## 龙芯架构

龙芯架构（Loongson Instruction Set Architecture,简称LoongArch）是一种RISC指令集架构，由我国龙芯团队开发。

龙芯架构继承了精简指令集计算机（RISC）的设计传统，其指令长度固定且编码格式规整，大多数指令为三操作数，仅有load/store访存指令可以访问内存。龙芯架构按照地址空间大小可分为32位和64位两个版本，分别简称为LoongArch32和LoongArch64，LoongArch64应用级向下二进制兼容LoongArch32。

龙芯架构具有完全自主、技术先进、兼容生态三方面特点。龙芯架构从整个架构的顶层规划，到各部分的功能定义，再到细节上每条指令和每个寄存器的编码、名称、含义，全部自主重新设计，具有充分的自主性。龙芯架构摒弃了传统指令系统中部分不适应当前软硬件设计技术发展趋势的陈旧内容，吸纳了近年来指令系统设计领域诸多先进的技术发展成果，易于硬件的高性能低功耗设计和软件的编译优化和操作系统、虚拟机的开发。龙芯架构在设计时充分考虑兼容生态需求，融合了各包括X86、ARM在内国际主流指令系统的主要功能特性，同时依托龙芯团队在二进制翻译方面十余年的技术积累创新，不仅能够确保现有龙芯电脑上应用二进制的无损迁移，而且能够实现多种国际主流指令系统的高效二进制翻译。

龙芯架构采用基础部分加扩展部分的模块化组织形式。一个兼容龙芯架构的CPU，除实现必需的基础部分（Loongson Base，简称LBase）外，可根据实际需求选择实现各扩展部分。目前龙芯架构已定义的扩展部分包括：虚拟化扩展（Loongson Virtualization，简称LVZ）、二进制翻译扩展（Loongson Binary Translation，简称LBT）、128位向量扩展（Loongson SIMD Extension，简称LSX）和256位高级向量扩展（Loongson Advanced SIMD Extension，简称LASX）。

==以下内容仅供参考

引言

我国的CPU研发应采用兼容指令系统还是自主研发指令系统是学术界和产业界长期争论的一个话题。兼容指令系统的优点是可利用现有的成熟软件生态，缺点是长远发展受制于人且不利于自主软件产业的发展。自主指令系统则反之。最近，构建自主可控的信息技术体系和产业生态已成为共识，而构建自主信息技术体系需要自主指令系统。本文介绍充分考虑兼容需求的龙芯自主指令系统架构（Loongson Instruction Set Architecture，简称LoongArch）。龙芯架构包括基础部分、向量扩展、虚拟化和二进制翻译扩展三个扩展部分，共计近2000条指令。

龙芯架构具有完全自主、技术先进、兼容生态三方面特点。龙芯架构从整个架构的顶层规划，到各部分的功能定义，再到细节上每条指令和每个寄存器的编码、名称、含义，全部自主重新设计，具有充分的自主性。龙芯架构摒弃了传统指令系统中部分不适应当前软硬件设计技术发展趋势的陈旧内容，吸纳了近年来指令系统设计领域诸多先进的技术发展成果，易于硬件的高性能低功耗设计和软件的编译优化和操作系统、虚拟机的开发。龙芯架构在设计时充分考虑兼容生态需求，融合了各包括X86、ARM在内国际主流指令系统的主要功能特性，同时依托龙芯团队在二进制翻译方面十余年的技术积累创新，不仅能够确保现有龙芯电脑上应用二进制的无损迁移，而且能够实现多种国际主流指令系统的高效二进制翻译。

软件生态是龙芯指令系统架构能否成功的基础和关键。除了迁移BIOS和操作系统内核到LoongArch，还需要3+3+3的主要编译系统。第一个“3”是三个C编译器，包括GCC、LLVM和GoLang。第二个“3”是三个重要虚拟机，包括Java虚拟机、JavaScript虚拟机和.NET虚拟机。第三个“3”是三个二进制翻译系统，包括从MIPS到LoongArch的二进制翻译系统、从X86到LoongArch的二进制翻译系统和从ARM到LoongArch的二进制翻译系统。在上述“3+3+3”主要编译系统的基础上，就能够突破指令系统的壁垒，构建LoongArch的软件生态体系。

