从 AMD Zen 看 CPU 发展趋势

从 AMD 新型 x86 核心架构 Zen 带来的改进看 CPU 架构的发展趋势

1 背景

1.1 AMD ZEN 架构简介

“Zen”核心是 AMD 公司推出的全新的 x86 处理器微架构设计。“Zen”结合了高吞吐和低功耗设计方法的最新思考，是一个为家用电脑、笔记本、数据中心和超级计算机设计的多功能平衡架构。(AMD, 2017)

首款 Zen 微架构的处理器，核心代号为“Summit Ridge”，正式品牌名称为“Ryzen”，中文名称为“锐龙”，于 2017 年 3 月 2 日上市。 (AMD, 2017)

1.2 AMD x86 微架构历史

AMD 的 x86 微架构有 K5、K6、Athlon/K7、K8、K9、10h/K10、Bulldozer、Zen。

1.2.1 推土机（Bulldozer）微架构

AMD Bulldozer（推土机）微架构是 Zen 微架构的前任架构，于 2011 年 9 月 19 日发布。 (Wikipedia, 2017)

Figure 1 推土机（Bulldozer）微架构模块框图展示了推土机微架构的基本组成：

1. 缓存
2. 分支预测
3. 指令抓取和译码
4. 转移旁路缓存
5. 整数单元
6. 浮点单元
7. 存取单元
8. 集成内存控制器
9. 其他单元

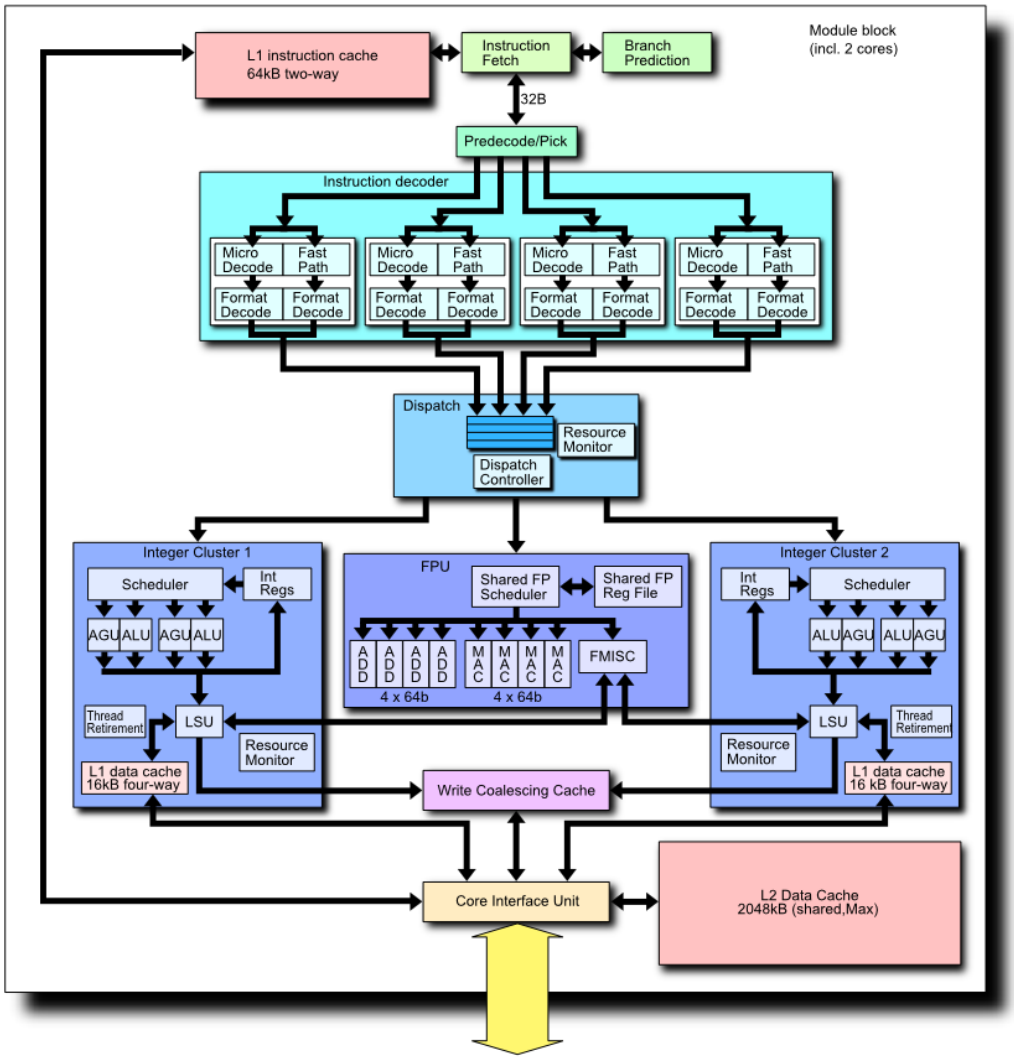


Figure 1 推土机（Bulldozer）微架构模块框图

下面介绍一些较为关键的，在 Zen 架构中将会得到改进的，一部分推土机微架构的技术规格。 (AMD, 2014) (Barragy, 2012)

1.2.1.1 缓存

推土机微架构有四种不同的缓存来加速指令执行和数据处理：

- L1 指令缓存
- L1 数据缓存
- 共享计算单位 L2 缓存
- 片上共享 L3 缓存

1.2.1.1.1 L1 指令缓存

推土机微架构包含一个 64KB 两路组相连映射的 L1 指令缓存，缓存每行 64 字节，但每个周期只抓取 32 字节。

与 L1 指令缓存相关的功能有：

- 指令加载
- 指令预取
- 指令预编码
- 分支预测

在 L1 指令缓存中未命中的请求将依次从 L2 缓存、L3 缓存或系统内存中获取。

1.2.1.1.2 L1 数据缓存

推土机微架构包含一个有 128 位端口的 16KB 四路预测 L1 数据缓存。这是一个透写（Write-Through）缓存，每个周期支持两个 128 字节的负载。

L1 数据缓存被分为 16 块，每块 16 字节宽。同时，L1 数据缓存通过奇偶校验避免 1 位数据的错误。

硬件预取器可以将数据存入 L1 数据缓存以避免未命中的情况。

一个周期只能加载一个 L1 数据缓存块。

1.2.1.1.3 L2 缓存

推土机微架构的每个计算单元有一个共享 L2 缓存。

L2 缓存是回写（Write-Back）缓存。每次某个核心存储数据，地址将被写入该核心的 L1 数据缓存以及 L2 缓存（双核共享）。

1.2.1.1.4 L3 缓存

推土机微架构支持每个晶粒最大 8MB 的 L3 缓存，分布在 4 个 L3 子缓存中（每个最大 2MB）。

L3 缓存是为多核处理器优化的非包含受害者缓存（non-inclusive victim cache）架构。

仅当 L2 删除缓存项时，L3 才会分配存储该项。

命中的 L3 缓存项，在当该项数据更可能被多核访问时，可以仍然保持在 L3 缓存中；当其可能只被某一核心访问时，可以从 L3 缓存中移除它，并将其放于只对应核心的 L1 缓存中，以在 L3 缓存中留出更多空间存放 L2 缓存的受害者缓存项。

