软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(8):3853−3869 [doi: 10.13328/j.cnki.jos.006595]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于 K Framework 的向量化机器学习指令语义形式化*

黄厚华, 刘嘉祥, 施晓牧

(深圳大学 计算机与软件学院, 广东 深圳 518060)

通信作者: 施晓牧, E-mail: xshi0811@gmail.com

摘　要: ARM 针对 ARMv8.1-M 微处理器架构推出基于 M-Profile 向量化扩展方案的技术, 并命名为 ARM Helium, 声明能为 ARM Cortex-M 处理器提升达 15 倍的机器学习性能. 随着物联网的高速发展, 微处理器指令执行正确性尤为重要. 指令集的官方手册作为芯片模拟程序, 片上应用程序开发的依据, 是程序正确性基本保障. 主要介绍利用可执行语义框架 K Framework 对 ARMv8.1-M 官方参考手册中向量化机器学习指令的语义正确性研究. 基于 ARMv8.1-M 的官方参考手册自动提取指令集中描述向量化机器学习指令执行过程的伪代码, 并将其转换为形式化语义转换规则. 通过 K Framework 提供的可执行框架利用测试用例, 验证机器学习指令算数运算执行的正确性.

关键词: ARMv8.1-M 架构; 向量化指令; 机器学习; K Framework; 形式化语义

中图法分类号: TP18

中文引用格式: 黄厚华, 刘嘉祥, 施晓牧. 基于K Framework的向量化机器学习指令语义形式化. 软件学报, 2023, 34(8): 3853–3869. http://www.jos.org.cn/1000-9825/6595.htm

英文引用格式: Huang HH, Liu JX, Shi XM. Semantics Formalization of Vectorized Machine Learning Instructions in K Framework. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3853–3869 (in Chinese). http://www.jos.org.cn/1000-9825/6595.htm

Semantics Formalization of Vectorized Machine Learning Instructions in K Framework

HUANG Hou-Hua, LIU Jia-Xiang, SHI Xiao-Mu

(College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China)

Abstract: ARM develops an M-Profile vector extension solution in terms of ARMv8.1-M micro processor architecture and names it ARM Helium. It is declared that ARM Helium can increase the machine learning performance of the ARM Cortex-M processor by up to 15 times. As the Internet of Things develops rapidly, the correct execution of microprocessors is important. In addition, the official manual of instruction sets provides a basis for developing chip simulators and on-chip applications, and thus it is the basic guarantee of program correctness. This study introduces these mantic correctness of vectorized machine learning instructions in the official manual of the ARMv8.1-M architecture by using K Framework. Furthermore, the study automatically extracts pseudo codes describing the vectorized machine learning instruction operation based on the manual and then formalizes them in semantics rules. With the executable framework provided by K Framework, the correctness of machine learning instructions in arithmetic operation is verified.

Key words: ARMv8.1-M architecture; vectorized instruction; machine learning; K Framework; formal semantics

ARM Helium 是 2019 年 ARM 推出的新技术. 其针对 ARMv8.1-M 版本的架构设计了 M-profile vector extension (MVE) 向量化扩展, 加强了 ARM Cortex-M 系列处理器在特定应用场景的计算性能. 例如, Helium 可以为 ARM Cortex-M 系列处理器提升 5 倍的信号处理性能以及 15 倍机器学习 (machine learning, ML) 性能, 其中能为终端设备深度学习的推理过程减少 50%–90% 的用时以及更少的能量消耗. Helium 指令集的特点是实现特定场景下频繁应用的运算向量化以加速复杂程序, 例如深度学习推理过程, 快速傅里叶变换, 滤波算法, 循环缓冲, 多项式展开, 密

* 基金项目: 深圳市科创委基础研究面上项目 (JCYJ20210324094202008); 国家自然科学基金 (62002228); 深圳市高等院校稳定支持计划 (20200810045225001)
本文由“智能系统的分析和验证”专题特约编辑明仲教授、张立军教授和秦胜潮教授推荐.
收稿时间: 2021-09-05; 修改时间: 2021-10-14; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28
CNKI 网络首发时间: 2023-01-19

码学大整数域运算等. ARM Cortex-M55 处理器是第 1 个采用 ARMv8.1-M 架构的处理器. STMicroelectronics 等公司已将 ARM Cortex-M55 集成到下一代人工智能终端及其生态环境. 开发人员已经在使用开源库 (如 CMSIS-DSP, CMSIS-NN) 和 ML 框架 (如用于微控制器的 TensorFlow Lite) 将机器学习应用程序移植到 Cortex-M 设备上, TensorFlow Lite 设计使用在内存占用非常小的处理器上运行机器学习模型, CMSIS-DSP 和 CMSIS-NN 库中添加了对 Cortex-M55 的支持, 这使得将 ML 应用程序移植到 Cortex-M55 变得更为容易, 开发人员可以使用它们熟悉的库和神经网络框架, 通过调用 Helium 技术的向量化指令集以及更强的数据处理能力, 提升机器学习性能. ARM 公司估计到 2022 年, 超过 20% 的物联网终端设备将拥有 ML 支持, 可以预计 Cortex-M55 在物联网中将会得到广泛应用. 由于物联网对于微处理器的需求量很大, 比如在无人机导航或控制, 终端人工智能, 机器人控制等, 因此若底层架构指令或官方文档出现错误, 将会造成巨大的隐患, 有可能使得物联网出现大规模的损失. 因此对于 ARMv8.1-M 微处理器架构而言, 其指令正确性尤为重要. 本文目的是验证 ARMv8.1-M 架构官方手册[1]中, ARM Helium 技术中向量化 ML 指令的语义正确性. 验证 Helium 技术的向量化 ML 指令, 存在以下的一些难点: (1) 与之前版本的 ARMv8 的 M 架构不同, ARMv8.1-M 的 Helium 技术的指令引入了新的加速数据存取以及操作的向量化特性; (2) 引入向量化寄存器及节拍 (Beat) 的概念; (3) 同一个助记符, 在操作寄存器类型不同时有不同的功能实现.

K Framework[2,3]作为实现语义形式化[4,5]的工具, 它具有以下优点: (1) 具备模块化的特性, 这有利于扩展现有架构的指令集, 当架构有新指令扩展时, 可以在已有的框架基础上实现新功能的语义扩展; (2) 可以自动生成可执行的解释器, 指令语义可以直接通过官方提供的测试集以及补充测试集进行测试, 在语义正确的基础下, 解释器等自动生成的工具具备 correct-by-construction 的特点, 帮助使用架构的开发者更快地熟悉指令功能; (3) 基于重写与 matching logic 理论, 提供程序中可达性等问题的验证能力; (4) K Framework 的标准模块支持整型, 二进制位向量和浮点数数据类型及相应的运算操作, 如数据类型转换, 位向量转换为浮点数或整型, 以及位向量的移位操作等, 有利于实现位向量指令语义的形式化. 所以本文应用 K Framework 作为向量化 ML 指令语义形式化的工具框架.

相关工作, 其中包括运用 K Framework 进行指令语义形式化的工作, 如 Dasgupta 等人[6]运用 K Framework 对 x86-64 架构的用户级指令集进行语义形式化并验证其语义正确性; 如 Wang 等人[7]在 K Framework 中定义了 RUST 语言的可执行语义 KRust 并成功测试了官方基准测试程序; 以及 C, Java 等语言上的实现. 另外相关工作还包括 ARMv8 架构指令集的验证工作, 如针对 ARMv8-A 架构指令进行语义形式化的工作, Armstrong 等人[8]运用 Sail 语言及其工具集对 ARMv8-A 架构的指令集体系结构 (ISA) 语义进行形式化并验证. Sail 是专用于描述 ISA[9] 的语言. 其作为中间语言负责将官方提供的 ISA 的架构规范语言 (ASL) 转换到用于仿真或验证的其他语言, 包括用于仿真的 C 与 OCaml, 用于验证的 Isabelle/HOL, HOL4, Coq. 用 Sail 实现指令语义的描述主要目的在于 ISA 仿真程序的生成, 或并发执行的研究. 由于这些用途, Sail 语言被设计的尽可能贴近 ARM 官方文本中的伪码, 且需要对 ISA 具体的特征建模. 而 K Framework 相比 Sail 用途更加广泛, K Framework 作为更为通用的语义描述语言更适用于描述特定应用的形式化语义. 在本工作中采用 K Framework 的优势有两点: (1) 相比 Sail, K Framework 可以根据需求对语义涉及的指令集架构的模型进行抽象, 如果我们关心的是指令伪码在运算操作上的正确性, 则对于一些内存管理与 IO 异常处理暂不建模也可以定义完整的指令语义规则; (2) K Framework 提供及时验证测试的能力, 可以根据现有语法语义及时生成解释器测试, 以及通过 Kprove 功能调用 SMT 验证语义规则满足规约. 而 Sail 的核心在于根据官方提供的官方架构描述 ASL, 直接生成可执行仿真程序, 或者生成可以用于验证的 Isabelle/HOL 等模型, 但对于 Sail 除了静态类型检测外其缺乏在建模过程中能及时测试验证的能力, 需要建立较完整的模型后才能通过转换成 C 来仿真测试, 或者转换成如 Coq 模型进行验证. 由于第一和第二条原因, Sail 的工作更加依赖于官方的 ASL, 如果没有该 ASL 则建模需要手动进行, 得到可以测试和验证工具的工作量很大, 不适合对于一个指令扩展集进行验证. Armstrong 等人[8]在 ARMv8-A 的工作依赖于官方提供的 ASL, 因此若想复用该自动化的工具链对其他架构进行指令语义的定义则需要官方提供该架构对应的 ASL, 但 ARMv8.1-M 的 ASL 并未发布. 另外, Reid 等人[10]对 ARMv8-A 和 ARMv8-M 架构官方手册进行规范并创建了 ASL, 对于 ARMv8-A 架构其开源网站上有长期维持架构版本的追踪与更新, 而 ARMv8-M 在其网站上没有开源, 且论文中对 M 架构进行规范的版本是 2006 年的官方手册. ARMv8 的 A 架构与 M 架构中的指令即使助记符相同, 但其功能却大不相同, 两

个架构差异很大. 因此, 迄今为止, 据我们所知还没有发现对 ARMv8.1-M 架构最新官方手册的指令语义进行形式化的工作. 本文运用 K Framework 对 ARMv8.1-M 架构最新版本手册 ARM Helium 技术的向量化 ML 指令进行语义形式化. 主要有以下 3 点贡献.

(1) 对 ARMv8.1-M 的最新版本架构扩展中 Helium 技术的向量化 ML 指令语义进行形式化验证.

(2) 在文档中自动提取向量化 ML 指令语义的伪代码, 利用 K Framework 的模块化定义其形式化语义.

