

学校代码：10246
学 号：23210240413

復旦大學

硕士 学位 论文

(专业学位)

面向 DSP 平台的编译器全局指令选择的实现、 优化及评估

**Implementation, Optimization and Evaluation of Global
Instruction Selection for DSP Platform Compilers**

院 系：计算与智能创新学院

专业学位类别(领域)： 计算机技术

姓 名： 周星宇

指 导 教 师： 吴俊 教授

完 成 日 期： 2026 年 1 月 10 日

指导小组成员

吴俊教授

目 录

摘要	v
Abstract	vii
第 1 章 绪论	1
1.1 研究背景与意义	1
1.1.1 DSP 架构简介	1
1.1.2 编译器研究背景	1
1.1.3 指令选择研究背景	3
1.1.4 研究意义	4
1.2 国内外研究现状	4
1.2.1 编译器研究现状	4
1.2.2 指令选择研究现状	6
1.3 论文研究内容和创新点	9
1.4 论文组织结构	10
第 2 章 面向 DSP 的全局指令选择实现	11
2.1 全局指令选择框架概述	11
2.1.1 传统指令选择方案及其局限性	11
2.1.2 全局指令选择原理及框架设计	14
2.2 GMIR 生成	15
2.2.1 通用机器表示	15
2.2.2 GMIR 生成方法	17
2.2.3 GMIR 生成实现	18
2.3 指令合法化	20
2.3.1 指令合法化方法	21
2.3.2 指令合法化实现	22
2.4 寄存器组选择	24
2.4.1 寄存器组	24
2.4.2 寄存器组选择方法	25

2.4.3 寄存器组选择实现	25
2.5 机器指令选择	29
2.5.1 匹配状态机	31
2.5.2 机器指令选择方法	32
2.5.3 机器指令选择实现	33
2.6 本章小结	34
 第 3 章 面向 DSP 的全局指令选择优化策略	 37
3.1 优化策略分析与设计	37
3.1.1 优化策略理论依据	37
3.1.2 关键优化技术分析	38
3.1.3 面向 DSP 的优化设计	40
3.2 合法化前优化策略	42
3.2.1 内存操作优化	42
3.2.2 拓展截断优化	43
3.3 合法化后优化策略	46
3.3.1 立即数装载优化	46
3.3.2 乘法优化	47
3.4 本章小结	50
 第 4 章 测试评估平台设计与实现	 51
4.1 DSP 编译器测试现状与改进	51
4.1.1 测试流程现状	51
4.1.2 测试体系优化需求与技术路径	53
4.1.3 随机测试生成技术与生成器选型	54
4.2 平台设计	55
4.2.1 平台整体设计	55
4.2.2 测试子系统设计	56
4.2.3 评估子系统设计	57
4.3 平台实现	57
4.3.1 面向 DSP 的 YARPGen 实现	57
4.3.2 测试子系统实现	59
4.3.3 评估子系统实现	61
4.4 本章小结	63

第 5 章 实验结果与性能分析	65
5.1 全局指令选择正确性验证	65
5.1.1 基本指令正确性验证	65
5.1.2 控制流指令正确性验证	68
5.2 全局指令选择性能分析	74
5.2.1 编译时间	75
5.2.2 代码尺寸	78
5.2.3 执行周期	79
5.3 测试评估平台功能展示	81
5.4 本章小结	83
第 6 章 总结与展望	85
6.1 论文总结	85
6.2 论文展望	86
参考文献	87
致谢	93

摘要

随着 DSP 架构复杂度的不断提高，编译器后端在指令选择阶段面临着更高的性能与工程化要求。GlobalISel 作为新一代指令选择框架，相较于传统基于 DAG 的指令选择机制，无论是在可扩展性上还是在编译效率上都具有优势，为此本文针对 DSP 架构的硬件及指令集特性，设计并实现了基于 GlobalISel 的指令选择优化方案。

在 GlobalISel 的实现中，对 GlobalISel 的整体框架及其关键阶段进行分析，结合 DSP 架构特性，完成了 GMIR 生成、指令合法化、寄存器组选择和机器指令选择等核心流程在 DSP 后端中的实现，并验证了其功能正确性。

针对 GlobalISel 在不同阶段产生的冗余与性能问题，本文提出并实现了多种指令选择优化策略，其中包括内存操作指令优化、常量乘法强度削弱优化以及冗余指令合并优化等，用于提升生成代码的质量和执行效率。

为了解决现有测试体系存在的问题，本文设计并实现了一套面向 DSP 编译器的测试评估平台。平台引入了随机测试生成技术，通过对 YARPGen 进行针对 DSP 架构的定制化扩展，来构建一套覆盖指令选择及后端优化场景的随机测试输入体系。同时，平台支持性能数据的自动采集、跨版本对比分析以及可视化展示等功能，为编译器优化效果评估和分析提供了支持。

实验结果表明，GlobalISel 相较于传统指令选择方案而言，在编译时间上具有一定的优势。在添加了针对 DSP 架构的优化之后，在代码尺寸和执行周期等指标上均取得了进步，这也反映了 GlobalISel 在 DSP 架构下的优化潜力。

关键词：编译器；全局指令选择；编译器优化；性能评估；LLVM

中图分类号：TP3

Abstract

Keywords: compiler; GlobalISel; compiler optimization; performance evaluation;
LLVM

CLC code: TP3

第 1 章 绪论

1.1 研究背景与意义

1.1.1 DSP 架构简介

DSP (Digital Signal Processor, 数字信号处理) 是一种用于处理数字信号的微处理器，其原理是先通过采样、量化与编码的技术手段，将连续变化的模拟信号转化为离散的数字信号，之后再借助相应算法对所得数字信号完成滤波、变换及压缩等一系列处理^[1]，DSP 的处理效率与精度直接影响终端设备的性能上限。自 20 世纪中叶诞生以来，DSP 已从早期单一功能的专用硬件电路，演进为具备可编程能力、高并行运算特性的专用处理器^[2]。近些年来，随着哈佛架构的引入^[3]、以及单指令多数据 (SIMD) 结构^[4]与超长指令字 (VLIW) 技术^[5-6]的结合，DSP 实现了从标量处理到大规模并行处理的跨越式发展。

随着 5G/6G 通信^[7]、边缘智能计算^[8]等新兴应用的爆发式增长，对 DSP 的性能提出了更为严格的要求：一方面，数据吞吐量需求呈指数级提升，需要更宽的并行运算单元与更高效的指令调度机制；另一方面，终端设备的低功耗需求日益突出，要求 DSP 在提升性能的同时实现精细化的功耗管控。在这个背景下，国内外科研机构与企业纷纷加大高性能 DSP 的研发投入，但由于核心架构设计、指令集开发等关键技术存在一定的技术壁垒，国内高性能 DSP 领域仍面临自主化程度不足的挑战，亟需具备完全自主知识产权的高性能 DSP 芯片及配套技术体系作为支撑。

本文所探讨的是一款由本实验室自主研发的高性能 DSP 芯片（在下文中简称为 DSP 芯片）。DSP 芯片基于哈佛架构设计，具备自主知识产权的指令集，通过四路并行向量处理 (VP) 单元、8 槽位 VLIW 指令发射机制及九级流水线设计，构建了高并行、高实时的运算架构；在寄存器的设计上，DSP 采用 32 个 32 位通用寄存器 GR、640 位宽向量寄存器 VR 以及专用循环访存寄存器组 MOB 等特殊寄存器的分层架构，能够适配标量与向量运算的需求。

1.1.2 编译器研究背景

编译器是一个将高级语言编写的程序转换成能在一台计算机上执行的等价目标代码或机器语言程序的软件系统^[9]。早期的计算机软件都是用汇编语言直接编写的，当人们发现为不同类型的处理器编写可重用软件的开销要明显高于编写

编译器时，高级编程语言应运而生。20世纪50年代末期，与机器无关的编程语言被首次提出。随后，人们开发了几种实验性质的编译器。第一个编译器是由Grace Hopper于1952年为A-0系统编写的。但是1957年由John Backus领导的FORTRAN^[10]则是第一个被实现出具备完整功能的编译器。1960年，COBOL^[11]成为一种较早的能在多种架构下被编译的语言。此时的编译器无标准化编译流程，每个编译器均为特定语言与硬件定制。

1964年IBM推出的PL/I语言编译器，首次实现“同时适配科学计算与商用数据处理”的多场景支持，验证了编译器的通用性潜力。1972年阿霍与乌尔曼提出的“词法分析-语法分析-语义分析-代码优化-代码生成”五阶段流程^[12]，如图1-1所示，成为全球编译器研发的标准框架；1975年UNIX系统与C语言的结合，催生了可移植编译器的研发——1978年贝尔实验室推出的C编译器^[13]，通过前后端分离的设计，首次实现同一前端解析C语言，不同后端适配不同硬件，为编译器的跨架构适配提供了范式。

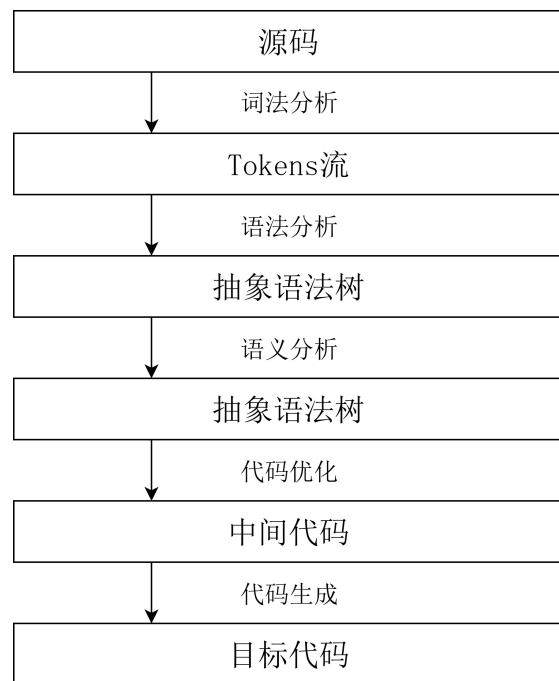


图 1-1 编译器五阶段流程图

1987年理查德·斯托曼发起GNU项目^[14]，推出GCC编译器，通过开源模式吸引全球开发者参与，逐步支持C、C++、Fortran、Java等多语言，适配x86、ARM等主流架构，成为开源生态的核心基础设施；LLVM^[15]由Chris于2000年在伊利诺伊大学创建，其模块化、可重定向的设计理念，为后续专用架构编译奠定了基础。随着GPU、DSP、AI加速器等异构架构的兴起^[16]，以及高性能计算、移动互联网等场景的需求，编译器的核心目标从通用适配转向架构专属优化，追求极致性能+低功耗的平衡。2003年NVIDIA推出CUDA^[17]平台及配套编译

器，首次实现 C 语言到 GPU 指令的编译优化，通过单指令多线程（SIMT）指令映射策略^[18]，充分挖掘 GPU 的并行计算潜力；2009 年 LLVM 2.6 版本发布，其模块化架构被广泛用于异构芯片编译，如苹果的 Clang 编译器、ARM 的商用编译器等，均通过定制 LLVM 后端实现架构专属优化。

1.1.3 指令选择研究背景

指令选择是编译器后端的核心环节，其核心任务是将编译器的中间表示 IR 转换为目标硬件支持的机器指令，其决策质量直接决定了生成代码的执行效率、资源占用及硬件适配性^[19]，是衔接编译前端语义表达与后端硬件执行的关键桥梁。从技术发展的视角来看，指令选择的发展始终围绕硬件架构复杂度提升与应用场景性能需求升级的双重驱动展开，形成了从局部适配到全局优化、从可执行性优先到能效比最优的迭代路径。

早期编译器处于基础架构搭建阶段，以支持基础编程语言和简单硬件架构为目标，核心需求是从中间表示 IR 映射到目标机器指令，确保代码可执行性，对全局优化和编译效率要求较低。在这样的需求背景下，指令选择技术以局部指令选择为核心，形成了宏拓展机制、树覆盖算法^[20]以及有向无环图（Directed Acyclic Graph, DAG）^[21]这三类主流实现方案。其中，DAG 匹配成为主流，通过将基本块内的指令表示为 DAG，寻找最优指令组合。DAG 匹配的核心思想是构建一个有向无环图，通过预定义的映射规则，实现 IR 到机器指令的精准匹配与局部优化。这种 DAG 图的优势在于能天然体现指令间的数据依赖关系，便于实现如公共子表达式消除这样的局部优化，不过其局限性也较为明显：DAG 图的构建与优化依赖局部子图分析，难以感知函数级的全局依赖关系，在 VLIW、宽位向量 DSP 等复杂架构中，易因局部最优决策导致全局资源浪费。

随着多核心、向量扩展（如 SIMD、SVE^[21]）、异构计算（GPU、加速器）等架构普及，传统局部指令选择难以适配复杂硬件的指令集特性；高性能计算、嵌入式系统等场景对代码执行效率要求严苛，局部优化的性能天花板逐渐显现，亟需全局视角的指令选择策略；大型项目编译周期长，DAG 等传统方法的编译开销成为瓶颈，需要更高效的指令映射方案。

LLVM 作为开源编译系统的主流选择，其生态需要一套能够统一多架构适配、兼顾编译速度与代码质量的指令选择框架。2019 年的 LLVM 全球开发者大会将 GlobalISel 列为核心主题，正式推动其成为传统方案的替代者。相比于 SelectionDAGISel，GlobalISel 解决了全局优化问题：以整个函数为操作粒度进行操作，保留完整的 IR 信息，拥有更大的视野，支持跨基本块的指令优化（如全局寄存器分配、跨块指令合并）；降低了编译开销：去除 DAG 中间表示，直接将 IR 转换为通用机器指令，简化指令映射流程；提升了模块化与复用性：采用

可配置的流水线架构，不同目标架构可复用基础流程，仅需定制架构相关规则，符合模块化的思想。除此之外，SelectionDAGISel 和 FastISel 截然不同，可以共享的代码非常少，而 GlobalISel 的构建方式使其能够实现代码的重用。

近年来，LLVM 社区持续推进 GlobalISel 的功能完善，针对 AArch64、RISC-V、PowerPC、AMDGPU 等主流架构完成适配，苹果、谷歌等企业也参与核心开发；从基础指令映射向深度优化演进，新增基于代价模型的指令合并、全局寄存器组选择、指令局部性优化等功能，缩小与传统方案的代码质量差距。

1.1.4 研究意义

随着数字化浪潮的持续推进，嵌入式技术已渗透到工业控制、智能终端、物联网等众多领域，尤其在数字信号处理应用场景中，DSP 处理器凭借其针对实时信号运算的专用架构设计、高并行处理能力及低功耗优势，成为自动驾驶、智能传感、通信基站等对响应速度与稳定性要求严苛的系统核心支撑。值得注意的是，即便当下 DSP 处理器的硬件算力已实现大幅提升，如何充分释放硬件潜能、通过编译优化与代码适配实现程序执行效率的精准提升，仍是尚未完全解决的关键课题。高性能硬件架构的潜力发挥取决于编译器的支撑能力，如何针对自研 DSP 的独特架构特性，设计高效的编译优化策略，提升代码执行效率并缩减代码尺寸，成为当前亟需解决的关键问题。

本研究聚焦于编译器的指令选择阶段，针对自研 DSP 架构的专用化设计需求与实时信号处理场景的高性能诉求，开展了系统性的技术研发与优化工作。首先，突破传统局部指令选择的局限，基于 GlobalISel 框架实现了函数级全局指令选择功能，解决了传统 DAG-based 方案跨基本块优化能力缺失、编译开销大等痛点，为充分挖掘硬件全局优化潜力奠定基础；其次，针对自研 DSP 架构的核心特性，定制开发了指令合并优化，在提升编译速度的同时，显著优化了生成代码的执行效率与资源利用率；最后，构建了涵盖运行时间、代码尺寸以及打包情况等多维度指标的一站式测试评估平台，实现了对优化效果的精准量化与迭代验证。该研究不仅为自研 DSP 架构提供了高效、适配性强的编译支撑，填补了专用架构编译优化工具的空白，还通过全局指令选择与架构特性的深度协同，有效释放了 DSP 硬件算力，为实时信号处理、嵌入式等领域的高性能应用开发提供了技术保障，具有重要的工程实践价值与技术推广意义。

1.2 国内外研究现状

1.2.1 编译器研究现状

编译器作为连接软件与硬件的核心枢纽，用于将源代码翻译成机器可以执行的目标代码。从技术演进脉络来看，编译器架构已从早期封闭的定制化设计，

逐步转向模块化、可扩展的开源框架，其中 LLVM 与 GCC 成为当前全球编译器研发的两大核心基石。

GCC 是第一个开源编译器，打破了传统的编译器技术格局，自 1987 年诞生以来，历经三十余年迭代，从单一 C 语言编译器成长为支持多语言、多架构的跨平台编译工具，深刻影响了 UNIX-like 系统、Linux 发行版及嵌入式领域的发展^[22]。GCC 沿用了编译器领域经典的前端—优化—后端三段式架构设计^[23]，在长期迭代中整合了常量传播、死代码消除、循环展开等一系列成熟的通用编译优化技术，能够满足多数场景下的基础编译需求。然而，受限于早期架构设计的历史惯性，该架构存在难以规避的固有缺陷：一方面，核心代码模块耦合度较高，编译流程的逻辑封装不够清晰，导致代码可读性欠佳，新开发者介入二次开发或功能调试时需花费大量时间梳理逻辑关联；另一方面，架构的拓展性不足，新增语言前端、适配新硬件指令集或集成定制化优化策略时，往往需要对现有核心代码进行大幅修改，开发成本高且易引入兼容性问题；此外，不同体系架构的适配逻辑缺乏统一的抽象层支撑，导致跨架构编译时的兼容性表现不佳，难以快速适配嵌入式、异构计算等场景下的小众架构或定制化芯片，这一问题在硬件架构日趋多元化的当下尤为突出。

2000 年 LLVM 的提出影响了编译器的发展轨迹，其突破传统编译器的紧耦合三段式架构，将编译流程拆解为独立的功能模块，各模块通过标准化接口通信，实现高度解耦，如图 1-2 所示。采用与目标平台、源语言无关的中间表示 IR，作为连接前端与后端的桥梁。IR 基于静态单赋值 (SSA)^[24] 形式设计，兼具高层语言的抽象性与底层指令的精准性，能够完整保留代码的语义信息与优化潜力。除此之外，LLVM 还将编译优化逻辑封装为独立的 Pass 单元，形成可配置、可组合的优化流水线^[25]。

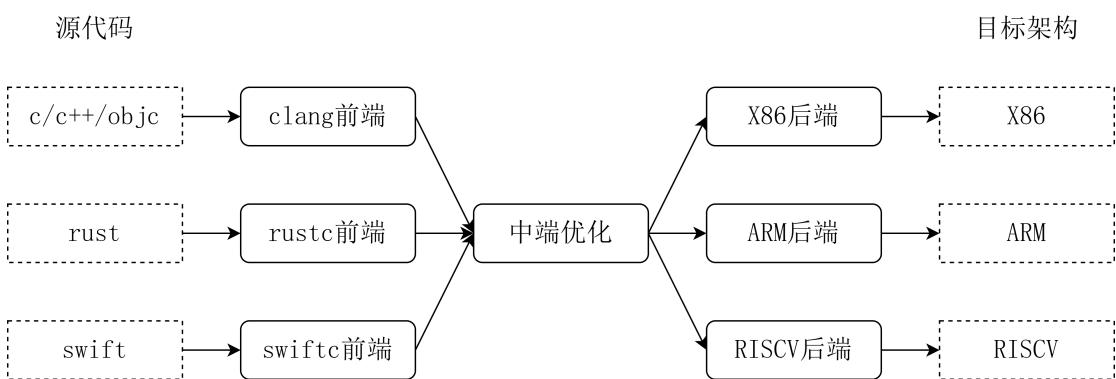


图 1-2 LLVM 架构图

依托模块化的整体架构设计、跨平台统一的中间表示以及可组合的优化机制，LLVM 已发展成为跨平台编译器实现与高性能代码生成领域的重要基础设

施。其良好的可扩展性与架构适配能力，使其被多种主流及新兴编程语言的编译器所采用，例如 Swift、TinyGo 和 Rust 等均在实现层面不同程度地依赖 LLVM 后端。除此之外，在国际上对 LLVM 的相关研究还有许多，如 Joseph N. Huber^[26]等人针对 ARM A64FX 处理器（面向高性能计算的可扩展向量架构），提出基于 LLVM 工具链的 SIMD 代码优化方法论；S Fang 等人^[27]针对 LLVM、GCC 等主流编译器自动向量化能力不足的问题，提出一套融合专用 IR 扩展 + 灵活向量化流水线的优化方案；Kavon Farvardin 等人^[28]对 SML/NJ 编译器^[29]开发了 LLVM 后端，解决了 SML 的模块系统、多态类型与 IR 的语义映射问题；Paschalis Mpeis 等人^[30]为 Android ART 开发了 LLVM 后端，解决了默认后端因优化粒度有限，难以满足交互式应用的性能需求问题。验证了 LLVM 在移动虚拟机编译器中的优化潜力；Lammich 等人^[31]提出了一种基于 LLVM 的逐步细化方法，将并行算法的验证推进至 LLVM 代码级别，通过结合 LLVM IR 和代码生成步骤，提高程序的性能；Walter Rocchia 等人^[32]提出利用 LLVM 编译器后端为 RISC-V 处理器设计新的 AI 指令集扩展，以提升边缘设备上神经网络的执行效率。Juan Carlos 等人^[33]提出一种基于遗传算法和 LLVM 优化的源代码混淆技术，旨在解决云计算等开放执行环境下的软件知识产权保护与反逆向工程问题；John Regehr 等人^[34]提出一款高吞吐量、形式化方法辅助的 LLVM 专用变异模糊测试工具，旨在高效发现 LLVM 编译器优化阶段的潜在缺陷。

随着国产芯片产业的快速发展，LLVM 由于其开源、模块化程度高等特点，逐渐被引入到多种自主研发硬件平台的工具链建设中。围绕 LLVM 的相关研究与工程实践，国内学界和工业界主要关注如何结合国产处理器架构特点进行适配与优化，并在此基础上提升编译器对底层硬件算力的利用效率。刘玉等人^[35]针对魂芯数字信号处理器的特殊架构，基于 LLVM 构建优化编译器，解决前代编译器代码质量低、硬件特性支持不足的问题。沈莉等人^[36]针对我国自主研制的神威新一代超级计算机解决异构系统可编程性差、硬件算力释放不足的问题，基于 LLVM 定制开发优化编译器；吴忧^[37]提出了基于启发式搜索的优化序列选择算法。

1.2.2 指令选择研究现状

在 LLVM 编译器生态中，有三种指令选择的实现方式：面向快速编译的快速指令选择（FastISel）、基于 DAG 图的指令选择（SelectionDAGISel）以及全局指令选择（GlobalISel）。表1-1给出了三者在设计目标、IR 形态与作用域等方面的差异。

FastISel 通常在 O0 优化等级下启用，主要面向的是需要快速编译的场景，其核心宗旨是通过牺牲生成代码的质量来提升编译速度。与传统指令选择依赖多

表 1-1 LLVM 不同指令选择框架对比

方案	设计目标	中间表示	作用域	编译速度	代码质量
FastISel	最大化编译速度	无	基本块	快	低
SelectionDAGISel	优化代码质量	DAG	基本块	慢	高
GlobalISel	平衡编译速度和代码质量	GMIR	函数	中	中

阶段中间表示处理不同，FastISel 在遍历 LLVM IR 过程中直接对指令进行处理。FastISel 通过对 IR 表达式结构进行递归访问，并结合目标架构中预定义的简化指令模式，完成指令的即时生成，从而避免了 DAG 构建及全局分析等开销较大的步骤。

SelectionDAGISel 是 LLVM 中应用最广泛的指令选择技术，自 LLVM 早期版本起就一直作为框架的核心实现。在 x86、ARM 以及 RISC-V 等主流架构中，尤其是在中高优化等级下仍是默认的指令选择方案。SelectionDAGISel 的核心思想是将中间表示转换为 SelectionDAG，并在有向无环图上进行模式匹配，以克服传统树匹配方法在表达共享子表达式方面的局限性。从研究角度来看，这种以 SSA 语义为基础、在图结构上完成匹配的模型被认为是对传统局部树结构指令选择的重要补充和发展。Ebner 等人^[38]对指令选择中的匹配模型进行了理论扩展，提出了一种基于 SSA 的 DAG 匹配方法，将匹配过程从局部的树结构提升为全局的 DAG 结构，不仅增强了模式匹配的表达能力，也提高了将机器无关中间表示映射为机器相关指令时的效率与准确性。其更早的工作也曾将指令选择形式化为 SSA 图上的图文法解析和组合优化问题，探索全函数范围上的更优覆盖与代价最小化^[39]。这些研究为更大范围、更强约束的指令选择提供了理论基础，但在工程上也带来求解复杂度与编译时间的挑战。

GlobalISel 最初在 LLVM 开发者会议上被提出，设计初衷是用于解决 SelectionDAG 在编译性能、可扩展性与复用性方面的痛点：SelectionDAG 引入了专用中间表示并伴随较高的构建和合法化开销，而不同后端在 DAG 层的大量定制也导致维护成本上升^[40]。GlobalISel 作为一种现代指令选择的替代方案已受到广泛的关注，同时它也是 AArch64 架构下 O0 优化等级的默认选择器。

除此之外，随着指令选择规则规模与复杂度增长，研究工作开始关注后端选择器相关的测试覆盖与缺陷发现，例如针对 GlobalISel 匹配表/选择路径的专门化测试与模糊测试。这些方向共同推动了指令选择从局部模式向可扩展、可验证、面向多目标优化的方向演进^[41]。

2015 年，苹果公司的 Quentin Colombet 提出：现有的指令选择框架 SelectionDAGISel 存在若干根本性局限，包括但不限于：编译速度缓慢、仅支持基本

块级别的局部作用域、架构设计过于单一化等。多年来，开发者们投入了大量精力通过增加 Target hooks 和优化 Pass 来规避这些局限，但这些方案本身也带来了新的问题（如启发式算法精度不足、需提前预测指令选择器的执行行为等）与局限。他认为，当前已具备推出新一代指令选择框架 GlobalISel 的条件，该框架将从根源上解决上述问题，同时为提升代码生成质量创造新的可能^[40]。次年，Quentin Colombet 等人分享了 GlobalISel 在设计与实现层面取得的阶段性成果，同时明确后续仍需重点研发的技术方向^[42]。

2017 年 3 月，苹果公司的 Justin Bogner 指出其团队在指令选择测试中应用模糊测试与输入生成技术的实验过程及核心成果。深入探讨了在寻找高价值测试输入过程中的技术权衡，以及验证生成代码有效性的核心方案^[43]。

2019 年，Daniel Sanders 指出目前 GlobalISel 的开发重心主要集中在对各类目标架构的基础支持上，优化相关的工作投入相对有限^[44]。近期，研发方向已转向优化能力的提升，目标是使其优化水平达到能够全面替代 SelectionDAGISel 的程度。并详细讲解 Combiner 的整体设计架构、支撑其运行的核心模块、它与 GlobalISel 其他组件的协同运作方式，以及该组件的测试和调试方法。

2021 年，Huang Zhufeng 等人提出基于代价模型的指令合并优化以及全局寄存器组选择优化以及指令局部性优化等^[45]，有效解决了传统指令选择的全局优化能力缺失、编译开销大等问题，在申威平台实现了编译速度与代码质量的平衡。

2022 年，Kai Nacke 等人分享了其为 PowerPC 目标架构实现 GlobalISel 框架的初步实践经验^[46]，详细阐述了在适配过程中面临的架构特性适配挑战、与传统 SelectionDAGISel 后端的功能衔接方案，以及达成的阶段性成果（如核心指令集的无回退匹配、基础基准测试的编译通过率提升等），为其他架构的 GlobalISel 适配工作提供可复用的实践参考。

2024 年，有关 GlobalISel 的研究骤然增长，其中 Pierre Houtryve 提出为 GlobalISel 组合器基础设施新增输入/输出 MIR 模式支持^[47]，该功能包含类 PatFrag 系统与类型推断机制，使开发者能够直接在 TableGen 中编写大量组合器规则；Tobias Stadler 提出不再生成 IR，而是直接输出 GMIR，从而跳过代码生成流水线的首个转换阶段^[48]。在作者的应用场景中，该方案使编译速度提升了约 20%；Jiahao Xie 提出面向可扩展向量的突破性 GlobalISel 的实现方案^[49]，该方案以 RISC-V 向量扩展为目标场景。演讲深入探讨了支持可扩展向量算术逻辑单元及加载/存储指令过程中面临的核心挑战与创新解决方案；Madhur Amilkanthwar 提出针对 GlobalISel 对部分指令和模式的支持不完善，导致其在该平台上需回退至传统的 SelectionDAGISel 的问题^[50]。做出的核心贡献如下：通过在 GlobalISel 各编译阶段引入补丁，消除了其回退现象；同时对 GlobalISel 生成的代码进行了

优化，显著缩小了其与 SelectionDAGISel 在 AArch64 高级 SIMD 平台上的性能差距。这些进展标志着 GlobalISel 框架的优化迈出了重要一步，使其向成为默认指令选择器的目标更近了一步。

2025 年，Nvidia 的 Neil Hickey 提出：GlobalISel 在所有后端的推广应用受到其指令选择场景覆盖不完全的限制，这导致其在部分场景下仍需回退至 SelectionDAGISel^[51]。为解决并监控这些局限性，Neil 等人开发了一套专用的持续集成系统。该系统每日会自动构建最新版本的 LLVM，通过编译一系列广泛的基准测试程序并记录回退事件，来为 LLVM 社区定位问题来源。

