1. **中间语言的作用**

中间语言顾名思义是介于c/c++这类高级语言和汇编低级语言之间的一种语言。例如LLVM编译器的LLVM-IR语言、GCC编译器的GIMPLE-IR语言。一般情况下，正常是几乎不可能会用中间语言来编写程序的，仅被用作编译过程中作为高级语言连接低级语言的桥梁。

中间语言与汇编语言的区别:

中间语言的语法格式往往与汇编语言非常类似，都是已ADD/SUB/MUL/DIV/STORE/LOAD这类的一个个指令来编写程序，其与汇编语言的根本性差别是屏蔽掉了CPU层面的机器差异。使得一套代码能在任意不同的CPU指令集架构下正确运行(这需要由CPU开发商设计出专门针对自身CPU的中间语言编译器)。此外，中间语言还完成了一个及其重要的任务:将函数中定义的堆栈局部变量的操作映射到对CPU物理寄存器的操作。从而极彻底解决了手动管理堆栈、CPU寄存器对程序员身心的折磨(写汇编代码最恶心的就是这点)。

中间语言与高级语言的区别:

中间语言最核心的任务还是完成对底层硬件细节的屏蔽，但其用起来依然很反人类。和高级编程语言的最大区别还是2点: 无法定义数组、类对象这种复合数据结构，无法定义接近于数学语言的表达式。

总结:一个完整的高级编译器由2端组成

前端:高级语言到中间语言的编译器(一般由各家语言设计厂商开发,如java/c/c++/rust)

后端:中间语言到汇编语言的编译器(一般由各家CPU设计厂商开发,如x86,arm,risc-v)

中间语言就是两端开发人员约定的一个统一的语法标准

1. **OpenQinling编译工具链的中间语言介绍**

作者致力于完整实现一整个高级语言编译器(前端完成一个c99标准的c语言编译器，后端就对接设计的OpenQinling023A指令集CPU)。故也就干脆自个定套中间语言标准了。为了方便中间语言的语法解析，以及考虑到一般情况下不会直接使用中间语言来写代码。所以我参考了XML/HTML的关系与设计思路。设计了一门类似于XML/JSON这种数据信息描述语言的PSDL(程序结构描述语言)。并在PSDL语言的基础上增加了拥有实际功能的中间语言PSDL-IR(也就类似于HTML基于XML)。

如此一来，可以极大的减轻中间语言的语法解析复杂度，同时让前端编译器更容易将编译出的中间表示信息传递给后端编译器。

1. PSDL语言的数据结构及语法

PSDL语言所展现的数据结构也比较类似于XML，但是为了在保障解析难度低的情况下兼顾一定的可读性，其语法格式与XML差距巨大。

PSDL数据结构是树状的，由一个树的节点构成。每个树的节点可以包含任意个节点参数和任意个分支，每个分支中可以有任意个子节点。一个PSDL树中必须有一个名为PSDL的根节点。

定义一个PSDL节点完整的语法:

[节点名]([节点的参数列表,用”,”分隔])

.[分支1的名字]{

#当前分支下的子节点列表，用”;”分隔

}.[分支2的名字]{

#当前分支下的子节点列表，用”;”分隔

}........任意个分支(用”.”分隔).....;

如果PSDL节点无任何参数，那么括号就可不写。例如PSDL的根节点:

PSDL.BODY{ #PSDL至少得有个BODY的分支

#分支节点

};

一个PSDL的分支下也可以无任何子节点，但是如果定义了依然在解析PSDL文本后还是能查询到目标分支。那么此时这个分支就承担了声明当前节点某些特殊属性的作用。

例如PSDL-IR中声明一个中断函数的节点:

FUN(函数名).VARS{

#声明函数中所用变量的子节点列

}.BODY{

#函数中具体执行逻辑的子节点列

}.INTERRUPT;

