果壳 Cache 验证报告

饶嘉怡, 岳俊宇, 杨泽辰

May, 2024

版本号： *v0.1*

目录

1 验证对象

1.1 Cache 简介

现代处理器使用硅工艺制造，在摩尔定律的驱使下，处理器的速度越来越快，但相比之下存储器的发展就慢了很多。在单核条件下，1980-2015 的 30 年间，CPU 性能提升了近 10000 倍，但是内存性能仅提升了 10 倍，不管是硬盘还是固态硬盘，他们的速度在当今的处理器面前都是慢了一个或几个数量级的，内存和 cpu 性能的差距越来越大，形成 CPU 与内存速度的“剪刀差”，这也导致了如何给 CPU 提供稳定的指令和数据是一个永不过时的话题。

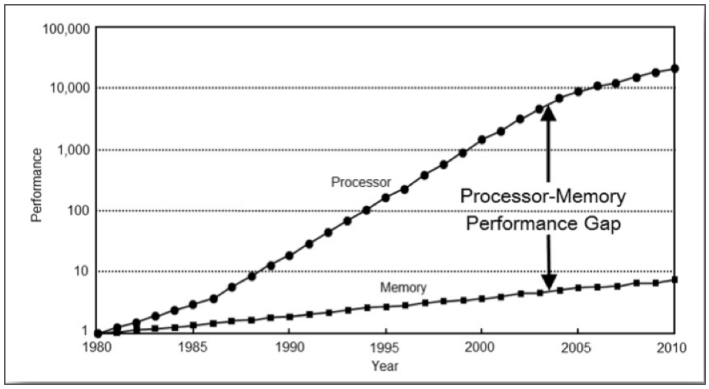


图 1: CPU 与内存之间速度的“剪刀差”

1.1.1 局部性

缓存（Cache）是存在于主存与 CPU 之间的存储器。Cache 之所以存在，是因为在计算机世界中，存在如下的两个现象：

（1）时间相关性（temporal locality）：如果一个数据现在被访问了，那么在以后很有可能还会被访问。

（2）空间相关性（spatial locality）：如果下一个数据现在被访问了，那么它周围的数据在以后有可能也会被访问。

现代处理器直接访问一次主存的时间内能够执行上千条指令，而 Cache 利用程序的局部性，缓存内存的一部分数据，这样 CPU 在访存时先访问速度更快的 Cache，能够显著降低 CPU 访问内存的开销。

1.1.2 Cache 的层次结构

为了进一步降低访存延迟，现代处理器通常采用多层缓存的结构，离处理器最近的为一级缓存（L1 Cache），次近的为（L2 Cache），依次类推。L1 Cache、L2 Cache 和处理器联系最紧密，使用和处理器一样的硅工艺制造，其中 L1 Cache 紧密地耦合在处理器的流水线中，是影响处理器性能的一个关键因素。

现代的超标量处理器都是哈佛结构，为了增加流水线效率，L1 Cache 一般都包括指令 Cache（I-Cache）和数据 Cache（D-Cache），I-Cache 只会被读取不会被写入，D-Cache 不仅需要读取，还需要考虑写入的问题，因此 D-Cache 更复杂一些。L1 Cache 一般都是使用 SRAM 来实现的，容量更大的 SRAM 需要更长的时间来找到一个指定地址的内容，而 L1 Cache 最靠近处理器，它是流水线的一部分，需要和处理器保持近似相等的速度，这决定了它的容量不会很大。

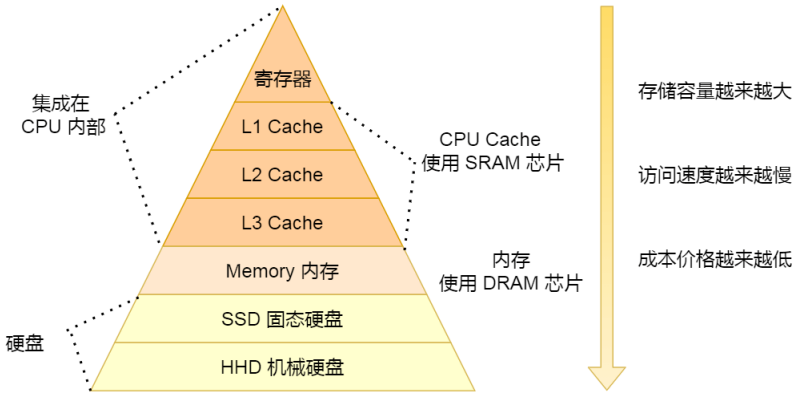


图 2: 金字塔存储层次

1.1.3 Cache 的组成方式

Cache 有三种主要的组成方式，直接映射（direct-mapped）Cache，组相连（set-associative）Cache 和全相连（fully-associative）Cache。对于物理内存（physical memory）中的一个数据来说，如果在 Cache 中只有一个地方可以容纳它，它就是直接映射的 Cache；如果 Cache 中有多个地方都可以放置这个数据，它就是组相连 Cache；如果 Cache 中的任何地方都可以放置这个数据，那么它就是全相连 Cache。直接映射和全相连这两种结构的 Cache 实际上是组相连 Cache 的两种特殊情况，现代处理器中的 Cache 一般属于上述三种方式中的某一个，例如 TLB 和 Victim Cache 多采用全相连结构，而普通的 I-Cache 和 D-Cache 则采用组相连结构等。

