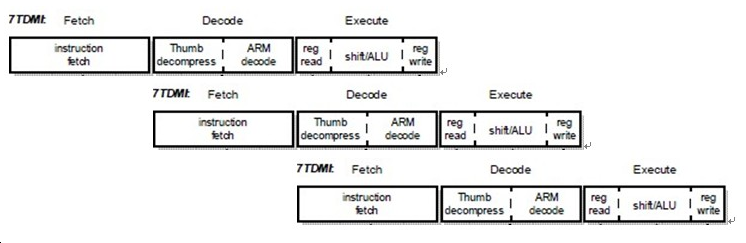
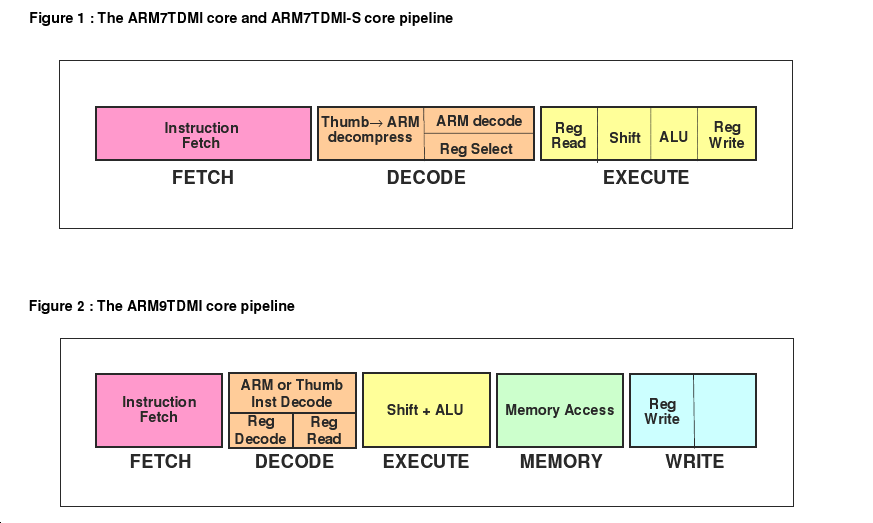
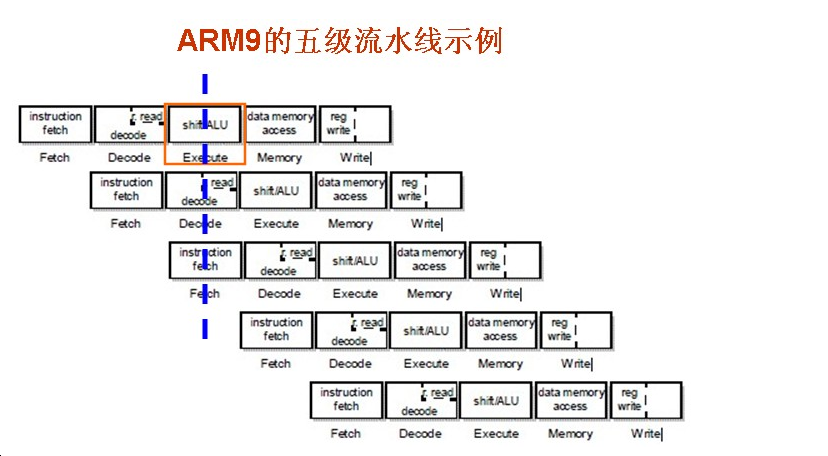
**流水线（PC=PC+8）**

ARM7中，CPU地址，即PC，为何有PC=PC+8这一说法, AMR7，是三级流水线，其细节见图,



**为何ARM9和ARM7一样，也是PC=PC+8**





有一句话要牢记，那就是：PC不是指向你正在运行的指令，而是PC始终指向你要取的指令的地址。

1. *指令周期Cycle1*
   1. *取指*

*PC总是指向将要读取的指令的地址（即我们常说的，指向下一条指令的地址），而当前PC=4，*

所以去取物理地址为4对对应的指令

ldr pc, [pc, #20]