1.2.1.2 分支预测

推土机微架构采用了以下分支预测机制：

- 下一地址逻辑单元（Next-Address Logic）
- 2 级分支目标缓冲（BTB，Branch Target Buffer）
- 混合分支预测（Hybrid Branch Predictor）
- 直接目标预测（Direct Target Predictor）
- 间接目标预测（Indirect Target Predictor）

- 返回地址栈（RAS，Return Address Stack）
- 抓取窗口跟踪结构（Fetch Window Tracking Structure，BSR）

1.2.1.2.1 下一地址逻辑单元（Next Address Logic）

该单元决定了抓取指令的地址。

如果当前抓取块中没有分支指令，该单元顺序计算下一个 32 字节的指令抓取块的起始地址。这个计算过程每个时钟周期执行一次。

如果分支指令存在，该单元重定向到分支目标，分支方向预测硬件会生成一个非顺序的指令抓取块地址。

1.2.1.2.2 分支目标缓冲（BTB，Branch Target Buffer）

分支目标缓冲是一个二级结构，使用当前指令抓取块的地址访问。

每个 BTB 项包括一个分支及其跳转目标的信息。

一级 BTB 是稀疏分支预测器，映射每个指令缓存行（64 字节）的前 2 个分支，包含 128 组 4 路共 512 项；

二级 BTB 是密集分支预测器，映射同一指令缓存行中每 8 个字节的指令块中额外的 2 个分支，包含 1024 组 5 路共 5120 项。

1.2.1.2.3 混合分支预测器

混合分支预测器用于预测条件分支，包含一个全局预测器、局部预测器和一个选择器。选择器会追踪每一个分支与全局或局部预测器的相关性的好坏。

选择器和局部预测器在一个线性地址哈希表中索引，全局预测器通过一个 2 位地址哈希表和一个 12 位全局历史表访问。

1.2.1.2.4 间接目标预测

推土机微架构实现了一个独立的 512 项间接目标数组用于预测间接分支。

1.2.1.2.5 返回地址栈

推土机微架构实现了一个 24 项返回地址栈来预测返回地址。

1.2.1.3 集群多线程（CMT，Clustered Multi-Threading）

推土机微架构引入了“集群多线程”技术，可以使处理器的一部分在线程间共享。

CMT 是一种与 SMT（并发多线程，Simultaneous Multi-Threading）具有相同设计哲学，但更为简单的多线程设计。

通过 Figure 1 推土机（Bulldozer）微架构模块框图可以看到，推土机微架构模块中有两个整数集群，每个整数集群中有 2 个 ALU 和 2 个 AGU，每个周期可以处理总共 4 个独立的算术和内存操作。而这两个集群共享一个 FPU。

因此，推土机的 CMT 模块在整数能力上相当于一个双核处理器，但浮点处理能力取决于浮点指令是否在模块的两个线程中同时存在，以及 FPU 处理的是 128 位还是 256 位浮点数——如果同时存在，则

相当于单核处理器的浮点处理能力。不过，CMT 架构在使用单线程时，会有大量整数执行单元闲置。(Wikipedia, 2017)

1.2.1.4 指令集

推土机微架构属于复杂指令集处理器架构，使用 x86-64/AMD64 指令集。它支持的扩展指令集有：

- SSE4：Streaming SIMD（Single Instruction，Multiple Data）Extension 4
- AES-NI：Advanced Encryption Standard Instructions
- AVX：Advanced Vector Extensions
- XOP：AMD 的 SSE5 修订
- FMA4：Fused Multiply-Add（FMAC）Instructions
- TBM：Trailing Bit Manipulation
- LWP：Lightweight Profiling

其中前三个是 Intel 公司提出的 x86 扩展指令集标准，后面四个是 AMD 的 x86 扩展指令集。(Hollingsworth, 2012)

2 AMD ZEN 的改进

AMD Zen 微架构做了很多架构上的改进，相比第四代推土机微架构（挖掘机微架构）的 IPC（Instructions Per Clock）性能有 40%的提升。 (AMD, 2016)

