Компилятор за выходные: наконец-то ассемблер

haqreu

Компилятор за выходные: наконец-то ассемблер

Средний

8 мин

8.2K

Продолжаем разговор об игрушечном компиляторе мной придуманного простейшего языка wend. На этот раз мы добрались до определённой вехи: наконец-то будем генерировать не питоновский код, а ассемблерный.

Ну а когда оно заработает, предлагаю решить задачу: как сэмулировать побитовые операции and-not-xor-or при помощи четырёх арифметических.

Это уже шестая статья из цикла, для понимания происходящего надо ознакомиться если не со всеми, то хотя бы с предыдущей.

Оглавление цикла

6. Избавляемся от зависимостей: пишем лексер и парсер сами

7. Оптимизирующий компилятор?

8. Сборщик мусора?

Работу со стеком и регистрами, которые доставляют наибольшее количество проблем новичкам в ассемблере, мы разобрали в предыдущей статье, так что на этот раз осталось всего ничего. На данный момент наш компилятор генерирует питоновский код, структура которого полностью соответствует желаемому ассемблерному. Единственный момент, с которым осталось разобраться — это с выводом на экран, всё остальное уже решительно готово.

Hello world, или вывод строк на экран

Чаще всего в учебных компиляторах выбирают ассемблер MIPS, но мне не нравится запускать код в эмуляторе, поэтому я недолго думал, и выбрал x86 GNU ассемблер, благо, он идёт в составе gcc. Не могу точно сказать почему, но захотелось мне 32-битную версию.
Для наших целей совершенно ни к чему быть ассемблерным гуру, но программы уровня хелловорлд писать надо уметь.

Давайте сделаем заготовку, от которой будем впоследствии отталкиваться. Представим, что у нас есть файл helloworld.s со следующим содержимым:

```
.global _start
        .data
hello: .ascii "hello world\n"
        hello_len = . - hello
        .align 2
        .text
_start:
        movl $4, %eax         # sys_write system call (check asm/unistd_32.h for the table)
        movl $1, %ebx         # file descriptor (stdout)
        movl $hello, %ecx     # message to write
        movl $hello_len, %edx # message length
        int  $0x80            # make system call

_end:
        movl $1, %eax   # sys_exit system call
        movl $0, %ebx   # error code 0
        int $0x80       # make system call
```

Тогда мы его можем скомпилировать при помощи команд as и ld следующим образом:

```
as --march=i386 --32 -o helloworld.o helloworld.s &&
ld -m elf_i386 helloworld.o -o helloworld &&
```

```
./helloworld
```

Если всё пошло хорошо, то на экране должно красоваться гордое приветствие. Теперь давайте разбираться, что же там происходит. А там только два системных вызова — sys_write и sys_exit. На сях то же самое можно было бы написать следующим образом:

```
#include <sys/syscall.h>
#include <unistd.h>

int main(void) {
        syscall(SYS_write, 1, "hello world\n", 12);
        return 0;
}
```

Если звёзды правильно сойдутся, то gcc сгенерирует примерно такой же ассемблерный код. Для наших нужд никаких других системных вызовов больше не нужно, write и exit нам хватит за глаза, ведь единственное взаимодействие с внешним миром в wend — это вывод на экран.

Wend не умеет никаких операций со строками, только вывод константных строк на экран, поэтому мой компилятор для каждой строки просто создаёт в заголовке уникальный идентификатор ровно как для нашего hello world. Для вывода на экран булевых значений две константные строки true и false. А что с числами? А вот тут придётся чуть-чуть поработать. Я лентяй, и мне неохота было разбираться с линковкой glibc и тому подобного, поэтому роскошь printf мне недоступна. Ну и ладно, мы и с sys_write управимся :)

Вывод на экран десятичных чисел

sys_write умеет выводить на экран строки, поэтому нам надо научиться конвертировать числа (у меня только знаковые 32-битные) в строковое представление. Для этого я закатал рукава и написал функцию print_int32:

```
.global _start
        .data
        .align 2
        .text
_start:
        pushl $-729
        call print_int32
        addl $4, %esp
_end:
        movl $1, %eax   # sys_exit system call
        movl $0, %ebx   # error code 0
        int $0x80       # make system call

print_int32:
        movl 4(%esp), %eax  # the number to print
        cdq
```

```
        xorl %edx, %eax
        subl %edx, %eax     # abs(%eax)
        pushl $10           # base 10
        movl %esp, %ecx     # buffer for the string to print
        subl $16, %esp      # max 10 digits for a 32-bit number (keep %esp dword-aligned)
0:      xorl %edx, %edx     #     %edx = 0
        divl 16(%esp)       #     %eax = %edx:%eax/10 ; %edx = %edx:%eax % 10
        decl %ecx           #     allocate one more digit
        addb $48, %dl       #     %edx += '0'       # 0,0,0,0,0,0,0,0,0,0,'1','2','3','4','5','6'
        movb %dl, (%ecx)    #     store the digit   # ^                   ^                   ^
        test %eax, %eax     #                       # %esp                %ecx (after)        %ecx (before)
        jnz 0b              # until %eax==0         #                     <----- %edx = 6 ----->
        cmp %eax, 24(%esp)  # if the number is negative                           |
        jge 0f              #                                                     |
        decl %ecx           # allocate one more character                         |
        movb $45, 0(%ecx)   # '-'                                                 |
0:      movl $4, %eax       # write system call                                   |
        movl $1, %ebx       # stdout                                              |
        leal 16(%esp), %edx # the buffer to print                                 |
        subl %ecx, %edx     # number of digits    <---------------------------------
        int $0x80           # make system call
        addl $20, %esp      # deallocate the buffer
        ret
```

