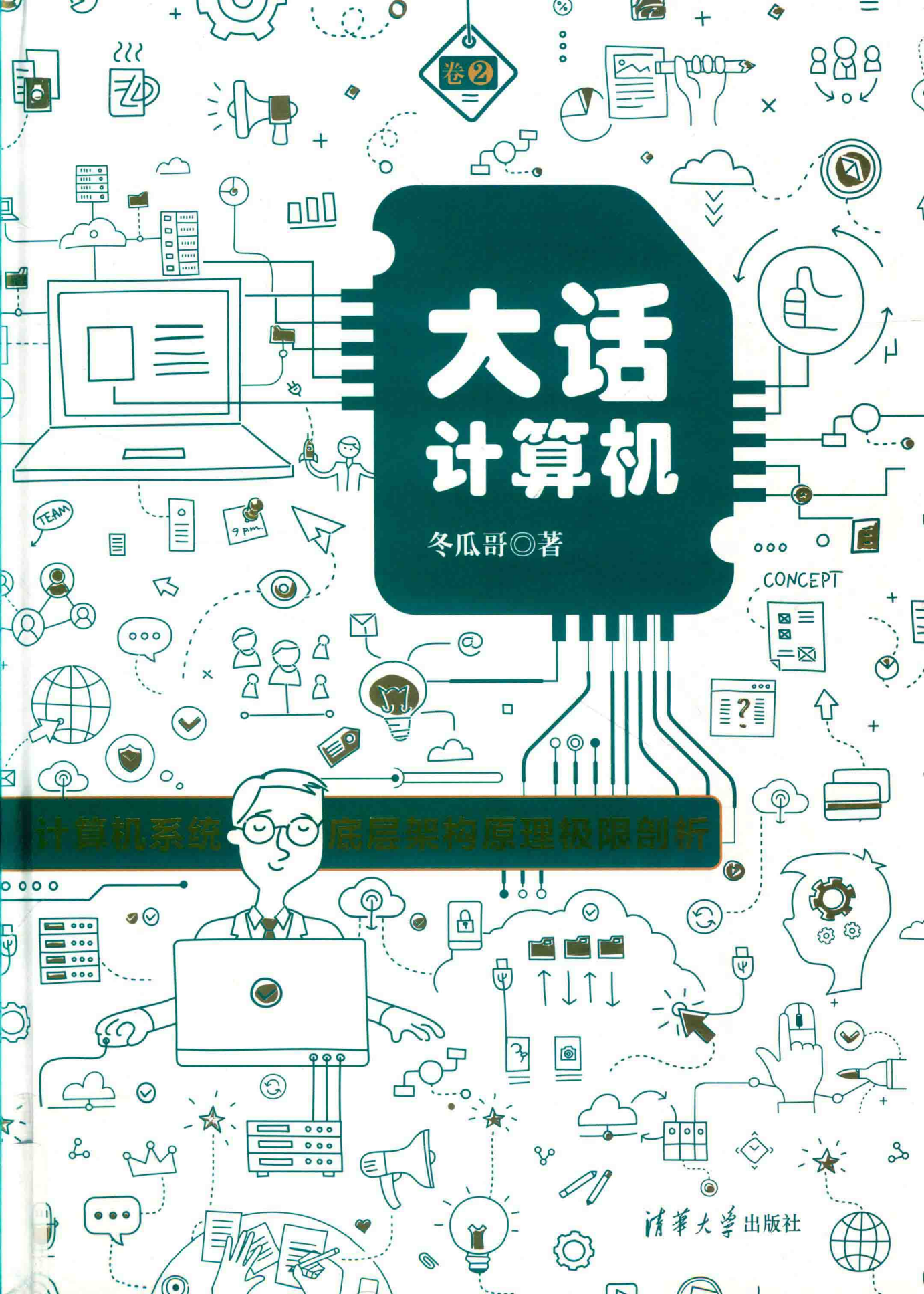
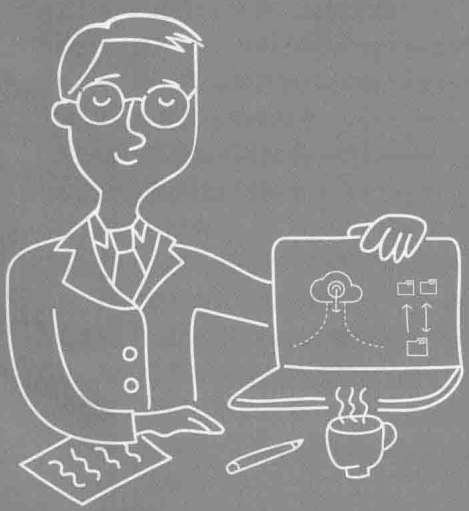

卷②
大话
计算机
冬瓜哥◎著
计算机系统底层架构原理极限剖析
清華大學出版社

大话计算机

计算机系统

底层架构原理极限剖析

冬瓜哥◎著



清華大學出版社
北京

内 容 简 介

现代计算机系统的软硬件架构十分复杂，是所有IT相关技术的根源。本书尝试从原始的零认知状态开始，逐步从最基础的数字电路一直介绍到计算机操作系统以及人工智能。本书用通俗的语言、恰到好处的疑问、符合原生态认知思维的切入点，来帮助读者洞悉整个计算机底层世界。本书在写作上遵循“先介绍原因，后思考，然后介绍解决方案，最终提炼抽象成概念”的原则。全书脉络清晰，带领读者重走作者的认知之路。本书集科普、专业为一体，用通俗详尽的语言、图表、模型来描述专业知识。

本书内容涵盖以下学科领域：计算机体系结构、计算机组成原理、计算机操作系统原理、计算机图形学、高性能计算机集群、计算加速、计算机存储系统、计算机网络、机器学习等。

本书共分为12章。第1章介绍数字计算机的设计思路，制作一个按键计算器，在这个过程中逐步理解数字计算机底层原理。第2章在第1章的基础上，改造按键计算器，实现能够按照编好的程序自动计算，并介绍对应的处理器内部架构概念。第3章介绍电子计算机的发展史，包括芯片制造等内容。第4章介绍流水线相关知识，包括流水线、分支预测、乱序执行、超标量等内容。第5章介绍计算机程序架构，理解单个、多个程序如何在处理器上编译、链接并最终运行的过程。第6章介绍缓存以及多处理器并行执行系统的体系结构，包括互联架构、缓存一致性架构的原理和实现。第7章介绍计算机I/O基本原理，包括PCIE、USB、SAS三大I/O体系。第8章介绍计算机是如何处理声音和图像的，包括3D渲染和图形加速原理架构和实现。第9章介绍大规模并行计算、超级计算机原理和架构，以及可编程逻辑器件（如FPGA等）的原理和架构。第10章介绍现代计算机操作系统基本原理和架构，包括内存管理、任务调度、中断管理、时间管理等架构原理。第11章介绍现代计算机形态和生态体系，包括计算、网络、存储方面的实际计算机产品和生态。第12章介绍机器学习和人工智能底层原理和架构实现。

本书适合所有IT行业从业者阅读，包括计算机（PC/服务器/手机/嵌入式）软硬件及云计算/大数据/人工智能等领域的研发、架构师、项目经理、产品经理、销售、售前。本书也同样适合广大高中生科普之用，另外计算机相关专业本科生、硕士生、博士生同样可以从本书中获取与课程教材截然不同的丰富营养。

