

深入了解内存

清水反应 [Ars technica](#)、[Aceshardware](#)、[simpletech](#) 等 独家编译

原文链接：[Ace's Guide to Memory Technology](#)

原文作者：Johan De Gelas(johan@aceshardware.com)

collected by stonefeng

[DRAM 和 SRAM 基础知识](#)

[RAM 模块基础](#)

[DRAM 读取过程](#)

[快页模式内存](#)

[SDRAM 读取过程分析](#)

[SDRAM 写入过程](#)

[Aceshardware 所写的关于 SDRAM 基本工作原理的文章](#)

DRAM 和 SRAM 基础知识

RAM (Random Access Memory) 随机存取存储器对于系统性能的影响是每个 PC 用户都非常清楚的, 所以很多朋友趁着现在的内存价格很低纷纷扩容了内存, 希望借此来得到更高的性能。不过现在市场是多种内存类型并存的, SDRAM、DDR SDRAM、RDRAM 等等, 如果你使用的还是非常古老的系统, 可能还需要 EDO DRAM、FP DRAM (块页) 等现在不是很常见的内存。

对于很多用户或者有一定经验的高级用户来说, 他可能能说出 Athlon XP 和 Pentium 4 的主要不同点, 能知道 GeForce3 和 Radeon 之间的区别, 但是如果真的让他说出各种内存之间的实现机理的主要差别或者解释 CAS 2 和 CAS 3 之间的主要差别的话, 就可能不是非常的清楚了。毕竟 CPU 和显卡之类的东西更容易引起我们的兴趣。我个人在这方面的知识也是比较片面甚至是一知半解的, 所以一直在收集这个方面的资料。在网上有很多很好的资源, 其中 Ars technica、Aceshardware、simpletech 等网站的资料对于我系统的了解这个方面的知识有很大的帮助。本文主要以 Ars technica 的文章为基础编写而成, 为大家比较详细的介绍 RAM 方面的知识。

虽然 RAM 的类型非常的多, 但是这些内存在实现的机理方面还是具有很多相同的地方, 所以本文的将会分为几个部分进行介绍, 第一部分主要介绍 SRAM 和异步 DRAM (asynchronous DRAM), 在以后的章节中会对于实现机理更加复杂的 FP、EDO 和 SDRAM 进行介绍, 当然还会包括 RDRAM 和 SGRAM 等等。对于其中同你的观点相悖的地方, 欢迎大家一起进行技术方面的探讨。

存储原理

为了便于不同层次的读者都能基本的理解本文, 所以我先来介绍一下很多用户都知道的东西。RAM 主要的作用就是存储代码和数据供 CPU 在需要的时候调用。但是这些数据并不是像用袋子盛米那么简单, 更像是图书馆中用有格子的书架存放书籍一样, 不但要放进去还要能够在需要的时候准确的调用出来, 虽然都是书但是每本书是不同的。对于 RAM 等存储器来说也是一样的, 虽然存储的都是代表 0 和 1 的代码, 但是不同的组合就是不同的数据。

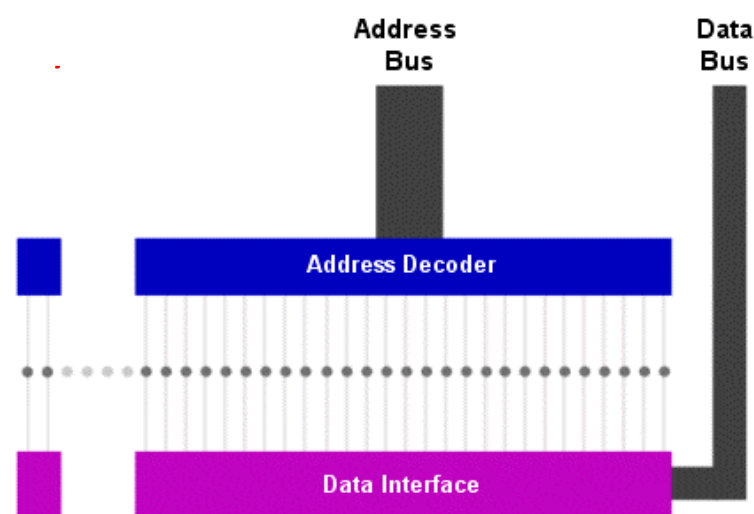
让我们重新回到书和书架上来, 如果有一个书架上有 10 行和 10 列格子 (每行和每列都有 0-9 的编号), 有 100 本书要存放在里面, 那么我们使用一个行的编号 + 一个列的编号就能确定某一本书的位置。如果已知这本书的编号 87, 那么我们首先锁定第 8 行, 然后找到第 7 列就能准确的找到这本书了。在 RAM 存储器中也是利用了相似的原理。

现在让我们回到 RAM 存储器上, 对于 RAM 存储器而言数据总线是用来传入数据或者传出数据的, 因为存储器中的存储空间是如前面提到的存放图书的书架一样通过一定的规则定义的, 所以我们可以通过这个规则来把数据放到存储器上相应的位置, 而进行这种定位的工作就要依靠地址

总线来实现了。

对于 CPU 来说, RAM 就象是一条长长的有很多空格的细线, 每个空格都有一个唯一的地址与之相对应。

如果 CPU 想要从 RAM 中调用数据, 它首先需要给地址总线发送地址数据定位要存取的数据, 然后等待若干个时钟周期之后, 数据总线就会把数据传输给 CPU。下面的示意图可以帮助你很好的理解这个过程。上图中的小圆点代表 RAM 中的存储



空间，每一个都有一个唯一的地址线同它相连。当地址解码器接收到地址总线送来的地址数据之后，它会根据这个数据定位 CPU 想要调用的数据所在的位置，然后数据总线就会把其中的数据传送到 CPU。

上面所列举的例子中 CPU 在一行数据中每次存取一个字节的数据，但是在现实世界中是不同的，通常 CPU 每次需要调用 32bit 或者是 64bit 的数据（这是根据不同计算机系统的数据总线的位宽所决定的）。如果数据总线是 64bit 的话，CPU 就会在一个时间中存取 8 个字节的数据，因为每次还是存取 1 个字节的数据，64bit 总线将不会显示出来任何的优势，工作的效率将会降低很多。

从“线”到“矩阵”

如果 RAM 对于 CPU 来说仅仅是一条“线”的话，还不能体现实际的运行情况。因为如果实际情况真的是这样的话，在实际制造芯片的时候，会有很多实际的困难，特别是在需要设计大容量的 RAM 的时候。所以，一种更好的能够降低成本的方法是让存储信息的“空格”排列为很多行 - 每个“空格”对应一个 bit 存储的位置。这样，如果要存储 1024bits 的数据，那么你只要使用 32x32 的矩阵就能够达到这个目的了。很明显，一个 32x32 的矩阵比一个 1024bit 的行设备更紧凑，实现起来也更加容易。请看下图 1：

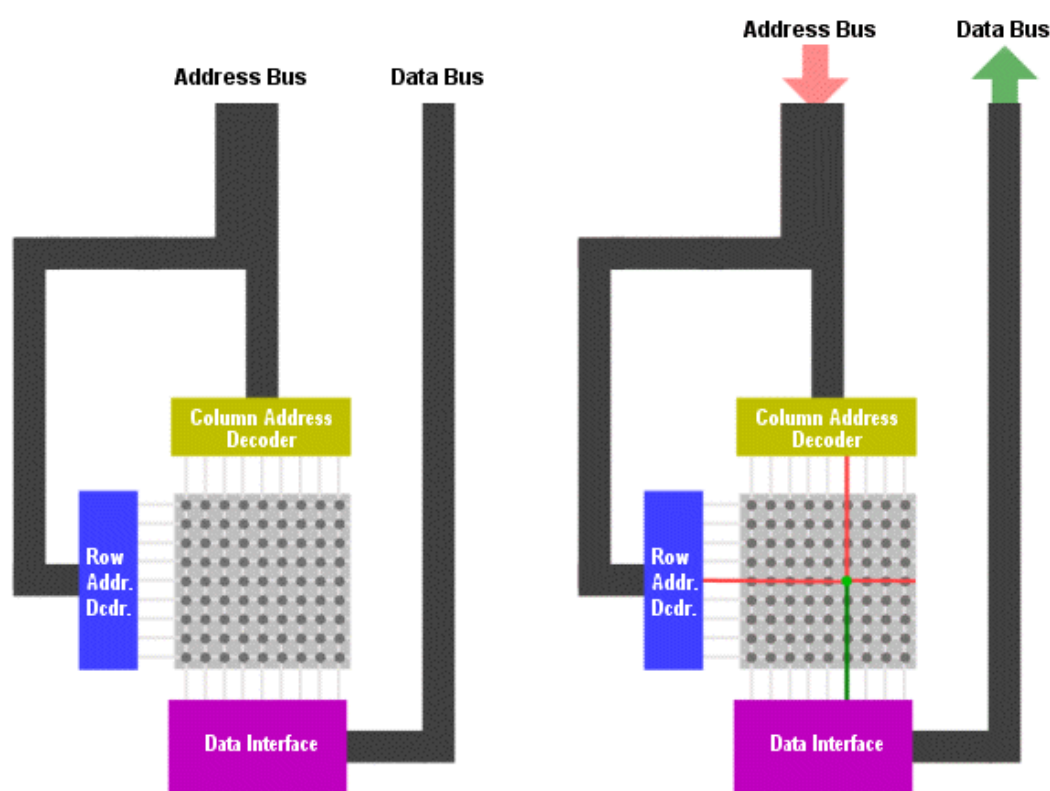


图 1

图 2

知道了 RAM 的基本结构是什么样子的，我们就下面谈谈当存储字节的过程是怎样的：

上面的示意图 1 显示的也仅仅是最简单状态下的情况，也就是当内存条上仅仅只有一个 RAM 芯片的情况。对于 X86 处理器，它通过地址总线发出一个具有 22 位二进制数字的地址编码 - 其中 11 位是行地址，另外 11 位是列地址，这是通过 RAM 地址接口进行分离的。行地址解码器（row decoder）将会首先确定行地址，然后列地址解码器（column decoder）将会确定列地址，这样就能确定唯一的存储数据的位置，然后该数据就会通过 RAM 数据接口将数据传到数据总线。另外，需要注意的是，RAM 内部存储信息的矩阵并不是一个正方形的，也就是行和列的数目不是相同的 - 行的数目比列的数目少。（后面我们在讨论 DRAM 的过程中会讲到为什么会这样）

上面的示意图 2 粗略的概括了一个基本的 SRAM 芯片是如何工作的。SRAM 是 “static RAM(静态随机存储器)” 的简称,之所以这样命名是因为当数据被存入其中后不会消失(同 DRAM 动态随机存储器是不同, DRAM 必须在一定的时间内不停的刷新才能保持其中存储的数据)。一个 SRAM 单元通常由 4-6 只晶体管组成, 当这个 SRAM 单元被赋予 0 或者 1 的状态之后, 它会保持这个状态直到下次被赋予新的状态或者断电之后才会更改或者消失。SRAM 的速度相对较快, 而且比较省电, 但是存储 1bit 的信息需要 4-6 只晶体管制造成本太高了 (DRAM 只要 1 只晶体管就可以实现)。

RAM 芯片

前面的介绍都相对比较简单、抽象。下面我们会结合实际的 RAM 芯片进行介绍。在谈到这个问题的时候, 我们会涉及到一个比较重要的技术: 封装。你应该听说过诸如 30 线 SIMMS、72 线 SIMMS 和 168 线 DIMMS 或者 RIMMs 其中的一个或者几个术语吧。如果要解释这些术语之间的不同, 就应该了解 RAM 的封装技术。

SRAM 芯片

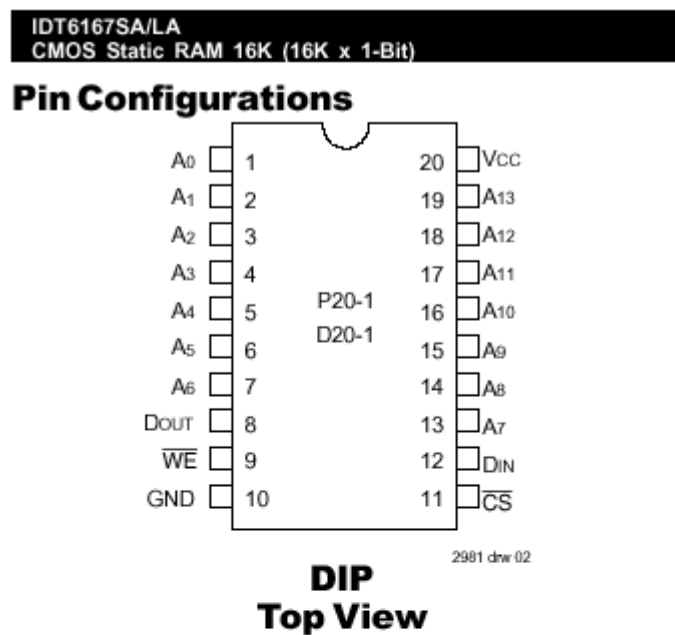
早期的 SRAM 芯片采用了 20 线双列直插 (DIP: Dual Inline Package) 封装技术, 它们之所以具有这么多的针脚, 是因为它们必须:

- 每个地址信号都需要一根信号线
- 一根数据输入线和一根数据输出线
- 部分控制线(Write Enable, Chip Select)
- 地线和电源线



上图显示的是 SRAM 芯片, 但是并不是下面示意图中的 SRAM 芯片

下面的是一个 16K x 1-bit SRAM 芯片的针脚功能示意图：



Pin Descriptions

Name	Description
A0 - A13	Address Inputs
\overline{CS}	Chip Select
\overline{WE}	Write Enable
Vcc	Power
DIN	DATAin
Dout	DATAout
GND	Ground

©2000 Integrated Device Technology, Inc.

2981 bit 01

- A0-A13 是地址输入信号引脚
- CS 是芯片选择引脚
在一个实际的系统中，一定具有很多片 SRAM 芯片，所以需要选择究竟从那一片 SRAM 芯片中写入或者读取数据
- WE 是写入启用引脚（如上表，在 CS、WE 上面的线我没有写入，表示低电平有效或者是逻辑 0 时有效）：当 SRAM 得到一个地址之后，它需要知道进行什么操作，究竟是写入还是读取，WE 就是告诉 SRAM 要写入数据
- Vcc 是供电引脚
- Din 是数据输入引脚
- Dout 是数据输出引脚
- GND 是接地引脚
- Output Enable (OE)：有的 SRAM 芯片中也有这个引脚，但是上面的图中并没有。这个引脚同 WE 引脚的功能是相对的，它是让 SRAM 知道要进行读取操作而不是写入操作。

从 Dout 引脚读取 1bit 数据需要以下的步骤：

SRAM 读取操作

1)通过地址总线把要读取的 bit 的地址传送到相应的读取地址引脚(这个时候/WE 引脚应该没有激活，所以 SRAM 知道它不应该执行写入操作)

2)激活/CS 选择该 SRAM 芯片

3)激活/OE 引脚让 SRAM 知道是读取操作

第三步之后，要读取的数据就会从 DOut 引脚传输到数据总线。怎么过程非常的简单吧？同样，写入 1bit 数据的过程也是非常的简单的。

SRAM 写入操作

1)通过地址总线确定要写入信息的位置(确定/OE 引脚没有被激活)

2)通过数据总线将要写入的数据传输到 Dout 引脚

3)激活/CS 引脚选择 SRAM 芯片

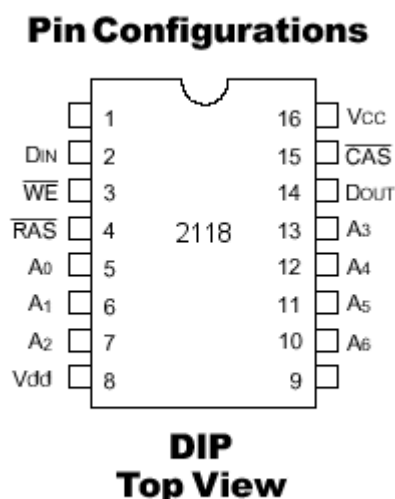
4)激活/WE 引脚通知 SRAM 知道要尽心写入操作

经过上面的四个步骤之后，需要写入的数据就已经放在了需要写入的地方。

DRAM 芯片

现在我们知道了在一个简单的 SRAM 芯片中进行读写操作的步骤了，然后我们了解一下普通的 DRAM 芯片的工作情况。DRAM 相对于 SRAM 来说更加复杂，因为在 DRAM 存储数据的过程中需要对于存储的信息不停的刷新，这也是它们之间最大的不同。下面让我们看看 DRAM 芯片的针脚的作用。

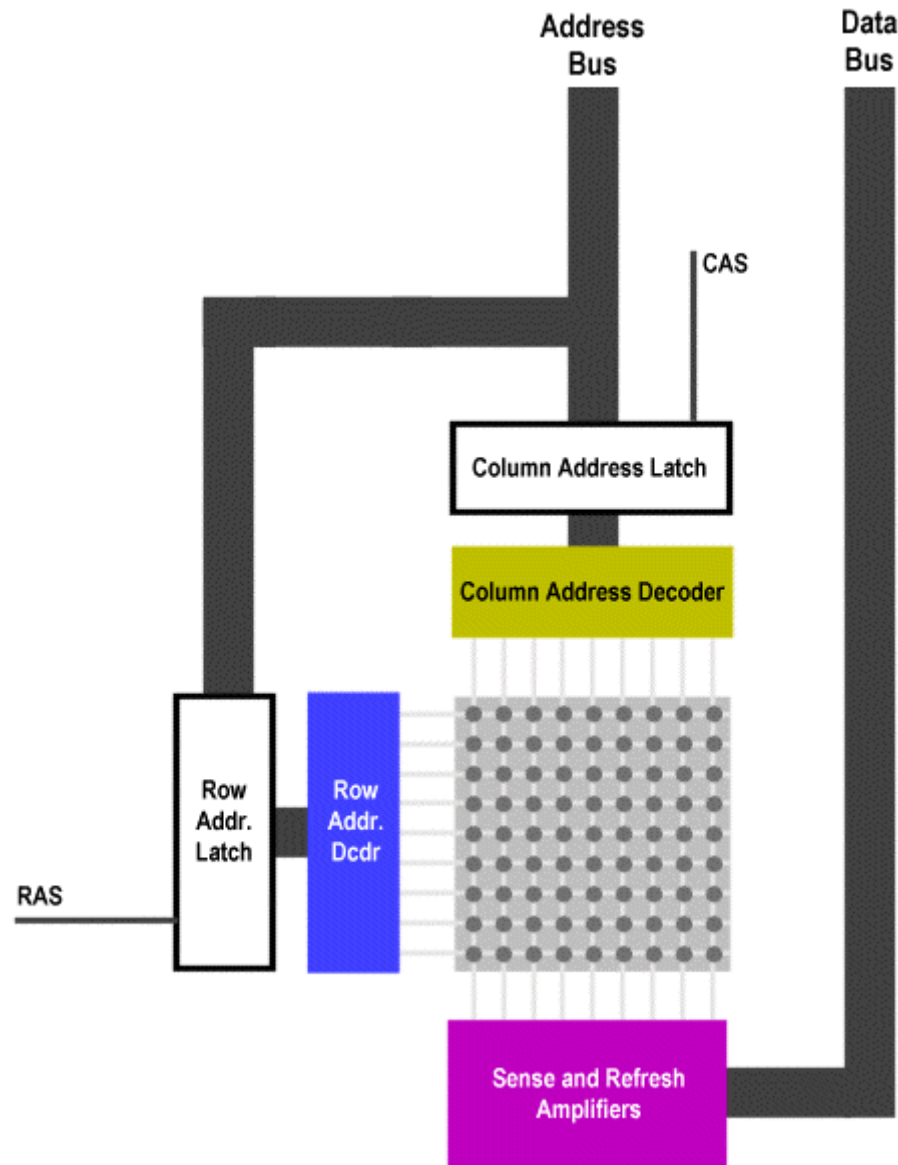
最早、最简单也是最重要的一款 DRAM 芯片是 Intel 在 1979 年发布的 2188，这款芯片是 16Kx1 DRAM 18 线 DIP 封装。“16K x 1”的部分意思告诉我们这款芯片可以存储 16384 个 bit 数据，在同一个时期可以同时进行 1bit 的读取或者写入操作。(很抱歉找不到这款芯片的实物图片，只好自己简单的画了一个示意图)



上面的示意图可以看出，DRAM 和 SRAM 之间有着明显的不同。首先你会看到地址引脚从 14 根变为 7 根，那么这颗 16K DRAM 是如何完成同 16K SRAM 一样的工作的呢？答案很简单，DRAM 通过 DRAM 接口把地址一分为二，然后利用两个连续的时钟周期传输地址数据。这样就达到了使用一半的引脚实现同 SGRAM 同样的功能的目的，这种技术被称为多路技术 (multiplexing)。