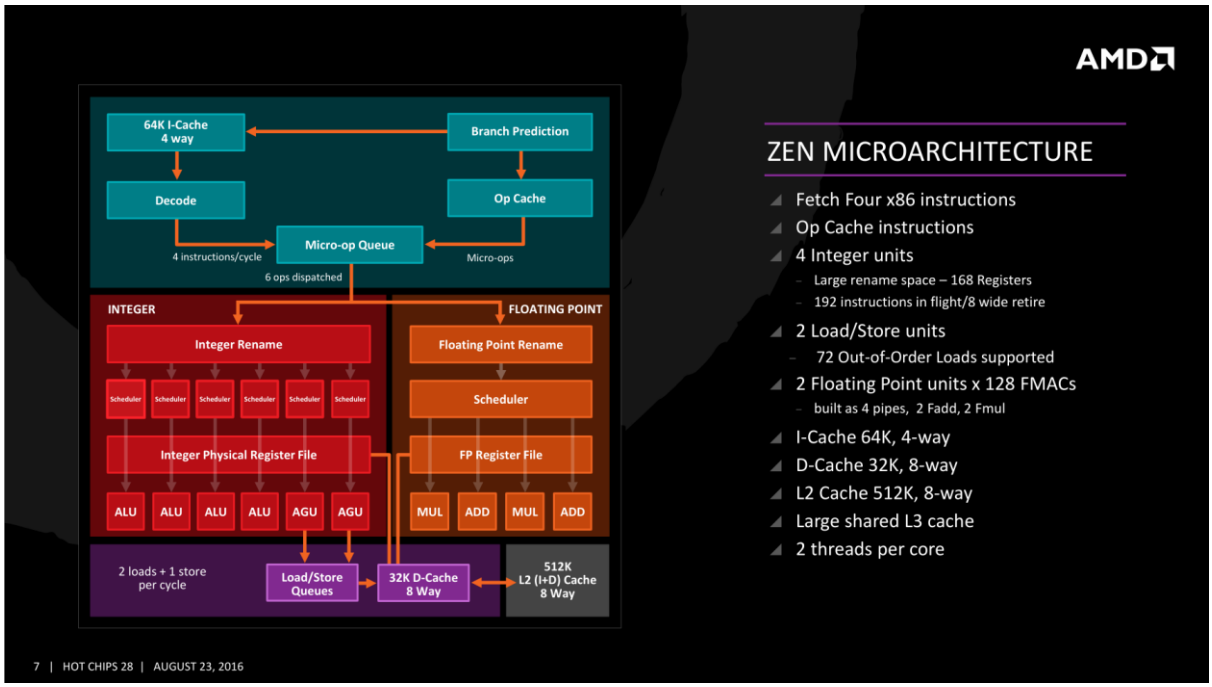


Figure 2 AMD Zen 微架构

2.1 缓存

AMD Zen 微架构对 L1 缓存做了特别改进，其写入策略从透写（Write-Through）改为回写（Write-Back），降低了延迟和功耗，提高了带宽。

同时，Zen 微架构提供了更快更大的 L2 和 L3 缓存，以及更好的 L1 和 L2 数据预取器。

2.2 分支预测

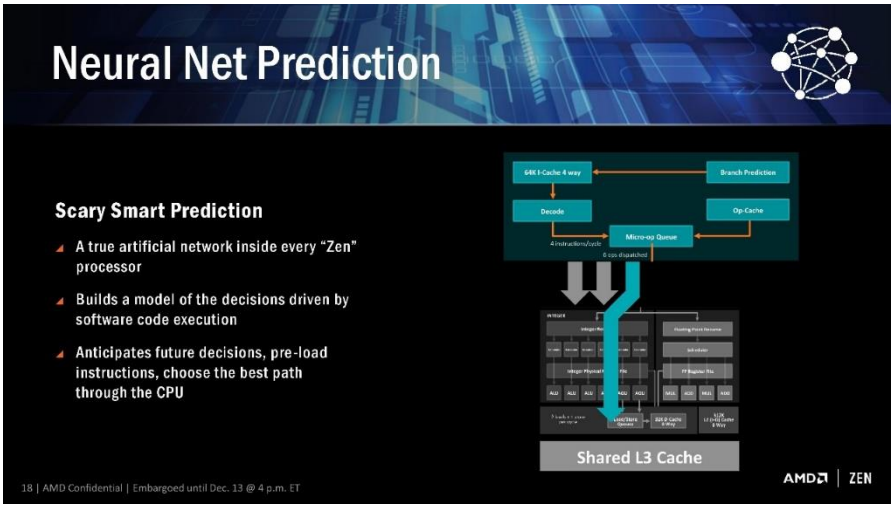


Figure 3 AMD Zen 微架构神经网络分支预测

Zen 微架构在分支预测上的改进有：

- 更大的 BTB
- 每个 BTB 项包含两个分支
- 32 项返回地址栈
- “神经网络”分支预测（感知器分支预测器）

感知器分支预测器通过学习执行的指令和数据通路，适应应用程序的模式，为未来的分支做出更准确的预测，减少潜在的延迟。 (Amata, 2017)

2.3 并发多线程（SMT，SIMULTANEOUS MULTI-THREADING）

Zen 微架构抛弃了 Bulldozer 微架构的集群多线程技术，转而采用和 Intel 基本一致的并发多线程技术。

SMT 技术不存在整数单元空闲的情况，在单线程状态下，所有的整数单元都会被单线程利用，提高了吞吐率，提升了单线程性能。 (AMD, 2016)

我们可以根据 Figure 4 Zen 微架构并发多线程（SMT）技术概览和 Figure 5 Bulldozer 和 Zen 微架构对比两图看到，Zen 微架构不再是两个整数集群（Cluster）构成，多线程将共享整数单元里的多个 ALU（4 个 ALU 和 2 个 AGU，参见 Figure 6 Zen 微架构概览图）。 (Wikipedia, 2017)

同时，资源共享简化了电路，使空间紧凑，扩展性强，功耗降低。CMT 技术中每个集群的独立单元造成空间的极大占用，却无法提供较高的单线程性能，在大规模集成上就存在劣势。 (Scali, 2012)

