



南京航空航天大学

# 本科毕业设计（论文）

题目      基于龙芯架构处理器的 GCC 编译器  
移植设计与实现

学生姓名	张润华
学    号	162010214
学    院	计算机科学与技术学院
专    业	计算机科学与技术
班    级	1620102
指导教师	施慧彬    副教授

二〇二四年六月



## 南京航空航天大学

### 本科毕业设计（论文）诚信承诺书

本人郑重声明：所呈交的毕业设计（论文）是本人在导师的指导下独立进行研究所取得的成果。尽我所知，除了文中特别加以标注和致谢的内容外，本设计（论文）不包含任何其他个人或集体已经发表或撰写的成果作品。对本设计（论文）所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

作者签名：张润华

日期：2024年6月15日

## 南京航空航天大学

### 毕业设计（论文）使用授权书

本人完全了解南京航空航天大学有关收集、保留和使用本人所送交的毕业设计（论文）的规定，即：本科生在校攻读学位期间毕业设计（论文）工作的知识产权单位属南京航空航天大学。学校有权保留并向国家有关部门或机构送交毕业设计（论文）的复印件和电子版，允许论文被查阅和借阅，可以公布论文的全部或部分内容，可以采用影印、缩印或扫描等复制手段保存、汇编论文。保密的论文在解密后适用本声明。

论文涉密情况：

☒ 不保密

☐ 保密，保密期（起讫日期：）

作者签名：张润华

导师签名：施慧彬

日期：2024年6月15日

日期：2024年6月15日



## 摘 要

现在国内计算机行业正在蓬勃发展，各种芯片、CPU、操作系统的新产品不断地涌出。现在在国产计算机领域做得比较好的是龙芯相关的一些产品。目前龙芯的计算机不但在军用和政府用途方面发挥巨大的作用，在民用方面也受到好评。近几年龙芯宣布不再借鉴 risc-v 指令集进行开发，而是全力研发自身的龙芯架构。龙芯指令集的 CPU 到现在为止已经发行了第三代。相对应的国产操作系统由于研发的历史比较长，也相对先进。编译器的发展和另外两者相比就比较缓慢。因此，怀有尝试解决这个问题的想法，本文将使用龙芯 32 位简化编译器将编译器移植到了目标的龙芯架构开发板上，验证移植工作的可行性。

龙芯 32 位简化编译链也称龙芯 GCC，是一个开源的供学习研究的编译链。在编译器进行自编译后就会形成能够使用的编译器。由于它的开源属性，学习者可以通过研究它的源代码，进一步学习编译器的内部结构，并且能够随意进行修改和调试。同时，龙芯 GCC 继承了 GCC 的优点，其中已经设置了许多目标机器的配套架构文件，这使得移植工作变得容易一些。要进行编译器的移植，修改操作系统、编译器的宏定义都是需要的。在文章中，首先对编译的理论流程和编译器的实际流程进行了介绍。然后会介绍编译器的测试操作系统 mos，并对 mos 的宏定义进行修改的部分和编译时的编译选项进行说明。为了体现龙芯编译器的可拓展性，本文会在编译器中新添加一条指令，这也是移植工作经常需要面对的问题。移植还需要对优化方面做一些修改，使得移植后速度不会变得很慢。

在本文的最后会进行相关指标的测试验证。需要验证主要有三点，一是使得编译器能够生成龙芯拓展指令集中的向量指令。二是能够正常生成和反编译添加的指令。三是修改的编译器编译的 mos 操作系统能够在开发板上运行。结果表明将生成的 mos 的文件系统和系统印象放到开发板上进行验证，结果说明基于编译器移植成功了。

**关键词：**编译器，指令生成，龙芯

## ABSTRACT

Now the domestic computer industry is booming, a variety of chips, CPU, operating system of new products constantly out. Now in the domestic computer field to do better is the loongson related products. At present, the Loongson computer not only plays a huge role in military and government use, but also in civilian use. In recent years, Loongson announced that it would no longer develop from the ratio c-v instruction set, but would fully develop its own Loongson architecture. The CPU of the Loongson instruction set has by now been issued for the third generation. The corresponding domestic operating system is relatively advanced due to its long history of research and development. Compiler development is relatively slow compared to the other two. Therefore, with the idea of trying to solve this problem, this paper will use the Loongson 32-bit simplified compilation chain to port the compilation chain to the target Loongson architecture development board to verify the feasibility of transplantation work.

Loongson 32-bit simplified compilation chain, also known as Loongson GCC, is an open source compilation chain for learning and research. After the compilation chain is self-compiled, a compiler can be used. Due to its open-source properties, learners can further learn the internal structure of the compilation chain by studying its source code, and are able to modify and debug at will. At the same time, Loongson GCC inherits the advantages of GCC, in which many supporting architecture files of the target machine have been set up, which makes the transplant work easier. To migrate the compiler, it is necessary to modify the operating system or the macro definition of the compiler. In this paper, the theoretical process of compilation and the practical process of compiler are introduced. The test operating system mos of the compiled chain is then introduced, and the modifications of the macro definition of mos and the compilation options at the compilation time are described. In order to reflect the scalability of the Loongson compilation chain, this article will add a new instruction to the compilation chain, which is also a problem that often needs to face in the transplantation work. The transplant also requires some modifications to the optimization so that the speed is not slow.

At the end of this paper, the relevant indicators will be tested and verified. There are three main points to be verified. First, the compilation chain to generate vector instructions in the Loongson extended instruction set. The second is to normally generate and decompress the added instructions. Third, the modified compiler compiled the mos operating system can run on the

---

development board. The results show that the file system and system impression of the generated mos are put on the development board for verification, and the results show that the compiler-based migration is successful.

**KEY WORDS:** Compiler, Instruction generation, Loongson

## 目录

第一章 绪论 .....	1
1.1 研究工作的背景和意义 .....	1
1.2 本次研究工作的任务 .....	1
1.3 国内外研究现状 .....	2
1.3.1 国外现状 .....	2
1.3.2 国内现状 .....	4
1.4 需要实现的任务和研究方法 .....	7
1.4.1 工作任务和要求 .....	7
1.4.2 工作流程 .....	8
1.5 论文的结构 .....	9
第二章 实验工具和理论 .....	11
2.1 龙芯的实验工具 .....	11
2.1.1 龙芯开发板 .....	11
2.1.2 龙芯 32 位简化编译器 .....	11
2.1.3 测试用操作系统 mos .....	12
2.1.4 龙芯 Qemu .....	13
2.2 编译器移植原理 .....	14
2.3 编译器的原理 .....	15
2.3.1 编译的流程 .....	15
2.3.2 编译器的工作流程 .....	16
2.4 优化原理 .....	17
2.4.1 并行计算的指令架构 SIMD .....	18
2.4.2 向量指令加速 .....	18
第三章 编译器移植工作介绍 .....	19
3.1 添加新的指令与指令级优化 .....	19
3.1.1 龙芯 32 位简化指令集介绍 .....	19
3.1.2 自定义指令选项 .....	20
3.1.3 修改 rtx 定义文件 .....	22
3.1.4 机器描述文件 .....	22
3.1.5 指令的匹配模式和窥孔优化 .....	23
3.2 向量指令优化 .....	25
3.2.1 明确编译选项 .....	25
3.2.2 使向量指令默认开启 .....	26
3.3 上板移植前后的准备 .....	27
3.3.1 安装龙芯 Qemu .....	27
3.3.2 修改和运行 mos .....	28
第四章 实验结果的验证 .....	32
4.1 验证的前置准备 .....	32
4.2 生成新指令部分的验证 .....	33
4.2.1 在汇编文件内产生自定义指令 .....	33
4.2.2 进行新指令的反汇编测试 .....	34
4.3 指令优化部分验证 .....	35
4.3.1 在汇编文件中生成浮点和向量指令 .....	35
4.3.2 和不使用浮点和向量指令的汇编代码进行对比 .....	36



4.4 在开发板上验证运行 mos .....	37
4.4.1 在开发板上运行 mos 系统 .....	37
4.5 实验结论 .....	41
第五章 工作总结与展望 .....	42
5.1 工作总结 .....	42
5.2 展望 .....	43
参考文献 .....	44
致谢 .....	45



## 第一章 绪论

### 1.1 研究工作的背景和意义

计算机高级语言的函数和变量在被处理的时候需要翻译成汇编语言，再被翻译成机器语言进行执行。编译器的作用是把高级语言翻译成可执行程序。不同的 GCC 编译器都处于高级语言和汇编语言之间，承担着翻译的功能。由于现在的操作系统存在着如 x86、mips、loongarch 等不同的架构，并且在未来还会架构的数量还会持续增多。GCC 编译器有时不能产生符合架构规定的代码，这时就需要对编译器进行移植，需要对帧栈结构进行调整，并为汇编代码的生成制定规则。

目前国产计算机公司都在努力研发和尝试应用国产的芯片。这样的公司主要的有六家：飞腾、鲲鹏、海光、龙芯、兆芯、申威。其中研究成果最为突出的是龙芯。龙芯不但推出了三代龙芯 CPU 芯片，还提供了配套的编译器以支持 CPU。龙芯编译器对 GCC 编译器做了改善，使得汇编代码的结构和指令种类变得更加兼容国产的 CPU 和操作系统。龙芯到现在为止一共发行了三个版本 1a32 的编译器，成功地将 x86 和 mips 架构等价映射到 1a32 架构上。在上述三个版本之外，龙芯公司还发行作为简化版本的 1a32r 供教育研究使用。由于 1a32r 的指令集是 1a32 指令集的子集，并且行为完全相同，所以能兼容 1a32r 的 CPU 必然能够兼容 1a32。然而由于龙芯编译器的技术还不是十分成熟，编译器产生的程序在一些龙芯机器上还不能完全兼容，需要修改编译器的宏定义文件和目标机器描述文件中的源代码才能执行程序。要实现计算机的全部功能需要操作系统、编译器、CPU 的合作，在国产操作系统和 CPU 都比较成熟的情况下，研究相对基础比较薄弱的编译器就显得十分重要了。由于制作新的编译器需要耗费太多时间，现在为新版本的 CPU 制作兼容的编译器都是靠原有相似的编译器进行移植实现的。这更显示出移植编译器工作的重要性。

综上所述，为了实现 1a32r 编译器在龙芯 CPU 开发板上的移植工作，本文将对 1a32r 的指令集进行拓展，同时兼顾优化问题，最终移植到目标的龙芯开发板上。目前国内对于 CPU 和操作系统的研究较多，编译器的研究就相对较少，所以本文对于以后的研究具有一定的参考价值。

### 1.2 本次研究工作的任务

本次研究任务有三点：第一点是 GCC 的工作流程和系统结构，包括前端、中间代码和后端代码，重点研究后端，包括目标宏定义文件和目标机器描述文件中源代码的修改方法。

研究龙芯处理器的硬件结构和指令集。在目标机器宏文件中添加龙芯处理器的专有寄存器、寄存器类别、帧栈布局和汇编输出相关的宏定义。第二点是在目标机器描述文件中定义和龙芯处理器指令集相关的指令模式和扩展模式，产生跳转操作、存储访问操作、算术操作等指令。第三点是进行生成龙芯处理器编译器所必需的环境配置，完成安装流程。然后使用 C 语言测试案例对编译器的功能进行测试，将生成的汇编文件在已有的开发板上进行验证。判断完成的指标有如下三点：

1. 能够编译生成新增加的指令
2. 能够优化指令的生成
3. 能够在开发板上运行测试

### 1.3 国内外研究现状

由于龙芯编译器属于国产的编译器，国外鲜有研究的报告，所以偏重普遍的编译器移植现状和编译器优化的方面来讲。而龙芯在国内近年发展势头迅猛。虽然发展较晚，龙芯编译器在三次迭代后也完成了国外需要三倍时间的研究量。所以，国内的现状方面会更侧重向龙芯编译器和编译器使用的龙芯指令集的优越性来介绍。

#### 1.3.1 国外现状

相比国内，国外的计算机发展历史比较长，因此计算机的种类也更多。为了应对不断涌现出来的不同架构的计算机，在国外基本是使用编译器移植的方法来解决这个问题。同时为了满足高级语言编程人员的特殊需求，GCC 的前后端都需要不断地形成特化的版本，这也需要移植来满足这个要求。因此，移植在国外显得尤为重要。

性能良好的编译器前端可以保护软件系统免受黑客利用潜在的安全漏洞的攻击，所以在一个编译器的移植后往往需要面对安全性的检测。但是现阶段国外对于编译器的检测程序并不全面，进行 debug 的时候有时会出现错误。例如，Csmith 和 Yarpngen 是两个著名的 C++ 程序生成器。然而它们主要生成完全语义有效的 C++ 测试程序，不具备产生类和模板的功能。其他的两种工具也有其他问题。Dharma 测试程序无法涵盖所有语法规则和替代方案，Grammarinator 在测试程序变得复杂时难以操作复杂的抽象语法树。Haoxin Tu 等人做出的 coft 以修正 GCC 和 Clang 的一些错误。它是通过将发现 bug 的步骤进行了细分，并设置了 bug 的过滤器，排除了大部分错误判别的可能性，使得错误率减少到原来的 39%[1]。在这种情况下，可以说移植的编译器很多问题都没有方法保证。

除此之外，国外最看重的还是编译器的编译效率，因此在一个 GCC 移植后都会被不断

地进行着优化。在一篇有关动态符号执行的论文中指出，软件测试在软件工程中起着重要作用，动态符号执行是白盒测试中流行的技术[2]。编译器优化可能会对 DSE 产生重大影响。实验中使用了编译器 KLEE，分别在有和没有编译器优化的情况下运行程序时。在优化前，需要近 85 秒；在优化后，需要 1.08 秒，差距十分明显。编译器可以通过设置约束求解、减少路径探索覆盖量来提升性能。这些操作都有可能被应用在移植后的优化上。这给本文的优化验证方法提供了灵感，可以通过编译器优化前后的状态进行对比。

国外更加先进的方法是通过递归神经网络对内存编译器性能进行分析[3]。传统的预测编译器性能的方法是给出内存的参数，如性能、功耗和面积，编译器为我们分配内存。而借助神经网络，用户可以通过输入数据（如总线速度、字深、字宽、库数）作为参数，让训练后的神经网络模型将预测尚未设计的内存的参数，以使用户可以相应地调整输入参数。使用递归神经网络，能够在极短的时间内准确预测给定编译器的参数，从而能够有效地搜索给定设计要求的编译器选项。这种方法不仅自动化了存储器选择和参数化过程，大大降低了其复杂性，而且还通过评估更广泛的可能参数化来改善参数结果。还有通过使用 TVM 深度学习的方法进行优化。优化使得在 CPU、GPU 和 DCU 平台上的平均加速比分别达到 1.02 倍、1.21 倍和 1.23 倍[4]。TVM 可以建立深度学习系统的编译器堆栈，它通过应用图级和算子融合优化遍历，使用调度原语将子任务图降级为 TE 表达式，使用自动调度模块搜索最佳调度配置，将每个子图降级为优化的 Tensor IR，最后使用目标编译器生成目标机器代码。

国外在进行操作系统和编译器的移植时，普遍认为使用嵌入式的移植，也就是结合开发板这种硬件设施的移植更加方便。广泛地使用 32 位嵌入式 CPU 不仅满足了高速提供处理的需求，而且能够使嵌入式快速系统稳定、可靠地运行。以 Linux 嵌入式系统的移植 Boot Loader 和 LED 驱动程序为例来说[5]。要移植的 Boot Loader 是用来完成系统启动和系统软件加载目标平台的操作程序存储在非易失性存储介质中[6]，在操作系统内核运行之前，其任务是通过建立内存空间映射来初始化硬件设备，使系统的软硬件环境进入适当的状态，完成操作系统内核的最终调用以准备正确的环境。移植的步骤是从官网下载移植的补丁，再加载 kernel。而 LED 驱动程序则需要做到通过一系列的功能和数据结构，它与硬件设备进行通信，同时遵循操作系统内核提供的统一接口；也支持动态添加到操作系统内核或由内核删除的独立组件。驱动程序还需要管理用户程序和外设和控制流之间的数据流。控制自己的地址空间，使得自己属于内核部分，通过设备文件自定义处理用户程序。

还有一篇移植基于 ARM 处理器的 Linux 处理器的文章[7]，详细地讲解了 ARM 的移植

方式。嵌入式 Linux 开发环境首先需要上位机、目标机和交叉编译器。上位机是一台安装了 Linux 并配置了交叉编译器的 PC, 可以交叉编译和调试; 除此之外还需要布置一些接口。之后需要配置嵌入式 Linux 内核、初始化设备驱动模块的管理机制、进行编译。编译后的内核镜像 zImage 表示在编译过程中会使用 gzip 算法对初始编译结果进行压缩。使用内核加载 zImage 就能移植成功。这和我使用操作系统的镜像文件来进行和验证移植的方法是有共通性的。我的很多后续操作也是从这里得到灵感的。

### 1.3.2 国内现状

国内现在的编译器有很大一部分是通过移植原有的编译器对新造的 CPU 进行适配。国内的几所大学在进行试验时都是通过修改 LLVM 的源码进行移植, 在嵌入式操作系统实现了指针、并行调度等操作[8]。可见编译器移植确实具有可行性。除此之外, 还有人将把 GCC 移植到矢量处理器上的[9], 也是通过在 mips 架构中修改目标宏定义文件和目标机器描述文件的源代码做到的。编译器通过不同的移植方案, 在对编译器优化了一些算法之后, 可以实现密码加密、能耗降低、仿真加速的功能。例如, 在优化了模调度的算法之后, 使得密码产生路径减少 15%, 启动间隔减少了 25%[10]。在 ILP 处理器上实现并行度调变算法优化, 可以使能耗降低 30%—65%[11]。