支持龙芯架构的龙芯3A5000处理器芯片已经流片成功。在3A5000上成功运行了完整的Linux操作系统，并通过二进制翻译技术高效运行原有龙芯计算机上的MIPS应用、X86计算机Linux和Windows系统上的各种应用。实测结果表明，在相同微结构的情况下，LoongArch比MIPS性能提高15%以上。从MIPS到LoongArch的二进制翻译系统效率达90%以上，从X86到LoongArch的二进制翻译系统效率达80%以上。上述测试结果表明，实现兼顾自主和兼容的指令系统是可行的，为建立自主可控的信息技术体系和产业生态打下坚实的基础。

本文第二节介绍龙芯指令系统架构LoongArch。第三节介绍实现LoongArch架构的龙芯3A5000及其基础软件。第四节介绍龙芯二进制翻译系统。第五节介绍性能测试与分析。第六节是结论和未来工作。

2 龙芯指令系统架构

龙芯架构继承了精简指令集计算机（RISC）的设计传统，其指令长度固定且编码格式规整，大多数指令为三操作数，仅有load/store访存指令可以访问内存。龙芯架构按照地址空间大小可分为32位和64位两个版本，分别简称为LoongArch32和LoongArch64，LoongArch64应用级向下二进制兼容LoongArch32。下面从指令编码、组成部分和对二进制翻译的支持三个方面对龙芯架构进行介绍。

2.1 指令编码

龙芯架构中的所有指令长度均为32位，且要求指令地址4字节边界对齐，当指令地址不对齐时将触发地址错例外。

龙芯架构中指令编码风格规整。所有寄存器操作数域都从第0比特开始从低到高依次摆放，操作码都是从第31比特开始从高到低依次摆放。如果指令中包含有立即数操作数，那么立即数域位于寄存器域和操作码域之间，根据不同指令类型有不同的长度。具体来说，包含9种典型的指令编码格式，即3种不含立即数的编码格式2R、3R、4R，以及6种含立即数的编码格式2RI8、2RI12、2RI14、2RI16、1RI21、I26。图1给出了这9种典型编码格式的具体定义。