(3) 通过官方提供的测试用例与自动生成的测试用例进行实验, 验证 ML 指令运算功能正确性.

本文第 1 节介绍了 K Framework 以及手册中向量化 ML 指令的相关知识. 第 2 节介绍了应用 K Framework 实现可执行的向量化 ML 指令语义的框架. 第 3 节讲述了 K Framework 定义向量化 ML 指令的语法与语义规则. 第 4 节描述了单指令语义形式化的具体实现. 第 5 节展示了关于可执行语义的实验. 第 6 节是总结与展望.

1 背景知识

本节首先简单介绍 K Framework 的背景知识, 然后介绍在 ARMv8.1-M 架构的官方手册中关于向量化 ML 指令语义的相关内容.

1.1 K Framework

K Framework (以下称 K 框架) 是基于重写逻辑的可执行语义工具框架, 它是一种用于设计和建模编程语言和软件/硬件系统的工具. K 框架的核心是一种称为 K 的编程, 建模和规范语言, 以及基于 matching logic 的验证能力. K 框架包括用于编译 K 描述的语义规则以构建解释器, 模型检查器, 验证器, 相关文档等的工具. 当给定语法与语义, K 框架可以自动生成该语言的解析器 (parser), 解释器 (interpreter) 以及演绎程序验证器 (deductive program verifiers) 等形式化分析工具, 不需要进行额外的开发. 运用 K 框架的解释器可以立即测试语言的语义以显著提高语义形式化开发的效率. 此外, 形式化分析工具有助于给定语言语义的形式化推导. 这在语义的适用性和语义本身的工程方面都有帮助.

简单来说, 在 K 框架中语言的语法 (syntax) 以传统的巴科斯范式 (BNF) 定义. 语言的语义 (semantics) 被定义为规则 (rule), 另外由配置 (configuration) 来定义状态. 配置是程序代码和状态规约的集合, 直观说它是一个元组, 这个元组的元素 (称为 cells) 代表着语义中的状态集合, 例如内存或寄存器的状态. K 框架中, 有一个命名为 k 的特殊 cell, 其包含一个将要被 K 框架执行运算 (computation) 的列表. 一个运算本质上是一个程序片段, 而原始程序被扁平化为一个或多个运算的线性序列. 规则描述的语义定义了配置之间的一步转换. 模块化的规则定义使得 K 框架定义的语义规则比其他框架下定义的形式化语义更简洁易读.

K 框架已被应用于很多项目中以实现语义形式化, 如 C[11,12], WebAssembly[13], LLVM[14], Python[15], Java[16,17], JavaScript[18], Boogie 等. C 语言的项目根据官方的 C11 标准定义了 K 框架下的语义规则, 还包括 C 语言的翻译, 链接和执行语义. 该 C 语言形式化项目已被用于构建运行时验证工具 RV-Match[19]. 其通过 C 语义运行测试套件来检测用户程序中未定义的行为. 还有 WebAssembly, KWasm 是它的 K 框架语义, WebAssembly 是一种底层 (但简单且流线型) 的汇编语言, 最初开发的目的是为基于浏览器的工具提供快速执行引擎. 最近, 它已在多个区块链智能合约平台中用作执行金融协议的基础语言. KWasm 已用于测量测试套件对 WebAssembly 代码的覆盖率并验证编译为 WebAssembly 的程序.

1.2 向量化机器学习指令

本文验证的对象是 ARMv8.1-M 架构的官方参考手册 (以下称 ARM 手册) 中关于 ARM Helium 技术的向量化机器学习 (machine learning, ML) 指令. 其中 ARM Helium 技术是 ARMv8.1-M 架构的新型 M-Profile Vector Extension (MVE) 向量扩展, 为架构带来计算能力的增强, 大幅提升机器学习以及信号处理等性能.

本文主要验证 ARM Helium 中关于 ML 的指令集扩展, 该扩展主要包含在机器学习中常用的向量大小比较和向量内积这两方面的向量化指令. 向量比较大小的指令可以用于如 ReLU 激活函数的实现, 神经网络分类结果判断等. 向量内积的指令则可以用于矩阵乘加, 卷积运算等. 向量化的指令使得同一时钟周期内可以执行更多的运算

操作. 本文涉及的向量化 ML 指令范围: 所要验证的 ML 指令包含 21 条 (区分整型和浮点数), 助记符 17 个, 且假设在应用模式下而非系统内核模式下进行语义形式化. 21 条指令中有 4 条调用浮点寄存器, 比较两个源寄存器值的大小, 并保存最大或最小值到目标寄存器; 4 条关于判断一个向量寄存器与通用寄存器的值的最大或最小值, 并保存在此通用寄存器中; 12 条是关于判断向量寄存器间的值的最大或最小值, 并将结果保存到目标向量寄存器中, 其中 4 条关于整型, 8 条关于浮点数; 还有 1 条指令是向量内积运算, 即机器学习中常用到的, 求两向量内对应元素的乘积的和的运算. 以上除了 4 条助记符相同的指令未涉及向量寄存器其他都是向量化指令, 所有统称为向量化 ML 指令 (以下简称 ML 指令). 这些 ML 指令用于计算神经网络分类结果, 如比较大小的操作. 由于我们重点关注的是 ML 指令的比较运算 (包括整数, 浮点数) 和向量内积运算 (整数) 执行的语义正确性, 所以文中重点描述算术运算的处理细节, 如 NaN (not a number) 处理. 由于涉及的指令与其他算术运算无关, 因此建模中不涉及进位等标志符 (向量内积运算不影响标志符). 本文验证的 ML 指令中, 要处理的数据类型分为整型和浮点数.

这 21 条 ML 指令的寻址方式都是寄存器寻址, 因此需要官方文档中寄存器寻址方式的定义. 因为 ML 指令大部分是向量化寄存器处理, 因此还需要与向量寄存器存取相关的定义. 如图 1 所示, 向量寄存器 Q 长度为 128 位, 由 2 个 64 位双精度浮点寄存器组成, 每个双精度浮点数寄存器由 2 个 32 位的单精度浮点数寄存器 S 组成. 官方文档中描述关于向量寄存器指令时用到节拍 (beat) 的概念. 一个指令执行分为 4 个节拍, 每个节拍按 32 位 (即一个 S 寄存器值) 执行一次运算. 因此每执行一次关于向量寄存器的 ML 指令都会运行 4 次操作.

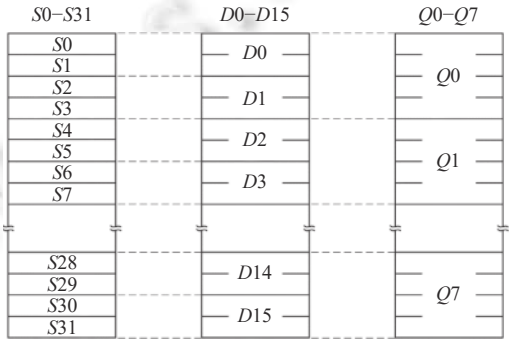

图 1　向量寄存器 *Q* 与浮点寄存器 *S*, *D*

在向量寄存器内, 每个元素的二进制位数大小必须相等. 对于整型值, 按照向量内每个元素的二进制位大小, 分为 8, 16 和 32 位. 因此向量寄存器内整型元素大小最长为 32 位, 最小为 8 位. 而对于浮点数, 元素大小分为 16 位, 32 位与 64 位. 当为 16 位时, 首位为符号位, 次 5 位为指数位, 最后 10 位为尾数位; 当为 32 位时, 首位为符号位, 次 8 位为指数位, 最后 23 位为尾数位. 64 位的首位是符号位, 次 11 位为指数位, 最后 52 位为尾数位. 浮点数的处理为 IEEE 754 标准. 关于指令语义的具体处理及其形式化在第 3 节中详细说明.

官网手册中 ML 指令描述结构由功能简述, 二进制编码, 语法, 译码伪代码, 汇编符号解析以及操作伪代码组成. 其中操作伪码即为诠释 ML 指令的执行语义, 因此着重于操作伪代码的分析. 如图 2 所示为指令 VMAXNM 的操作伪代码. 通常伪代码中应用的数据类型包含二进制位向量, 整型, 布尔值, 真实值 (数学意义上的值), 枚举, 元组, 结构和数组. 如 bits(*N*) 是长度为 *N* 的位向量, 整型就是以十进制数表示. 本文验证的 ML 指令大部分涉及向量寄存器的存取操作, 经常用到位操作. 在伪代码中, 取 *S* 寄存器的位向量中第 8 到 24 位, 则以“*S*[24:8]”的形式表示, 即当“:”前后是整数值时表示取位向量的相应连接位数. 另外当“:”前后为变量或者位向量时, 作为位向量连接符, 如将两个变量 *D*, *T* 的位向量连接, 表示为“*D*:*T*”. 这是在伪码提取时按项表达式区分的.

本文目标是验证应用模式下 ML 指令执行语义正确性, 在操作伪码提取时提取完整的伪代码作为形式化语义规则的基础, 以确保语义的完整. 如图 2 中 1, 2 行所示, 操作伪码先判断 ML 指令当前是否可以进入应用模式的状态, 根据判断结果决定是否会继续执行剩余的运算部分. 是否能够进入应用模式并正常执行指令是由一些内存映

射寄存器和特殊目的寄存器 (以下统称状态寄存器) 决定. 例如 CPACR (协同处理访问控制寄存器) 的 CP10 标志为“11”, NSACR (非安全访问控制寄存器) 的 CP10 标志为“1”, 且 CPPWR (协同处理功耗控制寄存器) 的 SU10 标志为“0”时, 代表拥有浮点数形式的访问权限.

如图 2 所示, 伪代码中关于函数的定义, 形式为字符串加上“()”或“[]”, 如图 2 第 2 行 “ExecuteFPCheck()”和第 7 行的“Q[n, curBeat]”. 大部分函数的详细语义在手册中有对应的伪代码描述, 而小部分函数则由自然语言描述其语义. 图 2 第 1 行的 EncodingSpecificOperations() 为当前指令的编码部分操作, 编码部分影响操作部分的信息, 包括图 2 中的 *n*, *m*, esize 和 absolute 都是从编码部分中获取的. 其中 *n* 代表源向量寄存器是第 *n* 个向量寄存器, *m* 代表源向量寄存器是第 *m* 个向量寄存器, esize 是向量寄存器中每个元素的位宽, absolute 用来判断指令是否进行绝对值操作. 第 2 行的 ExecuteFPCheck() 为检测指令当前是否处于浮点数操作状态. 第 4 行的 GetCurInstrBeat() 确定 ML 向量化指令当前处于哪个节拍, 以及获取元素掩码, 元素掩码决定最终结果是否写入到目标寄存器中 (应用模式下正常运行的元素掩码为“1111”). 第 6 行的 Zeros(32) 函数返回一个长度为 32 的由 0 组成的位向量, 用来初始化名为 result 的临时变量, 并最终写入目标寄存器. 第 7 行的 *Q*[n, curBeat] 根据当前节拍, 获取向量寄存器 *Q* 在当前节拍所对应的单精度浮点寄存器 *S* (1 个 *Q* 由 4 个 *S* 组成, 如 *Q*0 = *S*3:S2:S1:S0), 如 *Q*0 的第 0 个节拍为 *S*0, *Q*0 的第 3 个节拍对应 *S*3. 第 9 行的 for 循环根据每个 *S* 寄存器中元素的个数, 进行两个源寄存器的元素大小比较. 第 12 行的“Elem[op1, e, esize]” 函数, 获取 *S* 寄存器 (op1) 中第 *e* 个位宽为 esize 的元素. 接着用第 17 行的 FPMaxNum 函数比较浮点数大小, 最后将最大值返回到 result 中. 紧接着第 19 行的 for 循环, 将 result 中的位向量, 以字节为单位循环 4 次, 写入当前节拍向量寄存器 *Q* 所对应的 *S* 寄存器中.

Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[n, curBeat];
8  op2    = Q[m, curBeat];
9  for e = 0 to elements-1
10     // Avoid Floating-point exceptions on a predicated lane by checking the element mask
11     predicated = (elmtMask[e*(esize>>3)] == '0');
12     value1     = Elem[op1, e, esize];
13     value2     = Elem[op2, e, esize];
14     if absolute then
15         value1 = FPAbs(value1);
16         value2 = FPAbs(value2);
17     Elem[result, e, esize] = FPMaxNum(value1, value2, FALSE, predicated);
18
19 for e = 0 to 3
20     if elmtMask[e] == '1' then
21         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

图 2　VMAXNM 指令的操作伪代码

2　ML 指令语义形式化框架

本文运用 K 框架实现 ML 指令语义的形式化, ML 指令语义形式化以及测试的框架如图 3 所示.

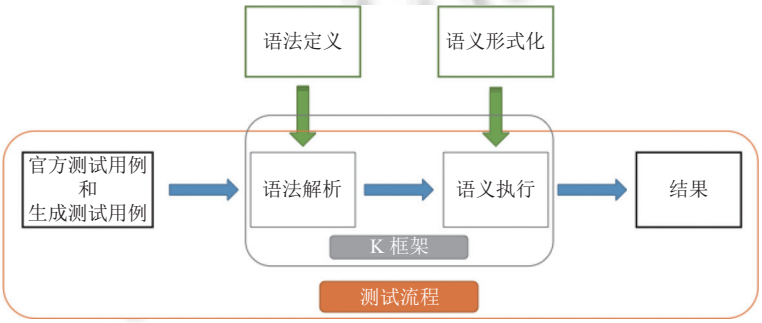

图 3　基于 K 框架的语义形式化与测试

第 1 部分 (图 3 上方两个方框), 定义 ML 指令的语法, 然后根据官方手册 ML 指令操作语义用 K 框架定义指令的执行规则. ML 指令语义执行基本流程如图 4 所示, 每一部分对应一组接口函数模块. 指令执行语义的定义是通过这些模块建立的, 将每个模块内接口函数功能用语义规则实现, 按照指令语义执行逻辑流程构建每条 ML 指令的语义. 图 4 中每一模块的形式化语义在第 3.2 节中详细说明.

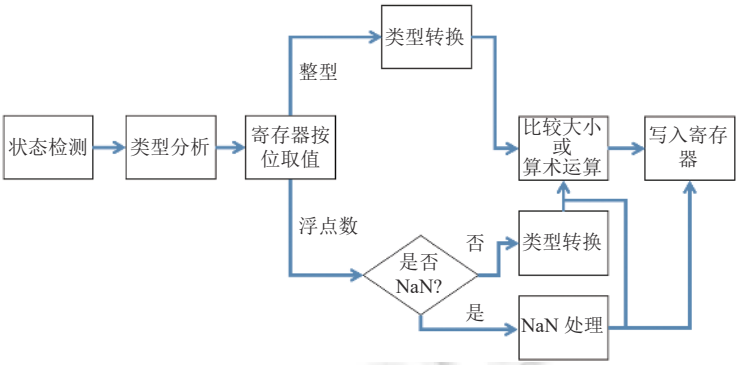

图 4 ML 指令语义执行流程

第 2 部分 (图 3 中间区域), 在 K Framework 下描述的 ML 指令语法语义可以自动生成解释器. 得到可执行语义还需要配置 (configuration) 描述系统的执行状态. 配置主要用于描述系统状态以及初始化. 其中系统状态包括寄存器与内存两部分, 寄存器为从寄存器名称到值的映射, 内存为从地址到值的映射. 值以 *W*'*V* 表示, 其中 *W* 为位宽, *V* 为数值. 初始化过程包括系统状态的初始化与程序初始化两部分. 系统状态初始化是将寄存器与内存映射表赋初值. 程序初始化内容包括指定程序与指定内存加载程序起始位置. 根据配置中程序初始化的内容, 语法解析模块逐行扫描指定程序并将符合语法的指令加载到指定的起始位置以及后续的内存地址中. 如果出现不符合语法的指令, 解析程序中断并返回当前系统状态, 可由加载程序的内存状态判断出哪条指令违反了语法. 当程序正确加载完成后将进入语义执行模块, 先将程序初始位置赋予程序计数器 PC 寄存器, 根据语义的转换规则逐条执行并更新配置中的系统状态 (寄存器与内存), 向量化寄存器的使用涉及节拍, 所以指令执行过程中会根据节拍更新状态. 每执行完一条指令后, 更新程序计数器 PC 指向下一个地址值.

第 3 步 (图 3 下方区域), 测试用例包括官方提供的测试用例与自动生成的数百条测试用例. 通过 K 框架自动生成的解释器运行并记录结果. 本部分内容在第 5 节详细介绍.

3 形式化语法与语义

本节首先介绍 ML 指令语法的定义, 接着介绍用 K 框架规则对 ML 指令执行语义的形式化.

3.1 ML 指令语法

图 5 给出了 ML 指令的语法的 BNF 形式定义. 本文验证的 ML 指令是关于元素比较和乘法累积的操作. 元素有不同的数据类型 (*DataType*), 如 *U*8 中的 *U* 代表着无符号数整数, 8 代表着位宽度为 8 位, *S*8 代表 8 位有符号数整数, *F*16 代表 16 位的浮点数. ML 指令的操作数 (Operand) 皆为寄存器 (Reg), 分为向量寄存器 (*Q*128), 双精度浮点寄存器 (*D*64), 单精度浮点寄存器 (*S*32) 和通用寄存器 (*R*32). 在向量寄存器中, 元素类型包含除 *F*64 外的所有数据类型, 而双精度浮点数 *F*64 只存在 *D*64 中. ARM-8.1 MVE 中向量寄存器有 8 个, 浮点寄存器 *D*64 有 16 个, 浮点寄存器 *S*32 有 32 个, 通用寄存器有 16 个. 其中向量寄存器 *Q* 由 4 个 *S*32 寄存器组成, 如 *Q*0 = *S*3:*S*2:*S*1:*S*0; 且 *D*64 寄存器由 2 个 *S*32 组成, 如 *D*0 = *S*1:*S*0. *SPReg* 指代特殊寄存器, 用于记录处理器当前的状态. 包括, FPSCR (浮点数状态与控制寄存器), CONTROL (控制寄存器, 控制处于何种状态模式) 等. 如 ML 指令中包含浮点数据类型, 在对浮点数进行处理的时候, 可能会出现 NaN 异常, 该状态下会修改 FPSCR 中的 IOC (invalid operation cumulative exception, 无效操作累积异常) 标志位, 即最低位, 并将其置 1.

```
DataType ::= U8 | U16 | U32 | S8 | ... | F16 | F32 | F64
Q128 ::= Q0 | Q1 | ... | Q7                R32 ::= R0 | R1 | ... | R15
S32 ::= S0 | S1 | ... | S31                D64 ::= D0 | D1 | ... | D15
SPReg ::= FPSCR | CONTROL | ...
Reg ::= Q128 | R32 | S32 | D64
Operand ::= Reg
Inst ::= VMAX.DataType  Q128, Q128, Q128      | VMAXA.DataType  Q128, Q128
    | VMAXNM.DataType  S32, S32, S32          | VMAXNM.DataType  D64, D64, D64
    | VMAXNM.DataType  Q128, Q128, Q128       | VMAXNMA.DataType  Q128, Q128
    | VMAXNMV.DataType  R32, Q128             | VMAXNMAV.DataType  R32, Q128
    | VMAXV.DataType  R32, Q128               | VMAXAV.DataType  R32, Q128
    | VMIN.DataType  Q128, Q128, Q128         | VMINA.DataType  Q128, Q128
    | VMINNM.DataType  S32, S32, S32          | VMINNM.DataType  D64, D64, D64
    | VMINNM.DataType  Q128, Q128, Q128       | VMINNMA.DataType  Q128, Q128
    | VMINNMV.DataType  R32, Q128             | VMINNMAV.DataType  R32, Q128
    | VMINV.DataType  R32, Q128               | VMINAV.DataType  R32, Q128
    | VMLAV.DataType  R32, Q128, Q128
Prog ::= Inst | Inst Prog
```

图 5　ML 指令语法

ML 指令的语法定义如图 5 中的 Inst 所示. ML 指令都是寄存器寻址方式, 且指令及有三地址码与二地址码两种格式. 当指令是三地址码时, 目标操作数为左边第 1 个操作数, 源操作数为余下两个操作数. 当指令为二地址码时, 目标操作数为左边第 1 个操作数, 两个均为源操作数. VMAX 指令是按照数据类型, 比较两个源向量寄存器中位向量的对应元素的值, 即 *Q*0 第 1 个元素与 *Q*1 第 1 个元素比较返回最大值, 直到所有元素对比较完毕, 将最终结果写入目标向量寄存器中, 如 VMAX.*U*8 *Q*2, *Q*0, *Q*1, 数据类型为 *U*8, 比较 *Q*0 和 *Q*1 中元素, 每 8 位为一个元素, 即分别比较 *Q*0 和 *Q*1 的第 1 到第 16 个元素, 将这 16 次比较中的较大值写入目标向量寄存器的相应位. VMAXA 指令是将 VMAX 指令的元素取绝对值后再进行比较. VMAXNM 指令是进行浮点数比较, 它的操作数分别包含着 *S*32, *D*64, *Q*128 这 3 种类型的寄存器. 对于同一指令 (如 VMAXNM), 不同操作数类型的操作有不一样的执行操作. 其中操作数类型标志为 *S*32 时, 数据类型为单精度浮点数 *F*32, 且 *S*32 中仅包含一个元素, 比较返回最大值; 当操作数类型标志为 *D*64 时, 数据类型为双精度浮点数 *F*64, *D*64 寄存器内仅包含一个元素; *Q*128 为操作数类型时, 数据类型可为 *F*16 和 *F*32, 向量寄存器 *Q*128 内包含多个元素, 比较也按对应元素位的值进行. VMAXNMA 指令是二地址格式, 比较源向量寄存器与目标向量寄存器的元素的绝对值, 返回最大值到目标向量寄存器中. VMAXNMV 指令是二地址格式, 将源向量寄存器的元素与目标通用寄存器的元素以浮点数型式比较, 将向量寄存器与通用寄存器中所有元素的最大值, 返回到目标通用寄存器中. VMAXNMAV 指令是 VMAXNMV 的绝对值形式. VMAXV 指令与 VMAXNMV 的差别, 在于比较的是整型数. VMAXAV 指令是 VMAXV 指令的绝对值形式. VMIN 指令与 VMAX 指令相似, 差异是返回的是最小值. 剩余的以 VMIN 为前缀的指令, 皆与对应的以 VMAX 为前缀指令相似, 差异是返回的是最小值. VMLAV 指令是将两个向量寄存器中各对应元素相乘, 并且将各乘积相加求和, 并将最终结果的低 32 位写入目标寄存器 (32 位通用寄存器) 中的运算. 关于程序的定义 *Prog*, 以单个指令或多个指令序列性构成.

