**中间表示的代码生成**

先说总体工作打算吧，先进行规划中间表示的具体数据结构，随后，针对每个语句节点进行代码生成工作。也就是说，这篇文档主要描述的是中间表示的规划设计以及由前端语法树生成中间表示的工作。

对于中间代码生成工作是否和语义分析放在一趟里面的问题。其实吧，理论上来说，如果能够混合在语义分析的遍历过程中是最好的，因为这会少很多代码。

但是问题一样存在，本来expr的语义分析上上下下2k行，如果再将中间代码生成放在里面，单个文件里面内容太多了。Declaration里面虽然东西多，但是真的需要考虑的无非是声明的变量的size而已，实际上中间代码生成并不会放进去太多内容，但是expr里面则不同。

目标是什么？

没必要完全对标LLVM，因为这是一个专门为了C编译器设计的框架，没必要兼容更多的东西，一些语法糖同样无关紧要。但是，中间表示的设计，需要尽可能的能让一些主流的编译优化的算法能够运行起来，而不出现一些因为中间表示设计的方案问题，引起的大规模的重构等等问题。

首先，因为源语言和目标平台是基本确定的（当然你非要说什么32位和64位的差别，我也没办法咯），所以完全可以针对性的，比如指令合法化（就是检测IR的指令是否能够直接翻译成目标平台的指令，不合法就用比如说多条目标平台指令来代替一条这样子的办法来解决掉）这样的pass本身可以在针对性的IR设计中去消除的，另外，由于设计目的是在后端使用gcc的asmer和linker，那么codeemission这样的事情，由于本编译器的设计实现，直接就没有必要了。因此，虽然可以参考LLVM的后端设计，也可以留下接口，但是，没必要太苛刻的去完全“学习”。

其次，设计的可扩展性。因为确实要赶工期了。害。这么说吧，如果可以，那么最好在IR之后，直接就可以不经过任何优化，就翻译成目标平台代码，然后通过模块化的各种优化pass，当成插件一样，去进行优化。当然这是最最理想的情况了。

**中间表示的代码表**

如果说前端核心是抽象语法树，那前端到后端的核心部件就是中间表示。

但是到底用那些中间表示的代码，这是个问题。

先说整体框架，我的打算确实是类似于LLVM的，就是module到function到BB到instruction。但是图的表示也依然需要。什么Uddu链也都是需要的。

整体的中间表示的语句，我也会一定程度的参考的，但是并不完全等同，起码会进行一定的删减。

总之，对于AST到IR的翻译，我认为先构建起中间表示才是最重要的。

* 支持的各个操作的表示

中间表示的操作选择相当复杂，从理论和直观来说，由于目前我的编译器的目标平台是x86，所以完全可以针对x86的模型去做，但是中间表示的设计又要求其本身尽可能泛用性，很有可能多条IR指令对应一条x86或者一条IR需要多条x86去翻译。

在做这个事情的时候，我不得不参照一下LLVM，这也是没办法的事情，我读书少。但是又有所修改，比如说对于switch，indirectbr这样的更加偏向于高级语言的中间表示，我选择了删除——尽管在理论上而言，这存在优化空间，但是并不利于我的编译器的简单实现。