目前编译器最需要突破的问题在于冯·诺依曼机的局限性, 就是 CPU 提升带来的计算速度的快速增长和内存速度的增长速度不相匹配, 优化的速度跟不上 CPU 的发展速度。有时候内存的空间不足以支持快速运算。特别是现在的延迟隐藏技术, 会进一步加剧内存的紧缺问题。在余龙龙有关申威 GCC 编译器的论文[12]中采取的是通过编译器优化 Pass, 先对间接访存进行分析, 再通过软件预读取未来可能需要的数据, 也可以通过提前计算未来的寄存器访问的地址, 并加载到缓存, 这可能是当前国产编译器优化的一条重要途径。在国外的研究中对预读取这类研究仅仅局限于模拟硬件上, 并没有进行实际的开发。最主要的研究由 Lee 和 Morwy 进行。Lee 的研究通过评测常规内存的访问, 并没有考虑其他的情况, 并不完善。Morwy 通过整数排序进行评测, 也存在不完善的地方。所以, 在这个方面国内的研究进展快于国外, 在这方面进行研究很有优势。

编译器在设计帧栈结构时还需要考虑安全的问题, 编译器的安全问题和计算机的基址保护机制具有很强的联系。编程中最常见的现象就是数组访问越位, 计算机会弹出访问出错并且出现了内存泄漏的可能性。研究信息安全的计算机安全学科主要包括密码学、网络安全以及系统安全, 三者虽看似独立, 实际上却是互为支柱, 其中软件安全是关系着系统安全一个很重要的支项, 除了操作系统本身的稳健性外, 上面运行的各种软件也需要做好

自己本身的安全防护，避免成为一个可被利用的突破口，进而让系统受到威胁[13]。在国内，赖策等人的方法在 C 语言的安全子集给出了 124 条规则[14]，将语言安全子集中的编程语言规范分为推荐类和强制类以区分错误的危险程度。通过风险分级来实现提升编译器在安全检测方面的能力的同时不降低编译效率的目的。

比较先进的编译器还有使用深度学习的方法使编译器动态优化自身的样例，这类编译器是为卷积神经网络工作的。在这方面比较突出的是 XLA[15]。XLA 是谷歌推出的高性能机器学习领域编译器，用来编译 Tensorflow 设计的模型，从而解决 Tensorflow 在设计时因只考虑灵活性和可扩展性而带来的计算性能欠缺问题。池昊宇的在这方面的研究是国内相关研究中最突出的。他以原始特征数据为输入，执行时间为输出，建立了一个尺度特定的矩阵乘法式执行时间预测模型，在此基础上构建了最优分块大小的预测模型，但该预测模型只是针对人工神经网络，无法适用于多层嵌套循环的卷积神经网络。

在国内的编译器领域做得相对较好的是龙芯编译器。龙芯编译器中使用的是龙芯自主研发的龙芯指令。龙芯指令集也给了龙芯编译器相当大的优势。相比 mips 指令集和 arm 指令集，龙芯指令集不但采用了精简指令集计算机设计理念，指令执行速度快，处理器的性能和效率都较高，还支持 64 位地址空间，可以处理更大的内存地址和更复杂的数据计算。同时，还支持单指令多数据流的操作，可以同时多个数据进行并行计算，提高了向量计算和多媒体处理的效率。虽然龙芯的编译器本身还不成熟，但是指令集弥补了一部分缺陷，使得编译速度和效率增加不少。

龙芯指令系统还融合了 x86 和 arm 的功能。目前龙芯已经能够量产龙芯指令系统的芯片，在上面运行龙芯指令的效率比运行 mips 要高 40%，同时 loongarch-Qemu 翻译模拟定点整数和浮点数的效率分别是 Qemu 的 3.6 倍和 47.0 倍。可以看出，相比较而言龙芯指令集有着巨大的优势，较大的发展空间。龙芯指令的指令码都是 32 位，比较规整。龙芯指令还删除了过时的不适应当前硬件的架构，精简了指令集；也增加了一些由于硬件技术提高已经可以做到的指令。如间接跳转加上偏移量的操作，降低了超大距离跳转的开销。除此之外，还优化了指令发射顺序和硬件处理功能、具体细分了用户态和系统态的状态。通过增加硬件设施，龙芯传递指令时可以减少或者不用周转的寄存器。一些寄存器还可以通过分用和复用使得通用寄存器的数量得到增加。

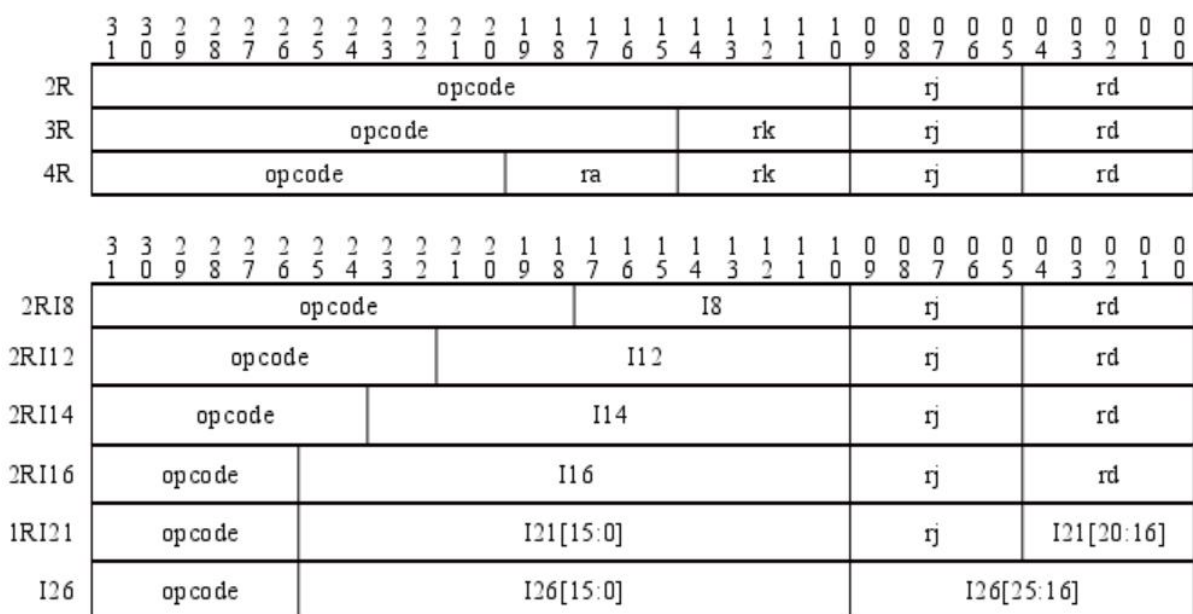


图 1.1 龙芯指令的九种编码格式图示

龙芯指令集的指令的编码较为工整。操作数的位置都是从最低位开始，操作码的位置总是在高  $n$  位。如果指令使用了立即数作为操作数，并且存在寄存器作为操作数，立即数的位置不会在最低位。立即数的长度和当前指令的种类有关系。龙芯架构总共有 9 种指令编码格式，其中有 3 种不含立即数的编码格式 2R, 3R, 4R，以及 6 种含立即数的编码格式 2RI8, 2RI12, 2RI14, 2RI16, 1RI21, I26[16]，具体的形式如图 1.1 所示。龙芯指令集的这种指令格式使得进行指令扩展时能够直接参照现有的指令标准，十分明确，扩展性好，而且对于 32 位的地址空间的利用率也极高。

在二进制翻译方面，loongarch 增加生成运算标志的运算指令以及根据运算标志生成转移条件的指令。x86 定义了独立的运算结果标志寄存 EFLAGS，这使得龙芯翻译的汇编代码比使用基础代码的编译器大幅减少。除此之外，loongarch 不仅拓展了 x86 的特殊寻址模式，还支持客户机虚地址到用户机实地址之间的直接翻译。这些都使得 loongarch 的编码数变得更少[16][17]。

龙芯还支持 mips 和 x86 到 loongarch 指令的二进制翻译以解决兼容性的问题。mips 与 loongarch 都是 risc 指令集，绝大多数常见指令都可以实现一对一翻译或者二对二翻译。在直接跳转和 switch 语句方面的翻译的效率更高。在翻译 x86 指令时，LATX 利用了 LBT 扩展的 x86 运算模拟指令、分支模拟指令以及浮点栈模式，大幅降低了对应指令的翻译开销。在翻译的时候能做到二对一的翻译。龙芯的翻译功能使得在不同目标机器之间的移植工作变得比较轻松。高效的翻译使得 loongarch 架构的目标机器的运行速度更快。龙



芯二进制翻译的过程如图 1.2 所示。

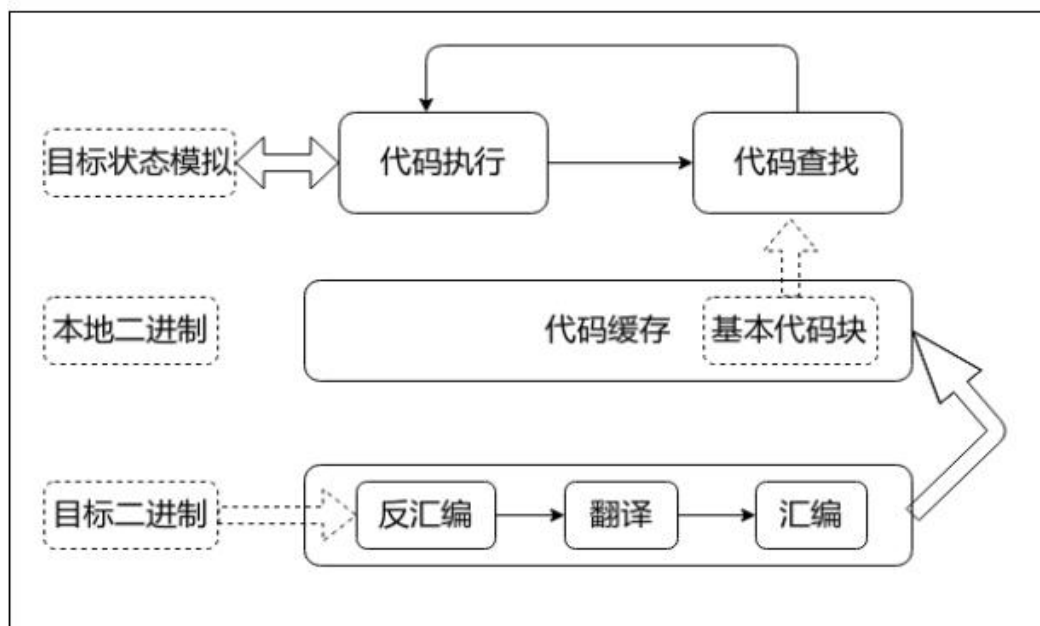


图 1.2 龙芯进行二进制翻译的过程

龙芯指令集也能够通过多线程的技术加快访问的速度。主要的做法是使用链接器优化和指令搬迁。如果能保证候选代码序列中的每条指令都与一个或多个重定位相关联，则 ABI 可以将语义分配给重定位类型，以确保代码序列转换是安全的。在这种情况下，可以将指令划分从单个的窗口，每次只分析窗口中的指令。每次窗口转换后，暴露的相邻窗口之间就有可能存在优化机会，因此可以多次调用窥视孔优化以最大限度地提高性能。指令搬迁是将动态重定位转化为静态重定位项。对于在链接阶段优化的指令，通常会在组装阶段为指令生成一个重定位项，该重定位项仅用于标记优化的指令。当链接器执行程序链接时，它会根据此重定位项修改指令[18]。后续的优化函数将使用该重定位条目进行条件判断，并最终实现相关指令优化。就是通过这两种方法，龙芯编译器的效率得到了进一步的提升。

## 1.4 需要实现的任务和研究方法

### 1.4.1 工作任务和要求

在这次毕设中需要实现的任务有三点，最主要的任务是编译器的移植问题。这项任务需要使用编译器编译测试作用的操作系统 mos，再将 mos 运行在目标开发板上观察结果。正常运行则代表实验完成。在实验过程中会用到 Qemu、vivado、PuTTY 等工具进行模拟和测试。同时还需要研究开发板的使用方法，据此修改软件的宏定义。

除此之外，还有两个支线任务，分别为实现新指令和实现指令优化。实现新指令首先需要研究编译器的工作流程和追踪过程中提供定义的文件。由于龙芯编译器的移植没有前

例，还需要借鉴 risv、arm 等框架的修改方法。然后需要在 Qemu 上进行测试，确认新指令不会出现不能识别或者识别出现问题的情况。再通过使用多种指令集的情况下证明新指令不存在冲突问题。实现指令优化需要研究龙芯的指令集的构造，尽可能使用扩展的指令集，以发挥高级指令的优势，减少组合基础指令的多余开销。同时也需要目标机器的宏定义架构，如寄存器和帧栈结构等信息，在对应的 md 文件内添加寄存器的匹配规则和限制条件。优化措施还可以通过优化定义来进行设置，例如寄存器优化和窥孔优化。

### 1.4.2 工作流程

毕设的工作的流程图如图 1.3 所示，一共有三个部分。第一部分是最上方的部分，这和硬件有关系，需要管理开发板和开发板的通信方法，需要仔细查看开发板的说明书。在查明开发板的串口型号和交换的波特率后，使用 vivado 和 PuTTY 两个软件对开发板进行连接调试。调试在初期和验证阶段都需要进行。初期需要对测试用操作系统 mos（详见第二章）的状态进行分析，并根据龙芯架构对其参数进行调整。后期需要对修改后的编译器编译后的 mos 进行运行验证，也需要用到开发板进行调试。

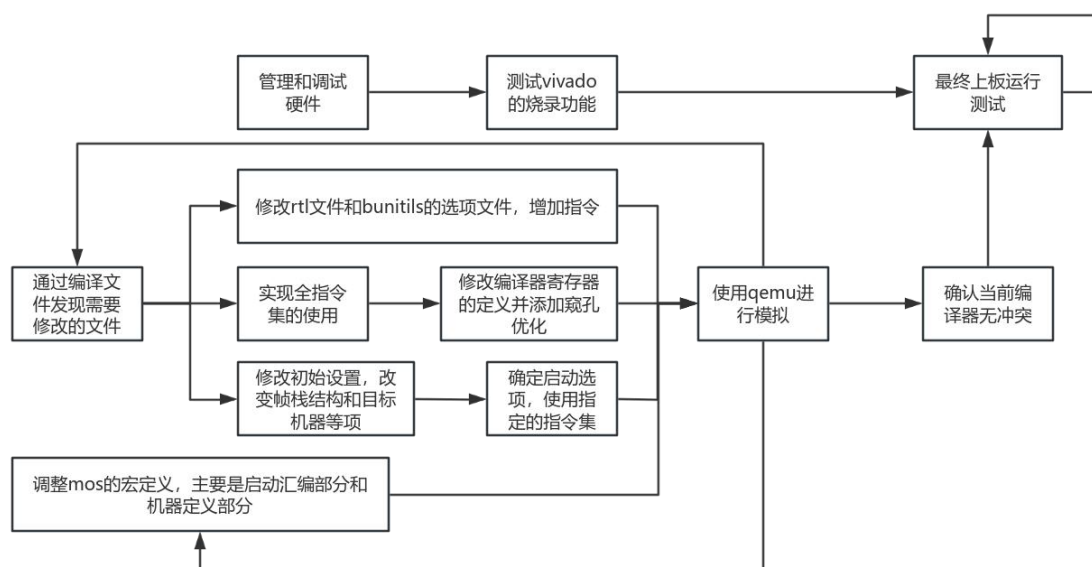


图 1.3 实验的工作流程图

中间的部分是最主要的部分，主要研究增加指令的方法、寄存器以及指令优化和指令集启动的条件，涉及的文件数量极多。其中三个支流分别对应了添加新的龙芯指令、指令优化和移植编译器三个实验目标。三个任务虽然并行，但验证的顺序却有不同。首先需要做的是新指令的添加，它涉及指令集的统一性问题，需要保证新指令与其他的龙芯指令保持兼容。这部分的任务主要在编译器的 bunitils 文件夹和 rtx 定义文件中实现。同时还需要在机器描述文件中添加相应的指令定义，这对指令的定义、匹配、行为和优化方式都

起到了定义的作用。接下来就可以进行指令的优化。指令的优化分为两个部分。第一部分是开启龙芯指令集的所有的指令使用权限。使用常规优化等级下的龙芯编译器时，龙芯编译器不会生成 fix 指令集之外的指令，也就是不会生成浮点运算指令和向量运算指令。通过对编译器初始化做特殊的设置和打开一些指令选项，在当前的待编译文件中如果存在浮点数计算或者数组级别的运算，就可以分别产生浮点数计算指令和向量指令。其中向量指令分为 128 位向量指令和 256 位向量指令，使用的情况也和当前的编译选项和数组计算的长度有关。指令优化的第二部分是关于实现一些寄存器的优化和窥孔优化，这部分和添加指令的部分逻辑相似，主要也是在机器描述文件以及附属的限制文件中做修改。但这部分的优化主要参考 risc-v 指令集的编写逻辑和原有的优化方案写出的，并不能够做到量化优化的效率，因此只会做简要介绍。在前两个任务完成后就可以开始完成编译器的移植了。首先需要在编译器的初始化时，对目标机器选项做指定，调整编译器运行时的帧栈结构和编码逻辑。编译器配置好只需要等待 mos 准备好就能够进行编译了。

下方部分测试用操作系统 mos 的帧栈结构和启动选项做修改。具体而言是对 mos 的宏定义进行修改，让它和编译器匹配。这部分属于编译器移植的一部分，需要在确定启动选项的部分之前添加。后期公共的部分有使用 Qemu 和开发板实验，这属于反复模拟和验证的过程。可以起到阶段性地对当前的移植工作做总结的作用。

## 1.5 论文的结构

论文总共有五章。章节的内容如下：

第一章是绪论，先是阐述了编译器移植的研究背景。再介绍了本次毕设的需要完成的任务和判断任务完成的指标。在这之后又介绍了国内外以 GCC 编译器为主的编译器的发展现状和未来发展趋势。阐明了移植编译器的重要性。再紧接着是对于毕设实验流程的介绍。在这章的最后介绍了论文的结构。

第二章主要叙述实验工具及其理论。使用的工具有龙芯 3A4000 开发板、龙芯 32 位简化编译器、龙芯 Qemu 和操作系统 mos。之后分别介绍了编译器移植的原理、编译原理和优化原理。为第三章的实验部分铺垫了理论基础。

