ассемблер в UNIX mini-faq

крис касперски, ака мыщъх, no-email

Q1:какие asm-трансляторы существуют?

Стандартный ассемблер для UNIX'a это — **as**, выходящий в бесплатно распространяемый комплект **binutils** (ftp://ftp.gnu.org/gnu/binutils), имеющийся в практически любом дистрибутива и поддерживающий AT&T синтаксис. Переваривает огромное количество процессоров (включая Intel Pentium-4 SSE3 и AMD x86-64). В качестве макропроцессора использует штатный препроцессор Си. Выходной формат: a.out.

NASM (Netwide Assembler – Расширенный Ассемблер) x86-транслятор ассемблера (16/32-разрядные режимы), поддерживающий синтаксис Intel и макроязык в стиле MASM. Выходные форматы: bin, aout, aoutb, coff, elf, as86, obj, win32, rdf, obj-ieee. Бесплатен, но содержит множество ошибок и странностей поведения: https://sourceforge.net/projects/nasm.

YASM ("Yes, it's an assembler" — "Да, это ассемблер". другие варианты расшифровки: "Your favorite assembler" — ваш любимый ассемблер и как "Yet another assembler" — "еще один ассемблер") гибридный транслятор, поддерживающий оба синтаксиса (AT&T + Intel), а потому совместимый с NASM'ом (попутно исправляя его ошибки) и частично с as. Переваривает команды Intel x86 (16/32) и AMD x86-64. Выходные форматы: bin, coff, elf. Распространяется бесплатно: http://www.tortall.net/projects/yasm;

FASM (Flat Assembler – Ассемблер Плоского Режима) довольно самобытный ассемблер со своим ни с чем не совместимым синтаксисом (в стиле Intel) и мощным макро-движком. Поддерживает Intel x86 (16/32) и AMD x86-64. Выходные форматы: bin, mz, pe, coff, elf. Бесплатен: http://flatassembler.net;

Q2: какой ассемблер выбрать?

Наибольшей популярностью пользуется **as**. Знать его синтаксис необходимо уже хотя бы затем, чтобы разбираться с чужими программами. Если вы раньше программировали под MS-DOS/Windows то, вероятно, лучшим выбором окажется NAMS/YASM (последний более предпочтителен). FASM (при всем уважении к нему) можно рекомендовать только его поклонникам.

Q3:что такое Intel и AT&T нотация?

AT&T синтаксис разрабатывался компанией AT&T в те далекие времена, когда никакого Intel'a вообще не существовало, процессоры менялись как перчатки и знание нескольких ассемблеров было вполне нормальным явлением. По сравнению с синтаксисом Intel, AT&T-синтаксис намного более избыточен, но это сделано умышленно с целью сокращения ошибок (хинт: на одном процессоре команда MOV может перемещать 32-бита, на другом 16, а на третьем вообще 64).

Отличия синтаксиса AT&T от Intel следующие:

- **имена регистров предваряются префиксом: "%":**
  - Intel: eax, ebx, dl;
  - AT&T: %eax, %ebx, %dl;
- **обратный порядок операндов: вначале источник, затем приёмник**:
  - Intel: mov eax, ebx;
  - AT&T: movl %ebx, %eax;
- **размер операнда задается суффиксом, замыкающим инструкцию, всего есть три типа суффиксов: "b" — байт (8-бит), "w" – слово (16-бит) и "l" – двойное слово (32-бита)**:
  - Intel: mov ah, al;
  - AT&T: movb %al, %ah;
  - Intel: mov bx, ax;
  - AT&T: movw %ax, %bx;
  - Intel: mov eax, ebx;
  - AT&T: movl %ebx, %eax;

- **в командах длинного косвенного перехода или вызова (indirect far jump/call), а так же дальнего возврата из функции (ret far) префикс размера ("l") ставится _перед_ командой (сокращение от long jmp/call) независимо от физического размера операнда, равного 32-бита в 16-разрядном режиме и 48-бит в 32-разрядном:**
  - Intel: jmp large fword ptr ds:[666h];
  - AT&T: ljmp *0x666;
  - Intel: retf;
  - AT&T: lret;
- **числовые константы записываются в Си-соглашении**:
  - Intel: 69h;
  - AT&T: 0x69;
- **для получения смещения метки используется префикс "$", отсутствие которого приводит к чтению содержимого ячейки:**
  - Intel: mov eax, offset label;
  - AT&T: movl $label, %eax;
  - Intel: mov eax, [label];
  - AT&T: movl label, %eax;
- **в тех случаях, когда метка является адресом перехода, префикс "$" опускается:**
  - Intel: jmp label;
  - AT&T: jmp label;
  - Intel: jmp –;
  - AT&T jmp 0x69;
- **для косвенного перехода по адресу используется префикс "*":**
  - Intel: jmp dword ptr ds:[69h];
  - AT&T: jmp *0x69;
  - Intel: jmp dword ptr ds:[label];
  - AT&T: jmp *label;
  - Intel: jmp eax;
  - AT&T: jmp *%eax;
  - Intel: jmp dword ptr ds: [eax];
  - AT&T: jmp *(%eax);
- **использование префикса "$" перед константой используется для получения ее значения. знак (если он есть) ставится после префикса. константа без указателя трактуется как указатель:**
  - Intel: mov eax, 69h;
  - AT&T movl $0x69, %eax;
  - Intel: mov eax, -69h;
  - AT&T movl $-0x69, %eax;
  - Intel: mov eax, [69h];
  - AT&T movl 0x69, %eax
- **для реализации косвенной адресации базовый регистр заключается в круглые скобки, перед которыми может присутствовать индекс, записанный в виде числовой константы или метки _без_ префикса "$":**
  - Intel: mov eax, [ebx];
  - AT&T: movl (%ebx), %eax;
  - Intel: mov eax, [ebx+69h];
  - AT&T: movl 0x69(%ebx), %eax;
  - Intel: mov eax, [ebx+label];
  - AT&T: movl label(%ebx), %eax;
- **если регистров несколько, то они разделаются через запятую:**
  - Intel: mov eax, [ebx+ecx];
  - AT&T: movl (%ebx, %ecx), %eax;
- **для задания коэффициента масштабирования (scale) перед первым регистром ставится ведущая запятая (при использовании базово индексной адресации запятая опускается), а сам коэффициент отделяется другой запятой, без префикса "$":**
  - Intel: mov eax, [ebx*8];

