

实验六、目标代码生成实验

学号：1120180207

姓名：唐小娟

班级：07111801

1.实验目的

- (1) 了解编译器指令选择和寄存器分配的基本算法；
- (2) 掌握目标代码生成的相关技术和方法，设计并实现针对x86/MIPS/RISC-V的目标代码生成模块；
- (3) 掌握编译器从前端到后端各个模块的工作原理，目标代码生成模块与其他模块之间的交互过程。

2.实验内容

基于BIT-MiniCC构建目标代码生成模块，该模块能够基于中间代码选择合适的目标指令，进行寄存器分配，并生成相应平台汇编代码。

如果生成的是MIPS或者RISC-V汇编，则要求汇编代码能够在BIT-MiniCC集成的MIPS或者RISC-V模拟器中运行。需要注意的是，config.xml的最后一个阶段“ncgen”的“skip”属性配置为“false”，“target”属性设置为“mips”、“x86”或者“riscv”中的一个。

如果生成的是X86汇编，则要求使用X86汇编器生成exe文件并运行。

本次实验使用的是x86汇编，完成了函数调用、基本运算、数组访问、循环语句、条件语句等的翻译工作，通过了0_BubbleSort.c、1_Fibonacci.c、2_Prime.c的测试用例。

3.实验过程

本次实验选用x86指令集作为格式，输入文件是中间代码生成实验中的四元式。并通过正则表达式，提取出op、res、opnd1、opnd2。

将四元式中已经出现变量名的作为局部变量，保存在栈中，将未定义的临时变量“%”格式的保存到寄存器中进行运算。这里根据了临时变量的特性，一次生成，一次使用，只要在第二次出现了该符号，就代表它的生存期已经结束了，那么就可以将它对应的寄存器进行释放。

在使用局部变量的时候，先load到寄存器中，运算完成后，再保存到栈中。使用临时变量的时候，查找该符号得到寄存器的标号，使用完成后立即释放该寄存器。

具体过程如下：

3.1 符号表的建立

为了方便搜索和及时查找，本实验建立了两个表，分别是hashMapLocal和hashMapTemp，保存了局部变量的符号和临时变量的符号。局部变量：变量名、分配的栈指针指向；临时变量：变量名、分配的寄存器序号，具体如下：

```

1 //局部变量符号
2 class LocalVar{
3     String name;
4     int id; //分配的内存序号
5 }
6 //临时变量符号
7 class TempVar{
8     int id;
9     String name;
10    boolean isArray;
11    boolean scanf_f;
12 }

```

临时变量符号这里为了保证数组和输入函数的不同，增加了两个属性isArray和scanf_f。

由于使用的是四元式，没有利用上一次实验的符号表，所以在进入函数的时候，先分配固定大小的帧，并且每个变量分配固定内存大小为4个字节。遇到符号表没有出现的变量，将该变量分配内存，如果该变量已经存在了，那么就利用ebp找到该变量在内存的位置。

3.2 基本表达式的计算

这里以双目运算符“+”为例进行说明。

根据四元式的结果 (op,opnd1,opnd2,res)，①先访问opnd1，判断是否是临时变量，如果是的话，在hashMapTemp中查找得到分配的寄存器，释放对应的寄存器；如果不是的话，则查找局部变量表，得到对应的栈指针位置，并且分配寄存器将局部变量的值加载到寄存器中进行运算。运算完后需要释放该寄存器（这里是因为在x86系统中，有两个在内存的操作数不可以直接进行运算）；②再访问opnd1，与opnd1同理，根据是否是临时变量和局部变量进行相应的操作；③最后访问res，判断是否是局部变量，如果是的话，则查找，将得到的结果赋值给对应的内存值，如果是临时变量，那么就分配寄存器给临时变量值，将结果存入到该寄存器中，并将该临时变量加入到临时变量符号表里。④最后一定要释放利用的中间寄存器。具体关键代码如下：

```

1  if(icCode.opnd1.contains("%"))
2  {
3      TempVar tv = hashMapTemp.get(icCode.opnd1);
4      reg_used[tv.id] = false;
5      hashMapTemp.remove(icCode.opnd1);
6      var1 = reg_name[tv.id];
7  }
8  else if(!isNumeric(icCode.opnd1))
9  {
10     LocalVar lv = hashMapLocal.get(icCode.opnd1);
11     reg1 = RegAlloc();
12     var1 = reg_name[reg1];
13     ncs.append("mov "+var1+", dword ptr[ebp-"+lv.id+"]\n");
14 }

