小型微型计算机系统
Journal of Chinese Computer Systems
DOI:10.20009/j.cnki.21-1106/TP.2023-0087
2024年8月 第8期
Vol.45 No.8 2024

映射字典导向的64位ARM到RISC-V汇编翻译

贾金成[1],朱家鑫[2,3],唐 震[2],王志鹏[4],王 伟[2]

[1](广西大学 计算机与电子信息学院,南宁 530004)
[2](中国科学院软件研究所,北京 100190)
[3](中国科学院大学 南京学院,南京 211135)
[4](中国电子技术标准化研究院 软件应用与服务研究中心,北京 100007)
E-mail:zhujiaxin@otcaix.iscas.ac.cn

摘 要: RISC-V是一个新兴开放的精简指令集架构,采用模块化设计,具有精简、可靠且支持多平台的优点.RISC-V指令集架构的推广需要其软件生态的支撑,但目前RISC-V的软件生态还不够丰富,亟需建设,软件生态建设过程中需要将其他架构的软件向RISC-V架构迁移适配,现有的ARM到RISC-V汇编迁移辅助工具还不够成熟,存在寄存器使用错误、程序分支控制错误等诸多问题.因此,本文设计和实现了一个64位ARM到RISC-V的汇编翻译工具,通过设计指令映射字典将指令映射关系与工具的其他模块松耦合,实现了工具的高拓展性;根据两种架构的应用二进制接口差异设计了寄存器映射字典,充分利用了RISC-V的寄存器与内存资源.与现有工具相比,本工具更易拓展,并且支持更多指令类型.

关 键 词: RISC-V;ARM;汇编语言;软件移植
中图分类号: TP313 **文献标识码:** A **文 章 编 号:** 1000-1220(2024)08-2041-08

64-bit ARM to RISC-V Assembly Translation Guided by Mapping Dictionary

JIA Jincheng[1],ZHU Jiaxin[2,3],TANG Zhen[2],WANG Zhipeng[4],WANG Wei[2]

[1](School of Computer and Electronic Information,Guangxi University,Nanning 530004,China)
[2](Institute of Software Chinese Academy of Sciences,Beijing 100190,China)
[3](University of Chinese Academy of Sciences,Nanjing,Nanjing 211135,China)
[4](Software Application and Service Research Center,China Electronics Standardization Institute,Beijing 100007,China)

Abstract: RISC-V is an emerging and open architecture of reduced instruction set,which adopts modular design and has the advantages of simplicity,reliability and support for multiple platforms.The promotion of the RISC-V instruction set architecture requires the support of its software ecosystem,but the current software ecosystem for RISC-V is not rich enough and needs to be built urgently.The process of building the software ecosystem requires the migration and adaptation of software from other architectures to the RISC-V architecture.The existing ARM to RISC-V assembly migration tools are not mature enough and have many issues such as incorrect use of registers and program branch control errors.Therefore,this article design and implement an assembly translation tool from 64-bit ARM to RISC-V,and the instruction mapping relationship is loosely coupled with other modules of the tool by designing an instruction mapping dictionary to achieve a high scalability of the tool;a register mapping dictionary is designed according to the difference of application binary interfaces between the two architectures,which makes full use of the register and memory resources of RISC-V.Compared to existing tools,this tool is easier to expand and supports more types of instruction.

Keywords: RISC-V;ARM;assembly language;software migration

0 引 言

RISC-V是加州大学伯克利分校的研究人员开发的一款开放的精简指令集架构[1].其开源、模块化等设计优势使得近年来RISC-V指令集架构硬件得到了快速发展,SiFive、阿里巴巴平头哥半导体公司均开始量产和应用RISC-V架构的CPU核以及芯片[2].指令集架构的应用和推广高度依赖生态体系建设,但现阶段RISC-V依旧缺乏完整生态系统的支持[3],现有的Linux操作系统发行版,如Ubuntu、Debian和openEuler等,正在适配RISC-V[4],并且各种应用程序的迁移工作也正在进行中[5].

ARM架构已经发展了30余年,拥有完整的生态系统,涵盖移动端、人工智能、云计算、高性能计算(HPC)、汽车、嵌入式和物联网等领域.AArch64是ARM的64位版本,采用

收稿日期:2023-02-22 收修改稿日期:2023-04-21 基金项目:中国科学院战略性先导科技专项项目(XDA0320000,XDA0320100)资助;国家自然科学基金联合基金项目(U20A6003)资助. 作者简介:**贾金成**,男,1995年生,硕士研究生,研究方向为软件工程;**朱家鑫**,男,1988年生,博士,副研究员,CCF会员,研究方向为软件工程;**唐 震**,男,1990年生,博士,高级工程师,CCF会员,研究方向为网络分布式计算与软件工程;**王志鹏**,男,1975年生,博士,高级工程师,研究方向为云计算;**王 伟**,男,1982年生,博士,研究员,CCF会员,研究方向为大数据与机器学习系统.

ARM A64 指令集,有丰富的软件可以向 RISC-V 迁移[6].