第三章部分主要讲解了分别讲解了为了实现三个实验目标需要做的工作。先是介绍添加新指令的做法，其中夹带了寄存器优化和窥孔优化的讲解，这两者的原理基本相同。然后讲解产生浮点指令和向量指令的部分，分为两种实现方法。最后讲解了在开发板上运行 mos 的实验前后需要准备。

第四章分别展示了三个实验目标的验证实验成果时的详细步骤和结果图，并分析了结果图符合毕设标准的原理以及原因。并且在最后对整个实验的成功做了总结。

第五章是对毕设中的工作做出了总结，分析出在毕设中完成的四项工作，并且分析了本次毕业设计的缺憾之处，展望了未来可以做的事。

## 第二章 实验工具和理论

### 2.1 龙芯的实验工具

#### 2.1.1 龙芯开发板

在这次实验中使用的是龙芯 3A4000 开发板。开发板的结构如图 2.1 所示。龙芯 3A4000 开发板搭载了龙芯 3A4000 处理器，这是龙芯公司推出的一款高性能处理器。该处理器采用 64 位 mips 架构，拥有多核心设计，支持 SIMD 指令集，具有较高的性能和能效比。这个开发板配备了 DDR4 内存插槽，支持最新的 DDR4 内存规格，可以提供高速和稳定的内存访问性能。龙芯开发板也提供了闪存存储器或固态硬盘接口，用于存储操作系统、应用程序和用户数据。同时开发板还提供了丰富的接口和扩展插槽，包括 USB、PCI Express、以太网、串口等，以支持外部设备的连接和扩展。此外，还包括了 HDMI、VGA 等视频输出接口。龙芯的开发板也支持多种操作系统，包括 Linux 发行版和其他嵌入式操作系统，为开发人员提供了灵活的软件开发环境。龙芯 3A4000 处理器在处理高性能计算、大规模数据处理和并行计算等工作负载时表现十分优异，具有较高的计算能力和能效比。

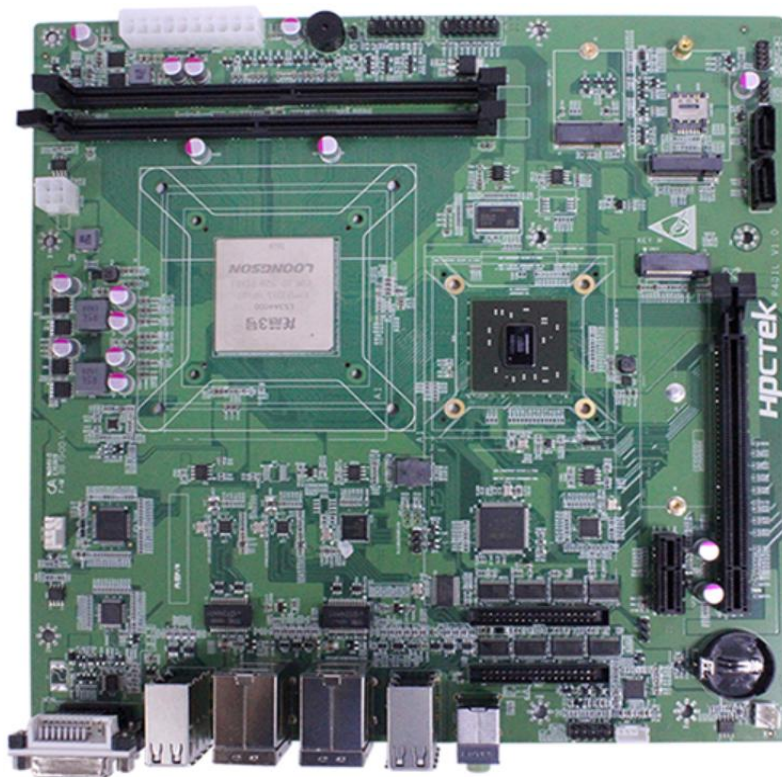


图 2.1 龙芯 3A4000 开发板

#### 2.1.2 龙芯 32 位简化编译器

本次实验使用的编译器是龙芯 32 位简化的编译器。在龙芯编译器当中有四个部分，

分别为 `la32r_binutils`、`la32r_gcc-8.3.0`、`la32r_glibc-2.28` 和 `la32r-Linux`。其中 `la32r_binutils` 主要是用来创建、操作和转换二进制文件的工具集合。提供的服务是处理目标文件、可执行文件和共享库等文件。一般在编译了这个文件夹下的文件之后就能生成编译程序了,但有时会缺少头文件的支持。实验中使用的 `loongarch32r-linux-gnusr-gcc`、`loongarch32r-linux-gnusr-as`、`loongarch32r-linux-gnusr-objdump` 都是在这生成的。`la32r_gcc-8.3.0` 储存的是一些机器相关和模式相关的配置代码。`la32r_gcc-8.3.0` 为编译器提供了许多基本的系统函数和服务,用于编写和运行 C 和 C++ 程序,是支持龙芯的 C 语言库文件。`la32r_glibc` 实现了 C 语言的标准库函数,如字符串操作、内存管理、文件操作、输入输出、时间和日期处理等。`la32r-Linux` 的文件主要是源代码文件、配置文件以及构建工具链。这些文件包括处理器的特定优化、寄存器定义、中断处理、内存管理和设备驱动等方面的代码。可以看出如果需要添加代码,就需要修改 `la32r_binutils` 的文件。如果需要改变生成指令的行为就需要修改 `la32r_gcc-8.3.0` 的内容。

### 2.1.3 测试用操作系统 mos

操作系统 `mos` 是北京航空航天大学研发的一款供移植测试的轻量级操作系统。由于其大小约为 5MB,编译起来十分方便,其组成结构如图 2.2 所示。在通过龙芯编译器编译后会在 `target` 文件夹下出现 `mos` 和 `fs.img` 两个文件,两者是可以在开发板上运行的系统镜像文件和文件系统文件。在开发板上运行时需要将这两个文件保存到 SD 卡上,再插入开发板运行操作系统。这样用户就可以通过在龙芯开发的 `la32-Qemu` 上模拟运行 `mos` 操作系统,为测试编译器提供了不少方便。

在 `mos` 文件系统下还存在一些用户应用,它们在 `user` 文件夹底下。运行 `mos` 之后,可以在里面模拟 linux 命令行的一些功能。比如说现在存在的有 `init.c`、`pingpong.c`、`ls.c` 分别模拟了操作系统启动功能、多线程功能、文件系统功能。可以通过修改 `user` 文件夹底下的 `include.mk` 添加自定义的用户程序。使用用户应用验证起来比较有说服力,这是我选择 `mos` 作为测试程序的原因。

除此之外,选择 `mos` 的原因还有 `mos` 的可塑性很好,能够轻易修改系统参数。在 2023 年 12 月毕设工作开始时,`mos` 的功能还不能支持复杂的浮点运算,用户文件夹下也不能运行用户程序,文件系统也有缺陷。当时做实验会出现许多软件上和逻辑上的错误。到 2024 年 2 月的版本就修复了文件系统和浮点运算的相关缺点。在同年 4 月的版本中增加了用户程序的功能,为最后的实验结果增加了说服力。在本文中,最终使用的版本是 4 月版本的 `mos`,已经十分完善。至今在实验中唯一不能实现的是向量指令的生成,因此向量指令的



使用未能在开发板上得到验证。

在实验中，目的是验证编译器编译出来的 mos 操作系统的汇编代码能够符合开发板的标准。这不仅需要在开发板上测试运行这个 mos 操作系统，还需要保证其中的用户程序能够在 mos 运行的时候能够正常运行。作为三个典型功能的代表，只需要验证上述三个功能就可以判断移植工作的成功。

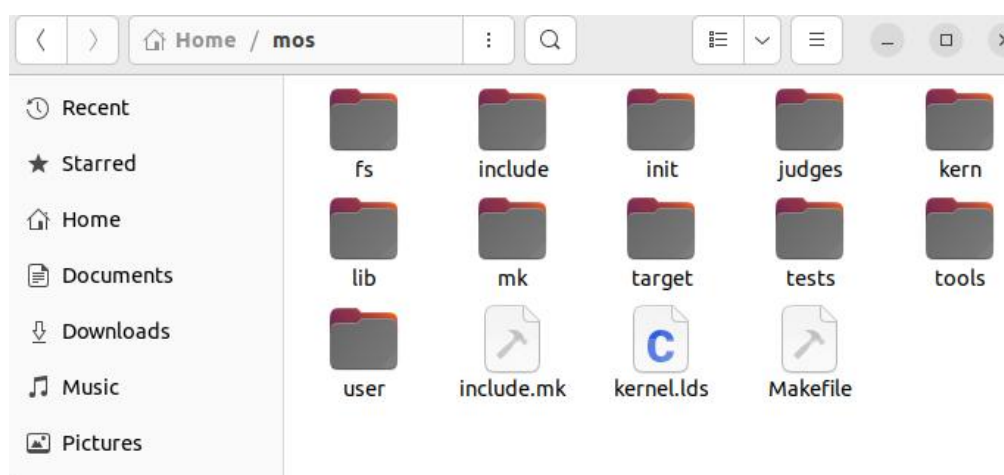


图 2.2 操作系统 mos 的内部结构

```
zrh@zrh-virtual-machine:~/la32r_ans$ make run
INITRD_OFFSET=3800000 qemu-system-loongarch32 -M ls3a5k32 -m 64M -nographic
reboot -kernel target/mos -initrd target/fs.img
loongson32_init: num_nodes 1
loongson32_init: node 0 mem 0x4000000
init.c: la32r_init() is called
Memory size: 57344 KiB, number of pages: 14336
to memory 8042a000 for struct Pages.
pmap.c: la32r vm init success
FS is running
icode: open /motd
superblock is good
read_bitmap is good
icode: read /motd
This is /motd, the message of the day.
Welcome to the MOS kernel, now with a file system!
```

图 2.3 启动 Qemu 打开 mos

#### 2.1.4 龙芯 Qemu

龙芯的 Qemu 是一款开源的虚拟化和模拟器软件，被用于模拟运行龙芯处理器架构的计算机系统。龙芯 Qemu 允许在不同的主机平台上运行虚拟的龙芯系统，从而方便开发者进行应用程序和操作系统的调试、测试和开发工作。龙芯 Qemu 可以在主机平台上模拟运行龙芯处理器架构的虚拟机。这使得开发者可以在主机系统上运行和测试适用于龙芯的软件，而无需考虑实际的龙芯硬件。龙芯 Qemu 支持指令级模拟，可以准确地模拟龙芯处理器的指令集架构，包括龙芯 1、龙芯 2 和龙芯 3 等不同版本。另外，龙芯 Qemu 支持快速创

建和管理虚拟磁盘映像，这使得用户可以在虚拟环境中运行和测试不同的操作系统和应用程序。正是这几个特点，使得龙芯 Qemu 可以省去烧录 SD 卡等步骤的过长的等待时间，直接进行结果的验证。龙芯 Qemu 运行 mos 的情况如图 2.3 所示，是在 mos 的文件夹下使用“make run”指令调用龙芯 Qemu，并成功模拟运行 mos 的结果图。

## 2.2 编译器移植原理

编译器移植是实质是指将编译器及操作系统从当前架构的机器移植到其他架构的机器的过程。具体来说是要实现编译器编译后的文件能够在目标机器上运行的目标。移植操作的关键在于改变编译产生的汇编代码。每一个框架下指令集中的指令的名称和作用都有所不同。原版 GCC 和龙芯编译器的编译相同一份 C 语言文件的差距如图 2.4 所示。即使使用同一架构下的指令集，根据发行版本也会有所不同。比如说实验中使用的 1a32r 编译器就分三个版本，除此之外还有 1sx、1asx 等几个分支版本。这几个产生代码的逻辑相同，但是在基础上龙芯指令集各自增加了自己的拓展指令集。

test2.s	test2.s
1 .file "test2.c"	1 .file "test2.c"
2 .text	2 .text
3 .align 2	3 .globl main
4 .globl main	4 .type main, @function
5 .type main, @function	5 main:
6 main:	5.LFB0:
7 addi.w \$r3,\$r3,-32	7 .cfi_startproc
8 st.w \$r22,\$r3,28	8 endbr64
9 addi.w \$r22,\$r3,32	9 pushq %rbp
10 la.local \$r12,.LC0	9 .cfi_def_cfa_offset 16
11 ld.w \$r13,\$r12,4	1 .cfi_offset 6, -16
12 ld.w \$r12,\$r12,0	2 movq %rsp, %rbp
13 st.w \$r12,\$r22,-24	3 .cfi_def_cfa_register 6
14 st.w \$r13,\$r22,-20	4 movsd .LC0(%rip), %xmm0
15 addi.w \$r12,\$r0,11 # 0xb	5 movsd %xmm0, -8(%rbp)
16 st.w \$r12,\$r22,-28	6 movl \$11, -12(%rbp)
17 or \$r12,\$r0,\$r0	7 pxor %xmm0, %xmm0
18 or \$r13,\$r0,\$r0	8 movq %xmm0, %rax
19 or \$r4,\$r12,\$r0	9 movq %rax, %xmm0
20 or \$r5,\$r13,\$r0	9 popq %rbp
21 ld.w \$r22,\$r3,28	1 .cfi_def_cfa 7, 8
22 addi.w \$r3,\$r3,32	2 ret
23 jr \$r1	3 .cfi_endproc
24 .size main, .-main	4.LFE0:
25 .section .rodata	5 .size main, .-main
26 .align 3	5 .section .rodata
27 .LC0:	7 .align 8
28 .word 1015124550	8.LC0:
29 .word 1076974194	9 .long 1015124550
30 .ident "GCC: (GNU) 8.3.0"	9 .long 1076974194
31 .section .note.GNU-stack,"",@progbits	1 .ident "GCC: (Ubuntu 11.4.0-1ubuntu1-22.04) 11.4.0"
	2 .section .note.GNU-stack,"",@progbits
	3 .section .note.gnu.property,"a"
	4 .align 8
	5 .long 1f - 0f
	6 .long 4f - 1f
	7 .long 5
	8: .string "GNU"

图 2.4 龙芯 GCC 编译的结果（左） 原版 GCC 编译的结果（右）

移植中使用的测试文件可以是单个的 C 语言文件，也可以是复杂的操作系统。如果移植的目标机器是本身没有可视化界面的开发板，由于操作系统便于远程调试的特点和复杂的编译环境，操作起来比较方便，并且能够得到几乎所有情况下的运行数据。因此，使用操作系统作为测试对象是非常合适的。能够在目标机器上运行表明编译器的编译逻辑符合目标机器的标准，同时也是移植成功的标准。如果使用的测试用操作系统不能支持某种指令的运行，比如说这次使用的操作系统 mos 就不能使用向量指令，这时就可以通过编写单



独的测试文件来进行测试相关指令的生成。还有这次新添加的指令，操作系统也没有它的相关定义。因此它的验证也是通过相同的方法做到的。

在移植的过程当中，需要对编译器本身进行移植，使得编译器生成的汇编代码能够被目标机器识别并处理。同时也需要对测试用操作系统本身做一些修改，使之符合编译器和开发板的栈定义和启动参数。编译器、操作系统和开发板三者的定义必须保持协同。所以说，这虽然是移植编译器的过程，其实也是移植操作系统的过程。

编译器移植主要包括四个步骤。首先需要分析目标平台和编译器源代码。这需要详细了解目标机器的体系结构、能够支持的操作系统等特性，以及编译器源代码的结构、模块、依赖关系等。其次是要修改编译器源代码。在这个步骤需要对编译器的源代码进行调整、添加新的目标平台相关的代码、修改代码生成器、调整优化策略等。移植完成后，需要对编译器进行调试和测试，以确保在新平台上的正确性、稳定性和性能。这包括编译器的功能测试、性能测试、兼容性测试等。还需要性能优化，一旦编译器在新平台上能够正常工作，为了弥补移植带来的效率损失，必须进一步优化编译器，以提高其在目标平台上的性能和效率。这个步骤涉及对代码生成器、优化策略、内存管理等方面的优化。

## 2.3 编译器的原理

### 2.3.1 编译的流程

编译的流程就是使用编译原理翻译代码的流程。总体而言会有六个步骤：词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成[19]，如图 2.5 所示。在六个线性步骤外，还需要进行符号表的管理。下面会具体讲一讲它们的作用。

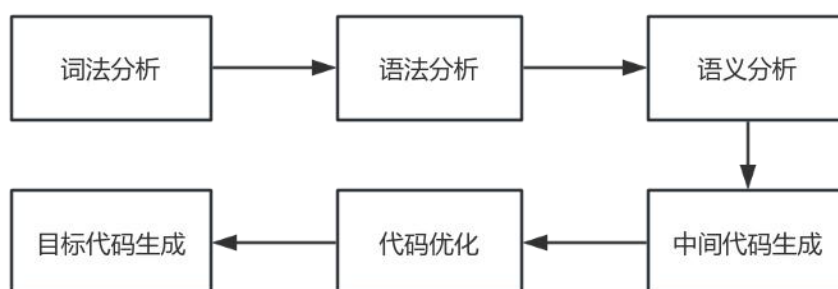


图 2.5 编译的六个步骤

词法分析根据编程语言的词法规则将源代码划分为一个个词法单元。它通过有限状态自动机或者正则表达式来扫描和识别源代码中的词法单元。词法分析还会过滤掉不必要的空白字符和注释。每个词法单元都有一个相关联的标识符。接下来的语法分析会将词法单

元序列组织成语法结构，最常见的形式是语法树或抽象语法树。语法分析会使用文法规则（通常是上下文无关文法）来验证源代码的语法正确性，使用算法（如递归下降、从左到右等）来解析语法结构，并生成一个根据语法规则推导出的语法树或抽象语法树。在语法分析的基础上，语义分析对识别出的语法结构进行语义检查。它检查和确保程序的语义正确性，例如类型检查、声明和使用检查、作用域解析等。语义分析使用符号表来存储和管理程序中的符号和标识符信息。在中间代码生成阶段，源代码被转化为一种介于高级语言和目标机器码之间的中间表示形式。这种中间代码通常更容易进行分析和优化。在中间代码生成阶段，中间代码可以是三地址代码、虚拟机代码等。这个阶段可能会使用一些基本优化，以提高中间代码的效率。

