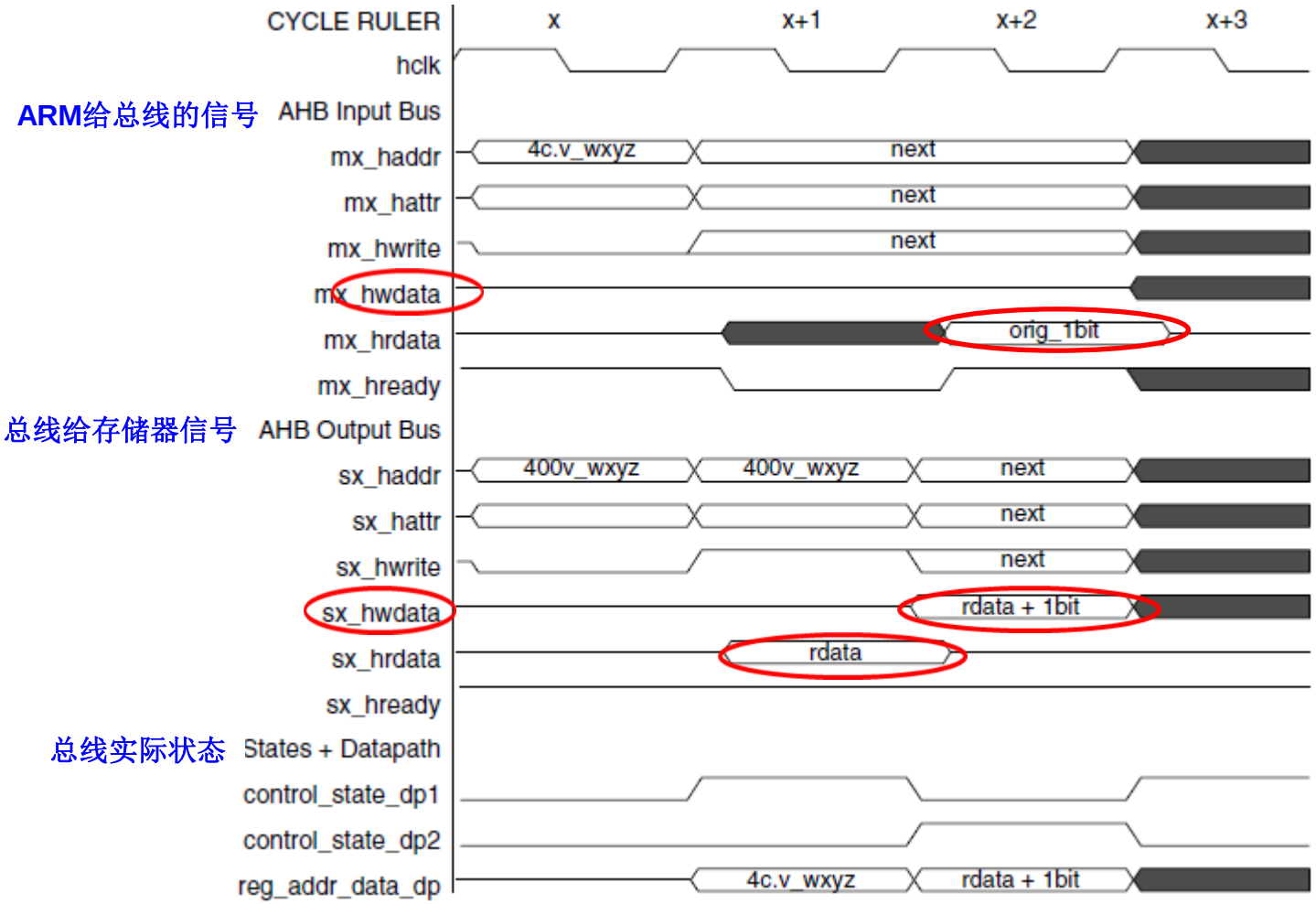
```
#define IOXORW(ADDR,WDATA) \ /*GNU的C宏汇编函数定义函数：写IO口与某数异或实现读改写*/
__asm(“ldr r3, =(3<<26);” \ /* b26、b27为%11 */
 “orr r3, %[addr];” \ /* 让r3指向 IO（或存储器）隐患指令的虚地址 */
 “mov r2, %[wdata];” \ /* 要参与异或的32b数在r2中 */
 “strb r2, [r3];” \ /* 写的是实地址，并实现XOR */
:: [addr] ″r″ (ADDR), [wdata] ″r″ (WDATA) : ″r2″, ″r3″);
      /*某r为实际地址ADDR，另一个r为要异或的数*/
```