```

3.3 数组定义和数组访问

3.3.1 数组定义

因为要提前分配内存大小，也就是对应数组的大小，所以在生成中间四元式的时候，需要重新改进，定义数组的时候，在四元式生成代码里增加一条(arr[],a,10,_)，前者代表数组的声明，opnd1代表数组名，opnd2代表分配的数组大小。在扫描到该四元式时，得到数组的大小，分配相应的内存，并且给数组变量名存入到局部变量符号表中，栈指针指向的数组的基地址。

```

1  else if(icCode.op.equals("arr[]")){
2      //声明数组，分配空间
3      int num = Integer.parseInt(icCode.opnd2);
4      in_var += num * 4-4;
5      LocalVar lv = new LocalVar(icCode.opnd1,in_var);
6
7      hashMapLocal.put(lv.name,lv);
8      in_var += 4;
9  }

```

3.3.2 数组访问

根据四元式 ([],xx,xx,xx)，先取出opnd1，也就是数组名，找到对应的基址baseadd。再判断opnd2也就是下标是否是数字，如果是数字，利用baseadd和数字4相加得到该元素的下标；如果是局部变量名，得到局部变量的值并且×4，体现代码就是mov reg,dword ptr[ebp-xx], imul reg, 4。如果是临时变量，则找到对应存放临时变量的寄存器，并释放该寄存器取出其中的值×4得到指针值。

```

1      //判断下标是否是数字
2      if(isNumeric(icCode.opnd2)){
3          addr = baseadd + 4*Integer.parseInt(icCode.opnd2);
4          name = "dword ptr[ebp-"+Integer.toString(addr)+"]";
5      }
6      else if(!icCode.opnd2.contains("%")){
7          LocalVar lv2 = hashMapLocal.get(icCode.opnd2);
8          int reg2 = RegAlloc();
9          reg_used[reg2]=false;
10         ncs.append("mov "+reg_name[reg2]+", dword ptr[ebp-
11         "+lv2.id+"]\n");
12         ncs.append("imul "+reg_name[reg2]+", 4\n");
13         name = "dword ptr[ebp-"+baseadd+"
14         ["+reg_name[reg2]+"]";
15     }else{
16         TempVar tv2 = hashMapTemp.get(icCode.opnd2);
17         reg_used[tv2.id] = false;
18         ncs.append("imul "+reg_name[tv2.id]+", 4\n");
19         name = "dword ptr[ebp-"+baseadd+"
20         ["+reg_name[tv2.id]+"]";
21     }
22     ncs.append("lea "+reg_name[reg]+", "+name+" \n");

```

3.4 跳转语句

3.4.1 跳转条件

这里以比较运算符后的结果作为跳转依据。

根据四元式 (<=,xx,xx,%x)，因为内存里的数据不可以直接访问，所以需要中间寄存器。先看opnd1：

- 是临时变量：（1）是数组：那么需要将根据临时变量的寄存器（这里保存的是地址）得到数组变量值，也就是mov reg,[reg_temp]。（2）如果不是数组，直接取出对应的寄存器即可。

- 是变量：取出局部变量的值到寄存器中。

- 是数字：直接用值比较即可。

opnd2同理。最后释放利用的中间寄存器。

```

1      //先看opnd1
2      int reg1 = -1;
3      jmp_id = JmpLabel.jg;
4      String var1 = icCode.opnd1;
5      int reg = -1;
6      if(icCode.opnd1.contains("%")){
7          //是否是数组
8          TempVar tv = hashMapTemp.get(icCode.opnd1);
9          if(tv.IsArray){
10             //是数组
11             reg1 = RegAlloc();
12             reg_used[tv.id] = false;
13             hashMapTemp.remove(icCode.opnd1);
14             ncs.append("mov "+reg_name[reg1]+",
["+reg_name[tv.id]+"]\n");
15             var1 = reg_name[reg1];
16         }else{
17             reg_used[tv.id] = false;
18             hashMapTemp.remove(icCode.opnd1);
19             var1 = reg_name[tv.id];
20         }
21     }
22     }
23     else if(!isNumeric(icCode.opnd1)){
24         LocalVar lv = hashMapLocal.get(icCode.opnd1);
25         reg1 = RegAlloc();
26         var1 = reg_name[reg1];
27         ncs.append("mov "+var1+", dword ptr[ebp-
"+lv.id+"]\n");
28     }
29     }
30     ncs.append("cmp "+var1+", "+var2+"\n");
31     RegFree(reg1);
32     RegFree(reg2);