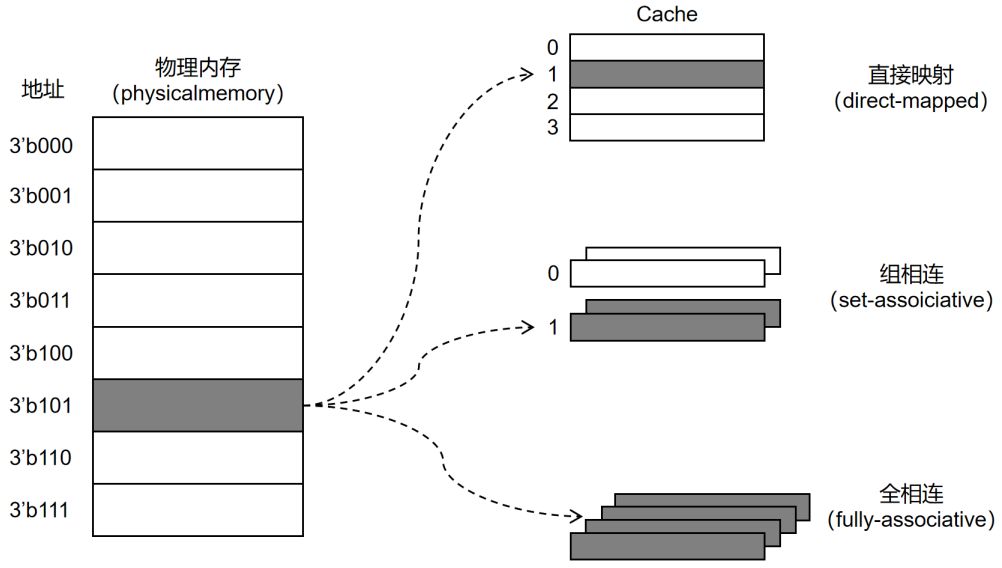


图 3: Cache 的三种组成方式

1.1.4 Cache 的写入

当执行一条写指令时，如果只是向 D-Cache 中写入数据，而并不改变它的下级存储器中的数据，这样就会导致 D-Cache 和下级存储器中，对于这一个地址有着不同的数据，即不一致（non-consistent）。要想保持一致性，一般 Cache 在写命中状态下有两种写入方式：

（1）写通（Write Through）：数据在写到 D-Cache 的同时，也写到它的下级存储器中。由于 D-Cache 的下级存储器需要的访问时间是比较长的，而 store 指令在程序中出现的频率又比较高，如果每次执行写指令时，都向这样的慢速存储器中写入数据，处理器的执行效率肯定不会很高了。

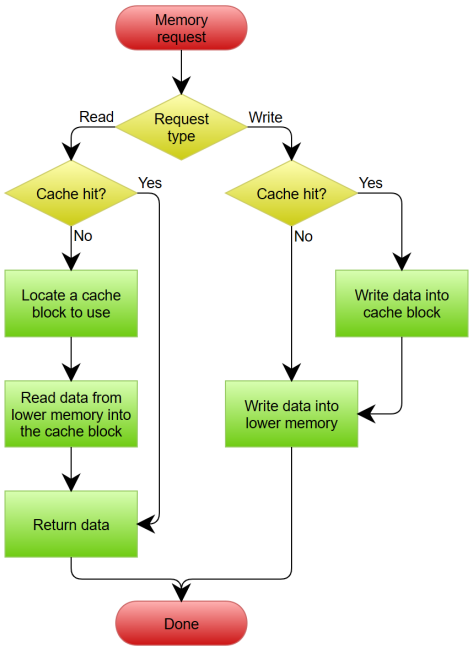
（2）写回（Write Back）：数据被写到 D-Cache 后，只是将被写入的 Cache line 做一个记号，并不将这个数据写到更下级的存储器中，只有当 Cache 中这个被标记的 line 要被替换式，才将它写到下级存储器中。这种方式能减少写慢速存储器的频率，从而获得更好的性能，但这种方式会造成 D-Cache 和下级存储器中有很多地址中的数据是不一致的，这会给存储器的一致性管理带来一定的负担

D-Cache 处理写缺失一般有两种策略：

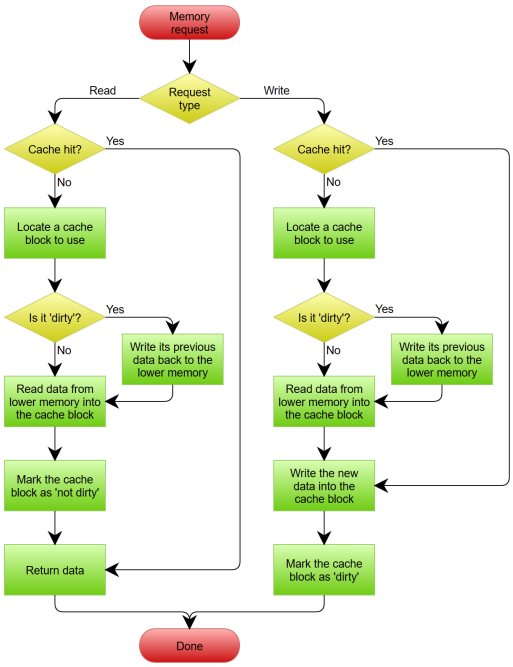
（1）非写分配（Non-Write Allocate）：将数据直接写到下级存储器中，而并不写到 D-Cache 中。

（2）写分配（Write Allocate）：如果写 Cache 时发生了缺失，会首先从下级存储器中将这个发生缺失的地址对应的整个数据块取出来，将要写入到 D-Cache 中的数据合并到这个数据块中，然后将这个数据块写到 D-Cache 中。

通过上面的描述可以看出，写通（Write Through）和非写分配（Non-Write Allocate）都会将数据直接写到下级存储器中，而写回（Write Back）和写分配（Write Allocate）都不会对下级存储器直接进行写操作，因此通常 D-Cache 的写策略搭配为：写通（Write Through）+ 非写分配（Non-Write Allocate），写回（Write Back）+ 写分配（Write Allocate），如图4。



(a) Write Through 和 Non-Write Allocation 工作流程图



(b) Write Back 和 Write Allocation 工作流程图

图 4: Cache 工作流程图[1]