b28:26 写：001=AND,010=OR,011=XOR,1xx=BFI；读:010=LAC,1011=LAS1,011=UFBX 等7条指令

这些看似对存储器和I/O虚地址的读写是MCU公司加上去的

是 “读-改-写” 一体化的原子操作，很有用！

MCU设计者用读存储器指令实现读后将某位置1的时序



读指令Load变成了读-改-写load-and-set 1-bit

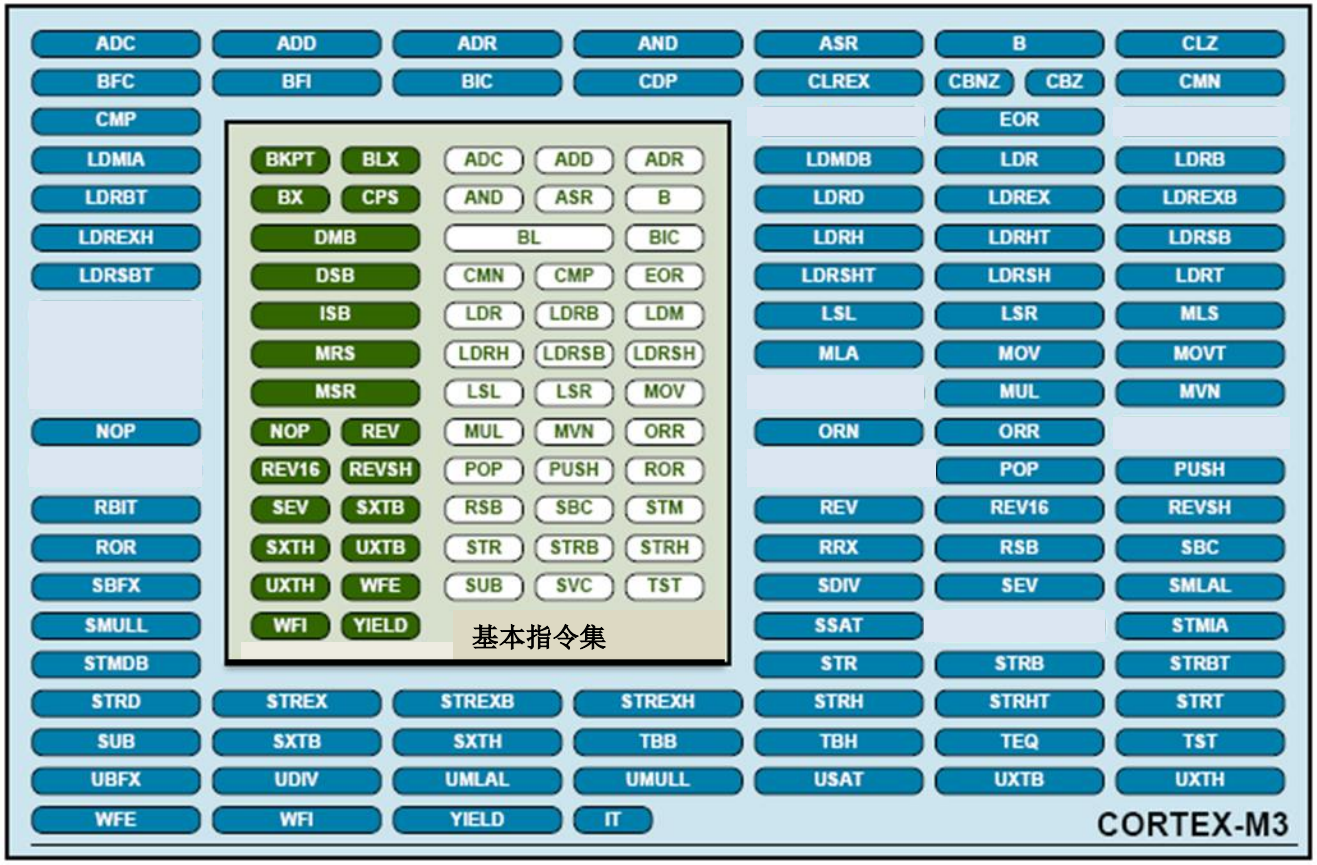
编译器生成的C代码

- 汇编中可调用C函数，C函数的参数代入和返回值：
  - 代入1、2、3个参数分别放在R0、R1、R2中，
  - 更多的参数可以以指向结构的指针形式代入
  - 局部变量占用的空间从栈中获取，子程序返回前释放占用空间
  - 返回值在R0中
  - 写汇编子程序要写成C可调用的格式
- Gcc 生成的C代码不含R8 ~ R11，即不使用R8 ~ R11
- CodeWarrior 实际上是包装了Gcc,增加了Processor Expert
- RTOS做任务切换时，不一定要保护R8 ~ R11
  - 响应中断时，由于R12～R15和R0~R3自动入栈，任务切换时中断服务子程序用到R4～R7时，需将R4～R7入栈
- Keil IAR Gcc 编译器的优缺点

为什么程序用C编译后必须优化?

- C是硬件无关语言，不能和CPU内部寄存器打交道
- C规定函数间参数传递必须用栈，以实现硬件无关
- RISC类CPU读写存储器很慢，所有运算得经内部寄存器完成
- ARM规定按R0、R1、R2.....顺序传递参数，不必用栈
- 优化可以使程序长度减少一般左右，速度至少快1倍
- 优化后的程序不再符合ANSI的C规范
- 我们的Monitor程序优化前～25KB，优化后～12KB

