上海交通大学

硕士学位论文

LLVM编译系统结构分析与后端移植

姓名：董峰

申请学位级别：硕士

专业：计算机系统结构

指导教师：付宇卓

20070119

LLVM 编译系统结构分析与后端移植

摘　要

伴随着复杂逻辑设计这样的挑战的出现，诞生了许多的基于平台的设计。在为硬件设计提供了一个好的解决方案的同时，基于平台的设计仍然需要给出一个完备的开发工具链，包括编译器、汇编器、仿真器等等。因此出现了 ADL(Architecture Description Language)以帮助进行工具链的生成。其中的 MADL(Mescal Architecture Description Language) 是一个良好设计但尚不完善的 ADL。UADL(Unified Architecture Description Language)项目致力于在 MADL 的基础之上提供一个工具链自动生成的更好方法。本文是 UADL 项目的一部分，提供了对 LLVM(Low Level Virtual Machine)编译系统的后端移植，这构成了 UADL 项目进行编译器自动生成研究的基础。LLVM 架构是美国伊利诺斯大学开发的开放源代码编译器架构，本文介绍了 LLVM 架构的基本构成，其组成部分的具体功能，详细分析了该架构的后端移植机制，包括寄存器描述、指令集描述、汇编输出等移植机制的各个方面，然后给出实现 ARM 后端移植的具体工作细节，最终实现了 LLVM 架构对 ARM 处理器后端的支持。

关键词：LLVM，编译器，ARM，后端移植

ANALYSIS OF LLVM COMPILER INFRASTRUCTURE AND BACKEND PORTING FOR ARM

ABSTRACT

With the rising challenges, such as the complexity of logic design, many platform-based designs came out. While providing a good solution for hardware design, platform-based designs still need to supply a qualified development toolchain, including compiler, assembler, simulator and etc. Then ADL (Architecture Description Language) came out to facilitate the generation of the toolchain. MADL (Mescal Architecture Description Language) is a well-designed but immature one. Our UADL (Unified Architecture Description Language) project aims to offer a better toolchain auto-generation methodology on the basis of MADL. This paper is part of the UADL project and gives a solution to LLVM (Low Level Virtual Machine) backend porting, which can be the basis of the research of compiler auto-generation for UADL project. LLVM Infrastructure is an open-source compiler structure developed by University of Illinois at Urbana-Champaign. In this paper, first the components of LLVM Infrastructure and their corresponding functions are introduced. Then LLVM Infrastructure's backend-porting scheme is analyzed, including register description, instruction description, assembly output etc. Finally, the details of porting LLVM Infrastructure on ARM are given and the LLVM Infrastructure's support for ARM backend is arrived.

KEY WORDS: LLVM, Compiler, ARM, Backend Porting

图片目录

表格目录

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：董峰

日期：2007 年 01 月 19 日

上海交通大学

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在___年解密后适用本授权书。

本学位论文属于

不保密√□。

（请在以上方框内打“√”）

学位论文作者签名：董峰　　指导教师签名：付宇卓

日期：2007 年 01 月 19 日　　日期：2007 年 01 月 19 日

1 绪论

1.1 课题研究背景

自从 1947 年美国贝尔实验室发明了晶体管以来，集成电路已经被广泛地使用，计算机、通信设备、汽车电子、消费电子，所有这些设备中都包含有集成电路，它们极大地改善了人们的日常生活。但伴随着集成电路的广泛使用，人们对它们的要求也越来越高。对产品用户而言，要求更快的速度、更多的功能、更低的功耗、更小的尺寸、更稳定的性能；对生产厂商而言，为了获得更大的利润，需要使用最少的成本在最短的时间内将产品上市。这一切都对现在的电子系统设计工程师和科学家提出了更多的要求，带来了更大的挑战。对应到电子系统设计，这些要求和挑战中最难解决的有：

a．功耗控制。传统电池提供的能量有限，如镍铬电池每磅仅能提供 20W*h 的能量，电源技术每 5 年最大能将电池的性能提高 30%[1]，而当今的 SoC 已达到百瓦量级，如 Intel 的 Itanium 2 功耗约 130 瓦[2]，这导致封装制冷成本提高。

b．复杂的逻辑设计。1965 年提出的摩尔定律距今已经 40 多年，芯片产业的确在按照这一规律在发展进步，即单位面积芯片上的晶体管数量大约每 18 个月增加一倍。图 1 为摩尔定律在 Intel 公司芯片上的体现[3]。工程师们要在短时间内充分利用单个芯片上可容纳的越来越多的晶体管，就必须采用更有效的快速电子系统设计方法。

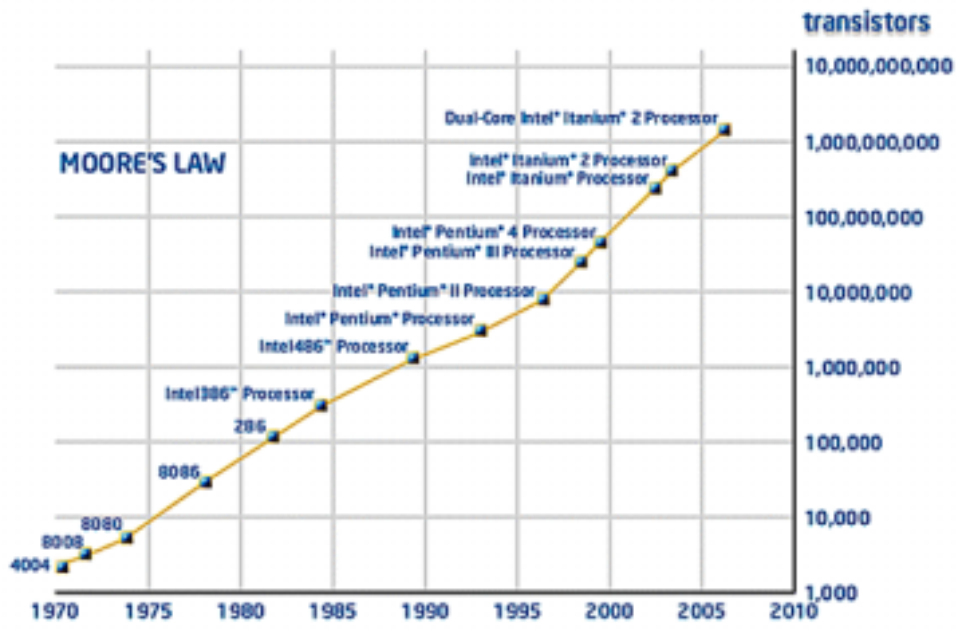

图 1 摩尔定律在 Intel 公司芯片上的体现

Fig. 1 Moore's Law on Intel's processors

c．生成产生的 NRE（Non-Recurring Engineering）费用。在生成芯片的过程中有一项是光学曝光[4]，随着工艺尺寸的不断缩小，不得不采用复杂的解析度增强技术，这些技术导致了更多的花费。图 2 显示了不同工艺尺寸下的掩模制作费用[5]。

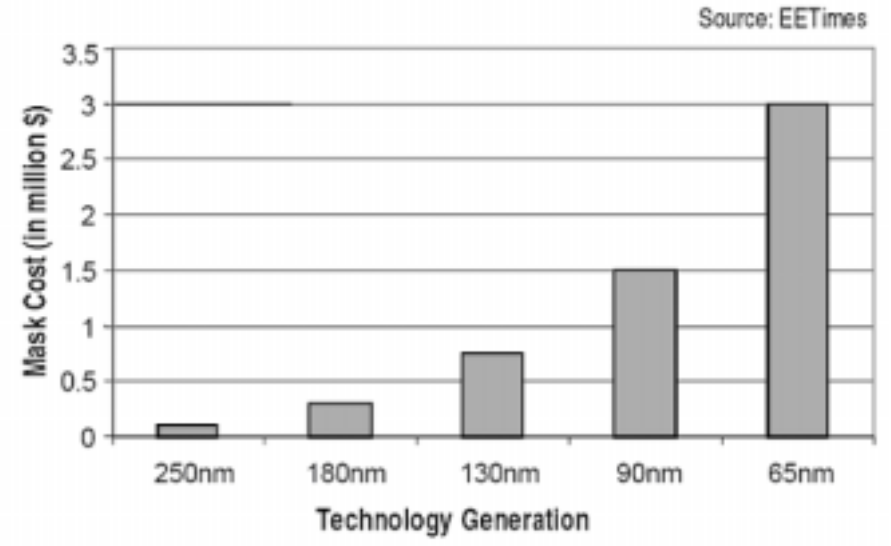

图 2不同工艺尺寸下的掩模制作费用

Fig. 2 Mask cost under different technology generation

除了以上这些困难，还有其他影响生产的因素，如信号完整性，电磁干扰等等。再加上越来越复杂的功能，和与之产生的相应的芯片测试和验证工作，导致了现在越来越多的设计失败和越来越多的流片次数（见图 3）。这些因素使得当今的电子系统设计是高风险和高成本的“双高”产业，即使是一个成功的设计也要通过巨大的产品生产量来平摊设计成本。调查显示，ASIC 项目的开展数目从 1998 年的大约 5000 个到 2002 年降为不到 1500 个[6]。

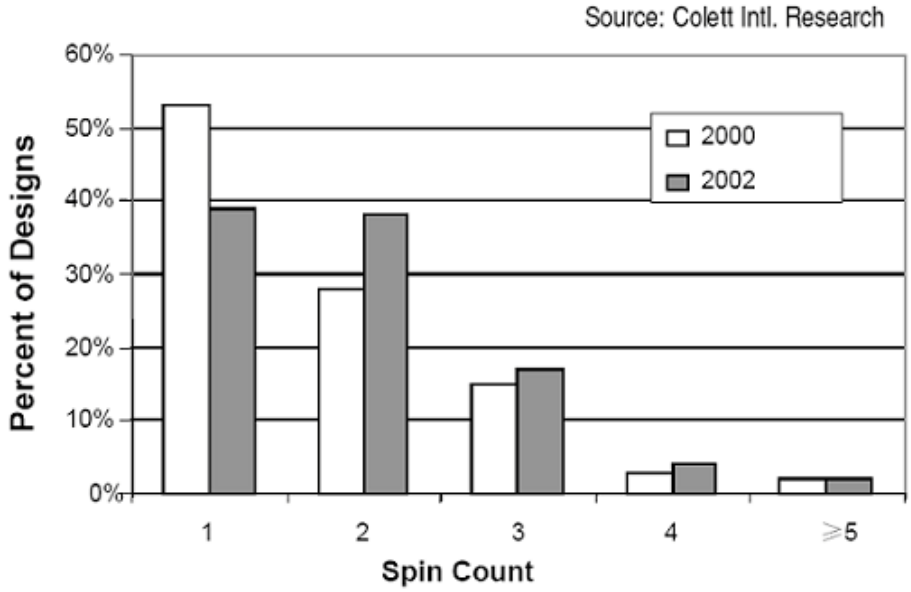

图 3 IC 设计中的不同流片次数

Fig. 3 Number of spins in IC Designs

由于设计的复杂性和生产效率之间的差距越拉越大，必须有一种全新的设计方法来简化设计的复杂性，减少重新流片的次数，降低 NRE 成本，让电子系统设计回到一个合理的范围，这些要求触发了基于平台设计的出现。Vincentelli 在“Defining platform-based design”一文中对基于平台的设计做了总结[7]。在该文中，平台被定义为“设计流程中的一个抽象层，它使得对设计流程中下一个抽象层的改进优化工作变得更加方便”。这个定义使平台这一概念可广泛地用于从抽象级最高的电子设计级到抽象级最低的硅片实现级。平台的本质是使设计分成若干可以被快速理解的不同层次，让设计的复杂程度降低到适当的程度，即让每一层次可以单独快速地被实现，能在每一层内进行充分有效地改进和优化，并使得因为层次划分而不能或不易进行的全局优化降到最低的一种设计方法。

在各种电子系统中，现有的硬件设计平台包括：基于全定制的版图的，基于可

重配置结构的，基于现场可编程门阵列（field-programmable gate array，FPGA）的，基于专用指令集处理器（application-specific instruction set processor, ASIP）的，基于数字信号处理器(digital signal processor, DSP)或通用处理器（general-purpose processor, GPP）的。所有这些平台依次提供相对越来越高的灵活性，但是同时是相对越来越低的效率（更高的功耗和更低的速度）。这使得对于不同的应用可以采用不同的平台进行电子系统设计。同时，这些平台的软件可编程性（software programmablitity）也依次提高，其中基于 ASIP 的平台由于具有以下的优势被越来越多的 SoC 设计者看好：

a．比全定制版图、可重配置结构、FPGA 有更好的灵活性；软件可编程带来的灵活性，可以加快开发周期，减少芯片流片次数，大大降低硬件实现相关的开发成本，使电子系统设计的 NRE 成本得到控制；

b．比 DSP 或 GPP 有更强的应用针对性，在去除不必要的通用性功能的同时提供优化特定应用的功能；对这些特定应用，有更好的效率和更低的功耗。

对于每一种平台，在提供相应执行应用的硬件基础的同时，都需要为用户提供相应的开发软件套件。这些套件包括高级语言编译器，汇编器，反汇编器，连接器，指令级仿真器，周期精确级仿真器，以及其他一些二进制工具，甚至可以提供在该平台上的操作系统。然而这些套件软件的开发却不是一件容易的事情，往往仅是一个高级语言编译器就需开发人员编写上万甚至是几十万行的代码。如此高的开发代价，对出片量很大的应用，如 GPP 来说，可以通过将开发成本平摊到每一片芯片的成本中去以降低单个芯片的成本。但是对其他的单个品种出片量不大，但是总量却占到整个芯片产业出片量 90%以上[8]的其他芯片而言，如此高的软件套件开发代价几乎是“ 不可承受之重 ”。

面对昂贵的开发代价，大多数的开发人员都放弃为新硬件平台开发全新的软件套件，而是转向复用一些已有的可移植的软件开发平台。在这些人中，又有许多开人选择了 GNU[50]工具链。GNU 全称为“ GNU's Not UNIX ”，是一个主要由 FSF（Free Software Foundation）[9]和一些志愿者支持开发的项目。GNU 软件工具链包括，高级语言编译器，汇编器，反汇编器，连接器，调试器等。它们在实现的时候就被设计成是可移植的，现在已经被移植到了许多的硬件上。然而，GNU 工具链的文档残缺不详，长久不更新，且软件工具链的移植工作也相当地困难，只有很少的一些程序员有能力做到。

目前国内有不少这方面的研究，如浙江大学的[11][12][13]，湖南大学的[14]。但这些研究的重点都是放在利用 GCC 进行手工后端移植上，很少考虑到移植过程

的自动化。即使考虑到移植的自动化[14]，也仅是利用已有的不很健全的工具[15]进行自动化，很少有自己的创新和较大改进。另外，关于工具链中其他工具，如汇编器，反汇编器的研究非常少，仅见的研究只有台湾国立台湾大学信息工程学研究所所做的工作[16]，该工作产生了一个可用于 ASIP 的自动二进制工具产生器。

相比而言，在工具链方面国外的应用相当多，如 GCC 后端支持的所有处理器几乎都是编译器移植方面的工作。但这类工作都是手工完成的，完全依靠工具自动生成的处理器移植仍未见报道。但有部分自动化的辅助工具，如 CGEN[10]。同样，手工移植其他二进制工具的工作也有许多，比较少见的是自动化的产生工具。

ADL 的出现使得这个问题有了更好的解决。ADL 是一类语言的统称，它通过对结构的详细描述信息自动生成电子系统设计所需的软件套件，可以包括指令级仿真器，周期精确级仿真器，高级语言编译器，汇编器，反汇编器，连接器，以及其他的一些二进制工具。目前的 ADL 可以大致分为三类：结构型 ADL，行为型 ADL 以及混合型 ADL[17][18]。

结构型 ADL 利用部件的网表（netlist）在处理器的逻辑级或者是微结构级（microarchitecture）描述结构细节，它们适合于综合成硬件和产生周期精确级的仿真器。MIMOLA[19]和 UDL/I[20]就是结构型 ADL。

行为型 ADL 描述处理器的指令集，它们则更适合于产生指令级仿真器和编译器。这种类型的 ADL 包括，ISDL[21]，CSDL[22]和早期版本的 nML[23]。

混合型 ADL 一般介于以上两种 ADL 的中间，比结构型 ADL 抽象级更高，它描述处理器的指令集，同时比行为型 ADL 抽象级略低，它可以用于描述较粗粒度（coarse-grained）的结构信息，因此，利用它可以同时产生指令级仿真器，周期精确级仿真器和编译器。大多数的 ADL 是混合型 ADL，这其中有：UPFAST[24]，BUILDABONG[25]，LISA[26]，RADL[27]，ArchC[28]，EXPRESSION[29]，PRMDL[30]，HMDES[31]，Maril[32]，TIE[33]，BABEL[34]和最新版本的 nML[35]。

国内在电子系统设计自动化方面的研究相对较少，所知的仅有的两个关于 ADL 的研究分别是来自中国科学技术大学和台湾清华大学的项目。中国科学技术大学的项目是关于 XP-ADL（eXtensible exPloration ADL）的，相关具体情况见该校两名博士的毕业论文[36][37]。项目主要提出并部分实现了支持低功耗 SoC 体系结构设计的设计环境框架，以 XML 为基础定义的通用体系结构描述语言 XP-ADL，该 XP-ADL 继承了 XML 的语言特征，其描述文档具有清晰的语义，并具有良好的可扩展性，并以此实现了以指令级功耗模型和微结构层功耗模型综合而构成的 SoC 系统级功耗模型，给出了基于此模型进行周期精确的功耗评估的方法。该项目是大

陆目前可见唯一的关于 ADL 的项目，其中有许多工作是之前大陆研究领域未做过的，但同时也有相当大的不足：

a．该项目只是将功耗和性能的优化集成在代码生成过程中，而且代码生成框架只能局限在基本块的范围内，可进行的优化方法相对少很多，优化力度更是有限；

b．不支持流水线的重定向，不能解决流水冲突，更不用说异常处理。而如今的处理器几乎无一例外地采用了流水线重定向和流水冲突解决技术，以此来提高处理器的有效执行速度；没有这两项技术的处理器流水线深度越深，执行效率越差，流水线停顿代价极大；

c．只能产生编译器和仿真器，不支持其他的软件套件的产生；

d．未来的 SoC 将包含支持多线程体系结构的异构多处理器、ASIC 和多种存储器(如 DRAM 和闪存)，而 XP-ADL 仅支持单处理器，不适合进行多核处理器设计探索。

台湾清华大学的项目主要是由该校的集成电路设计技术研发中心领导进行的。该项目以“网路安全处理器研发”为基点，进行相关处理器架构，设计平台，验证与测试的研究。研究的预计成果包括一个可用于 SoC 芯片设计的 ADL，利用该 ADL 可自动生成相应的软件套件以及可综合的硬件代码。其中项目编译器部分将采用 GNU 项目的 GCC 编译器作为可移植的编译器。项目紧密结合工程研究院及产业界公司，如创意电子源捷科技等，预计将帮助台湾 SoC 产业加速整合，取得技术领先 [37][38] 。

相对于国内的研究情况，国外在电子系统设计自动化方面的研究较多，同时也产生了许多的 ADL 语言，如前文中提到过的 MIMOLA[19]，UDL/I[20]，ISDL[21]，CSDL[22]，nML[23][35]， UPFAST[24]，BUILDABONG[25]，LISA[26]，RADL[27]，ArchC[28]，EXPRESSION[29]，PRMDL[30]，HMDES[31]，Maril[32]，TIE[33]，BABEL[34]。

然而这些 ADL 中没有一个是完美的，它们有些侧重于自动生成可综合硬件的代码，有的侧重于自动生成仿真器，有些侧重于自动生成编译器，有些侧重于二进制工具链的产生。这其中比较出色的是 LISA 和 EXPRESSION。前者已经被产品化，可用于自动高级语言编译器，汇编器，连接器，仿真器和调试器前端[39]。后者能产生周期精确的仿真器和指令级并行（Instruction Level Parallel，ILP）优化的编译器。

