2021/03/08_x86逆向_第3课_基本运算(乘除法)

笔记本: x86逆向-C
创建时间: 2021/3/8 星期一 10:02
作者: ileemi

数学优化

a+b-c 对应的波兰表达式:
sub(add(a, b), c);
-+abc

add(a, b)
mov ecx, a
add ecx, b
mov eax, ecx ; 保存返回

sub(a, b)
mov ecx, a
sub ecx, b
mov eax, ecx ; 保存返回

a+b-c:
mov ecx, a
add ecx, b
mov eax, ecx
mov ecx, eax
sub ecx, c
mov eax, ecx

折叠
传播
公共

数学优化：
a * a + 2 * a * b + b * b
(a + b) * (a + b)

> 使用空函数以及scanf会打断常量传播（防止变量被常量化）。"lea" 汇编指令除了可以取地址操作外也常用于数学表达式中（数学运算的变形操作）。

示例代码：

```
void  foo(int  *pb){
    *pb  =  5;
}
int  main(int  argc,  char*  argv[]){
    int  b;
    int  a;
    //  scanf("%d",  &b);
    foo(&b);
    foo(&a);
    printf("%d\n",  argc*b  +  b*a);
  // Release 版会进行优化
  // (argc + a) * b
    return  0;
}
```

编译器的优化选项

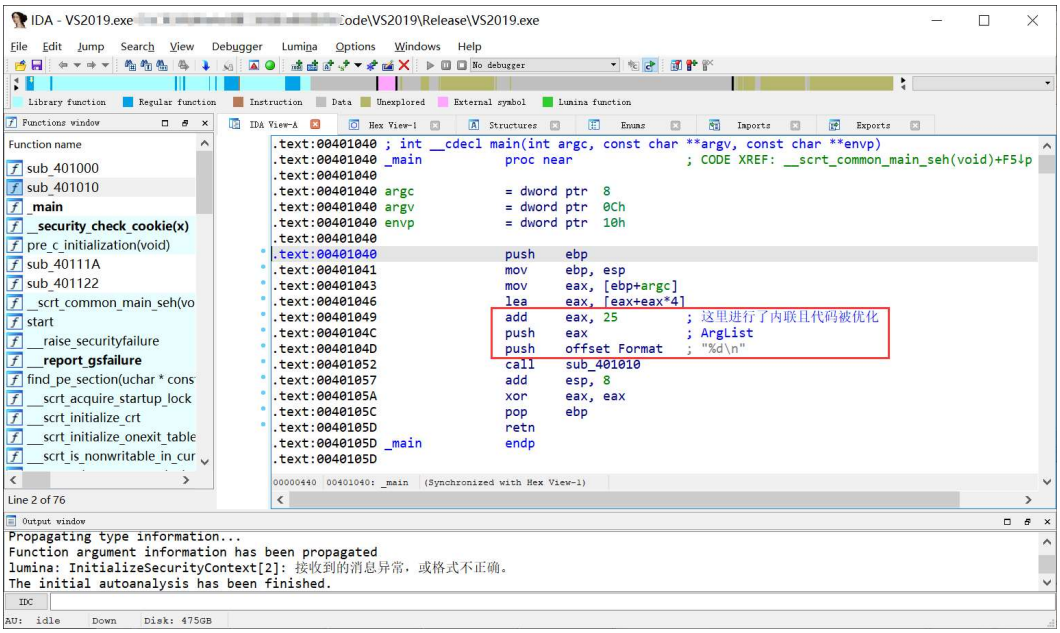
- VC++6.0 的优化是函数局部优化，不支持全程序优化。
- VS 的高版本（Release）支持全程序优化（默认开启），会检查一些简单函数，将其内敛到调用位置上，之后在对其进行优化。

