计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2021,30(11):20−26 [doi: 10.15888/j.cnki.csa.008347]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于 LLVM 的 RISC-V 自定义扩展指令支持方法①

王　鹏[1,2], 陈　影[1,3], 邢明杰[1]

[1](中国科学院 软件研究所, 北京 100190)
[2](中国科学院 深圳先进技术研究院, 深圳 518055)
[3](合肥工业大学 数学学院, 合肥 230601)
通讯作者: 邢明杰, E-mail: mingjie@iscas.ac.cn

摘　要: RISC-V 指令集架构具有模块化、可扩展等特性. 基于 RISC-V 架构的处理器, 可以在整数指令集的基础之上, 有选择地支持官方标准指令集扩展, 以及非标准的用户自定义指令集扩展. 这也意味着, 对于每个新增的自定义扩展指令集, 用户都需要自己在编译工具链中实现相应支持. 通过分析 LLVM 编译框架, 研究 RISC-V 自定义扩展指令支持的通用方法, 并以玄铁 C910 自定义指令集为例进行实现和验证. 为基于 LLVM 基础架构的 RISC-V 自定义指令集扩展研究与实现提供借鉴.

关键词: LLVM; RISC-V; 玄铁 C910

引用格式: 王鹏,陈影,邢明杰.基于 LLVM 的 RISC-V 自定义扩展指令支持方法.计算机系统应用,2021,30(11):20–26. http://www.c-s-a.org.cn/1003-3254/8347.html

Method of LLVM-Based RISC-V Custom Extended Instructions Support

WANG Peng[1,2], CHEN Ying[1,3], XING Ming-Jie[1]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China)
[3](School of Mathematics, Hefei University of Technology, Hefei 230601, China)

Abstract: The RISC-V instruction set architecture features modularization and scalability. On the basis of the integer instruction set, the RISC-V architecture based processors can optionally support the official standard and non-standard custom instruction set extensions. This also means that, for each new custom extended instruction set, users need to implement corresponding support in the compiler toolchain. After analyzing the LLVM compilation framework and researching the general methods supporting RISC-V custom extended instructions, we conduct the implementation and verification with the XuanTie C910 custom instruction set as an example. The results can provide references for the research and implementation of RISC-V custom instruction set extension based on LLVM infrastructure.

Key words: LLVM; RISC-V; XuanTie C910

由于 RISC-V 指令集架构具有开源、模块化、可扩展等特性, 近年来在许多领域迅速兴起. 国内外也出现了一些基于 RISC-V 进行指令集扩展的研究和实现. 例如, 神经网络指令集扩展研究[1−4], 加密指令集扩展研究[5], 平头哥公司发布的玄铁 C910 处理器[6] 等. 对于标准指令集扩展, RISC-V 社区会提供完整的工具链

① 基金项目: 中国科学院战略性先导科技专项 (C 类)(XDC05040200)
Foundation item: CAS Strategic Priority Program (Category C)(XDC05040200)
本文由“RISC-V 技术与生态”专题特约编辑武延军研究员、李玲研究员以及邢明杰高级工程师推荐.
收稿时间: 2021-04-27; 修改时间: 2021-05-21, 2021-06-08; 采用时间: 2021-06-11; csa 在线出版时间: 2021-10-22

支持[7], 而对于非标准的自定义指令集扩展, 则意味着需要用户自己实现工具链支持.

LLVM 编译框架具有模块化、可复用等特性[8–10], 适合用于快速搭建原型系统和二次开发. 目前 LLVM 社区已经对 RISC-V 体系结构进行支持. 本文通过对 LLVM 现有框架进行分析, 研究在 RISC-V 后端对自定义扩展指令集的支持方法, 为基于 LLVM 基础架构的 RISC-V 自定义指令集扩展研究与实现提供借鉴. 文章组织如下: 第 1 节介绍 RISC-V 指令集扩展, 包括标准指令集扩展和非标准的自定义指令集扩展; 第 2 节对 LLVM 框架进行分析, 重点分析现有的 RISC-V 体系结构相关部分; 第 3 节研究基于 LLVM 实现扩展指令支持的方法, 并以玄铁 C910 为例进行实现和验证; 第 4 节给出结论与展望.