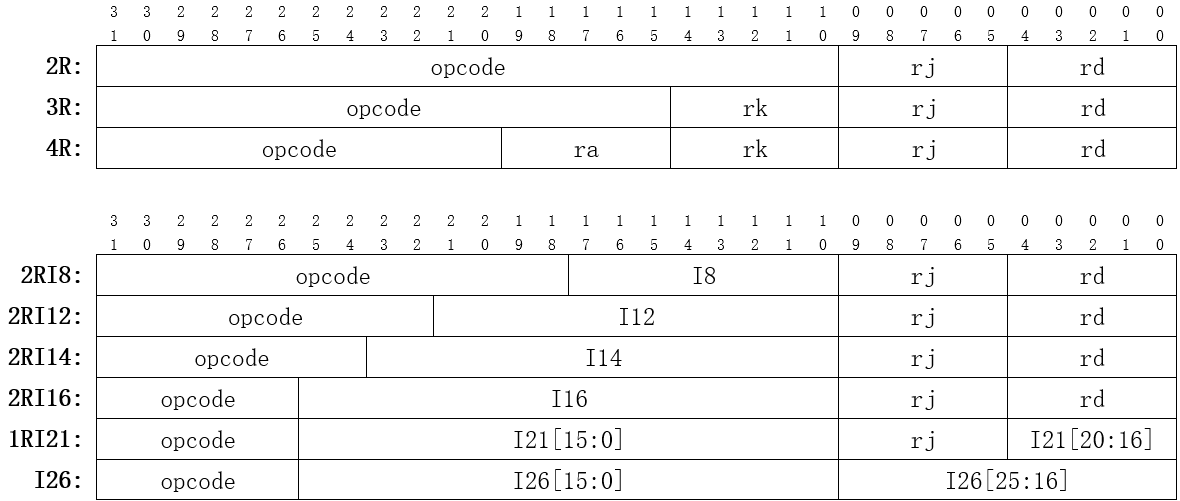


Figure 1 Instruction Format of LoongArch

图 1 龙芯架构指令编码格式

2.2 组成部分

龙芯架构采用基础部分加扩展部分的模块化组织形式。一个兼容龙芯架构的CPU，除实现必需的基础部分（Loongson Base，简称LBase）外，可根据实际需求选择实现各扩展部分。目前龙芯架构已定义的扩展部分包括：虚拟化扩展（Loongson Virtualization，简称LVZ）、二进制翻译扩展（Loongson Binary Translation，简称LBT）、128位向量扩展（Loongson SIMD Extension，简称LSX）和256位高级向量扩展（Loongson Advanced SIMD Extension，简称LASX）。

**龙芯架构的基础部分**包含用户态和核心态两方面内容。用户态部分定义了常用的整数和浮点数指令，能够充分支持现有各主流编译系统生成高效的目标代码。核心态部分在处理器特权等级、例外和中断处理、存储管理以及配套的控制状态寄存器等方面给出了明确规范，旨在支持目前主流的类Unix操作系统。与龙芯CPU原来实现的MIPS架构相比，龙芯基础架构充分利用后发优势，摒弃了传统指令系统中部分不适应当前软硬件设计技术发展趋势的陈旧内容，同时积极吸纳了近年来指令系统设计领域诸多先进的技术发展成果，主要特点包括：

（1）取消传统RISC指令系统中一些过时指令。包括取消陷阱指令和进行溢出判断的运算指令，整数乘除运算的结果将直接写入通用寄存器而非单独的HI/LO寄存器。这些过时指令或者不利于指令流水线的高效实现，或者当前主流编程语言已经很少使用。

（2）增加一些便于实现且有利于软件提高性能的指令。引入基于PC的运算指令并为间接跳转指令添加立即数偏移，同时增加相对PC跳转指令的偏移范围，这些调整有利于改善位置无关代码中长跳转和数据访问的指令（序列）的执行效率，并且能够大幅度减小全局偏移表（Global Offset Table，简称GOT表）的规模，从而降低因GOT表规模过大带来的维护和访问开销。增加原子访存修改指令，解决传统LL/SC指令在大规模并发执行情况下失效重试开销急剧增加的问题。

（3）取消转移指令延迟槽。对于现代多发射的乱序动态流水线，要求转移指令后面一条指令（即延迟槽）一定执行无法起到提升性能的作用，还容易成为实现负担。

（4）由硬件负责所有流水线冲突的处理，允许普通访存指令的地址非对齐访问，由硬件维护指令和数据Cache之间的数据一致性。这些功能简化了软硬件的界面，降低了应用迁移的成本。

（5）优化处理器特权态、例外系统和计时系统。处理器特权等级从MIPS的用户态、监管态和核心态三个状态调整为PLV0、PLV1、PLV2、PLV3四个状态。例外系统从多个例外公用入口调整为每个例外独立入口。计时系统的计时频率恒定，不再随处理器核频率变化而变动，消除处理器动态功耗管理带来的问题。

（6）取消地址空间的固定分段方式以及地址段与特权等级、映射方式间的固定绑定，代之以单一平整（flat）寻址空间且所有存储管理配置信息软件均可动态调整。

（7）支持控制寄存器的原子修改、规范核外控制寄存器且使用独立的寻址空间。增加控制寄存器的原子修改支持以简化系统软件实现。大幅度拓展核内控制寄存器寻址空间，为指令系统核心态部分的后续演进解除束缚。规范核外控制器并使用独立的寻址空间，将各种多核多路系统的底层硬件信息封装为固定统一格式呈现给软件。

