1 Регистры

Регистр — это небольшой объем очень быстрой памяти, размещённой на процессоре. Он предназначен для хранения результатов промежуточных вычислений, а также некоторой информации для управления работой процессора. Так как регистры размещены непосредственно на процессоре, доступ к данным, хранящимся в них, намного быстрее доступа к данным в оперативной памяти.

Все регистры можно разделить на две группы: пользовательские и системные. Пользовательские регистры используются при написании «обычных» программ. В их число входят основные программные регистры (англ. basic program execution registers; все они перечислены ниже), а также регистры математического сопроцессора, регистры MMX, XMM (SSE, SSE2, SSE3). Системные регистры (регистры управления, регистры управления памятью, регистры отладки, машинно-специфичные регистры MSR и другие) здесь не рассматриваются. Регистры общего назначения (РОН, англ. General Purpose Registers, сокращённо GPR). Размер — 32 бита.

- %eax: Accumulator register — аккумулятор, применяется для хранения результатов промежуточных вычислений.
- %ebx: Base register — базовый регистр, применяется для хранения адреса (указателя) на некоторый объект в памяти.
- %ecx: Counter register — счетчик, его неявно используют некоторые команды для организации циклов (см. loop).
- %edx: Data register — регистр данных, используется для хранения результатов промежуточных вычислений и ввода-вывода.
- %esp: Stack pointer register — указатель стека. Содержит адрес вершины стека.
- %ebp: Base pointer register — указатель базы кадра стека (англ. stack frame). Предназначен для организации произвольного доступа к данным внутри стека.
- %esi: Source index register — индекс источника, в цепочечных операциях содержит указатель на текущий элемент-источник.
- %edi: Destination index register — индекс приёмника, в цепочечных операциях содержит указатель на текущий элемент-приёмник.

Эти регистры можно использовать «по частям», так же как и в MASM. Всё отличие лишь в том, что надо ставить знак "%"перед названием регистра. Сегментные регистры:

- %cs: Code segment — описывает текущий сегмент кода.
- %ds: Data segment — описывает текущий сегмент данных.
- %ss: Stack segment — описывает текущий сегмент стека.
- %es: Extra segment — дополнительный сегмент, используется неявно в строковых командах как сегмент-получатель.
- %fs: F segment — дополнительный сегментный регистр без специального назначения.
- %gs: G segment — дополнительный сегментный регистр без специального назначения.

Регистр флагов eflags и его младшие 16 бит, регистр flags. Содержит информацию о состоянии выполнения программы, о самом микропроцессоре, а также информацию, управляющую работой некоторых команд. Регистр флагов нужно рассматривать как массив битов, за каждым из которых закреплено определённое значение. Регистр флагов напрямую не доступен пользовательским программам; изменение некоторых битов eflags требует привилегий. Ниже перечислены наиболее важные флаги.

cf: carry flag, флаг переноса: 1 — во время арифметической операции был произведён перенос из старшего бита результата; 0 — переноса не было; zf: zero flag, флаг нуля: 1 — результат последней операции нулевой; 0 — результат последней операции ненулевой; of: overflow flag, флаг переполнения: 1 — во время арифметической операции произошёл перенос в/из старшего (знакового) бита результата; 0 — переноса не было; df: direction flag, флаг направления. Указывает направление просмотра в строковых операциях: 1

— направление «назад», от старших адресов к младшим; 0 — направление «вперёд», от младших адресов к старшим.

Есть команды, которые устанавливают флаги согласно результатам своей работы: в основном это команды, которые что-то вычисляют или сравнивают. Есть команды, которые читают флаги и на основании флагов принимают решения. Есть команды, логика выполнения которых зависит от состояния флагов. В общем, через флаги между командами неявно передаётся дополнительная информация, которая не записывается непосредственно в результат вычислений. Все флаги полностью идентичны флагам в MASM.

Указатель команды eip (instruction pointer). Размер — 32 бита. Содержит указатель на следующую команду. Регистр напрямую недоступен, изменяется неявно командами условных и безусловных переходов, вызова и возврата из подпрограмм.