图书在版编目(CIP)数据

大话计算机：计算机系统底层架构原理极限剖析 / 冬瓜哥著. — 北京：清华大学出版社，2019（2019.6重印）
ISBN 978-7-302-52647-6

Ⅰ.①大… Ⅱ.①冬… Ⅲ.①计算机系统—基本知识 Ⅳ.①TP303

中国版本图书馆CIP数据核字（2019）第045444号

责任编辑： 栾大成
封面设计： 杨玉芳
版式设计： 方加青
责任校对： 徐俊伟
责任印制： 杨 艳

出版发行： 清华大学出版社
网 址： http://www.tup.com.cn，http://www.wqbook.com
地 址： 北京清华大学学研大厦A座 **邮 编：** 100084
社 总 机： 010-62770175 **邮 购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者： 北京亿浓世纪彩色印刷有限公司
经 销： 全国新华书店
开 本： 188mm×260mm **印 张：** 96.25 **字 数：** 3546千字
（附海报11张）
版 次： 2019年5月第1版 **印 次：** 2019年6月第2次印刷
定 价： 598.00元（全三册）

产品编号：082577-01

目 录

第6章 多处理器微体系结构——多核心与缓存

第7章 计算机I/O子系统

第8章 绘声绘色——计算机如何处理声音和图像

第9章 万箭齐发——加速计算与超级计算机

我们在第4章中介绍过流水线、多发射、超标量等概念和对应的示意设计。这些技术无一不是为了将指令并行地执行而设计的。那时候，咱们还不知道有“线程”这个概念，还是假设CPU上只有一个线程在运行，这个线路穿起来一堆的代码，利用流水线、多发射，将该线路中的这一堆代码三三两两地并行执行，并在ROB中重新排序，最后提交到数据寄存器。可以在同一个时刻并行发射多条指令的前提是：指令之间要做好去除部件冲突导致的伪相关（除了RAW之外的相关），比如利用寄存器重命名机制解决寄存器争抢冲突、放置多个计算ALU/FPU以解决对计算单元的争抢冲突导致的排队串行化。

6.1 从超线程到多核心

而现在我们知道了，内存中的一堆代码，可能有多个执行线路，每个线路是一个线程，每个线程轮流被调度到流水线上执行。也就是说，流水线上正在执行的指令总是某个线程内的一堆指令。我们在第5章中也说过，这种高速切换线程执行所产生的效果，可以给人脑一种错觉，认为所有线程在被“同时”执行。然而，能不能让多个线程真的在同一时刻都并行执行呢？

6.1.1 超线程并行

试想一下，线程是什么。线程除了是一堆被串起来的代码之外，它还有：入口地址、上一次被打断时执行到哪里了（断点地址）、其运行时的栈帧被放在哪里了（栈顶和栈底指针）、断点时各个数据寄存器的值、它的代码和数据（放在内存里）、线程的虚拟地址页表基地址，等等。只要知道这些信息，起码就可以把这个线程继续运行起来。线程运行的时候，需要什么资源呢？需要一个地址指挥棒（PC指针不断自增），需要流水线中的译码、执行、写回等硬件模块，需要内存来存放它的代码和数据，需要各个数据寄存器。那么，现在需要用一条流水线、一套寄存器来同时运行多个线程的代码的话，也就是说线程调度器要让CPU同一时刻将多个线程的指令混杂地载入流水线执行，就需要为这些线程提供好这些资源。流水线可以大家一起同时用，因为流水线是分了好几个级的，可以同时容纳多条指令的不同执行阶段，那就可以让多个线程的指令被同时载入流水线，只不过位于不同级，做到真的同时；那么，PC指针寄存器可以共用么？显然不行，怎么可能在一个寄存器中同时放置多个线程的下一条指令的地址呢。所以，需要新设立对应数量的PC指针寄存器，各自存储各自所归属线程的下一条指令的地址；同理，栈指针寄存器也不可以公用，也得设立多份；再者，单个可见数据寄存器无法同时存多个数，但是咱们之前不是设计了一堆的物理寄存器么？利用寄存器重命名，就可以同时存储多个线程指令中的运算中间值，做到同时。页表基地址寄存器也不可能同时保存多个基地址指针，所以也得给每个线程设立一份。最后剩下内存了，内存可以公用么？当然，多个线程的代码和数据本来就被同时存储在内存的不同位置，互不冲突。实际上，需要复制出来多套的东西还有很多，图6-1所示为Intel x86 CPU

The following features are duplicated for each logical processor:

- General purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP)
- Segment registers (CS, DS, SS, ES, FS, and GS)
- EFLAGS and EIP registers. Note that the CS and EIP/RIP registers for each logical processor point to the instruction stream for the thread being executed by the logical processor.
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM7) and the MXCSR register
- Control registers and system table pointer registers (GDTR, LDTR, IDTR, task register)
- Debug registers (DR0, DR1, DR2, DR3, DR6, DR7) and the debug control MSRs
- Machine check global status (IA32_MCG_STATUS) and machine check capability (IA32_MCG_CAP) MSRs
- Thermal clock modulation and ACPI Power management control MSRs
- Time stamp counter MSRs
- Most of the other MSR registers, including the page attribute table (PAT). See the exceptions below.
- Local APIC registers.
- Additional general purpose registers (R8-R15), XMM registers (XMM8-XMM15), control register, IA32_EFER on Intel 64 processors.

图6-1 Intel x86 CPU超线程所需复制的硬件模块一览

超线程所需复制的硬件模块一览。

经过这样设计，不同线程拥有各自的PC寄存器、栈指针寄存器、物理寄存器（这些寄存器统称为线程上下文寄存器，Thread Context Register）、内存中数据。各自的PC寄存器被各自的指令引导着，要么自增，要么跳转，各增各的，各跳各的，互不干扰。至此好像资源都准备好了，可以执行了么？还有个关键的地方。假设第一个线程中的一条Divide A B C指令，和第二个线程中的一条Add C B A指令同时被载入了流水线，你说这两条指令冲突么？看似是发生了RAW冲突，Add需要利用Divide指令的结果。但是它俩实际上根本就是毫无关系的两条指令，因为这完全是两个线程，两条路线，还记得上一章中的“你的一天”和“猪的一天”这两个线程么？虽然小猪也要吃饭，你也要吃饭，比如其中某一步使用了相同的指令，都要吃，但是该指令操作的数据可是完全不同的，也就是你和小猪吃的饭是不同的，这个指令之前、之后的逻辑也很有可能是不同的，也就是上下文是不同的。如果底层不加以区分，把小猪要吃的东西作为你的指令中的操作数怎么办？所以，这里多个线程的代码只是公用了寄存器而已，属于一种部件访问冲突（结构相关），是可以通过新设一套对应的部件来解决的，但是这里需要用同一条流水线来节省资源，那就得想其他办法解决。必须要让流水线控制部件感知到这两条指令是不同路线里的，毫不相关。可以在保留站中增加一列，记录每一条指令到底是属于哪个线程的（线程ID），这个ID可以由硬件自动加入，对程序代码完全透明，比如，第一路PC寄存器取进来的指令属于线程ID1，第二个PC寄存器取进来的指令属于线程ID2。这样就可以让发射控制单元区分出来了，只要将这两条指令的目标操作数映射到物理寄存器中就可以解决冲突，就可以并行执行。这里有多种设计思路，比如可以在物理上将寄存器分隔开，如每个线程分配16个物理寄存器，共支持4线程，一共有64个物理寄存器；也可以让所有线程混杂使用这64个寄存器，使用单独的ID来记录哪个寄存器目前被哪个线程所使用。

现在思考一个引申问题，多个线程的指令到底应该怎么被塞入流水线？你一条我一条他一条，一人一条地来交织（Interleave），还是两条两条地交织？还是按照某种更小的时间窗口来交织，比如你执行1ms，我执行1ms，他再执行1ms？根据不同的设计思路，实际中的设计各有不同。比如有些设计采用事件触发的形式，如线程1发出了一个Load指令，结果没有命中缓存，需要访问SDRAM，此时可以立即决定切换到线程2执行，因为访问SDRAM需要很长时间，空等不划算。缓存不命中这个事件触发了线程切换。（注：这里所谓触发了线程切换并不是指触发了操作系统中的程序对线程进行切换，此时操作系统认为多个线程都处于运行过程中，但是CPU内部却同一时刻只在运行一个线程的代码。操作系统线程切换流程详见第10章。）