```

3.4.2 跳转地址

根据比较的条件后，跳转到固定标签处，这里在四元式的体现是 (jf,%x,,%x),或者 (jmp,__,%x)跳转的位置是res处。将跳转标记的%改成L标号即可，因为在四元式中，已经体现了标号的位置了。而jf具体体现值根据跳转条件的比较符号。例：如果是>则是jle。

```

1  String l_id = icCode.res.replace("%","");
2  String lable = "L" + l_id;
3  ncs.append(jmp_id.name()+" "+lable+"\n");

```

3.4.3 标签语句

根据四元式中的 (label,__,%x)可以得到该位置的具体标号值，只需要将%改成L，增加语句L(x):即可。

```

1  String l_id = icCode.res.replace("%","");
2  String lable = "L" + l_id;
3  ncs.append(lable+": \n");

```

3.5 函数

3.5.1 函数声明

因为在四元式中已经有了函数模块的划分了，体现为（proc 函数名）和（endp 函数名）所以只需要根据四元式的标记，就可以知道函数的开头和结尾了。因为x86的汇编需要对函数的调用有一定的初始化。在函数开头时，就增加一定的初始化代码，分配固定大小的帧，并将寄存器压入栈中，当遇到ret时，需要将计算的结果返回到eax中。

```
1 //函数开头
2 if(icCode.op.equals("proc")) {
3     ncs.append("\n"+icCode.res+" proc C\n");
4     ncs.append("push ebp"+"n");
5     ncs.append("mov ebp,esp"+"n");
6     //分配固定栈大小
7     ncs.append("sub esp,160\n" + "push ebx\n" +
8         "push ecx\n" +
9         "push edx\n" +
10        "push esi\n" +
11        "push edi\n"+
12        "\nlea edi,[ebp-160]\nmov ecx,40\nmov eax,0ccccccc\nrep
    stos dword ptr[edi]\n\n");
13     in_var = 4;
14     pop_var = 8;
15 }
16 //函数结束
17 else if(icCode.op.equals("endp")){
18     ncs.append(icCode.res+" endp \n");
19 }
20 //最后的恢复
21 else if(icCode.op.equals("ret")){
22     if(!icCode.opnd1.equals('_')){
23         if(isNumeric(icCode.opnd1)){
24             ncs.append("mov eax," + icCode.opnd1+"\n");
25         }
26         else{
27             LocalVar lv = hashMapLocal.get(icCode.opnd1);
28             ncs.append("mov eax, dword ptr[ebp-"+lv.id+"]\n");
29         }
30     }
31     ncs.append("\n" +
32         "pop edi\n" +
33         "pop esi\n" +
34         "pop edx\n" +
35         "pop ecx\n" +
36         "pop ebx\nmov esp,ebp\n" +
37         "pop ebp\n"+"ret\n");
38 }
```

3.4.2 函数调用

因为四元式中，函数调用是采用先压入参数的方式，在函数体开头如果有相应的参数则pop出来。所以先讨论push运算符。

(1) 压入的如果是字符串，需要在数据区分配相应的值，由于该实验是一遍扫描，我单独把data区拿出来，如果遇到字符串则将该字符串增添进去。并且要给相应的传递参数的pop_var数量+=4，方便得到在栈中的相对位置[ebp-pop_var]。

(2) 如果是数字，直接压入，pop_var+=4。

(3) 如果是变量，找到变量对应的内存地址，直接压入，pop_var+=4。

(4) 如果是临时变量，如果是数组，则需要压入该临时变量对应地址的对应值，如果不是则直接压入寄存器的值即可，pop_var+=4。

```
1  if(icCode.res.charAt(0)==' '){
2      //说明压入是字符串
3      //创建一部分数据区
4      pop_var += 4;
5      String name = "_string"+stringId;
6      if(icCode.res.contains("\n")){
7          icCode.res = icCode.res.replace("\n","");
8          if(!icCode.res.equals("")){
9              ncs_data.append(name+"    db    "+icCode.res+",0ah,0\n");
10         }
11         else{
12             ncs_data.append(name+"    db    0ah,0\n");
13         }
14     }
15     else{
16         ncs_data.append(name+"    db    "+icCode.res+",0\n");
17     }
18     stringId ++;
19     ncs.append("push offset "+name+"\n");
20 }
21 else if(isNumeric(icCode.res)){
22     //压入数字
23     ncs.append("push "+icCode.res+"\n");
24     pop_var += 4;
25 }
26 else if(!icCode.res.contains("%")){
27     LocalVar lv = hashMapLocal.get(icCode.res);
28     //hashMapLocal.put(lv.name,lv);
29     ncs.append("push dword ptr [ebp-"+lv.id+"]\n");
30     pop_var += 4;
31 }else{
32     TempVar tv = hashMapTemp.get(icCode.res);
33     reg_used[tv.id] = false;
34     if(tv.IsArray){
35         ncs.append("push dword ptr["+reg_name[tv.id]+"]\n");
36     }else{
37         ncs.append("push "+reg_name[tv.id]+\n");
38     }
39     pop_var += 4;
40 }
```