[1]图源 wiki · cache

1.1.5 Cache 的读出

当执行一条读指令时，如果命中，可以直接读回；如果缺失，则需要从下级存储器中读出所需的数据块，重填进 Cache，再进行读取。

1.1.6 Cache 的替换策略

读写 D-Cache 发生缺失时，需要从对应的 Cache Set 中找到一个 cache 行，来存放从下级存储器中读出的数据，如果此时这个 Cache Set 内的所有 Cache 行都已经被占用了，那么就需要替换掉其中一个，如何从这些有效的 Cache 行找到一个并替换它，这就是替换策略，本节介绍几种最常用的替换策略。

近期最少使用法会选择最近被使用次数最少的 Cache 行，因此这个算法需要追踪每个 Cache 行的使用情况，这需要为每个 Cache 行都设置一个年龄（age）部分，每当一个 Cache 行被访问时，它对应的年龄部分就会增加，或者减少其他 Cache 行的年龄值，这样当进行替换时，年龄值最小的那个 Cache 行就是被使用次数最少的了，会选择它进行替换。

随机替换算法硬件实现简单，这种方法发生缺失的频率会更高一些，但是随着 Cache 容量的增大，这个差距是越来越小的。在实际的设计中，很难实现严格的随机，一般采用一种称为时钟算法（clock algorithm）的方法实现近似的随机，它的工作原理本质上是一个时钟计数器，计数器的宽度由 Cache 的路的个数决定，当要替换时，就根据这个计数器选择相应的行进行替换。这种方法硬件复杂度较低，也不会损失较多的性能，因此是一种折中的方法。

1.2 果壳 Cache 简介

果壳（NutShell）是一款由 5 位中国科学院大学本科生设计的基于 RISC-V RV64 开放指令集的顺序单发射处理器 (NutShell · Github), 隶属于国科大与计算所“一生一芯”项目。

果壳 Cache（NutShell Cache）是其缓存模块，采用可定制化设计（L1 Cache 和 L2 Cache 采用相同的模板生成，只需要调整参数），具体来说，L1 Cache（指令 Cache 和数据 Cache）大小为 32KB，L2 Cache 大小为 128KB, 在整体结构上，果壳 Cache 采用三级流水的结构，这个 Cache 模块的结构图如下所示：

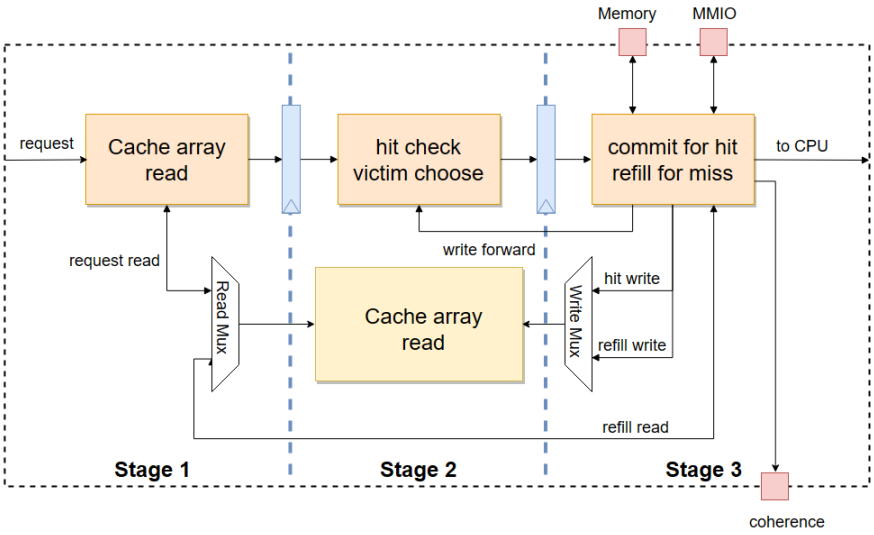


图 5: Cache 模块结构图[2]

[2]图源 NutShell · GitBook

在本验证实例中，验证对象为**果壳 L1 Cache**，不支持一致性（coherence）[3]请求。

NutShell Cache 映射方式采用的是四路组相联映射,Cache 的替换策略为随机替换, 并采取写回的策略。每个 Cache 行的大小为 64B。顺序核 Cache 目前采用实地址作为标签 (tag), 实地址作为索引 (index)。因此访问 Cache 的地址为经过 TLB(Translation Lookaside Buffer) 进行虚实地址转化后的 32 位实地址。32 位地址格式（物理地址）:

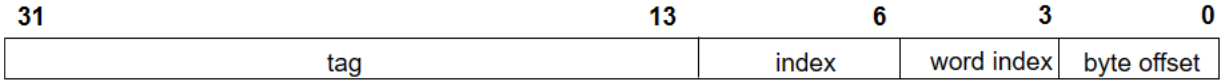


图 6: Cache 物理地址划分

NutShell 使用内存映射 I/O（Memory-mapped I/O, MMIO）的方式来访问外设。MMIO 是一种访问外设的方式，它将外设的寄存器映射到内存地址空间中，通过读写内存地址来访问外设。在 NutShell 中，MMIO 地址空间为 32'h30000000 至 32'h7fffffff。

果壳 Cache 的顶层接口以及说明如下:

表 1: 果壳 Cache 的顶层接口及说明

信号	说明
clock reset	时钟 复位信号
io_flush io_empty io_in_*	 请求总线信号 (req & resp)
io_out_mem_*	cache 向内存请求的总线信号
io_mmio_*	cache 向 MMIO 请求的总线信号
io_out_coh_*	一致性相关的信号
victim_way_mask	受害者相关信号，即被替换的 cache 块相关信息

上下游通信总线采用 SimpleBus 总线，包含了 req 和 resp 两个通路，其中 req 通路的 cmd 信号表明请求的操作类型，可以通过检查该信号获得访问类型。SimpleBus 总线共有七种操作类型，由于 NutShell 文档未涉及 probe 和 prefetch 操作，在验证中只出现五种操作：read、write、readBurst、writeBurst、writeLast，前两种为字读写，后三种为 Burst 读写，即一次可以操作多个字。