其对应二进制代码为e59ff014。

此处取指完之后，自动更新PC的值，即PC=PC+4（单个指令占4字节，所以加4）=4+4=8

1. *指令周期Cycle2*
   1. *译指*

翻译指令e59ff014

* 1. 同时再去*取指*

PC总是指向将要读取的指令的地址（即我们常说的，指向下一条指令的地址），而当前PC=8，

所以去物理地址为8所对应的指令“ldr pc, [pc, #20]” 其对应二进制代码为e59ff014。

此处取指完之后，自动更新PC的值，即PC=PC+4=8+4=12=0xc

1. *指令周期Cycle3*
   1. *执行*（指令）

执行“e59ff014”，即

ldr pc, [pc, #20]

所对表达的含义，即PC

= PC + 20

= 12 + 20

= 32

= 0x20

此处，只是计算出待会要赋值给PC的值是0x20，这个0x20还只是放在执行单元中内部的缓冲中。

* 1. *译指*

翻译e59ff014

* 1. *取指*

此步骤由于是和上面（1）中的执行同步做的，所以，未受到影响，继续取指，而取指的那一时刻，PC为上一Cycle更新后的值，即PC=0xc，所以是去取物理地址为0xc所对应的指令

ldr pc, [pc, #20]

对应二进制为e59ff014

其实，分析到这里，大家就可以看出：

在Cycle3的时候，PC的值，刚好已经在Cycle1和Cycle2，分别加了4，所以Cycle3的时候，PC=PC+8，而同样道理，对于任何一条指令的，都是在Cycle3，指令的Execute执行阶段，如果用到PC的值，那么PC那一时刻，就是PC=PC+8。

所以，此处虽然是五级流水线，但是却不是PC=PC+16，而是PC=PC+8。

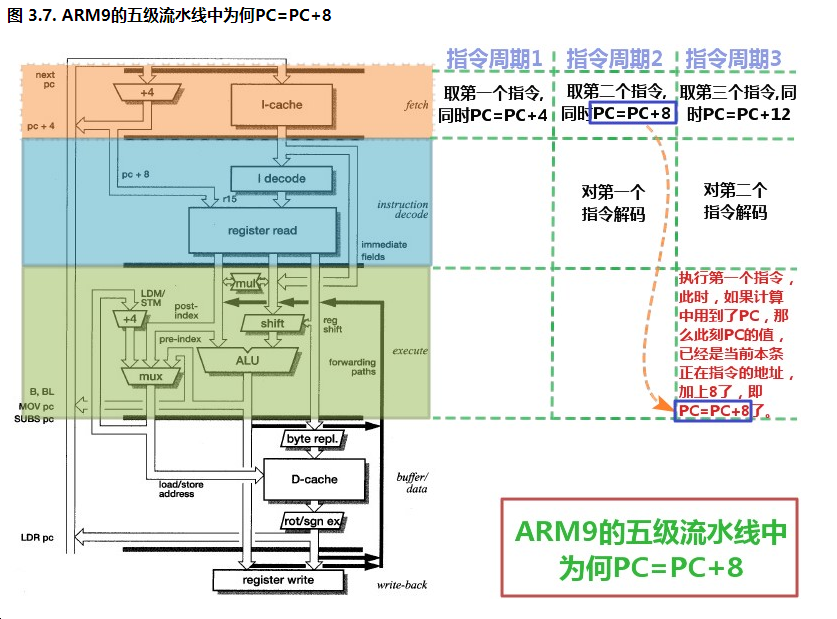
进一步地，我们发现，其实PC=PC+N的N，是和指令的执行阶段所处于流水线的深度有关，即此处指令的执行Execute阶段，是五级流水线中的第三个，而这个第三阶段的Execute和指令的第一个阶段的Fetch取指，相差的值是 3 -1 =2，即两个CPU的Cycle，而每个Cycle都会导致PC=+PC+4，所以，指令到了Execute阶段，才会发现，此时PC已经变成PC=PC+8了。