**龙芯架构的虚拟化扩展部分**定义了一系列硬件支持特性，旨在提升系统虚拟化实现的性能，涉及处理器虚拟化、内存虚拟化和I/O虚拟化三个方面。其具体内容主要包括客户机专用的运行模式和特权资源、例外分级处理、TLB两级地址翻译支持以及中断虚拟化。

**龙芯架构的二进制翻译扩展部分**引入了一系列硬件特性，进而以软硬件协同的方式大幅度提升跨指令系统二进制翻译执行效率。有关这部分的详细介绍将在本文的2.2节展开。

**龙芯架构下的向量扩展部分**定义了一系列单指令多数据（SIMD）指令，利用数据级并行性提升程序执行性能。向量扩展部分可进一步分为向量扩展（LSX）和高级向量扩展（LASX），后者在前者基础上将操作向量位宽从128位增至256位并进一步扩展。

上述基础部分加扩展部分的组成形式对于LoongArch32和LoongArch64架构均适用。从功能角度而言，各部分中属于LoongArch32的内容均包含在LoongArch64中。各部分中的用户态相关内容，LoongArch64向下二进制兼容LoongArch32。

2.3 龙芯架构的ABI优化

LoongArch的应用程序二进制接口（Application Binary Interface，简称ABI）与龙芯CPU原使用MIPS相比对寄存器使用约定进行了优化。表1和表2分别是64位MIPS和LoongArch的通用寄存器使用约定。与MIPS相比，LoongArch有如下差异：

（1）取消了汇编暂存器（$at）。MIPS的一些汇编宏指令用多条硬件指令合成，汇编暂存器用于数据周转。LoongArch指令系统的宏指令可以不用周转寄存器或者显式指定周转寄存器，汇编暂存器不再必要了。这可以增加编译器可用寄存器的数量。

（2）取消了预留给内核的专用寄存器（$k0/$k1）。MIPS预留两个寄存器的目的是支持高效异常处理，可以省去异常处理过程中保存上下文到内存中的开销。LoongArch提供了便签寄存器来高效暂存数据，可以在不预留通用寄存器的情况下保持高效实现，给编译器留下了更多的可用寄存器。

（3）取消了$gp寄存器。MIPS中用$gp寄存器指向GOT（Global Offset Table）表以协助动态链接器计算可重定位的代码模块的相关符号位置。LoongArch指令集支持基于PC的运算指令，能够用其他高效的方式实现动态链接，不再需要额外花费一个通用寄存器。

（4）复用参数寄存器和返回值寄存器，参数寄存器$a0/$a1也被用作返回值寄存器。这也是现代指令系统比较常见的做法，它进一步增加了通用暂存器的数量。

（5）增加了线程指针寄存器$tp，用于高效支持多线程实现。$tp总是指向当前线程的TLS（Thread Local Storage）区域。

Table 1 MIPS N64 Integer register usage

表1 MIPS N64 整数寄存器使用约定

| **寄存器编号** | **MIPS N64助记符** | **使用约定** |
| --- | --- | --- |
| 0 | zero | 总是为0 |
| 1 | at | 汇编暂存寄存器 |
| 2~3 | v0,v1 | 子程序返回值 |
| 4~11 | a0~a7 | 子程序的前八个参数 |
| 12~15 | t0~t3 | 不需保存的暂存器 |
| 16~23 | s0~s7 | 寄存器变量，过程调用时需要存储和恢复 |
| 24~25 | t8,t9 | 暂存器 |
| 26~27 | k0,k1 | 为异常处理保留 |
| 28 | gp | 全局指针 |
| 29 | sp | 栈指针 |
| 30 | s8/fp | 寄存器变量，或作为帧指针 |
| 31 | ra | 子程序返回地址 |