1 RISC-V 指令集扩展

RISC-V 指令集架构被设计成由基础整数指令集和各种扩展指令集组成. 其中, 基础整数指令集非常精简 (目前最新版本的 RV32I 仅包含 40 条指令), 同时功能又足以支持编译器和操作系统[7–9]. 扩展指令集分为标准扩展指令集和非标准扩展指令集. 扩展指令集不仅支持固定宽度指令, 还可以支持可变长指令和 VLIW 指令. 为了能够有效支持各种指令集扩展, RISC-V 指令集架构在编码空间和命名约定等方面做了详细的设计和规划.

1.1 标准指令集扩展

标准指令集扩展涵盖了常用的功能支持, 并且相互之间不能存在指令编码冲突. 非特权指令集使用单个字母或者以 Z 开头的字母组合来命名, 特权指令集则使用 S (Supervisor 级别)、H (Hypervisor 级别) 或者 Zxm (Machine 级别) 开头的字母组合来命名. 其中, 字母 G 用来表示通用指令集扩展组合 IMAFDZicsr_Zifencei, 依次表示整数、乘除法、单精度浮点、双精度浮点、控制状态寄存器访问、取指栅栏指令集. 指令集名称不区分大小写, 名称之间可以使用下划线来分割, 并且后面可以有版本号信息.

RISC-V 国际基金会下面设有专门的任务工作组来负责标准指令集扩展规范的制定. 同时还设有工具链相关的任务工作组来负责推动开源社区工具链对标准指令集的支持, 从而对 RISC-V 的生态建设起到很好的支撑作用.

1.2 自定义指令集扩展

RISC-V 指令集架构允许并鼓励用户根据自己的需求来定制指令集扩展. 自定义的非标准指令集可以与它不支持的标准扩展或者非标准扩展之间存在指令编码冲突. 不过为了减少冲突, RISC-V 指令集规范也为自定义扩展指令集预留了 4 个主编码字段: 0b0001011 (custom-0), 0b0101011 (custom-1), 0b1011011 (custom-2) 和 0b1111011 (custom-3). 自定义扩展指令集使用以 X 开头的字母组合来命名.

比较有代表性的自定义指令集扩展实现是平头哥推出的玄铁 C910, 一款 12 级超标量流水线、3 发射、乱序执行的高性能 64 位嵌入式多集群多核 RISC-V 处理器. 其标准指令集架构为 RV64GCV, 并在此基础上增加了自定义扩展指令集和相应的控制状态寄存器, 用于增强计算、存储和多核等方面的性能. 扩展指令集的总体信息如表 1 所示.

表 1 玄铁 C910 扩展指令集

指令类别	描述
Cache	实现DCache、ICache、L2Cache的清除脏表项、无效表项等功能
多核同步	实现多核之间的同步、同步清空、同步广播等功能
算术运算	实现寄存器移位相加、乘累加、循环右移、寄存器传送等功能
位操作	实现比特或字节为0测试、字节倒序、快速找0或1、寄存器连续位提取等功能
存储	实现移位加载/存储、符号或零扩展加载/存储、基地址自增加载/存储、双寄存器存储/加载等功能

新增指令的位宽为 32 位固定长度, 其指令主编码使用的是 custom-0 预留编码. 新增控制状态寄存器的总体信息如表 2 所示.

表 2 玄铁 C910 扩展寄存器

控制状态寄存器类别	描述
用户模式	包括用户模式下的扩展浮点控制寄存器组
超级用户模式	包括超级用户模式下的控制状态扩展寄存器组、MMU扩展寄存器组
机器模式	包括机器模式下的控制状态扩展寄存器组、Cache访问扩展寄存器组、处理器型号扩展寄存器组

此外, 扩展指令集需要在机器模式控制状态寄存器 MXSTATUS 中开启扩展指令集使能位 THEADISAEE 的时候才能正常运用, 否则会出现非法指令异常.