那么为什么好减少地址引脚呢？这样做有什么好处呢？前面我们曾经介绍过，存储 1bit 的数据 SRAM 需要 4-6 个晶体管但是 DRAM 仅仅需要 1 个晶体管，那么这样同样容量的 SRAM 的体积比 DRAM 大至少 4 倍。这样就意味着你没有足够空间安放同样数量的引脚（因为引脚并没有因此减少 4 倍）。当然为了安装同样数量的针脚，也可以把芯片的体积加大，但是这样就提高芯片的生产成本和功耗，所以减少针脚数目也是必要的，对于现在的大容量 DRAM 芯片，多路寻址技术已经是必不可少的了。

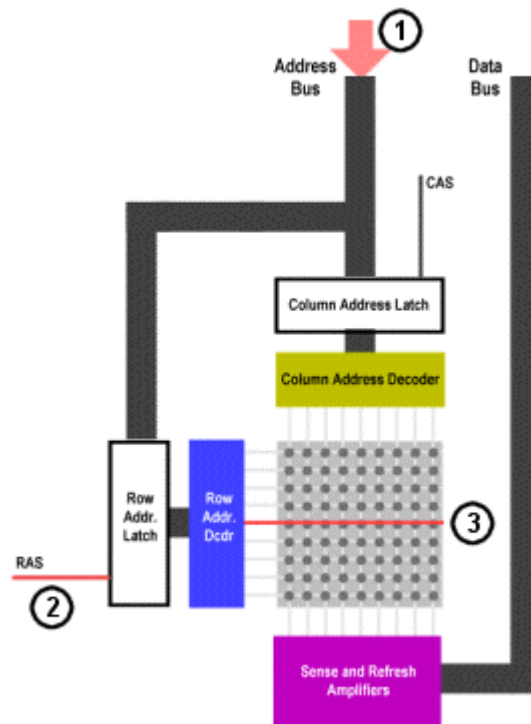
当然多路寻址技术也使得读写的过程更加复杂了，这样在设计的时候不仅仅 DRAM 芯片更加复杂了，DRAM 接口也要更加复杂，在我们介绍 DRAM 读写过程之前，请大家看一张 DRAM 芯片内部结构示意图：



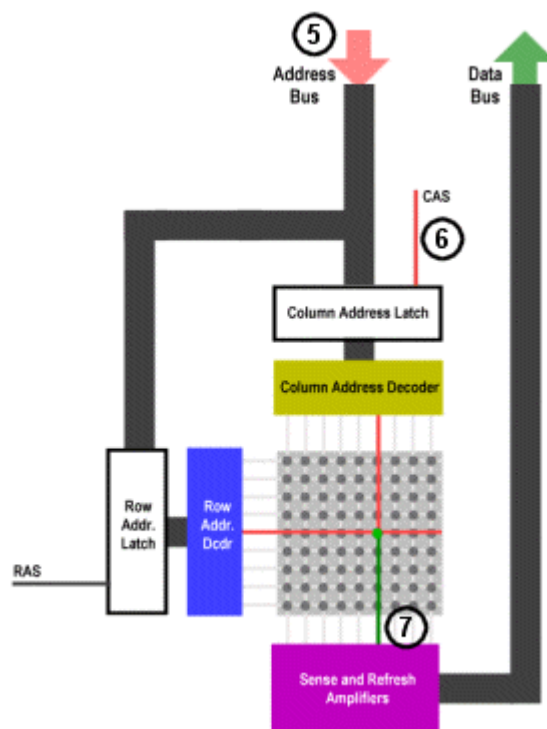
在上面的示意图中，你可以看到在 DRAM 结构中相对于 SRAM 多了两个部分：由/RAS (Row Address Strobe：行地址脉冲选通器)引脚控制的行地址门选线路（Row Address Latch）和由/CAS(Column Address Strobe：列地址脉冲选通器)引脚控制的列地址门选线路（Column Address Latch）。

DRAM 读取过程

- 1)通过地址总线将行地址传输到地址引脚
- 2)/RAS 引脚被激活，这样行地址被传送到行地址门线路中
- 3)行地址解码器根据接收到的数据选择相应的行



- 4)/WE 引脚被确定不被激活，所以 DRAM 知道它不会进行写入操作
- 5)列地址通过地址总线传输到地址引脚
- 6)/CAS 引脚被激活，这样列地址被传送到行地址门线路中
- 7)/CAS 引脚同样还具有/OE 引脚的功能，所以这个时候 Dout 引脚知道需要向外输出数据。



8) /RAS 和/CAS 都不被激活，这样就可以进行下一个周期的数据操作了。

其实 DRAM 的写入的过程和读取过程是基本一样的，所以如果你真的理解了上面的过程就能知道写入过程了，所以这里我就不赘述了。（只要把第 4 步改为/WE 引脚被激活就可以了）。

DRAM 刷新

我们已经提到过，DRAM 同 SRAM 最大的不同就是不能比较长久的保持数据，这项特性使得这种存储介质对于我们几乎没有任何的作用。但是 DRAM 设计师利用刷新的技术使得 DRAM 称为了现在对于我们最有用处的存储介质。这里我仅仅简要的提及一下 DRAM 的刷新技术，因为在后面介绍 FP、EDO 等类型的内存的时候，你会发现它们具体的实现过程都是不同的。

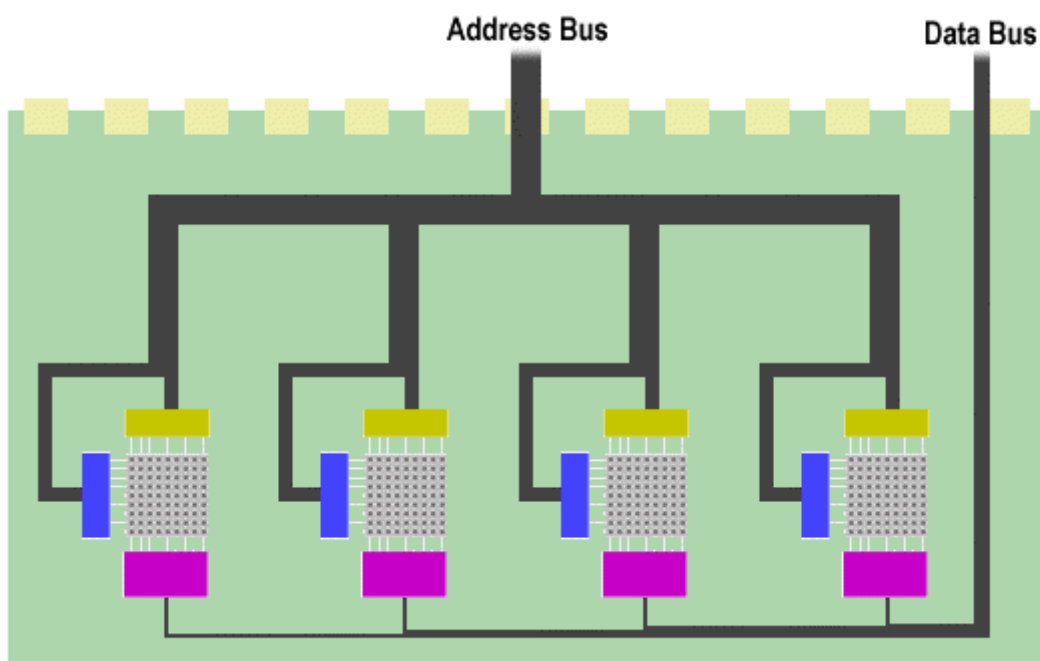
DRAM 内仅仅能保持其内存存储的电荷非常短暂的时间，所以它需要在其内的电荷消失之前就进行刷新直到下次写入数据或者计算机断电才停止。每次读写操作都能刷新 DRAM 内的电荷，所以 DRAM 就被设计为有规律的读取 DRAM 内的内容。这样做有下面几个好处。第一，仅仅使用/RAS 激活每一行就可以达到全部刷新的目的；第二，DRAM 控制器来控制刷新，这样可以防止刷新操作干扰有规律的读写操作。在文章的开始，我曾经说过一般行的数目比列的数据少。现在我可以告诉为什么会这样了，因为行越少用户刷新的时间就会越少。

RAM 模块基础

在前面的一节文章中我们对于 DRAM 和 SRAM 的基本工作原理做了一些简单的介绍，在我们所列举的例子中都是介绍了最基本的存储单元的工作模式，所以应该不难理解，看到很多朋友对于这个方面的东西很感兴趣，今天我就继续介绍关于 RAM (Random Access Memory) 的部分知识。理解这个部分知识，是更好的了解以后我们介绍各种 RAM 的实际工作情况的基础。

在 SRAM 或者 DRAM 的每一个基本存储单位(也就是上一节中介绍用来存储 1bit 信息的存储单位)都只能存储 0 或者 1 这样的数据，而且在上一节中 IDT6167 和 Intel 2188 芯片都仅仅只有 Din (数据输入) 和 Dout (数据输出接口)，而 CPU 存取数据的时候是按照字节 (也就是 8bit) 来存储的，那么 RAM 究竟如何满足 CPU 的这样的要求呢？

首先为了能存储 1 字节(8 bit)的信息,就需要 8 个 1bit RAM 基本存储单元堆叠在一起，这也意味着这 8 颗芯片被赋予了同样的地址。下面的示意图可以帮助你比较形象的了解这一点 (下图所示的图例中仅仅画了 4 个存储单元，大家当成 8 个来看就可以了)。

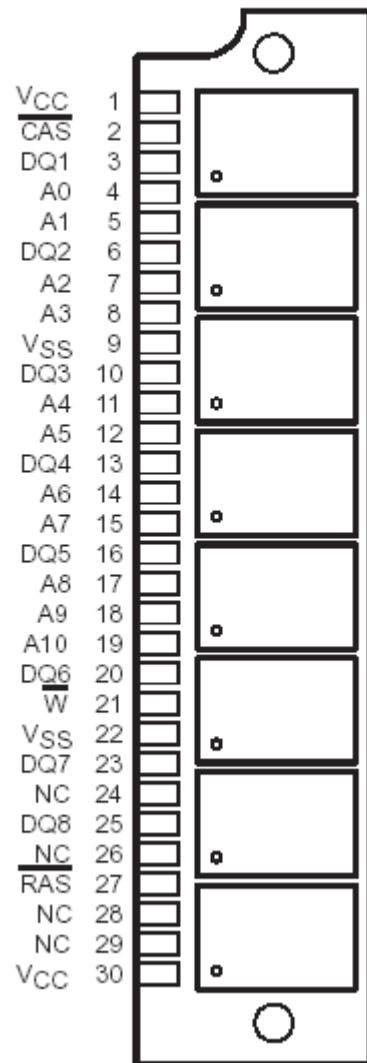


通常这 8 颗 1bit 芯片是通过地址总线 and 数据总线在 PCB (印刷电路板) 上连接而成的，对于 CPU 来说它就是一颗 8bit 的 RAM 芯片，而不再是独立的 8 个 1 bit 芯片。在上图所示的地址总线位宽是 22bit，这样这个地址总线所能控制的存储模块的容量应该是 $2^{22}=4194304\text{bit}$ ，也就是 4MB 的容量；数据总线的位宽是 8bit，就是通过刚才提到的 8 个 1bit 的基本存储单元的 Dout 并联在一起实现的 - - 这样也能够满足 CPU 的要求了。(对于这种存储颗粒我们称之为 4194304 x 8 模块或者 4Mx8，注意这里的“M”不是“MByte”而是“Mbit”)。

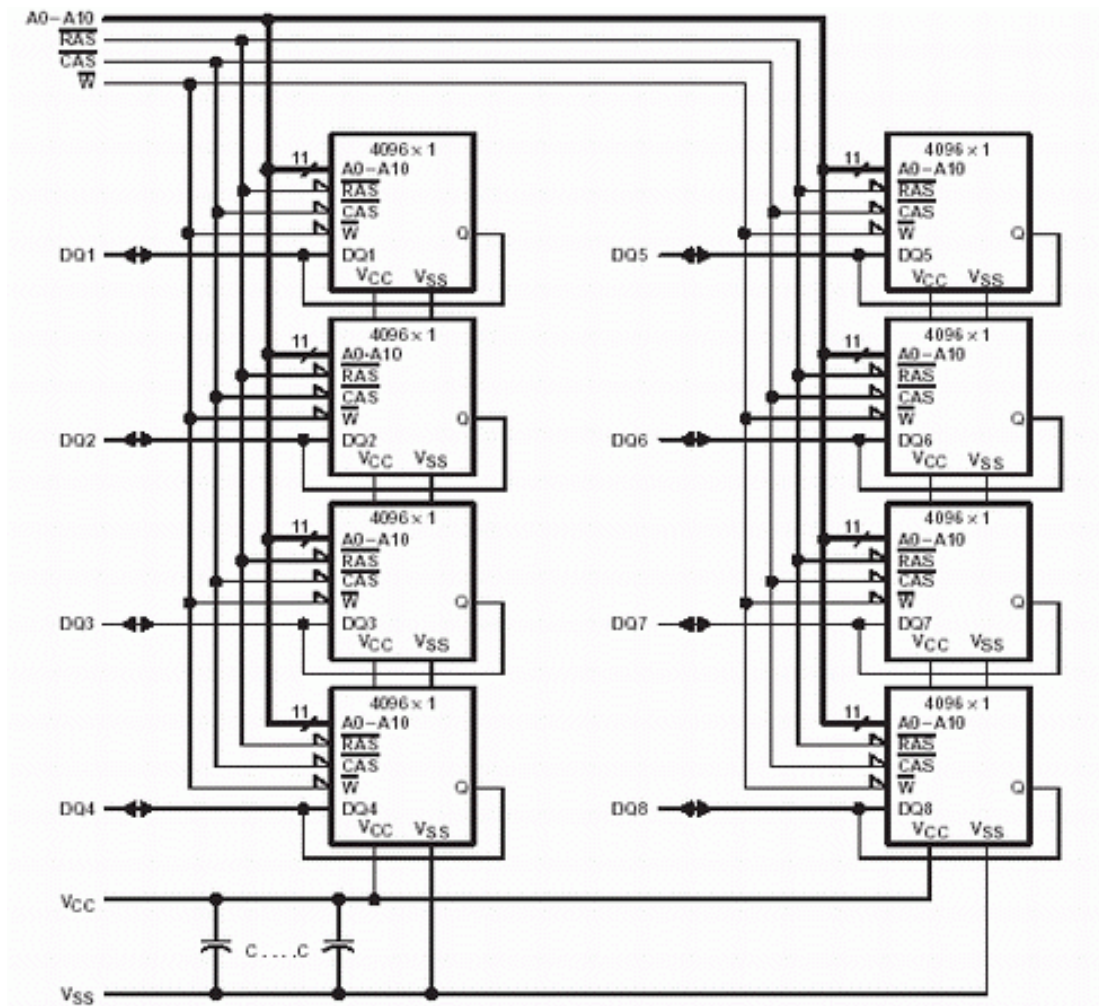
为了举例说明，我们用一条 TI (德仪公司) 出品的 TM4100GAD8 SIMM 内存为例来说明，因为这种内存的构造相对比较简单，便于大家理解。TM4100GAD8 基于 4M x 8 模块制造，容量 4MB，采用 30 线 SIMM 封装。如果前面我说的东西你看明白了，就应该知道这条内存采用了 4Mx1 DRAM 颗粒。下面的数据是我在 TI 官方网站上找到的 (目前很少有公司的网站还提供自己以前产品的数据)：

- 构造：4194304 x 8
- 工作电压：5-V
- 30 线 SIMM (Single In-Line Memory Module：SIMM)
- 采用 8 片 4Mbit DRAM 内存颗粒，塑料 SOJs 封装
- 长刷新期 16 ms(1024 周期)

SINGLE IN-LINE MODULE
(TOP VIEW)



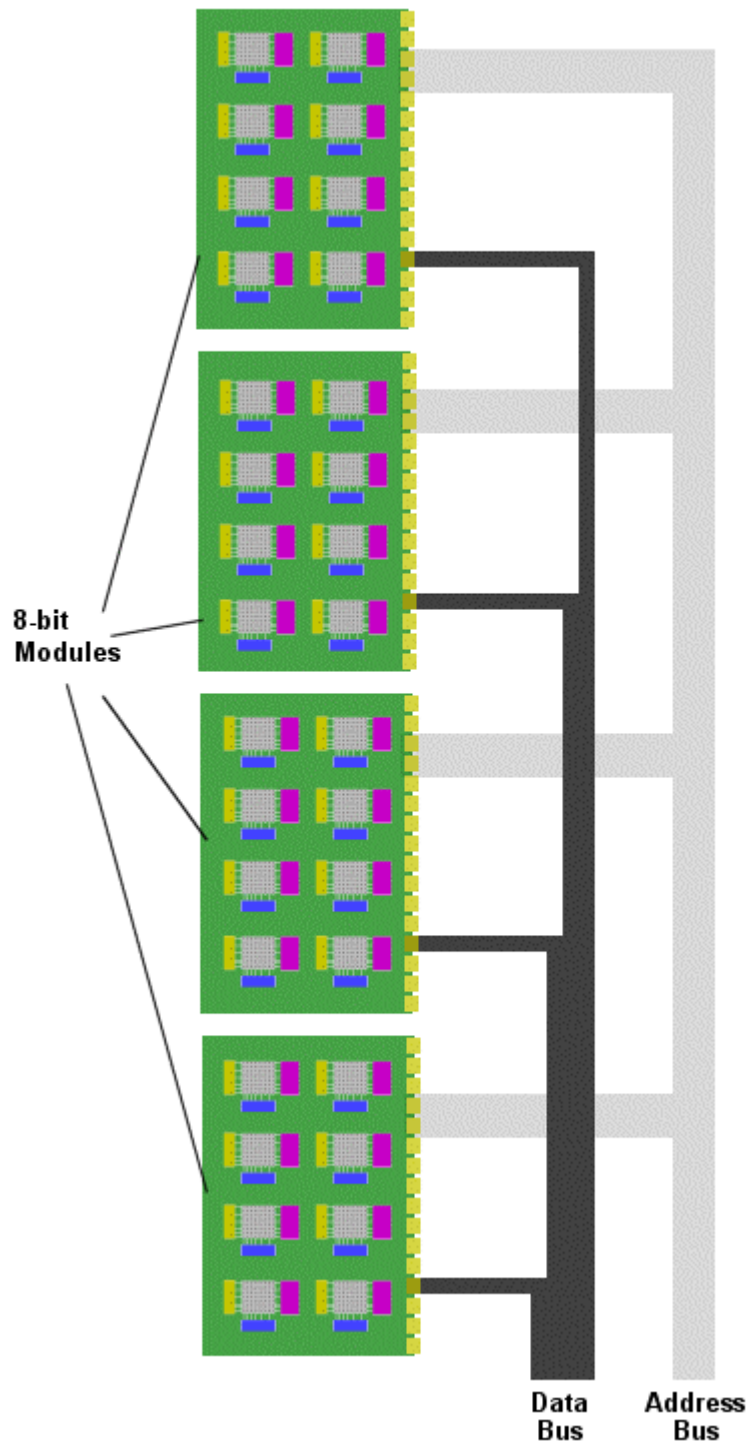
- 在上面的示意图中，A0 – A10 是地址输入引脚
- /CAS:行地址脉冲选通器引脚
- DQ1 – DQ8：数据输入/数据输出引脚
- NC：空信号引脚
- /RAS：列地址脉冲选通器引脚
- V_{SS}：接地引脚
- /W：写入启用引脚
- V_{CC} 5V 供电引脚



上面的电路示意图应该能够让我们更加清楚的理解这个问题, TM4100GAD8 由 8 片 4096x1bit 芯片组成, V_{CC} 和 V_{SS} 为所有的芯片提供 5v 的电压。每个芯片都具有 \overline{RAS} 、 \overline{CAS} 、 \overline{W} 引脚同内存相应的引脚连通。每个芯片都具有不同的数据输出/输出接口。这样我们就应该就能够知道 RAM 是如何满足 CPU 存取数据的需要的了。

关于 Bank 的问题

前面我们讲述的都是 8bit 的内存，现在这种东西我们基本上都接触不到了，更常用的是 32bit、64bit 或者 128bit。由于前面我们已经讲到了 4Mx1bit 模块实现 bit 输入输出的方法，所以我们很容易想到我们把足够多的芯片放在一个模块中就可以了。不过在实际应用中，仅仅这样做还是不行，这里就需要引入 bank 的概念，bank 是由多个模块组成的。请看下面的示意图：



上面的示意图显示的是由4组8bit模块组成的一个bank,如果构成模块的是4194304x1芯片,那么每个模块的架构应该是4194304x8(4MB),这样4个模块就能组成一个位宽为32bit的bank,容量为16MB。当存储数据的时候,第一模块存储字节1,第二个模块存储字节2,第三个模块存储字节3,第四个模块存储字节4,第五个模块存储字节5.....如此循环知道达到内存所能达到的最高容量。