前三个操作和毕设的任务关系并不是很大。对于三个分析，只要上层使用的语言是相同的，它的分析逻辑就不会变化，这是平台无关的。这些功能主要是由 `bison` 和 `flex` 分析器承担的。在这两个程序下通过编写正则表达式来构建 `yytranslate` 和 `yylex` 等文件进行规定。实验中需要做的是移植工作，所以和这些没有直接联系。中间代码的优化实质是指高级指令转化为第一类中间代码（`gimple`）指令的过程，也是平台无关的。

代码优化具有可选择性。它在中间代码生成和目标代码生成之间发生。这一阶段应用各种优化技术和策略，以改进程序的执行效率和资源利用率。例如，常量折叠、循环优化、控制流优化和数据流分析都是常见的优化技术。在目标代码生成阶段，中间代码被转换为与特定目标体系结构相关的机器代码。这包括指令选择、寄存器分配、栈帧管理、代码布局等操作。目标代码生成器使用目标机器的特性和约束，生成一系列可在目标体系结构上执行的机器指令。

代码优化是从第二类中间代码（`rtl`）生成的一个阶段。在项目对中间代码的寄存器匹配模式进行调整实际上就是影响了这个过程。这是能够进行优化的第一个地方。也正是因为这样，在编译器优化选项被关闭的时候，这类优化仍然能够生效。目标代码生成阶段的本质是中间代码到汇编代码的过程。因此它是本文工作最集中的部分，添加代码、代码优化、修改宏定义和帧栈结构等工作都会在这部分实现[20]。

符号表管理包括符号的声明和定义的记录、作用域解析、类型检查等功能。编译器通过符号表进行名称解析和语义检查。这是在六个操作之外的维护工作，其实就是上方提到的 `bison` 和 `flex` 的工作项目。三个分析工作都是依靠符号表来工作的。

### 2.3.2 编译器的工作流程

编译器的处理流程主要有三个步骤，先是从高级语言转化为 `gimple` 指令，再从 `gimple`

指令转化为 rtl 指令，最后从 rtl 指令转化为汇编指令。这三个步骤只有最后一个和目标机器有关，其他两个和目标机器无关。关于 rtl 指令转化为汇编指令的过程在后面具体提到。如图 2.6 所示。



图 2.6 编译器的工作流程图示

从高级代码到 gimple 代码的基本思想基本就是 2.1.1 提到的编译原理。具体流程有三点：一是词法分析和语法分析。在这个过程中，编译器首先会对源代码进行词法分析和语法分析，将其转换为抽象语法树（AST）或类似的中间表示形式。二是语义分析，编译器会进行语义分析，检查源代码是否符合语言规范，同时收集变量、函数等的声明信息，并构建符号表。最后是中间表示生成，在这个阶段，编译器会将源代码转换为中间表示，其中 gimple 是 GCC 的一种 IR 格式。gimple 代码是一种类似于 SSA（静态单赋值）的指令，它简化了后续优化和代码生成的过程。

从 gimple 到 rtl 主要是 gimple 的分析优化和 gimple 到 rtl 转换。编译器会对 gimple 进行各种分析和优化，例如常量传播、死代码消除、循环优化等。这些优化旨在提高程序的性能和效率。接下来，编译器将 gimple 转换为 rtl。rtl 是一种低级表示形式，它更接近于目标机器的硬件特性，包括寄存器和指令级别的操作。

从 rtl 到汇编代码有两步，寄存器分配和调度和汇编代码生成。在寄存器分配中，编译器会对 rtl 进行寄存器分配和调度，将虚拟寄存器映射到物理寄存器，并优化指令的调度顺序，以提高程序的性能。在汇编代码生成的最后，编译器将经过寄存器分配和调度后的 rtl 转换为目标机器的汇编代码。这些汇编代码是机器可以直接执行的指令序列，它们与目标机器的体系结构密切相关。

## 2.4 优化原理

本文实现的优化方法是将常规指令转化为浮点运算指令和向量运算指令。仿照学习的有寄存器匹配优化和窥孔优化。在这里主要说明的是向量运算指令对于指令优化的意义，即向量指令的加速能力和与常规指令相比的优势。寄存器匹配优化和窥孔优化可以参考第三章第一节的介绍。

### 2.4.1 并行计算的指令架构 SIMD

SIMD 是一种并行计算的指令集架构，用于同时对多个数据执行相同的操作。在 SIMD 架构中，一条指令可以并行地作用于多个数据元素，从而实现高效的数据并行处理。这种并行计算方式适用于各种计算密集型应用，例如图形处理、信号处理、数字信号处理、科学计算等。SIMD 架构中的基本概念是向量化操作。在向量化操作中，一条指令可以同时为一个向量（即多个数据元素）中的所有元素执行相同的操作，而不是逐个元素地执行。这样就可以在单个时钟周期内处理多个数据元素，从而提高了计算效率。在大长度的数进行数值运算通常可以使得用时和 32 位数加减用时相当，相当于提高了数倍运算能力。

SIMD 架构通常由多个处理单元组成，每个处理单元都可以同时执行相同的指令，但作用于不同的数据元素。这样，即使是在单个处理器中，也可以实现并行计算的效果。一般而言，SIMD 处理器还会提供一些特殊的指令，用于加载、存储和操作向量数据，以及进行数据的重新排列和归约等操作。龙芯的处理器往往有四个以上的 CPU 用作处理单元，所以能够支持向量化的运算，使用向量化运算至少能够加速 4 倍。

### 2.4.2 向量指令加速

向量指令在上文提到的 SIMD 架构的加速中起到了主要的作用，向量指令能够加速程序执行的原因有四点。一是能够减少指令的执行次数，向量指令允许一条指令同时作用于多个数据元素，而不是逐个执行。例如，一条向量指令可以同时为一个向量中的所有元素执行相同的操作。这样可以减少指令的执行次数，提高了指令级并行性。二是能够利用硬件并行性，现代处理器通常具有专门的向量处理单元，可以同时处理多个数据元素。通过向量指令，处理器可以将多个数据元素同时加载到向量寄存器中，并且在同一时钟周期内对这些数据元素执行相同的操作。这样可以充分利用处理器的硬件并行性，加速计算过程。三是能够实现数据的重用。向量指令通常会将多个数据元素一次性加载到向量寄存器中，并且在寄存器中重复使用这些数据元素。这样可以减少内存访问的次数，并且提高了数据重用的效率。最后一点是能够内存访问优化。向量指令通常会对内存访问进行优化，例如向量指令可以通过数据预取、数据对齐等技术来提高内存访问的效率。这样可以减少内存访问的延迟，并且提高了内存带宽的利用率。

### 第三章 编译器移植工作介绍

#### 3.1 添加新的指令与指令级优化

##### 3.1.1 龙芯 32 位简化指令集介绍

龙芯的 32 位简化指令集具有 risc 指令集的典型特征。它的指令长度固定且编码格式规整，绝大多数指令只有两个源操作数和一个目的操作数，采用 load/store 架构，即仅有 load/store 访存指令可以访问内存，其他指令的操作对象均是处理器核内部的寄存器或指令码中的立即数。龙芯架构采用基础部分加扩展部分的组织形式。其中扩展部分包括：二进制翻译扩展、虚拟化扩展、向量扩展（LSX）和高级向量扩展（LASX）。龙芯架构内部的结构如图 3.1 所示。

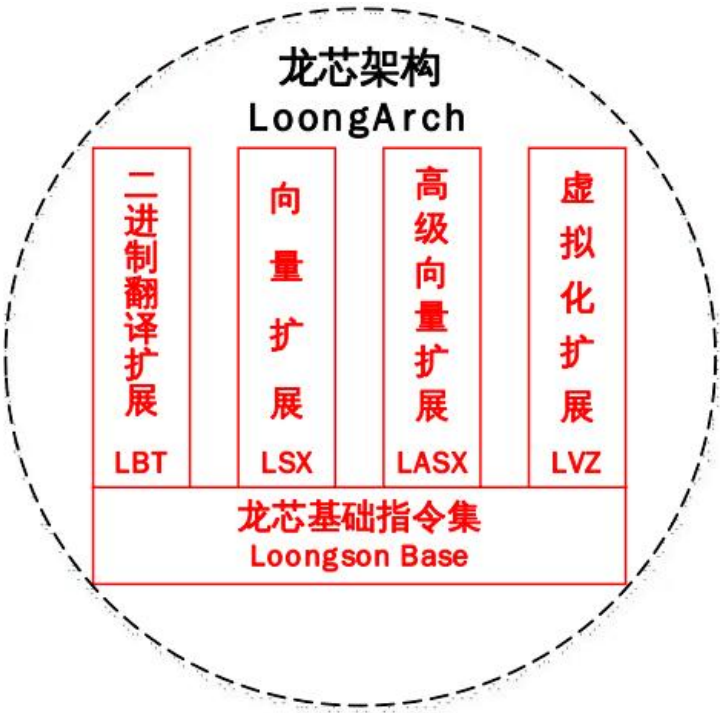


图 3.1 龙芯架构的内部结构

龙芯的基本功能基本存在于 fix 指令集。test 指令集只包含一部分整型数学运算指令集的内容。fix 指令集包含了宏指令集、特权指令集、整型数学运算指令集、存取指令集、跳转指令集。除此之外，还有独立存在作为拓展指令集的 vec\_128 指令集和 vec\_256 指令集，用来解决大长度的算术计算。龙芯 32 位简化编译链的基础整型指令如表 3.1 所示。每条浮点指令和向量指令在形式上都存在对应的整型指令，并且功能上也一一对应，只是用来处理比整型数运算操作数长度更长的同类型运算。

表 3.1 龙芯 32 位简化编译链的基础整型指令

算术运算类指令	ADD.W, SUB.W, ADDI.W, ALSL.W, LU12I.W, SLT, SLTU, SLTI, SLTUI, PCADDI, PCADDU12I, PCALAU12I, AND, OR, NOR, XOR, ANDN, ORN, ANDI, ORI, XORI, MUL.W, MULH.W, MULH.WU, DIV.W, MOD.W, DIV.WU, MOD.WU
移位运算类指令	SLL.W, SRL.W, SRA.W, ROTR.W, SLLI.W, SRLI.W, SRAI.W, ROTR.W
位操作指令	EXT.W.B, EXT.W.H, CLO.W, CLZ.W, CTO.W, CTZ.W, BYTEPICK.W, REVB.2H, BITREV.4B, BITREV.W, BSTRINS.W, BSTRPICK.W, MASKEQZ, MASKNEZ
转移指令	BEQ, BNE, BLT, BGE, BLTU, BGEU, BEQZ, BNEZ, B, BL, JIRL
访存指令	LD.B, LD.H, LD.W, LD.BU, LD.HU, ST.B, ST.H, ST.W, PRELD
原子访存指令	LL.W, SC.W
栅障指令	DBAR, IBAR
其它杂项指令	SYSCALL, BREAK, RDTIMEL.W, RDTIMEH.W, CPUCFG

龙芯架构的基础部分包含非特权指令集和特权指令集两个部分，其中非特权指令集部分定义了常用的整数和浮点数指令，能够充分支持现有各主流编译系统生成高效的目标代码。龙芯架构的虚拟化扩展部分用于为操作系统虚拟化提供硬件加速以提升性能。这部分涉及的基本上都是特权资源，包括一些特权指令和控制状态寄存器，以及在例外和中断、存储管理等方面添加新的功能。龙芯架构的二进制翻译扩展部分用于提升跨指令系统二进制翻译在龙芯架构平台上的执行效率。其在基础部分之上进行扩展，同样包含非特权指令集和特权指令集两个部分。龙芯向量指令扩展和高级向量指令扩展两部分均是采用 SIMD 指令来加速计算密集型应用。两个扩展部分在指令功能上基本一致，区别在于向量指令扩展操作的向量位宽是 128 位而高级向量指令扩展操作的向量位宽是 256 位。

### 3.1.2 自定义指令选项

指令选项存在于 loongarch-opc.c 文件中，主要由四部分组成。第一部分是 match，是一条指令的唯一识别匹配的标志，它的工作方式类似于计算机网络的网号。第二部分是 mask，它在某些位上的值是 1。与网络掩码不同，它的值为 1 的位不一定是连续的，所以使用起来更加灵活。当它和当前指令的指令码执行相与的操作后，如果能够使结果和 match 的数值相同的话，就匹配成功。初始的龙芯指令集考虑到这一点，将 match 和 mask 的长度和数值设定为特定的值，不会出现两条代码冲突的情况。如果再加入新指令就有可能起冲突，因此需要和当前的指令集下的所有指令的掩码和匹配码进行比较。设计规则是如果掩码和与原先的指令的掩码相同，那么它们和指令的指令码相与之后的结果必须不同。反之情况也相同。同时还需要考虑指令的操作数数量的问题。这决定了到底最多有多少位能够被用作匹配码。



第三个部分是指令的名称，这个属性也具有特殊性。在生成指令的时候，匹配码匹配到相应的指令后，通过读取指令的名称和指令模板进行验证，判断一致后才能输出在汇编代码文件中。而且指令取名有其特殊的含义，比如说所有 128 位向量指令的指令名以字母“V”开头；所有 256 位向量指令的指令名以字母“XV”开头。除此之外，绝大多数指令通过指令名中“.XX”形式的后缀来指示指令的操作对象，且这种形式的后缀仅用来表征指令操作对象的类型。对于操作对象是整数类型的，指令名后缀为.B、.H、.W、.D、.BU、.HU、.WU、.DU 分别表示该指令操作的数据类型是有符号字节、有符号半字、有符号字、有符号双字、无符号字节、无符号半字、无符号字、无符号双字。

第四个部分是指令格式。它的意义在于表示指令在 32 位操作码的什么地方存放立即数或地址。比如“r0:5, r5:5, u16:6, u10:6”就具有四个操作数，分别在从低到高的 0-4 位、5-9 位上储存着两个长度为 5 的地址，在 10-15、16-21 位上存着两个六位的立即数。这个部分规定了指令的具体的格式，考虑匹配码和掩码的分配时，需要注意不能使匹配码长度过长触及操作数部分，使得匹配结果出现异常[21]。

其他部分还有的就是 macro, include, exclude, pinfo 选项，这些项分别对应了宏、内联指令、外联指令、指令信息的信息。这些选项在作为龙芯指令集参考的 risc-v 指令集上是有意义的，而龙芯指令集目前并没有把这些功能体现在 loongarch-opc.c 中。所以，在指令集这些项的值都为 0，没有起到区分作用。图 3.2 给出了一些指令的例子。

```
0x00800000, 0xffc00000, "bstrins.d", "r0:5,r5:5,u16:6,u10:6", 0, 0, 0, 0},
0x00c00000, 0xffc00000, "bstrpick.d", "r0:5,r5:5,u16:6,u10:6", 0, 0, 0, 0},
0x00008000, 0xffff8000, "cube.w", "r0:5,r5:5,r10:5", 0, 0, 0, 0},
```

图 3.2 指令的例子以及添加的指令

在这三个指令中，“bstrins.d”和“bstrpick.d”指令是原来的龙芯指令集中就存在的，而“cube.w”是在实验中添加的实验指令，它具有三个操作数，所以指令中数值部分占有的是低 15 位的位置。可以看出 cube.w 匹配码为上 17 位，值为 0x1。而从掩码长度可以看出，其他的两条指令的匹配码只有高 10 位，且与它们的掩码相与后不可能为 17 位的 0x1，所以可以区分开这几条指令。对应的方法也可运用在其他的指令上，必须所有的指令都符合与 cube.w 的匹配码相异的条件才能使得 cube.w 生效。除此之外，cube.w 的掩码没有干涉到数据位，这也使得本身的匹配不会出错。后续的实验也证明这条指令可以被编译器所识别，详细情况见第四章。

之前在实验中也设置过二操作数的 cube.w 指令，它的几个参数如下所示：

```
{0x0000A000, 0xFFFFFC00, "cube.w", "r0:5,r5:5", 0, 0, 0, 0}
```

在这种情况下匹配码的长度为 22 位，剩余 10 位正好给数据位。当时用这套指令掩码做三操作数的指令实验时，会出现反汇编失败的情况。这就是和指令的数值位发生冲突情况了。

### 3.1.3 修改 rtx 定义文件

rtx 定义文件一般指的是 rtl.def 文件。rtl.def 文件是用于描述寄存器模式的文件。它主要管理着 rtl 语言中的符号和指令格式的定义。rtl 语言是 GCC 内部表示的一种中间表示，用于描述指令级的操作和寄存器之间的转移。也就是说 rtl.def 是根据寄存器匹配制定转换的规则的文件。rtl.def 还定义了 rtl 模式中出现的各种操作和模式，以及模式的被选中的条件。模式包括了寄存器-寄存器操作、内存-寄存器操作等。rtl.def 文件中还包括了转换规则，描述了如何将一个 rtl 操作转换为另一个操作，以及如何进行优化和代码生成。如果需要添加新的指令就必须在里面添加对应的操作定义，例如图 3.3 就是在实验中新增加指令的相应定义。

```
458 /* Comparison, produces a condition code result. */
459 DEF_RTL_EXPR(COMPARE, "compare", "ee", RTX_BIN_ARITH)
460
461 /* plus */
462 DEF_RTL_EXPR(PLUS, "plus", "ee", RTX_COMM_ARITH)
463
464 DEF_RTL_EXPR(CUBE, "cube", "ee", RTX_COMM_ARITH)
465
466 /* Operand 0 minus operand 1. */
467 DEF_RTL_EXPR(MINUS, "minus", "ee", RTX_BIN_ARITH)
468
```

图 3.3 自定义的指令 cube.w 在 rtl.def 中添加的定义

在图 3.3 中，有四条关于指令运算的定义，分别是比较、加法、cube 运算、减法。括号中的几个参数，分别为指令运算的名称、指令运算引用时的名称、寄存器的数量与种类，优化时采用的逻辑类型。其中，编译器通过执行优化逻辑判断当前的指令的类型，以及如何在 rtl 级别执行优化。例如 cube.w 指令，它的大名为 CUBE，在 3.1.4 中的指令模板中被引用时就写为 cube。它的运算包含两个操作数，且为两个一般的寄存器。RTX\_COMM\_ARITH 表明 cube.w 暂定为算术运算指令，在 rtl 层进行优化时，为它实现算术的优化规则。如果需要优化，就需要进一步加入优化逻辑。但本文的优化主要集中在产生汇编代码时期的优化，这里就暂时留下了可以进行补充的地方。

