|  |
| --- |
|  |
| 关于ARM指令流水线知识 |
| （周方辉） |
|  |
|  |
| **2012/10/22** |

编改记录

|  |  |
| --- | --- |
| 编写日期 | 内容概要 |
| 2012-10-22 | 创建 |
|  |  |
|  |  |

|  |
| --- |
|  |

目 录

[1 参考文献 2](#_Toc338688382)

[1.1 内部参考文献 2](#_Toc338688383)

[1.2 外部参考文献 2](#_Toc338688384)

[2 名词解释 2](#_Toc338688385)

[3 指令执行三步骤 2](#_Toc338688386)

[4 指令流水线（ARM指令） 3](#_Toc338688387)

[4.1 三级指令流水线 3](#_Toc338688388)

[4.2 五级指令流水线 5](#_Toc338688389)

[4.3 六级指令流水线 6](#_Toc338688390)

[4.4 其它级指令流水线 6](#_Toc338688391)

# 参考文献

## 内部参考文献

内部参考文献指的是周方辉自生的百度博文中的文件。

无内部参考文献。

## 外部参考文献

外部参考文献指的是相对于上述内部参考文献以外的文件。

无外部参考文献

# 名词解释

**CPI：** 指令周期数，一段时间内走过的指令时钟数除以被执行的指令条数，CPI>=1。

**F：** Fetch的缩写，取指令的意思，用在分析指令流水线中。

**D：** Decode，解指令码，用在分析指令流水线中。

**E：** Execute，执行指令，用在分析指令流水线中。

**M**：Memory，内存操作。

**W：** Writeback回写。

**S**： Stall，拖延clock。

**L：** Linkret，连接返回。

**A：** Adjust，调整流水线。

**DI：Decode IRQ**解析中断指令。

**EI：Execute IRQ**执行中断指令。

**I：Interlock**，内部锁状态。

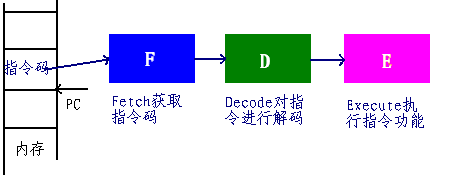
# 指令执行三步骤

一般计算机指令码与数据码没有区别，存在内存中，都属于二进制数字信息。指令码和数据码的区别是一般用PC指针从内存中读取的数据为指令码，否则就当数据码处理。

到目前为止，一般计算机执行指令是用CPU部件来执行的，通常分成：

1. 获取指令，通过PC指针，从内存中获取指令码；
2. 解析指令，使用CPU内部的指令解码器对指令码进行解析，从而得知指令功能。
3. 执行指令，按照解码器得知的功能，CPU调用相应部件来执行该条指令。

三个步骤完成，我们可以抽象理解成，每条指令都通过：取指令模块——>解析指令模块——>执行指令模块，这三个模块。如下图所示：



这三个步骤的协调工作是依靠指令时钟来推动完成的，，指令时钟并不等于CPU的时钟，一个指令时钟可能有几个CPU时钟组成，这看具体的CPU而定。

每个步骤至少需要一个指令时钟来完成。如执行指令这模块，可能要完成内存访问，即数据向内存回写等，就需要多个指令时钟了。

如何让最少的指令时钟执行更多的指令，成为CPU对指令执行速度的标志之一，我们使用CPI值来指示这一特征。

CPI是指在一定的时间范围内，指令时钟除以被执行了的指令条数。CPI值总是大于1，越接近1，表明CPU执行指令越块。

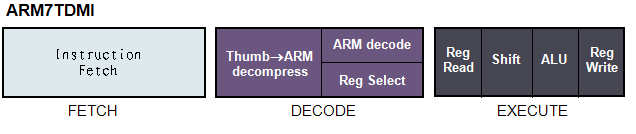
为此，诞生了指令流水线感念。

# 指令流水线（ARM指令）

我们为了提升CPU的指令执行速度，就希望取值、解码和执行3个模块同时都处于工作状态，例如，当我们第一条指令在执行时，第二条指令处于解码阶段，二第三条指令已被读到CPU中；当我们执行第二条指令时，第三条指令处于解码阶段，而第四条指令已被读到CPU中。这样对指令的执行就处于流水线操作状态。这就是指令流水线的概念。

## 三级指令流水线

ARM公司的ARM7核使用三级流水线。

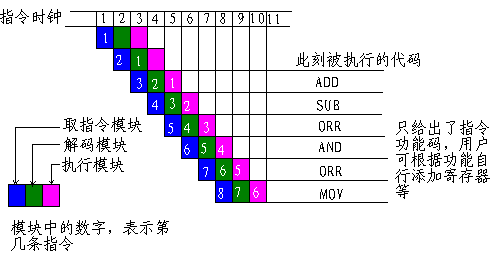


1. Fetch：只负责通过PC从内存中读取指令码数据。
2. Decode：要完成Thumb和ARM指令的识别、功能和参与寄存器的设别。
3. Execute：调用寄存器、ALU（及Shift）运算单元，和内存及寄存器的回写功能来完成操作。