2 Стек

В работе со стеком нет никаких отличий от MASM, за исключением лишь того, что регистр esp надо писать

Порядок байт числа в оперативной памяти такой же как в MASM: сначала располагаются младшие байты. Этот порядок называется интеловским или little-endian. Именно он используется в процессорах x86. Как написать Hello, world! на Си.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
  printf("Hello, world!\n");
  exit(0);
}
```

Вот только printf(3) — функция стандартной библиотеки Си, а не операционной системы. «Чем это плохо?» — спросите вы. Да, в общем, всё нормально, но, читая этот учебник, вы, вероятно, хотите узнать, что происходит «за кулисами» функций стандартной библиотеки на уровне взаимодействия с операционной системой. Это, конечно же, не значит, что из ассемблера нельзя вызывать функции библиотеки Си. Просто мы пойдём более низкоуровневым путём.

Как вы уже, наверное, знаете, стандартный вывод (stdout), в который выводит данные printf(3), является обычным файловым дескриптором, заранее открываемый операционной системой. Номер этого дескриптора — 1. Теперь нам на помощь придёт системный вызов write(2).

```
WRITE(2)        Руководство программиста Linux        WRITE(2)

ИМЯ
        write - писать в файловый дескриптор

ОБЗОР
        #include <unistd.h>

        ssize_t write(int fd, const void *buf, size_t count);

ОПИСАНИЕ
        write пишет count байт в файл, на который ссылается файловый
        дескриптор fd, из буфера, на который указывает buf.


А вот и сама программа:

#include <unistd.h>

int main(int argc, char* argv[])
{
```

```
  char str[] = "Hello, world!\n";
  write(1, str, sizeof(str) - 1);
  _exit(0);
}
```

Почему sizeof(str) - 1? Потому, что строка в Си заканчивается нулевым байтом, а его нам печатать не нужно.

Теперь скопируйте следующий текст в файл hello.s. Файлы исходного кода на ассемблере имеют расширение .s.

Стоит дополнительно указать директиву ассемблеру .code32в самом начале листинга, так как часть ассемблеров в GNU/Linux системах настроено на 64 битную архитектуру.

```
.data                           /* поместить следующее в сегмент данных
                                                                    */

hello_str:                      /* наша строка                      */
        .string "Hello, world!\n"

                                /* длина строки                     */
        .set hello_str_length, . - hello_str - 1

.text                           /* поместить следующее в сегмент кода */

.globl  main                    /* main - глобальный символ, видимый
                                   за пределами текущего файла      */
.type   main, @function         /* main - функция (а не данные)     */


main:
        movl    $4, %eax        /* поместить номер системного вызова
                                   write = 4 в регистр %eax         */

        movl    $1, %ebx        /* первый параметр - в регистр %ebx;
                                   номер файлового дескриптора
                                   stdout - 1                       */

        movl    $hello_str, %ecx  /* второй параметр - в регистр %ecx;
                                     указатель на строку            */

        movl    $hello_str_length, %edx /* третий параметр - в регистр
                                           %edx; длина строки       */

        int     $0x80           /* вызвать прерывание 0x80          */

        movl    $1, %eax        /* номер системного вызова exit - 1   */
        movl    $0, %ebx        /* передать 0 как значение параметра  */
        int     $0x80           /* вызвать exit(0)                    */

        .size   main, . - main    /* размер функции main            */
```

Напомним, сейчас наша задача — скомпилировать первую программу. Подробное объяснение этого кода будет потом.

```
[user@host:~]$ gcc hello.s -o hello
[user@host:~]$
```

Если компиляция проходит успешно, GCC ничего не выводит на экран. Кроме компиляции, GCC автоматически выполняет и компоновку, как и при компиляции программ на C. Теперь запускаем нашу программу и убеждаемся, что она корректно завершилась с кодом возврата 0.

