ARM

Weng Kai
2016 Spring

ARM

ARM公司

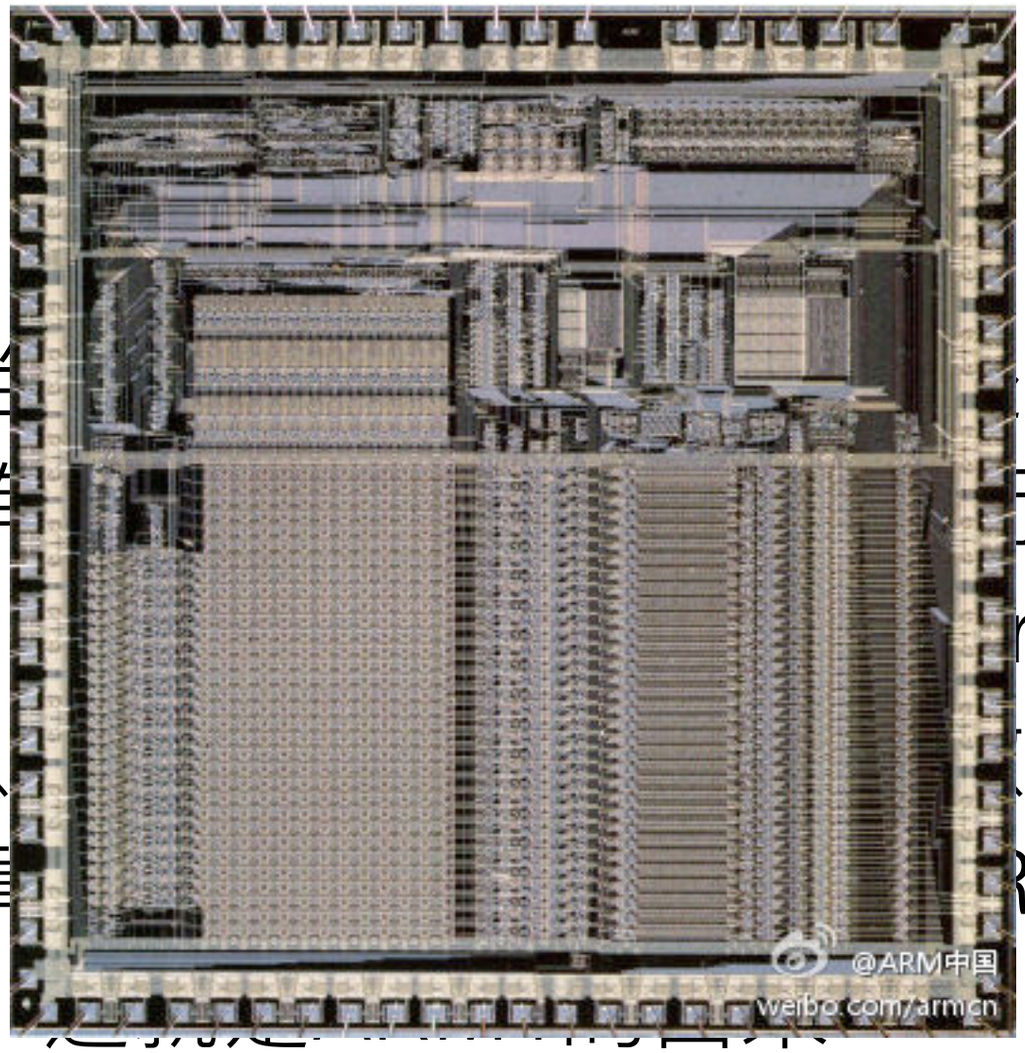
- ARM是Advanced RISC Machines的缩写，它是一家总部位于英国的微处理器行业知名企业，该企业设计了大量高性能、廉价、耗能低的RISC处理器。

- ARM公司的特点是只设计芯片，而不生产。它将技术授权给世界上许多著名的半导体、软件和OEM厂商，并提供服务。

ARM

- Acorn电脑当时考察了几种市面上的处理器，由于种种原因都没有最终使用，于是开始自行研发。1985年4月26日，Roger Wilson和Steve Furber设计出了第一代32位、6MHz的处理器，用它做出了一台RISC指令集的计算机，简称ARM（Acorn RISC Machine）。这就是ARM1的由来

ARM



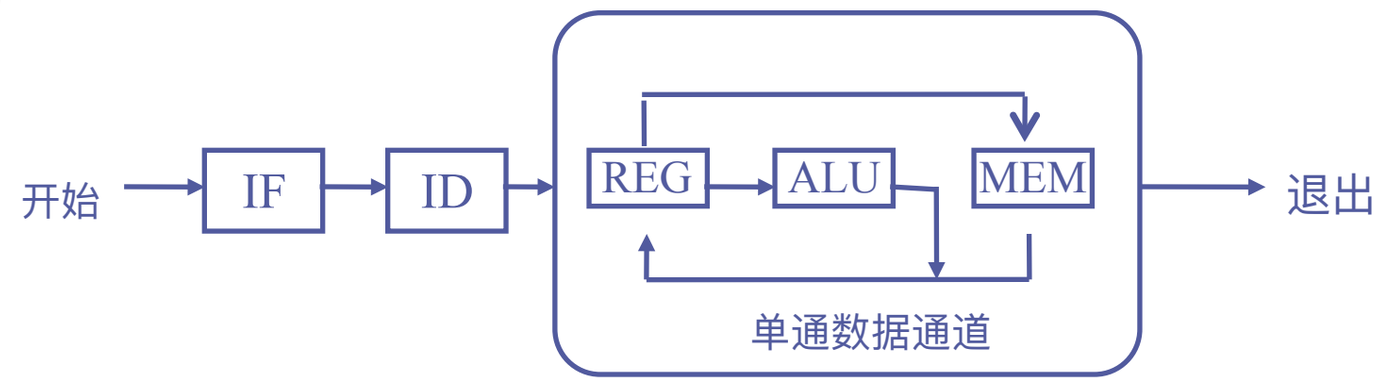
- Acorn电脑当[illegible]理器，由于种种原因都没有[illegible]行研发。1985年4月26日，[illegible]rber设计出了第一代32位、[illegible]出了一台RISC指令集的计算[illegible]RISC Machine）。[illegible]

ARM架构的特征

- Load/Store架构
- 大量的寄存器，都可用于多种用途
- 三地址指令（两个源操作数寄存器和结果寄存器独立设定）
- 每条指令都可以条件执行，包含非常强大的多寄存器Load和Store指令
- 能在单时钟周期内完成操作数的移位操作
- 能过协处理器指令集来扩展ARM指令集，包括在编程模式下增加了新的寄存器和数据类型
- 在Thumb体系结构中以高密度16位压缩形式表示指令集

Load/Store架构

- 用来简化CPU设计而且能改善性能
  - 内存墙：CPU越来越比内存快，内存访问拖慢了CPU，限制了编译器的优化能力
  - 修改指令集，使得大多数指令与内存无关，数据处理指令只能访问寄存器
    - 把数据装入寄存器（load）
    - 处理数据（计算）
    - 把数据存会内存（store）
  - 寄存器越多效率越高
- 寄存器/存储器架构
  - 数据处理指令可以访问内存或寄存器
  - 相对于较慢的CPU速度（如低于50MHz），内存墙不是非常明显



ARM：Load/Store结构

ARM架构的寄存器

- 37个寄存器
  - 31个通用32位寄存器
  - 6个状态寄存器，R16－CPSR（当前程序状态寄存器），5个SPSR（程序状态保存寄存器），用于当异常发生时保存CPSR的状态
    - 每一种处理器模式中，可见：15个通用寄存器(R0～R14，其中：R13－堆栈指针(sp)、R14－链接寄存器(lr))、R15：程序计数器PC，以及1～2个状态寄存器
- 可见的寄存器取决于处理器的模式
- 其它寄存器(the banked registers)在处理器状态切换时被屏蔽

ARM工作模式

- 模式切换的方法
  - 软件控制
  - 外部中断
  - 异常处理

ARM工作模式

处理器模式	说明	备注
用户 (usr)	正常程序执行模式	不能直接切换到其它模式
系统 (sys)	运行操作系统的特权任务	与用户模式类似，但具有可以直接切换到其它模式等特权
快中断 (fiq)	支持高速数据传输及通道处理	FIQ异常响应时进入此模式
中断 (irq)	用于通用中断处理	IRQ异常响应时进入此模式
管理 (svc)	操作系统保护模式	系统复位和软件中断响应时进入此模式
中止 (abt)	用于支持虚拟内存和/或存储器保护	在ARM7TDMI没有大用处
未定义 (und)	支持硬件协处理器的软件仿真	未定义指令异常响应时进入此模式

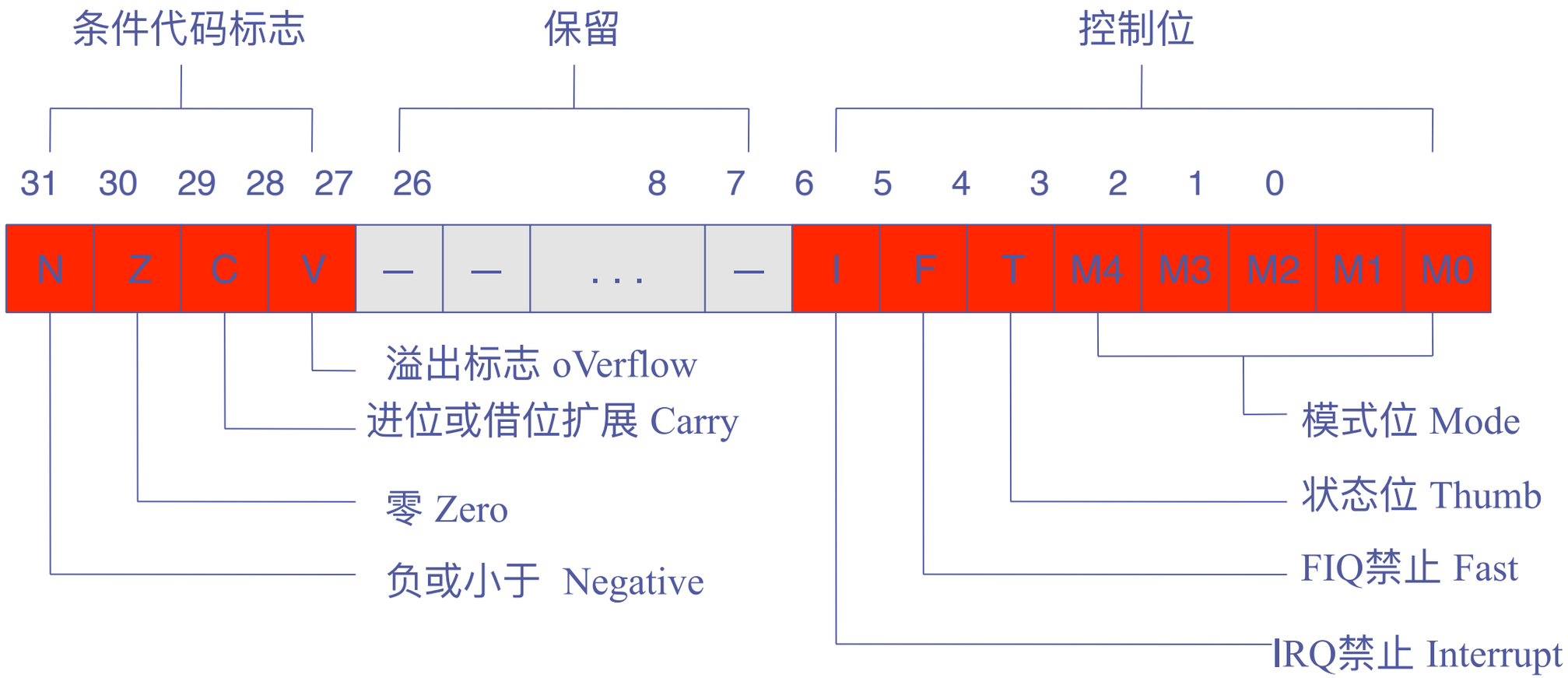
- 模式切换的方法
  - 软件控制
  - 外部中断
  - 异常处理

寄存器与模式

寄存器类别	寄存器在汇编中的名称	各模式下实际访问的寄存器						
		用户	系统	管理	中止	未定义	中断	快中断
通用寄存器和程序计数器	R0(a1)	R0						
	R1(a2)	R1						
	R2(a3)	R2						
	R3(a4)	R3						
	R4(v1)	R4						
	R5(v2)	R5						
	R6(v3)	R6						
	R7(v4)	R7						
	R8(v5)	R8						R8_fiq
	R9(SB,v6)	R9						R9_fiq
	R10(SL,v7)	R10						R10_fiq
	R11(FP,v8)	R11						R11_fiq
	R12(IP)	R12						R12_fiq
	R13(SP)	R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
	R14(LR)	R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
	R15(PC)	R15						
状态寄存器	CPSR	CPSR						
	SPSR	无		SPSR_abt	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