1.3 论文研究内容和创新点

结合前文分析可以看出，传统的 SelectionDAGISel 在实践中暴露出多方面的局限性。一方面，其实现依赖规模较大的代码体系，随着功能扩展不断膨胀，增加了开发与调试成本；另一方面，受限于 SelectionDAG/SDNode 的数据结构设计，指令选择过程需要频繁的进行内存分配与释放，不仅限制了优化策略的表达空间，也在一定程度上增加了编译时间开销。

作为 LLVM 官方推崇的替代方案，GlobalISel 在架构设计上具有更好的模块化与可扩展性，已被 AArch64、AMDGPU、RISC-V 以及 X86 等主流后端采用。然而，在大部分平台上，其当前生成代码质量仍与成熟的 SelectionDAGISel 存在差距，尤其是在目标相关指令匹配与指令组合方面，GlobalISel 仍有较大的优化空间。

现有 CI 测试流程的性能评估缺陷：DSP 工具链已具备 Gitlab 托管与提交后 CI 测试机制，在测试通过且负责人审核通过后方可合并代码，但现有测试仅反馈测试样例通过或未通过的结果，存在很多不足：缺乏 Cycles（执行周期）、Code Size（代码体积）等关键性能统计信息；无法支持不同 Commit 版本的性能对比分析；大量含 Bug 或测试性提交的无效结果混杂，不便于有效结果检索与定位。为解决上述问题，本文聚焦以下三方面研究：

1. 针对 DSP 硬件指令集以及寄存器布局等硬件特性，完成全局指令选择框架的定制化实现。通过扩展 TableGen 中的指令描述模板，补充 DSP 架构相关的指令语义与操作码映射关系，并结合 DSP 的硬件特点，设计了相应的寄存器组划分规则以及类型合法化逻辑，解决了通用 GlobalISel 在嵌入式 DSP 平台上适配性不足的问题。在此基础上，构建了适用于 DSP 后端的指令选择基础框架，替代原有的 SelectionDAGISel，为后续指令选择优化与功能扩展提供了更具可维护性的实现基础。
2. 针对 DSP 硬件特性（如硬件指令集、寄存器布局等），对全局指令选择机

制进行针对性优化，提升生成代码质量。

3. 设计并实现与 Gitlab CI 深度结合的性能测试评估平台：在代码合并后，系统会自动调用脚本来启动模拟器和芯片端上的测试任务，在测试结束后将测试数据传到 Perf 仓库；后端服务通过设置定时任务来拉取 Perf 仓库的数据并更新数据库；前端则负责用图表展示性能数据。这套方案不但支持多版本之间的性能对标，还能够清晰地捕捉到每一次提交导致的性能提升或退步，实现了优化效果的精准量化。

1.4 论文组织结构

本文一共分为六个章节，针对全局指令选择的实现与优化以及性能评估平台的实现来进行研究，论文结构如下：

本文一共分为六个章节，围绕全局指令选择的实现与优化方法以及性能评估平台的设计与实现展开研究，论文整体结构安排如下。

第一章为绪论部分，介绍了论文的研究背景与意义，重点分析了 DSP 架构与编译器技术的发展现状，梳理了指令选择技术，尤其是全局指令选择相关研究的国内外研究现状与存在的问题，在此基础上明确了本文的研究内容、技术路线和主要创新点。

第二章为全局指令选择框架的实现部分。本章先对传统指令选择方案及其局限性进行了分析，并给出全局指令选择的基本概念和设计思路，之后分节详细阐述了全局指令选择的 GMIR 的生成、指令合法化、寄存器组选择以及机器指令选择这四个核心阶段的设计与实现。

第三章为全局指令选择的优化策略的实现部分。本章先结合 DSP 架构特点从理论角度分析了全局指令选择优化的必要性和可行性，之后针对内存操作指令、乘法指令等典型应用场景，提出并实现了多种优化策略，提升了生成代码的执行效率和质量。

第四章为测试评估平台的设计与实现部分。本章根据现有 DSP 编译器测试流程中存在的问题，确定了测试体系优化的需求与技术路径，之后从系统架构层面给出了两个子系统的整体设计方案，并在此基础上实现。

第五章为实验和展示部分。实验部分从编译时间、代码尺寸和执行周期等多个维度来分别对全局指令选择实现的正确性及优化的有效性进行了验证，展示部分则通过运行截图来直观地展示测试评估平台的实现情况。

第六章为总结与展望部分。本章对全文的研究工作进行了系统的总结，归纳了本文在全局指令选择实现、优化策略以及测试评估平台方面取得的主要成果，并对存在的不足与未来改进方向进行讨论。

第 2 章 面向 DSP 的全局指令选择实现

2.1 全局指令选择框架概述

编译器后端指令选择阶段的任务是将 LLVM IR (Intermediate Representation, 中间表示) 转换为目标架构 MI (Machine Instruction, 机器指令)。指令选择阶段的核心目标是在保证程序语义等价的基础上，生成符合目标架构约束且性能尽可能优的指令序列。该阶段直接影响生成代码的质量、资源利用率以及编译器对复杂硬件架构的适配能力，是连接前端与后端的重要桥梁。

随着处理器架构的不断演进以及 LLVM 编译器基础设施的持续发展，指令选择的实现方式也在不断演化。为适应不同应用场景和性能需求，LLVM 先后发展并引入了快速指令选择、基于有向无环图的指令选择以及全局指令选择等三种主流技术路线。这些方案在设计理念、适用场景和生成代码质量等方面各有侧重，共同构成了 LLVM 指令选择技术体系的发展脉络。

2.1.1 传统指令选择方案及其局限性

在 LLVM 的早期发展阶段，指令选择的目标是保证 LLVM IR 能够正确、稳定地映射为目标架构支持的机器指令，并在开启优化选项时生成质量较高的代码。为此，LLVM 采用了基于 DAG 的指令选择方案。该方案是 LLVM 长期以来的主流指令选择方案，也是优化模式下的默认选择，其核心定位是代码质量优先，支持复杂架构指令集，能够处理 X86、ARM 以及 RISC-V 等各类架构的复杂指令选择需求。

SelectionDAG 指令选择方案通过将基本块内的 LLVM IR 转换为 SelectionDAG 结构，在此基础上完成指令合法化 (Legalize)、节点组合 (Combine) 以及指令匹配与替换等阶段性处理，从而实现从中间表示到目标机器指令的映射过程。其整体处理流程如图2-1所示。

SelectionDAGISel 的设计思想是将 LLVM IR 转换为 DAG 来表示指令间的数据流依赖关系，之后通过模式匹配与树覆盖的算法将 DAG 节点映射为目标架构的机器指令。具体流程可分为三个阶段：第一阶段，框架将 LLVM IR 指令序列转换为 SelectionDAG。在 DAG 图中，每个节点代表一个操作，边代表数据依

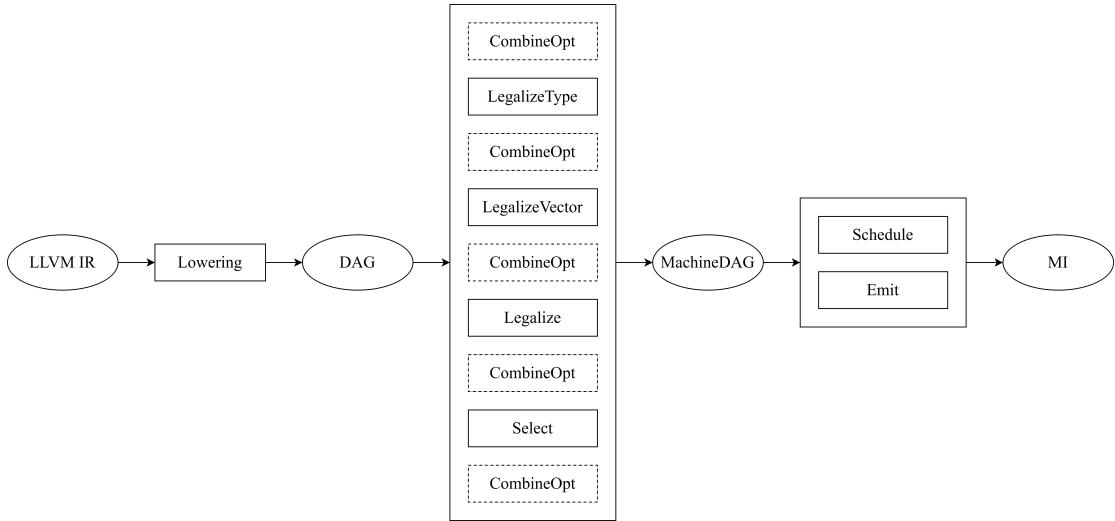


图 2-1 SelectionDAGISel 流程图

赖，这种结构能清晰体现指令间的数据流关系，从而避免了线性扫描的局限性；第二阶段，为了简化后续的匹配流程，框架对 SelectionDAG 进行标准化操作及优化；第三阶段，框架采用基于 TableGen 自动生成的模式匹配器，对优化后的 SelectionDAG 进行拓扑排序，通过自底向上遍历节点来匹配目标架构在 td 文件中定义的 DAG 模式，之后将匹配成功的节点替换为对应的机器指令，最终生成目标架构的机器代码。图2-2展示了一个简单自增运算函数在指令选择阶段生成、调度之前的部分 SelectionDAG 图。

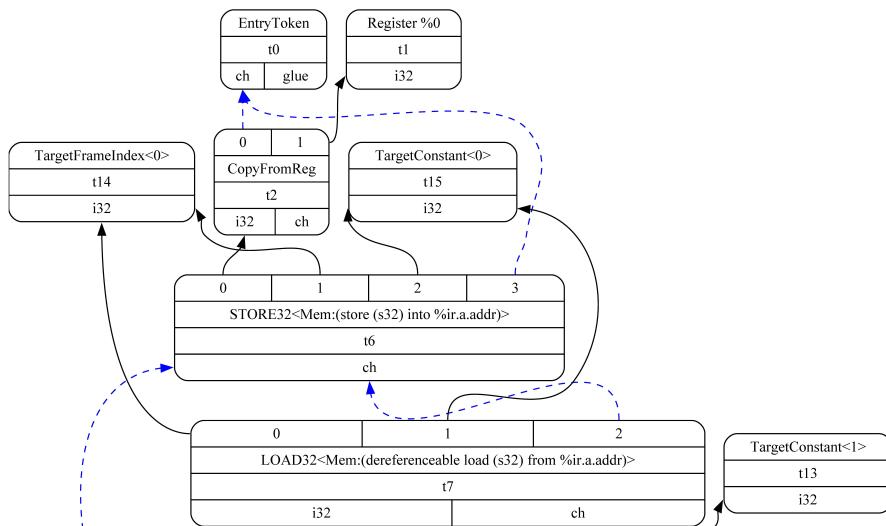


图 2-2 自增运算函数在指令选择阶段生成的部分 SelectionDAG 示例

通过构建 SelectionDAG，基于 DAG 的指令选择能够实现对全局数据流的深度分析，因此其能够进行复杂的指令优化与组合，并显著提升生成代码的质量。除此之外，开发者仅需在描述文件中定义指令模式即可自动生成匹配代码，这会大幅降低了新架构适配的开发成本。但该指令选择方案的设计复杂度很高，这

是因为 SelectionDAG 的构建、优化和拓扑排序等过程涉及到了大量复杂的逻辑，并且 DAG 作为一种专用中间表示，其与 LLVM IR 和机器指令的衔接需要额外的转换层。

随着 LLVM 编译器生态的持续迭代与发展，其凭借模块化的设计与良好的跨平台兼容性，已经在各类大规模工业级项目开发及复杂程序调试场景中得到广泛的应用。在这样的背景下，编译速度在低优化等级下成为关键诉求。为了满足项目调试构建阶段对快速编译的需求，LLVM 在原有的 SelectionDAG 框架之外引入了 FastISel 框架作为补充路径。FastISel 是 LLVM 为平衡编译速度与代码质量设计的轻量级指令选择方案，FastISel 的核心定位是优先考虑编译速度，同时兼顾基础指令选择需求，其主要用于 O0 优化等级下的代码生成。

从设计原理来看，快速指令选择采用线性扫描 + 模式匹配的策略，以单个指令为处理单位，不构建复杂的中间表示结构。其核心流程为：遍历 LLVM IR 指令序列，对每条指令直接匹配目标架构的简单指令模式，匹配成功则直接生成机器指令，匹配失败则回退到基于 DAG 的指令选择。该流程如图2-3所示。

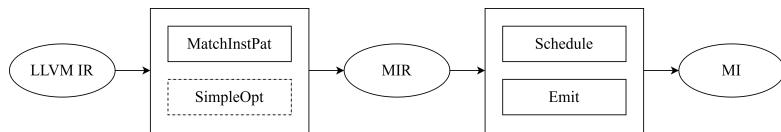


图 2-3 FastISel 流程图

为了提高编译速度，快速指令选择采用手写 C++ 代码的方式来实现匹配规则。这种方式能够有效的规避 TableGen 工具链的解析与代码生成产生的开销，同时省去了指令间的依赖分析以及复杂的优化逻辑，从而显著缩短编译耗时。这种设计使得快速指令选择的编译效率极高，在调试模式下能显著缩短编译时间，但 FastISel 也存在明显局限性，由于其仅支持简单的指令模式匹配，无法处理如向量运算、指令融合这样复杂的指令组合，这会导致生成的代码质量较低。除此之外，由于 FastISel 缺乏全局视角，导致其无法优化跨指令的数据流和控制流，在性能敏感的场景下难以满足需求。因此，快速指令选择通常仅仅作为调试模式的默认选择，主要用于调试和快速编译的场景，在性能敏感的优化模式下仍需依赖更强的指令选择机制。

随着编译器应用场景的持续扩展，传统指令选择方案无论是在编译效率，还是架构适配灵活性方面都难以适配新需求。具体而言，SelectionDAGISel 的专用中间表示及其配套优化机制增加了编译流程复杂度及维护成本，并且受限于其局部化的视角与寄存器约束的处理逻辑，SelectionDAGISel 在跨基本块优化、复杂寄存器组绑定等复杂场景中显现出明显的不足。这些问题共同推动了新一代指令选择框架的提出与发展。

2.1.2 全局指令选择原理及框架设计

全局指令选择是 LLVM 为解决基于 DAG 的指令选择的局限性而设计的新一代指令选择框架，其核心目标是兼顾编译速度、代码质量和架构适配灵活性，同时统一 LLVM IR 到机器指令的转换流程。

从设计理念来看，全局指令选择摒弃了基于 DAG 的专用中间表示，转而使用 GMIR（Generic Machine Intermediate Representation，通用机器中间表示）作为其核心载体。GMIR 本质上是对 LLVM IR 的轻量级扩展，在保留了 LLVM IR 的语义特征的同时引入了目标架构的基础信息，这使得在指令选择的过程无需在多种中间表示间进行频繁转换，从而简化了整体流程。

GlobalISel 的核心优势在于全局性和灵活性。全局性体现在 GlobalISel 以函数为粒度，通过结合控制流和数据流的全局视角来进行指令选择，因此 GlobalISel 能优化跨基本块的指令序列。灵活性则体现在 GlobalISel 支持手动和自动两种模式匹配方式，自动模式下除了可以基于 TableGen 匹配，还能重用基于 DAG 的指令选择的部分指令集描述信息，这会显著降低前期的开发成本。同时还可以通过自定义代码来补充 TableGen 无法描述的复杂指令，这些特性降低了架构适配的成本。

全局指令选择的执行流程分为两个核心阶段：第一阶段为 GMIR 预处理，包括将 LLVM IR 转换为原始 GMIR、指令合法化、寄存器类型选择，最终生成附着目标架构信息的合法 GMIR；第二阶段为机器指令选择，以函数为粒度，采用基于树覆盖的指令选择算法，通过自动匹配模块和手动匹配模块，将 GMIR 指令树映射为 MIR（Machine Intermediate Representation，目标架构的机器中间表示），同时将通用虚拟寄存器限制到具体的寄存器类中，并传递 COPY 指令至后续的寄存器分配阶段。

全局指令选择将两个阶段涉及的每个功能都拆分成独立的 Pass，这样既能使整体的架构更加清晰，也能够方便利用 LLVM Pass 的相关基础设施进行代码维护和问题分析定位。LLVM 实现中将全局指令选择所必需的基本功能划分为 4 个 Pass，其中第一阶段包含 3 个 Pass，而第二阶段为 1 个 Pass。还有窥孔类的优化也会以独立 Pass 进行实现，并可以放置在 4 个基础 Pass 之间的任意位置，不同的架构可以根据需要配置一到多个这种优化 Pass。该流程如图2-4所示。

相比于前两种指令选择方案，GlobalISel 实现了在编译效率、代码质量和架构适配性之间的多维度平衡。从编译速度上来看，GlobalISel 摆弃了 SelectionDAG 中复杂的处理流程，编译效率接近快速指令选择；从代码质量上来看，GlobalISel 的全局视角和复杂模式匹配能力能生成与基于 DAG 的指令选择相当的代码；从架构适配性上来看，GlobalISel 的模块化设计使其易于扩展和维护，由于其具有通用的 GMIR 表示和灵活的模式匹配机制，这使得 GlobalISel 能更便捷地适配

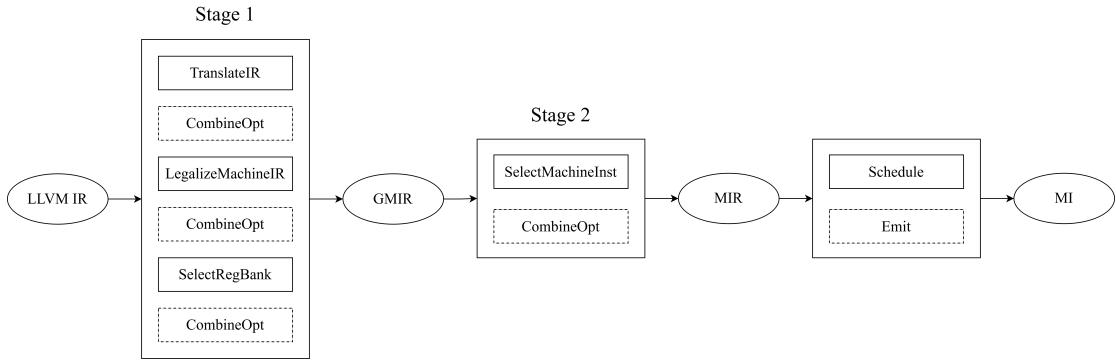


图 2-4 GlobalISel 流程图

新的架构，尤其是具有特殊寄存器组和向量指令的架构。

2.2 GMIR 生成

GMIR 的生成是 GlobalISel 第一阶段的第一个 Pass，也称为 IRTTranslator。IRTranslator 阶段的任务是完成 LLVM IR 到 GMIR 的语义下沉与架构适配，核心是要完成寄存器处理、栈管理以及调用约定处理等操作。IRTranslator 的实现以 DSP 目标平台的 ABI (Application Binary Interface，应用二进制接口) 约定、指令集特性及数据类型支持为核心依据，通过模块化的调用处理机制，确保函数调用、参数传递、返回值处理的正确性与高效性。

2.2.1 通用机器表示

GMIR 是 GlobalISel 框架的核心中间层表示，在 LLVM IR 与目标架构强相关的 MIR 之间起到承上启下的关键作用。GMIR 的设计目标是在保留通用机器语义的同时，延迟引入具体架构细节。这样做好处是可以为跨架构的指令选择、合法化以及后续的优化提供统一而稳定的基础。通过这一设计，GlobalISel 能够在更大作用域内进行分析与决策，避免过早绑定目标架构而减少优化空间。GMIR 由四个要素组成：

1. 通用虚拟寄存器 (Generic Virtual Register)：不绑定具体物理寄存器，由 MachineRegisterInfo 统一管理生命周期，是 GMIR 中数据传递的核心载体，如函数参数、运算结果均存储于虚拟寄存器。
2. 通用指令集 (Generic Instructions)：覆盖算术运算、逻辑运算、内存访问、分支跳转、函数调用与返回等基础操作，指令格式标准化，避免架构专属指令的碎片化。
3. LLT (Low-Level Type，低级别类型)：基于位宽和类型类别定义 (如标量、

指针、向量)，摒弃了 LLVM IR 的高级类型（如结构体、类），更贴近硬件数据处理逻辑。

4. 架构约束元数据：通过 MachineMemOperand（内存访问属性）、RegMask（寄存器掩码）、CCValAssign（调用约定分配信息）等来系统化地表示目标架构约束，从而为后续合法化与指令选择提供依据。

在 GlobalISel的整体流程中，GMIR 贯穿指令选择前后的多个关键阶段。首先，IRTranslator 以函数为单位遍历 LLVM IR，将其直接映射为 GMIR 的通用指令和虚拟寄存器表示，例如将 LLVM IR 中的 call 指令转换为 GMIR 的 G_CALL 指令并生成相应的参数传递逻辑。在这一阶段，编译器可以结合目标架构进行必要的特化处理。比如 DSP 架构会对 64 位浮点运算进行拆分。随后进入合法化阶段，框架会对 GMIR 中目标架构不直接支持的指令及参数类型进行修正，确保其完全落入硬件能力范围。在寄存器组选择阶段，框架将通用虚拟寄存器映射到目标架构定义的寄存器组中，为后续的物理寄存器分配奠定基础。最后，在指令选择阶段，框架会将 GMIR 的通用指令替换为目标架构的机器指令，最终生成与目标硬件紧密绑定的 MIR 表示。为进一步明确 GMIR 在 LLVM 编译流程中的抽象层级及其过渡性定位，表2-1对 LLVM IR、GMIR 与 MIR 在抽象层级、核心元素及架构相关性等方面进行了对比。

表 2-1 LLVM 不同中间表示层次对比

类型	抽象层级	核心元素	架构相关性	核心作用
IR	高级语言无关抽象 (函数、模块级)	函数、基本块、 Value、LLVM IR 指令	完全无关	跨平台程序分析与 中端优化
GMIR	通用机器语义抽象 (指令、寄存器级)	通用虚拟寄存器、通用指令、LLT 类型	部分相关	衔接 LLVM IR 与架构相关 MIR，统一 指令选择流程
MIR	目标架构专属抽象 (硬件指令、物理寄存器级)	物理寄存器、目标架构指令、硬件约束	完全相关（绑定具体硬件）	目标架构指令调度 与寄存器分配

与传统 LLVM IR 到 SelectionDAG 的映射流程相比，GlobalISel 的 LLVM IR 到 GMIR 的转换路径在流程简化、语义完整性以及优化潜力等方面都具有明显优势。GlobalISel 通过直接生成 GMIR，避免了 SelectionDAG 流程中 LLVM IR、SelectionDAG 和 MachineDAG 之间的多次转换，从而显著降低了编译流程的复杂度和开销。同时，GMIR 以函数为基本单元来进行构建，这能完整保留 LLVM IR 中跨基本块的控制流和数据依赖信息，为全局优化提供了基础。由于通用虚

拟寄存器能够跨基本块存在，并且 LLT 对异构类型系统具有良好的适配能力，GMIR 得以在更大作用域内发挥作用，为跨基本块的指令合并、全局寄存器分配等高级优化提供支撑，从而展现出更为显著的整体优化潜力。

2.2.2 GMIR 生成方法

GMIR 的指令生成逻辑分为架构无关与架构相关两大类别，这样做的好处是既能保留通用优化的灵活性，又能通过架构特化逻辑适配不同硬件的底层约束。其中，架构无关指令涵盖算术运算、逻辑运算等基础操作。由于这类指令仅描述通用机器语义，并不依赖具体硬件特性，于是可以由 GlobalISel 框架统一生成，无需目标架构单独开发适配代码；而架构相关指令则聚焦于与硬件约束强绑定的场景，如函数调用、形式参数处理等。由于架构相关指令依赖目标架构的 ABI 调用约定（如寄存器分配规则、栈帧布局、参数传递方式等），因此必须结合具体架构特性进行定制化的实现。

IRTranslator 是 LLVM IR 到 GMIR 的翻译器，负责整体指令翻译流程的调度。在 GlobalISel 框架中，IRTranslator 以函数为基本处理单元，遍历 LLVM IR 中的各类指令，并将指令逐步转换为对应的通用机器指令表示，从而完成从高级中间表示到通用机器语义的转换。

在翻译过程中，IRTranslator 的模块化设计使其能避免直接处理与目标架构强相关的细节，将函数调用等依赖 ABI 约定的任务交给目标后端来实现。其中，CallLowering 是 IRTranslator 的关键组件，负责参数传递、返回值处理以及调用约定的具体实现。IRTranslator 在处理函数入口、调用指令和返回指令时，通过统一接口来调用目标后端提供的 CallLowering，从而实现通用翻译框架与目标架构定制逻辑之间的解耦。

基于这种分层协作机制，GlobalISel 能够在保持 IRTranslator 通用性的同时，还能实现对不同目标架构 ABI 差异的良好适配，从而实现通用指令翻译逻辑与目标架构定制逻辑之间的有效解耦。IRTranslator 与 CallLowering 之间的关系及协作方式如图2-5所示。

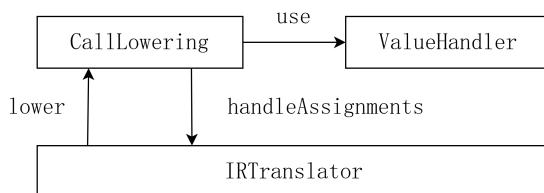


图 2-5 GMIR 生成流程图

以函数调用约定处理为例，在函数调用的过程中需要将 LLVM IR 中抽象的函数调用、参数传递语义，转换为符合目标架构规范的 GMIR。由于不同架构的

寄存器布局、栈对齐要求、参数存储优先级（寄存器优先或栈优先）存在显著差异，因此需要由各目标架构单独实现专属的 CallLowering 模块，模块通过解析自身的 ABI 约定来完成 LLVM IR 到 GMIR 的精准映射，从而确保后续指令选择与硬件执行的兼容性。

GMIR 生成的执行过程是以函数为粒度进行的，与 SelectionDAGISel 是以基本块为粒度不同，由于函数可以分为包含形参信息的函数头与函数体，函数体又有基本块等表示，所以按处理的函数信息不同，将执行过程分为 4 个主要的阶段，如图2-6所示。

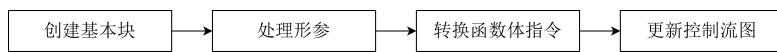


图 2-6 GMIR 生成流程图

1. 创建基本块：这个阶段会遍历函数的 LLVM IR 基本块，依次为每个基本块创建对应的 GMIR 的基本块，并且会保留相关的控制流信息，形成初始的控制流图。还会为每个函数添加一个额外的基本块（EntryBB），作为函数入口。
2. 处理形参：根据目标架构的调用约定规则处理函数的入参，为每个人参生成一条从传参寄存器复制到虚拟寄存器的 GMIR 复制指令，或者为入参生成一条从栈上加载到虚拟寄存器的 GMIR 加载指令，并将生成的指令放入 EntryBB 中。
3. 转换函数体指令：按 RPOT (Reverse Post-Order Traversal，逆后序遍历) 的方式遍历函数的控制流图，对于每个基本块，以自顶向下的顺序将基本块里的每条指令转换成一组 GMIR 指令。
4. 更新控制流图：在第 3 阶段的指令转换过程中，有些原本不是跳转的指令会被翻译成跳转指令（例如，跳转指令的条件码是由多条连续的逻辑运算指令生成的，此时就有可能会拆分逻辑运算生成多个跳转指令），导致原有基本块被拆分出多个新基本块，破坏原有的控制流。因此在基本块指令转换好后，需要维护好这些新基本块与原有基本块的控制流边，形成新的控制流图。此外，转换过程还可能会将 EntryBB 和函数体中的第一个基本块合并，此时控制流信息也要跟着更新。

2.2.3 GMIR 生成实现

按照 2.2.2 节的设计可知，GMIR 的指令生成逻辑主要分为架构无关和架构相关两类，算术运算、逻辑运算等架构无关指令的处理由框架统一实现，而与

DSP 架构强绑定的函数调用、形参传递、返回值处理等架构相关指令，则需基于 DSP 的 ABI 规范、寄存器布局及数据传输规则来对 CallLowering 模块进行定制化开发。对于 DSP 架构而言，上述场景在寄存器宽度受限、数据类型支持不完整以及端序特性等方面均存在显著差异，因此成为 IRTTranslator 实现中的重点与难点。

为适配 DSP 的架构特性，本文基于 ValueHandler 的接口实现了 DSPIncomingValueHandler 与 DSPOutgoingValueHandler 两个辅助类。这两个辅助类分别负责函数入参出参的传递与返回值的接收，其核心结构如图2-7所示。这一设计将参数与返回值的处理逻辑从 IRTTranslator 的主流程中解耦出来，使调用相关的架构特化代码能够集中管理，提升了整体实现的可维护性与可扩展性。