比如有些采用严格的单指令轮流交织，有些则动态地调整交织粒度，有时候一人一条，有时候你一条我两条他三条，不定。有些则根据时间窗来交织。有些则在不同的执行阶段采用不同策略，比如AMD在2017年推出的Ryzen锐龙系列CPU在指令队列中采用Round Robin轮流方式，在寄存器重命名阶段采用加权优先级方式调度，而在Load/Stor时采用静态分区的方式物理上把资源隔开给多个线程使用。可见，上述动态调整交织粒度的做法最为复杂，需要在硬件中记录更多状态和判断规则，比如CPU发现线程A的代码内部的RAW相关度太高，而且都是些耗时的RAW，如Divide A B C、Add C B A、Sub A B D、Mul D A B这几个指令，由于Divide指令耗时较长，后续这三条指令都卡在保留站里等待被发射，等待期间加减法ALU被闲置了，乘法单元也被限制了。如果这类指令太多，很快就会塞满保留站，导致流水线整体停顿。与其在这干耗着，不如择机载入一段线程B的代码执行着，把流水线中的资源利用起来。而根据时间窗来交织，实现最为简单，因为不会出现上文中说的那种两条分属不同线程的指令需要被区分的问题。

如果按照时间窗来交织，也就是让CPU一段时间内载入线程A的代码执行，然后切换到线程B执行一段时间，这段时间比如可以是几毫秒。这样就可以保证多个线程的代码不会相互无序掺杂在一起，也就不会出现上面那个两条看似相关的指令分属不同线程，其本质其实不相关，必须增加线程ID而且在发射控制单元中增加判断逻辑的问题。

咦？在第5章中我们不是已经介绍过利用时钟中断强行切换线程的思路和设计了么？这不就是时间窗交织么？靠线程调度器来切换不就可以了么？但是，线程内部代码之间的相关度如何，以及代码能否能够充分利用流水线，这两个信息线程调度器是根本无法发现的，这些只能靠CPU的硬件电路模块在运行时动态地判断出来，所以必须由CPU决定运行A线程多长时间，然后切换到B线程运行多长时间。既然这样，可以这样设计，只要CPU决定切换线程，就自动将自己中断，就像发生了一个时钟中断一样，然后在保护现场之后，跳转到时钟中断服务程序执行，最后执行到线程调度这一步，就可以调度其他线程执行了。但是这样做不太理想，上一章我们看到了，产生一个中断的代价是很高的，能不能不用中断，而在CPU决定载入其他线程代码时直接载入呢？因为我们已经给其他线程准备好它所需要的资源了（PC寄存器、栈指针寄存器等），这样可以省掉现场保护、载入新线程的相关寄存器这些步骤。既然这样，线程调度器就要被设计为一次调度多个线程到CPU上执行，然后CPU硬件决定某段时间内执行哪个就执行哪个。时钟中断到来后，线程调度器再调度另外的多个线程（或者可能

还是这几个，或者换掉其中某个/几个）到CPU上执行。这样的话，时钟中断之后的处理方法保持不变，变化的则是每次需要调度多个而不是只有一个线程到CPU上，把每个被调度线程的PC寄存器、栈寄存器的值安装到前文中所述的每一份线程的对应寄存器中即可，这些寄存器在线程运行过程中不断各自变化。在两次时钟中断的时间窗内（比如10ms），CPU内部的电路可能会在这几个线程之间来回切换，多次将它们的指令载入流水线，比如每1ms切换一个线程，这样可以让多线程的切换粒度更细腻，还避免了中断导致的开销。这样不但能够让用户体验更连续流畅，而且还能充分利用流水线资源。单指令粒度交织其实与时间窗交织本质上没有区别，单指令交织可以看作是最小时间窗交织。

而更加智能的动态交织做法就会比较复杂，但是效果也会更好。它的做法就是见缝插针，比如在执行线程A时，本来是执行到某处时不得不插入几条NOOP指令空泡，但现在CPU可以不插入空泡，而将线程B的几条指令塞入流水线，利用了这个空泡。时钟中断到来时，CPU也要同时将这多个线程的现场寄存器保护下来，再跳转到时钟中断服务程序执行。

这样设计看上去不错。那么，为了让线程调度器可以将多个线程的寄存器的值从内存中（断点时被保存到内存中的线程状态表中）载入到各自线程的PC、栈、页表基地址等寄存器，是不是要给每个线程的这些寄存器编个号，然后新设计一套专门用于多线程场景的指令，比如Jmp_PC0、Jmp_PC1（分别表示将断点地址载入0号、1号线程的PC寄存器），或者LPTBR0、LPTBR1（Load Page Table Baseaddres Register，分别表示载入0号、1号线程的页表基地址指针），然后修改线程调度器，使用这些新指令来实现多线程并行调度到CPU上执行。如果这样的话，实现就有点复杂了。人们其实用了一个更加绝妙的办法。

在收到时钟中断之后，大家都知道CPU要跳到时钟中断服务程序运行然后跳到线程调度器运行。现在不是有多套线程上下文状态寄存器了么？那就让这多个PC指针都跳转到同一个/同一份时钟中断服务程序，然后跳到线程调度器执行。什么？多个线路都在执行同一份代码？怎么，不可以么？记得冬瓜哥在前面章节就提到过，多个厨师可以看同一份菜谱步骤（代码），各自做出各自的菜（运算结果数据）。但是，前提是，菜谱里需要这么写：

#1. 打着火；

#2. 烧热油；

#3. 如果是厨师A在掌勺请跳到6号指令，如果不是则跳到4号指令；

#4. 如果是厨师B在掌勺请跳到7号指令，如果不是则跳到5号指令；

#5. 警报警报！无法识别的掌勺人！（产生错误消息并停止运行）；

#6. 放入葱花爆锅，放胡萝卜丝、木耳、肉丝、酱料爆炒出锅；

#7. 放入蒜蓉爆锅，放胡萝卜丝、木耳、肉丝、酱料爆炒出锅。

这是一份鱼香肉丝的代码，但是不同厨师执行的时候却走了不同的分支，输出了不同口味的数据，也就是发生了不同的结果。那么谁说不同的执行者（厨师）不能运行同一份代码？完全可以，即使代码中不加判断，充其量两个执行者做出了完全一样的菜肴罢了。看到这里你应该隐约就知道了，在线程调度器中，加入“判断当前是谁在执行”的代码。怎么得知这个信息？需要利用CPU指令集中的一条叫作“CPUID”（CPU Identity）的指令。CPU执行这条指令之后，会将自己的ID号以及最多支持多少个线程的并行执行等信息放到数据寄存器中，然后程序再用Stor指令将数据寄存器中的内容保存到内存中某处就可以拿到对应信息了。调度器程序根据最大支持的并行线程数（内部做了多少套线程上下文状态寄存器）以及当前是哪套线程上下文状态寄存器在驱动着电路执行的，从而将不同线程的上下文状态寄存器的上一次的断点值复制到当前的上下文状态寄存器中接续执行。

这样设计之后，Jmp、装载栈指针、装载页表指针等指令都可以保持不变，因为这些指令操纵的就是当前的硬件。比如1号PC寄存器对应的硬件执行了装载页表指针这条指令，那么其装载的页表指针一定是这样一个线程的页表指针：该线程由线程调度器指定将、将要被调度到由第一套线程上下文状态控制部件驱动的硬件上运行。同理第二套上下文状态控制寄存器也在做同样的事情，只不过它所载入的将会是另外线程的页表指针。这样就完成了调度任务。