Cortex-M0/1/M0+指令集，基本指令集



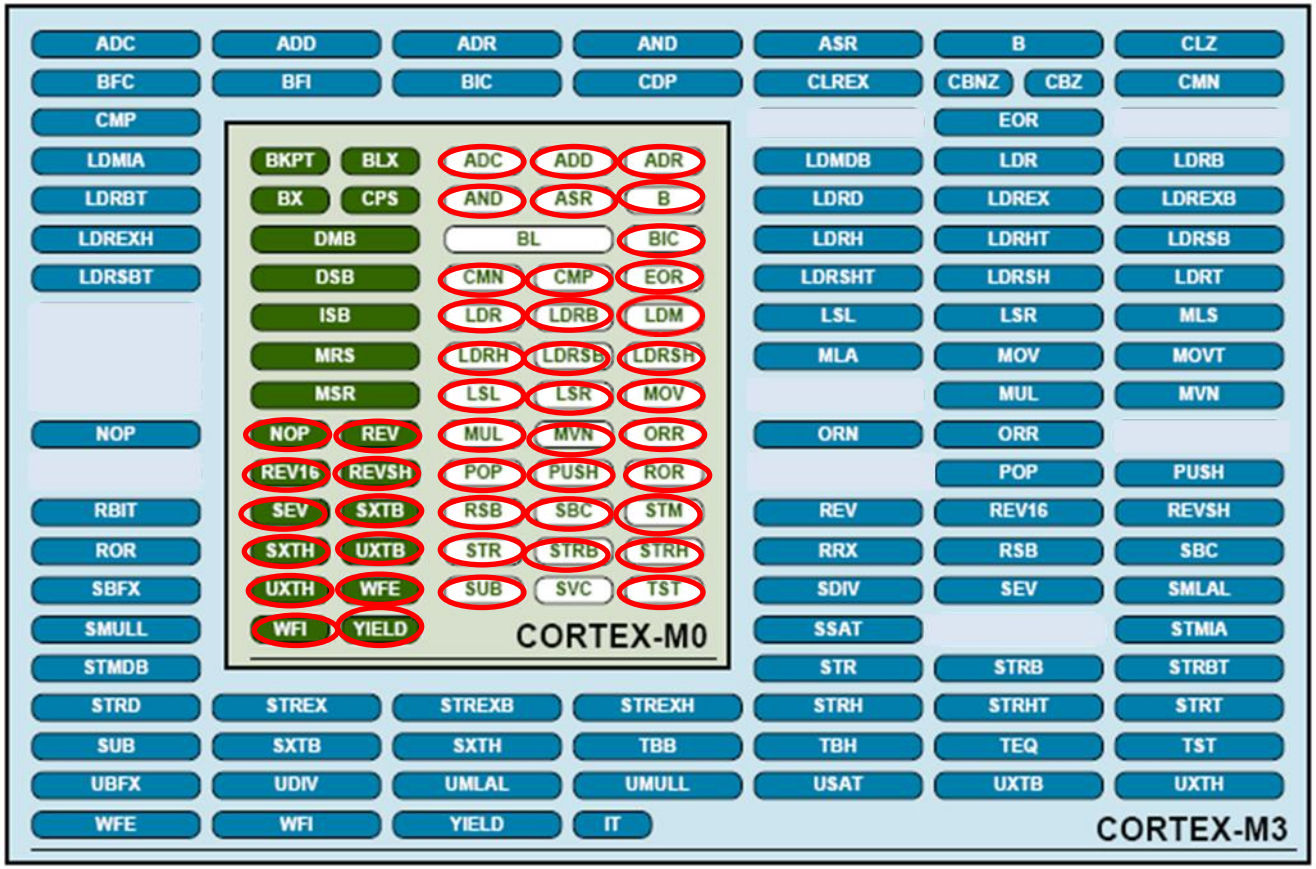
和Cortex M3/4比较，M0+大约却掉了一半指令

提供了无法匹敌的代码密度

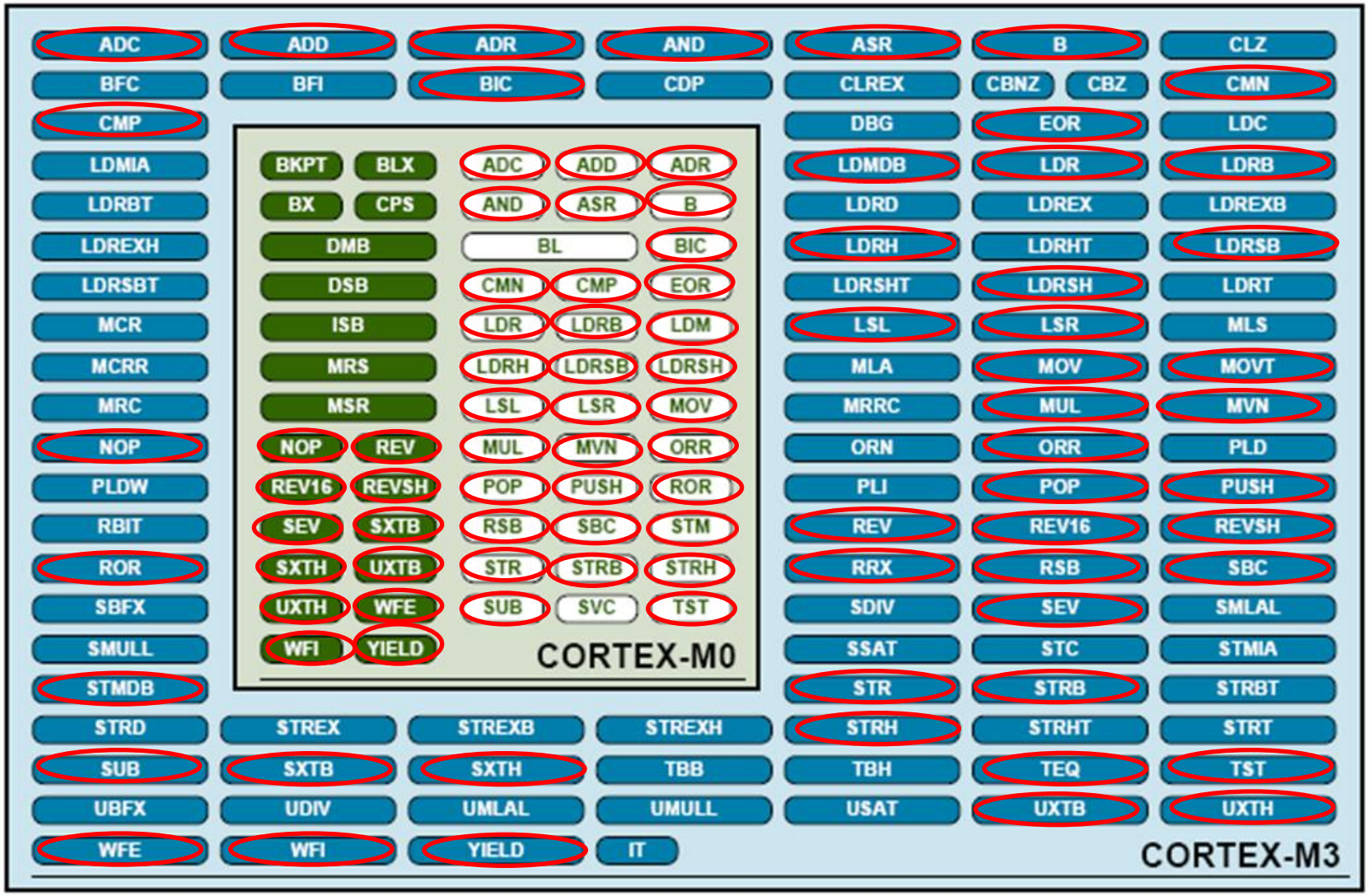
特别要注意的是，去掉了CLZ指令和除法指令

画圈的16b指令都有32b指令码

32b指令对高寄存器也有效，附带移位功能



M3/M4支持16b/32b两种格式的指令

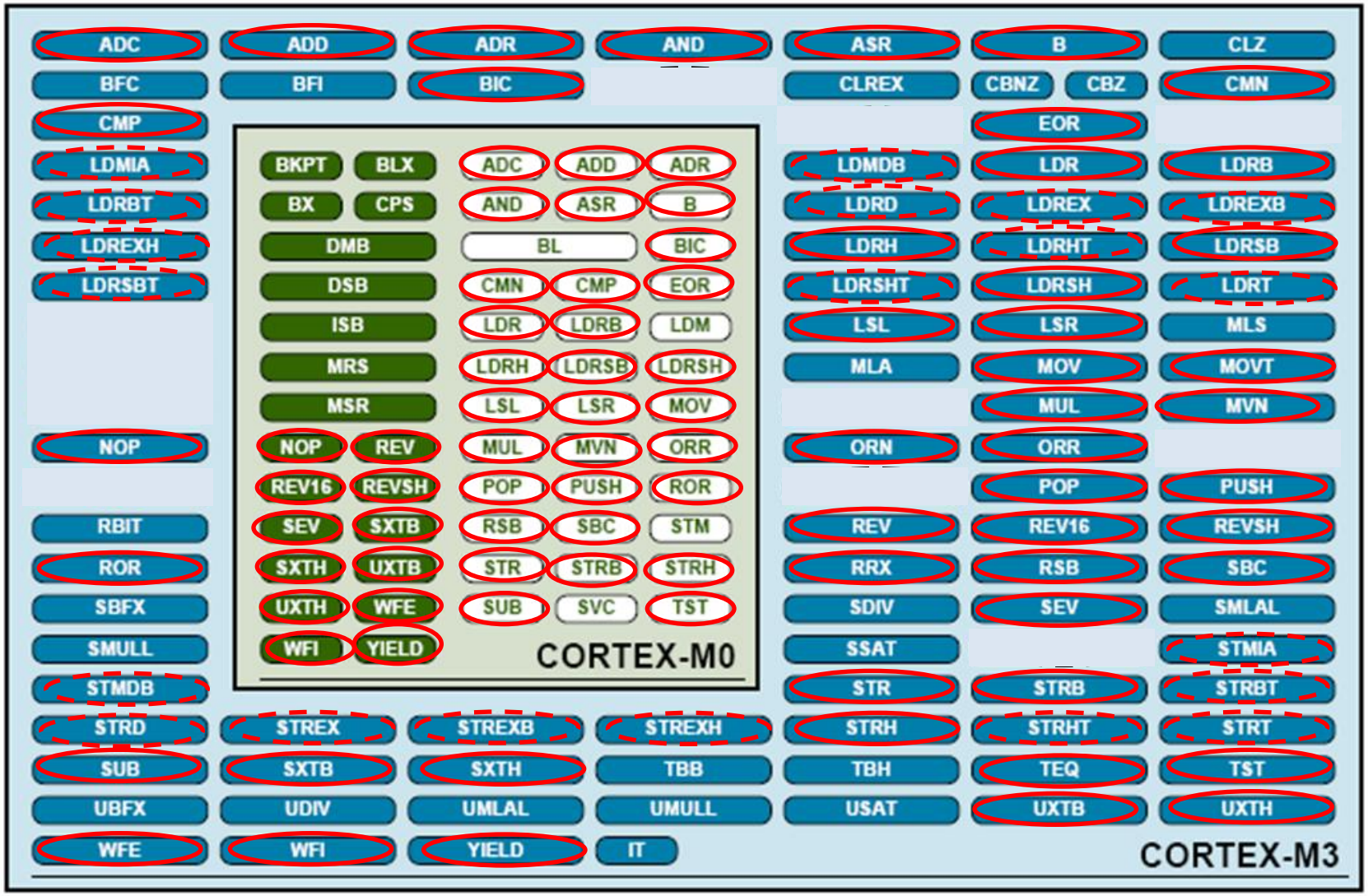


CB{N}Z <Rn>,<label>；Compare Branch

尽量少用高组寄存器：让高组寄存器实现同样功能，指令长度就得16位 了！

