BOOM v2
一个开源乱序执行 RISC-V 处理器核心

Translated by Tao Miao | Email: taomiao@pku.edu.cn

摘要：

本文展示了 BOOM 处理器第二版，第一版在[3]中介绍过。该设计方案是通过使用工厂提供的标准元器件库来进行综合，布局，布线的。并且，存储器使用台积电 28 纳米 HPM 工艺中的存储器编译器。

BOOM 是一个开源的处理器，它实现了 RV64G RISC-V 指令集架构（ISA）。像其它高性能内核，BOOM 是超标量的，支持乱序执行。BOOM 用 Chisel 语言实现，支持参数化配置，可以综合生成 FPGA 和 ASIC。

BOOMv2 是一个使用现代工业生产工具的新产品。我们也有访问标准的单端口和双端口存储器和相应的编译器，使我们能够探索设计权衡使用不同的 SRAM 存储器并进行比较对合成的触发器阵列。相对于 BOOMv1，主要的区别是功能包括更新的 3 阶段前端，采用更大的集合关联分支目标缓冲区；一个流水线寄存器重命名阶段；拆分浮点和整数寄存器文件；一个专用的浮点流水线；用于浮点，整数和内存的单独发射窗口和访存微码；以及分离选择发射阶段和读寄存器阶段。

管理寄存器文件的复杂性，极大的阻碍了 BOOM 时钟频率的提高。 我们花费相当大的努力设计一个半定制的 9-端口寄存器文件来探索潜在的改进，结合微架构技术来减少寄存器文件的大小和端口数量。 BOOMv2 具有 37 个四分频（FO4）延迟，在布线后有 50 个 FO4 延迟，相比 BOOMv1 的 65 FO4 减少了 24%。但是，每个周期的指令（IPC）性能下降高达 20%，主要是由于 load 指令的延迟和指令相关导致的。 不过，新的 BOOMv2 物理设计为 IPC 提高铺平了道路。

背景介绍：

BOOM 处理的设计来源于 MIPS R10000 和 Alpha 21264，这两个处理器都非常详细的研究了微结构。然而，这两个处理器频率的提高都得益于复杂的定制的逻辑设计。其中，Alpha 21264 有 15 个 FO4 延时[4]。而同一时期的 Xtensa 处理器有 44 个 FO4 延时。

对于 BOOM 处理器，我们根据其微结构找出了其关键路径，加了更多的流水线去提高 IPC 和频率。研究中，我们只用到了现有的单端口或者双端口存储器，并没有自己设计存储器。应该注意到的是，在处理器晶体管变得便宜，而功耗变得更严格的今天，很多新的复杂的技术已经很难去应用了[1]。现代高性能处理器已经极大的限制了他们的设计，而通常用规则的结构如阵列和寄存器文件。

2 BOOMv1

BOOMv1 延续了 MIPS R10000 的 6 级流水线------取指，译码/重命名，发射/读寄存器，执行，存储器操作，写回。在译码阶段，指令被映射成微操作；在重命名，逻辑寄存器被映射到物理寄存器。考虑到设计的简单性，全部微操作都被放到一个统一的发射窗口。同时，所有的物理寄存器都在一个寄存器文件里。执行部件包含一个定点处理单元和一个浮点处理单元。这大大的简化了浮点存储器指令和浮点整点转换指令，因为这些指令可以从一个寄存器文件里读取数据。BOOMv1 也支持了一个 2 阶段的前端流水线。条件转移预测在分支指令被译码的时候发生。

BOOMv1 的设计也部分得益于教育工具库和综合工具。虽然教育库可以帮助发现设计中的控制逻辑信号错误，但其对于数据流错误却没太大用。更要命的是，我们没有商业存储编译器。即使可以用 Cacti 这样的工具去建模存储特性，但它只能做单端口小规模的存储。然而我们的 BOOM 需要这样的 SRAM 阵列用来做比如 BPT，转移预测快照，BTB 等等。

在分析了 BOOMv1 的时序后，我们定位到了如下的关键路径：

（1）发射选择操作
（2）寄存器重命名忙表读操作
（3）条件转移预测重定向
（4）寄存器文件读操作

上述第（4）个只在布局布线后才表现在关键路径中。

3 BOOMv2

BOOMv2 是 BOOMv1 的一个升级版本，总结了 BOOMv1 在台积电 28nm 工艺下的优缺点后。我们探索了设计空间通过工厂提供的单/双端口存储器编译器和手写的标准单元构成的多端口寄存器文件。

BOOMv2 的工作从 4 月 9 日一直延续到 8 月 9 日，其中包括 4948 增加的代码行和 2377 删除的代码行。接下来的章节介绍了 BOOMv2 的主要工作。

3.1 前端（取指）

设计前端的目的是取指令以给后端的执行部件提供指令。处理器会在前端取指部件能不断的提供指令流的时候表现得很好。这就要求前端能利用分支预测技术取预测指令的哪一个分支是真正会被执行的。任何错误的预测都会影响取指直到分支指令被真正的执行。在一个错误预测发生时，这条指令后所有指令都会被赶出处理器，前端必须重新开始取正确的指令。