### 3.1.4 机器描述文件

机器描述文件是一类描述汇编代码的生成逻辑和优化方案的文件。它的文件往往以“.md”结尾。里面存储着指令模板，这些模板用来决定最终 rtl 是如何生成汇编代码。



在这次实验中修改的就是 loongarch.md。

机器描述文件在 gcc 代码中的主要作用是：

1. 在使用 gimple 指令生成 rtx 指令时，机器描述文件可以通过相关定义将 gimple 指令翻译成对应的 rtx 指令。

2. 在对已经生成的 rtx 指令再次进行优化时，rtx 指令对应的机器描述文件以及输出的指令格式需要重新确定，解析函数的生成也要依赖机器描述文件，gen 函数就是在这种情况下生成的。

3. rtx 指令转化为汇编代码时，机器描述文件可以确定每个操作数的位置，以及汇编代码的输出格式。

在这次任务之中，需要关注的是第三点，这是由于 gimple 指令到 rtx 指令和 rtx 指令优化的过程是平台无关的，这表明无论移植到哪个架构的操作系统和 CPU 上，这些步骤是不变的。需要注意的是 rtx 指令到汇编代码的部分。这部分涉及 rtl 语言描述，详见下一节的讲解。

### 3.1.5 指令的匹配模式和窥孔优化

在本实验中，指令的匹配模式是在 loongarch.md 中实现的，实验中的寄存器调整和窥孔优化也是在这个文件中编写的。该文件的作用是构建帧栈布局和寄存器结构的部分定义和逻辑。编写这些定义使用的是 rtl 语言。

指令的匹配模式的功能通过定义 `define_insn` 来实现，一个匹配模式一共含有五个部分。第一个部分是指令的名称，指令的名字是可以任意选用的。不过如果指令有一个确切的名字，并且在 `rtl.def` 中被定义，在生成 rtl 代码时就可以执行对应的操作，否则就会被认为是无名的指令，不执行任何操作，如图 3.4 的 611 行就是一条加法指令的模板。然后是 rtl 匹配模板，它描述了指令模式的匹配规则，rtl 匹配模板由指令的操作符、操作数、匹配条件构成。对应的是图 3.4 的 612-614 行的内容。第三个是条件表达式，它一般起到判断当前指令模式是否符合的作用，也有判断寄存器关系的。如果为空，则条件总是为真，如图 3.4 的 615 行。下一个是汇编指令的输出模板，可见当前指令输出的汇编指令是 `fadd.w`、`fadd.d` 等 `fadd` 衍生的汇编指令。“%0”对应的是 rtl 模板标记的寄存器的编号数值。最后一个是指令属性。它是一个可选填的项，内容是和该模式匹配的指令的属性，这些属性值定义在指令属性模式中，主要是提示当前的指令应该执行哪种类型的优化。

```

611 (define_insn "add<mode>3"
612   [(set (match_operand:ANYF 0 "register_operand" "=f")
613         (plus:ANYF (match_operand:ANYF 1 "register_operand" "f")
614                   (match_operand:ANYF 2 "register_operand" "f")))]
615   "TARGET_HARD_FLOAT"
616   "fadd.<fmt>\t%0,%1,%2"
617   [(set_attr "type" "fadd")
618     (set_attr "mode" "<UNITMODE>")])

```

图 3.4 指令匹配模式的例子

本文中添加的 `cube.w` 指令也需要在这里添加相应的定义，如图 3.5 所示。其中指令的名称为 `cube`，尾缀为当前的模式名。指令有两个操作数，一个是普通寄存器类型的操作数，另一个是算术类型的操作数。`cube.w` 暂定的运算类型是加法，有两组寄存器匹配模式。指令的生成没有特定的要求，并且生成固定的 `cube.w` 指令，并使用和 `add` 相同的优化，和 `rtl.def` 文件里相对应。

添加的时候还为 `cube.w` 增加了寄存器匹配的组数。这也是在 `define_insn` 内实现的。具体修改的地方在图 3.4 的 621-623 行的 “`=r, r`”， “`r, r`”， “`r, I`”。原本的代码是 “`=r`”， “`r`”， “`r`”。先介绍这个属性有关寄存器的匹配，`r` 代表的是匹配寄存器、`f` 匹配的是浮点数、`I` 匹配的是立即数。每一个竖行为一个匹配模式。所以原来的匹配方案只有三个操作数为 “`r, r, r`” 才能匹配，否则交给匹配失败的函数再进行处理，现在给它增加了 “`r, r, I`” 的匹配方案。这样在寄存器和立即数进行加法运算时会快一些。

```

620 (define_insn "cube<mode>3"
621   [(set (match_operand:GPR 0 "register_operand" "=r,r")
622         (plus:GPR (match_operand:GPR 1 "register_operand" "r,r")
623                   (match_operand:GPR 2 "arith_operand" "r,I")))]
624   ""
625   "cube.w\t%0,%1,%2";
626   [(set_attr "alu_type" "add")
627     (set_attr "mode" "<MODE>")])

```

图 3.5 自定义指令 `cube.w` 在 `loongarch.md` 中的定义

窥孔优化是一种对已经生成代码进行再优化的方案。窥孔优化的操作类似于一个滑动窗口，它只会对范围中指令进行分析和优化，这个窗口的大小是可变的。编译器会对产生的汇编代码进行线性扫描，尝试删除无用代码、优化循环操作和简化指令。窥孔优化具有使用起来比较快速和占用内存小的优点 [22]。初期的编译器一般都是使用 `define_peephole` 直接匹配指令和替换汇编文本。但是现在大都使用 `define_peephole2` 进行优化，它在寄存器分配之前就进行优化，优化出的代码效率更高。图 3.6 是 `peephole2` 的例子，现在的龙芯编译器已经不存在使用 `peephole` 的情况了。现在的存在 `peephole2`

都是对读内存的操作进行优化的，几乎涵盖了所有情况。

peephole2 主要的部分只有三个。第一个部分在图 3.6 的 3841-3844 行，是需要检测到的指令模板。第二个部分位于图 3.6 的 3845 行，是使用优化的限制条件。如果同时满足匹配规则和条件，窥孔优化会把匹配的语句转化为类似图 3.6 的 3846-3849 行的样形式。peephole 也类似 peephole2 的写法，只不过第三部分直接给出了转化后的代码。图 3.6 中代码的作用是将三条连续存储的指令转化为一条存储指令，主要用于数组运算的优化。

```

3840 (define_peephole2
3841   [(set (match_operand:SI 0 "register_operand")
3842         (any_extend:SI (match_operand:HI 1 "non_volatile_mem_operand"))
3843       (set (match_operand:SI 2 "register_operand")
3844             (any_extend:SI (match_operand:HI 3 "non_volatile_mem_operand"))))]
3845   "loongarch_load_store_bonding_p (operands, HImode, true)"
3846   [(parallel [(set (match_dup 0)
3847                   (any_extend:SI (match_dup 1)))
3848               (set (match_dup 2)
3849                   (any_extend:SI (match_dup 3))))]])
3850   "")
3851
3852

```

图 3.6 peephole2 优化的例子

以此为基础，本人在文件中加入了一条简单的 peephole 优化语句，将默认和优化规则明确地写了出来，在某些特定的情况下可以生效，如图 3.7 所示。它的意图是在出现寄存器乘以常数 0 时，不进行计算，直接将 0 这个数值通过 move 指令移入目的寄存器内。这样可以稍稍节省时间，不需要去进行乘 0 的无用运算了。

```

3821 (define_peephole
3822   [(set (match_operand:SI 0 "register_operand" "=r")
3823         (mult:SI (match_operand:SI 1 "register_operand" "r")
3824                 (const_int 0)))]
3825   ""
3826   ""
3827   {
3828     output_asm_insn ("move.w %0,0", operands);
3829     return "";
3830   })

```

图 3.7 新建的 peephole 优化

## 3.2 向量指令优化

### 3.2.1 明确编译选项

在这次实验使用到的和向量指令相关的编译选项主要有四个。一是“-march=”。它的作用是指明需要生成的系统。在 target 目录下存在着数十个目标机器的指令生成规则的限制文件。在使用龙芯编译环境的情况下还存在着一些分支选项，如 la464, lsx、lsx2

等选项，分别对应了龙芯的不同架构的生成环境。这次需要移植的操作系统使用的编译环境是 loongarch32r。但是实际上使用的龙芯简化编译器将几个选项进行了融合，使得最终选择操作系统的目标机器的选项只需要确保使用的目标机器是龙芯 CPU 就行了，不需要精确区分这些细则。但是这个定义在编译选项是需要明确指定的，否则会默认使用“-march=loongarch”情况下的汇编代码结构。这种情况下默认只能生成 la32r 指令集的 test 和 fix 指令集的指令。这里如果使用选项 lsx 和 lsx2 可以直接动用 128 位向量汇编指令。如果使用选项 lasx 就能直接使用 256 位向量汇编指令。

第二个是 -O 选项，这个选项是在选择编译器的优化程度。根据优化的程度可以分成 -O0, -O1, -O2, -O3 四种选项。-O0 选项代表不用任何重组汇编代码方面的优化，但是可以在匹配方案层面上进行优化。-O1 是单个代码之间的调整和合并优化，最主要的就是 peephole 和 peephole2 的优化方案。-O2 是对循环类型的代码进行优化，-O3 执行的是排列和删除代码块的优化。有的龙芯编译器还支持 -Os 类型的更进一步的优化，但是龙芯 32 位简化编译器没有这种功能，就暂且不予考虑。高等级的优化策略会在进行汇编代码检测、重组上花费大量时间，但是会使得生成的目标程序运行速度变得很快。考虑到这一点，就需要权衡两者的利弊。本文提到的三种优化中，寄存器优化任何时候都发挥作用，窥孔优化处于 -O1 阶段，向量优化处于 -O3 阶段。由于在实验中目前能够观察到的优化方案都在 -O1 和 -O3 等级中，所以实验中一般使用的选项就是这两个。

第三，四个是“-msimd=lsx”和“-mfpu=64”，这两个选项是配套使用的。“msimd=lsx”表明当前计算机使用的是单指令多数据流的结构，可以支持四字节的并行向量计算。

“-mfpu=64”代表能够启动多精度的浮点运算功能，这样就可以使用 128 位的向量指令进行运算。在使用长度为 4 字节以上的数组或数字运算时，汇编代码中就会生成相关代码。同理，“-msimd=lasx”和“-mfpu=64”组合时，在使用长度为 4 字节以上、8 字节以下的数组或数字运算时，会产生 128 位的向量指令。在使用长度为 8 字节以上的数组或数字运算时，会产生 256 位的向量指令。

### 3.2.2 使向量指令默认开启

除了上文中使用四个编译选项能够使得编译过程中能够产生向量指令，还可以通过改变选项文件中的一些定义使得指令选项默认打开。比如说在 loongarch-opc.h 文件中，有两个定义地点。如图 3.8 所示，下方是一个名为 loongarch\_ASEs\_option LARCH\_opts 结构体，里面含有指令集和其他选项的一些定义。以 .ase 开头的几个选项表示几个指令集默认是否打开。现在所有的指令都没有默认打开，但是在初始化的时候会使用 test 和 fix

两个指令集。下方的指令是用来设定地址字长、指令长度、立即数长度的，和这次实验没有直接关系，保持默认值就行。使用浮点运算时需要将.ase\_float 置为 1，使用 128 位向量指令需要将.ase\_128vec 置为 1，使用 256 位向量指令需要将.ase\_256vec 置为 1。ase256 的功能包含.ase128 的功能。

除此之外，还有一处需要一同修改，如图 3.9。这是细分过的指令集定义，多出来的指令集是由 fix 指令集细分而来。这里依旧是需要默认使用哪个指令集，就需要将该指令集的最后一个参数设置为 1。这样就能够在不添加相关参数时默认使用该指令集了。但是不推荐在工作中改变默认值，由于开发板不同，可能有些开发板不支持向量运算，生成会导致出错。只使用默认的指令可能会造成效率不高，但绝对是安全的。

```

4 struct loongarch_ASEs_option LARCH_opts =
5 {
6     .ase_test = 0,
7     .ase_fix = 0,
8     .ase_float = 0,
9     .ase_128vec = 0,
10    .ase_256vec = 0,
11
12    .addrwidth_is_32 = 0,
13    .addrwidth_is_64 = 0,
14    .rlen_is_32 = 0,
15    .rlen_is_64 = 0,
16    .la_local_with_abs = 0,
17    .la_global_with_pcrel = 0,
18    .la_global_with_abs = 0,
19
20    .abi_is_lp32 = 0,
21    .abi_is_lp64 = 0,
22 };

```

图 3.8 决定指令集是否默认启用的选项

```

2211 struct loongarch_ase loongarch_ASEs[] = {
2212 {&LARCH_opts.ase_test, loongarch_test_opcodes, 0, 0, {0}, 0, 0},
2213 {&LARCH_opts.ase_fix, loongarch_macro_opcodes, 0, 0, {0}, 0, 0},
2214 {&LARCH_opts.ase_fix, loongarch_lmm_opcodes, 0, 0, {0}, 0, 0},
2215 {&LARCH_opts.ase_fix, loongarch_privilege_opcodes, 0, 0, {0}, 0, 0},
2216 {&LARCH_opts.ase_fix, loongarch_jmp_opcodes, 0, 0, {0}, 0, 0},
2217 {&LARCH_opts.ase_fix, loongarch_load_store_opcodes, 0, 0, {0}, 0, 0},
2218 {&LARCH_opts.ase_fix, loongarch_fix_opcodes, 0, 0, {0}, 0, 0},
2219 {&LARCH_opts.ase_float, loongarch_4opt_opcodes, 0, 0, {0}, 0, 0},
2220 {&LARCH_opts.ase_float, loongarch_float_opcodes, 0, 0, {0}, 0, 0},
2221 {&LARCH_opts.ase_128vec, loongarch_128vec_opcodes, 0, 0, {0}, 0, 0},
2222 {&LARCH_opts.ase_256vec, loongarch_256vec_opcodes, 0, 0, {0}, 0, 0},
2223
2224 {0},
2225 };

```

图 3.9 另外的指令集选项

### 3.3 上板移植前后的准备

#### 3.3.1 安装龙芯 Qemu

由于每次对编译器进行修改都需要重新编译安装，所以将编译器的安装放入第四章的准备工作里讲解，在这里只介绍龙芯 Qemu 的安装过程。



在从龙芯官网上下载了龙芯 Qemu 源码后，首先需要在主目录下建立存放编译安装文件的目录 build，然后在 build 目录下执行指令“`../configure --target-list=loongarch32-softmmu --disable-werror --enable-debug`”。这是一条配置指令，目标机器是面向龙芯架构，禁用了严重错误的报告，启用了 debug 的功能。如果有其他的需要，可以对里面的参数进行修改。当不禁用严重错误报告时，就不能正常编译，判断可能是设置了一些自陷指令的原因。配置完之后，在当前目录下执行 make 指令，完成对 Qemu 的安装。接下来再执行“`make install`”指令将其装入系统中去。也可以直接将 build 文件夹加入系统路径下，这样 mos 就可以直接通过“`make run`”指令调用龙芯 Qemu 进行仿真。

还需要注意的是，龙芯 Qemu 当前没有设置 loongarch64 的用户选项，但是为了之后进行添加拓展，官方在编译的时候会检测 loongarch64 的用户文件夹的存在。因此，在编译之前还需要在 linux-user 文件夹之下将 loongarch32 的文件夹复制，并改名为 loongarch64，这样会执行同一套用户操作逻辑，不会出现错误。各个文件应该如图 3.10 中所示。

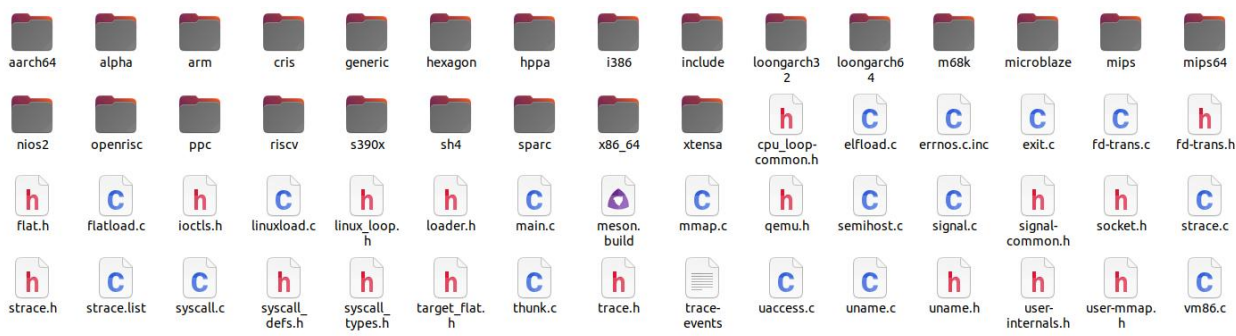


图 3.10 linux-user 文件夹下应该存在的文件

### 3.3.2 修改和运行 mos

操作系统 mos 由系统定义文件、用户应用文件所组成。在移植的时候最需要注意的就是 include、init 文件夹。除此之外，还需要注意三个后缀为“.lds”的文件，它们也作为宏文件控制着操作系统的基础部分。由于 mos 不能引用 gcc 库里面的头文件对操作系统的用户指令进行编译，include 文件夹储存着所有关于启动 mos 的参数和函数以及 gcc 中的部分头文件。

init 文件夹下有 cache.c、start.S 两个文件，cache.c 规定了龙芯操作系统的缓存的存储方式。由于 mos 和编译链都属于龙芯体系，不需要修改。start.S 直接由汇编代码组成，作用是构建上板所使用的内核和所需要的文件系统，在这个文件中的许多数值都需要改变来使得 mos 能够兼容龙芯系统。start.S 文件内部如图 3.11 所示。在这个文件中 19-33 行是用来启动操作系统和文件系统的。20-23 行的代码用来使得几个标志寄存器的