文章读到这里,我们应该能知道,当我们的系统使用这种类型的内存时,可以通过两种方式来增加这种类型内存的容量。第一种就是通过增加每一个独立模块的容量来增加bank的容量,另外一个方法就是增加bank的数目。这样如果让这种类型的内存的容量提升到32MB,可以把每个模块的容量从4MB提升到8MB或者增加bank的数目。

前面我们用来举例的这种 30 线的 SIMM 一般是用在 486 级别的电脑上的，而现在的 Pentium 级别的电脑所使用的内存同这个是不同的。而截止到现在，我的这篇文章还没有涉及到我们目前所使用的内存，不过不要着急，相信充分的理解我现在所谈论的东西将有助于你理解以后的内容。不过这里可以先告诉大家的是 Pentium 级别的内存和 486 系统的内存之间的主要差异在于它们的 RAM 芯片。

SIMM 和 DIMM

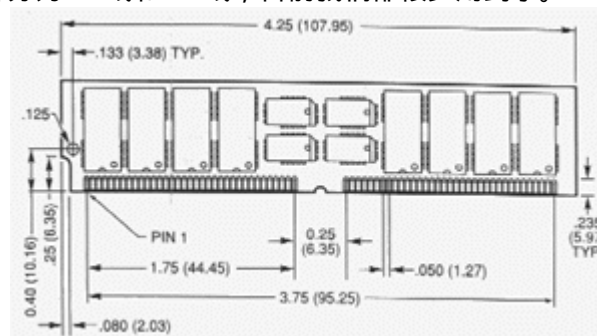
前面我们既然提到了 30 线的 DIMM，那么我们就来介绍一下 SIMM 以及与之相对应的 DIMM。其实 SIMM 和 DIMM 都是内存条的封装形式的一种（这里说的不是芯片的封装形式），因为每片内存颗粒无法直接同计算机进行连接并且通讯的，并且它们单颗颗粒的容量有限而且涉及到前面提及的数据传输位宽等方面的原因，所以内存厂商需要通过一定的形式把它们组织到一起，这样就产生了不同的内存封装形式。

首先我们来介绍一下 SIMM，如下图（上面一条是 30 线 DIMM 内存，下面一条是 72 线 DIMM 内存）：



在 DIMM 内存中的颗粒采用了 DIP（Dual Inline Package：双列直插封装）封装，如上图图中黑色的芯片。早期的内存颗粒是直接焊接在主板上面的，这样如果一片内存出现故障，那么整个主板都要报废了。后来在主板上出现了内存颗粒插槽，这样就可以更换内存颗粒了，但是热膨胀的缘故，每使用一段时间你就需要打开机箱把内存颗粒按回插槽。

除了这些原因，更重要的是我们前面提到的数据总线位宽等方面的原因使得工程师着手设计了 SIMM（Single Inline Memory Module）封装和 DIMM（Double Inline Memory Module）的内存，它们通过主板上的内存插槽同主板进行通讯。这样的设计解决了原来所有的问题。SIMM 内存根据引脚分为 30 线和 72 线，目前我们都很少用到了。



SIMM Diagram

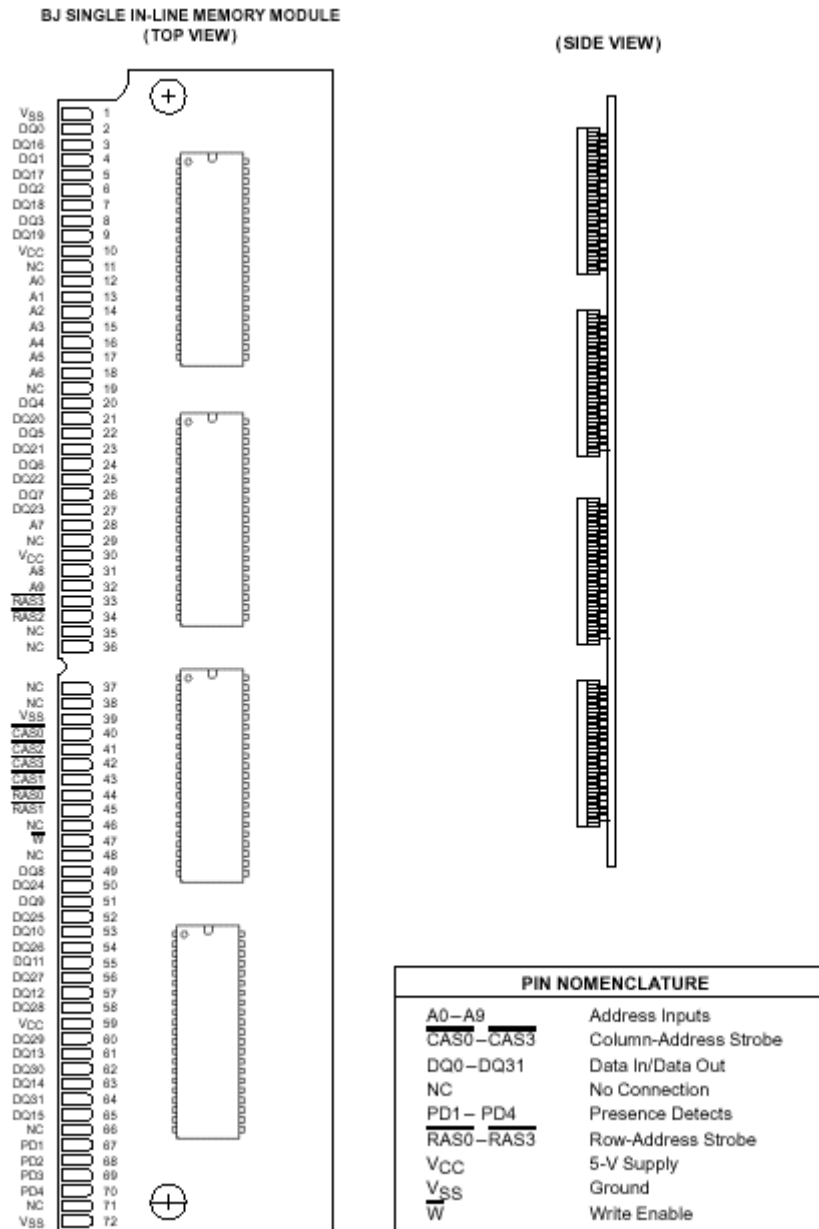
SIMM 根据内存颗粒分布可以分为单面内存和双面内存，一般的容量为 1、4、16MB 的 SIMM 内存都是单面的，更大的容量的 SIMM 内存是双面的。在我们本文中所列举的 TM4100GAD8 就是一款 30 线的内存，它每次仅能传输 8bit 的数据 - - 从前面的示意图中我们也知道这 30 线引脚中有 11 线是地址引线，8 线是数据引线，还有其它的控制引线，对于当时的封装工艺这已经是比较不错了。比较细心的读者会问为什么还有三条空信号引脚？因为这种内存的数据输出总线位宽只有 8bit，所以即使将空信号引脚转换为地址总线提高寻址范围，但是并没有足够多的引脚用于数据的输出。72 线的 SIMM 内存的容量不但可以更大，而且数据总线的位宽也得到了极大的提高。一条 72 线 SIMM 内存的数据总线位宽是 32bit，它的数据输出能力大大提高了。



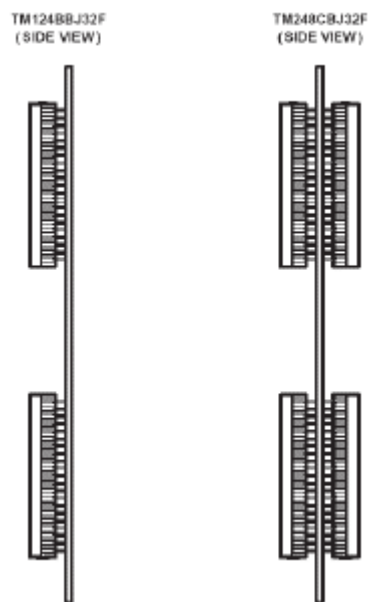
DIMM 是目前我们使用的内存的主要封装形式,比如 SDRAM、DDR SDRAM、RDRAM,其中 SDRAM 具有 168 线引脚并且提供了 64bit 数据寻址能力。DIMM 的工作电压一般是 3.3v 或者 5v,并且分为 unbuffered 和 buffered 两种。上图上面的内存就是 168 线的 SDRAM,而下面的内存是 72 线的 SIMM。需要指出的是在 SIMM 和 DIMM 内存之间不仅仅是引脚数目的不同,另外在电气特性、封装特点上都有明显的差别,特别是它们的芯片之间的差别相当的大。因为按照原来内存制造方法,制造这种内存的时候是不需要把 64 个芯片组装在一起构成一个 64bit 的模块的,得益于近年来生产工艺的提高和改进,现在的高密度 DRAM 芯片可以具有不止一个 Din 和 Dout 信号引脚,并且可以根据不同的需要在 DRAM 芯片上制造 4、8、16、32 或者 64 条数据引脚。

如果一个 DRAM 芯片具有 8 个数据引脚,那么这个基本储存单元一次就可以输出 8bit 的数据,而不像是在原来的 TM4100GAD8 SIMM 芯片中每次仅仅能输出 1bit 数据了。这样的话,如果我们需要制造一个同 TM4100GAD8 一样容量的内存,那么我们可以不使用前面所使用的 4M x 1bit 芯片,而是采用 1M x 8bit 芯片,这样仅仅需要 4 片芯片就可以得到一个容量为 4MB,位宽为 32bit 的模组。芯片数目减少最直接的好处当然是可以减少功耗了,当然也简化了生产过程。

下面的图只是为了说明这个问题而制作的,它展示的是一种 72 线的 4MB SIMM 内存,采用了 4 片 1Mx8bit DRAM 芯片。但是至于是不是真的有这样的一款产品我也不能确定,因为目前为之我找不到实际的产品相关资料,所以这个只是为了帮助大家理解这个问题,不要对于是否有这样的产品而斤斤计较。

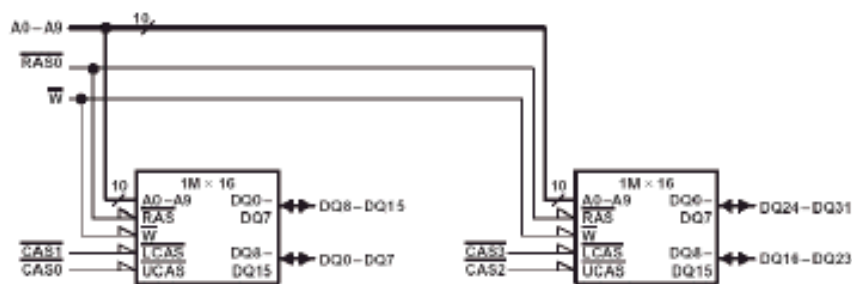


这样一来，只要 4 片采用具有 8bit 位宽的内存颗粒就可以达到同样的容量，当然这样的内存条工作原理在理解的过程中比原来略微复杂一点。我们看到在上面的 4Mbit × 8bit 芯片中，依然还是有 10 条地址总线引脚，但是/CAS 和/RAS 引脚却从原来的 1 条增加到 4 条。当然数据输入输出引脚线数目是 32 条。其实 TI 公司的 TM124BBJ32F 和 TM248CBJ32F 前面的我所列举的例子是比较相似的：



这两款内存的容量均为 4MB,位宽为 32bit,当然也属于 DRAM 了。TM124BBJ32F 内存为单面而 TM248CBJ32F 双面的两种模式,不过其中单面 TM124BBJ32F 有些奇怪,在它的内存条上只有两颗内存芯片,这样每颗内存芯片应该是 2MBx16bit。另外,双面的 TM248CBJ32F 由 4 片 1Mx8bit DRAM 芯片组成。

functional block diagram (TM124BBJ32F and TM248CBJ32F, side 1)



functional block diagram (TM248CBJ32F, side 2)

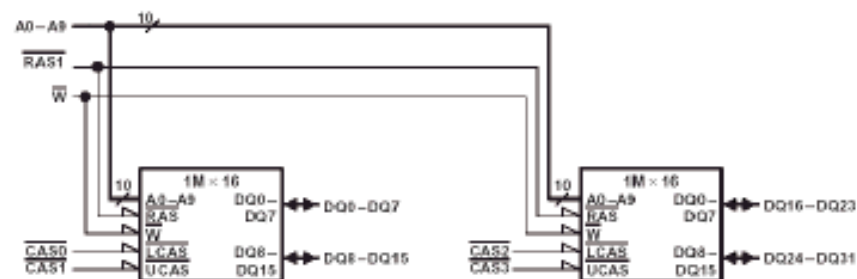


Table 1. Connection Table

DATA BLOCK	RASx		CASx
	SIDE 1	SIDE 2†	
DQ0-DQ7	RAS0	RAS1	CAS0
DQ8-DQ15	RAS0	RAS1	CAS1
DQ16-DQ23	RAS2	RAS3	CAS2
DQ24-DQ31	RAS2	RAS3	CAS3

† Side 2 applies to the TM248CBJ32F only

上面的示意图和表格是 TM124BBJ32F 和 TM248CBJ32F 的示意图和表格,我们可以很容易的理解它们的工作模式。

对于 TM124BBJ32F 来说：因为是 2MBx16bit 的颗粒，所以当 RAS0 引脚为低电平时，DQ0-DQ15 输出/输入引脚有效，所以它可以同时传送一个 16bit 数据；让 RAS1 引脚为低电平时，DQ16-DQ31 输出/输入引脚有效，也可以一次传送一个 16bit 数据。

对于 TM248CBJ32F 来说：因为是 1MBx8bit 的颗粒，所以情况同前面是不同的，当 RAS0 引脚为低电平时，DQ0-DQ7 输出/输入引脚有效，所以它可以同时传送一个 8bit 数据；让 RAS1 引脚为低电平时，DQ6-DQ15 输出/输入引脚有效，也可以一次传送一个 8bit 数据；让 RAS2 引脚为低电平时，DQ16-DQ23 输出/输入引脚有效，也可以一次传送一个 8bit 数据；让 RAS3 引脚为低电平时，DQ24-DQ31 输出/输入引脚有效，也可以一次传送一个 8bit 数据 (注意这里虽然都是控制输出 8bit 或者 16bit 地址，但是它们之间分别代表的含义是不同的)。当然在确定地址的时候，还是需要 CAS 控制电路配合的。

今天我们对于 32bit 的内存做了进一步的了解。这些东西虽然对于现在的用户似乎有些不太实际，但是对于进一步了解现在的内存还是有相当的帮助的。(未完待续.....)

DRAM 读取过程

这个系列的文章已经写了两篇了，但是我们几乎还没有谈到我们都关心的一些问题，比如 CAS-2 和 CAS-3 之间的区别什么的。现在我们对于 DRAM 的基础知识已经有了一个基本的了解，下面的文章就是给大家介绍一些现代的内存技术。

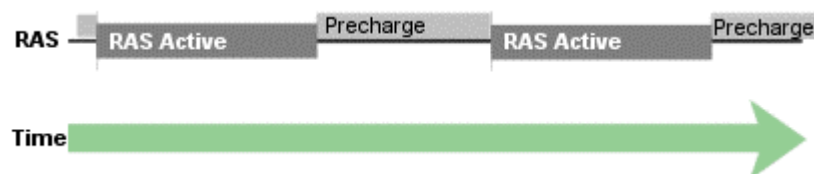
DRAM 读取过程

其实我们在以前的文章中已经讨论过 DRAM 的读写过程了，不过对于内存比较了解的朋友都会发现在前面的介绍中仅仅是对于内存的大致读取过程进行了简述，很多重要的细节都没有详细的讨论。所以我们在文章的这一节的内容中对于这个过程进行详细的讨论。下面就是异步内存的读取过程的步骤，因为异步 DRAM 的运行并不需要同处理器同频，它的时序信号控制、寻址等操作基本上说是独立控制的，也就是由内存芯片本身所控制，所以在讨论起来比较简单，我们仅仅需要考虑 DRAM 本身的情况就可以了（这个系列的文章也是本着循序渐进的原则让大家更好的理解内存的工作原理的）：[BLOCKQUOTE]

- 1) 行地址通过地址总线传输到地址引脚
- 2) /RAS 引脚被激活，列地址就会被放入行地址选通电路(Row Address Latch:在文章的前面部分我们把它翻译为列地址门电路)
- 3) 行地址解码器 (Row Address Decoder) 选择正确的行然后送到传感放大器 (sense amps)
- 4) /WE 引脚此时不被激活，所以 DRAM 知道它们不是进行写操作
- 5) 列地址通过地址总线传输到地址引脚
- 6) /CAS 引脚被激活，列地址就可以被送到列地址选通器 (Column Address Latch)
- 7) /CAS 引脚也被当作输出启动信号 (Output Enable)，因为一旦 /CAS 信号被放到传感放大器，就因为这时需要的数据已经找到，所以 Dout 引脚开始有效，数据可以从内存中传输到系统了
- 8) /RAS 和 /CAS 引脚停止激活，等待下一个读取命令[/BLOCKQUOTE]

在内存的读取过程中，需要我们考虑的有两个主要类型的延迟。第一类的是连续的 DRAM 读操作之间的延迟。内存不可能在进行完一个读取操作之后就立刻进行第二个读取操作，因为 DRAM 的读取操作包括电容器的充电和放电另外还包括把信号传送出去的时间，所以在两个读取操作中间至少留出足够的时间让让内存进行这些方面的操作。

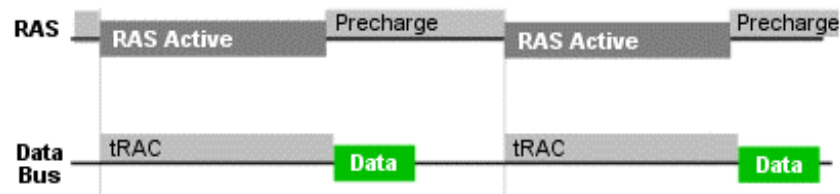
在连续的两次读取操作之间，第一种类型的延迟包括 /RAS 和 /CAS 预充电延迟时间。在 /RAS 被激活并且失活之后，你必须给它足够的时间为下次激活做好准备。下图可以帮助你更好了解这个过程。



CAS 预充电的过程是一样的，你只要把上图种的“RAS”换成“CAS”就可以了。

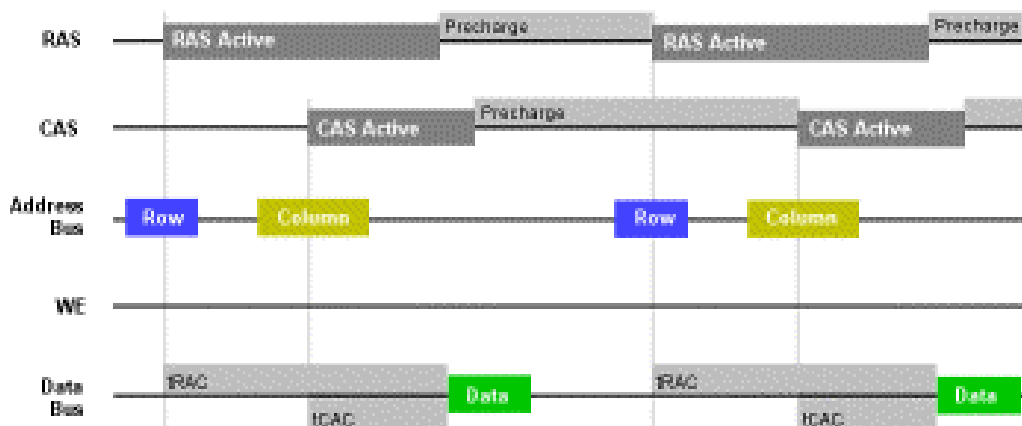
从前面我们介绍的 DRAM 读取过程的 8 个步骤中，我们可以了解到 /RAS 和 /CAS 预充电过程是依次进行的，所以我们在一定的时间里只能进行有限次数的读取操作。特别是在第 8 个步骤中，当一次读取操作周期结束之后，我们必须让 /RAS 和 /CAS 引脚都失活。实际上，在你让它们失活之后，必须等待预充电过程结束之后才能开始下一个操作（或者是读取操作、或者是写入操作、或者是刷新操作）。

当然在两次读取操作之间的预充电时间不是限制 DRAM 速度的唯一因素。第二种延迟类型是叫做内部读取延迟 (inside-the-read)。这种延迟同两次读取操作之间的延迟非常的相似，但是不是由停止 /RAS 和 /CAS 激活而产生的，而是由于要激活 /RAS 和 /CAS 而产生的。比如，行存取时间(t_{RAC}) - - 它就是在你激活 RAS 和数据最终出现在数据总线之间的时间。同样的列存取时间 (t_{CAC})就是激活 /CAS 引脚和数据最终出现在数据总线上之间的时间。下面的示意图可以帮助你更好的理解这两种类型的延迟：