前端会依赖几种分支预测技术来预测指令流，其中每种技术都是精确度，面积，关键路径和流水线惩罚的折中，当做一个分支预测的时候。

分支目标缓冲器（BTB） BTB 中保存一些映射表，这类表把 PC 映射到一个分支目标地址。当查找的时候，用查找的地址索引 BTB 看时候有匹配的地址。如果匹配上一个，BTB 则预测将会跳转到目标地址并改变前端的取指地址。一些 bit 位（相当于 BHT）用于帮助确定分支是否发生。BTB 是一个成本非常高的结构，它必须为每一个入口保存一个 tag（部分 tag 20bits，全 tag 64bits）和一个目标地址（64 bits）。

返回地址栈（RAS） RAS 预测函数返回地址。寄存器跳转指令是一种比较难预测的分支指令，因为它们的目标指令取决于一个寄存器值。然而，函数通常通过一个 call 指令进入 A 地址执行，然后执行完后返回 A+1 地址。RAS 可以检测 call 指令，计算返回地址保存到 RAS 给函数返回时使用。为了支持嵌套的函数调用，RAS 被设计成一个栈结构。

条件分支预测器（BPD） BPD 保存一个预测表和一个 BHT。BPD 只做分支是否发生的预测，因此它依赖于其它部件来给他提供一个指令是否是分支的信息以及分支的目标地址信息。BPD 可以使用 BTB 的信息或者可以等到指令译码得到的信息。因为 BPD 不保存分支目标地址，所以他可以设计得更大，做更精确的分支转移预测。当 BTB 只能保存 80-128 项的时候，BPD 可以做成全局历史预测器。全局历史预测器通过用 Hash 表记录前 N 个分支执行情况来进行预测。对于一个复杂的 BPD 来说，这个 Hash 表的 Hash 函数可以很复杂。BOOM 的分支预测器用一个单端口的 SRAM 存储。即使很多预测器表都是高瘦型的（比如 2*1K），但是 Chisel 通常会把他们转换成比较矮胖的样子，这样更适合存储编译器工作。

图三展示了前端设计的流水线组织。我们发现 BOOMv1 的一个关键路径是在 BPD 预测和重定向分支方向上，因为 BPD 必须首先译码新进来的指令然后计算可能的分支指令。对于 BOOMv2，我们做了一整个时钟周期去译码。我们也用了一整个周期去做 Hash，这样做可以把 Hash 操作从 Next-PC 的关键路径上去除。

我们为 BPD 添加了可选的继续在 F2 阶段做预测的条件，可以通过 BTB 来提供译码和目标地址信息。然而，我们发现宣统预测表和重定向指令流需要重新设计一个表。

另一个前端的关键路径在 BTB 上。我们发现 40 个表项已经是之前全相连 BTB 的极限了。所以我们重写了 BTB 为组相连并用单端口存储器实现它。我们尝试了在 flip-flop 存储器和 SRAM 上实现 tag，从表项来看，SRAM 综合比较慢但布局布线比较好。

3.2 分布的发射窗口

发射窗口保存所有的流水线中未执行的指令的微操作码。每一个发射端口从准备好的指令中选择一个发射。有些处理器，比如 Intel 的 Sandy Bridge 处理器，用一个统一的保存站来保存所有的指令。另外一些处理器为每一个功能部件设置一个单端口发射窗口。这两种方法各有优劣。

发射窗口的大小决定了可以同时存在未执行的，可以被选择乱序执行的指令条数。发射窗口越大，就可以容纳更多的指令来被调度。对于 BOOM 来说，发射窗口被设计成为循环队列以使得最先进入的指令可以被排在最前面。对于发射选择，一个多级优先编码选择器选择最先进入的准备好的指令发射。这条关键路径可能会被增加入口数或者增加发射端口数而延时增加。对于 BOOMv1，我们实现的 20 项 3 发射端口的发射窗口就被发现是比较激进的，所以我们选择用三个分离的发射窗口（分别是整数，存储，浮点），每个 16 项。这样我们解决了发射选择增加关键路径长度的问题同时还增加了发射窗口总大小。不过，要保持每个周期执行两个整点 ALU 指令和一个存储指令的话，BOOM 需要在整点发射端口用两个发射选择端口。

3.3 寄存器文件设计

乱序执行处理器的一个关键部件，并且最难设计的部分是一个多端口的寄存器文件。因为存储器代

价比较高，现代处理器架构用寄存器来临时保存工作集数据。这些寄存器聚集成一个寄存器文件。指令直接访问寄存器文件，把数据传递给执行部件，或者把执行部件的数据写回寄存器文件。现代支持多发射的处理器通常有 6 个读口和 3 个写口。

BOOMv1 的寄存器文件遇到了很多挑战，读出数据在关键路径上，传递数据到功能部件比较难做路由，并且，寄存器文件需要违反工厂设计工具规则才能成功综合。寄存器数量和读写端口数量都加剧了设计难度。