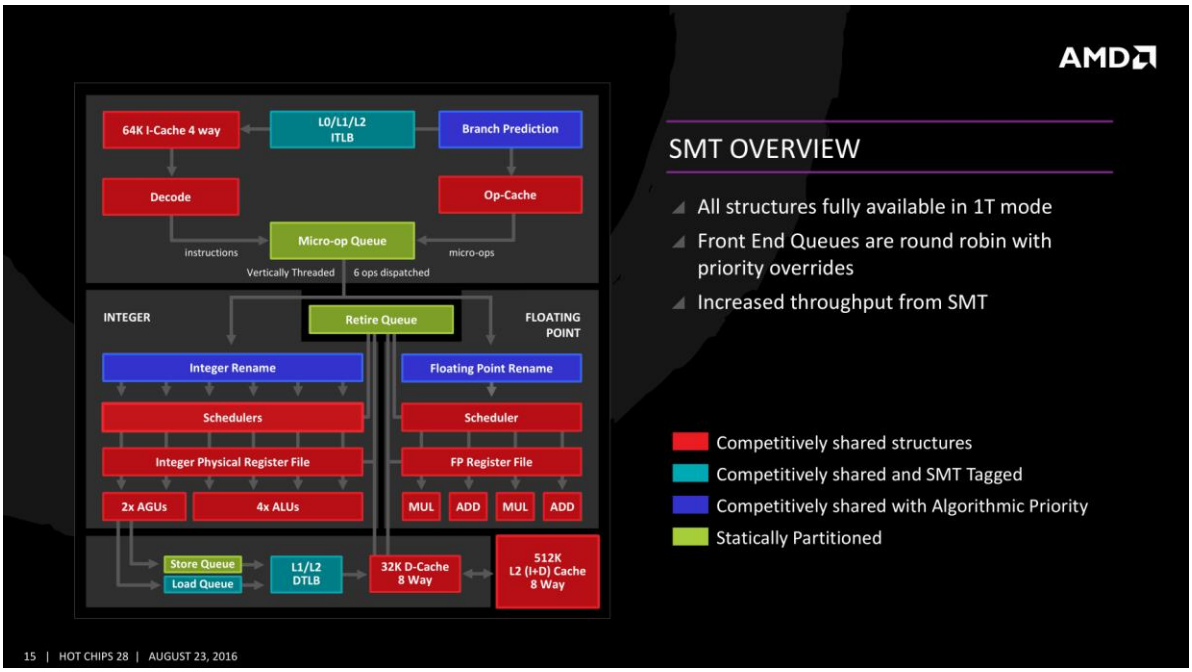


Figure 4 Zen 微架构并发多线程（SMT）技术概览

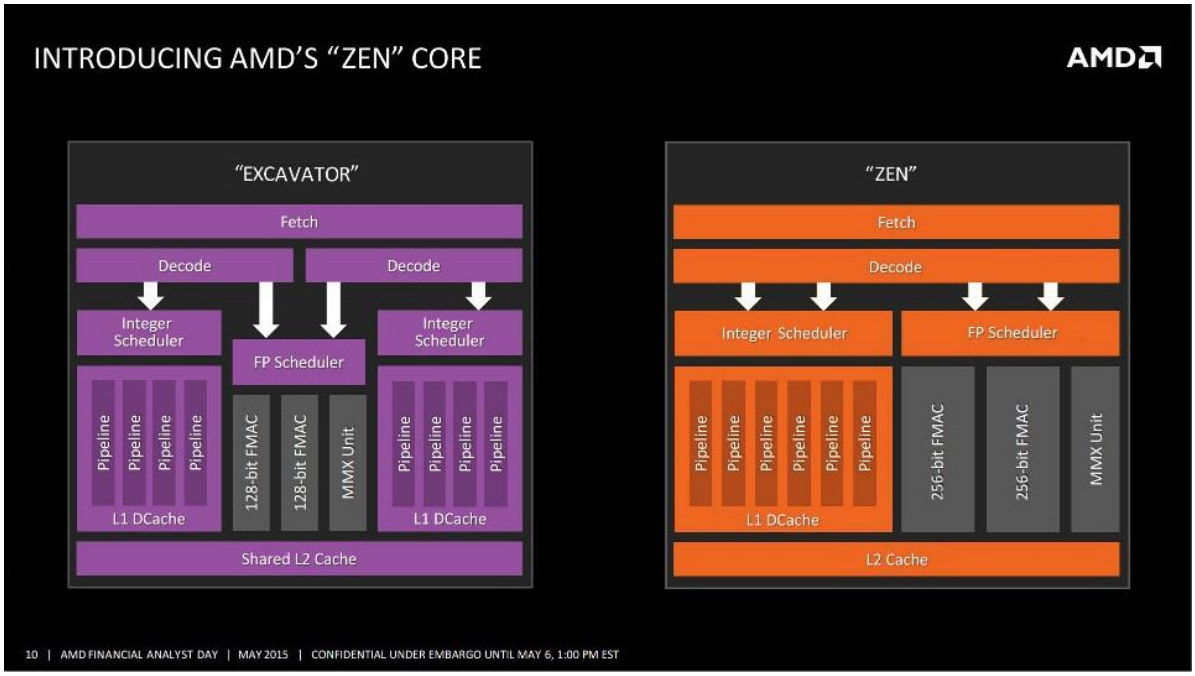


Figure 5 Bulldozer 和 Zen 微架构对比

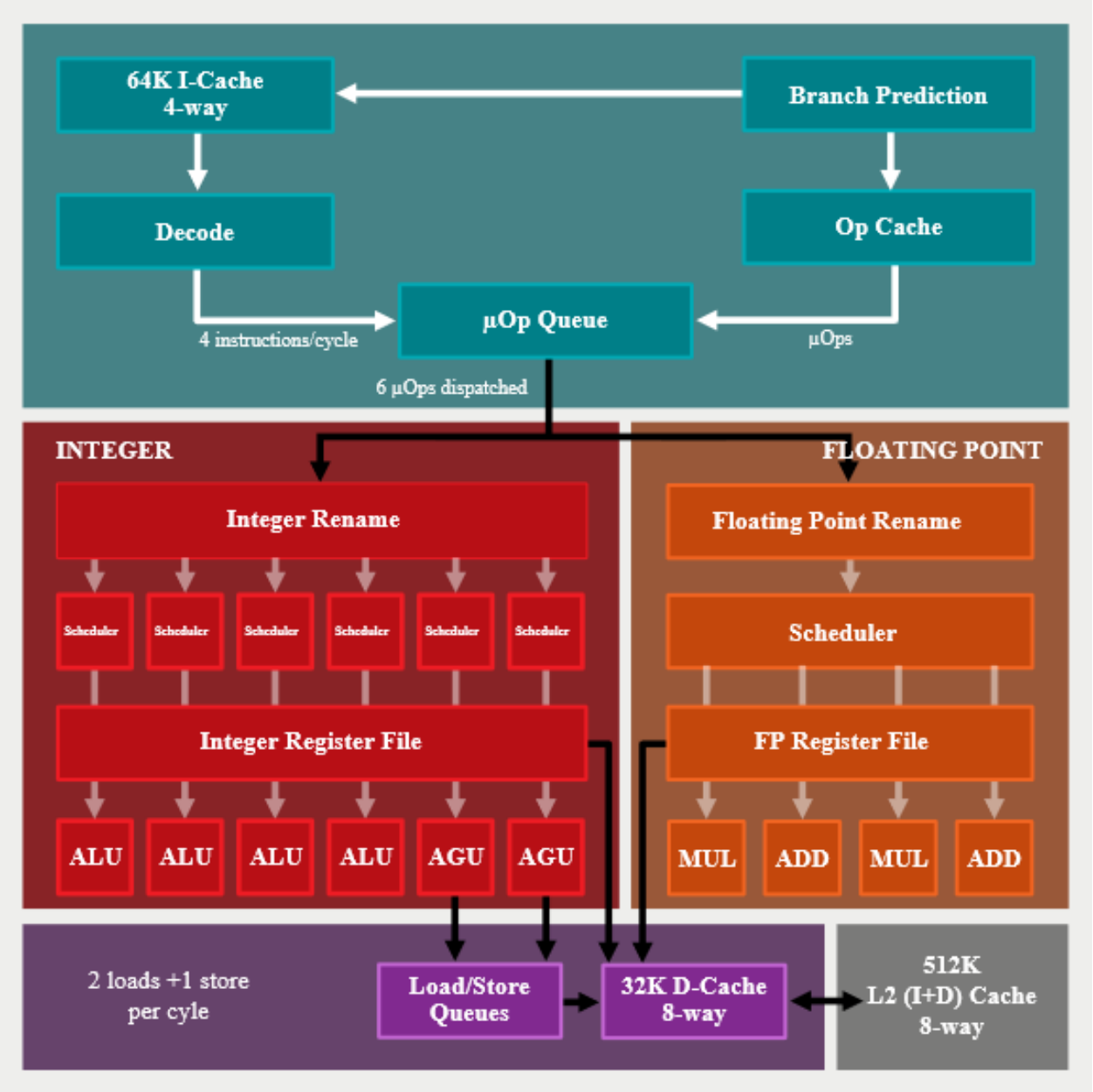


Figure 6 Zen 微架构概览图

2.4 指令集

Zen 微架构取消了 4 个 AMD 特有的指令集：TBM、FMA4、XOP 和 LWP。

新增了对大量主流指令（尤其是 Intel 支持的指令）的支持，如：AVX2、BMI、BMI2、FMA3 等，保持了与 Intel 处理器的良好兼容性。 (AMD, 2017)