示例代码（VS2019 Release开启全程序优化）：

```
void  foo(int  *pb){
    *pb  =  5;
}
int  main(int  argc,  char*  argv[]){
    int  b;
    int  a;
    //  scanf("%d",  &b);
    foo(&b); // b  =  5; -- 编译器会将简单函数内联，之后在优化，会跨函数传播
    foo(&a); // a  =  5;
    printf("%d\n",  argc*b  +  b*a);
  // Release 版会进行优化
  // (argc + a) * b
```

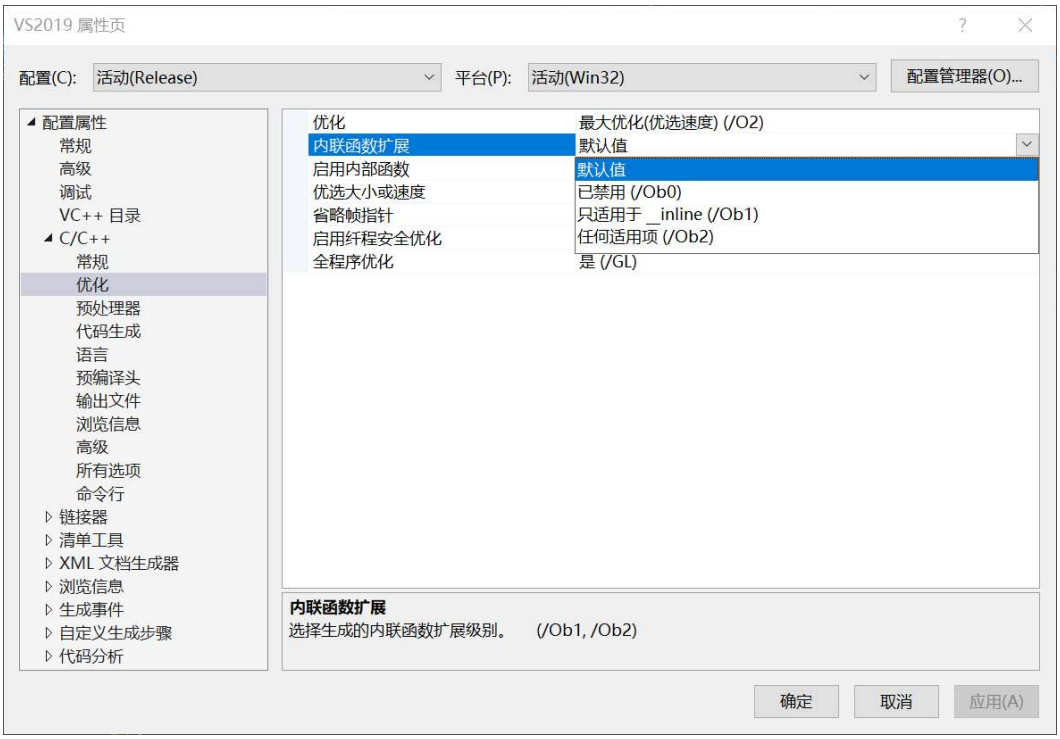
```
    return 0;
}
```

根据上下文汇编代码可以分析处 lea 是取地址还是表达式求值。



> 集成开发环境中的内联函数扩展选项：

- /Ob0：不管什么情况都禁止内联
- /Ob1：当有 "inline" 关键字的时候才内联
- /Ob2：根据情况能内联就内联



一些数学优化不能依靠编译器优化。

> 比例因子寻址支持下面的写法：
> lea reg, [base + reg * n + imm]

乘法

乘法的三种表示方法（下面的操作数为32位寄存器）：

- 单操作数：imul r32 --> edx.eax = eax * r32
- 双操作数：imul r32, r32 --> r32 = r32 * r32
- 三操作数：imul r1, r2, imm --> r1 = r2 * imm （Release版会被优化调）