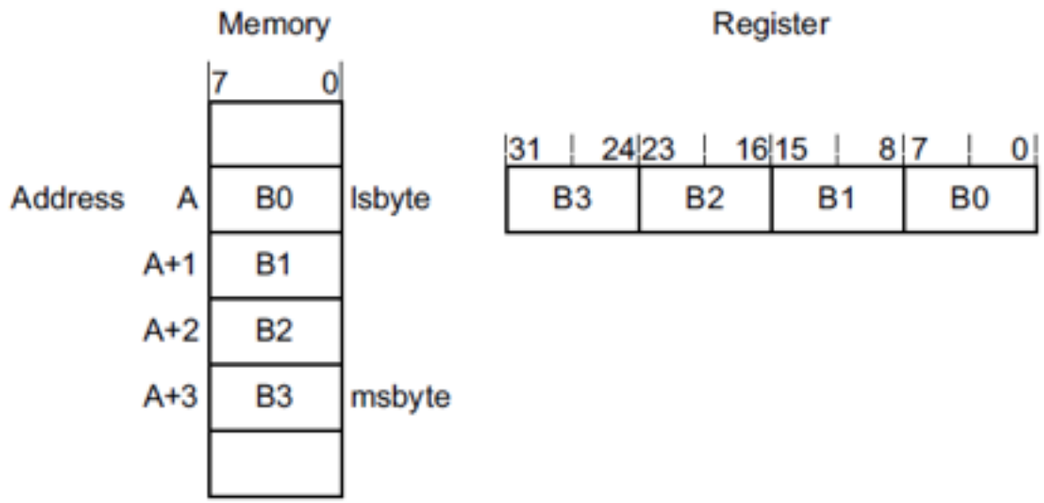
状态寄存器

CPSR寄存器的格式

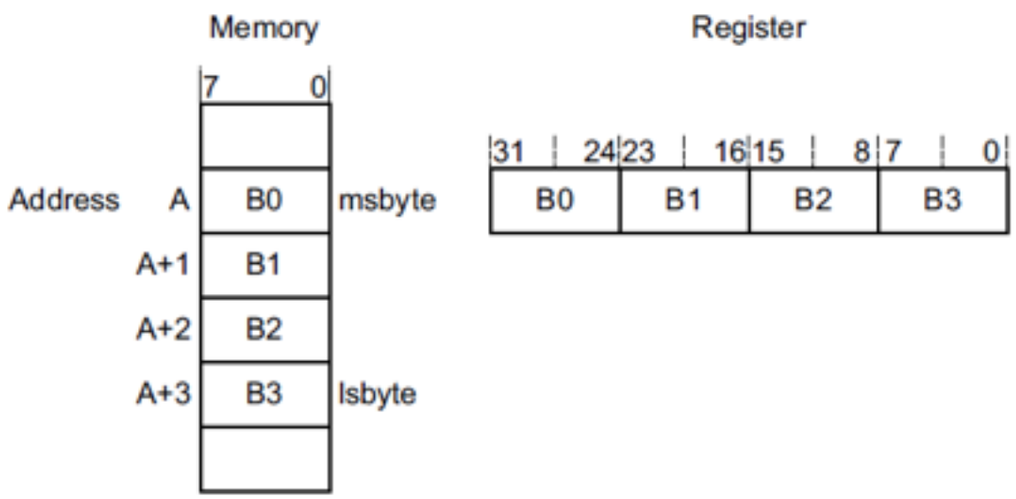


大小端

- 小端（低位在前）：
  从最低位的字节开始



- 大端（高位在前）：
  从最高位的字节开始



ARM的大小端

- 从前ARM是大端的
- 现在的Cortex-M中
  - 指令始终是小端的
  - 对私有的外围部件总线的读和写（load/store）始终是小端的
  - 数据：取决于芯片的具体实现，或是启动时的配置
    - 绝大多数厂家的实现是小端的

指令基本格式

- ARM是三地址指令格式，指令的基本格式如下：

```
<opcode> {<cond>} {S}    <Rd> ,<Rn>{,<operand2>}
```

- 其中<>号内的项是必须的，{}号内的项是可选的。各项的说明如下：

opcode：指令助记符；　　cond：执行条件；

S：是否影响CPSR寄存器的值；

Rd：目标寄存器；　　Rn：第1个操作数的寄存器；

operand2：第2个操作数；

- 如：

指令语法	目标寄存器(Rd)	源寄存器1(Rn)	源寄存器2(Rm)
ADD r3, r1, r2	r3	r1	r2

条件执行

```
<opcode> {<cond>} {S}    <Rd> ,<Rn>{,<operand2>}
```

- 使用条件码“cond”可以实现高效的逻辑操作(节省跳转和条件语句)，提高代码效率。

- 所有的ARM指令都可以条件执行，而Thumb指令只有B（跳转）指令具有条件执行功能。如果指令不标明条件代码，将默认为无条件（AL）执行。

条件码

操作码	条件助记符	标志	含义
0000	EQ	Z=1	相等
0001	NE	Z=0	不相等
0010	CS/HS	C=1	无符号数大于或等于
0011	CC/LO	C=0	无符号数小于
0100	MI	N=1	负数
0101	PL	N=0	正数或零
0110	VS	V=1	溢出
0111	VC	V=0	没有溢出
1000	HI	C=1,Z=0	无符号数大于
1001	LS	C=0,Z=1	无符号数小于或等于
1010	GE	N=V	有符号数大于或等于
1011	LT	N!=V	有符号数小于
1100	GT	Z=0,N=V	有符号数大于
1101	LE	Z=1,N!=V	有符号数小于或等于
1110	AL	任何	无条件执行 (指令默认条件)
1111	NV	任何	从不执行(不要使用)

条件执行

C代码：

```
if(a > b)
  a++;
else
  b++;
```

普通CPU

对应的汇编代码：

```
CMP   R0,R1       ;R0（a）与R1（b）比较
JLE   L1          ;若R0<=R1,则跳转
ADD   R1,R1,#1    ;b++
JMP   L2
L1:
ADD   R0,R0,#1    ;a++
L2:
```

ARM

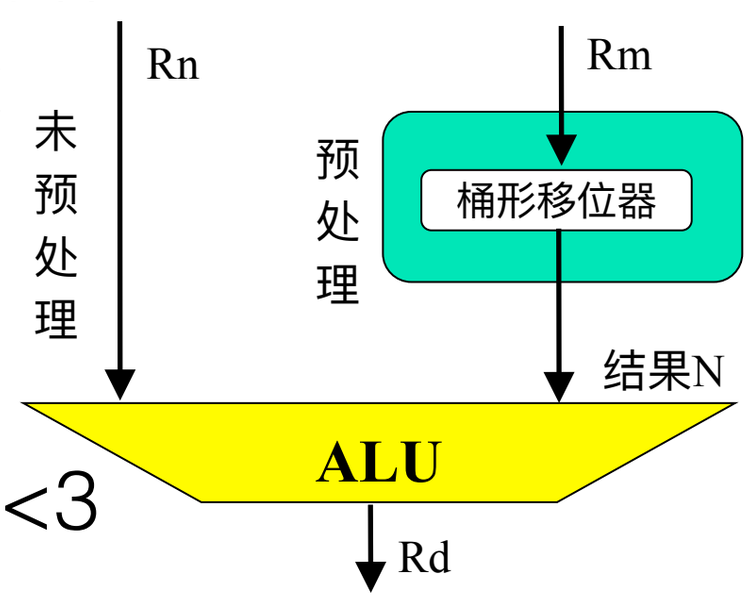
对应的汇编代码：

```
CMP   R0,R1       ;R0（a）与R1（b）比较
ADDHI R0,R0,#1    ;若R0>R1，则R0=R0+1
ADDLS R1,R1,#1    ;若R0≤R1，则R1=R1+1
```

移位操作

- ARM指令集对第2个源操作数，提供指令内的移位操作功能

- 将寄存器的移位结果作为操作数（移位操作不消耗额外的时间），但Rm值保持不变



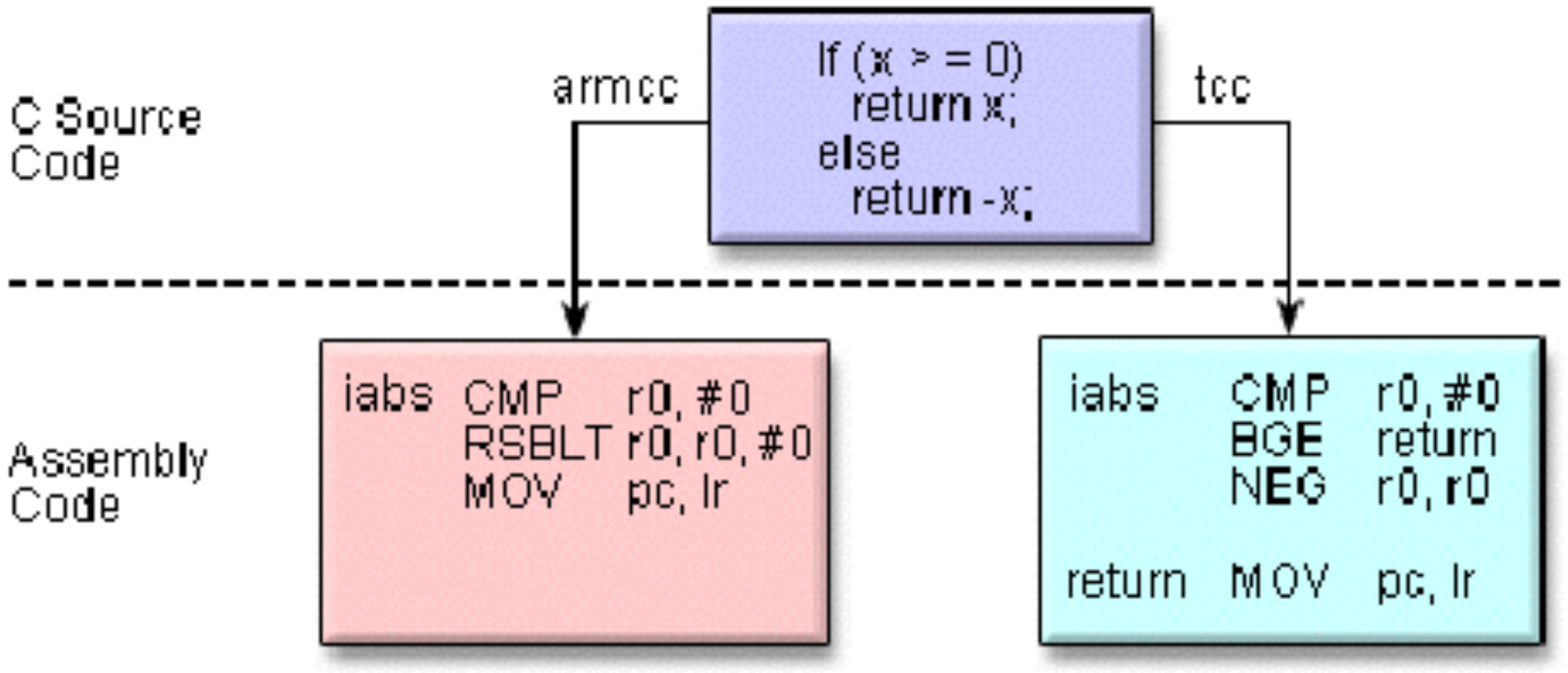
- 寄存器移位方式举例：
  - ADDR1,R1,R1,LSL #3　;R1=R1+R1<<3
  - SUB R1,R1,R2,LSR R3　;R1=R1-R2>>R3

ARM vs Thumb

- ARM指令是为资源丰富、高性能计算系统而优化设计的
  - 深度流水线处理器、高的时钟频率、宽的内存总线（如32位）
- 低端嵌入式计算系统是不同的
  - 较慢的时钟频率、浅的流水线
  - 不同的成本因素——比如代码大小更为重要、比特和字节运算很重要
- 修改了ARM指令集来适应低端嵌入式计算

- 1995：Thumb指令集
  - 16位指令，降低了内存需求但是也降低了性能
- 2003：Thumb-2指令集
  - 增加了一些32位指令
  - 改进了速度，只需很小的内存额外开销
- CPU根据是Thumb状态还是ARM状态来解码指令，状态由T位控制

ARM vs Thumb



Comparison

Code	Instruction	Size (Bytes)	Normalized
ARM	3	12	1.0
Thumb	4	8	0.67

ARM A32指令集

指令集

- 寻址方式
- 指令
- 函数调用约定

寻址方式

- 立即寻址
- 寄存器寻址
- 寄存器移位寻址
- 寄存器间接寻址
- 基址寻址
- 多寄存器寻址
- 堆栈寻址
- 相对寻址

立即寻址

- 立即寻址指令中的操作码字段后面的地址码部分即是操作数本身，也就是说，数据就包含在指令当中，取出指令也就取出了可以立即使用的操作数(这样的数称为立即数)。立即寻址指令举例如下：

- SUBS R0,R0,#1 ;R0减1，结果放入R0，并且影响标志位

- MOV R0,#0xFF000 ;将立即数0xFF000装入R0寄存器

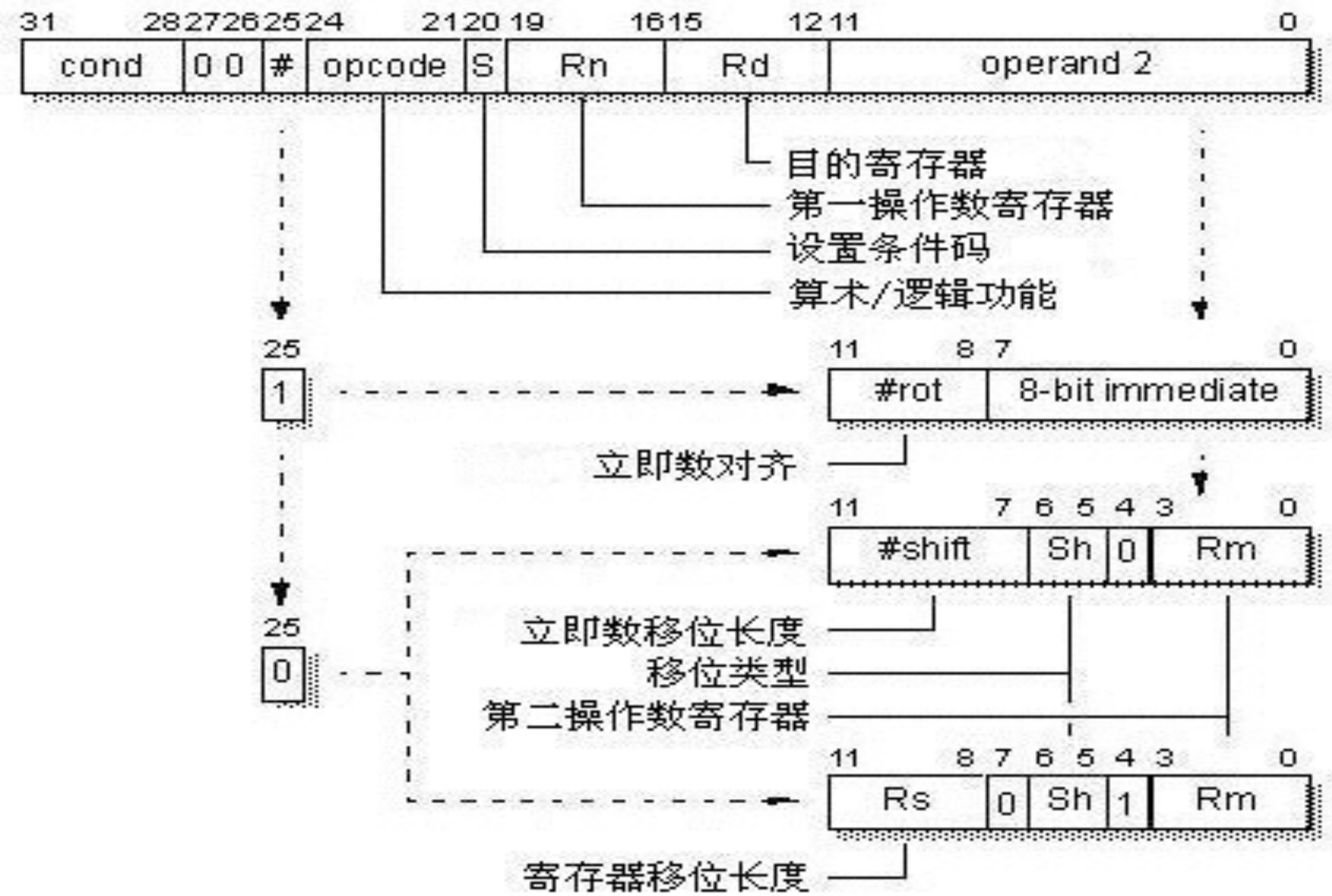
立即数范围

- 在ARM数据处理指令中，当参与操作的第二操作数为立即数型时，每个立即数都是采用一个8位的常数循环右移偶数位而间接得到。其中循环右移的位数由一个4位二进制的两倍表示，如果立即数记作<immediate>，8位常数记作immed_8，4位的循环右移值记作rotate_imm，有效的立即数是由一个8位的立即数循环右移偶数位得到。