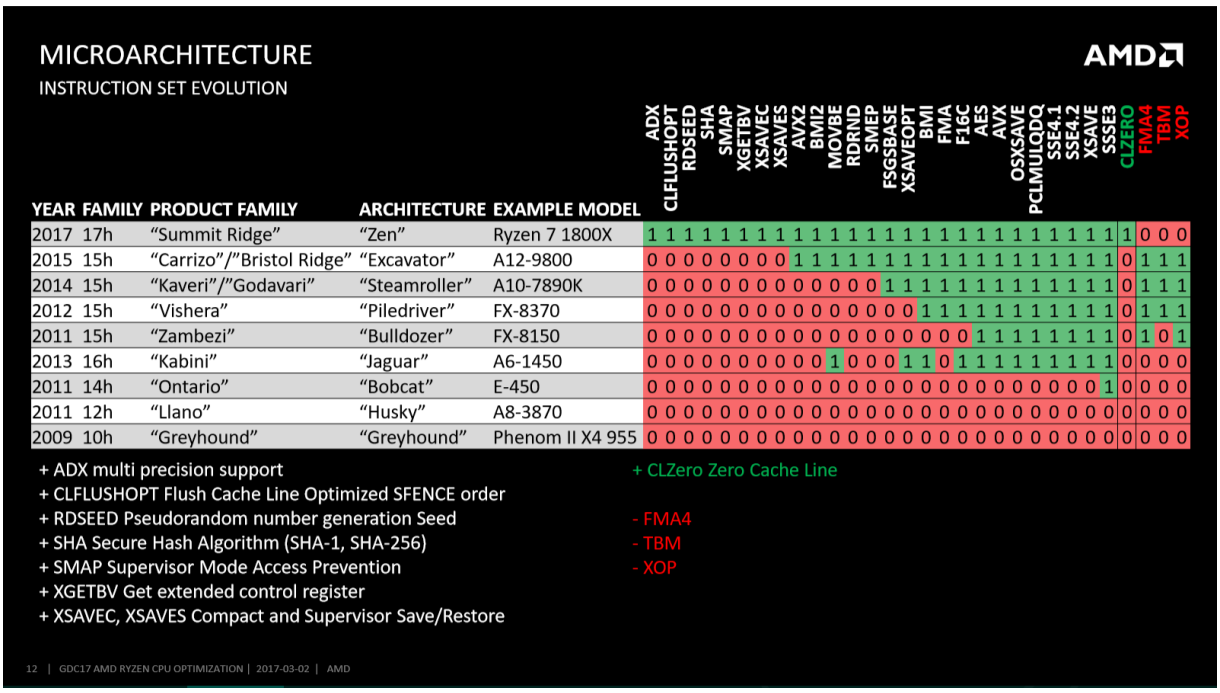


Figure 7　AMD 指令集发展

3 技术发展趋势

3.1 缓存

3.1.1 容量与延迟

增加缓存容量可以提高缓存命中率，但同时可能增加访问延迟。

缓存的访问延迟有两种表现：

- 时钟周期的延长（CPU 最大频率降低）
- 缓存占用的流水线层数增加（缓存访问的周期数增加）

一般现代 CPU 的 L1 缓存访问都会占用 3-4 个时钟周期，即使用上述第二种方式。

时钟周期的延长或周期数的增加都会导致性能的下降。

L1 缓存容量位于 32KB-128KB 之间时，可能可以获得较好的性能；超出范围后，延迟带来的性能下降会凸显，同时成本也过高。

L2、L3 缓存也应当在容量、延迟与成本之间做出妥协，同时进一步提高 CPU 的时钟频率也有助于降低缓存延迟的影响。 (Prybylski, Horowitz, & Hennessy, 1988)

3.1.2 写入策略

3.1.2.1 透写（Write-Through）

- 每次未命中（无论读写）都需要向内存访问一个块
- 每次存储新数据都要写入内存

3.1.2.2 回写（Write-Back）

- 每次未命中（无论读写）都需要向内存访问一个块
- 当某一缓存项将被剔除时，需要向内存写入修改过的数据

3.1.2.3 比较

透写更慢，但结构更清晰，硬件构造简单，内存数据保持更新；

回写更快，但结构复杂，硬件设计难度大，尤其是在多核共享内存时。 (Weatherspoon, 2013)

3.1.2.4 发展

AMD Zen 微架构的 L1 缓存就从 Bulldozer 微架构使用的透写策略改为了回写策略，提高了 L1 缓存的性能。

为了缓存性能提升，硬件设计复杂度的部分提高是值得和必要的。

3.1.3 专用缓存

现代处理器越来越多使用专用缓存，下面列举一些常用的专用缓存，并介绍他们分别带来的性能优化。

CPU 缓存方面未来的发展可能依赖更多更专用的缓存，但不同厂商也应当做出选择，专用缓存带来的性能提升未必明显，需要与其他硬件设计综合考虑。

3.1.3.1 受害者缓存

受害者缓存是一个较小的全相联缓存：

- 包含上一级缓存最近替换掉的缓存块
- 如果上一级缓存未命中，而受害者缓存命中，则交换受害者缓存块到上一级缓存对应块
- 如果都未命中，上一级缓存块剔除到受害者缓存中

受害者缓存用于维持刚被剔除的缓存块，进一步降低了未命中的可能性。

Intel Skylake 系列处理器的 L4 缓存就是受害者缓存，但 Intel 在后续产品中改变了 L4 缓存的用途；

AMD Bulldozer 和 Zen 微架构处理器的 L3 缓存都是受害者缓存。

3.1.3.2 追踪缓存（Trace-Cache）

由于跳转和分支指令的存在，指令的抓取未必是连续的。处理器就需要额外的逻辑和硬件支持来获取非连续指令。

追踪缓存动态地按照执行的顺序存储指令。当这些指令再一次被抓取，可以通过追踪缓存抓取而无视具体的执行分支，减少抓取指令的延迟。 (Rotenberg, Bennett, & Smith, 1996)

这项技术由 Intel Pentium IV 引入，但后来实践发现性能提升有限，硬件设计复杂度却大大增加。于是，Intel 后续产品不再包含追踪缓存。

3.1.3.3 微操作缓存（Micro-operation Cache，又名 UC，即 Uop Cache、μop Cache）

微操作缓存接收指令译码器和指令缓存的结果，存储译码后指令的微操作。

当指令需要被译码时，检查微操作缓存是否有可复用的缓存。

在一系列性能测试中和超过 90%的多媒体应用及高耗能测试中，一个中等大小的微操作缓存减少了大约 75%的指令译码。在 Intel P6 处理器中，译码的减少在没有影响性能的条件下节约了大约 10%的能耗。 (Solomon, Mendelson, Orenstein, Almog, & Ronen, 2001)

3.2 分支预测

分支预测可以帮助在分支指令后预抓取后续需要执行的指令，提高执行效率。 (Kubiatowicz, 2011)

3.2.1 机构齐全

诸如分支目标缓冲（BTB）和分支历史表（BHT）、返回地址栈、相关分支预测（适应性预测）等分支预测机构应当配备齐全，提高分支预测准确性，同时控制功耗。

3.2.1.1 分支历史表（BHT，Branch History Table）

BHT 每一项是一个预测器，对应一个含分支指令的 PC（指令地址）的是否跳转状态。即每个预测器是一个状态机，状态决定了是否跳转。状态机可以是 1 位、2 位或多位的。