线程调度器感知到的其实是多个虚拟CPU（可以让计算机操作员明确地看到系统中的逻辑CPU的个数，比如可以通过程序发出CPUID指令获取虚拟CPU的数量然后列出到屏幕上），或者说逻辑CPU，每套线程上下文状态寄存器和控制部件就是一个逻辑CPU，它们被中断后各自都跳到同一个地址入口，也就是线程调度程序入口，当执行到调度程序中的CPUID指令时，后续执行路径发生了差异，代码开始产生分支（因为每个逻辑CPU返回的信息都不一样），从而控制着每个逻辑CPU做了不同的事情（各自载入不同的线程执行）。Windows 10操作系统的任务管理器中就可以展示出物理CPU或者逻辑CPU的负载情况视图。

提示 ▶▶▶

那么系统具体是如何在初始时向这些逻辑CPU派发线程运行呢？现代操作系统一般会给每个逻辑CPU在内存中特定位置准备对应的线程队列，线程

调度模块把要在该CPU上执行的所有线程的对应的入口地址等信息写入到该队列里。系统启动时会有一个主核心负责初始化操作，祝贺信在初始化流程后期采用IPI（处理期间中断，详见第10章）来通告其他逻辑CPU也开始运行线程调度模块（线程调度模块的入口地址会被包含在IPI中断消息中传递给对方CPU），线程调度模块根据当前CPU的ID决定到不同的运行队列中调度对应的线程运行。

至此总结一下。这种用同一条流水线、同一套数据寄存器，但是每个线程各自对应一套上下文状态寄存器，然后利用不同的交织粒度同时运行多个线程指令的设计，称为超线程（Hyper Threading）。如果交织的粒度比较粗，比如基于固定时间窗或者一定数量的指令或者某些事件（比如缓存不命中）触发为一个切换粒度的话，被称为粗粒度超线程；如果时间窗较小，比如达到了单条指令为切换粒度，这被称为细粒度超线程。

在早期的超线程设计中，流水线的各个执行级中只处理同一个线程的指令，但是不同级可以处理不同线程的指令，比如取指令单元在某个时钟周期内只从某个线程取回一定量的指令（利用该线程的PC寄存器），与此同时，译码级电路可能正在译码另外一个线程的指令；而在下一个时钟周期，取指令单元切换到使用另一个线程的PC寄存器来取指令，与此同时译码单元则开始译码上一步取回的上一个线程的指令。后来Intel公司做了改进，在取指令级上，每个周期切换一次线程的PC寄存器取出对应指令，进入译码阶段，译码阶段每个周期也就跟着分别译码某个线程的指令，达到细粒度超线程级别。然而，在所有指令进入保留站之后，在发射阶段，则允许同时发射多个线程的指令到运算单元执行，而不是像传统细粒度超线程那样发射阶段必须发射同一个线程的指令。这种改进之后的超线程技术被称为同时多线程（SMT，Simultaneous Multi Threading）。SMT技术是真正意义上的“并行”，也就是同一时刻多个线程的代码齐头并进地被发射执行，其他超线程技术本质上都是超细时间粒度的切换线程而已。

可以看到，在使用流水线、多发射等技术实现了单个线程内部多个指令可以并行执行之后，利用超线程技术实现了多个线程的指令流级别的并行执行，前者被称为指令级并行（ILP，Instruction Level Parallel），后者则被称为线程级并行（TLP，Thread Level Parallel）。线程级并行的效果是用户可以感受到多个独立的任务同时执行而且切换比较流畅，而单线程内部的指令级并行，用户能感受到的效果则是单个任务的执行速度很快，响应延迟变低。当然，如果二者兼具，则同时提升了并发度和降低了延迟，体验会更好。

值得一提的是，分支预测单元也应当感知超线程的存在，也就是需要为超线程做对应的设计变更，它必须为每个线程单独预测分支，甚至可以准备单独的BTB、PHT等，并更新各自线程的PC寄存器为预测出来的目标地址，而不能把多个线程的指令混杂起来进行预测，这样就乱套了，预测正确率将会大大降低。

超线程对流水线上的执行部件是多线程争用的，BTB、保留站、各种缓冲队列等都需要同时存放多个线程的指令/数据，有些设计完全采用物理平均分割的方式来分割上述这些资源，比如，保留站总体上有256项，打开x2超线程之后，每个线程只能利用128项，超出就不能使用了。这样做硬件实现最为简单但是也很不灵活，比如假设当前只有1个线程在执行，线程2发生了Cache Miss正在等待返回数据，那么线程1也只能利用128项，如果程序的行为导致经常发生这种情况，那么此时可以关闭超线程，则单个线程就可以利用全部资源了，总体性能反而能够提升。有些超线程设计是动态分配资源的，用多少占多少，某线程暂时不用则另一个可以全用满，这样更合理，性能也更好。有些CPU设计里，对资源A对半分，对资源B又动态分，这导致更难以判断超线程到底是否会增益整体性能。

注意 ▶▶

超线程并不是任何时候都可以提升性能的，如果同一条流水线上的多个线程运行时都由于指令的RAW相关或者由于访存等因素导致流水线阻塞的话，此时就只能怪运气差了。

6.1.2　多核心/多CPU并行

我想，早在大家阅读第4章的时候，一定有不少人会产生这个想法：为什么不能在CPU里做上多条流水线、多套寄存器、多套PC/栈/页表基地址寄存器呢？这本质上不就等价于多个CPU了么？是的，早期的高端计算机就是使用多个独立的CPU芯片共同连接到SDRAM上的，但是随着芯片制造工艺的不断提升，上面这些模块其实完全可以做成多套并被放到同一个芯片内部，不需要多个独立芯片。同一个芯片内部每一套上述这堆部件，被称为该CPU芯片内部的一个核心（Core）。多个线程之前住的是合租房（公用流水线和数据寄存器），现在终于有了自己的全配套设施单间了，真的可以完全各干各地并行执行了。线程被调度到这些核心上的过程与超线程是一样的，每个核心都是一个执行者，线程调度器同样会感知到多个逻辑CPU，只不过此时每个逻辑CPU是全配套设施的，而不是有些设施公用的。实际上，线程调度器等软件程序除非使用CPUID等控制类指令来获取深层次信息，否则也无法感知某个逻辑CPU到底是被CPU内部超线程模块虚拟出来的，还是物理上独立的核心。

现在实际的产品中普遍都在核心中保留了超线

程，比如Intel x86 CPU每个核心支持2个超线程，而IBM Power8 CPU每个核心支持8个超线程。这样，一个8核心Power8 CPU对外其实体现为16个逻辑CPU，每个逻辑CPU每个时刻可以执行一个线程，如果两个逻辑CPU落在同一个核心，那么这两个逻辑CPU所执行的两个线程会共用流水线。至于具体是如何分配流水线资源的，上文中也描述过那些交织粒度，在此不再赘述。

在单个核心中保留超线程的原因是因为单个线程无法100%利用流水线，比如当有大范围连续发生的RAW相关时，或者缓存不命中导致花费大量时间去访问RAM时，会导致流水线大范围空泡，此时如果有另一个线路中的指令能够顶上来占用这些时隙，就可以充分利用资源。超线程其实是一举两得的事情，既充分利用了流水线资源，又能让多线程以更小的时间窗齐头并进，提升用户体验。

注意

值得一提的是，超线程被发明的时候，其初衷并不是为了让线程更加细粒度地并行提升用户体验，因为靠时钟中断一样可以做到线程看上去并行，而纯粹是看到流水线没有被单线程完全利用觉得可惜，所以强行插入其他线程。

此外，多个多核心CPU也可以同时运行多个线程，比如4个8核心2超线程的CPU，总共的逻辑CPU数量为64，那么这个系统在同一时刻可以同时执行64个线程，其中有一半数量的线程并行得不是那么充分，因为它们使用的是公用设施。可以用特殊指令来关闭超线程，这样CPU内部就不会使能其他线程的状态寄存器了，只保留一套。

