实验四报告：RISC-V 代码生成

本编译器已实现将 Zero IR 有效翻译为 RISC-V 汇编，解决了代码生成中的关键问题。

1. 指令选择（Instruction Selection）

指令选择阶段将每条 Zero IR 指令翻译为一条或多条 RISC-V 汇编指令，指令使用自定义的 C++ 类表示，定义在 `codegen/asm.hpp` 中。该转换过程由 `InstSelector` 类（实现于 `codegen/inst_selector.cpp` 和 `codegen/inst_selector.hpp`）负责完成。

关键 IR 到 ASM 的映射：

- **常量与赋值：**
  - `a = #t`（IR `LoadImm`）→ `li reg(a), t`（ASM `Li`）
  - `a = b`（IR `Assign`）→ `mv reg(a), reg(b)`（ASM `Mv`）
- **算术/逻辑操作：**
  - `a = b op c`（IR `Binary`）→ `op reg(a), reg(b), reg(c)`（ASM `Arith`）
  - `a = b op #imm`（IR `Binary` 含立即数）→ `opi reg(a), reg(b), imm`（适用于 `add` / `sub` 的 ASM `ArithImm`）。对于其他操作，立即数先加载到 `t0`，然后使用寄存器形式操作。
  - `a = op b`（IR `Unary`，如取负）→ `sub reg(a), zero, reg(b)`（ASM `Arith`）
- **控制流：**
  - `LABEL label:`（IR `Label`）→ `label:`（ASM `Label`）
  - `GOTO label`（IR `Goto`）→ `j label`（ASM `Jump`）
  - `IF x op y GOTO label`（IR `If`）→ `branch_op reg(x), reg(y), label`（ASM `Branch`，如 `bgt`、`ble`）
- **内存操作：**
  - `x = &y`（IR `LoadAddr`）→ `la reg(x), y`（ASM `La`）
  - `*x = y` 或 `*(x + #k) = y`（IR `Store`）→ `sw reg(y), k(reg(x))`（ASM `Store`）。大偏移通过中间寄存器 `t0` 计算地址。
  - `x = *y` 或 `x = *(y + #k)`（IR `Load`）→ `lw reg(x), k(reg(y))`（ASM `Load`），大偏移处理同 `Store`。
  - `DEC x #k`（IR `Dec`，局部数组/变量声明）→ `addi reg(x), sp, offset`。栈空间由 `current_func->alloc_dec()` 管理，`offset` 为偏移量。
- **函数定义与调用：**
  - `FUNCTION func:`（IR `Function`）→ `.globl func`（ASM `GlobalLabel`）和 `func:`（ASM `Function`）
  - `ARG x, k`（IR `Arg`）：前 8 个参数放入 `a0-a7`，其余通过 `sw` 存于栈，由 `current_func->alloc_temp()` 分配空间。
  - `a = CALL f`（IR `Call`）：

    - 调用前将 t0-t2 保存至栈（ sw ），调用后恢复（ lw ）。
    - 使用 call f 调用函数。
    - 若有返回值，从 a0 移动到 reg(a) （ mv reg(a), a0 ）。
  - PARAM x, k （IR Param ）：前 8 个参数从 a0-a7 取值，超过部分从调用者栈帧中通过 lw 取值。
  - RETURN a （IR Return ）：返回值赋给 a0 的逻辑在函数结尾（Epilogue）处理， selectReturn 本身不生成指令。

2. 寄存器分配（Register Allocation）

使用了朴素的寄存器分配策略，定义在 RegAllocator 类中（文件 codegen/reg_allocator.cpp/hpp ）。

- **策略说明**：
  i. 对于使用虚拟寄存器的指令：
    - **加载操作数**：若 rs1 或 rs2 为虚拟寄存器，从栈槽加载至物理临时寄存器（ t0-t2 ，轮转分配）；
    - **执行指令**：使用这些物理寄存器执行操作；
    - **存储结果**：若结果是虚拟寄存器，将其值从物理寄存器存回其栈槽。
  ii. 物理寄存器（如 sp ， fp ， a0 ）不参与分配，直接使用。
  iii. 结构性指令（如 Label ， Jump ， Call ， Ret ）不涉及虚拟寄存器，直接保留。
- 每个虚拟寄存器的栈偏移由 Function::alloc_temp() 分配与记录，确保一致性。

