内存的错误检查与纠正

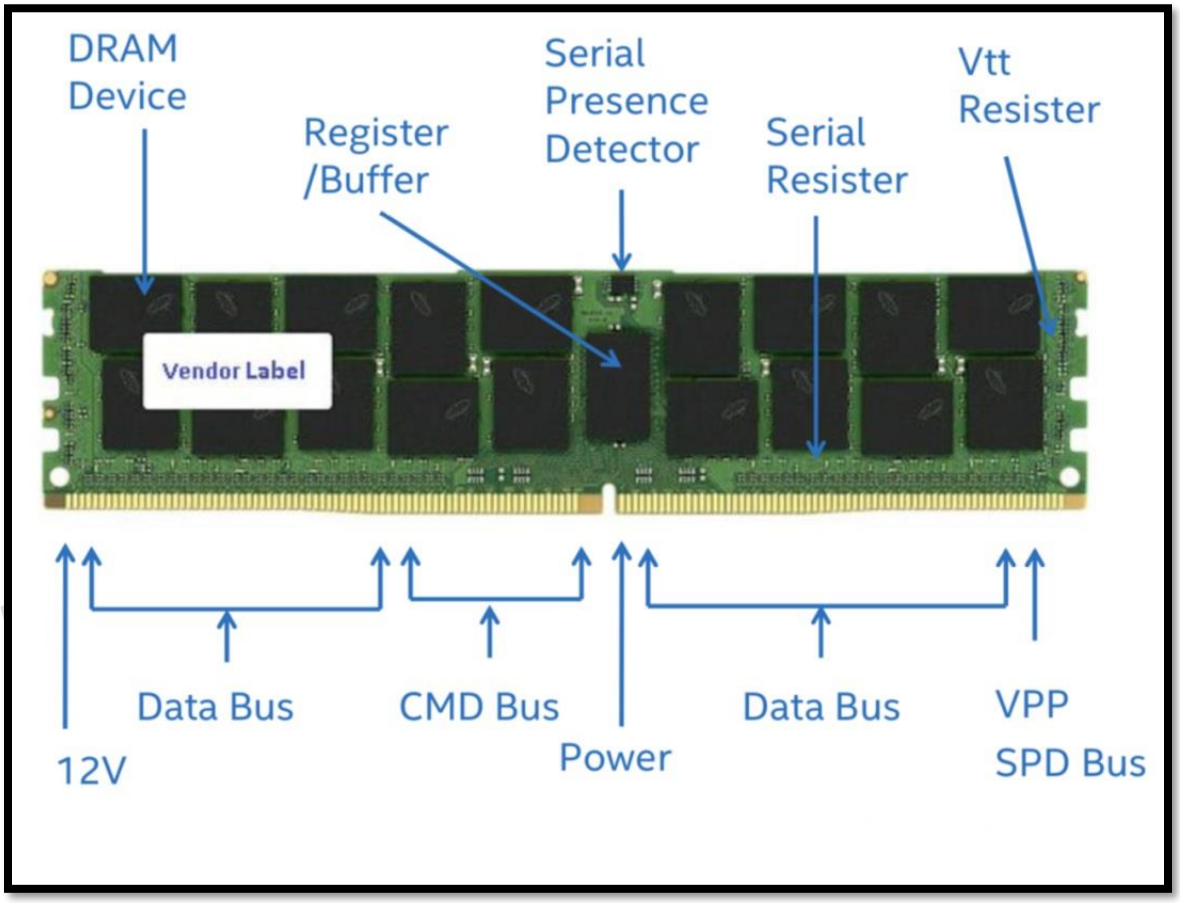


图 （1）

图（1）是一个带有 ECC 的 RDIMM，图中我们已经将各个组件和关键的金手指信号区域标示出来。首先，我们来认识一下这几个关键词：

Device：内存颗粒，根据其存放内容不同，又分为数据颗粒和 ECC 颗粒。通常有 X4，X8 和 X16，代表每个颗粒对外的数据线路是 4 lane，8 lane 和 16 lane。

Channel：一个 Channel 由一个或者多个 Rank 组成，其宽度由控制器决定。当前主流的个人电脑和服务器中，一个 Channel 的宽度为 64bit，可根据内存控制器是否支持 ECC 而扩展额外的 8bit。也就是说如果不支持 ECC 的 Channel，其宽度为 64bit，而支持 ECC 的 Channel，其宽度为 72bit。市面上两种内存条都有销售。