现在很多开展的工作是围绕完善以上提到的这些 ADL，但是由于这些 ADL 形

式上的不同，描述的对象以及包含的信息的不同，对不同 ADL 所做的工作不能用在其他的 ADL 上，这样就导致了很多的努力结果不能被共享。这本质上是由于这些 ADL 是基于不同抽象层的，或者说它们的抽象层定义不明确导致的。为此，普林斯顿大学的秦巍博士设计了 MADL[40]这一用于 ASIP 建模和软件工具链自动生成的 ADL。

MADL 建立在一个良好设计的模型 OSM（Operation State Machine，操作状态机）基础之上，该模型本身内在的特性决定了它可以很容易地支持流水线结构，解决流水线冲突，实现重定向，甚至支持其他所有 ADL 均不支持的多处理器建模[41][42]。

1.2 课题研究目的

工具链的手工移植代价很大，所需信息繁多，工具链的自动生成将使许多的电子系统开发人员从中解脱出来。目前的 MADL 仅能支持工具链中指令级仿真器和周期精确级仿真器的自动生成，然而它良好的模型设计，清晰的层次划分（Operation Level and Hardware Level，操作层和硬件层）及层次间通信方式，使得它的结构易于扩展。为了能够支持生成除指令级仿真器和周期精确级仿真器外的其他软件开发工具链中的工具，需要对软件工具链中每一个工具增加为支持该种工具自动产生所需的信息，由此提出了 UADL 项目。UADL 全称为统一结构描述语言，是一种新型的描述语言，该语言能描述处理器以及开发工具链支持处理器所需的信息等。通过 UADL 和相关工具，可进行处理芯片设计空间探索（DSE，Design Space Exploration），对软件、硬件、网络、操作系统、应用程序等多维综合优化。UADL 的建立基于一种新的描述模型 UOSM，即 Unified Operation State Machine。UOSM 模型通过对 OSM 模型进行扩展，使之具备以上提到信息的描述及抽取功能，提供软件工具链中工具的自动生成所需信息，简化工具链自动生成工具的实现工作，进而支持软件工具链的产生，从而使得 UADL 成为一个具备全流程工具链自动生成的优秀的 ADL。

本文的研究是 UADL 项目的一部分，力图通过对 LLVM[43]编译系统进行结构分析及后端移植来为 UADL 项目关于编译器的自动生成提供一个可行的参考方案，对 UADL 语言的及 UOSM 模型的进一步研究打下一个良好的基础，帮助最终实现基于 UADL 的软件开发工具链自动生成。LLVM 是近来出现的一个优秀的编译系统项目，是美国伊利诺斯大学开展的一个开放源代码项目。该项目在重用了

GCC 的前端高级语言处理的同时，采用了自创的代码优化机制，针对 GCC 的不足做了大量改进，尤其使整个程序的全局优化成为可能。其生成代码的编译时间明显少于 GCC[46]，并有着更清晰的架构层次和更详细的文档。如今正逐渐被越来越多的科研机构和商业项目所使用。这当中几乎都是国外的项目，如 SSAPRE 项目[44]，国内可见的唯一应用 LLVM 的项目是国防科学技术大学计算机学院的一个项目[45]。该项目为国家“863”计划软件重大专项基金资助项目“服务器操作系统内核”的一部分，在 LLVM 编译系统下实现了文中提出一个扩展的流不敏感指针分析算法，该算法主要用于缓冲区溢出静态分析，通过把程序控制流图（CFG，Control Flow Graph）转换为静态单指派（SSA，Static Single Assignment），再循环调用一个流不敏感指针分析，生成每个指针变量精确的指向集，更新指针变量的定义－引用链中约束信息。这两个项目均不是对 LLVM 进行后端移植的项目，国外除 LLVM 项目组本身开发的处理器后端外尚未见其它成功的后端移植项目，而国内目前关于 LLVM 后端移植的研究暂时空白。

在本文中，将介绍 LLVM 编译系统的组成，深入研究其移植机制，并给出对处理器移植的实例。

1.3 本文所作的工作

本文先分析 LLVM 编译系统的结构，然后深入探讨了 LLVM 的后端移植机制，并采用 2006 年 4 月发布的 LLVM 1.7 版本，对被广泛使用的 ARM 处理器后端目标是进行了移植。移植工作最终得到了支持 ARM 后端的 LLVM 编译器及其他相关工具。全文的安排如下：

第二章，介绍 LLVM 编译系统的各组成部分，重点分析其专有的虚拟指令集、集成库、系统工具集中各工具的功能和相互关系，并说明了它的编译流程；

第三章，详细剖析 LLVM 移植接口的各个方面，包括寄存器描述、指令集描述、帧栈布局、中间代码转换、汇编输出、处理器子系列支持、处理器 JIT 支持、全局描述等；

第四章，给出了移植 ARM 后端的实例，该章首先分析了 ARM 处理器的体系结构特征，然后给出了移植工作的具体过程；

第五章，该章总结了全文的工作，提出了后续可进行的工作以及对今后的展望。

2 LLVM 编译系统结构分析

LLVM 项目致力于程序整个生命周期，包括编译过程，链接过程，执行过程等阶段的优化，并提供大量用于建立编译器的可重用模块，可帮助减少开发新编译器的工作量。目前编译前端已经支持 C、C++等高级编程语言，后端则支持生成 C 语言代码以及 X86、Sparc、PowerPC 等处理器的汇编代码。

整个 LLVM 编译系统的组成可分成三部分：LLVM 中间代码[47]，用于分析、优化、代码生成等工作的集成库，以及建立在以上集成库基础之上的工具，包括汇编器、链接器、调试器等等。本章将先依次分析这三个组成部分，然后再对 LLVM 的编译流程作详细的说明。

2.1 LLVM 中间代码

LLVM 中间代码是一种采用 SSA 形式的 IR（Immediate Representation，中间表达），使用的指令集为 LLVM 虚拟指令集。该指令集是一个类似 RISC（Reduced Instruction Set Computer，精简指令集计算机）的三地址指令集，含有简单的控制指令和使用带类型指针的访存指令，拥有独立于高级语言和目标处理器的语法，易于进行代码分析和优化。使用 LLVM 虚拟指令集的 LLVM 中间代码可以三种方式存在：存在于内存中的编译器 IR、存在磁盘上的字节码（bytecode）、以及可供人阅读的汇编代码。这三种存在方式是同一中间代码的不同表达形式，可分别用于：在编译器执行编译遍时提供方便高效的中间代码转换或者中间代码分析、作为 JIT 编译器（Just-In-Time Compiler）在本地执行客户处理器代码的格式、以及方便开发人员进行代码调试的汇编代码。LLVM 中间代码由虚拟指令集、中间代码高层结构、以及中间代码类型系统三方面组成。

虚拟指令集可分为两大部分：单指令和内函数（Intrinsic Functions）。其中，单指令即具有一定功能的单一操作；而内函数在使用时以函数形式存在，编译时将被翻译成一条或若干条单指令。单指令包含以下几类：终结指令、算术运算指令、逻辑运算指令、向量运算指令、访存及地址操作指令、类型转换指令及其他指令。

其中终结指令是指每个基本块的最后一条指令，该条指令决定了当前基本块结束后将进入的下一个基本块。内函数则包含可变参数处理内函数、精确垃圾回收内函数、代码生成内函数、标准 C 库内函数、位操作内函数、以及调试内函数（详细的虚拟指令列表请见附录 1）。

构成 LLVM 中间代码的高层结构主要有以下这些：模块、链接类型、调用约定、全局变量、函数、模块级内嵌汇编。LLVM 模块是构成 LLVM 程序的基本元素，每一个模块都是从输入的程序翻译得到，由函数、全局变量、符号表入口组成。全局变量用于定义在编译时就分配的内存空间，可对它做初始化，并可放置在指定的特定段显示指定对齐。LLVM 函数描述组成一个函数的各个方面，如返回类型、函数名、参数类表、一个包含一系列基本块的函数体，以及可选的链接类型、调用约定、对齐等。每个 LLVM 函数定义时均需使用关键字‘ declare ’，两个函数可以同名，但是此时必须有不同的参数列表。对对上述的全局变量和函数均需指定链接类型，而对每个 LLVM 函数以及 call 和 invoke 指令，均可以可选地指定一个调用约定(可用的链接类型及调用约定请见附录 2)。类似于 GCC 的文件内嵌汇编，在 LLVM 中，模块可以内嵌模块级汇编。语句“ module asm "xxx yyy zzz" ”即为内嵌模块级汇编的语法格式，其中"xxx yyy zzz"为需内嵌的汇编代码。

普通的无类型的三地址中间表达必须在转换之后才能进行优化，而 LLVM 提供了一个较完备的类型系统，这使得在进行任何转换之前就可以在 LLVM 中间代码之上可以进行许多优化。LLVM 类型系统由两大部分组成：原子类型以及衍生类型。

原子类型是建立 LLVM 类型系统的基础类型，目前已有的类型可见下表：

表 1 LLVM 原子类型

Tab. 1 LLVM atomic types

类型	描述	类型	描述
void	空值	bool	布尔类型
ubyte	无符号 8 位数	sbyte	有符号 8 位数
ushort	无符号 16 位数	short	有符号 16 位数
uint	无符号 32 位数	int	有符号 32 位数
ulong	无符号 64 位数	long	有符号 64 位数
float	32 位浮点数	double	64 位浮点数
label	跳转的目标		

衍生类型使用以上的原子类型扩展 LLVM 类型系统可以描述的类型。这些衍生类型有数组类型、函数类型、结构类型、打包的结构类型、指针类型、打包类型等(详见附录 3)。

2.2 集成库

LLVM 提供了许多的集成库实现了编译器开发过程中需要的大部分基本功能，如描述程序基本块、函数、模块的类，程序编译过程中的编译遍添加管理，数据流分析，参数类型转换 (Arg Promotion)，调试，性能估计，以及其他各种优化。

这些集成库可以分为：核心库、分析库、转换库、代码生成库、目标处理器库、运行库。其中核心库主要用于中间代码的解析读写及相关处理、分析库提供各种数据分析及控制分析、转换库提供各种优化转换、代码生成库用于目标代码生成、目标处理器库提供已支持的后端描述、运行库提供解释执行机制及虚拟机器引擎（详细列表请见附录 4）。

2.3 工具集

LLVM 提供了大量现成的工具，方便以此为基础进行的后续开发工作。从总体上来说，这些工具可大致分为基本工具、编译工具、调试工具、后端工具四大类。在下图中列出了这些工具并显示了它们之间的相互关系。

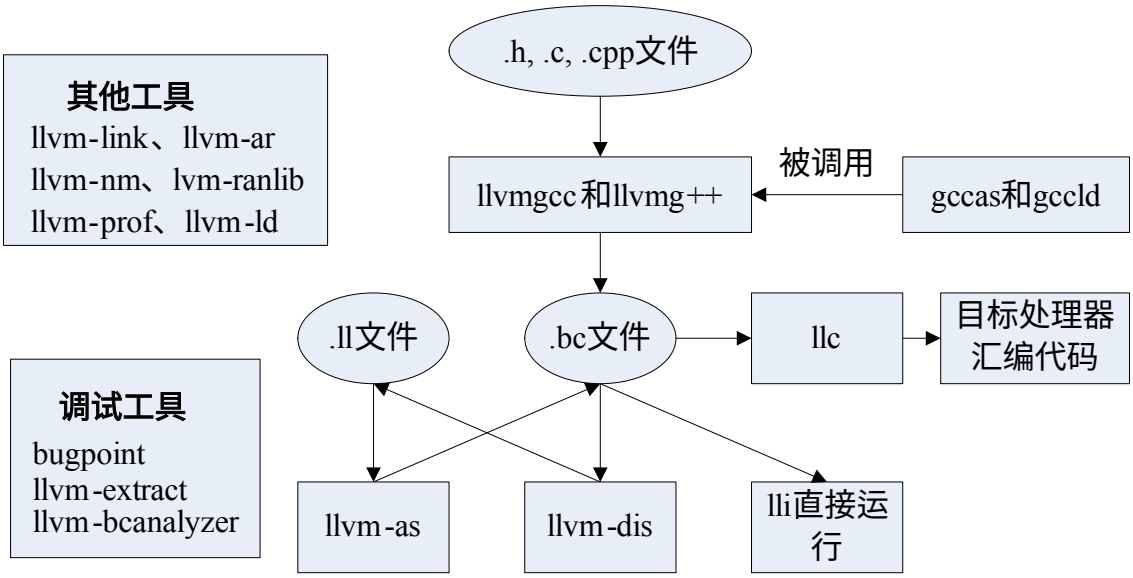

图 4 LLVM 工具及相互关系

Fig. 4 LLVM tools and relations between them

LLVM 提供了类似 GNU 工具链的全套基本工具，包括 llvm-as、llvm-dis、llvm-link、llvm-ar、llvm-nm、llvm-ranlib、llvm-prof、llvm-ld 等，区别在于 LLVM

定义有自己的汇编文件类型.ll 文件和二进制目标文件类型.bc 文件，并可用工具 llvm2cpp 将.bc 文件转成调用 LLVM C++ API 的 C++源文件。

编译工具包括基于 GCC 的 C 及 C++前端 llvmgcc 和 llvmg++，两者分别类似于 GCC 的 gcc 和 g++，而 gccas 和 gccld 则是被前两个工具分别用于编译和链接阶段的优化器，见图 1。工具 opt 可方便用户在.bc 文件的基础上逐遍进行编译遍转换或优化。

调试工具中 bugpoint 用于定位 LLVM 工具或编译遍的错误，llvm-extract 可从一个大型程序中剥离出指定的函数，llvm-bcanalyzer 则用于检查.bc 文件的编码。

后端工具中最重要的工具是 llc， lli 和 tblgen。llc 用于从.bc 文件生成目标处理器的汇编代码；而对支持 JIT 编译器（LLVM Just-In-Time compiler，LLVM 即时编译器）的目标处理器，lli 可直接在本地运行目标处理器的.bc 代码。tblgen 用于将目标处理器的描述转化为相应的描述源代码文件，由此来简化后端目标处理器的移植工作，它要求用 LLVM 定义的.td 格式文件来描述，是 LLVM 后端移植机制的重要组成部分。

2.4 LLVM 编译流程

LLVM 的编译流程可以分为三大部分：高级语言前端、中间代码优化器、后端代码生成器。高级语言前端将使用高级语言编写的代码转换到 LLVM 中间代码，相对高级语言前端和目标处理器后端均独立的中间代码优化器则对转换得到的 LLVM 中间代码作优化，经过优化后的中间代码通过后端代码生成器生成针对目标处理器的机器代码。大致流程可见下面图 2：

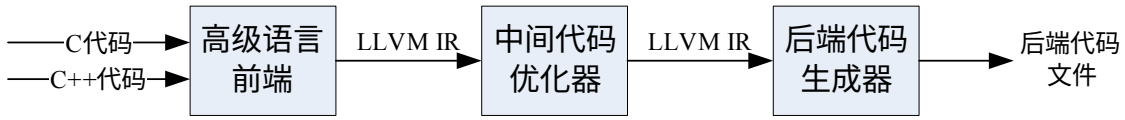

图 5 LLVM 编译流程结构图

Fig. 5 Diagram of LLVM compiling flow

LLVM 的编译前端结构可见下面图 3。LLVM 采用 GCC 的高级语言前端来解析代码，现已支持 C、C++、FORTRAN、Ada、Java 等高级语言，并且可以通过前端的移植接口添加对新的高级语言的支持。又由于该高级语言前端完全独立于之后的中间代码生成器以及后端代码生成器，对以后两个两阶段的任何改进和优化可以使得所有的高级语言前端获益，这大大提高了模块的复用程度，减少不必要的重

复工作。

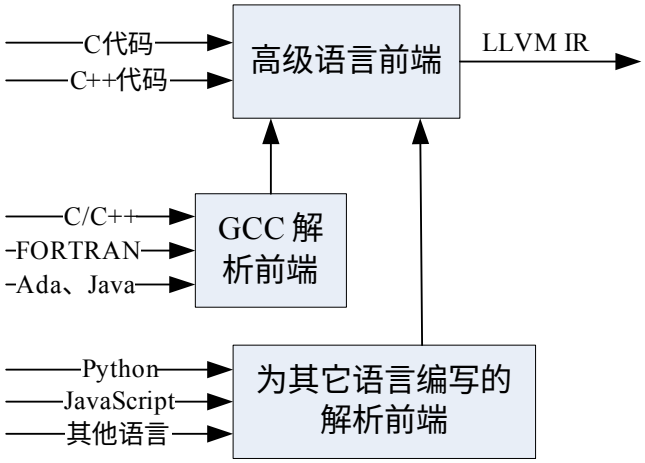

图 6高级语言前端结构图

Fig. 6 Diagram of high-level language front-end

LLVM 的中间代码优化器是建立在前文已述的 LLVM 虚拟指令集基础之上的，它同样独立于其他的两个步骤。在这个阶段，中间代码优化器将运用标准的标量和循环优化、以及 IPO（Interprocedural Optimizations，进程间优化）等。其结构可见下面图 4：

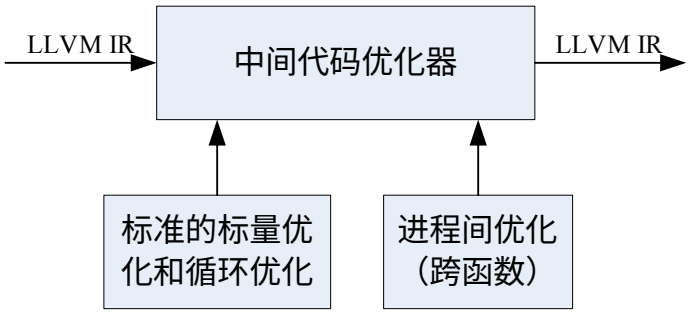

图 7中间代码优化器结构图

Fig. 7 Diagram of intermediate representation optimizer

后端代码生成器主要由以下部分组成：指令选择、遍前调度、寄存器分配、后期机器代码优化、代码输出。其中在指令选择前使用 LLVM 中间代码，之后均使用目标处理器的特定代码。下面图 5 给出了代码生成器的结构图。

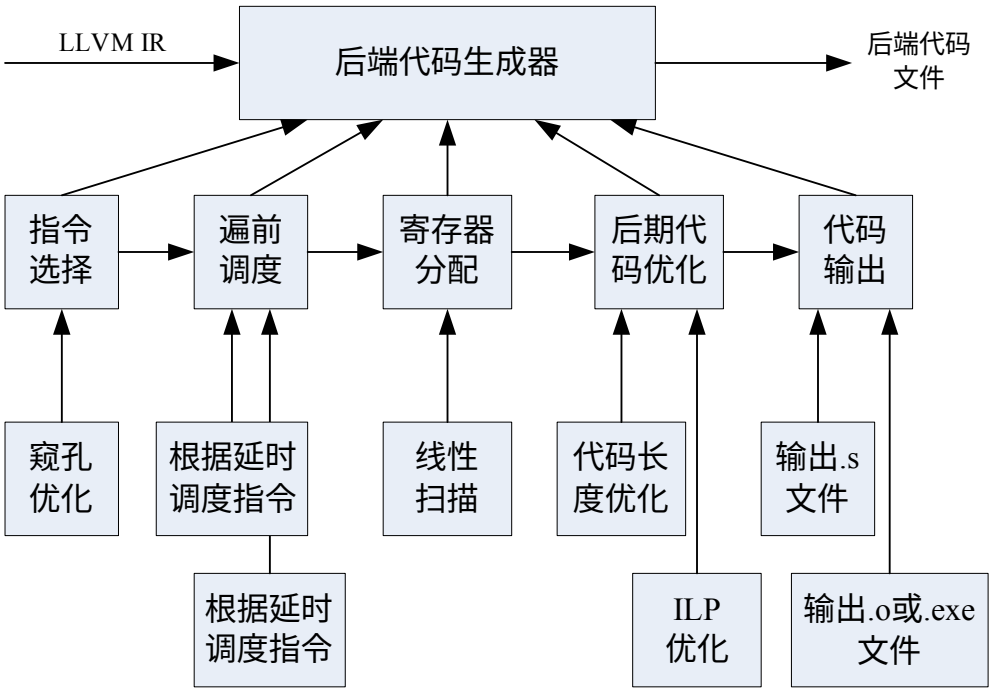

图 8代码生成器的结构图

Fig. 8 Diagram of coder generator

下面简要介绍代码生成器的各个组成部分：

指令选择，就是将输入给后端代码生成器的 LLVM 中间代码翻译成目标处理器的特定机器指令的过程，过程中可进行窥孔优化等工作。

遍前调度，就是根据目标处理器指令在执行时占用处理器功能单元的资源使用情况对程序的指令序列进行重新安排；同时根据程序使用寄存器的情况对指令序列进行重新安排。

寄存器分配，将原来使用的没有个数限制的虚拟寄存器映射到真实的目标处理器寄存器，由于真实寄存器的个数限制，不能放下的源虚拟寄存器中的值将被迫放入内存中。在这个过程中，LLVM 采用线性扫描（linear scan）作为默认的寄存器分配算法。