ML 指令的数据类型总共 9 种, 不同指令能使用的数据类型种类有区别. 且对于不同指令, 操作数类型是有限制的, 比如 VMAXNM 指令对不同类型操作数实现功能各不相同.

3.2　ML 指令语义

3.2.1　状态检测与类型分析

本文对处理器状态的假设是其运行在应用模式下对 ML 指令的执行语义进行验证. 指令执行语义的第 1 部分是状态监测与类型分析. 因此, 需要将处理器中的特殊寄存器预设为满足应用模式下运行状态的值. 通过判断寄存器标志, 让 ML 指令正常进入应用模式并开始运行算数运算的操作部分. 状态寄存器的预设, 通过将状态寄存器的

特殊位赋予相应的值. 如控制寄存器 CONTROL, 它的第 2 位 (最低位开始) 影响当前运行状态是否允许浮点数扩展, 当为 0 时浮点数扩展不激活, 为 1 时激活. 当指令需要在浮点数状态下运行时, 可以通过判断 CONTROL 的第 2 位是否等于 1 来表示是否执行浮点数的相关操作. 而当其等于 0 时, 不可以执行浮点数的相关操作, 但可以执行整型的相关操作. 其他影响 ML 指令运行状态的状态寄存器以相同的方式检测.

类型分析是分析当前执行的 ML 指令涉及的数据类型. 本文所验证的 ML 指令因为大部分都涉及元素的大小比较, 且分为整数和浮点数. 数据类型分为 9 种类型, 即语法中的 *DataType*. 因此, 当从指令表中获取当前执行的 ML 指令语法后, 由于指令助记符中包含了 *DataType* 的标志, 所以可以通过判断输入的 *DataType* 是否等于语法中所定义的类型, 且 *DataType* 是否为当前指令所能接受的数据类型. 若都判断为真, 则进入此 *DataType* 标志对应的操作, 以此实现检测数据类型. 如当前执行的 ML 指令为: VMAX.*U*16 *Q*0, *Q*1, *Q*2 时, 因为 *U*16 为正确的数据类型, 且 VMAX 指令可以接受包含 *U*16 数据类型, 那么它会进入对应 *U*16 的操作语义中实现 VMAX 指令的操作部分.

3.2.2 寄存器取值

向量寄存器是由 4 个 32 位浮点寄存器 *S*32 构成 (如 *Q*0 = *S*3:*S*2:*S*1:*S*0), 并且根据节拍, 提取当前节拍中, 向量寄存器对应的 *S*32 寄存器. 图 6 所示的 *convSubVecRegsToRegs* 规则, 构成根据节拍获取向量寄存器对应元素的 *S*32 寄存器的语义, subR 代表构成向量寄存器的子寄存器. *convSubVecRegsToRegs* 规则根据向量寄存器以及当前的节拍值 (从 0 开始), 返回向量寄存器当前节拍的子寄存器 (即 *S*32 寄存器), 如 *convSubVecRegsToRegs*(*Q*0, 0) ⇒ *S*0, *convSubVecRegsToRegs*(*Q*1, 2) ⇒ *S*6. 其中, 以图 6 的 *convSubVecRegs-ToRegs* 规则为例, 解释语义规则图中的规则形式所代表的含义. 比如, 在横线右侧的字符串代表着当前规则的名称. 在横线上方的各项, 代表着执行此规则所需要满足的条件, 如 *R*:*Reg* 则代表需要满足 *R* 是 *Reg* 寄存器类型 (图 6), *Beat in*(0,1,2,3) 代表着要满足 *Beat* 是 (0,1,2,3) 这个集合的其中一项. 而横线下方则代表着当前语义规则的操作, 当满足横线上方的条件后, 调用 *convVecRegsToSubRegs* 规则就会根据给定的向量寄存器以及第几个 *Beat* 得到目标子寄存器 *subR*. 其他规则在文中表示形式与 *convSubVecRegsToRegs* 规则形式相似.

$$\frac{R:Reg \quad (Beat\ in\ (0,1,2,3))}{convVecRegsToSubRegs(R,\ Beat) \Rightarrow\ subR}\ convSubVecRegsToRegs$$

$$\frac{R:Reg \quad RSMap(R \mid \rightarrow W'V)}{getRegisterValue(R,\ \ RSMap) \Rightarrow W'V}\ getRegisterValue$$

$$\frac{W'V \quad start \geqslant 0 \quad start < W \quad end > 0 \quad end \leqslant W}{extractMInt(W'V, start, end) \Rightarrow W'V[(W - start - 1):(W - end)]}\ extractMInt$$

图 6 寄存器取值语义规则

图 6 中的第 2 和第 3 条语义规则为构成 ML 指令中寄存器取值和按位取值规则. ML 指令中常用到寄存器按位取值操作, 比如 ML 指令, 必须先从向量中取出元素再作比较或相乘, 因此需要实现寄存器取值和按位取值操作. 其中 *getRegisterValue* 规则表示从寄存器中取对应的位向量, 只要满足寄存器 *R* 是通用寄存器 *Reg* 类型, 且寄存器映射表 *RSMap* 中包含 *R* 的映射关系, 当在 *RSMap* 中, *R* 指向 (|→) 位向量 *W'V*, 那么规则最终会返回 *W'V*.

按位取值操作则由图 6 中的 *extractMInt* 规则实现, 当要在位向量 *W'V* 中取某一段位向量时, 可通过 *extractMInt*(*W'V*, *start*, *end*) 实现, 需要满足 *start* (开始位) 大于等于 0 并且小于 *W*, *end* (终止位) 大于 0 并且小于等于 *W*. 此处需要注意的是, *W'V* 的最低位是在右边第 1 位, 而 *extractMInt* 规则提取位向量时, 以位向量的左边第 1 位作为最低位, 此处的 *start* 和 *end* 都是以 *extractMInt* 规则的标准来对齐位向量中的位, 且 *end* 所对应的位不被提取, 所以转换为 *W'V*[(*W* – *start* – 1):(*W* – *end*)]. 例如, 对于位向量“1100”, 即 4′6, *extractMInt*(4′6, 0, 2) = 4′6[3:2], 即提取位向量“1100”的第 2 和第 3 位, “11”, 而 *start*(0) 对应位向量左边数起第 1 位: “1”, *end*(2) 对应位向量左边数起第 3 位: “0”.

3.2.3 类型转换

ML 指令大部分是关于获取一组数据中最大或最小值的操作, 与之伴随的是比较大小. 比较就涉及类型转换, 而且 K 框架中的比较规则涉及整型, 浮点数以及位向量, 比较时需将比较的元素转换为相同的对应类型后才能比

较. 按照原语义, 需将位向量转换为对应的整型或浮点数后, 运用比较规则进行比较. 接下来说明位向量转换为浮点数和整型类型的语义实现.

图 7 中, 列出了位向量转换为浮点数表示的实数的语义规则. *MInt2Float* 规则通过调用多条规则, 实现将位向量转换为浮点数的语义. 先介绍一下图 7 的一些基础符号语义, *W'V* 是所要转换的位向量, *M* 是尾数位宽度加 1, *E* 是指数位宽度, *Manti* 是尾数位向量代表的无符号整数值, *Exp* 是指数位向量代表的无符号整数值, *Sign* 是符号位对应的符号, *MaxExp*(*E*) 是代表 *E* 位的指数位所能表达的最大无符号整数值.

$$\frac{W'V}{MInt2Sign(W'V)\Rightarrow uvalueMInt(extractMInt(W'V,\ 0,\ 1))}MInt2Sign \qquad \frac{W'V\quad E:Int}{MInt2Exp(W'V,E)\Rightarrow uvalueMInt(extractMInt(W'V,\ 1,\ 1+E))}MInt2Exp$$

$$\frac{W'V\quad M:Int}{MInt2Manti(W'V,M)\Rightarrow uvalueMInt(extractMInt(W'V,\ 1+E,\ M+E))}MInt2Manti$$

$$\frac{E:Int}{MaxExponent(E)\Rightarrow 1\ll(E-1)}MaxExponent \qquad \frac{E:Int}{MinExponent(E)\Rightarrow 2-MaxExponent(E)}MinExponent$$

$$\frac{E:Int\quad Exp:Int}{ExpBitstoMul(E,Exp)\Rightarrow \mathrm{minInt(maxInt(Exp\ -MaxExponent(E)+1, MinExponent(E)),\ MaxExponent(E))}}ExpBitstoExp$$

$$\frac{M:Int\quad E:Int\quad Exp:Int}{ExpBitstoMul(M,E,Exp)\Rightarrow roundFloat(2.0,M,E)\hat{}\ \ Int2Float(ExpBitstoExp(E,Exp),\ M,\ E)}ExpBitstoMul$$

$$\frac{M:Int\quad E:Int\quad Manti:Int\quad (I=1\ ||\ I=2)}{MantissaBitsToFixedPoint(M,E,Manti,I)\Rightarrow \mathrm{Int2Float(\ (I\ll(M-1))\ |\ Manti,M,E)\ /\ (roundFloat(2.0,M,E)\hat{}Int2Float(M-1,M,E))}}MantissaBitsToFixedPoint$$

$$\frac{M:Int\quad E:Int\quad Sign\quad Exp:Int\quad Manti:Int}{Mint2FloatNormalized(M,E,\ Sign,\ Exp,\ Manti)\Rightarrow roundFloat(\ Sign*\ ExpBitstoMul(M,E,Exp)\ *\ MantissaBitsToFixedPoint(M,E,Manti,1),\ M,\ E\ )}MInt2FloatNormal$$