[3]Cache 一致性：https://zhuanlan.zhihu.com/p/115114220

2 验证方案与验证框架

2.1 验证方案

传统的验证往往基于 System Verilog 编程语言进行，但是该语言用户集数少，并且缺乏高效的库工具来进行验证工作。Picker 工具支持将模块的 RTL 代码转换成 python 等语言的库，通过 Picker 工具的桥接我们可以在保留硬件特征的前提下，使用 python 中的 pytest、协程等软件工具便捷快速地进行验证工作。此外，Picker 工具同时支持导出 RTL 行覆盖率数据，亦方便评估验证结果。

综合以上，我们得到以下的验证方案：

1. 首先使用mcv-ut 工具拆解 NutShell 中的 Cache，得到其 RTL 源码
2. 再使用 Picker 工具将 Nutshell Cache 的 RTL 代码转换成 python 形式的待测设计（Design Under Test, DUT）
3. 以软件的形式驱动其顶层接口，观察特定接口行为和结果反馈。

2.2 验证框架

通过 Picker 工具生成 DUT 的封装之后，我们还需要进一步构造验证环境。为了满足 cache 的 memory 和 mmio 请求，需要创建对应的外围设备模拟，同时为了方便后续测试用例中对 cache 的读写操作，还需要对 DUT 的顶层做进一步抽象；由于封装的 cache 是黑盒，较难探知 cacheline 等内部信息，于是我们需要创建参考模型来模拟部分 cache 的功能；除此之外，为了验证功能点是否被触发以及是否被正确实现，我们需要将信号汇总，交由一个模块来综合判断。

综上，验证框架可以分为 5 个部分：**激励生成（PyTest）、外围设备（CacheWrapper& Memory & MMIO）、参照模型（Ref Cache）、功能覆盖统计（PyTest-Cov）**以及 **DUT**。

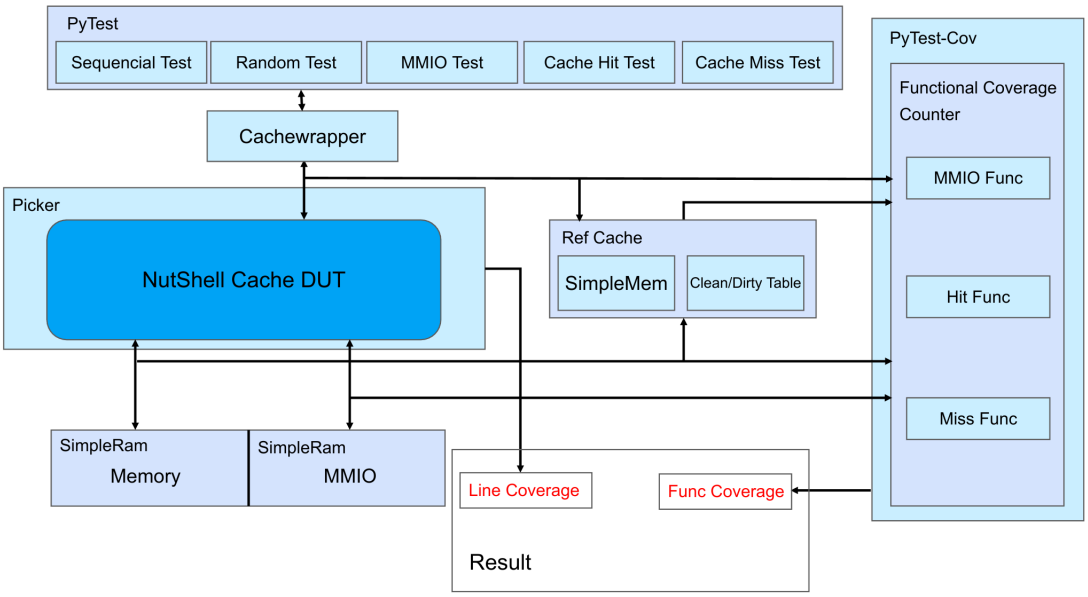


图 7: 验证框架

2.2.1 激励生成模块

该部分使用 PyTest 工具自动运行。包括 5 个测试用例：Sequencial Test（顺序读写测试）、Random Test（随机读写测试）、MMIO Test（MMIO 读写测试）、Cache Hit Test（Cache 命中测试）以及 Cache Miss Test（Cache 缺失测试）。

对于每个测试用例，其会根据测试目标向 CacheWrapper 发出相应的命令，产生相应的激励信号驱动 Cache DUT。笔者将在**测试用例**一节进一步阐述。

2.2.2 外围设备模块

外围设备则是对 Cache 运行需要的外部环境的模拟。

CacheWrapper 是对 Cache 顶层信号的封装。其维护了两个队列：请求队列（req_queue）和回复队列（resp_queue）。在时钟上升沿到来时，通过回调函数扫描请求队列情况，若存在未处理的请求则在 DUT 的请求端置起相应信号发送请求；同时，若 DUT 的回复端有新的回复信号，则将其记录到回复队列中。通过 CacheWrapper 可以简单方便地向 Cache 发起读写请求。

Memory 是对内存区域的模拟，其可以响应 Cache 发出的普通内存访存请求。需要注意的是该模型为理想模型，并未模拟延迟的情形。

MMIO 则是对 MMIO 区域的模拟，可以响应 Cache 发出的 MMIO 访存请求。该模型同样没有延迟。

2.2.3 参照模型模块

参照模型用于比对 Cache 的行为是否正确。其主要包括两个部分：**SimpleMem** 用于模拟不使用 Cache 时的内存和 MMIO 区域情况，**Clean/Dirty Table**（下简称 C/D Table）则记录了每个块是否为脏块以及是否在 cache 中。