Чужой ассемблерный код читать непросто, поэтому давайте я приведу питоновский эквивалент нашей функции:

```
def print_int32(n):
    buffer = [ None ]*16 # output string buffer
    ecx = 0              # number of characters stored in the buffer

    eax = abs(n)
    while True:
        edx = eax %  10
        eax = eax // 10
        buffer[ecx] = chr(edx + ord('0'))
        ecx += 1
        if eax == 0: break

    if n<0:
        buffer[ecx] = '-'
        ecx += 1

    print(''.join(buffer[ecx-1::-1]))

print_int32(-729)
```

Write я буду вызывать только один раз, поэтому нужно подготовить строковый буфер. 32-битное число не потребует больше 11 символов, поэтому я выделяю 16 под буфер (чтобы стек был выровнен по краю машинного слова). Затем конвертирую в строку модуль заданного

числа, и в конце приклеиваю минус, если число было отрицательным.

При помощи такой нехитрой гимнастики мы можем выводить на экран строки и числа, и, что характерно, без головной боли линковки с какой-нибудь 32-битной версией libc на 64-битной системе.

А такой вывод на экран необходим не только для поддержки инструкции print нашего языка, но и для отладки. GDB это для слабых духом, вставка вывода на экран где ни попадя — это наше всё ;)

Собираем всё вместе

Ну, собственно, и всё. Теперь берём шаблон генерации питоновского кода, и вместо вывода питоновского `print()` достаточно написать `call print_int32`, вместо `eax = eax * ebx` написать `imull %ebx, %eax`. Таким образом планомерно переводим питоновские инструкции в ассемблерные, и дело в шляпе! Никаких тонкостей не осталось, долгожданный компилятор почти готов. Можно взять релиз v0.0.5 и с ним поиграть.

Давайте уже программировать на wend! Побитовые операции.

Если вы внимательно следили за происходившим, то видели, что из численных типов у меня только знаковые 32-битные целые числа, и над ними разрешены только базовые арифметические операции, в то время как более развитые языки позволяют, например, делать побитовые логические операции. А я что, рыжий? Я тоже могу :)

На самом деле, это отличное упражнение, рекомендую в мой код не смотреть и попробовать написать код самостоятельно. Обратите внимание, что я железно знаю, что у меня числа хранятся в дополнительном коде, и нет никаких undefined behavior при знаковых переполнениях :)

AND

Давайте попробуем сэмулировать побитовое «и» при помощи стандартной арифметики. Я, недолго думая, обрабатываю самый старший бит, а затем просто пускаю тривиальный цикл по 31 оставшемуся биту:

```
//           _    _ _____
//      /\   | \ | |  __ \
//     /  \  |  \| | |  | |
//    / /\ \ | . ` | |  | |
//   / ____ \| |\  | |__| |
//  /_/    \_\_| \_|_____/
    fun and(a:int, b:int) : int {
```

```
        var result:int;
        var pow:int;

        result = 0;
        if (a<0 && b<0) {
            result = -2147483648;
        }
        if (a<0) {
            a = a + 2147483648;
        }
        if (b<0) {
            b = b + 2147483648;
        }
        pow = 1;
        while a>0 || b>0 {
            if a % 2 == 1 && b % 2 == 1 {
                result = result + pow;
            }
            a = a / 2;
            b = b / 2;
            pow = pow * 2;
        }
        return result;
    }
```

Код весьма дубовый, если кто-то может предложить более изящный подход, с удовольствием его перейму!