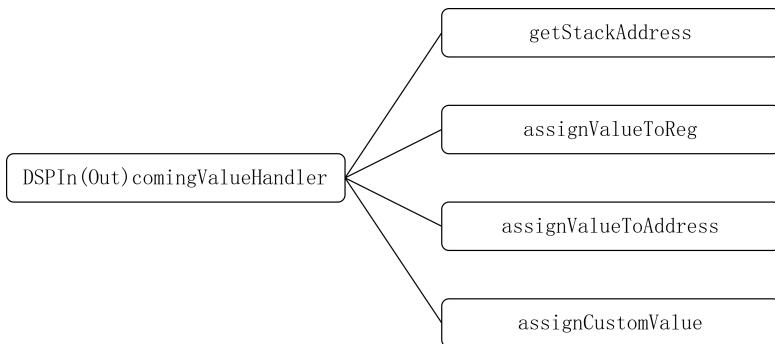


图 2-7 DSPValueHandler 核心函数

1. `getStackAddress` 用于生成 DSP 架构下栈帧空间的目标地址虚拟寄存器，根据上下文选择基于帧索引还是直接操作栈指针，为出参的栈上存储提供地址支持。
2. `assignValueToReg` 用于将目标值分配至物理寄存器，核心完成 COPY 指令构建，并按需标记虚拟寄存器的状态属性。
3. `assignValueToAddress` 用于完成栈上数据的加载和存储，将其加载或存储在合适的位置并在需要时进行扩展。
4. `assignCustomValue` 是针对 DSP 架构的核心定制化实现，通过数据的合并与拆分逻辑，解决了非法类型的跨函数传输问题。具体而言，函数根据 DSP 的端序特性调整高低位数据的存储顺序，确保跨函数数据传输的一致性。除此之外，函数通过 G_COPY 指令将拆分后的数据映射至目标物理寄存器，并支持 Thunk 回调机制来实现复制逻辑的延迟执行，从而提升了编译流程的灵活性。

在具体的实现中，IRTranslator 需要分别完成函数调用、形式参数以及返回值的降级处理等操作。其中，以 lowerCall（函数调用的降级过程）为例，其主要流程可概括为以下几个阶段：

1. 调用合法性校验：约束调用约定（如 DSP 仅支持 C 语言调用约定），确保跨函数调用的二进制兼容性；校验入参和返回值类型是否为架构支持的类型，提前拦截非法调用以避免后续流程异常。
2. 栈帧预处理与调用指令构建：lowerCall 通过 DSP::ADJCALLSTACKDOWN 指令创建栈帧调整起始指令，为调用过程预留栈空间；根据被调用者类型动态选择指令：寄存器寻址或 PIC 场景使用间接调用的 DSP::CALLR 指令，其余场景使用直接调用的 DSP::CALL 指令，并标记栈指针为隐式定义，添加调用保存寄存器掩码，确保调用过程中寄存器状态的一致性；添加被调用者标识至调用指令，为后续指令选择提供目标地址信息。
3. 出参传递与栈空间校准：lowerCall 将参数按 DSP 数据类型规则拆分，生成适配架构的 ArgInfo 列表，并通过 DSPCCState 结合 ABI 约定，解析参数的寄存器/栈分配策略，由 DSPOutgoingValueAssigner 完成分配计算及参数传递。最后获取 DSP 架构栈对齐要求，将栈空间大小按对齐要求校准，确保栈访问符合硬件约束。
4. 调用指令提交与间接调用约束：lowerCall 将构建完成的调用指令（CALL/-CALLR）插入当前基本块，若为 CALLR 指令，通过 constrainAllUses 结合 DSP 的 RegisterBank 信息及目标指令信息，约束指令操作数的合法性，避免跨银行访问或非法寄存器使用。
5. 返回值处理与栈帧恢复：如果返回值非空，lowerCall 则通过 DSPIncomingValueAssigner 解析返回值分配规则，将物理返回寄存器中的值映射至通用虚拟寄存器，并通过 DSP::ADJCALLSTACKUP 指令恢复栈指针，释放调用过程占用的栈空间，从而完成整个调用流程。

通过上述实现，IRTranslator 在 DSP 架构下实现了对函数调用、参数传递与返回值处理的完整支持，有效解决了通用调用降低逻辑在寄存器宽度受限和非法类型处理方面的不足。该实现不仅保证了 GMIR 生成阶段的语义正确性，也为后续指令合法化与指令选择阶段提供了结构清晰、约束明确的中间表示基础。

2.3 指令合法化

在经过 IRTranslator 阶段的转换后，以 LLVM IR 表示的函数已经转变成了 GMIR 表示的函数。这一转换并非简单的语法映射，而是伴随语义适配的深度处

理。为了衔接后续架构适配流程，IRTranslator 会在转换过程中嵌入部分目标架构的如寄存器组初步约束、调用约定相关标记等基础信息。但从设计本质来看，GMIR 作为通用机器中间表示的核心属性并未改变，转换生成的绝大多数 GMIR 指令仍保持架构无关性，这种设计虽保障了跨架构复用性，但这也必然导致一个核心问题：部分 GMIR 指令可能超出目标架构的硬件能力范围，成为非法指令，即有部分 GMIR 指令会存在目标架构不支持的情况。

非法 GMIR 指令的产生源于架构能力的固有差异，最典型的场景便是数据位宽不匹配。例如，针对 16 位字长的嵌入式架构，其硬件指令集仅支持 16 位或 32 位运算，若 IRTranslator 因 LLVM IR 中原有的 64 位数据运算生成了 64 位的 G_ADD 指令，该指令便会因超出硬件处理能力而成为非法指令。类似地，部分架构不支持向量运算或特定类型的内存访问，对应的 GMIR 向量指令、特殊寻址模式指令也会被判定为非法。

为解决非法 GMIR 指令的问题，框架需要实现一个独立的合法化 Pass，这个 Pass 是衔接 GMIR 与目标架构特性的关键枢纽。该 Pass 的工作原理是首先加载目标架构的完整指令集描述信息，涵盖其支持的数据类型、运算操作以及寄存器约束等关键元数据，随后以函数为单位遍历 GMIR 代码，将其中所有非法指令逐一替换为语义等价的合法 GMIR 指令序列，从而使生成的指令组合能够完全适配目标架构的硬件能力。

2.3.1 指令合法化方法

合法化过程以函数为基本粒度，采用结构化遍历的执行逻辑，从而确保指令转换结果的完整性与正确性。在遍历策略上，Pass 采用 RPOT 算法自函数入口开始依次扫描所有基本块，该遍历顺序能够保证父基本块中的指令优先于子基本块被处理，从而有效避免由控制流分支依赖引发的转换冲突。在进入单个基本块之后，指令按照自顶向下的顺序逐条处理，对每一条 GMIR 指令进行合法性检查，一旦发现非法指令，立即触发相应的转换逻辑。待当前基本块内所有指令均完成合法化后，再继续处理下一个基本块，直至整个函数范围内的 GMIR 指令全部完成合法化。

指令合法化的处理过程实际上包含了两个关键子问题的处理，二者共同构成了指令合法化的完整技术链路：

1. 非法指令识别：Pass 通过查询目标架构提供的 LegalizerInfo 接口来进行判断。一方面需要判断校验指令操作码与数据类型的组合是否支持，如 G_ADD 指令是否支持 64 位数据。另一方面需要检查指令的操作数约束是否符合架构的要求。例如，针对 ARM 架构的 LegalizerInfo 会明确标记 G_FADD（浮点加法）仅支持 32 位和 64 位，若 GMIR 中出现 16 位浮点加

法指令，则会被Pass直接判定为非法。此外，部分架构对指令的隐含约束也会纳入识别逻辑，用于确保识别结果的精准性。

2. 非法指令转换：当检测到非法指令时，Pass需要通过指令重构来生成一组语义不变且实现架构兼容合法指令序列。由于非法指令产生原因不同，转换策略通常可分为两类。针对数据位宽不匹配的情形，采用拆分与重组相结合的方法，例如将64位的G_ADD运算拆解为两个32位G_ADD指令，并通过进位标志的传递实现语义等价的计算。对于目标架构不支持的操作类型，如在缺乏硬件除法单元的体系结构上遇到G_SDIV指令，则通过算法模拟的方式加以替代，利用减法、移位等基础指令组合实现相同的除法语义。除此之外，在整个转换过程中还必须同步完成虚拟寄存器类型的调整以及指令间依赖关系的维护，从而确保重构后的指令序列在数据流和执行效果上与原始指令保持一致。

在上述两个子问题得到解决后，合法化Pass最终实现了GMIR从通用到架构适配的关键转换。这为后续的寄存器组选择以及目标指令生成等流程提供了合法且可靠的输入基础。

2.3.2 指令合法化实现

根据2.3.1的设计，合法化阶段的目标是非法指令的识别与非法指令的转化。在具体实现上，合法化流程首先需要明确目标架构所支持的LLT类型集合，该集合不仅包括各类数据类型，还包括地址类型与内存访问相关的类型信息。上述类型集合构成了指令合法性判定的基础边界，用以描述目标架构在位宽支持、对齐约束以及寻址能力等方面的硬件限制。

在此基础上，需要构建一组合法化规则集，用于对GMIR指令进行系统化的合法性判定与处理策略选择。LegalityQuery对象是规则集的核心载体，该对象通过结构化封装的方式，将合法性判定所需的关键信息统一组织为标准数据结构。其中包括指令操作码、各操作数索引对应的LLT类型、MachineMemOperand中描述的内存访问字节大小，以及针对内存类指令的原子性与顺序约束等信息。DSP架构的部分合法化规则集如下所示。

```

1 DSPLegalizerInfo :: DSPLegalizerInfo(const DSPSubtarget &ST)
2   : STI(ST), XLen(STI.getXLen()), SXLen(LLT::scalar(XLen)) {
3     const LLT S1 = LLT::scalar(1);
4     const LLT S8 = LLT::scalar(8);
5     const LLT S16 = LLT::scalar(16);
6     const LLT S32 = LLT::scalar(32);
7     const LLT S64 = LLT::scalar(64);
8     const LLT P0 = LLT::pointer(0, 32);

```

```

9   const LLT SDoubleXLen = LLT::scalar(2 * XLen);
10  .....
11
12  auto &ShiftActions = getActionDefinitionsBuilder({G_ASHR,G_LSHR,G_SHL});
13  if (ST.is64Bit())
14    ShiftActions.customFor({{S32, S32}});
15  ShiftActions.legalFor({{S32, S32}, {S32, SXLen}, {SXLen, SXLen}})
16    .widenScalarToNextPow2(0)
17    .clampScalar(1, S32, SXLen)
18    .clampScalar(0, S32, SXLen)
19    .minScalarSameAs(1, 0);
20  .....

```

这种设计在工程实践中具有明显的优势。一方面，其他编译 Pass 可直接基于 LegalityQuery 对象查询指令合法性，而无需重复解析指令内部结构，从而有效地降低实现复杂度。另一方面，合法性判定过程不依赖于实际指令实例的构造，这样做的好处是能够避免额外的对象创建与析构开销，在提升编译效率的同时，也使 LegalityQuery 对象能够直接作为谓词约束的输入参数，支撑更为复杂和精细的合法性判断逻辑。

合法化规则集的生成与执行遵循统一的标准流程。通过调用 getActionDefinitionsBuilder 接口，可为指定操作码构建专属的合法化规则集。当输入多个操作码时，该接口会自动将其绑定至同一规则集中，适用于操作语义相近、合法化策略一致的指令类型。规则集中的各条规则按照自上而下的优先级顺序依次匹配，当某条指令成功匹配并完成合法性判定后，其后续处理即告结束，下一条指令将重新从规则集起始位置开始匹配，从而保证每条指令都能获得唯一且确定的合法化结果。

如果某条指令未匹配到任何规则，则会被直接判定为合法化失败。当规则集中未显式声明指令为合法时，系统将自动触发类型扩展、拆分或重写等数据类型转换逻辑，为后续指令选择阶段提供可执行的中间表示基础。通过这种机制，合法化阶段能够在保持规则清晰性的同时，兼顾灵活性与健壮性。

对于一些合法化逻辑复杂的指令，框架提供的标准化的规则集往往不足以无法完成目标架构适配，因此需要实现自定义合法化函数 legalizeCustom 来承载专有的合法化逻辑。与此同时，对于架构强相关的内建函数，还需要额外实现 legalizeIntrinsic 的处理入口，以完成对应内建函数的合法化。以 G_VASTART 指令为例，该指令用于在可变参数函数中标记可变参数的起始位置，其合法化过程通常需要通过 legalizeCustom 实现定制化转换，将通用的 G_VASTART 语义映射为 DSP 架构可执行的内存存储序列，并显式体现与栈帧布局相关的约束关系。相关实现代码如下所示。

```

1  bool DSPLegalizerInfo :: legalizeVAStart(MachineInstr &MI,
2      MachineIRBuilder &MIRBuilder) const {
3      assert(MI.getOpcode() == TargetOpcode::G_VASTART);
4      MachineFunction *MF = MI.getParent()->getParent();
5      DSPMachineFunctionInfo *FuncInfo = MF->getInfo<DSPMachineFunctionInfo>();
6      int FI = FuncInfo->getVarArgsFrameIndex();
7      LLT AddrTy = MIRBuilder.getMRI()->getType(MI.getOperand(0).getReg());
8      auto FINAddr = MIRBuilder.buildFrameIndex(AddrTy, FI);
9      assert(MI.hasOneMemOperand());
10     MIRBuilder.buildStore(FINAddr, MI.getOperand(0).getReg(), *MI.memoperands()
11         ()[0]);
12     MI.eraseFromParent();
13     return true;
}

```

2.4 寄存器组选择

虽然指令合法化阶段引入了目标架构的指令信息，并将 GMIR 生成阶段生成的 GMIR 转换为符合目标架构约束的合法 GMIR，但此时的 GMIR 指令仍然没有具体的寄存器分配信息。指令里的虚拟寄存器操作数仅携带用于表示其类型和位数大小的数据类型描述。因此，全局指令选择模块需要进一步为 GMIR 中的虚拟寄存器确定其可映射的寄存器组类型，即完成寄存器组选择过程。

寄存器组选择是全局指令选择模块的第三个基本 Pass，它会利用目标架构的寄存器信息，自顶向下的为合法化后的 GMIR 指令中的虚拟寄存器操作数分配合适的寄存器组类型，并且它还可以利用 GMIR 指令之间的关系选取一个较优的寄存器组类型。这也是不能直接在 GMIR 生成处理中为直接分配寄存器类型的原因之一：在 GMIR 生成阶段生成指令时，LLVM IR 还没有全部转换成 GMIR，无法利用指令之间的关系进行寄存器类型择优。

2.4.1 寄存器组

在 GlobalISel 框架中，Register Bank（寄存器组）是目标架构层面的关键抽象组件，其设计目标是在满足硬件指令访问约束的前提下，通过弱约束分组机制平衡寄存器分配的灵活性与跨寄存器资源的数据交互开销，从而兼顾编译效率与生成代码性能。与传统编译器的 Register Class（寄存器类）不同，Register Bank 仅关注寄存器的最大数据宽度及支持的操作集合，而不对具体物理编号和精确位宽做强约束，这使得编译器在分配阶段拥有更大的选择空间。

寄存器组的提出源于硬件架构的实际限制。许多嵌入式处理器、DSP 及异构计算平台将物理寄存器划分为多个相互独立的寄存器文件，不同文件之间的寄存器无法被同一条指令同时访问。若操作数分布在不同寄存器文件中，必须

通过额外的数据复制指令完成中转，从而引入不必要的性能开销。寄存器组通过将物理寄存器按文件属性进行逻辑分组，使编译器能够在寄存器分配阶段优先将关联变量分配到同一寄存器组中，从源头减少跨文件数据复制。

在实际架构中，寄存器组通常与运算单元类型紧密对应，例如通用寄存器组（GPR）和浮点寄存器组（FPR）。GMIR 指令在寄存器组使用上遵循硬性约束 + 柔性适配的原则：一方面，特定指令必须使用指定寄存器组（如浮点运算只能使用 FPR）；另一方面，部分指令允许在多个寄存器组间灵活分配，其最终选择由后续运算需求决定，从而避免不必要的跨组数据复制。该弱约束设计不仅降低了寄存器分配复杂度，也显著减少了跨寄存器文件的数据搬运开销，尤其适合嵌入式与 DSP 等资源受限架构。同时，寄存器组的抽象方式具有良好的扩展性，新架构仅需在目标描述中定义寄存器组属性即可完成适配，无需修改编译器核心逻辑。表2-2列出了 DSP 架构下使用的寄存器组与寄存器类划分情况，为后续寄存器银行选择与物理寄存器分配过程提供基础。

表 2-2 DSP 架构下寄存器抽象类型与分类

寄存器抽象层级	具体实例
Register Bank	GPRBRegBank、CRBRegBank、VPRBRegBank
Register Class	GPR32EVEN、GPR32、GPR64、VTRegs、OBMRegs、VPR4Out、VPR8Out、VPR16Out、VPRSf、VPRHf 等

2.4.2 寄存器组选择方法

寄存器组选择阶段的核心任务是在合法化完成之后，为 GMIR 指令中引入的虚拟寄存器操作数分配合适的寄存器组，从而确保后续指令选择与物理寄存器分配过程能够满足目标架构的硬件访问约束。在 DSP 目标架构下，寄存器组选择模块的整体设计结构如图2-8所示。该模块主要由 DSPGenRegisterBankInfo 与 DSPRegisterBankInfo 两部分构成：前者由 TableGen 根据 DSPRegisterBanks.td 文件自动生成，负责描述目标架构中可用的寄存器组及其基础属性；后者则作为目标后端的核心实现类，封装了寄存器组选择所需的全部元信息与决策逻辑，包括指令到寄存器组的映射规则、操作数属性以及不同映射方案之间的代价评估。

2.4.3 寄存器组选择实现

基于上一节对寄存器组选择机制的设计分析，本节将从寄存器组定义、映射结构定义及映射逻辑实现三个核心维度，详细阐述 DSP 架构编译器后端寄存器

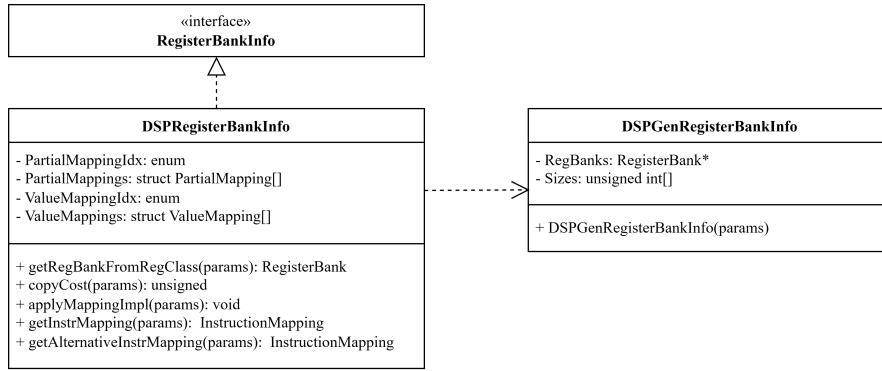


图 2-8 DSP 架构下寄存器组选择模块结构

组选择功能的具体实现方案。

1. 寄存器组的定义

DSP 处理器内部包含了多种不同类型的寄存器，不同寄存器在位宽、用途及访问方式上存在显著差异。为保证寄存器分配与硬件执行语义的一致性，需要首先对目标架构中可参与寄存器分配的寄存器资源进行建模。表2-3给出了 DSP 架构中主要寄存器类的数量、位宽及功能特性，为后续寄存器组划分与寄存器组选择策略的设计提供硬件基础。表中列出的仅为 DSP 架构下部分不透明寄存器类及其功能描述，其中不透明寄存器类是指对编译器后端可见、并参与寄存器分配与调度的寄存器集合；而对开发人员透明的寄存器类（如中断返回地址寄存器、基址寄存器、偏移寄存器及取模寄存器等）由于其使用方式固定，不参与寄存器分配过程，故不在此列出。

表 2-3 DSP 架构主要寄存器类特性

寄存器类	数量	宽度 / bit	功能描述
通用寄存器 (GR)	32	32	使用频率最高的寄存器，用于存放标量指令中的运算数据及地址信息，支撑整型运算与内存访问等基础操作
矢量寄存器 (VR)	4×16	640	用于存放矢量运算数据，采用四路并行设计，支持 SIMD 模式下多数据元素的并行加载、存储与计算
控制寄存器 (CR)	1	32	用于保存处理器状态信息，包括进位标志、溢出标志以及比较指令的结果标志等，参与指令执行控制与条件判断

在 DSP 架构编译器后端的 RegisterInfo 类中，通用寄存器被划归至 CPURegs 寄存器类，矢量寄存器被划归至 CPUVecRegs 寄存器类，控制寄存器被划归至 CCR 寄存器类。上述寄存器类的划分构成了寄存器组定义的基础，基于该分类体系完成了 DSP 架构寄存器组的定义如下：

```

1 def GPRBRegBank : RegisterBank<"GPRB", [GPR32, GPR64]>;
2 def VPRBRegBank : RegisterBank<"VPRB", [VPR4Out, VPR8Out, VPR16Out]>;

```

```
3 def CRBRegBank : RegisterBank<"CRB", [CCR]>;
```

2. 映射结构的定义

图2-9直观展示了 GlobalISel 框架中寄存器组选择阶段采用的分层映射结构，以及该结构在不同类型指令上的具体应用方式。该映射体系自底向上由 PartialMapping、ValueMapping 和 InstructionMapping 三个层次逐级构成，用于描述指令操作数到寄存器组的映射关系及其代价评估过程。

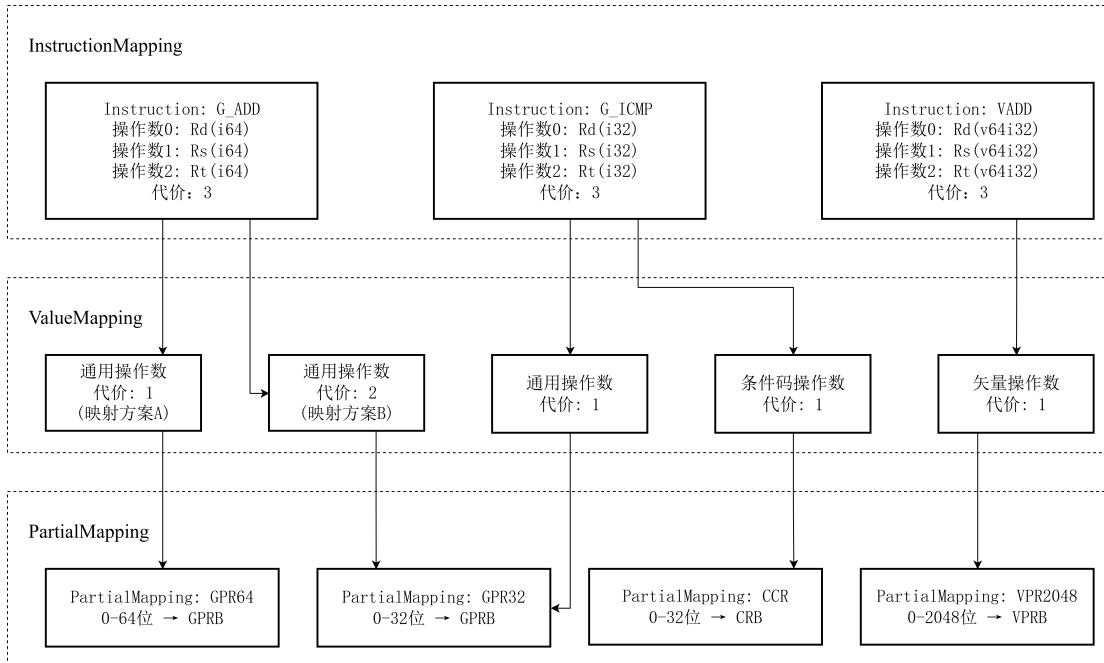


图 2-9 寄存器组分层映射结构图

在最底层，PartialMapping 用于描述操作数中某一连续比特区间到特定寄存器组的映射关系，支持大整数拆分、向量拆分以及非对齐布局等情况。例如，对于 64 位整型加法指令 G_ADD，其操作数可通过一个 PartialMapping 将 0 ~ 64 位整体映射至通用寄存器组 GPRB；而对于 32 位整型比较指令 G_ICMP，其操作数仅需将 0 ~ 32 位映射至 GPRB。对于控制类操作数，则通过 PartialMapping 将对应比特区间映射至控制寄存器组 CRB；在矢量指令 VADD 场景下，宽达 2048 位的矢量操作数整体映射至矢量寄存器组 VPRB。

ValueMapping 由一个或多个 PartialMapping 组合而成，用于描述单个操作数到寄存器组的完整映射方案。图中展示了同一类通用操作数可能存在多种 ValueMapping 方案的情况，不同方案对应不同的映射代价，用于反映寄存器使用效率或潜在的跨寄存器组数据复制开销。如图中的 64 位 G_ADD，其操作数既可以用一个 GPR64 来映射，又可以用两个 GPR32 来映射。

在最上层，InstructionMapping 对一条指令的所有操作数映射进行统一建模，其不仅包含每个操作数对应的 ValueMapping，还关联了该指令整体映射方案的

代价值。以图中所示为例，G_ADD、G_ICMP 以及 VADD 等指令均可生成各自的 InstructionMapping，其总代价由各操作数映射代价综合决定。寄存器组选择阶段将基于这些代价信息，在满足指令硬性约束的前提下选择最优映射方案。

在具体实现中，主要有两种寄存器组分配策略。其中，Fast 策略仅为指令选择默认可用的寄存器组组合，不对备选方案进行枚举与评估，因此实现简单、运行开销低，适用于对编译速度敏感的场景；而 Greedy 策略则在默认映射方案的基础上，进一步枚举目标架构允许的所有合法寄存器组组合，逐一计算其映射代价，并最终选择代价最低的方案。该策略能够在更大搜索空间内优化寄存器组分配，减少跨寄存器组的数据复制开销，但相应地增加了寄存器组选择阶段的计算成本。

在 LLVM 的 RegisterBankInfo 实现中，PartialMapping、ValueMapping 等结构通常以静态表的方式定义，用于描述不同位宽、不同操作数类型在目标架构下的寄存器组映射规则。通过这种表驱动的实现方式，寄存器组选择阶段能够在不依赖具体指令实例的前提下，高效查询操作数的映射方案及其对应代价，从而支撑后续 InstructionMapping 的构建与选择。下面代码片段给出了 DSP 目标架构中 PartialMapping 与 ValueMapping 的典型定义方式，其中 PartialMapping 用于描述操作数比特区间到寄存器组的基础映射，而 ValueMapping 则通过组合一个或多个 PartialMapping，形成对单个操作数的完整寄存器组映射描述。

```

1 PartialMapping PartMappings[] = {
2     {0, 32, GPRBRegBank},
3     {0, 64, GPRBRegBank},
4     {0, 32, CRBRegBank},
5     .....
6 };
7
7 ValueMapping ValueMappings[] = {
8     {nullptr, 0},
9     {&PartMappings[PMI_GPRB32], 1},
10    {&PartMappings[PMI_GPRB32], 1},
11    {&PartMappings[PMI_GPRB32], 1},
12    {&PartMappings[PMI_GPRB64], 1},
13    {&PartMappings[PMI_GPRB64], 1},
14    {&PartMappings[PMI_GPRB64], 1},
15    .....
16 };

```

通过上述分层映射结构，寄存器组选择阶段实现了从比特级映射到指令级决策的逐层抽象，既避免了过早绑定具体物理寄存器，又能够在全指令粒度上评估不同寄存器组分配方案的代价，为后续指令选择与寄存器分配阶段提供了清晰的寄存器资源使用边界。

3. 映射逻辑的实现

寄存器组选择的核心逻辑通过重载 RegisterBankInfo 类中的相关接口函数实现，其中最关键的是 getInstrMapping 与 getInstrAlternativeMapping 两个函数。前者用于为指令生成默认的寄存器组映射方案，后者则用于在必要时提供备选映射方案，以支持更灵活的寄存器组分配策略。

getInstrMapping 的实现流程如算法1所示。其实现采用以规则驱动为核心的寄存器组映射构建流程。该流程首先以指令操作码为入口，对可直接由 TableGen 规则覆盖的场景进行快速判定。对于非通用指令及 PHI 指令，优先尝试使用 TableGen 自动生成的映射结果；一旦匹配成功，即可直接返回对应的 InstructionMapping，从而避免后续不必要的分析与代价评估。

当 TableGen 映射不可用时，算法进入基于操作数语义的通用映射阶段。此阶段以操作数为粒度，遍历指令中的所有寄存器操作数，并依据其 LLT 进行映射决策。在完成默认映射后，算法进一步结合指令语义对映射结果进行修正。对于使用条件值的指令，其条件操作数需显式映射至控制寄存器组，以保证条件语义与目标架构控制寄存器模型的一致性。最终，算法基于上述映射规则构建并返回完整的 InstructionMapping。

2.5 机器指令选择

机器指令选择是 GlobalISel 框架的第二阶段核心任务，承接第一阶段（GMIR 生成、指令合法化、寄存器类型选择）的处理结果，其核心目标是将经过合法化的、附着目标架构信息的通用机器中间表示 GMIR，完整转换为目标架构专属的机器中间表示 MIR，为后续基于 MIR 的寄存器分配、指令调度、窥孔优化等后端流程提供合法输入，是衔接通用中间表示与架构专属表示的关键枢纽。

机器指令选择以单个函数为处理粒度，基于树覆盖的指令选择算法，通过将 GMIR 指令序列构建为 AST，再通过树模式匹配映射为目标架构指令序列，确保语义等价性与架构兼容性。目前框架实现了两种树覆盖的方式：一种是基于表驱动的自动状态机进行的自动覆盖方式；另一种是基于固定模式的手动覆盖方式。这两种覆盖方式在每次覆盖的时候都只会产生一种成功匹配的树模式，因此可以直接生成对应的 MIR 指令序列^[52]。值得注意的是，GlobalISel 的机器指令选择与传统基于 DAG 的指令选择存在显著设计差异，后者因引入 DAG 中间表示，需在选择阶段额外执行拓扑排序流程，导致实现复杂度与设计难度显著提升，而 GlobalISel 通过直接基于 GMIR 的树覆盖匹配规避了这一环节，简化了整体设计逻辑。