$$\frac{M:Int\quad E:Int\quad Sign\quad Exp:Int\quad Manti:Int}{Mint2FloatNormalized(M,E,\ Sign,\ Exp,\ Manti)\Rightarrow roundFloat(\ Sign*\ ExpBitstoMul(M,E,Exp)\ *\ MantissaBitsToFixedPoint(M,E,Manti,0),\ M,\ E\ )}MInt2FloatDenormal$$

$$\frac{M:Int\quad Sign\quad Exp>0\quad Exp<MaxExp(E)\quad Manti:Int}{translate2Float(M,E,Sign,Exp,Manti)\Rightarrow MInt2FloatNormal(M,\ E,\ Sign,\ Exp,\ Manti)}translate2Float\ \ Case1$$

$$\frac{M:Int\quad Sign\quad E:Int\quad Exp=0\quad Manti:Int}{translate2Float(M,E,Sign,\ 0,Manti)\Rightarrow MInt2FloatDenormal(M,\ E,\ Sign,\ 0,\ Manti)}translate2Float\ \ Case2$$

$$\frac{M:Int\quad Sign\quad Exp=MaxExp(E)\quad Manti=0}{translate2Float(M,E,Sign,Exp,0)\Rightarrow sign*roundFloat(GetInfinity(M,E),\ M,\ E)}translate2Float\ \ Case3$$

$$\frac{M:Int\quad Exp=MaxExp(E)\quad Manti>0}{translate2Float(M,E,\_,Exp,Manti)\Rightarrow roundFloat(GetNaN(M,E),\ M,\ E)}translate2Float\ \ Case4$$

$$\frac{M:Int\quad E:Int}{Getinfinity(M,E)\Rightarrow roundFloat(2.0\ \hat{}\ 1000000,M,E)}GetInfinity$$

$$\frac{M:Int\quad E:Int}{GetNaN(M,E)\Rightarrow roundFloat(0.0,M,E)}GetNaN$$

$$\frac{W'V\qquad W=M+E}{MInt2Float(W'V,\ M,\ E)\Rightarrow translate2Float(M,\ E,\ MInt2Sign(W'V),\ MInt2Exp(W'V,\ E),\ MInt2Manti(W'V,\ M))}MInt2Float$$

图 7　位向量转为浮点数语义规则

图 7 中, 第 1 行的 *MInt2Sign* 规则是将位向量中的最高位 (左边第 1 位) 取出作为符号位, 其中 *uvalueMInt* 规则 (K 框架标准模块内置) 是将位向量转换为无符号整数. 第 1 行的 *MInt2Exp* 规则是根据指数位宽 *E*, 通过按位取值操作在位向量中取出指数位向量, 并将指数位向量转换为整数类型. 第 2 行 *MInt2Manti* 规则是将位向量的尾数位转换为整数类型, 根据 *M*, 通过按位取值操作在位向量中取出尾数位向量, 并将其转换为整数类型. 第 3 行的 *MaxExponent* 规则, 返回 1 左移 (<<) *E* − 1位的整数值. 第 3 行的 *MinExponent* 返回 2 减去 *MaxExponent*(*E*) 的差的整数值.

第 4 行的 *ExpBitstoExp* 规则, 实现在指数值 *Exp* 和指数位宽 *E* 所代表的当前浮点数精度下, 按所需偏移值将指数值偏移化. 当指数值大于 0 时, 偏移值为– (*MaxExponent*(*E*) – 1) , 当指数值等于 0 时, 偏移值为 *MinExponent*(*E*). 如半精度浮点数 (16 位), 其指数位宽为 5, 当指数位值 *Exp* 大于 0 时, 偏移值为−15, 当 *Exp* 等于 0 时, 其偏移值为−14. 单精度浮点数 (32 位), 其指数位宽为 8, 当 *Exp* 大于 0 时, 偏移值为−127, 当 *Exp*=0 时, 偏移值为−126. 双精度浮点数 (64 位), 指数位宽为 11 位, 当 *Exp*>0 时, 偏移值为−1 023, 当 *Exp*=0 时, 偏移值为−1 022.

第 5 行的 *ExpBitstoMul* 规则, 是将经过第 4 行 *ExpBitstoExp* 规则偏移化后的指数值, 作为 2.0 的指数位, 返回真实浮点数的指数值. 其中的 *roundFloat* 规则, 是 K 框架中关于浮点数模块的内置规则, 功能是将浮点值按指数位精度为 E, 尾数位精度为 M 进行舍入操作规则. 其规则定义满足 IEEE754 标准, 可以直接使用.

第 6 行的 *MantissaBitsToFixedPoint* 规则是对尾数位向量进行规格化或非规格化处理. 规格化语义是先将 *Manti* 乘 $2.0^{(-M+1)}$ 再加上 1. 0. 当用规则实现时, 我们将小数点前一位, 用 *I* 代替, 再对 *I* 左移尾数位 (*M* – 1), 接着和 *Manti* 进行或操作 (|), 最后再将结果除以 $2.0^{(-M+1)}$, 得到规格化后的尾数值. 这样当小数点前一位是 1 时, 即为规格化操作, 其中 *Int*2*Float* (K 框架内置) 实现整数转换为浮点数功能. 而非规格化 (小数点前一位为 0) 操作, 即将 *I* 设为 0 即可.

第 7 行的 *Mint*2*FloatNormal* 规则, 是实现将位向量进行规格化处理. 当其指数值 *Exp* 大于 0 且小于 *MaxExp*(*E*) 时, 需要对位向量进行规格化处理. 指数位的处理运用第 5 行的 *ExpBitstoMul* 规则进行指数偏移化. 尾数位的规格化处理运用第 6 行的 *MantissaBitsToFixedPoint* 规则, 将 *I* 设为 1.

第 8 行的 *Mint*2*FloatDenormal* 规则, 是实现将位向量进行非规格化处理. 当其指数值 *Exp* 等于 0 时, 需要对位向量进行非规格化处理. 指数位的处理运用第 5 行的 *ExpBitstoMul* 规则进行指数值偏移化. 尾数位的非规格化处理运用第 6 行的 *MantissaBitsToFixedPoint* 规则, 将 *I* 设为 0.

第 9 到第 12 行的 *translate*2*Float* 规则, 是按情况将位向量转换为浮点数, 分为 4 种情况. Case1 的 *translate*2*Float* 规则是将位向量以规格化处理转换为浮点数. 当指数值 *Exp* 大于 0 并且小于 *MaxExp*(*E*) 时, 运用第 7 行的 *Mint*2*FloatNormal* 规则实现规格化转换. Case2 的 *translateFloat* 规则是将位向量以非规格化处理转换为浮点数. 当指数值 *Exp* 等于 0 时, 运用第 8 行的 *Mint*2*FloatDenormal* 规则实现非规格化转换. Case3 的 *translate*2*Float* 规则是将位向量转换为代表无穷的浮点数. 当指数值 *Exp* 等于 *MaxExp*(*E*) 且尾数值 *Manti* 等于 0 时, 位向量代表无穷, 因此运用第 13 行 *GetInfinity* 规则, 将位向量转换为浮点值 2.0 的 1 000 000 次方. Case4 的 *translate*2*Float* 规则是当位向量为 NaN 时, 按照手册语义返回浮点值 0.0 值, 用第 14 行的 *GetNaN* 规则实现. 但是手册的语义中, NaN 处理在比较操作前完成, 因此当出现 NaN 时, 不会运用到这一返回值 0.0. 具体的 NaN 处理在第 3.2.4 节详细说明.

最后一行的位向量到浮点数转换规则 *Mint*2*Float*, 是以上规则的汇总. 输入位向量 *W'V*, 当其指数位宽 (*E*) 和尾数位宽加 1 (*M*) 的和等于位向量的宽度 *W* 时, 即可调用 *translate*2*Float* 规则实现位向量到浮点值的类型转换.

位向量转换为整型数的类型转换规则, 可以用 K 框架的标准模块 MINT 中内置的位向量转换为整数的规则实现. 比如, 将位向量转换为有符号数整数, 就可以调用 *svalueMInt*(*W'V*), 此规则将 *W'V* 转换为有符号整数形式; 当需要将位向量转换为无符号数整数时, 可以调用 *uvalueMInt*(*W'V*), 此规则将 *W'V* 转换为无符号整数形式.

3.2.4 NaN 处理

从第 3.2.3 节可知, 当指数位向量所代表的整数值与指数位为 E 所能表示的最大整数值相等, 并且尾数位向量所代表的整数值大于 0 时, 位向量所代表的浮点数就是 NaN. 因此, 需要对位向量进行 NaN 处理. 官方手册中, ML 指令处理 NaN 的语义将 NaN 分为 QNaN 和 SNaN 两种类型. QNaN 与 SNaN 的不同之处在于, QNaN 的尾数部分最高位 (即 *M* – 2 位, 因为 *M* 值为尾数位宽加 1) 定义为 1, SNaN 最高位定义为 0. QNaN 一般表示未定义的算术运算结果, 最常见的是除以 0 运算; 而 SNaN 一般被用于标记未初始化的值, 以此来捕获异常.

