基于图着色的寄存器分配器设计与实现实验报告

项目简介

本项目实现了一个针对 TinyRISC 虚拟架构的后端模块，核心功能是分配寄存器，并将使用虚拟寄存器变量的汇编风格IR转换为使用真实寄存器的汇编代码。项目重点在于寄存器分配算法的实现，采用了经典的干扰图着色算法。系统能够对控制流图进行指令流分析，计算 Use 和 Def 集合，完成活性分析。在此基础上，构建寄存器干扰图，利用启发式的 Simplify 和 Spill 算法将无限的虚拟寄存器映射到有限的 K 个物理寄存器。当发生寄存器溢出时，系统具备重写控制流图的能力，能够自动插入内存加载与存储指令，并重新开始分配流程，直至生成合法的汇编代码。

编译与运行说明

项目构建

见README

设计实现重点

在实现过程中，最关键的挑战在于干扰图的构建与溢出处理机制。为了保证分配的正确性，我在构建干扰图时不仅处理了同一程序点活跃变量（LiveIn 集合）之间的两两干扰，还特别处理了定义变量（Def）与出口活跃变量（LiveOut）之间的干扰。这是为了防止某条指令定义的变量复用了该指令之后仍需使用的变量的寄存器，从而导致数据被覆盖的严重错误。在溢出处理方面，我实现了一个迭代式的重写框架。当分配器发现无法为图进行 K 着色时，会选择度数较高的节点进行溢出，为其分配栈槽，并在 IR 中插入 Load 和 Store 指令。修改后的 IR 会被送入下一轮编译循环，重新进行活性分析和干扰图构建，直到所有变量都驻留在寄存器或栈中。此外，代码生成阶段还处理了函数调用约定的栈帧管理，生成标准的 Prologue 和 Epilogue。

测试方案与结果分析

为了全面验证分配器的正确性和鲁棒性，项目中集成了多个维度的测试用例，在构建完成后运行可执行文件demo和test_all即可看到。
可执行文件test_all的程序入口在 `tests/test_all.cpp` 。
这部分测试包括了一些最基本的正确性测试，具体包含：

1. **Use/Def 分析**:
    - 测试 `computeUseDef` 函数能否正确识别指令中的**使用（Use）**和定义（Def）**的虚拟寄存器。
2. **活跃变量分析 (Liveness Analysis)**:
    - 测试 `liveness` 函数能否在基本块（Basic Block）中生成非空的入口（In）和出口（Out）活跃集。
3. **干扰图构建 (Interference Graph)**:
    - 测试 `buildInterference` 函数能否基于活跃变量构建出非空的干扰图（Interference Graph）。
4. **寄存器分配与溢出处理 (Register Allocation & Spilling)**:
    - **溢出测试 (K=2)**: 当物理寄存器数量极少（K=2）时，验证编译器是否会触发**溢出（Spill）**机制，即生成的汇编代码中包含 `LOAD` 和 `STORE` 指令。
    - **无冲突分配测试 (K=12)**: 当物理寄存器充足（K=12）时，验证：
        - 所有相互干扰的变量是否被分配了**不同**的寄存器（图着色正确性）。
        - 分配的寄存器编号是否在合法范围内（0 到 K-1）。
        - 生成的汇编代码中使用的最大寄存器编号是否小于 K。
        -

demo的程序入口在project1.cpp，可以进一步测试以下功能

1. **测试用例 1** 旨在验证基本块内的纯数据流依赖与寄存器复用。该测试计算多项式 $y = x^2 + 2x - 5$，配置物理寄存器数量 $k = 8$。从输出的汇编代码可以看出，虽然我们提供了 8 个寄存器，但生成的代码仅使用了 `R0` 、 `R1` 和 `R2` 三个寄存器。代码首先将 `x` 加载到 `R2` 中，计算 x^2存入 `R1` ，随后在计算 $2x$时复用了 `R0` 。这测试了活性分析准确地判断了变量生命周期的结束点，分配器成功回收并

复用了不再活跃的寄存器，生成了紧凑高效的代码，且堆栈槽使用量为 0，表明在资源充足时未发生不必要的溢出。

2. **测试用例 2** 侧重于验证对源程序中显式内存操作的处理以及对死代码的兼容性。我们将物理寄存器数量 K 设置为 3。输出结果显示，源程序中的 `Load` 指令被正确翻译为栈偏移访问，例如 `Operand mem 100` 被转换为加载地址 `fp` 减 404，这验证了后端对栈帧偏移量计算公式的正确性。此外，虽然代码中定义了一个死变量 `v4` 赋值为 999 且后续从未被使用，但生成的汇编中依然包含了一条 `MOV R0, #999` 指令。这表明分配器具有良好的鲁棒性，能够安全地为死变量分配寄存器而不导致程序崩溃或破坏其他活跃变量的数据流，证明了干扰图构建逻辑的完整性。

3. **测试用例 3** 构建了一个菱形控制流结构，这是验证跨基本块活性分析最关键的场景。程序的控制流从入口分叉为左右两个分支，分别对变量 `v4` 进行不同的计算，最后在汇合块中使用 `v4`。配置 $k = 3$。观察输出的汇编代码可以发现一个关键细节：在左分支 `LEFT_BRANCH` 中，计算结果被写入了 `R0`；在右分支 `RIGHT_BRANCH` 中，计算结果同样被写入了 `R0`；而在随后的 `MERGE` 块中，乘法指令直接使用了 `R0` 作为源操作数。这强有力地证明了分配器的活性分析算法正确处理了控制流的汇合点，成功将汇合块的 `LiveIn` 信息反向传播到了两个前驱分支的 `LiveOut` 中，从而迫使图着色算法在整个控制流图中为该虚拟变量分配了同一个物理寄存器 `R0`，保证了程序逻辑的正确执行。

4. **测试用例 4** 是高压循环溢出测试，用于验证资源受限时的溢出处理机制。输入逻辑包含 `8` 个跨越循环活跃的常量以及循环计数器和累加器，总活跃变量数远超配置的物理寄存器数量 `4`。输出的汇编代码显示了显著的溢出特征：在 `ENTRY` 块中，常量加载到寄存器后立即执行了 `STORE` 指令将其保存到栈中，例如 `STORE [fp-8], R0`。而在 `LOOP_BODY` 循环体内，在执行 `ADD` 加法指令之前，代码插入了 `LOAD R0, [fp-8]` 等指令从栈中重新加载数据。最终统计显示使用了 `13` 个栈槽位。这些细节证明了当着色失败时，分配器能够正确识别溢出候选者，重写控制流图并插入必要的内存访问指令，通过以空间换时间的方式，在极端受限的寄存器资源下生成了逻辑正确的代码。