Table 2 LoongArch64 Integer register usage

表2 LoongArch64 定点寄存器使用约定

| **寄存器编号** | **LA64助记符** | **使用约定** |
| --- | --- | --- |
| 0 | zero | 总是为0 |
| 1 | ra | 子程序返回地址 |
| 2 | tp | thread pointer，指向线程私有存储区 |
| 3 | sp | 栈指针 |
| 4~11 | a0~a7 | 子程序的前八个参数 |
| 4~5 | v0~v1 | v0/v1是a0/a1的别名，用于表示返回值 |
| 12~20 | t0~t8 | 不需保存的暂存器 |
| 21 | reserved | 暂时保留不用 |
| 22 | fp | frame pointer，栈帧指针 |
| 23-31 | s0~s8 | 寄存器变量，子程序使用需要保存和恢复 |

2.4 龙芯架构对二进制翻译的支持

二进制翻译技术是实现跨指令系统兼容的重要手段。二进制翻译技术在宿主机（host）上用软件模拟出一个目标机/客户机（guest）指令系统兼容的CPU来，从而在宿主机上执行客户机的二进制代码。如在MIPS计算机上模拟X86指令系统，从而实现与X86兼容。二进制翻译的最大问题是效率问题，用软件模拟的CPU比硬件直接实现的CPU慢很多，一般有数量级的差异。

通过硬件支持和软硬件协同可以有效提高二进制翻译的效率。LoongArch在认真分析X86、ARM、MIPS、RISC-V指令系统主要特点的基础上，增加了为提高X86、ARM、MIPS、RISC-V二进制翻译性能至关重要的指令功能。下面举例说明LoongArch对X86二进制翻译的具体支持。

（1）增加生成运算标志的运算指令以及根据运算标志生成转移条件的指令。X86定义了独立的运算结果标志寄存器EFLAG，由运算指令在产生结果数值的同时予以更新，转移指令依据EFLAG的值进行跳转判断。若采用纯软件方式模拟一条X86运算指令对于EFLAG的更新需要几十条指令。图2(b)给出了模拟图2(a)中X86指令“SUB ecx,edx”运算结果及常用的四位标志（SF、ZF、OF、CF）的LoongArch基础指令序列，需要近30条指令。为了降低软件模拟运算结果标志的二进制翻译开销，LoongArch定义了一组专门用于产生X86运算结果标志的指令，这类指令仅将运算结果标志写入到通用寄存器中。同时，LoongArch还定义了根据通用寄存器中的运算结果标志值，生成X86转移指令相对应的转移条件的指令。图2(c)给出了采用二进制翻译扩展的LoongArch翻译图2 (a)中X86指令“SUB ecx,edx”的指令序列，与仅采用基础指令的翻译结果相比指令数大幅度减少。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 |  | SUB | ECX | EDX |  |  |
| 1 |  | JE | X86\_target |  |  |  |
| (a) 原始X86代码片段 | | | | | | |
| 0.00 |  | SUB.W | Result, | Recx, | Redx |  |
| 0.01 |  | SRLI.W | Rsf, | Result, | 31 | /\*SF=Result[31]\*/ |
| 0.02 |  | BEQ | Result, | R0, | L1 |  |
| 0.03 |  | ADDI.W | Rzf, | R0, | 0 | /\*ZF=0\*/ |
| 0.04 |  | B | L2 |  |  |  |
| 0.05 | L1: | ADDI.W | Rzf, | R0, | 1 | /\*ZF=1\*/ |
| 0.06 | L2: | SRLI.W | Rtmp1, | Result, | 31 |  |
| 0.07 |  | SRLI.W | Rtmp2, | Redx, | 31 |  |
| 0.08 |  | SRLI.W | Rtmp3, | Recx, | 31 |  |
| 0.09 |  | BEQ | Rtmp1, | Rtmp3, | L3 |  |
| 0.10 |  | BEQ | Rtmp2, | Rtmp3, | L3 |  |
| 0.11 |  | ADDI.W | Rof, | R0, | 1 | /\*OF=1\*/ |
| 0.12 |  | B | L4 |  |  |  |
| 0.13 | L3: | ADDI.W | Rof, | R0, | 0 | /\*OF=0\*/ |
| 0.14 | L4: | SRLI.W | Rhigh2, | Recx, | 16 |  |
| 0.15 |  | SRLI.W | Rhigh1, | Redx, | 16 |  |
| 0.16 |  | BEQ | Rhigh2, | Rhigh1, | L5 |  |
| 0.17 |  | BLT | Rhigh2, | Rhigh1, | L7 |  |
| 0.18 |  | B | L6 |  |  |  |
| 0.19 | L5: | SLLI.W | Rlow2, | Recx, | 16 |  |
| 0.20 |  | SRLI.W | Rlow2 | Rlow2, | 16 |  |
| 0.21 |  | SLLI.W | Rlow1, | Redx, | 16 |  |
| 0.22 |  | SRLI.W | Rlow1, | Rlow1, | 16 |  |
| 0.23 |  | BLT | Rlow2, | Rlow1, | L7 |  |
| 0.24 | L6: | ADDI.W | Rcf, | R0, | 0 | /\*CF=0\*/ |
| 0.25 |  | B | L8 |  |  |  |
| 0.26 | L7: | ADDI.W | Rcf, | R0, | 1 | /\*CF=1\*/ |
| 0.27 | L8 | ADD.W | Recx, | Result, | R0 |  |
| 1.00 |  | BNE | Rzf, | R0, | LA\_target |  |
| (b) LoongArch基础指令翻译代码片段 | | | | | | |
| 0.0 |  | SUB.W | Result, | Recx, | Redx | /\*生成Sub 指令运算结果\*/ |
| 0.1 |  | X86SUB.W | Reflag, | Recx, | Redx | /\*生成Sub指令EFLAG结果\*/ |
| 0.2 |  | SETX86J | Rtmp, | Reflag, | EQ | /\*根据EFLAG判断转移条件\*/ |
| 1.0 |  | BNE | Rtmp, | R0, | LA\_target |  |
| (c) LoongArch含二进制翻译扩展后的翻译代码片段 | | | | | | |