后期代码优化，进行代码长度优化、ILP（Instruction-Level Parallelism，指令级并行）优化等一系列优化工作。

代码输出，根据要求输出目标处理器的汇编代码、目标文件、或者可执行文件。

2.5 本章小节

本章对 LLVM 编译系统的作了总体介绍，深入分析了 LLVM 中间代码的虚拟指令集及其语法系统、LLVM 各个集成库的功能、各个 LLVM 工具的功能及它们之间的相互关系，最后详细解释整个 LLVM 编译系统编译流程的各个步骤。

3 LLVM 后端移植接口

对经过前端翻译后生成的 LLVM 中间代码，通过后端代码生成器可以生成对特定后端处理器的后端代码。生成的后端代码可以两种形式存在：一种是以目标处理器的汇编代码形式，可以通过汇编器编译后得到相应的目标处理器二进制代码，并能运行在目标处理器上；另一种是直接以二进制代码存在，不能运行在目标处理器上，但可以使用 JIT 编译器直接在本地运行。

LLVM 将后端代码生成器分成相对独立的两大部分：目标处理器独立的代码生成器以及后端移植接口。目标处理器独立的代码生成器提供了实现一个后端代码生成器的所有基本功能，但不依赖于任何后端处理器的代码生成器，因此不能直接生成针对某种处理器的后端代码。后端移植接口是后端代码生成器提供的移植接口，对任何一个新增加的目标处理器，需要按照移植接口的要求提供目标处理器的特征描述。对不同目标处理器的后端代码生成器来说，生成代码的过程是相同的，不同的只是过程中对不同的目标处理器需要根据其相应的体系结构特征进行不同的处理。用目标处理器独立的代码生成器实现后端代码生成器的代码生成过程，并根据后端移植接口提供的目标处理器的特定体系结构特征来进行过程中不同的处理，这种划分大大减少了开发后端代码生成器的代价，使得对每一个目标处理器的后端代码生成器的实现工作减少为实现该目标处理器的后端移植接口。

本章首先对 LLVM 后端移植接口的总体情况进行说明，再介绍 LLVM 提供用于简化后端移植的 TableGen，最后根据一般移植的步骤深入分析接口中的各个部分。以下均假设所要实现的目标处理器的名称为 XXX。

3.1 移植接口综述

LLVM 后端移植接口由一系列的目标处理器描述组成，它们被封装成一个个抽象类，这些抽象类描述了目标处理器各个方面的属性。对每一个特定的处理器，首先继承这些类，然后根据处理器的体系结构特征实现相关的虚函数，就可以获得对该处理器的后端支持。这些抽象类包括：TargetMachine、TargetData、TargetLowering、

MRegisterInfo、TargetInstrInfo、TargetFrameInfo、TargetSubtarget、TargetJITInfo 等。它们分别描述了目标处理器的各个方面：全局情况、中间代码转换、寄存器、指令集、帧栈布局、处理器子系列支持、处理器 JIT 支持等。

同时，LLVM 提供 TableGen 以减少后端描述的工作量，将大量简单但工作量大的后端描述用符合 TableGen 语法的.td 格式文件描述，并用工具 tblgen 解析后生成这些描述的 C++代码。因此，移植工作就可分为使用 TableGen 描述目标处理器，以及编写 C++代码补充描述 TableGen 不能描述的目标处理器体系结构特性两大方面。下面给出的图 2 给出了 LLVM 后端移植的结构图。

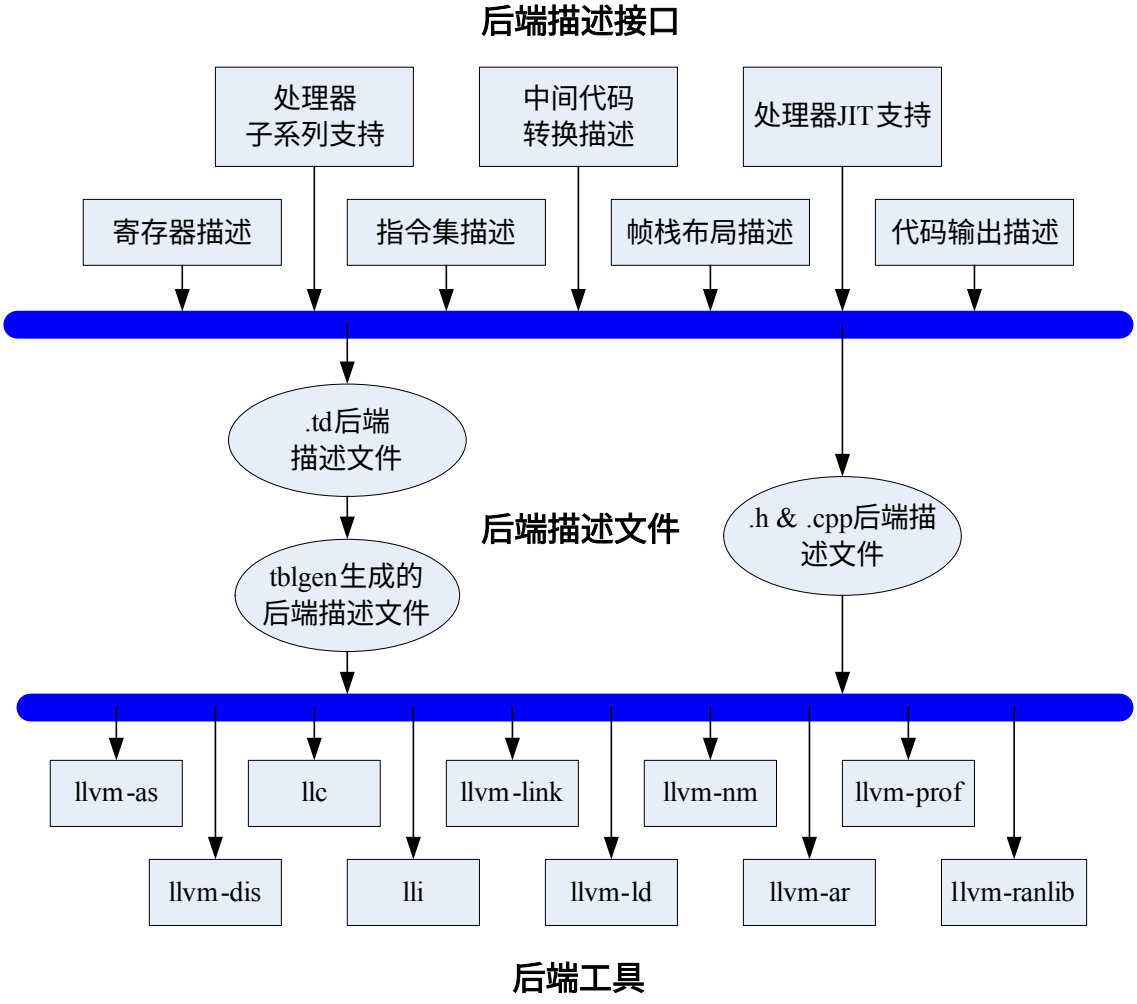

图 9 LLVM 后端移植结构

Fig. 9 Diagram of LLVM backend porting

3.2 TableGen 简介

在为后端代码生成器添加对一个新的目标处理器的支持时，一般使用 TableGen 来描述该目标处理器的抽象属性，tblgen 解析后将生成大部分移植所需的目标处理器抽象属性描述。

TableGen 本身用于帮助开发和维护某一领域相关的信息记录，它是一种描述信息记录的一套数据结构和语法规范。对某些应用来说，可能有大量的记录要开发或者维护，而这些记录之间有许多部分是相同的。如果手工编写这些记录会花费大量的精力，每次对类似记录的小改动也会占用大量的时间，并且很容易产生错误。TableGen 就是设计来简化这些记录的编写的，它允许用灵活的描述来减少记录间公共部分的编写，能更方便地组织记录的数据结构。对用 TableGen 描述的信息记录，使用工具 tblgen 可以对该描述作解析并生成出完整的信息记录，这样就可以大大减少对信息记录编写的工作量，并减少手工编写的出错可能。

TableGen 描述文件是由许多的记录（record）组成的。每一个记录都有一个唯一的名字，包含一系列的数据，这些数据就是 TableGen 文件所描述的领域所需的信息。使用 tblgen 工具解析文件时，指定这些数据所针对的不同领域，就可以解析出该领域所需的记录信息。

记录可以分为两种，一种称为定义（definition），一种成为类（class）。定义是最基本的记录类型，它的定义中没有不确定的成分，并且使用关键字‘ def ’标记。类是一种抽象的记录，帮助建立和描述其他的记录。使用类可以用来描述特定领域所使用数据的抽象数据结构，也可以用来减少对记录中数据公共部分的编写。对所有的类，TableGen 都会记下它们的结构，并在所有使用类来建立的记录中实现这些结构。

例如，“ def A { bit c = 0; } ”就是一个定义，它建立了一个名为 A 的定义，它只有一个数据，类型为位，值为 0。“ class B{ bit d = 0; } ”就是一个类，并且所有建立时使用类 B 的记录都将包含类型为位的数据 d。“ def E : B { bits<3> num = 0b101; } ”建立了一个定义 E，E 包含两个数据 d 和 num，d 为位类型且值为 0，num 为位串类型且值为 0b101。

通过使用‘ let ’关键字，可以改变建立记录时所使用的类中数据的默认值。例如，“ def F : B { let d = 1; bits<3> num = 0b101; } ”建立的记录 F 中的数据 d 的值为 1。还可以用‘ let ’来指定多个记录中的同样的数据的值，比如：

```
let isStore = 1, noResults = 1 in {
```

```
    def STRimmOffAddW : STW_imm<
                        ( ops GPRC:$rS, memri:$src ),
                        "str $rS, $src",
                        [(store GPRC:$rS, iaddr:$src)]
                    >;
    def STRimmOffAddH : STW_imm<
                        ( ops GPRC:$rS, memri:$src ),
                        "strh $rS, $src",
                        [(truncstore GPRC:$rS, iaddr:$src, i16)]
                    >;
    def STRimmOffAddB : STW_imm<
                        ( ops GPRC:$rS, memri:$src ),
                        "strb $rS, $src",
                        [(truncstore GPRC:$rS, iaddr:$src, i8)]
                    >;
}
```

这里建立了三个记录 STRimmOffAddW、STRimmOffAddH、STRimmOffAddB，分别用来描述字、半字及字节的立即数寻址模式的存储指令，并设置 isStore 及 noResults 均为 1 来表明三种指令都是存储指令且指令没有返回结果。

另外，TableGen 还允许建立类时使用模板参数。如“ class addrMode < bits<2> mode > { bits<2> addressMode = mode; } ”建立了一个名为 addrMode 的类，接受长度为 2 的位串作为参数。以此可方便地定义以下寻址模式：

```
def addrImm : addrMode<0>;
def addrIdx : addrMode<1>;
def addrIdxOnly : addrMode<2>;
def addrImmshift : addrMode<3>;
```

在定义记录时，TableGen 规定了以下可以使用的基本类型：

- "bit" – 位，值为布尔类型，只可以取 0 或者 1
- "int" – 整型，可以表达 32 位的整型数据
- "string" – 字符串，可以表达任意长度的字符序列

· "bits<n>" – 位串，可以表达长度固定为 n 的位类型的数据序列

· "list<ty>" – 列表，可以表达一个组成元素为 ty 类型的列表，ty 可以是任意类型，也可以本身是一个列表

· Class type – 类，在和列表类型一起使用时，用于指出列表中组成元素的类型为该类的子类

· "code" – 代码段，可以表达一段代码

· "dag" – dag 类型，可以表达有向连接图中一个组成元素

对上述的基本类型，目前支持的值和表达式有以下几种：

· ? – 未初始化的

· 0b1001011 – 二进制整数值（以 0b 起始）

· 07654321 – 八进制整数值（以 0 起始）

· 7 – 十进制整数值

· 0x7F – 十六进制整数值（以 0x 起始）

· "foo" – 字符串值

· [{ ... }] – 代码段（其中…可为各种代码）

· [X, Y, Z] – 列表值

· { a, b, c } – 对长度为 3 的位串的初始化值

· value – 对 value 的引用

· value{17} – 对位串中一个位的引用，引用 value 中标号为 17 的位

· value{15-17} – 对位串中几个位的引用，引用 value 中标号为 15 到 17 的位

· DEF – 对一个记录定义的引用

· CLASS<val list> – 引用一个指定模板参数的匿名类

· X.Y – 对一个值的子域的引用，引用 X 的子域 Y

· list[4-7,17,2-3] – 对列表中一个片断的引用，包括标号为 4、5、6、7、17、2、3 的元素

· (DEF a, b) – 一个 dag 类型的值，第一个元素必须是一个记录定义，剩下的可以是任意的值，包括 dag 类型的值

使用 TableGen 进行的描述一般存入以‘ td ’(target description，目标描述）为后缀名的文件中。在需要将描述分别放在数个文件中时，可以使用关键字‘ include ’在一个文件中包含另一个文件中的内容。例如，在编写描述指令集的文件

‘ ARMInstrInfo.td ’ 时，可以使用“ include “ ARMInstrFormats.td ”” 引入已在文件‘ ARMInstrFormats.td ’ 中建立的描述处理器指令格式的类。而在文件中需要使用注释时，可以使用类似于 C++语言中的注释标记“ // ”和“ /* */ ”。

对编写好的描述文件，使用工具 tblgen 就可以解析得到相应的领域的记录文件。例如，使用下面的命令可以分别生成描述处理器寄存器的三种文件：

tblgen ARM.td -gen-register-enums -o ARMGenRegisterNames.inc

tblgen ARM.td -gen-register-desc -o ARMGenRegisterInfo.inc

tblgen ARM.td -gen-register-desc-header -o ARMGenRegisterInfo.h.inc

3.3 寄存器描述

主要从两方面来描述目标处理器的寄存器，实现关于目标处理器寄存器的 TableGen 描述以及继承并实现 MRegisterInfo 类，也即需要实现文件“ XXXRegisterInfo.td ”、“ XXXRegisterInfo.h ”和“ XXXRegisterInfo.cpp ”。

在文件“ XXXRegisterInfo.td ”中，需要描述目标处理器每个寄存器的属性，寄存器之间的别名关系以及程序运行时的寄存器分配方案等，这通过 TableGen 提供的描述寄存器的四个记录类实现：Register、RegisterGroup、RegisterClass、DwarfRegNum。

Register 类用于描述每个寄存器的属性，该类包括的数据有：

string Namespace　　指定所属的命名空间

string Name　　寄存器的名字

int SpillSize　　寄存器溢出时保存寄存器所需的位的个数

int SpillAlignment　　寄存器溢出时保存寄存器要求的对齐

list<Register> Aliases　　读/写本寄存器将读/写的其它寄存器的列表（在后文中将简称为“ 别名 ”）

int DwarfNumber　　gcc/gdb 内部用于识别寄存器的编号

例如，下面建立了一个名为 GPR 的寄存器类，属于命名空间“ XXX ”，有一个长度为 2 的位串类型数据 num.。用 GPR 可以建立 4 个记录定义 R0～R3，描述了 4 个属于命名空间“ XXX ”的寄存器。

```
class GPR< bits<2> num, string n > : Register<n>{
    field bits<2> Num;
```

```
    let Namespace = "XXX";
    Num = num;
}
def R0 : GPR< 0, "R0">;
def R1 : GPR< 1, "R1">;
def R2 : GPR< 2, "R2">;
def R3 : GPR< 3, "R3">;
```

RegisterGroup 类继承了 Register 类，用于描述有别名的寄存器，其结构如下：

```
class RegisterGroup<string n, list<Register> aliases> : Register<n>{
    let Aliases = aliases;
}
```

例如，“ def S0 : RegisterGroup<"S0", [R0]>; ”说明了 S0 与 R0 之间的别名关系。

在用 Register 类定义寄存器，并用 RegisterGroup 类指出所有的别名情况后，RegisterClass 类帮助对寄存器进行分类，将不同功能的寄存器分为不同的寄存器类，并规定不同类型的寄存器在进行寄存器分配时的分配顺序。该类包括的数据有：

string Namespace = namespace　　指定所属的命名空间

list<ValueType> RegTypes　　指令该类寄存器类型

int Size　　寄存器溢出时保存寄存器所需的位的个数

int Alignment　　当寄存器被写入内存时需要的对齐

list<Register> MemberList　　列出所有属于本寄存器类的寄存器，并作为没有提供寄存器分配方案 allocation_order_*时默认的寄存器分配顺序

code MethodProtos　　提供寄存器分配方案 allocation_order_*的函数原型

code MethodBodies　　实现寄存器分配方案 allocation_order_*函数

以下给出了一个寄存器类的记录定义：

```
def GPRC : RegisterClass< "XXX", [i32], 32, [R0, R1, R2, R3] >{
    let MethodProtos = [{
        iterator allocation_order_begin(MachineFunction &MF) const;
        iterator allocation_order_end(MachineFunction &MF) const;
```

```
    }];
    let MethodBodies = [{
        GPRCClass::iterator GPRCClass::allocation_order_begin(
                                    MachineFunction &MF) const
        { return begin() + 1; }
        GPRCClass::iterator GPRCClass::allocation_order_end(
                                    MachineFunction &MF) const
        { return end() - 1; }
    }];
}
```

上面定义的寄存器类 GPRC 包含 4 个寄存器 R0、R1、R2、R3，该寄存器类中的寄存器的类型为 i32，对齐为 32，所属命名空间的名字为“ XXX ”，并且在 allocation_order_begin 和 allocation_order_end 的函数实现中分别规定 R0 和 R3 不参与寄存器分配。

DwarfRegNum 类提供了 llvm 寄存器与 gcc/gdb 寄存器编号之间的映射，其结构如下：

```
class DwarfRegNum<int N>{
    int DwarfNumber = N;
}
```

例如，“ def R0 : GPR< 0, "R0">, DwarfRegNum<0>; ”将 R0 映射到寄存器编号 0。

对需移植的目标处理器，通过以上这四个记录类来描述寄存器，就可完成文件“ XXXRegisterInfo.td ”的实现。

在文件“ XXXRegisterInfo.h ”和“ XXXRegisterInfo.cpp ”中，则需继承 MRegisterInfo 类，并实现 MRegisterInfo 类中的虚函数。这些虚函数接口主要有：

提供将寄存器中的值放入栈槽的目标处理器指令

```
virtual void storeRegToStackSlot(MachineBasicBlock &MBB,
                MachineBasicBlock::iterator MI, unsigned SrcReg,
                int FrameIndex, const TargetRegisterClass *RC) const = 0;
```

提供从栈槽取出值放入寄存器中的目标处理器指令

```
virtual void loadRegFromStackSlot(MachineBasicBlock &MBB,
                MachineBasicBlock::iterator MI, unsigned DestReg,
                int FrameIndex, const TargetRegisterClass *RC) const = 0;
```