NOT

Побитовое «не» выглядит существенно проще:

```
//   _   _  ____ _______
//  | \ | |/ __ \__   __|
//  |  \| | |  | | | |
//  | . ` | |  | | | |
//  | |\  | |__| | | |
//  |_| \_|\____/  |_|
```

```
    fun not(a:int) : int {
        return -1 - a;
    }
```

XOR и OR

Ну а поскольку «и» и «не» хватает для моделирования всех остальных операций, то «или» сделать тривиально:

```
//  __   _____  _____
//  \ \ / / __ \|  __ \
//   \ V / |  | | |__) |
//    > <| |  | |  _  /
//   / . \ |__| | | \ \
//  /_/ \_\____/|_|  \_\
    fun xor(a:int, b:int) : int {
        return a - and(a,b) +  b - and(a,b);
    }

//    ____  _____
//   / __ \|  __ \
//  | |  | | |__) |
//  | |  | |  _  /
//  | |__| | | \ \
//   \____/|_|  \_\
    fun or(a:int, b:int) : int {
        return xor(xor(a,b),and(a,b));
    }
```

Тест

Запускаем тест на заботливо подготовленных случайных данных, и смотри-ка, работает!

```
 bitwise and     -1804289383      1681692777      1957747793      -719885386       596516649      1025202362       783368690     -2044897763
 -1804289383     -1804289383        70555657       338728977     -1810617712          268809       336599192        70254736     -2079059431
  1681692777        70555657      1681692777      1680906305      1142163488       537663529       605558824       607125600        68947977
  1957747793       338728977      1680906305      1957747793      1410353168       545266689       873486352       615530576        68178961
  -719885386     -1810617712      1142163488      1410353168      -719885386        17173280       353585330        68239794     -2078981612
   596516649          268809       537663529       545266689        17173280       596516649       554309672       578814240        34346505
  1025202362       336599192       605558824       873486352       353585330       554309672      1025202362       739328178        68767768
   783368690        70254736       607125600       615530576        68239794       578814240       739328178       783368690       101793808
 -2044897763     -2079059431        68947977        68178961     -2078981612        34346505        68767768       101793808     -2044897763

  bitwise or     -1804289383      1681692777      1957747793      -719885386       596516649      1025202362       783368690     -2044897763
 -1804289383     -1804289383      -193152263      -185270567      -713557057     -1208041543     -1115686213     -1091175429     -1770127715
  1681692777      -193152263      1681692777      1958534265      -180356097      1740545897      2101336315      1857935867      -432152963
  1957747793      -185270567      1958534265      1957747793      -172490761      2008997753      2109463803      2125585907      -155328931
  -719885386      -713557057      -180356097      -172490761      -719885386      -140542017       -48268354       -4756490      -685801537
   596516649     -1208041543      1740545897      2008997753      -140542017       596516649      1067409339       801071099     -1482727619
  1025202362     -1115686213      2101336315      2109463803       -48268354      1067409339      1025202362      1069242874     -1088463169
   783368690     -1091175429      1857935867      2125585907        -4756490       801071099      1069242874       783368690     -1363322881
```

```
-2044897763     -1770127715      -432152963      -155328931      -685801537     -1482727619     -1088463169     -1363322881     -2044897763

bitwise xor     -1804289383      1681692777      1957747793      -719885386       596516649      1025202362       783368690     -2044897763
-1804289383               0      -263707920      -523999544      1097060655     -1208310352     -1452285405     -1161430165       308931716
 1681692777       -263707920               0       277627960     -1322519585      1202882368      1495777491      1250810267      -501100940
 1957747793       -523999544       277627960               0     -1582843929      1463731064      1235977451      1510055331      -223507892
 -719885386      1097060655     -1322519585     -1582843929               0      -157715297      -401853684       -72996284      1393180075
  596516649     -1208310352      1202882368      1463731064      -157715297               0       513099667       222256859     -1517074124
 1025202362     -1452285405      1495777491      1235977451      -401853684       513099667               0       329914696     -1157230937
  783368690     -1161430165      1250810267      1510055331       -72996284       222256859       329914696               0     -1465116689
-2044897763       308931716      -501100940      -223507892      1393180075     -1517074124     -1157230937     -1465116689               0

                -1804289383      1681692777      1957747793      -719885386       596516649      1025202362       783368690     -2044897763
bitwise not      1804289382     -1681692778     -1957747794       719885385      -596516650     -1025202363      -783368691      2044897762
```

Подводим итоги

Промежуточная цель достигнута: мы научились компилировать собственный язык в настоящий x86 gnu ассемблер. На данный момент для парсинга я пользуюсь сторонней библиотекой sly, но по пути меня слегка занесло, и выяснилось, что написать свой парсер совсем несложно. А заодно можно будет грамматику языка подправить, чтобы приятнее писать можно было!

Таким образом, следующие две статьи будут о том, как самостоятельно сделать лексер и парсер, готовый код уже лежит в репозитории.

А вот что будет потом... Есть у меня большое подозрение (я ненастоящий сварщик, я маску нашёл!), что самое интересное начинается именно потом. Я попробую разобраться, и заодно рассказать вам, как работает оптимизирующий компилятор. В качестве промежуточного представления я выберу LLVM IR, чтобы можно было в любой момент запускать код при помощи LLVM, но при этом всё будет сделано вручную **без** LLVM. Бюджета на «оптимизирующий» (только показывающий принцип) компилятор я себе отведу в районе пятисот дополнительных строк питоновского кода. Посмотрим, что получится :)

Stay tuned, have fun!

Если эта публикация вас вдохновила и вы хотите поддержать автора — не стесняйтесь нажать на кнопку