Lab8 目标代码生成

模拟器环境配置

本次实验生成代码的目标平台是RISC-V

因为在默认条件下使用`nc_gen`生成的代码无法在`BIT-MINICC`运行，并且对应的`RISC-V`模拟器`RARS0.1`没有相应的文档（据说模拟的指令集甚至不包括有乘除指令的`rv32m`），为了防止后续实验产生不愉快事件……于是将`lib/Rars.jar`替换为了GitHub上有更齐全文档的`Rars1.5`：

TheThirdOne/rars: RARS -- RISC-V Assembler and Runtime Simulator (github.com)

将`bit/minisys/minicc/simulator/RISCVSimulator.java`进行一下修改，就能按原样运行：

```
...
@Override
    public void run(String input) throws Exception {

        if(input == null) {
            String[] args = new String[0];
            rars.Launch.main(args);
            return;
        }

        String[] args = new String[1];
        args[0] = input;
        rars.Launch.main(args);

        System.out.println("8. Simulate not finished!");
    }
...
```

由于框架发生了变化，因此需要将`Mars`框架中的`Mars_PrintStr`等函数重新进行一下封装：

Environment Calls · TheThirdOne/rars Wiki (github.com)

a0: 传入参数以及返回参数，a7: 对应函数

`Mars_PrintStr`->`a7`: 4

`Mars_GetInt`->`a7`: 5

`Mars_PrintInt`->`a7`: 1

```
Mars_PrintStr:
    li a7,4
    ecall
    ret

Mars_GetInt:
    li a7,5
    ecall
    ret

Mars_PrintInt:
```

```
    li a7,1
    ecall
    ret
```

汇编标记

`.data`：存放数据

`.asciz`：相当于`.string`，声明字符串

`.text`：代码段

```
.data
str1: .asciz "Please input a number:\n"
str2: .asciz "This number's fibonacci value is :\n"

.text

main:
...RISCV代码
```

需要使用的指令

以下是本次实验主体中会使用的`RISC-V`指令：

- `add dest,src1,src2`
- `sub dest,src1,src2`
- `addi dest,src,立即数`：将立即数和对应的值放入目标寄存器
- `sw rs2, 偏移量(基址)`：将寄存器的值存入内存
- `lw rd,偏移量(基址)`：将内存的值加载到寄存器
- `la rd,符号`：将对应符号的值载入寄存器（打印字符串时需要）
- `b** rs1,rs2,label`：有条件跳转。如果满足rs1`<,>,==...`条件时，跳转到指定位置
- `jal x0,label`：无条件跳转
- `ret`：跳转到`ra`，即`x1`处所指的位置
    - 相当于`jalr x0, 0(x1)`
- `call 符号`：函数调用。由于对应真实指令的函数调用复杂，因此直接使用伪指令

RISC-V 函数调用

在开始实验前，需要研究一下`RISC-V`函数调用的约定（Calling Convention），才好以正确的形式

主函数`call`的时候，会将返回地址放入`ra`（`x1`）

子函数`ret`的时候，会转向`ra`的地址

参数传入：使用`a0~a7`

返回值：使用`a0`

`t0~t6`在子函数调用过程中会被破坏，如果需要保留这个值，在递归调用子函数的时候调用者应该保存这个值。

对于`s0~s11`，如果函数过程中需要使用，则需要保存其中原先存在的值，并且在退栈时将值还原，以让父函数的值不被破坏。

函数调用过程中特殊的寄存器：

- `ra (x1的别名)`，返回地址，在调用`ret`伪指令的时候会将此寄存器的值赋给`pc`
- `sp (x2的别名)`，栈指针
- `fp (x8的别名，也有个别名s0)`，栈帧指针

函数调用步骤：

1. `addi sp,sp,-framesize`，
2. 保存`ra`
3. 保存`s0`（`fp`）
4. 让新`s0`指向老`sp`
5. 进入函数体，在新`s0`的基址上开工
6. 还原`ra`
7. 还原老`s0`
8. 还原`sp`