- 因此有效立即数immediate可以表示成：

  - <immediate>=immed_8循环右移

ARM指令编码


31 28 27 26 25 24 21 20 19 16 15 12 11 0
cond
0 0
#
opcode
S
Rn
Rd
operand 2
目的寄存器
第一操作数寄存器
设置条件码
算术/逻辑功能
25
1
11 8 7 0
#rot
8-bit immediate
立即数对齐
11 7 6 5 4 3 0
#shift
Sh
0
Rm
25
0
立即数移位长度
移位类型
第二操作数寄存器
11 8 7 6 5 4 3 0
Rs
0
Sh
1
Rm
寄存器移位长度

寄存器寻址

- 操作数的值在寄存器中，指令中的地址码字段指出的是寄存器编号，指令执行时直接取出寄存器值来操作。

寄存器寻址指令举例如下：

- MOV R1,R2 ;将R2的值存入R1
- SUB R0,R1,R2 ;将R1的值减去R2的值，结果保存到R0

寄存器移位寻址

- 寄存器移位寻址是ARM指令集特有的寻址方式。当第2个操作数是寄存器移位方式时，第2个寄存器操作数在与第1个操作数结合之前，选择进行移位操作。寄存器移位寻址指令举例如下：

- MOV　R0,R2,LSL #3　　;R2的值左移3位，结果放入R0，即是R0=R2×8

- ANDS　R1,R1,R2,LSL R3　;R2的值左移R3位，然后和R1相“与”操作，结果放入R1

移位方式

- LSL：逻辑左移，空出的最低有效位用0填充。
- LSR：逻辑右移，空出的最高有效位用0填充。
- ASL：算术左移，由于左移空出的有效位用0填充，因此　它与LSL同义。
- ASR：算术右移，算术移位的对象是带符号数，移位过程中必须保持操作数的符号不变。如果源操作数是正数，空出的最高有效位用0填充，如果是负数用1填充。
- ROR：循环右移，移出的字的最低有效位依次填入空出的最高有效位。
- RRX：带扩展的循环右移。将寄存器的内容循环右移1位，空位用原来C标志位填充。

移位位数

- 移位位数可以用立即数方式或者寄存器方式给出,如下所示:

  - ADD　R3，R2，R1，LSR #2　；R3 <—R2 + R1÷4
  - ADD　R3，R2，R1，LSR R4　；R3 <—R2 + R1÷2R4

- 寄存器R1的内容分别逻辑右移2位、R4位（亦即R1÷4、R1÷2R4），再与寄存器R2的内容相加，结果放入R3中。

寄存器间接寻址

- 寄存器间接寻址指令中的地址码给出的是一个通用寄存器的编号，所需的操作数保存在寄存器指定地址的存储单元中，即寄存器为操作数的地址指针。寄存器间接寻址指令举例如下：
  - LDR R1,[R2] ;将R2指向的存储单元的数据读出保存在R1中
  - SWP R1,R1,[R2] ;将寄存器R1的值和R2指定的存储单元的内容交换
  - STR R0, [R1] ;/*[R1]←R0*/

基址寻址

- 基址寻址就是将基址寄存器的内容与指令中给出的偏移量（<4K）相加/减，形成操作数的有效地址。基址寻址用于访问基址附近的存储单元，常用于查表、数组操作、功能部件寄存器访问等。寄存器间接寻址是偏移量为0的基址加偏移寻址。

- 基址寻址指令举例如下(前索引寻址)：

  - LDR R2,[R3,#0x0C]　　;读取R3+0x0C地址上的存储单元的内容，放入R2

  - STR R1,[R0,#-4]!　　;先R0=R0-4，然后把R0的值寄存到保存到R1指定的存储单元

基址加偏址寻址

- 前变址模式：
  - LDR R0，[R1，#4]　　；R0←[R1＋4]
- 自动变址模式：
  - LDR R0，[R1，#4]!　　；R0←[R1＋4]、R1←R1＋4
- 后变址模式：
  - LDR R0，[R1]，#4　　；R0←[R1]、R1←R1＋4

基址加偏址寻址

- 基址寄存器的地址偏移可以是一个立即数，也可以是另一个寄存器，并且在加到基址寄存器前还可以经过移位操作，如下所示：
  - LDR　r0，[r1，r2]　　；r0<—mem32[r1+r2]
  - LDR　r0，[r1，r2，LSL #2]；r0<—[r1+r2*4]
- 但常用的是立即数偏移的形式，地址偏移为寄存器形式的指令很少使用。

多寄存器寻址

- 一次可传送几个寄存器值，允许一条指令传送16个寄存器的任何子集或所有寄存器。多寄存器寻址指令举例如下：

- LDMIA R1!,{R2-R7,R12} ;将R1指向的单元中的数据读出到R2～R7、R12中(R1自动加4)

- STMIA R0!,{R2-R7,R12} ;将寄存器R2～R7、R12的值保存到R0指向的存储; 单元中(R0自动加4)

多寄存器寻址

多寄存器寻址

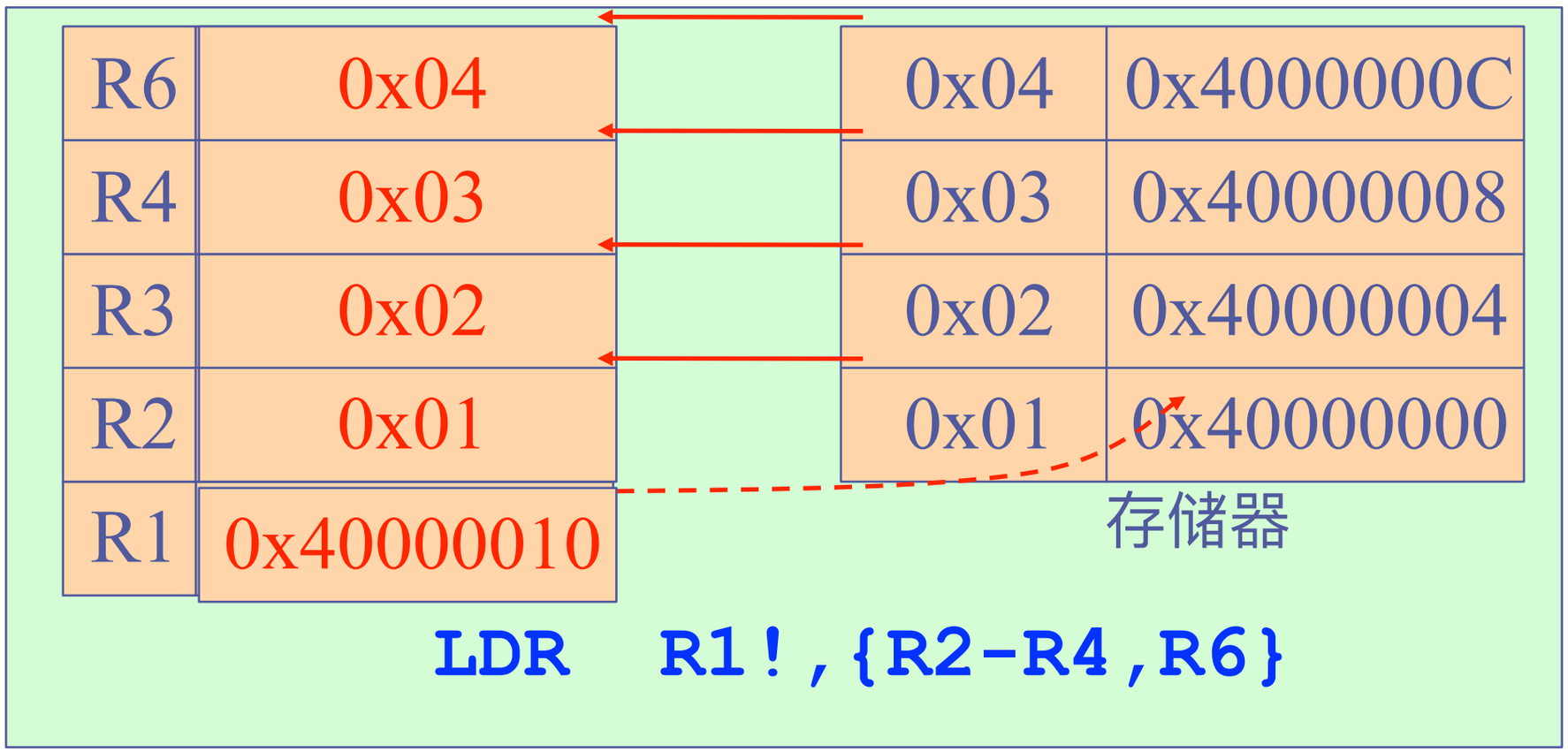
R6	0x??
R4	0x??
R3	0x??
R2	0x??
R1	0x40000000

0x04	0x4000000C
0x03	0x40000008
0x02	0x40000004
0x01	0x40000000

存储器

```
LDR   R1!,{R2-R4,R6}
```

多寄存器寻址


R6
0x04
0x04
0x4000000C
R4
0x03
0x03
0x40000008
R3
0x02
0x02
0x40000004
R2
0x01
0x01
0x40000000
R1
0x40000010
存储器
LDR R1!,{R2-R4,R6}

堆栈寻址

- 堆栈是一个按特定顺序进行存取的存储区，操作顺序为“后进先出”。堆栈寻址是隐含的，它使用一个专门的寄存器(堆栈指针)指向一块存储区域(堆栈)，指针所指向的存储单元即是堆栈的栈顶。存储器堆栈可分为两种：

  - 向上生长：向高地址方向生长，称为递增堆栈
  - 向下生长：向低地址方向生长，称为递减堆栈

堆栈方式

- 根据堆栈指针指向是否为空，以及堆栈生长方向，可以组合出四种类型的堆栈方式：

  - 满递增：堆栈向上增长，堆栈指针指向内含有效数据项的最高地址。指令如LDMFA、STMFA等；

  - 空递增：堆栈向上增长，堆栈指针指向堆栈上的第一个空位置。指令如LDMEA、STMEA等；

  - 满递减：堆栈向下增长，堆栈指针指向内含有效数据项的最低地址。指令如LDMFD、STMFD等；

  - 空递减：堆栈向下增长，堆栈指针向堆栈下的第一个空位置。指令如LDMED、STMED等。

相对寻址

- 相对（PC）寻址是基址寻址的一种变通。由程序计数器PC提供基准地址，指令中的地址码字段作为偏移量，两者相加后得到的地址即为操作数的有效地址。相对寻址指令举例如下：

```
        BL      SUBR1       ;调用到SUBR1子程序
        BEQ     LOOP        ;条件跳转到LOOP标号处
    ...
LOOP  MOV    R6,#1
    ...
SUBR1     ...
```

ARM指令编码

31–28	27–26	25	24–23	22	21	20	19–16	15–12	11–8	7–5	4	3–0	
Cond	0 0	I	Opcode			S	Rn	Rd	Operand 2				**Data Processing PSR Transfer**
Cond	0 0 0 0 0 0				A	S	Rd	Rn	Rs	1 0 0 1		Rm	**Multiply**
Cond	0 0 0 1 0			B	0 0		Rn	Rd	0 0 0 0	1 0 0 1		Rm	**Single Data Swap**
Cond	0 1	I	P	U	B	W	L	Rn	Rd	offset			**Single Data Transfer**
Cond	0 1 1	XXXXXXXXXXXXXXXXXXX									1	XXXX	**Undefined**
Cond	1 0 0	P	U	S	W	L	Rn	Register List					**Block Data Transfer**
Cond	1 0 1	L	offset										**Branch**
Cond	1 1 0	P	U	N	W	L	Rn	CRd	CP#	offset			**Coproc Data Transfer**
Cond	1 1 1 0	CP Opc					CRn	CRd	CP#	CP	0	CRm	**Coproc Data Operation**
Cond	1 1 1 0	CP Opc				L	CRn	Rd	CP#	CP	1	CRm	**Coproc Register Transfer**
Cond	1 1 1 1	ignored by processor											**Software Interrupt**