值复位。需要修改的是 25、27 行和 31 行的内容。25 行的加载的数字是用来表示未包含 cach 的操作系统的地址的，具体的地址需要根据具体开发板的参数决定。比如龙芯 Qemu 规定的数值 0xa0000019，而开发板上规定的数值是 0x80000009，这与当前环境的寄存器数量有关系。27 行是包含了 cach 之后的操作系统的地址。在开发板中里面 cach 的大小为 32MB。所以一般来说，27 行的数值比 25 行大 0x20000000。31 行的内容要与前两项相对应，这行规定了整个系统最低地址是 0x80000000，两类系统最终都不能越过这个界限运行。

```

6_start:
7      /* initialize */
8      #if !defined(LAB) || LAB >= 3
9          la      a0, exc_gen_entry
10         csrwr   a0, csr_eentry
11     #endif
12     #if !defined(LAB) || LAB >= 2
13         la      a1, tlb_miss_entry
14         li.w    a0, 0xffffffff
15         and     a1, a1, a0
16         csrwr   a1, csr_tlbrentry
17     #endif
18
19     /* enable mem map and jump to high address */
20     li.w    a0, 0xb0      // PLV = 0, ID = 0, DA = 0, PG = 1, DATF = DATM = 01
21     csrwr   a0, csr_crmd
22     li.w    a0, 0x0c000000
23     csrwr   a0, csr_tlbidx
24
25     li.w    t1, 0xa0000009 // uncached window
26     csrwr   t1, csr_dmw0
27     li.w    t1, 0x80000009 // cached window
28     csrwr   t1, csr_dmw1
29
30     la      a0, 1f
31     li.w    t0, 0x80000000
32     or      a0, a0, t0
33     jr      a0
34
35 1:
36     li.w    t0, ESTAT_TI
37     csrwr   t0, csr_ectl
38     /* hint: you can refer to the memory layout in include/mmu.h */
39     /* set up the kernel stack */
40
41     li.w    sp, KSTACKTOP
42
43     /* clear bss segment for C env. */
44     la      t0, .bss
45     la      t1, bss_end

```

图 3.11 文件 start.S 的内容

除此之外，还需要规定用于交互的串口设备 MMIO 地址，这需要根据 Qemu 和开发板具体参数进行设置。比如，在使用的时候，QEMU 版本的串口设备 MMIO 地址位于 0x1fe001e0 处，而上板版本的串口设备 MMIO 地址则位于 0x1fe40000 处。这需要修改 megasoc.h 中关于对这个 MMIO 地址的定义选项 MEGASOC\_SERIAL\_BASE。在模拟和在开发板上操作之前要提前切换数值。两者转化的逻辑如图 3.12 所示。下面的几个定义的数值也需要根据编译环境来设置。MEGASOC\_SERIAL\_DATA 定义了用于传输数据的寄存器地址，而 MEGASOC\_SERIAL\_LSR 定义了用于获取线路状态的寄存器地址。MEGASOC\_SERIAL\_DATA\_READY 表示接收到的数据已准备好读取，而 MEGASOC\_SERIAL\_THR\_EMPTY 表示传输寄存器为空，可以写入新的数据。

MEGASOC\_FPGA\_HALT\_ADD 定义了 FPGA 设备中用于控制重新启动的地址，而 MEGASOC\_FPGA\_HALT\_VAL 则定义了重新启动的值。在查看了龙芯编译器的帧栈说明书后，根据上面推荐的设置数值填写了这些定义。

还需要考虑的是不同机器和操作系统之间的编码打印问题，mos 操作系统原本构造的是类 linux 的编译环境，将“\n”视为一行结束的标志，在龙芯的 Qemu 上也是这样规定的，所以不会出现乱码。但是龙芯开发板采用的是类似 windows 的打印逻辑，使用“\r\n”代表一行的结束，这样就会导致发生输出的错误。需要在输出“\n”的情况下在之前输出“\r”进行组合，才能够发挥功能。这个功能在 kern\machine.c 里，需要修改其中的函数实现这个功能。修改如图 3.13 所示，添加的是第十五行的语句。

```

1 #ifndef MEGASOC_H
2 #define MEGASOC_H
3
4 /*
5  * QEMU MMIO address definitions.
6  */
7
8 /*
9  * 16550 Serial UART device definitions.
10 */
11 #define MEGASOC_SERIAL_BASE (0x1fe001e0)
12 // #define MEGASOC_SERIAL_BASE (0x1fe40000)
13
14 #define MEGASOC_SERIAL_DATA (MEGASOC_SERIAL_BASE + 0x0)
15 #define MEGASOC_SERIAL_LSR (MEGASOC_SERIAL_BASE + 0x5)
16 #define MEGASOC_SERIAL_DATA_READY 0x1
17 #define MEGASOC_SERIAL_THR_EMPTY 0x20
18
19 /*
20  * MEGASOC reboot on QEMU.
21 */
22 #define MEGASOC_FPGA_HALT_ADD (0x1fe78030)
23 #define MEGASOC_FPGA_HALT_VAL (42)
24
25 #endif

```

图 3.12 megasoc.h 的内部内容

```

11 */
12 void printcharc(char ch) {
13     while (!((volatile uint8_t *) (KSEG1 + MEGASOC_SERIAL_LSR)) & MEGASOC_SERIAL_THR_EMPTY)) {
14     }
15     if (ch=='\n') printcharc('\r');
16     *((volatile uint8_t *) (KSEG1 + MEGASOC_SERIAL_DATA)) = ch;
17 }

```

图 3.13 修正打印乱码的错误

由于在 3.2.1 中说到编译器并不能识别龙芯下的具体的目标机器，如果使用原来的 loongarch32r 作为选择机器的标识，就会导致编译器不认识这个机器名称。这个选择目标机器的代码位置在 mos 的所有“.lds”文件内，需要把相应所有选项都修改为“loongarch”。还需要将程序的入口设定为\_start，需要和 start.S 里面的启动函数的名称相同。具体如



图 3.14 所示。然后把启动地址修改为开发板上或 Qemu 的标准地址。除此之外，还可以设置选用的应用程序和 mos 的操作系统模式。这些不在系统文件中修改，而是在命令行参数中体现，这些请参考 3.2.1。

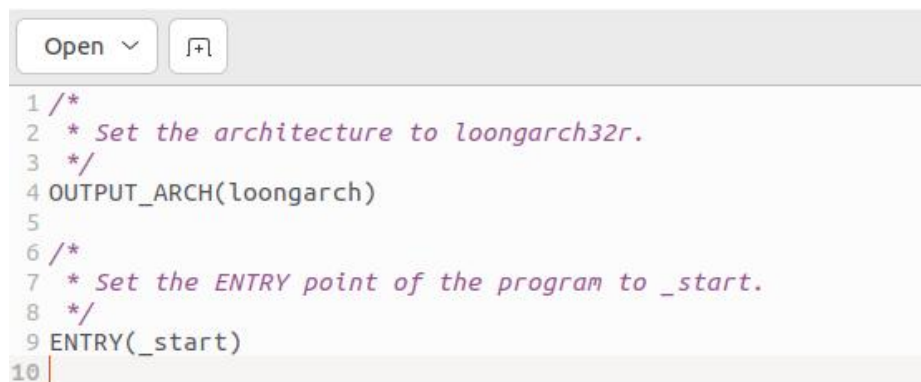


图 3.14 .lds 文件需要做的修改

模拟运行首先需要在 mos 的主目录下使用指令“make test lab=6\_2”构建整个操作系统，在 target 文件夹下生成 fs.img 和 mos。如果是在 Qemu 内进行模拟，就可以直接使用“INITRD\_OFFSET=3800000 Qemu-system-loongarch32 -M ls3a5k32 -m 64M -nographic -no-reboot -kernel target/mos -initrd target/fs.img”指令启动 mos 系统。在这条指令下，模拟了 SD 卡在开发板上的情况。将操作系统安装在虚拟 SD 卡起始位置偏移量为 3800000 的位置，设置了 ls3a5k32 类型的龙芯虚拟编译环境，不使用图形化界面，并且指定了 mos 为内核，fs.img 为文件系统。在开发板的运行 mos 是验证移植工作的成功与否的关键，需要串口助手、vivado 等软件的辅助，这些会在之后的第四章讲解。

## 第四章 实验结果的验证

### 4.1 验证的前置准备

如果要进行验证，首先需要对改变后的编译器进行重新构建。在进行了上章的修改后，编译器的改变主要有两点，一是在相关文件之中加入了 `cube.w` 指令的定义、二是打开了浮点运算和向量运算的选项。首先需要确认是否安装 `bison`、`flex`、`gawk` 和 `ninja` 等支持固件包，确保不会因为不能找到支持文件而构建失败。还需要在 `gcc` 文件夹底下更新适合当前版本的 `isl`、`gmp`、`mpc`、`mpfr` 文件，使构建时的逻辑符合标准。在使用 `build.sh` 在对 `bunitils` 和 `gcc-8.3.0` 进行构建后，在 `install/bin` 文件夹下会出现和 `GCC` 相关的可执行程序，如 `loongarch32r-linux-gnusrf-gcc`、`loongarch32r-linux-gnusrf-as` 等。在产生可执行程序的步骤完成后，还需要将 `bin` 文件夹加入系统路径。完成后就可以进行下一步的操作。如图 4.1 所示的这些就是生成的可执行程序。这里在 `build.sh` 在构造编译器前需要在文件中把目标机器的定义改为 `loongarch32r`，如图 4.2 所示，否则编译器可能会使用其他目标机器的指令生成逻辑。图 4.3 展示的是编译成功结束的标志，不会出现 `Error`。

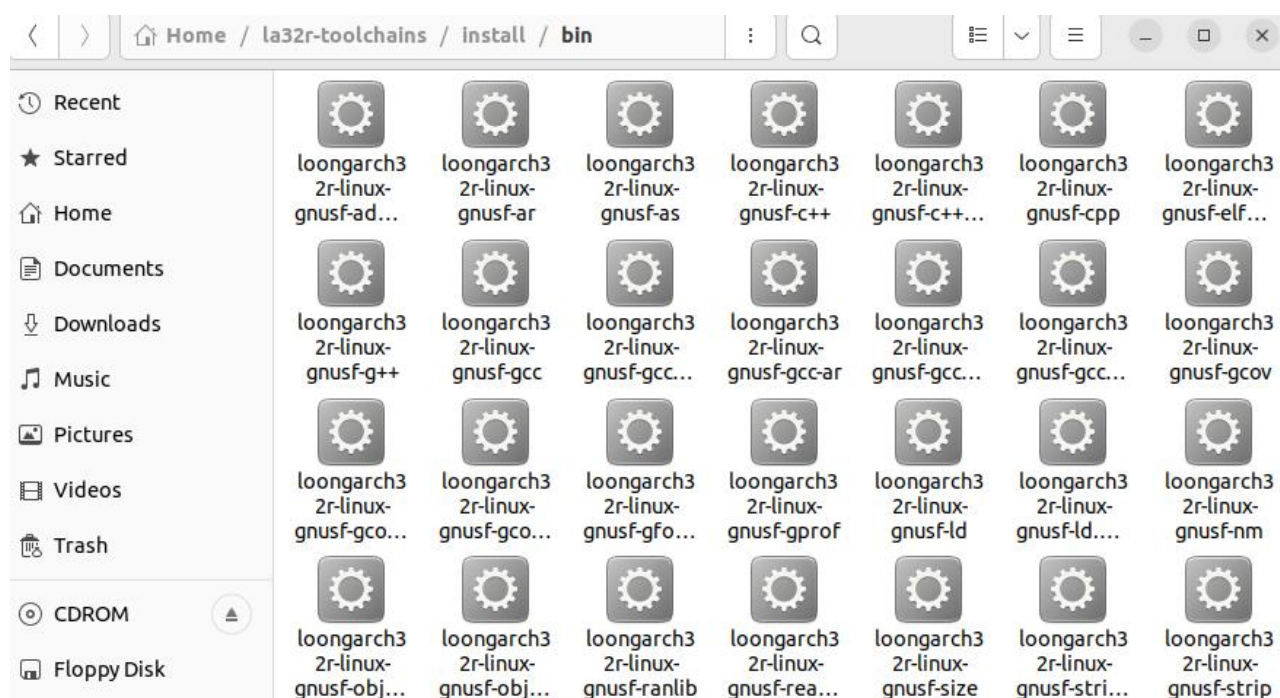


图 4.1 编译后的龙芯二进制程序

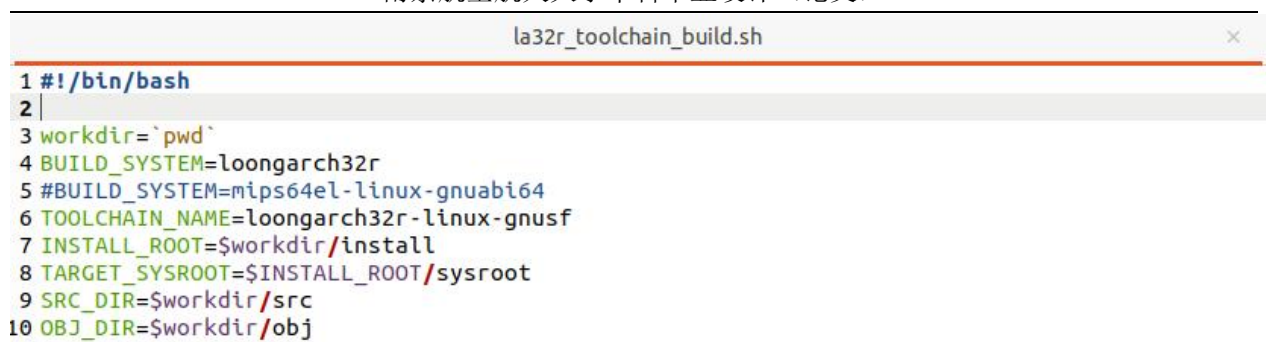


图 4.2 改变后的目标机器设置

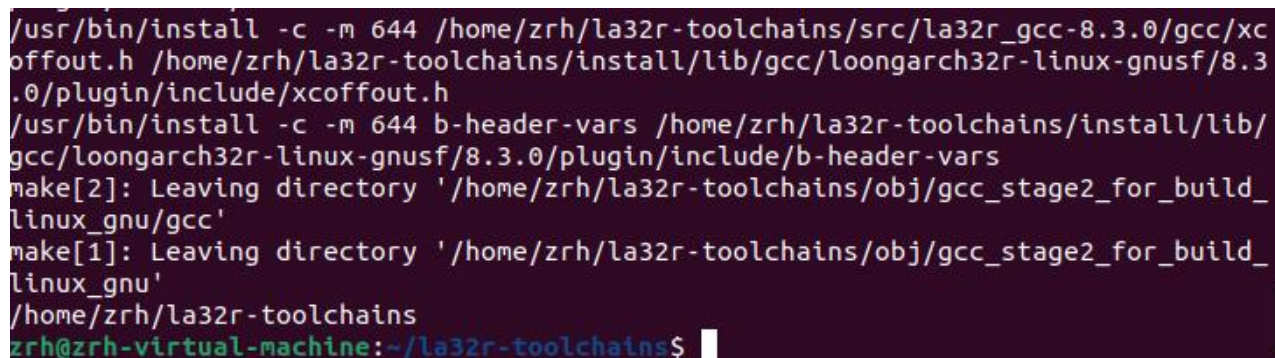


图 4.3 编译成功的图像

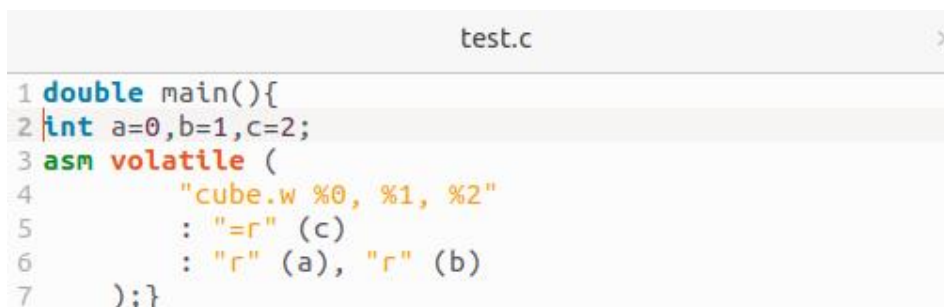
最后需要修改 mos 里的编译选项文件使得 mos 可以使用编译器的所有代码，起到优化的作用。具体的做法是在 mos 的 include.mk 里的 CFLAG 的选项之中，加入 CFLAGS 编译选项“-O3 -msimd=lsx -mfpu=64”。这几个选项之前也有提到，功能是允许 mos 使用所有扩展指令集。这样就可以排除 mos 不能使用除向量指令以外的其他来自操作系统的问题。

## 4.2 生成新指令部分的验证

这项任务的完成指标有两点。一是加入的指令能够被翻译成汇编代码。二是在生成可执行程序后，可以通过编译器产生的反汇编程序将可执行程序。

### 4.2.1 在汇编文件内产生自定义指令

在实验中使用的是如图 4.4 所示的文件，里面含有生成 cube.w 的指令和 128 位的向量运算。使用的指令是“loongarch32r-linux-gnusr-gcc -S -O3 -msimd=lsx -mfpu=64 test.c”。实验的文件中只定义了三个整型数和一条内联汇编指令。其意义是在将 c、a、b 三个参数分别代入 cube.w 的三个寄存器参数中进行运算。由于 cube.w 指令的用途不明确，所以编译时-O3 没有将其简化掉，如果用其他原生指令进行测试，则必须把-O3 选项去掉。否则指令会被直接优化掉。