但是在现代CPU和现代的多数程序场景下，超线程对性能的提升并不是很明显，因为现代CPU在乱序执行、分支预测等方面的设计优化几乎做到了极致，流水线空闲的概率较低，在这种场景下多个线程就不是共同利用流水线了，而是共同争抢流水线，反而可能产生一些开销，比如第一个线程运行得好好的，没有空泡，结果非要调度第二个线程来乱入第一个线程，这样的话，配套的分支预测、乱序执行顺序提交等模块就要切换各自的上下文状态到第二个线程。为什么非要调度第二个线程呢？如果CPU看到第一个线程充分利用了流水线，就让它一直执行下去不就好了么？不可以，那此时其他线程不就被饿死了么，永远得不到执行。既然你选择了使用超线程，就会有线程被调度上去，你又不让人家上台执行，最后那些线程就会卡住，影响用户体验。

图6-2描述了从最原始的核心演变到超线程核心再到双物理核心的过程。

6.1.3 idle线程

现在思考一个问题。我们在前面章节中说到过，CPU必须运行点什么，就像汽车发动机总得转着一样，一旦熄火，就得手动打火；同理，如果你不想让CPU运行了，那就发送一条poweroff指令给CPU，CPU在做完一些善后工作之后，内部的供电控制电路直接切断自己的供电，以后就再也不会靠自己运行起来了，必须重新手动再按一下电源按钮。这也是为什么在上一章中需要让计算机运行一个无限循环的Loader程序（Shell）的原因之一，让计算机等待命令输入。

现在，我们的CPU内部出现了多个逻辑核心（超线程生成），以及多个物理核心，相当于之前的单缸发动机变成了双缸、四缸、六缸发动机，那就必

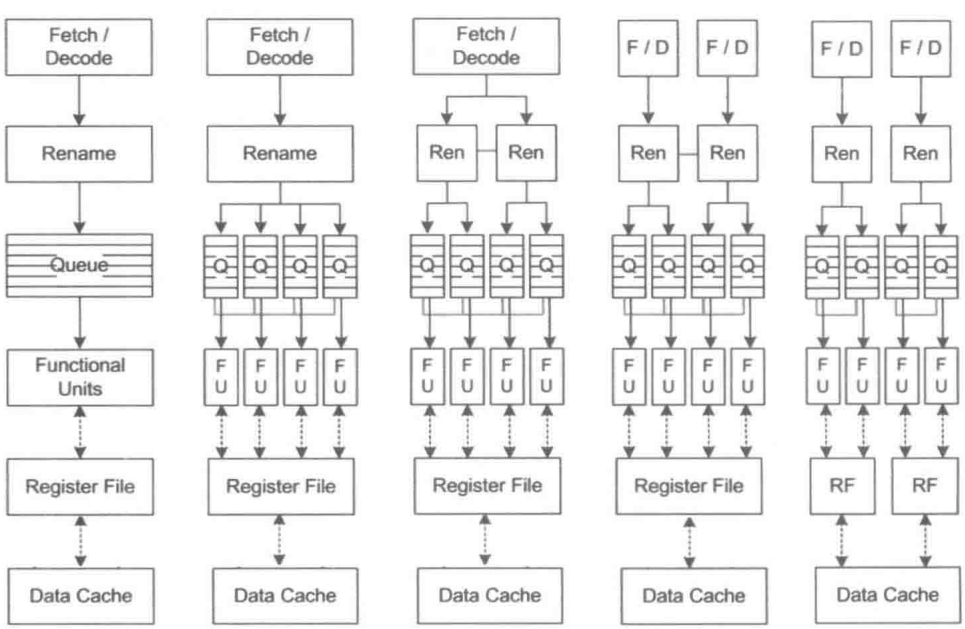


图6-2 从最原始的核心演变到超线程核心再到双物理核心的过程

须给这些逻辑CPU的每一个都喂上一个线程，因为它们一跑起来就停不下来了。那么，假设刚加电启动到Loader之后，还没有任何程序需要执行，唯一的一个Loader线程被喂给了其中一个逻辑CPU，其他逻辑CPU怎么办呢？是否可以让其他的每个逻辑CPU都载入Loader运行呢？可以，但是毫无意义，而且会得到莫名其妙的结果，比如多个CPU的每一个都在向屏幕输出同样的信息，而且可能因为速度不同导致闪屏，除非有多个屏幕，不同CPU输出到不同屏幕。可以这样，调度一个大循环while(1){noop;}不断发送NOOP指令的线程给这些CPU，让它们不断地原地空转，什么也不输出，这样就好了。可以，这些逻辑CPU执行NOOP指令也是需要消耗电的，电子还在电路中拉锯发热。最好的办法是让这些无事可做的逻辑CPU直接进入节能状态，而不是循环执行NOOP哇哇哭叫“NOOP，N~O~O~P”。所以，需要开发一个CPU“安抚奶嘴”，只要CPU一执行这个线程，就立即进入节能状态，比如将电路主模块的时钟直接脱挡（Clock Gating）。这就需要CPU的指令集中提供一条特殊指令，比如Intel CPU里使用MWAIT（Monitor Wait）指令让CPU静默，将一些电路模块的时钟脱挡。这个线程叫作idle线程，也就是在Windows任务管理器中看到的那个“System Idle Process”，其内部的主要逻辑就是发出让CPU进入节能态的指令。

思考一个问题，如果一个物理核心上虚拟出两个超线程逻辑CPU，其中一个逻辑CPU被灌输了MWAIT指令，执行后是不是可能会将整个物理核心的时钟脱挡了？这显然不行。所以物理核心内部的超线程电路部分会保证只将该逻辑CPU自己的那部分电路进入节能态，比如每线程一套的线程上下文寄存器等。

再思考一个问题，执行了MWAIT的逻辑CPU，如果后续想让它醒过来执行其他线程的话怎么办呢？是不是还得有一条Wakeup指令？问题是，这个逻辑CPU已经被静默了，它不会去取指令执行，甚至连时钟中断都不会响应，什么都听不见了，如果它能取指令就证明它根本没睡。所以，需要一种强制唤醒机制。为此，Intel CPU是这样设计的：其CPU内部增加了一个专门用于监控特定内存地址写入动作的硬件电路模块，在执行MWAIT指令之前，线程必须先用一条MONITOR指令将需要监控的内存地址范围（放在寄存器中作为指令的操作数）告诉这个电路模块，比如“我要监控内存地址0～127”，然后再执行MWAIT指令让CPU睡觉，当然，这个监控电路模块不能睡，一直在监控地址0～127字节是否有人写入，一旦有人写入，则该模块将强制叫醒该逻辑CPU接续执行排在MWAIT指令之后的指令。在其他逻辑CPU上运行的线程调度程序（还记得线程调度器是怎么得到执行机会的么？CPU被时钟中断强制中断之后可以进入，或者其他正在运行的线程调用它）可以将“是否有线程需要执行”的信息写入到0～127地址中的某处，这样就可以唤醒之前被睡眠的其他逻辑CPU，被唤醒的CPU继续执行MWAIT之后的指令（比如可以跳转到线程调度器执行，从而调度新的线程到该逻辑CPU执行）。

如果有多个逻辑CPU都在睡觉，它们都被设置为监控0～127字节的写入，那么这些逻辑CPU都会被唤醒，各自继续执行MWAIT的下一条代码。这条代码依然处于idle线程内，在这里，idle线程调用线程调度器函数，比如类似的逻辑idle () {MONITOR 0~127; MWAIT; Scheduler();}。调度器执行时，便可以调度其他线程到该逻辑CPU了，或者继续调度idle线程给该CPU。

线程调度器可以进一步判断，如果长时间没有多余的线程可运行，那就进入深度节能状态，比如向CPU发送Halt指令，直接关闭主要模块的电源（Power Gating），而不是只将时钟脱挡——时钟脱挡只是让电子不再拉锯振荡，但是半导体管中依然会有漏电流存在，而关闭电源之后，漏电流就消失了，功耗进一步降低。当然，每个逻辑CPU执行Halt时也只是关掉自己的那部分电路，不会把整个物理核心都关掉。

6.1.4　乱序执行还是SMT?