其中`framesize`是本栈帧中会用到的所有栈区内存的大小（以字节为单位）

对于32位系统，在一个函数调用过程的栈帧中：

- `0(s0)`：旧`sp`
- `-4(s0)`：`ra`
- `-8(s0)`：旧`s0`
- `-12(s0)`：自定义的第一个栈区地址

和`x86`不一样的是，`RISC-V`并没有`push`、`pop`等原子指令，栈顶全靠自行移动。并且在每次函数调用过程中，栈顶指针是被“封顶”的——一个函数要用到的栈区的大小总是提前给出，先把栈顶移动后，在进行栈区内存的访问，在一次函数的内部，`sp`的值一般不再变化。（而`x86`在一个函数内如果出现`push`、`pop`，`sp`必然会上下浮动。）

```
    |         |
    |  父函数  |
    |---------|<- s0 (即fp)
    |  旧sp   |
    |---------|
    | 保存的ra |
    |---------|
    |  旧s0   |
    |---------|
    |   ...   |
    | 临时变量 |
    |   ...   |
    |---------|<- sp(一个函数内部不会变化)
    |         |
```

由于`sp`降低的栈帧要提前给出，因此必须将本次函数调用所有需要的变量大小算出来后，才可以进行函数体代码的书写。

Wzc Scan —— 寄存器分配

由于时间安排上的不允许，因此我并没有充分的时间研究`图着色`或者`线性扫描`等寄存器分配算法，并将其运用于LLVM的SSA上。

因此引入一个比最简单的`on the fly`算法效能更高，但同样简单寄存器分配算法——`Wzc Scan`。

在实现的概念上，有如下要求：

- 不对语句进行优化，例如基本块或者DAG的更改。只对LLVM-IR对应指令进行原始输出（优化只来自LLVM-IR自身）。
- 对一个函数先做扫描，收集信息后再进行目标代码生成
- 在代码生成完后，再回到函数开头及结尾，完善对函数调用的补全

具体算法

在介绍算法前，先引入几个在代码生成过程中必须使用的数据结构以及函数以及自定义的概念：

- SSA寄存器符号表：记录了每个虚拟寄存器当前对应的物理寄存器或者是地址。若在对此寄存器使用时发现其记录对应的是一个地址，则要对其进行值加载。
- `alloca`寄存器：对应于LLVM-IR中通过`alloca`操作声明的栈区变量。对于`alloca`寄存器，每一条`alloca`语句不会对应任何一条汇编语句，但是会在读入`alloca`操作时进行SSA寄存器符号表的添加，记录对应的栈区内存偏移量。
- 可用寄存器队列：即可用的物理寄存器。按照`t`、`a`、`s`依次排序，会优先使用`t`中的`t0~t6` 7个寄存器，倘若`t`用完，则使用`a`中的可用寄存器...直到`s`用完，如果`s`用完，则会产生“寄存器溢出”(spill)。注意的是，如果使用`s`系列的寄存器，需要对其进行标注，每增加一个`s`寄存器的使用，都需要在函数开头增加一个对对应寄存器的保存操作，并且在函数退出时还原其值，以满足`RISC-V`函数调用的约定。
    - 因为如果产生对`s`系列寄存器使用时，需要在整个函数的开头进行压栈操作，这是一个必然事件；但如果对`t`系列寄存器使用，保存值的情况只会出现在：
        - 当前寄存器生命周期未结束（生命周期会在下面提到），且要进行函数调用时
    - 由于上述情况可能仅在一个分支出现，仅仅是一个可能性事件，因此可在一定程度减小访存概率。（但在某些情况下，比如产生函数调用时，当前生命周期未结束的变量比较多，此时使用`s`系列寄存器可能更划算，但是为了方便起见，先不考虑这种优化）
- DAG变量引用表：在一个基本块内，每一个语句都有可能对前序的语句产生引用，这个引用表规定了对应变量最迟出现的时刻，可用通过这个对应一个寄存器在基本块内的生命周期。