2 LLVM 框架分析

最新的 LLVM 代码已经开始对 RISC-V 体系结构进行支持. 因此, 在 LLVM 中增加自定义扩展指令支持需要首先对 LLVM 框架[11], 特别是 RISC-V 相关部分有所熟悉[12].

2.1 LLVM 整体框架

LLVM 可以看作是一个编译基础设施, 由一系列的功能模块以及基于这些模块构建的工具集组成[13–15]. 其整体框架如图 1 所示.

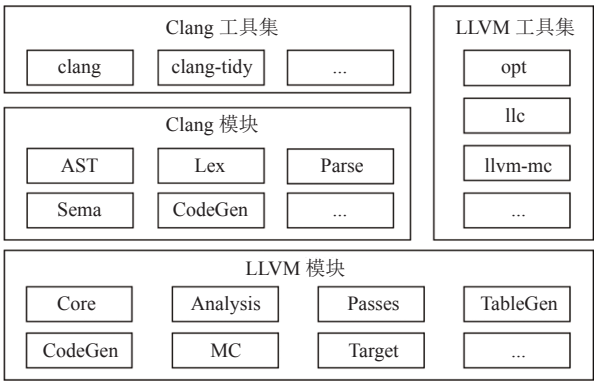

图 1 LLVM 整体框架

LLVM 主要涉及到编译器的中、后端, 其代码以模块的形式进行划分和实现, 包括中间表示、代码分析、优化和代码生成等[16–18]. 基于这些模块实现的工具集有优化器 (opt)、生成器 (llc)、汇编器 (llvm-mc) 等[19]. Clang 主要涉及到编译器的前端, 也是采用类似的形式进行模块化实现, 包括抽象语法树、词法分析、语法分析、语义分析和 LLVM 中间代码生成等. 基于这些模块实现的工具集有编译器 (clang)、静态检查工具 (clang-tidy) 等[20].

2.2 RISC-V 体系结构相关部分

为了能够支持多种目标体系结构 (X86、ARM、RISC-V 等), LLVM 的代码结构被划分成体系结构无关部分和相关部分, 如图 2 所示.

体系结构无关部分使用通用的算法来实现各种分析、优化以及代码生成 (涉及指令选择、指令调度、寄存器分配等), 并通过抽象接口来获取体系结构相关的信息, 执行相应的处理. 其中 RISC-V 体系结构相关信息主要包括:

1) 芯片特性. 描述芯片所支持的特性、对应的命令行参数和说明信息等.

2) 寄存器信息. 描述寄存器的编号、大小、存放数据类型、名称、类别、分配优先级等.

3) 指令信息. 描述指令格式、编码、操作数、指令选择匹配模式、对应的汇编代码等.

4) 调用约定. 描述需要被调用函数保存的寄存器列表等.

5) 调度模型. 描述调度资源、指令延迟周期、对调度资源的使用情况等.

6) 处理器模型. 描述处理器所支持的指令调度模型、芯片特性等.

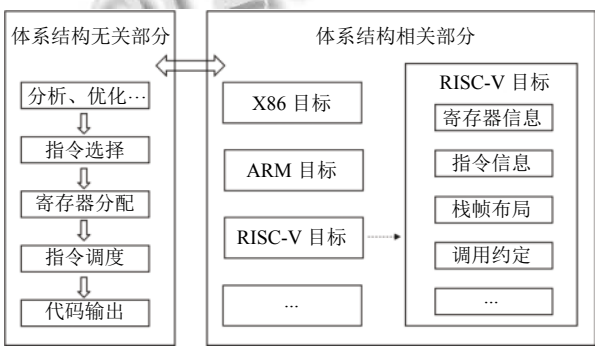

图 2 多目标体系结构支持框架

这些信息都是通过 LLVM 自带的 TableGen 语言来编写. TableGen 是一个领域专用语言, 用来帮助 LLVM 开发者来处理大规模的信息描述, 简化代码编写和维护工作. 其语法形式借鉴了 C++的类和模板, 并增加一些用于处理指令选择、指令编码的数据类型和操作.

