姓名：乔羿童
学号：21311111
邮箱：qiaoyt3@mail2.sysu.edu.cn

Lab4_实验报告

编译和测试方法

1. `cd Code/`：进入源代码和`Makefile`文件所在的路径下
2. `make`：使用`make`命令对程序代码进行**编译构建**
3. `./parser ../Test/xxx.cmm ../result/temple.s`：对指定的 C--代码`xxx.cmm`进行`MIPS32`汇编代码的生成，将输出结果写入到`result`文件夹下名为`temple.s`的文件中。 也可以修改`Makefile`文件中的`test`部分的内容，然后使用`make test`命令进行测试和`MIPS32`汇编代码的输出。本人`Makefile`文件中已经将`Test`文件夹中需要测试的代码的路径均写入，同时输出`Test`文件夹中多个测试文件的测试结果。

本人已经使用`SPIM Simulator`对`Test`文件夹中 3 个测试文件对应的汇编代码进行测试，均能够正常运行。并且已经将生成的 3 个汇编代码文件存储到`result`文件夹中。

本次实验中的 **todo** 部分涉及：寄存器分配、操作数装载、中间代码翻译。

_Todo1

要求：存在空闲寄存器的情况下，为变量描述符分配寄存器。

解决思路：对于【不存在空闲寄存器】的情况中采用最长时间未使用算法，定位到合适的寄存器`i`后进行寄存器的分配。 因此此处存在空闲寄存器的情况直接参考定位到合适寄存器之后的代码，进行空闲寄存器`i`的分配，返回分配的寄存器编号。

下面是代码实现过程：

1. 分配寄存器：设置寄存器描述符的`var`字段，将变量分配到空闲寄存器`i`中。
2. 更新寄存器使用间隔：调用`updateInterval`函数更新寄存器的使用间隔。
3. 装载变量值到寄存器：根据`load`参数判断是否需要将变量值装载到寄存器中。如果`load`置为`1`，则进行装载操作。后续设置`load`参数处理不同情况，若分配寄存器后直接进行值覆盖，则`load`置为`0`，没必要装载值。装载操作依赖于变量的类型：
    - 常量：使用`li`**指令将常量值直接装载到寄存器中**。

```
fprintf(fp, "  li %s, %d\n", regs[i]->name, var->op->value);
```

    - 变量或临时变量：使用`lw`**指令将存储在栈上的变量值装载到寄存器中。**

```
fprintf(fp, "  lw %s, %d($fp)\n", regs[i]->name, -var->offset);
```

_Todo2

要求：完成地址操作数的装载。参考值操作数的装载进行代码编写。

解决思路：`GET_ADDR_OP`类型的操作数，要确定并装载其内存地址。关键步骤是使用帧指针（`$fp`）结合变量在当前函数栈帧中的偏移量，计算出其实际内存地址。

下面是代码实现过程：

1. 分配寄存器并获取存放操作数的寄存器编号：调用`getReg`函数为`op->opr`分配寄存器。`op->opr`是`GET_ADDR_OP`类型操作数指向的变量或临时变量。
2. 获取当前函数的栈帧描述符：调用`findCurrFrame`函数获取当前函数的栈帧描述

符。栈帧描述符包含了函数中所有局部变量的信息，包括它们在栈上的位置。

3. 创建或获取变量描述符：使用 `createVarDes` 函数在当前栈帧中搜索或创建对应于 `op->opr` 的变量描述符 `var`。根据该函数的实现，如果该变量已经存在于栈帧中，则返回现有的描述符；否则，创建一个新的描述符。变量描述符 `var` 包含该变量在栈帧中的偏移量信息。

```
VarDes var = createVarDes(op->opr, frame);
```

4. 计算变量内存地址：使用 `addi` 加立即数指令，**指令 `addi %s, $fp, %d` 计算相对于帧指针 `$fp` 的变量地址，即帧指针+变量偏移量，并将其存储在分配的寄存器中**。栈向下增长，需要将偏移量 `var->offset` 的值取反保证地址正确。

```
fprintf(fp, "  addi %s, $fp, %d\n", regs[reg]->name, -var->offset);
```

_Todo3

要求：实现将中间代码翻译为目标代码并输入到文件部分的 `case ASSIGN_IR`。

解决思路：对于赋值语句的处理，关键是根据左侧操作数的类型（变量或间接赋值）选择不同的 MIPS 汇编指令。如下为中间代码->MIPS32 指令。

x := y -> move reg(x), reg(y)　　*x = y -> sw reg(y), 0(reg(x))