程序的可移植性是指使用一种编程语言在一个系统平台上编写的程序经过很少改动或者不需要修改就可以在其他系统平台上运行的特性. 高级语言编写的程序具有较强的可移植性,向新平台移植只需进行小部分修改重新编译即可. 汇编语言是指令集架构强相关的语言,可移植性差. 但汇编语言编写的程序能够充分利用指令集架构的特性,还具有占用存储空间少、运行速度快、执行代码短的优点. 因此,高级语言通常会增加调用汇编语言程序的接口或允许与汇编语言混合编程,以获得汇编语言的优势. 高级语言编写的软件中也常包含由汇编语言编写的源程序. 由于不同架构的汇编代码不同,在跨架构迁移软件时,需要迁移人员按照原架构汇编源程序的功能重新编写新架构对应的汇编程序,如图 1. 但与高级语言相比,编写相同功能的汇编代码,需要更多的时间和精力. 虽然单个汇编指令容易理解,但汇编程序的可理解性会随着指令数量的增加而极大降低. 此外,跨架构迁移汇编源程序要求迁移人员至少熟悉原架构和目标架构两种汇编语言,而学习汇编语言的学习成本很高,以上种种原因提高了源码迁移工作的门槛. 综上,采用人工进行汇编源码迁移具有人工投入大、准确率低、效率低下、技术门槛高等缺点[7]. 因此,需要迁移辅助软件实现汇编程序的自动翻译,提高汇编源程序的迁移效率与准确率以及降低人员成本与技术门槛.

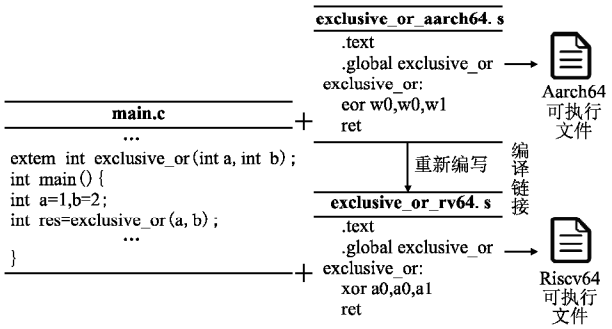

图 1 AArch64 向 RISC-V64 汇编程序迁移过程

Fig. 1 AArch64 to RISC-V64 assembler migration process

当前学术界和产业界存在一些软件迁移辅助工具,例如,华为公司开发了 DevKit 开发套件,旨在方便用户将软件从 x86 平台移植到华为鲲鹏平台,其中包括汇编翻译功能,可实现 x86 架构到 AArch64 架构的汇编翻译[7]. Cao 等提出了面向 64 位 RISC-V 的基础数学库自动化移植框架,能够实现申威架构向 RISC-V 架构汇编的自动翻译[8]. Moshe Schorr 和 Matan Ivgi 设计并实现了 ARM2RISCV,一个 ARM AArch64 到 RISC-V64 汇编转换器,能够实现 AArch64 到 RISC-V64 的汇编的自动转换,但是,经过测试,该工具存在寄存器使用错误、程序分支控制错误、指令翻译错误、指令拓展困难等缺点,无法用于实际的软件迁移工作[9].

本文通过分析 AArch64 与 RISC-V64 两种架构的差异,设计了寄存器映射字典和易于拓展的指令映射字典,实现了一个准确且易于拓展的 AArch64 到 RISC-V64 汇编翻译工具.

1 汇编翻译工具框架

翻译工具的功能是将应用程序中由 AArch64 汇编编写的源代码静态翻译为 RISC-V64 汇编代码,解决 AArch64 指令集架构到 64 位 RISC-V 指令集架构汇编源程序的自动移植问题.

图 2 展示了该翻译工具的结构框架,包含 4 个模块,即预处理、指令映射、寄存器映射和辅助寄存器分配. 指令映射字典和寄存映射字典分别存储指令映射关系和寄存器的替换规则.

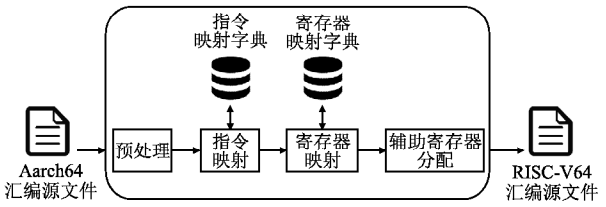

图 2 汇编翻译工具的框架

Fig. 2 Framework of assembler translation tool

预处理模块将汇编源码的文本文件解析为注释、伪操作、指令、标签并存储在相应的数据结构当中,并对汇编源源文件进行分块、常量符号统计、流程控制指令优化检测、寄存器使用情况统计等. 指令映射模块根据指令映射字典中的映射关系将 AArch64 指令替换为 Risc-V64 指令序列. 寄存器映射模块根据寄存器映射关系对指令中的寄存器进行替换. AArch64 指令向 RISC-V64 指令翻译过程中,由于部分 AArch64 指令的功能需要多条 RISC-V 指令实现,需要额外的寄存器传递多条指令产生的中间临时数据,但由于 AArch64 与 RISC-V64 架构寄存器的总数量相近,RISC-V 架构没有充分的寄存器存储这些中间数据,辅助寄存器模块根据预处理模块统计的寄存器使用情况实现了存储中间数据的辅助寄存器的分配功能.

下面以一个的例子阐述汇编自动翻译工具的工作流程,如图 3 所示.

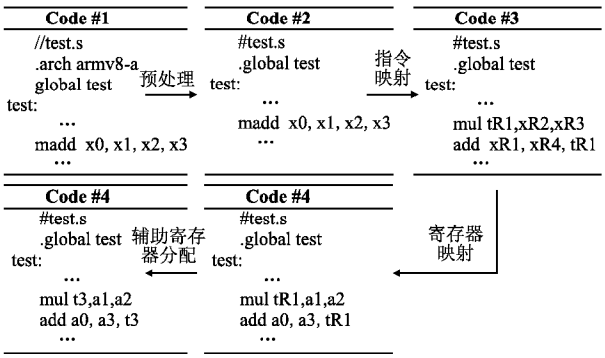

图 3 汇编翻译工具工作流程

Fig. 3 Workflow of assembler translation tool

1)Code#1 是 AArch64 部分汇编源程序,预处理模块将 Code#1 中的注释格式进行修改,且删除了伪操作. arch armv8-a,最终生成 Code#2.