机器指令选择采用 PROT+ 自底向上处理的执行逻辑，确保指令依赖关系的正确性与处理效率，具体流程如下：首先以单个函数为处理单元，采用 RPOT 算法从函数底部开始依次扫描所有基本块，该遍历顺序可保障父基本块的指令处

Algorithm 1: Core Logic of getInstrMapping

Input: MI: Machine Instruction

Output: InstructionMapping

```

1 Opc ←opcode of MI;
2 if MI is non-generic or Opc is G_PHI then
3   | Mapping ←getInstrMappingImpl(MI);
4   | if Mapping is valid then
5     |   | return Mapping;
6   | end
7 end
8 Initialize operand mapping array OpdsMapping;
9 foreach operand op in MI do
10  | if op is a register and op.reg is valid then
11    |   | Obtain the low-level type Ty of op.reg;
12    |   | if Ty is invalid or bit width of Ty ≤ 32 then
13    |   |   | Assign op to a 32-bit GPR container;
14    |   | end
15    |   | else
16    |   |   | Assign op to a 64-bit GPR container;
17    |   | end
18  | end
19 end
20 if Opc consumes a condition value then
21   | Override the mapping of the condition operand to the control register
      | bank (CRB);
22 end
23 if Opc produces a comparison result then
24   | Force the result operand to use a 32-bit GPR container;
25 end
26 Construct and return the final InstructionMapping from OpdsMapping;
```

理滞后于子基本块，避免因分支依赖导致的未定义指令引用问题；进入单个基本块后，按自底向上的顺序逐行处理每条 GMIR 指令，处理前先判断当前指令是否已生成对应的 MIR 指令，若已生成则直接跳过以避免重复处理，若未生成则以该指令为根节点触发自动匹配模块与手动匹配模块的协同匹配流程；匹配成功后由指令生成模块输出对应的 MIR 指令序列，若匹配失败则直接触发编译报错；最后以基本块为单位迭代执行上述流程，直至整个函数的所有 GMIR 指令均成功转换为 MIR 指令或触发报错，完成该函数的机器指令选择任务。

2.5.1 匹配状态机

在 LLVM 的指令选择体系中，匹配状态机是支撑基于表驱动的树覆盖指令选择的核心底层组件，其本质是一种由 TableGen 工具自动生成的 FSM (Finite State Machine，有限状态机)，其核心作用是将 GMIR 指令构成的 AST 与目标架构的指令模式进行自动化匹配，从而确定可映射为 MIR 的合法目标指令模式。

在生成阶段，通过在 td 文件中以树模式的形式描述目标架构指令的结构特征。TableGen 工具对所有指令树模式进行解析，提取其中的关键信息，包括操作码、操作数类型、寄存器组约束以及数据位宽等，并将这些特征系统化地编码为状态机的状态节点与转移条件。其中，状态节点对应指令树在不同层级上的特征匹配状态，而转移条件则描述从当前特征匹配到下一层特征的合法规则。最终，TableGen 将所有状态节点与转移关系整合为完整的状态机代码，覆盖从初始状态、匹配中状态到成功匹配状态或失败状态的完整流程，且每一个成功匹配状态均唯一对应一条目标架构指令模式。

在机器指令选择阶段，匹配状态机针对单条 GMIR 指令的执行流程可概括为以下几个步骤：

1. 指令树构建：以当前待处理的 GMIR 指令为根节点，递归遍历其操作数、子指令，构建完整的抽象语法树。
2. 状态机初始化：将状态机置为初始状态，从指令树的叶子节点开始向上遍历。
3. 状态转移匹配：在遍历过程中，根据当前 AST 节点的特征触发状态机的转移并进入对应状态；若在某一节点处不存在满足条件的转移路径，则判定该子树匹配失败，状态机回退至最近的分支点并尝试其他可行路径；当遍历至根节点且状态机进入成功匹配状态时，即输出对应的目标架构指令树模式。

为保证匹配结果的确定性，LLVM 中的指令匹配状态机被设计为 DFA (Deterministic Finite Automaton，确定性有限状态机)。在该设计下，同一指令树在匹

配过程中仅会触发唯一的一条状态转移路径，并最终生成唯一的成功匹配结果，从机制上避免了多模式冲突问题。

匹配状态机在指令选择阶段的主要优势体现在自动化与可扩展性。开发时无需为每条目标指令手工编写匹配逻辑，只需在 `td` 文件中定义相应的指令树模式，即可由 `TableGen` 自动生成匹配状态机，从而降低目标架构适配所需的代码量。当目标架构新增指令时，也仅需补充对应的树模式并重新生成状态机，无需修改编译器核心代码，保证了指令选择框架的稳定性与可维护性。此外，匹配状态机采用确定性遍历机制，在匹配过程中避免了对所有指令模式的穷举搜索。相较于手动遍历或回溯式匹配方法，该机制能够显著提升指令匹配效率。

2.5.2 机器指令选择方法

从功能划分的角度来看，机器指令选择可划分为三个模块：自动匹配模块、手动匹配模块以及指令生成模块。三者共同完成从 `GMIR` 到 `MIR` 的语义映射过程。

1. 自动匹配模块

自动匹配模块负责构建基于表驱动的自动状态机，通过状态转移的方式自动完成 `GMIR` 指令与目标架构指令模式之间的匹配，其优势在于无需人工干预即可适配大部分通用指令的匹配需求，提升编译器对指令集扩展的自适应能力。这种设计不仅显著减少了手写匹配代码的工作量，也提升了编译器在面对指令集扩展或新增指令时的可维护性与可扩展性。

自动匹配模块的工作流程可进一步划分为两个阶段：第一阶段发生在编译器构建期，由 `TableGen` 工具解析目标架构指令的树模式描述，并将其编码为确定性有限状态机；第二阶段发生在编译期，指令选择器利用已生成的状态机，对单条 `GMIR` 指令构建的抽象语法树进行自底向上的遍历，通过状态转移自动判定其可匹配的目标指令模式。

为兼容 `SelectionDAGISel`，`GlobalISel` 对通用算术与逻辑指令直接复用既有指令模式，而对依赖 `SDNode` 的架构专属信息则重新建模，并通过 `GINodeEquiv` 显式建立 `GMIR` 操作码与 `ISD` 操作码的等价关系，实现指令模式的复用。以下示例展示了加法指令的映射。

```

1 Class GINodeEquiv<Instruction i, SDNode node> {
2     Instruction I = i;
3     SDNode Node = node;
4 }
5
6 def : GINodeEquiv<G_ADD, add>;

```

2. 手动匹配模块

尽管自动匹配模块能够覆盖大多数通用指令的选择需求，但在实际工程中仍存在部分目标架构特有的复杂指令，其语义或操作数约束难以通过 TableGen 的树模式进行完整描述。针对这类场景，需要通过自定义 C++ 代码实现精细化的指令匹配逻辑。

手动匹配模块通常以内置固定树模式或显式条件判断的形式存在，通过逐条检查 GMIR 指令的操作码、操作数类型及寄存器约束，判定其是否符合特定架构专属指令的语义特征。一旦匹配成功，即直接生成对应的目标指令。该模块作为自动匹配的补充机制，主要用于处理高度架构相关、对性能或硬件语义要求严格的指令场景，弥补纯表驱动匹配在表达能力上的不足。

3. 指令生成模块

指令生成模块位于机器指令选择流程的末端，其输入为自动匹配或手动匹配阶段选定的指令树模式。该模块依据模式与目标架构指令之间的映射关系，构造语义等价的 MIR 指令序列，并完成必要的操作数绑定与属性设置。通过该模块，GMIR 中的通用指令被最终转换为与目标架构紧密绑定、可直接参与后续寄存器分配与指令调度的机器指令表示。

2.5.3 机器指令选择实现

机器指令选择阶段以单条 GMIR 指令为处理粒度，其整体流程可概括为预处理、特殊指令处理、合法性校验、自动匹配以及手动匹配五个阶段，如算法2所示。

1. 预处理

为适配 DSP 目标架构的指令语义，指令选择前需对部分通用指令执行前置 Lowering，将其规范化为更贴近硬件的等价形式。典型场景是指针相关操作：由于 DSP 架构缺少独立的指针运算指令，指针通常以整型寄存器保存并参与运算，因此需要将带指针语义的通用指令转换为等价的整型语义指令，并同步修正结果寄存器的 LLT，保证后续匹配与约束的一致性。

2. 特殊指令处理

对于语义或结构特殊、难以通过树模式描述的指令，需要在自动匹配之前进行单独处理，主要包括以下几类。

- PHI 指令用于描述控制流汇合处的 SSA 值合并，属于编译器内部语义，不对应具体硬件指令，其处理重点在于保证定义寄存器与各输入操作数满足一致的寄存器约束，以确保后续寄存器分配的合法性。
- COPY 指令仅承担数据搬运语义，不包含可匹配的运算结构，且可能涉及虚拟寄存器与物理寄存器之间或跨寄存器资源的数据复制，通常无法通过

通用树模式完成映射，因此需要采用专门的选择逻辑。

- 此外，由于 DSP 架构的 ISA 限制，对于依赖控制寄存器 Condition 位的指令，如 G_BRCOND、G_ICMP、G_FCMP 以及 G_SELECT 等，需在 TableGen 驱动的通用匹配流程之前进行提前处理。通过在指令选择阶段直接识别并生成符合目标架构条件码语义的指令序列，可以避免条件值经由普通数据路径再映射至控制寄存器所带来的冗余指令与数据搬移开销。

3. 合法性校验在进入模式匹配前，对指令结构进行必要的合法性检查。例如，通用指令不应携带未显式声明的隐式操作数；若指令形态异常，直接返回失败以避免产生不一致的选择结果，并便于定位后端实现问题。

4. 自动匹配对通过前述阶段筛选的通用指令，调用 selectImpl 执行基于 TableGen 的树覆盖匹配。该路径覆盖面广、维护成本低，能够将绝大多数通用算术、逻辑、访存等指令自动映射为目标指令模式，是指令选择的主要实现方式。

5. 手动匹配当指令无法被自动匹配覆盖时采用手动匹配。手动匹配通常以专用选择函数的形式存在，对操作数、立即数约束及目标指令序列进行精确构造。例如，针对寄存器内符号扩展的 G_SEXT_INREG，可通过检查扩展位宽并构造等价的目标指令序列完成选择，同时对新生成指令施加寄存器约束，保证与后续流程兼容。

手动匹配在处理指令集差异时尤为关键。以 UMULH（无符号高半乘法）为例：对于原生支持该指令的芯片型号，可以直接通过 TableGen 的 Pattern 将 GMIR 操作映射为硬件指令；而对于不支持 UMULH 的型号，则需要在手动路径中将其展开为等价指令序列。

这种设计使得条件值的生成、传递与使用能够保持在最小且明确的指令路径上，不仅简化了指令选择过程，降低了指令序列中冗余复制操作的数量，还减少了寄存器压力和执行延迟，对分支频繁、条件判断密集的程序尤为有利，有助于提升最终生成代码的整体执行效率。

2.6 本章小结

本章围绕 GlobalISel 在 DSP 架构上的实现，对全局指令选择框架的关键组成与实现路径进行了系统阐述。首先，对 LLVM 现有指令选择体系进行了对比分析，指出传统 SelectionDAGISel 在流程复杂度、维护成本与灵活性方面的局限，以及 FastISel 在代码质量与覆盖能力上的不足，从而引出 GlobalISel 以 GMIR 为核心载体、以函数为粒度的整体设计思路与流程拆分方式。

随后，本章给出了 GlobalISel 第一阶段 GMIR 生成的实现要点，重点说明了通用翻译逻辑与目标架构相关逻辑的分层协作机制。针对 DSP 架构在 ABI、寄

Algorithm 2: Core Logic of Instruction Selector

Input: MI: Machine Instruction
Output: Boolean: Instruction selection result

```

1 Initialize context objects for MI (MBB, MF, MRI, MIB);
2 Execute pre-selection lowering on MI;
3 if MI is a copy instruction then
4   | return Process copy instruction selection for MI;
5 end
6 Get opcode (Opc) of MI;
7 if Opc is G_PHI then
8   | Extract target register (DefReg) of PHI;
9   | Derive target register class (DefRC) of DefReg;
10  | if DefRC derivation fails then
11    |   | return false;
12  | end
13  | for each operand in PHI instruction do
14    |   | if Operand cannot be constrained to DefRC then
15    |   |   | return false;
16    |   | end
17  | end
18  | Update PHI instruction descriptor;
19  | return Constrain DefReg to DefRC;
20 end
21 if MI has implicit operands not declared explicitly then
22   | return false;
23 end
24 if Opc is a condition-related generic instruction then
25   | return Apply condition-specific selection;
26 end
27 if table-driven selectImpl succeeds then
28   | return true;
29 end
30 if Opc is a special-type generic instruction then
31   | return Apply special-type instruction selection;
32 end
33 return false;
```

存器宽度与数据类型支持方面的约束，本文通过定制 ValueHandler，实现了参数传递、返回值处理及调用指令构建等关键环节，保证了 LLVM IR 到 GMIR 的语义下沉过程在 DSP 平台上的正确性与可扩展性。

在合法化阶段，本章阐述了基于 LegalizerInfo 规则集的合法性判定机制，并结合 DSP 平台的类型集合与指令支持范围构建合法化规则；同时针对通用规则难以覆盖的复杂场景，给出了基于 Custom 的专用处理方式，以确保 GMIR 能够完全落入目标硬件能力边界，为后续流程提供稳定输入。

在寄存器组选择阶段，本章阐述了 Register Bank 抽象在多寄存器文件架构中的工程意义，并结合 DSP 寄存器资源划分，给出了寄存器组与寄存器类的建模方式。进一步地，本文基于 Mapping 的三层映射结构，说明了寄存器组选择的映射构建与代价评估机制，并给出 getInstrMapping 及备选映射接口在 DSP 后端中的实现策略，为降低跨寄存器组数据搬移开销提供支撑。

最后，在机器指令选择阶段，本章总结了 GlobalISel 基于树覆盖的选择方法，并给出了 DSP 指令选择器的核心流程：通过预处理与特殊指令处理保障输入形态一致性，在此基础上优先采用 TableGen 驱动的自动匹配完成通用指令选择，同时以手动匹配补充复杂指令或架构特化语义。至此，本章完成了从 LLVM IR 到 DSP 架构 MIR 的关键路径打通，为后续章节针对全局指令选择的优化策略设计与性能评估奠定了实现基础与实验前提。

第3章 面向 DSP 的全局指令选择优化策略

3.1 优化策略分析与设计

在经过 GMIR 生成、指令合法化、寄存器组选择以及机器指令选择 4 个核心 Pass 处理之后，原始的 LLVM IR 已经成功被转换成 MIR。尽管这一流程已实现指令选择阶段的目标，但是生成的代码质量相比于 SelectionDAGISel 较差。代码质量的差距主要体现在指令数量冗余、硬件专用指令利用率低以及寄存器资源分配不够高效等方面。

为了缩小与 SelectionDAGISel 在代码质量上的差距，本文引入了包括 CSE (Common Subexpression Elimination, 公共子表达式消除)、已知比特分析 (Known-Bits Analysis) 以及合并优化 (Combiner) 等多项优化技术。这些优化策略通过降低指令冗余、精确推断操作数的已知状态并简化表达式，从而显著提升了生成代码的性能。

3.1.1 优化策略理论依据

程序在编译过程中，尤其是在从高级的 LLVM IR 向低级的目标相关指令转换时，会不可避免地引入大量的冗余。这些冗余体现为计算冗余（公共子表达式重复计算）、数据冗余（多余的拷贝指令）和表示冗余（复杂的指令序列可用更高效的指令替代）。优化的根本任务就是系统性地识别并消除这些冗余，在保证程序语义不变的前提下，提高生成代码的执行效率、代码密度及硬件利用率。

GlobalISel 与 SelectionDAGISel 的代码质量差距，本质上源于两者设计目标与实现路径的差异。SelectionDAGISel 针对特定架构定制化设计，其核心表示为 SelectionDAG，天然具备对局部计算图进行重写与合并的能力；而 GlobalISel 为适配不同架构的指令集特性，采用通用中间表示 + 模块化 Pass 的设计，导致代码生成过程中不可避免地引入冗余，具体根源可归结为三点：

1. 通用化转换带来的语义冗余：为保证跨架构语义的一致性，GlobalISel 在 LLVM IR 向 GMIR 转换及指令合法化阶段，会将复杂运算拆分为一系列简单的原子指令。例如，将 DSP 架构的乘加运算 (MAC) 拆分为乘法与加法

指令，将宽类型数据访问拆分为多次窄类型加载/存储指令，这种拆分虽保证了语义正确性，却引入了大量指令级冗余。

2. 寄存器组选择的局部性局限：寄存器组选择阶段以基本块为粒度分配寄存器组，未充分考虑函数级的全局数据流依赖，易导致跨基本块的频繁跨组数据复制。例如，循环变量在不同基本块中被分配到不同寄存器组，需通过专用复制指令完成数据迁移，显著增加了执行延迟。
3. 硬件特性适配的滞后性：机器指令选择阶段的主要目标是完成从通用 GMIR 到目标指令的语义映射，其默认策略往往采用局部的、一对一的映射。由于复杂指令模式的识别被推迟到较晚阶段，编译器往往难以及时捕捉跨多条 GMIR 指令的复合语义，这会导致硬件中专用功能单元的性能潜力无法被充分挖掘。

从理论上来看，高效优化的前提是精确可靠的程序分析。数据流分析（如 CSE 所依赖的可用表达式分析）揭示了值是如何在程序中产生和传播的；而信息流分析（如 KnownBits 分析）则从位级推断值的可能范围。这些分析为优化变换提供了可行性证明和安全保障。

从实际上来看，DSP 处理器的指令集包含大量面向特定计算模式设计的复合指令，如乘加、向量加载/存储及多数据并行运算指令。优化需要具备对指令模式的识别能力，能够在合适的阶段将通用操作序列映射为高度优化的硬件指令。只有在分析与变换机制的共同支撑下，编译器才能在保证通用性的同时，生成兼具高性能与高效率的目标代码。

3.1.2 关键优化技术分析

1. 公共子表达式消除

公共子表达式消除是一种经典的编译优化技术，其核心思想是消除程序中重复计算相同表达式的冗余操作，避免冗余的运算操作与寄存器占用，从而精简代码体积、降低执行延迟。在指令选择阶段，如果一个表达式的值已经被计算过，并且其操作数未被重新定义，则可以复用之前的计算结果，避免重复计算，从而减少冗余指令的生成。如下代码块3.1和代码块3.2所示，该样例在优化前 $b+c$ 被计算了 3 次，优化后只需要计算 1 次。

与传统的 CSE 不同，GlobalISel 采用了持续 CSE (Continuous CSE) 策略。这种策略通过在指令创建时即时检查是否存在等价的表达式，从而消除不必要的冗余计算。持续 CSE 能够在基本块内高效工作，避免了全函数扫描的代价高昂性。这种局部的优化策略能够大幅提升编译效率，同时保持优化效果。

```

1 a = b + c;
2 d = b + c;
3 e = b + c * 2;
4 f = (b + c) * 3;
5
6

```

Listing 3.1 CSE 优化前

```

1 temp = b + c;
2 a = temp;
3 d = temp;
4 e = b + c * 2;
5 f = temp * 3;
6

```

Listing 3.2 CSE 优化后

CSE 不仅能够消除冗余指令，为后续优化创造更多的机会，而且还能减少寄存器的使用。在指令合并优化阶段，复用公共子表达式的计算结果往往能够暴露出更复杂的合并模式。例如，如果两个计算共享相同的子表达式，合并优化可能通过识别这些冗余的计算来进一步简化指令序列，提升代码执行效率。

2. 已知比特分析

已知比特分析是一种静态推断寄存器各位已知状态的技术，主要用于优化常量传播和指令简化，通过跟踪每个位的已知状态来帮助优化器做决策。核心目标是在不引入额外运行时开销的前提下，尽可能提前获取寄存器值的精确信息，从而辅助常量传播、指令简化以及后续的合并优化决策。该分析通过对寄存器中必然为 0 或必然为 1 的比特位进行建模，为优化器提供细粒度的位级语义信息。

已知比特分析以寄存器的初始比特状态作为分析起点。这些初始信息可能来源于多种途径，例如：函数参数在 ABI 层面的位宽约束、全局变量的初始化值、字面量常量的比特分布，或前序优化阶段已推导出的比特属性。随后，分析沿着指令的数据流依赖关系，对寄存器的比特信息进行逐指令传播与更新。

在实现层面，每个寄存器通常维护两个并行的比特掩码：KnownZeros 和 KnownOnes。其中，KnownZeros 标记所有必然为 0 的比特位，KnownOnes 标记所有必然为 1 的比特位，其余未被标记的比特则视为未知。随着指令流的推进，这两个掩码会根据指令语义持续更新，从而逐步积累更精确的寄存器状态信息。下面的示例展示了已知比特分析在简单算术与逻辑运算中的效果：

```

1 int test(int a) {
2     int b = a & 0xFFFF;    // 低16位保留，高16位必为0
3     int c = b | 0x10000;  // 第17位必为1
4     return c;
5 }
6 // b: KnownZeros = 0xFFFF0000, KnownOnes = 0x00000000
7 // c: KnownZeros = 0xFFE0000, KnownOnes = 0x00010000

```

通过上述分析结果，编译器能够准确获知变量 b 的高 16 位始终为 0，而变量 c 的第 17 位必然为 1。这类信息在后续优化阶段具有重要价值：一方面可用

于消除冗余的掩码、扩展或比较指令；另一方面也为指令合并与目标指令匹配提供更精确的语义依据，从而生成更紧凑、高效的机器代码。

3. 合并优化

在 GlobalISel 框架中，合并优化是提升代码质量的核心手段之一。合并优化的主要目标是通过删除冗余的指令、简化表达式、合并常见的操作模式来减少代码的规模，并提高执行效率。它能充分利用目标架构的复杂指令，减少不必要的操作和存储器访问。

合并优化本质上可以视为窥孔优化（Peephole Optimization）的推广形式，但其作用范围已不再局限于固定长度的局部指令窗口，而是能够在更大的指令序列尺度上进行模式识别与重写，甚至支持跨多条指令、跨基本块的优化决策。该优化策略的理论基础主要来源于以下几个方面的观察。

首先，在低级代码生成过程中，编译器往往会为了表达中间计算结果而生成一系列结构简单但语义相关的指令序列，这类指令在指令级层面存在显著冗余。若目标架构提供语义等价但更为复杂的指令形式，这些冗余序列通常可以被单条高效指令所替代，从而减少指令数量和中间结果的产生。

其次，指令之间普遍存在较强的数据流局部性。相邻或近邻指令往往共享操作数或存在直接的定义-使用关系，这为编译器识别可合并的操作模式提供了天然条件，使得合并优化能够在保持语义正确性的前提下，对指令序列进行整体重构。

最后，现代处理器指令集日益丰富，广泛支持乘加（MAC）、融合算术、向量化等复合指令形式。合并优化正是连接通用中间表示与这些复杂目标指令之间的关键桥梁，其效果直接决定了编译器对目标架构计算能力的利用程度。

在所有这些优化中，合并优化承担着将优化机会最终转化为性能收益的核心职责。其主要目标是通过删除冗余指令、简化表达式、合并常见的操作模式来减少代码规模，并提高执行效率。例如，它将两个连续的零扩展指令 G_ZEXT(G_ZEXT(x)) 合并为单一的 G_ZEXT(x)，或者将独立的乘法（G_MUL）和加法（G_ADD）指令融合为一条高效的乘加（MAC）指令。正是通过这种深度的指令融合，才能充分利用目标架构的复杂指令，减少不必要的操作和存储器访问。

3.1.3 面向 DSP 的优化设计

单一的优化策略通常难以应对复杂的代码生成需求，因此需要一个多策略协同的优化体系。本章将深入分析以合并优化为核心，并辅以公共子表达式消除、合法化策略以及窥孔优化等多种策略的全局指令选择优化方法。由于公共

子表达式消除属于通用优化且已经在框架中实现，只需要调用相应的接口即可，下文主要针对合并优化。

为在 GlobalISel 框架下实现高效、可扩展的 DSP 优化，本文采用基于 TableGen 的 GICombiner 描述机制与 C++ 回调逻辑相结合的实现路径。该设计充分利用了两种机制在表达能力与工程组织上的互补优势，实现了合并优化规则的高效描述与可靠执行。

具体而言，基于 TableGen 的 GICombiner 主要用于描述可合并指令的结构化模式。通过在 td 文件中定义 GICombineRule，可以以声明式的方式刻画指令之间的拓扑关系、操作码组合以及操作数约束条件。TableGen 在编译阶段自动生成对应的匹配执行器代码（如合法化前的 DSPGenPreLegalizeGICombiner.inc），从而避免了手工编写复杂的匹配逻辑，并保证了规则匹配过程的高效性与一致性。

另一方面，C++ 回调机制则用于承担 TableGen 难以表达的复杂语义判断与指令重构工作。在合并过程中，许多优化决策依赖于位级信息推断（如 KnownBits 分析）、子目标特性（如 DSP 硬件功能单元支持情况）、以及对合法性与数据依赖关系的精细控制。这类逻辑通常需要借助 C++ 代码访问分析结果并进行条件判断。因此，在每条合并规则中，具体的匹配判定与变换执行均由 C++ 实现，用于根据匹配结果构造新的 GMIR 指令序列，并维护虚拟寄存器类型、数据流关系及指令依赖的一致性。

仅采用基于 TableGen 的 GICombiner 难以准确表达复杂的语义依赖与分析驱动的优化条件，而完全依赖 C++ 回调又会导致合并规则分散在大量手写代码中，降低规则的可读性与可维护性。基于上述考虑，本文采用 TableGen 负责结构化模式描述、C++ 回调负责语义判断与指令重写的混合设计方案，在保证合并优化表达能力的同时，有效提升了工程可维护性与扩展性。

在实现层面，合并优化规则首先以 GICombineRule 的形式在 td 文件中声明，并由 TableGen 自动生成对应的匹配执行器代码。随后，这些规则由 Combiner 框架在 MachineFunction 粒度上统一调度与执行，对 GMIR 指令流进行遍历和合并，从而在指令合法化前后逐步消除冗余指令、重构指令序列，并提升最终生成代码的执行效率。

在 GlobalISel 框架中，指令合法化阶段不仅构成代码生成流程中的关键功能边界，同时也是优化策略划分的重要依据。由于指令在合法化前后，其类型信息、操作数形态及目标相关约束的完备程度存在显著差异，编译器在不同阶段所能实施的优化类型与优化空间也随之发生变化。基于这一特点，本文在 DSP 后端中将优化策略按生效阶段进行系统化划分：一类面向通用 GMIR 表达，主要针对语义拆分冗余消除与表达式规范化，统一在 Pre-Legalize Combiner 阶段实施；另一类则依赖合法化后稳定的类型与操作数信息，侧重于目标相关指令匹

配与代码质量提升，在 Post-Legalize Combiner 阶段完成。下文将结合具体优化实例，对上述两类优化策略的设计与实现进行详细分析。

3.2 合法化前优化策略

合法化之前，GMIR 仍处于高度通用的表示阶段，指令尚未被约束为 DSP 架构所支持的具体形式。该阶段的优化不涉及目标指令集的具体限制，重点在于语义层面的简化与结构性重写，以减少冗余并规范表达式形态，为后续 Legalizer 与 Instruction Selector 提供更加稳定的匹配条件，同时保留足够的优化自由度，主要包括以下几类：

- 表达式合并与规范化，如消除冗余的类型扩展 ($G_ZEXT(G_ZEXT(x))$)、合并重复的算术或逻辑操作，为后续指令选择创造更简洁的模式。
- 基于语义等价的指令融合，例如将 G_MUL+G_ADD 的通用表示提前重构为更利于匹配硬件乘加指令 MAC 的形式。
- 分析驱动的优化，如利用 KnownBits、CSE 等分析结果进行常量折叠、冗余计算消除和条件简化。

由于此阶段允许暂时存在非法指令，优化变换不必立即满足目标指令集的约束，从而具备更大的自由度，能够在保持语义不变的前提下，对指令结构进行深度重构。

3.2.1 内存操作优化

作为 Pre-Legalize Combiner 阶段的典型优化之一，内存操作指令的优化主要针对通用 GMIR 中高层内存操作所引入的函数调用冗余问题。在 GlobalISel 默认实现中，如 G_MEMCPY 、 $G_MEMMOVE$ 等高层内存操作指令通常会被直接降级为库函数调用。然而，对于小规模内存操作而言，函数调用所引入的额外开销往往远大于实际的内存访问成本，从而导致显著的性能浪费。为此，本文在合法化之前引入针对高层内存操作的合并优化策略：主动将小规模内存拷贝操作转换为显式的 load/store 指令序列，以避免不必要的函数调用。

在 DSP 后端的 Pre-Legalize Combiner 中，针对内存操作指令的处理逻辑如下：

Algorithm 3: lowerMemCpyFamily

Input: MachineInstr MI , unsigned $MaxLen$

Output: LegalizeResult

```

1  $Opc \leftarrow$  opcode of  $MI$ ;
2 if  $Opc \notin \{G\_MEMCPY, G\_MEMMOVE, G\_MEMSET\}$  then
3   return UnableToLegalize;
4 extract  $Dst$ ,  $Src$ ,  $Len$  and memory operands from  $MI$ ;
5 obtain  $DstAlign$ ,  $SrcAlign$  (if applicable);
6 if  $Len$  is not constant then
7   return UnableToLegalize;
8 if  $Len = 0$  then
9   erase  $MI$ ; return Legalized;
10  $IsVolatile \leftarrow$  volatility flag;
11 if  $Opc = G\_MEMCPY\_INLINE$  then
12   return lowerMemcpyInline(...);
13 if  $IsVolatile$  or ( $MaxLen > 0$  and  $Len > MaxLen$ ) then
14   return UnableToLegalize;
15 if  $Opc = G\_MEMCPY$  then
16   compute store limit from target lowering;
17   return lowerMemcpy(...);
18 if  $Opc = G\_MEMMOVE$  then
19   return lowerMemmove(...);
20 if  $Opc = G\_MEMSET$  then
21   return lowerMemset(...);
22 return UnableToLegalize;
```