提供寄存器拷贝的目标处理器指令

```
virtual void copyRegToReg(MachineBasicBlock &MBB,
                MachineBasicBlock::iterator MI, unsigned DestReg,
                unsigned SrcReg, const TargetRegisterClass *RC) const = 0;
```

将对栈槽进行读写的中间代码转为目标处理器的访存指令

```
virtual MachineInstr* foldMemoryOperand(MachineInstr* MI,
                unsigned OpNum, int FrameIndex) const;
```

在进行序言和尾声代码插入时，调用这个接口函数去除建立和销毁帧的伪代码

```
virtual void eliminateCallFramePseudoInstr(MachineFunction &MF,
                MachineBasicBlock &MBB, MachineBasicBlock::iterator MI) cons;
```

实现本函数接口以在函数帧布局结束之前增加其它处理

```
virtual void processFunctionBeforeFrameFinalized(MachineFunction &MF) const;
```

实现本函数接口以消除 LLVM 的虚拟帧索引

```
virtual void eliminateFrameIndex(MachineBasicBlock::iterator MI) const;
```

添加进入函数之前运行的序言代码

```
virtual void emitPrologue(MachineFunction &MF) const = 0;
```

添加退出函数之前运行的尾声代码

```
virtual void emitEpilogue(MachineFunction &MF,
                MachineBasicBlock &MBB) const = 0;
```

将目标处理器的寄存器映射到 dwarf 的寄存器编号

```
virtual int getDwarfRegNum(unsigned RegNum) const = 0;
```

返回当前使用的帧指针所在的寄存器

```
virtual unsigned getFrameRegister(MachineFunction &MF) const = 0;
```

返回目标处理器的链接寄存器

```
virtual unsigned getRARegister() const = 0;
```

对给定的帧索引偏移 index 返回实际位置 ML

```
virtual void getLocation(MachineFunction &MF, unsigned Index,
                         MachineLocation &ML) const;
```

返回出现在所有函数入口的目标处理器移动指令

```
virtual void getInitialFrameState(std::vector<MachineMove *> &Moves) const;
```

3.4 指令集描述

指令集描述也同样分为两方面，实现关于目标处理器指令集的 TableGen 描述以及继承并实现 TargetInstrInfo 类，也即需要实现文件“ XXXInstrInfo.td ”、“ XXXInstrInfo.h ”和“ XXXInstrInfo.cpp ”。

在文件“ XXXInstrInfo.td ”中，需要描述目标处理器的指令集，指令功能，指令的寻址方式,指令操作数,指令编码,指令汇编代码的输出格式,以及指令与 LLVM 虚拟指令集的匹配关系等，这些描述可以由以下几个 TableGen 提供的记录类实现：

Operand 类用于描述指令操作数，包含的数据有：

ValueType Type	操作数的数据类型
string PrintMethod	打印输出操作数的函数的名字
int NumMIOperands	操作数对应的目标处理器的操作数的个数
dag MIOperandInfo	操作数对应的目标处理器的操作数的信息

记录定义 ops，用于标识指令的操作数列表。例如：“ ops GPRC:$dst, GPRC:$src ”，表明指令有两个操作数 dst 和 src，且都是 GPRC 类型的。

记录定义 variable_ops，用于标识指令有不定数量的操作数。

Predicate 类用于描述指令进行匹配选择时所需的特定条件，Requires 类为这些特定条件的组合，它是一个以 Predicate 类为元素的列表。这两个记录类的结构如下：

```
class Predicate<string cond> {
    string CondString = cond;
}
class Requires<list<Predicate> preds> {
    list<Predicate> Predicates = preds;
}
```

FuncUnit 类用于建立目标处理器的功能单元，对每一个功能单元需指定一个值来代表该单元。这些功能单元将被视为可以分配使用的受限资源，即在一定时间内只能有一条指令在执行时占用该单元。例如，“ def ICU : FuncUnit; ”及“ def FCU : FuncUnit; ”分别建立了一个整数运算单元和一个浮点数运算单元。

InstrStage 类用于描述指令执行中的某一阶段，包括完成该阶段可以选择的功能单元列表以及完成所需的目标处理器运行周期，其结构如下：

```
class InstrStage<int cycles, list<FuncUnit> units> {
    // 完成该指令执行阶段所需的目标处理器周期
    int Cycles = cycles;
    // 完成该指令执行阶段可以使用的功能单元列表
    list<FuncUnit> Units = units;
}
```

每一条指令执行可能经历不同的阶段，使用不同的功能单元组合，需要不同的执行时间。InstrItinClass 类用于建立这些不同的指令执行类别。例如，“ def IntSimp : InstrItinClass; ”和“ def IntComp : InstrItinClass; ”建立了两个指令执行类别，可分别用于描述简单整型运算和复杂整型运算。

而 InstrItinData 类代表指令执行绑定，即对指令执行类别和指令执行阶段的绑定，其结构如下：

```
class InstrItinData<InstrItinClass Class, list<InstrStage> stages> {
    InstrItinClass TheClass = Class;          // 指令执行类别
    list<InstrStage> Stages = stages;         // 指令执行阶段列表
}
```

下面给出了一个建立绑定的例子，设定指令执行类别 IntComplex 需要两个执行周期，可用功能单元 IntComp1 或 IntComp2 完成：

```
InstrItinData<IntComplex, [InstrStage<2, [IntComp1, IntComp2]>]>,
```

ProcessorItineraries 类用于对目标处理器的所有指令执行绑定，其结构如下：

```
class ProcessorItineraries<list<InstrItinData> iid> {
    list<InstrItinData> IID = iid;
}
```

Instruction 类用于描述处理器指令，包含的数据有：

string Name	指定指令名
string Namespace	指定所属的命名空间
dag OperandList	指定指令的操作数列表
string AsmString	指定指令的汇编代码字符串
list<dag> Pattern	设定指令所匹配的 LLVM 虚拟指令模式
list<Register> Uses	设定将使用的非操作数寄存器
list<Register> Defs	设定将修改的非操作数寄存器
list<Predicate> Predicates	设定指令进行模式匹配时所需的特定条件列表
// 以下为描述指令高级语法信息的标志位	
bit isReturn	判断指令是否为返回指令的标志位
bit isBranch	判断指令是否为条件跳转指令的标志位
bit isBarrier	判断控制流能否穿过本指令的标志位
bit isCall	判断指令是否为函数调用指令的标志位
bit isLoad	判断指令是否为读内存指令的标志位
bit isStore	判断指令是否为写内存指令的标志位
bit isTwoAddress	判断指令是否为两地址指令的标志位
bit isConvertibleToThreeAddress	判断指令是否能转化为三地址指令的标志位
bit isCommutable	判断指令是否为可交换指令的标志位

bit isTerminator　　判断指令是否为某个基本块的结束部分的标志位
bit hasDelaySlot　　判断指令是否有延时槽的标志位
bit usesCustomDAGSchedInserter 判断指令是否为需客制处理的伪指令的标志位
bit hasCtrlDep　　判断指令是否读或写控制流的标志位
bit noResults　　判断指令是否产生结果的标志位
InstrItinClass Itinerary　　设定指令执行时的调度和执行阶段

InstrInfo 类用于提供目标处理器的全局参数，其结构如下：

```
class InstrInfo {
    // 当目标处理器需要给指令设定一些目标处理器相关的信息时，列表
    // TSFlagsFields 给出了这些信息的字符串列表，而列表 TSFlagsShifts
    // 给出了将这些字符串对应到可用的 32 个位的位置信息列表
    list<string>   TSFlagsFields = [];
    list<int>      TSFlagsShifts = [];
    // 指定目标处理器是否是小端编码的
    bit isLittleEndianEncoding = 0;
}
```

在文件“ XXXInstrInfo.h ”和“ XXXInstrInfo.cpp ”中，则需继承 TargetInstrInfo 类，并实现 TargetInstrInfo 类中的虚函数。下面列出部分重要的虚函数接口：

用于判断一条指令是否是寄存器间移动指令，并给出源寄存器和目标寄存器

```
virtual bool isMoveInstr(const MachineInstr& MI,
                         unsigned& sourceReg, unsigned& destReg) const;
```

用于判断一条指令是否是读栈槽指令，是的时候给出目标寄存器编号和要读取栈的槽的偏移量

```
virtual unsigned isLoadFromStackSlot(MachineInstr *MI, int &FrameIndex) const;
```

用于判断一条指令是否是写栈槽指令，是的时候给出源寄存器编号和要写入栈槽的偏移量

```
virtual unsigned isStoreToStackSlot(MachineInstr *MI, int &FrameIndex) const;
```

在允许将两地址指令转换为三地址指令的目标处理器上，本函数需提供相应的指令转换

virtual MachineInstr *convertToThreeAddress(MachineInstr *TA) const;

当目标处理器包含有可交换操作数的指令，且交换操作数需做一定转换时，重载本函数以提供相应的转换

virtual MachineInstr *commuteInstruction(MachineInstr *MI) const;

重载以提供目标处理器的空闲指令

virtual void insertNoop(MachineBasicBlock &MBB, MachineBasicBlock::iterator MI) const;

3.5 帧栈布局描述

栈的描述主要是通过继承和实现 TargetFrameInfo 类来实现，也即需要实现文件“XXXFrameInfo.h”和“XXXFrameInfo.cpp”。TargetFrameInfo 类包含以下三个私有数据来描述栈的布局：

StackDirection StackDir;

unsigned StackAlignment;

int LocalAreaOffset;

其中 StackDirection 类型的变量 StackDir 用来描述栈的生长方向，即向栈中添加数据是增长地址还是减少地址，分别称为升序栈和降序栈，StackDir 应设置为 StackGrowsUp 和 StackGrowsDown。变量 StackAlignment 用来设置目标处理器的栈的对齐大小，而变量 LocalAreaOffset 的值表明在函数入口时栈指针与当前函数可以存放本地临时变量的地址的偏移。

另外，在可用的寄存器分配完毕后，调用函数需保存的寄存器可能需要被置入栈中，此时如果目标处理器对某些寄存器置入栈中的位置有特定要求时，必须实现虚函数 getCalleeSaveSpillSlots()，其函数声明如下：

virtual const std::pair<unsigned, int> * getCalleeSaveSpillSlots(
unsigned &NumEntries) const;

该函数返回一个指向一个元素为 pair 的数组，数组中的每一个 pair 描述了一个调用函数需保存的寄存器的编号以及当它在被迫置入栈中时需存入的特定位置，

该位置通过位置与栈指针间的偏移来描述。

3.6 中间代码转换描述

中间代码转换是指将 LLVM 的中间代码根据目标处理器的情况转为相应的目标处理器的代码。中间代码转换描述可分为三方面：操作数合法化、指令匹配选择、以及其他转化描述。

LLVM 中间代码的类型系统与目标处理器的类型系统很可能不一致，对于目标处理器不支持的 LLVM 类型，需要将该类型转化为目标处理器支持的类型，该过程称为操作数的合法化。两种情况可能产生这种不一致：

1．目标处理器支持的最小类型比 LLVM 的类型要大，此时将该 LLVM 类型的数据提升（Promote）为目标处理器类型的数据以进行下一步工作

2．目标处理器支持的最大类型比 LLVM 的类型要小，此时将该 LLVM 类型的数据拆成数个目标处理器可以支持的类型的数据以进行下一步工作

该合法化主要通过继承并实现 TargetLowering 类中关于操作数合法化的部分达成。这些部分包括：

1．添加目标处理器支持的寄存器类，表明目标处理器能支持的数据类型

2．重载 TargetLowering 类中的函 LowerArguments，在该函数中将函数调用中的参数进行合法化

3．重载 TargetLowering 类中的函数 LowerCallTo，在该函数中对函数调用中的返回结果进行合法化

LLVM 中间代码使用的指令集为 LLVM 虚拟指令集，该指令集几乎必然会和目标处理器的指令不一致。这种不一致主要表现在两方面：

1．目标处理器不支持 LLVM 虚拟指令集的部分指令

2．目标处理器支持 LLVM 虚拟指令集中的某种指令，但支持的操作数类型不一致

3．目标处理器存在 LLVM 虚拟指令集中没有的指令

指令匹配选择则用于解决这种不一致，并且同样是通过继承并实现 TargetLowering 类实现。共有 4 种情况需要实现：

1．设定目标处理器支持的指令及该指令的操作数类型

2．设定目标处理器不支持的指令及该指令的操作数类型

3．对于目标处理器不直接支持，但可变相支持的指令进行扩展支持

4．添加 LLVM 虚拟指令集不支持的目标处理器指令

对于上述的 1、2、3，这三种情况的描述可用下面的语句实现：

setOperationAction(ISD::Operation, MVT::Type, LegalizeAction);

其中“ ISD::Operation ”为某个 LLVM 虚拟指令，“ MVT::Type ”为该指令的操作数的类型，而“ LegalizeAction ”则为对该指令进行匹配时的处理。对应之前解决不一致的步骤，共有 4 种处理方式：

Legal　　目标处理器完全支持这条指令

Promote　目标处理器的指令需在更大的数据类型上进行操作

Expand　　目标处理器不支持该条指令，应转为其他指令或使用其他工具库

Custom　　利用“ 重载函数 LowerOperation ”这个接口实现对该条虚拟指令的目标处理器实现

对于上述的情况 4，类似前述情况 3 的处理方法，将某条与目标处理器特有指令 B 相关的 LLVM 指令 A 的处理类型设定为 Custom，当发现当前的 A 指令的上下文环境使得 A 指令或 A 指令与其上下若干条指令可用目标处理器特有指令 B 替代时，在重载的函数 LowerOperation 中进行该替换操作。

除去操作数以及指令的转化描述之外，还有许多其他的转化描述：

设定用于移位指令的移位大小的类型，默认为目标处理器的指针类型

void setShiftAmountType(MVT::ValueType VT)

设定存放 setcc 指令结果数据的类型，默认为目标处理器的指针类型

void setSetCCResultType(MVT::ValueType VT)

设定如何将 setcc 指令的结果扩展放置到寄存器中，如“ 零扩展 ”或“ 符号扩展 ”

void setSetCCResultContents(SetCCResultValue Ty)

设定目标处理器进行调度的优先选项，如“ 最小延时 ”或“ 最小寄存器压力 ”

void setSchedulingPreference(SchedPreference Pref)

设定目标处理器如何处理对于超出范围的移位，如“扩展”或“未定义”
void setShiftAmountFlavor(OutOfRangeShiftAmount OORSA)

设定应该保存的栈指针寄存器
void setStackPointerRegisterToSaveRestore(unsigned R)

告诉代码生成器在目标处理器上进行类似 setcc 这样的条件跳转指令是否代价很大，是否有必要将该指令转化为其他的指令
void setSetCCIsExpensive()

告诉代码生成器在目标处理器上进行整数除法这样的指令是否代价很大，是否有必要将该指令转化为其他的指令
void setIntDivIsCheap(bool isCheap = true)

为目标处理器添加它支持的所有寄存器类，以表明可以本地支持的数据类型
void addRegisterClass(MVT::ValueType VT, TargetRegisterClass *RC)

在添加了所有的寄存器类之后计算它们的属性
void computeRegisterProperties();

另外，还可能需要实现重载的函数有：
给出所有的目标处理器新添加的需特殊处理的指令或操作数的名字
virtual const char *getTargetNodeName(unsigned Opcode) const;

对给定的限制字符，返回在目标处理器上的限制类型
virtual ConstraintType getConstraintType(char ConstraintLetter) const;

添加在基本块末尾需要进行的特殊处理代码
virtual MachineBasicBlock *InsertAtEndOfBasicBlock(MachineInstr *MI,
MachineBasicBlock *MBB);

完成以上三方面的描述可以实现文件“XXX Lowering.h”、“XXX

Lowering.cpp ”，对于需要更加复杂的特殊匹配，可以通过重载函数 select 来进行处理，完成后实现文件“ XXXISelDAGToDAG..cpp ”，并最终完成 LLVM 中间代码对目标处理器转化的描述。

3.7 汇编输出描述

汇编输出描述也同样分为两方面，实现目标处理器汇编输出的 TableGen 描述以及继承并实现 AsmPrinter 类。

对于 TableGen 描述，则使用用于描述汇编输出的记录类为 AsmWriter 类建立目标处理器需要支持的汇编代码生成器即可，其结构如下：

```
class AsmWriter {
    // 给出汇编代码生成器的名字前缀
    string AsmWriterClassName    = "AsmPrinter";
    // 可指定用于输出打印指令的格式名字
    string InstFormatName = "AsmString";    //
    // 一个汇编代码打印器可以有好几个变种，
    // 这些变种用于在目标处理器有好几种汇编语法格式时做区分
    int Variant = 0;
}
```

而在文件“ XXXAsmPrinter.cpp ”中实现对 AsmPrinter 类的继承。可对需要定制汇编代码输出格式的目标处理器进行设定，并实现指令集描述中指令汇编名和操作数的输出。

基本的格式设定包括：

```
// 设定注释的起始字符串
const char *CommentString;                  // 默认为"#"
// 设定所有全局符号的前缀字符串
const char *GlobalPrefix;                   // 默认为""
// 设定所有私有全局符号的前缀字符串
const char *PrivateGlobalPrefix;            // 默认为"."
// 设定所有全局变量的前缀/后缀字符串
const char *GlobalVarAddrPrefix;            // 默认为""
```

```
const char *GlobalVarAddrSuffix;          // 默认为""
// 设定所有函数的前缀/后缀字符串
const char *FunctionAddrPrefix;           // 默认为""
const char *FunctionAddrSuffix;           // 默认为""
// 设定函数内嵌汇编的起始/结束字符串
const char *InlineAsmStart;               // 默认为"#APP\n"
const char *InlineAsmEnd;                 // 默认为"#NO_APP\n"
// 设定用于在当前段中描述一定数量零字节的标识
const char *ZeroDirective;                // 默认为"\t.zero\t"
// 允许产生包含标准 C 转义符的 ascii 字符串
const char *AsciiDirective;               // 默认为"\t.ascii\t"
// 用于对目标处理器中以零为结尾的字符串进行特殊处理
const char *AscizDirective;               // 默认为"\t.asciz\t"
// 设定用于在当前段中描述整型数据的标识
const char *Data8bitsDirective;           // 默认为"\t.byte\t"
const char *Data16bitsDirective;          // 默认为"\t.short\t"
const char *Data32bitsDirective;          // 默认为"\t.long\t"
const char *Data64bitsDirective;          // 默认为"\t.quad\t"
// 设定对齐的标识
const char *AlignDirective;               // 默认为"\t.align\t"
// 设定对齐是否以字节为单位描述，否则用对数描述
bool AlignmentIsInBytes;                  // 默认为真
// 设定切换段时的标识
const char *SwitchToSectionDirective;     // 默认为"\t.section\t"
// 设定在切换到为函数开设的常数段时需要的标识
const char *ConstantPoolSection;          // 默认为"\t.section .rodata\n"
// 设定在切换到描述静态构建列表时需要的标识
const char *StaticCtorsSection;           // 默认为"\t.section .ctors,\"aw\",@progbits"
// 设定在切换到描述静态析构列表时需要的标识
const char *StaticDtorsSection;           // 默认为"\t.section .dtors,\"aw\",@progbits"
// 如果目标处理器支持的话，用来设定在“ .bss ”、“ .data ”段中的局部块
const char *LCOMMDirective;               // 默认为空
```

```
const char *COMMDirective;                    // 默认为"\t.comm\t"
// 设定是否带第三个参数来指定对齐
bool COMMDirectiveTakesAlignment;     // 默认为真
// 设定目标处理器是否包含标识“ .type ”及“ .size ”
bool HasDotTypeDotSizeDirective;          // 默认为真
```

其它的函数接口则主要有：

```
在开始使用汇编代码打印器时做初始化工作
bool doInitialization(Module &M);
在关闭汇编代码打印器时做收尾工作
bool doFinalization(Module &M);
依次打印出指定函数中指令的汇编代码
bool runOnMachineFunction(MachineFunction &F);
```