Figure 2 Example of X86 EFLAGS translation

图 2 X86 EFLAG翻译示例

（2）增对X86浮点寄存器特殊寻址模式和数据格式的支持。X86架构的X87浮点部件有8个浮点寄存器采用栈寻址模式，即X86浮点指令码中只有所访问寄存器相对于栈顶的偏移值，其加上浮点状态字中的TOP域的值才是要访问的寄存器编号。如果在二进制翻译中动态计算上述浮点寄存器号，每条访问浮点寄存器的浮点指令需额外花费多条指令。为解决这一问题，LoongArch定义了3位的TOP寄存器、浮点寄存器的TOP寻址模式以及用于操作TOP寄存器和切换TOP寻址模式的指令。当处于TOP寻址模式时，普通浮点指令中浮点操作数的寄存器号都是这些域的值与TOP寄存器中的值相加后的结果，从而省去了软件计算实际寄存器号的翻译开销。此外，X87浮点部件支持40位的扩展单精度和80位的扩展双精度浮点运算，与RISC处理器中32位单精度和64位双精度不同，为此LoongArch定义了将X87的浮点数与LoongArch的浮点数进行格式转换的指令。

（3）扩展TLB MMU功能支持客户机虚地址到宿主机实地址的硬件直接翻译。系统级二进制翻译过程中所有客户机访存指令都要两级虚实地址转换，即先把客户机的虚地址翻译成客户机的物理地址，再将客户机的物理地址作为宿主机的虚地址翻译成宿主机的物理地址。在QEMU[1]中，客户机虚地址到客户机物理地址的转换是通过软件模拟完成的。图3(b)中给出模拟图3 (a)中X86访存指令时，使用LoongArch基础指令模拟将客户机虚地址转化为客户机物理地址所需的指令序列，共有14条指令且其中包含两条访存。为降低实现开销，LoongArch的TLB可以同时存放“宿主机虚地址=>宿主机物理地址”和“客户机虚地址=>宿主机物理地址”两种类型的页表项，前一类型的页表项由宿主机操作系统中的普通页表提供，后一类型的页表项由二进制翻译虚拟机中维护的影子页表提供。两类页表项在填入TLB的过程中将被标记上不同的页表类型标识。访存指令在查找TLB的过程中，也将根据自身携带的是宿主机虚地址还是客户机虚地址，仅查找页表类型标识与之对应的TLB表项。当查找过程发生异常时，不同页表类型标识所触发的TLB例外类型及其入口也不同。访存指令所携带虚地址的类型判定，可以采用预设地址空间或访存指令前缀指定两种方式。图3 (c)中给出了采用前缀指令方式判定的翻译指令序列，只需要两条指令。具体的芯片实现还可以选择用未使用的某些虚拟地址位来表示地址空间，这样可进一步省去前缀指令，只需要一条指令。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 |  | MOV | %EAX | 4(%ESP) |  |  |
| (a) 原始X86程序 | | | | | | |
|  | | | | | | |
| 0.1 |  | ADDI.W | Rr11, | Resp, | 4 | /\* 计算虚拟地址 \*/ |
| 0.2 |  | SRLI.W | Rr12, | Rr11, | 12 | /\* 取虚拟页面基地址 \*/ |