1/load/store

2/int类型的运算，加减乘除

3/int类型的位运算

4/real类型的运算（没有位运算这种）

5/控制转移指令，跳转，包括无条件跳转和条件跳转，functioncallret等等

6/内存分配

7/其他

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | Dst | Src1 | Src2 |
| Ret | Void/除了array之外的任何类型，因为很显然，C中的struct也是可以返回的。 | None | None |
| Br | None | Cond（reg类型）（条件跳转情况） | Label类型（如果没有就加label） |
|  | None | Label类型（无条件跳转） | None |
| call | None | Func类型（其中的参数压栈问题需要处理） | None |
| Recv | 寄存器类型 | 寄存器类型（但是是固定绑定的寄存器） |  |
| Binaryoperation（直接抄的，没啥好说的，大家都得有） | | | |
| Add | 寄存器类型（int） | 立即数（int）或者寄存器类型 | 立即数（int）或者寄存器类型 |
| Fadd | 寄存器类型（real） | 立即数（real）或者寄存器类型 | 立即数（real）或者寄存器类型 |
| Sub | 寄存器类型（int） | 立即数（int）或者寄存器类型 | 立即数（int）或者寄存器类型 |
| Fsub | 寄存器类型（real） | 立即数（real）或者寄存器类型 | 立即数（real）或者寄存器类型 |
| Mul | 寄存器类型（int） | 立即数（int）或者寄存器类型 | 立即数（int）或者寄存器类型 |
| Fmul | 寄存器类型（real） | 立即数（real）或者寄存器类型 | 立即数（real）或者寄存器类型 |
| Udiv | 寄存器类型（int） | 立即数（int）或者寄存器类型 | 立即数（int）或者寄存器类型 |
| Sdiv | 寄存器类型（int） | 立即数（int）或者寄存器类型 | 立即数（int）或者寄存器类型 |
| Fdiv | 寄存器类型（real） | 立即数（real）或者寄存器类型 | 立即数（real）或者寄存器类型 |
| Urem | 寄存器类型（int） | 立即数（int）或者寄存器类型 | 立即数（int）或者寄存器类型 |
| Srem | 寄存器类型（int） | 立即数（int）或者寄存器类型 | 立即数（int）或者寄存器类型 |
| Frem | 寄存器类型（real） | 立即数（real）或者寄存器类型 | 立即数（real）或者寄存器类型 |
| Bitwise operation | | | |
| Shl（逻辑左移位） |  |  |  |
| Shr（逻辑右移位） |  |  |  |
| Sal（算数左移位） |  |  |  |
| Sar（算数右移位） |  |  |  |
| rol（循环左移位） |  |  |  |
| Ror（循环右移位） |  |  |  |
| Rcl（带进位位循环左移位） |  |  |  |
| Rcr（带进位位循环右移位） |  |  |  |
| And |  |  |  |
| Or |  |  |  |
| Xor |  |  |  |
| Not（按位取反） |  |  |  |
| Fnot（LLVM中有个fneg） |  |  |  |
| 存储 | | | |
| Alloca |  |  |  |
| Load |  |  |  |
| Store |  |  |  |
| Conversion（主要是用于类型的转换） | | | |
| Trunc（整数类型截断） |  |  |  |
| Zext（零扩展） |  |  |  |
| Sext（符号扩展） |  |  |  |
| Ftrunc（float类型截断） |  |  |  |
| Fext（float类型扩展） |  |  |  |
| Ftoui（float转为无符号int） |  |  |  |
| Ftosi（float转为符号int） |  |  |  |
| Uitof（无符号int转为float） |  |  |  |
| Sitof（符号int转为float） |  |  |  |
| 其他 | | | |
| Icmp |  |  |  |
| Fcmp |  |  |  |
| Phi |  |  |  |

这里面，我直接用的栈帧模型。

在转换的过程中，LLVM还有bitcast以及ptr和int的转换。但是，我看了一下，要么可以直接拿来用（也就是说他的位值不变）要么可以通过零扩展或者符号扩展进行转换，我不觉得这需要特意拿一个指令去做。而是应当写在代码逻辑中完成.

* 中间表示的寄存器和基本类型

LLVM拥有无限多的寄存器，对于表达式的翻译，其中间结果，肯定是存放在寄存器中的。而上面的想法所不同的是，对于内存变量，LLVM同样将他们视作寄存器。而在后面的pass中进行内存布局的确定。换句话说，那些可以确定的变量是Static Alloca，而中间结果，则是Dynamic Alloca。

由于死变量消除等等问题，在中间表示不直接使用内存分配，而在经过优化之后再确定如何使用，我认为是非常合理的。否则会造成内存空间的浪费。所以就这样了。

这里面需要分配寄存器标号。

对于寄存器占用的大小，我反正直接用64bit和32bit两种寄存器，一方面，无论是longdouble还是longlongint都是64位表示的，这是最大的数据量了，另一方面，考虑的目标平台是x86-64，基本上64位和32位寄存器能够表达除了complex的绝大多数东西了，而且complex其实就是一个长度为2的数组，我也懒得说。就这么定了。