图 3 给出了 TableGen 代码的处理流程: 在构建 LLVM 的时候, 用户使用 TableGen 编写的代码 (文件名通常以 td 为后缀), 会先通过工具 llvm-tblgen 进行解析, 然后在构建目录下生成 C++数据结构和代码片段 (文件名通常以 inc 为后缀), LLVM 源文件通过#include 方式将这些生成的文件包含进来.

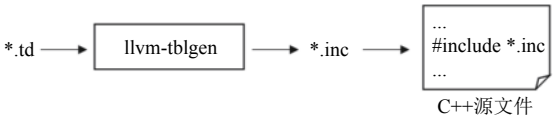

图 3 TableGen 代码处理流程

TableGen 本身只是描述信息记录, 至于具体生成什么样的 C++代码, 则需要 LLVM 开发者来实现相应

的 C++代码生成后端. 除此之外, 还有一些目标体系结构抽象接口不适合使用 TableGen 来描述和自动生成, 这部分则直接使用 C++代码来实现, 主要包括:

1) 栈帧布局. 处理栈空间的增长方向、地址对齐方式、局部变量地址偏移以及在函数开头和结尾处插入栈帧维护代码等.

2) 部分指令选择处理. 例如指令 DAG 图构建过程中的类型和操作合法化、函数调用和返回的处理、特殊 DAG 节点的处理等.

3) 部分寄存器信息. 例如获取预留寄存器列表、消除帧指针等.

4) 部分指令信息. 例如判断指令是否为对栈槽进行加载或存储、对分支跳转指令的分析和处理等.

5) 汇编器和反汇编器的接口函数实现.

6) 机器代码层 (MC) 的处理. 例如 ELF 文件写出、重定位信息、汇编指令打印等.

2.3 LLVM 测试框架

LLVM 源码包中自带的测试用例有两种: 回归测试与单元测试. 其中单元测试使用 Google C++测试框架编写, 用来测试 LLVM 的功能单元. 回归测试使用 LLVM 测试框架编写, 用来验证特定功能点或者已经修复的问题. 这些测试用例需要在每次提交代码之前运行通过, 从而避免新的改动出现回退现象.

3 扩展指令支持方法

我们可以将基于 LLVM 的扩展指令支持分为汇编层面支持和编译层面支持. 其中, 编译层面支持是指可以将用户编写的高级语言程序转换成含有扩展指令的汇编程序或者机器指令编码. 编译层面支持有两种常见的方式: 一是在高级语言中定义新的数据类型和编译器内建函数, 使得用户可以直接通过函数调用的形式来使用扩展指令; 二是通过编译优化技术将中间代码自动转换成机器特定的扩展指令.

本文主要研究汇编层面的支持方法. 汇编层面支持是指可以将用户编写的含有扩展指令的汇编程序转换成机器指令编码. 根据前面对 RISC-V 指令扩展的介绍以及 LLVM 框架的分析, 可以看到汇编层面支持大体需要完成如下工作:

1) 定义新的芯片特性, 添加命令行选项;

2) 针对新增加的寄存器, 实现相应的寄存器信息描述以及可能涉及到的抽象接口;

3) 针对新增加的指令, 实现相应的指令信息描述以及可能涉及到的抽象接口;

4) 根据指令集扩展情况, 可能需要对汇编器和反汇编器的接口函数进行更新实现;

5) 根据指令集扩展情况, 可能需要在机器代码层增加相应的处理;

6) 编写测试用例, 对新增加的指令集扩展进行测试和验证.

接下来, 我们将以玄铁 C910 的扩展指令支持为例, 对主要涉及到的工作内容进行具体介绍. 我们已经将完整的代码实现进行了开源, 项目地址为: https://github.com/isrc-cas/c910-llvm.