回过头来反观ARM7的三级流水线，也是同样的道理，指令的Execute执行阶段，是处于指令的第三个阶段，同理，在指令计算数据的时候，如果用到PC，就会发现此时PC=PC+8。

同理，假如ARM9的五级流水线，把指令的Execute执行阶段，设计在了第四个阶段，那么就是PC=PC+（第4阶段-1）\*4个字节 = PC= PC+12了。

用图来说明PC=PC+8个过程

对于上面的文字的分析过程，可能看起来不是太容易理解，所以，下面这里通过图表来表示具体的流程，就更容易看懂了。其中，下图，是以ARM9的五级流水线的内部架构图为基础，而编辑的出来用于说明为何ARM9的五级流水线，也是PC=PC+8：



|  |  |
| --- | --- |
|  | **PC（execute）=PC（fetch）+ 8** |
| 对于PC=PC+8中的两个PC，其实含义不完全一样.其更准确的表达，应该是这样：  *PC（execute）=PC（fetch）+ 8*  其中：  PC（fetch）：当前正在执行的指令，就是之前取该指令时候的PC的值  PC（execute）：当前指令执行的计算中，如果用到PC，则此时PC的值。 |
|  | **不同阶段的PC值的关系** | |
| 对应地，在ARM7的三级流水线（取指，译指，执行）和ARM9的五级流水线（取指，译指，执行，存储，写回）中，可以这么说：  PC， 总是指向当前正在被取指的指令的地址，  PC-4，总是指向当前正在被译指的指令的地址，  PC-8，总是指向当前的那条指令，即我们一般说的，正在被执行的指令的地址。 | |

关于mcr的来龙去脉：

<http://apps.hi.baidu.com/share/detail/32319228>

ARM 微处理器可支持多达 16 个协处理器，用于各种协处理操作，在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略 ARM 处理器和其他协处理器的指令。ARM 的协处理器指令主要用于 ARM 处理器初始化 ARM 协处理器的数据处理操作，以及在ARM 处理器的寄存器和协处理器的寄存器之间传送数据，和在 ARM 协处理器的寄存器和存储器之间传送数据。 ARM 协处理器指令包括以下 5 条：

1. CDP 协处理器数操作指令
2. LDC 协处理器数据加载指令
3. STC 协处理器数据存储指令
4. MCR ARM 处理器寄存器到协处理器寄存器的数据传送指令
5. MRC 协处理器寄存器到ARM 处理器寄存器的数据传送指令

......

CP15系统控制协处理器

CP15 —系统控制协处理器 （the system control coprocessor）他通过协处理器指令MCR和MRC提供具体的寄存器来配置和控制caches、MMU、保护系统、配置时钟模式（在bootloader时钟初始化用到）

CP15的寄存器只能被MRC和MCR（Move to Coprocessor from ARM Register ）指令访问

一些要说明的内容，见下：:

<http://infocenter.arm.com/help/topic/com.arm.doc.ddi0151c/ARM920T_TRM1_S.pdf>

you can only access CP15 registers with MRC and MCR instructions in a privileged mode. The assembler for these instructions is:

MCR/MRC{cond} P15,opcode\_1,Rd,CRn,CRm,opcode\_2

The CRn field of MRC and MCR

instructions specifies the coprocessor register to access. The CRm field and opcode\_2 fields specify a particular action when addressing registers. The L bit distinguishes between an MRC (L=1) and an MCR (L=0).

Note:

Attempting to read from a nonreadable register, or to write to a nonwritable register causes unpredictable results.

The opcode\_1, opcode\_2, and CRm fields should be zero, except when the values specified are used to select the desired operations, in all instructions that access CP15.

Using other values results in unpredictable behavior

CP15有很多个寄存器，分别叫做寄存器0(Register 0)，到寄存器15（Register 15），

每个寄存器分别控制不同的功能，而且有的是只读，有的是只写，有的是可读写。

而且这些寄存器的含义，随着版本ARM内核版本变化而不断扩展，详情请参考：[Processor setup via co-processor 15 and about co-processors](http://www.heyrick.co.uk/assembler/coprocmnd.html)