问，如何保证能够在目标平台上跑，因为目标平台只有一种长度可能，所以这里面依然需要合法化的步骤。如果64位跑到32位的，那么需要拆分计算，如果32位跑到64位的，那么需要扩展。

1. **存储绑定**

存储绑定也就是前面说的寄存器无限分配的设计方面。

一个变量需要给定其存储单元。

­­方法一：变量分为全局变量，全局栈变量，栈变量，栈静态变量。

全局变量和静态存储类的变量分配固定可重定位地址，或者是一个相对于基址寄存器的偏移。

栈变量则分配为相对于栈帧的地址。

也就是说，给出当前栈帧的相对地址即可。

方法二：也就是上面说的LLVM所使用的方法

给他们分配名义的符号寄存器，这种名义寄存器本身是无限的，并且可以按照累加方式分配。不过一般不用于数组或者聚合类型。

使用这种方法需要配合后续的图着色寄存器分配算法等算法，在IR到目标指令的过程中指定寄存器。

另一个问题在于类型系统，LLVM存在更多的类型，但是就C这门语言的翻译来说，所要求的类型并没有那么夸张和复杂，~~我打算直接按照占用内存大小来区分。~~我对此的看法是，对于非聚合类型的变量可以用寄存器的方法，但是聚合类型的还是在栈上面分配吧。

对此，在实现中，对于存储位置的翻译主要有两种，一种是存储在静态存储区的，主要包括全局变量以及声明了static的变量，另一种是分配在栈上面的，对于分配在堆上面的部分，其管理并不是我们需要考虑的，还有就是thread local的部分，我认真查看了一下，两者在gcc中生成的汇编代码的区别在于，当使用了thread local的时候，其不再是data段，而是下面这个

|  |
| --- |
| .section .tdata,"awT",@progbits |

而在使用到了该线程变量的时候，

在gcc中我生成的代码变成了

|  |
| --- |
| %fs:t@tpoff |

这里面t是变量名字，前面的fs是fs段，查询得知，fs段寄存器里面存放了一堆和线程相关的东西，其中包括线程堆栈顶部的指针，这句话的意思就很容易理解，就是从线程堆栈顶部找到这个变量。

但总体而言，他还是被当做global来处理，只不过在最终生成代码的时候需要参考一下这个字段，结果存在一些差异罢了。

1. 还有就是**边界对齐**。

边界对齐的方法，对于普通变量，既可以按照声明的顺序去分配他们。也可以先分配64位的变量，再分配32位的，以此类推。或者先分配最小的，再分配最大的。

对于较大的数据结构，比如说数组。一个问题是，立即寻址的长度位数是有限的，通常达不到32位（我要是没记错，MIPS是20位立即数好像），有可能数组的长度太长了。一种方法是，使用一个指针指向其位置，再使用加法来进行偏移寻址。

1. **寄存器使用的优化**

不再简单的使用的时候从内存取东西到寄存器，算好立即存储。

一种改进是追踪每个寄存器的内容，如果已经在合适类型的寄存器中，不再冗余的取它，而是直接使用，如果类型不符，但是在某个寄存器，那么可以进行类型传送，比如浮点到整数的传送指令。

如果用完了寄存器，可以选择一个新的（就是说，存储一个值到内存，然后空出来一个寄存器，就可以了。

不是算好一个值就立即存储，而是推迟到基本块结束时或者寄存器用完的时候再存储。

对于跨基本块的寄存器分配，如果只有一个前驱块，那么前驱的出口寄存器状态就是后继的入口状态。多个前驱可以取交集。

更多的可以考虑全局寄存器分配算法。

1. **中间表示的形式以及我打算采用的形式**

四元式：不想解释

三元式：相比于四元式，结果并不显式的给出。而是使用对于该三元式的标号来使用其结果

树：不想解释，由四元式生成树应当是很简单很基本的事情。

DAG：在树中，假设同一个变量被使用，会使用两个叶子结点。但是在DAG中，则是两个引用指针即可。通常来说，这被用于基本块之内。

前缀波兰表示：也就是树的遍历形式而已，其实质并没有太大的差别。

关于我个人对于中间表示想要采用的形式：

首先，针对于源语言翻译成中间表示来说，肯定是下面描述的这种线性结构更为合适的。但是在这一步之后，我认为中间表示应当在后面的步骤中被转换为DAG形式。这也是LLVM所实现的。