具体步骤如下：

1. 对基本块进行扫描，迭代后记录每个基本块的活跃变量INL和OUTL
2. INL和OUTL中记录了逻辑寄存器，对“不来自`alloca`寄存器”的逻辑寄存器对应的SSA寄存器表进行寄存器的分配
3. 出现在每个基本块OUTL处的物理寄存器在基本块内部不能释放以及改变
4. 进入一个基本块内进行基本块内代码生成
    1. 扫描基本块DAG，找到变量引用流，在DAG变量引用表填写对应变量最后一次引用语句的行数-1的值
    2. 进行语句生成。在生成前，先查DAG变量引用表，如果当前行数大于DAG变量引用表对应寄存器的行数，则将其对应寄存器释放，放入寄存器队列中。
    3. 生成时，对于3地址指令，目的地址来自可用寄存器队列
    4. 如果存在常量，则在使用常量前需要进行立即数加载
    5. 如果对应操作数在SSA寄存器符号表中是地址，则要进行值加载
5. 如果要进行函数调用等可能改变当前运行环境的操作，则将其称作“破坏性操作”。对于这些操作，如果`t`系列或者`a`系列寄存器中有未结束生命周期的寄存器，则需要将寄存器变量压栈，将对应SSA寄存器符号表值改为地址，这样就能使得在调用完毕后引用时会重新进行此值的加载。
6. 寄存器溢出：从未结束生命周期的物理寄存器中踢出一个寄存器，保存其值并将其SSA寄存器符号表的值改为地址，这样这种行为在下次值加载时可规约为一次对“破坏性操作”的恢复

7. 因为在一次调用过程中，可能出现多次“破坏性操作”，这些破坏性操作对栈帧改变的值会被累计。最后累计的值+4即为栈帧大小。
8. 在函数开头处和结尾处分别附上栈初始化以及调用结束后的退栈操作。

一个实例

以斐波那契数列的函数做参考：

```
int fibonacci(int num){
    int res;
    if(num < 1){
        res = 0;
    }else if(num <= 2){
        res = 1;
    }else{
        res = fibonacci(num-1)+fibonacci(num-2);
    }
    return res;
}
```

它对应的LLVM-IR如下：在LLVM-IR中，每一个基本块的前方都有一个标号（即使那个标号可能不会被跳转到）

%0 : num

(1)

```
%2 = alloca i32, align 4
%3 = alloca i32, align 4
store i32 %0, i32* %2, align 4
%4 = load i32, i32* %2, align 4
%5 = icmp slt i32 %4, 1
br i1 %5, label %6, label %7
```

(2)

```
6:
store i32 0, i32* %3, align 4
br label %20
```

(3)

```
7:
%8 = load i32, i32* %2, align 4
%9 = icmp sle i32 %8, 2
br i1 %9, label %10, label %11
```

(4)

```
10:
store i32 1, i32* %3, align 4
br label %19
```

(5)