例如，1 位 BHT 在循环中会造成两次误判：

1. 退出循环时，它仍预测继续循环的分支
2. 首次进入循环，第一次循环后，它会预测跳出循环

因此，2 位 BHT 比较常见，如 Figure 8 2 位 BHT 的预测器状态机。

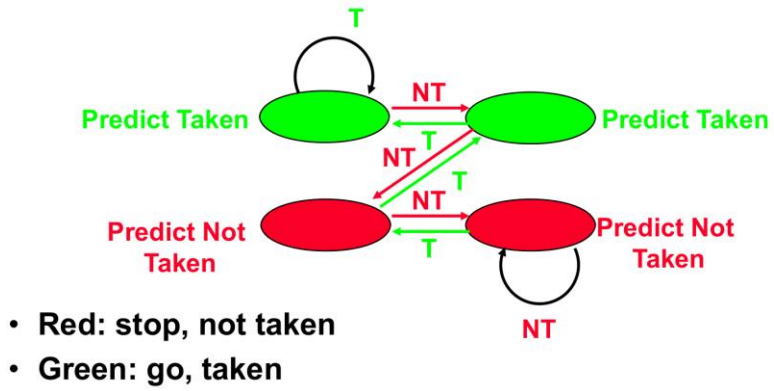


Figure 8 2 位 BHT

3.2.1.2 分支目标缓冲（BTB）

由于 BHT 仅仅是一个预测器，包含状态信息，没有实际的跳转目标地址的存储，实际预测时仍然需要耗费较多周期。因此采用 BTB，缓冲跳转的分支目标地址。

BTB 相比 BHT 更复杂，成本更高，因此大多将二者混合使用：

- 在指令抓取时，首先访问 BTB
- 若 BTB 未命中，使用 BHT 预测跳转
- 最后更新二者

3.2.1.3 返回地址栈

用于预测子程序返回之后跳转的目标地址。

每次调用时，压栈返回地址；每次返回时，出栈预测地址。

3.2.1.4 相关分支（Correlating Branch）预测

相关分支预测基于一个假设：最近的几个分支是相关的，也就是说最近执行的分支影响当前分支的预测。

一般有两种情况，当前分支依赖于：

1. 最后 m 条最近执行的分支
2. 最后 m 条由同一分支最近跳转的目标

前者属于全局适应（GA，Global Adaptive），后者属于地址适应（PA，Per-address Adaptive）。

因此，一般记录 m 条最近执行的分支是否跳转，使用特定模式选择合适的 BHT 项。

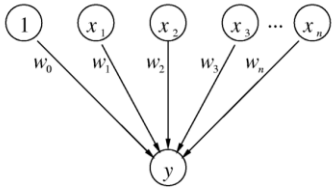
(m, n)预测器表示记录最后 m 个分支来从2^m项历史表项中选择，每个表项有 n 位计数器。 (Mirkovic, 2005)

3.2.2 机器学习（感知器分支预测器）

AMD Zen 微架构引入了神经网络分支预测技术，实质上是一种感知器分支预测器。

感知器的输入为分支跳转历史（x_i即第i位分支历史），1 表示跳转，-1 表示不跳转；输出为预测结果，负数为预测不跳转，非负数为预测要跳转。 (Jimenez & Lin, 2001)

有一系列权重需要经过在线训练，最终输出结果即权重和输入的点积（线性模型，如 Figure 9 分支预测感知器）。



$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$

Figure 9 分支预测感知器

训练算法如 Figure 10 感知器分支预测器训练算法：

$x_{1..n}$ is the n-bit history register, x_0 is 1.
$w_{0..n}$ is the weights vector.
t is the Boolean branch outcome.
θ is the training threshold.

```
if |y| ≤ θ or ((y ≥ 0) ≠ t) then
    for each 0 ≤ i ≤ n in parallel
        if t = x_i then
            w_i := w_i + 1
        else
            w_i := w_i − 1
        end if
    end for
end if
```