* 中间表示的modulefunctionBBinstruction结构

本来就这四个层次的结构没什么好说的，但是有那么几点，所以拿来说

首先是BB，这里面需要构建每个BB（其实也有function来做的）的DAG，这毫无疑问。而经过指令选择之后，需要将BB（或function）下的ins链转化为DAG图，这个数据结构如何处理？也就是说，instruction结构如何能在表示指令链条的同时表示DAG图。我个人其实比较喜欢function来做DAG。因为这样可以有全局的指令优化空间。问，这样一来BB这部分怎么处理呢？

其次是，IR的表示，和经过指令选择之后，实现的目标平台代码之间的兼容性，因为我才懒得再去做一个数据结构来表示MachineInst和MCInst呢，自找麻烦是不是，那么这样一个兼容从IR到目标平台代码的数据结构和表示，设计起来，也不算简单。

其实就我的看法，说到底，无非这两样，

1指令类型（以及对应的操作数的数量，0操作数，1操作数，2操作数，3操作数）这个四元组最经典的东西，虽然我不会用就是了。

2，指令的操作数从哪里来，分别有立即数（也就是常量了），寄存器，内存位置，指令位置（比如跳转之类的）这四种。

1. 内存位置这个，只有loadstore这两个操作可以使用，并且其操作数数量是相同且确定的，因此，尽管x86的寻址方式复杂，却可以在IR生成的时候就写在里面，并且能够封装好。

另外，考虑到上面说的使用全局变量和局部变量这两种类型，而且还有thread段，因此，在内存位置中，又细分为，全局静态数据段，thread段，栈区域（相对于ebp的偏移，64位使用signed long int类型，而32位则使用signed int类型。）（目前我还不知道是否存在别的段需要考虑，但是大不了再加嘛）

对于这部分的判定，则是根据符号表来进行。

这里需要注意一点，关于linkage，首先是解析语法树的时候，type向量里面存放了修饰类型，包括了linkage的设定，但是在declarator的解析过程中，其又被具体指定为external internal和none三种类型，这是不矛盾的，并且在这里，都需要考虑进来（当然理论上使用define和declare两个字段应该也行吧）。对于声明了extern的，仅仅只是声明，而不是定义，但这个符号本身会有一个external的linkage。因此，在file作用域下面，声明了external的，在这个文件中是不需要考虑的。当然，block作用域下面则不同，虽然声明了extern，但是他可能是internal的linkage的，所以这两个字段都是有用的。

对于这三类符号的判定。

首先上面说了，如果是file作用域声明了extern或者extern thread local的，丢掉不管。

其次，函数不管他，因为这是后面的代码段的部分。

第三，剩下的部分，无论是哪个作用域，只要声明了thread local，那么就是thread存储位置，而且受到相关的初始化约束。

第四，在file作用域下面的，都是全局静态数据区。

第五，在block作用域下面，具有static声明的，是当做全局静态数据区的内容的

第六，在block作用域下面，具有extern声明的，这玩意吧，还得分情况，如果在外面的符号表中，找到了在外面定义的同名符号，那就和前一个一样，如果找不到，说明是在其他模块定义的。但是考虑到实际的实现，我让同一个符号表项，插入了不同的符号表，所以两者指向同一个符号表项内容，因此，首先看是否有已经存在的存储类型，有的话，就是这种情况，如果没有，说明不存在prior的符号，所以，在这里碰见extern，一定是别的文件里面的。另外，这里不能直接返回，因为是对declarator列表里面的东西都需要做一个查询。