ARM指令

ARM指令分类

- 存储器访问指令
- 数据处理指令
- 乘法指令
- ARM分支指令
- 杂项指令

存储器访问———单寄存器加载

存储器访问——单寄存器加载

助记符	说明	操作	条件码位置
LDR Rd,addressing	加载字数据	Rd←[addressing], addressing索引	LDR{cond}
LDRB Rd,addressing	加载无符号字节数据	Rd←[addressing], addressing索引	LDR{cond}B
LDRT Rd,addressing	以用户模式加载字数据	Rd←[addressing], addressing索引	LDR{cond}T
LDRBT Rd, addressing	以用户模式加载无符号字节数据	Rd←[addressing], addressing索引	LDR{cond}BT
LDRH Rd, addressing	加载无符号半字数据	Rd←[addressing], addressing索引	LDR{cond}H
LDRSB Rd, addressing	加载有符号字节数据	Rd←[addressing], addressing索引	LDR{cond}SB
LDRSH Rd, addressing	加载有符号半字数据	Rd←[addressing], addressing索引	LDR{cond}SH

存储器访问——单寄存器存储

助记符	说明	操作	条件码位置
STR Rd, addressing	存储字数据	[addressing]←Rd, addressing索引	STR{cond}
STRB Rd,addressing	存储字节数据	[addressing]←Rd, addressing索引	STR{cond}B
STRT Rd,addressing	以用户模式存储字数据	[addressing]←Rd, addressing索引	STR{cond}T
STRBT Rd,addressing	以用户模式存储字节数据	[addressing]←Rd, addressing索引	STR{cond}BT
STRH Rd,addressing	存储半字数据	[addressing] ←Rd, addressing索引	STR{cond}H

- LDR/STR指令用于对内存变量的访问、内存缓冲区数据的访问、查表、外围部件的控制操作等。若使用LDR指令加载数据到PC寄存器，则实现程序跳转功能，这样也就实现了程序跳转。

- 所有单寄存器加载/存储指令可分为“字和无符号字节加载存储指令”和“半字和有符号字节加载存储指令。

存储器访问———多寄存器存取

- 多寄存器加载/存储指令可以实现在一组寄存器和一块连续的内存单元之间传输数据。LDM为加载多个寄存器；STM为存储多个寄存器。允许一条指令传送16个寄存器的任何子集或所有寄存器。它们主要用于现场保护、数据复制、常数传递等。

存储器访问——多寄存器存取

助记符	说明	操作	条件码位置
LDM{mode} Rn{!},reglist	多寄存器加载	reglist←[Rn...]，Rn回写等	LDM{cond}{mode}
STM{mode} Rn{!},reglist	多寄存器存储	[Rn...]←reglist,Rn回写等	STM{cond}{mode}

- 多寄存器加载/存储指令可以实现在一组寄存器和一块连续的内存单元之间传输数据。LDM为加载多个寄存器；STM为存储多个寄存器。允许一条指令传送16个寄存器的任何子集或所有寄存器。它们主要用于现场保护、数据复制、常数传递等。

存储器访问——寄存器和存储器交换

- SWP指令用于将一个内存单元(该单元地址放在寄存器Rn中)的内容读取到一个寄存器Rd中，同时将另一个寄存器Rm的内容写入到该内存单元中。使用SWP可实现信号量操作。

- 指令格式如下：

  - SWP{cond}{B}　Rd,Rm,[Rn]

- 其中，B为可选后缀，若有B，则交换字节，否则交换32位字；Rd用于保存从存储器中读入的数据；Rm的数据用于存储到存储器中，若Rm与Rd相同，则为寄存器与存储器内容进行互换；Rn为要进行数据交换的存储器地址，Rn不能与Rd和Rm相同。

存储器访问——寄存器和存储器交换

助记符	说明	操作	条件码位置
SWP Rd,Rm,Rn	寄存器和存储器字数据交换	Rd←[Rn]，[Rn]←Rm（Rn≠Rd或Rm）	SWP{cond}
SWPB Rd,Rm,Rn	寄存器和存储器字节数据交换	Rd←[Rn]，[Rn]←Rm（Rn≠Rd或Rm）	SWP{cond}B

- SWP指令用于将一个内存单元(该单元地址放在寄存器Rn中)的内容读取到一个寄存器Rd中，同时将另一个寄存器Rm的内容写入到该内存单元中。使用SWP可实现信号量操作。

- 指令格式如下：

  - SWP{cond}{B} Rd,Rm,[Rn]

- 其中，B为可选后缀，若有B，则交换字节，否则交换32位字；Rd用于保存从存储器中读入的数据；Rm的数据用于存储到存储器中，若Rm与Rd相同，则为寄存器与存储器内容进行互换；Rn为要进行数据交换的存储器地址，Rn不能与Rd和Rm相同。

数据处理———数据传送

- 注：当后缀S时，这些指令根据结果更新标志N和Z，在计算Operand2时更新标志C，不影响标志V。

数据处理——数据传送

助记符	说明	操作	条件码位置
MOV　Rd,operand2	数据传送	Rd←operand2	MOV{cond}{S}
MVN　Rd,operand2	数据非传送	Rd←(～operand2)	MVN{cond}{S}

- 注：当后缀S时，这些指令根据结果更新标志N和Z，在计算Operand2时更新标志C，不影响标志V。

数据处理———算术运算

- 注：这些指令带S时影响N，Z，C和V标志位。

数据处理———算术运算

助记符	说明	操作	条件码位置
ADD　Rd, Rn, operand2	加法运算指令	Rd←Rn+operand2	ADD{cond}{S}
SUB　Rd, Rn, operand2	减法运算指令	Rd←Rn-operand2	SUB{cond}{S}
RSB　Rd, Rn, operand2	逆向减法指令	Rd←operand2-Rn	RSB{cond}{S}
ADC　Rd, Rn, operand2	带进位加法	Rd←Rn+operand2+Carry	ADC{cond}{S}
SBC　Rd, Rn, operand2	带进位减法指令	Rd←Rn-operand2-(NOT)Carry	SBC{cond}{S}
RSC　Rd, Rn, operand2	带进位逆向减法指令	Rd←operand2-Rn-(NOT)Carry	RSC{cond}{S}

- 注：这些指令带S时影响N，Z，C和V标志位。

数据处理———逻辑运算指令

- 注：当后缀S时，这些指令根据结果更新标志N和Z，在计算Operand2时更新标志C，不影响标志V。

数据处理——逻辑运算指令

助记符	说明	操作	条件码位置
AND　Rd, Rn, operand2	逻辑与操作指令	Rd←Rn & operand2	AND{cond}{S}
ORR　Rd, Rn, operand2	逻辑或操作指令	Rd←Rn | operand2	ORR{cond}{S}
EOR　Rd, Rn, operand2	逻辑异或操作指令	Rd←Rn ^ operand2	EOR{cond}{S}
BIC　Rd, Rn, operand2	位清除指令	Rd←Rn & (~operand2)	BIC{cond}{S}

- 注：当后缀S时，这些指令根据结果更新标志N和Z，在计算Operand2时更新标志C，不影响标志V。

数据处理———比较指令

- 注：这些指令影响N，Z，C和V标志位。

数据处理——比较指令

助记符	说明	操作	条件码位置
CMP　Rn, operand2	比较指令	标志N、Z、C、V←Rn－operand2	CMP{cond}
CMN　Rn, operand2	负数比较指令	标志N、Z、C、V←Rn+operand2	CMN{cond}
TST　Rn, operand2	位测试指令	标志N、Z、C、V←Rn　&operand2	TST{cond}
TEQ　Rn, operand2	相等测试指令	标志N、Z、C、V←Rn　^operand2	TEQ{cond}

- 注：这些指令影响N，Z，C和V标志位。

数据处理———乘法指令

数据处理——乘法指令

助记符	说明	操作	条件码位置
MUL Rd,Rm,Rs	32位乘法指令	Rd←Rm*Rs（Rd≠Rm）	MUL{cond}{S}
MLA Rd,Rm,Rs,Rn	32位乘加指令	Rd←Rm*Rs+Rn（Rd≠Rm）	MLA{cond}{S}
UMULL RdLo,RdHi,Rm,Rs	64位无符号乘法指令	(RdLo,RdHi) ←Rm*Rs	UMULL{cond}{S}
UMLAL RdLo,RdHi,Rm,Rs	64位无符号乘加指令	(RdLo,RdHi) ←Rm*Rs+ (RdLo,RdHi)	UMLAL{cond}{S}
SMULL RdLo,RdHi,Rm,Rs	64位有符号乘法指令	(RdLo,RdHi) ←Rm*Rs	SMULL{cond}{S}
SMLAL RdLo,RdHi,Rm,Rs	64位有符号乘加指令	(RdLo,RdHi) ←Rm*Rs+ (RdLo,RdHi)	SMLAL{cond}{S}

分支指令

- 带状态切换的分支指令——BX指令，该指令可以根据跳转地址（Rm）的最低位来切换处理器状态。其跳转范围限制在当前指令的±32M字节地址内(ARM指令为字对齐，最低2位地址固定为0)。指令格式如下：
  - BX{cond}　Rm

分支指令

助记符	说明	操作	条件码位置
B label	分支指令	PC←label	B{cond}
BL label	带链接的分支指令	LR←PC-4，PC←label	BL{cond}
BX Rm	带状态切换的分支指令	PC←Rm，切换处理器状态	BX{cond}
BLX label	带链接和状态的分支		

- 带状态切换的分支指令——BX指令，该指令可以根据跳转地址（Rm）的最低位来切换处理器状态。其跳转范围限制在当前指令的±32M字节地址内(ARM指令为字对齐，最低2位地址固定为0)。指令格式如下：
  - BX{cond}　Rm

跳转地址 Rm[0]	跳转后	
	CPSR标志T位	处理器状态
0	0	ARM
1	1	Thumb

杂项指令

杂项指令

助记符	说明	操作	条件码位置
SWI　　immed_24	软中断指令	产生软中断，处理器进入管理模式	SWI{cond}
MRS　　Rd,psr	读状态寄存器指令	Rd←psr，psr为CPSR或SPSR	MRS{cond}
MSR　　psr_fields, Rd/#immed_8r	写状态寄存器指令	psr_fields←Rd/#immed_8r，psr为CPSR或SPSR	MSR{cond}

函数调用规范

用link寄存器

- 没有CALL/RET指令，BL指令在跳转时将当前PC+4存入R14
- 返回时将R14移入PC（R15）即返回

```
  BL SUBR

  …

SUBR:

  …

  MOV PC, R14
```