乱序执行纵使可以尽量让流水线保持忙碌，但是其代价是功耗会变得较高。相比顺序执行（In-Order）的CPU核心而言，支持乱序执行的核心又被称为乱序执行（Out-of-Order）核心。乱序执行核心即使在单线程场景下也可以较好地利用流水线资源，最终结果是单线程执行速度较快。而有了超线程技术之后，多个线程的指令混合交织，天然就消除了RAW相关，就算不重排序，顺着执行，流水线资源也能够较好地利用。那么，此时是不是可以抛弃乱序执行，而重归顺序执行呢？因为顺序执行的控制部件简单，功耗会降低不少。

有些业界的学者对顺序执行核心+SMT的思路做了一些研究和仿真，结果证明效果不错。比如Khubaib团队设计的MorphCore（可变身的核心）。他们发现，不管是使用乱序执行核心还是顺序执行核心，在线程达到8个的时候，乱序和顺序执行核心达到了相同的单线程平均性能。如图6-3所示，在单线程时，顺序执行核心的单线程性能显然较乱序执行核心差了一大截；但是随着线程数量的增加，顺序执行核心内部的RAW相关越来越少，逐渐达到了与乱序执行核心相同的单线程平均性能。

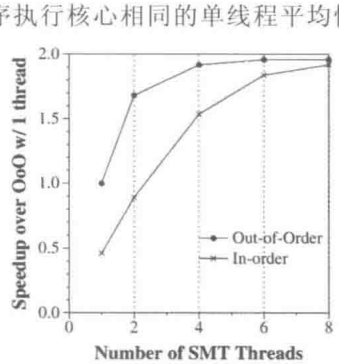


图6-3　顺序和乱序执行内核性能对比

为了同时兼顾单线程和多线程，MorphCore的核心设计思路是在单线程运行时依然采用乱序执行模式，而当被调度的线程达到了一定数量之后，自动切换为顺序执行模式，这就是其“变身”的含义。模式切换时对包括线程调度器在内的所有线程保持完全透明。线程调度器感知到的永远都是8个逻辑CPU核心，当线程调度器分配了1或者2个线程给某个核心时（向该核心虚拟的8个逻辑CPU中的任意1或者2个逻辑CPU），该核心采用乱序执行模式；当调度的线程大于2个时，该核心自动切换为顺序执行模式。只要线程调度器将某个线程写入了线程上下文寄存器，而且这个线程没有运行MWAIT指令，就证明该线程不是idle线程，而是用户线程，电路是可以识别出来的；如果调度器将某个线程用MWAIT指令暂停了，电路也可以检测到，从而知道该线程被静默了，当运行着的线程降低到2个及以下时，电路切换到乱序执行模式。电路的切换过程是将当前所有运行着的线程的上下文寄存器数据全部保存到RAM中某处固定区域，然后内部切换各个MUX/DEMUX路径，准备好之后，再将之前保存的上下文重新装回电路中，从而继续执行。

在顺序执行模式下，保留站、物理寄存器等公用部件会被切分为多份，每个线程一份，大小固定，这样可以简化硬件设计。MorphCore的流水线模块分了两套，一套专供乱序执行，另一套则用于顺序执行，然后根据当前所处的模式，用Mux和Demux来选择使用哪一套电路来执行，利用门控时钟来将另一套电路模块时钟脱挡以节能。当然，有一些电路模块是两个模式公用的。比如，取指令单元的架构示意图如图6-4所示，可以看到8个线程的PC寄存器被输入到一个MUX。指令存储器（i-cache）公用一份，取出的指令缓冲则是每个线程一份独立的。

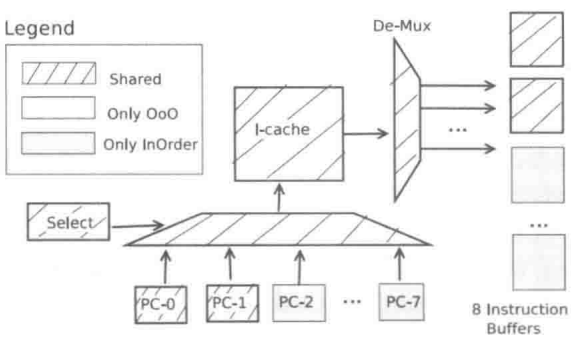


图6-4 MorphCore中的取指令单元示意图

当然，至于是不是各种场景下该理论都适用，还需要经过长时间的检验。

6.1.5 逆超线程?

现在的商用CPU的核心数量越来越多，要想发挥出多核心的并发性，就得在程序设计上考虑尽量切分为多个线程，而且还得保证每个线程都得有活干，而不是滥竽充数，因为线程是可以主动让出CPU的（本章其他节会详细介绍线程的各种状态和运行机制）。如果有太多不干活的线程和少数累到死的线程，多核心的性能就得不到提升。但是有些程序的确难以切分为多线程，其本身就是没有并发性的，对于这种程序，单核心的性能就非常关键了。但是提升单核心的性能，势必会占用更多的硬件资源，在相同面积的芯片上，核心数量也就做不上去，不利于多线程程序。为了兼顾这两种场景，人们提出了一种设想：CPU能否做到动态适配，在运行多线程场景时，采用每个核心运行一个线程的方式，而当运行单线程时，大量的核心是否可以同时运行该单个线程中的不同部分，然后将结果汇总提交？这个突发奇想被称为逆超线程，或者核心融合（core fusion），其基本思想也是动态地适配应用场景。

逆超线程与执行调度部件调度同一个线程中无关联的指令同时并行载入多个执行部件并行执行并无本质区别，只不过此时不但要将代码调度到同一个核心的多个执行部件，还要调度到不同核心的执行部件。可想而知，不同的核心相隔得实在是太远了，它们连L1缓存都不共享，这么做势必难度很大。试想一种设计思路：多个核心的取指令单元预先定义好，核心1从该线程的第1条指令开始，取8条指令执行，然后核心2取从第9条指令开始的8条指令载入执行，以此类推。假设共4个核心，每核心取8条指令执行，大家各自执行，但是结果必须提交到一个统一的Reorder Buffer以供判断哪个核心上的哪条指令依赖哪个核心上哪条指令的结果，也就是跨核心RAW相关。还有，假设核心1执行了指令Add_i 1 A B（将1和寄存器A中的数据相加写入寄存器B），而核心2执行了指令Add_i B A C，那么此时，核心1必须将寄存器B中的结果传送给核心2。如果是在同一个核心内部出现这种RAW相关，则可以采用前递方式节省一个时钟周期，但是跨核心前递就不太好实现了。目前在一些论文中普遍采用单独的硬件模块来将两个核心中的资源做整合，比如各种buffer、Register File，当逆超线程模式开启时，这些硬件模块将所有资源全局统一地管理和分配。现阶段要想实现逆超线程，在工程化设计上还比较难，实际使用效果上暂时还体现不出绝对优势。

6.1.6 线程与进程

现在回来思考一个问题。我们在上一章中提到过，可以把音乐播放器这个大程序拆分成多个小线程，各自独立运行。比如其中的三个线程：从磁盘读出音频文件放到内存、对音频文件进行解码、将解码后的数据发送给声卡进行发声。在上一章中我们也描述过，为了防止程序之间的相互影响，我们将物理内存虚拟成了多个独立的地址空间，然后用页表来追踪每个线程的虚拟地址到物理地址的映射，把每个程序关在了罩子里运行。既然如此，上述的读数据线程读