在此处定义了一个无任何子节点的分支，名为INTERRPUT。当PSDL-IR语法分析器检测到该函数定义节点下存在这么个分支,就能知道这是一个中断处理函数。此时分支语法也就承担了声明属性的功能。只是这些个属性都只能是布尔值，定义了为真，没定义为假。

定义PSDL节点时，节点名、分支名的注意点: 只能是0-9,a-z,A-Z组成的名字，且首字符只可为字母。

填写PSDL节点参数时的注意点: 参数内容中不可有”()”、”{}”、”,”这些个限制符，其余无任何限制。另外参数可以用””双引号括起来当做定义的是一个字符串参数值，字符串参数值中哪怕出现了限制符也没有问题。

1. PSDL-IR的基本语法

1.PSDL-IR的根节点:

PSDL.BODY{

#可在其中定义函数、全局/静态变量、导入外部函数/变量

};

1. 定义一个全局/静态变量

支持的数据类型:

64位整型:LONG 64位无符号整型:ULONG 64位浮点数:DOUBLE

32位整型: INT 32位无符号整型:UINT 32位浮点数:FLOAT

16位整型:SHORT 16位无符号整型:USHORT 指针:POINTER

8位整型:BYTE 8位无符号整型:UBYTE

普通数值型初始值的语法:

<初始值的数据类型:常量值>

如定义一个float初始值: <FLOAT:32.3223>

定义一个指向其它变量、函数的指针型初始值语法

[函数/变量名 +/- 偏移值]<高限位:低限位>

如定义指向函数main的初始值: [main]

1. 定义一个普通数据类型的变量

不需要初始化:

VAR(数据类型,数据名称);

需要初始化:

VAR(数据类型,数据名称,数据初始值);

1. 定义一个数组类型变量

定义不初始化数组:

ARRAY(数组名,字节数);

需要初始化:

ARRAY(数组名).BODY{

#数组内的初始化节点

};

数组内初始化可写的节点:

普通数据类型变量: VAR(数据类型,初始值);

非初始化块: ARRAY(字节数);

字符串: STRING(“字符串文本”);

1. 定义一个变量组

GROUP(变量组名).BODY{

#其中可以定义普通数据类型变量和不初始化数组

};

如果一个指针型初始值要指向变量组中的一个变量/数组时的语法:

[变量组名.变量/数组名 +/- 偏移值]<高限位-低限位]

将定义的一个变量设置为常量型

在变量定义节点中增加分支属性 .CONST

将定义的一个变量设置为全局变量(默认是静态变量)

在变量定义节点中增加分支属性 .EXPORT

示例:定义一个C语言的全局变量 int a = 32;等效的变量

VAR(INT,a,<int:32d>).EXPORT;

1. 定义一个函数

定义函数的节点名和支持的主要分支:

FUN(函数名).ARGS{

#定义函数参数的分支

}.VARS{

#定义函数内局部变量的分支

}.BODY{

#定义函数内具体运行指令的分支

};

参数/局部变量定义的节点语法:

普通数据类型的参数/变量: DATA(数据类型,名称);

数组类型的参数/变量: ARRAY(名称,字节数);

【字节数可为0,一般stdcall方式的可变参数数量函数中最后一个参数就为一个字节数=0的数组,从而方便后续取出stdcall可变参数的起始地址】

定义函数为stdcall方式调用(默认为cdecl方式调用): 添加分支属性 .STDCALL

定义为全局函数: 添加分支属性 .EXPORT

(特殊函数扩展属性:不同的CPU可以不同)

将函数定义为系统内核的中断响应处理函数: 添加分支属性 .INTERRUPT

将函数定义为驱动层请求系统功能的处理函数: 添加分支属性 .DRIVERCALL

将函数定义为应用层请求系统功能的处理函数: 添加分支属性 .DRIVERCALL

1. 函数体中的运行指令的节点定义

基本运算指令节点:

四则运算(a/b/c的数据类型必须相同,c可为常量)