对于 NaN 处理, 我们按照指令原语义实现. NaN 处理有 3 种情况, 如图 8 所示. 第 1 种, *NaNProcess Case*1, 当比较的两个元素的位向量皆为 QNaN 类型时, 即当它们指数值都是 E 位指数位所能表示的最大值, 且尾数位最高位等于 1 时 (规则中以 *W'V*[*n*], 表示位向量的第 *n* 位), 规则中的 *SetDefault* 会将默认位向量 (符号位为 0, 指数位全

为 1, 尾数位最高位是 1 且余下的都是 0) 返回到结果. 第 2 种, *NaNProcess Case*2, 当比较的两元素中皆为 NaN 且存在 SNaN 时, 也会返回默认位向量, 并且会将 FPSCR 的最低位置 1 (*SetFPSCR*). 第 3 种, *NaNProcess Case*3, 当比较中的某一个元素是 NaN 且另一个为非 NaN (指数位不全为 1) 时, 若元素 NaN 为 SNaN 则先将其重置为 QNaN, 下一步统一将为 QNaN 的元素位向量重置为无穷小或无穷大, 按照比较最大值或最小值的语义, 返回的结果由另一个非 NaN 元素决定. 当出现 NaN 是 Case2 和 Case3 的情况的时候, 当出现 SNaN 的时候, 对于 SNaN 的处理, 出现了名为 FPExc_InvalidOp 的异常, 即浮点值执行的无效操作, 因此会改变 FPSCR (浮点数状态与控制寄存器) 的最低位, 将其置 1, 而其他的两种情况不会改变 FPSCR 的标志位. 此处 FPSCR 对应的位向量, 根据官方手册语义, 每当本文验证的 ML 指令执行浮点值操作时, 即每个节拍, 都会将其位向量预设为“0000 0011 0000 0000 0000 0000 0000 0000”.

$$\frac{W_1'V_1 \quad W_2'V_2 \quad Exp_1 = Exp_2 = MaxExp(E) \quad Manti > 0 \quad W_1'V_1[M-2] = 1 \quad W_2'V_2[M-2] = 1}{NaNProcess(W_1'V_1, W_2'V_2, Exp_1, Exp_2) \Rightarrow SetDefault}\ NaNProcess\ \ Case1$$

$$\frac{W_1'V_1 \quad W_2'V_2 \quad Exp_1 = Exp_2 = MaxExp(E) \quad Manti > 0 \quad (W_1'V_1[M-2] = 0 \ \|\ W_2'V_2[M-2] = 0)}{NaNProcess(W_1'V_1, W_2'V_2, Exp_1, Exp_2) \Rightarrow (SetDefault\ \&\&\ SetFPSCR)}\ NaNProcess\ \ Case2$$

$$\frac{W_1'V_1 \quad W_2'V_2 \quad Exp_1 < MaxExp(E_1) \quad Exp_2 = MapExp(E_2) \quad W_2'V_2[M-2] = 1}{NaNProcess(W_1'V_1, W_2'V_2, Exp_1, Exp_2) \Rightarrow W_1'V_1}\ NaNProcess\ \ Case3$$

图 8 NaN 处理语义规则

该 NaN 检测位于比较操作之前, 即未经类型转换, 数据类型仍为位向量. 因此保证了不会有 NaN 参与类型转换和比较操作, 与官方手册的语义一致.

3.2.5 比 较

经过 NaN 处理与类型转换后, 指令执行可以进入元素间的大小比较阶段. 在 K 框架的标准模块中, 包含着各种数据类型的比较规则, 如整型, 浮点值和位向量. 本文应用了位向量和浮点值的比较规则, 以实现整型 ML 指令和浮点数 ML 指令的比较操作. K 框架中的 MInt 模块中, *sgeMInt* 规则表示带符号位的大于等于, *ugeMInt* 规则表示无符号位的大于等于, 以位向量为类型作比较. 对于浮点数数据类型, 在 K 框架的 Float 模块中也定义了以“≤Float, <Float, ==Float, ≥Float, >Float”符号进行浮点数间的比较, 可以直接应用在项目中, 不再累述.

3.2.6 算术运算

本节介绍为指令语义形式化而定义的算术运算规则. ML 指令中的 VMLAV 指令实现向量内积的功能, 是将两个源操作数, 即两个向量寄存器中的对应元素, 做乘法运算, 再将得到的乘积做累加求和, 最终取结果的低 32 位写入目标寄存器 (32 位通用寄存器) 中. VMLAV 指令的数据类型是整型, 区分无符号和有符号整型. 并且该指令不会改变任何标识符, 如进位等, 因为过程中的乘法与加法运算是以整数的形式 (先将位向量转换为对应的整型) 进行, 再将运算得到的整型结果转换为位向量形式的最终结果, 并且只保留结果的低 32 位, 因此不涉及进位和溢出等标志符, 亦不会影响其他任何标志符.

如图 9 所示是算术运算中相关操作的语义规则. 第 1 行的 *+Int* 规则是关于两个整数的加法运算, 需要满足整数 I_3 等于 I_1 与 I_2 的和的条件. 第 2 行的 *MulofUInt* 规则是关于两个位向量的无符号整型的乘法运算, 调用 *uvalueMInt* 规则取出位向量所代表的无符号整数值, 并调用 K 框架内置的乘法规则 (*), 最终将乘积转换为位向量, 其位宽为 *W*+*W*. 由于元素之间的位宽都是相等的, 以 *W* 表示元素位宽. 第 3 行的 *MulofSInt* 规则是关于两个位向量的有符号整型的乘法运算, 调用 *svalueMInt* 规则取出位向量所代表的有符号整数值, 并调用 K 框架内置的乘法规则 (*), 最终将乘积转换为位向量, 其位宽为 *W*+*W*.

图 9 第 4 行的 *SInnerProd* 规则是关于有符号运算的向量内积规则. 其中在条件中 (横线上方), $W'V_1$ 和 $W'V_2$ 代表一个 *Beat* 中, 两个向量寄存器在此 *Beat* 下分别对应的浮点值寄存器所存储的位向量, 因此 $W = 32$. *esize* 是指向量内单个元素的位宽, 包含 8 位, 16 位和 32 位. *n* 代表单个浮点值寄存器内 (即单个 *Beat* 内) 包含的元素个数. 在规则中 (横线下方), $Ext1_i$ 代表 $W'V_1$ 的第 *i* 个 (*i* 属于 1 到 *n*) 元素的位向量, 调用图 6 的 *extractMInt* 规则实现, $Ext2_i$ 是 $W'V_2$ 的第 *i* 个元素的位向量. 调用第 3 行的 *MulofSInt* 规则进行有符号乘法, 将两向量在某一 *Beat* 下的的第 *i* 个元素相乘, 获取元素对的乘积位向量, 并调用 *svalueMInt* 规则提取乘积位向量对应的整数值. 其中以

$SumofIP_n$ 代表 n 对元素的乘积的加法和, 它调用第 1 行的+*Int* 规则实现加法运算. 并且最终将整数和存放到 W 位 (32 位) 的位向量中. 注意, 当乘积和的带符号整数结果可以用 32 位位向量表示时, 则将整数结果转换为其对应的 32 位位向量. 若整数结果超出了 32 位位向量能表示的范围, 则将其整数对应位向量的低 32 位取出作为最终结果, 调用图 6 的 *extractMInt* 规则提取结果的低 32 位, 即按照手册中的指令语义, 不管溢出部分, 直接取低 32 位.

$$\frac{I_1:Int \quad I_2:Int \quad I_3:Int \quad I_3 = I_1 + I_2}{I_1 \ +Int \ I_2 \Rightarrow I_3} +Int$$

$$\frac{W'V_1 \quad W'V_2}{MulofUInt(W'V_1, W'V_2) \Rightarrow (W+W)'(uvalueMInt(W'V_1) * uvalueMInt(W'V_2))} MulofUInt$$

$$\frac{W'V_1 \quad W'V_2}{MulofSInt(W'V_1, W'V_2) \Rightarrow (W+W)'(svalueMInt(W'V_1) * svalueMInt(W'V_2))} MulofSInt$$

$$\frac{W'V_1 \quad W'V_2 \quad W=32 \quad (esize \ in \ (8,16,32)) \quad i \in [1,n] \quad n = 32/esize}{SInnerProd(W'V_1, W'V_2) \Rightarrow W'(SumofIP_n\left(svalueMInt\left(MulofSInt(Ext1_i, Ext2_i)\right)\right))} SInnerProd$$

$$\frac{W'V_1 \quad W'V_2 \quad W=32 \quad (esize \ in \ (8,16,32)) \quad i \in [1,n] \quad n = 32/esize}{UInnerProd(W'V_1, W'V_2) \Rightarrow W'(SumofIP_n\left(uvalueMInt\left(MulofUInt(Ext1_i, Ext2_i)\right)\right))} UInnerProd$$

图 9 算术运算的语义规则

图 9 第 5 行的 *UInnerProd* 规则是关于无符号运算的向量内积规则. 其中条件与第 4 行 *SInnerProd* 规则相似, 此处不再累述. 在规则中 (横线下方), 与 *SInnerProd* 不相同的是, 在向量元素对做乘法运算时, 调用了第 2 行的 *MulofUInt* 规则进行无符号乘法, 以及将乘积位向量转换为整数时, 调用 *uvalueMInt* 规则, 转换为无符号整数. 其他部分与 *SInnerProd* 规则的描述相同. 注意, 当乘积和的无符号整数结果能用 32 位位向量表示时, 则将整数结果转换为其对应的 32 位位向量. 若整数结果超出 32 位位向量能表示的范围时, 则将其整数对应位向量的低 32 位取出作为最终结果, 调用图 6 的 *extractMInt* 规则提取结果的低 32 位.

3.2.7 寄存器按位写入

操作数为向量寄存器的 ML 指令一般包含 4 个节拍, 每个节拍的操作相同. 以图 2 为例, 操作部分根据数据类型改变用于比较的 for 循环次数 (如图 2 中第 1 个 for 循环), 而第 2 个 for 循环则循环 4 次, 以字节为单位将结果写入向量寄存器在当前节拍对应的浮点寄存器 *S*32 中. 如图 2 中的操作代表一个节拍内的操作, 中间会将比较的结果写入临时变量 *result* 中, *result* 的数据类型为位向量. 当完成第 1 个 for 循环后, *result* 中存有各元素比较后的结果, 最后通过第 2 个 for 循环以字节为单位将 *result* 中的结果写入目标寄存器. 因此, 第 1 个 for 循环内每次循环的最后一步是将比较结果的位向量按位替换到目标位向量中, 引入按位替换规则 *plugInMask* 实现按位写入语义, 如图 10 所示, 分为 4 种情况. 其中 *Wdes'Vdes* 为目标位向量, *Wsrc'Vsrc* 为写入的位向量 (源位向量), *Wsrc* 就是写入位向量的宽度, Start 为插入的开始位 (最低位为右边第 1 位). 连接规则 *concatenateMInt* 是 K 框架的标准模块 *MInt* 中的内置规则, 功能是将两个位向量连接如 *concatenateMInt*(‘1111’, ‘0000’) 转换为“11110000”. Case1 的 *plugInMask* 规则, 当插入的开始位是 0, 且目标位向量和源位向量的宽度相等时, 可以直接将源位向量赋给目标位向量. Case2 的 *plugInMask* 规则, 当插入的开始位是 0, 且目标位向量的宽度比源位向量的宽度大时, 通过位向量连接规则, 将目标位向量中没有被写入的位向量取出来后, 与源位向量相连, 可以得到写入的结果. Case3 的 *plugInMask* 规则, 当插入的开始位不等于 (!=)0, 且目标位向量的宽度, 等于源位向量的宽度与插入的开始位值的和时, 将源位向量写入目标位向量的左边的 *Wsrc* 位, 即运用连接规则, 将源位向量作为最后的 *Wsrc* 位, 与目标位向量中没有被写入的位向量相连. Case4 的 *plugInMask* 规则, 当插入的开始位不等于 0, 且目标位向量的宽度大于源位向量的宽度与插入的开始位值的和时, 在目标位向量的中间写入源位向量, 需要运用两次连接规则, 最外层的连接规则将目标位向量的左边没有被写入的位向量, 和右边的位向量相连, 而第 2 个连接规则就是将右边的位向