Rank：一个 Channel 里面，同一个 CS（Chip Select）信号选中的所有 Device 就是一个 Rank。同一个 Rank 中所有的 Device 共用命令，地址和控制信号。拿读操作举例，内存控制器发起的一个读操作，其实将作用于该 Channel 的某个 Rank 中所有的 Device。所有 Device 的数据线共同输出达到内存控制器所需的宽度。例如，采用 X4 的颗粒，组成不带 ECC 功能的一个 Rank 则需要 64/4 = 16 个 X4 的 Device。大家可以计算一下如果采用 X8 或者 X16 宽度的颗粒，需要多少个呢？

注：本文我们将主要以 X4 的 Device 来讨论

注：X16 的颗粒一般不被用来组成带 ECC 的 Rank

Cacheline：Cacheline 通常是指是处理器中 Cache Unit（缓存模块）缓存一笔数据的标准大小。根据处理器的不同，Cacheline 的大小是不一样的。当前主流的个人电脑和服务器中，Cacheline 的大小为 64 Byte。为了设计方便，处理器内部搬运可被缓存的数据也采用同样的大小 64B。为了满足该需求，一个 Rank 被设计成了 64bit 的数据位宽，而 JEDEC（DDR 标准组织）设计了 burst 传输。一个 Burst 的长度可以是 8，从而一次读操作，可以让颗粒一次吐出 8 笔数据。从而达到 64bit X 8 = 64B 的大小。具体参考图（2）。