另外，在静态存储类型中，别的都可以不说，但是，如果存储了指针类型，而该指针类型指向某个代码段的地方，比如说，函数指针，在这种情况下的处理，同样需要考虑进去，来看看gcc的做法（可以看func\_test\_07.c）

|  |
| --- |
| int a(void){      return 1;  }  int b(void){      return 2;  }  int (\*f[2])(void)={a,b}; |

其编译出来的结果是这样的（我是64位的，所以是quad）

|  |
| --- |
| f:  .quad a  .quad b |

同样的，在该测例里面，增加了结构体中的函数指针，也即是说，在某个初始化的对象中，可能并不完全都是函数指针。

对于这个的处理办法：

没办法了，只能使用一个值向量来记录了，向量的每个元素都需要记录下其类型是普通的填值还是指向某个函数的函数指针。

1. 立即数的常量，那没啥说的，直接写就好了。我在语义分析折腾了半天的常量表达式，难道是用来看的嘛（但是仍然要考虑常量折叠）
2. 指令位置，也就是存放在代码段里面的东西，但是，其在编译阶段不确定具体的位置，而是在链接阶段确定，一般用于跳转到某个标号那里这种东西呢，一定存在于转移指令（包括无条件转移和条件转移指令对吧。所以在编译阶段，是用标号所唯一确定的，使用指针即可。
3. 其他地方都应当而且必须是寄存器。在这里，由于铁了心要弄ssa，所以寄存器（因为无限数量嘛），只会被赋值一次，所以，也其实没那么麻烦就是了。

上述的分类是按照所处位置来进行考虑的，而实际上，无论是静态还是其他类型，都可以再进行细分。

* 目标平台寄存器和可选指令集

这里面因为平台的32和64位的不同，所以依然要实现指令选择。（叹气）

首先，无论如何都要感谢不知道叫什么的老哥，对x86-64指令集的整理。我把链接贴出来哈：

<http://ref.x86asm.net/#column_op>

这里有较为详细的x86 32位及64位指令集的汇总整理——总不能再为了这个项目去翻intel手册吧。

当然，无论如何我都会为32和64位的代码表使用机器而不是人工方式来进行整理——2k多个代码，人工整理，那我岂不是智障。

* IEEE-754-2008

在中间表示的类型中，对于浮点数的类型表示，参照IEEE 754 2008的描述进行。但是相比于普通学到的754标准，确实有点东西哈，我简单描述一下，网上参考资料也不少，反正我是直接找了一个参考资料的，懒得详细说了。

<https://pan.baidu.com/s/10soDctgCJ84MDs3PyhJBcw?pwd=lzku>

但对于浮点数的处理，还确实是要用到这个的。

* X86 amx和mmx类型

既然是要做x86，而且没说不考虑向量矩阵运算，这两个类型也是需要考虑的，当然如果是其他目标架构，理论上可以就不管它。

* 函数调用和函数调用约定

函数调用，我认为需要提一下的，一个是plt和got，这在对应的md文档中写了，举的是linux下plt和got的例子。另一个则是动态链和静态链，这个概念本身是相对于前端语言转换成中间语言的过程的，因此放在这边说。

动态链是直接调用当前活动栈帧的一个调用链，指向上一个调用者，我认为没什么好说的，看过C函数调用的都明白有一个指向上一个栈帧的指针。而静态链，其目的是用于解析非局部符号表。

先说说前提，首先静态链本身针对的是能够在函数中嵌套函数定义的语言，C显然不是这样一种语言，因此，如果只是想看C编译器怎么搞，这部分别管了。

静态链指向的是其直接外层调用者，也就是说，自递归的静态链并不是指向自己，而是指向自己外面那一层。因为其目的是找非局部符号，在这样一个函数嵌套定义的过程中，上层函数的符号，在下一层函数中也是可以使用的，因此，才有了这样一个静态链。