嵌套调用

- 函数要调用其他函数时需要保存R14

BL SUB1

…

SUB1

STMFA R13!, {R0-R2, R14}

BL SUB2

…

LDMFA R13!, {R0-R2, PC}

…

SUB2

…

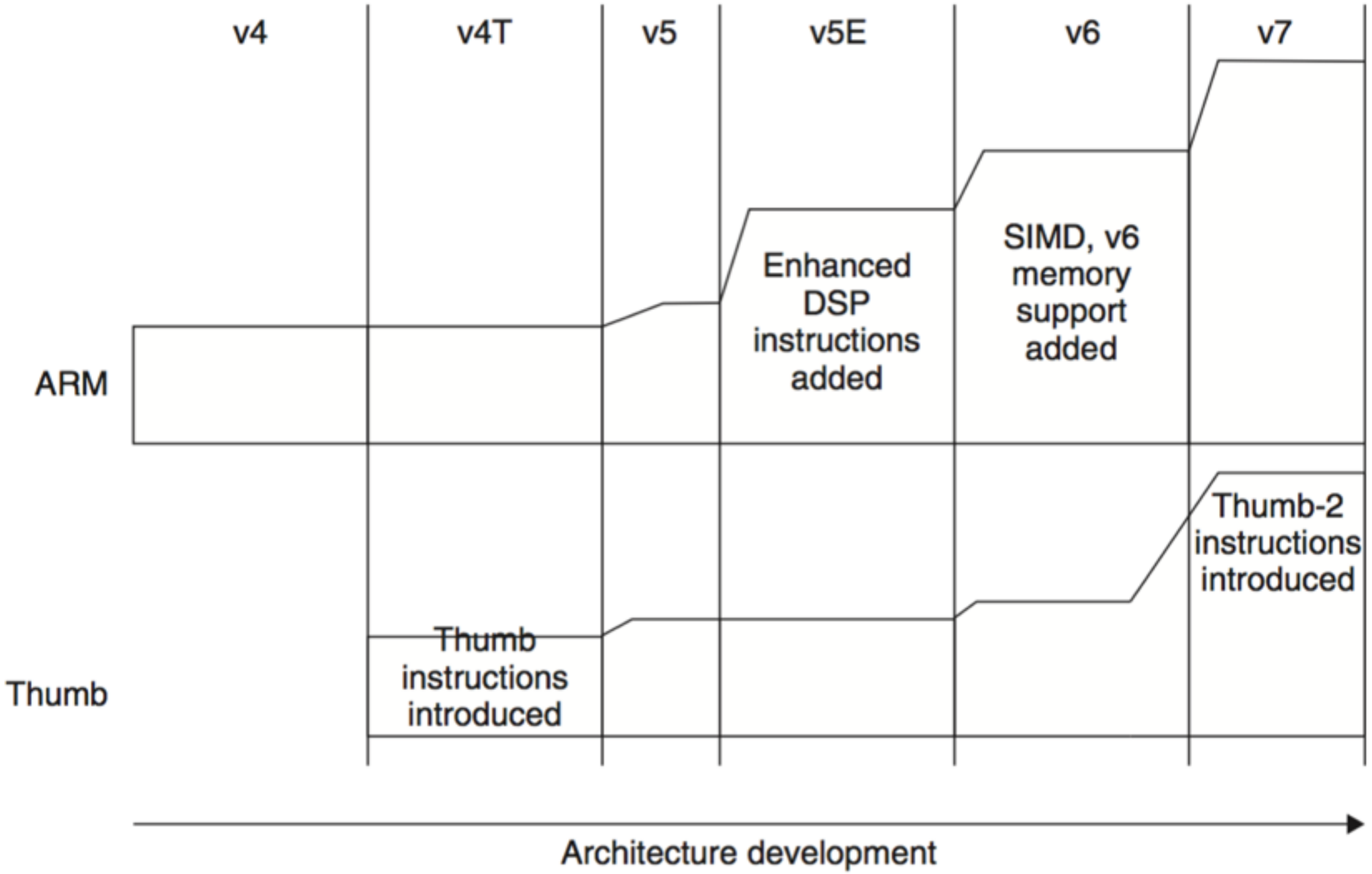
MOV PC, R14

函数调用规范

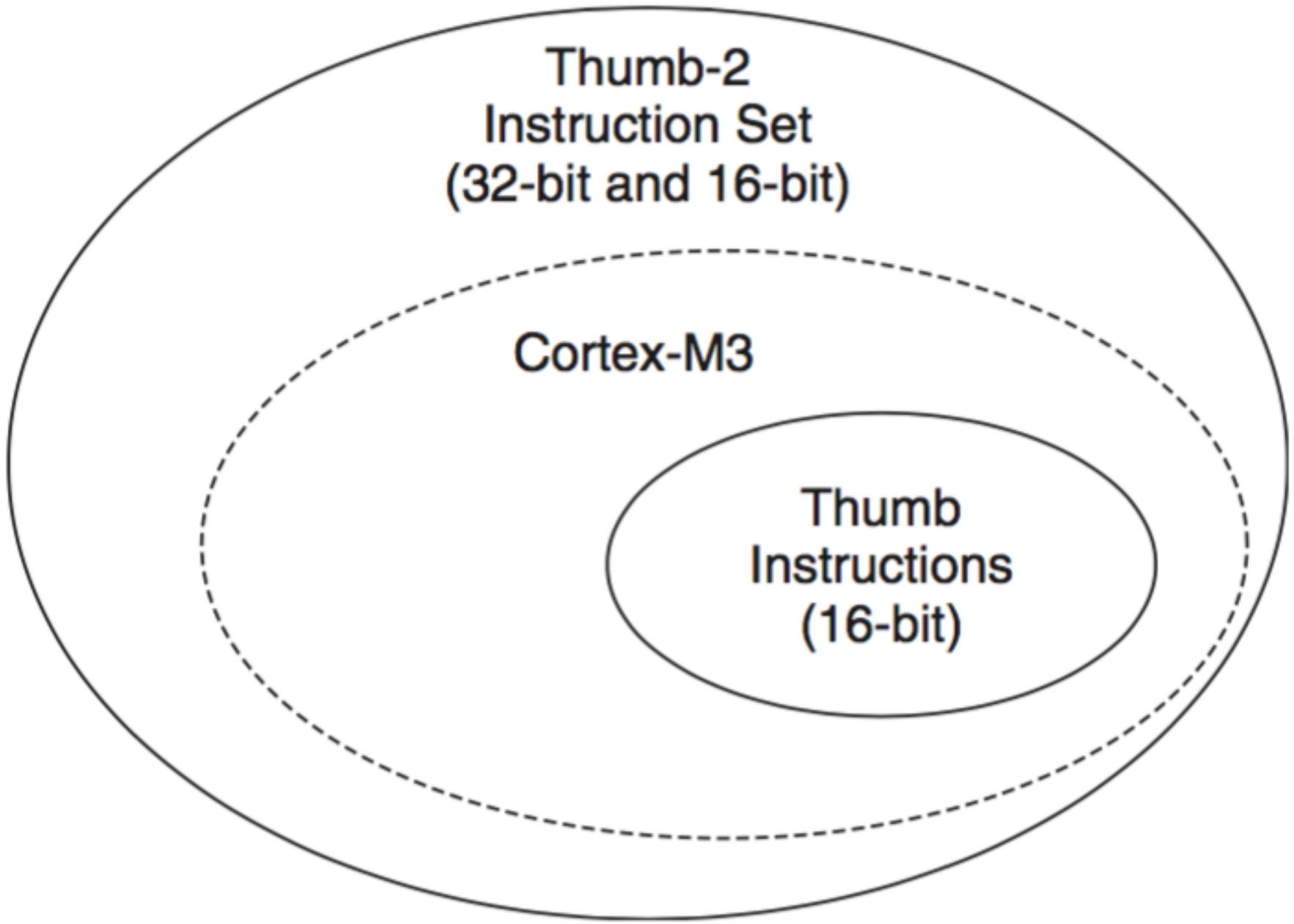
- 使用栈的规范：
  - 数据栈为FD（满递减）类型
- 在函数调用之间传递/返回参数
  - 4个以下参数，由R0～R3传递，其中arg1－R0
  - 大于4个参数时，通过堆栈传递
  - 返回结果存在R0中
  - 寄存器R4～R10用于本地变量或临时存储

Cortex-M ISA

ISA演进

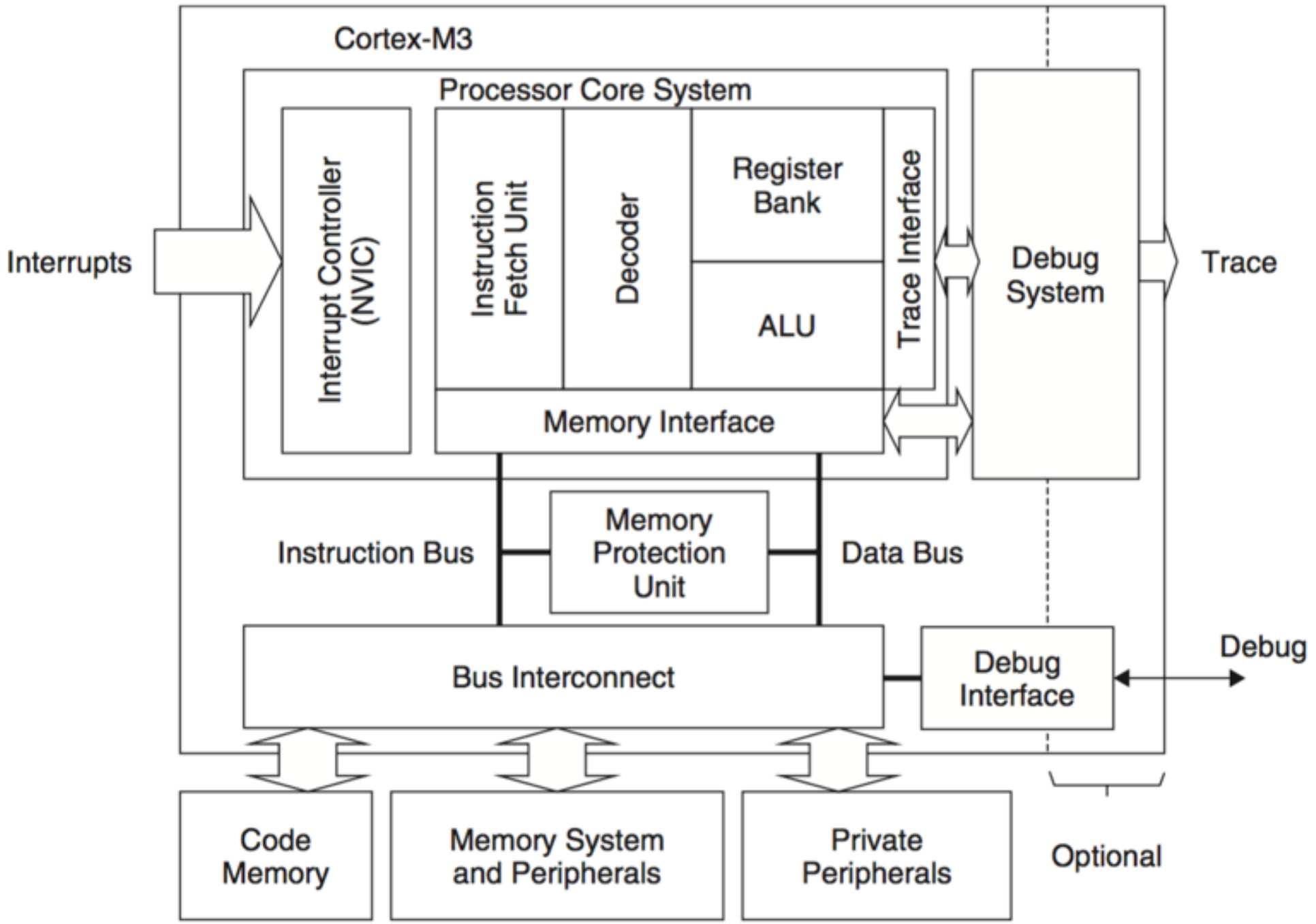

v4
v4T
v5
v5E
v6
v7
ARM
Enhanced DSP instructions added
SIMD, v6 memory support added
Thumb
Thumb instructions introduced
Thumb-2 instructions introduced
Architecture development

Thumb vs Thumb-2



A Brief on the Cortex-M Architecture

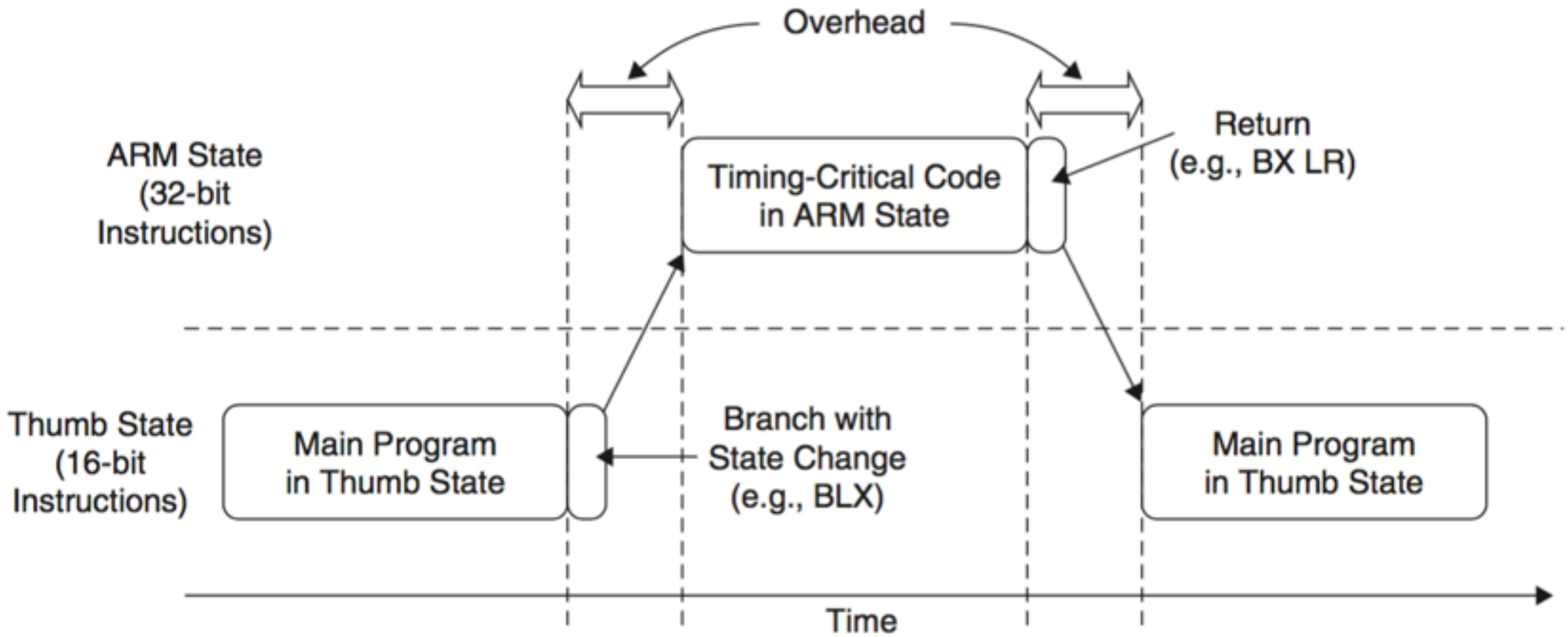
A Simple View



Registers

Name		*Functions (and Banked Registers)*	
R0		General-Purpose Register	Low Registers
R1		General-Purpose Register	
R2		General-Purpose Register	
R3		General-Purpose Register	
R4		General-Purpose Register	
R5		General-Purpose Register	
R6		General-Purpose Register	
R7		General-Purpose Register	
R8		General-Purpose Register	High Registers
R9		General-Purpose Register	
R10		General-Purpose Register	
R11		General-Purpose Register	
R12		General-Purpose Register	
R13 (MSP)	R13 (PSP)	Main Stack Pointer (MSP), Process Stack Pointer (PSP)	
R14		Link Register (LR)	
R15		Program Counter (PC)	
xPSR		Program Status Registers	Special Registers
PRIMASK		Interrupt Mask Registers	
FAULTMASK			
BASEPRI			
CONTROL		Control Register	

Thumb-2的优势



- ARM与Thumb并存时，切换存在额外开销

Program Status Registers

	31	*30*	*29*	*28*	*27*	*26:25*	*24*	*23:20*	*19:16*	*15:10*	*9*	*8*	*7*	*6*	*5*	*4:0*
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Exception Number				