- AT&T movl (,%ebx, 8), %eax
- Intel: mov eax, [ebx*8+label];
- AT&T movl label(,%ebx, 8), %eax
- Intel: mov eax, [ecx+ebx*8+label];
- AT&T: movl label(%ecx, %ebx, 8);
- Intel: mov eax, [ebx+ecx*8+label];
- AT&T: movl label(%ebx, %ecx, 8);

- **сегментная адресация с использованием сегментных регистров отличается от Intel'а использованием круглых скобок вместо квадратных:**
  - Intel: mov eax, es:[ebx];
  - AT&T: movl %es:(%bx), %eax;
- **в командах переходов и вызовов функций непосредственные сегмент и смещение разделяется не двоеточием, а запятой** ~~***(причем если сегмент меньше 0Fh, то происходит "нормализация" к сегменту 00h ,что создает проблемы при работе с селекторами, для которых правила нормализации неприемлемы в принципе – проверялось на as 2.11.2 и 2.15 – баг в IDA 4.7, 5.0)***~~**:**
  - Intel: jmp far 10h:100000h (псевдоконструкция!)
  - AT&T: jmp $0x10, $0x100000;
  - Intel: jmp far ptr 10:100000
  - AT&T: jmp $10, $0100000 – транслируется в –> jmp far ptr 0:0F4240h;

Q4: как выглядит программа на asm?

Простейшая ассемблерная программа, работающая через штатную библиотеку LIBC, и выводящая "hello, world!" на консоль выглядит так:

```
.text

// объявляем глобальную метку main
.global main

main:
        pushl   $len                    // длина строки
        pushl   $msg                    // указатель на строку
        pushl   $1                      // stdout
        call    write                   // функция записи
        addl    $12,%esp                // выталкиваем аргументы из стека

        ret                             // возвращаемся в стартовый код

.data
        msg: .ascii "hello, world!\n" // строка для вывода
        len = . - msg                   // вычисление длины строки
```

Листинг 1 исходный текст программы demo-asm-libc.S, работающей через LIBC

Q5: как ее собрать и запустить?

Проще и правильнее всего транслировать ассемблерные программы с помощью… компилятора gcc, однако, в этом случае у файла должно быть расширение .S, иначе компилятор не поймет, что это ассемблерная программа:

```
# компилируем и линкуем
$gcc demo-asm-libc.S -o demo-asm-libc

# убиваем символьную инфу (для сокращения размеров файла)
$strip demo-asm-libc

# запускаем на выполнение
$./demo-asm-libc
hello, world!
```

Листинг 2 сборка ассемблерной программы при помощи gcc

Q6: есть ли у UNIX'а API?

Да, у UNIX'а есть API — высокоуровневые библиотеки и в первую очередь LIBC (условный аналог KERNEL32.DLL в win32), пример использования которой был продемонстрирован в листинге 1.

Некоторые хакеры тяготеют к использованию системных вызовов (syscall'ов), представляющих собой своеобразный Native-API, по разному реализованный в различных системах, что затрудняет написание программ, работающих более, чем на одной машине. Тем не менее, применение syscall'ов оправдано в shell-коде, червях и вирусах в силу простоты и компактности их вызова.

Q7: покажите программу с системными вызовами!

Программа, выводящая "hello, world!" на консоль, переложенная на системные вызовы выглядит так:

```
.text

// точка входа, которую ищет линкер по умолчанию
.globl  _start

_start:
        movl    $4,%eax         // системный вызов #4 "write"
        movl    $1,%ebx         // 1 -- stdout
        movl    $msg,%ecx       // смещение выводимой строки
        movl    $len,%edx       // длина строки
        int     $0x80           // write(1, msg, len);

        movl    $1, %eax        // системный вызов #1 "exit"
        xorl    %ebx,%ebx       // код возврата
        int     $0x80           // exit(0);

.data
        msg: .ascii "hello,elf\n"
        len = . - msg
```

Листинг 3 исходный текст программы demo-asm-80h.S, работающей через syscall'ы

Q8: и как же ее собрать?

```
# транслируем
$as -o demo-asm-80h.o demo-asm-80h.S

# линкуем
$ld -s -o demo-asm-80h demo-asm-80h.o

# убиваем символьную инфу (для сокращения размеров файла)
$strip demo-asm-80h

# запускаем на linux
./demo-asm-80h
hello, world!

# запускам на xBDS (через эмулятор системных вызовов)
brandelf -t Linux demo-asm-80h
./demo-asm-80h
hello, world!
```

Листинг 4 сборка ассемблерной программы без помощи gcc