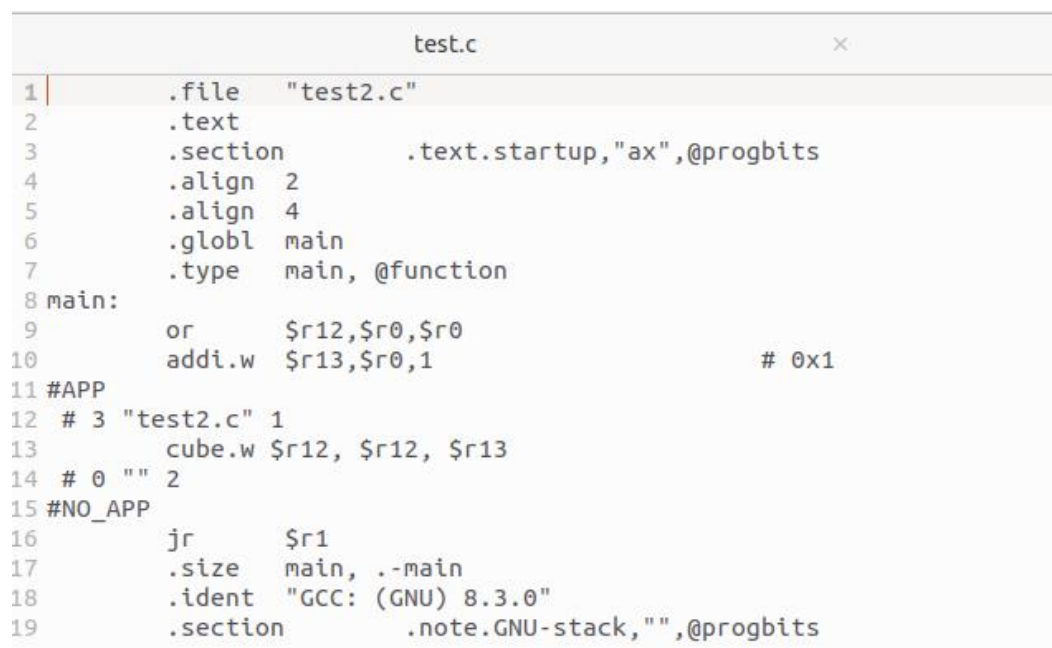


```

1 double main(){
2 int a=0,b=1,c=2;
3 asm volatile (
4     "cube.w %0, %1, %2"
5     : "=r" (c)
6     : "r" (a), "r" (b)
7 );}

```

图 4.4 验证新生成指令的测试文件



```

1 .file "test2.c"
2 .text
3 .section .text.startup,"ax",@progbits
4 .align 2
5 .align 4
6 .globl main
7 .type main, @function
8 main:
9 or $r12,$r0,$r0
10 addi.w $r13,$r0,1 # 0x1
11 #APP
12 # 3 "test2.c" 1
13 cube.w $r12, $r12, $r13
14 # 0 "" 2
15 #NO_APP
16 jr $r1
17 .size main, .-main
18 .ident "GCC: (GNU) 8.3.0"
19 .section .note.GNU-stack,"",@progbits

```

图 4.5 测试文件产生的汇编代码

汇编代码的生成如图 4.5 所示。在汇编代码的 13 行会出现本文自定义的 `cube.w` 指令。这是本人在 C 语言程序当中使用内联汇编的方法生成的。这种方式可以直接指定某一条汇编指令的产生。虽然不是通过直接的方式生成的，但是如果不符合格式，编译器就会输出未找到这条指令的错误，这表明编译器有能够识别指令结构的能力，从侧面表明了编译器能够识别 `cube.w`。

#### 4.2.2 进行新指令的反汇编测试

反汇编是先通过“`loongarch32r-linux-gnuf-as test.s`”把上文生成的汇编文件编译成龙芯架构下的二进制程序 `a.out`。再使用指令“`loongarch32r-linux-gnuf-objdump -d a.out`”将 `a.out` 反汇编为汇编程序。图 4.6 是反汇编后的结果，可以看到上文提及的 `cube.w` 存在于反汇编文件中，说明当前的编译器能够识别这些指令，添加指令和使用向量指令的工作是成功的。这是由于编译器的主要工作就是将高级语言转化为可执行程序 and 反汇编，在这两种工作下指令都能正常产生，说明编译器的优化和产生新指令的成功。编译器的链



接等其他功能在这里不会影响结果，因此不需要考虑。

```

zrh@zrh-virtual-machine:~/test$ loongarch32r-linux-gnustf-objdump -d a.out

a.out:      file format elf32-loongarch

Disassembly of section .text:

00010390 <main>:
10390:      0015000c      move     $r12,$r0
10394:      0280040d      addi.w   $r13,$r0,1(0x1)
10398:      0000b58c      cube.w   $r12,$r12,$r13
1039c:      4c000020      jirl     $r0,$r1,0

```

图 4.6 反汇编的结果

### 4.3 指令优化部分验证

#### 4.3.1 在汇编文件中生成浮点和向量指令

在实验中使用的是如图 4.7 所示的文件，里面含有能够产生 128 位的向量运算。使用的指令是“loongarch32r-linux-gnustf-gcc -S -O3 -msimd=lsx -mfpu=64 test.c”。其中 `__vector_quad` 构造了一个四个数字的数组。设定了一个虚拟的函数 `loongarch_lsx_v4sf`，使用它可以将四个数组的数当作一个整体进行运算，此时这个数字的长度达到了 128 位。这时结合编译选项中添加生成向量指令的参数，这时如果实验是成功的，就会出现 `vld`、`vst` 两类指令，这是由于汇编指令在引用函数的时候，如果函数体本身不明确且函数传递了参数，就会通过产生 `st` 和 `ld` 操作示意将参数传递过去。当然，在这次实验中使用的是向量指令来实现这个传递 128 位整型参数的功能。

```

test.c
1 #include <stdio.h>
2 typedef struct {
3     float data[4];
4 } __vector_quad;
5 __vector_quad loongarch_lsx_v4sf(__vector_quad v);
6 int main() {
7     __vector_quad vx = {{1.0f, 2.0f, 3.0f, 4.0f}};
8     __vector_quad vy = loongarch_lsx_v4sf(vx);
9     printf("LSX Result: %.2f, %.2f, %.2f, %.2f\n", vy.data[0],
10     vy.data[1], vy.data[2], vy.data[3]);
11     return 0;
12 }

```

图 4.7 指令优化的测试程序

图 4.8 展示的是实验中得到的结果。可以看到在汇编代码文件的第 15 和 16 行出现了 `vld` 指令和 `vst` 指令，这两条指令都是在 128 位向量指令集出现的存取指令。除此之外，汇编代码中还混有许多以 `f` 开头的浮点指令，如 `fld.s`、`fct.d` 等，这也说明了添加了向量指令的选项后，浮点指令也是可以生成的，提高了编码程序的整体效率。

```

1 | .file "test.c"
2 | .text
3 | .section .rodata.str1.4,"aMS",@progbits,1
4 | .align 2
5 | .LC2:
6 | .ascii "LSX Result: %.2f, %.2f, %.2f, %.2f\012\0"
7 | .section .text.startup,"ax",@progbits
8 | .align 2
9 | .align 4
10 | .globl main
11 | .type main, @function
12 main:
13 | addi.w $r3,$r3,-80
14 | la.local $r12,.LC1
15 | vld $vr0,$r12,0
16 | vst $vr0,$r3,32
17 | ld.w $r15,$r3,32
18 | ld.w $r14,$r3,36
19 | ld.w $r13,$r3,40
20 | ld.w $r12,$r3,44
21 | addi.w $r5,$r3,16
22 | addi.w $r4,$r3,48
23 | st.w $r1,$r3,76
24 | st.w $r15,$r3,16
25 | st.w $r14,$r3,20
26 | st.w $r13,$r3,24
27 | st.w $r12,$r3,28
28 | bl %plt(loongarch_lsx_v4sf)
29 | fld.s $f0,$r3,60
30 | fld.s $f3,$r3,56
31 | fld.s $f2,$r3,52
32 | fld.s $f1,$r3,48
33 | la.local $r4,.LC2
34 | fcvt.d.s $f0,$f0
35 | fst.d $f0,$r3,0
36 | fcvt.d.s $f0,$f2
37 | fcvt.d.s $f3,$f3
38 | movfr2gr.s $r8,$f0
39 | movfrh2gr.s $r9,$f0
40 | fcvt.d.s $f0,$f1
41 | movfr2gr.s $r10,$f3
42 | movfrh2gr.s $r11,$f3
43 | movfr2gr.s $r6,$f0
44 | movfrh2gr.s $r7,$f0
45 | bl %plt(sprintf)
46 | ld.w $r1,$r3,76
47 | or $r4,$r0,$r0
48 | addi.w $r3,$r3,80
49 | jr $r1
50 | .size main, .-main
51 | .section .rodata.cst16,"aM",@progbits,
52 | .align 4
53 | .LC1:
54 | .word 1065353216
55 | .word 1073741824
56 | .word 1077936128
57 | .word 1082130432
58 | .ident "GCC: (GNU) 8.3.0"
59 | .section .note.GNU-stack,"",@progbits

```

图 4.8 测试文件产生的汇编代码

### 4.3.2 和不使用浮点和向量指令的汇编代码进行对比

在这个对比实验中使用的是和 4.2.1 中相同的测试文件，里面含有能够产生 128 位的向量运算。使用的指令是“loongarch32r-linux-gnusr-gcc -S test.c”。这样虽然文件中还是会有 128 位的运算，但是龙芯编译器在不使用拓展指令集的情况下，只能生成 32 位的存取指令。结果如图 4.9 所示。可以看出在不使用拓展指令集的情况下，确实只会生成 fix 指令集的常规指令，其数量达到了 90 条，而使用向量指令之后就会减少为 55 条。在实验理论的章节中讲到过向量指令属于单指令多数据流的指令。执行一条向量指令所花费的时间和一条常规指令几乎相同。所以使用向量指令可以使得编译器提速 39%，优化得到了体现。这样优化的部分也可以认为是完成了。

```

1 | .file "test.c"
2 | .text
3 | .globl __extendsfdf2
4 | .section .rodata
5 | .align 2
6 | .LC1:
7 | .ascii "LSX Result: %.2f, %.2f, %.2f, %.2f\012\0"
8 | .align 2
9 | .LC0:
10 | .word 1065353216
11 | .word 1073741824
12 | .word 1077936128
13 | .word 1082130432
14 | .text
15 | .align 2
16 | .globl main
17 | .type main, @function
18 main:
19 | addi.w $r3,$r3,-96
20 | st.w $r1,$r3,92
21 | st.w $r22,$r3,88
22 | st.w $r24,$r3,84
23 | st.w $r25,$r3,80
24 | st.w $r26,$r3,76
25 | st.w $r27,$r3,72
26 | st.w $r28,$r3,68
27 | st.w $r29,$r3,64
28 | addi.w $r22,$r3,96
29 | la.local $r12,.LC0
30 | ld.w $r15,$r12,0
31 | ld.w $r14,$r12,4
32 | ld.w $r13,$r12,8
33 | ld.w $r12,$r12,12
34 | st.w $r15,$r22,-48
35 | st.w $r14,$r22,-44
36 | st.w $r13,$r22,-40
37 | st.w $r12,$r22,-36
38 | addi.w $r16,$r22,-64
39 | ld.w $r15,$r22,-48
40 | ld.w $r14,$r22,-44
41 | ld.w $r13,$r22,-40
42 | ld.w $r12,$r22,-36
43 | st.w $r15,$r22,-80
44 | st.w $r14,$r22,-76
45 | st.w $r13,$r22,-72
46 | st.w $r12,$r22,-68
47 | addi.w $r12,$r22,-80
48 | or $r5,$r12,$r0
49 | or $r4,$r16,$r0
50 | bl %plt(loongarch_lsx_v4sf)
51 | ld.w $r12,$r22,-64
52 | or $r4,$r12,$r0
53 | bl %plt(__extendsfdf2)
54 | or $r24,$r4,$r0
55 | or $r25,$r5,$r0
56 | ld.w $r12,$r22,-60
57 | or $r4,$r12,$r0
58 | bl %plt(__extendsfdf2)
59 | or $r26,$r4,$r0
60 | or $r27,$r5,$r0
61 | ld.w $r12,$r22,-56
62 | or $r4,$r12,$r0
63 | bl %plt(__extendsfdf2)
64 | or $r28,$r4,$r0
65 | or $r29,$r5,$r0
66 | ld.w $r12,$r22,-52
67 | or $r4,$r12,$r0
68 | bl %plt(__extendsfdf2)
69 | or $r29,$r5,$r0
70 | or $r13,$r5,$r0
71 | st.w $r4,$r3,0
72 | st.w $r5,$r3,4
73 | or $r10,$r28,$r0
74 | or $r11,$r29,$r0
75 | or $r8,$r26,$r0
76 | or $r9,$r27,$r0
77 | or $r6,$r24,$r0
78 | or $r7,$r25,$r0
79 | la.local $r4,.LC1
80 | bl %plt(sprintf)
81 | or $r12,$r0,$r0
82 | or $r4,$r12,$r0
83 | ld.w $r1,$r3,92
84 | ld.w $r22,$r3,88
85 | ld.w $r24,$r3,84
86 | ld.w $r25,$r3,80
87 | ld.w $r26,$r3,76
88 | ld.w $r27,$r3,72
89 | ld.w $r28,$r3,68
90 | ld.w $r29,$r3,64
91 | addi.w $r3,$r3,96
92 | jr $r1
93 | .size main, .-main
94 | .ident "GCC: (GNU) 8.3.0"
95 | .section .note.GNU-stack,"",@progbits

```

图 4.9 未使用向量指令时产生的汇编代码

## 4.4 在开发板上验证运行 mos

### 4.4.1 在开发板上运行 mos 系统

#### （1）编译改变过后的 mos

首先需要将所有的参数全部转换为开发板所要求的数值。等到数值设置结束后，在 mos 的文件夹底下执行“make test lab=6\_2”指令，运行构建当前最新版本的 mos 系统，在 target 文件夹底下生成 fs.img 和 mos 文件。与在龙芯 Qemu 上运行不同，这时还需要使用“loongarch32r-linux-gnusrf-objcopy ./target/mos ./target/mos.bin -O binary”指令，将 mos 转变为可以上板载入的二进制 bin 文件，方便 bootloader 加载。这样 mos 的编译部分就结束了。

#### （2）使用 vivado 烧录启动文件

首先将串口连接到开发板。等到主机的串口和开发板的串口连接完毕，开发板上的灯亮起后，如图 4.10 所示，需要使用 vivado 打开 complex\_soc.xpr 文件，这个文件起到搭建计算机到开发板之间的联系的作用。之后使用 Generate Bitstream 将连接的文件组织激活，测试计算机是否能够与开发板连接起来，并生成比特流文件。在比特流文件生成后就可以进入烧写配置 FPGA 的阶段了。具体操作是：在比特流文件生成完成的窗口选择“Open Hardware Manager”，确认后进入硬件管理界面，连接 FPGA 开发板的电源线和与电脑的下载线，打开 FPGA 电源。与此同时可以将 mos 的 fs.img 和 mos.bin 复制到 SD 卡上。完成后将 SD 插入开发板侧面的凹槽里。插入后就可以对目标硬件编程了。



图 4.10 串口连接完毕后，开发板上的灯亮起

#### （3）使用串口助手和 PuTTY 远程连接开发板

首先需要判断开发板的波特率，只有在波特率相同的情况下开发板和串口助手才能正



常通信,否则会出现乱码。打开串口助手,设置串口号为 COM3,同时设置传输协议为 Xmodem send 进行通信。点击发送文件把 u-boot.bin 发送到开发板上。u-boot.bin 的作用是为 SD 卡提供可运行的操作环境。插入烧录的 SD 卡后,可以在开发板加载并运行烧录的程序。图 4.11 展示的是发送后的图像。发送结束后,使用 PuTTY 连接开发板进行调试,连接后 PuTTY 会出现向图 4.12 的窗口,这样就可以对在开发板上打开的 mos 进行调试了。

