实验四：目标代码生成 实验报告

成员	学号	院系	邮箱
张路远	221220144	计算机学院	craly199@foxmail.com
姜帅	221220115	计算机学院	jsissosix1221@gmail.com

代码结构

1. **全局定义与常量**
   - `REG_NAME[32]`：定义了 MIPS 寄存器名称数组，用于生成指令时引用寄存器。
   - `reserved[]`：保存了一些常用的 MIPS 操作符字符串，用于名称冲突检测。
   - 宏定义 `REG_START`、`REG_END`、`RESERVED_COUNT`：分别表示可用于分配的寄存器编号起始、结束位置，以及保留字数量。
2. **数据结构**
   - `Register regs[32]`：用于维护 32 个寄存器的使用状态、绑定的变量以及最近使用信息。
   - `Frame frameStack` 与 `Frame currentFrame`：用于跟踪各个函数的活动记录（栈帧）；`frameStack` 是全局链表，`currentFrame` 指向正在翻译的函数的栈帧。
3. **保留字处理**
   - `is_reserved(const char *name)`：检测给定名称是否与 MIPS 保留字冲突。
   - `changename(const char *name)`：若与保留字冲突，则在名称后追加下划线，确保标签或函数名合法。
4. **栈帧管理与变量链表**
   - `updateFramePointer(char *name)`：根据函数名称在 `frameStack` 中查找并更新 `currentFrame`。
   - `initFrames(InterCode ir)`：遍历 IR 链表，为每个函数创建栈帧节点，并为其中涉及的变量预先分配栈偏移。
   - `lookupVariable(Operand op, Frame frame)`、`insertVariable(Operand op, Frame frame)`、`handleVariable(Operand op, Frame frame)`、`tryHandleVariable(Operand op, Frame frameStack)`：一组函数用于在当前帧中查找或插入变量，并维护变量在栈区的偏移信息。
5. **寄存器管理**
   - `initRegisters()`：为 32 个寄存器结构体分配内存，并初始化其编号、使用标志等。
   - `freeRegisters()`、`clearRegisters()`：分别在发射每条指令后或函数入口时，重置可用于分配的寄存器状态。
   - `getRegForVar(FILE *fp, Variable var)`：根据变量请求，分配寄存器。优先使用空闲寄存器，若寄存器耗尽则采用 LRU（最近最久未使用）策略执行写回与替换。
   - `updateRound(int no)`：更新各寄存器的最近使用轮次（用于 LRU 算法）。
6. **操作数处理**
   - `handleOp(FILE *fp, Operand op, bool load)`：核心函数，用于将 IR 中的操作数（常量、变量、取地址、取值）转换为相应的 MIPS 代码片段，并返回其所在寄存器编号。
7. **函数参数与寄存器保存/恢复**
   - `pushRegsToStack(FILE *fp)`、`popStackToRegs(FILE *fp)`：在函数调用时，将可用寄存器压栈或从栈中恢复，以保证调用者保存的寄存器不会被覆盖。
8. **目标代码文件初始化**

- `initObjectCode(FILE *fp)`：预先输出 `.data` 段与 I/O 子例程 `read`、`write`，包括提示字符串与系统调用序列。

9. **主调度函数**
   - `printObjectCode(char *filepath, InterCode ir)`：整体驱动函数，负责打开目标汇编文件，调用各初始化函数，遍历 IR 链表，根据 IR 节点类型生成相应 MIPS 指令，并在合适时机清理寄存器状态，最终关闭文件。

实现方法

1. 寄存器管理

- **寄存器初始化**：
  - 在调用 `printObjectCode` 时首先执行 `initRegisters()`，为所有 32 个寄存器分配 `Register_t` 结构体，并将其设置为未使用状态（`isInUse = false`）。
  - 仅编号 8 到 25 这 18 个寄存器（`$t0`–`$t7`，`$s0`–`$s7`，`$t8`–`$t9`）会用于普通变量的分配，其余寄存器用于固定用途（如 `$zero`，`$at`，`$v0` 等）。
- **寄存器分配算法 (LRU)**：
  1. 当 `handleOp` 请求为某个变量或常量加载到寄存器时，调用 `getRegForVar`：
     - 遍历编号 8–25 的寄存器列表，若发现空闲寄存器（`isInUse == false`），直接分配给该变量。
     - 若某个寄存器已绑定到同一变量，说明之前已经加载；此时直接复用该寄存器。
  2. 若所有可用寄存器均被占用，则执行 LRU 置换：
     - 遍历所有可用寄存器，选取 `lastUse` 值最大的寄存器（最近最久未使用）。
     - 生成 `sw $reg, offset($fp)` 代码，将被替换寄存器中原变量的值写回栈。
     - 将该寄存器重新绑定到新变量，重置其 `lastUse` 为 0。
  3. 每次分配或复用寄存器后都调用 `updateRound(i)`：
     - 遍历可用寄存器，自增 `lastUse`，并将被分配寄存器的 `lastUse` 置 0，以保持 LRU 信息。
- **寄存器清理**：
  - 在发射每条 IR 对应的 MIPS 指令后，调用 `freeRegisters()` 将编号 8–25 所有寄存器的 `isInUse` 重置为 `false`，仅清空使用标志；而 `clearRegisters()` 在函数入口处额外将 `var` 与 `lastUse` 置空、置 0。
  - 这样可保证在每条指令之间，不会因为残留状态导致新的操作数分配不正确。