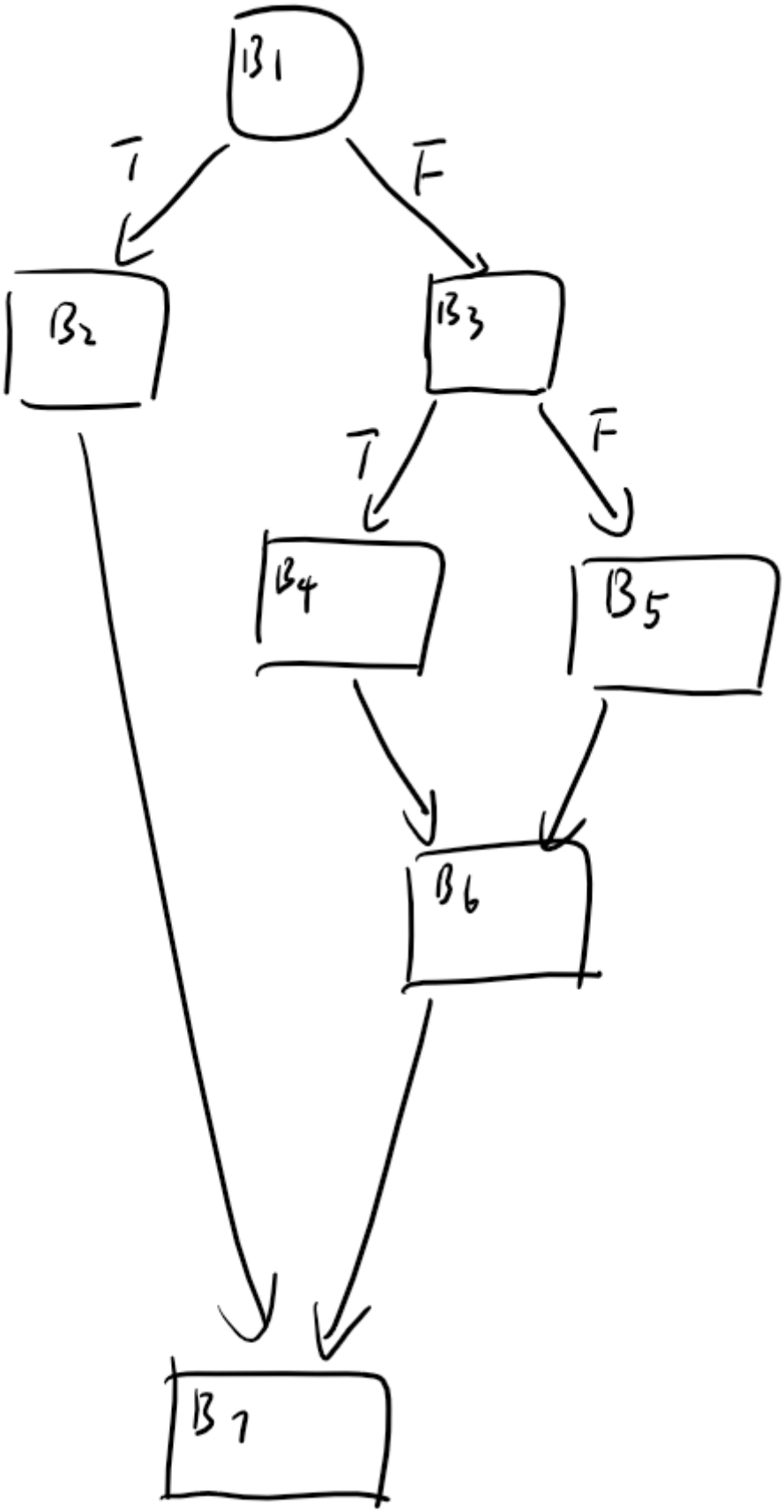
```
11:

%12 = load i32, i32* %2, align 4
%13 = sub nsw i32 %12, 1
%14 = call i32 @fibonacci(i32 %13)
%15 = load i32, i32* %2, align 4
%16 = sub nsw i32 %15, 2
%17 = call i32 @fibonacci(i32 %16)
%18 = add nsw i32 %14, %17
store i32 %18, i32* %3, align 4
br label %19
```

```
19:
  br label %20
```

```
20:
  %21 = load i32, i32* %3, align 4
  ret i32 %21
```