```
[user@host:~]$ ./hello
Hello, world!
[user@host:~]$ echo $?
0
```

3 Команды

Команды ассемблера — это те инструкции, которые будет исполнять процессор. По сути, это самый низкий уровень программирования процессора. Каждая команда состоит из операции (что делать?) и операндов (аргументов). Операции мы будем рассматривать отдельно. А операнды у всех операций задаются в одном и том же формате. Операндов может быть от 0 (то есть нет вообще) до 3. В роли операнда могут выступать:

Конкретное значение, известное на этапе компиляции, например числовая константа или символ. Записываются при помощи знака $, например: $0xf1, $10, $hello_str. Эти операнды называются непосредственными.

Регистр. Перед именем регистра ставится знак %, например: %eax, %bx, %cl. Указатель на ячейку в памяти (как он формируется и какой имеет синтаксис записи – далее в этом разделе).

Неявный операнд. Эти операнды не записываются непосредственно в исходном коде, а подразумеваются. Нет, конечно, компьютер не читает ваши мысли. Просто некоторые команды всегда обращаются к определённым регистрам без явного указания, так как это входит в логику их работы. Такое поведение всегда описывается в документации.

Почти у каждой команды можно определить операнд-источник (из него команда читает данные) и операнд-назначение (в него команда записывает результат). Общий синтаксис команды ассемблера такой:

Операция Источник, Назначение

```
movl    $4, %eax                /* поместитьномерсистемноговызова
                                   write = 4 врегистр  %eax */
```

Как видим, источник — это непосредственное значение 4, а назначение — регистр %eax. Суффикс l в имени команды указывает на то, что ей следует работать с операндами длиной в 4 байта. Все суффиксы:

- b (от англ. byte) — 1 байт,
- w (от англ. word) — 2 байта,
- l (от англ. long) — 4 байта,
- q (от англ. quad) — 8 байт.

Таким образом, чтобы записать $42 в регистр %al (а он имеет размер 1 байт):
movb $42, %al
Как формируется указатель на ячейку памяти? Синтаксис:
смещение(база, индекс, множитель)
Вычисленный адрес будет равен $база + индекс множитель + смещение$. Множитель может принимать значения 1, 2, 4 или 8. Например:

- (%ecx) адрес операнда находится в регистре %ecx. Этим способом удобно адресовать отдельные элементы в памяти, например, указатель на строку или указатель на int;
- 4(%ecx) адрес операнда равен %ecx + 4. Удобно адресовать отдельные поля структур. Например, в %ecx адрес некоторой структуры, второй элемент которой находится «на расстоянии» 4 байта от её начала (говорят «по смещению 4 байта»);
- -4(%ecx) адрес операнда равен %ecx - 4;
- foo(,%ecx,4) адрес операнда равен $foo + %ecx * 4$, где foo — некоторый адрес. Удобно обращаться к элементам массива. Если foo – указатель на массив, элементы которого имеют размер 4 байта, то мы можем заносить в %ecx номер элемента и таким образом обращаться к самому элементу.

Ещё один важный нюанс: команды нужно помещать в секцию кода. Для этого перед командами нужно указать директиву .text. Вот так:

```
.text
        movl    $42, %eax
        ...
```

4 Данные

Существуют директивы ассемблера, которые размещают в памяти данные, определенные программистом. Аргументы этих директив – список выражений, разделенных запятыми.

- .byte – размещает каждое выражение как 1 байт;
- .short – 2 байта;
- .long – 4 байта;
- .quad – 8 байт.

Например:

```
.byte   0x10, 0xf5, 0x42, 0x55
.long   0xaabbaabb
.short  -123, 456
```

Также существуют директивы для размещения в памяти строковых литералов:

- .ascii "STR"размещает строку STR. Нулевых байтов не добавляет.
- .string "STR"размещает строку STR, после которой следует нулевой байт (как в языке Си). У директивы .string есть синоним .asciz (z от англ. zero – ноль, указывает на добавление нулевого байта).

Строка-аргумент этих директив может содержать стандартные escape-последовательности, которые вы использовали в Си, например,
n,
r,
t,

,
"и так далее.

Данные нужно помещать в секцию данных. Для этого перед данными нужно поместить директиву .data. Вот так:

```
.data
        .string "Hello, world\n"
        ...
```

Если некоторые данные не предполагается изменять в ходе выполнения программы, их можно поместить в специальную секцию данных только для чтения при помощи директивы .section .rodata:

```
.section .rodata
        .string "program version 0.314"
```

Для соблюдения выравнивания в распоряжении программиста есть директива .p2align.

.p2align степень двойки, заполнитель, максимум

Директива .p2align выравнивает текущий адрес до заданной границы. Граница выравнивания задаётся как степень числа 2: например, если вы указали .p2align 3 — следующее значение будет выровнено по 8-байтной границе. Для выравнивания размещается необходимое количество байт-заполнителей со значением заполнитель. Если для выравнивания требуется разместить более чем максимум байт-заполнителей, то выравнивание не выполняется.

Второй и третий аргумент являются необязательными.

Примеры:

```
.data
        .string "Hello, world\n"    /* мы вряд ли захотим считать,
                                       сколько символов занимает эта
                                       строка, и является ли следующий
                                       адрес выровненным           */
        .p2align 2                  /* выравниваем по границе 4 байта
                                       для следующего .long        */
        .long 123456
```

5 Метки

Вы, наверно, заметили, что мы не присвоили имён нашим данным. Как же к ним обращаться? Очень просто: нужно поставить метку. Метка — это просто константа, значение которой — адрес.