出的数据，又怎么会被解码线程看到呢？

显然，这套机制需要改，改成这样：让需要相互传递数据的多个线程运行在同一个虚拟地址空间中，运行在同一个罩子里就可以解决了，但是这些线程之间会相互受到各自bug的影响。人们把运行在同一个地址空间中的所有线程称为一个进程（Process），比如上述的音乐播放器整体上可以被设计为一个进程，在进程中包含多个线程。每个进程有各自独立的虚拟地址空间，进程中的所有线程共享同一个虚拟地址空间。图6-5为一个假想中的多线程进程在虚拟内存地址空间中的布局示意图，实际中不同的设计各有不同考虑。

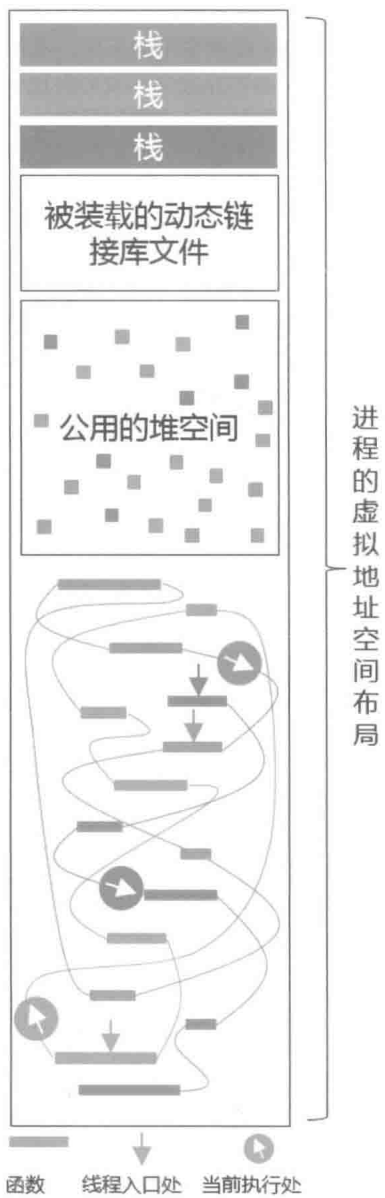


图6-5　多线程示意图

那么，多线程程序是怎么被系统运行起来的呢？可能会很自然地想到这种设计思路：在ELF/PE格式可执行文件的头部增加一个针对多线程的特殊控制字段，用于描述该可执行文件内部共有多少个执行线路，以及每个线路的入口函数所在的偏移量和参数，装载器可以根据这些信息直接将这些线程载入到同一个或者不同的CPU核心上运行。这样会非常不灵活，比如很多场景需要根据程序运行之后的各种条件动态创建线程，如某个网络服务程序，根据网络访问量动态地创建越来越多的线程；有些场景则需要动态地从网络、键盘等输入渠道来获取线程入口函数的参数，然后传递给线程入口函数。而上述这些动态的条件，在程序编写的时候是根本无法获知的，所以将线程数量和参数写死在可执行文件中理论上是可行的，但是却没有现实意义。

实际中，程序在运行的时候可以自行创建新的执行线路然后立即被调度执行，但是由于创建新线程势必要更新底层的线程调度管理方面的各种系统表，上一章中提到过，对这些系统级关键数据结构的管理和更新必须由内核级程序来负责，这些内核级程序通过系统调用的方式来被调用。所以为了方便统一起见，可以将这些底层的调用封装起来，形成更加易用的创建和管理线程的函数，比如起名为create_thread ()函数。对于线程管理和调度方面会在后续章节中详细介绍。

6.1.7　多核心访存基本拓扑

所谓“CPU核心”，我想本书进行到这一步，也该给大家一幅全景图了。在第2章中，我们介绍了一个简易CPU的基本组成模块：取指令单元、译码单元、运算单元等驱动着数据向前流动同时做运算的部件；以及主内存、指令存储器、数据存储器、数据寄存器等存储/暂存数据的地方。到了第4章，我们又向大家介绍了流水线以及额外追加的更加复杂的控制部件，包括：流水线阻塞控制单元、分支预测控制单元、寄存器重命名控制单元、结果侦听单元、发射控制单元、提交控制单元，以及内部私有的物理寄存器、流水线中间寄存器/队列、BTB/BHT/PHT、RAT、保留站、ROB等用于存储流水线运行时关键的映射表、控制位等的特殊内部存储器。上一章结尾介绍了中断处理模块、虚拟内存管理模块MMU；本章上一节又向大家介绍了超线程控制相关模块、节能控制模块。一个CPU核心所包含的主要模块除了上述这些部件之外，还要把缓存及其控制模块也算进去。最终我们将核心抽象为如图6-6所示的样子，当然，实际的CPU核心中还包含更多其他模块，随着本书的逐步讲解，你会逐渐了解到，比如图中所示的页表项缓存（Translation Lookaside Buffer，TLB），其作用为缓存MMU查询过的页表项，从而加速虚拟-物理地址的转换过程。

图中橘色导线表示控制信号，比如告诉对方读还是写，以及其他各种用于传输控制、状态等的信号；蓝色导线表示地址信号，用于告诉对方读/写的是哪个

地址上的数据；绿色导线表示数据信号，用于向对方传送/从对方接收对应地址上的数据。

所以，每个模块之间的各种导线可以归成控制、地址、数据这三大类信号。每一类信号的导线又可以有多根，比如地址信号导线有32根（32位宽），则一共可以表示2^{32}个地址，如果每个地址上存储一字节数据，那么这32位的地址空间中可容纳4G字节的数据。控制信号的根数则由需要多少种控制信号而定；数据导线的位宽如果是64位，则每个时钟周期可以最大传输8字节数据（通过给出一个起始地址，再给出一个要读取数据的长度控制信号来实现）。

取指令单元只会读入数据，其发出的读请求信号会被MMU发送给一级指令缓存控制器，指令缓存控制器的处理逻辑比较简单，就是不断地从二级缓存中预读入当前正在执行的地址的后续地址上的内容即可，因为指令总是顺着读取执行的。当然，当发生跳转的时候，之前被读入的内容可能就会被作废。而Load/Stor单元（或简称L/S单元）既读又写，对于Stor指令，其存入的数据会被存入Stor Queue（或者叫Stor Buffer）中排队，然后异步写入到一级数据缓存中。往一级数据缓存中写入数据的过程还稍有些复杂，后文中会有描述。使用Stor Queue相当于在L/S单元与一级缓存之间又增加了一层缓冲，Stor指令存入的数据写入了Stor Queue，就被视作写入完成。

图6-6中可以看到一级缓存（L1 Cache），以及二级缓存（L2 Cache）。实际中可以有多层缓存，比如L3、L4。目前Intel的Xeon CPU有3级缓存，其中L1缓存和L2缓存是每个核心私有的，L3缓存则是所有核心共享的，详情待本章后面介绍。