另外，对要求进行特殊格式打印的目标处理器指令操作数或寻址方式，需要实现相应的打印处理函数。

3.8 处理器子系列支持

对设计出的每个芯片架构来说，往往能生产出一系列仅有些许差别的处理器。这系列中的处理器往往有一个相同的指令集基，每款处理器在这个指令集基上添加一些特别的指令，或是指令集基中的指令的执行特性不同。这些处理器被称为处理器的子系列，而这些不同的特性被称为处理器的子特性。一个处理器可能拥有数个子特性，也可能只包含最基本的功能，不具有其他的子特性。

目标处理器子系列及子特性支持的实现分为两个方面：实现子特性支持的 TableGen 描述、继承并实现 TargetSubtarget 类。也即需要实现文件“ XXX.td ”、“ XXXSubtarget.h ”和“ XXXSubtarget.cpp ”。

在文件“ XXX.td ”中，可定义目标处理器各系列的各种子特性，以及每一个子系列所提供的子特性的组合。这些子特性可以由 TableGen 提供的记录类描述：

SubtargetFeature 类，用于描述处理器所支持的某种特性，其结构如下：

```
class SubtargetFeature<string n, string a,    string v, string d> {
    // 设定该处理器子特性的名字，之后在使用各种编译工具时便可
```

```
    // 用“ -mattr= ”选项对支持该子特性的处理器做处理
    string Name = n;
    string Attribute = a;        // 设定该处理器子特性的属性名称
    string Value = v;            // 设定该处理器子特性的属性值
    // 设定该处理器子特性的描述信息，之后在使用各种编译工具时
    // 便可用“ -mattr= ”选项来显示这个帮助信息
    string Desc = d;
}
```

Processor 类用于建立能支持的处理器子系列中的某一款，并描述该款处理器所拥有的子特性。其结构如下：

```
class Processor<string n, ProcessorItineraries pi, list<SubtargetFeature> f> {
    // 给定处理器款式名称，之后在使用各种编译工具时便可
    // 用“ -mcpu= ”选项对该款式的处理器做处理
    string Name = n;
    // 给定用于该款处理器的指令执行绑定信息
    ProcessorItineraries ProcItin = pi;
    // 给定该款处理器支持的子特性列表
    list<SubtargetFeature> Features = f;
}
```

例如，对 Intel 的处理器来说，MMX、SSE、SSE2 是三个关于多媒体处理的指令集，一款处理器可能不支持或支持其中的若干种。由此便可建立以下三个子特性：

```
def FeatureMMX : SubtargetFeature<"mmx","X86SSELevel", "MMX",
                                  "Enable MMX instructions">;
def FeatureSSE : SubtargetFeature<"sse", "X86SSELevel", "SSE",
                                  "Enable SSE instructions">;
def FeatureSSE2 : SubtargetFeature<"sse2", "X86SSELevel", "SSE2",
                                  "Enable SSE2 instructions">;
```

而对这三个处理之子特性，Pentium2、Pentium3、Pentium4 这三种 Intel 处理器的子系列分别支持其中的 1 至 3 种，则根据之前说明的 Processor 类，可以建立

三种处理器子系列描述：

```
def : Proc<"pentium2", NoItineraries, [FeatureMMX]>;
def : Proc<"pentium3", NoItineraries, [FeatureMMX, FeatureSSE]>;
def : Proc<"pentium4", NoItineraries, [FeatureMMX, FeatureSSE, FeatureSSE2]>;
```

在文件“ XXXSubtarget.h ”和“ XXXSubtarget.cpp ”中，继承 TargetSubtarget 类添加目标处理器的子系列支持。之后便可使用选项-mcpu=及-mattr=来指定特定的子系列支持进行编译。

3.9 处理器 JIT 支持

JIT（LLVM Just-In-Time compiler，LLVM 即时编译器）可以使目标处理器的代码直接在本地上用工具 lli 运行。通过继承 TargetJITInfo 类可以让目标处理器取得 JIT 支持。目标处理器的 JIT 支持实现分为两个方面：实现 JIT 支持的 TableGen 描述、继承并实现 TargetJITInfo 类。也即需要实现文件“ XXX.td ”、“ XXXJITInfo.h ”和“ XXXJITInfo.cpp ”。

实现 JIT 支持的 TableGen 描述主要是对于目标处理器指令的二进制编码描述，即在指令集描述时对每条指令添加一个指令二进制编码的数据。这里给出一个例子：

```
class InstrARM< bits<4> cond, dag ops, string asmstr, list<dag> pattern >
    : Instruction
{
    field bits<32> Inst;
    let Inst{31-28} = cond;
    bits<4> op;
    let Inst{24-21} = op;
    let Namespace = "ARM";
    let OperandList = ops;
    let AsmString     = asmstr;
    let Pattern = pattern;
}
```

上面中的 Inst 即为存放指令二进制编码的数据，长度为 32，用于描述 32 位的

指令集结构。目前分割出了两部分：Inst{31-28}存放指令的条件码，Inst{24-21}存放指令的操作码，具体的值以及 Inst 其他部分的值由利用类 InstrARM 建立的具体指令给出（Instruction 类的说明见前“指令集描述”一节，此处不赘）。

在文件“XXXJITInfo.h”和“XXXJITInfo.cpp”中，则需继承 TargetJITInfo 类，并实现 TargetJITInfo 类中的虚函数。这些虚函数接口主要有：

重载函数以在编译遍管理器中添加编译遍来实现针对目标处理器的快速代码生成器

```
virtual void addPassesToJITCompile(FunctionPassManager &PM) = 0;
```

使得调用机器代码在 Old 的函数变成调用在 New 的函数

```
virtual void replaceMachineCodeForFunction(void *Old, void *New) = 0;
```

使用指定的机器代码产生器 MCE 来产生一小段目标处理器代码以调用在指定的函数

```
virtual void *emitFunctionStub(void *Fn, MachineCodeEmitter &MCE);
```

返回一个 typedef 来代表未解析的函数调用应该调用的函数

```
typedef void (*LazyResolverFn)();
```

返回一个 typedef 来代表编译函数后对应一个 stub 的 JIT 函数

```
typedef void* (*JITCompilerFn)(void *);
```

用于初始化 JIT

```
virtual LazyResolverFn getLazyResolverFunction(JITCompilerFn);
```

在 JIT 运行产生的一小块代码之前，必要将所有对全局符号的引用替换为实际的地址

```
virtual void relocate(void *Function, MachineRelocation *MR,
                      unsigned NumRelocs, unsigned char* GOTBase);
```

3.10 全局描述

目标处理器基本属性的描述实现分为三个方面：实现基本属性的 TableGen 描述、继承并实现 TargetMachine 类以及声明为目标处理器添加的特定编译遍。也即需要实现文件“ XXX.td ”、“ XXXTargetMachine.h ”和“ XXXTargetMachine.cpp ”、“ XXX.h ”。

在文件“ XXX.td ”中，需要描述目标处理器的机器指针类型、指定函数调用时需保护的寄存器集合、指定汇编代码输出函数以及给出指令集描述等。这些基本属性描述可以由以下几个 TableGen 提供的记录类实现：

Target 类，提供整个目标处理器的全局信息，包含的数据有：

```
// 目标处理器的指针类型，一般为 i32 或 i64
ValueType PointerType
// 函数调用时调用函数需保护的寄存器列表
list<Register> CalleeSavedRegisters
// 给出目标处理器的指令集描述记录
InstrInfo InstructionSet
// 给出目标处理器可用的汇编代码输出函数列表
list<AsmWriter> AssemblyWriters
```

在文件“ XXXTargetMachine.h ”和“ XXXTargetMachine.cpp ”中，需要实例化处理器子系列描述、目标处理器指令集描述、目标处理器帧栈描述、目标处理器指令模式匹配描述等，继承 TargetMachine 类以实现 TargetMachine 类中的虚函数，并在全局的命名空间中使用语句“ RegisterTarget<XXXTargetMachine> X("xxx", "XXX"); ”注册目标处理器。其中处理器子系列描述、目标处理器指令集描述、目标处理器帧栈描述、目标处理器指令模式匹配描述等实现结构可见前文，不赘。下面主要列出 TargetMachine 类的函数接口：

TargetMachine 类的构造函数，需设定目标处理器的名称、大小端，以及在目标处理器上布尔类型、字节类型、短整型、整型、长整型、单精度浮点、双精度浮点、指针的对齐大小以及指针的大小，并提供了这些设置的默认值。其函数原型如下：

```
TargetMachine(const std::string &name, bool LittleEndian = false,
    unsigned char PtrSize = 8, unsigned char PtrAl = 8,
```

```
unsigned char DoubleAl = 8, unsigned char FloatAl = 4,
unsigned char LongAl = 8, unsigned char IntAl = 4,
unsigned char ShortAl = 2, unsigned char ByteAl = 1,
unsigned char BoolAl = 1);
```

函数返回一个对特定模块进行匹配的值，用于在没有设定命令选项“ -march ”时让 LLC 工具决定使用那个后端处理器

```
static unsigned getModuleMatchQuality(const Module &M);
```

类似于 getModuleMatchQuality 函数，返回值用于描述当前本地对目标处理器 JIT 的支持程度

```
static unsigned getJITMatchQuality();
```

重载函数以在编译遍管理器中添加针对目标处理器的编译遍来生成特定的文件

```
virtual bool addPassesToEmitFile(PassManager &PM, std::ostream &Out,
                                 CodeGenFileType FileType, bool Fast);
```

重载函数以在编译遍管理器中添加针对目标处理器的编译遍来生成目标处理器机器代码

```
virtual bool addPassesToEmitMachineCode(FunctionPassManager &PM,
                                        MachineCodeEmitter &MCE);
```

在文件“ XXX.h ”中，主要是放置添加目标处理器所需要的全局函数的声明，例如添加要在编译遍管理器中添加的编译遍函数的声明等。

3.11 本章小节

本章从后端移植机制的角度分析了进行 LLVM 后端接口的各个部分，并给出了实现各个部分的方法以及步骤。

4 实现 LLVM 的 ARM 后端支持

本文完成的后端实现支持生成 ARM 指令集中的大多数常用 32 位指令，包括算术运算、逻辑运算、跳转指令，内存访问指令等。暂不支持浮点运算、Thumb 指令集、DSP 扩展指令生成以及 JIT 编译器。

整个实现过程分为 ARM 处理器结构分析、后端移植实现、环境设置与 ARM 后端生成三大部分，下文中将对三个部分逐个进行说明。

4.1 ARM 处理器结构分析

以下从处理器支持的数据类型与存储对齐、寄存器、指令集、寻址方式、帧栈布局、进程调用约定以及汇编格式等方面对 ARM 处理器[48]进行分析。

4.1.1 数据类型与存储对齐

一般来说，基本数据类型主要分为三大类：整数类型、浮点数类型以及指针类型。ARM 支持所有这三类基本数据类型，并详细定义各类基本类型对应的 ARM 支持的机器数据类型，包括这些机器数据类型的大小，对齐等信息，具体情况见下表。

表 2 ARM 处理器数据类型与对齐

Tab. 2 Data types and alignments of ARM processors

基本类型	对应的机器数据类型	大小（字节）	对齐（字节）
整数类型	无符号字节（Unsigned byte）	1	1
	有符号字节（Signed byte）	1	1
	无符号半字（Unsigned half-word）	2	2
	有符号半字（Signed half-word）	2	2
	无符号单字（Unsigned word）	4	4
	有符号单字（Signed word）	4	4
	无符号双字（Unsigned double-word）	8	8
	有符号双字（Signed double-word）	8	8

浮点数类型	单精度浮点（Single precision） IEEE 754	4	4
	双精度浮点（Single precision） IEEE 754	8	8
指针类型	数据指针（Data pointer）	4	4
	代码指针（Code pointer）	4	4

4.1.2 寄存器

ARM 主要使用 16 个 32 位的整数寄存器，分别被命名为 R0～R15。在下表中，列出了它们的别名、特殊用途情况以及在进程调用时的作用。

表 3 ARM 处理器寄存器功能表

Tab. 3 Functions of registers in ARM processors

寄存器名	别名	特殊用途	标准进程调用时的功能
R15		PC	程序计数器
R14		LR	链接寄存器
R13		SP	栈指针寄存器
R12		IP	进程间调用临时寄存器
R11	V8		变量寄存器 8
R10	V7		变量寄存器 7
R9		V6 SB TR	平台寄存器 寄存器的用途是平台相关的
R8	V5		变量寄存器 5
R7	V4		变量寄存器 4
R6	V3		变量寄存器 3
R5	V2		变量寄存器 2
R4	V1		变量寄存器 1
R3	A4		参数寄存器 4 / 临时寄存器 4
R2	A3		参数寄存器 3 / 临时寄存器 3
R1	A2		参数寄存器 2 / 结果寄存器 2 / 临时寄存器 2
R0	A1		参数寄存器 1 / 结果寄存器 1 / 临时寄存器 1

由上表可知，ARM 有 4 个参数寄存器 R0～R3（A1～A4），它们是调用函数自己需要保护的寄存器。由于个数只有 4 个，当函数传递的参数的大小小于 4 个 32 位寄存器可容纳的大小时，可直接将参数从参数寄存器 A1 开始放置直至放下所有的参数；当函数传递的参数的大小大于 4 个 32 位寄存器的容量时，前 4 个字大小的数据仍然依次放入 A1 至 A4 中，而将剩下的参数放入内存中。

同样对函数的返回结果来说，ARM 只有 2 个结果寄存器，只能容纳下大小小

于 2 个字的结果，并在此时将结果放入寄存器 A1 至 A2 中。对于大于 2 个字的结果来说，将对原来的函数做变形，使之增加一个额外的参数。如对下面的函数 f 的调用，将变成函数 f ’。

原函数 f，返回参数类型为 TT：

TT tt = f(x, ...);

改变后的函数 f ’，无返回参数，先定义一个类型为 TT 的变量 tt，然后增加变量 tt 的地址作为额外的参数：

TT tt;

f’(&tt, x, ...);

寄存器 R4～R8（V1～V5）作为变量寄存器，它们是被调用函数需要保护的寄存器。而对于其他的 7 个寄存器，一般在特定的时候作为特殊寄存器使用，详细的使用方法可见上表。

4.1.3 指令集

ARM 指令为 32 位指令，除去前 4 位后的 28 位指令域类似于其他架构处理器的指令编码情况：根据指令的功能及寻址方式的不同，分成了许多不同的编码格式；对每种编码格式一般设定一个指令码，即 opcode，并设定相应的指令的操作数，个数可为 0～3 个。下面给出了 ARM 的编码规范总图：

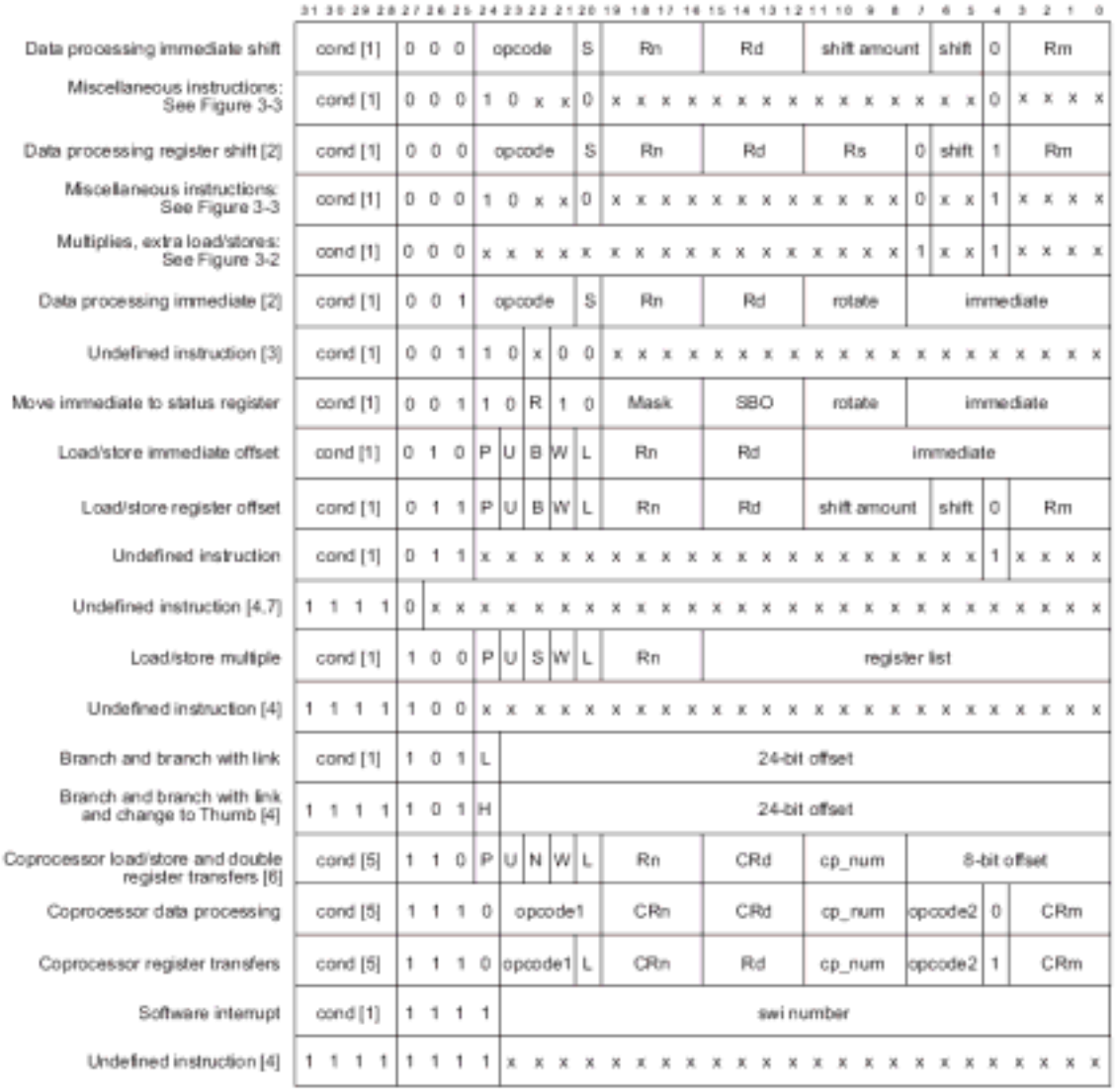

图 10 ARM 指令集编码

Fig. 10 Encodings of ARM instructions

由上图即可知 ARM 在编码时不同于其他处理器的地方：大多数 ARM 指令编码中的前 4 位为条件域，用于指定程序运行到该条指令时执行该指令的条件。这些编码的情况可见下图：

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	(NV)	See *Condition code 0b1111* on page A3-5	-