2)指令映射模块先将 Code#2 中的指令转换为 AArch64 中间指令,再通过查询指令映射字典,将 AArch64 中间指令替换成 RISC-V64 的中间指令,即先将指令 madd x0,x1,x2,x3 转换为 madd xR1,xR2,xR3,xR4,然后替换为两条 RISC-V

中间指令 mul tR1,xR2,xR3 和 add xR1,xR4,tR1 得到 Code#3.

3)寄存器映射模块通过查询寄存器映射字典,对 Code#3 中具有映射关系的寄存器进行替换,即将 xR1 替换为 a0、xR2 替换为 a1、xR3 替换 a2,、xR4 替换成 a3,得到 Code#4.

4)辅助寄存器模块对整个程序的寄存器使用情况进行分析,选择适当的寄存器存储指令翻译过程需要存储的临时数据,将 Code#4 中 tR1 替换为 t3 得到 Code#5,也就是最终的 RISC-V64 汇编代码.

2　AArch64 与 RISC-V64 汇编差异分析

AArch64 架构到 RISC-V64 架构汇编的转换,最主要的是指令到指令的映射. 下面将从指令的操作数和指令的设计特点来介绍两种架构汇编差异.

2.1　操作数差异

AArch64 和 RISC-V64 操作数的差异主要存在于立即数操作数和寄存器操作数两方面.

2.1.1　立即数差异

立即数操作数的差异体现在立即数表示范围不同,AArch64 的立即数范围为 0 ~ 4095,RISC-V64 立即数的范围为-2048 ~ 2047,所以对于不同范围的立即数采用不同的映射方式.

2.1.2　寄存器差异

寄存器的差异体现在指令对于不同长度寄存器的使用方式以及调用约定对于各个寄存器的用法规定不同.

AArch64 包括 31 个 64 位宽的通用寄存器 X0 ~ X30 以及栈指针寄存器(SP)和程序计数器(PC),用 W 来表示对应寄存器的低 32 位,如 W0 表示 X0 的低 32 位数据. AArch64 包含 32 个 64 位双精度浮点寄存器 D0 ~ D31,用 S 来表示对应寄存器的低 32 位,代表 32 位单精度浮点寄存器,如 S0 表示 D0 的低 32 位数据[10].

RISC-V64 包括 32 个 64 位宽的通用寄存器 X0 ~ X31 和程序计数器(PC),32 个 64 位浮点寄存器 f0 ~ f31,RISC-V64 的 64 位通用寄存器和浮点寄存器也可以作为 32 位寄存器使用,但与 AArch64 不同,RISC-V64 不是用寄存器名区分操作数的长度而是通过操作码后缀,如表 1 所示,为两种架构对于不同长度操作数指令对比示例.

表 1　不同长度寄存器的指令对比

Table 1　Instruction comparison of different length registers

寄存器类型	AArch64	RISC-V64
32 位整型	add w1,w2,w3	addw x1,x2,x3
64 位整型	add x1,x2,x3	add x1,x2,x3
32 位单精度浮点	fadd s1,s2,s3	fadd. s f1,f2,f3
64 位双精度浮点	fadd d1,d2,d3	fadd. d f1,f2,f3

应用程序二进制接口(Application Binary Interface,ABI)定义了应用程序和操作系统之间或其他应用程序的低级接口[11]. ABI 规定了函数调用约定,函数调用时,调用函数将参数传递给被调用函数,被调用函数运行结束后将返回值返回给调用函数. 函数调用约定规定了函数调用过程中参数的传递顺序、参数的传递方式、寄存器分配方式. 函数调用约定,将寄存器主要分为,参数寄存器,保存寄存器,临时寄存器,及特殊用途寄存器[12].

参数寄存器:函数调用过程中,两个模块之间通过参数寄存器传递参数. 临时寄存器:在函数调用之前,调用函数需要在自己的栈上对这些寄存器进行备份,被调用函数可以直接修改临时寄存器. 保存寄存器:被调用函数在使用保存寄存器时需先将保存寄存器中的数据备份到自己的栈上,在被调用函数执行结束返回调用函数之前将备份的数据恢复到保存寄存器中.

表 2　AArch64 与 RISC-V64 通用寄存器调用约定

Table 2　AArch64 and RISC-V64 general register call convention

架构	参数寄存器	临时寄存器	保存寄存器	特殊用途
AArch64	X0—X7/ W0—W7	X9—X15/ W9—W15	X19—X28/ W19—W28	sp、lr 等
RISC-V	A0—A7	T0—T6	S1—S11	sp、ra 等

表 3　AArch64 与 RISC-V64 浮点寄存器调用约定

Table 3　AArch64 and RISC-V64 floating point register call convention

架构	参数寄存器	临时寄存器	保存寄存器
AArch64	S0—S7/ D0—D7	S8—S15/ D8—D15	S16—S31/ D16—D31
RISC-V	FA0—FA7	FT0—FT11	FS0—FS11

表 2、表 3 为 AArch64 和 RISC-V64 的寄存器调用约定比较,为了区分两种架构寄存器的名字,RISC-V64 寄存器采用了 ABI 规定的寄存器别名. 由于调用约定的不同,两种架构每一类的寄存器数量不完全相同.