V7中更多的32位存储器读写指令

增加了互斥读写（虚线）



协处理器指令支持

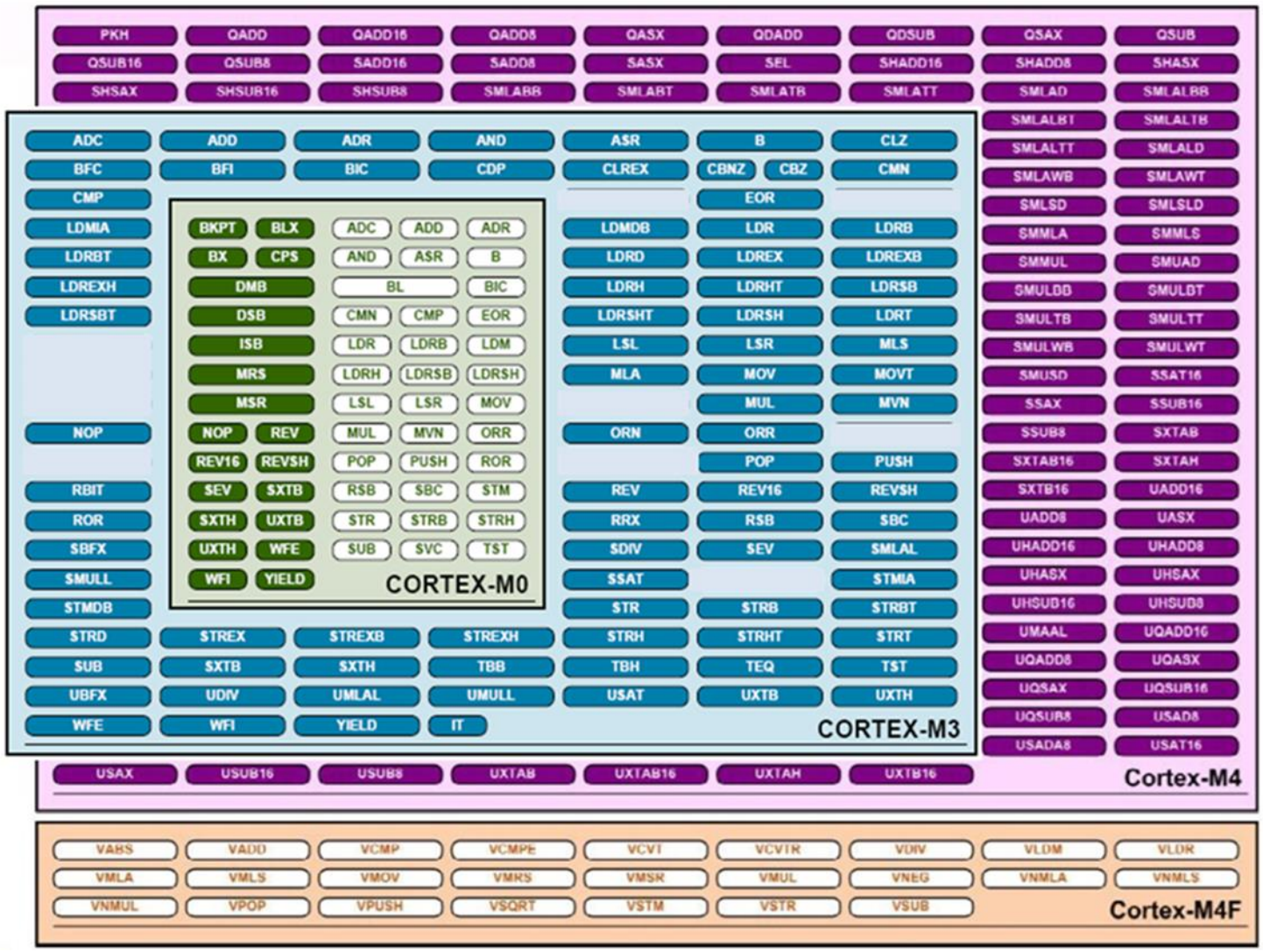
从 v7到v6 精简掉的指令基本上都是32位指令

- **CLZ** ；数出前置零的数目
- **MUL** ；32位乘法 （结果64位）
- **MLA** ；乘加/乘减
- **DIV** ；除法
- **SAT** ；饱和运算
- **RBIT** ；寄存器中位序的180度反转
- **LDREX STREX** ；互斥读写，并在标明进入了互斥访问状态
- **CLREX** ；互斥复位，清互斥状态标志（先前LDREX 做的标记）
- **TBB/TBH** ；从数组表中选一个8/16b前向跳转地址并转移
- **IT** ；If than（ 16b指令）
- **BFC / BFX** ；有/无符号的位段清/ 位段复制并扩展

涉及存储器、I/O读写的指令，需要MCU设计者配合实现

与中断相关的，如WFI，也需要 MCU设计者配合实现

Cortex-M0/M3/M4比较



性能：

(DMIP/MHz)

M3:~1.25

80486:~0.81

M0+:~0.89

8051:~0.11

门数(stages)：

M0＋: 1.5万

80486:118万

RISC运算在内部寄存器间完成，快，多数指令在1个时钟周期内完成。读写存储器慢，需要2个周期，但2个周期可读入2条指令16b指令，平均每周期读入1条指令，由于有2~3条流水线，平均1周期1指令