图 11 ARM 条件码

Fig. 11 ARM condition code

从功能上，常用指令可以分为以下几类：算术运算指令、逻辑运算指令、访存指令、转移类指令等。

基本指令有：mov、mvn、cmp、cmn、teq、tst 等。

算术运算指令有：add、sub、adc、sbc、rsb、rbc 等。

逻辑运算指令有：and、orr、eor 等。

访存指令有：str、strh、strb、ldr、ldrh、ldrb 等。

转移类指令：b、bl 等。

4.1.4 寻址方式

ARM 的寻址方式可分为四大类：数据处理类、单字内的访存、多字的访存、与协处理器相关的。

数据处理类的寻址方式又可以分为三类：立即数、立即数移动、寄存器移动。它们的编码方式见下图：

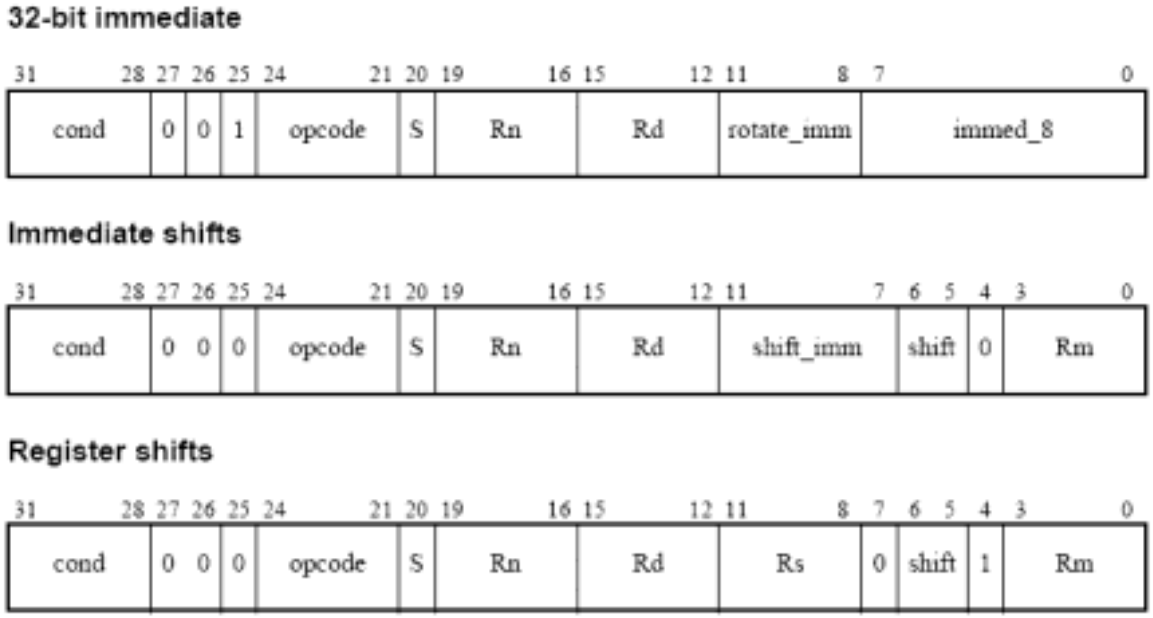

图 12三种数据处理类指令编码

Fig. 12 Three encodings of data processing instructions

对每种数据处理类的寻址方式，它们的操作数情况如下：

对立即数方式，它最多可以有三个操作数，其中两个是寄存器 Rn 和 Rd，另一个为整数立即数。该立即数由 8 位二进制数 immed_8 向右循环移位 2×rotate_imm 位得到。

对立即数移动方式，同样最多可以有三个操作数，其中两个是寄存器 Rn 和 Rd，另一个也为整数立即数。该立即数由寄存器 Rm 进行移动 shift_imm 得到，移动的种类包括：逻辑左移、逻辑右移、算术右移、循环右移。

对寄存器移动方式，同样最多可以有三个操作数，其中两个是寄存器 Rn 和 Rd，另一个为整数。该整数由寄存器 Rm 进行移动寄存器 Rs 中的值得到，移动的种类包括：逻辑左移、逻辑右移、算术右移、循环右移。

对单字内的访存，主要指对字、半字以及字节的访存操作，包括读取和存入两个方向。方式主要包括：立即数偏移寻址、寄存器偏移寻址、前变址寻址，后变址寻址以及比例寻址。这些寻址方式的编码如下：

图 13单字内的访存指令编码

Fig. 13 Encoding of less than one-word memory access instructions

对每种单字内访存类的寻址方式，它们的操作数情况如下：

对立即数偏移/立即数变址，最多两个操作数，一个为寄存器 Rd，另一个为内存地址。该内存地址的值在立即数偏移时为 Rn 加（减）offset_12；在立即数前变址寻址时为 Rn 加（减）offset_12，且当目前的条件码状态满足该条指令的条件域时，更新 Rn 的值为原来的 Rn 值加（减）offset_12；在立即数后变址寻址时为 Rn，且当目前的条件码状态满足该条指令的条件域时，更新 Rn 的值为原来的 Rn 值加（减）offset_12。

对寄存器偏移/寄存器变址，情况类似立即数偏移/立即数变址，只是将原来的 offset_12 变成了寄存器 Rm，不赘述。

对寄存器比例偏移/寄存器比例变址，情况也类似立即数偏移/立即数变址，只是将原来的 offset_12 变成了寄存器 Rm 进行移动，移动包括：逻辑左移、逻辑右移、算术右移、循环右移。不赘述。

多字的访存以及与协处理器相关的寻址方式可细分为许多种，本文尚未实现这些寻址方式，在此略去不赘。

4.1.5 帧栈布局

栈是内存中用于局部变量存储以及子进程调用缺少足够参数寄存器时传递额外

参数的连续内存区域。APCS（ARM Procedure Call Standard，ARM 进程调用规范）规定 ARM 为满递减堆栈，使用寄存器来 SP 做为栈指针，并规定栈的对齐大小为 8 个字节，也即 SP mod 8 = 0。

4.1.6 进程调用约定

函数入口，出口，调用约定（参数传递，函数返回）

当一个函数 A 调用另外一个函数 B 时，A 称为 B 的祖先。当从函数 A 进入到 B 后，为了在完成函数 B 的运行后重新回到函数 A，必须要保存一定的信息，该信息可称为栈回溯结构。在该结构中首先需要保存的是返回函数 A 时的链接地址；其次，由于函数 B 将开辟自己的新局部存贮空间，将有可能修改原来的帧指针 FP 和栈指针 SP，因此也要保存 FP 及 SP；最后，可能需要保存变量寄存器 R4～R10（V1～V7）及传递函数参数的参数寄存器 R0～R3（A1～A4）。由上可得到如下的栈回溯结构（其中“ [] ”内的为可选的保护，即需要保护时才保存）：

当前函数 FP 的指向的地址	[fp]
返回链接地址的值	[fp, #-4]
返回的 SP 值	[fp, #-8]
返回的 FP 值	[fp, #-12]
[保存的 V7 值]	
... ...	
[保存的 V2 值]	
[保存的 V1 值]	
[保存的 A4 值]	
[保存的 A3 值]	
[保存的 A2 值]	
[保存的 A1 值]	

在从函数 A 进入函数 B 时需要在栈中保存上述的栈回溯结构，而在函数 B 执行结束返回 A 时则释放占用的栈空间，设置可能的返回值，并利用保存的信息回到函数 A 继续函数 A 的执行[49]。

4.1.7 汇编格式

ARM 的指令输出可以根据以下几方面进行分类：操作数个数，操作数类型，条件码，下面给出这三种分类的情况。

指令的操作数个数一般为 1 个、2 个或 3 个：

一个操作数　“ B target_address ”，　跳转指令

两个操作数　“ MOV PC, LR ”，　移动指令

三个操作数　“ ADD R1, R2, R3 ”，　加法指令

操作数的类型为寄存器，立即数，立即数偏移，寄存器偏移：

寄存器　“ SUB R3, R5, R7 ”　其中三个操作数均为寄存器

立即数　“ AND R1, R2, #10 ”　其中‘ #10 ’为立即数

立即数偏移　“ LDR R4, [R8, #8] ”　其中‘ [R8, #8] ’为立即数偏移

寄存器偏移　“ STRH R2, [R1, R9] ”　其中‘ [R1, R9] ’为立即数偏移

条件码有 EQ、NE、LT、GT 等多种，因此一条指令也可以有多种形式，以跳转指令 B 为例：

BEQ target_address 该指令使用条件码 EQ，仅当条件标志 Z 设置时执行跳转

BVS target_address 该指令使用条件码 VS，仅当条件标志 V 设置时执行跳转

4.2 后端移植实现

下文中将根据之前对 ARM 处理器结构分析以及前一章对移植接口的描述，逐步实现添加 ARM 处理后端的各个方面，这些接口的实现包括 4 个.td 文件和 13 个.h 及.cpp 文件的编写。

4.2.1 寄存器描述实现

在文件“ ARMRegisterInfo.td ”中定义 ARM 的 16 个 32 位寄存器 R0-R15，其中 R11 为 FP（帧指针），R13 为 SP（栈指针），R14 为 LR（链接寄存器），R15 为 PC（程序计数器），将它们归为 GPRCClass 类，并指明这个寄存器类中可为被调用函数分配使用的寄存器。

在文件“ ARMRegisterInfo.h ”及“ ARMRegisterInfo.cpp ”中，主要实现了以下功能：在 ARM 下将寄存器中内容进出栈的指令；将 LLVM 调整栈指针的伪指令转成实际的 ARM 指令；将 LLVM 的虚拟帧索引（frame index）根据调用函数的具体情况转成栈指针 SP 或是帧指针 FP；实现函数进入前的序言(Prologue)及退出前的尾声（Epilogue）以符合 ARM 的过程调用标准。

实现的具体步骤如下：

首先建立描述 ARM 寄存器的寄存器类 ARMReg，由之前的分析知一共有 16 个寄存器，因此只需 5 个位即可对这些寄存器编号。目前只支持整数寄存器，建立该寄存器的记录类 GPR，这两个类的结构如下：

```
class ARMReg< string n > : Register<n>
{
    field bits<5> Num;
    let Namespace = "ARM";
}
class GPR< bits<5> num, string n > : ARMReg<n>
{
    let Num = num;
}
```

利用记录类 GPR 即可建立所有 16 个整数寄存器的记录定义：

```
def R0 : GPR< 0, "R0">, DwarfRegNum<0>;
def R1 : GPR< 1, "R1">, DwarfRegNum<1>;
def R2 : GPR< 2, "R2">, DwarfRegNum<2>;
… …
… …
… …
def R11 : GPR< 11, "R11">, DwarfRegNum<11>; // FP, 帧指针
def R12 : GPR< 12, "R12">, DwarfRegNum<12>; // 进程间调用临时寄存器
def SP : GPR< 13, "SP">, DwarfRegNum<13>;     // 栈指针
def LR : GPR< 14, "LR">, DwarfRegNum<14>;     // 链接寄存器
def PC : GPR< 15, "PC">, DwarfRegNum<15>;     // 程序计数器
```

这 16 个寄存器的名字分别为 R0，R1，R2，... ，R11，R12，SP，LR，PC，依次映射到 gcc/gdb 寄存器编号 0～15，且互相之间不存在别名关系。所有寄存器定义放入命名空间“ ARM ”中。

由于寄存器都是整数寄存器，对齐都为 32 位，所以可以建立整数寄存器类定义 GPRC，同时设定这 16 个寄存器的 SpillSize 及 SpillAlignment。GPRC 描述的寄存器为 i32 类型，在寄存器分配策略上，采用了比较保守的方式：让参数寄存器和结果寄存器不参与分配，让 SP、LR、PC、 R11，R12 也不参与分配。下面为该定

义的结构：

```
def GPRC : RegisterClass< "ARM", [i32], 32,
[R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, SP, LR, PC] >{
    let MethodProtos =  [{
        iterator allocation_order_begin(MachineFunction &MF) const;
        iterator allocation_order_end(MachineFunction &MF) const;
    }];
    let MethodBodies =  [{
        GPRCClass::iterator GPRCClass::allocation_order_begin(
                                    MachineFunction &MF) const{
            return begin() + 4; // 即不分配寄存器 R0、R1、R2、R3
        }
        GPRCClass::iterator GPRCClass::allocation_order_end(
                                    MachineFunction &MF) const{
            return end() - 5; // 即不分配寄存器 SP、LR、PC、 R11，R12
        }
    }];
}
```

以上的寄存器类 GPRC 描述放入命名空间“ ARM ”中。至此完成了文件“ ARMRegisterInfo.td ”的实现。

实现后，可调用 tblgen 工具生成继承 MRegisterInfo 类的结构 ARMGenRegisterInfo，该命令如下：

```
tblgen ARMRegisterInfo.td -gen-register-desc-header -o ARMGenRegisterInfo.h.inc
```

再建立类 ARMRegisterInfo 继承结构 ARMGenRegisterInfo 即可实现对 MRegisterInfo 类的继承。然后实现以下 MRegisterInfo 类中的虚函数：

```
void ARMRegisterInfo::storeRegToStackSlot( MachineBasicBlock &MBB,
                    MachineBasicBlock::iterator MI,unsigned SrcReg, int FI,
                    const TargetRegisterClass *RC) const;
```

在对寄存器类 GPRC 中的寄存器进行进栈操作时，使用 ARM 的 str 指令：

```
BuildMI( MBB, MI, ARM::STRimmOffAddW, 3).addReg(SrcReg)
                                    .addFrameIndex(FI).addImm(0);
```

```
void ARMRegisterInfo::loadRegFromStackSlot(MachineBasicBlock &MBB,
                    MachineBasicBlock::iterator I,unsigned DestReg, int FI,
                    const TargetRegisterClass *RC) const;
```

在对寄存器类 GPRC 中的寄存器进行出栈操作时，使用 ARM 的 ldr 指令：

```
BuildMI(MBB, I, ARM::LDRimmOffAddW, 2, DestReg)
                                        .addFrameIndex(FI).addImm(0);
```

```
void ARMRegisterInfo::copyRegToReg(MachineBasicBlock &MBB,
                    MachineBasicBlock::iterator I, unsigned DestReg,
                    unsigned SrcReg, const TargetRegisterClass *RC) const;
```

在对寄存器类 GPRC 中的寄存器移动操作时，使用 ARM 的 orr 指令：

```
BuildMI(MBB, I, ARM::ORRrr, 2, DestReg).addReg(SrcReg).addReg(SrcReg);
```

```
MachineInstr *ARMRegisterInfo::foldMemoryOperand(MachineInstr* MI,
                                        unsigned OpNum, int FI) const;
```

在存入内存时使用 ARM 的 str 指令；在读取内存时使用 ARM 的 ldr 指令。建立指令的格式见上，不赘。

```
void ARMRegisterInfo::eliminateCallFramePseudoInstr(MachineFunction &MF,
                    MachineBasicBlock &MBB, MachineBasicBlock::iterator I) const;
```

由于 LLVM 中间代码描述程序时采用虚拟指针来描述，这里需使用 ARM 指令进行真实的栈指针调整。调整的大小可由原来的虚拟指令得知，调整的方向可由原来的虚拟指令编码 ARM::ADJCALLSTACKDOWN 或 ARM::ADJCALLSTACKUP 得知。最后再建立一条 ARM 加法指令 add 即可完成栈指针的调整。

```
void ARMRegisterInfo::eliminateFrameIndex(MachineBasicBlock::iterator II) const;
```

在使用 LLVM 中间代码建立帧时，使用的是虚拟帧指针，此处根据程序情况将当前的虚拟帧指针绑定为 ARM 的帧指针 R11 或者栈指针 SP。同时，ARM 的立即数运算指令的立即数范围有限制（只能为对 8 位二进制数进行偶数移位可表达的数，本文中称为 ARM 的 8 位立即数）。因此，根据需要建立的帧的大小，建立一条（帧的大小可用 ARM 的 8 位立即数表示，用一个操作数为立即数的算术运算指

令）或若干条 ARM 指令（先导入该立即数，用两个操作数均为寄存器的算术运算指令）来进行指针调整。

void ARMRegisterInfo::emitPrologue(MachineFunction &MF) const;

根据程序的情况，进行 ARM 函数调用时的进入调整。如调整当前的栈指针，将其值减少当前需要在栈中分配的数据的大小。并根据该大小是否能用 ARM 的 8 位立即数表示，使用不同的 ARM 指令实现。

void ARMRegisterInfo::emitEpilogue(MachineFunction &MF,
MachineBasicBlock &MBB) const;

根据程序的情况，进行 ARM 函数调用时的退出调整。如调整当前的栈指针，将其值增加当前栈中可以释放的数据的大小。并根据该大小是否能用 ARM 的 8 位立即数表示，使用不同的 ARM 指令实现。

unsigned ARMRegisterInfo::getRARegister() const;

返回 ARM 的链接寄存器 LR

unsigned ARMRegisterInfo::getFrameRegister(MachineFunction &MF) const;

根据当前的程序调用情况，返回 ARM 的帧指针 R11 或者栈指针 SP。

至此可完成文件 “ ARMRegisterInfo.h ”和“ ARMRegisterInfo.cpp ”的实现，并全部完成了后端代码生成器对 ARM 寄存器的描述。

4.2.2 指令集描述实现

在文件“ ARMInstrFormats.td ”中，根据 ARM 指令的二进制格式将他们进行分类，文件“ ARMInstrInfo.td ”根据“ ARMInstrFormats.td ”中定义的分类定义指令。指出指令的操作数，汇编输出格式类型，以及与对应的 LLVM 虚拟指令间的匹配关系及匹配的特殊要求等指令描述信息。

如 ARM 的 add 指令可对应 LLVM 的虚拟指令 add，但是所有的 ARM 立即数操作指令对立即数都有特殊要求，必须是可通过对 8 位二进制数循环移位偶数次能得到的 32 位二进制数，因此不符合要求时要匹配成其他的指令序列。

下面将从指令操作数、寻址方式、算术运算指令、逻辑运算指令、访存指令、函数调用返回指令、条件设定指令、条件跳转指令、伪指令等方面介绍实现过程。

在实现过程中，大部分的指令操作数可以使用 LLVM 已建立的类型描述并打印输出，而有些 ARM 指令对汇编格式有特殊要求，则需要建立记录指定它们的打印函数名称，并在文件“ ARMAsmPrinter.cpp ”中实现。例如：

条件跳转指令的操作数，指定打印函数“ printBranchOperand ”

```
def brtarget : Operand<OtherVT> { let PrintMethod = "printBranchOperand"; }
```

函数调用指令的操作数，指定打印函数“ printCallOperand ”

```
def calltarget : Operand<i32> { let PrintMethod = "printCallOperand"; }
```

条件跳转指令的指令名打印，指定函数“ printCCOperand ”

```
def CCOp : Operand<i32> { let PrintMethod = "printCCOperand"; }
```

操作数为内存中数据的，根据寻址方式可分为两类：

立即数变址寻址，操作数含两个数据分别为 i32imm 和 GPRC 类型

```
def memri : Operand<i32> {
    let PrintMethod = "printMemRegImm";
    let NumMIOperands = 2;
    let MIOperandInfo = (ops i32imm, GPRC);
}
```

寄存器变址寻址，操作数含两个数据均为 GPRC 类型，

```
def memrr : Operand<i32> {
    let PrintMethod = "printMemRegReg";
    let NumMIOperands = 2;
    let MIOperandInfo = (ops GPRC, GPRC);
}
```

较为特殊的寻址方式目前主要有两种，其描述如下：

```
def iaddr : ComplexPattern<i32, 2, "SelectAddrImm", []>;
def xaddr : ComplexPattern<i32, 2, "SelectAddrIdx", []>;
```

这里“ SelectAddrImm ”与“ SelectAddrIdx ”为进行寻址方式匹配的函数名，在文件“ ARMISelDATToDAG.cpp ”中实现（见后文“ 中间代码转换描述实现 ”一节）。

算术运算指令的描述较为简单，以下为寄存器寻址及立即数寻址的加法指令：

```
def ADDrr   : DPI_2_1<
                ( ops GPRC:$dst, GPRC:$src1, GPRC:$src2),
                "add $dst, $src1, $src2",
                [(set GPRC:$dst, (add GPRC:$src1, GPRC:$src2))]
              >;
def ADDri   : DPI_2_1<
                ( ops GPRC:$dst, GPRC:$src1, i32imm:$src2),
                "add $dst, $src1, $src2",
                [(set GPRC:$dst, (add GPRC:$src1, immed_8:$src2))]
              >;
```

两条指令实现的都是‘ add ’指令，不同之处在于第三个操作数 src2，寄存器寻址时为 GPRC 类型即寄存器类型，立即数寻址时为 i32imm 类型即 32 位整数类型。且由于 ARM 指令的限制，在进行立即数寻址指令匹配时，仅当 LLVM 虚拟指令 add 的第三个立即数操作数‘ src2 ’符合‘ immed_8 ’时才能匹配成功。其中‘ immed_8 ’为自定义用于描述“ 能通过对 8 位二进制数用偶数次移位得到的整数 ”（这是 ARM 指令集的限制）的限制条件。