该方法简化了分配逻辑，牺牲性能换取正确性。

3. 栈帧管理（Stack Frame Management）

栈帧管理用于函数调用、局部变量存储和寄存器保存，核心由 ASMEmitter 类（ codegen/asm_emitter.cpp/hpp ）与 Function 对象中的栈空间信息协同完成。

- **函数序言（Prologue）**：
  i. 为 fp 和 ra 分配栈槽（由 alloc_temp() 获得偏移量）；
  ii. sp 向下移动函数所需总栈空间（包括临时变量、保存寄存器、局部数组、调用参数）；
  iii. 将旧的 ra 和 fp 保存至栈；
  iv. 设置新的 fp： addi fp, sp, stack_size 。
- **函数结尾（Epilogue）**：
  i. 若有返回值，将其赋给 a0 （ mv a0, reg(x) ）；
  ii. 从栈中恢复 ra 和 fp ；
  iii. sp 恢复至调用前位置；
  iv. 发出 ret 指令。

- **调用者/被调用者职责**：
  - **调用者**（在 `InstSelect::selectCall` 中实现）：保存其 `t0-t2`；传递参数超出 8 个时入栈；
  - **被调用者**（在 `ASMEmitter::emit`遇到`Function`类型的`inst` 时实现）：设置栈帧；保存 `ra` 和 `fp`；从寄存器或栈获取参数。
- **局部变量管理**：通过 `DEC x #k` 指令触发栈空间分配，由 `alloc_dec()` 管理偏移，再用 `addi` 存储地址。

4. 汇编生成（Assembly Emission）

`ASMEmitter` 类将 `ASM::Inst` 对象序列转为最终汇编代码文本。

- **输出结构**：
  - 若设置 `use_venus`，则额外输出 `read` 和 `write` 系统调用辅助函数；
  - `.data` 区：存储全局变量，由 `selectGlobal` 生成；
  - `.text` 区：输出函数体。
- **指令格式化**：每个指令类均实现 `to_string()` 方法，用于生成其对应的文本形式。
- **处理大立即数**：
  - 对于 `addi`，`lw`，`sw` 等立即数超出 12 位的指令，拆分为：
    a. `li temp_reg, imm`
    b. 原操作指令改用该寄存器（例如 `add rd, rs1, temp_reg`）
  - 使用 `t4` 作为临时寄存器。
- **寄存器替换**：在输出前调用 `inst->replace_all(reg_map)` 替换虚拟寄存器；由于 `RegAllocator` 已处理替换，通常此步骤无实际作用，仅作保险。

实现亮点

- **清晰的汇编指令类层级结构**：`ASM::Inst` 各子类封装了指令的表示与行为（如字符串转换、寄存器使用/定义集合），使得代码更模块化、更易维护。
- **统一的栈空间管理体系**：`Function` 类集中管理所有栈空间的分配，通过 `alloc_temp` 和 `alloc_dec` 等方法维护 `temp_stack_size`，`reg_stack_size`，`temp_offset` 等字段，确保代码生成各阶段一致性。
- **模块划分明确**：`InstSelector` 负责 IR 到 ASM 的翻译，`RegAllocator` 负责寄存器分配与溢出，`ASMEmitter` 负责汇编输出与函数框架维护，结构清晰，利于后续优化。
- **针对 RISC-V 特性的优化**：如大立即数处理（超出 12 位的值通过 `li` 和中转寄存器实现）和兼容 Venus 的系统调用封装等，充分考虑目标架构特点。
- **遵循 RISC-V 调用约定**：正确区分调用者与被调用者的职责，按规则处理参数传递、返回值、寄存器保存等，增强代码的通用性与可执行性。