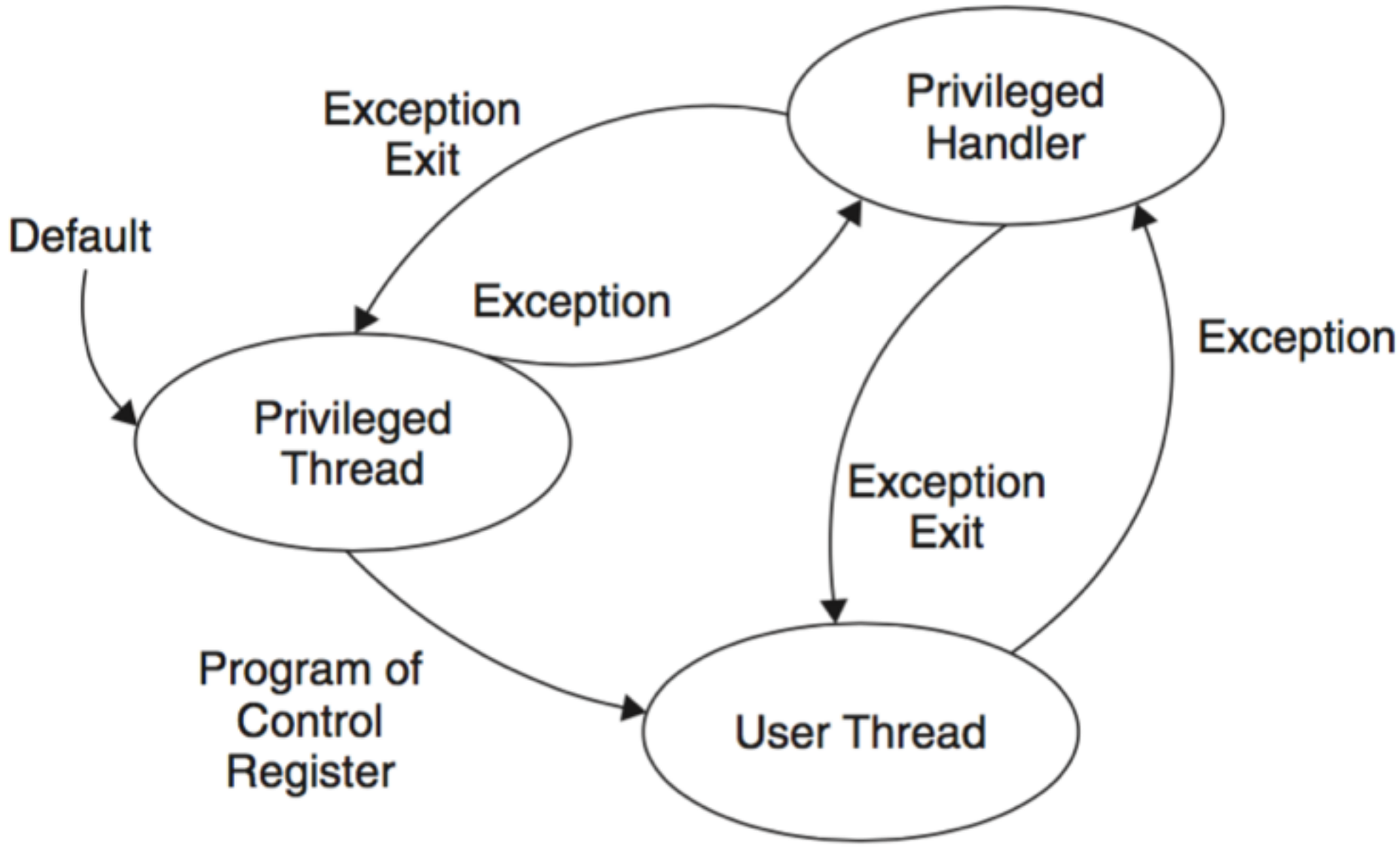
- MRS r0, PSR ; Read the combined program status word
- MSR PSR, r0 ; Write combined program state word

The Control Register

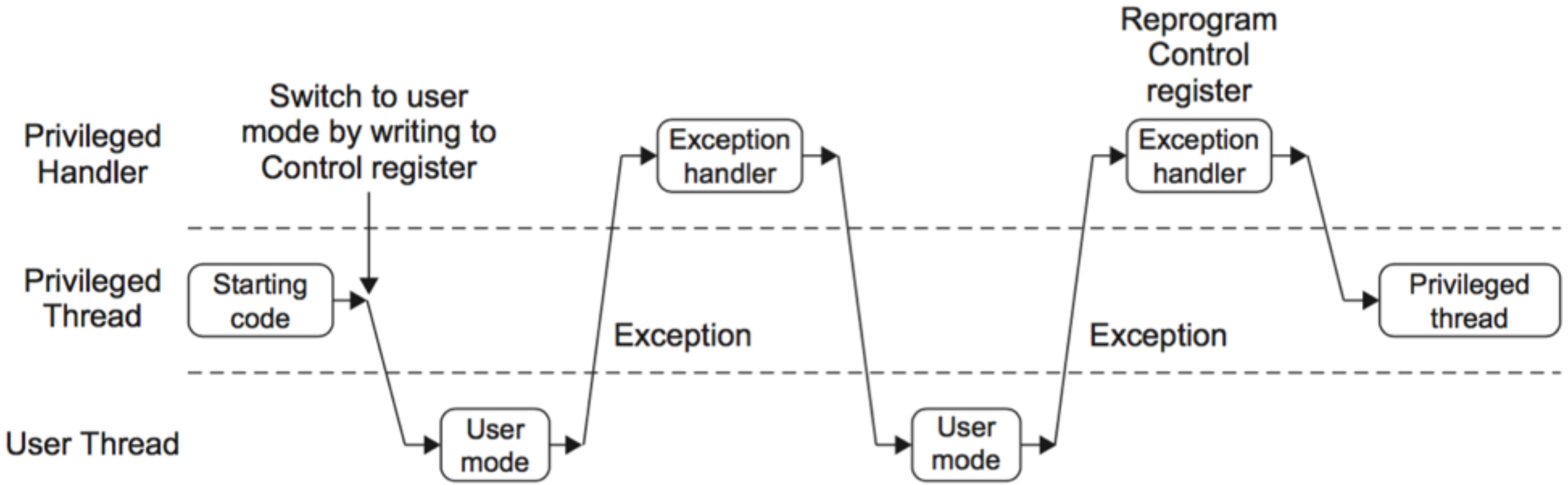
Bit	Function
CONTROL[1]	Stack status: 1 = Alternate stack is used 0 = Default stack (MSP) is used If it is in the Thread or base level, the alternate stack is the PSP. There is no alternate stack for handler mode, so this bit must be zero when the processor is in handler mode.
CONTROL[0]	0 = Privileged in Thread mode 1 = User state in Thread mode If in handler mode (not Thread mode), the processor operates in privileged mode.

- Thread mode：用户程序
- Handler mode：中断处理程序

Operation Models



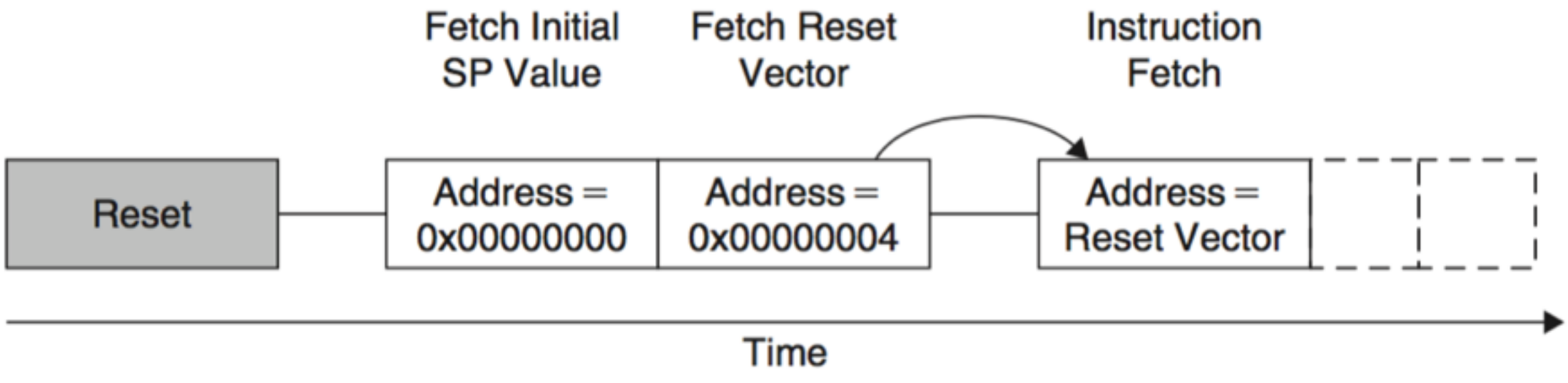
the Modes



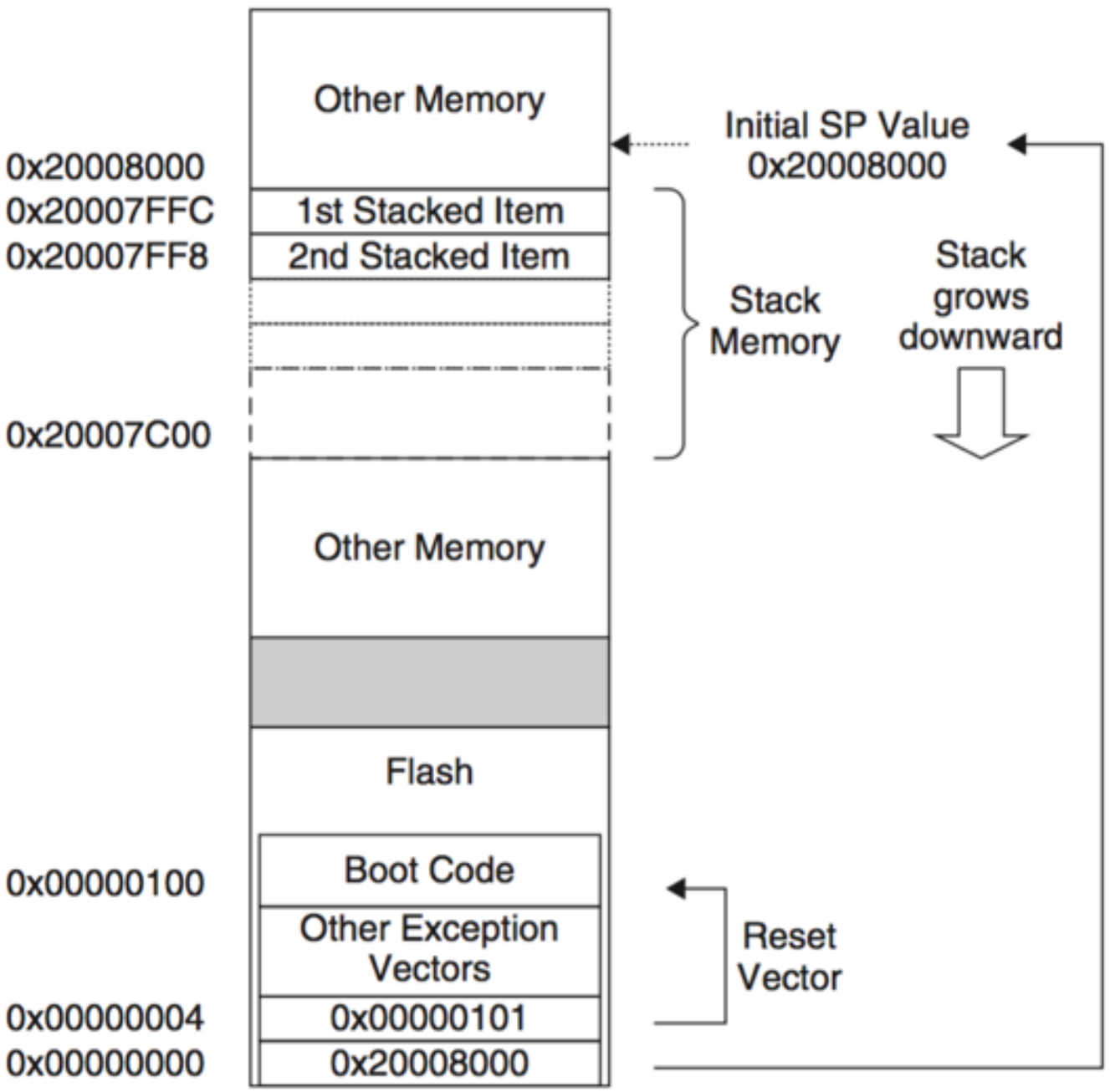
- ARM用异常表示中断和异常，中断是异常的一部分，表示从CPU外部来的异常。这与一般的术语体系是相反的。

Reset Sequence

- Address 0x00000000: Starting value of R13 (the stack pointer)
- Address 0x00000004: Reset vector (the starting address of program execution; LSB should be set to 1 to indicate Thumb state)



Initial Memory



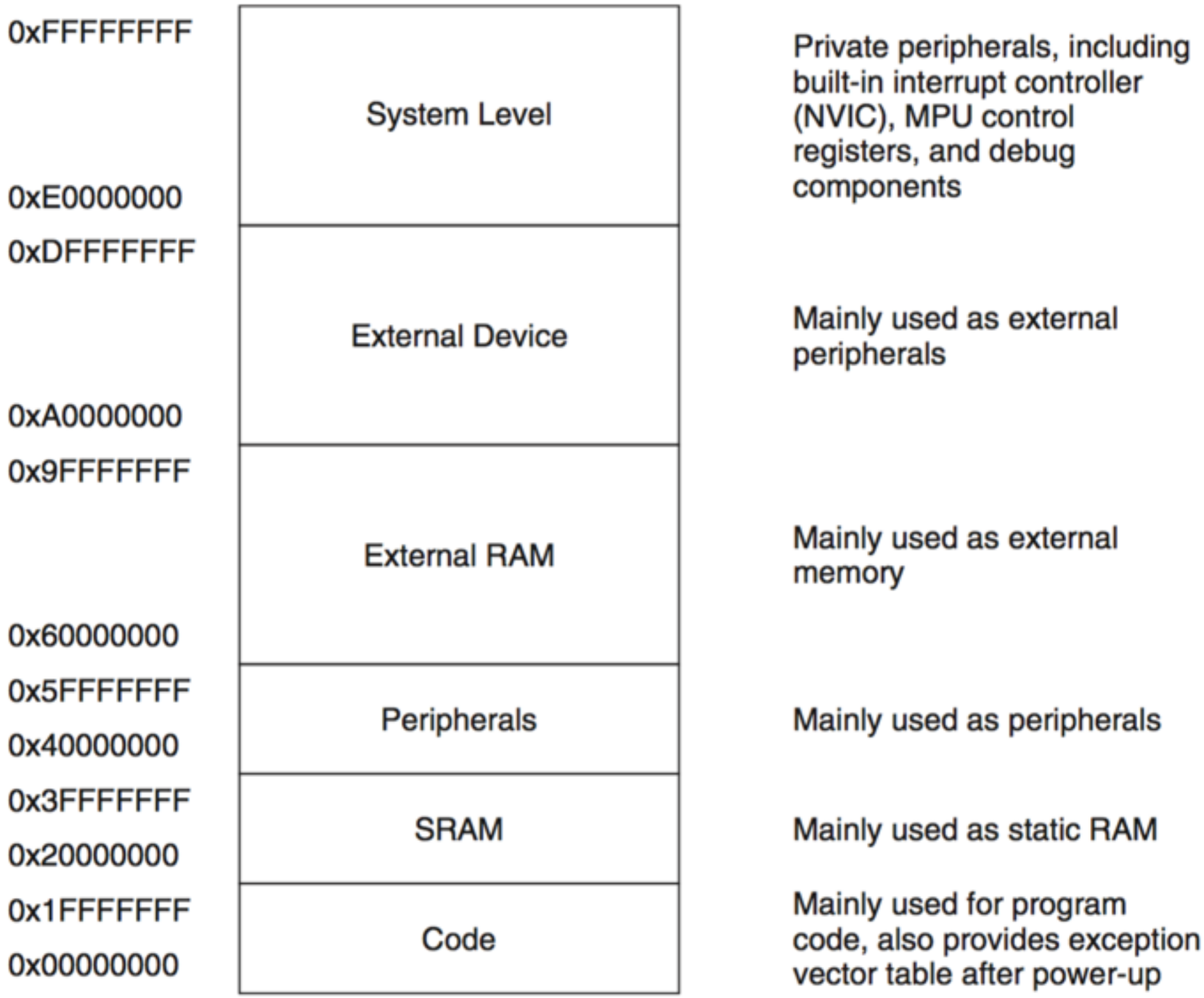
- 启动是一个中断，而不是从0地址开始执行
- 启动中断是中断向量表的第一项
- 在0地址的不是中断向量表的第一项
- 初始堆栈指针由0地址的值指定