在时钟上升沿到来时，Ref Cache 通过回调函数侦听请求总线上的情况，并对 SimpleMem 进行一致的读写操作。例如，若通过 Cache 向某个地址写入了某个数据，则需要对 SimpleMem 中的地址进行相应的操作。此外，若该请求为写操作，Ref Cache 则会在 C/D Table 中将相应的 cache 块标记成 “dirty”。同时，Ref Cache 也会侦听内存总线上的情况，若出现内存写请求（这意味这有 cache 块被替换），则需要将被替换的块标记成 “out”。

2.2.4 功能覆盖统计模块

该模块主要作用是正确性检查以及功能点检测。

当时钟上升沿到来时，功能覆盖统计模块通过回调函数侦听各总线上的情况。若此是通过请求总线向 Cache 发起了读请求，则需要进行正确性检验，即判断 Cache 回复的读数据是否与 Ref Cache 中的数据一致。同时，该模块回根据各总线上的数据情况以及 Ref Cache 中的 C/D Table 判断某个功能点是否被覆盖，具体见**测试用例**部分。

同时，为了建模 Cache 的时序方面的特征（例如三级流水），笔者维护了一个带时间戳的队列（msg_queue），当请求总线上有请求时，将相应请求打上时间戳（当前的周期数）并从右侧入队；若 Cache 回复了一个请求，则将请求从队列的左侧出队。这样，当出队一个请求时，则可以通过其时间戳以及当前时间判断其经历的周期数，进而判断时序相关的功能点是否被覆盖。

2.2.5 DUT 模块

Design Under Test。该部分为 Picker 工具生成的待测模块。

2.3 框架模块交互

在这一小节我们将简单介绍验证框架中的模块如何交互。下表展示了某次读操作中各模块的工作，假设该次读操作未能命中：

表 2: 交互流程

顺序	激励生成	外围设备	DUT	参照模型	功能覆盖统计
1	读 0x1000	cachewrapper: 封装该请求并转发			
2			接受请求	监听请求总线，读操作时无动作	监听请求总线，将请求加入 msg_queue
3			Memory 端口发出读 cacheline 请求	监听内存请求总线，侦听到读请求，在 CD 表中对应位置标记 [in, clean]	监听内存请求总线，侦听到读请求并标记
4		Memory: 响应 Memory 端口读请求，返回 cacheline 数据			
5			回复请求	监听请求总线，侦听到回复了一个读请求，无动作	监听请求总线，侦听到回复读请求，将 msg_queu 队首出队，并且检查 DUT 回复的数据是否等于 SimpleMem[0x1000]
6		cachewrapper: 封装回复			
7	收到回复，读完成				

3 功能点和测试点

如前所述，Cache 的功能是降低访存的时间开销，其功能本质上和内存是一致的。也就是说，不论是向 Cache 存数还是取数，其都应该和直接向内存存取的数是一样的。因此，Cache 的基础读写功能将成为我们的第一个功能点。

进一步，访问 Cache 的地址空间分为 MMIO 和内存。其中，访问 MMIO 的地址空间时，Cache 一定会 Miss，然后将请求转发到 MMIO 端口上。而访问内存的地址空间时，Cache 则会根据该地址所在的 Cache Line 是否在 Cache 中而触发 Hit 或者 Miss。Hit 则直接返回响应，Miss 则会将请求转发到内存端口。如果被替换的受害者行之前被写过，是 dirty 的，则要先将受害者行写回（write-back）内存，否则直接从内存加载缺失的 Cache Line，重填（refill）回 Cache。也就是说，Cache 的行为可以大致分为下面几种：

访问 MMIO 地址空间：
　　Miss
访问内存地址空间：
　　Hit（命中）
　　Miss → **Refill**（缺失）
　　Miss → **Write-back** → **Refill**（缺失）

因此，我们可以继续将 MMIO 地址空间的访问、内存地址空间访问命中、内存地址空间访问缺失，作为接下来的三个功能点。

并且，我们可以根据每个功能点的行为，继续将功能点拆分为测试点，如下表所示。测试点的拆分思路将在下文解释。

表 3: 测试点及解释

功能点	测试点	简述
基础功能	1	Cache 基础读写功能
MMIO	2.1.1	MMIO 读写请求会被转发
	2.1.2	Cache 响应 MMIO 时不会发 Burst 请求
	2.2.1	MMIO 请求会阻塞流水线
Cache 命中	3.1.1	Cache 采用写回策略
	3.2.1	Cache 命中时周期数少
Cache 缺失	4.1.1	Cache 缺失阻塞流水线
	4.1.2	重填时关键字优先
	4.1.3	Cache 缺失时周期数多
	4.2.1	替换脏块时先写回
	4.3.1	替换干净块不写回

3.1 功能点 1: 基础功能

首先，Cache 从本质上来说是内存的备份，这意味着访问 Cache 得到的结果和没有 Cache 直接访问内存得到的结果是一致的。由此可以得到第一个测试点，这个测试点可以在每一个测试用力中被覆盖到：

测试点 1：Cache 是内存的备份
测试用例：**测试用例 1**, **测试用例 2**, **测试用例 3**, **测试用例 4**, **测试用例 5**,

3.2 功能点 2: MMIO

3.2.1 MMIO 读写

Cache 会根据地址所在的区间，判断是否发生 MMIO 请求。如果发生 MMIO 请求则会将请求转发到 MMIO 的端口上，而不会发生 Cache 行的读写。这是一个测试点：

测试点 2.1.1：对 MMIO 地址的请求会被转发到 MMIO 端口上
测试用例：**测试用例 3**

此外，MMIO 请求不是 Burst 请求，每次只会写入或读出一个地址的数据，而不是一个 Cache 行的数据。因此，在 MMIO 端口上不应当观测到 Burst 的请求类型，这也作为一个测试点：