2. 栈帧与变量管理

- **栈帧节点创建（`initFrames`）**：
  - 初始遍历整个 IR 链表，遇到 `FUNC_` 节点时，创建一个新的 `Frame_t` 结构体，拷贝函数名到 `frame->name`，初始化 `varlist = NULL`，并将新节点链入 `frameStack`。
  - 对于其他涉及变量的节点（如 `ASSIGN_`、`PLUS_`、`IF_GOTO_`、`RETURN_`、`CALL_`、`ARG_`、`PARAM_`、`READ_`、`WRITE_`），调用 `tryHandleVariable`：若操作数是普通变量或临时变量，则先调用 `handleVariable`，通过 `lookupVariable` 和 `insertVariable` 建立或查找该变量在当前 `frameStack` 的偏移。
  - `insertVariable` 为新变量分配偏移：若是该帧的第一个变量，则偏移 = 该类型大小（通过 `getSizeof(type)`）；否则在链表头部累加上上一个头部变量的偏移，保证每个变量占用栈空间大小连续分配。
- **变量查找与插入**：
  - `lookupVariable`：遍历当前 `frame` 的 `varlist` 链表，比较 `Operand` 的 `kind` 与名字/编号是否一致；若找到返回对应 `Variable_t`。

- `insertVariable`：为新变量分配一个 `Variable_t`，设置其 `offset`，将其链入 `frame->varlist`，并返回该节点。
- `handleVariable` 封装了查找与插入逻辑，确保对某个操作数多次请求时只插入一次。

- **更新函数栈帧指针（`updateFramePointer`）**：
  - 在实际发射每个函数体、函数调用与返回指令时，根据函数名切换到对应 `Frame`。
  - 在 `printObjectCode` 的主循环中，遇到 `FUNC_` 节点时先调用 `updateFramePointer` 更新 `currentFrame`，保证后续的变量分配与栈偏移都作用在正确的帧上。

3. 操作数与指令生成

- 此部分设计原理与实验3没有区别，不再赘述

4. 函数调用约定与栈布局

- **栈帧格式**（由高地址向低地址增长）：
  - 调用者保存区：当函数调用时，通过 `pushRegsToStack` 将编号 8–25 可用寄存器依次 `sw` 到栈中，共占 18*4=72 字节。
  - 保存原函数 `$fp`：在函数入口由 `sw $fp, 0($sp)` 保存旧帧指针。
  - 新的 `$fp`：`move $fp, $sp`。
  - 局部变量区：`addi $sp, $sp, -<localsize>` 为本地变量分配空间，所有临时及局部变量通过 `-offset($fp)` 访问。
  - 参数传递区：
    - 前 4 个参数由调用方放在 `$a0`–`$a3`；被调用方在函数入口时将其存入栈帧对应偏移。
    - 第 5 个及之后的参数由调用方通过栈压入，被调用方在入口处通过 `$fp` 访问。
- **寄存器保存与恢复**：
  - 在调用前，调用方将编号 8–25 的寄存器压栈(`pushRegsToStack`)；调用结束后，调用方再执行 `popStackToRegs` 恢复这些寄存器。
  - 函数入口自己会保存 `$fp`，并在返回前通过 `$fp` 恢复前一个 `$fp`。

5. 特殊设计

- **保留字检测**：在生成函数或标签名称时，调用 `is_reserved` 检查是否与 `reserved[]` 中的已有操作符冲突，若冲突调用 `changename` 在末尾加下划线，确保不会与 MIPS 指令同名。保留字列表可以按需编辑。
- **刷新寄存器状态**：在每条 IR 指令生成完成后，调用 `freeRegisters()` 将所有临时寄存器标记为空闲。这种解法简单、低效但管用。
- **输出格式化**：在 `printObjectCode` 中，针对不同 IR 节点类型，通过 `fprintf` 精准控制 MIPS 指令的格式与缩进，实现易读的汇编输出。