3.2.2 拓展截断优化

除上述典型优化外，DSP 后端中还实现了一些针对特定冗余模式和代码生成细节的补充优化。其中，部分优化在 Pre-Legalize 阶段生效，用于简化通用指令结构；另一部分则依赖于目标指令特性，在 Post-Legalize 阶段完成。

1. 元余截断与比较指令优化

在通用指令生成与合法化前的阶段，编译器常会引入形如截断 + 比较的冗余指令序列。这类模式通常表现为：先对一个宽位宽寄存器执行截断操作 (G_TRUNC)，再将截断结果与零进行比较 (G_ICMP)。该现象多源于高层 IR 中的类型收缩、布尔表达式转换以及通用化指令拆分策略。

从语义角度分析，若被截断寄存器的高位仅由符号扩展产生，则这些高位在逻辑上并不携带有效信息。在这种情况下，对截断结果进行零比较，与直接对原始宽位宽值进行零比较在语义上是等价的。基于这一观察，本文在合法化前引入了针对冗余截断 + 比较模式的合并优化。

该优化首先利用 KnownBits 分析对被截断操作数进行位级信息推断，判断其高位是否完全由符号位扩展而来；若满足该条件，则说明截断操作并不会改变数值的符号与零值判定结果。此时，优化过程将原本以截断结果为操作数的比较指令，直接重写为对宽位宽操作数与零进行比较，同时移除多余的 G_TRUNC 指令。一个典型的示例如代码块3.3和代码块3.4所示。

```

1 %0:_(s64) = COPY $d0
2 %1:_(s32) = G_CONSTANT i32 0
3 %2:_(s32) = G_TRUNC %0:_
4 %3:_(s1)   = G_ICMP slt, %1:_, %2:_
5

```

Listing 3.3 CSE 优化前

```

1 %0:_(s64) = COPY $d0
2 %1:_(s32) = G_CONSTANT i32 0
3 %3:_(s1)  = G_ICMP slt, %0:_, %1:_
4
5

```

Listing 3.4 CSE 优化后

通过这一变换，不仅可以减少一条通用中间指令和对应的虚拟寄存器定义，还能显著简化数据流依赖关系，使比较操作更贴近硬件语义。在 DSP 架构下，该优化有助于后续指令选择阶段更直接地匹配硬件比较指令与条件跳转指令，从而提升代码密度并降低执行延迟。

2. 拓展下推优化

在 GMIR 中，算术指令往往作用于经过符号扩展 G_SEXT 或零扩展 G_ZEXT 后的操作数。这种形式虽然在语义上是正确的，但在实际工程中会引入多余的扩展指令，导致指令数量的增加及后续目标指令匹配与融合的成功率的降低。为此，本文引入一种 EPO (Extension Pushing Optimization，扩展下推优化)，优化通过重写算术运算与扩展指令的组合形式，将算术操作下推至扩展之前，从而获得更加规范、紧凑的中间表示。

考虑如下典型模式：

```
add(zext x, zext y)
```

该形式在 GMIR 中表现为两个扩展指令后接一条算术指令。由于 DSP 指令集中包含带扩展的算术指令，允许在较小位宽上直接完成算术运算，再对结果进行一次统一扩展。因此，将算术操作推到扩展之前能够减少冗余的扩展指令数量，降低指令条数。除此之外，还能形成更规范的 IR 结构，提升指令选择阶段的模式匹配成功率。

该优化针对 ADD 和 SUB 指令，适用于零扩展与符号扩展两种情形，其核心重写规则可概括为：

$$\text{op}(\text{ext } x, \text{ ext } y) \Rightarrow \text{ext}(\text{op}(x, y))$$

其中， op 表示算术操作 ADD 或 SUB， ext 表示扩展操作 ZEXT 或 SEXT。需要注意的是，变换过程中必须保证扩展类型的一致性，并在符号扩展场景下正确维护符号语义。在 Combiner 中，该优化通过参数化的 TableGen 规则进行描述。其核心定义如下：

```

1 class epo_base<Instruction opcode, Instruction extOpcode>
2 : GICombineRule <
3   (defs root:$root),
4   (match (extOpcode $ext1, $src1):$ExtMI,
5    (extOpcode $ext2, $src2),
6    (opcode $dst, $ext1, $ext2):$root,
7    [{ return matchEPO(*${root}, MRI, $dst, $src1, $src2); }]),
8   (apply [{,
9     applyEPO(*${root}, MRI, B,
10    ${ExtMI}->getOpcode() == TargetOpcode::G_SEXT, $dst, $src1, $src2);
11  }]);
12 >;

```

Listing 3.5: EPO 的 TableGen 规则

该规则通过模板参数抽象了算术操作类型 ADD/SUB 以及扩展类型 ZEXT-/SEXT，从而避免重复定义多条相似规则。在匹配阶段，规则要求两个操作数均来源于相同类型的扩展指令，并由指定的算术指令使用；在应用阶段，根据扩展类型生成新的算术指令，并在必要时重新插入对应的扩展操作。该优化在语义上是安全的，其正确性基于如下事实：

- 对于零扩展 ZEXT，扩展前后的值在无符号算术意义下保持一致。
- 对于符号扩展 SEXT，在保证算术运算不发生未定义行为的前提下，先运算后扩展与先扩展后运算在数学语义上等价。

- 优化过程中不改变数据依赖关系，仅调整指令组合方式。

EPO 属于一种 IR 规范化优化，其主要作用并非直接降低执行时间，而是为后续目标相关优化与指令选择提供更优的输入形式。在 DSP 架构上，该优化提升了复合指令匹配的成功率，并间接减少了最终生成代码中的指令数量。

3.3 合法化后优化策略

合法化之后，所有 GMIR 指令均已被转换为目标架构支持的合法形式，指令的类型、操作数位宽及内存访问方式均受到严格约束。此时优化的重点转向目标相关的性能优化与代码质量提升，主要包括以下几类：

- 目标指令匹配与替换优化，如将合法化后形成的指令序列进一步合并为 DSP 架构提供的伪指令或复合指令，以提升执行效率。
- 指令调度友好的局部合并，通过消除多余拷贝、简化寄存器间的数据搬移，降低寄存器压力，为后续寄存器分配与指令调度创造更优条件。
- 面向代码密度和执行效率的微优化，如立即数折叠、访存地址计算合并以及针对硬件流水线特性的局部重写。

由于合法化后指令已经紧密贴合目标架构，这一阶段的优化虽在语义变换自由度上受限，但能够更直接地转化为实际的性能收益。

3.3.1 立即数装载优化

在指令选择阶段，G_CONSTANT 和 G_FCONSTANT 需要被转化为目标指令集可执行的装载立即数序列。对于 DSP 后端而言，指令采用固定 32 位编码格式。由于指令由操作码与操作数组成，且立即数字段的编码宽度通常不超过 16 位，单条指令中可用于表示立即数的位数受到严格限制，因此 32 位常量一般无法通过单条指令直接完成装载。因此，对于 32 位立即数，若不加区分地统一采用长立即数装载伪指令 MOVI，则会引入额外的指令开销。

针对上述问题，本文结合 DSP 平台的 ISA 特性，在全局指令选择阶段对立即数装载过程进行定制化优化。核心思想是根据常量的位型特征对立即数进行分类，并优先选择编码长度更短、执行开销更低的装载形式，从而在保证语义正确性的前提下减少冗余指令。以 32 位立即数为例，可将其划分为以下三类情况。

1. 立即数的高 16 位全 0

当立即数的高 16 位全 0 时，常量可视为一个零扩展的 16 位无符号数，只需要装载低 16 位即可。由于硬件指令集的限制，即使高 16 位全为 0，也无法省略

对高位赋值的操作。为此，本文引入基于零寄存器的装载策略，通过设置一个零寄存器并将零寄存器作为源操作数，结合立即数加法或专用低位装载指令来完成常量生成，从而避免使用通用的长立即数序列。

```

1 MOVIGH GR2 0x0;
2 MOVIGL GR2 0x1;
3

```

Listing 3.6 CSE 优化前

```

1 ADDI ZERO 0x1;
2
3

```

Listing 3.7 CSE 优化后

2. 立即数的高 17 位全 1

当立即数的高 17 位全为 1，即为可用 16 位有符号立即数表示的常量时，可以利用 MOVIGLX 指令来实现将一个 16 位的有符号立即数进行符号位扩展至 32bit 后复制到 rd 中。于是当常量满足 $\text{Imm} \in [-2^{15}, 2^{15} - 1]$ 时，可将 imm16 符号扩展到 32 位写入 Rd。

```

1 MOVIGH GR2 0xFFFF;
2 MOVIGL GR2 0xF123;
3

```

Listing 3.8 CSE 优化前

```

1 MOVIGLX GR2 0xF123;
2
3

```

Listing 3.9 CSE 优化后

3. 其他情况

对于不满足上述情况的立即数，由于无法通过单条指令完成装载，只能使用伪指令拓展的长立即数装载方式，伪指令会在后续阶段展开为 MOVIGH 和 MOVIGL 两条机器指令。其中 MOVIGH 装载立即数的高 16 位到寄存器高半区，而 MOVIGL 装载立即数的低 16 位到寄存器低半区。

上述优化策略在指令选择阶段完成判定与映射，避免了在后续低层次优化中再对立即数序列进行修正。通过对立即数位型的精细化分类，该方法在不增加指令选择复杂度的前提下，有效减少了冗余指令的生成，为后续寄存器分配与指令调度提供了更紧凑的输入形式。该优化能够在控制流密集与常量使用频繁的程序中显著降低代码尺寸，并对整体执行效率产生积极影响。

3.3.2 乘法优化

对于乘数为编译期常量的整数乘法，直接生成乘法指令不是最优选择。一方面，整数乘法的硬件实现通常具有较高的延迟和功耗；另一方面，许多常量乘法

在算术上可以等价地分解为移位与加减操作，从而以更低的执行成本完成相同计算。基于这一考虑，本文在合法化后阶段对G_MUL指令引入常量乘法强度削减优化。当乘数为满足特定形式的编译期常量时，编译器将乘法操作替换为由移位G_SHL与加法G_ADD或减法G_SUB组成的等价指令序列。

1. 理论基础

从算术等价关系来看，若常量 C 满足以下形式之一：

$$C = 2^n \pm 1 \quad (3.1)$$

或

$$C = (2^n \pm 1) \times 2^m \quad (3.2)$$

则乘法运算可以通过移位与加减运算等价实现：

$$x \times (2^n + 1) = (x \ll n) + x \quad (3.3)$$

$$x \times (2^n - 1) = (x \ll n) - x \quad (3.4)$$

$$x \times ((2^n \pm 1) \times 2^m) = ((x \ll n) \pm x) \ll m \quad (3.5)$$

这些变换在保持程序语义完全一致的前提下，能够显著降低指令执行代价，并减少对乘法硬件单元的依赖。

2. 匹配阶段

在匹配阶段，优化逻辑以G_MUL指令为目标，对其操作数及常量特征进行系统分析，整体匹配流程如算法4所示。具体而言，该阶段主要包括以下三个步骤：

1. 常量识别：通过常量传播与等价查询机制，判断右操作数是否为编译期常量，获取乘法右操作数的常量值。
2. 位型分析：利用位运算分析常量的二进制结构，判断其是否满足上述形式。
3. 约束检查：为避免干扰后续指令融合或破坏已存在的扩展语义，匹配阶段还会检查源操作数是否仅被当前指令使用以及排除可能影响符号扩展、零扩展或寄存器重用的场景。

Algorithm 4: Constant-Multiply Strength Reduction

Input: MachineInstr MI
Output: InstructionMapping or fail

```

1 if  $opcode(MI) \neq G\_MUL$  then
2   | return fail;
3 end
4 if  $RHS$  is not an integer constant then
5   | return fail;
6 end
7  $C \leftarrow sext(const(RHS));$ 
8  $TZ \leftarrow countr\_zero(C);$ 
9  $C' \leftarrow ashr(C, TZ);$ 
10 if  $C' > 0$  then
11   | if  $(C' - 1)$  is power of two then
12     |   Rewrite as  $(x \ll \log_2(C' - 1)) + x;$ 
13   | else
14     |   if  $(C' + 1)$  is power of two then
15       |     Rewrite as  $(x \ll \log_2(C' + 1)) - x;$ 
16     |   end
17     |   else
18       |     | return fail;
19     |   end
20   | end
21 else
22   | if  $(-C' + 1)$  is power of two then
23     |   Rewrite as  $x - (x \ll \log_2(-C' + 1));$ 
24   | else
25     |   if  $(-C' - 1)$  is power of two then
26       |     Rewrite as  $-((x \ll \log_2(-C' - 1)) + x);$ 
27     |   end
28     |   else
29       |     | return fail;
30     |   end
31   | end
32 end
33 return success;

```

3. 重写阶段

在重写阶段，编译器根据匹配到的常量形式，生成对应的移位与加减指令序列，并替换原有的 G_MUL 指令。

3.4 本章小结

本章主要分析并设计了面向 DSP 架构的全局指令选择优化策略。首先，针对 GlobalISel 框架在 DSP 后端的性能瓶颈，提出了包括 CSE、KnownBits Analysis 和 Combiner 等技术的优化方案，旨在减少冗余指令，提升代码执行效率和硬件利用率。通过对这些优化技术的理论基础和实现机制进行详细分析，揭示了优化策略在不同阶段的应用方式。

在具体优化技术分析中，介绍了 CSE 如何通过消除重复计算、减少寄存器占用来精简代码，已知比特分析如何为常量传播和指令简化提供支持，合并优化则通过删除冗余指令、重构指令序列，提高代码密度和执行效率。此外，本章还针对 DSP 架构的特点，设计了符合目标架构需求的优化策略，如立即数装载优化和乘法优化，以进一步提升指令生成效率和硬件资源利用率。

最后，本章通过系统的优化设计与实现，展示了如何通过多策略协同来提升全局指令选择框架在 DSP 架构下的代码生成性能，为后续的代码生成与优化工作奠定了坚实的基础。

第 4 章 测试评估平台设计与实现

4.1 DSP 编译器测试现状与改进

随着 DSP 编译器后端功能的持续扩展，尤其是全局指令选择框架及相关优化策略的引入，编译器在指令匹配、寄存器组选择以及指令合法化等阶段的行为呈现出更强的输入敏感性和组合多样性。相较于传统的基于 DAG 的指令选择方案，全局指令选择虽然在实现结构上更加模块化，但其指令选择结果由多个 Pass 的协同决策共同决定，整体行为对输入程序结构更加敏感。这种全局视角和延迟决策机制显著扩大了可观察的指令选择行为空间，使得潜在缺陷更具隐蔽性，也对测试体系在输入覆盖和结果分析方面提出了更高要求。

DSP 工具链虽然已经建立了基本完善的 CI (Continuous Integration, 持续集成) 测试流程，能够在一定程度上保证主分支代码的功能正确性。然而，该测试体系主要形成于传统编译流程阶段，其设计目标更多集中于功能可用性验证，对于全局指令选择及其相关优化在复杂场景下的正确性和性能稳定性缺乏针对性支持。随着 GlobalISel 在 DSP 工具链中的逐步落地，现有测试体系在测试结果可分析性、性能回归检测以及复杂输入场景覆盖等方面逐渐暴露出不足，有必要对其进行系统分析。

4.1.1 测试流程现状

DSP 工具链拥有独立的线上代码托管与协作开发平台。开发者在提交代码后，CI 系统会自动触发一套预定义的测试流程，对最新提交的工具链代码进行编译、链接以及部分运行验证。只有在测试全部通过并经项目负责人审核确认后，相关提交才能被合并至主分支。

该测试流程在当前开发阶段发挥了积极作用。一方面，它能够在代码提交阶段及时发现编译失败、链接错误及明显的功能性缺陷，从流程上保证主分支代码的基本可用性；另一方面，通过统一的 CI 流水线规范了代码提交流程，减少了人工测试的参与程度，在一定程度上降低了开发成本。

总体来看，尽管现有测试体系在功能正确性验证方面具备基础能力，适用于发现显性错误和阻断明显缺陷进入主分支。然而，该体系主要以是否通过测试为评判标准，对编译器优化效果、性能变化以及复杂场景下的鲁棒性缺乏系统性支撑，难以满足 DSP 工具链对高性能与高可靠性的长期发展需求。具体体现

在测试用例覆盖率不足以及测试产物可读性与分析能力不足这两个问题上。

1. 测试集来源单一，场景覆盖局限于常见路径，缺乏随机化测试机制

测试用例的覆盖完整性直接决定了 DSP 工具链发现潜在缺陷的能力。然而，现有 DSP 工具链在测试集构成与场景覆盖方面仍存在明显不足，难以对核心编译流程的鲁棒性进行充分验证。从测试来源来看，当前 DSP 工具链的测试集主要包括以下三类：

- 基于库函数的手写测试用例：该类测试通常围绕基础功能验证设计，重点覆盖 C 标准库、数学库以及运行时支持库等核心库函数的调用场景。
- 移植的开源 Benchmark 程序：以通用计算或嵌入式系统性能评估为目标，例如 Embench 等，用于衡量工具链在典型应用负载下的性能表现。
- 已知缺陷的最小复现样例：在开发过程中针对已暴露的问题提炼出的最小测试用例，主要用于回归测试，防止同类缺陷再次引入。

上述测试用例在验证工具链基础功能正确性以及复现已知缺陷方面具有一定价值，但其整体覆盖范围仍然较为有限。

一方面，测试样例的设计高度依赖人工经验，往往集中于库函数的常规调用路径和标准输入输出场景，对于极端数据类型、异常输入以及复杂控制流等边界条件覆盖不足。这种以“常见路径”为主的测试策略，使得许多非常规但合法的程序形态难以被有效触达。

另一方面，DSP 工具链内部的关键编译阶段缺乏针对复杂场景的系统性测试，尤其是在指令选择、寄存器分配和指令合法化等核心模块中。例如，多寄存器组之间的交叉分配、复杂指令序列的模式匹配，以及非标准位宽数据的合法化转换等场景，往往涉及大量条件组合和路径分支，人工构造测试用例难以穷尽所有可能情况。

这种覆盖不足直接带来两类风险：其一，工具链在通用应用场景下表现正常，但在 DSP 的特定应用场景中容易暴露出逻辑错误；其二，编译器内部的条件触发型缺陷难以及早发现。这类缺陷通常仅在特定指令组合或数据模式下才会被触发，例如指令选择阶段的模式匹配漏判、寄存器分配中的资源冲突，或合法化转换引入的语义偏差。一旦这些问题未能在测试阶段暴露，往往会在实际应用中引发程序崩溃或性能异常，后果较为严重。

对于引入 GlobalISel 后的 DSP 编译流程而言，上述问题进一步被放大。由于指令选择结果往往由多个 Pass 的协同决策共同决定，例如寄存器组选择与指令合法化之间的相互影响、不同合法化路径对后续指令匹配结果的约束关系等，这类行为通常只会在特定输入结构和数据特征组合下才会显现。依赖人工经验

构造测试样例难以覆盖这些复杂交互场景，导致部分潜在缺陷长期隐藏，直至在真实 DSP 应用中才被触发，从而带来较高的工程风险。

2. 测试产物可读性与分析能力不足

现有测试体系在测试结果的表达和分析能力方面同样存在明显不足。当前 CI 流程主要以测试是否通过作为判定标准，测试产物缺乏对编译器后端行为的量化描述，未系统记录程序执行周期、代码尺寸、指令打包效率等关键性能指标，也未对生成的汇编代码进行结构化管理。这使得测试结果只能用于验证功能正确性，而无法反映全局指令选择及相关优化策略对代码性能的实际影响。

此外，测试结果未进行历史化存储和跨版本管理，开发者难以对不同提交之间的性能变化进行对比分析，也无法从整体上观察工具链性能随时间演进的趋势。随着 CI 测试记录不断积累，大量中间提交和实验性版本产生的结果进一步加剧了测试数据的冗余，显著提高了有效信息筛选和问题定位的成本。

综上所述，现有测试体系在输入覆盖能力和结果分析深度方面均难以支撑全局指令选择框架下编译器后端行为的系统性验证，已无法满足 DSP 工具链在高性能和高可靠性方向持续演进的需求。

4.1.2 测试体系优化需求与技术路径

基于前述分析可以看出，随着全局指令选择框架及相关优化策略在 DSP 工具链中的引入，单纯依赖人工构造测试样例以及是否通过这一单一判定标准，已难以对编译器后端行为进行系统性验证。为保障全局指令选择在复杂应用场景下的功能正确性和性能稳定性，现有测试体系亟需从测试输入覆盖能力和测试结果分析深度两个层面进行针对性优化。

在测试输入层面，测试体系需要具备更强的输入多样性和场景覆盖能力，以支撑全局指令选择框架下条件触发型行为的验证。由于指令选择、寄存器组绑定和合法化等阶段的决策往往依赖于输入程序的控制流结构、数据分布以及运算模式，人工设计测试样例难以系统覆盖所有潜在组合路径。因此，有必要引入随机测试生成机制，通过自动化方式生成大量具有结构多样性和数据随机性的测试程序，从而覆盖人工测试难以触及的边缘场景。这种随机测试方式能够有效暴露隐藏在特定输入条件下的潜在缺陷，提升编译器后端对复杂输入场景的整体鲁棒性。

在测试结果层面，仅验证程序功能是否正确已无法满足全局指令选择优化验证的实际需求。全局指令选择及其相关优化策略的引入，往往不会直接导致功能错误，而是可能以性能退化、代码尺寸膨胀或指令级行为异常等形式体现出来。为准确评估其对 DSP 架构性能的影响，有必要构建一套能够自动采集、存储和分析性能指标的评估平台，对程序执行周期、代码尺寸及指令级行为进行

量化记录，并支持跨版本的对比分析和趋势观察。通过将测试结果结构化存储并以可视化方式呈现，可有效降低性能分析门槛，辅助开发者快速定位性能回退和异常变化。

综上所述，面向全局指令选择框架的 DSP 工具链测试体系，应当以随机测试驱动的高覆盖输入验证为基础，以性能指标采集与可视化分析为核心支撑，构建一套覆盖广、可量化、可追溯的测试评估机制。基于上述需求，本文在后续章节中设计并实现了一套集随机测试生成与性能评估于一体的测试评估平台，为全局指令选择及其优化策略的验证提供系统化支撑。

4.1.3 随机测试生成技术与生成器选型

随机测试生成技术是自动化测试领域的核心技术之一，其核心原理是依托随机或伪随机算法，按照预设的语法、语义规则及测试目标生成大规模多样化的测试输入，以此验证软件系统在不同输入场景下的正确性、稳定性与鲁棒性。相较于传统人工设计测试用例的方式，随机测试能够在更短时间内覆盖更广泛的输入空间，有利于发现隐藏较深、难以通过经验手段构造的边界错误和组合缺陷。

在编译器测试场景中，随机测试技术通常通过专用的随机测试生成器加以实现。这类工具通过随机组合表达式、控制流结构、数据类型以及运算符等语言构造要素，来生成大量结构各异的测试程序，并将其作为输入交由编译器处理，从而验证编译器前端解析、优化过程以及后端代码生成等多个阶段的正确性与健壮性。

在编译器随机测试工具的发展过程中，CSmith 是较早被广泛应用的一类 C 语言随机程序生成器，核心目标是发现编译器在处理常规 C 程序时可能存在的逻辑错误。其特点是生成的随机程序严格遵循 C99 标准且严格规避 UB (Undefined Behavior, 未定义行为)，从而确保测试结果可复现。CSmith 在验证编译器基本语义一致性和防止功能性错误方面发挥了重要作用，但其测试目标主要聚焦于功能是否正确，对编译器在高性能优化场景下的行为覆盖能力相对有限。

随着现代处理器架构复杂度的提升以及编译器优化策略的不断增强，仅依赖功能正确性验证已难以满足编译器工具链的测试需求。在这一背景下，YARPGen (Yet Another Random Program Generator) 被提出，其设计初衷正是针对 CSmith 在性能相关测试方面覆盖不足的问题，重点面向编译器后端的高性能优化阶段开展随机化测试。

与以语义正确性为核心目标的随机测试生成器不同，YARPGen 则更加关注编译器在高性能优化场景下的行为表现，重点发现那些在功能层面正确、但在性能或架构适配方面存在问题的深层次缺陷。它可以在不依靠动态检查的情况下，

使用生成规则来生成没有未定义行为的表达式，同时提升了生成代码的多样性，触发更多编译器优化的可能。YARPGen 在代码生成策略上倾向于构造密集型循环、数组连续访问、数据并行计算以及向量运算等代码模式，从而有针对性地覆盖循环向量化、指令级并行、缓存优化以及指令调度等编译器后端的关键优化阶段。

在语言和架构支持方面，YARPGen 不仅支持 C 语言，还扩展支持 C++ 的部分核心特性，并能够生成面向 SIMD、向量指令等现代处理器架构扩展的代码。这一特性使其在测试面向高性能计算、嵌入式 DSP 以及多媒体处理场景的编译器工具链时具有一定的实用价值。

基于上述特点，本文在测试体系设计中选择基于 YARPGen 并结合 DSP 架构特性进行定制化扩展，构建面向全局指令选择及其相关优化策略的随机测试输入生成机制。通过生成具有高性能特征的随机测试程序，测试体系能够更有效地验证编译器在指令选择、寄存器组绑定以及指令合法化等阶段的行为表现，为后续性能评估与回归分析提供可靠的测试基础。

4.2 平台设计

4.2.1 平台整体设计

为解决 4.1.1 节提到的测试覆盖不足与结果分析能力弱等问题，本文设计并实现一套面向 DSP 编译器的性能测试评估平台。平台整体由测试子系统与评估子系统两部分构成，前者在原有 CI 测试平台上进行新功能的添加与旧架构的重构，后者则为全新搭建的独立模块。

测试子系统的核心改进在于移植并定制随机测试生成器 YARPGen，为其添加针对 DSP 指令集架构的内建函数 Intrinsic 及必要的环境适配逻辑，使其能够在每次测试触发时自动生成一批随机样例，并与原有固定测试集并行执行。

评估部分与线上 CI 测试平台连接，通过脚本实现编译流程的自动化触发：当编译器代码合并至仓库时。编写脚本使得每次编译器的代码合并到仓库中都会自动在线上模拟器平台运行，同时在连接到远程服务器上的长期工作的芯片运行；测试完成后将结果返回到 Gitlab 的 Perf 仓库（便于管理以及迁移）；后端定时拉取 Perf 仓库并更新数据库，当前端访问时返回查询的结果，并在前端进行展示，可以直观地反映新优化带来的性能提升或退步。

系统整体设计如图 4-1 所示，整体架构分为测试子系统和评估子系统两部分。测试子系统完成从测试样例生成、编译到执行的全自动化流程；评估子系统则通过前端、后端与数据库的协作，实现测试结果的结构化存储、查询、对比分析与可视化展示。两个子系统在功能上相互解耦，通过 Perf 仓库进行数据交互。该

设计既保证了测试流程的高效运行，又提升了测试结果的可分析性与可追溯性。

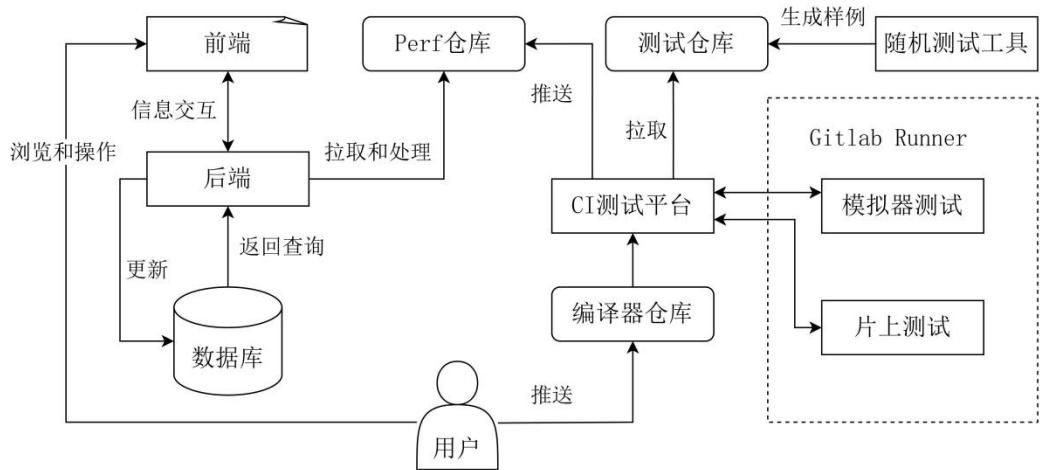


图 4-1 系统整体设计图

4.2.2 测试子系统设计

测试子系统的核心目标是构建全自动的流水线，覆盖从测试用例到结果产出的全流程，具体模块与流程如下：

- 随机测试样例生成模块随机测试工具用于自动生成适配 DSP 架构的随机测试样例，重点覆盖密集循环、向量运算等性能相关场景。随机测试工具在执行测试时再生成，并将错误的样例推送至测试仓库，与其他测试样例一起被统一管理。二者共同为 CI 测试提供稳定的用例来源。
- 编译与测试模块 CI 测试平台作为调度核心，从编译器代码仓库中拉取开发者提交的最新 DSP 工具链代码，并从测试仓库中获取待执行的测试脚本、测试样例以及需要的测试工具。首先进行编译，编译成功后将测试任务分发至 Gitlab Runner，触发两类测试流程：一类是在 DSP 架构模拟器上执行的测试，用于快速验证功能正确性及基础性能表现；另一类是在实际 DSP 硬件芯片上执行的测试，用于发现硬件上的 Bug，以及获取更加准确的性能指标。
- 结果处理模块测试完成后，CI 测试平台会将所有测试结果统一归档，并推送至 Perf 仓库。归档内容不仅包括测试通过或失败状态，还涵盖性能指标数据以及生成的汇编代码，为后续评估分析提供完整、可靠的数据基础。除此之外，为了解决 4.1.2 中提到的冗余问题，设置过滤器来保证只有主分支的代码才会传到 Perf 仓库中。