测试点 2.1.2：Cache 响应 MMIO 时不会发出 Burst 请求
测试用例：**测试用例 3**

3.2.2 MMIO 阻塞

NutShell 手册指出，在检测出 MMIO 请求后会阻塞流水线。

测试点 2.2.1：Cache 检出 MMIO 请求时会阻塞流水线
测试用例：**测试用例 3**

3.3 功能点 3: Cache 命中

3.3.1 Cache 写命中

NutShell 的 Cache 采用写回策略，即若发生写请求，直接在 cache 块中对应的位置做修改，同时标记该块为脏块；当发生行替换时，若被替换的为脏块，再将其写到内存中。Cache 是否真的采用了写回的策略也是一个测试点：

测试点 3.1.1：写回策略
测试用例：**测试用例 1**

3.3.2 Cache 命中时序

由于 Cache 命中时直接从 Cache 中得到数据，不需要写回或重填，因此收到回复所需要的周期数更少。

测试点 3.2.1：Cache Hit 时周期数更短
测试用例：**测试用例 1**

3.4 功能点 4: Cache 缺失

为了创造 Cache Miss 的测试环境，**首先需要通过一系列的 Load 操作先将 Cache 填满**。后续需要触发 Cache Miss 时，只需要访问上述 Load 覆盖范围之外的地址即可。

3.4.1 Cache 缺失

首先，发生 Cache Miss 时也会阻塞流水线。

测试点 4.1.1：Cache Miss 时阻塞流水线
测试用例：**测试用例 2**

其次，NutShell Cache 重填时采用**关键字优先方案**，因此 Cache 向内存请求数据时，发出的首个地址应当是向 Cache 发出请求时的地址。例如，假设向 Cache 发出 0x1000 地址的读请求，此时发生 Cache Miss，Cache 会向内存发出读请求，这个请求的首地址应当是 0x1000。

测试点 4.1.2：关键字优先重填
测试用例：**测试用例 2**

当然 Cache Miss 时，回复所需要的周期数会更长。

测试点 4.1.3：Cache Miss 时周期数更长
测试用例：**测试用例 2**

3.4.2 Cache 替换脏块

当需要替换的 Cache 块是脏块时，首先会进行写回的操作。当然，首先需要创建脏块的环境，由于 NutShell Cache 采用随机替换的策略，因此我们考虑将整个 Cache 都设置成脏块。操作也是简单的，在上述的 Load 的基础上，只需要在每个 CacheLine 的起始地址进行一次 Store 操作即可。

测试点 4.2.1：在替换块为脏块的时候，首先写回该脏块。
测试用例：**测试用例 2**

3.4.3 Cache 替换干净块

当需要替换的 Cache 块是干净的时，不会写回这个 Cache 块。

测试点 4.3.1：替换干净的 Cache 块时不会发生写回操作
测试用例：**测试用例 2**

4 测试环境

4.1 硬件环境

AMD Ryzen 7 7840H，32GB

4.2 软件环境

操作系统：Ubuntu 22.04LTS
测试工具集:

名称	版本号
*python	3.10.12
*pytest	8.1.1
*pytest-cov	5.0.0
*pytest-html	4.1.1
*picker	0.1.0-master-418d502

其中，标注 * 的测试工具为必要的环境。测试代码详见XS-MLVP-Nutshell Cache verification。

5 测试用例

为了覆盖上面所有功能点和拆分出来的测试点，我们设计了如下五个测试用例。由于读写数据的正确性在每个测试用例中都会检测，因此下面所有的测试用例都覆盖了**功能点 1**。其中，第一个测试用例主要测试 Cache 命中，覆盖**功能点 3**；第二个测试用例主要测试 Cache 缺失，覆盖**功能点 4**；第三个测试用例测试 MMIO 的功能，覆盖**功能点 4**。之后，为了模拟 Cache 真正运行时的行为，还设计了两个测试用例，分别是随机读写和顺序读写。