Thumb2 v6 指令速查表1 （影响标志的指令）

运算		汇编指令	标志	操作	说明	周期
移动	8位立即数	MOVS Rd, #<imm8>	N Z	Rd := imm	Imm8 范围为 0-255	1
	寄存器低到低	MOVS Rd, Rm	N Z	Rd := Rm	同：LSLS Rd, Rm, #0	1
	含高低寄存器	MOV Rd, Rm	不影响	Rd := Rm	同：CPY<Rd>, <Rm>	1
	任何寄存器之间	MOV Rd, Rm	不影响	Rd := Rm	任何寄存器之间。	1
加法	3位立即数加	ADDS Rd, Rn, #<imm3>	N Z C V	Rd := Rn + imm3	Imm3 范围为 0-7	1
	限低寄存器	ADDS Rd, Rn, Rm	N Z C V	Rd := Rn + Rm		1
	含高低寄存器	ADD Rd, Rd, Rm	不影响	Rd := Rd + Rm	不是 Lo 到 Lo。	1
	任何寄存器之间	ADD Rd, Rd, Rm	不影响	Rd := Rd + Rm	任何寄存器之间	1
	8位立即数加	ADDS Rd, Rd, #<imm8>	N Z C V	Rd := Rd + imm 8	imm 范围为 0-255	1
	带进位加	ADCS Rd, Rd, Rm	N Z C V	Rd:=Rd + Rm + C-bit		1
	SP值加8位立即数	ADD SP, SP, #<imm8>	不影响	SP := SP + imm8	范围：0-1020（字对齐）	1
	SP 所存储的地址	ADD Rd, SP, #<imm7>	不影响	Rd := SP + imm7	范围：0-508（字对齐）	1
	PC 所存储的地址	ADR Rd, <label>	不影响	Rd := label	标号范围：PC-PC+1020,字对齐	2
减法	寄存器低到低	SUBS Rd, Rn, Rm	N Z C V	Rd := Rn – Rm		1
	偏移量3位 立即数	SUBS Rd, Rn, #<imm3>	N Z C V	Rd := Rn – imm3	范围为 0-7	1
	偏移量8位 立即数	SUBS Rd, Rd, #<imm8>	N Z C V	Rd := Rd – imm8	范围为 0-255	1
	带借位减	SBCS Rd, Rd, Rm	N Z C V	Rd := Rd–Rm–借位 C		1
	SP 减7位立即数	SUB SP, SP, #<imm7>	不影响	SP := SP – imm7	范围为 0-508（字对齐）	1
	求负	RSBS Rd, Rn, #0	N Z C V	Rd := – Rn	同：NEGS Rd, Rn	1
乘	16位数乘法	MULS Rd, Rm, Rd	N Z	Rd := Rm * Rd	16b*16b，32b 结果在 Rd 中	*
比较	比较 Rn, Rm	CMP Rn, Rm	N Z C V	更新 Rn-Rm 的标志位	含高低寄存器	1
	与 –Rm 比较	CMN Rn, Rm	N Z C V	更新 Rn + Rm 的标志位		1
	与8位立即数比较	CMP Rn, #<imm8>	N Z C V	更新 Rn–imm8 的标志位	立即数范围：0-255	1
逻辑运算	与	ANDS Rd, Rd, Rm	N Z	Rd := Rd AND Rm		1
	异或	EORS Rd, Rd, Rm	N Z	Rd := Rd EOR Rm		1
	或	ORRS Rd, Rd, Rm	N Z	Rd := Rd OR Rm		1
	位清零	BICS Rd, Rd, Rm	N Z	Rd :=Rd AND NOT Rm		1
	取反移动	MVNS Rd, Rd, Rm	N Z	R d := NOT Rm		1
	测试位	TST Rn, Rm	N Z	更新 Rn&Rm 标志位		1
移位	逻辑左移	LSLS Rd, Rm, #imm5	N Z C	Rd := Rm << shift (0~31)	允许移 0-31 位。 移0位，不影响 C	1
		LSLS Rd, Rd, Rs	N Z C	Rd := Rd << Rs[7:0]	如果 Rs[7:0]为 0，则不影响 C	1
	逻辑右移	LSRS Rd, Rm, #imm5	N Z C	Rd := Rm >> shift(1~32)	允许移动 1-32 位,0 表示移 32 位	1
		LSRS Rd, Rd, Rs	N Z C	Rd := Rd >> Rs[7:0]	如果 Rs[7:0] 为 0，则不影响 C	1
	算术右移	ASRS Rd, Rm, #imm5	N Z C	Rd := Rm ASR shift(1~32)	允许移动 1-32 位,0 表示移 32 位	1
		ASRS Rd, Rd, Rs	N Z C	Rd := Rd ASR Rs[7:0]	如果 Rs[7:0] 为 0，则不影响 C	1
	向右循环移	RORS Rd, Rd, Rs	N Z C	Rd := Rd ROR Rs[7:0]	如果 Rs[7:0] 为 0，则不影响 C 标	1

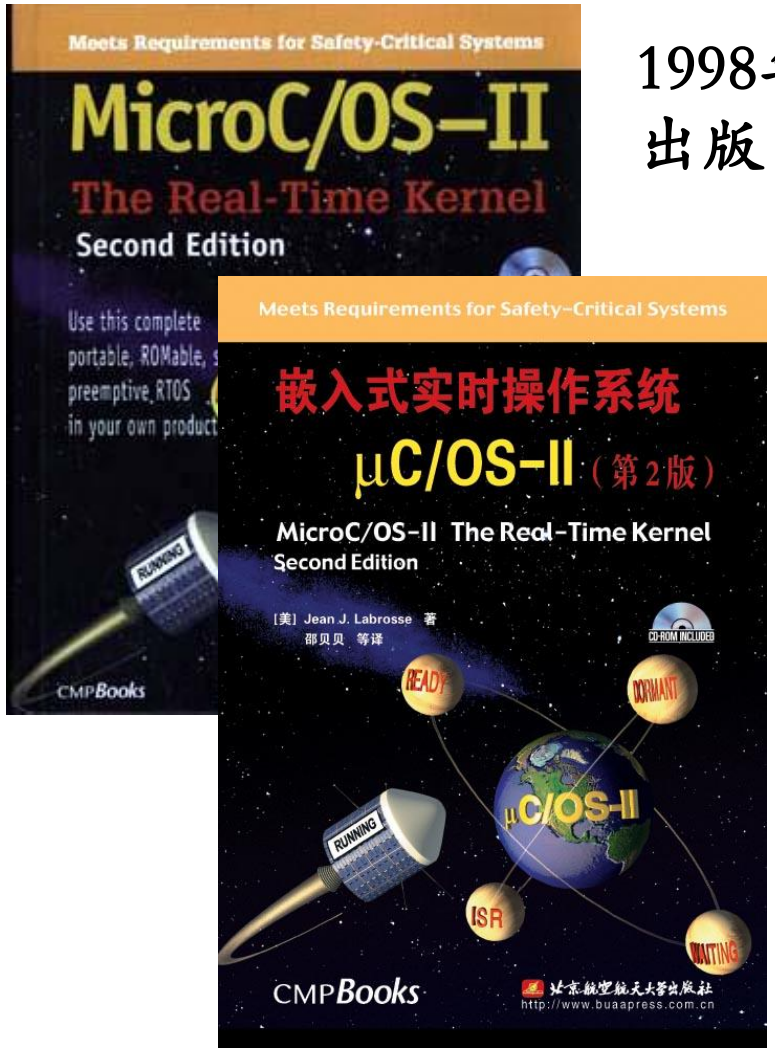
*： 1 到 16，取决于被乘数中 1 的数目，乘数与被乘数为 16 位数，不区分有符号还是无符号数。