2.2　指令差异

汇编指令的一般格式为:

< Opcode > < Rd > , < Rn > {, Operand2}

其中,Opcode 为操作码,说明指令需要执行的操作类型;Rd 为目标寄存器,指令结果写入该寄存器;Rn 为第 1 个操作数的寄存器;Oprand2:第 2 个操作数. AArch64 拥有灵活的第 2 个操作数,AArch64 的第 2 操作数可以是立即数、寄存器或者带有移位操作的寄存器. 当第 2 操作数是带有移位操作的寄存器时,AArch64 的一条指令可以先对一个寄存器中的寄存器进行移位操作,再进行这个指令的主操作,如 add x1,x2,x3,lsl 8 指令,而 RISC-V 单条指令无法同时实现移位和其他操作,需要多条指令才能实现 AArch64 指令的功能,如 sslli t4,a3,8 和 add a1,a2,t4 两条指令实现了上述 AArch64 一条指令的功能.

AArch64 和 RISC-V64 虽然都是精简指令集,但相对于 AArch64,RISC-V 更加精简. 如果一个功能可以通过已有的多条指令实现,RISC-V64 就不会为这个功能单独设计指令,而 AArch64 会为部分这类指令设计单独指令,如 AArch64 的整型乘加指令 madd x1,x2,x3,x4,该指令先将寄存器 x2 与寄存器 x3 指令相乘,之后将乘积与寄存器 x4 相加,结果保存在寄存器 x1 中. 而 RISC-V 没有为该操作设计单独指令,而是用已有的加指令和乘指令两条指令实现该功能,即 mul t3,a2,a3 和 add a1,t3,a4.

AArch64 的流程控制使用标志位机制[13](N、C、Z、V 标

志位),操作码后缀为 S 的指令或者比较指令可以修改标志位,后续带有条件码的指令可以根据标志位的值进行条件跳转,条件码为标志位的不同组合,如表 4 所示. RISC-V 分支指令不采用标志码,没有标志位机制,而是将比较和跳转判断合并为一条指令,如 AArch64 实现条件跳转需两条指令 cmp x1,x2 和 bge LABEL,而 RISC-V64 使用 bge x1,x2,LABEL 一条指令就可实现条件跳转.

表 4 AArch64 条件码

Table 4 AArch64 condition code

条件码	标志位	定 义
EQ	Z = 1	相等
NE	Z = 0	不相等
CS	C = 1	无符号大于或等于
CC	C = 0	无符号小于
MI	N = 1	负值
PL	N = 0	正值或 0
VS	V = 1	溢出
VC	V = 0	无溢出
HI	C = 1 且 Z = 0	无符号大于
LS	C = 0 或 Z = 1	无符号小于或等于
GE	N = V	无符号大于或等于
LT	N! = V	有符号小于
GT	Z = 0 且 N = V	有符号大于
LE	Z = 1 或 N! = V	有符号小于或等于

3 汇编翻译的实现

根据汇编翻译工具的框架的设计以及两种架构的汇编差异,对各个模块进行了具体的实现. 下面分别对预处理模块、寄存器映射模块和寄存器映射字典、指令映射模块和指令映射字典以及辅助寄存器分配模块的设计与实现做具体介绍.

3.1 预处理

预处理模块首先对 AArch64 架构的汇编源文件进行解析,并将解析后的数据存入相应的数据结构中,之后对汇编源码的各个部分进行处理,以便于后续对程序的翻译. 主要预处理分为 6 部分:

1)注释的修改:因 RISC-V GNU GCC[14] 不支持“//”注释符,所以需要改为“#”注释符,本工具选择在翻译过程中保留源程序的注释,以方便开发人员理解程序.

2)伪操作修改:对两种架构不同的伪操作进行修改,如. xword 替换为. dword.

3)程序分块:标签为开始点,以其他标签、ret 指令或者文件尾为结束点,将源程序分为若干块,之后根据各块之间的调用关系,将各块组合为函数块.

4)寄存器统计:对分块后的各个函数块进行寄存器的使用情况统计,并提供给辅助寄存器分配模块使用.

5)常量统计:由于 AArch64 和 RISC-V64 架构的立即数表示范围不同,预处理模块要对程序中的常量进行统计,后续翻译过程中会对不同范围的常量采用不同的翻译方式.

6)流程控制指令优化检测:模拟程序执行流,统计所有执行流中每条条件跳转指令执行前影响该指令执行时标志位的指令,如果多于一条标记为不能优化,如果只有一条则标记为可以优化.

3.2 指令映射字典与指令映射

翻译工具采用映射字典的形式将翻译程序与指令映射关系松耦合,只需在指令映射字典中按照指令映射规范对指令映射关系进行增加和修改就可以达到对翻译工具所支持指令的增加和修改的目的.

映射字典的键和值都采用了中间指令的形式,为每种指令操作数赋予一个别名,如表 5 所示为部分操作数类型及其别名,以 wRn 为例,wRn 代表一条指令中的第 n 个 32 位整型寄存器,如 add wR1,wR2,wR3 代表将两个 32 位整数相加,结果保存在 wR1 寄存器中.

表 5 指令操作数别名

Table 5 Instruction operand alias

操作数类型	别 名
32 位整型寄存器	wRn
64 位整型寄存器	xRn
32 位单精度浮点寄存器	sRn
64 位双精度浮点寄存器	dRn
立即数	Immn
标志位模拟寄存器	zTagR、nTagR、vTagR、cTagR
中间数据寄存器	tRn

不同类型 AArch64 第 2 操作数的指令与 RISC-V64 指令的映射字典如表 6 所示,以加法指令为例,AArch64 加法指令的第 2 操作数为寄存器时,两种架构可以找到指令一一对应,当第 2 操作数为立即数时,由于 AArch64 的立即数范围为 0 ~ 4095,RISC-V64 立即数的范围为-2048 ~ 2047,所以对于 RISC-V 立即数表示范围内的指令,只需一条指令就能实现立即数加的功能,而对于超出 RISC-V 立即数表示范围的指令需要两条指令才能实现 AArch64 指令的加法操作. 当第 2 操作数为带移位的寄存器时,需要两条 RISC-V 指令,第 1 条指令实现移位操作,第 2 条指令实现加操作.