图8是功能点、测试点、测试用例的对应关系。

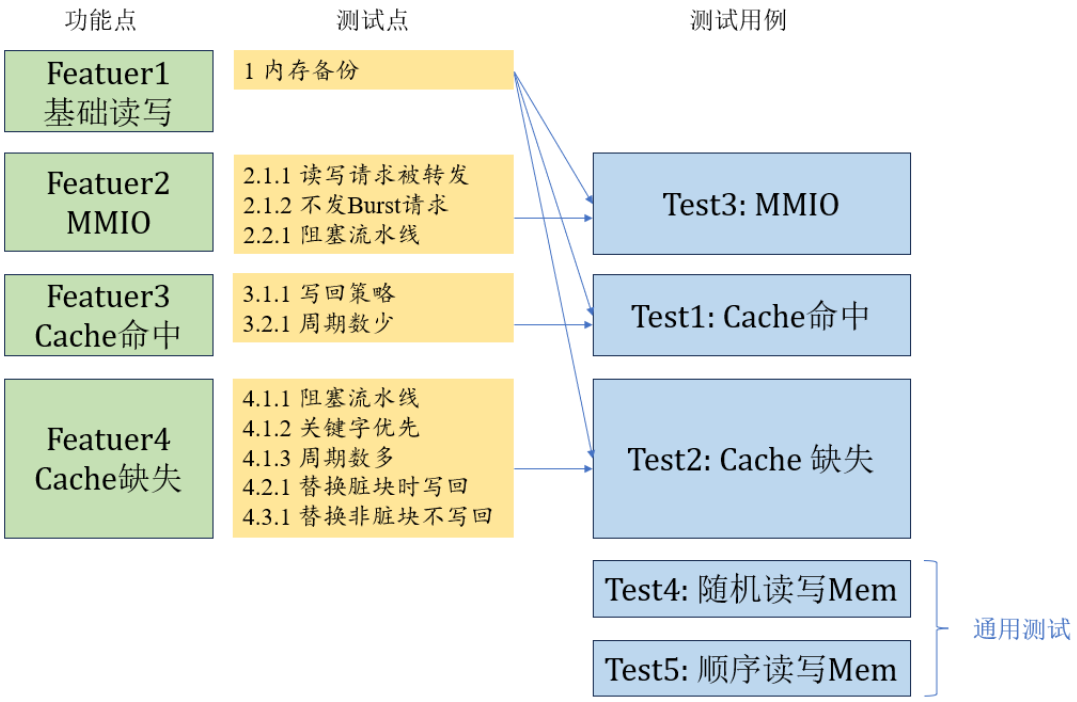


图 8: 功能点、测试点、测试用例的对应关系

注：通过 tb_cache.py 中的 testlist 指定需要运行的测试用例。

5.1 测试用例 1: Cache Hit Test

这个测试用例主要测试 Cache 命中的情况。采用先顺序读，再顺序写的方式，使得对 Cache 块是否 dirty 有准确预期，就能够在替换 Cache 块时准确判断是否应该写回，从而验证写回策略。读写结果和响应周期都容易验证。

表 5: Cache 命中的测试用例

用例	目标功能点	运行方法
测试用例 1 (cache_hit_test)	**测试点 1** **测试点 3.1.1** **测试点 3.2.1**	修改 tb_cache.py 中的 testlist 为”cache_hit”, 运行命令 make pytest
用例说明		

步骤 1: Cache 初始化 **步骤 2**: 顺序读 0x0000 到 0x8000 地址内容（32KB，恰好为 cache 大小），以保证 cache 已被完全填充 **步骤 3**: 随机读写 0x0000 到 0x8000 地址的内容。
预期结果 **测试点 1**: Cache 读写结果与 Ref Cache 一致。下同，不再赘述 **测试点 3.1.1**: 对干净块进行写操作时，内存端口上 (out_mem) 不应观察到读写操作；替换干净块时，内存端口上不应观察到写操作 **测试点 3.1.2**: 读写请求被接受后，回复时间在 3 个周期内

5.2 测试用例 2: Cache Miss Test

这个测试用例主要测试 Cache 缺失的情况。采用先顺序读，再顺序写，使得对 Cache 块是否 dirty 有准确预期，可以检测受害者行是脏块时写回，不是脏块时不写回。并且，以 Cache Line 大小为粒度顺序写的方式，使得写时会连续 Miss，这样就可以验证连续 Miss 时是否阻塞流水线。读写结果和响应周期都容易验证。

表 6: Cache 缺失的测试用例

用例	目标功能点	运行方法
测试用例 2 (cache_miss_test)	**测试点 1** **测试点 4.1.1** **测试点 4.1.2** **测试点 4.1.3** **测试点 4.2.1** **测试点 4.3.1**	修改 tb_cache.py 中的 testlist 为”cache_miss”，运行命令 make pytest
用例说明 **步骤 1**: Cache 初始化 **步骤 2**: 顺序读 0x0000 到 0x8000 地址内容以填满 Cache。 **步骤 3**: 以 cacheline 大小（64B）为粒度访问 0x9000 到 0x10000 的地址。具体来说，访问的地址序列为 0x9000, 0x9040, 0x9080...		
预期结果 **测试点 4.1.1**: 连续发出 Miss 的请求后，观测到 cache 请求端口 (io_in_req) 的 ready 信号拉低 **测试点 4.1.2**: 向内存发出读请求时，发出的地址应当是请求时的地址 (io_in_req_bits_addr) 按 8B 对齐后的地址 **测试点 4.1.3**: 请求被接受后，回复时间在 3 个周期以上 **测试点 4.2.1**: 替换块为脏块时，mem 端口可以观测到两次请求，第一次请求为写操作，地址为该 cacheline 的起始地址 **测试点 4.3.1**: 替换块为干净块时，mem 端口仅观测到一次请求，为读请求		

5.3 测试用例 3: MMIO Test

这个测试用例主要测试 MMIO 的读写情况。直接顺序读写，MMIO 端口上是否有请求信号、控制信号是否是 Burst、Cache 请求端口是否阻塞，这些都容易观测。

表 7: MMIO 的测试用例

用例	目标功能点	运行方法
测试用例 3 (mmio_test)	**测试点 1** **测试点 2.1.1** **测试点 2.1.2** **测试点 2.2.1**	修改 tb_cache.py 中的 testlist 为"mmio_serial"，运行命令 make pytest
用例说明 **步骤 1**: Cache 初始化 **步骤 2**: 以 4B 为粒度，读写访问 0x30000000 到 0x30001000 的地址 (MMIO 范围)		
预期结果 **测试点 2.1.1**: 发出 MMIO 请求后，在 MMIO 端口上可以观测到一样的请求信号 **测试点 2.1.2**: 发出 MMIO 请求时，MMIO 端口上不应出现 Burst 读写请求控制信号 **测试点 2.2.1**: 连续发出 MMIO 请求后，观测到 cache 请求端口 ready 信号拉低		

5.4 测试用例 4: Random Test

表 8: 随机读写测试的测试用例

用例	目标功能点	运行方法
测试用例 4 (random_test)	General Test	修改 tb_cache.py 中的 testlist 为"random"，运行 make pytest
用例说明 **步骤 1**: Cache 初始化 **步骤 2**: 对 0x0 到 0xffffffff 全地址空间进行有阻塞的随机读写。有阻塞是指需要等 cache 回复当前读写请求后再发起下一次请求		
预期结果 仅作正确性验证 (测试点 1)		

5.5 测试用例 5: Sequencial Test

表 9: 顺序读写测试的测试用例

用例	目标功能点	运行方法
测试用例 5 (sequencial_test)	General Test	修改 tb_cache.py 中的 testlist 为"seq"，运行 make pytest
用例说明 **步骤 1**: Cache 初始化 **步骤 2**: 对 0x1000 到 0x9000 进行地址连续的无阻塞读写测试。粒度为 4B，每次发出 3 个请求。		
预期结果 仅作正确性验证 (测试点 1)		