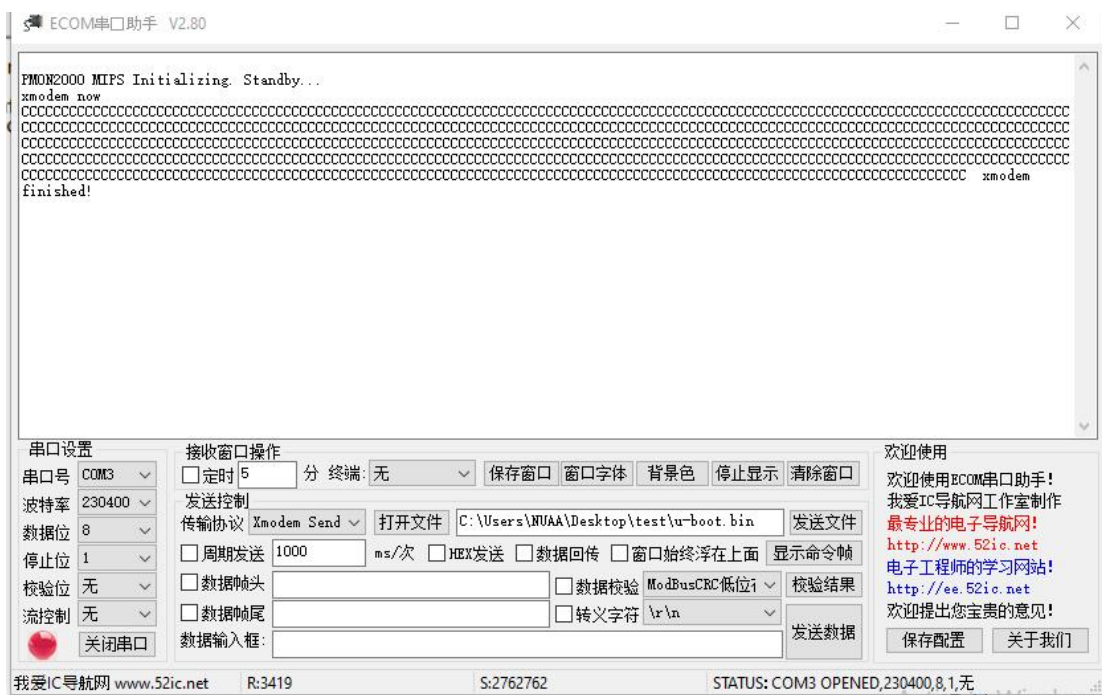


图 4.11 使用串口助手发送文件 u-boot. bit

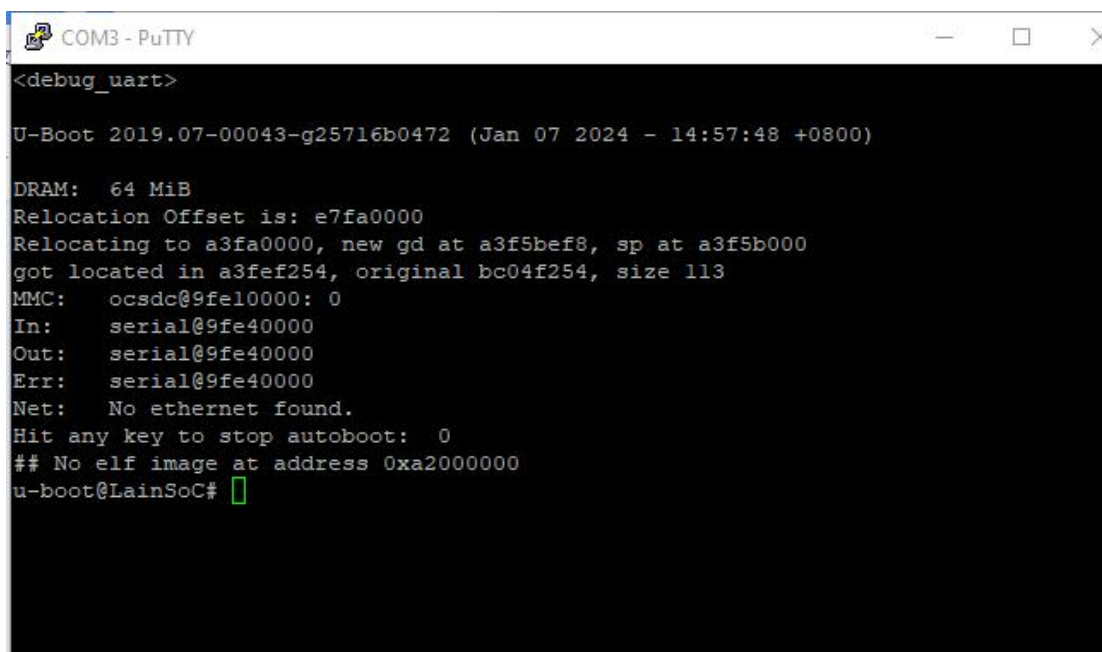


图 4.12 使用 PuTTY 进行远程连接

#### (4) 通过操作 mos 验证移植的成功



这时候通过输入如图 4.13 的指令就能启动 mos 了。上面的两条指令是 fat32 分卷在 SD 卡上加载分卷资源的指令。如果对 mos 的数据进行过修改，那么 go 指令后面的地址很有可能会变化，这需要事先反编译 mos.bin 文件查看启动地址。及时将 go 指令后面跟随的参数，也就是启动地址，进行更换。执行过 go 指令之后，如果成功，就会出现类似图 4.14 的启动界面。可以看到在 mos 在执行 init.b 进行启动时，输出的格式正常，并且通过输出语句显示出当前的操作系统中的文件系统正在顺利运行，之后的 ls.b 能够输出当前的根文件夹下的文件也说明了这一点。然后 init.b 正常输出了当前系统的状态，包括运行状态、启动参数、系统环境、在运行的程序。完成了之后又启动了 sh.b 程序，全程没有出现问题，说明 mos 能够正常运行。

```
fatload mmc 0 0xa0000000 mos.bin
fatload mmc 0 0xa3800000 fs.img
go 0xa0021964
```

图 4.13 需要执行的启动指令

```
kexec: read /motd  
This is /motd, the message of the day.  
  
Welcome to the KOS kernel, now with a file system!  
  
kexec: close /motd  
kexec: spawn /init  
kexec: exiting  
init: running  
init: data seems okay  
init: bss seems okay  
init: args: 'init' 'initarg1' 'initarg2'  
init: running sh  
init: starting sh  
  
:::  
::  
:: MOS Shell 2023 ::  
::  
:::  
  
$ ls
```




图 4.14 在开发板上运行了 mos 操作系统

之后还可以验证用户程序（在前文提到的三个典型的用户程序）的使用，如图 4.15 所示。可以看出，三个用户程序都正常地运行了。`init.b` 显示出了应该输出的语句，`ls.b` 输出了 `user` 文件夹的文件，`pingpong.b` 也展示了两个进程的轮流运行的情形，所以当前

用龙芯编译器编译后的 mos 在开发板上的运行是成功的。在进行实验的过程中，每一次向开发板上传输一次指令，在运行用户程序时，开发板上的一排灯都会闪亮，表示当前开发板正在处理程序。如图 4.16，可以看到电源灯常亮，指示灯闪烁，是正确处理程序的表现。

```
i am killed ...
i am killed ...

$ init.b
init: running
init: data seems okay
init: bss seems okay
init: args: 'init.b'
init: running sh
panic at init.c:47: opencons: 2
i am killed ...
i am killed ...

$ ls.b
testshell.sh test.s script lorem init.b pingpong.b num.b sh.b motd testbss.b tes
tpipe.b newmotd testptelibrary.b echo.b aaa.txt test.b halt.b testarg.b testpipe
race.b ls.b cat.b testfdsharing.b
i am killed ...
i am killed ...

$ pingpong.b
8005 am waiting.....

@@@@@send 0 from 7804 to 8005
7804 am waiting.....
8005 got 0 from 7804

@@@@@send 1 from 8005 to 7804
```

图 4.15 用户程序的实验

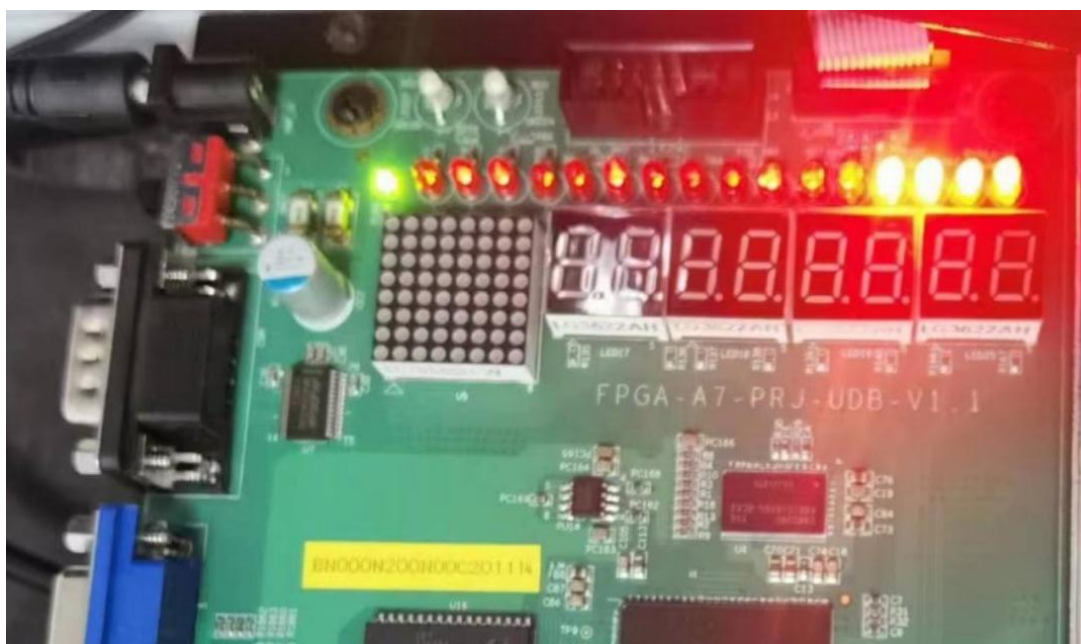


图 4.16 开发板处理程序时灯会亮起

## 4.5 实验结论

目前的任务都得到了完成。首先是运行自己定义的指令的任务。通过在虚拟机上使用编译器编译测试文件里面出现了 `cube.w`，且使用 `objdump` 工具后，反编译也会出现相应的指令。第二个任务是生成浮点和向量指令进行指令优化。从实验的现象中得到的信息来看，确实在汇编代码文件出现了自定义的 `cube.w` 指令，所以这个任务可以认为已经被解决了。从这些现象来看，这个任务也得到了解决。第三个任务是要求编译后的 `mos` 操作系统在开发板运行。这点也确实得到了解决，从 `PuTTY` 上的调试上可以看出，`mos` 是能够在开发板上运行的。因此，整个毕设的预设目标都得到了实现。

## 第五章 工作总结与展望

### 5.1 工作总结

龙芯的国产计算机正在快速发展和分化。不但衍生出多种多样为政务机关和军队服务特化的 CPU 和操作系统，其本身的版本也在不断迭代更新。移植编译器的技术在为新的 CPU 和操作系统实现高效翻译方面起到了至关重要的作用。在本文中完成了移植 1a32r 编译器到型号不同的龙芯开发板上的实验。在此过程中，还为指令集本身加入了一条自定义指令，并且学习了指令的优化方法，做了一些简单的修改。

本人完成的任务总结如下：

(1) 分析了编译器移植的工作背景，对比了国内外使用和移植现状，说明了编译器移植对于做好 CPU 的适配的重要意义。因此，本文的实验重心就是完成编译器的移植，附带指令的添加和优化。在分析了龙芯在国产计算机开发的地位和先进性之后，决定使用龙芯的编译器和开发板进行实验，并对实验工作进行了整理。

(2) 在本文中分析了龙芯编译器的工作原理和实际的编译器处理流程。GCC 的工作由词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成组成，并且实际处理流程上有高级代码到 gimple、gimple 到 rtl、rtl 到汇编代码。最后指出毕业设计的重点是代码优化和目标代码生成两个部分，也就是改变 rtl 到汇编代码的过程。然后介绍了测试用操作系统 mos，分析了 mos 的内部构造和关键部位。同时，还介绍了编译器的移植原理和优化原理，分析了两者的可行性。最后重点分析了向目标机移植编译器的方法，这和修改操作系统的宏定义文件、编译器的目标机器描述文件以及启动时的选项有关，并介绍了在移植时改变了的语法的作用和操作的意義。

(3) 介绍了龙芯编译工具的逻辑和功能，找出需要修改的两个主要区域。然后分析了指令的五种属性，得出了添加指令编码的规律。由此产生了新的指令设计了编码方式，并在实验得到了验证。同时还对机器描述文件进行，给原有的指令添加了新的寄存器匹配机制。同时研究了一部分的 GCC 的优化机制，并阐述了其含义，并试验性地在文件中添加了一些 peephole 优化代码。除此之外，还做出了向量指令的优化和 mos 的细节修改。

(4) 编写了测试文件，并使用 mos 操作系统验证了前一阶段在 Qemu 上的实验现象。在本文中介绍了实验的操作流程和每一步产生的结果，比如在开发板上也成功运行了 mos、常规指令的效率做对比的结果、可以对新的指令进行生成和反汇编以及能够产生向量指令

等现象。种种结果表明经过改造的龙芯编译器编译的操作系统能够运行在龙芯开发板上，成功证明了编译器的移植成功。

## 5.2 展望

本文对于任务方面的介绍到此结束。由于龙芯关于编译器移植的研究还在起步阶段，能查找到的资料比较少，有些还并不完善。在实验中，关于龙芯编译器的研究工作还有众多并未能完全解决，还有一些可以进一步研究的地方：

（1）本文提到添加的指令 `cube.w`，目前只能通过内联汇编的方法调用，不能通过指令直接产生这个汇编指令。而且目前这条指令还不能实现运算操作。究其原因，实际上只有 CPU 认识这条指令后，才能设定这条指令的行为并且规划寄存器的分配规则。目前的 CPU 并没有装载这条指令，就不能实现完全的自动化处理。

（2）当前指令的优化还停留在对寄存器的匹配模式的改变和简单的代码替换上。虽然加入了 `peephole` 的优化代码，但是对于提到的 `peephole2` 优化，虽然能够理解其中的含义，但是进行修改还存在困难。这是因为机器描述文件还受着其他限制文件的制约，还需要修改有联系的其他文件的定义。这还需要深入了解 GCC 的整体结构才能进行下去。

（3）目前使用的 `mos` 系统不支持向量运算，现在是通过自己编写的测试验证改造后的编译器能够生成向量指令的。其他的功能也有些制约。如果需要增加功能，就还需要添加兼容 `risc-V` 指令集或龙芯指令集的库文件。等到 `mos` 更加完善之后再进行测试，可能会得到更多的数据结果。

（4）由于现在使用的龙芯编译器本身就设计得比较完善了，还根据 `-O` 进行划分了优化等级，基本上已经将每一个分级的优化方案全部想到了。比较简易的优化不起作用，实施比较强的优化方案又需要对编译器有很深的理解。如果能够再深入地理解编译器的工作方式细节，也许可以做出更好的效果。

## 参考文献

- [1] Tu et al., Detecting C++ Compiler Front-End Bugs via Grammar Mutation and Differential Testing[J], March 2023, 343-357.
- [2] J. Shen, The Impact of Compiler Optimizations on Symbolic Execution[C], SC, USA, 2021, 161-165.
- [3] S. Hussain, M. Raheem and A. Ahmed, Memory Compiler Performance Prediction using Recurrent Neural Network[C], Kalyani, India, 2023, 490-495.
- [4] G. Zheng, J. Li, W. Gao, L. Han, Y. Li and J. Xu, Operator Fusion Scheduling Optimization for TVM Deep Learning Compilers[C], Chengdu, China, 2023, 273-277.
- [5] R. Zeng, Embedded Linux Operating System Network Accelerated Operation Method Based on ARM Processor[C], Shenyang, China, 2021, 315-319.
- [6] H. Guo, Z. Wang and X. Wang, Transplant of Linux and Embedded System of Boot Loader and LED Driver[C], Kaifeng, China, 2010, 733-736.
- [7] J. Chang, Y. Wang, Q. Meng and Q. Li, Implementation of Thread Local Storage Optimization Method Based on LoongArch[C], Tianjin, China, 2023, 2104-2109.
- [8] 赵晔. 基于 LLVM 的交叉编译器的设计与实现[D]. 西安: 西安电子科技大学, 2015.
- [9] 罗旋. 基于 GCC 的通信专用矢量处理器编译器的研究与开发[D]. 成都: 电子科技大学, 2017.
- [10] 章俊. 面向粗粒度可重构安全芯片的编译器和仿真器研究与设计[D]. 南昌: 南昌大学, 2022.
- [11] 佟玉凤. 应用于 ILP 处理器的编译器导向低功耗并行度调变算法[D]. 天津: 天津大学, 2017.
- [12] 余龙龙. 基于申威 GCC 编译器的间接预取算法设计与实现[D]. 郑州: 中原工学院, 2022.
- [13] 李宏哲. qy\_一种基于编译器的保护返回位址的方法[D]. 高雄: 義守大學, 2017.
- [14] 赖策, 李明东, 刘茜, 等. 面向龙芯处理器的编译检测技术研究[J]. 南充: 西华师范大学学报(自然科学版), 2017, 4:462-466.
- [15] 张磊. 面向 CNN 专用加速器的深度学习编译器优化设计与实现[D]. 重庆: 重庆大学, 2022.
- [16] 胡伟武, 汪文祥, 吴瑞阳, 等. 龙芯指令系统架构技术[J]. 北京: 计算机研究与发展, 2023, 60(1):2-16.
- [17] 董峻峰. 基于龙芯 2 号结构特征对 GCC 的分析与优化[D]. 北京: 中国科学院研究生院(计算技术研究所), 2006.
- [18] 刘玲. 链接后优化在龙芯上的实现[D]. 北京: 中国科学院研究生院(计算技术研究所), 2004.
- [19] 李磊. C 编译器中间代码生成及其后端的设计与实现[D]. 成都: 电子科技大学, 2016.
- [20] 邵阳, 韩昌刚, 全雨, 等. 基于 Qemu risc-V 架构的 OpenHarmony 标准系统移植[J]. 北京: 计算机系统应用, 2023, 32(11):21-28.
- [21] 李雪, 尹健, 贾光帅. 基于 risc-V 向量指令集的内嵌汇编函数设计与实现[J]. 北京: 中国集成电路, 2023, 32(12):36-39+65.
- [22] 胡浩. 基于申威平台 LLVM 编译器的窥孔优化研究[D]. 郑州: 郑州大学, 2020.

## 致 谢

转眼间大学四年就过去了，回过头进入学校仿佛还在昨天，真是感慨万分。在大学四年内我受到了很多人的关照，无论是为人还是学习都有了很大的提高，这四年能顺利通过少不了你们的帮助，于此诚挚地感谢你们。

这次毕业设计可以圆满地完成，运用了我在大学学习的几乎所有课程学习的知识。所以需要首先感谢在南京航空航天大学学习过程中我遇到的所有老师，如果缺少其中一门课程，我可能都无法完成任务。其中特别需要感谢是我这次的毕设导师施慧彬，他有着严谨的治学态度和渊博的知识。在毕设的实验期间，施老师对我的问题总是及时回答，给出的意见也引人思考，具有启发性，总能指导我突破难关。更需要感谢的是在四月下旬的时候，我的毕设一直不能顺利进行下去。期间产生了一些畏难的想法，情绪也不太稳定。施慧彬老师在期间一直耐心地指导我，终于在五月开始的时候解决了让我万分为难的任务。这些更是让我难以忘记。

也需要感谢 mos 的开发者，北航的学长王哲，他在使用 mos 的方面给了我许多建议。在编译器的移植方面也给出有效的建议。期间还夹杂着其他方面的疑问，有时并不是他做的项目，他也能提出有建设性的意见，帮了我很大的忙。真的非常感谢他。

同时还需要感谢与我同组的同学，他们时常关心我的毕设进度，也会提醒我一些重要的时间点和关键事件。在毕设过程中，我们经常一起在实验室或者教室内一起学习，使得毕业设计不再枯燥。感谢你们，让我度过了一段充实而快乐的时光。

在这之后，我要感谢我的家人和朋友，在我整个学习生涯中给予了我无尽的支持和鼓励。没有他们的支持，我无法顺利完成这篇论文。

最后还要感谢在百忙之中参与作者论文审阅、评议和参加答辩的各位专家、教授。