表 6 不同类型 AArch64 第 2 操作数的映射字典

Table 6 Mapping dictionaries for the second operand of different types of AArch64

Oprand2	AArch64	RiscV64
寄存器	add xR1,xR2,xR3	add xR1,xR2,xR3
立即数(大于 0 小于 2048)	add xR1,xR2,imm1	addi xR1,xR2,imm1
立即数(大于等于 2048)	add xR1,xR2,IMM1	li tR1,IMM1add xR1,xR2,IMM1
带移位操作的寄存器	add xR1, xR2, xR3, lsl imm1	slli tR1,xR3,imm1add xR1,xR2,tR1

如表 7 所示为 AArch64 的复杂指令与 RISC-V64 指令的映射字典,通过分配辅助的临时寄存器传递多条 RISC-V64 指令的中间数据,实现 AArch64 一条指令的功能.

虽然 RISC-V 将比较和跳转判断合并为一条指令,但无法覆盖 AArch64 的所有条件码,且在迁移过程中存在无法直接用一条 RISC-V 比较跳转指令替换 AArch64 比较指令和条件跳转指令特殊情况. 如代码 1 所示,bge label1 指令执行之前,有两种情况,第 1 种为 bge label 根据 cmp x1,x2 执行后的标志位信息进行判断跳转,第 2 种为 bge label1 根据 cmp x3,

x4 执行后的标志位信息进行判断跳转,bge label1 与两个比

表 7　AArch64 复杂指令映射字典

Table 7　AArch64 complex instruction mapping dictionary

功能	AArch64	RiscV64
整型乘加	madd xR1, xR2, xR3,xR4	mul tR1,xR2,xR3add xR1,tR1,xR4
立即数和 32 位寄存器与操作	and wR1,wR2,imm1	li tR1,imm0and wR1,wR2,tR1

较指令关联,并且这两个比较指令比较的是不同寄存器中的内容,因此不能将 AArch64 的比较指令和跳转指令用一条 RISC-V64 比较跳转指令.

代码 1. 无法优化流程控制指令

```
    …
    cmp x1,x2
    …
loop1:
    …
    bge label1
    …
    cmp x3,x4
    …
    bls loop1
    …
```

所以对于流程控制指令采用两种方式,对于能够找到对应指令的条件码的分支指令且预处理流程控制指令优化检测为可优化,将比较指令和条件跳转指令用一条指令替换. 否则,采用模拟标志位的方式进行翻译[15],即比较指令和条件跳转指令单独翻译,如表 8 所示. 在内存中分配 4 个内存区域分别代表 AArch64 架构中的 N,C、Z、V 标志位,对于 cmp 指令的翻译为计算出它对 N、C、Z、V 的值,先存入 nTagR、cTagR、zTagR、vTagR 中,之后存入内存对应位置,而对于条件跳转指令的翻译为先从内存中取出 N、C、Z、V 存入 TagR、cTagR、zTagR、vTagR 中,用 RISC-V64 条件跳转指令中比较这些标志位值进行跳转.

表 8　流程控制指令映射字典

Table 8　Flow control instruction mapping dictionary

AArch64	RISC-V64
cmp xR1,xR2bge label1	bge xR1,xR2,label1
cmp xR1,xR2	neg tR1,xR2
	add tR2,xR1,tR1
	slti nTagR,tR2,0
	sltu zTagR,x0,tR2
	sltu cTagR,tR2,xR1
	slti tR3,xR1,0
	slt tR2,tR2,tR1
	xor vTagR,tR2,tR3
bge label1	beq nTagR,vTagR,label1

目前工具映射字典中除流程控制指令外,AArch64 与 RISC-V64 指令的映射关系都为一对一或一对多的映射关系,翻译过程中不会产生二义性. 对于流程控制指令的翻译在预处理模块中单独设置了跳转指令优化模块,检测多对一翻译是否有二义性,如果存在二义性则采用模拟标志位机制的方式,对比较指令和条件跳转指令分别进行一对多翻译,如果没有二义性就采用比较指令和条件跳转指令用一条 RISC-V 指令翻译的方式. 对流程控制指令采用以上两种方式相结合的方式翻译能在保证翻译正确的前提下,减小翻译后指令数量.

指令映射模块首先将预处理后得到的指令转换为中间指令形式,即映射字典中的键,同时保存寄存器别名与指令操作数的对应关系以及立即数别名与立即数的对应关系. 然后查询指令映射字典,将 AArch64 中间指令替换为 RISC-V64 的中间指令.

3.3　寄存器映射字典与寄存器映射

由于 RISC-V64 的指令设计较 AArch64 更加精简,部分 AArch64 指令的功能需要多条 RISC-V 指令才能实现,所以在翻译过程中就需要额外的寄存器存储中间数据,以及两种架构 ABI 规定的各类型寄存器的数量不完全相同,所以寄存器不能全部一一映射. 寄存器的映射采用两种方法:寄存器直接映射和内存模拟寄存器.

寄存器直接映射方法是 AArch64 的寄存器在 RISC-V 中有一个专用的寄存器与之对应. 内存模拟寄存器方法是在 RISC-V 的内存中有一个专用的内存空间,AArch64 指令操作对该类寄存器进行操作时对应的 RISC-V 指令对对应的内存进行相应操作,但是这种方法需要额外的开销,即内存读写.