Thumb2 v6 指令速查表2（不影响标志的指令）

运算		汇编指令	操作	说明	周期
读存储器到寄存器	带直接偏移量，字	LDR Rd, [Rn, #<imm5>]	Rd := [Rn + imm5]	范围 0-124 范围内 4 的倍数	2
	半字	LDRH Rd, [Rn, #<imm5>]	Rd:=ZeroExtend([Rn+imm5][15:0])	b31:16清零，范围 0-62，偶数	2
	字节	LDRB Rd, [Rn, #<imm5>]	Rd:= ZeroExtend([Rn + imm5][7:0])	b31:8清零，范围为 0-31	2
	带寄存器偏移量，字	LDR Rd, [Rn, Rm]	Rd := [Rn + Rm]		2
	半字	LDRH Rd, [Rn, Rm]	Rd := ZeroExtend([Rn + Rm][15:0])	b31:16 清零	2
	有符号半字	LDRSH Rd, [Rn, Rm]	Rd := SignExtend([Rn + Rm][15:0])	b31:16 = b15	2
	字节	LDRB Rd, [Rn, Rm]	Rd := ZeroExtend([Rn + Rm][7:0])	b31:8清零	2
	有符号字节	LDRSB Rd, [Rn, Rm]	Rd := SignExtend([Rn + Rm][7:0])	b31:8 = b7	2
	相对 PC	LDR Rd, <label>	Rd := [label]	标号范围：PC到 PC+1020（字对齐）	2
	相对 SP	LDR Rd, [SP, #<imm8>]	Rd := [SP + imm8]	范围0-1020内的 4 的倍数	2
	多寄存器，不含Rn	LDM Rn!, <loreglist>	读多存储器到多寄存器（不括 Rn）	每读一次后Rn+4，并回写Rn	2
	多寄存器，含Rn	LDM Rn, <loreglist>	读多存储器到多寄存器（包括 Rn）	Rn不回写	2
寄存器到存储器写	带立即数偏移量，字	STR Rd, [Rn, #<imm5>]	[Rn + imm] := Rd	范围 0-124内的 4 的倍数。	2
	半字	STRH Rd, [Rn, #<imm5>]	[Rn + imm][15:0] := Rd[15:0]	忽略Rd[31:16]，范围 0-62，偶数	2
	字节	STRB Rd, [Rn, #<imm5>]	[Rn + imm][7:0] := Rd[7:0]	忽略 Rd[31:8]，范围 0-31	2
	带寄存器偏移量，字	STR Rd, [Rn, Rm]	[Rn + Rm] := Rd		2
	半字	STRH Rd, [Rn, Rm]	[Rn + Rm][15:0] := Rd[15:0]	忽略 Rd[31:16]	2
	字节	STRB Rd, [Rn, Rm]	[Rn + Rm][7:0] := Rd[7:0]	忽略 Rd[31:8]	2
	相对 SP，字	STR Rd, [SP, #<imm8>]	[SP + imm] := Rd	imm 为 0-1020 范围内的 4 的倍数	2
	多寄存器到存储器	STM Rn!, <loreglist>	存储寄存器列表始	之后增加，回写加后的值到基址寄存器	2
入栈	推入堆栈	PUSH <loreglist>	将寄存器推入满降序堆栈	每次SP-4	1+N*
	带返回地址的推入	PUSH <loreglist+LR>	将 LR 和寄存器推入满降序堆栈		1+N*
出栈	弹出堆栈	POP <loreglist>	从满降序堆栈中弹出寄存器	每次Sp +4	1+N*
	弹出并返回	POP <loreglist+PC>	弹出IR寄存器到 PC 实现返回		3+N*
跳转	无条件跳转	B <label>	PC := label	Label：当前PC ±2KB 范围之内	2
	条件跳转	B{cond} <label>	如果 {cond}，则 PC := label	范围:当前指令的 –252B 到 +258B	#
	子程序调用	BL <label>	LR:=返回地址	32位指令，Lable范围：±16MB	3
	子程序返回	BLX LR	PC:=LR	也用于中断服务子程序返回	2
扩展	有符号，半字到字	SXTH Rd, Rm	Rd[31:0] := SignExtend(Rm[15:0]		1
	有符号，字节到字	SXTB Rd, Rm	Rd[31:0] := SignExtend(Rm[7:0])		1
	无符号，半字到字	UXTH Rd, Rm	Rd[31:0] := ZeroExtend(Rm[15:0]		1
	无符号，字节到字	UXTB Rd, Rm	Rd[31:0] := ZeroExtend(Rm[7:0])		1
顺序调整	字中的字节	REV Rd, Rm Rd	[31:24] :=Rm[7:0], Rd[23:16] :=Rm[15:8], Rd[15:8] :=Rm[23:16], Rd[7:0] :=Rm[31:24]		1
	两个半字中的字	REV16 Rd, Rm	Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:24] := Rm[23:16], Rd[23:16] :=Rm[31:24]		1
	低半字中的字节	REVSH Rd, Rm	Rd[15:8] := Rm[7:0]，Rd[7:0] := Rm[15:8]，Rd[31:16] := Rm[7] * &FFFF		1
改变状态	关中断	CPSID i	PRIMASK.PM 置1		1
	开中断	CPSIE i	PRIMASK.PM 清零		1
	软中断	SVC <imm8>	系统调用，软中断	指令中编码中带入8 位立即数，相当于 SWI	3
	设断点	BKPT <imm8>	设置调试断点，产生断点中断	指令中编码中带入8 位立即数	3
	读特殊寄存器	MRS <Rd>,<spec_reg>	读特殊寄存器到Rd	特殊寄存器spec_reg指：PRIMASK，CONTROL，	3
	写特殊寄存器	MSR <spec_reg>,<Rn>	Rd写入特殊寄存器	xPSR，MSP，PSP，IEPSR	3
空操作与提示	空操作	NOP	空操作	编码同 MOV R8, R8	1
	设置事件	WFI	等待中断	MCU制造商或用户提供相关函数	2
	等待事件	SEV	向多处理器系统发送事件信号	同 NOP 用于向上兼容和多处理器,M0+不用	1
	等待中断	WFE	多处理器时等待事件	同 NOP 用于向上兼容和多处理器,M0+不用	2
	Yield	YIELD	生成对其他线程的控制	同 NOP 用于向上兼容和多处理器,M0+不用	1