上面的图仅仅是一个示意图，下面的时序图可以帮助你了解不同的延迟时间发生的顺序：

DRAM Read



现在让我们花一点时间结合前面介绍的读取过程来研究一下上面的这张示意图：

1) 首先看上图第一行，在预充电期间行地址通过地址总线传输到地址引脚，这个期间 RAS 未被激活，在第三行 Address BUS 中我们看到数据在这个期间正在行地址总线上，这个期间 CAS 也处于预充电状态；

2) 依然看上图第一行，/RAS 引脚被激活（RAS Active，灰色的部分），列地址就会被放入行地址选通电路(第三行 Address BUS 中所示)，这个期间 CAS 依然处于预充电状态；在 /RAS 被激活的同时，tRAC（行存取时间）开始 - - 如上图最后一行 Data BUS 所示。

3) 在 /RAS 被激活以后，行地址解码器（Row Address Decoder）选择正确的行然后送到传感放大器（sense amps）

4) 在这个期间 /WE 引脚一直处于不激活的状态，所以 DRAM 知道它们不是进行写操作 - - 这个状态将一直持续到开始执行写操作才结束

5) 列地址通过地址总线传输到地址引脚

6) /CAS 引脚被激活（如上图第三行），列地址就可以被送到列地址选通器（Column Address Latch），这个时候 tCAC（列地址访问时间）开始计时

7) 在 /CAS 处于激活状态期间的末尾，/RAS 停止激活 - - 也就大约在这个时间附近找到的数据被传送到数据总线进行数据传送（如图 data BUS），在数据总线进行数据传输的过程中，地址总线是处于空闲状态的，它并不接受新的数据 - - 在数据开始创送的同时 tRAC 和 tCAC 都结束了。

8) 就在数据在数据总线上传输期间 /CAS 引脚也被停止激活 - - 就是得到一个高电平，从而开始进入到预充电期。RAS 和 CAS 会同时处于预充电期，直到下次 /RAS 被激活进入到下一个读取操作的周期。

相信经过这样的说明大家应该了解 DRAM 的读取过程了。在这个基础上我们就可以开始认识 SIMM 或者 DIMM 的潜伏期（latency）问题了。首先我们来继续澄清一下几个概念。

DRAM 潜伏期类型分为两种：访问时间（access time）和周期时间（cycle time）。其中访问时间（access time）同前面我们谈论的第二种类型的延迟有关，也就是同读取周期中的延迟时间；而周期时间（cycle time）同我们前面谈论的第一种类型的延迟有关，也就是受

到两个读取周期之间的延迟时间影响。当然潜伏期的时间很短，都是用纳秒来衡量的。

对于异步 DRAM 芯片，访问时间就是从行地址到达行地址引脚的时间起截至到数据被传输到数据引脚的时间段。这样，访问时间为 60 纳秒的 DIMM 意味着当我们下达读取数据的命令后，地址数据被送到地址引脚之后要等待 60 纳秒才能达到数据输出引脚。周期时间，从字面上理解就是从两个连续读取操作之间的时间间隔。如何尽可能的减小 DRAM 的周期时间和访问时间是我们这篇文章后半部分将要详细的讨论的问题。

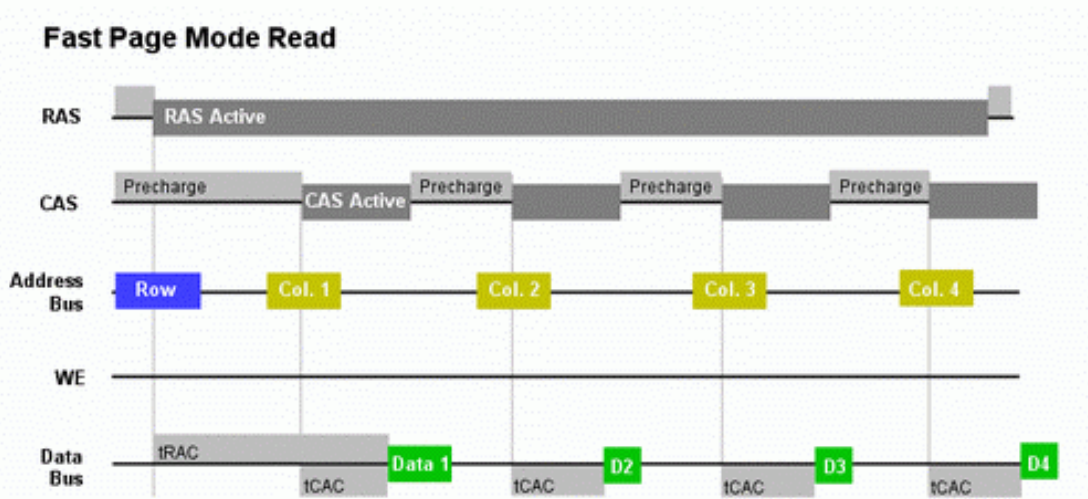
我们平时说到 DRAM 内存是多少多少纳秒,这里指的一般是访问时间(我们也会对于为什么采取这样的标称方法进行解释)。我们都知道访问时间越短,意味着内存工作频率会越高。当然内存工作频率越高,意味着可以适应外频更高的处理器。如果处理器的时钟周期较短,而 DRAM 的潜伏期较长,处理器在很多时间里都是等到 DRAM 传送数据。因此当 DRAM 一定时,比如时潜伏期为 70 纳秒,那么一颗 1GHz PIII 等待数据的时间将会比一颗 400MHz PII 处理器长。当然出现这样的现象是每个用户都不愿意看到的,当的使用的内存速度越慢或者说你的处理器相对越快,你的处理器就会由更多的性能都被这样的等待浪费了。(未完待续.....)

快页模式内存

在《深入了解内存（三）——DRAM 读取过程》一文中我们介绍了 DRAM 的详细读取过程，在本文的这一节中我们将要开始了解 FPM DRAM (Fast Page Mode DRAM)，也就是我们常说的快页内存。之所以称之为快页内存，因为它以 4 字节突发模式传送数据，这 4 个字节来自同一列或者说同一页。

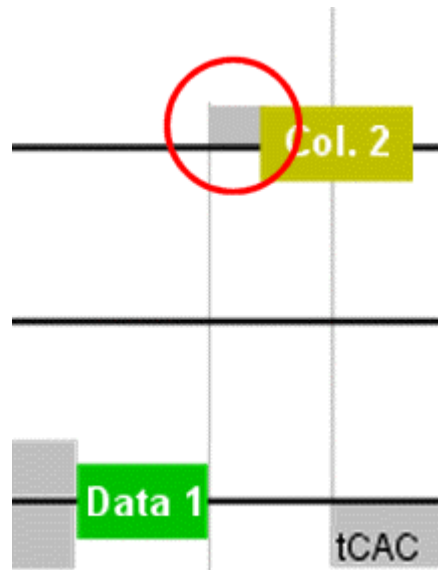
如何理解这种读取方式呢？FPM DRAM 如果要突发 4 个字节的数据，它依然需要依次地读取每一个字节的数据，比如它要读取第一个字节的数据，这个时候的情况同前面介绍的 DRAM 读取方式是一样的（我们依然通过读取下面的 FPM 读取时序图来了解它的工作方式）：

1. 首先行地址被传送到行地址引脚，在 /RAS 引脚被激活之前，RAS 处于预充电状态，CAS 也处于预充电状态，当然 /WE 此时依然是高电平，FPM 至少知道自己不会进行写操作。
2. /RAS 引脚被赋予低电平而被激活，行地址被送到行地址选通器，然后选择正确的行送到传感放大器，就在 /RAS 引脚被激活的同时，tRAC 开始计时
3. CAS 一直处于预充电状态，直到列地址被传送到列地址引脚并且 /CAS 引脚得到一个低电平而被激活（tCRC 时间开始计时），然后下面的事情我们也应该很清楚了，列地址被送到列地址选通器，然后需要读取的数据位置被锁定，这个时候 Dout 引脚被激活，第一组数据就被传送到数据总线上
4. 对于原来介绍的 DRAM，这个时候一个读取周期就结束了，不过对于 FPM 则不同，在传送第一组数据期间，CAS 失活（RAS 依然保持着激活状态）并且进入预充电状态，等待第二组列地址被传送到列地址引脚，然后进行第二组数据的传输，如此周而复始直至 4 组数据全部找到并且传输完毕
5. 当第四组数据开始传送的时候，RAS 和 CAS 相继失活进入到预充电状态，这样 FPM 的一个完整的读取周期方告结束。FPM 之所以能够实现这样的传输模式，就是因为所需要读取的 4 个字节的行地址是相同的但是列地址不同，所以它们不必为了得到一个相同的列地址而去做重复的工作。
6. 这样的工作模式显然相对于普通的 DRAM 模式节省了很多的时间，特别是节省了 3 次 RAS 预充电的时间和 3 个 tRAC 时间，从而进一步提高的效率。

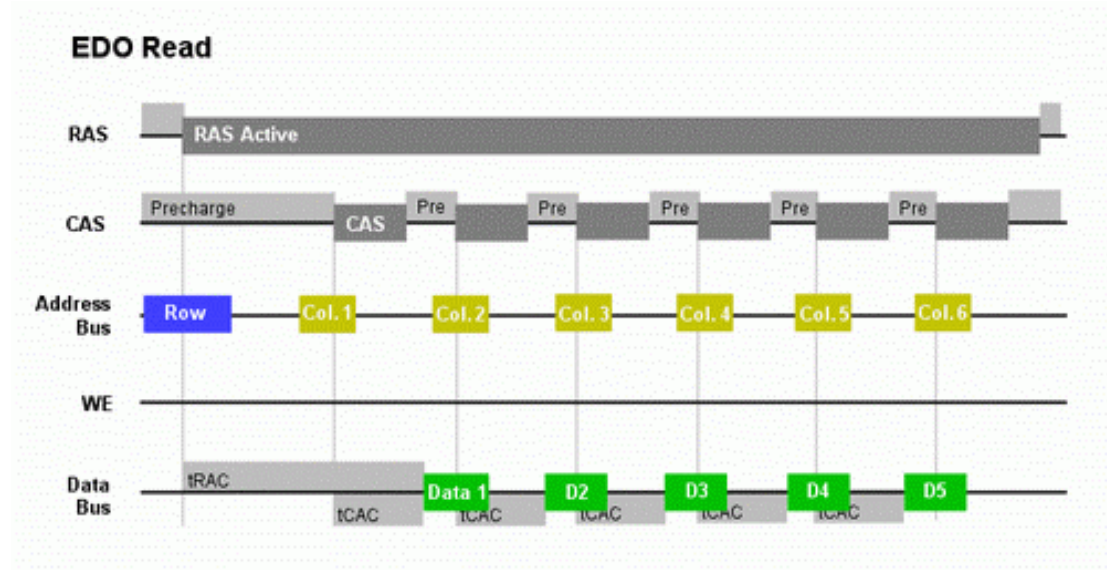


我想你一定看到过诸如 6 - 3 - 3 - 3 这样的内存标注方法，其中的 6 表示从最初状态读取第一组数据需要 6 个时钟周期，而读取另外三个数据仅仅需要 3 个时钟周期就能达到目的了。

需要特别指出的是，在上面的时序图中，我们并没有标注出 FPM DRAM 进行第二个、第三个、第四个数据输出的前进行新的列地址选通的时间，但是从上面的示意图中我们可以看到 Col.2 同 Data1 和 D2 之间都没有重叠，所以这三个数据的输出是进行完毕一个再进行的另一个，因此再上一次数据传输完毕到下一次列地址传输之间还有一点小小的延迟。请看下图：



EDO DRAM (Extended Data Out DRAM : 扩展数据输出 DRAM) 在介绍 FPM 的读取过程的最后我着重提到了 FPM DRAM 是在上一次的数据读取完毕才会进行下一个数据的读取，但是对于 EDO DRAM 却是完全不一样的。 EDO DRAM 可以在输出数据的同时进行下一个列地址选通,我们依然结合下面的 EDO 读取时序图来了解 EDO DRAM 读取数据的过程：



1. RAS 在结束上一次的读取操作之后，进入预充电状态，当接到读取数据的请求之后，行地址首先通过地址总线传输到地址引脚，在这个期间 CAS 依然处于预充电状态
2. /RAS 引脚被激活，列地址开始经过行地址选通电路和行地址解码器进行行地址的选择，就在这个同时 tRAC 周期开始，因为是读取操作 /WE 引脚一直没有被激活，所以内存知道自己进行的是读取操作而不是写操作
3. 在 CAS 依然进行预充电的过程中，列地址被送到列地址选通电路选择出来合适的地址，当 /CAS 被激活的同时 tCAC 周期开始，当 tCAC 结束的时候，需要读取的数据将会通过数据引脚传输到数据总线。
4. 从开始输出第一组数据的时候，我们就可以体会到 EDO 同 FPM 之间的区别了：在 tCAC 周期结束之前，CAS 失活并且开始了预充电，第二组列地址传输和选通也随即开始，第一数据还没有输出完毕之前，下一组数据的 tCAC 周期就开始了 - - 显然这样进一步的节省了时间。就在第二组数据输出前，CAS 再次失活为第三组数据

传输列地址做起了准备.....

5. 如此的设计使得 EDO 内存的性能比起 FPM 的性能提高了大约 20-40%.
6. 正是因为 EDO 的速度比 FPM 快,所以它可以运行在更高的总线频率上。所以很多的 EDO RAM 可以运行在 66MHz 的频率上,并且一般标注为 5 - 2 - 2 - 2。

SDRAM

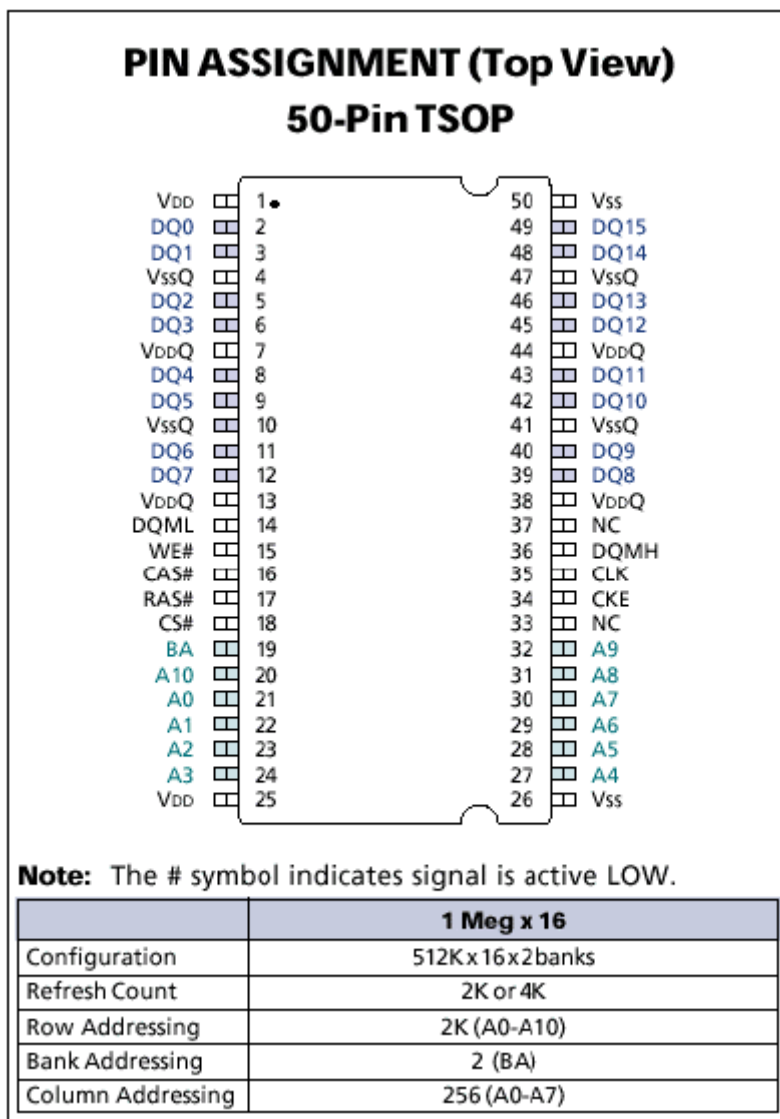
主要谈论我们大家都能接触到的 SDRAM 内存了,首先得承认 SDRAM 同我们之前介绍的异步 DRAM 是差别非常大的。它的基本原理同前面提到的 DRAM 还是基本一样的(比如基本存储单元都是按照阵列排列,都有 RAS 和 CAS 的概念),不过这些存储单元的组织和控制同 DRAM 就具有相当大的差别了。在前面我们讨论其它类型的内存都是采用了以具体的产品为例来讲述的,所以这里我们依然采用这种形式,这次我们以 MICRON MT48LC4M4A1 16MB SDRAM 为例。(如果你感兴趣可以去 www.micronsemi.com 网站查找相应的资料)。

如果你还记得我们在前面介绍的 DRAM 相关内容,那么应该还记得 DRAM 是以 bank 来组织存储单元的。因为每个内存 bank 的位宽是同数据总线阿位宽一样的。所以对于 SIMM,你必须把多个 SIMM 放在一个 bank 之中来满足 32bit 或者 64bit 数据总线的要求。DIMM 具有更多阿引脚,所以单个 DIMM 可以提供足够的同数据总线相适应的位宽 - - 这样每个 bank 只要一个 DIMM 就可以了。而且 SDRAM 更进一步的是可以在单个的 DIMM 中存在多个 bank,这样不但能够满足数据总线的需要还能进一步的提高总线的性能。

下面就让我来解释一下:

在我们前面讨论的 DRAM 读取方式中,当一个读取周期结束后,/RAS 和/CAS 都必须停止激活然后有一个短暂的预充电期才能进入到下一次的读取周期中。但是一个具有两个 bank 的 SDRAM 的模组中,其中一个 bank 在进行预充电的期间另一个 bank 却可以被调用 - - 这样当你需要读取已经预充电的 bank 的数据时,就无需等待而是可以直接调用了。为了实现这个功能,SDRAM 就需要增加对于多个 bank 的管理,这样就可以实现控制其中的 bank 进行预充电,并且在需要使用的时候随时调用了。这样一个具有两个 bank 的 SDRAM 一般会多一根叫做 BA0 的引脚,实现在两个 bank 之间的选择 - - 一般的 BA0 是低电平表示 Bank0 被选择,而 BA0 是高电平 Bank1 就会被选中。

可见,虽然 SDRAM 在基本的原理上比如基本存储的结构都是基本一样的,但是在整个内存架构的组织上是不同的,而且在存储单元的控制上也是有着相当大的区别的。因为异步 DRAM 同处理器和芯片的时钟并没有什么关系,所以芯片组只能按照 DRAM 内存的时序要求“被动”的操作 DRAM 控制引脚。SDRAM 因为要同 CPU 和芯片组共享时钟,所以芯片组可以主动的在每个时钟的上升沿发给引脚控制命令。



上图显示的就是 MT48LC4M4A1 16MB SDRAM 内存颗粒的引脚示意图，它采用了 50 引脚的 TSOP 封装，符合 PC100 规范。这种内存颗粒将同系统时钟同步运行。这种内存颗粒的架构 1Mx16-512Kx16x2，每 bank 行地址数目是 11，列地址数目是 8。我们首先来介绍一下这种内存颗粒的引脚定义：

- A0 - A10：地址输入引脚，当 ACTIVE 命令和 READ/WRITE 命令时，来决定使用某个 bank 内的某个基本存储单元。
- CLK：时钟信号输入引脚
- CKE：Clock Enable，高电平时有效。当这个引脚处于低电平期间，提供给所有 bank 预充电和刷新的操作
- /CS：芯片选择 (Chip Select)，SDRAM DIMM 一般都是多存储芯片架构，这个引脚就是用于选择进行存取操作的芯片
- /RAS:行地址选择(Row Address Select)
- /CAS:列地址选择(Column Address Select)
- /WE:写入信号(Write Enable)
- DQ0 - DQ15:数据输入输出接口
- BA：Bank 地址输入信号引脚，BA 信号决定了由激活哪一个 bank、进行读写或者预充电操作；BA 也用于定义 Mode 寄存器中的相关数据。
- NC：空引脚
- DQM: 这个引脚的主要用于屏蔽输入/输出，功能相当于/OE 引脚(Output Enable)。
- VDDQ：DQ 供电引脚，可以提高抗干扰强度

- VSSQ：DQ 供电接地引脚
- VSS：内存芯片供电接地引脚
- VDD：内存芯片供电引脚，提供+3.3V \pm 0.3V

(上面的列表项目和示意图中，前面标有“/”或者“#”标记的表示在低电平下有效)

下面的表格在不同的状态下（或者说不同命令下）的各个引脚的信号。“H”代表高电平，“L”代表低电平，“X”代表可以是任何状态，也就是该引脚同该命令并没有直接的关系。