在相同的嵌套层次，那么从上一层的静态链复制，如果被调用过程高n层，那么底层的局部变量是不能被看见的，所以需要回溯n层来寻找。

嵌套层次显示表display表，我看了一下就是对每一层的嵌套都指出其最新的函数的静态链（最新的含义是，类似于树的深搜，而每一层都有一个当前路径的节点这样）

另一个重要考点是C的函数调用约定和函数调用栈。

以x86为例子，ebp或者rbp指明栈帧的基地址，也就是ebp表明的4byte或者8byte是上一个栈帧的ebp。

Ebp+4（或者64位下+8）则是返回地址，也就是调用函数的下一条指令。再往上，ebp+8（16）开始则依次是压栈的参数。

Ebp向下则是局部变量。

不同的函数调用约定的区别在于寄存器参数传递的问题。

这玩意数量太多了，自己去看LLVM的函数调用约定吧，最基本的c calling convention，fastcc到各种各样的玩意儿，哎。

在这里我只列举几个，先是x86，之所以硬件特化，实在是我看LLVM下的很多调用约定仅仅在x86下进行了实现，至于其他平台那基本凉凉。

cdecl，gcc默认的，32位下的，就是压栈就完事了。我个人看法，和LLVM默认的调用约定一样。

显然这里面不涉及任何寄存器传参，所以问题不大。

Stdcall 32位win下的一种调用约定，百度一下都有，反正在汇编上面的表示更加复杂。就C来说，还有函数后面加@和参数字节数量，返回用retn x而不是ret，自动清栈，而cdecl则手动清栈等等。如果C++的话，则更为复杂，不过，额跟mcc没有任何关系。

Microsoft x64：同样看名字就知道是win64下的约定。有四个寄存器用于传参数，rcx，rdx，r8和r9。（顺序不能乱哦，这里是先rcx，再rdx），另外，对于浮点寄存器栈和向量（主要是在simd指令那里进行了涉及）的问题也需要特别考虑。

<https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170>

（当然，限于编译和调试环境在gcc下面，这玩意，我肯定只留个接口，有兴趣的去做吧）

System V AMD64，则是基于64位x86的一种调用约定，也是mcc在64位下的一种约定。存放前6个整形参数，rdi，rsi，rdx，rcx，r8，r9。注意这里面是rdx，rcx的顺序，超过这个数，才进行压栈。

对于浮点参数，则依次存放在xmm0-7的浮点寄存器中去。

对于返回值，整形返回值放在rax中，而浮点返回值则会放在xmm0中。

另外，对于不定长度的，还需要在al中传入参数数量。

Riscv的调用约定，我给网址吧（这东西难道还有win嘛，肯定没有的咯）：

<https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf>

我看了一下，大概是尽可能的用a0-a7和fa0和fa7进行传参。目前我并没有考虑要实现这部分的打算。

由此，在mcc的实现中，不去考虑什么Microsoft的调用约定。

在类型中，遇到使用了x64的情况，则将0-6号寄存器绑定于用于存放函数调用的参数，当然同样的碰到了浮点数的情况也一样，同样生成固定的寄存器编号用于存放浮点数。而在之后寄存器分配中，再进行实际和对应寄存器的绑定。

这当然会引出一个问题，这不再满足静态单赋值了！既然存在了绑定，这当然会引起一堆问题，我个人对此的看法是，做部分SSA优化，而对于这些赋值的语句则不做优化考虑。

显然，这部分寄存器属易失性寄存器，并不会影响什么东西。但是，在另一方面，对于callee来说，如果有需要则需要将其保存。（我就琢磨着吧，这保存不还是需要当成局部变量压栈么？我查看gcc弄出来的汇编代码，都有个把rdi里面的值（就是第一个参数）挪到rbp减去某个常数（就是栈上面）的操作，而如果，这保存的值是不需要的，那当然不需要挪到栈上面，但这样一来，就会被优化掉，并不明白这样做的意义在哪里）

在数据结构的表示上面，因为调用库函数的话必须和ABI相对应，因此，还是要在函数的结构体中声明其使用的调用约定。

并且，在使用函数调用的时候，需要进行函数调用约定的检查，看看caller和callee的调用约定是否相同。

为了不在后面的分析中出什么幺蛾子，我增加了一个recv指令，其含义就是在栈上面压栈一个收到的参数，并依照对应的调用约定将值赋值给他。他虽然会使用固定的某个寄存器作为入参，但是在后面分析中，是当做和alloca一样的terminal节点来使用的。