ADD(a,b,c); a=b+c

SUB(a,b,c); a=b-c

MUL(a,b,c); a=b\*c

DIV(a,b,c); a=b/c

取模运算(a/b/c的数据类型必须相同且为整型,c可为常量)

REM(a,b,c); a=b%c

移位运算(a和b的数据类型必须相同且为整型,c可为常量)

SAL(a,b,c); a=b<<c

SAR(a,b,c); a=b>>c

位运算(a/b/c数据类型必须相同且为整型)[not中b可为常量,其它c可为常量]

NOT(a,b); a=~b

AND(a,b,c); a=b&c

OR(a,b,c); a=b|c

XOR(a,b,c); a=b^c

布尔运算(a必须是8位整型，b/c数据类型必须相同且为整型)[not中b可为常量,其它c可为常量]

BNOT(a,b); a=!b

BAND(a,b,c); a=b&&c

BOR(a,b,c); a=b||c

比较运算(a必须是8位整型，b/c数据类型必须相同且为整型,c可为常量)

CMEC(a,b,c); a=(b==c)

CMNEC(a,b,c); a=(b!=c)

CMMC(a,b,c); a=b>c

CMLC(a,b,c); a=b<c

CMMEC(a,b,c); a=(b>=c)

CMLEC(a,b,c); a=(b<=c)

内存操作指令(a无数据类型限制,b必须是指针,b可为常量)

STORE(a,b); \*b=a

LODE(a,b); a=\*b

读取一个局部变量的内存地址到另一个局部指针变量中,c只可为数值型常量

ADDRESS(a,b,c); a=(&b)+c

条件执行程序块(.IF和.ELSE至少要有一个)

a/b必须相同数据类型,b可为常量

比较运算指令可为:

\* EC b==c

\* NEC b!=c

\* MC b>c

\* LC b<c

\* MEC b>=c)

\* LEC b<=c)

PROCESS(比较运算指令,a,b).JUDGE{ #可不写

#jmp语句判断是执行if还是else块的代码,通过设置决定变量的值来实现

}.IF{

#决定变量的值!=0 是执行的代码

}.ELSE{

#决定变量的值==0 是执行的代码

}

无条件跳转

LABEL(标记名); #标记一个跳转位置

GOTO(标记名); #跳转到标记点指定的位置

函数调用

调用cdecl的函数: CALL(指向函数地址的变量/或函数地址常量,函数参数....);

调用stdcall的函数:

CALL(指向函数地址的变量/或函数地址常量,函数参数....).STDCALL;

【选错调用方式会造成堆栈平衡被破坏，程序崩溃。必须要谨慎】

OpenQinling023A的函数调用方式：

对于函数参数,就通过将参数数据入栈来传入(与x86的传参方式相同)

对于函数返回值的如何返回，PSDL中间语言层没有进行规范，可以自行在高级语言中自定义方式。一般选择将返回值的地址作为一个指针型参数传入，在被调函数RETURN前解引用传入的返回值指针对指针指向地址进行赋值，来完成数据的返回。如果不需要接收返回值，那就给返回值指针参数传入NULL，被调函数RETURN前做一下判断，来确定是否要返回。

函数退出

RETURN;

1. 导入外部的函数、变量

EXTERN(函数/变量名);

**PSDL-IR代码示例程序:**

**原C语言代码:**

int printf(const char\*fstr,...);

static int add(int a,int b){

return a+b;

}

void main(){

int a = 10;

int b = 43;

float c = add(a,b);

print(“a+b=%f\r\n”,c);

};

**转为PSDL-IR代码:**

EXTERN(printf);#导入c标准库的printf函数

FUN(add).ARGS{

DATA(pointer,retPtr);#传入返回值的指针作为第一个函数参数

DATA(int,a);

DATA(int,b);