功能	/CS	/RAS	/CAS	/WE	DQM	ADDR.
COMMAND INHIBIT (NOP)	H	X	X	X	X	X
NO OPERATION (NOP)	L	H	H	H	X	X
ACTIVE (选择 bank 并且激活相应的行)	L	L	H	H	X	Bank/Row
READ (选择 bank 和列地址，并且开始突发读取)	L	H	L	H	X	Bank/Col
WRITE (选择 bank 和列地址，并且开始突发写入)	L	H	L	L	X	Bank/Col
BURST TERMINATE(停止当前的突发状态)	L	H	H	L	X	X
PRECHARGE (让相应的 bank 中的行失活或者让该 bank 失活)	L	L	H	L	X	Code
AUTO REFRESH(进入自动刷新模式)	L	L	L	H	X	X
LOAD MODE REGISTER	L	L	L	L	X	Op-code
写入启用/输出启用					L	
写入禁止/输出禁止					H	

如果你对于我们前面介绍的内容有了真的有所了解了，看到上面的芯片引脚示意图和各个针脚的功能示意图就基本对于 SDRAM 的工作工程有了一个基本的了解了，在下面的章节里我们就对于这个过程进行详细的介绍，首先我们对于一些基本的概念做一些了解。

这条 SDRAM 颗粒采用了双 bank（每 bank 512K x 16 DRAM）的工作电压是 3.3V，并且采用同步接口方式（所有的信号都是时钟信号的上升沿触发）。每一个 512K x 16-bitbank 由 2,048 行乘以 256 列个基本存储单元构成，输出数据位宽是 16 bit。Read 和 write 操作都是通过突发导向模式访问 SDRAM 的；这种访问模式以访问指定的区域开始的，然后按照预先设定的方式定位其它的数据的所在。每次访问都是以 ACTIVE 命令启动的，然后仅仅跟着一个 READ 或者 WRITE 命令。不过在进行所有这些操作之前，SDRAM 必须首先进行初始化。

• 初始化

SDRAM 在上电之后，必须首先按照预定的方式进行初始化才能正常的运行。一旦 VDD 和 VDDQ 被同时供电并且时钟稳定下来，SDRAM 就需要一个 100 微秒的延迟，在这个时间段中 COMMAND INHIBIT 和 NOP 指令有效，这个过程实际上就是内存的自检过程，一旦这个过程通过之后一个 PRECHARGE 命令就会紧紧随着最后一个 COMMAND INHIBIT 或者 NOP 指令而生效，这个期间所有的内存都处于空闲（idle）状态，随后会执行两个 AUTOREFRESH 周期、当 AUTOREFRESH 周期完毕之后，SDRAM 为进行 Mode Register 编程做好了准备。因为 Mode Register 上电会引起一个为止的状态，它会在进行所有正常指令之前被载入。至此，初始化过程完成。

• MODE REGISTER

Mode Register 一般被用于定义 SDRAM 运行的模式。其中包括了突发长度（burst length）、突发类型（burst type）、CAS 延迟（CAS latency）、运行方式（operating mode）和写入突发模式（如 Figure 1 所示）。Mode Register 通过 LOAD MODE REGISTER 命令进行编程，这组信息将会一直保存在 Mode Register 中直到内存掉电之后才会消失。Mode Register 中的 M0-M2 是用来定义突发长度（burst length）的，M3 定义突发类型（sequential 或者

interleaved),M4-M6 定义 CAS 延迟,M7 和 M8 定义运行模式, M9 定义写入突发模式 (write burst mode),M10 和 M11 目前保留。Mode Register 必须在所有的 bank 都处于 idle 状态下才能被载入,在所有初始化工组都进行完毕之前,控制器必须等待一定的时间。在初始化过程中发生了任何非法的操作都可能导致初始化失败从而导致整个计算机系统不能启动

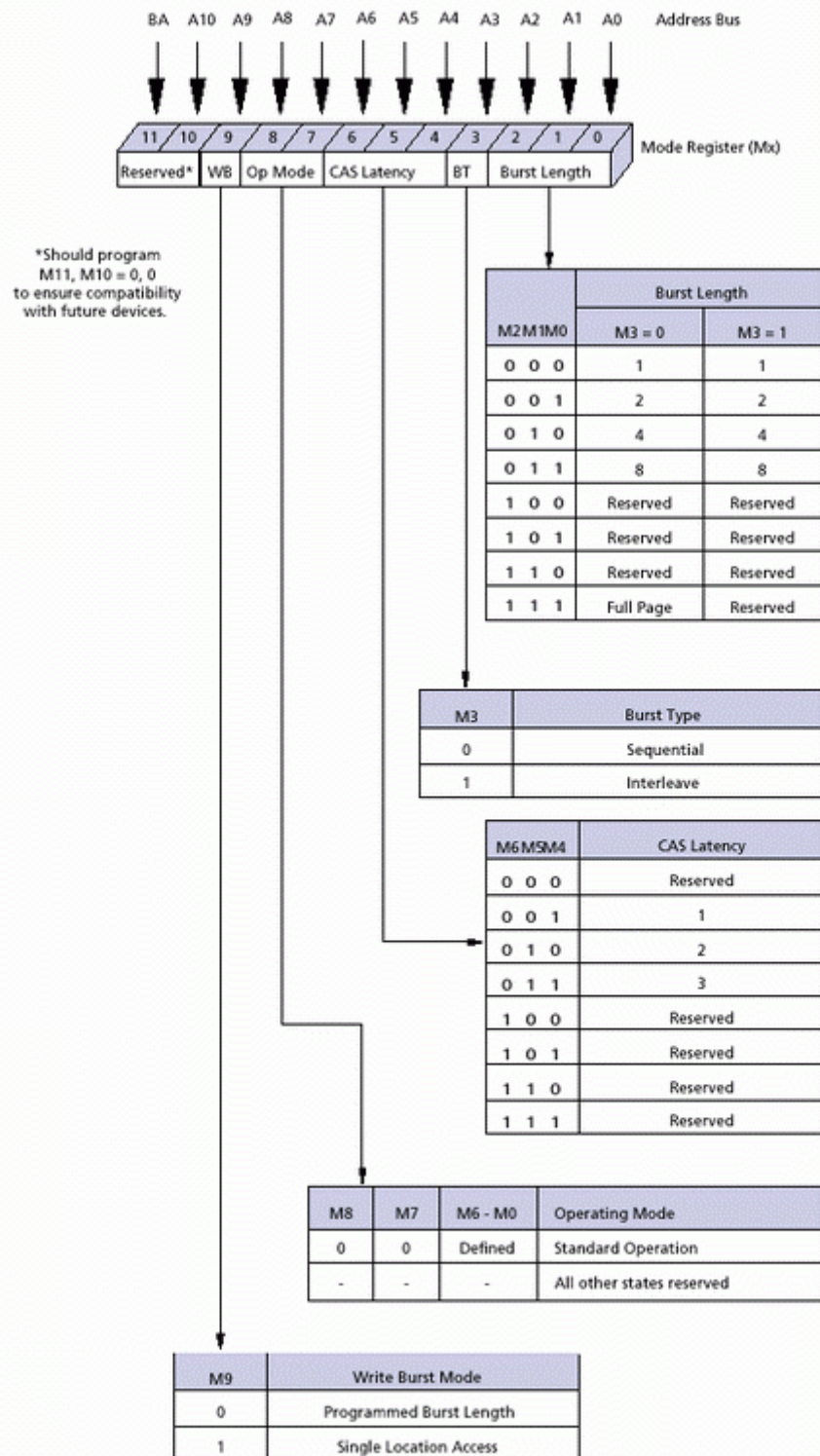


Figure 1
Mode Register Definition

- 突发长度 (Burst Length)

Read 和 write 操作都是通过突发模式访问 SDRAM 的,当然突发模式的长度都是在初始化过程中载入 Mode Register 中载入的参数,这些参数当然是由厂商或者用户定义的。在 Figure 1 中我们看到突发长度决定了 READ 或者 WRITE 命令能够访问的列地址的最大数目。对于 sequential 和 interleaved 这两种突发模式它们的突发长度是 1、2、4、8,另外全页 (full - page) 突发模式仅仅适用于 sequential 类型。全页突发可以用 BURST TERMINATE 命令连接来产生任意的突发长度。保留状态 (Reserved states) 主要用于应付未来的不兼容的情况而准备的。当一个 READ 或者 WRITE 命令被发出之后,这个时候突发长度就被选定了。所有的访问操作都会以这个突发长度为限进行读取操作。当突发长度设为 2 时,A1-A7 将会作为数据输入输出的列地址线;当突发长度设定为 4 时,A2-A7 将会作为数据输入输出的列地址线;当突发长度被设定为 8 时,A3-A7 将会作为数据输入输出的列地址线。

- 突发类型 (Burst Type)

突发类型主要分为两种:sequential 和 interleaved——主要由 M3 所决定。访问顺序主要由突发长度、突发类型和起始列地址所决定,如下表:

Burst Definition

Burst Length	Starting Column Address	Order of Accesses Within a Burst	
		Type = Sequential	Type = Interleaved
2	A0		
	0	0-1	0-1
	1	1-0	1-0
4	A1 A0		
	0 0	0-1-2-3	0-1-2-3
	0 1	1-2-3-0	1-0-3-2
	1 0	2-3-0-1	2-3-0-1
	1 1	3-0-1-2	3-2-1-0
8	A2 A1 A0		
	0 0 0	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7
	0 0 1	1-2-3-4-5-6-7-0	1-0-3-2-5-4-7-6
	0 1 0	2-3-4-5-6-7-0-1	2-3-0-1-6-7-4-5
	0 1 1	3-4-5-6-7-0-1-2	3-2-1-0-7-6-5-4
	1 0 0	4-5-6-7-0-1-2-3	4-5-6-7-0-1-2-3
	1 0 1	5-6-7-0-1-2-3-4	5-4-7-6-1-0-3-2
	1 1 0	6-7-0-1-2-3-4-5	6-7-4-5-2-3-0-1
	1 1 1	7-0-1-2-3-4-5-6	7-6-5-4-3-2-1-0
Full Page (256)	n = A0-A7 (location 0-255)	Cn, Cn+1, Cn+2 Cn+3, Cn+4... ...Cn-1, Cn...	Not supported

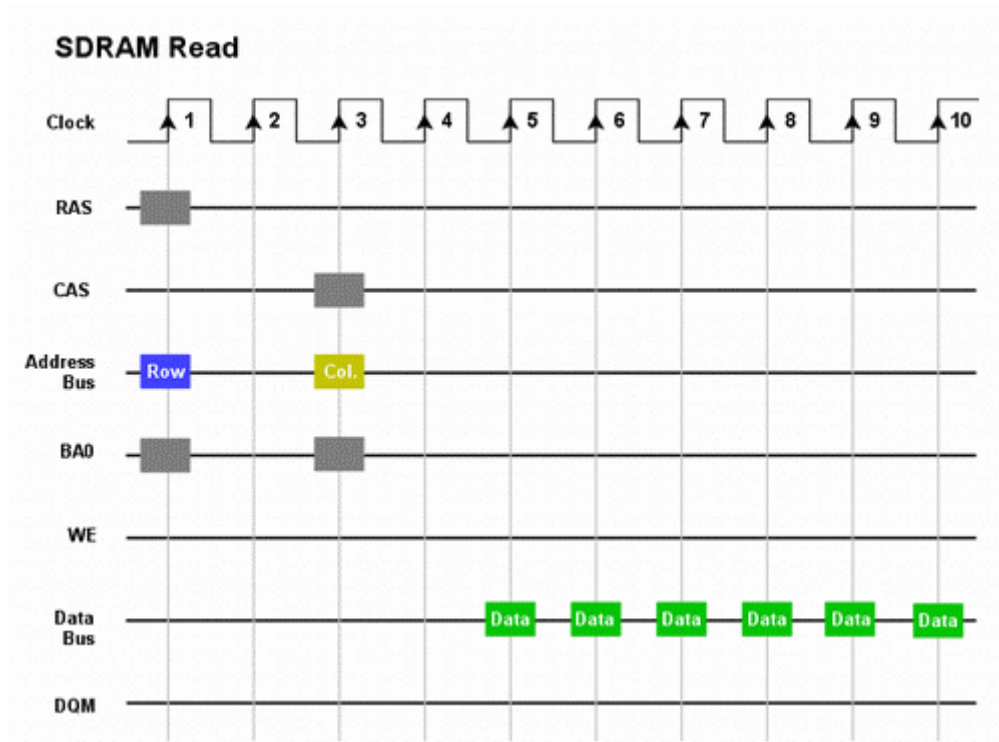
- 禁止指令 (COMMAND INHIBIT) 和空指令 (NOP)

这两条指令是 CS 信号的两个不同的状态。前面我们提到 /CS 信号可以赋予芯片两种状态:参与工作和休息。/CS 处于未激活状态(也就是禁止指令生效的时候),SDRAM 此时不对于任何传送到这个引脚上的指令作出反应;/CS 引脚处于激活状态的情况下才会对于传送到引脚上的指令作出反应。

空指令 (NOP) 这个指令将激活 /CS,但是它仅仅告诉芯片什么说不作——但是为什么要有这样的看似多余的指令呢?在后面的 CAS 延迟时间中我们将会涉及到。

- ACTIVATE、READ 和 WRITE

如果要了解基本的 READ 和 WRITE 操作，那么你就应该需要了解这三条指令。ACTIVATE 指令的主要就是选择一个 bank 并且激活相应的行；READ 指令就是读取指定的行的数据；WRITE 指令就是在指定的 bank 和列中写入数据。下面就让我们看看 SDRAM 读取时序图：



1) 行地址通过地址总线传输到地址引脚 (Address Bus 行)，当在第一个时钟周期的上升沿，通过 ACTIVATE 指令通过 /CS 激活了相应的行地址 - - 当然确定指定的行之前 BA0 引脚会选择相应的 bank。

2) /WE 引脚在这个过程中一直不会被激活，所以 SDRAM 知道它们不是进行写操作

3) 第三个时钟周期的开始，ACTVATE 指令激活了/CAS 并且得到指定的列地址

4) 第五个到第十个时钟周期的上升沿都会有数据输出到数据总线。

我们看到 SDRAM 基本的读取过程非常的简单。今天就先谈到这里，在以后的文章中我们将会对于 SDRAM 相关的问题继续进行讨论。(未完待续.....)

SDRAM 读取过程分析

在《深入了解内存（四）》一文中，我们对于 SDRAM 的读取过程中的基本概念做了比较详细的了解。在这个系列的文章中，我们继续对于 SDRAM 的读取过程进行更加详细的讨论。

BANK/行的激活过程

在进行任何的 READ 或者 WRITE 命令之前，SDRAM 首先要选择进行操作的 bank，并且还要打开这个 bank 中的相应的行。完成这个任务要通过 ACTIVE 命令来实现，（具体的说，ACTIVE 命令主要在 subsequent 访问模式用于打开/激活所选 bank 中的行。其中 BA 引脚输入的信号决定 bank 的选择，而 A0 - A10 地址引脚决定行地址。被打开的行保持打开/激活状态直到下一次所在 bank 执行 Precharge 命令 - 一般的一个 Precharge 命令用于在同一个 bank 中打开不同的行之前）如下图所示显示的 CS（芯片选择）信号是低电平处于有效状态；RAS 信号是低电平处于有效状态；CAS 信号是高电平，表示还没有进行列选择；WE 信号是高电平，表现芯片不会进行写操作：

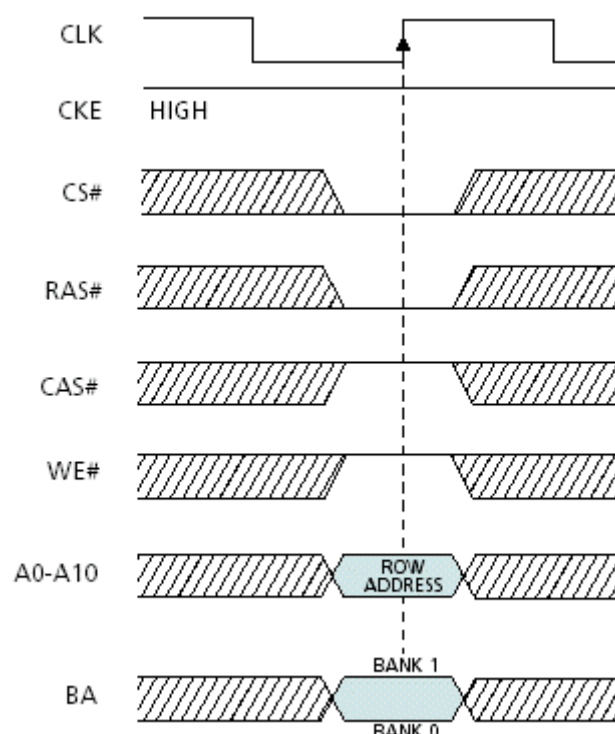


图 1

当 ACTIVE 命令执行完毕之后，需要进行操作的 bank 中的行就会被打开，这个时候就可以进一步执行 READ 或者 WRITE 命令，当然进行这个命令要受限于 t_{RCD} (Time - RAS to CAS Delay) 规格。 $t_{RCD}(\text{MIN})$ 就是从开始执行 ACTIVE 命令到执行 READ 或者 WRITE 命令的时间差。比如说， t_{RCD} 如果为 20ns，当时钟频率为 125MHz（也就是每周期 8ns）时， t_{RCD} 所占据的时钟周期为 2.5-3 个时钟周期。请看图 2，一般的情况下 $2 < t_{RCD}(\text{MIN})/t_{CK}$ 3。

当 SDRAM 需要在同一个 bank 中打开另外一行时，Precharge 命令会先把已经打开的行关闭，然后由一个 ACTIVE 来打开新的行。同一个 bank 中，在两个相邻的 ACTIVE 命令之间的时间间隔叫做 t_{RC} 。在不同的 bank 之间，执行两个 ACTIVE 命令之间的时间间隔叫做 t_{RRD} 。

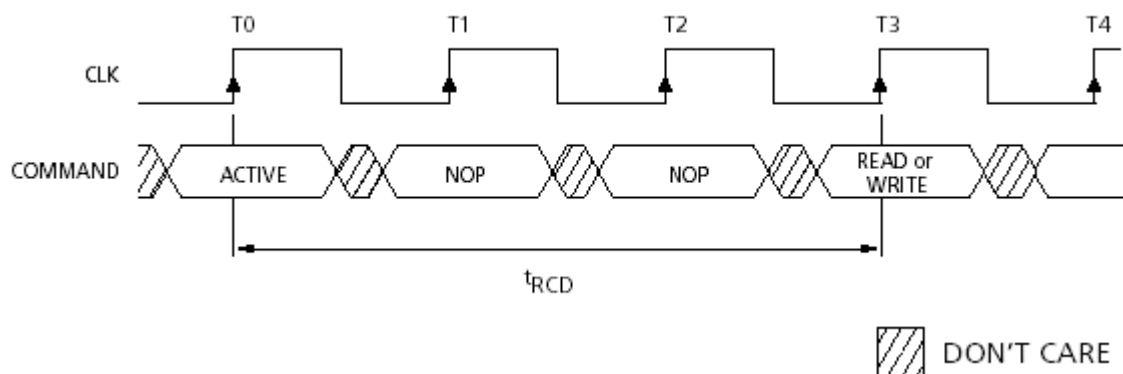


图 2

SDRAM 读取状态进阶分析

当行地址选定并且相应的行被打开之后，READ 命令将要开始执行，如图 3 所示。BA 引脚依然决定对于哪个 bank 进行操作，A10 引脚的信号决定了是否进行 AUTO PRECHARGE - - 一般的读取操作中 AUTO PRECHARGE 处于低电平也就是无效状态，如果它处于高电平就说明在读取突发进行完毕之后所读取的行会进入预充电状态，该行也会从打开状态变为关闭状态。A0 - A7 传输列地址数据。CS 依然处于低电平状态，保证对于需要操作芯片的选择。RAS 此时处于高电平，因为该行已经打开直到执行 PRECHARGE 命令才会关闭，所以 RAS 此时处于无效状态。因为这个时候是对于列的选择，所以 RAS 处于低电平状态，进行列地址的选择。因为是读取操作 WE 当然是高电平，处于无效的状态。