6 结果分析

6.1 测试用例分析

在所有测试用例的运行过程中，均覆盖了预期测试的功能点，完成了既定的目标。

6.2 覆盖率分析

6.2.1 行覆盖率

表 10: 行覆盖率

名称	命中行数	总行数	行覆盖率
Cache.v	1180	1241	95.1%

实际的行覆盖率超过 95.1%，原因是代码中存在保证安全性的 assertion（例如 MMIO 请求不会触发 Cache Hit 事件）覆盖不到。

即使去除 assertion 语句之后，测试的给出的代码覆盖率没有达到 100%。没有覆盖到的代码包括以下关键词：**victimWayMask**，**io_mmio_req_ready**，**probe/coh**，**readBurst**。

（1）**io_mmio_req_ready** 是翻转覆盖率相关，因为采用的 SimpleRam 模型是理想模型，没有延迟，因此 Cache 总是认为 MMIO 端可以接受请求。同时，与 Memory，由于 MMIO 不会请求 Burst，因此 Ready 信号总是拉高。

（2）**probe/coh** 是一致性相关的信号，不在此次验证的考察范围之内。

（3）**readBurst** 是当 Cache 为 L2 Cache 信号时才会使用到的信号，此次验证将 Cache 作为 I/D Cache 来进行验证，因此没有触发该信号。

（4）**victimWayMask** 涉及到选择替换的 Cacheline 的，是随机选择的一个边界情况。

剔除上述因素后，实际行覆盖率为 **97.3%**。

6.2.2 功能覆盖率

测试点的具体测试方法如下表所示：

表 11: 测试点具体说明

测试点	说明
测试点 1	测试运行时，参考模型会同步监听 cache dut 顶层的请求信号。当为写信号时，同步中参考模型中完成写入操作；当为读信号时，自动比对 cache dut 顶层返回的读数据，判断是否与参考模型中的数据一致。不一致则报错。

测试点 2.1.1	测试运行时，参考模型会同步监听 cache dut 顶层的请求信号，同时模拟一个请求队列。当监测到 cache dut 请求时，将请求信息入队；当监测到 cache dut 请求被响应时，将请求信息出队（以下不再叙述该队列模型）。监测 cache 的 MMIO 端口，当 MMIO 端口有请求时，比对请求的信息（地址、控制信号等）是否与队首的记录一致。不一致则报错，否则认为功能点被覆盖。
测试点 2.1.2	测试运行时，若 MMIO 端口有请求，检查其请求类型，若为 Burst 请求则报错，否则认为功能点被覆盖。
测试点 2.2.1	测试运行时，若检测到 cache dut 的顶层信号中 io_req_ready 信号无效，并且当前队首记录的请求类型为 MMIO 请求，则认为功能点被覆盖。
测试点 3.1.1	测试运行时，若队首记录为一个普通写请求，查询参考模型中的 cacheline 状态，若该 cacheline 显示为在 cache 中，则检查 cache 的 memory 端口是否发起了写请求，若发出了写请求则认为 cache 没有将写的内容缓存在 cache 中，而是直接写到了内存，进而认为没有采用写回策略，功能点未覆盖。
测试点 3.2.1	测试运行时，若 cache 响应某个请求并将其出队，检查该项入队的时间戳（周期数），若其与当前的周期数差值小于 3，则认为功能点被覆盖。
测试点 4.1.1	测试运行时，若检测到 cache dut 的顶层信号中 io_req_ready 信号无效，并且当前队首的请求类型为普通请求，则认为功能点被覆盖。
测试点 4.1.2	测试运行时，若 Memory 端口发出请求，其发出的首地址，若与当前队首请求的地址一致，则认为功能点被覆盖。
测试点 4.1.3	测试运行时，若 cache 响应某个请求并将其出队，检查该项入队的时间戳（周期数），若其与当前的周期数差值超过 3，则认为功能点被覆盖。
测试点 4.2.1	测试运行时，若 cache 的 Memory 端口发出写请求，首先查询参考模型该 cacheline 的情况，若不为 dirty 则报错，然后置标志 is_miss_dirty；当 Memory 端口发出读请求时，若 is_miss_dirty 则认为已经完成了先写的操作，此时认为功能点被覆盖，然后将标志位清空。
测试点 4.3.1	测试运行时，若 cache 的 Memory 端口发出读请求，检查上一条中的标志位，若标志位为 0，则认为替换的是干净块，进而认为功能点被覆盖。

下表是测试的功能覆盖情况：

表 12: 功能覆盖情况

功能点	测试点	是否覆盖
基础功能	**Featrue 1**	✓
MMIO	**测试点 2.1.1**	✓
	测试点 2.1.2	✓
	测试点 2.2.1	✓
Cache 命中	**测试点 3.1.1**	✓
	测试点 3.2.1	✓
Cache 缺失	**测试点 4.1.1**	✓
	测试点 4.1.2	✓
	测试点 4.1.3	✓
	测试点 4.2.1	✓
	测试点 4.3.1	✓
Functional Coverage: 100%		
Hit:✓, Miss:−, Fail:×		

可以看到所有功能均被覆盖且正确。

7 缺陷分析

验证结果显示果壳 Cache 未存在功能性缺陷，主要缺陷存在于设计文档。

7.1 设计文档缺陷：前递相关

设计文档中缺少对前递信号（write forward）的说明，导致设计用例覆盖相关代码行时难度较大。

7.2 设计文档缺陷：一致性相关

设计文档中缺乏对一致性相关功能（coherence）的说明，缺乏对相关接口信号的描述，导致难以设计用例覆盖代码行。

8 测试结论

针对 MMIO、Cache 命中以及 Cache 缺失的情况设计的用例测试显示，Nutshell L1 Cache 功能正常；随后通过随机读写和顺序读写的综合测试，进一步验证了 Nutshell L1 Cache 的健壮性，可以认为其功能正确。综上所述，在验证内容范围内，NutShell L1 Cache 功能正确。