4.2.3 评估子系统设计

评估部分以测试结果高效利用与直观呈现为目标，采用前端、后端和数据库协同工作的三层架构，对测试数据进行统一管理和深度分析。具体模块与流程如下：

- 后端模块负责从 Perf 仓库拉取性能报告，对原始数据进行结构化解析并将数据存储至数据库；同时，后端对外提供统一的数据接口，支持测试结果查询、跨版本对比以及性能趋势分析等功能，为前端的可视化展示提供服务支撑。
- 数据库模块作为测试结果的持久化存储中心，统一保存所有历史测试数据，支持按照芯片型号、Commit 号、优化等级、测试集以及测试样例等多维条件进行快速检索。这一设计为性能回归分析、版本对比以及长期趋势观察提供了坚实的数据基础。
- 前端模块则为用户提供直观、交互友好的可视化界面。用户可以通过前端查看单个提交版本的测试结果，对比不同提交之间的性能差异，或以时间序列方式观察性能变化趋势等。同时，前端支持灵活的数据筛选操作，例如仅查看片上测试的性能指标，相关请求由后端实时处理并从数据库中返回结果进行展示。

4.3 平台实现

4.3.1 面向 DSP 的 YARPGen 实现

YARPGen 是一个通用的编译器测试用例生成工具，其原生设计主要针对通用 CPU 架构及标准 C/C++ 运行环境。在默认实现中，YARPGen 假设目标平台具备标准的运行时库支持，并依赖 stdio 等通用接口完成程序结果的输出与校验。然而，DSP 后端在硬件特性、调试机制以及运行环境方面均与通用处理器存在显著差异，这使得原生 YARPGen 难以直接应用于 DSP 编译器流水线的自动化验证。

为使随机测试生成机制能够适配到 DSP 编译器后端，本文对 YARPGen 进行了针对 DSP 架构的定制化扩展。整体改动遵循可复用和可扩展的设计原则，在保留 YARPGen 原有随机生成能力的基础上，引入对 DSP 平台特性的适配支持。具体改动主要体现在以下几个方面。

1. 引入 DSP 专用调试头文件

DSP 编译器的测试与验证依赖架构专用的调试与输出接口，而非标准的 stdio 输出机制。为保证生成的随机测试程序能够在 DSP 环境中正确输出运行

结果，需在测试用例中引入特定头文件以支持调试输出和结果对比。在 YARPGen 的 emitCheckFunc 阶段，通过条件编译区分通用平台和 DSP 平台，为 DSP 环境添加专用调试头文件，并声明调试输出相关的函数接口。相关代码片段如下所示：

```

1 void emitCheckFunc( std :: ostream &stream ) {
2     std :: ostream &out_file = stream ;
3     out_file << "#ifdef DSP_VALIDATION\n";
4     out_file << "#include <swift_debug.h>\n";
5     out_file << "#else\n";
6     out_file << "#include <stdio.h>\n\n";
7     out_file << "#endif\n";
8     .....
}
```

其中，DSP_VALIDATION 用于标识当前测试是否运行于 DSP 验证环境。该宏由测试流水线在编译阶段统一注入，从而保证同一套测试生成逻辑可同时服务于通用平台与 DSP 平台，避免代码分叉维护。通过该方式，YARPGen 在 DSP 模式下能够调用调试头文件中提供的调试输出接口，实现与 DSP 仿真器或硬件调试环境的对接。

2. 添加 DSP 专用调试输出逻辑

在通用平台中，YARPGen 通过标准输出对程序执行结果进行校验；而在 DSP 平台上，测试结果需要通过专用调试通道输出，以便测试框架进行采集与比对。为此，在 YARPGen 的 emitMain 阶段中，通过条件编译为 DSP 环境下添加调试函数调用，用于输出校验结果。

3. 引入目标架构选项与条件编译参数

原生 YARPGen 只区分生成语言，并不区分目标后端架构。为支持 DSP 平台相关逻辑，本文在 YARPGen 中引入了目标架构选项，并通过条件编译参数控制测试生成与代码发射阶段的具体行为。这一改动为后续引入 DSP 架构的 Intrinsic 提供了基础设施支持。

4. 添加 DSP 架构的 Intrinsic

为了更有效地测试 DSP 编译器在指令选择与优化阶段的行为，本研究在 YARPGen 中引入了 DSP 架构专用 Intrinsic，并将其纳入随机测试生成范围。由于 YARPGen 不支持 Intrinsic，于是需要拓展其功能，在 IRNode 中引入新的 Intrinsic 选项，并根据操作数数量进行分类，如图4-2所示。以 SIN 函数为例，对 YARPGen 的 IRNode 生成逻辑进行了扩展。

- **createHelper**: 核心作用是创建 SIN 函数调用表达式，包括生成算术表达式参数、处理布尔类型转换，最终返回封装后的 IntrinsicExpr 指针。这是

YARPGen 中表达式生成逻辑的关键部分，用于随机生成 SIN 调用的测试用例。

- `propagateType`: 核心作用是对 SIN 函数的两个参数表达式进行类型推导、类型提升和类型统一，最终确定 SIN 函数调用表达式的返回值类型。这是代码生成过程中保证类型安全和语法正确性的关键步骤。
- `evaluate`: 在测试生成或运行时阶段计算 SIN 函数调用的结果，包括处理参数的类型转换、未定义行为判断，以及最终返回计算后的标量值。这是 YARPGen 中表达式求值逻辑的关键部分，用于确定 SIN 函数调用的实际结果值。
- `emitCDefinitionImpl`: 核心作用是为 C 语言环境生成 SIN 函数的宏定义实现。因为 C 语言标准库中没有通用的 SIN 函数，该方法通过输出 C 语言的语句表达式宏，为 SIN 提供兼容 C 语言的实现，保证生成的测试用例在 C 环境下能正常编译运行。通过上述的定制化扩展，YARPGen 被成功适配到 DSP 编译器的流水线测试场景中。基于条件编译的设计使得：同一套随机测试生成逻辑可同时服务于通用平台与 DSP 平台；DSP 专用 Intrinsic 能够被系统性纳入随机测试覆盖范围。

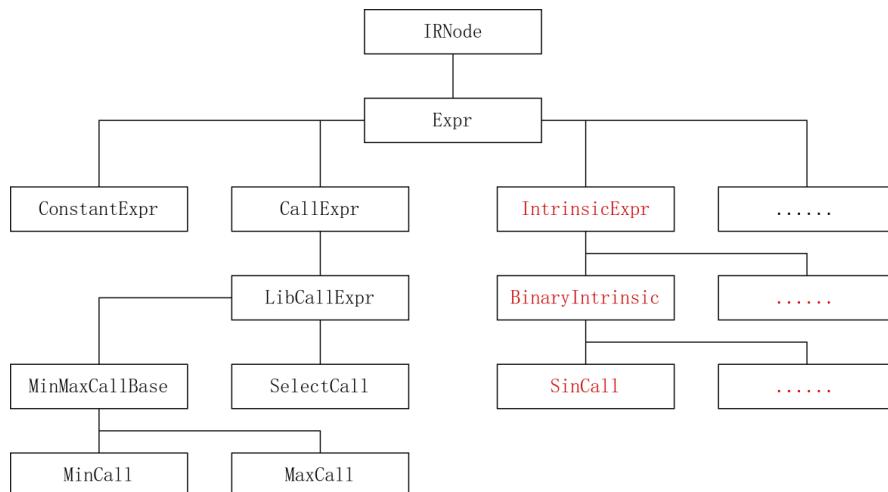


图 4-2 拓展后的 IRNode 结构图

对应的类层次结构如图4-3所示。

4.3.2 测试子系统实现

测试部分是性能测试评估平台的核心组成之一，其主要任务是在原有功能正确性验证的基础上，引入对编译器性能表现的自动化测量能力，并将测试流

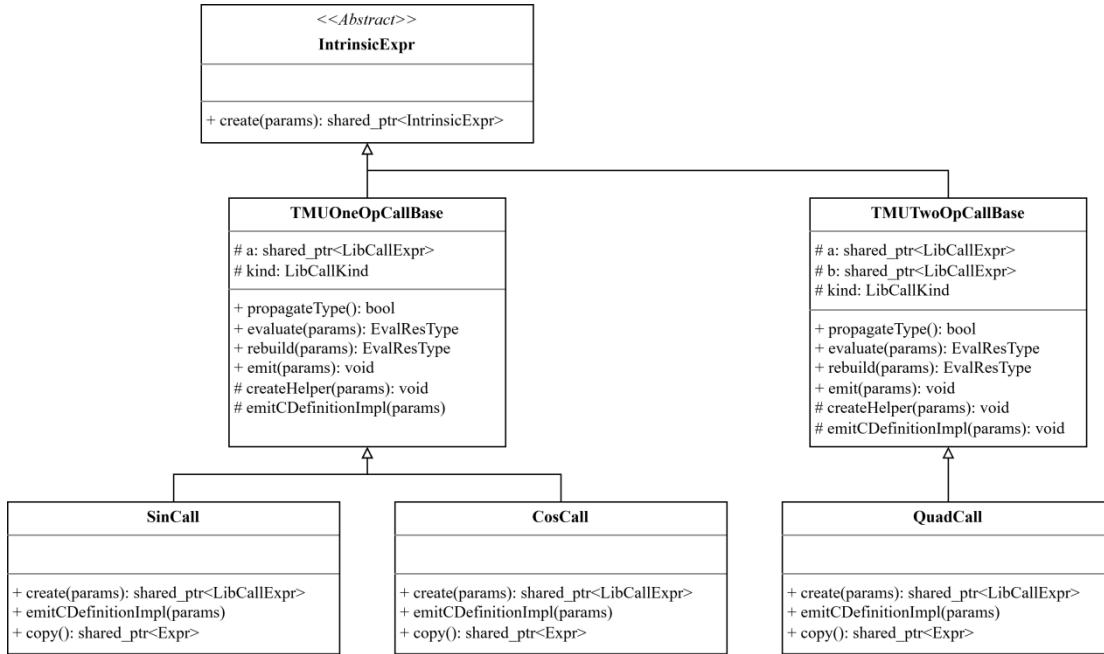


图 4-3 扩展后的类结构

程整体融入持续集成流水线中，实现性能测试的标准化与可重复执行。围绕这一目标，测试部分的实现主要包括性能指标测量支持的引入以及测试脚本逻辑的重构两个方面。

(1) 添加测量性能指标的支持衡量 VLIW 架构的 DSP 编译器性能的核心指标包括程序运行周期数、生成代码的大小、汇编代码质量以及指令的打包效率等。上述指标分别从执行效率、存储资源占用以及指令层级优化效果等角度反映了编译器后端的整体性能表现。针对不同性能指标的获取方式，测试系统采用了差异化的实现策略。代码大小的测量通过集成开源工具完成，工具通过在编译完成后对目标文件进行分析，来自动统计生成代码所占用的存储空间。汇编代码的获取则依赖于 DSP 工具链自带的反汇编器，在编译阶段结束后对目标文件进行反汇编，生成对应的汇编输出，为后续的指令级分析提供基础数据。指令打包效率由工具链的模拟器进行统计，该模拟器能够在模拟执行过程中收集指令发射和打包相关信息，从而反映编译器在指令调度与并行化方面的优化效果。

运行周期数的获取在实现上具有一定复杂性。对于模拟器环境，程序运行周期数可以在执行过程中直接由模拟器提供；然而在实际 DSP 芯片上，周期数的测量需要依赖底层驱动库所提供的硬件计数器接口。为使周期数测量能够无缝融入自动化测试流水线，平台对原有驱动库进行了精简与重构，将其裁剪为可重定向的独立文件，仅保留初始化和时间戳获取两个核心函数。在链接阶段，这些函数被统一链接至测试程序的主函数中，从而实现对程序执行前后时间戳的自动采集。通过这一方式，芯片上的运行周期数测量能够与编译、执行流程紧

紧密结合，实现流水线化和自动化。

(2) 重构测试脚本并添加 YARPGen 原先的测试脚本的逻辑是只判断样例执行正确与否，无法满足性能评估平台对多维性能数据采集的需求，为此，需要对测试脚本的整体逻辑进行重写和扩展。在新的脚本中，测试流程在完成编译与执行后，会自动触发性能数据采集模块，分别获取运行周期数、代码大小、汇编输出以及指令打包效率等指标，并将其统一整理为结构化的性能报告。

性能报告以标准化格式生成，便于后续自动解析和存储。在报告生成完成后，测试脚本会将测试结果进行打包，并自动上传至 Perf 仓库，作为后续评估和可视化分析的数据源。此外，为提升测试覆盖度，CI 脚本中还集成了 YARPGen 的随机样例生成逻辑，在每次测试触发时可自动生成一批新的随机测试程序，并与已有测试样例共同参与测试执行。通过这一机制，测试流水线不仅能够验证固定测试集上的功能和性能表现，还能够持续引入新的输入场景，提升对潜在缺陷的发现能力。

4.3.3 评估子系统实现

评估部分用于对测试阶段产生的大量性能数据进行集中管理、分析与展示，其实现采用典型的三层架构，包括前端展示层、后端服务层以及数据库存储层。各层在功能上相互解耦，在数据流转上通过标准化接口进行协作，从而保证系统的可扩展性与维护性。

1. 数据展示层的实现

数据展示层（下文简称前端）面向编译器性能分析与回归验证的实际需求，用于解决性能数据可视化难、跨版本对比成本高的问题。前端部分基于 Vue 框架实现交互界面，并结合 Apache ECharts 实现性能数据的图形化展示。通过前端界面，用户可以直观地查看单个提交版本的测试结果，对比不同提交之间的性能差异，并以时间序列形式观察性能指标的变化趋势。可视化展示方式有效降低了性能分析门槛，使开发者能够快速定位性能回退或异常波动问题。在实现过程中，前端通过组件化方式对不同功能模块进行拆分，具体模块如下：

- Dashboard：从整体层面展示 DSP 平台与竞品芯片在典型测试集上的性能对比结果，用于快速评估工具链性能水平。
- Summary：基于完整测试集统计不同提交版本下的平均代码大小与执行周期，用于观察性能随时间的整体演进趋势。
- History：针对单个测试样例，以时间序列方式展示其在不同提交下的性能变化，用于精确定位性能回退首次引入的位置。

- Compare：对任意两个提交版本进行逐样例性能对比，并支持汇编代码对照分析，用于解释性能差异的底层原因。
- Custom：支持上传本地测试结果并与主分支数据进行对比，便于在本地优化验证阶段提前发现潜在性能问题。

前端通过调用后端提供的 REST 接口获取性能数据，并在本地对数据进行初步处理和筛选，以减少不必要的数据传输。性能数据的图形化展示采用 Apache ECharts 实现，针对不同分析场景选用合适的图表类型：例如，使用柱状图展示与竞品芯片在相同测试集下执行周期数或代码大小上的对比结果，使用折线图展示执行周期数及代码大小随提交版本变化的趋势情况。通过这种方式，前端能够直观反映性能变化特征，帮助开发者快速识别性能回退或异常波动。

2. 业务逻辑层的实现

业务逻辑层（下文简称后端）是评估子系统的核心控制单元，负责性能数据的自动获取、解析建模以及对外接口服务。后端部分基于 Spring Boot 框架实现，负责性能数据解析、业务逻辑处理以及接口服务等任务。后端在启动或触发定时任务时，会从 Perf 仓库中拉取测试结果，并按照预定义的数据格式对原始 JSON 数据进行解析。解析过程中，后端会将测试环境信息（如目标处理器、优化等级、测试平台）与单个测试样例的性能指标进行关联建模，从而形成结构化的数据对象。这种建模方式使得不同维度的性能数据能够被统一管理，为后续的多条件查询和对比分析提供基础。

随后，这些数据通过 MyBatis 持久层框架映射到数据库表结构中，实现性能数据的持久化存储。为保证在多用户查询或批量数据写入场景下系统的稳定性，后端引入 Druid 数据库连接池，对数据库连接进行统一管理与复用，有效降低了连接创建和释放的开销。

在接口设计方面，后端向前端提供了一组面向性能分析任务的 REST 接口，包括按提交版本查询、跨版本对比以及历史趋势获取等功能。复杂的数据筛选与聚合逻辑均在后端完成，从而降低前端实现复杂度，提高系统整体的模块化程度。

3. 数据存储层的实现

数据存储层采用 MySQL 据库实现，用于对测试评估平台产生的性能数据进行长期持久化存储。在设计过程中，数据库并非简单保存测试输出结果，而是将每一次测试视为编译器在特定提交版本和测试环境下的一次性能状态快照，从而为性能回归分析和历史对比提供结构化支撑。

在数据建模方面，系统以提交版本、测试样例和测试环境为核心维度，对性能数据进行规范化建模，并通过外键关系维护不同实体之间的关联。这种设计

使得系统能够支持跨版本、跨优化等级以及跨测试平台的多维性能查询与对比分析。

为保证测试结果的可复现性与可解释性，每条性能记录均绑定完整的测试上下文信息，包括编译器提交版本、优化等级、目标平台以及测试样例标识，避免由于环境差异导致的性能误判。同时，数据库保留测试执行时间信息，以支持长期性能演化趋势分析和回归问题的时间定位。

考虑到性能测试数据会随着工具链的持续迭代不断积累，数据库在设计时对常用查询维度建立索引，以提升历史数据查询和对比分析的效率。该数据存储机制为评估子系统提供了稳定、可扩展的数据基础，有效支撑了后续章节中针对全局指令选择性能影响的实验分析。

4.4 本章小结

本章围绕 DSP 编译器在引入 GlobalISel 及相关优化策略后面临的测试挑战，分析了现有 CI 测试流程在输入覆盖与结果分析能力方面的不足：一方面，测试集来源相对单一且高度依赖人工经验，难以覆盖指令选择、寄存器组绑定与合法化等阶段中由多 Pass 协同决策触发的复杂交互场景；另一方面，缺乏对代码大小、执行周期、汇编质量与指令打包效率等关键性能指标的系统记录与跨版本对比能力，难以支撑性能回归检测与趋势分析。

针对上述问题，本章提出并实现了一套面向 DSP 工具链的测试评估平台。平台在架构上由测试子系统与评估子系统构成，并通过 Perf 仓库完成解耦的数据交互：测试子系统在原有 CI 流水线基础上引入随机测试生成机制，移植并定制 YARPGen 以适配 DSP 运行环境与架构特性，新增 DSP 调试输出、目标选项与 Intrinsic 扩展能力；同时重构测试脚本，实现代码大小、运行周期、反汇编输出与指令打包效率等多维性能数据的自动采集与结构化归档。评估子系统采用前端展示层、后端服务层与数据库存储层的三层架构，后端定时拉取并解析 Perf 仓库中的结果数据，完成建模入库与 REST 接口服务；前端基于可视化组件实现单版本查看、跨版本对比与历史趋势分析等功能，降低性能分析与回归定位成本。

通过上述设计与实现，本章形成了一套覆盖样例生成—流水线执行—性能指标采集—结果持久化—对比分析展示的完整测试评估流程，为后续对 GlobalISel 及其优化策略的性能影响开展定量实验与回归定位提供了统一的数据来源与可复现的工程支撑。

第 5 章 实验结果与性能分析

5.1 全局指令选择正确性验证

本节通过一系列有代表性的 LLVM IR 示例程序，对基于 GlobalISel 框架的 DSP 后端代码生成流程进行正确性验证。本节围绕 LLVM IR 到 GMIR 的转换、指令合法化、寄存器组选择以及机器指令选择四个环节展开验证，通过对中间结果进行对比分析，系统性地验证了全局指令选择流程在 DSP 后端中的功能正确性与稳定性。

5.1.1 基本指令正确性验证

本节旨在验证 DSP 后端基本指令的正确性，重点关注算术运算指令与内存访问指令的指令选择结果。基于此，本节构造了一个示例程序一，该示例程序包含函数参数传递、内存读写操作以及整数加法与乘法运算等典型计算模式，能够覆盖 DSP 后端中常见的指令选择路径。

由于 LLVM IR 具有良好的目标无关性，且从 C 语言到 LLVM IR 的转换过程并非本文关注重点，本文在分析过程中省略前端生成 IR 的具体细节，直接展示从 LLVM IR 到目标机器码的转换流程，以突出指令选择与后端代码生成阶段的行为特征。

1. LLVM IR 输入

示例程序一的 LLVM IR 如代码块5.1所示。

```
1 define dso_local i32 @testArith(i32 noundef %a, i32 noundef %b) {
2     entry:
3     %a.addr = alloca i32
4     %b.addr = alloca i32
5     store i32 %a, ptr %a.addr
6     store i32 %b, ptr %b.addr
7     %0 = load i32, ptr %a.addr
8     %1 = load i32, ptr %b.addr
9     %add = add i32 %0, %1
10    %2 = load i32, ptr %a.addr
11    %mul = mul i32 %add, %2
12    ret i32 %mul
13 }
```

Listing 5.1: 示例程序一的 LLVM IR 表示

2.GMIR 生成阶段

如图5-1所示，在 GMIR 生成阶段，LLVM IR 被转换为与目标架构无关的 GMIR。从中间结果可以观察到：

- add 与 mul 分别被映射为 G_ADD 与 G_MUL。
- load 与 store 分别被映射为 G_LOAD 与 G_STORE。
- 函数参数通过 COPY 指令从物理寄存器拷贝到虚拟寄存器。

```
# *** IR Dump After IRTranslator (irtranslator) ***:
# Machine code for function testArith: IsSSA, TracksLiveness
Frame Objects:
| fi#0: size=4, align=4, at location [SP]
| fi#1: size=4, align=4, at location [SP]
Function Live Ins: $a0, $a1

bb.1.entry:
liveins: $a0, $a1
%0:_(s32) = COPY $a0
%1:_(s32) = COPY $a1
%2:_(p0) = G_FRAME_INDEX %stack.0.a.addr
%3:_(p0) = G_FRAME_INDEX %stack.1.b.addr
G_STORE %0:_(s32), %2:_(p0) :: (store (s32) into %ir.a.addr)
G_STORE %1:_(s32), %3:_(p0) :: (store (s32) into %ir.b.addr)
%4:_(s32) = G_LOAD %2:_(p0), debug-location !17 :: (dereferenceable load (s32) from %ir.a.addr); ./arith.c:2:13
%5:_(s32) = G_LOAD %3:_(p0), debug-location !18 :: (dereferenceable load (s32) from %ir.b.addr); ./arith.c:2:17
%6:_(s32) = nsw G_ADD %4:_, %5:_, debug-location !19; ./arith.c:2:15
%7:_(s32) = G_LOAD %2:_(p0), debug-location !10 :: (dereferenceable load (s32) from %ir.a.addr); ./arith.c:2:22
%8:_(s32) = nsw G_MUL %6:_, %7:_, debug-location !21; ./arith.c:2:20
$v0 = COPY %8:_(s32), debug-location !22; ./arith.c:2:5
RET implicit $lr, implicit $v0, debug-location !22; ./arith.c:2:5

# End machine code for function testArith.
```

图 5-1 示例程序一的 GMIR 生成阶段 Dump 图

该结果表明，LLVM IR 到 GMIR 的转换过程保持了原有运算语义与数据依赖关系，为后续后端处理提供了正确的输入。

3. 合法化阶段

指令合法化阶段的主要任务是将 GMIR 中不被目标架构直接支持的指令形式转化为架构可直接处理的合法指令。在本示例程序中，32 位的 G_ADD、G_MUL、G_LOAD 以及 G_STORE 等指令均为 DSP 架构原生支持的操作，因此并未触发类型扩展、操作数分裂或指令拆解等复杂合法化过程。如图5-2所示，合法化前后指令结构保持一致，这表明当前 DSP 合法化规则能够正确处理基本的算术运算与内存访问指令。

```

# *** IR Dump After Legalizer (legalizer) ***:
# Machine code for function testArith: IsSSA, TracksLiveness, Legalized
Frame Objects:
  fi#0: size=4, align=4, at location [SP]
  fi#1: size=4, align=4, at location [SP]
Function Live Ins: $a0, $a1

bb.1.entry:
  liveins: $a0, $a1
  %0:_(s32) = COPY $a0
  %1:_(s32) = COPY $a1
  %2:_(p0) = G_FRAME_INDEX %stack.0.a.addr
  G_STORE %0:_(s32), %2:_(p0) :: (store (s32) into %ir.a.addr)
  %3:_(p0) = G_FRAME_INDEX %stack.1.b.addr
  G_STORE %1:_(s32), %3:_(p0) :: (store (s32) into %ir.b.addr)
  %4:_(s32) = G_LOAD %2:_(p0), debug-location !17 :: (dereferenceable load (s32) from %ir.a.addr); ./arith.c:2:13
  %5:_(s32) = G_LOAD %3:_(p0), debug-location !18 :: (dereferenceable load (s32) from %ir.b.addr); ./arith.c:2:17
  %6:_(s32) = nsw G_ADD %4:_, %5:_, debug-location !19; ./arith.c:2:15
  %7:_(s32) = G_LOAD %2:_(p0), debug-location !20 :: (dereferenceable load (s32) from %ir.a.addr); ./arith.c:2:22
  %8:_(s32) = nsw G_MUL %6:_, %7:_, debug-location !21; ./arith.c:2:20
  $v0 = COPY %8:_(s32), debug-location !22; ./arith.c:2:5
  RET implicit $lr, implicit $v0, debug-location !22; ./arith.c:2:5

# End machine code for function testArith.

```

图 5-2 示例程序一的合法化阶段 Dump 图

4. 寄存器组选择阶段

在寄存器组选择阶段，编译器需要为每个虚拟寄存器分配合适的寄存器组。在 DSP 后端实现中，所有参与整数算术运算的虚拟寄存器都会被统一映射至 GPRB 中。如图5-3所示，从寄存器组选择后的中间结果可以观察到，所有算术指令的操作数及结果均被分配到 GPRB 寄存器组，并且寄存器类型与对应操作数的位宽保持一致，符合 DSP 架构对整数运算的寄存器使用约束。

```

# *** IR Dump After RegBankSelect (regbankselect) ***:
# Machine code for function testArith: IsSSA, TracksLiveness, Legalized, RegBankSelected
Frame Objects:
  fi#0: size=4, align=4, at location [SP]
  fi#1: size=4, align=4, at location [SP]
Function Live Ins: $a0, $a1

bb.1.entry:
  liveins: $a0, $a1
  %0:gprb(s32) = COPY $a0
  %1:gprb(s32) = COPY $a1
  %2:gprb(p0) = G_FRAME_INDEX %stack.0.a.addr
  G_STORE %0:gprb(s32), %2:gprb(p0) :: (store (s32) into %ir.a.addr)
  %3:gprb(p0) = G_FRAME_INDEX %stack.1.b.addr
  G_STORE %1:gprb(s32), %3:gprb(p0) :: (store (s32) into %ir.b.addr)
  %4:gprb(s32) = G_LOAD %2:gprb(p0), debug-location !17 :: (dereferenceable load (s32) from %ir.a.addr); ./arith.c:2:13
  %5:gprb(s32) = G_LOAD %3:gprb(p0), debug-location !18 :: (dereferenceable load (s32) from %ir.b.addr); ./arith.c:2:17
  %6:gprb(s32) = nsw G_ADD %4:gprb, %5:gprb, debug-location !19; ./arith.c:2:15
  %7:gprb(s32) = G_LOAD %2:gprb(p0), debug-location !20 :: (dereferenceable load (s32) from %ir.a.addr); ./arith.c:2:22
  %8:gprb(s32) = nsw G_MUL %6:gprb, %7:gprb, debug-location !21; ./arith.c:2:20
  $v0 = COPY %8:gprb(s32), debug-location !22; ./arith.c:2:5
  RET implicit $lr, implicit $v0, debug-location !22; ./arith.c:2:5

# End machine code for function testArith.