其他架构的启动实现

- 启动从固定地址开始，通常是0地址
- 中断向量表如果从0地址开始，则启动从高地址开始，或0x100这样的地址
- 可能安排地址重新映射来解决中断向量表不可写的问题
- 初始堆栈指针由程序代码写入

- 启动是一个中断，而不是从0地址开始执行
- 启动中断是中断向量表的第一项
- 在0地址的不是中断向量表的第一项
- 中断向量表不可写
- 初始堆栈指针由0地址的值指定

Memory Map


0xFFFFFFFF
System Level
0xE0000000
0xDFFFFFFF
External Device
0xA0000000
0x9FFFFFFF
External RAM
0x60000000
0x5FFFFFFF
Peripherals
0x40000000
0x3FFFFFFF
SRAM
0x20000000
0x1FFFFFFF
Code
0x00000000
Private peripherals, including built-in interrupt controller (NVIC), MPU control registers, and debug components
Mainly used as external peripherals
Mainly used as external memory
Mainly used as peripherals
Mainly used as static RAM
Mainly used for program code, also provides exception vector table after power-up

Cortex-M ISA

Thumb-2 Support

- With the new Thumb-2 instruction support, some of the operations can be handled by either a Thumb instruction or a Thumb-2 instruction. For example, R0 R0 1 can be implemented as a 16-bit Thumb instruction or a 32-bit Thumb-2 instruction. With UAL, you can specify which instruction you want by adding suffixes:

```
ADDS   R0, #1 ; Use 16-bit Thumb instruction by default,for smaller size

ADDS.N R0, #1 ; Use 16-bit Thumb instruction (N=Narrow)

ADDS.W R0, #1 ; Use 32-bit Thumb-2 instruction (W=wide)
```

16-bit or 32-bit?

- In most cases, applications will be coded in C, and the C compilers will use 16-bit instructions if possible due to smaller code size. However, when the immediate data exceeds a certain range or when the operation can be better handled with a 32-bit Thumb-2 instruction, the 32-bit instruction will be used.

- 32-bit Thumb-2 instructions can be half word aligned:

  0x1000 : LDR r0,[r1] ; a 16-bit instructions (occupy 0x1000-0x1001)

  0x1002 : RBIT.W r0 ; a 32-bit Thumb-2 instruction (occupy 0x1002-0x1005)

Commonly Used Memory Access Instructions

- LDRB Rd, [Rn, #offset] Read byte from memory location Rn offset
- LDRH Rd, [Rn, #offset] Read half-word from memory location Rn offset
- LDR Rd, [Rn, #offset] Read word from memory location Rn offset
- LDRD Rd1,Rd2, [Rn, #offset] Read double word from memory location Rn offset
- STRB Rd, [Rn, #offset] Store byte to memory location Rn offset
- STRH Rd, [Rn, #offset] Store half word to memory location Rn offset
- STR Rd, [Rn, #offset] Store word to memory location Rn offset
- STRD Rd1,Rd2, [Rn, #offset] Store double word to memory location Rn offset

Multiple Memory Access Instructions

- LDMIA Rd!,<reg list> Read multiple words from memory location specified by Rd. Address Increment After (IA) each transfer (16-bit Thumb instruction).

- STMIA Rd!,<reg list> Store multiple words to memory location specified by Rd. Address Increment After (IA) each transfer (16-bit Thumb instruction).

- LDMIA.W Rd(!),<reg list> Read multiple words from memory location specified by Rd. Address increment after each read (.W specified it is a 32-bit Thumb-2 instruction).

- LDMDB.W Rd(!),<reg list> Read multiple words from memory location specified by Rd. Address Decrement Before (DB) each read (.W specified it is a 32-bit Thumb-2 instruction).

- STMIA.W Rd(!),<reg list> Write multiple words to memory location specified by Rd. Address increment after each read (.W specified it is a 32-bit Thumb-2 instruction).

- STMDB.W Rd(!),<reg list> Write multiple words to memory location specified by Rd. Address Decrement Before each read (.W specified it is a 32-bit Thumb-2 instruction).

Pre-Indexing Memory Access Instructions

- Pre-indexing load instructions for various sizes (word, byte, half word, and double word)
    - LDR.W Rd,[Rn, #offset]!
    - LDRB.W Rd,[Rn, #offset]!
    - LDRH.W Rd,[Rn, #offset]!
    - LDRD.W Rd1,Rd2,[Rn, #offset]!
- Pre-indexing load instructions for various sizes with sign extend (byte, half word)
    - LDRSB.W Rd,[Rn, #offset]!
    - LDRSH.W Rd,[Rn, #offset]!
- Pre-indexing store instructions for various sizes (word, byte, half word, and double word)
    - STR.W Rd,[Rn, #offset]!
    - STRB.W Rd,[Rn, #offset]!
    - STRH.W Rd,[Rn, #offset]!
    - STRD.W Rd1,Rd2,[Rn, #offset]!

Post-Indexing Memory Access Instructions

- Post-indexing load instructions for various sizes (word, byte, half word, and double word)
    - LDR.W Rd,[Rn], #offset
    - LDRB.W Rd,[Rn], #offset
    - LDRH.W Rd,[Rn], #offset
    - LDRD.W Rd1,Rd2,[Rn], #offset
- Post-indexing load instructions for various sizes with sign extend (byte, half word)
    - LDRSB.W Rd,[Rn], #offset
    - LDRSH.W Rd,[Rn], #offset
- Post-indexing store instructions for various sizes (word, byte, half word, and double word)
    - STR.W Rd,[Rn], #offset
    - STRB.W Rd,[Rn], #offset
    - STRH.W Rd,[Rn], #offset
    - STRD.W Rd1,Rd2,[Rn], #offset

Stack Operations

- PUSH {R0, R4-R7, R9} ; Push R0, R4, R5, R6, R7, R9 into stack memory
- POP {R2,R3} ; Pop R2 and R3 from stack
- PUSH {R0-R3, LR} ; Save register contents at beginning of subroutine
- POP {R0-R3, PC} ; restore registers and return

Special Registers

- MRS R0, PSR ; Read Processor status word into R0

- MSR CONTROL, R1 ; Write value of R1 into control register

Immediate Data

- MOV R0, #0x12 ; Set R0 to 0x12
- MOVW.W R0,#0x789A ; Set R0 to 0x789A
- MOVW.W R0,#0x789A ; Set R0 lower half to 0x789A
- MOVT.W R0,#0x3456 ; Set R0 upper half to 0x3456. Now R0=0x3456789A

LDR and ADR Pseudo Instructions

- LDR R0, address1 ; R0 set to 0x4001

... address1

0x4000: MOV R0, R1 ; address1 contains program code

- LDR R0, =address1 ; R0 set to 0x4000

... address1

0x4000: DCD 0x0 ; address1 contains data

- ADR R0, address1

... address1

0x4000: MOV R0, R1 ; address1 contains program code

- LDR obtains the immediate data by putting the data in the program code and uses a PC relative load to get the data into the register. ADR tries to generate the immediate value by adding or subtracting instructions (for example, based on the current PC value).

Arithmetic Instructions

- ADD, ADC
- ADDW: ADD register with 12-bit immediate value
- SUB, SBC, RSB.W
- MUL
- UDIV, SDIV
- SMULL RdLo, RdHi, Rn, Rm ;{RdHi,RdLo} = Rn * Rm
- SMLAL RdLo, RdHi, Rn, Rm ;{RdHi,RdLo} += Rn * Rm
- UMULL, UMLAL

Logic Operation Instructions

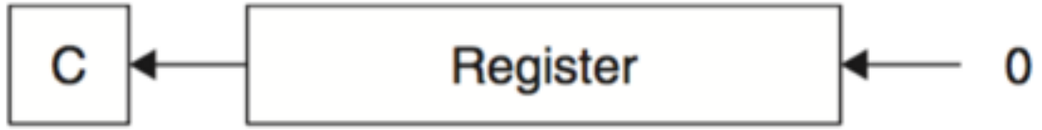
- AND, ORR
- BIC: Rd = Rd & (~Rn)
- ORN: Rd = Rn |(~Rd)
- EOR: Rd = Rd ^ Rn

Shift and Rotate Instructions

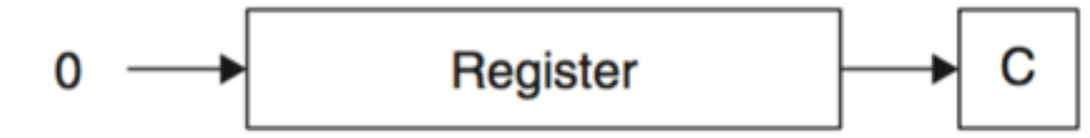
- ASR: Arithmetic shift right
- LSL: Logical shift left
- LSR: Logical shift right
- ROR: Rotate right
- RRX.W: Rotate right extended (including C)

Shift and Rotate

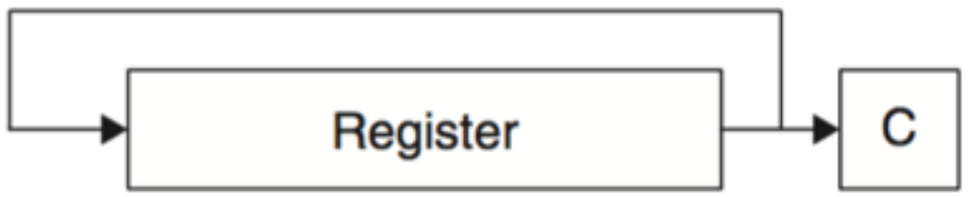
Logical Shift Left (LSL)



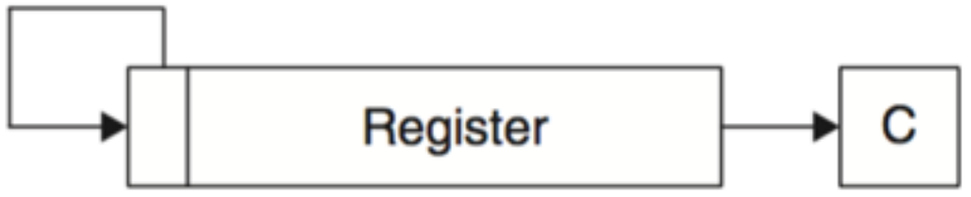
Logical Shift Right (LSR)



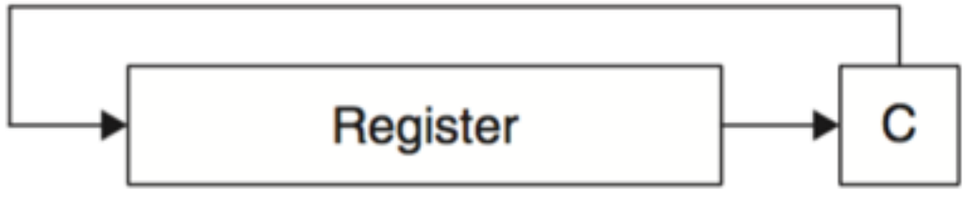
Rotate Right (ROR)



Arithmetic Shift Right (ASR)



Rotate Right Extended (RRX)



Why Is There Rotate Right But No Rotate Left?

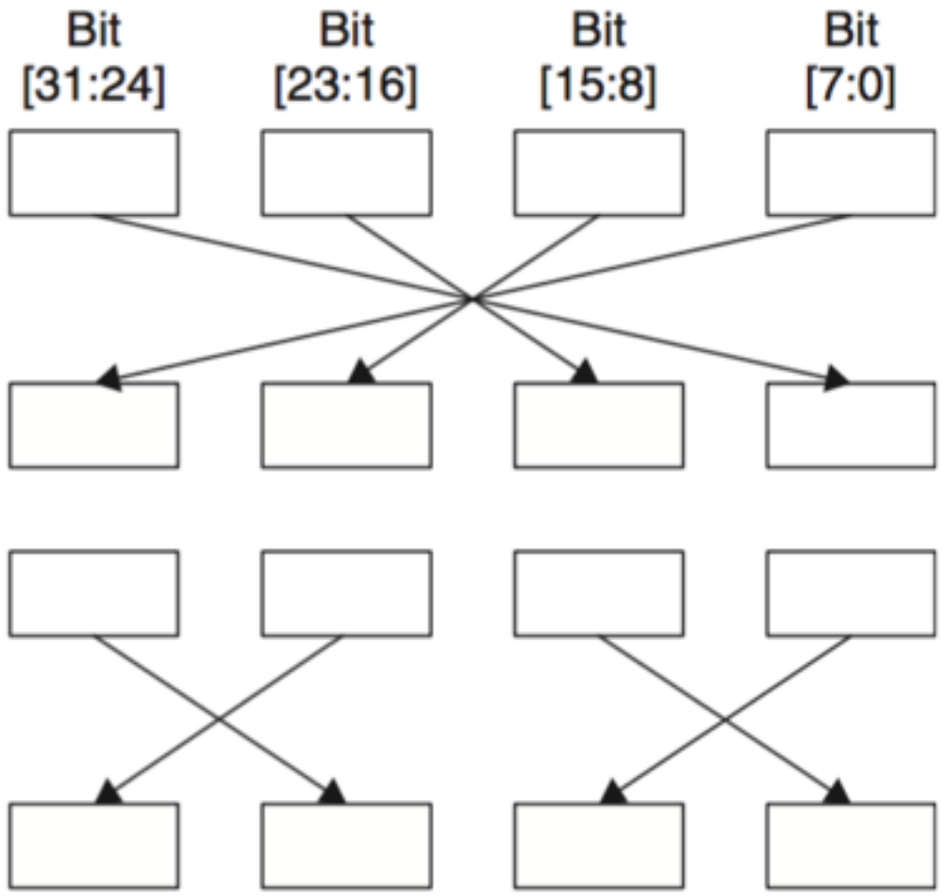
- The rotate left operation can be replaced by a rotate right operation with a different rotate offset. For example, a rotate left by 4-bit operation can be written as a rotate right by 28- bit instruction, which gives the same result and takes the same amount of time to execute.