逻辑运算指令类似算术运算，以下给出逻辑异或指令的描述实现：

```
def EORrr   : DPI_2_1<
                ( ops GPRC:$dst, GPRC:$src1, GPRC:$src2),
                "eor $dst, $src1, $src2",
                [(set GPRC:$dst, (xor GPRC:$src1, GPRC:$src2))]
              >;
def EORri   : DPI_2_1<
                ( ops GPRC:$dst, GPRC:$src1, i32imm:$src2),
                "eor $dst, $src1, $src2",
                [(set GPRC:$dst, (xor GPRC:$src1, immed_8:$src2))]
              >;
```

访存指令则较为复杂，有立即数变址寻址与寄存器变址寻址两种寻址方式，以及字节、半字、字三种存取指令，读写内存一共有 12 种情况。以下仅以写内存指

令为例说明，读内存指令类似。

```
def STRimmOffAddW : STW_imm<
                    ( ops GPRC:$rS, memri:$src ),
                    "str $rS, $src",
                    [(store GPRC:$rS, iaddr:$src)]
                >;
```

store 为 LLVM 的字写内存指令，匹配 ARM 的 str 指令，iaddr 为自定义的立即数变址寻址方式（详见前文寻址方式），memri 表明操作数的类型，需要特殊的打印方式（详见前文立即数变址寻址的操作数）。类似地，若将 iaddr 改为 xaddr 则表明为寄存器变址寻址方式，memri 需改为表明寄存器变址寻址时操作数的 memrr 类型。

```
def STRimmOffAddH : STW_imm<
                    ( ops GPRC:$rS, memri:$src ),
                    "strh $rS, $src",
                    [(truncstore GPRC:$rS, iaddr:$src, i16)]
                >;
```

truncstore 为 LLVM 的部分写内存指令，最后指明的 i16 表明为半字操作，匹配 ARM 的 strh 指令。类似地，若将 i16 改为 i8 则表明为字节操作，匹配 ARM 的 strb 指令。

函数调用指令是指 bl 指令，必须定义函数调用可能破坏而非被调用函数需要保护的寄存器集合，并设置 isCall 为 1 指明 bl 指令是一个函数调用指令，且设置 noResults 为 1 表明指令没有运算结果。其实现如下：

```
let isCall = 1, noResults = 1,
    Defs = [R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, LR] in{
    def BL   : Branch< (ops calltarget:$func, variable_ops),  "bl $func", [] >;
}
def : Pat<(ARMcall tglobaladdr:$dst), (BL tglobaladdr:$dst)>;
def : Pat<(ARMcall texternalsym:$dst), (BL texternalsym:$dst)>;
```

bl 指令不直接匹配任何 LLVM 虚拟指令，因此以上后两个记录建立了从自定义的虚拟指令 ARMcall 匹配 bl 指令。其中 ARMcall 是在中间代码生成过程中使用

的虚拟的 ARM 函数调用指令，其记录结构如下：

```
def SDT_ARMCall : SDTypeProfile<0, 1, [SDTCisVT<0, i32>]>;
def ARMcall : SDNode<"ARMISD::CALL", SDT_ARMCall,
        [SDNPHasChain, SDNPOptInFlag, SDNPOutFlag]>;
```

前一条记录描述了自定义虚拟指令 ARMcall 的类型为 SDT_ARMCall，后一条记录描述了 ARMCall 在中间代码生成时使用的指令名‘ ARMISD::CALL ’。

ARM 处理器没有单独的函数返回指令，其函数返回指令通过将链接寄存器中的函数返回地址放入程序计数器的移动自动实现。其记录建立如下：

```
let isReturn = 1, isTerminator = 1, noResults = 1 in{
    def RET: DPI_2_1< ( ops ), "mov pc,lr", [(retflag)] >;
}
def : Pat<(ret), (RET)>;
```

其中 isReturn 表明是返回指令，isTerminator 表明的终结指令（详见 2.1.1 节“ 终结指令 ”）。其中 retflag 是自定义的虚拟指令，其记录结构如下：

```
def SDT_ARMRetFlag : SDTypeProfile<0, 0, []>;
def retflag : SDNode<"ARMISD::RET_FLAG", SDT_ARMRetFlag,
        [SDNPHasChain, SDNPOptInFlag]>;
```

前一条记录描述了自定义的虚拟指令 retflag 的类型为 SDT_ARMRetFlag，后一条记录描述了 retflag 在中间代码生成时使用的指令名‘ ARMISD::RET_FLAG ’。

本文实现中的条件码设定指令是指 ARM 的改变状态寄存器的指令，使用的是 ARM 的 subs 指令，用于匹配自定义的虚拟指令 ARMcmpcc，其实现如下：

```
def SUBCCrr : DPI_2_1<
                ( ops GPRC:$dst, GPRC:$src1, GPRC:$src2),
                "subs $dst, $src1, $src2",
                [(set GPRC:$dst, ( ARMcmpcc GPRC:$src1, GPRC:$src2))]
            >;
```

同时需定义如下条件码：

```
class CC_VAL<int N> : PatLeaf<(i32 N)>;
def CC_NE : CC_VAL< 9>;  // 不等
def CC_EQ : CC_VAL< 1>;  // 等于
```

```
def CC_GT : CC_VAL<10>; // 大于
def CC_LE : CC_VAL< 2>;  // 小于等于
def CC_GE : CC_VAL<11>; // 大于等于
def CC_LT : CC_VAL< 3>;  // 小于
def CC_HI : CC_VAL<12>; // 无符号大于
def CC_LS : CC_VAL< 4>;  // 无符号小于等于
def CC_CS : CC_VAL<13>; // 进位清零/无符号大于等于
def CC_CC : CC_VAL< 5>;  // 进位设置/无符号小于
def CC_PL : CC_VAL<14>; // 正值
def CC_MI : CC_VAL< 6>;  // 负值
def CC_VC : CC_VAL<15>; // 溢出清零
def CC_VS : CC_VAL< 7>;  // 溢出设置
```

类似于条件设定指令，条件跳转指令用于匹配自定义的虚拟指令 ARMbrcc：

```
def BCOND : Branch<
            (ops brtarget:$dst, CCOp:$cc),
            "b$cc $dst",
            [(ARMbrcc bb:$dst, imm:$cc)]
        >;
```

重要的伪指令有两条，均为栈调整指令，实现如下：

```
let hasCtrlDep = 1 in {
def ADJCALLSTACKDOWN : Pseudo<(ops i32imm:$amt),
        "; ADJCALLSTACKDOWN", [(callseq_start imm:$amt)]>;
def ADJCALLSTACKUP : Pseudo<(ops i32imm:$amt),
        "; ADJCALLSTACKUP", [(callseq_end imm:$amt)]>;
}
```

其中 callseq_start 与 callseq_end 均为目标独立匹配节点但与目标格式相关它们的描述如下：

```
def SDT_ARMCallSeq : SDTypeProfile<0, 1, [SDTCisVT<0, i32>]>;
def callseq_start : SDNode<"ISD::CALLSEQ_START",
                    SDT_ARMCallSeq, [SDNPHasChain]>;
```

```
def callseq_end : SDNode<"ISD::CALLSEQ_END",
                         SDT_ARMCallSeq, [SDNPHasChain]>;
```

文件“ ARMInstrInfo.h ”与“ ARMInstrInfo.cpp ”则实现判断指令的类型，如是否是移动指令（move），是否是从栈顶取值到寄存器的指令等。

4.2.3 帧栈布局描述实现

在本文的移植中对 ARM 实现降序栈，同时由之前对 ARM 帧栈布局的分析可知，对 TargetFrameInfo 类中关于栈布局的描述 StackDir，StackAlignment 及 LocalAreaOffset 可分别设置为 StackGrowsDown，8 及 0。另外，ARM 对调用函数需保存的寄存器在溢出时在栈中的保存位置没有特殊的要求，因此也无需实现函数 getCalleeSaveSpillSlots()。

4.2.4 中间代码转换描述实现

针对 ARM 处理器的中间代码转化主要需实现文件“ ARMLowering.h ”、“ ARMLowering.cpp ”及文件“ ARMISelDATToDAG.cpp ”。

在文件“ ARMLowering.h ”及“ ARMLowering.cpp ”中，建立继承 TargetLowering 的类 ARMTargetLowering。在新建立的类中中完成的功能包括以下几方面：

1．将 LLVM 的函数参数转化为合法的 ARM 的函数参数
2．将 LLVM 的函数返回值转化为合法的 ARM 的函数返回值
3．设定 LLVM 虚拟指令对 ARM 指令的指令匹配情况
4．其他描述
5．其他函数重载

对于上述功能 1，在重载的函数 LowerArguments 中将所有小于 MVT::i32 的整数类型，如 MVT::i1、MVT::i8、MVT::i16，全部提升转化（根据原来数据的类型进行符号扩展或无符号扩展）为 ARM 的最小整数寄存器的类型 MVT::i32；对所有大于 MVT::i32 的类型的数据全部拆分为 MVT::i32 类型的数据。在将所有的参数都转化为 MVT::i32 类型之后，将这些函数参数分配到 ARM 进程调用规范规定的参数寄存器 R0～R3 中。并在当所有参数所需存放的寄存器个数超过可能用的总数（4 个）时，添加相应的 ARM 指令将其余的参数置入内存中。

对于上述功能 2，在重载的函数 LowerCallTo 中对函数的返回值类型进行检查。若返回值类型均不大于 MVT::i32 的整数类型，如 MVT::i1、MVT::i8、MVT::i16、

MVT::i32，则将小于 MVT::i32 类型的数据全部提升转化（根据原来数据的类型进行符号扩展或无符号扩展）为 ARM 的最小整数寄存器的类型 MVT::i32。此时，将转化后的返回值放入 ARM 进程调用规范规定的返回寄存器 R0。而当返回值类型大于 MVT::i32 时，则添加相应的 ARM 指令将该返回数据置入内存中返回。

对于上述功能 3，描述设定了大多数 LLVM 指令与 ARM 指令的关系。对于需进行“ Promote ”操作的指令，如 LLVM 指令“ ISD::TRUNCSTORE ”，ARM 指令提供了对于 8 位字节、16 位半字类型的操作，但是不支持 1 位的比特位操作，因此需提升操作数类型的“ Promote ”操作：

setOperationAction(ISD::TRUNCSTORE, MVT::i1, Promote);

对于许多 ARM 不支持的指令，或是目前本文的移植暂不支持的 LLVM 虚拟指令，则指定该指令为需要扩展（“ Expand ”操作）的。目前这些指令包括：

MVT::i32 类型的 ISD::SDIV、ISD::UDIV、ISD::SREM、ISD::UREM、ISD::CTTZ、ISD::CTLZ、ISD::CTPOP 等

MVT::f32 及 MVT::f64 类型的 ISD::FADD、ISD::FSUB、ISD::FDIV、ISD::FREM、ISD::FCOPYSIGN、ISD::FCOPYSIGN 等

MVT::v8i8、MVT::v4i16、MVT::v2i32、MVT::v16i8、MVT::v8i16、MVT::v4i32、MVT::v2i64 等向量类型的运算操作如 ISD::ADD、ISD::VECTOR_SHUFFLE 等

另外，需进行以下设定以指示需要进行特殊匹配处理的 LLVM 虚拟指令：

setOperationAction(ISD::SELECT_CC, MVT::i32, Custom);

setOperationAction(ISD::BR_CC, MVT::i32, Custom);

setOperationAction(ISD::RET, MVT::Other, Custom);

setOperationAction(ISD::GlobalAddress, MVT::i32, Custom);

setOperationAction(ISD::ConstantPool, MVT::i32, Custom);

实际的匹配处理则在函数 LowerOperation 这个接口中实现。例如，ISD::BR_CC 指令实际是通过两条 ARM 指令‘ subs ’及‘ bxx ’实现。其中，‘ subs ’实现原虚拟指令的比较部分，并设置比较结果的标志；而‘ bxx ’指令则根据原虚拟指令的跳转条件码设定‘ xx ’，进行特定情况下的跳转动作。

对于上述功能 4，主要包括以下几个方面：

a．添加 ARM 处理器的寄存器类型，即通过语句“ addRegisterClass(MVT::i32, ARM::GPRCRegisterClass) ”添加目前支持的 32 位整数寄存器类型

b．通过语句“ setStackPointerRegisterToSaveRestore(ARM::SP) ” 设定 ARM 的栈指针为 R13

c．通过语句“ setIntDivIsCheap(false) ”表明目前不支持支持除法

d．通过语句“ setSetCCIsExpensive() ”表明执行 setcc 指令开销很大

对于上述功能 5 中所需重载的函数主要有以下几个：

a．在函数 InsertAtEndOfBasicBlock 中对基本块的选择使用 ARM 的条件跳转指令 bxx，其中条件‘ xx ’为之前匹配得到的条件跳转指令的条件

b．在函数 getTargetNodeName 中需特殊处理的指令的名字，包括 ARMISD::SELECT_CC 、 ARMISD::CMPCC 、 ARMISD::BRCC 、 ARMISD::CALL 等

文件“ ARMISelDATToDAG.cpp ”中，重载 select 函数以实现对 ARM 的复杂寻址方式的匹配支持，目前实现了对立即数偏移寻址、寄存器偏移寻址的支持，即前文“ 指令集描述实现 ”一节中提到的函数“ SelectAddrImm ”与“ SelectAddrIdx ”。其中，函数 SelectAddrImm 用于判断一个地址是否可以用寄存器加立即数偏移的方式得到，并在可以时给出基址和偏移值；函数 SelectAddrIdx 则尽量将原来得到地址的运算变为寄存器加寄存器的方式，并在可以时给出基址寄存器和偏移寄存器。

4.2.5 汇编输出描述实现

对于一般的指令的汇编输出，在建立指令记录的文件“ ARMInstrInfo.td ”中同时已经给出了其汇编格式。如对于立即数寻址的逻辑与指令，其建立的记录如下：

```
def ANDri : DPI_2_1<
            ( ops GPRC:$dst, GPRC:$src1, i32imm:$src2),
            "and $dst, $src1, $src2",
            [(set GPRC:$dst, (and GPRC:$src1, immed_8:$src2))]
        >;
```

其中"and $dst, $src1, $src2"给出了指令的整体格式，指令名为 and，同时由“ ops GPRC:$dst, GPRC:$src1, i32imm:$src2 ” 知三个操作数中的 dst 与 src1 为 GPRC 类型，可以用打印寄存器的方式打印；而操作数 src2 为 i32imm 类型，可以用打印立即数的方式打印。

对于复杂的指令打印，即指令集打印中指定特殊打印函数的操作数或函数名，则需要实现相应的打印函数。

例如，实现操作数为立即数偏移或寄存器偏移的指令的打印。

而对于根据条件码需打印不同的指令名的指令，其实现如下：

```
def BCOND : Branch<
```

```
    (ops brtarget:$dst, CCOp:$cc),
    "b$cc $dst",
    [(ARMbrcc bb:$dst, imm:$cc)]
>;
```

其中‘ cc ’为该条指令的条件码，该条件码从进行指令匹配的自定义的 LLVM 虚拟指令 ARMbrcc 处得到，并在打印时对‘ cc ’使用该类型‘ CCOp ’的特殊打印函数即可实现。

所有之前提到汇编指令打印同时均需根据 GCC 的汇编器 Gas 的输入文件格式要求设定生成汇编文件的格式，如注释使用的特殊符号、全局变量的前缀，汇编文件头及汇编文件尾等。

最后，在文件“ ARMAsmPrinter.cpp ”中还需实现为 ARM 后端添加汇编代码输出编译遍函数 createARMAsmPrinter，在该函数中返回之前实现的继承 AsmPrinter 类的 ARMAsmPrinter 类的对象。

4.2.6 子系列支持实现

需添加文件“ ARMSubtarget.h ”及“ ARMSubtarget.cpp ”，目前实现的是一个通用的 32 位 ARM 指令集，暂时未实现对特殊处理器子特性的支持。目前继承了包含处理器子系列支持的 TargetSubtarget 类，并提供了相应的接口，可在今后添加对 ARM 子特性和子系列的支持。

4.2.7 JIT 支持实现

部分实现了对指令集的机器码编码描述，目前暂不支持 JIT。

4.2.8 全局描述实现

在文件 ARM.td 中指明 ARM 的指针类型为 32 位，并确定被调用函数所需保护的寄存器。在“ ARMTargetMachine.h ”和“ ARMTargetMachine.cpp ”中，除了对 TargetMachine 类实例化处理器子系列描述、目标处理器指令集描述、目标处理器帧栈描述以及目标处理器指令模式匹配描述，还需要实现三个函数：TargetMachine 的构造函数、模块匹配质量函数 getModuleMatchQuality 以及向编译遍管理器添加编译遍的函数 addPassesToEmitFile。

对 TargetMachine 构造函数，从之前对 ARM 处理器进行的数据格式及存储对齐分析可知，可以设定 TargetMachine("ARM", false, 4, 4, 8, 4, 4, 4, 2, 1, 1)。

对函数 getModuleMatchQuality，可以让它在模块中匹配到字符串“ arm- ”时返回较高的匹配度如 20，而在不匹配的时候返回一个较低值，如 5 或 0。

对函数 addPassesToEmitFile，需要将针对 ARM 处理器后端进行处理的编译遍，即编译遍函数 createARMISelDag 和 createARMAsmPrinter，加入编译遍管理器，需要添加以下语句：

```
PM.add(createARMISelDag(*this));
PM.add(createARMAsmPrinter(Out, *this));
```

在文件“ ARM.h ”中，则相应添加这两个函数的声明。主要用来在编译遍管理器中添加指令选择匹配的编译遍和添加汇编代码打印的编译遍：

```
FunctionPass *createARMISelDag(ARMTargetMachine &TM);
FunctionPass *createARMAsmPrinter(std::ostream &OS,
                                  ARMTargetMachine &TM);
```

另外，在文件“ ARM.h ”中注册 ARM 后端，可以通过在全局命名空间中添加语句“ RegisterTarget<ARMTargetMachine> X("arm", " ARM") ”。并在该文件中声明 ARM 的条件码及确定其与 LLVM 预定义条件码之间的对应关系。

4.3 环境设置与 ARM 后端生成

首先在 LLVM 源代码树的根目录下的文件“ Makefile.config ”中 TARGETS_TO_BUILD 项中添加 ARM 后端。其次在同样根目录下的文件“ configure ”中的 llvm_cv_target_arch 项设置中添加 ARM 选择项，并在相应的其他项中作设置，如 TARGET_HAS_JIT 设为 0 表明不支持 JIT 编译器。

在需要生成工具的 Makefile 中添加 ARM 处理项。如在 llc 工具的 Makefile 中添加 USEDLIBS+=LLVMARM。

将 ARM 后端的描述文件放在 LLVM 源代码根目录下的/lib/Target/ARM/下，并编写编译这些文件所需的 Makefile。

执行前文所述修改后的 configure 脚本，并加入选项--enable-targets=arm，运行后即可得到支持 ARM 后端的 LLVM 编译系统，得到支持 ARM 后端的 LLVM 编译器及其他工具。

例如，通过以下调用即可生成 ARM 汇编代码：

1. llvm-gcc / llvm-g++ -c xxx.c / xxx.cpp 生成 xxx.o
2. llc -march=arm -o xxx.s xxx.o 生成 xxx.s

再利用 GCC 的 ARM 交叉编译工具便可生成 ARM 的可执行代码。

本文的测试包括单指令测试以及函数测试。单指令测试编写单条 LLVM 虚拟指令，通过检查编译后对应的 ARM 指令检验生成代码的正确性。函数测试通过编写不同复杂程度的函数来测试生成函数正确性。其中可按函数类型来测试，如：无参数无返回值函数、单个简单参数无返回值函数、无参数简单返回值函数等等。这些测试的大部分可利用 LLVM 源代码目录下的/test/Regression/CodeGen/目录下的大量 LLVM 汇编代码完成。具体步骤如下：

1. 运行命令 llvm-as xxx.ll 生成 xxx.bc
2. 运行命令 llc -march=arm xxx.bc 生成 ARM 汇编文件 xxx.s

通过检查文件 xxx.s 检验生成代码的正确性，本文最后生成的 ARM 后端已通过大部分 LLVM 提供的相关测试。

另外，参与 UADL 项目的同学实现了 ARM 处理器的仿真器，该仿真器支持直接运行 ARM 的汇编代码，部分对 LLVM 的 ARM 后端的测试通过使用该仿真器进行，即将 LLVM 生成的 ARM 汇编代码在 ARM 仿真器上运行进行测试验证结果的正确性。这些测试包括一些含有简单运算的代码段，测试同样表明本文生成的 ARM 后端可以正确生成 ARM 汇编代码。

4.4 本章小节

本章在深入分析 ARM 处理器体系结构各个方面的特性之后，结合前一章对 LLVM 后端移植接口的分析，对照 ARM 后端移植工作的实际步骤，逐步实现了为支持 ARM 后端所需的接口的各个部分，并给出了完成接口后生成后端工具的方法以及测试的步骤和结果。

5 总结与展望

本文介绍了当前最新最完善的开放源代码编译系统——LLVM 编译系统，详细分析了该系统的各个组成部分及功能之后，深入探讨了该编译系统的后端移植接口，并实现了广泛使用的 ARM 处理器的后端支持。目前移植的 ARM 后端已经可以生成 ARM 的汇编代码，并已验证了对简单程序生成汇编代码的正确性。由于时间原因，现阶段尚未完成对 ARM 的 Thumb 指令集、DSP 扩展指令集以及浮点运算指令集等一些 ARM 子系列的子特性的支持，对 JIT 编译器的支持也只完成了部分。随着工作的继续进行，增加上述功能的同时还可对生成的代码质量作进一步的优化和改进。

本文的工作是 UADL 项目的一部分，之前项目已经实现对 ARM 公司的 ARM7TDMI 处理器的仿真器生成[51]，至此也完成了 LLVM 编译系统的移植。通过对仿真器生成及 LLVM 编译系统后端移植的工作进展的分析，发现两种软件开发工具所需要的移植信息有相当大的部分是类似甚至是一致的，因此整合这些信息并进一步建立完善 UADL 语言及 UOSM 模型来统一描述将大大减少软件开发工具移植的工作量，并可能实现自动生成各种开发工具，由此将使国内在软件工具链方面的开发能力得到增强，极大地减少芯片开发过程中配套工具链的完成时间，改进现有的芯片开发流程，加快电子系统开发速度，降低开发成本，为具体应用快速开发量体裁衣的芯片及系列支撑软件打下坚实基础，大大降低研制自主知识产权高性能处理芯片的门槛，显著加速产业界的产品整合和研究项目的产业化，加快芯片的应用和推广。

参考文献

[1] 张天骐，林孝康，余翔。“ SoC 系统的低功耗设计 ”，单片机及嵌入式系统应用 No.6 P.17-19，2004 年。

[2] H. Muljono, S. Rusu, B. Cherkauer, and J. Stinson. Two new 130nm Itanium 2 processors for 2003. In HOT Chips 15 Proceedings, 2003.