表 9　存器映射字典

Table 9　Register mapping dictionary

类型	AArch64	RISC-V	调用约定
寄存器直接映射	X0-X7/W0-W7	A0-A7	参数寄存器
	x9-x11/w9-w11	t0-t2	临时寄存器
	x19-x28/w19-w28	s1-s10	保存寄存器
	x29/w29	s0(fp)	帧指针寄存器
	x30/w30	Ra	返回地址寄存器
	x31	Sp	栈指针寄存器
	S0-S7/D0-D7	FA0-FA7	浮点参数寄存器
	S8-S15/D8-D15	FS0-FS7	浮点保存寄存器
	S16-S27/D16-D27	FT0-FT11	浮点临时寄存器
	S28-S31/D28-D31	FS8-FS11	——
内存模拟寄存器	x8(xr)	1	保存子程序返回地址
	x12-x15	2-5	临时寄存器
	x16-x17	6-7	子程序内部调用寄存器
	x18	8	平台寄存器

寄存器映射字典见表 9,两种架构的参数寄存器数量相同,所以参数寄存器采用寄存器映射的方式. 临时寄存器部分采用寄存器映射,部分采用内存模拟寄存器,这是因为本工具保留了 RISC-V64 的 4 个临时寄存器 t3-t6,专用于保存中间数据,没有与 AArch64 的临时寄存器进行映射,采用临时寄存器保存中间数据不需要进行备份,可以减少内存读写和翻译后指令数量. RISC-V64 保存寄存器的数量多于 AArch64,保存寄存器全部采用寄存器直接映射的方式. 帧指针寄存器,返回地址寄存器和栈指针寄存器,这 3 种寄存器两种架构都有对应的寄存器[16],所以采用寄存器直接映射. AArch64 临时浮点寄存器多于 RISC-V 临时浮点寄存器,保存寄存器少于 RISC-V 的保存寄存器,采用寄存器直接映射方法后,AArch64 剩余 4 个临时浮点寄存器,RISC-V 剩余 4 个保存浮

点寄存器,用 RISC-V 这 4 个保存寄存器映射 AArch64 的剩余临时浮点寄存器,但是在程序入口处需先对这 4 个保存浮点寄存器进行备份,函数调用返回时恢复保存浮点寄存器,保证调用程序能正常运行.

本工具将模拟寄存器,模拟标志位,寄存器数据备份保存内存的 tdata 段,该段是线程局部存储段(Thread Local Storage,TLS),TLS 是一种为不同线程分配不同对象的机制,即该段所存储的数据仅当前线程访问,不会被其他线程修改. 在 tdata 段声明若干 8 字节连续空间,每个空间与一个模拟寄存器,模拟标志位,被备份的寄存器相对应. 工具将该连续空间的首地址指针保存在寄存器 t6 中,即在函数入口处插入指令 lui t6,% tprel_hi(SimRegion) 和 add t6, t6, tp,% tprel_add(SimRegion),当需要对该区域的数据进行操作时,使用 ld tR0,% tprel_lo(SimRegion + offset)(t6)指令和 sd tR0,% tprel_lo(SimRegion + offset)(t6)指令对内存中的数据进行存取,其中 tR0 是辅助寄存器分配模块分配的寄存器,offset 是数据相对于 SimRegion 的偏移量,声明 tdata 内存空间代码如代码 2 所示.

代码 2. tdata 内存空间声明

```
    . section . tdata
SimRegion:
    . dword 0
    . dword 0
    …
```

寄存器映射模块通过查询指令映射字典和指令映射模块保存的寄存器别名和寄存器的映射关系,将除中间数据寄存器、标志位模拟寄存器和内存模拟寄存器外的别名替换为对应的 RISC-V64 寄存器,以及将查询指令映射模块保存期立即数别名与立即数的映射关系,将立即数别名替换为对应的立即数.

3.4 辅助寄存器分配

辅助寄存器分配模块为中间数据寄存器 tRn,标志位模拟寄存器 zTagR、nTagR、cTagR、VTagR 以及内存模拟寄存器分配寄存器,分配策略如下:

表 10 UART 初始化函数翻译前后对比

Table 10 Comparison before and after translation of UART initialization function

AArch64

```
    . global uart16550Init
      . text
uart16550Init:
    // Check the input base address
    cbz   x0,init_fail
    // Check baud rate and uart clock for sanity
    cbz   w1,init_fail
    cbz   w2,init_fail
    // Program the baudrate
    // Divisor =   Uart clock / (16 * baudrate)
    lsl   w2,w2,#4
    udiv   w2,w1,w2
    and   w1,w2,#0xff
    lsr   w2,w2,#8
    and   w2,w2,#0xff
    ldr   w3,[x0,#UARTLCR]
    orr   w3,w3,#UARTLCR_DLA
    str   w3,[x0,#UARTLCR]
    str   w1,[x0,#UARTDLL]   // program DLL
    str   w2,[x0,#UARTDLLM]   // program DLLM
    mov   w2,#~UARTLCR_DLAB
    and   w3,w3,w2
    str   w3,[x0,#UARTLCR]
    // data length 8bit,1 stop bit
    mov   w3,#3
    str   w3,[x0,#UARTLCR]
    //no interrupt
    mov   w3,#0
    str   w3,[x0,#UARTIER]
#ifdef TI_16550_MDR_QUIRK
    //UART must be enabled on some platforms via the MDR register
    str   w3,[x0,#UARTMDR1]
    #endif // TI_16550_MDR_QUIRK
    //enable fifo,DMA,FIXMELIY_TODO is need to enable fifo,DMA?
    mov   w3,#(UARTFCR_FIFOEN | UARTFCR_DMAEN)
    str   w3,[x0,#UARTFCR]
    // DTR + RTS
    mov   w3,#3
    str   w3,[x0,#UARTMCR]
    mov   w0,#1
    ret
init_fail:
    mov   w0,#0
    ret
```