Data Manipulation

- SXTB.W Rd, Rm ; Rd = signext(Rn[7:0])
- SXTH.W Rd, Rm ; Rd = signext(Rn[15:0])
- REV.W Rd, Rn ; Rd = rev(Rn)
- REV16.W <Rd>, <Rn> ; Rd = rev16(Rn)
- REVSH.W <Rd>, <Rn> ; Rd = revsh(Rn)

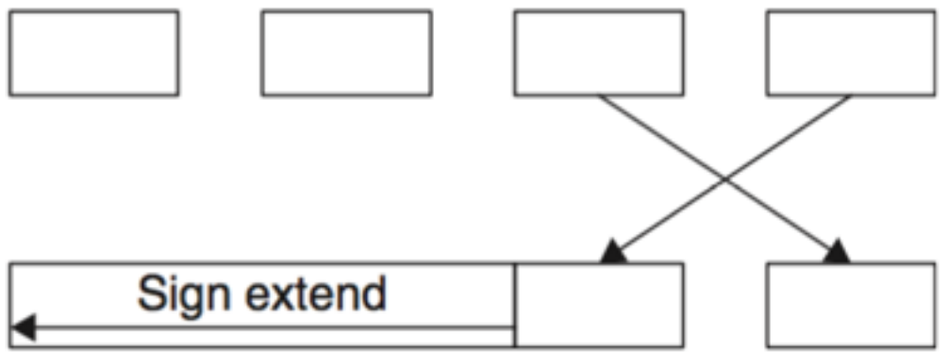
Reverse Operations

REV.W
(Reverse bytes in word)



REV16.W
(Reverse bytes in half word)

REVSH.W
(Reverse bytes in bottom
half word and sign extend results)



Bit Field Processing

- BFC.W Rd, Rn, #<width> Clear bit field within a register
- BFI.W Rd, Rn, #<lsb>, #<width> Insert bit field to a register
- CLZ.W Rd, Rn Count leading zero
- RBIT.W Rd, Rn Reverse bit order in register
- SBFX.W Rd, Rn, #<lsb>, #<width> Copy bit field from source and sign extend it
- UBFX.W Rd, Rn, #<lsb>, #<width> Copy bit field from source register

Call and Unconditional Branch

- B label ; Branch to a labeled address
- BX reg ; Branch to an address specified by a register
- BL label ; Branch to a labeled address and save return address in LR
- BLX reg ; Branch to an address specified by a register and save return address in LR.

Flags in ARM Processors

- Z (Zero) flag: This flag is set when the result of an instruction has a zero value or when a comparison of two data returns an equal result.

- N (Negative) flag: This flag is set when the result of an instruction has a negative value (bit 31 is 1).

- C (Carry) flag: This flag is for unsigned data processing—for example, in add (ADD) it is set when an overflow occurs; in subtract (SUB) it is set when a borrow did not occur (borrow is the invert of carry).

- V (Overflow) flag: This flag is for signed data processing; for example, in an add (ADD), when two positive values added together produce a negative value, or when two negative values added together produce a positive value.

Conditions

- EQ Equal
- NE Not equal
- CS/HS Carry set/unsigned higher or same
- CC/LO Carry clear/unsigned lower
- MI Minus/negative
- PL Plus/positive or zero
- VS Overflow
- VC No overflow
- HI Unsigned higher
- LS Unsigned lower or same
- GE Signed greater than or equal
- LT Signed less than
- GT Signed greater than
- LE Signed less than or equal
- AL Always (unconditional)

Flags

- PSR flags can be affected by the following:
  - 16-bit ALU instructions
  - 32-bit (Thumb-2) ALU instructions with the S suffix; for example, ADDS.W
  - Compare (e.g., CMP) and Test (e.g., TST, TEQ)
  - Write to APSR/PSR directly
- Most of the 16-bit Thumb arithmetic instructions affect the N, Z, C, and V flags. With 32-bit Thumb-2 instructions, the ALU operation can either change flags or not change flags.

CMP and TST

- The CMP (Compare) instruction subtracts two values and updates the flags (just like SUBS), but the result is not stored in any registers. CMP can have the following formats:
  - CMP R0, R1 ; Calculate R0 – R1 and update flag
  - CMP R0, #0x12 ; Calculate R0 – 0x12 and update flag
- A similar instruction is the CMN (Compare Negative). It compares one value to the negative (two's complement) of a second value; the flags are updated, but the result is not stored in any registers:
  - CMN R0, R1 ; Calculate R0 – (-R1) and update flag
  - CMN R0, #0x12 ; Calculate R0 – (-0x12) and update flag
- The TST (Test) instruction is more like the AND instruction. It ANDs two values and updates the flags. However, the result is not stored in any register. Similarly to CMP, it has two input formats:
  - TST R0, R1 ; Calculate R0 and R1 and update flag
  - TST R0, #0x12 ; Calculate R0 and 0x12 and update flag

Conditional Branches

- BEQ label ; Branch to address ‘label’ if Z flag is set

- BEQ.W label ; Branch to address ‘label’ if Z flag is set

Combined Compare and Conditional Branch

- two new instructions are provided on the Cortex-M3 to supply a simple compare with zero and conditional branch operations.
- These are CBZ (Compare and Branch if Zero) and CBNZ (Compare and Branch if Nonzero).

Combined Compare and Conditional Branch

- two new instructions are provided on the Cortex-M3 to supply a simple compare with zero and conditional branch operations.
  - These are CBZ (Compare and Branch if Zero) and CBNZ (Compare and Branch if Nonzero).

```
i = 5;
while (i != 0 ){
    func1(); ; call a function
    i--;
}
```

Combined Compare and Conditional Branch

- two new instructions are provided on the Cortex-M3 to supply a simple compare with zero and conditional branch operations.
  - These are CBZ (Compare and Branch if Zero) and CBNZ (Compare and Branch if Nonzero).

```
i = 5;
while (i != 0 ){
    func1(); ; call a function
    i--;
}
```

```
      MOV  R0, #5          ; Set loop counter
loop1 CBZ  R0, loop1exit   ; if loop counter = 0 then exit the loop
      BL   func1           ; call a function
      SUB  R0, #1          ; loop counter decrement
      B    loop1           ; next loop
loop1exit
```

Conditional Branches Using IT Instructions

- IT<x><y><z> <cond> ; IT instruction (<x>, <y>, ; <z> can be T or E)

- instr1<cond> <operands> ;1st instruction (<cond> must be same as IT)

- instr2<cond or not cond> <operands> ; 2nd instruction (can be ; <cond> or <!cond>

- instr3<cond or not cond> <operands> ; 3rd instruction (can be ; <cond> or <!cond>

- instr4<cond or not cond> <operands> ; 4th instruction (can be ; <cond> or <!cond>

If-Then-Else

```
if (R1<R2) then
    R2=R2-R1
    R2=R2/2

else

    R1=R1-R2
    R1=R1/2
```

If-Then-Else

```
if (R1<R2) then
    R2=R2-R1
    R2=R2/2
else
    R1=R1-R2
    R1=R1/2
```

```
CMP       R1, R2 ; If R1 < R2 (less then)
ITTEE     LT ; then execute instruction 1 and 2
(indicated by T) ; else execute instruction 3 and 4
(indicated by E)
SUBLT.W   R2,R1    ; 1st instruction
LSRLT.W   R2,#1    ; 2nd instruction
SUBGE.W   R1,R2    ; 3rd instruction (notice the GE is
; opposite of LT)
LSRGE.W   R1,#1    ; 4th instruction
```

Instruction Barrier and Memory Barrier Instructions

- In some cases, if memory barrier instructions are not used, race conditions could occur.

- For example, if the memory map can be switched by a hardware register, after writing to the memory switching register you should use the DSB instruction. Otherwise, if the write to the memory switching register is buffered and takes a few cycles to complete, and the next instruction accesses the switched memory region immediately, the access could be using the old memory map. In some cases, this might result in an invalid access if the memory switching and memory access happen at the same time. Using DSB in this case will make sure that the write to the memory map switching register is completed before a new instruction is executed.

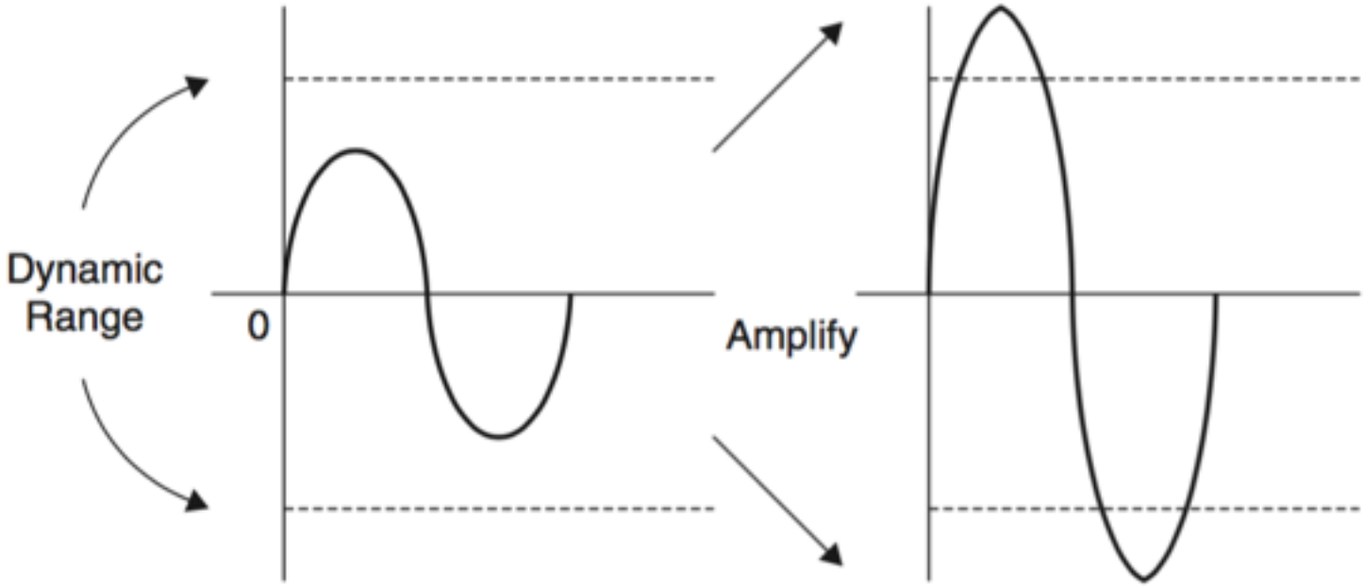
Barrier Instructions

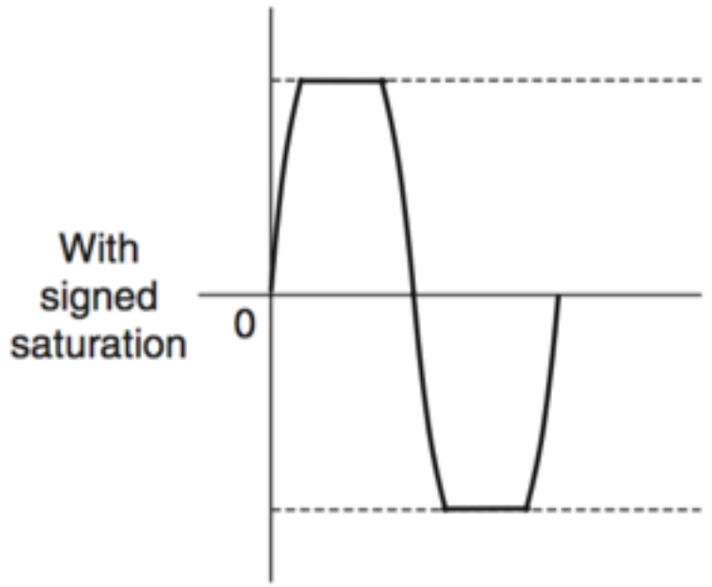
- DMB : Data Memory Barrier; ensures that all memory accesses are completed before new memory access is committed

- DSB : Data Synchronization Barrier; ensures that all memory accesses are completed before next instruction is executed

- ISB : Instruction Synchronization Barrier; flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

Saturation(饱和) Operations





Saturation Instructions

- Saturation for signed value
  - SSAT.W <Rd>, #<immed>, <Rn>, {,<shift>}
- Saturation for a signed value into an unsigned value
  - USAT.W <Rd>, #<immed>, <Rn>, {,<shift>}

Saturation Instructions

- SSAT.W R1, #16, R0

Input(R0)	Output(R1)	Q-flag
0x00020000	0x00007FFF	set
0x00008000	0x00007FFF	set
0x00007FFF	0x00007FFF	Unchanged
0x00000000	0x00000000	Unchanged
0xFFFF8000	0xFFFF8000	Unchanged
0xFFFF8001	0xFFFF8000	set
0xFFFE0000	0xFFFF8000	set