hw13实验过程记录

参考资料

1. 虎书11.4节：图着色的实现
2. week13.pptx

实验过程

准备工作

在进行寄存器分配之前，需要获取每个函数的冲突图。由此，需要对SSA destruction后的伪arm源码重新进行CFG的构建、活跃性分析，最后根据新的Liveness Graph调用ig.h/c模块得到变量冲突图。

寄存器分配

该模块的输入为SSA destruction后的伪arm源码和冲突图，输出为变量着色表 (temp到reg的映射) 和修改好的指令，即：

```
typedef struct RA_result_ *RA_result;
struct RA_result_ {
  Temp_map coloring;
  AS_instrList il;
};
```

代码整体分为两部分：对图着色 (COL模块) 和处理溢出并修改指令 (RA模块)。

Color

> 此部分ARM寄存器和浮点寄存器一起进行。

着色的思路即不断简化度小于 k 的节点，直到无法再简化时取一个度不小于 k 的节点溢出 (将对应边也从冲突图中去掉)，然后重复以上步骤直到所有节点都被移除。

着色过程用到了下面的数据结构：

1. 着色的结果：包括着色表、溢出变量集合，还有两个辅助变量用于记录简化过程中用到的寄存器数 (以便后续确认栈大小)。

   ```
   typedef struct COL_result_ *COL_result;
   struct COL_result_ {
     Temp_map coloring;
     Temp_tempList spills;
     int maxRegId;
     int maxFloatRegId;
   };
   ```

2. 变量信息表：记录每个节点 (变量) 在冲突图中的度，还有对应的寄存器id。

```
typedef struct COL_tempInfo_ *COL_tempInfo;
struct COL_tempInfo_ {
  G_node node;
  int degree;
  int regId;
};

static TAB_table colEnv = NULL;
```

3. `simplyfyWorklist`：记录所有待简化的节点。
4. `spillWorklist`：记录所有待溢出的节点。
5. `selectStack`：记录从图中删除的简化节点的栈。

着色过程整体流程大致如下：

1. 初始化：
    1. 将所有预着色节点放入变量着色表 `coloring` 里
    2. 初始化工作表：只考虑未预着色的节点，将其中度小于 k 的节点放入 `simplyfyWorklist`，剩余节点放入 `spillWorklist`。

> **注意**：在着色完成后需要将所有工作表清空 (设为NULL)，否则会影响到下一个函数的处理。

2. 循环进行以下过程：
    1. 简化：从 `simplyfyWorklist` 取出一个节点 n，并将其压入 `selectStack` 栈中，同时需要减少 n 的所有邻接节点的度数。
        - 减少度数的操作提取到函数 `COL_decrementDegree` 中：
            1. 将节点度数减1
            2. 如果该节点度数变为 $k-1$，说明该节点之前在 `spillWorklist` 中，应该从其中删除并加入 `simplyfyWorklist` 等待简化

```
static void COL_decrementDegree(G_node m) {
  ASSERT(m, "m is NULL");
  Temp_temp temp = G_nodeInfo(m);
  if (Temp_isPrecolored(temp)) return;

  COL_tempInfo info = TAB_look(colEnv, temp);
  info->degree--;
  switch (temp->type) {
    case T_int: {
      if (info->degree == MAX_NUM_REG - 1) {
        COL_removeNode(m, &spillWorklist);
        simplyfyWorklist = G_NodeList(m, simplyfyWorklist);
      }
      break;
```

```
    }
    case T_float: {
      if (info->degree == MAX_NUM_FLOATREG - 1) {
        COL_removeNode(m, &spillWorklist);
        simplyfyWorklist = G_NodeList(m, simplyfyWorklist);
      }
      break;
    }
    default:
      ASSERT(0, "Unknown temp type");
  }
}
```

2. 溢出：从 `spillWorklist` 选取一个节点 m 从冲突图中删除，其中
    1. 此处选择节点的启发式就是选择工作表中度最大的节点
    2. 为简化，直接将其加入结果的溢出集合 `res->spill` 中 (real spill)
    3. 同样，需要调用 `COL_decrementDegree` 减少 m 的所有邻接节点的度数
3. 着色：不断从 `selectStack` 中弹出待简化的节点 n，
    1. 遍历其所有的邻接节点，按id顺序找到第一个可用的寄存器，分配给 n
    2. 将 n 到该寄存器的映射关系存储在 `res->coloring` 中
    3. 维护用到的最大寄存器id值
4. 清空工作表