我们采用了两种不同的方法去提升寄存器文件。第一种是在微架构上下功夫。我们把发射阶段和读寄存器文件分开，发射选择现在用一个时钟周期来选这发射的微码，另外再用一个时钟周期来读寄存器文件。我们通过把整点寄存器文件和浮点寄存器文件分离来减少寄存器文件大小。这样，我们同时减少了寄存器文件的读端口数量。

第二种方法是通过改进物理设计。一个显著的问题在寄存器文件布局布线的时候出现的是面积问题------逻辑上分开的两条线在物理上可能重叠了。这是因为寄存器文件的连线过密导致的。BOOMv2 的 70 项整点寄存器文件，每个有 6 个读端口和 3 个写端口，一共是 4480 bits，每一个需要 18 条连线。这就导致了综合阵列和布线的不匹配。

由于以上原因，我们选择了舍弃 Chisel 提供的寄存器文件而用工厂标准工具手写寄存器文件。我们然后让工具自动连线。虽然这解决了布线问题，不过用三态门驱动 70 个读端口还是很难。因此我们有设计了层次位线，把寄存器文件分成簇，三态门分别驱动每个簇的读端口，然后通过选择器选择其中一个读端口的数据。

对于比较小的浮点寄存器文件，我们能直接简单综合了。

4 学到的教训

把 RTL 设计通过现代 VLSI 工具链做成产品的过程是一个很有意义的过程。

处理很多端口的存储器和高度拥挤的布线需要很多微架构的解决方案。处理关键路径上由于存储带来的问题通常会影响 IPC，这就需要更多的微结构上的改变。早期缺少好的存储模型和布局造成了一系列问题。后来人工设计的方案对于探索设计空间有很大的帮助。

存储时序对于频率很敏感；高瘦型的存储设计不现实。我们写了一个 Chisel 的生成器来自动把高瘦型存储转变成矮胖型结构，这个生成器改变 Hash 生成函数并且利用读写口的 bit 掩码。

去降低所有关键路径的延迟不是一个聪明的决定。最重要的关键路径是读寄存器文件，从我们在布局布线的时候观察来看。修复综合后的关键路径问题可能会导致更坏的 IPC，得不偿失。

用生成器来描述一个硬件被证明是一个有用的技术；多个设计点子可以被生成并且评估好坏，而且，最后的设计可以在设计的靠后阶段决定。我们也可以更加确信某些关键路径不应该去尝试解决；通过移除功能部件和寄存器读端口，我们可以估计减少的端口数量和寄存器文件带来的性能提升。

Chisel 是一个很好的表达语言。在一个合适的软件工程基础上，对于数据流水线激进的改进可以被很快的实现。奋力寄存器和发射窗口花了大约一周的工作，寄存器重命名流水化也花了一周时间。然而，物理设计是很难的一块。细小的改变还好，但较大的改变就可能改变物理布局，严重影响关键路径，对于新的设计也有影响。

5 怎样才能更快

在把 BOOM 设计到 35 FO4 延迟以下的过程中有很多挑战。第一个就是要重新设计 L1 指令缓存和数据缓存。这两个部件都要求一个时钟周期内返回数据。这个路径差不多 35 FO4 延时。一个新的技术可以增加时钟频率，但是却会影响缓存访问延迟。

在这个分析中，我们用常规门限 RVT SRAM。然而，BTB 是一个很重要的结构，通常需要在一个时钟周期内给出预测结果，所以，这也是一个重要的工作。这其中还有很多这样的结构需要更多人力解决，比如功能部件。在乱序执行处理器中 CAM 是一个重要的结构，比如 load/store 部件和 TLB；并且，发射选择逻辑的复杂度决定了发射窗口可以做多大，也就决定了同时可以进入发射队列的指令多少。

然而，任何能提高 BOOM 的时钟频率的技术都必须平衡对于 IPC 的影响。比如，BOOM 的新前端会被额外的气泡指令影响，即使分支预测正确。需要更多的策略来减少这些气泡，当分支预测正确的时候。

6 结论

现代乱序执行处理器依靠很多不同的存储结构，并且他们中的很多出现在关键路径中。他们对于关键路径的影响很难用现在的 flip-flop 阵列或者模型来评估；因为它们可能根本不能真正物理实现或者实

现了性能很差。人工重新设计，可综合的寄存器文件，以及 Chisel 生成器使得我们可以找出真正的关键路径，区分假的关键路径。这种设计方法可以帮助我们减少人工工作量。

BOOM 的设计还在进行中，我们可以预期一个更好的处理器。因为 BOOMv2 在 BOOMv1 基础上已经很大的提高了关键路径效率，我们可以预测 BOOMv2 在 IPC 上有不错的表项。用 Coremark 实际测试，我们发现 BOOMv2 在 IPC 上有 20%的下降。超过一半的性能下降来自于相关指令间增加的延迟。BOOM 目前还没有采用一些可用的技术来解决这个问题。不过，BOOMv2 增加了很多参数，允许设计去配置重命名和读寄存器部件流水线深度，使得 BOOMv2 可以弥补损失的 IPC，以换取更长的时钟周期。