在遇到call运算符时，因为Mars_GetInt等这些是特殊的函数，需要有特殊处理，在遇到调用结果有返回值的时候，还需要给返回值进行赋值，并调用完毕后需要对堆栈进行平衡处理。因为四元式的逻辑，函数返回值都是临时变量，之后才会给具体的局部变量赋值。这里因为scanf函数的特殊，需要提前压入地址，输入字符后，得到相应需要的值，该情况需要单独处理。关键如下：

```

1 ncs.append("call "+icCode.opnd1+"\n");
2 reg_used[0] = true;
3 int reg = RegAlloc();
4 TempVar tv = new TempVar(icCode.res,reg);
5 hashMapTemp.put(tv.name,tv);
6 reg_used[0] = false;
7 ncs.append("mov "+reg_name[reg]+", eax\n");
8 ncs.append("add esp, "+(pop_var-8)+"\n");
9 pop_var = 8;

```

在遇到pop操作是，因为会有参数的传递问题，则需要把对应的参数值赋值给当前函数的局部变量处。具体如下：

```

1 //分配内存
2 LocalVar localVar = new LocalVar(icCode.res,in_var);
3 hashMapLocal.put(icCode.res,localVar);
4 //把参数传入
5 int reg = RegAlloc();
6 reg_used[reg] = false;
7 ncs.append("mov "+reg_name[reg]+", dword ptr[ebp-"+pop_var+"]\n");
8 ncs.append("mov dword ptr[ebp-"+in_var+"],"+reg_name[reg]+" \n");
9 in_var += 4;

```

4.运行结果

4.1 利用0_BubbleSort.c测试

4.1.1 数据区部分代码

```

.data
InMsg db '%d',0
IntPrinMsg db '%d',0ah,0
_string1 db "please input ten int number for bubble sort:",0ah,0
_string2 db "before bubble sort:",0ah,0
_string3 db 0ah,0
_string4 db "after bubble sort:",0ah,0

```

4.1.2 循环体部分代码

```

lea ebx, dword ptr[ebp-40][ecx]
mov ecx, dword ptr[ebx]
mov dword ptr[ebp-56], ecx
mov ecx, dword ptr[ebp-52]
imul ecx, 4
lea ebx, dword ptr[ebp-40][ecx]
mov ecx, dword ptr[ebp-52]
add ecx, 1
mov edx, ecx
imul edx, 4
lea ecx, dword ptr[ebp-40][edx]
mov edx, dword ptr[ecx]
mov dword ptr[ebx], edx
mov ebx, dword ptr[ebp-52]
add ebx, 1
mov ecx, ebx
imul ecx, 4
lea ebx, dword ptr[ebp-40][ecx]
mov ecx, dword ptr[ebp-56]
mov dword ptr[ebx], ecx
L28:
mov ebx, dword ptr[ebp-52]
mov dword ptr[ebp-52], ebx

```

4.1.3 执行结果截图

```

please input ten int number for bubble sort:
4
2
2
4
5
1
6
7
3
5
before bubble sort:
4
2
2
4
5
1
6
7
3
5
after bubble sort:
1
2
2
3
4
4
5
5
6
7
请按任意键继续. . .

```


4.2 利用1_Fibonacci.c测试

4.2.1 数据区部分代码

```
.386
.model flat,stdcall
option casemap:none
includelib msvcrt.lib
printf    PROTO    C:dword,:vararg
scanf     PROTO    C:dword,:vararg

.data
InMsg  db    '%d',0
IntPrinMsg  db    '%d',0ah,0
_string1  db    "Please input a number:",0ah,0
_string2  db    "This number's fibonacci value is :",0ah,0

.code
```