Figure 10 感知器分支预测器训练算法

即如果输出结果 y 的绝对值小于等于训练阈值，或输出结果 y 预测与实际分支结果 t 不一致，则更新权重。

对每个分支跳转历史，与当前实际分支结果一致则权重自增，否则权重自减。

可知，权重w_0即分支跳转的概率，w_1~w_n则是预测分支和分支历史跳转概率的比例（相关性）。

w_i和x_i点积则是考虑历史相关性的预测分支跳转概率。θ训练阈值用于防止训练过度，以更快适应新情况。

3.2.2.1 优势

这种方法综合利用了分支跳转历史，利用权重自适应算法，提高了预测的准确性。同时线性的感知器模型避免了大量的硬件使用，硬件复杂度不高。

分支预测属于预测分类问题，正是机器学习算法的用武之地，因此未来可能更多的利用新型的机器学习理念和算法优化 CPU 的分支预测等设计。

3.3 多线程技术

3.3.1 背景

在一个流水线中，可能存在命令的依赖关系等，造成时钟周期的浪费，即流水线冒险。

一般可以采用流水线互锁或旁路直通方法解决，但前者效率低，后者不能解决所有的冒险情况。

单线程的优化很难进行，现在多采用在同一流水线执行程序的不同线程实现指令的交叉执行。

多线程技术利用延迟来换取吞吐量：

- 共享处理器使单线程延迟增加
- 但改善了多线程总体的延迟
- 提高了 CPU 利用率

3.3.2 实现

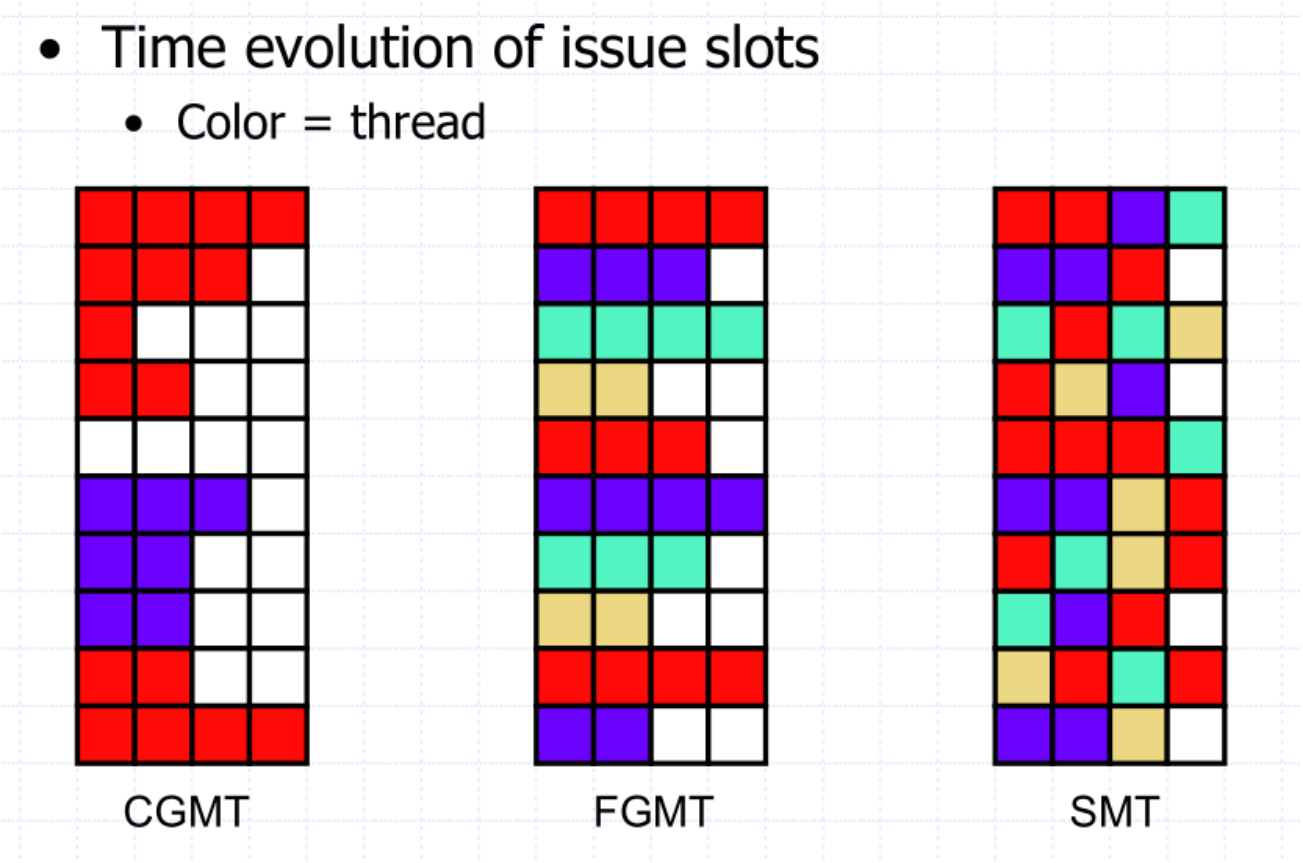


Figure 11 三种多线程实现 对比

3.3.2.1 粗粒度多线程（CGMT，Coarse-Grain Multi-Threading）

粗粒度多线程指定一个优先线程 A 持续执行 (Martin & Roth, 2005)

- 当 L2 未命中时切换线程 B
- 当线程 A 的 L2 未命中后的获取结果返回时，切换回原来的优先线程。

粗粒度多线程的特点：

- 优点：牺牲很少的单线程性能

- 缺点：只减少了 CPU 利用中一定程度的纵向浪费，只能容忍较大的延迟（L2 未命中），CPU 利用率仍然不够高

3.3.2.2 细粒度多线程（FGMT，Fine-Grain Multi-Threading）

细粒度多线程每个周期切换一次线程。

这意味着实现细粒度多线程，需要存在很多线程，也就意味着大量的寄存器文件（用于恢复线程环境）。

细粒度多线程的特点：

- 优点：去除了 CPU 利用中的纵向浪费，容忍所有的延迟（包括 L2 未命中、分支预测错误等）
- 缺点：牺牲了显著的单线程性能
- 缺点：需要大量线程

3.3.2.3 并发多线程（SMT，Simultaneous Multi-Threading）

并发多线程也是每个周期切换一次线程，但它是针对乱序执行处理器的多线程技术。

- 优点：去除了 CPU 利用中的横向和纵向浪费，容忍所有的延迟
- 缺点：牺牲部分单线程性能

3.3.2.3.1 与多核处理器对比

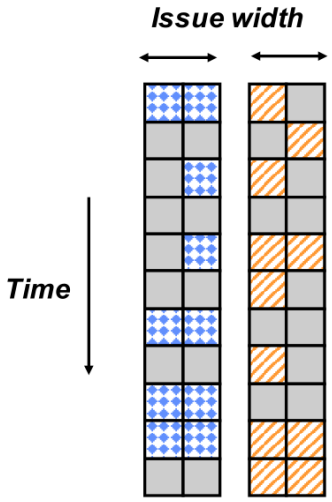


Figure 12 多核处理器的 CPU 利用

多核处理器实质上实现了线程级并发处理（TLP，Thread-Level Parallelism），减少了 CPU 利用中一定的横向浪费，保留了一些纵向浪费。

而并发多线程实现了指令级并发处理（ILP，Instruction-Level Parallelism），最大程度减少了 CPU 利用率的浪费。 (Wawrzynek, 2016)

现代处理器会综合以上两种方法，既实现多核处理器，也实现并发多线程。

3.3.3 发展

AMD Bulldozer 中使用的集群多线程（CMT）实质上也是一种 SMT，但对于资源共享和利用的设计上与主流 SMT 设计有所差异。虽然 AMD Bulldozer 的 CMT 多线程性能较高，但在单线程任务中闲置了部分运算器，牺牲了较多的单线程性能。

AMD Zen 抛弃 CMT，改用 SMT，在保证单线程性能的同时也提高了多线程性能。Intel 的 Hyper Threading 技术是 Intel 对 SMT 的商业命名，技术原理是一致的。

指令级并发处理（ILP）也是一直以来处理器技术的热点问题，多线程技术将不断改进，以更好的实现 ILP。

3.4 指令集

3.4.1 兼容性

AMD Zen 微架构取消了 AMD 专有的几个指令集，加入了 Intel 主导的几个扩展指令集。

Intel 的 x86 处理器是市场主流，17 年第一季度占据了市场份额约 80%。 (PassMark, 2017)