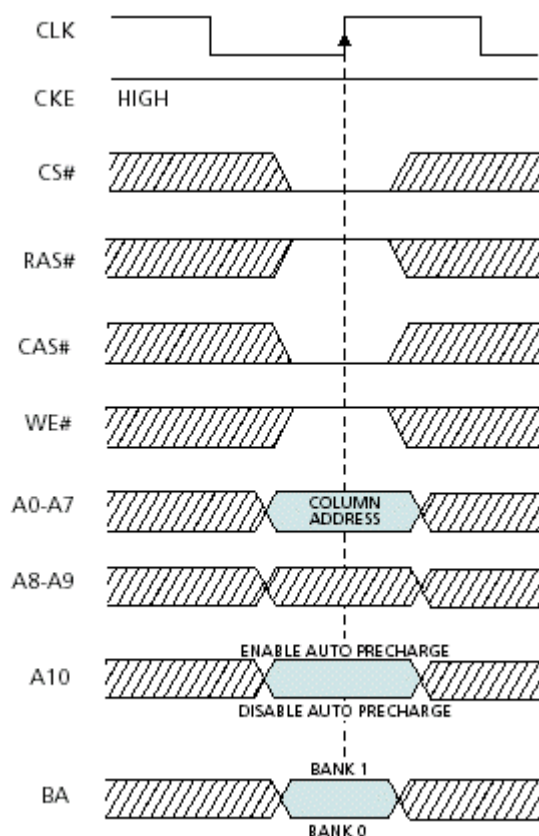


图 3

在读取突发过程中，当 READ 命令开始执行之后经过 CAS 延迟之后的时间数据就会出现现在数据总线上，当然随后相邻的数据都是在紧接着的时钟周期的上升沿依次发出。图 4 所示的就是在各种 CAS 延迟设定下的从 READ 命令到数据输出的情况：

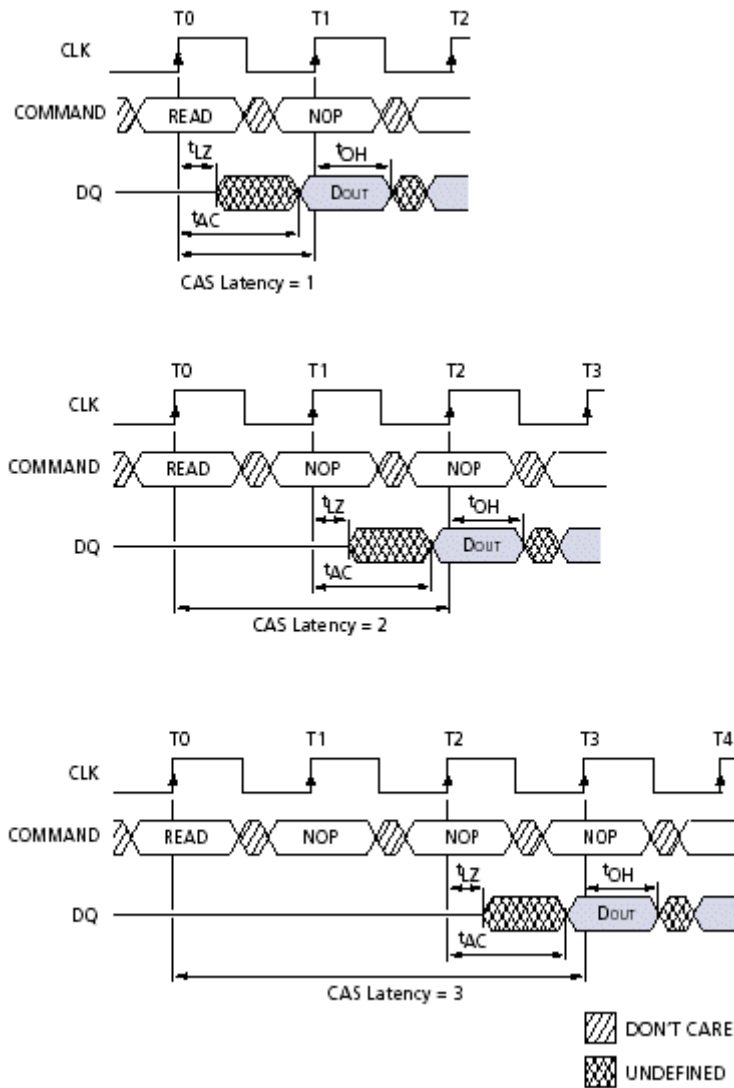


图 4

当完成了读取突发操作之后，如果这个时候没有其它的命令进行初始化，那么 DQs 将会进入到 HighZ 状态。然后内存就会进入全页突发模式，直到被中止 - - 否则的话，即使是读取到了页尾，它会从该行的列 0 重新开始读取。任何的 READ 突发能够被持续 READ 命令所中止，并且具有新的突发长度的 READ 命令所产生的数据后可以紧紧跟着前面 READ 突发命令所产生的数据发出。这样就能够保持数据的连贯性。图 5 可以让你理解这种情况：

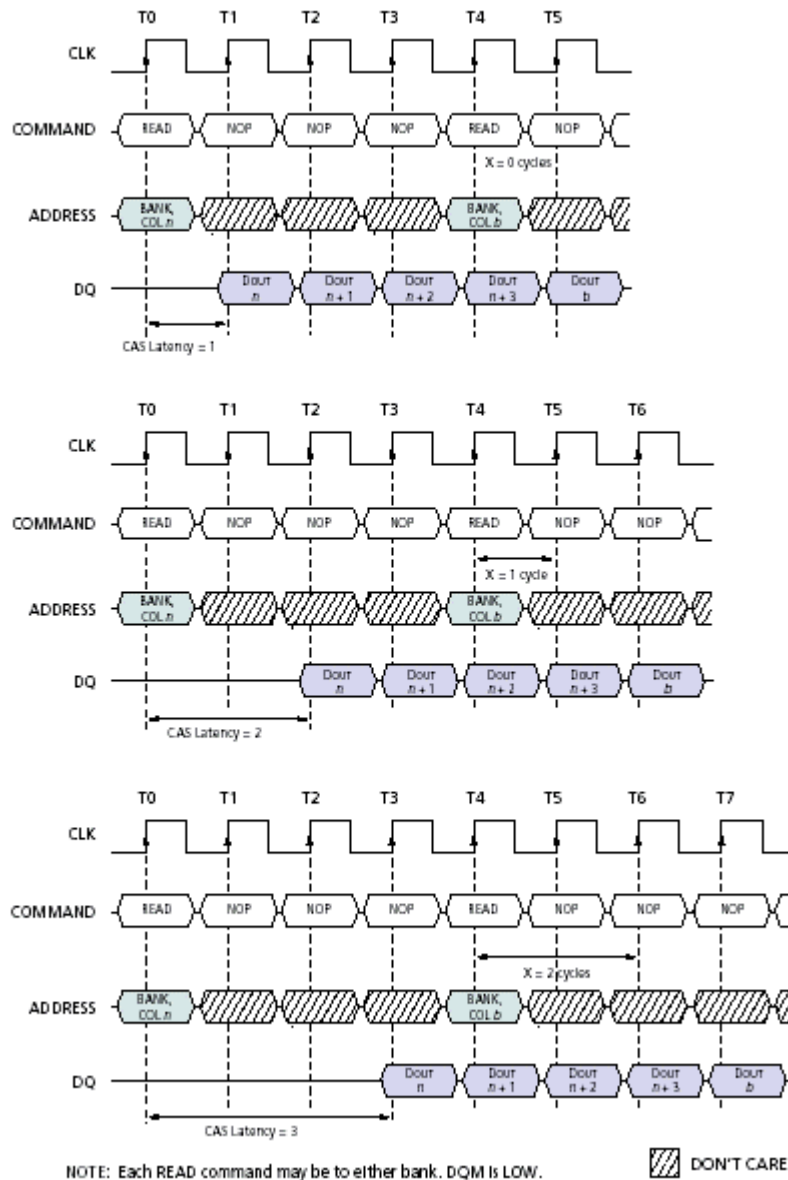


图 5 (顺序读取)

从上图可以看到，为了保证数据的连贯性，新的 READ 突发命令会在上一个 READ 突发命令所产生的数据之前的 x 个周期执行。在这里 x 等于 CAS 延迟时间减 1。图 5 中可以看到，当 CAS 延迟等于 1 时，在 $Dout\ n+3$ 数据传输的时钟周期的上升沿，新的 READ 命令就开始执行 ($1 - 1 = 0$)；同样在 CAS 延迟时间等于 3 时，新的 READ 指令在 $Dout\ n+1$ 数据传输的时钟周期的上升沿开始执行——提前了 $3 - 1 = 2$ 个周期。可见一个 READ 命令可以在前一个 READ 读取过程中的任何一个时钟周期开始初始化。

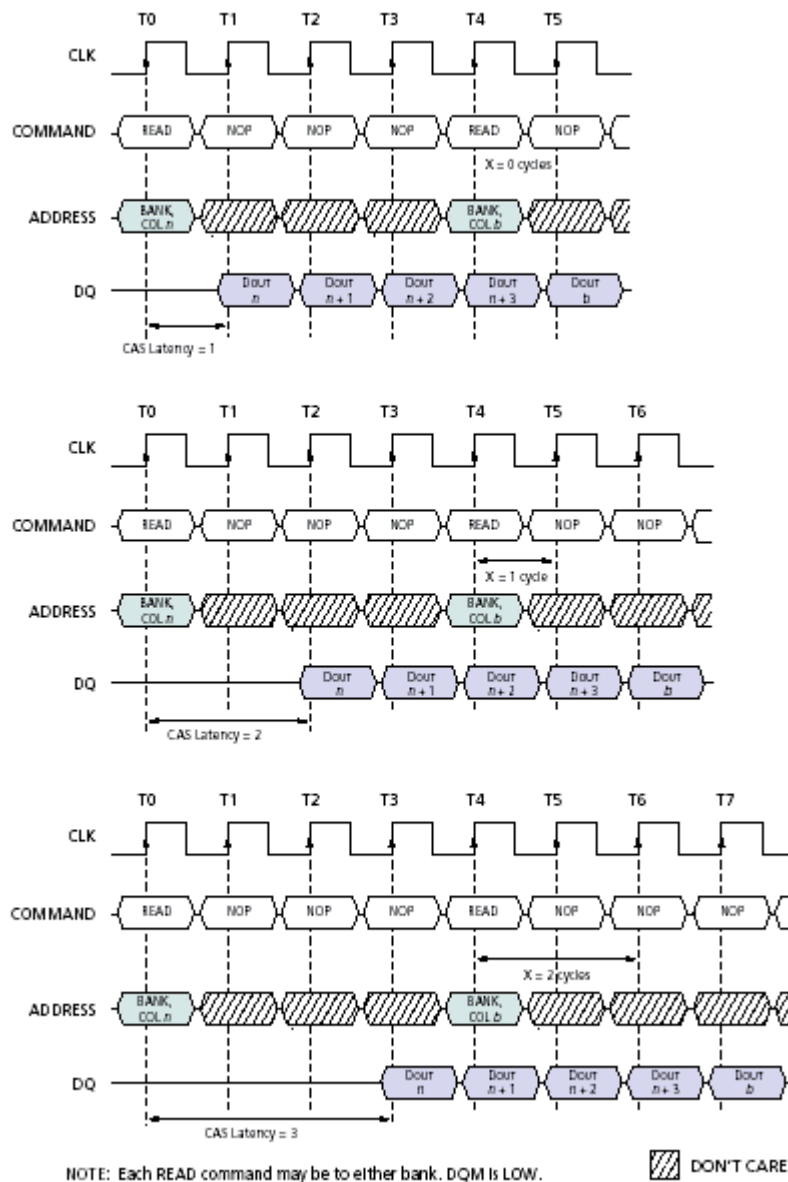


图 6(随机读取)

图 6 所显示的是在 CAS 分别设置尾 1、2、3 的状态下，进行随机读取的操作的情况。当 CAS1 时情况很好理解，新的 READ 指令总是在前一个 READ 指令所产生的数据传输的时钟周期的上升沿开始初始化；在 CAS3 的情况下，为了保证数据输出的连续性，新的 READ 指令必须提前 2 个时钟周期初始化，这样我们就看到新的 READ 指令是紧接着上一个 READ 指令进行初始化的。

这个地方之所以引入 DQM 信号，主要是为了避免 I/O 连接如图 7、8 所示的情况出现。DQM 信号必须维持两个时钟周期的高电平抑制了 READ 数据的输出才能进行 WRITE 命令的初始化 - - 一旦 WRITE 命令被触发，DQ 将会变为 High-Z 状态 - - 这个状态同 DQM 信号的状态并没有关系。只有这样才能进行 WRITE 操作，但是如果数据输出没有被抑制，那么在第二个 WRITE 会发生非法 WRITE 操作。比如，如果图 8 中的 T4 周期 DQM 是低电平，那么在 T5 和 T7 周期的 WRITE 操作将会是有效的，但是在 T6 周期的写入操作将会是非法的。在执行写入操作的过程中，DQM 信号必须成为低电平，这样才能保证写入的数据没有被屏蔽。图 7 和图 8 的主要区别就是后者在写入数据和前一次的数据输出之间插入了 NOP 指令。

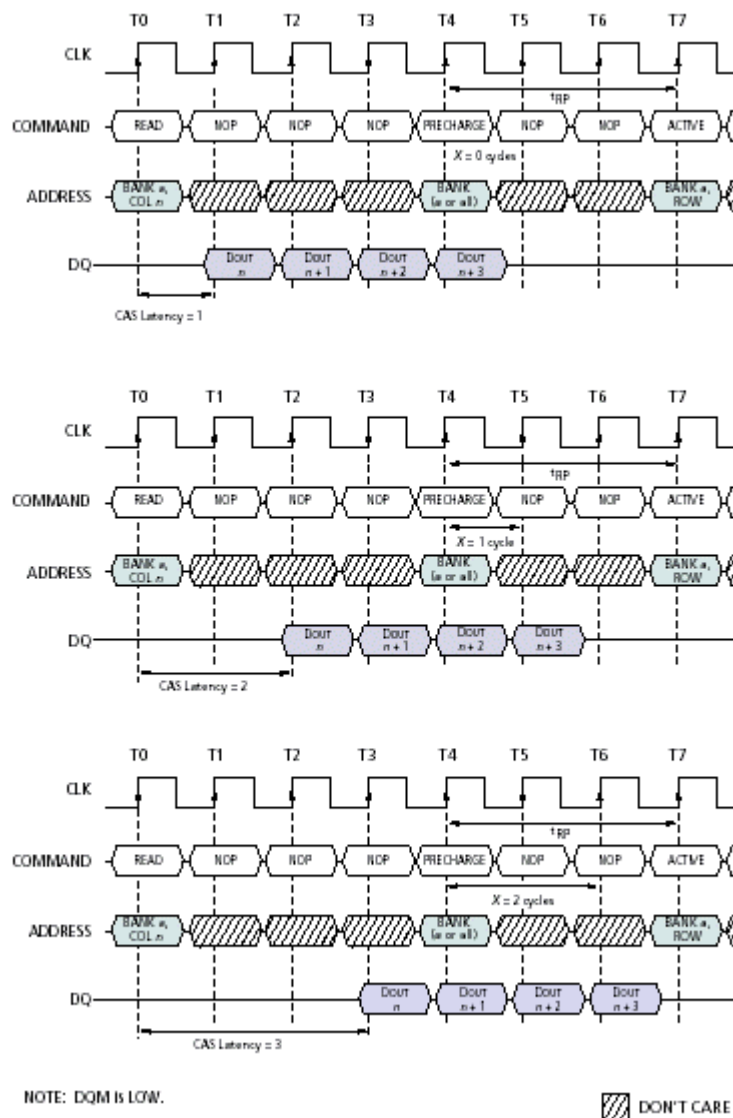


图 9 (READ - 预充电)

前面我们介绍的数据读取都是在打开同一个行的前提下进行的, 现在我们讨论的是进行不同行的数据的读取或者是中止数据的读取。PRECHARGE 命令一般用于同一个 bank 间的不同的行之间的数据读取。PRECHARGE 会在上一次 READ 命令输出数据最后一个时钟周期的前 $(x - 1)$ 个时钟周期执行, 在图 9 中显示的是当 CAS 在各种可能状态下的的情况。当 CAS 等于 1 的时候, PRECHARGE 会提前 $(1 - 1)$ 个时钟周期执行预充电操作, 从这个 PRECHARGE 命令到下一个 ACTIVE 命令之间的时间价格叫做 t_{RP} 。需要注意的是部分预充电时间同上一次的数据传输时间是有重叠的。随着 CAS 延迟时间的延长, 预充电时间同上次读取操作数据输出的重叠时间就越多。因为 PRECHARGE 指令要求命令总线和地址总线都必须在指定的时间段内空闲, 这是应该是它的缺点; 而 PRECHARGE 命令的优点就是它可以被用来中止可变长度或者全页突发 READ 指令。

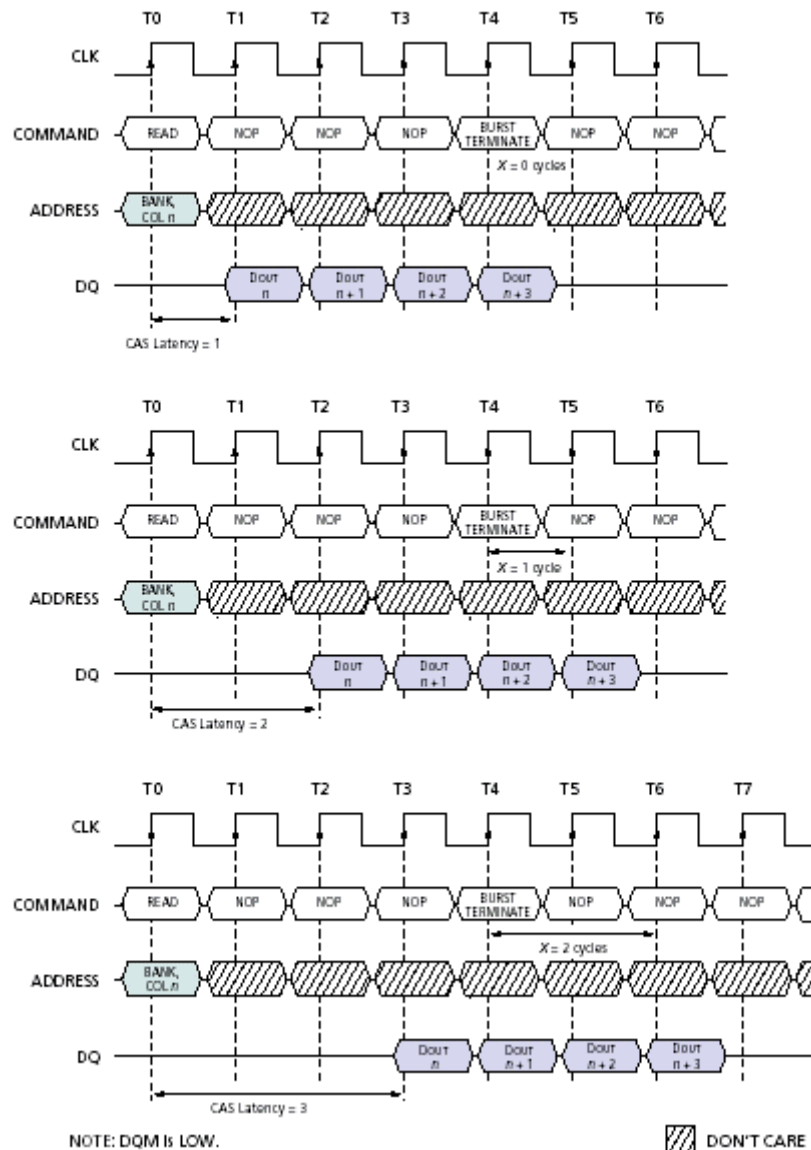


图 10 (READ - 状态中止)

全页 READ 突发和可变长度 READ 突发都可以用 BURST TERMINATE 指令来中止。BURST TERMINATE 指令一般在最后一个所需要的数据传输前的 x 时钟周期有效，这里的 x 依然代表是 CAS 延迟周期减一。在图 10 中很清楚的显示出来了 BURST TERMINATE 的供桌状态，因为前面我已经同大家一起分析了其它情况的工作状态，所以这里我就详细的分析了，留给大家自己思考一下。

今天我们用本章节的篇幅介绍了 SDRAM 读取数据的各种情况，看上去有些复杂，但是非常的有规律，应该不难理解。在随后的文章的中我们会继续对于 SDRAM 的其它方面的情况进行介绍，敬请期待……

SDRAM 写入过程

在前面部分的文章中我们对于 SDRAM 的读取过程进行了详细的讨论，不过今天我发现在其中有个比较重要的概念并没有单独的提出来讲解一下，但是前面对于 SDRAM 读取过程的讲解中如果仔细领会也应该对于 CAS 的概念比较了解了。在本章节的开始我们还是对于这个问题进行一下专门的讨论吧。

SDRAM CAS 延迟

如果你的主板支持对于内存进行比较全面的控制，那么你应该在其中看到有关于 CAS 的选项，一般的可以选择 CAS 1、CAS 2 或者 CAS 3。你选择不同的 CAS 值取决于你想要内存在什么样子的频率上运行。在本章节中主要就是告诉 CAS latency 值对于计算机系统究竟意味着什么以及为什么它们同总线速度有关。