3.1 定义芯片特性, 添加命令行选项

我们在 RISCV.td 文件中, 通过 TableGen 语言来描述玄铁 C910 所支持的指令扩展特性. 参见代码示例 1, 其中特性 FeatureExtXuantie 继承自 SubtargetFeature, 并通过模板参数给出名字、属性、属性值、文字描述信息. 同时, 定义一个断言 HasExtXuantie, 可以在指令描述中用来设置指令选择和汇编指令匹配的判断条件.

代码示例 1. 定义芯片特性

```
def FeatureExtXuantie
   :SubtargetFeature<"xuantie", "HasExtXuantie",
      "true", "'Xuantie' (Xuantie Custom Instructions)">;
def HasExtXuantie
   :Predicate<"Subtarget->hasExtXuantie()">,
      AssemblerPredicate<"FeatureXcache">;
```

除此之外, 还定义了一个命名为 c910 的处理器模型. 从而, 用户可以在汇编器 llvm-mc 的命令行中使用 -mattr=+xuantie 或者-mcpu=c910 来开启对玄铁 C910 扩展指令的支持特性.

3.2 描述寄存器信息

由于玄铁 C910 只对控制状态寄存器进行了扩展, 并没有增加新的通用寄存器或者其他用来存放数据、参与寄存器分配的寄存器, 因此, 基于现有的 RISC-V 代码框架, 对这部分进行支持所需要做的工作比较简单. 我们在 RISCVSystemOperands.td 文件中, 使用 TableGen 语言对它进行描述. 代码示例 2 给出了部分扩展控制状态寄存器的描述, 例如, MXSTATUS 寄存器继承自父类 SysReg, 并通过模板参数给出它的名字和编码.

代码示例 2. 描述状态寄存器信息

```
def MXSTATUS : SysReg<"mxstatus", 0x7C0>;
def MHCR : SysReg<"mhcr", 0x7C1>;
def MCOR : SysReg<"mcor", 0x7C2>;
def MCCR2 : SysReg<"mccr2", 0x7C3>;
```

现有的 RISC-V 代码框架已经实现了 SysReg 的定义, 以及相应的支持. 所以, 用户只需要添加一行 TableGen 描述即可. 然后在汇编程序中, 便可以使用该寄存器的名字作为指令的符号操作数.

3.3 描述指令信息

我们以玄铁 C910 的位操作扩展指令 EXT (寄存器连续位提取符号位扩展指令) 和 EXTU (寄存器连续位提取零扩展指令) 为例介绍如何使用 TableGen 语言来描述指令信息. 图 4 给出了这两条指令的编码格式.

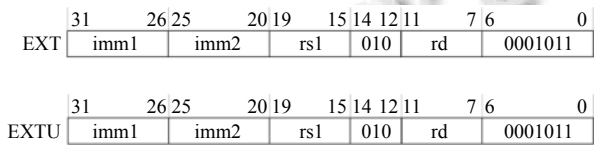

图 4 位操作扩展指令 EXT 和 EXTU

其汇编语法形式如下:

1) ext rd, rs1, imm1, imm2

2) extu rd, rs1, imm1, imm2

可以看到 EXT 和 EXTU 两条指令具有相同的操作数, 只不过是 12–14 位的编码不同. 因此, 可以将相同指令格式提取出来, 使用 TableGen 的 class 类型定义成一个模板类, 从而避免冗余的指令信息描述. 图 5 给出了两个指令格式, 其中模板类 RVInst 是在 RISCVInstrFormats.td 文件中定义, 用来表示 32 位的 RISC-V 指令格式. 目前 LLVM 中所有的 RISC-V 扩展指令都是继承自该父类.