```

图 5-3 示例程序一的寄存器组选择阶段 Dump 图

该结果表明所有参与算术运算的虚拟寄存器均被正确分配至通用寄存器组，同时未引入额外的跨寄存器组数据复制或冗余中间指令。

5. 机器指令选择阶段

在机器指令选择阶段，GMIR 被转换为 DSP 架构的目标机器指令。如图5-4所示，从中间结果可以观察到，除了 COPY 指令外的所有指令都被正确的映射到 DSP 后端机器指令，COPY 指令需要在后续阶段中单独处理。

```
# *** IR Dump After InstructionSelect (instruction-select) ***:
# Machine code for function testArith: IsSSA, TracksLiveness, Legalized, RegBankSelected, Selected
Frame Objects:
  fi#0: size=4, align=4, at location [SP]
  fi#1: size=4, align=4, at location [SP]
Function Live Ins: $a0, $a1

bb.1.entry:
  liveins: $a0, $a1
  %0:gpr32 = COPY $a0
  %1:gpr32 = COPY $a1
  STORE32 %0:gpr32, %stack.0.a.addr, 0 :: (store (s32) into %ir.a.addr)
  STORE32 %1:gpr32, %stack.1.b.addr, 0 :: (store (s32) into %ir.b.addr)
  %4:gpr32 = LOAD32 %stack.0.a.addr, 0, debug-location !17 :: (dereferenceable load (s32) from %ir.a.addr); ./arith.c:2:13
  %5:gpr32 = LOAD32 %stack.1.b.addr, 0, debug-location !18 :: (dereferenceable load (s32) from %ir.b.addr); ./arith.c:2:17
  %6:gpr32 = nsw ADD %4:gpr32, %5:gpr32, implicit-def dead $carry, debug-location !19; ./arith.c:2:15
  %7:gpr32 = LOAD32 %stack.0.a.addr, 0, debug-location !20 :: (dereferenceable load (s32) from %ir.a.addr); ./arith.c:2:22
  %8:gpr32 = nsw MUL32 %6:gpr32, %7:gpr32, debug-location !21; ./arith.c:2:20
  $v0 = COPY %8:gpr32, debug-location !22; ./arith.c:2:5
  RET implicit $lr, implicit $v0, debug-location !22; ./arith.c:2:5

# End machine code for function testArith.
```

图 5-4 示例程序一的机器指令选择阶段 Dump 图

6. 最终代码生成阶段

在完成寄存器分配以及指令调度等一系列的后端处理后，编译器最终生成了符合 DSP 指令集规范的可执行机器码。如图5-5所示，生成的目标代码在指令顺序、操作数选择以及控制流结构等方面均与原始程序语义保持一致。这表明在该示例程序中，基于 GlobalISel 框架的代码生成流程能够在 DSP 后端上正确运行，并完成从 LLVM IR 到目标机器码的端到端映射。

5.1.2 控制流指令正确性验证

除基本算术与内存访问指令这些基本指令外，控制流指令的正确生成和映射同样是验证编译器后端正确性的重要体现。本节针对对控制流相关指令的处理过程进行验证，重点覆盖比较与条件分支、循环结构以及函数调用三类典型控制流场景。

1. 比较 + 分支 + 跳转指令

比较与条件分支是程序控制流构建的基础。在 LLVM IR 中，比较与条件分支通过 icmp 与 br 组合表达。为验证 DSP 后端对该类控制流指令的处理正确性，

```

# *** IR Dump After DSP NOP Rewriter (enableNOPRewriter) ***
# Machine code for function testArith: NoPHIs, TracksLiveness, NoVRegs, Legalized, RegBankSelected, Selected, TiedOpsWritten
Frame Objects:
  fi#0: size=4, align=4, at location [SP-4]
  fi#1: size=4, align=4, at location [SP-8]
Function Live Ins: $a0, $a1

bb.1.entry:
  liveins: $a0, $a1
  $sp = ADDI $sp, -8, implicit-def $carry
  CFI_INSTRUCTION def_cfa_offset 8
  STORE32 killed $a0, $sp, 4 :: (store (s32) into %ir.a.addr)
  STORE32 killed $a1, $sp, 0 :: (store (s32) into %ir.b.addr)
  $v0 = LOAD32 $sp, 4, debug-location !17 :: (dereferenceable load (s32) from %ir.a.addr); ./arith.c:2:13
  NOP debug-location !18; ./arith.c:2:17
  $v1 = LOAD32 $sp, 0, debug-location !18 :: (dereferenceable load (s32) from %ir.b.addr); ./arith.c:2:17
  NOP debug-location !19; ./arith.c:2:15
  $v0 = nsw ADD killed $v0, killed $v1, implicit-def dead $carry, debug-location !19; ./arith.c:2:15
  $v1 = LOAD32 $sp, 4, debug-location !20 :: (dereferenceable load (s32) from %ir.a.addr); ./arith.c:2:22
  $v0 = nsw MUL32 killed $v0, killed $v1, debug-location !21; ./arith.c:2:20
  $sp = ADDI $sp, 8, implicit-def $carry
  RET implicit $lr, implicit killed $v0, debug-location !22; ./arith.c:2:5
  JNOP debug-location !22; ./arith.c:2:5
  JNOP debug-location !22; ./arith.c:2:5

# End machine code for function testArith.

```

图 5-5 示例程序一的最终阶段 Dump 图

本节构建了一个同时包含比较、条件分支及无条件跳转的示例程序二，其 LLVM IR 如代码块5.2所示。

```

1 define dso_local i32 @testBranch() {
2   entry:
3     %retval = alloca i32
4     %a = alloca i32
5     %b = alloca i32
6     store i32 10, ptr %a
7     store i32 20, ptr %b
8     %0 = load i32, ptr %a
9     %1 = load i32, ptr %b
10    %cmp = icmp slt i32 %0, %1
11    br i1 %cmp, label %if.then, label %if.else
12
13    if.then:
14      store i32 0, ptr %retval
15      br label %return
16
17    if.else:
18      store i32 1, ptr %retval
19      br label %return
20
21    return: %2 = load i32, ptr %retval
22    ret i32 %2
23 }

```

Listing 5.2: 示例程序二的 LLVM IR 表示

针对该示例程序，在 GlobalISel 各关键阶段中的处理过程如下：

- 在 GMIR 生成阶段，LLVM IR 中的比较指令被正确转换为 G_ICMP，其比较谓词与操作数保持一致；条件跳转指令被转换为 G_BRCOND，并以比较结果作为分支条件。
- 在合法化阶段，由于 DSP 架构支持整数比较操作，无需进一步拆分，相关指令保持原有结构。
- 在寄存器组选择阶段，编译器为比较操作及其结果所涉及的虚拟寄存器分配了合适的寄存器组，确保比较结果能够以符合 DSP 架构约定的方式参与后续控制流指令。
- 在机器指令选择阶段，G_ICMP 与后续的 G_BRCOND 会被联合处理，映射为 DSP 架构下的比较指令与条件跳转指令，比较结果通过条件码寄存器传递，从而完成分支控制。

如图5-6所示，在函数入口基本块中，比较指令首先生成条件码寄存器状态，随后通过条件跳转指令根据比较结果在两个后继基本块之间进行分支选择，验证了比较与条件分支指令的正确映射。

```
# *** IR Dump After DSP NOP Rewriter (enableNOPRewriter) ***
# Machine code for function testBranch: NoPHIs, TracksLiveness, NoVRegs, Legalized, RegBankSelected, Selected, TiedOpsRewritten
Frame Objects:
fi#0: size=4, align=4, at location [SP-4]
fi#1: size=4, align=4, at location [SP-8]
fi#2: size=4, align=4, at location [SP-12]

bb.1.entry:
successors: %bb.2, %bb.3

$sp = ADDI $sp, -16, implicit-def $carry
CFI_INSTRUCTION def_cfa_offset 16
$v0 = MOVI GLZ 10, debug-location !14; ./branch.c:2:9
STORE32 killed $v0, $sp, 8, debug-location !14 :: (store (s32) into %ir.a); ./branch.c:2:9
$v0 = MOVI GLZ 20, debug-location !16; ./branch.c:2:17
STORE32 killed $v0, $sp, 4, debug-location !16 :: (store (s32) into %ir.b); ./branch.c:2:17
NOP debug-location !17; ./branch.c:3:9
$v0 = LOAD32 $sp, 8, debug-location !17 :: (dereferenceable load (s32) from %ir.a); ./branch.c:3:9
NOP debug-location !19; ./branch.c:3:13
$v1 = LOAD32 $sp, 4, debug-location !19 :: (dereferenceable load (s32) from %ir.b); ./branch.c:3:13
GE killed $v0, killed $v1, implicit-def $con, implicit-def $con, debug-location !20; ./branch.c:3:11
JC %bb.3, implicit $con, implicit $con, debug-location !20; ./branch.c:3:11
JNOP debug-location !20; ./branch.c:3:11
JNOP debug-location !20; ./branch.c:3:11
JMP %bb.2, debug-location !20; ./branch.c:3:11
JNOP debug-location !20; ./branch.c:3:11
JNOP debug-location !20; ./branch.c:3:11
```

图 5-6 示例程序二入口基本块的最终阶段 Dump 图

如图5-7所示，条件判断成立时进入 if.then 基本块，不成立则进入 if.else 基本块，两个基本块各自执行相应的逻辑操作，且在块末尾均通过无条件跳转指令，汇合到同一个返回基本块中。从最终生成的机器码结果来看，各分支基本块

间的跳转关系，以及这些分支块与返回块的连接方式，均与 LLVM IR 所定义的控制流结构一致，未产生任何额外的控制流变动。

```

bb.2.if.then:
; predecessors: %bb.1
successors: %bb.4(0x80000000); %bb.4(100.00%)

$v0 = MOVI GLZ 0, debug-location !21; ./branch.c:4:9
STORE32 killed $v0, $sp, 12, debug-location !21 :: (store (s32) into %ir.retval); ./branch.c:4:9
JMP %bb.4, debug-location !21; ./branch.c:4:9
JNOP debug-location !21; ./branch.c:4:9
JNOP debug-location !21; ./branch.c:4:9

bb.3.if.else:
; predecessors: %bb.1
successors: %bb.4(0x80000000); %bb.4(100.00%)

$v0 = MOVI GLZ 1, debug-location !23; ./branch.c:6:9
STORE32 killed $v0, $sp, 12, debug-location !23 :: (store (s32) into %ir.retval); ./branch.c:6:9
JMP %bb.4, debug-location !23; ./branch.c:6:9
JNOP debug-location !23; ./branch.c:6:9
JNOP debug-location !23; ./branch.c:6:9

bb.4.return:
; predecessors: %bb.3, %bb.2

$v0 = LOAD32 $sp, 12, debug-location !25 :: (dereferenceable load (s32) from %ir.retval); ./branch.c:8:1
$sp = ADDI $sp, 16, implicit-def $carry
RET implicit $lr, implicit killed $v0, debug-location !25; ./branch.c:8:1
JNOP debug-location !25; ./branch.c:8:1
JNOP debug-location !25; ./branch.c:8:1

```

图 5-7 示例程序二条件分支基本块的最终阶段 Dump 图

实验结果表明，从 LLVM IR 到最终机器码的转换过程中，比较条件与分支目标均保持一致，程序控制流未发生偏移或错误跳转。实验验证了比较与分支指令选择的正确性。

2. 循环指令

循环结构由条件判断、分支跳转以及回边构成。在 LLVM IR 中，循环通过多个基本块以及 br 指令共同表示。在 GMIR 中，这一多基本块结构及其控制流关系被完整保留下来，为后端代码生成提供了清晰的控制流语义。

为验证 DSP 后端对循环指令的处理正确性，本文构建了一个包含循环指令的示例程序三，其 LLVM IR 如代码块5.3所示。

```

1 define dso_local i32 @testLoop() {
2     entry:
3     %i = alloca i32
4     store i32 0, ptr %i
5     br label %for.cond
6
7     for.cond:
8     %0 = load i32, ptr %i

```

```

9  %cmp = icmp slt i32 %0, 10
10 br i1 %cmp, label %for.body, label %for.end
11
12 for.body:
13 br label %for.inc, !dbg !21
14
15 for.inc:
16 %1 = load i32, ptr %i
17 %inc = add nsw i32 %1, 1
18 store i32 %inc, ptr %i
19 br label %for.cond
20
21 for.end:
22 ret i32 0
23 }

```

Listing 5.3: 示例程序三的 LLVM IR 表示

针对该示例程序，在 GlobalISel 各关键阶段中的处理过程如下：

- 在 GMIR 生成阶段，各基本块及其控制流关系被准确映射为 MBB，循环条件与跳转关系通过 G_BR 与 G_BRCOND 指令表达。
- 在合法化阶段，合法化前后指令结构保持一致，表明 DSP 后端的合法化规则能够覆盖该类循环场景的基本操作需求。
- 在寄存器组选择阶段，循环体内的算术与内存操作被正确映射至 DSP 通用寄存器组，循环条件判断所依赖的比较结果亦能够被正确传播。
- 在机器指令选择阶段，循环控制流相关的 G_ICMP 与 G_BRCOND 被联合分析并映射为 DSP 架构下的比较指令与条件跳转指令：比较结果通过条件码寄存器生成，并由条件跳转指令读取实现分支决策。

如图5-8所示，在循环条件基本块中，编译器首先将循环变量与上界进行比较，再借助条件码寄存器生成对应的分支条件，随后根据比较结果在循环体与循环结束块之间跳转，由此验证了循环条件判断与分支决策逻辑的正确性。

如图5-9所示，循环体与自增基本块在执行完成后通过无条件跳转形成回边，并重新进入条件判断块；当循环条件不满足时，控制流正确跳转至循环结束块并完成函数返回，表明循环控制流在后端生成过程中保持一致。

最终生成的机器码显示，循环入口、循环体以及循环回边对应的跳转指令均与原始 LLVM IR 的控制流结构保持一致，且不存在多余或缺失的跳转指令。这说明 GlobalISel 在 DSP 后端中能够正确处理多基本块循环结构，确保循环语义的正确无误。

```

# *** IR Dump After DSP NOP Rewriter (enableNOPRewriter) ***:
# Machine code for function testLoop: NoPHIs, TracksLiveness, NoVRegs, Legalized, RegBankSelected, Selected, TiedOpsWritten
Frame Objects:
| fi#0: size=4, align=4, at location [SP-4]

bb.1.entry:
successors: %bb.2(0x80000000); %bb.2(100.00%)

$sp = ADDI $sp, -8, implicit-def $carry
CFI_INSTRUCTION def_cfa_offset 8
$v0 = MOVIGLZ 0, debug-location !15; ./loop.c:2:14
STORE32 killed $v0, $sp, 4, debug-location !15 :: (store (s32) into %ir.i); ./loop.c:2:14
JMP %bb.2, debug-location !16; ./loop.c:2:10
JNOP debug-location !16; ./loop.c:2:10
JNOP debug-location !16; ./loop.c:2:10

bb.2.for.cond:
; predecessors: %bb.1, %bb.4
successors: %bb.3, %bb.5

BUNDLE implicit-def $v1, implicit-def $v0, implicit $sp, debug-location !19; ./loop.c:2:23 {
    $v1 = MOVIGLZ 10, debug-location !19; ./loop.c:2:23
    $v0 = LOAD32 $sp, 4, debug-location !17 :: (dereferenceable load (s32) from %ir.i); ./loop.c:2:21
}
GE killed $v0, killed $v1, implicit-def $con, implicit-def $con, debug-location !19; ./loop.c:2:23
JC %bb.5, implicit $con, implicit $con, debug-location !19; ./loop.c:2:23
JNOP debug-location !19; ./loop.c:2:23
JNOP debug-location !19; ./loop.c:2:23
JMP %bb.3, debug-location !20; ./loop.c:2:5
JNOP debug-location !20; ./loop.c:2:5
JNOP debug-location !20; ./loop.c:2:5

```

图 5-8 示例程序三循环条件基本块的最终阶段 Dump 图

```

bb.3.for.body:
; predecessors: %bb.2
successors: %bb.4(0x80000000); %bb.4(100.00%)

JMP %bb.4, debug-location !21; ./loop.c:3:5
JNOP debug-location !21; ./loop.c:3:5
JNOP debug-location !21; ./loop.c:3:5

bb.4.for.inc:
; predecessors: %bb.3
successors: %bb.2(0x80000000); %bb.2(100.00%)

BUNDLE implicit-def $v1, implicit-def $v0, implicit $sp, debug-location !23; ./loop.c:2:30 {
    $v1 = MOVIGLZ 1, debug-location !23; ./loop.c:2:30
    $v0 = LOAD32 $sp, 4, debug-location !23 :: (dereferenceable load (s32) from %ir.i); ./loop.c:2:30
}
$v0 = nsw ADD killed $v0, killed $v1, implicit-def dead $carry, debug-location !23; ./loop.c:2:30
STORE32 killed $v0, $sp, 4, debug-location !23 :: (store (s32) into %ir.i); ./loop.c:2:30
JMP %bb.2, debug-location !24; ./loop.c:2:5
JNOP debug-location !24; ./loop.c:2:5
JNOP debug-location !24; ./loop.c:2:5

bb.5.for.end:
; predecessors: %bb.2

$v0 = MOVIGLZ 0, debug-location !28; ./loop.c:4:5
$sp = ADDI $sp, 8, implicit-def $carry
RET implicit $lr, implicit killed $v0, debug-location !28; ./loop.c:4:5
JNOP debug-location !28; ./loop.c:4:5
JNOP debug-location !28; ./loop.c:4:5

```

图 5-9 示例程序三循环体基本块的最终阶段 Dump 图

3. 函数调用指令

函数调用是另一类重要的控制流操作，其正确性取决于调用约定、参数传递以及返回值处理等多个环节。在 LLVM IR 中，函数调用通过 call 指令表示。在 GlobalISel 框架下，函数调用的处理过程则涉及 CallLowering 以及指令选择等多个阶段。

为验证 DSP 后端在 GlobalISel 框架下对函数调用指令的处理正确性，构造了一个简单的示例程序四，其中 main 函数调用前文定义的 testBranch 函数，其 LLVM IR 如代码块5.4所示。

```

1 define dso_local i32 @main() {
2     entry:
3     %retval = alloca i32
4     store i32 0, ptr %retval
5     %call = call i32 @testBranch()
6     ret i32 %call
7 }
```

Listing 5.4: 示例程序四的 LLVM IR 表示

针对该示例程序，函数调用在各阶段的处理过程如下：

- 在 GMIR 生成阶段，call 指令被转换为与调用约定相关的一系列包括参数准备、调用指令以及返回值接收等操作的 GMIR 指令。
- 在合法化阶段，合法化前后指令结构保持一致，表明 DSP 后端的合法化规则能够覆盖该类函数调用场景的基本操作需求。
- 在寄存器组选择阶段，函数参数与返回值在符合 DSP ABI 的规则下被分配至 DSP 架构规定的寄存器或栈位置。
- 在机器指令选择阶段，与调用相关的 GMIR 指令会被映射为 DSP 架构支持的跳转与返回指令。

如图5-10所示，实验结果表明 DSP 后端在 GlobalISel 框架下能够正确完成函数调用相关控制流的指令选择，从而保证了跨函数控制流的正确传递。

5.2 全局指令选择性能分析

本章主要针对编译时间、代码大小以及执行周期三个方面来对基于 DAG 的指令选择、未优化的全局指令选择以及优化的全局指令选择来进行对比。

```

# *** IR Dump After DSP NOP Rewriter (enableNOPRewriter) ***
# Machine code for function main: NoPHIs, TracksLiveness, NoVRegs, Legalized, RegBankSelected, Selected, TiedOpsRewritten
Frame Objects:
| fi#0: size=4, align=4, at location [SP-8]
| fi#1: size=4, align=4, at location [SP-4]

bb.1.entry:
$sp = ADDI $sp, -16, implicit-def $carry
CFI_INSTRUCTION def_cfa_offset 16
STORE32 killed $lr, $sp, 12 :: (store (s32) into %stack.1)
CFI_INSTRUCTION offset $lr, -4
$v0 = MOVIGLZ 0
STORE32 killed $v0, $sp, 8 :: (store (s32) into %ir.retval)
CALL @testBranch, <regmask $fp $lr $d8 $d9 $d10 $d11 $s0 $s1 $s2 $s3 $s4 $s5 $s6 $s7>, implicit-def $lr, implicit $lr
, implicit-def $sp, implicit-def $v0, debug-location !27; ./branch.c:12:12
NOP debug-location !27; ./branch.c:12:12
NOP debug-location !27; ./branch.c:12:12
$lr = LOAD32 $sp, 12, debug-location !28 :: (load (s32) from %stack.1); ./branch.c:12:5
$sp = ADDI $sp, 16, implicit-def $carry
RET implicit $lr, implicit killed $v0, debug-location !28; ./branch.c:12:5
JNOP debug-location !28; ./branch.c:12:5
JNOP debug-location !28; ./branch.c:12:5

```

图 5-10 示例程序四的最终阶段 Dump 图

5.2.1 编译时间

编译时间是衡量编译器的可用性与开发效率的重要指标之一，尤其是在编译器需要持续迭代优化、频繁进行回归验证的实际开发场景中，后端编译开销的变化将直接影响开发效率。因此，本节围绕编译时间这一维度，对不同指令选择方案在 DSP 后端中的表现进行全面的对比分析。

1. 度量方法与相关指标说明

本节基于 Clang 自带的编译时间追踪机制来对编译时间进行量化分析，该机制能够对编译过程中各阶段的执行时间进行精细化统计，从而为分析编译器性能瓶颈与优化效果提供可靠的数据支撑。图5-11中展示了部分关键时间指标及其在不同编译阶段的分布情况。

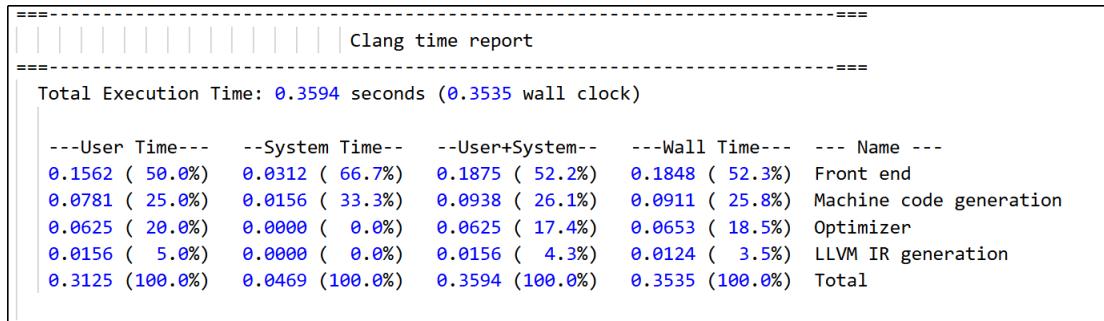


图 5-11 不同编译阶段时间指标图

其中，各时间指标的含义如下：

- User Time（用户态时间）：表示 CPU 在用户态执行编译器自身代码所消耗

的时间，仅反映编译器逻辑计算的开销，不包含操作系统内核相关操作。

- System Time（系统态时间）：表示 CPU 在内核态执行系统调用（如文件读写、内存管理、进程调度等）所消耗的时间，属于编译过程中的系统辅助开销。
- User+System Time（用户态加系统态时间）：反映 CPU 在该阶段用于计算的总耗时，是用户态和系统态时间的累加，但不包含 I/O 等待和 CPU 空闲时间。
- Wall Time（壁钟时间）：指从编译阶段开始到结束所消耗的实际物理时间，其中涵盖了 CPU 计算、I/O 等待以及进程调度等因素。在实际工程场景中，这一指标最能直观反映用户实际感知到的编译等待时长，因此也是本文关注的核心量化指标。

LLVM 编译流程可粗略划分为以下几个阶段：

- Front end（前端）：负责源代码解析、语法分析、语义分析等核心操作，核心作用是将高级语言代码转换为中间表示形式。
- LLVM IR generation（LLVM IR 生成）：负责将前端分析的结果转换为 LLVM IR，为后续优化和代码生成奠定基础。
- Optimizer（优化器）：负责对 LLVM IR 进行多级优化，包括控制流优化、数据流优化及目标无关优化等。
- Machine code generation（机器码生成）：作为编译器后端的核心环节，该阶段负责指令选择、寄存器分配、指令调度以及最终目标机器码的生成。

由于指令选择阶段在 LLVM 后端中与其他流程的耦合度较高，难以单独抽离出来进行精确量化，因此本节在编译时间的分析中，将关注点放在后端整体执行时间上。具体而言，统计 Machine Code Generation 阶段的 Wall Time，以此作为后端编译开销的统一度量标准。这种统计方式在保证分析结果可比性和稳定性的同时，能够有效反映后端相关优化对实际编译时间的影响，为后续性能评估提供依据。

2. 实验设置与统计方法

实验基于 Embench 测试集进行，Ebench 是一个面向嵌入式系统与微控制器平台的开源基准测试套件，主要用于评估编译器、指令集架构以及代码生成

质量在资源受限环境下的表现。本文采用多次重复实验的统计方法，用于降低单次测量中噪声与偶然因素对实验结果的影响。具体而言，在相同实验环境与配置条件下，对每个测试样例分别执行 10 次编译。在结果统计阶段，剔除两个最大值和两个最小值，对剩余数据取平均值作为该样例在对应优化等级下的编译时间。最后对 Embench 测试集中 22 个基准程序的结果进一步取平均，得到不同优化等级下的平均编译时间，用于不同指令选择方案之间的对比分析。

3. 实验结果与分析

图 5-12 给出了在 Embench 测试集上两种指令选择方案在不同优化等级下的平均编译时间对比结果。

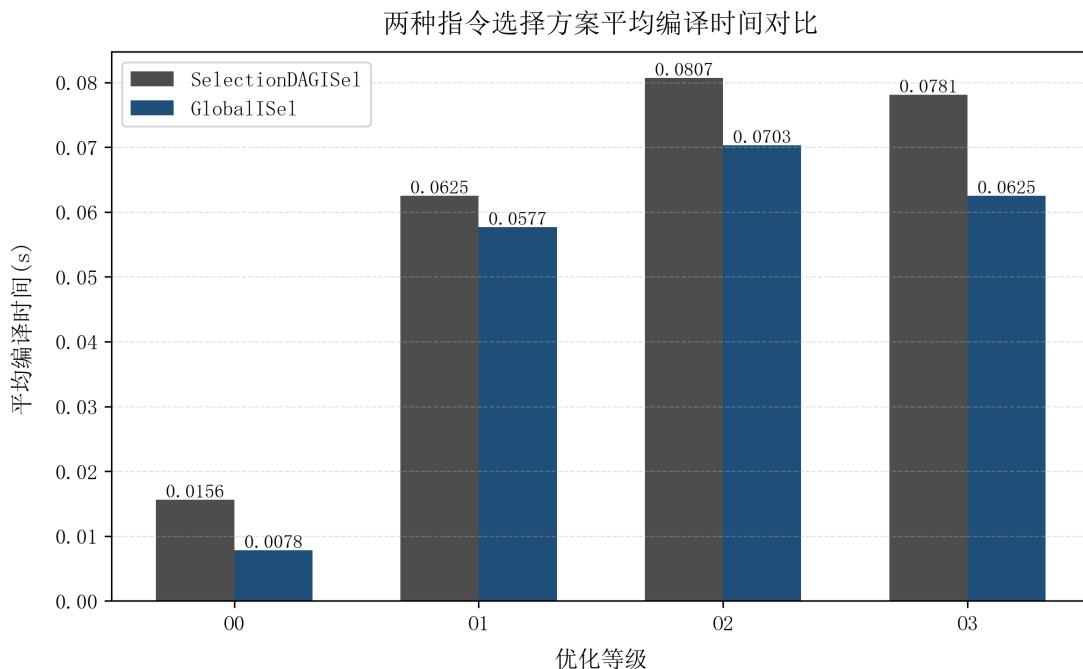


图 5-12 两种指令选择方案平均编译时间对比图

图 5-12 中结果显示，四个优化等级下 GlobalISel 的平均编译时间均更低。该趋势在 O0 与 O1 优化等级下更加明显，这表明在中端优化尚未充分展开的情况下，指令选择框架自身的设计差异就已对后端编译效率产生了影响。之所以出现这样的结果，主要是因为 GlobalISel 在指令选择阶段采用的基于规则匹配与直接构造 Machine IR 的流程，相较于传统基于 DAG 的指令选择方式，其构建与遍历成本更低，整体流程更线性。此外，GlobalISel 在 GMIR 层面引入的早期规范化与合并优化，也在一定程度上减少了后续 Pass 的处理负担，从而降低了后端编译的整体开销。

4. 实验结论

从实验结果上来看，DSP 后端中 GlobalISel 在编译时间上相比 SelectionDAG 具备稳定优势，且该优势在各优化等级下均保持一致。这说明该优势并非源于特定优化配置，而是来自指令选择框架本身在流程设计和中间表示处理上的效率优化。这一特性使 GlobalISel 更适合应用在需要频繁编译和快速迭代的工程场景，为后续编译器优化和性能验证提供了扎实的基础。

5.2.2 代码尺寸

代码尺寸是衡量编译器后端代码生成质量的重要指标之一，尤其在 DSP 这类嵌入式场景中，受限的程序存储空间使得指令密度成为影响系统规模与功耗水平的关键因素。因此，分析不同指令选择方案对最终生成代码体积的影响，对于评估 GlobalISel 框架下的优化效果具有重要意义。

1. 度量方法

本节基于 `llvm-size` 工具来完成代码尺寸的量化，并以目标文件中 `text` 段大小作为代码尺寸的度量指标。`text` 段大小能够直接反映指令选择与代码生成阶段所产生的指令数量与编码密度，同时有效规避调试信息、符号表等非执行内容对统计结果的干扰。

2. 实验设置与统计方法

在实验的设置上，本节选取 Embench 测试集，对每个基准程序分别在四个优化等级下进行编译，并统计对应生成目标文件的 `text` 段大小。对于同一测试样例与优化等级，取单次编译结果作为该样例的代码尺寸。随后，把测试集中所有基准程序的结果进行平均计算，得到各个优化等级下的平均代码段尺寸，用于不同指令选择方案之间的对比分析。

3. 实验结果分析

图5-13给出了在 Embench 测试集上，未引入优化的 GlobalISel 与引入优化后的 GlobalISel 在不同优化等级下的平均代码段尺寸对比结果。

从整体上看，在 4 个优化等级下的优化后 GlobalISel 方案最终生成的平均代码体积更小，而且在 O0 性能等级下体现更加明显，表明即使是不进行高层 LLVM IR 的优化，在后端指令选择和指令组合方面提高的优化手段亦可以减小指令冗余、提高指令密度。高优化等级下，代码尺寸下降更加收敛，因为高层