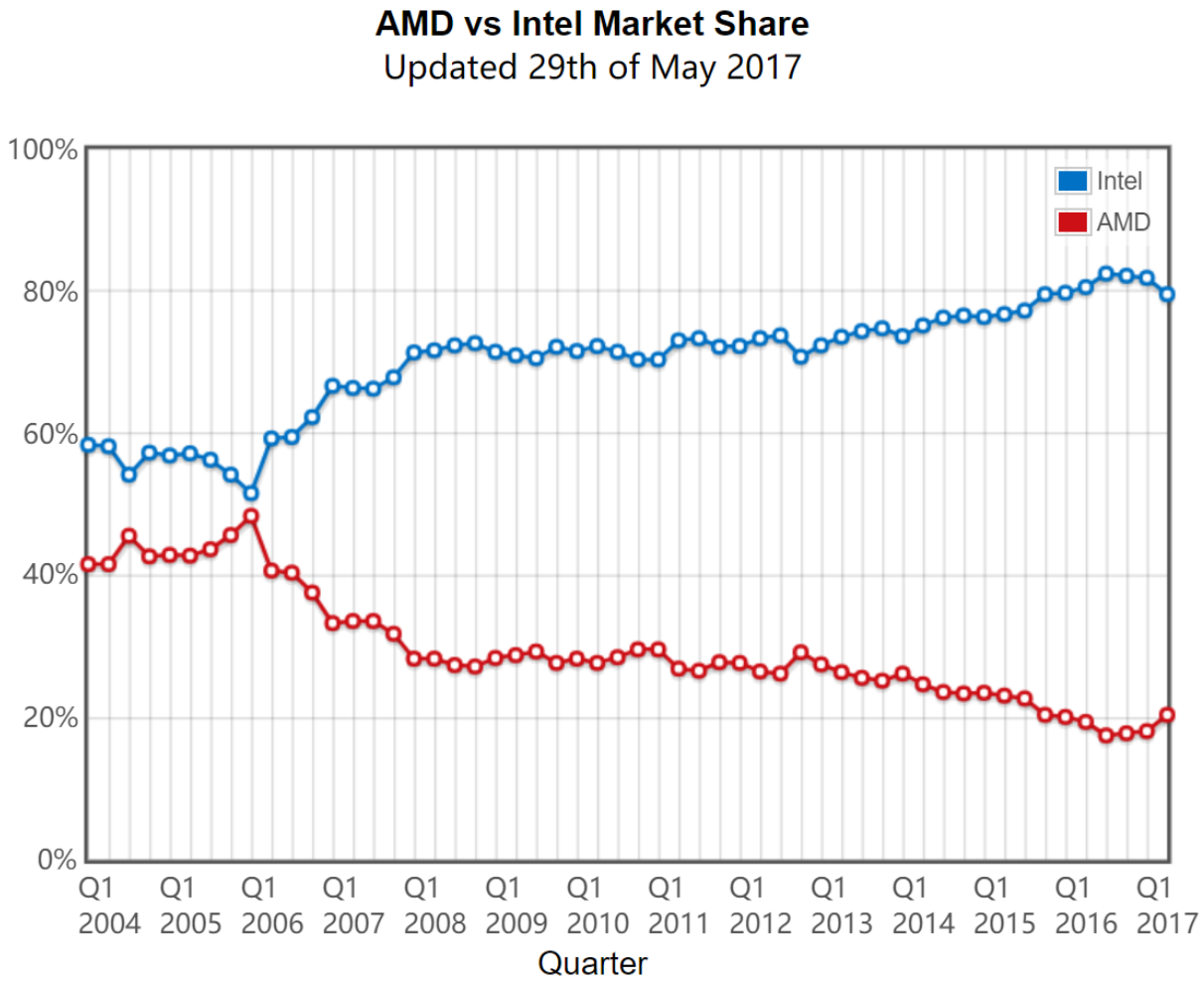


Figure 13 Intel 和 AMD 的 CPU 市场份额

因此，保持和 Intel 的兼容性，更能被市场所接受。

4 引用

Amata. (2017 年 1 月 13 日). Exclusive Interview With AMD’s Robert Hallock on Ryzen | Architecture, Performance & Chip Details. 检索来源: Exclusive Interview With AMD’s Robert Hallock on Ryzen | Architecture, Performance & Chip Details: http://www.redgamingtech.com/exclusive-interview-with-amds-robert-hallock/

AMD. (2014 年 1 月). Software Optimization Guide for AMD Family 15h Processor. United States. 检索来源: https://support.amd.com/TechDocs/47414_15h_sw_opt_guide.pdf

AMD. (2016 年 8 月 23 日). AMD and the new “Zen” High Performance x86 Core at Hot Chips 28. 检索来源: https://www.slideshare.net/AMD/amd-and-the-new-zen-high-performance-x86-core-at-hot-chips-28/1

AMD. (2017 年 5 月 20 日). "Zen" 核心架构. 检索来源: AMD: https://www.amd.com/zh-hans/technologies/zen-core

AMD. (2017 年 3 月 2 日). AMD Ryzen CPU Optimization. 检索来源: http://32ipi028l5q82yhj72224m8j.wpengine.netdna-cdn.com/wp-content/uploads/2017/03/GDC2017-Optimizing-For-AMD-Ryzen.pdf

AMD. (2017 年 3 月 2 日). 锐龙 AMD Ryzen7 台式处理器今日起全球上市！性能超越想象！. 检索来源: AMD: http://www.amd.com/zh-cn/press-releases/Pages/press-releases-2017Mar02.aspx

BarragyTed. (2012 年 1 月 23 日). Bulldozer. 检索来源: https://www.olcf.ornl.gov/wp-content/uploads/2012/01/TitanWorkshop2012_Day1_AMD.pdf

HollingsworthBrent. (2012 年 10 月). New “Bulldozer” and “Piledriver” Instructions - A step forward for high performance software development . 检索来源: https://developer.amd.com/wordpress/media/2012/10/New-Bulldozer-and-Piledriver-Instructions.pdf

JimenezD.A., & LinC. (2001). Dynamic branch prediction with perceptrons. High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on. Monterrey, Nuevo Leon, Mexico: IEEE.

KubiatowiczJohn. (2011 年 2 月 14 日). Prediction/Speculation (Branches, Return Addrs). 检索来源: https://people.eecs.berkeley.edu/~kubitron/courses/cs252-S11/lectures/lec08-prediction.pdf

Martin, & Roth. (2005). Multithreading. UNIVERSITY OF PENNSYLVANIA, United States. 检索来源: https://www.cis.upenn.edu/~milom/cis501-Fall05/lectures/12_smt.pdf

MirkovicJelena. (2005 年 10 月 27 日). Correlating Branch Predictors. 检索来源: https://www.eecis.udel.edu/~sunshine/courses/F05/CIS662/class14.pdf

PassMark. (2017 年 5 月 29 日). AMD vs Intel Market Share. 检索来源: CPU Benchmarks: https://www.cpubenchmark.net/market_share.html

PrybylskiS., HorowitzM., & HennessyJ. (1988). Performance tradeoffs in cache design. Proceeding ISCA '88 Proceedings of the 15th Annual International Symposium on Computer architecture (页 290-298). Honolulu, Hawaii, USA: IEEE Computer Society Press Los Alamitos, CA, USA ©1988.

RotenbergE., BennettS., & SmithJ.E. (1996). Trace cache: a low latency approach to high bandwidth instruction fetching. Microarchitecture, 1996. MICRO-29.Proceedings of the 29th Annual IEEE/ACM International Symposium on. IEEE.

Scali. (2012 年 2 月 14 日). The myth of CMT (Cluster-based Multithreading). 检索来源: Scali's OpenBlog: https://scalibq.wordpress.com/2012/02/14/the-myth-of-cmt-cluster-based-multithreading/

SolomonBaruch, MendelsonAvi, OrensteinDoron, AlmogYoav, & RonenRonny. (2001). Micro-operation cache: a power aware frontend for the variable instruction length ISA. Proceeding ISLPED '01 Proceedings of the 2001 international symposium on Low power electronics and design (页 4-9). Huntington Beach, California, USA: ACM.

WawrzynekJohn. (2016 年 10 月 25 日). Multithreading. University of California, Berkeley, United States. 检索来源: http://www-inst.eecs.berkeley.edu/~cs152/fa16/lectures/L14-Multithread.pdf

WeatherspoonHakim. (2013). Caches (Writing). United States: Cornel University. 检索来源: http://www.cs.cornell.edu/courses/cs3410/2013sp/lecture/18-caches3-w.pdf

Wikipedia. (2017 年 5 月 9 日). Bulldozer (microarchitecture). 检索来源: Bulldozer (microarchitecture): https://en.wikipedia.org/wiki/Bulldozer_(microarchitecture)

Wikipedia. (2017 年 5 月 21 日). Zen (microarchitecture). 检索来源: Zen (microarchitecture): https://en.wikipedia.org/wiki/Zen_(microarchitecture)