RISCV64

```
    . global uart16550Init
    . text
uart16550Init:
    # Check the input base address
    beqz a0,init_fail
    # Check baud rate and uart clock for sanity
    beqz a1,init_fail
    beqz a2,init_fail
    # Program the baudrate
    # Divisor =   Uart clock / (16 * baudrate)
    slliw a2,a2,4
    divuw a2,a1,a2
    li t4,0xff
    and a1,a2,t4
    srliw a2,a2,8
    li t4,0xff
    and a2,a2,t4
    lw a3,UARTLCR(a0)
    li t4,UARTLCR_DLAB
    or a3,a3,t4
    sw a3,UARTLCR(a0)
    sw a1,UARTDLL(a0)# program DLL
    sw a2,UARTDLLM(a0)# program DLLM
    li a2,~UARTLCR_DLAB
    and a3,a3,a2
    sw a3,UARTLCR(a0)
    # data length 8bit,1 stop bit
    li a3,3
    sw a3,UARTLCR(a0)
    # no interrupt
    li a3,0
    sw a3,UARTIER(a0)
#ifdef TI_16550_MDR_QUIRK
    # UART must be enabled on some platforms via the MDR register
    sw a3,UARTMDR1(a0)
#endif # TI_16550_MDR_QUIRK
    # enable fifo,DMA,FIXMELIY_TODO is need to enable fifo,DMA?
    li a3,(UARTFCR_FIFOEN | UARTFCR_DMAEN)
    sw a3,UARTFCR(a0)
    # DTR + RTS
    li a3,3
    sw a3,UARTMCR(a0)
    li a0,1
    ret
init_fail:
    li a0,0
    ret
```

1)辅助寄存器的数量小于等于 3,使用 RISC-V 保留的 t3、t4、t5 寄存器

2)辅助寄存器的数量大于 3

a)预处理时统计 AArch64 的寄存器参数寄存器以及临时寄存器中使用次数为 0 的寄存器所映射的 RISC-V 寄存器

b)将该指令序列没有用到的其他寄存器备份到内存,使用完之后恢复

4　实验验证

本节展示工具的实际应用场景示例,以及使用了具有代表性的覆盖了常用指令类型的汇编程序进行测试,并与现有工具 ARM2RISCV 进行翻译效果对比.

4.1　实际场景示例

如表 10 所示,为本工具在实际开发场景中的具体应用,为某国产实时操作系统内核从 AArch64 架构向 RISC-V64 架构迁移的示例,该源码功能为实现 UART 初始化,翻译后的代码能够正确实现 UART 初始化功能.

4.2　实验设计

本文使用 QEMU 搭建测试环境. QEMU[17]是一个开源计算机系统模拟器,QEMU 的机器模式可以运行 RISC-V64 架构的 LINUX 操作系统. 本文测试环境最底层为 X86 架构 Ubuntu 20.04,通过 QEMU 的机器模式运行 RISC-V64 的 Ubuntu 22.04 操作系统. 本文所有测试用例都运行在该 RISC-V64 架构的 Ubuntu 上.

本文从相关开源项目中选取了 6 段典型汇编程序作为测试用例,其中包含了常用的指令类型,如表 11 所示,其中○表示该测试用例包含该指令类型,╳表示该测试用例不包含该指令类型. 测试用例涵盖了算术运算指令(arithmetic)、逻辑运算指令(logical)、移位指令(shift)、存取指令(load/store)、系统调用指令(system call)、分支指令(branch)等. 测试用例覆盖了常用的指令类型,生成的目标代码包含在 RV64G 内.

表 11　测试用例覆盖指令类型

Table 11　Test case covers instruction type

编号	指令类型									
	Aithmetic	Load/store	Logical	Shift	Atomic	SystemCall	Branch	Privileged	Register Transfer	Conditonal Selection
1 算数运算	○	○	○	○	╳	╳	╳	╳	○	╳
2 原子运算	○	○	╳	╳	○	╳	╳	╳	╳	╳
3 输入输出	○	○	╳	╳	╳	○	╳	╳	╳	╳
4 矩阵运算	○	○	╳	╳	╳	╳	○	╳	╳	╳
5 浮点比较	○	○	╳	╳	╳	╳	╳	╳	╳	○
6 特权指令	○	○	╳	╳	╳	╳	╳	○	╳	╳

4.3　实验结果

如表 12 所示,为本工具与 ARM2RISCV 测试结果对比. 测试用例 1 和 2 结果显示,对于常用的算术运算、存储、逻辑运算、寄存器转换和原子指令两种工具都能支持. 测试用例 3 显示,ARM2RISCV 不支持系统调用指令的翻译,而本工具能够支持系统调用指令的翻译. 测试用例 4 和 5,对矩阵预算和浮点运算中常用的分支指令和条件选择指令进行测试,因为 RISC-V64 的条件跳转指令无法全部覆盖 AArch64 的条件码,ARM2RISCV 只对能对应条件码的 AArch64 条件跳转指令支持翻译,存在指令翻译错误的情况,而本工具对于没有对应条件码的 AArch64 条件跳转指令采用模拟标志位的方式进行支持. 测试用例 6 是对特权指令进行测试,由于两种架构控制状态寄存器(Control and Status Register,CSR)差异较大,无法构建寄存器映射关系,两种工具目前都无法支持.