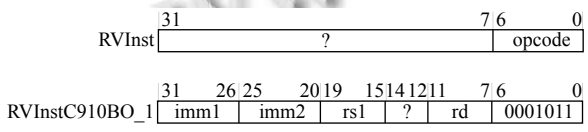

图 5 指令格式

我们参照 LLVM 现有的 RISC-V 代码框架, 新增加一个 RISCVInstrFormatsC910.td 文件用来定义扩展指令格式. 其中模板类 RVInstC910BO_1 用来表示 EXT 和 EXTU 这样的位操作扩展指令格式, 它的主编码为 0b0001011 (指令集规范中预留的 custom-0 主编码). 然后新增加一个 RISCVInstrInfoC910.td 文件用来定义具体的扩展指令, 以及细化的指令格式模板子类.

代码示例 3 给出了 RVInstC910BO_1 的定义, 其模板参数分别为 12–14 位的编码, 指令的输出和输入操作数, 汇编指令字符串. 然后, 在类的定义中根据这些参数来设置指令相应字段的值. 具体的代码含义可以参照 TableGen 语言文档资料. 通过 TableGen 提供的这种抽象和继承机制, 我们可以很方便的实现对玄铁 C910 扩展指令集的支持.

代码示例 3. 描述指令信息

```
class RVInstC910BO_1<bits<3> funct3, dag outs, dag ins, string
opcodestr, string argstr>
  : RVInst<outs, ins, opcodestr, argstr, [], InstFormatOther> {
  bits<6> imm1;
  bits<6> imm2;
  bits<5> rs1;
  bits<5> rd;
  let Inst{31-26} = imm1;
  let Inst{25-20} = imm2;
  let Inst{19-15} = rs1;
  let Inst{14-12} = funct3;
  let Inst{11-7} = rd;
  let Opcode = OPC_CUSTOM0.Value;
}
```

3.4 测试验证

最后, 我们通过编写测试用例, 来验证对玄铁 C910 扩展指令的支持情况. 参照现有测试框架, 在 test/MC/RISCV 目录下新增一个 c910-valid.s 文件用来测试有效的汇编指令, 同时新增一个 c910-invalid.s 文件用来测试对无效指令的错误处理. 代码示例 4 中给出了测试用例的开头部分.

代码示例 4. 测试用例

```
# RUN: llvm-mc %s -triple=riscv64 -mcpu=c910 -riscv-no-aliases -
show-encoding \
# RUN: | FileCheck -check-prefixes=CHECK-ASM,CHECK-ASM-
AND-OBJ %s
# CHECK-ASM-AND-OBJ: ext a0, a1, 4, 1
# CHECK-ASM: encoding: [0x0b,0xa5,0x15,0x10]
ext a0, a1, 4, 1
```

前两行是要执行的测试命令, 由 RUN 开头并且嵌套在代码注释中. LLVM 测试工具 llvm-lit 会根据这些命令来调用汇编器 llvm-mc, 然后将输出结果传送给检查工具 FileCheck. FileCheck 工具会根据注释中 CHECK 关键字开头的内容来对比汇编生成结果.

代码示例 5 是汇编指令测试用例, 有效汇编指令 c910-valid.s 文件中包含了新增的 99 条自定义玄铁 C910

指令集, 我们运行 llvm-lit 来单独测试 c910-valid.s 汇编文件的正确性, 运行结果输出 Expected Passes 1, 说明所有的新增自定义指令的汇编编码都是正确的.

代码示例 5. 汇编指令测试用例

```
$ ./bin/llvm-lit -v ../test/MC/RISCV/c910-valid.s
-- Testing: 1 tests, single process --
PASS: LLVM :: MC/RISCV/c910-valid.s (1 of 1)
Testing Time: 0.30s
  Expected Passes : 1
```

代码示例 6 是新增寄存器测试用例, 在控制与状态寄存器文件 user-csr-names.s 中, 我们添加了玄铁 C910 扩展寄存器 fxcr, 它的功能是用于浮点扩展功能开关和浮点异常累积位, 我们对新增寄存器进行指令编码, 然后用 llvm-mc 求出 fxcr 的编码, 同时将寄存器别名添加到 user-csr-names.s 汇编文件中.