[3] Intel Corporation. Moore's Law. http://www.intel.com/technology/mooreslaw/index.htm

[4] 白杉。“ 光刻机曝光技术演进 ”，集成电路应用，2003 年 12 月。

[5] M. Lapedus. IC makers brace for $3 million mask at 65nm, EETimes, September 2003.

[6] C. Souza. Semiconductor IP houses struggle to survive as ASIC design starts continue to dwindle http://www.my-esm.com/showArticle.jhtml?articleID=1010007205/26/2003

[7] A. Vincentelli. Defining platform-based design, EEDesign of EETimes, February 2002.

[8] J. Turley. The two percent solution http://www.embedded.com/shared/printableArticle.jhtml?articleID=9900861 December 2002.

[9] The Free Software Foundation. http://www.fsf.org

[10] CGEN. http://sources.redhat.com/cgen/docs-1.0/cgen.html#SEC_Top

[11] 朱少波。“ 基于 GCC 开发 C 编译器的研究与实践 ”，浙江大学硕士学位论文，2003 年 02 月。

[12] 蔡杰。“ GCC 编译系统结构分析与后端移植实践 ”，浙江大学硕士学位论文，2004 年 02 月。

[13] 冯钢。“ 基于 GCC 的嵌入式系统编译器研究与开发 ”，浙江大学硕士学位论文，2004 年 03 月。

[14] 任小西。“ 嵌入式系统编译器的快速生成方法研究 ”，湖南大学硕士学位论文，2004 年 03 月。

[15] P. Podge. Retargettable Code Generation using Sim-nML Machine Description. Department of Computer Science & Engineering, Indian Institute of Technology, Kanpur. April 2000.

[16] 陈建智。“ Binary Utilities 自動移植生成器 ”，国立台湾大学信息工程学研究所硕士学位论文，2005 年。

[17] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture description languages for system-on-chip design. In Proceedings of the Sixth Asia Pacific Conference on Chip Design Language (APCHDL), 1999.

[18] W. Qin and S. Malik. Architecture description languages for retargetable compilation. In Y. N. Srikant and P. Shankar, editors, Compiler Design Handbook: Optimizations & Machine Code Generation, pages 535-564. CRC Press, 2002.

[19] G. Zimmerman. The MIMOLA design system: A computer-aided processor design method. In Proceedings of Design Automation Conference, pages 53–58, June 1979.

[20] H. Akaboshi. A Study on Design Support for Computer Architecture Design. PhD thesis, Department of Information Systems, Kyushu University, Japan, 1996.

[21] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In Proceedings of Design Automation Conference, pages 299–302, June 1997.

[22] N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, pages 176–192, 1998.

[23] A. Fauth, J. V. Praet, and M. Freericks. Describing instructions set processors using nML. In Proceedings of Conference on Design Automation and Test in Europe, pages 503–507, Paris, France, 1995.

[24] S. Őnder and R. Gupta. Automatic generation of microarchitecture simulators. In Proceedings of the IEEE International Conference on Computer Languages, pages 80–89, May 1998.

[25] J. Teich, R. Weper, D. Fischer, and S. Trinkert. A joined architecture/compiler environment for ASIPs. In Proceedings of International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pages 26–33, San Jose, CA, Nov 2000.

[26] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In Proceedings of Design Automation Conference, pages 933–938, 1999.

[27] C. Siska. A processor description language supporting retargetable multi-pipeline DSP program development tools. In Proceedings of the International Symposium on System Synthesis, pages 31–36, 1998.

[28] S. Rigo, R. J. Azevedo, and G. Araujo. The ArchC architecture description language. Technical Report 15, Institute of Computing of the University of Campinas, Brazil, 2003.

[29] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In Proceedings of Conference on Design Automation and Test in Europe, pages 485– 490, 1999.

[30] A. Terechko, E. Pol, and J. van Eijndhoven. PRMDL: A machine description language for clustered VLIW architectures. In Proceedings of Conference on Design Automation and Test in Europe, page 821, 2001.

[31] J. C. Gyllenhaal, W. Hwu, and B. R. Rao. HMDES version 2.0 specification. Technical Report IMPACT-96-3, University of Illinois at Urbana-Champaign, 1996.

[32] D. G. Bradlee, R. R. Henry, and S. J. Eggers. The Marion system for retargetable instruction scheduling. In Proceedings of the Conference on Programming Language Design and Implementation, June 1991.

[33] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In Proceedings of Design Automation Conference, pages 184–188, 2001.

[34] W. S. Mong and J. Zhu. A retargetable micro-architecture simulator. In Proceedings of Design Automation Conference, pages 752–757, 2003.

[35] Target Compiler Technologies N.V. http://www.retarget.com

[36] 陆岚。“ 嵌入式系统可重定向功耗优化编译器研究 ”，中国科学技术大学博士学位论文，2004 年 05 月。

[37] J. Lee，T. Hwang。“ 前瞻网络安全处理器及相关 SOC 设计与测试技术研发 ”，台湾清华大学集成电路设计技术研发中心

http://pllab.cs.nthu.edu.tw/research/MOEA_B2.ppt
[38] “高移植性 GCC 编译器，结构描述语言 ADL，自动化系统工具软件产生”，台湾清华大学集成电路设计技术研发中心 http://pllab.cs.nthu.edu.tw/research/pllab_and_ossti.doc
[39] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen and H. Meyr. A Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using the Machine Description Language LISA, Integrated Signal Processing Systems, RWTH Aachen, Germany.
[40] W. Qin. Modeling and Description of Embedded Processors for the Development of Software Tools, department of electrical engineering, Princeton University, November 2004.
[41] P. Schaumont, B. Cheng, C. Lai, W. Qin, I. Verbauwhede. Cooperative Multithreading on Embedded Multiprocessor Architectures Enables Energy-scalable Design,l DAC 2005, June 13-17, Anaheim, California, USA.
[42] B. Cheng, C. Lai, P. Schaumont, W. Qin, I. Verbauwhede. Energy and Performance Analysis of Mapping Parallel Multithreaded Tasks for An On-Chip Multi-Processor System, International Conference on Computer Design (ICCD), 2-5 October 2005, San Jose, CA, USA.
[43] The LLVM Compiler Infrastructure. http://llvm.org/
[44] T. Brethour, J. Stanley, B. Wendling. An LLVM Implementation of SAPRE[C]. UIUC, December 2002.
[45] 张明军，罗军。“缓冲区溢出静态分析中的指针分析算法”，计算机工程，第 31 卷第 18 期，2005 年 9 月。
[46] C. Lattner, V. Adve. Architecture for a Next-Generation GCC[C]. First Annual GCC Developers' Summit, Ottawa, Canada, May 2003.
[47] V. Adve, C. Lattner, M. Brukman. LLVA: A Low-level Virtual Instruction Set Architecture[C]. Proc. of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36), San Diego, CA, December 2003.
[48] ARM Architecture Reference Manual [M]. ARM Limited, 2000.
[49] Procedure Call Standard for the ARM Architecture [M]. ARM Limited, 2006.
[50] The GNU Project. http://www.gnu.org
[51] 马潇，谢凯年，董峰。“基于 MADL 语言的时钟精确级仿真器生成”，计

算机仿真（已录用）。

附录 1 LLVM 虚拟指令列表

1. 单指令

终结指令共有 6 条：

ret	用于从函数返回到调用者
br	从函数中的一个基本块有条件或无条件地转移到另一个基本块
switch	根据条件将控制流从一个基本块转移到数个基本块之一
invoke	用于调用一个函数
unwind	用于实现例外处理
unreachable	用于告诉代码优化器某一段代码是不可到达的

算术运算指令包含以下常用指令：

add	加法	sub	减法	mul	乘法
udiv	无符号数除法	sdiv	有符号数除法	fdiv	浮点数除法
urem	无符号数取余	srem	有符号数取余	frem	浮点数取余

逻辑运算指令包括以下常用指令：

and	逻辑与	or	逻辑或	xor	逻辑异或
shl	左移	lshr	逻辑右移	ashr	算术右移

向量运算指令包括以下指令：

extractelement	从一个向量中取出一个指定的向量元素
insertelement	将一个向量元素加入向量中的指定位置
shufflevector	根据掩码从两个向量中挑选向量元素组成新的向量

访存及地址操作指令包括以下常用指令：

malloc	在系统的堆中分配一块内存空间
free	释放一块之前在系统的堆中用 malloc 分配的内存空间
alloca	在当前进程的栈帧中分配内存，并在函数返回时自动释放
load	从内存中读数据
store	将数据写入内存

getelementptr　取得一个聚合数据结构中元素的地址

类型转换指令包括以下常用指令：

trunc .. to　将较大整数类型表达的整数转成用较小整数类型表达
zext .. to　将较小整数类型表达的整数无符号扩展到用较大整数类型表达
sext .. to　将较小整数类型表达的整数有符号扩展到用较大整数类型表达
fptrunc .. to　将较大浮点类型表达的浮点数转成用较小浮点类型表达
fpext .. to　将较小浮点类型表达的浮点数扩展到用较大浮点类型表达
fptoui .. to　将用浮点类型表达的浮点数转成用无符号整数类型表达的整数
fptosi .. to　将用浮点类型表达的浮点数转成用有符号整数类型表达的整数
uitofp .. to　将用无符号整数类型表达的整数转成用浮点类型表达的浮点数
sitofp .. to　将用有符号整数类型表达的整数转成用浮点类型表达的浮点数
ptrtoint .. to　将指针类型的指针转成用整数类型表达的整数
inttoptr .. to　将用整数类型表达的整数转成用指针类型表达的指针
bitcast .. to　将一类型表达的数据转成与原类型大小相等的另一类型的数据

注：本文所述整数类型以及浮点类型，其“较大”、“较小”以及“相等”均指该类型表达数据使用的位数的多少，较多称为较大，较少称为较小，一样称为相等。

其他指令主要有：

icmp　比较两个整数是否满足指定的条件关系，如大于、小于等
fcmp　比较两个浮点数是否满足指定的条件关系，如大于、小于等
phi　用于实现表达函数的 SSA 图的φ节点，必须是基本块的第一条指令
select　根据当前条件比较的结果从两个数据中选择一个返回
call　用于调用一个函数，是 invoke 指令的简化版本
va_arg　用于访问允许可变参数的函数的每个参数

2. 内函数

可变参数处理内函数包含以下几种：

llvm.va_start　用于初始化之后可为 va_arg 指令使用的参数列表
llvm.va_end　用于释放之前使用 llvm.va_start 建立的参数列表
llvm.va_copy　用于复制参数列表

精确垃圾回收内函数包含以下几种：

llvm.gcroot　　用于向代码生成器声明一个垃圾回收根的存在

llvm.gcread　　用于识别出对堆引用的读操作

llvm.gcwrite　　用于识别出对堆引用的写操作

代码生成内函数包含以下几种：

llvm.returnaddress　　尝试计算出代表当前函数或它的调用者地址的目标相关值

llvm.frameaddress　　尝试对指定的栈帧计算出目标相关的帧指针值

llvm.stacksave　　用于记录当前函数的栈状态

llvm.stackrestore 用于恢复之前 llvm.stacksave 记录的函数栈状态

llvm.prefetch　　用于提示代码生成器插入可能支持的预取指令

llvm.pcmarker　　提供将程序计数器输出给仿真器或其他工具的方法

llvm.readcyclecounter 在支持的目标处理器上提供访问周期计数寄存器的方法

标准 C 库内函数包含以下几种：

llvm.memcpy.*　　将源地址的一段内存数据复制到目标地址处

llvm.memmove.*　　将源地址的一段内存数据移动到目标地址处

llvm.memset.*　　将一段内存中的数据用指定的字节填充

llvm.isunordered.*　　判断两个浮点数据是否是不可排序的

llvm.sqrt.*　　返回指定数据的平方根值

llvm.powi.*　　返回指定数据的指定次数的方值

位操作内函数包含以下几种：

llvm.bswap.*　　将数据按原来的字节逆序排列，*可为 i16、i32、i64

llvm.ctpop.*　　计算数据中置为‘ 1 ’的位的个数，*可为 i8、i16、i32、i64

llvm.ctlz.*　　计算数据中前导的‘ 0 ’的位个数，*可为 i8、i16、i32、i64

llvm.cttz.*　　计算数据中尾部的‘ 0 ’的位个数，*可为 i8、i16、i32、i64

附录 2 链接类型与调用约定

链接类型可有以下几种：

internal　　表明只在当前模块被使用

linkonce　　类似于 internal，不同在于当链接两个模块时会将两个 linkonce 变量或函数丢弃，主要用于实现内联函数

weak　　类似于 linkonce，不同在于未被引用的 weak 全局符号不会被丢弃

appending　仅应用于指向数组类型的指针，在链接模块时将两个数组连在一起

externally visible　　不是以上几种类型时，用此类型表明外部可见

调用约定有以下几种：

"ccc"　　C 语言调用约定，为默认值，允许支持可变参数函数，并容忍一些在原型声明时的不一致

"csretcc"　　C 结构返回调用约定，类似于 C 语言调用约定，但是该约定规定函数的一个参数必须为指针且返回类型为空，主要用于返回聚合类

"fastcc"　　快速调用约定，该种调用约定力图使得调用过程最快，允许使用各种支持的技巧来优化，但不支持可变参数且函数原型必须与函数定义匹配

"coldcc"冷调用约定，在假设调用一般不会运行的情况下用于使调用者的效率提高，即这些调用保存所有的寄存器以不影响调用者

"cc <n>"　　计数的调用约定，通过该方式可以定义目标处理器需要的调用约定，n 的取值从 64 开始

附录 3 LLVM 衍生类型

数组类型用于描述一系列线性的数据，如“ [10 x ubyte] ”表示一个含 10 个无符号 8 位数的类型。

函数类型用于描述函数，由返回类型和参数类表组成。如“ bool (int, float) ”表示一个返回参数为布尔类型，有两个参数分别为 int 类型和 float 类型的函数。

结构类型用于描述一系列类型数据的组合，由一系列的类型组成，结构中各类型之间的填充大小由目标处理器决定。如“ { bool, int, long, double} ”表示一个由四个分别为 bool、int、long、double 类型组成的结构。

打包结构类型类似结构类型，区别在于各组成类型的数据之间无填充。

指针类型用于描述执行某类型数据的指针。如“ [5 x int] * ”表示一个指向一个数组，该数组由四个 int 类型的数据组成。

打包类用于描述适用于 SIMD（Single Instruction Multiple Data，单指令多数据）指令的操作数，该类型由一个向量个数及一个向量元素类型定义。向量个数必须为 2 的指数个，向量元素类型必须为原子类型。如“ < 8 x short > ”就定义了一个含有 8 个 short 类型元素的打包类型。

附录 4 LLVM 集成库列表

核心库包括：

LLVMArchive	用于读写 LLVM 压缩格式
LLVMAsmParser	用于解析 LLVM 汇编代码
LLVMBCReader	用于读 LLVM 字节码（bytecode）
LLVMBCWriter	用于写 LLVM 字节码（bytecode）
LLVMCore	用于提供 LLVM 中间表达描述
LLVMDebugger	用于提供源代码级调试支持
LLVMLinker	用于提供字节码及压缩链接接口
LLVMSupport	用于提供一般的支持
LLVMSystem	用于提供操作系统级的抽象层
LLVMbzip2	用于提供 BZip2 压缩算法支持

分析库包括：

LLVMAnalysis	用于提供各种分析编译遍
LLVMDataStructure	用于提供数据结构分析编译遍
LLVMipa	用于提供跨进程的分析编译遍

转换库包括：

LLVMInstrumentation	Instrumentation passes.
LLVMipo	用于提供所有的跨进程优化编译遍
LLVMScalarOpts	用于提供所有的标量优化编译遍
LLVMTransforms	用于提供其他未分类的转换编译遍
LLVMTransformUtils	用于提供转换的工具

代码生成库包括：

LLVMCodeGen	用于提供生成目标处理器代码的基本框架
LLVMSelectionDAG	用于提供适用于有向无循环图的积极指令选择器

运行库包括：

LLVMInterpreter	用于提供字节码解释执行器

LLVMJIT	用于提供字节码 JIT 编译器
LLVMExecutionEngine	用于提供虚拟机器的引擎

致谢

本项目及本论文是在导师付宇卓教授、谢凯年副教授的悉心指导下完成的。谢老师开展的 UADL 项目让我接触到了最新的 LLVM 编译系统，并给了我许多新的想法和思路；付老师对我在项目的进行及论文的撰写过程中给予了很多指点，让我少走了很多弯路，大大减少了工作量。没有这两位老师的帮助，本文是无论如何不能完成的。

同时也感谢所有参与“开发工具链自动生成”项目的同学，是大家的努力使得项目有了进展，并希望能将这个项目继续进行下去。

最后，我要感谢我的父母，是他们让我能全身心地投入到研究生的学习生活之中，并在我身后给予我一如既往的支持！

谢谢大家！

攻读学位期间发表的学术论文

1. 董峰，付宇卓。“ 基于 LLVM 架构的 ARM 后端移植 ”，信息技术（已录用）。
2. 马潇，谢凯年，董峰。“ 基于 MADL 语言的时钟精确级仿真器生成 ”，计算机仿真（已录用）。

LLVM编译系统结构分析与后端移植

作者：董峰
学位授予单位：上海交通大学