CAS 延迟 (CAS latency) 指的就是在 READ 命令触发到第一次数据输出之间的时间, 这个时间的单位是以时钟周期。这个延迟时间可以设定为 1、2、3 个时钟周期。也就是说, 如果 READ 命令在第 n 个时钟上升沿被触发, 延迟时间为 m 个时钟周期, 那么数据将会在第 n + m 个时钟上升沿开始输出。下面的图表 (图片都是来自 Micron 的数据库中关于 MT48LC4M4A1 的资料) 可以帮助你更好的了解这个问题:

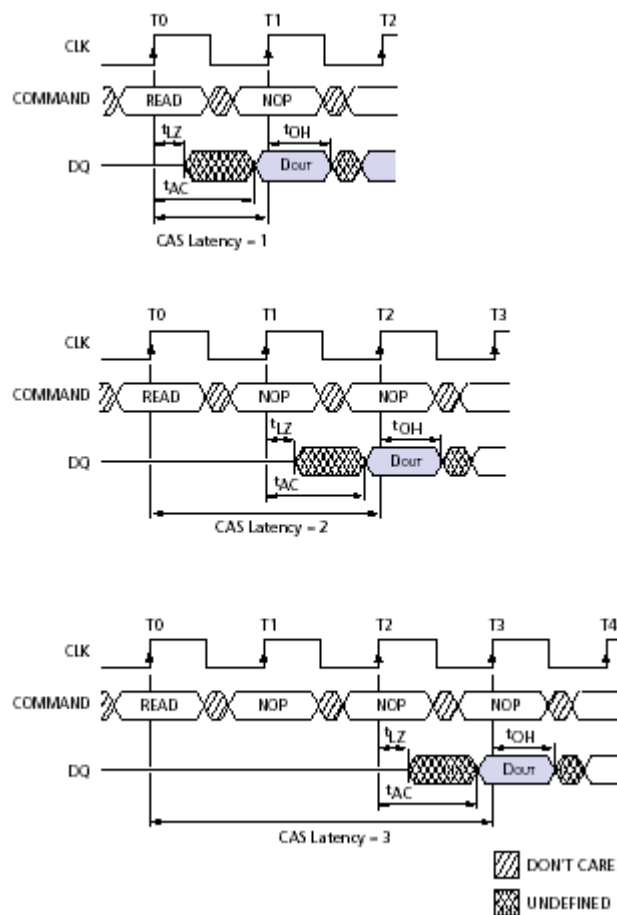


图 1

另外，DQ 会在第 $(n + m - 1)$ 个时钟周期之后的上升沿开始输出数据，也就是说数据将在第 $(n + m)$ 个时钟周期有效。比如说，CAS 延迟被设定为 2，那么如果 READ 命令在 T0 触发，DQ 将会在 T1 开始驱动，而所需要读取的数据在 T2 有效并且开始输出。上面的示意图显示了 CAS 延迟分别在 1、2、3 下的情况。

在上面的三个示意图中我们都可以看到 NOP (NO OPERATION : 空指令)。SDRAM 插入一个 NOP 空指令是因为 SDRAM 所有的操作都要同系统的基频同步, 大部分的操作都不是一个时钟周期所能完成的, 因此在等待或者空闲状态下为了防止其它的命令被触发, 所以

需要不停的发送 NOP 命令来抑制其他的指令，但是并不会干扰当前正在进行的操作。当然需要注意的是 NOP 指令的发送并不是随机的，而是根据用户的设定已经预设好的。

CAS Latency

SPEED	ALLOWABLE OPERATING FREQUENCY (MHz)		
	CAS LATENCY = 1	CAS LATENCY = 2	CAS LATENCY = 3
-6	≤ 50	≤ 125	≤ 166
-7	≤ 40	≤ 100	≤ 143
-8A	≤ 40	≤ 77	≤ 125

表 1

CAS 延迟和总线时钟之间的关系可以同下面的 CAL LATENCY 表格中看出，一般系统总线速度越快，CAS 延迟也会越长。这是因为目前的内存基本存储单元架构决定了其所能达到的最高运行频率，为了保证同目前总线时钟的同步，只能采用这样的方法了。

WRITE 过程分析

WRITE 突发过程是以 WRITE 指令的初始化开始的，在初始化过程中起始列和 bank 地址将会被确定。如果 AUTO PRECHARGE 有效，被访问的行将会在突发结束后进行预充电。而对于如图 2 一般的 WRITE 指令来说，AUTO PRECHARGE 指令是会被屏蔽的。

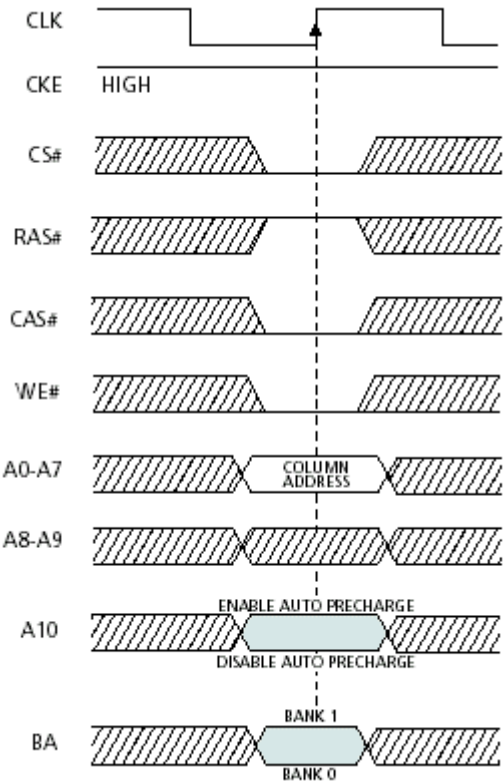
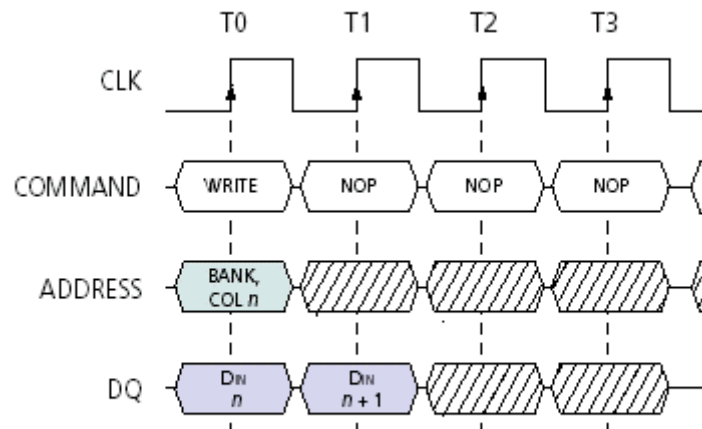


图 2、Write Command

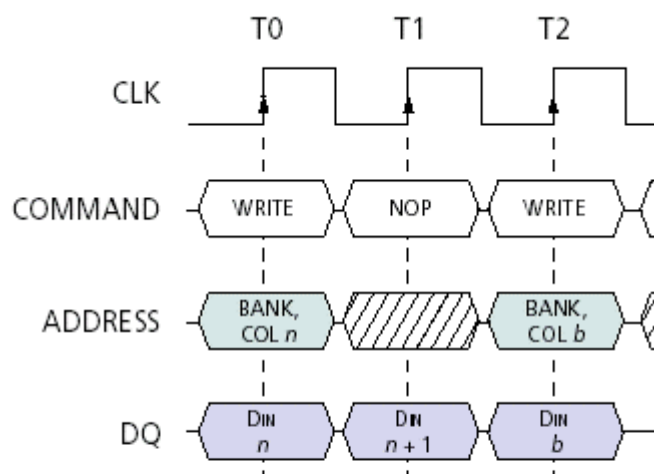
如图 2 所示，这是 WRITE 命令的写入时的状态，在这个时钟的上升沿，CS # 也进行相应的芯片选择，BA 已经确定了需要进行操作的 Bank，需要操作的行已经打开，直到下次读取操作或者 PRECHARGE 命令才会关闭该行，所以这个时候 RAS # 是高电平处于无效状态，因为需要进行列地址的选择，所以 CAS # 处于低电平，当然因为要进行写入操作 WE # 也是低电平状态。



NOTE: Burst length = 2. DQM is LOW.

图 3、Write Burst

在 WRITE 突发过程中，第一个有效数据在 WRITE 命令被触发的同时开始写入，如果还有连续的数据，它们会在随后的每个时钟周期的上升时间进行写入操作。在固定长度的突发操作中，假设没有其他的指令被初始化，DQ 将会一直保持 High-Z 状态，这个时候其它的输入数据将不会被执行（图上图 3），这个时候当然进入了全页突发的写入模式，在没有中止指令之前这种模式会一直持续而且会循环往复。这个时候如果有新的 WRITE 指令，那么原来的写入过程将会被中止。同我们前面了解的读取过程一样，在固定长度的 WRITE 突发模式中可以在任何之前的时钟周期中预先执行持续的 WRITE 命令。



NOTE: DQM is LOW. Each WRITE command may be to any bank.


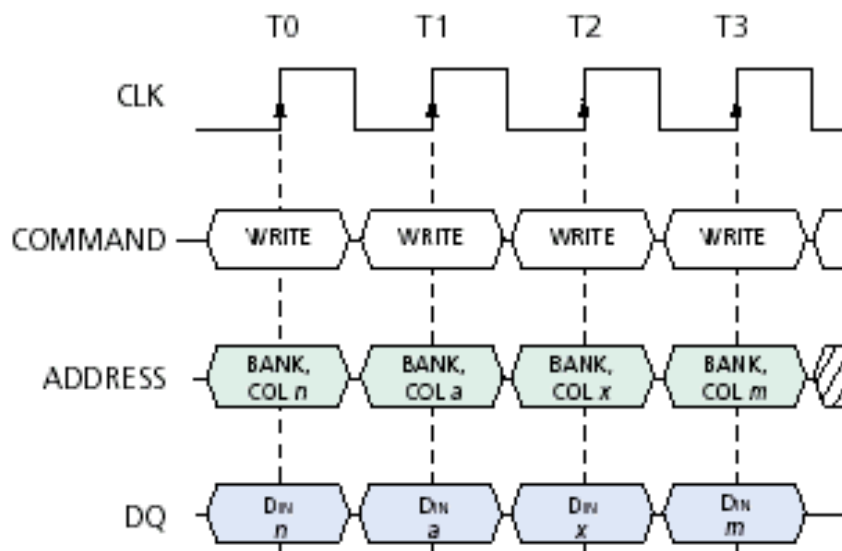
 DON'T CARE

图 4、WRITE to WRITE

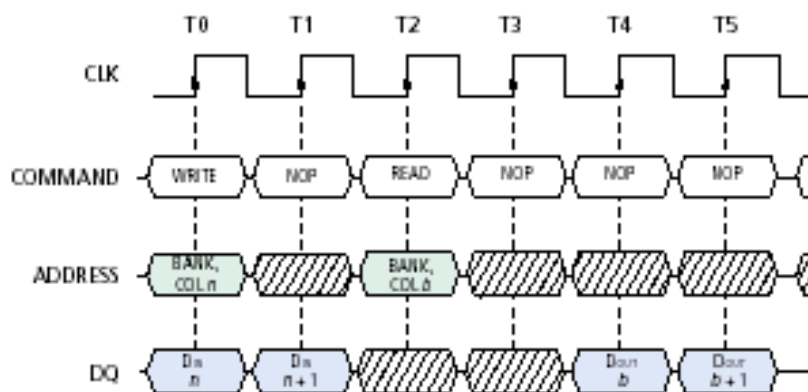
如上图 4，Data $n + 1$ 是上一次写操作的最后一个写入的数据。一个新的 WRITE 命令可以在以前 WRITE 命令过程中的任何一个时钟周期进行初始化。



NOTE: Each WRITE command may be to any bank.
DQM is LOW.

图 5 (随机写入)

在同一页中全速、随机写入的执行情况请参考上面的图 5。任何形式的 WRITE 突发模式都可以被持续的 READ 命令所中止,而在之前进行的 WRITE 命令操作所写入最后一个数据之后的时钟周期会马上执行持续的 READ 操作。



NOTE: The WRITE command may be to any bank, and the READ command may be to any bank. DQM is LOW. CAS latency = 2 for illustration.

图 6 (WRITE to READ)

如上图 6 所示, Data $n + 1$ 是 T0 开始的写入操作的最后一个写入数据。在这个数据之后,可以紧紧跟着固定长度的 WRITE 突发操作或者被 PRECHARGE 命令中止 - - 当然这样的操作也是在同一个 bank 之中的。关于 PRECHARGE 我们会在后面的示意图中讲述。这里我们看到在 Data $n + 1$ 后的第一个时钟周期 READ 命令开始执行,一旦 READ 命令被触发,数据输入将会被忽略,WRITE 命令将不会被执行。因为 CAS = 2,所以在 T4 时钟周期第一组数据开始输出了。

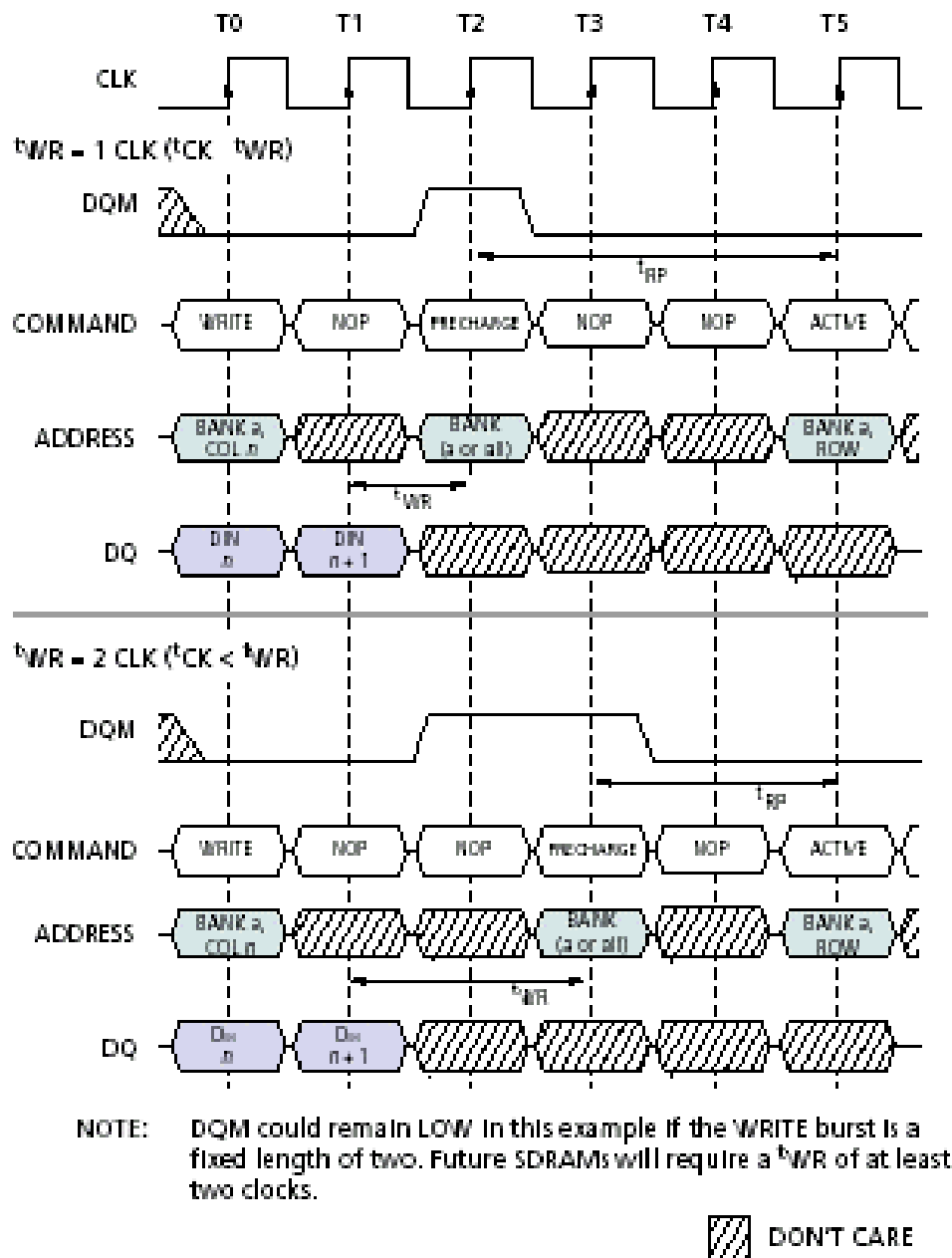
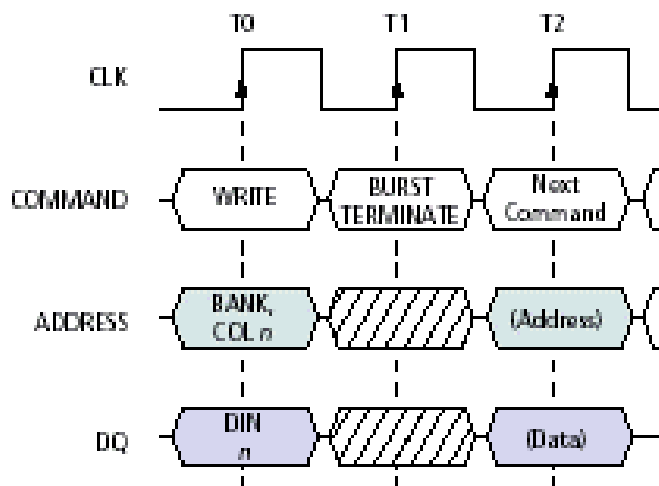


图 7 (WRITE to PRECHARGE)

在图 7 显示的是两种情况，第一是 $t_{WR} = 1\text{CLK}$ 的情况：在 T0 时钟 WRITE 命令执行，ADDRESS 所选择的列地址确定了需要写入数据的存储单元，也就在这个时钟周期数据开始写入。然后随后的每一个时钟周期都会进行数据的写入，在 $D_{in, n+1}$ 个数据写入之后的时钟周期，PRECHARGE 命令执行，这个时候已经打开的列地址被关闭进入预充电状态。在最后一个数据写入的时钟周期的上升沿到 PRECHARGE 命令开始执行的时钟周期的上升沿之间的时间间隔就是 t_{WR} 时间。你一定还记得从 PRECHARGE 命令开始执行的时钟周期的上升沿到下一个 ACTIVE 命令的执行的时钟周期的上升沿之间的时间间隔是 t_{RP} ，上图中显示占用了 3 个时钟周期，期间插入了两个 NOP 指令。这个时候就可以进行其他的操作了。

第二种情况也就是 $t_{WR} = 2\text{CLK}$ 的情况下，情况基本是一样的，只是 t_{WR} 时钟周期不同而已。另外我们看到，在 PRECHARGE 命令执行期间 DQM 信号是高电平。PRECHARGE 命令的缺点很明显，就是需要同时占用命令总线和地址总线，优点就是可以中止固定长度或者全页突发模式。



NOTE: DQMs are low

图 8

固定长度或者全页 WRITE 突发可以被 BURST TERMINATE 指令中止。当中止了一个 WRITE 突发操作，以后的输入数据将都会被忽略。如上图 8 所示，Din n 是前一个 WRITE 命令的最后一个写入的数据，在随后的一个时钟周期的上升沿 BURST TERMINATE 将会执行，该 WRITE 操作将会被中止，然后内存就可以接受其它的命令。

小节：

关于 SDRAM 内存的工作原理到这里我们基本上讨论完毕了。最后再对于 SDRAM 标识问题进行一下讨论。我们经常看到再很多场合这样的标识 SDRAM：X - Y - Z

比如说有的 SDRAM 标识为 3-2-2 这个含义同前面的文章介绍的异步 DRAM 的 x-y-y-y 是完全不同的：这三个数字依次表示为 CAS 延迟、RAS-to-CAS 延迟、RAS 预充电时间。这三项指标再加上内存的运行频率就可以比较详细的表明这款内存的各个方面的情况。说到内存运行频率，它是以 MHz 为衡量单位的 - - 之前的 DRAM 都是以纳秒为单位的。