代码示例 6. 新增寄存器测试用例

```
User@dacent:~/tools/c910-project/c910-llvm/test/MC/RISCV
$ vim user-csr-names.s
# fxcr
# name
# CHECK-INST: csrrs t1, fxcr, zero
# CHECK-ENC: encoding: [0x73,0x23,0x00,0x80]
# CHECK-INST-ALIAS: csrr t1, fxcr
# uimm12
# CHECK-INST: csrrs t2, fxcr, zero
# CHECK-ENC: encoding: [0xf3,0x23,0x00,0x80]
# CHECK-INST-ALIAS: csrr t2, fxcr
# name
csrrs t1, fxcr, zero
# uimm12
csrrs t2, 0x800, zero
```

代码示例 7 是 99 条新增玄铁 C910 指令中 ff0 指令, ff0 指令是快速找 0 指令, 我们用 llvm-mc 进行测试, 编译选项选择 mcpu=c910 和 mattr=+c910, 可以确定 ff0 指令对应的编码形式.

代码示例 7. ff0 指令汇编测试用例

```
User@dacent:~/tools/c910-project/c910-llvm/build/bin$ echo "ff0 a0,a1" | ./llvm-mc --triple=riscv64 -mcpu=c910 -mattr=+c910 -show-encoding -show-inst
  .text
  ff0a0, a1
  # encoding: [0x0b,0x95,0x05,0x84]
  # <MCInst #399 FF0
  # <MCOperand Reg:11>
  # <MCOperand Reg:12>>
```

代码示例 8 是利用 llvm-mc, 在选定了编译选项是 mcpu=c910 和 mattr=+c910 之后, 对汇编编码进行反汇编测试, 查看执行之后得出的是否是对应的 ff0 汇编指令.

代码示例 8. ff0 指令反汇编测试用例

```
User@dacent:~/tools/c910-project/c910-llvm/build/bin$ echo "0x0b,0x95,0x05,0x84" | ./llvm-mc -disassemble --triple=riscv64 -mcpu=c910 -mattr=+c910 -show-encoding -show-inst
  .text
  ff0a0, a1
  # encoding: [0x0b,0x95,0x05,0x84]
  # <MCInst #399 FF0
  # <MCOperand Reg:11>
  # <MCOperand Reg:12>>
```

表 3 中列出了我们目前支持的所有新增玄铁 C910 指令的汇编测试, 反汇编测试, 编译选项 mcpu=c910 测试和无效操作数测试, 说明了新增玄铁 C910 自定义扩展指令集在 LLVM 中具有功能完备性支持.

表 3 功能完备性测试

指令类别	指令总个数	汇编测试	反汇编测试	编译选项测试	无效操作数测试
Cache指令	20	20	20	20	20
多核同步	4	4	4	4	4
算术运算	11	11	11	11	11
位操作	7	7	7	7	7
存储指令	57	57	57	57	57

除此之外, 我们还在一个 C 文件 test.c 中, 使用内联汇编的方式编写了一条上述已定义的玄铁 C910 扩展指令, 使用 clang 编译生成汇编文件, 然后用 llvm-mc 将汇编文件 test.s, 编译选项是 mcpu=c910, 编译成目标文件, 之后可以通过反汇编来验证正确性.

代码示例 9. 玄铁 C910 新增指令内联汇编测试用例

```
$ ./bin/clang --target=riscv64-unknown-elf test.c -S -o test.s
$ cat test.c
int main(){
  int a,b,c;
  a = 1;
  b = 2;
  asm volatile
  (
    "mula %[z], %[x], %[y]\n\t"
    : [z] "=r" (c)
    : [x] "r" (a), [y] "r" (b)
  );
  if ( c == 0 ){
```

```
    return -1;
  }
  return 0;
}
$ ./bin/llvm-mc test.s -triple=riscv64 -mcpu=c910 -show-encoding -show-inst --filetype=obj -o=test.o
```

4 结论与展望

本文通过对 RISC-V 指令集扩展和 LLVM 框架的分析, 给出了在 LLVM 中实现对 RISC-V 自定义指令集扩展的支持方法. 结合玄铁 C910 的例子可以看到, 在现有 LLVM 框架下, 对于 32 位指令集扩展的汇编层面支持比较容易实现.

