编译原理实验四

实验目标

以实验三输出的中间代码为输入，生成mips目标代码

实验步骤

代码翻译

将中间代码对应到一条或多条目标代码。

使用结构体和枚举类型对语句和寄存器进行抽象。

```
enum OpCode {
    OP_ADD, OP_SUB, OP_MUL, OP_DIV, OP_MFLO,
    OP_ADDI, OP_LI,
    OP_MOVE, OP_LW, OP_SW, OP_LA,
    OP_J, OP_JAL, OP_JR,
    OP_BEQ, OP_BNE,
    OP_BLT, OP_BGT, OP_BLE, OP_BGE,
    OP_SYSCALL,
    OP_LABEL,
    OP_NOP
};

enum Register {
    ZERO,
    AT,
    V0, V1,
    A0, A1, A2, A3,
    T0, T1, T2, T3, T4, T5, T6, T7,
    S0, S1, S2, S3, S4, S5, S6, S7,
    T8, T9,
    K0, K1,
    GP,
    SP,
    FP,
    RA
};

struct TargetCode {
    ListHead list;
    enum OpCode op;
    enum Register rd, rs, rt;
    uint32_t off, imm;
    char *label;
};
```

寄存器分配

在中间代码中，可以使用的中间变量数量不受限制。但在目标代码中，寄存器数量有限，所以需要进行寄存器分配。

我使用的方法是朴素算法。翻译一个新函数时先扫描函数体，统计变量和中间变量的数量，将他们的空间全部分配到栈空间中。在使用一个变量前，先从栈空间中加载到寄存器中，修改后再写回栈空间。这种方法只需要使用少量的寄存器，不需要但心寄存器耗尽，但会增加读写内存的次数，效率不如其他的寄存器分配算法。

```c
static ListHead* translate_binop(InterCode ir) {
    ListHead *binop_code = new_list_head();
    if (ir->code.binop.y.kind == LABEL_CONSTANT) {
        TargetCode *li = new_target_code(OP_LI, T0, 0, 0, 0, ir->code.binop.y.value, 0);
        list_add_tail(binop_code, &li->list);
    } else if (ir->code.binop.y.kind == LABEL_VARIABLE) {
        TargetCode *lw_t0 = new_target_code(OP_LW, T0, FP, 0, lookup_offset(ir->code.binop.y.name), 0, 0);
        list_add_tail(binop_code, &lw_t0->list);
    }
    if (ir->code.binop.z.kind == LABEL_CONSTANT) {
        TargetCode *li = new_target_code(OP_LI, T1, 0, 0, 0, ir->code.binop.z.value, 0);
        list_add_tail(binop_code, &li->list);
    } else if (ir->code.binop.z.kind == LABEL_VARIABLE) {
        TargetCode *lw_t1 = new_target_code(OP_LW, T1, FP, 0, lookup_offset(ir->code.binop.z.name), 0, 0);
        list_add_tail(binop_code, &lw_t1->list);
    }
    if (ir->code.binop.op == BINOP_PLUS) {
        TargetCode *add = new_target_code(OP_ADD, T2, T0, T1, 0, 0, 0);
        list_add_tail(binop_code, &add->list);
    } else if (ir->code.binop.op == BINOP_MINUS) {
        TargetCode *sub = new_target_code(OP_SUB, T2, T0, T1, 0, 0, 0);
        list_add_tail(binop_code, &sub->list);
    } else if (ir->code.binop.op == BINOP_STAR) {
        TargetCode *mul = new_target_code(OP_MUL, T2, T0, T1, 0, 0, 0);
        list_add_tail(binop_code, &mul->list);
    } else if (ir->code.binop.op == BINOP_DIV) {
        TargetCode *div = new_target_code(OP_DIV, 0, T0, T1, 0, 0, 0);
        TargetCode *mflo = new_target_code(OP_MFLO, T2, 0, 0, 0, 0, 0);
        list_add_tail(binop_code, &div->list);
        list_add_tail(binop_code, &mflo->list);
    }
    TargetCode *sw_t2 = new_target_code(OP_SW, FP, T2, 0, lookup_offset(ir->code.x.name), 0, 0);
    list_add_tail(binop_code, &sw_t2->list);
    return binop_code;
}
```

栈空间管理

在调用函数和函数被调用时需要进行栈空间管理操作。包括保存返回地址和旧栈帧指针，分配栈空间，保存参数和 上下文寄存器的值。

```c
// 函数入口
TargetCode *func_code = new_target_code(OP_LABEL, 0, 0, 0, 0, 0, ir->code.x.name);
list_add_tail(function_code, &func_code->list);

// 分配栈空间
TargetCode *addi_sp = new_target_code(OP_ADDI, SP, SP, 0, 0, -size, 0);
list_add_tail(function_code, &addi_sp->list);

// 保存返回地址和旧帧指针
TargetCode *sw_ra = new_target_code(OP_SW, SP, RA, 0, size - 4, 0, 0);
list_add_tail(function_code, &sw_ra->list);
TargetCode *sw_fp = new_target_code(OP_SW, SP, FP, 0, size - 8, 0, 0);
list_add_tail(function_code, &sw_fp->list);

// 设置新帧指针
TargetCode *addi_fp = new_target_code(OP_ADDI, FP, SP, 0, 0, size, 0);
list_add_tail(function_code, &addi_fp->list);

// 保存函数参数到栈
for (int i = A0; i <= A3; i++) {
    TargetCode *sw_ai = new_target_code(OP_SW, FP, i, 0, -8 - (i - A0 + 1) * 4, 0, 0);
    list_add_tail(function_code, &sw_ai->list);
}
```

```c
// 保存调用者保存寄存器
TargetCode *addi_sp = new_target_code(OP_ADDI, SP, SP, 0, 0, -12, 0);
list_add_tail(call_code, &addi_sp->list);
for (int i = T0; i <= T2; i++) {
    TargetCode *sw_ti = new_target_code(OP_SW, SP, i, 0, (i - T0) * 4, 0, 0);
    list_add_tail(call_code, &sw_ti->list);
}

// 传递参数
InterCode nowIR = ir->prev;
int arg_num = 0;
while (nowIR != NULL && nowIR->code.kind == CODE_LABELOP && nowIR->code.labelop.op == LABELOP_ARG) {
    arg_num++;
    nowIR = nowIR->prev;
}
if (arg_num > 4) {
    addi_sp = new_target_code(OP_ADDI, SP, SP, 0, 0, -4 * (arg_num - 4), 0);
    list_add_tail(call_code, &addi_sp->list);
}
```

```
// 调用函数
TargetCode *jal = new_target_code(OP_JAL, 0, 0, 0, 0, 0, ir->code.call.f.name);
list_add_tail(call_code, &jal->list);

// 恢复栈空间
for (int i = T0; i <= T2; i++) {
    TargetCode *lw_ti = new_target_code(OP_LW, i, SP, 0, (i - T0) * 4, 0, 0);
    list_add_tail(call_code, &lw_ti->list);
}
addi_sp = new_target_code(OP_ADDI, SP, SP, 0, 0, 12, 0);
list_add_tail(call_code, &addi_sp->list);
TargetCode *sw_v0 = new_target_code(OP_SW, FP, V0, 0, lookup_offset(ir->code.x.name), 0, 0);
list_add_tail(call_code, &sw_v0->list);
```

总结

本次实验只对翻译的正确性有要求，对代码效率没有要求。后续可以优化寄存器分配算法，先使用活跃变量分析得到变量的生存周期，再使用图染色算法进行分配。这样可以减少内存访问次数，提高代码效率。