RegAlloc

该模块主要是处理溢出变量，并确定栈大小。

溢出变量的处理策略参考week13的slides，预留一些寄存器专门用作溢出：

1. 对ARM寄存器，预留 `r9` 和 `r10` (callee saved)
2. 对浮点寄存器，预留 `s14` 和 `s15` (caller saved) (无需存储到栈中)

该模块的整体流程大致如下：

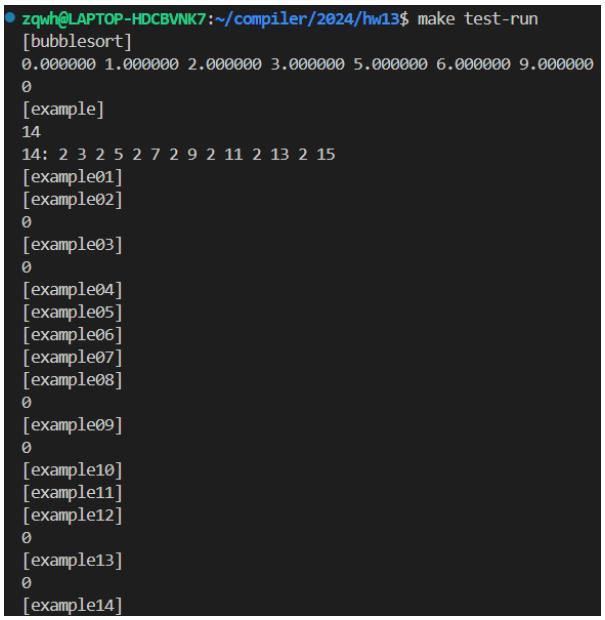
1. 计算栈大小和溢出变量对应的偏移量：
    1. 由于只需要real spill，遍历着色阶段得到的溢出集合，将对应的偏移量 (按顺序) 记录到 `spillMap` 中
    2. 计算用到的预留ARM寄存器数
    3. 计算之前简化过程中，用到的callee saved寄存器
2. 遍历指令序列，修改指令：
    1. 根据上一步的2和3，修改之前的prolog和return语句中的callee saved寄存器压栈/出栈的指令，并插入为溢出变量开辟/恢复栈顶 `sp` 的指令
    2. 对每条指令，如果src/dst变量有溢出的，根据偏移量表插入对应的 `ldr/str` 指令
    3. 对 `AS_Move` 指令，如果src和dst分配到同一个寄存器 (或者都溢出了)，删除该指令

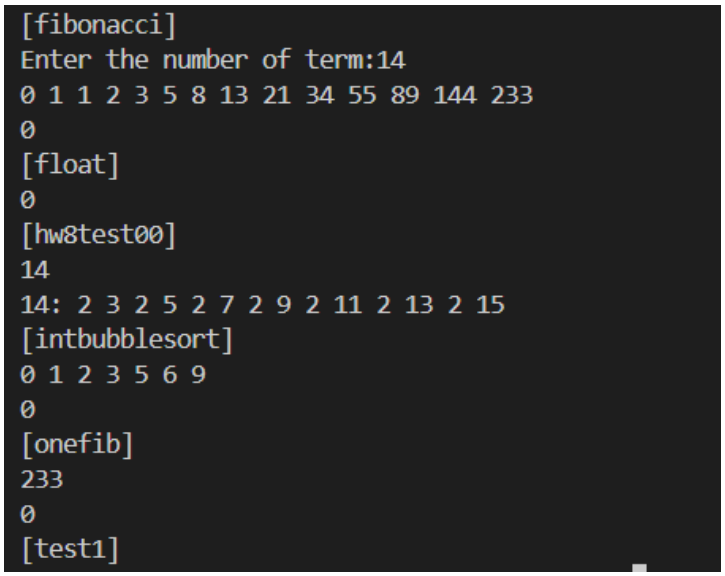
遇到的问题

1. cjump翻译成cmp+br时，br的AS_Target要完整保留 (即true和false的label都保留)，否则无法得到正确的冲突图。

测试结果

运行 `make test-run`，测试结果如下：





开发过程

git提交记录如下：