| 0.3 |  | ANDI | Rr13, | Rr12, | 0xff | /\* 计算TLB index，取页面基地址最低8位 \*/ |
| 0.4 |  | LU12I.W | Rtmp, | %hi20(tlb) |  | /\* 取TLB首地址 \*/ |
| 0.5 |  | ORI | Rtmp, | Rtmp, | %low12(tlb) |  |
| 0.6 |  | SLLI.W | Rr14, | Rr13, | 3 | /\* 由TLB index计算在表中的偏移量，  每个TLB条目占8字节\*/ |
| 0.7 |  | ADD.W | Rr14, | Rr14, | Rtmp | /\* TLB条目的地址 \*/ |
| 0.8 |  | LD.W | Rr15, | Rr14, | 0 | /\* 取TLB中的虚拟页面基地址 \*/ |
| 0.9 |  | BNE | Rr15, | Rr12, | miss | /\* TLB不命中 \*/ |
|  |  |  |  |  |  |  |
|  | hit: |  |  |  |  |  |
| 0.10 |  | LD.W | Rr15, | Rr14, | 4 | /\* 取TLB中的物理页面基地址 \*/ |
| 0.11 |  | ANDI | Rtmp, | Rr11, | 0xfff | /\* 取页内偏移量 \*/ |
| 0.12 |  | ADD.W | Rtmp, | Rtmp, | Rr15 | /\* 形成物理地址 \*/ |
| 0.13 |  | LD.W | Reax, | Rtmp, | 0 | /\* 取数据 \*/ |
|  |  |  |  |  |  |  |
|  | miss: |  |  |  |  |  |
| 0.14 |  | B | tlb\_refill |  |  | /\* 查页表，反填TLB \*/ |
| (b) LoongArch基础指令翻译代码片段 | | | | | | |
| 0.1 |  | SETVPT | guest\_va |  |  | /\* 前缀指令指示后续访存携带客户机虚地址 \*/ |
| 0.2 |  | LW | Reax | 4(Resp) |  | /\* 访存指令通过独立X86 TLB进行虚拟地址转换 \*/ |
| (c) LoongArch含二进制翻译扩展后的翻译代码片段 | | | | | | |

Figure 3 Example of X86 memory access instruction translation

图3 X86访存指令翻译示例

（4）增加宿主机寄存器数量，避免用宿主机的内存模拟客户机的寄存器。对客户机寄存器的高效模拟是提高二进制翻译性能的关键。如果宿主机没有足够的寄存器，用内存单元来模拟目标机器的寄存器，就会大幅度降低性能。X86有8个通用寄存器、8个浮点寄存器和16个256位的浮点/向量混用寄存器，少于RISC架构下通用寄存器、浮点/向量寄存器的个数，可以采用一一对应的模拟方式。在MIPS等RISC架构中，采用一一对应的模拟方式极容易出现翻译过程中无临时寄存器可用的情况。为此龙芯架构定义了若干便签寄存器，用于二进制翻译系统临时存储数据。