三级指令执行流水线，正好将三个模块充分使用。我们举例说明其工作机制：

假设我们有下面这几条指令被执行（仅仅给出指令功能码）：

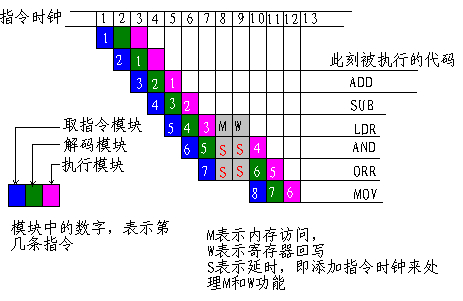
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 指令编号 | 指令 | 指令编号 | 指令 | 指令编号 | 指令 |
| 1 | ADD | 4 | AND | 7 | ROR |
| 2 | SUB | 5 | ORR | 8 | EOR |
| 3 | ORR | 6 | MOV | 9 | LSL |



从上述图表看出，从第5个指令时钟开始，CPU指令执行的各个模块都处于工作状态，每一个指令时钟都有一条指令被执行完成。

从第1个指令时钟开始到第10个指令时钟，一共11个指令时钟，这期间共有6条指令被执行，CPI值达1.8（11/6=1.8333）；假设，我们从第5个指令时钟开始算，到第10个指令时钟，共花费6个指令时钟，一共执行6条指令，这时，我们的CPI=1（6/6=1），这是理想状态下的CPU最快执行指标。

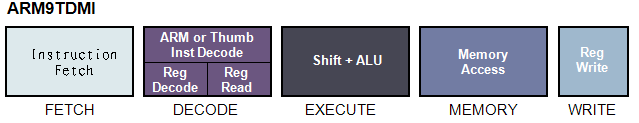
上述三级流水线仅仅理想状态下，如果我们有内存访问指令如LDR Rd，[Rn]，CPI就不能达到1了，因为内存访问需要消耗指令时钟。例如，我们将上述的第3条指令ORR改成LDR类（或者STR类）指令，其指令流水线如下：

在第3条LDE指令的执行阶段多插入连个指令周期来完成M和W动作（从内存中取出数据放入寄存器中）。这样一来，从第5个指令时钟到第12个指令时钟，共花费8个，而执行了6条指令，CPI=1.3（8、6=1.3333），降慢了CPU对指令执行速度。

不仅如此，遇到分支跳转指令，原先已经取得的指令和解码的指令都将废弃，从分支的新的指令地址开始从新装配流水线，CPI值也会上升。遇到中断类的是他就更加严重，中断的识别，跳转等都需要消耗大量指令时钟，CPI值上升得更加快。

## 五级指令流水线

ARM的五级指令流水线是在执行阶段中，添加两级，用来专门执行M和W之类功能，ARM9就使用了5级流水线。



也就是当我们执行STR（或者LDR，内存与寄存器访问）之类指令时，将内存访问和寄存器回写功能交付给Memory Acess和Register Write流水线来完成，它们的指令时钟将于后续指令的取指、解码和执行的指令时钟共用，从而它们不单独消指令时钟。这样就保证了指令执行又处于CPI=1状态。

不过，这种方式也有缺陷。当相邻两条指令同时出现相同的寄存器时，并且前一条指令是带有内存操作的，例如：

LDR R4,[R2] 将R2指定的内存中的数据加载到R4中

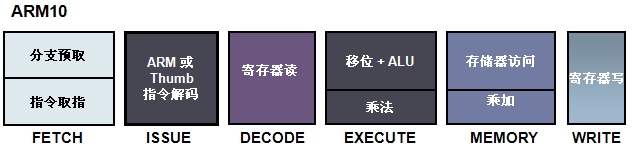
SUB R3,R4,R5 将R4-R5值保存到R3中。

因为LDR对R4的操作依赖于第4和第5条流水线，而这两条流水线执行，又是与后面SUB指令共用指令时钟，也就是两条指令将会同时访问R4寄存器，将产生冲突。这时，系统将插入一个interlock互锁指令时钟来完成，也就是，依旧采用添加指令时钟（提升CPI值）方式解决。当然用户也可调整一下程序顺序，或在其之间插入一条其它指令来缓和该问题，当然纸条指令必须是必要的，否则还不如不加。

这种想象经常发生在LDM和STM等与内存访问密切相关的指令中。

## 六级指令流水线

引起CPI值上升的还有另外一个重要的原因就是跳转分支指令。ARM10采用六级流水线，其在五级流水线基础上，在FETCH中添加了分支预取流水线。



如果取值得到一条是分支指令，则启动分支预取流水线，预测程序会往哪里跳转，事先将要跳转的指令加载到流水上，减少，一跳转就抛弃原先取出来的指令和解码好的指令。

分支预取流水线也不独立占用三级流水线的指令时钟。

## 其它级指令流水线

ARM公司指令流水线已经发展到10多级，不管如何，三级指令流水线是一种标准，CPI=1是追求的目标。

所以在程序代码中计算一些中断返回，或函数调用返回时，都是采用三级流水线计算，即：当前执行的指令是PC-8取出的指令，而下条（第二条）要执行的指令是PC-4，当前的PC取出的指令将是第三条要执行的指令。如果Thumb指令集下，分别为PC-4（当前）、PC-2（第二条）和PC（第三条）指令。