量连接, 即连接中间的源位向量和目标位向量右边没有被写入的位向量, 如 *plugInMask*(‘10000’, ‘11’, 1) 转换为“10110”, 将“11”写入到“10000”中的第 1 和第 2 位.

$$\frac{Wdes'Vdes \quad Wsrc'Vsrc \quad Start{:}\,Int \quad Start = 0 \quad Wdes = Wsrc}{plugInMask(Wdes'Vdes, Wsrc'Vsrc, Start) \Rightarrow Wsrc'Vsrc}\, plugInMask\ \ Case1$$

$$\frac{Wdes'Vdes \quad Wsrc'Vsrc \quad Start{:}\,Int \quad Start = 0 \quad Wdes > Wsrc}{plugInMask(Wdes'Vdes, Wsrc'Vsrc, Start) \Rightarrow \\ concatenateMInt(extractMInt(Wdes'Vdes, 0, Wdes - Wsrc),\ Wsrc'Vsrc)}\, plugInMask\ \ Case2$$

$$\frac{Wdes'Vdes \quad Wsrc'Vsrc \quad Start{:}\,Int \quad Start\ !=0 \quad Wdes = Wsrc + Start}{plugInMask(Wdes'Vdes, Wsrc'Vsrc, Start) \Rightarrow \\ concatenateMInt(Wsrc'Vsrc,\ extractMInt(Wdes'Vdes, Wsrc, Wdes))}\, plugInMask\ \ Case3$$

$$\frac{Wdes'Vdes \quad Wsrc'Vsrc \quad Start{:}\,Int \quad Start\ !=0 \quad Wdes > Wsrc + Start}{plugInMask(Wdes'Vdes, Wsrc'Vsrc, Start) \Rightarrow \\ concatenateMInt(\ extractMInt(Wdes'Vdes, 0, Wdes - Wsrc - Start), \\ concatenateMInt(Wsrc'Vsrc,\ extractMInt(Wdes'Vdes,\ Wdes - Start,\ Wdes))\ )}\, plugInMask\ \ Case4$$

$$\frac{Inst \quad (Beat\ in\ (0,1,2,3))}{execinstrBeat(Inst,\ Beat) \Rightarrow\ Operation}\, execinstrBeat$$

图 10　按位写入与单节拍指令执行的语义规则

由此, 可以运用 *plugInMask* 规则对 for 循环进行语义转换, 将 *plugInMask* 规则以内嵌的形式实现元素按宽度写入 *result* 和将 *result* 按字节写入目标寄存器的语义, 如 VMAX 指令, 类型 *U*16, 每个元素 16 位, 一个 *S*32 寄存器中包含 2 个元素, 因此进行两次 for 循环获取最大值, 我们运用 *plugInMask* 规则, 以 *plugInMask* plugInMask $((32'V, 16'Vlowmax, 0), 16'Vtopmax, 16)$ 的内嵌形式实现 for 循环的等价转换. 这里内部的 *plugInMask* 规则作为外部 *plugInMask* 规则的第 1 个参数, 先将低 16 位的最大值位向量 ($16'Vlowmax$) 写入目标寄存器对应位向量 ($32'V$) 的低 16 位中, 接着运用外部的 *plugInMask* 规则, 将高 16 位的最大值位向量 ($16'Vtopmax$) 写入目标寄存器对应位向量 ($32'V$) 的高 16 位中. 另外, 对于 ML 指令的操作部分, 即使分为 4 个节拍, 但操作是相同的, 改变的只是元素, 所以用图 10 的 *execinstrBeat* 规则实现单个节拍的操作语义. 根据节拍值 *Beat*, 获取相应的寄存器与寄存器的位向量以进行比较. *Operation* 是单个指令所要实现的语义, 通过上文列出的规则实现.

4　单指令实现

本节以三地址指令 VMAXNM.*F*32 *Q*2, *Q*1, *Q*0 为例子, 说明 ML 指令如何通过上文中的形式化语法语义规则实现指令的解析与执行.

指令 VMAXNM.*F*32 *Q*2, *Q*1, *Q*0 作为输入, (1) 进行状态初始化, 将各寄存器初始化为符合应用模式的位向量. (2) 进行语法分析, 将目标指令进行语法解析, 当语法满足 *Inst* 的语法时, 将指令写入指令表, 当程序中所有指令都写入后, 进入语义执行, 即调用 *execinstrBeat* 规则. (3) 可执行的语义来源于我们对指令操作规则的形式定义, 每个 ML 指令的语义, 均可以使用上文中模块化的规则构成, 以 *execinstrBeat* 规则为入口实现 ML 指令规则的归纳. 如图 4 所示, 首先从指令表顺序提取 ML 指令 *Inst*, 作为 *execinstrBeat* 的规约, 随后 根据 Inst 调用对应的规则. 例子 VMAXNM.F32 指令, 首先会通过判断相关状态寄存器的位向量的相关标志位, 以确定是否满足应用模式且是否满足浮点数模式, 若满足, 则进入操作部分, 若不满足, 则返回未定义位向量 *undefMInt* (对于非应用模式的状态, 退出运行). 当符合 ML 指令执行状态后, 首先进行数据类型检测, 如第 3.2.1 节所描述的, *F*32 是 VMAXNM 的数据类型, 由于手册 VMAXNM 语义中, 该指令包含 *F*32 数据类型, 因此可以进行关于 *F*32 的操作. 以节拍数 0 为例, 当数据类型为 *F*32 时, 先提取 *Q*1 和 *Q*0 在节拍 0 时对应的寄存器, 调用 *convSubVecRegsToRegs* 规则, 得到 *Q*1 和 *Q*0 在 0 节拍的浮点寄存器分别为 *S*4 和 *S*0. 然后调用寄存器取值规则 *getRegisterValue* 在寄存器内存映射表中获取 *S*4, *S*0 对应的值. *F*32 表示单精度浮点数, *S*4, *S*0 中 32 位的位向量直接取出作为两个即将进行浮点数类型比较的元素. 接着要检测两个位向量是否为 NaN, 运用按位取值 *extractMInt* 规则以及第 3.2.4 节中描述的 NaN 检测规则. 若出现 NaN, 则按规则描述的 3 种情况进行处理; 若没有出现 NaN, 则进入数据类型转换. 将位向量转换为浮点数, 调用 *MInt*2*Float* 规则, 转换为浮点值后, 调用 K 框架内置的浮点值比较规则, 如“≤Float”等进行取最大值的

操作. 再将较大值的位向量返回 *result* (初始值为全 0 的 32 位位向量) 中, 用 *plugInMask* 规则将比较得到的最大值结果写入 *result* 中. 最后, 将 *result* 中的结果以字节为单位, 从低位到高位 (最高有效位格式) 写入目标寄存器 *S*8 中 (*Q*2 在节拍 0, 对应的寄存器是 *S*8), 依然是调用 *plugInMask* 规则, 如第 3.2.6 节中描述的内嵌形式实现以字节为单位写入 *S*8.

5 实 验

K 框架实现 ML 指令语义的形式化, 可以自动生成可执行的 correct-by-construction 的指令解释器. 本文利用该解释器, 对官方的 ML 指令测试集以及我们自动生成的多个测试用例进行实验, 检测 ML 指令语义的正确性. 此处的官方测试集是指 ARM Cortex-M55 官方手册[20]中提供的测试用例. 因 ARM Crotex-M55 运用 ARMv8.1-M 架构, 所以手册中包含 ARMv8.1-M 架构的 ML 指令. 并且每个 ML 指令都包含一个测试用例, 其给出了该指令操作数以及执行后的系统返回状态. 由于手册中的官方测试用例数量不多, 没有涵盖操作数类型为双精度浮点寄存器的指令, 也没有涵盖 NaN 检测的用例. 因此, 我们用 C 语言脚本对每条 ML 指令生成多个测试用例. 对每条 ML 指令, 输出 30 个测试用例以供各 ML 指令测试. 此处的 30 条测试用例, 在测试时是在一个脚本下连续运行的 30 条指令, 每条指令开始时会重置系统状态. 最终输出时, 不仅会输出指令的最终状态, 还会输出关于指令运行过程的中间状态的列表. 根据中间状态与最终状态的列表判断每条指令的正确性. 生成的测试用例是遵循指令语义生成的, 覆盖原伪码定义的指令语义中涉及的所有语义分支, 其中包括标志位判断, 位长判断, 以及在浮点数下的 NaN 处理分类判断. 生成的测试用例中, 当 ML 指令操作数都是向量寄存器时, 通过生成 128 位的位向量作为测试数据. 当 ML 指令操作数是通用寄存器 (R) 或 32 位单精度浮点寄存器 (S) 时, 则生成 32 位的位向量作为测试数据. 当 ML 指令操作数是双精度浮点寄存器 (D) 时, 则生成 64 位的位向量作为测试数据. 在数据类型上包括了该指令能够接受的不同位长的位向量, 以及针对浮点数指令生成不同模式的 NaN 的情况. 由于 Cortex-M55 的模拟器并未开源, 与其相关的芯片也并未正式发售, 因此实验中脚本生成的测试用例尚未能与官方模拟器进行对比实验. 关于测试用例运算正确性判断. 我们根据官方手册 ML 指令伪码用 C 语言实现了指令功能模拟器. 手册中伪码与指令过程式语言相近, 可以直观的转换为 C 语言实现的模拟程序, 所以将其作为测试用例返回结果的参考对象. 当在相同输入下, C 模拟器运行得到的结果与 K 的指令解释器运行得到的结果一致, 作为判断测试用例在 K 框架解释器下的运算正确的依据.

表 1 和表 2 分别为各 ML 指令关于官方测试用例和自动生成测试用例所得到实验结果. 如表 1 所示, 第 1 列第 1 行代表当前图表是 Benchmarks, 即属于官方的测试用例. 第 1 列中的第 2, 3 行代表 ML 指令的数据类型是整型和浮点数. 第 2 列代表各 ML 指令. 第 3 列代表每个 ML 指令的单个测试程序中, 包含不同测试数据和数据类型的指令个数. 第 4 列代表浮点数 ML 指令的测试中包含 NaN 的例子个数. 第 5 列代表单个测试程序运行的平均时间, 用秒 (s) 为单位. 第 6 列代表测试程序的所有测试数据, 是否都能通过执行 ML 指令形式化语义得到正确结果, 若都成功则“√”, 出现错误则“×”. 最后一列代表浮点数测试中, 是否能成功检测到 NaN 异常. 表中的“—”符号代表空. 表 2 为生成的测试用例, 其他格式与表 1 相同.