}.VARS{

DATA(int,tmp);#临时变量，缓存结果

}.BODY{

#计算a+b结果，暂存到tmp

ADD(tmp,a,b);

#判断retPtr是否不为0，如果不为0说明需要返回结果

PROCESS(NEC,retPtr,<UINT:0D>).IF{

STORE(tmp,retPtr);

};

#末尾RETURN可以不写

};

ARRAY(constStr).BODY{

STRING(“a+b=%f\r\n”);

}.CONST;

FUN(main).VARS{

DATA(INT,a);

DATA(INT,b);

DATA(FLOAT,c);

DATA(POINTER,cPtr); #暂存c变量的指针

}.BODY{

#初始化a,b的值

MOV(a,<int:10d>);

MOV(b,<int:43d>);

#等效于c = add(a,b);

ADDRESS(cPtr,c,0d);

CALL([add],cPtr,a,b);

#printf输出c的结果

CALL([printf],[constStr],c).STDCALL;

};

**编译生成的023A汇编代码:**

IMPORT printf;

CODE add{

#将函数运算过程中要用到的数据寄存器数据入栈缓存

PUSH DWORD,R1;

PUSH DWORD,R2;

PUSH DWORD,R3;

#函数中有用到堆栈存储局部变量数据，创建堆栈存储空间

USUB SP,4D;

#读取传入参数a,b;将其相加结果存入临时变量tmp

SRD DWORD,R1,24D;

SRD DWORD,R2,20D;

ADD R3,R2,R1; #tmp变量数值暂存在r3寄存器

#读取返回值指针，比较其与0是否相同

SRD DWORD,R1,16D;

UNEC R1,0D;

SWE DWORD,R3,0D; #将tmp变量数值写入堆栈

JNE [this.\*if\_else\*0];

#如果返回值指针不等于0，将tmp数值写入到返回值指针指向的位置

SRD DWORD,R1,16D;

SRD DWORD,R2,0D;

WE DWORD,R2,R1;

\*if\_else\*0> #如果返回值指针等于0，程序直接从JNE位置跳转到此处

\*return\*>

#释放局部变量占用的堆栈空间

UADD SP,4D;

#恢复入栈保存的寄存器数据

POP DWORD,R3;

POP DWORD,R2;

POP DWORD,R1;

#释放实参变量占用的堆栈空间

UADD SP,12D;

#退出函数

RT;

};

DATA constStr{

STR "a+b=%f\r\n";

};

CODE main{

#将函数运算过程中要用到的数据寄存器数据入栈缓存

PUSH DWORD,R1;

PUSH DWORD,R2;

PUSH DWORD,R3;

#函数中要调用其他函数，将链接寄存器数据入栈保存

PUSH DWORD,TPC;

#函数中有用到堆栈存储局部变量数据，创建堆栈存储空间

USUB SP,4D;

#开始函数中真正的执行逻辑

MOV R1,10D; #初始化变量a数据

MOV R2,43D; #初始化变量b数据

ADD R3,SP,0D; #取变量c的地址到cPtr

#将a/b/cPtr的数据依次入栈

PUSH DWORD,R2;

PUSH DWORD,R1;

PUSH DWORD,R3;

#设置链接寄存器指向add函数地址

INIT TPC,[add];

RT; #跳转入add函数

#将c变量数据和[constStr]依次入栈

SRD DWORD,R1,0D;

PUSH DWORD,R1;

INIT R2,[constStr];

PUSH DWORD,R2;

#设置链接寄存器指向printf函数地址

INIT TPC,[printf];

RT; #跳转至printf函数中

#printf函数执行完毕，因为是stdcall调用约定，清理实参使用的堆栈

UADD SP,8D;

\*return\*>

#释放函数中局部变量使用的堆栈空间

UADD SP,4D;

#恢复入栈保存的寄存器数据

POP DWORD,TPC;

POP DWORD,R3;

POP DWORD,R2;

POP DWORD,R1;

RT;

};