『编译技术』SysY-Mips编译器设计——目标代码Mips生成

章节目录

零. 任务目标

由生成的中间代码LLVM转化为目标代码Mips，实现简单的寄存器分配，不考虑后端优化。

一. 前置准备

Mips基本分区

data常量段

包含所有准备输出的字符串。

text代码段

全局变量：使用`GP`做偏移

函数和主函数代码段：其中变量定义，入栈时使用`FP`做偏移

关键寄存器

1. `$gp`：全局寄存器 记录全局变量
2. `$sp`：栈顶寄存器
3. `$fp`：栈帧寄存器，记录的是活动记录基地址 开局`li $fp, 0x10040000`设置栈帧寄存器
4. `$t0~$t7,$s0~$s6`：待分配寄存器
5. `$v0, $v1, $t8, $t9`：机动寄存器

Mips指南

1. `syscall`的使用，`$v0`中存储指令代码
    1. `syscall 1`：打印整数，将整数参数存储在`$a0`寄存器中，然后使用 syscall 1 来打印这个整数
    2. `syscall 4`：打印字符串，将字符串的地址存储在`$a0`寄存器中，然后使用 syscall 4 来打印这个字符串。
    3. `syscall 5`：读取整数，将整数的地址存储在`$v0`寄存器中，然后使用`syscall 5`来从用户输入中读取整数
    4. `syscall 10`：终止程序

5. `syscall 11`：打印字符，将字符的数值存入 `$a0` 寄存器中，然后使用`syscall`打印字符

2. `li $x, 0x1`：将立即数赋值给指定寄存器
3. `la $x, addr`：将地址赋值给指定寄存器，用于输出字符串
4. `move $x1, $x2`：将`$x2`中的内容赋值给`$x1`

二. 指令转化

- `cmp` 的各种比较符号完全替换成 MIPS 的`sgt,sge`等指令即可
- `br` 无条件跳转替换为`j`，有条件跳转如 `br i1 label1 label2` 可替换为 `bne $x, $zero, label1; beq $x, $zero, label2` 两条语句
- `call` 替换为 `jal`

四. MIPS 符号表

首先说明本人的内存管理非常规，`$sp,$fp`并非指栈顶栈底，而是将其视为两个栈底，函数调用的活动变量用`$sp`记录，局部变量用`$fp`记录。

不能完全复用错误处理时的符号表，因为虚拟寄存器与符号表记录的信息相比多了许多中间变量，而这些中间变量也需要被记录，则需要**在生成中间代码时重新构造一张符号表**，而构建起两张符号表的联系是必要的。

中间代码的每个虚拟变量由`Value`记录，因此我们直接使用`Value`作为中间代码符号表的单元，在`Value`上额外记录一些属性即可。

新增属性：

```
int offset; // 对于基地址，是基于Fp或Gp的偏移
bool isGp; // 标记全局Alloc的属性
bool isFp; // 标记局部Alloc的属性
bool isInReg; // 该Value是否在寄存器内
```

五. 函数调用

分为调用方和被调用方，调用函数时，此时已经使用过的寄存器要压入栈，返回时重新填入。

调用方调用函数流程

1. 将前四个参数放入`A`中
2. 将当前`FP，RA`，当前函数未释放的寄存器保存到`SP`中，并移动`SP`位置，即保存上下文现场
3. 将`FP`增长至被调用函数的栈帧处
4. 将剩余参数存入`FP`首地址的位置
5. 执行跳转 `jal xxx`
6. 返回后首先将`FP，RA`，当前函数未释放的寄存器从`SP`中取回，再将`SP`还原，即还原上下文现场
7. 将返回值转移到其他寄存器，继续运行

被调用方处理

函数开始时需要将参数全部装入当前函数的`FP`栈帧中。计算时可能会出现寄存器不够的情况，若传递的参数大于当前寄存器可用的数量，要**压栈**。

```
mipsInstructions.add(new Sw(V0, fpOffset, FP));
irInstruction.offset = fpOffset;
fpOffset += 4;
```

六. 机动寄存器

在参数传递时，若多于四个参数，此时`A0~A3`也无法作为中转寄存器，那我们规定`V0,T8,T9`作为**立即数转移**的机动寄存器，也作为**压栈**的临时寄存器，`V1`作为存储地址的机动寄存器，他们的共同特点就是使用后立即释放。

- `V1`地址机动寄存器：由于地址每次都重新计算，因此只要规定不随意使用，`V1`一个就够。

七. 相对与绝对地址

值与地址

我们尝试找到一种寻址比较易于理解的方法。

`Value`具有几个关键属性：

- `isFp，isGp`：这两个标识**仅**出现在`Alloc`的`Value`中，根据全局或局部变量设置，目的是配合`offset`设置定义的变量地址`fp/gp + offset`，对于数组而言是首地址。
- `isInReg`，`reg`：标识该`Value`的值是否在寄存器中，若在，则位于`reg`号寄存器。
- `offset`：偏移量，若`isFp/isGp`为True，表示定义的变量的相对偏移量；若为 false，意为由于寄存器不足，将临时变量存入内存中，保存的地址相对于当前栈帧`FP`的偏移量。

`offset`在这两种情况表达的含义不同，由于寄存器不足而入栈的`Value`的`offset`，指的是**Value的值存入的地址相对于栈帧的偏移**；而`alloc`的`Value`本身就是地址，`offset`是`Value`值的偏移。

首先明确，我们想要得到一个Value的**值**，一共有两种方法：

1. 从内存中取，获得所在内存的地址又分为两种方法
    1. `GP + offset`：`offset`是相对于全局基地址的偏移
    2. `FP + offset`：`offset`是相对于当前栈帧基地址的偏移
2. 从寄存器中取，此时寄存器中保存着`Value`的值

针对`store,load,getelementptr`这三条指令的转化是重点，我们默认地址都指绝对地址，只是存储方式有区别。

当`Value`存的值是一个地址时，即上述三条指令会涉及，这个地址(`Value`值)可能有两种情况。

1. 该地址基地址，即`alloc`的`Value`，这时绝对地址就是`Value`中`FP/GP + offset`。
2. 该地址是经过`elementptr`转化过的目标地址，这时通过上述的取值方法可得到地址的值。

区别在于，第一种情况只需要`Add($a, FP, offset)`，而第二种若`Value`在内存中则需要`Lw($a, offset, FP)`，`$a`为绝对地址。

绝对与相对寻址

一旦寄存器或内存中保存了地址，则一定为绝对地址。当涉及到传指针传地址时，传的应当是绝对地址，这是由于若是相对地址，则由于不同函数的`FP`不同，将无法得到正确的地址。而传指针时必然会涉及到`getelementptr`，因此我们规定`getelementptr`的`Value`一律保存绝对地址。

在本部分，第一是要正确理解值，地址，内存的概念，第二是要理解相对只是绝对的另一种描述方式，本质上寻址都是基于绝对寻址。

八. 优化方法

由于LLVM的临时变量(除去定义)用完即释放，操作数的寄存器一旦被用了一次即可释放，因此寄存器会比较够用，也会不易产生入栈读写内存的开销。然而这些应当是由于LLVM优化不够导致的，正如上一篇LLVM生成文档所言，一旦一个临时变量可以被多次使用，那么我们便不能轻易释放寄存器(但凡后面还会用到该`value`)，寄存器之间的冲突会加剧，调度与分配会更为复杂，这些在优化LLVM后应当着手考虑。