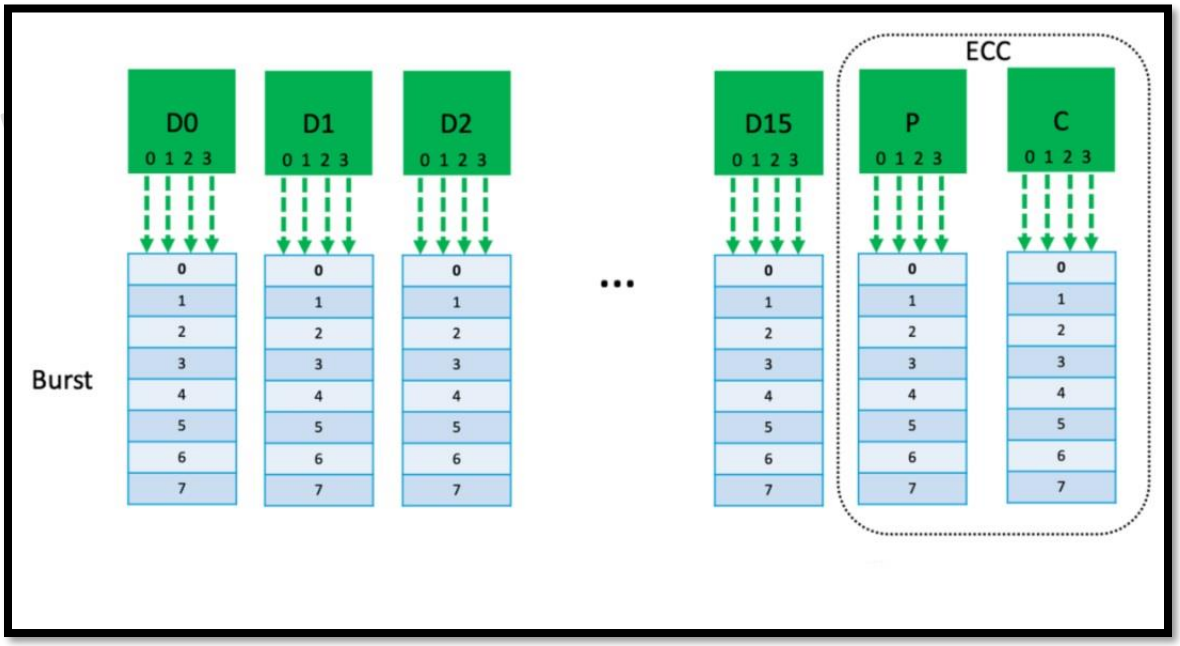


图 （2）

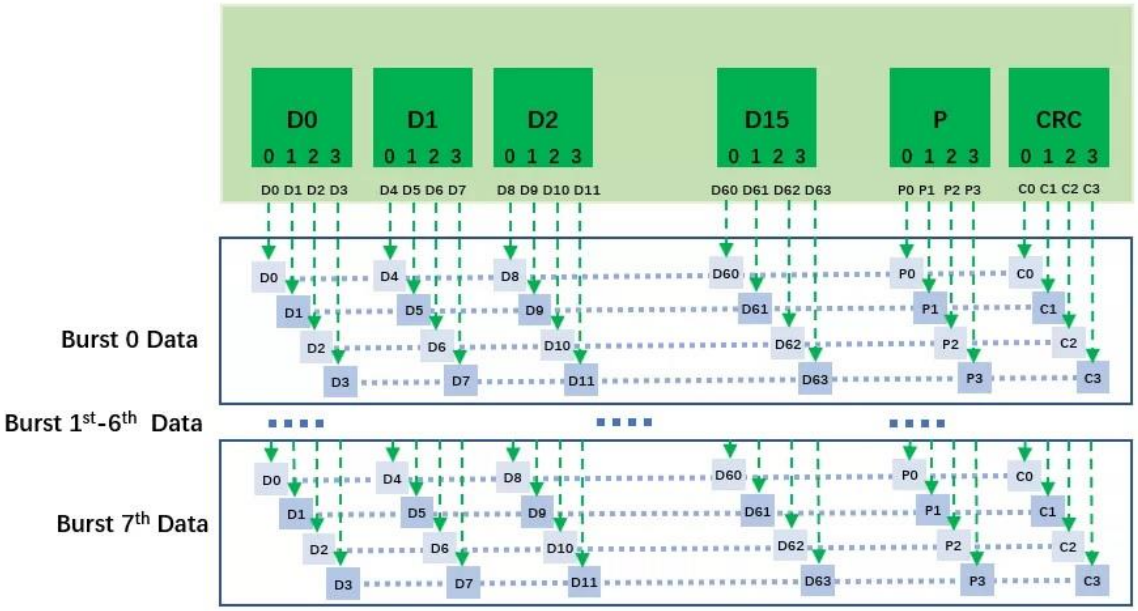
CE(CorrectableError)：可纠正错误是指硬件（芯片）可以直接纠正的错误。由于内存控制器设计不一样，对于可纠正错误的能力可能存在不同。例如，主流 x86 服务器的内存控制器（支持带 ECC 的内存条），在一次读操作中，一个 X4 宽度的 Device 内的任意错误都是可纠正的，包括 ECC 的 Device。如果 Rank 是 X8 宽度的 Device 组成，其纠正能力还是与 X4 的 Device 宽度及位置保持一

致。在 X8 的一个 Device 中，只有 DQ0-3，或者 DQ4-7 可以被纠正。如果是 DQ2-5，虽然是 X4 宽度但位置与 X4 时不对应，也无法纠正。
注：DQ0 即 D0，或者 D0_0，DQ63 则是 D63，或者 D15_3

UCE(Uncorrectable Error)： 不可纠正错误是指硬件（芯片）无法直接纠正的错误。例如，在一次读操作中，错误数据位分布在不同 X4 的 Device 范围，以现有内存控制设计来看，属于不可纠正错误。

下面我们简单介绍一下内存控制器是如何侦错和纠错的。由于 ECC 具体算法属于各家的 IP，这里介绍的方法只是帮助大家理解该功能。首先，内存控制器能够纠错，就必须先能发现错误。如果每次消费的数据大小是 64B，在不增加额外信息的情况下，我们是无法知道该数据是否有改变的，因为 64B 的数据可以是任何 01 的组合，即任意数据都是合法的。另一方面，额外的信息需要额外的存储，从成本考虑，这额外信息应该越小越好。JEDEC 组织提出增加额外 8 x 8 = 64bit 的数据来帮助一个 64B 的数据完成 ECC。
从物理角度看，一个 X4 Device 组成的 Rank 将会增加两个 Device 用于 ECC。一种可行的做法是，其中一个 Device 负责存放 CRC（Cyclic Redundancy Check）校验信息用于侦错，另一个 Device 负责存放奇偶校验信息（Parity），配合纠正错误。