如表 1 所示, 在官方提供的测试用例中, VMAXNM 指令与 VMINNM 指令关于操作数类型为双精度浮点寄存器 (VMAXNM-D 和 VMINNM-D), 在官方手册中没有提供相应的测试用例, 因此为空. VMAXNM-S 和 VMAXNM-Q 分别代表操作数类型为单精度浮点寄存器和向量寄存器时的 VMAXNM 指令, VMINNM-S, VMINNM-D 和 VMINNM-Q 与 VMAXNM 所代表的含义一致. 可以得到, 各 ML 指令的形式化语义可以正确通过官方测试用例.

由于官方的测试用例没有涵盖操作数类型为双精度浮点寄存器的指令, 以及没有涵盖 NaN 检测的用例, 本文从而生成了满足上述两种情况的测试用例, 并覆盖到所有上文定义的语法语义. 从表 2 可以得到, 对于生成的测试用例, ML 指令的可执行语义皆能得到正确结果. 且对于浮点数类型的 ML 指令, 当生成的数据中含有 NaN 的情况时能成功检测到异常. 因此, 基于官方手册与 K Framework 得到的 ML 指令可执行语义是被验证符合预期的.

表 1 官方测试用例实验结果

Benchmarks	指令	测试数量	含NaN用例	正确	NaN异常
整型	VMAX	1	—	√	—
	VMAXA	1	—	√	—
	VMAXV	1	—	√	—
	VMAXAV	1	—	√	—
	VMIN	1	—	√	—
	VMINA	1	—	√	—
	VMINV	1	—	√	—
	VMINAV	1	—	√	—
	VMLAV	1	—	√	—
浮点数	VMAXNM-S	1	0	√	0
	VMAXNM-D	—	—	—	—
	VMAXNM-Q	1	0	√	0
	VMAXNMA	1	0	√	0
	VMAXNMV	1	0	√	0
	VMAXNMAV	1	0	√	0
	VMINNM-S	1	0	√	0
	VMINNM-D	—	—	—	—
	VMINNM-Q	1	0	√	0
	VMINNMA	1	0	√	0
	VMINNMV	1	0	√	0
	VMINNMAV	1	0	√	0

表 2 生成测试用例实验结果

生成测试用例	指令	测试数量	含NaN用例	正确	NaN异常
整型	VMAX	30	—	√	—
	VMAXA	30	—	√	—
	VMAXV	30	—	√	—
	VMAXAV	30	—	√	—
	VMIN	30	—	√	—
	VMINA	30	—	√	—
	VMINV	30	—	√	—
	VMINAV	30	—	√	—
	VMLAV	30	—	√	—
浮点数	VMAXNM-S	30	0	√	0
	VMAXNM-D	30	0	√	0
	VMAXNM-Q	30	2	√	2
	VMAXNMA	30	0	√	0
	VMAXNMV	30	3	√	3
	VMAXNMAV	30	4	√	4
	VMINNM-S	30	2	√	2
	VMINNM-D	30	0	√	0
	VMINNM-Q	30	4	√	4
	VMINNMA	30	6	√	6
	VMINNMV	30	2	√	2
	VMINNMAV	30	1	√	1

上述的实验是对单个指令进行的测试, 此处举出一个运用不同指令组成的程序的测试用例. 此用例实现了神经网络中常用到的最大池化操作和全连接层操作的功能. 图 11 为用例中的神经网络的最大池化层和全连接层结构图. 在最大池化层中 (调用 VMAXV 指令实现), 输入是 16×16 的二维向量, 最大池化的窗口为 4×4, 因此输出为 4×4 的二维向量. 在全连接层中, 输入为 4×4 的二维向量通过向量内积操作 (调用 VMLAV 指令), 最终输出 4 个特征值. 同样的, 我们用 C 模拟器实现相同功能的比较对象. 关于正确性的判定, 是在同一数据下, 当 C 模拟器和 K 框架输出的结果相同时, 判定 K 框架得到的输出是正确的. 在此例子中, 我们生成了多组数据, 经过对比实验, C 模拟器和 K 框架得到的最终结果都是一致的, 作为判断测试用例是正确的依据.

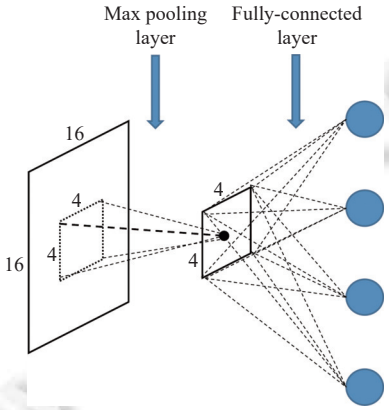

图 11 最大池化层与全连接层

6 总结与展望

通过分析 ARMv8.1-M 架构官方手册中关于 ML 的指令的语法与语义, 运用 K Framework 定义了关于 ARM

Helium MVE 技术的向量化 ML 指令的形式化执行语义. 详细介绍了如何运用 K 框架模块化的实现向量化 ML 指令语义规则以及如何利用 K 框架配置得到可执行的语义. 最终通过多个测试用例和官方基准用例, 对形式化的可执行语义进行测试, 验证了所有 K 框架中定义的 ML 指令语法语义及其运算逻辑.

本文所实现的 ML 指令只是 ARMv8.1-M 架构官方手册中 ARM Helium MVE 技术指令集的一部分, 且重点关注的是指令的算术运算执行逻辑, 因此未来工作包括: (1) 不仅关注向量化 ML 指令的功能语义的形式化, 还会关注其他向量化指令; (2) 运用 K 框架内置的验证功能, 如其内置的可达性验证功能和 SMT 工具调用功能, 验证程序正确性; (3) 处理器状态, 内存状态检验的操作语义, 以及异常处理等方面的形式化实现; (4) 并利用指令形式化语义对等价指令转换, 代码优化等方面进行验证.

References:

[1] ARMv8.1-M architecture reference manual. 2021. https://developer.arm.com/documentation/ddi0553/bo

[2] Roşu G, Şerbănuţă TF. An overview of the K semantic framework. The Journal of Logic and Algebraic Programming, 2010, 79(6): 397–434. [doi: 10.1016/j.jlap.2010.03.012]

[3] Şerbănuţă TF, Arusoaie A, Lazar D, Ellison C, Lucanu D, Roşu G. The K primer (version 3.3). Electronic Notes in Theoretical Computer Science, 2014, 304: 57–80. [doi: 10.1016/j.entcs.2014.05.003]

[4] Qian ZJ, Huang H, Song FM. Method of formal design and verification of OS on assembly layer. Ruan Jian Xue Bao/Journal of Software, 2016, 27(12): 3143–3157 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4851.htm [doi: 10.13328/j.cnki.jos.004851]

[5] Guo Y. Model, methods and tools for formal verification of assembly language [MS. Thesis]. Nanjing: Nanjing University, 2015 (in Chinese with English abstract).

[6] Dasgupta S, Park D, Kasampalis T, Adve VS, Roşu G. A complete formal semantics of x86-64 user-level instruction set architecture. In: Proc. of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Phoenix: ACM, 2019. 1133–1148. [doi: 10.1145/3314221.3314601]

[7] Wang F, Song F, Zhang M, Zhu XR, Zhang J. KRust: A formal executable semantics of rust. In: Proc. of the 2018 Int'l Symp. on Theoretical Aspects of Software Engineering. Guangzhou: IEEE, 2018. 44–51. [doi: 10.1109/TASE.2018.00014]

[8] Armstrong A, Bauereiss T, Campbell B, Reid A, Gray KE, Norton RM, Mundkur M, Wassell M, French J, Pulte C, Flur S, Stark I, Krishnaswami N, Sewell P. ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS. Proc. of the ACM on Programming Languages, 2019, 3: 71. [doi: 10.1145/3290384]

[9] Armstrong A, Bauereiss T, Campbell B, Flur S, Gray KE, Mundkur P, Norton RM, Pulte C, Reid A, Sewell P, Stark I, Wassell M. Detailed models of instruction set architectures: From pseudocode to formal semantics. In: Proc. of the 25th Workshop on Automated Reasoning (ARW 2018). 2018. 23–24.

[10] Reid A. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In: Proc. of the 2016 Formal Methods in Computer-aided Design. Mountain View: IEEE, 2016. 161–168. [doi: 10.1109/FMCAD.2016.7886675]

[11] Ellison C, Rosu G. An executable formal semantics of C with applications. ACM SIGPLAN Notices, 2012, 47(1): 533–544. [doi: 10.1145/2103621.2103719]

[12] Stefanescu A. MatchC: A matching logic reachability verifier using the K framework. Electronic Notes in Theoretical Computer Science, 2014, 304: 183–198. [doi: 10.1016/j.entcs.2014.05.010]

[13] Hjort R. Formally verifying WebAssembly with KWasm [MS. Thesis]. Gothenburg: Chalmers University of Technology, 2020.

[14] Li LY, Gunter EL. K-LLVM: A relatively complete semantics of LLVM IR. In: Proc. of the 34th European Conf. on Object-oriented Programming (ECOOP 2020). Berlin: ECOOP, 2020. 1–29. [doi: 10.4230/LIPIcs.ECOOP.2020.7]

[15] Guth D. A formal semantics of Python 3.3 [MS. Thesis]. Urbana: University of Illinois at Urbana-Champaign, 2013.

[16] Bogdănaş D, Roşu G. K-Java: A complete semantics of Java. In: Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Mumbai: ACM, 2015. 445–456. [doi: 10.1145/2676726.2676982]

[17] Legunsen O, Ul Hassan W, Xu XY, Roşu G, Marinov D. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: IEEE, 2016. 602–613.

[18] Park D, Stefănescu A, Roşu G. KJS: A complete formal semantics of JavaScript. In: Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Portland: ACM, 2015. 346–356. [doi: 10.1145/2737924.2737991]

[19] Guth D, Hathhorn C, Saxena M, Roşu G. RV-Match: Practical semantics-based program analysis. In: Proc. of the 28th Int'l Conf. on

Computer Aided Verification. Toronto: Springer, 2016. 447–453. [doi: 10.1007/978-3-319-41528-4_24]
[20] ARM Cortex-M55 processor deviceS. 2020. https://developer.arm.com/documentation/101273/r0p2

附中文参考文献:

[4] 钱振江, 黄皓, 宋方敏. 操作系统汇编级形式化设计和验证方法. 软件学报, 2016, 27(12): 3143–3157. http://www.jos.org.cn/1000-9825/4851.htm [doi: 10.13328/j.cnki.jos.004851]
[5] 郭毅. 汇编语言形式化验证的模型、方法和工具 [硕士学位论文]. 南京: 南京大学, 2015.

黄厚华(1995－), 男, 硕士生, CCF 学生会员, 主要研究邻域为形式化方法程序验证.

施晓牧(1987－), 女, 博士, CCF 专业会员, 主要研究领域为形式化方法, 定理证明技术及其在安全关键系统的应用.

刘嘉祥(1987－), 男, 博士, 助理教授, CCF 专业会员, 主要研究领域为形式化方法, 程序验证, 神经网络验证.