*： 1+N、3+N 个周期，N=表中寄存器数目

嵌入式实时操作系统 RTOS μC/OS-II



1998年
出版

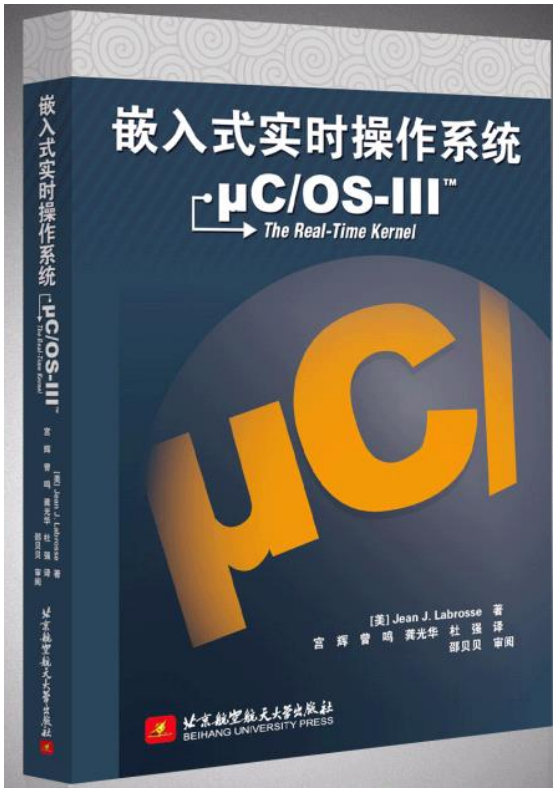
μC/OS-II特有的软件优先级调度查表算法，定位于低端单片机嵌入式应用的很好用的RTOS内核
适用于ARM Cortex-M0/M1/M0+

2003年
翻译出版

µC/OS-III ，适用于有硬件算法指令CLZ的ARM Cortex-M3/4/7

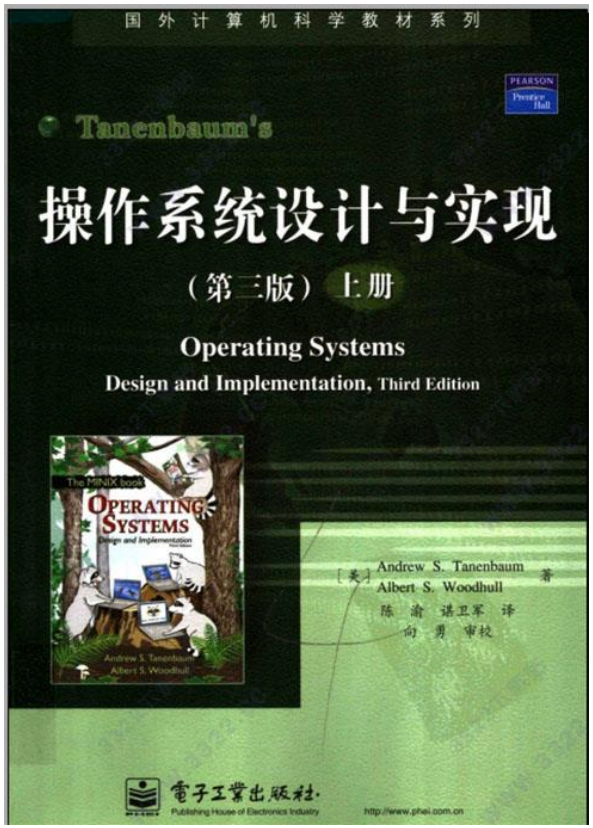
- ARM CortexM3/4/7 ,有以CLZ指令为代表的RTOS硬件算法指令，使用µC/OS-II不合理。应该使用

µC/OS-III



2013年

操作系统的权威教材



本书分上下册，上册讲操作系统原理，下册是Minix代码作者Andrew S．Tanenbaum曾获MIT理学学士学位和加周大学伯克利分校的哲学博士学位，现为荷兰阿姆斯特丹Vrije大学的计算机科学系当教授。他从70年代初一直参与Unix的开发，并在大学开操作系统课。16年后，Unix5有了商业价值，不准许他讲了。为了上课，1987年，他编写了MINIX，一个用于操作系统教学的类UNIX的小操作系统。以及后来的Amoeba分布式操作系统，是一个高性能的微内核分布式操作系统。在因特网上免费得到MINIX及Amoeba，用于教学和研究。

著名的技术作家、教育家和研究者，IEEE高级会员、ACM高级会员、荷兰皇家艺术和科学院院士、1994年ACM Karl V. Karlstrom杰出教育奖、1997年ACM计算机科学教育杰出贡献奖、2002年Texty卓越教材奖、第10届ACM操作系统原理研讨会杰出论文奖

小结

- ARM营销模式的创新结束了MC的U群雄割据，统一了CPU开发平台成为嵌入式应用的主流
- 精心规划的Cortex，系列兼容，无后顾之忧
- 良好的开发环境，但隔离了ARM内核，不利于基础教学
- 学生在校期间最好能把最基础的ARM搞清楚(M0＋或软核M1)
- 精心设计的指令系统很值得欣赏
- ARM仅为内核，给MCU和ASIC设计者留有扩展空间
- Cortex-M有现成的嵌入式实时操作系统可以使用

参考文献

- ARM System-on-Chip Architecture (2nd Edition)
- ARMv6-M Architecture Reference Manual
- ARMv7-M Architecture Reference Manual
- Cortex™-M0+ Devices Generic User Guide
- Cortex™-M0+ Technical Reference Manual
- ARM® Compiler toolchain Version 5.03 Using the Assembler
- KL25 Sub-Family Reference Manual
- KE06 Sub-Family Reference Manual
- Joseph Yiu ARM Cortex-M3与Cortex-M4权威指南（第3版)
- AMBA® 3 AHB-Lite Protocolv1.0 Specification