各个控制器的运行频率可能不同，有快有慢，它们之间采用异步FIFO等方式来接收和发送请求。发送方将地址、控制、数据信号放置到导线上，接收端进

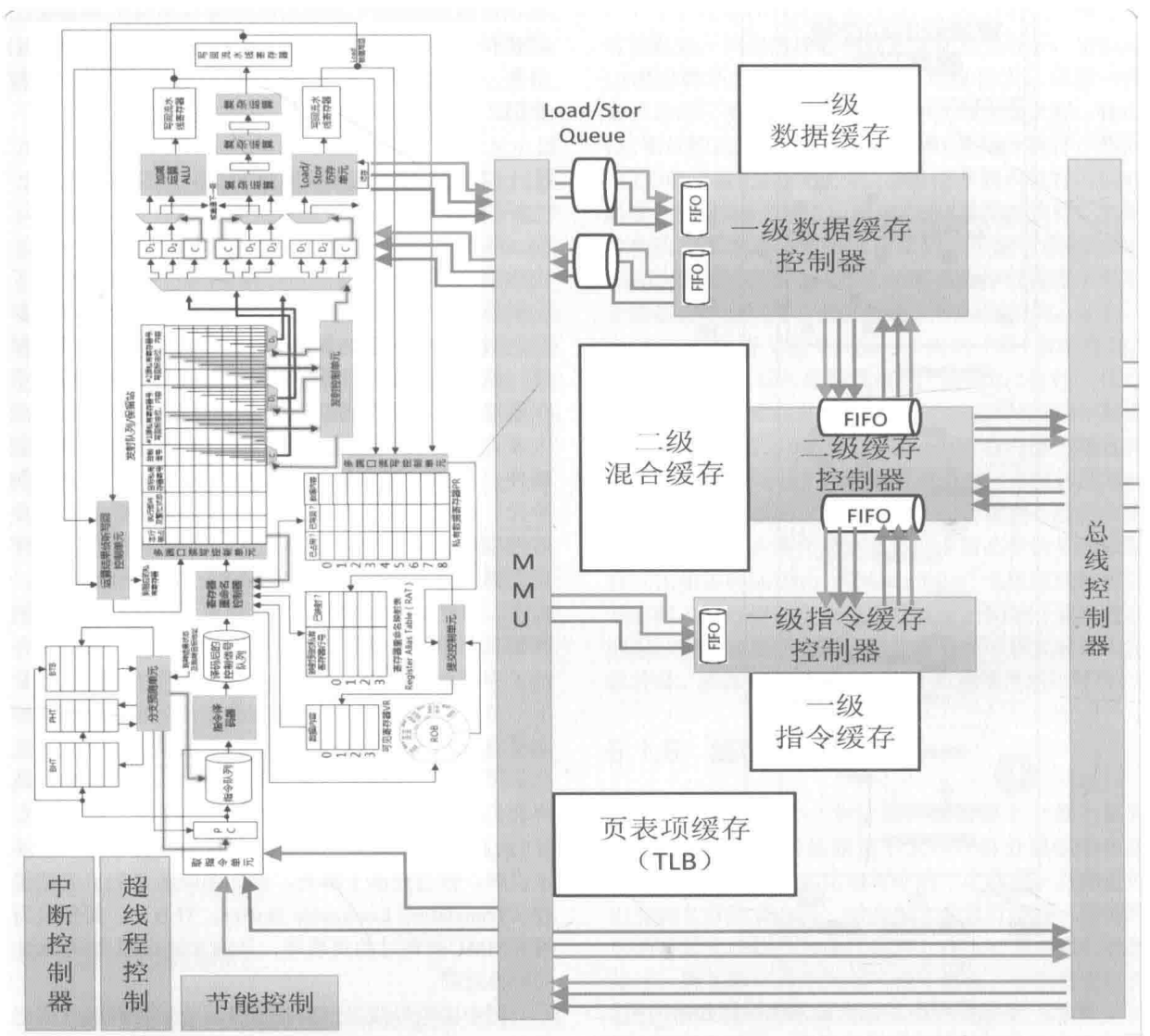


图6-6　CPU核心示意图

行异步采样，将对应的信号锁存到接收方寄存器中，然后处理该请求，并返回对应的状态信号给发送方。

如图6-7所示，左图是曾经在第3章中给出的，如果当时你还没有更加感性地理解图中的这些方方块块的东西是什么的话，那么和右图对比一下可能会更有感觉。当然，右图中的模块只是示意图，实际中会细分成更加密密麻麻的小块。

那么，这个核心与其他核心之间，以及与SDRAM大容量主存储器之间，又是怎么连接起来的呢？首先思考一个问题：多个核心之间并无关联，既然它们是各干各的，就没有必要相互连接起来啊。（实际上是需要的，在本章后面你会知道，由于缓存的存在，多个核心之间的缓存需要实时同步，必须以超高速度交换信息。）

但是多个核心都需要从SDRAM中来取数据进行处理，那么它们就都需要连接到SDRAM控制器上，对应的拓扑如图6-8所示。如果给每个核心单独连接一份SDRAM，这样的话就相当于多个独立的计算机了，这些核心之间也就不共享SDRAM内存（各自拥有各自的独立地址空间）。在前文中也说过，人们更加希望的是单台可以同时运行多个线程/进程的计算机。另外，对于一个进程，其运行时可能会创建多个线程，运行在多个不同核心上，而该进程对应的可执行文件的代码、数据是被载入到同一个地址空间中的，如果用多台独立的机器来运行同一个进程，运作起来就比较复杂（并不是不可以，比如本书后面章节中介绍的超算场景就是这样的）。所以，多核心共享内存是一个基本期望。

假设某个CPU有4个核心，那么可以设计一个有4个读写通道的SDRAM控制器。在第3章中大家可以看到SDRAM控制器内部的架构，如果做上4套读写接口，成本将会比较高，但不是不可以。最早的时候，人们采用了共享总线的形式来将多个核心连接到SDRAM控制器，如图6-9所示。

图6-10所示的是另一种核心、缓存、SDRAM的互联拓扑。可以看到，相比图6-8中所示的拓扑而言，其多了一个多核心全局共享的L3混合缓存，如果没有命中L3缓存，则由L3缓存控制器向SDRAM控制器发起数据读写请求。另外还可以看到，L3缓存控制器、SDRAM控制器、I/O桥使用了一套单独的总线互

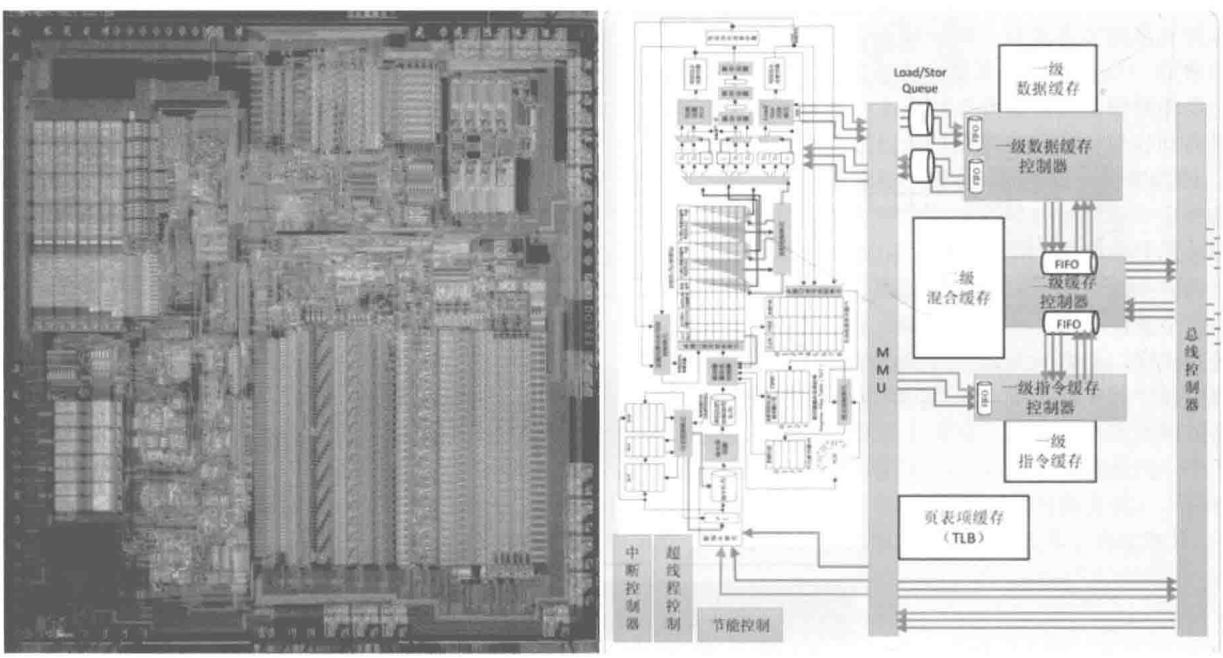


图6-7　实际的CPU核心芯片显微图与架构示意图比较

图6-8　CPU核心与SDRAM的连接示意图