表 12　测试结果

Table 12　Test results

测试用例编号	结　果	
	本工具	ARM2RISCV
1 算数运算	通过	通过
2 原子运算	通过	通过
3 输入输出	通过	不支持
4 矩阵运算	通过	错误
5 浮点比较	通过	错误
6 特权指令	不支持	不支持

ARM2RISCV 支持 101 条指令,但扩展指令困难,每增加对一条指令的支持需要对程序进行修改,新增一个方法单独对该指令进行翻译. 而本工具目前支持 230 条指令,采用指令映射字典的方式,除流程控制指令外,将工具本身与指令映射关系松耦合,增加新的指令支持只需按照映射字典格式向指令映射字典中加入新的指令映射关系就可实现.

5　结　语

本文设计了一个高可拓展性的汇编源程序自动翻译工具,实现了 AArch64 架构到 RISC-V64 架构的汇编源程序自动翻译. 根据 AArch64 架构和 RISC-V64 架构的 ABI 规定差异,设计了寄存器映射字典,通过寄存器直接映射和内存模拟寄存器两种方式,实现寄存器的映射. 根据 AArch64 架构和 RISC-V 架构汇编指令的差异设计了汇编指令映射字典,实现了指令映射关系与翻译程序的解耦,实现了工具的高可拓展性. 支持的指令数量是现有工具 ARM2RISCV 的 2.3 倍,且更易拓展,后续会将指令映射规则上传到开源社区,让更多的迁移工作者加入进来,拓展指令映射字典. 该工具可以帮助汇编源程序移植人员提高效率,加快项目移植进度,也可以为熟悉 AArch64 架构的程序员学习 RISC-V64 架构提供帮助.

References:

[1] Waterman A, Lee Y, Patterson D A. The RISC-V instruction set manual volume I: unprivileged ISA [EB/OL]. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf,2023-02-15.

[2] Chen Chen, Xiang Xiao-yan, Liu Chang, et al. Xuantie-910: a com-

mercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: Industrial product[C]//ACM/IEEE 47th Annual International Symposium on Computer Architecture(ISCA),2020:52-64.

[3] BAO Y G,SUN N H. Opportunities and challenges of building CPU ecosystem with open-source mode[J]. Bulletin of Chinese Academy of Sciences,2022,37(1):24-29.

[4] Mezger B W,Santos D A,Dilillo L,et al. A survey of the RISC-V architecture software support [J]. IEEE Access, 2022, 10: 51394-51411.

[5] LIU C,WU Y J,WU J Z,et al. Survey on RISC-V system architecture research[J]. Journal of Software,2021,32(12):3992-4024.

[6] Kusswurm D. Modern arm assembly language programming: covers Armv8-A 32-bit,64-bit,and SIMD[M]. New York: Apress Media, 2020.

[7] Huawei Technologies Co. ,Ltd. Kunpeng code migration tool[EB/OL]. https://www.hikunpeng.com/document/detail/zh/kunpengdevps/porting/usermanual/kunpengpt_06_0002.html,2023-02-15.

[8] CAO H,GUO S Z,LIU D,et al. Automatic portion of basic mathematics Library for 64-bit RISC-V[J]. Computer Science,2021,48(6):41-47.

[9] Moshe Schorr,Matan Ivgi. ARM2RISCV: an arm to RISC-V transpiler[EB/OL]. https://github.com/schorrm/arm2riscv/blob/master/docs/project_documentation.md,2023-02-15.

[10] ARM Holdings plc. Arm architecture reference manual for A-profile architecture[EB/OL]. https://developer.arm.com/documentation/ddi0487/latest,2023-02-15.

[11] LIU Y S. Design and development of ARM automatic compilation tools[D]. Xi'an: Xidian University,2014.

[12] HU W W,WANG W X,WU R Y,et al. Loongson instruction set architecture technology[J]. Journal of Computer Research and Development,2023,60(1):2-16.

[13] Breternitz M,Manikonda A,Ommermann M,et al. Design tradeoffs and experience with motorola PowerPC migration tools[C]//Proceedings International Conference on Computer Design, VLSI in Computers and Processors,Austin,TX,USA,1996:301-308.

[14] GCC. The GNU compilers[EB/OL]. https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/,2023-02-15.

[15] Cempron J P,Salinas C,Conzales J B,et al. A static transliteration approach for assembly language translation[C]//IEEE Region 10 Symposium(TENSYMP),Bali,Indonesia,2016:332-336.

[16] CHENG Y H,HUANG L B,CUI Y J,et al. Design and implementation of embedded multiple-ISA processor based on RISC-V[J]. Acta Electronica Sincia,2021,49(11):2081-2089.

[17] QEMU. A generic and open source machine emulator and virtualizer[EB/OL]. https://qemu.org,2023-02-15.

附中文参考文献:

[3] 包云岗,孙凝晖. 开源芯片生态技术体系构建面临的机遇与挑战[J]. 中国科学院院刊,2022,37(1):24-29.

[5] 刘 畅,武延军,吴敬征,等. RISC-V 指令集架构研究综述[J]. 软件学报,2021,32(12):3992-4024.

[7] 华为技术有限公司. 鲲鹏代码迁移工具[EB/OL]. https://www.hikunpeng.com/document/detail/zh/kunpengdevps/porting/usermanual/kunpengpt_06_0002.html,2023-02-15.

[8] 曹 浩,郭绍忠,刘 聃,等. 面向 64 位 RISC-V 的基础数学库自动化移植[J]. 计算机科学,2021,48(6):41-47.

[11] 刘跃生. ARM 自动编译工具的设计与开发[D]. 西安:西安电子科技大学,2014.

[12] 胡伟武,汪文祥,吴瑞阳,等. 龙芯指令系统架构技术[J]. 计算机研究与发展,2023,60(1):2-16.

[16] 成元虎,黄立波,崔益俊,等. 基于 RISC-V 的嵌入式多指令集处理器设计及实现[J]. 电子学报,2021,49(11):2081-2089.