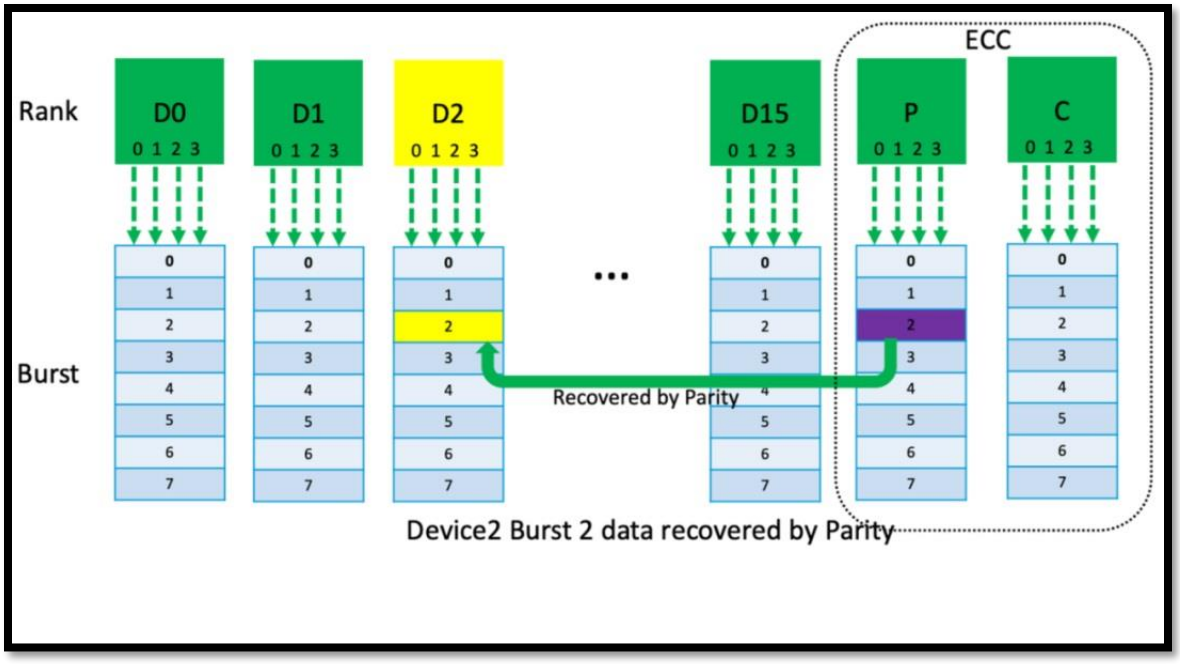
Parity：Parity 基本功能是发现保护数据中是否有 bit 翻转。保护方法是统计保护数据中 1 的个数，如果是偶校验，当保护数据中 1 的个数是偶数时，Parity 为 0，否则为 1，所以 Parity 只需要一个 bit 就能发现保护数据中是否有一个 bit 的数据翻转（0 到 1 或者 1 到 0）。当然对于奇数个 bit 都有一样的检测效果。但当偶数 bit 翻转的时候，Parity 将无法知道。在了解了 Parity 基本功能后，我们来看看内存控制器是如何计算 Parity 并存放的。如图（3）所示。



图（3）

Burst 传输中每一笔 64bit 数据，4bit Parity 和 4bit CRC 的具体对应关系如下：
P0=D0_0+D1_0+D2_0+…D15_0+C0P1=D0_1+D1_1+D2_1+…D15_1+C1P2=D0_2+D1_2+D2_2+…D15_2+C2P3=D0_3+D1_3+D2_3+…D15_3 +C3
注：D15_3 为 Device15 的 DQ3 信号，从 Rank 角度看，为图中的 D63

假设 Device 2 在 Burst 的第三笔数据中有 bit 翻转，则无论是 D2_0, D2_1, D2_2, D2_3 或者都错了，请参考图（4），我们都可以通过 Parity bits 反算回来，前提是 burst 的第三笔数据中其他 Device 没有出现错误。具体计算如下：
D2_0=P0(-)(D0_0+D1_0+D3_0…D15_0+C0)
D2_1=P1(-)(D0_1+D1_1+D3_1…D15_1+C1)
D2_2=P2(-)(D0_2+D1_2+D3_2…D15_2+C2)
D2_3=P3(-)(D0_3+D1_3+D3_3…D15_3+C3)



图（4）

CRC：我们怎样知道读取的 Cacheline 数据是正确的还是错误的？这里将会用到 CRC 来进行校验。一种比较简单的校验方式就是除法。我们设计一个除数，让被保护数据（被除数）去除以这个除数，然后会得到商和余数。通常余数比设计的除数要小。在存储一个 Cacheline 大小数据到内存条上的时候，内存控制器会计算 CRC 的值，并存放到 CRC 的 Device 中去。读取的时候再计算一遍，然后和内存条读回来的 CRC 的值进行比较。如果一致，则认为数据没有发生变化。否则，认为数据出错。从上述理论可以推出，CRC 校验位越多，则侦错能力越强。CRC 设计不一样，侦错不同数据翻转的能力不同。可能存在数据错了，但侦错不了的情况。既然有漏测的情况，为什么我们还会继续使用？这就和错误类型的概率有关了。通常情况下，一个 bit 翻转的可能性比较高，多 bit 同时翻转的可能性比较低。多 bit 翻转在同一个 device 里的几率比较高，多 device 同时翻转的概率比较低。

举个例子，当一个 Cacheline 的数据从内存条里读出来后，通过 CRC 校验，我们会发现数据有可能已经发生改变。这个时候，我们先假设出现了 CE（Correctable Error）问题。则通过 Parity 信息反算 Device 数据，需要一个 Device 一个 Device 的假设，然后重新计算 CRC 和之前存储的 CRC 进行比较。所以最多的情况可能要假设 18 次。如果全部弄完仍然 CRC 对不上，则属于 UCE（Uncorrectable Error）问题啦。当然，大家会发现，ECC 校验过程会影响内存读写延时。