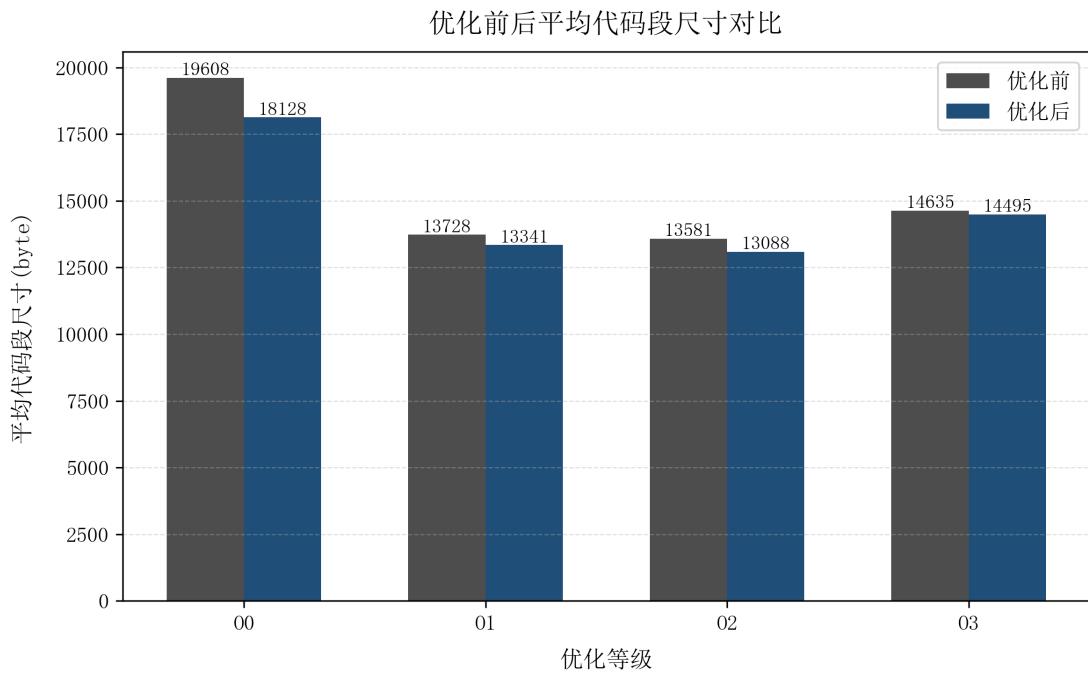


图 5-13 优化前后平均代码段尺寸对比图

LLVM IR 已经展开得很充分，有些冗余已经在前端或中端去掉，后端指令选择优化对代码体积的影响更多是精细化的指令密度改进，而非数量级的减少。

4. 实验结论

从实验结果来看，相较于未优化的 GlobalISel 实现，引入针对 DSP 架构特点的指令选择与指令组合优化后，编译器在各优化等级下均生成了更紧凑的目标代码。这表明本文提出的优化策略不仅在性能层面有效，还在代码尺寸这一关键指标上取得效果，为后续嵌入式场景的实际部署提供了有力支撑。

5.2.3 执行周期

执行周期是衡量编译器后端优化效果最直接、也是最具代表性的性能指标之一。对于 DSP 这种以吞吐率和实时性为核心设计目标的处理器架构而言，程序执行周期的变化能够直观反映编译器后端各阶段的决策对实际运行性能的影响。因此，本节围绕执行周期这一维度，对不同指令选择方案在 DSP 平台上的运行表现进行对比分析。

1. 度量方法

本文的执行周期测量基于 DSP 工具链配套的仿真器环境完成。在仿真环境中，仿真器能够在程序执行过程中精确统计指令发射与执行情况，并直接给出

程序完成所消耗的总周期数。

为保证不同指令选择方案在执行周期统计上的可比性，本节在测量时统一采用相同的测试程序、输入数据以及运行环境配置。除此之外，本节关注程序主体计算阶段的执行周期，避免初始化、调试输出等非核心逻辑对统计结果产生干扰。实验最终得到的执行周期能够真实地反映编译器后端生成的代码在 DSP 架构上的运行效率。

2. 实验设置与统计方法

执行周期实验同样基于 Embench 测试集进行。对于每个基准程序，在相同硬件平台和配置条件下分别运行多次，以降低偶然扰动对测量结果的影响。具体统计方法如下：对每个测试样例在对应优化等级下执行 10 次，剔除两个最大值和两个最小值，对剩余结果取平均值作为该样例的执行周期；随后，对测试集中全部基准程序的执行周期结果进一步取平均，得到不同优化等级下的平均执行周期，用于不同指令选择方案之间的对比分析。

该统计方式能够有效平衡测量精度与实验成本，在保证数据稳定性的同时，避免个别异常运行结果对整体趋势判断产生干扰。

3. 实验结果与分析

图5-14给出了在 Embench 测试集上，不同指令选择方案在各优化等级下的平均执行周期对比结果。

从实验结果可以观察到，相较于未优化的 GlobalISel 实现，引入针对 DSP 架构的指令选择与指令组合优化后，程序在所有优化等级下的平均执行周期均呈现出下降趋势。这表明，在高层 LLVM IR 优化与后端优化协同作用的场景中，GlobalISel 所引入的目标相关指令匹配、冗余指令消除以及寄存器组绑定优化，能够进一步挖掘 DSP 架构的指令级并行潜力，从而在整体执行效率上获得持续收益。

4. 实验结论

从实验结果来看，DSP 后端在引入针对其目标架构特性的 GlobalISel 优化后，编译器生成代码在执行周期维度上取得了稳定且可观的性能提升。并且该提升在不同优化等级下均具有一致性，说明其并非依赖特定编译配置，而是源于指令选择框架与目标相关优化策略在整体设计上的协同改进。这一结果进一步验证了前文提出的优化方法在实际运行性能上的有效性。

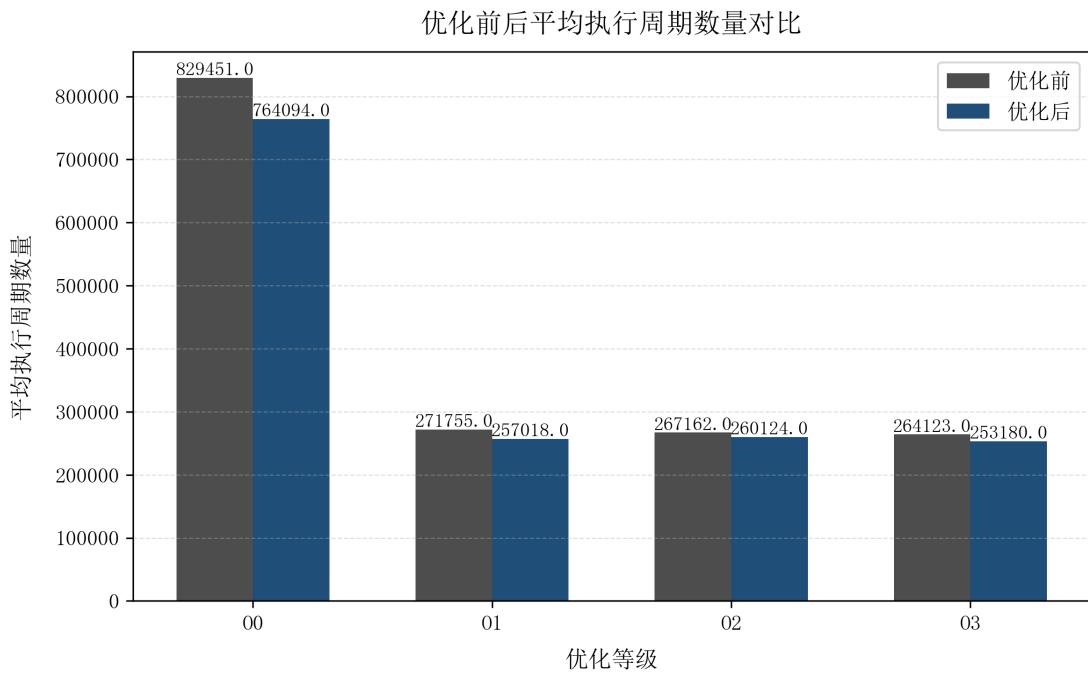


图 5-14 不同指令选择方案平均执行周期对比图

5.3 测试评估平台功能展示

测试评估平台围绕编译器工具链的持续迭代需求来进行设计。平台集成了性能数据采集、历史回归分析、版本对比以及汇编级诊断等功能。平台通过系统化地记录与分析不同 Commit 在统一测试集上的性能表现，已在实际开发过程中协助作者发现并定位多个编译器缺陷，为编译器优化与回归验证提供了重要支撑。下文结合平台主要功能模块，详细展示其核心功能。

1. 全局性能趋势分析模块

如图5-15所示，该模块用于展示历史 Commit 的整体性能概况。模块基于完整测试集，对不同 Commit 下的平均代码体积和平均执行周期进行统计与可视化展示，便于快速识别性能回退或异常波动，为后续的分析提供依据。除此之外，模块还支持按测试集、测试平台以及芯片版本区分数据，同时展示测试集在不同优化等级及版本下的平均代码尺寸与平均执行周期。

2. 单测试用例性能演化分析模块

如图5-16所示，该模块用于展示单个测试用例在不同 Commit 下的性能变化情况，并支持按时间序列展示其执行周期、代码尺寸以及指令打包效率等关键指标。通过该模块，开发者可以精确定位某一性能变化首次引入的 Commit，从而显著提升性能回归分析的效率与准确性。



图 5-15 全局性能趋势分析界面展示图

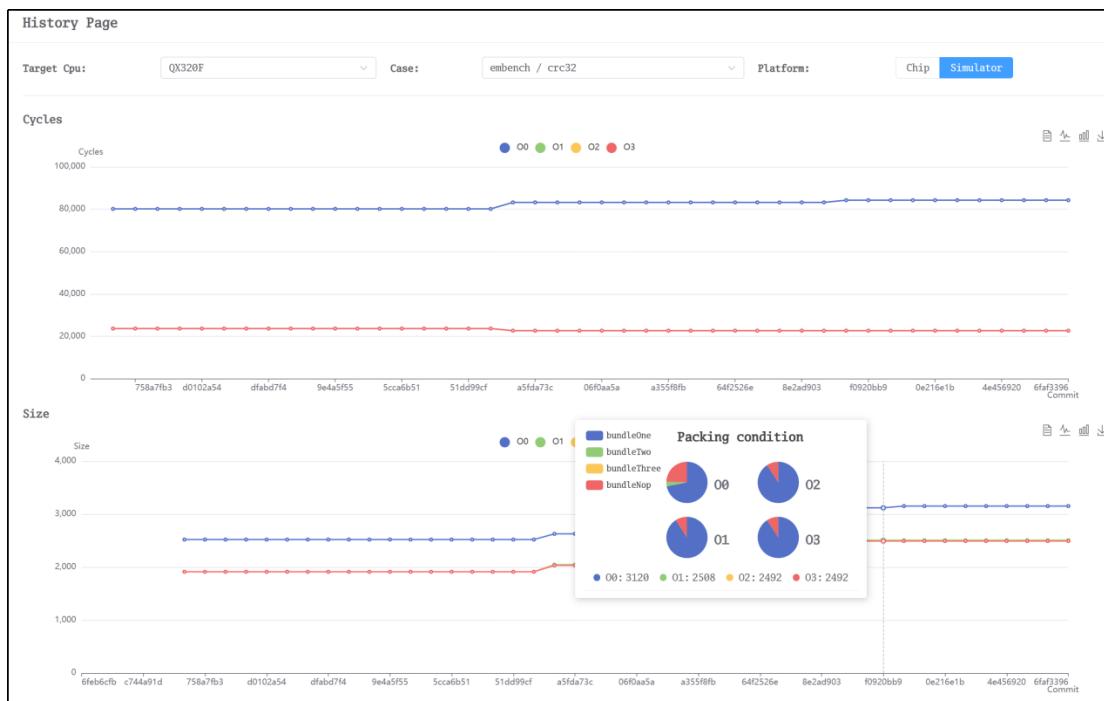


图 5-16 单测试用例性能演化分析界面展示图

3. 跨版本性能差异对比分析模块

如图5-17所示，该模块用于对任意两个 Commit 之间的性能指标进行对比分析。模块不仅提供测试用例级别的执行周期与代码体积对比结果，还提供样例汇编代码对比的功能，以此帮助开发者从底层代码生成角度分析性能差异的根本原因，适用于优化效果验证与回归问题定位的情景。

The screenshot shows a web-based performance comparison tool. At the top, there are dropdown menus for 'Target Cpu' (QX320F), 'Suite' (scalar), 'Opt level' (O0), and 'Platform' (Chip Simulator). Below these are two sets of commit details:

Source Commit Detail			Target Commit Detail		
Author	Hash	Date	Author	Hash	Date
xingyu zhou <3355789077@qq.com>	6ae361cb	2025-07-21T14:49:12.000+08:00	Bai Lu <1764339350@qq.com>	a355f8fb	2025-07-28T15:43:12.000+08:00
Message			Message		
fix: 为QUADF指令添加延时槽 See merge request compiler/dsp-llvm-project!67			fix: 修复跳转到bundle出错的bug See merge request compiler/dsp-llvm-project!61		

Below this is a table comparing execution cycles and code size for 10 test cases:

No.	CaseName	Cycle Num		Code Size		Assembly
		Source	Target	Source	Target	
1	addc.c	174	174	3216	3232	<button>View</button>
2	addition.c	2272	2272	3228	3244	<button>View</button>
3	and_or_if.c	340	340	3840	3856	<button>View</button>
4	argsum.c	632	633 ▲	3464	3476	<button>View</button>
5	argtest.c	None	None	None	None	<button>View</button>
6	argument_fir.c	50871	50872 ▲	4368	4392	<button>View</button>
7	bestoptim.c	925	926 ▲	3820	3844	<button>View</button>
8	binary_insert.c	1098	1102 ▲	4008	4036	<button>View</button>
9	binary_search.c	391	395 ▲	3728	3752	<button>View</button>
10	bool_ygh.c	2273	2273	3232	3248	<button>View</button>

图 5-17 跨版本性能差异对比分析界面展示图

4. 本地性能测试报告上传模块

如图5-18所示，该模块用于上传本地生成的性能测试结果，并与 Master 分支的结果进行对比分析。该模块为开发者在本地修改编译器或新增优化策略后的快速验证提供了便利。

5.4 本章小结

本章主要围绕基于 GlobalISel 框架的 DSP 后端的代码生成正确性和性能进行了详细的验证与分析。首先通过一系列代表性的示例程序，系统地验证了从 LLVM IR 到目标机器码的转换过程，包括指令合法化、寄存器组选择以及机器指令选择等关键阶段，确保了编译器在 DSP 平台上的功能正确性。

在验证了正确性之后，本章接着对编译时间、代码尺寸和执行周期等性能指标进行了全面对比分析，评估了不同指令选择方案的优化效果。实验结果表明，各关键环节均能保持原始程序语义与数据依赖关系，生成的目标代码符合 DSP

The screenshot shows a web-based form titled "Custom Page". It includes fields for "Author" (zhouxingyu), "Hash" (dadasdha232323bkghj), and a "Message" box containing "This is a test.". There are "Submit" and "Reset" buttons. To the right is a file upload area with a cloud icon and the text "Drop files here or click to upload", noting a limit of 10 files. Below this is a message stating "limit 10 files, files will upload atomically". At the bottom left are red "Pure all report" and "Pure" buttons.

图 5-18 本地性能测试报告上传界面展示图

指令集规范与控制流逻辑，充分验证了 GlobalISel 在 DSP 后端应用的功能正确性与稳定性。

此外，本章介绍了测试评估平台的核心功能，包括全局性能趋势分析、单测试用例性能演化追踪、跨版本性能对比和本地测试报告上传等。这些功能为实验的正确性验证和性能分析提供了高效的数据支持与可视化手段，并有助于开发者快速定位性能回退与代码生成缺陷。

第 6 章 总结与展望

6.1 论文总结

本文围绕 DSP 架构下基于 LLVM 的 GlobalISel 框架的工程化实现与优化问题，系统开展了架构分析、优化设计、实现验证以及性能评估等研究工作，形成了一套面向 DSP 后端的全局指令选择实现与测试评估方案。全文的主要研究内容与成果可以总结为以下几个方面。

1. 首先，在指令选择框架层面，本文对 GlobalISel 的整体架构与关键阶段进行了系统分析，重点围绕 GMIR 生成、指令合法化、寄存器组选择以及机器指令选择等核心流程，结合 DSP 架构特性，完成了全局指令选择在 DSP 后端中的功能性实现。通过对基本算术指令、控制流指令以及函数调用等典型场景的端到端验证，证明了 GlobalISel 框架在 DSP 后端中的可行性与正确性，为后续优化工作奠定了基础。
2. 其次，在优化策略设计方面，本文针对 GlobalISel 在不同阶段产生的冗余与性能问题，提出了分阶段的指令选择优化思路，并围绕内存操作指令、乘法指令以及其他通用指令模式，设计并实现了多项具有针对性的优化策略。其中，合法化前的通用合并优化侧重于消除表达式级冗余和规范化指令形态，而合法化后的目标相关优化则更加关注 DSP 专用指令匹配、代码密度提升以及寄存器使用效率。实验结果表明，这些优化能够在保证语义正确性的前提下，有效改善生成代码的质量。
3. 再次，在测试与评估体系方面，本文设计并实现了一套面向 DSP 编译器的测试评估平台。该平台以随机测试生成技术为基础，结合 DSP 架构特点对 YARPGen 进行定制化扩展，构建了覆盖指令选择与后端优化场景的随机测试输入体系。在此基础上，平台实现了从测试执行、数据采集到结果分析与可视化展示的完整流程，能够支持跨版本性能对比、历史趋势分析以及回归问题定位，为编译器优化提供了可靠的实验支撑。
4. 最后，通过在 Embench 等测试集上的系统实验，本文从编译时间、代码尺寸和执行周期等多个维度，对比分析了基于 DAG 的指令选择方案、未优化的 GlobalISel 以及优化后的 GlobalISel 实现。实验结果表明，在 DSP 架

构下，GlobalISel 在编译效率方面具有一定优势，结合本文提出的优化策略后，在代码尺寸和执行效率等方面亦表现出较为稳定的改进效果，验证了本文方法在工程实践中的有效性。

综上所述，本文围绕 DSP 后端的 GlobalISel 指令选择实现与优化问题，完成了从架构分析、优化设计到测试验证的系统性研究，为后续在 DSP 平台上进一步推进 GlobalISel 的应用和优化提供了实践基础。

6.2 论文展望

尽管本文在 DSP 架构下对 GlobalISel 指令选择及其优化进行了较为系统的研究，但仍存在进一步拓展和深化的空间，后续工作可从以下几个方向展开：

1. 在指令选择优化层面，本文当前的优化策略主要集中在局部模式合并和指令级重写，尚未充分结合更高层次的全局信息。未来可以进一步探索将数据流分析、控制流分析与 GlobalISel 的合并优化机制相结合，在函数级甚至跨基本块层面挖掘更具潜力的优化机会，以进一步提升代码质量和执行效率。
2. 在 DSP 架构特性利用方面，本文的优化设计主要围绕通用算术指令和部分专用指令展开。随着 DSP 指令集和微架构特性的不断演进，未来可进一步针对流水线结构、指令并行执行能力以及专用加速单元，设计更细粒度的目标相关优化策略，使 GlobalISel 在 DSP 平台上的优势得到更充分发挥。
3. 在测试评估体系方面，当前平台主要侧重于性能指标的离线统计与可视化分析。后续工作可以考虑引入更自动化的性能回归检测机制，例如基于阈值或趋势分析的自动告警策略，以提升平台在持续集成和日常开发中的实用性。同时，也可扩展测试输入类型，引入更多面向实际应用场景的负载程序，以增强测试结果的代表性。
4. 从整体工具链角度来看，GlobalISel 仍处于持续演进阶段。未来可结合 LLVM 社区在 GlobalISel 方向上的最新进展，对本文实现进行持续迭代，并探索将相关优化经验推广至其他嵌入式或专用处理器架构，为构建统一、高效的指令选择框架提供更多实践参考。

参考文献

- [1] PROAKIS J G. Digital signal processing: principles, algorithms, and applications, 4/e[M]. Pearson Education India, 2007.
- [2] WANG Y, LI C, LIU C, et al. Advancing dsp into hpc, ai, and beyond: challenges, mechanisms, and future directions[J]. CCF Transactions on High Performance Computing, 2021, 3(1): 114-125.
- [3] SONG W, XU D, CHEN L. Overview of dsp architecture development[J]. Microelectronics & Computer, 2023, 40(4): 1-7.
- [4] FLYNN M J. Some computer organizations and their effectiveness[J]. IEEE transactions on computers, 2009, 100(9): 948-960.
- [5] FISHER J A. Very long instruction work architectures and the eli-512[C]//25 years of the International symposia on Computer architecture (selected papers). 1998: 263-273.
- [6] RAU B R, FISHER J A. Instruction-level parallel processing: History, overview, and perspective[M]//Instruction-Level Parallelism: A Special Issue of The Journal of Supercomputing. Springer, 2011: 9-50.
- [7] GIORDANI M, POLESE M, MEZZAVILLA M, et al. Toward 6g networks: Use cases and technologies[J]. IEEE communications magazine, 2020, 58(3): 55-61.
- [8] SATYANARAYANAN M, BAHL P, CACERES R, et al. The case for vm-based cloudlets in mobile computing[J]. IEEE pervasive Computing, 2009, 8(4): 14-23.
- [9] MUCHNICK S. Advanced compiler design implementation[M]. Morgan kaufmann, 1997.
- [10] BACKUS J W, BEEBER R J, BEST S, et al. The fortran automatic coding system [C]//Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability. 1957: 188-198.

- [11] CONWAY M E. Design of a separable transition-diagram compiler[J]. Communications of the ACM, 1963, 6(7): 396-408.
- [12] AHO A V, ULLMAN J D. The theory of parsing, translation, and compiling: Vol. 1[M]. Prentice-Hall Englewood Cliffs, NJ, 1972.
- [13] JOHNSON S C, RITCHIE D M. Unix time-sharing system: Portability of c programs and the unix system[J]. The Bell System Technical Journal, 1978, 57(6): 2021-2048.
- [14] STALLMAN R M, MCGRATH R, SMITH P. Gnu make[J]. Free Software Foundation, Boston, 1988.
- [15] LATTNER C, ADVE V. Llvm: A compilation framework for lifelong program analysis & transformation[C]//International symposium on code generation and optimization, 2004. CGO 2004. IEEE, 2004: 75-86.
- [16] 刘颖, 吕方, 王蕾, 等. 异构并行编程模型研究与进展[J]. 软件学报, 2014, 25(7): 17.
- [17] GUIDE D. Cuda c programming guide[J]. NVIDIA, July, 2013, 29(31): 6.
- [18] NICKOLLS J. Scalable parallel programming with cuda introduction[C]//2008 IEEE Hot Chips 20 Symposium (HCS). IEEE Computer Society, 2008: 1-9.
- [19] BEZBARUAH M, DHAKULKAR S, PANDEY P, et al. Comparative analysis of gcc and llvm for performance optimization on aarch64[C]//2024 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2024: 1-6.
- [20] HJORT BLINDELL G. Tree covering[M]//Instruction Selection: Principles, Methods, and Applications. Springer, 2016: 31-76.
- [21] SMOTHERMAN M, KRISHNAMURTHY S, ARAVIND P, et al. Efficient dag construction and heuristic calculation for instruction scheduling[C]//Proceedings of the 24th annual international symposium on Microarchitecture. 1991: 93-102.
- [22] GRIFFITH A. Gcc: the complete reference[M]. McGraw-Hill, Inc., 2002.
- [23] JOHNSON S C. A portable compiler: Theory and practice[C]//Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1978: 97-104.

- [24] BILARDI G, PINGALI K. The static single assignment form and its computation [J]. Cornell University Technical Report, 1999.
- [25] LEIDEL J, KABRICK R, DONOFRIO D. Toward an automated hardware pipelining llvm pass infrastructure[C]//2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). IEEE, 2021: 39-49.
- [26] HUBER J, WEI W, GEORGAKOUDIS G, et al. A case study of llvm-based analysis for optimizing simd code generation[C]//International Workshop on OpenMP. Springer, 2021: 142-155.
- [27] FANG S, ZHENG W. Retrofitting control flow graphs in llvm ir for auto vectorization[A]. 2025.
- [28] FARVARDIN K, REPPY J. A new backend for standard ml of new jersey[C]// Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages. 2020: 55-66.
- [29] APPEL A W, MACQUEEN D B. Standard ml of new jersey[C]//International Symposium on Programming Language Implementation and Logic Programming. Springer, 1991: 1-13.
- [30] MPEIS P, PETOUMENOS P, HAZELWOOD K, et al. Developer and user-transparent compiler optimization for interactive applications[C]//Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2021: 268-281.
- [31] LAMMICH P. Refinement of parallel algorithms down to llvm[C]//13th International Conference on Interactive Theorem Proving (ITP 2022). Dagstuhl, 2022: 24-1.
- [32] BALASUBRAMANIAN K K, DI SALVO M, ROCCHIA W, et al. Designing risc-v instruction set extensions for artificial neural networks: An llvm compiler-driven perspective[J]. IEEE Access, 2024, 12: 55925-55944.
- [33] DE LA TORRE J C, JAREÑO J, ARAGÓN-JURADO J M, et al. Source code obfuscation with genetic algorithms using llvm code optimizations[J]. Logic Journal of the IGPL, 2025, 33(5): jzae069.

- [34] FAN Y, REGEHR J. High-throughput, formal-methods-assisted fuzzing for llvm [C]//2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2024: 349-358.
- [35] 刘玉, 刘谷, 耿锐. 基于 LLVM 实现的国产 DSP 优化编译器[J]. 中国集成电路, 2020, 29(7): 24-28.
- [36] 沈莉, 周文浩, 王飞, 等. swLLVM: Optimized Compiler for New Generation Sunway Supercomputer[J]. Journal of Software, 2023, 35(5): 2359-2378.
- [37] 吴忧. 基于 LLVM 编译器的自动调优算法研究与实现[D]. 西安电子科技大学, 2024.
- [38] EBNER D, KRALL A, SCHOLZ B. Instruction code selection[C]//Lecture Notes in Computer Science: Vol. 12608 Compiler Construction. Springer, 2021: 17-32.
- [39] EBNER D, BRANDNER F, SCHOLZ B, et al. Generalized instruction selection using ssa-graphs[C]//Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems. 2008: 31-40.
- [40] COLOMBET Q. Global instruction selection[C]//LLVM Developers' Meeting. San Jose, CA, USA: LLVM Foundation, 2015.
- [41] RONG Y, YU Z, WENG Z, et al. Irfuzzer: Specialized fuzzing for llvm backend code generation[A]. 2024.
- [42] COLOMBET Q, LARIN S, et al. Globalisel: Design, implementation, and future directions[C]//Proceedings of the LLVM Developers' Meeting. San Jose, CA, USA: LLVM Foundation, 2016.
- [43] BOGNER J. Fuzzing and testing instruction selection in llvm[C]//Proceedings of the LLVM Developers' Meeting. San Jose, CA, USA: LLVM Foundation, 2017.
- [44] SANDERS D. The globalisel combiner[C]//Proceedings of the LLVM Developers' Meeting. San Jose, CA, USA: LLVM Foundation, 2019.
- [45] ZHUFENG H, JIANDONG S. Optimization based on llvm global instruction selection[C]//Journal of Physics: Conference Series: Vol. 1856. IOP Publishing, 2021: 012004.

-
- [46] NACKE K, et al. Bringing globalisel to powerpc[C]//Proceedings of the LLVM Developers' Meeting. San Jose, CA, USA: LLVM Foundation, 2022.
 - [47] HOUTRYVE P. Extending the globalisel combiner with mir pattern support[C]//Proceedings of the LLVM Developers' Meeting. San Jose, CA, USA: LLVM Foundation, 2024.
 - [48] STADLER T. Directly generating gmir to speed up llvm compilation[C]//Proceedings of the LLVM Developers' Meeting. San Jose, CA, USA: LLVM Foundation, 2024.
 - [49] XIE J. Supporting scalable vectors in globalisel: A risc-v case study[C]//Proceedings of the LLVM Developers' Meeting. San Jose, CA, USA: LLVM Foundation, 2024.
 - [50] AMILKANTHWAR M. Eliminating globalisel fallback on aarch64 SIMD[C]//Proceedings of the LLVM Developers' Meeting. San Jose, CA, USA: LLVM Foundation, 2024.
 - [51] HICKEY N. Tracking and eliminating globalisel fallbacks with continuous integration[C]//Proceedings of the LLVM Developers' Meeting. San Jose, CA, USA: LLVM Foundation, 2025.
 - [52] 彭成寒, 李灵, 戴贤泽, 等. 深入理解 LLVM 代码生成[M]. 北京: 机械工业出版社, 2024.

致 谢

致谢

复旦大学 学位论文独创性声明

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。论文中除特别标注的内容外，不包含任何其他个人或机构已经发表或撰写过的研究成果。对本研究做出重要贡献的个人和集体，均已在论文中作了明确的声明并表示了谢意。本声明的法律结果由本人承担。

作者签名：_____ 日期：_____

复旦大学 学位论文使用授权声明

本人完全了解复旦大学有关收藏和利用博士、硕士学位论文的规定，即：学校有权收藏、使用并向国家有关部门或机构送交论文的印刷本和电子版本；允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。涉密学位论文在解密后遵守此规定。

作者签名：_____ 导师签名：_____ 日期：_____