代码解读

整体流程

目标代码生成是编译器工作流程中的最后一个关键步骤。在该步骤中：

- 输入：中间代码
- 输出：MIPS 汇编代码
- 实现：目标代码生成器

实验目的：将实验三中得到的中间代码经过与具体体系结构相关的指令选择、寄存器选择以及栈管理之后，转换为 MIPS32 汇编代码，并在 SPIM Simulator 上运行

代码解读

整个目标代码生成部分文件分为 objectcode.c 和 objectcode.h，其中核心代码在 objectcode.c 中

objectcode.c 代码结构很简单，核心函数就是 printObjectCodes 进行目标代码生成，这也是目标代码生成的起点（main.c 中调用） 其他函数都可以看作是辅助函数（寄存器相关操作、初始化等）

核心函数 `printObejectCodes` 代码解读

起点（main.c 中调用）是 `printObjectCodes` 函数

这个函数通过读取中间代码并根据不同的指令类型将 IR 翻译成相应的符合 MIPS 汇编指令格式的目标代码，并写入到指定的文件中。

- 在中间代码中，包含了函数、局部变量和临时变量等符号信息，而在目标代码中这些符号信息需要转化为具体的地址。
- 在中间代码中，我们使用临时变量如#1、#2 存储运算的中间结果，而在将中间代码翻译为目标代码时，这些临时变量需要被存储到寄存器或内存中。

1. LABEL_IR

生成一个标签（例如 `label1:`），用于跳转

2. FUNC_IR（含 PARAM_IR 处理）

包括函数栈帧初始化、局部变量分配和 PARAM_IR 的处理：

- **函数名打印**：`curr->ops[0]->name`，是函数在汇编中的标签
- **栈帧初始化**：
  - `addi $sp, $sp, -4`：将栈指针 `$sp` 向下移动 4 字节，为保存 `$fp`（帧指针）留出空间
  - `sw $fp, 0($sp)`：将当前的帧指针 `$fp` 保存到栈中，以便恢复函数调用时的栈帧指针
  - `move $fp, $sp`：更新 `$fp` 为当前的栈指针 `$sp`，标记新的栈帧开始，从 `$fp` 开始访问局部变量
- **非 `main` 函数：**
  - 通过 `pushAllRegs(fp)` 将所有可用寄存器的值压栈，保存当前函数调用时的寄存器状态，防止在函数调用时寄存器内容丢失
  - 调整栈指针 `$sp`，在栈上为局部变量分配空间。分配的空间大小由当前函数的栈帧描述符（`frame`）中的 `vars->offset` 决定
- **`main` 函数：**在栈上为局部变量分配空间，栈指针移动 `frame->vars->offset` 字节
- `clearRegs()`清除寄存器的状态，准备处理函数参数中间代码 `PARAM_IR`
- **处理函数参数（`PARAM_IR`）**
  - 循环遍历所有的参数声明指令 `PARAM_IR`
  - 前四个参数传递给特定寄存器：MIPS 使用寄存器 `$a0` 至 `$a3` 来传递前四个参数。如果参数数量少于或等于 4，则将这些参数从**寄存器**加载到 `$a0`, `$a1`, `$a2`, `$a3` 中
  - 更多的参数通过栈传递：对于超过四个的参数，从**栈**中按至顶向下的顺序加载到寄存器中（使用 `lw` 指令）
- `spillReg(regs[reg], fp)`这行代码将当前寄存器的内容保存到栈中，以防止后续操作中寄存器被覆盖

3. ASSIGN_IR

负责生成将右操作数赋值给左操作数的 MIPS 汇编代码。根据左操作数的类型：

- 如果是变量或临时变量，使用 `move` 指令将右操作数的值传递给相应的寄存器
- 如果是取值操作（例如取指针指向的值），则使用 `sw` 指令将右操作数存储到左

操作数指向的内存地址

4. PLUS_IR、SUB_IR、MUL_IR 和 DIV_IR

加减乘除算术运算，步骤都差不多：

1. 获得操作数（左、右共三个）
2. 对每个右操作数（`right1` 和 `right2`），通过 `handleOp(right, fp, 1)` 函数将其转换为对应的寄存器，并获取其在寄存器中的索引，即将右操作数被正确加载到寄存器中，以供后续运算使用
3. 处理左操作数：分类处理，是变量或临时变量（`VARIABLE_OP` 或 `TEMP_VAR_OP`）/ 是取值操作（`GET_VAL_OP`）
4. 生成相应的算术汇编指令（加法、减法、乘法、除法）
5. 运算并存储结果