其基本块间的数据流如下：



按照上述算法最终生成的代码以及生成的对应法则的注释如下：

```
fibonacci:
    # 栈帧的初始化语句在最后才生成。
    addi sp,sp,-24 # 比如-28只有在得知 %14: -24(fp) 这一语句的最大栈偏移为24后，才能将28=24+4得出
```

```
    sw ra,20(sp)
    sw s0,16(sp) #如果有需要的话，这里也会保存各种功能参数
    addi s0,sp,24


    sw a0,-12(fp)
    lw t0,-12(fp) # %4: t0
    li t1,1 # 加载立即数,1:t1
    bge t0,t1,label_7


    sw x0,-16(fp)
    jal x0,label_20 #无条件跳转

    label_7:
    lw t0,-12(fp) # %8: t0
    li t1,2
    bgt t0,t1,label_11


    li t0,1
    sw t0,-16(fp)
    j label_19

    label_11:
    lw t0,-12(fp) # %12:t0
    addi t0,t0,-1 # %12在使用前已释放

    addi a0,t0,0 # mov a0,t0 函数传参
    call fibonacci # 同时释放%13，%14: a0

    lw t0,-12(fp) # %15: t0
    addi t1,t0,-2 # %16: t1

    # 破坏性操作，保存a0，%15:t0已被释放
    sw a0,-20(fp) # %14: -20(fp)

    addi a0,t1,0
    call fibonacci # %17 :a0

    # %14现在符号表对应的标记是一个地址，需要值加载
    lw t0, -20(fp)
    add t0, t0, a0 # 同时释放14, %18: t0
    sw t0,-16(fp)
    j label_19

    label_19:
    j label_20 # 来自LLVM代码的冗余

    label_20:
    #返回值，直接加载到a0
    lw a0,-16(fp)

    # 恢复堆栈

    lw ra,20(sp)
    lw s0,16(sp)
    addi sp,sp,24
```

```
    ret
```

对其中较复杂的5号基本块（对应开头标号为11）进行考察：

其DAG变量引用表如下：

虚拟寄存器	%12	%13	%14	%15	%16	%17	%18
1. %12 = load i32, i32* %2	✓						
2. %13 = sub i32 %12, 1		✓					
3. %14 = call i32 @fibonacci(i32 %13)			✓				
4. %15 = load i32, i32* %2			✓	✓			
5. %16 = sub i32 %15, 2			✓	✓	✓		
6. %17 = call i32 @fibonacci(i32 %16)			✓			✓	
7. %18 = add i32 %14, %17						✓	✓

生成1号语句时，在可用寄存器队列中获取t0，而到了2号语句时，由于`%12`对应的DAG变量引用表的项目已经为空，因此在进行减法时，可以直接获取到`%12`已经释放的`t0`作为目标寄存器。两个语句对应汇编代码如下：

```
lw t0,-12(fp) # %12:t0
addi t0,t0,-1 # %12在使用前已释放
```

在执行6号语句时，在表中，`%14`寄存器的生命未结束（`%17`是本语句保存的虚拟寄存器，生命周期刚开始），因此需要在"破坏性"操作前，对其进行压栈操作。将其对应的SSA寄存器表项的值换为`-20(fp)`和`-24(fp)`，这样在下次操作时，即可进行对其的地址的值加载。

```
# 破坏性操作，保存t1,a0, %15:t0已被释放
sw a0,-20(fp) # %14: -20(fp)

addi a0,t1,0
call fibonacci # %17 :a0
```

在执行7号语句时，由于对应的SSA寄存器表是一个地址，因此使用前对其进行了值加载：

```
# %14现在符号表对应的标记是一个地址，需要值加载
lw t0, -24(fp)
add t0, t0, a0 # 同时释放14, %18: t0
sw t0,-16(fp)
jar x0,label_19
```

在这个算法下，有两种会对源代码产生优化的例外：

- 叶函数
- `ret`指令

叶函数即不产生函数嵌套调用的函数，因为它们不会产生进一步的调用，因此不会产生“破坏性操作”。于此同时，其`ra`以及`s0`也不会涉及到改变和恢复等操作，对于这种函数，在理想情况下是不存在进入函数前的栈帧初始化和退栈的访存操作的。

对于`ret`指令，由于大多数LLVM-IR的`ret`指令是先`load`一个值，然后再对此值进行返回。但是在具体的机器上，由于返回值存在`a0`中，因此此时只需要直接将最后一次寄存器分配使用`a0`作为目的地址（无视`a0`的占用情况），并直接返回即可。

参考资料

- RISC-V-Reader-Chinese
- riscv-asm-manual/riscv-asm.md at master · riscv/riscv-asm-manual (github.com)