变量乘变量

> 不会进行优化，会直接使用乘法相关的汇编指令。

常量乘常量

> 会折叠

变量乘常量

> 会优化

不使用imul等功能指令，实现变量乘常量（编译器Release版不使用乘法相关的汇编指令是考虑指令周期的原因）。
示例1（VC6 Release）：

```
printf("%d\n", argc * 51);
/*
VC6 Release版做法：假设 argc = 1
mov ecx, [esp+argc] --> ecx = 1
mov eax, ecx --> eax = 1
shl eax, 4 --> eax = 16
add eax, ecx  --> eax = 17
lea eax, [eax+eax*2] -- 17 + 17 * 2 = 51
*/
printf("%d\n", argc * 25);
mov eax, argc
lea eax, [eax + eax * 4]
lea eax, [eax + eax * 4]
```

示例2：

```
printf("%d\n", argc + argc * 8 + 999);
// VC6 Release:
mov eax, [esp+arg_0]
```

```
lea eax, [eax, eax*8+999]
// VS2019 Release:
mov eax, [esp+arg_0]
lea eax, [eax, eax*8]
add eax, 3E7h
```

除法

有符号乘：**imul**
无符号乘：**mul**
有符号除：**idiv**
无符号除：**div**

- VS2019 - Debug：
  有符号乘以无符号：imul
  下面的结果应该倒向
  有符号除以无符号：div
  无符号除以有符号：div
  无符号乘以无符号：imul
- VS2019 - Release：
  下面的结果应该倒向
  有符号除以无符号：div
  无符号除以有符号：div
  无符号除以无符号：div
  有符号除以有符号：idiv
  无符号乘以无符号：imul

> 有符号和无符号混除都是无符号。

除法定义：

- 定义1：已知两个因数的积与其中一个因数，求另一个因数的运算，叫做除法。
- 定义2：在整数除法中，只有能整除与不能整除两种情况，当不能整除时，就会产生余数。

除法取整

1. 向下取整
   往负无穷大的方向找最接近x的整数值，也就是取得不大于x的最大正数。
   例如：+3.5 向下取整得到3，-3.5向下取整得到-4。
   右移位相当于向下取整（C语言移位取得不大于数学移位得最大整数）：
   1011111>>3 --> 1011
   向下取整得除法。当除数为2的幂的时候，可以直接用有符号右移指令（sar）来完成，但是向下取整存在一个问题：
   (-a/b) 向下取整 != - ((a/b) 向下取整) (假设a/b的结果不为整数)
   可能是因为这个原因，C语言整数除法没有使用float方法。

2. 向上取整
   往正无穷大的方向找最接近x的整数值，也就是取得不小于x的最小整数。
   例如：+3.5 向上取整得到4，-3.5向上取整得到-3。
   在标准 C 语言的 math.h 中有定义 ceil 函数，其作用就是向上取整，也有人称之为 "天花板取整"。向上取整也存在一个问题：
   (-a/b) 向上取整 != - ((a/b) 向上取整) (假设a/b的结果不为整数)
3. 向零取整（C语言向0取整）
   所谓对x向零取整，就是取得往0方向最接近x的整数值，放弃小数部分。
   例如：+3.5 向零取整得到3，-3.5向零取整得到-3。
   向零取整的除法满足：
   (-a/b) 向零取整 = (a/-b) 向零取整 = - ((a/b) 向零取整)
   当 a/b >= 0时，(a/b) 向零取整 = (a/b) 向零取整
   当 a/b < 0时，(a/b) 向零取整 = (a/b) 向零取整

在 C 语言和其它多数高级语言中，对整数除法规定为向零取整。有人也称之为 "截断除法"

取模

> a / b = q, r
> r = a - qb;
> 代码示例：

```
int  main(int  argc,  char*  argv[])
{
    printf("%d\n",  10  %  3);       //  10 - 3*3 = 1
    printf("%d\n",  10  %  -3);      //  10 - (-3*-3) = 1
    printf("%d\n",  -10  %  3);      //  -10 - (3*-3) = -1
    printf("%d\n",  -10  %  -3);   //  -10 - (-3*3) = -1
    return  0;
}
```

> 余数的符号跟被除数走。