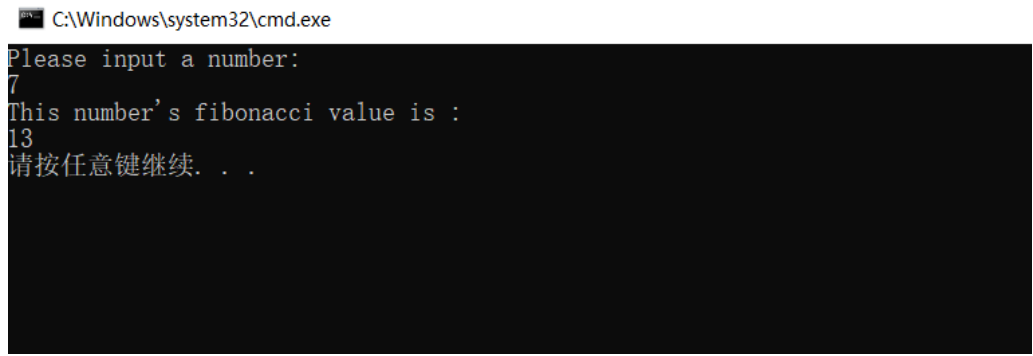
4.2.2 函数体部分代码

```
fibonacci proc C
push ebp
mov ebp,esp
sub esp,160
push ebx
push ecx
push edx
push esi
push edi

lea edi,[ebp-160]
mov ecx,40
mov eax,0cccccccch
rep stos dword ptr[edi]

mov eax, dword ptr[ebp+8]
mov dword ptr[ebp-4],eax
mov eax, dword ptr[ebp-4]
cmp eax, 1
jge L2
mov dword ptr[ebp-8], 0
jmp L3
```

4.2.3 执行结果截图



```
C:\Windows\system32\cmd.exe
Please input a number:
7
This number's fibonacci value is :
13
请按任意键继续. . .
```

4.3 利用2_Prime.c测试

4.3.1 数据区部分代码

```
.386
.model flat,stdcall
option casemap:none
includelib msvcrt.lib
printf PROTO C:dword,:vararg
scanf PROTO C:dword,:vararg

.data
InMsg db '%d',0
IntPrinMsg db '%d',0ah,0
_string1 db "Please input a number:",0ah,0
_string2 db "The number of prime numbers within n is:",0ah,0
```

4.3.2 函数体部分代码

```

mov eax, dword ptr[ebp-16]
mov ebx, dword ptr[ebp-20]
idiv ebx
mov eax, edx
cmp eax, 0
jne L10
mov dword ptr[ebp-12], 0
jmp L5
L10:
mov eax, dword ptr[ebp-20]
mov dword ptr [ebp-20], eax
add dword ptr [ebp-20],1
jmp L4
L5:
mov eax, dword ptr[ebp-12]
cmp eax, 1
jne L13
mov eax, dword ptr[ebp-8]
mov dword ptr [ebp-8], eax
add dword ptr [ebp-8],1
push dword ptr [ebp-16]
push offset IntPrinMsg

```

4.3.3 执行结果截图

```

C:\Windows\system32\cmd.exe
Please input a number:
6
2
3
5
The number of prime numbers within n is:
3
请按任意键继续. . .

```

由上可知，上述实验都可生成x86的代码，并且能在目标平台上运行成功。

5.实验心得

在这次实验中，完成了目标代码生成部分，虽然寄存器分配的效率不高，但基本可以生成目标代码。开始生成1和2测试样例的代码时，不需要考虑数组，实现起来还是比较容易的，但是在数组这里花费了较多的时间，一方面需要修改中间代码部分，另一方面因为数组在赋值的时候需要取出内存，和一般的局部变量有所区别。除此之外，在这次实验中也遇到了一些困难，比如scanf的函数特殊性，它需要提前压入需要赋值的内存，但是在老师给的测试代码中，赋值的值作为返回值，生成四元式的时候，后面才会出现该变量。所以需要提前给临时变量%x做个标记，并且把它的id值设置为栈指针的值而不是寄存器标号。还有在分配寄存器，因为释放的问题，导致了寄存器在生命周期前就释放了或者很久未释放，该问题主要体现在j++表达式中，因为四元式对j++生成的值是作为临时变量的，但是该临时变量可以不用第二次出现，这样该变量其实已经是死亡的了，需要进行回收，为了方便，我把四元式改成了

(+,1,j,j)直接对j进赋值。总之，在完成该实验中，中间代码的效率和表达方式一定程度上决定了目标代码生成的复杂度。每一个环节都十分紧凑，不可分割。

一个简单的编译器终于实现了，虽然在效率上和复杂度上都有一定的缺陷，但是在这个过程中，我明白了代码的运行后面的机理，对如何写出更优雅的代码有了更深的认识，尤其是汇编底层，以及各个寄存器的分配效果，都让我对计算机的本质有了更进一步的了解。同时，这个实验将近上万行的代码，极大的提高了我的编程能力和解决问题的能力。

感谢计卫星老师的教诲和布置的实验作业，给了我们挑战自己提高自己的机会。也十分感谢实验过程以来同学们对我的帮助以及一起探讨问题的经历。相信在今后的学习中，这次的实践经历会不断指导我走得更远。