下面是代码实现过程：

1. 提取操作数：从中间代码 `curr` 中提取左侧（目标）和右侧（源）操作数。左侧操作数 `left` 表示被赋值的变量（x），而右侧操作数 `right` 表示赋值来源（y）。
2. 装载操作数至寄存器： `handleOp` 函数将 `right` 装载到寄存器 `regRight` 。
3. 根据左侧（目标）操作数的类型进行分支处理：
    - 如果 `left` 是普通变量（`VARIABLE_OP`）或临时变量（`TEMP_VAR_OP`），则通过 `getReg` 为它分配一个寄存器 `regLeft`，但不加载其当前值（因为接下来会被新值覆盖）。使用 **`move` 指令将寄存器 `regRight` 的值（右侧操作数 `right`）复制到寄存器 `regLeft` 中**。最后使用 `spillReg` 函数将可操作寄存器中的变量保存到栈上，确保即使在寄存器被其他变量重用时，变量值也得以保存。
    - 如果 `left` 是解引用类型（`GET_VAL_OP`），表示需要将值赋给一个指针指向的位置，首先为 `left->opr`（指针本身）分配寄存器 `regLeft`，并加载其指向的地址（`load` 置为 `1`）。使用 **`sw` 指令将 `right` 的值存储到 `left->opr` 指向的内存位置**。这实现了间接赋值操作，对应于中间代码 `*x = y`。

```
fprintf(fp, "  move %s, %s\n", regs[regLeft]->name, regs[regRight]->name);
```

_Todo4

要求：实现将中间代码翻译为目标代码并输入到文件部分的 `case SUB_IR`。

解决思路：参照 PLUS_IR 的实现，将其输出的汇编指令中的 add 更换为 sub 即可，关键点也是根据目标操作数的类型（变量或间接赋值）选择不同的 MIPS 汇编指令。

下面是对代码实现中根据结果变量 `left` 的类型进行分支处理的简单叙述：

- 如果 `left` 是普通变量或临时变量，则直接使用 **`sub` 指令执行减法操作**，进行寄存器的值覆盖，调用 `spillReg` 函数将可操作寄存器中的变量保存到栈上。
- 如果 `left` 是解引用类型，先计算并获取减法操作的结果存储于寄存器 `regLeft1`，将需要赋值的内存地址加载到寄存器 `regLeft2`，使用 **`sw` 指令将减法结果从 `regLeft1` 存储到 `regLeft2` 指向的内存位置**。

_Todo5

要求：实现将中间代码翻译为目标代码并输入到文件部分的 `case DIV_IR`。

解决思路：根据文档中的中间代码与 MIPS32 指令对应的示例进行代码编写。同 todo3 与 todo4，需要根据目标操作数的类型（变量或间接赋值）选择不同的 MIPS 汇编指令。如下为中间代码->MIPS32 指令。

x := y / z -> div reg(y), reg(z) mflo reg(x)

经查询资料： 在 MIPS32 架构中定义了 3 个特殊寄存器，`PC`（程序计数器）、`HI`（乘除结果高位寄存器）、`LO`（乘除结果低位寄存器）。**`div` 指令只接受被除数和除数两个寄存器作为输入，结果（商）会自动存储在 `LO` 寄存器中，余数存储在 `HI` 寄存器中，即不直接将结果存储在通用寄存器中。**使用 `mflo` 指令从 `LO` 寄存器中提取商，即将 `LO` 寄存器中的商移动到一个通用寄存器中，再继续进行其他操作。 下面是代码实现中的重要内容：

执行除法操作：使用 `div` 指令执行除法操作，其中 `regRight1` 存储被除数，`regRight2` 存储除数，除法操作结果自动存储在特殊寄存器 `LO`（商）和 `HI`（余数）。

- 如果 `left` 是普通变量或临时变量，为 `left` 分配一个寄存器 `regLeft`，使用 **`mflo` 指令从 `LO` 寄存器中提取商**，并将其存储在 `regLeft` 中，最后调用 `spillReg` 函数将可操作寄存器中的变量保存到栈上。
- 如果 `left` 是解引用类型，表示结果需要存储到一个指针指向的内存位置。则类似之前的做法，分配寄存器 `regLeft1` 后，使用 `mflo` 指令提取商，并将结果暂存于该寄存器中。 然后为 `left->opr`（指针本身）分配寄存器 `regLeft2`，加载其指向的地址，最后使用 **`sw` 指令将除法结果（商值）从 `regLeft1` 存储到 `regLeft2` 指向的内存位置**，完成间接赋值操作。

```
fprintf(fp, "  div %s, %s\n", regs[regRight1]->name, regs[regRight2]->name);
if (left->kind == VARIABLE_OP || left->kind == TEMP_VAR_OP) {
    int regLeft = getReg(left, fp, 0);
    fprintf(fp, "  mflo %s\n", regs[regLeft]->name);
    spillReg(regs[regLeft], fp); }
else if (left->kind == GET_VAL_OP) {
    int regLeft1 = getReg(left->opr, fp, 0);
    fprintf(fp, "  mflo %s\n", regs[regLeft1]->name);
    int regLeft2 = getReg(left->opr, fp, 1);
    fprintf(fp, "  sw %s, 0(%s)\n", regs[regLeft1]->name, regs[regLeft2]->name); }
```

总结

本次实验通过几个 **todo** 部分，完成 MIPS 汇编代码生成的关键模块。实验的关键点在于根据不同的指令和操作数类型，正确选择并生成合适的 MIPS 汇编代码，同时确保寄存器的有效和高效分配。虽然仅仅补充完成了代码中的 **todo** 部分，而不是自己从零进行代码构建，但是通过参考代码上下文和通读实验指导书进行完成，收获仍颇丰。