SDRAM 我们就讨论到这里，以后我们会继续讨论其它的主流 RAM，比如 DDR、RDRAM 等等。（未完待续.....）

深入了解内存（七）

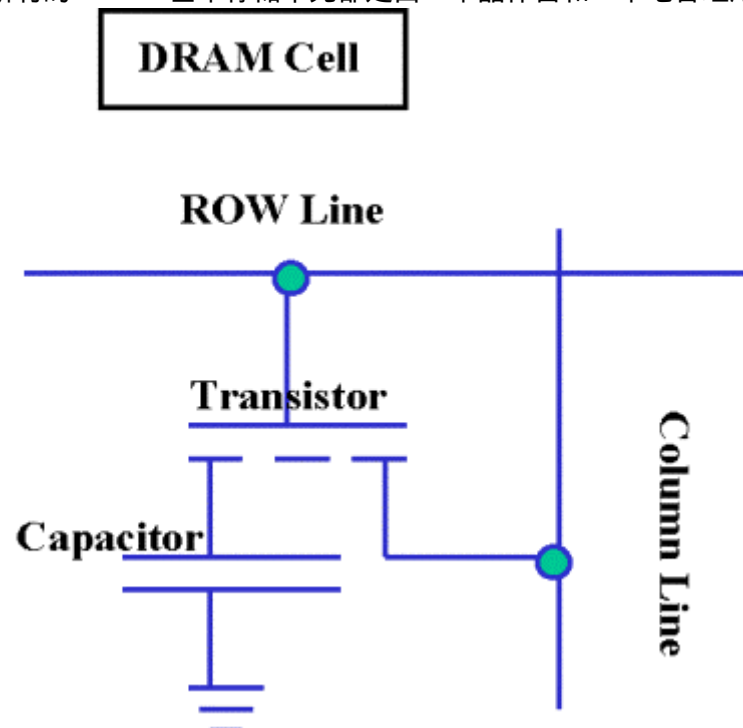
Aceshardware 所写的关于 SDRAM 基本工作原理的文章

在本站前一段时间推出了一系列的关于内存技术的文章，其中对于 DRAM 内存的基本原理都做了介绍，特别详细的介绍了 SDRAM 的工作情况。相信能够帮助读者们了解内存的基本工作原理。今天介绍给大家的这篇文章是 Aceshardware 所写的关于 SDRAM 基本工作原理的文章，这篇文章的同我们前面所写的文章可以说能够相互补充。

它首先介绍了 DRAM 基本存储单元的结构，然后结合芯片组来介绍了内存工作的时序，相对于原来比较微观的介绍，这篇文章的是从**整体的概念**上入手的。

DRAM 基本存储单元结构

不管你相信不相信，目前主流内存中的 RDRAM(Rambus)、DDR SDRAM、SDRAM 甚至是 EDO RAM 的基本结构都是相同的，它们都是属于 DRAM (Dynamic RAM：动态随机访问存储器)。所有的 DRAM 基本存储单元都是由一个晶体管和一个电容组成。



这样的基本存储单元的架构是目前最经济的方式，电容的状态决定着内存基本存储单元的逻辑状态是“0”还是“1” - - 充满电荷的电容器代表逻辑“1”，“空”的电容器代表逻辑“0”，不过正是因为使用了电容器所以产生了一些局限性。电容的内存储的电荷一般是会慢慢泄漏的，这也就是为什么内存需要不时的刷新的缘故。电容需要电流进行充电，而电流充电的过程也是需要一定时间的，一般是 0.2-0.18 微秒（由于内存工作环境所限制，不可能无限制的提高电流的强度），在这个充电的过程中内存是不能被访问的。

从技术上讲，实现内存的定时刷新并不是什么难事，DRAM 厂商指出这种刷新操作必须每 64ms 进行一次，这也就意味着 DRAM 基本存储单元大约有 1% 的时间用在了刷新上。对于 DRAM 来说最大的问题是，读取内存会造成内存基本存储单元中的电荷丢失，所以每当 DRAM 被访问之后都要进行刷新，以维持访问之前的状态，否则就会造成数据丢失。当然拿出专门的时间进行刷新，也就增加了访问时间，提高了延迟。

SRAM (Static RAM) 则不存在刷新的问题。一个 SRAM 基本存储单元由 4 个晶体管和两个电阻器构成，它并不利用电容器来存储数据，而是通过切换晶体管的状态来实现的，如同 CPU 中的晶体管通过切换不同的状态也能够分别代表 0 和 1 这两个状态。正是因为这种结构，所以 SRAM 的读取过程并不会造成 SRAM 内存储的的信息的丢失，当然也就不存在什么刷新的问题了。

SRAM 可以比 DRAM 高的频率来运行，主要是因为获取前 8 个字节的时间延迟大大缩

短了。SRAM 需要 2 - 3 个时钟周期来得到想要的数 据 (这里我们暂时忽略 CPU、芯片组和内存 DIMM 控制电路之间的延迟), 不过同样的过程 DRAM 需要大约 3 - 9 个时钟周期。当然因为构造不同, SRAM 和 DRAM 存储 1bit 数据的成本是不同的, 前者大约是后者的 4 倍 - - 因为它的所需要的晶体管数目是后者的 4 倍以上。SRAM 因为存取延迟时间非常的短, 所以它的工作频率能够达到很高, 因此可以带来更高的带宽。

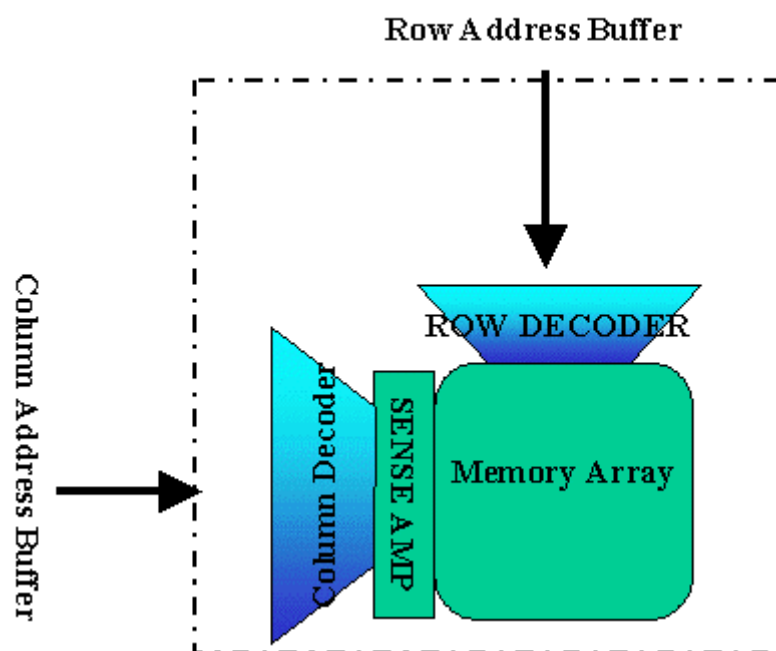
前面我们已经提到获取第一个字大约需要 3 - 9 个时钟周期的延迟时间。为什么? 在我们讨论不同的 DRAM 之间的性能差异之前, 我们应该首先了解 DRAM DIMM 的内部架构。

SDRAM 基本结构

基本存储单元是内存芯片中储存信息的最小的单位, 每个存储单元可以存储 1 个 bit 的信息, 并且有一个由行地址和列地址共同定义的唯一地址。我们都知道 8 个 bit 可以在一起组成一个 byte (这也就意味着 1byte 具有 2^8 种可能的数值), 而 byte 是内存中最小的可寻址的单元。虽然内存基本存储单元具有唯一的地址, 但是并不能进行独立的寻址——这 will 要求内存芯片有数以百计的引脚同计算机通讯, 显然这是不可能的。现在内存架构是处于同一列的基本存储单元共用一条列地址线, 而处于同一行的基本存储单元共用一条行地址线, 组成一个基本存储单元构成的矩阵架构。而这些矩阵架构构成一个内存 bank。

大多数的 SDRAM 芯片具有这样的 4 个 bank, 一个 SDRAM DIMM (Dual Inline Memory Module) 可以包括 8 或者 16 片芯片。SDRAM DIMM 具有 14 条地址线和 64bit 数据线。如果一条 DIMM 使用 8bit SDRAM 芯片, 那么在 DIMM 上可以发现 8 片芯片, 如果 DIMM 使用了 4bit SDRAM 芯片, 那么你可以在 DIMM 上找到 16 片这样的芯片。

Memory Bank



现在就让我们仔细的看看内存 bank, 上图所示的就是内存 bank 的示意图, 每个 bank 中包含一个内存阵列、传感放大器、行地址解码器和列地址解码器。为了理解内存 bank 内部工作情况, 让我们当 CPU 需要的数据没有调入处理器缓存, 处理器从内存中调用数据的过程:

- CPU 需要 32 byte 的数据, 然后将这个请求发送给芯片组。这个过程大约需要 1 个时钟周期。
- 芯片组首先通过 14 条地址总线发送行地址。这意味着这个行地址被发送到 DIMM 中的所有芯片上。所有的使用同一个行地址的存储单元组成了一个“页”, 也就是说, 当芯片组发送行地址到 DIMM 之后在 DIMM 打开了一个页。

- 每一个内存 bank 都有一个传感放大器,它用于在读取或者写入过程中改变电荷到适于操作的程度。根据发送过来的行地址,传感放大器可以读取正确的行。这个读取过程是需要花费时间的,这段时间就叫做“RAS 到 CAS 延迟”(T_{RCD}),SDRAM 质量不同,T_{RCD} 的值大约是 2 或者 3 个时钟周期。
- 当进行到这个时候,我们已经找到数据所在的行了,但是我们依然不知道数据存储在哪个基本存储单元。CAS 延迟内存找到正确的列的时间 - - 对于一个矩阵来说,知道了行地址和列地址之后我们就能够确定我们所需要的数据所在的位置。CAS 延迟一般是 2 或者 3。
- 被找到的数据被送到 DIMM 的输出缓存,这个时候芯片组就能读取它们了。
- 现在我们已经读取了第一个字(8 个字节)了,目前因为正确的行地址已经在传感放大器中了,那么得到下 24 个字节的过程就简单了。一个内部计数器可以直接把下一个相邻列地址所决定的基本存储单元中的数据传送到输出缓存中,每一个时钟都可以传送 8 个字节,这种传送数据的方式就叫做“突发模式”。

下面列出了整个内存系统(从 FSB 到 DRAM)各个阶段的延迟时间:

1. FSB 和芯片组之间的延迟时间大约是 ± 1 个时钟周期
2. 芯片组和 DRAM 之间的延迟时间是 ± 1 个时钟周期
3. RAS 到 CAS 延迟时间大约 2-3 时钟周期,这个期间找到了正确的行
4. CAS 延迟时间大约 2-3 时钟周期,这个期间找到了正确的列
5. 1 个时钟周期的时间用于传送数据
6. DRAM 输出缓存中的数据通过芯片组传送到处理器大约需要 ± 2 个时钟周期

这个过程只是得到了前 8 个字节的数据,优质的 PC100 SDRAM CAS 2 这个过程一般需要 9 个时钟周期,下 3 个时钟周期可以传送剩下的 24 个字节。PC100 SDRAM 传送这 32 个数据一共需要大约 12 个时钟周期。

如果你想要计算 CPU 所等待的延迟时间,只要用这个内存子系统延迟时间乘以 CPU 倍频。这样一个 500MHz (5x100MHz) 处理器将要有 5x9 个时钟周期。也就是说,当 CPU 不能在二级缓存中找到需要的数据的话,至少要等待 45 个时钟周期才能调用到正确的数据。本文到这里可以让你了解基本的 DRAM 工作情况了,下面让我们看看影响 RAM 技术速度的主要因素。

延迟时间

究竟是什么影响了 DRAM 的速度?因为 SDRAM 是一个多 bank 架构,当芯片组访问过某一 bank 的某一行之后,这一行处于一个“开”的状态。如果下一次访问请求还是同一个行,芯片组不必等待传感放大器充电,这样的情况就叫做页命中。这种情况下 RAS 到 CAS 延迟将成为 0 个时钟周期,只要经过 CAS 延迟时间之后数据就可以被送到输出缓存中了。换句话说,在页命中的情况下我们仅仅等待寻找正确的列的时间就可以了。

当然还有其它的可能性,就是所请求的行并不是已经打开的行,这种情况叫做页面失效(page miss)。在这种情况下,RAS 到 CAS 延迟将是 2 - 3 个时钟周期,这根据 SDRAM 内存的质量不同而不同。这种情况就是前面我们详细讨论的情况。

如果芯片组已经打开了某个 bank 中的某一行,而请求的数据在同一 bank 中的另外的行中,情况就会变得更糟了。这意味着传感放大器必须在选择新的行之前回写已经打开的旧的行。这个过程叫做预充电时间(Precharge time,简称 T_{TP}),这是所有情况中最糟糕的情况。

内存实际带宽

为了便于大家理解延迟和带宽之间的关系,我们以一个 PC100 SDRAM-222 内存为例来论述这个问题,这里的第一个 2 表示 CAS 延迟时间等于 2,第二个数字表示 RAS 到 CAS 延迟时间,第三个数据表示预充电时间。

请看下表,它所显示的是在各种情况的下的延迟时间的。其中第三列所示的 DRAM 延迟是得到第一个列地址所需要的时间。比如在页面失效的情况下,我们必须等到 2 个时钟周期的预充电时间 (Row to Cas Delay, RCD),然后继续等待 2 个时钟周期(也就是 CAS 延迟时间)这样就是能找到需要找的列地址了。而在表格中把 2 个时钟周期的预充电时间和 CAS 延迟时间合称为 DRAM 延迟时间。

在第四列我们看到时间都比前面的延迟时间多了 5 个时钟周期,这 5 个时钟周期中,前 2 两个时钟周期是地址数据从 CPU 经过芯片组传输到 DIMM 模组的时间,中间 1 个时钟周期是数据传输到输出缓存的时间,另外两个时钟周期是找到的数据经过芯片组传回到 CPU 的时间。

页面命中	+/- 55%	CL = 2	7	7-1-1-1= 10 cycles	320 MB/s
“正常”页面失效	+/- 40%	RCD+CL = 4	9	9-1-1-1 = 12 cycles	267 MB/s
页面失效,传感放大器但是已经打开的页面尚未关闭	+/- 5%	RP+RCD +CL = 6	11	11-1-1-1= 14 cycles	229 MB/s

倒数第二列显示的就是传送全部的 32 个字节所需要的时间,最后一列显示的就是在这种情况下内存所能够达到的最大带宽。我们看到即使在最理想的情况下,也就是在页面 100 %命中的情况下,实际所能达到的带宽也只有 320MB/s(对于 CAS = 2 的 PC100 SDRAM 来说,可以在 10 个时钟周期中得到 32 个字节的数据,也就是说需要 100 纳秒的时间得到 32 字节的数据,这样当然就能估算中实际带宽是 320MB/s - - 当然这样也是理想状态,平时我们所需要的数据不可能都是每 32 个字节连续的)。

从上面计算出来的数据我们可以看出,即使对于最好的 PC100 SDRAMs (222)内存,在最理想的状态下(页面命中率 100%的情况下),实际所能达到的带宽也不过是理论带宽(800 MB/s) 的 40%。对于 Pentium III 缓存所使用的 SRAM 传送 32 个字节所需要的时间延迟为 3-1-1-1,也就是 6 个时钟周期(注意它的频率可以达到 300MB/s 以上),这样它的带宽至少是 1600MB/s。那么对于我们现在所普遍使用的 PC133 的情况是怎么样的呢?请看下表所示的是 PC133 CAS2、PC133 Cas3、PC100 CAS2 的情况:

DRAM	总体延迟时间	传输 32 字节所需延迟	最大带宽	相对于 PC 100 的性能增长
PC 133-CAS 2	+/- 7 cycles	7-1-1-1= 10 cycles	427 MB/s	33%
PC 133-CAS 3	+/- 8 cycles	8-1-1-1 = 11 cycles	387 MB/s	21%
PC 100-CAS 2	+/- 7 cycles	7-1-1-1= 10 cycles	320 MB/s	N/A

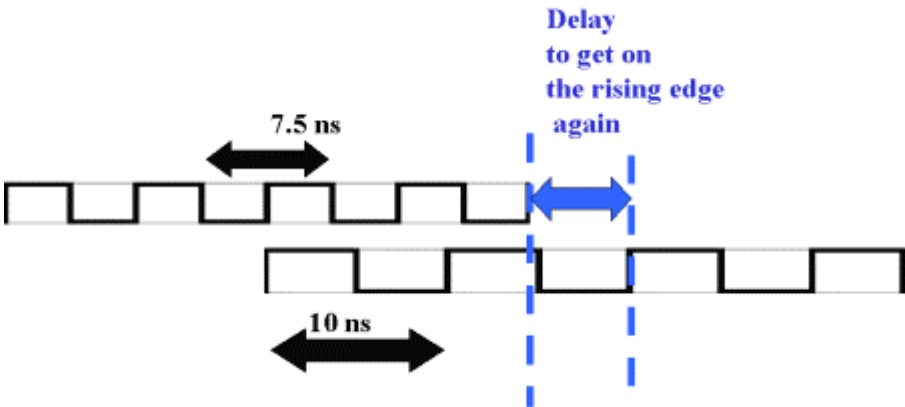
这里我们看到得到前 8 个字节的数据,PC133 CAS3 需要 8 个时钟周期而 PC100 CAS2 却仅仅需要 7 个时钟周期,如果这样就觉得 PC100 CAS2 的性能就比 PC133 CAS3 好了,应该说不全面的。对于系统总线分别为 100MHz 和 133MHz 的系统,如果分别使用 PC100 CAS2 和 PC133 CAS3,虽然后者得到前 8 个字节所需的时间比前者多 1 个时钟周期,但是考虑到系统总线速度依然比较快,所以计算起来依然是 PC133 系统快一些,但是对于都是 PC100 的系统,即使使用 PC133 CAS3 的内存条,它的性能也不可能超过 PC100 CAS2 的内存条 - - 不过好在一般

的 PC133 CAS3 内存都能在 PC100 CAS2 上稳定的运行，所以如果你遇到这个情况应该知道怎么解决了。

异步和同步芯片组的相关问题

对于这些问题有了基本的概念之后，我们一起讨论 BX 时代的芯片组的相关问题，现在很用用户一定还在使用 BX 芯片组主板或者 VIA Apollo Pro133A、AMD750 等芯片组。我们知道 VIA Apollo Pro133A 可以支持 133 外频，但是它的内存子系统的性能相对于超频到 133MHz 是略低的。为什么会这样，这就涉及到一个芯片组采用的异步内存或者同步的内存的问题。BX/AMD750 芯片组是同 FSB 时钟频率一致的，而 VIA Apollo Pro133A 芯片组的采用的是异步解决方案 - - 因为它的可以支持 PC66、PC100 和 PC133 SDRAM 芯片组，所以必须采用这种解决方案。

现在让我们比较以下三种系统的情况：
超频到 133MHz 的 BX 芯片组、使用异步内存的 VIA Apollo Pro133A 芯片组和运行在 100MHz 额定频率下的芯片组 - - 每种芯片组都采用了相应频率的 SDRAM 内存。如果内存运行的频率不同于系统的总线频率，那么至少一个时钟周期用于同步。因为 133 MHz 总线的时钟周期是 7.5 纳秒，而 100 MHz 总线的时钟周期是 10 纳秒，如下图所示，在某一个时间两种频率的波同时达到上升沿，当 7.5 纳秒的的某个时钟周期结束的时候，10 纳秒的时钟周期才进行到某一个地方。如果让这两个不同的频率的波同时达到上升沿至少需要一个时钟周期的延迟。



下面让我们看看各种情况下的延迟情况：

DRAM	芯片组	前 8 字节延迟时间	得到 32 字节所需延迟时间	最大带宽
PC 133 CAS2	BX Sync 同步	+/- cycles	7 7-1-1-1= 10 cycles	32 byte/(10 cycles of 7.5 ns) =427 MB/s
PC 133-CAS2	VIA 133 MHz 异步	+/- cycles	8 8-1-1-1= 11 cycles	32 byte/(11 cycles of 7.5 ns) =387 MB/s
PC 133-CAS3	VIA 133 MHz 异步	+/- cycles	9 9-1-1-1 =12 cycles	32 byte/(12 cycles of 7.5 ns) =355 MB/s
PC 100-CAS2	BX/VIA 同步	+/- cycles	7 7-1-1-1= 10 cycles	32 byte/(10 cycles of 10 ns) =320 MB/s

一个 133 MHz 异步内存系统得到前 8 个字节的延迟时间高于同步内存子系统大约 15%，这也就意味着它的带宽比同步内存子系统低大约 10%。上面我们仅仅假设异步内存系统之间需要一个时钟周期来进行同步，如果需要更多的时钟周期来同步，那么异步内存系统的实际带宽将会更低。

结论

从以上的内容我们可以看到，DRAM 内存为了保持其内的内容需要经常的刷新，这种特性限制了这种内存不可能达到太高的工作频率，在实际工作中得到需要的数据也是必须经过一定的延迟时间才能够得到的。一般的 SDRAM 的带宽在 800 - 1066 MB/s 之间，但是实际的带宽也不过是它的理论带宽的 40% 左右（这是在 100% 命中的情况下的结果）。

在今天文章所介绍的东西涉及到 BX 这个基本的芯片组，这些产品正在快速的退出市场，让位给我们现在主流使用的东西。但是对于我进一步了解内存的工作原理依然有相当的帮助。不过目前的 Athlon、Ahtlon XP 系统配置的都是 DDR SDRAM 内存，而 Pentium 4 也从 RDRAM 逐渐的过渡到了 DDR SDRAM，所以下一篇文章我们依然从这样的角度来了解 RDRAM 和 DDR SDRAM 的工作原理和特点。

collected by stonefeng