请完成 TODO1：实现 SUB_IR 的目标代码生成，可仿照 PLUS_IR 的过程

5. TO_MEM_IR

目的是将右操作数的值存储到由左操作数（变量或临时变量）指定的内存位置中：

- 先将右操作数加载到寄存器
- 再通过 sw 指令将其存储到由左操作数所指定的内存地址

6. GOTO_IR 和 IF_GOTO_IR

- GOTO_IR：无条件跳转，通过 `j label%d` 指令实现跳转，标签编号由 `curr->ops[0]->no` 提供
- IF_GOTO_IR：有条件跳转，根据给定的条件（两个操作数的比较）生成条件跳转指令
  - 比较 `left` 和 `right` 操作数，使用 `handleOp` 将它们加载到寄存器 `regLeft` 和 `regRight`
  - 根据 `curr->relop` 的值（如 `==`, `!=`, `>`, `<` 等）确定合适的条件跳转指令（如 `beq`, `bne`, `bgt`, `blt` 等）
  - 根据 `relop` 生成不同的条件跳转汇编指令（如 `beq`, `bne` 等），跳转到指定的标签 `label%d`，其中 `%d` 是 `curr->ops[2]->no`（目标标签的编号）
    - 示例：`beq $regLeft, $regRight, labelN`（如果 `regLeft ==`

`regRight`，跳转到标签 `labelN`

7. RETURN_IR

负责函数的返回操作，处理返回值、栈帧恢复、寄存器恢复以及跳转到返回地址，确保函数的返回过程正确执行

8. DEC_IR

DEC 指令不需要翻译，因为在预先扫描的过程中已经为所有变量在栈中分配了空间

9. ARG_IR

传参代码一定是在 CALL 指令之前，所以不单独翻译，在 CALL_IR 部分翻译

10. CALL_IR（含 ARG_IR 处理）

这段代码处理函数调用指令的目标代码生成

1. **处理函数参数传递（ARG_IR）**：
    - 通过 `preCode` 遍历与当前 `CALL_IR` 相关的所有 `ARG_IR`（参数指令），并为每个参数分配寄存器。
    - 如果参数个数少于或等于 4，将参数分别存放在特定的寄存器中（`$a0` 到 `$a3`）。
    - 如果参数超过 4，则将其存储到栈上，保证参数在栈中按顺序从后往前压栈。
    - 使用 `regNos[]` 数组来保存超过四个参数的寄存器编号，按顺序从后往前将这些参数压栈。
2. **保存返回地址**：将返回地址 `$ra` 压栈，以确保函数返回后能够跳回正确的位置。
    - 执行 `addi $sp, $sp, -4` 和 `sw $ra, 0($sp)` 将返回地址保存到栈上。
3. **函数调用**：使用 `jal` 指令跳转到目标函数（`jal %s`），执行函数调用操作。
4. **恢复返回地址**：调用返回后，恢复栈中的返回地址 `$ra`：
    - 执行 `lw $ra, 0($sp)` 和 `addi $sp, $sp, 4`。
5. **处理返回值**：
    - 如果调用的函数有返回值，检查操作数类型（`VARIABLE_OP`、`TEMP_VAR_OP` 或 `GET_VAL_OP`）。
    - 如果返回值存储在变量或临时变量中（`VARIABLE_OP`、`TEMP_VAR_OP`），使

用 `move` 指令将返回值 `$v0` 存储到相应寄存器，并将寄存器的内容写回栈或存储区。

- 如果返回值是通过 `GET_VAL_OP` 存储的，则将 `$v0` 存储到指定内存地址。

11. READ_IR 和 WRITE_IR

以 READ_IR 为例逐行解释

- `fputs("  addi $sp, $sp, -4\n", fp);`将栈指针（$sp）减少 4，创建栈空间，为接下来的栈操作（保存 $ra 寄存器）腾出空间
- `fputs("  sw $ra, 0($sp)\n", fp);`将返回地址寄存器 $ra 保存到栈上（栈指针 $sp 指向的位置）。这样做是为了在函数调用之后能够恢复返回地址，确保程序的控制流正确
- `fputs("  jal read\n", fp);`调用 read 函数
- `fputs("  lw $ra, 0($sp)\n", fp);`从栈顶加载 $ra 寄存器的值，恢复返回地址。这样 read 函数执行完后，程序能正确跳回到调用函数的地方
- `fputs("  lw $ra, 0($sp)\n", fp);`恢复栈指针，将栈指针回到原来位置
- 剩下的代码做的事就是存储 read 函数的返回值，具体可以看代码里的注释

请完成 TODO2：实现 WRITE_IR 的目标代码生成，已给出具体提示，可参考 READ_IR 的过程

辅助函数在代码中已有注释，这里不再赘述