对于其他宽度的指令扩展支持, 包括可变长指令和 VLIW 指令扩展的支持, 还需要做进一步的分析研究. 除此之外, 对扩展指令的编译层面支持涉及到编译器的前、中和后端多个方面. 后续工作中, 将重点研究这部分内容.

参考文献

1 娄文启, 王超, 宫磊, 等. 一种神经网络指令集扩展与代码映射机制. 软件学报, 2020, 31(10): 3074–3086. [doi: 10.13328/j.cnki.jos.006071]

2 Patsidis K, Konstantinou D, Nicopoulos C, *et al*. A low-cost synthesizable RISC-V dual-issue processor core leveraging the compressed instruction set extension. Microprocessors and Microsystems, 2018, 61: 1–10. [doi: 10.1016/j.micpro.2018.05.007]

3 Jiao Q, Hu W, Wen Y, *et al*. Design of a convolutional neural network instruction set based on RISC-V and its microarchitecture implementation. In: Qiu MK, ed. Algorithms and Architectures for Parallel Processing. Cham: Springer, 2020. 82–96.

4 Wu N, Jiang T, Zhang L, *et al*. A reconfigurable convolutional neural network-accelerated coprocessor based on RISC-V Instruction set. Electronics, 2020, 9(6): 1005. [doi: 10.3390/electronics9061005]

5 Marshall B, Newell GR, Page D, *et al*. The design of scalar AES instruction set extensions for RISC-V. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020, 2021(1): 109–136.

6 Chen C, Xiang XY, Liu C, *et al*. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: Industrial product. 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). Valencia: IEEE, 2020. 52–64.

7 Waterman A, Lee Y, Avizienis R, *et al*. The RISC-V instruction set. 2013 IEEE Hot Chips 25 Symposium (HCS). Stanford: IEEE, 2013. 1.

8 Lattner C, Adve V. The LLVM compiler framework and infrastructure tutorial. Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing. Berlin Heidelberg: Springer, 2004. 15–16.

9 Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. International Symposium on Code Generation and Optimization (CGO). San Jose: IEEE, 2004. 75–86.

10 Kenneth CC. 编译原理及实践 (计算机科学丛书). 北京: 机械工业出版社, 2011.

11 Lattner C. LLVM language reference manual. https://llvm.org/docs/LangRef.html. [2020-10-05].

12 Waterman A, Asanović K. The RISC-V instruction set manual. https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf. (2017-05-07) [2020-10-05].

13 胡敏. 基于 LLVM 编译架构的 CSKY 后端移植 [硕士学位论文]. 杭州: 浙江大学, 2014.

14 Kim JJ, Lee SY, Moon SM, *et al*. Comparison of LLVM and GCC on the ARM platform. 2010 5th International Conference on Embedded and Multimedia Computing. Cebu: IEEE, 2010. 1–6.

15 Aho AV, Lam MS, Sethi R, *et al*. Compilers: Principles, Techniques, and Tools. 2nd ed. New York: Addison-Wesley Longman Publishing Co. Inc., 2006.

16 张祖羽. 基于 LLVM 的 C*Core 后端移植研究 [硕士学位论文]. 哈尔滨: 哈尔滨工程大学, 2012.

17 任胜兵, 卢念, 张万利, 等. 基于 LLVM 架构的 Nios II 后端快速移植. 计算机应用与软件, 2011, 28(12): 22–25, 50. [doi: 10.3969/j.issn.1000-386X.2011.12.006]

18 Brethour T, Stanley J, Wendling B. An LLVM implementation of SSAPRE [Master 's thesis]. Urbana: University of Illinois at Urbana-Champaign, 2002.

19 刘俊波. 基于 LLVM 的编译器移植中关键技术研究 [硕士学位论文]. 天津: 南开大学, 2012.

20 董峰. LLVM 编译系统结构分析与后端移植 [硕士学位论文]. 上海: 上海交通大学, 2007.