```
hello_str:
        .string "Hello, world!\n"
```

Сама метка, в отличие от данных, места в памяти программы не занимает. Когда компилятор встречает в исходном коде метку, он запоминает текущий адрес и читает код дальше. В результате компилятор помнит все метки и адреса, на которые они указывают. Программист может ссылаться на метки в своём коде. Существует специальная псевдометка, указывающая на текущий адрес. Это метка . (точка).

Для создания нового символа используется директива .set. Синтаксис:

```
 .set    символ, выражение

Например, определим символ foo = 42:

.set    foo, 42

Пример:

hello_str:
        .string "Hello, world!\n"                    /* наша строка  */
        .set    hello_str_length, . - hello_str - 1  /* длина строки */
```

Сначала определяется символ hello_str, который содержит адрес строки. После этого мы определяем символ hello_str_length, который, судя по названию, содержит длину строки. Директива .set позволяет в качестве значения символа использовать арифметические выражения. Мы из значения текущего адреса (метка «точка») вычитаем адрес начала строки — получаем длину строки в байтах. Потом мы вычитаем ещё единицу, потому что директива .string добавляет в конце строки нулевой байт (а на экран мы его выводить не хотим).

6 Неинициализированные данные

Часто требуется просто зарезервировать место в памяти для данных, без инициализации какими-то значениями. Например, у вас есть переменная, значение которой определяется параметрами командной строки. Действительно, вы вряд ли сможете дать ей какое-то осмысленное начальное значение, разве что 0. Такие данные называются неинциализированными, и для них выделена специальная секция под названием .bss. В скомпилированной программе эта секция места не занимает. При загрузке программы в память секция неинициализированых данных будет заполнена нулевыми байтами.

Хорошо, но известные нам директивы размещения данных требуют указания инициализирующего значения. Поэтому для неинициализированных данных используются специальные директивы:

```
 .space  количество_байт
 .space  количество_байт, заполнитель
```

Директива .space резервирует количество_байт байт.

Также эту директиву можно использовать для размещения инициализированных данных, для этого существует параметр заполнитель — этим значением будет инициализирована память.

Например:

```
.bss
long_var_1:                 /* по размеру как .long               */
        .space 4

buffer:                     /* какой-то буфер в 1024 байта        */
        .space 1024

struct:                     /* какая-то структура размером 20 байт */
        .space 20
```

7 Адресация

Прямая или абсолютная адресация:

```
.data
num:
        .long    0x12345678

.text
main:
        movl    (num), %eax   /* Записать в регистр %eax операнд,
                                 который содержится в оперативной
                                 памяти по адресу метки num          */

        addl    (num), %eax   /* Сложить с регистром %eax операнд,
                                 который содержится в оперативной
                                 памяти по адресу метки num и записать
                                 результат в регистр %eax            */

        ret
```

Непосредственная адресация:

```
.text
main:
        movl    $0x12345, %eax       /* загрузить константу 0x12345 в
                                        регистр %eax.                */
```

Косвенная адресация:

```
.data
num:
        .long   0x1234

.text
main:
        movl    $num, %ebx      /* записать адрес метки в регистр
                                   адреса %ebx                       */

        movl    (%ebx), %eax    /* записать в регистр %eax операнд из
                                   оперативной памяти, адрес которого
                                   находится в регистре адреса %ebx  */
```

Регистровая адресация:

Предполагается, что операнд находится во внутреннем регистре процессора.

Пример:

```
.text
main:
        movl    $0x12345, %eax  /* записать в регистр константу 0x12345
                                                                      */
        movl    %eax, %ecx      /* записать в регистр %ecx операнд,
                                   который находится в регистре %eax */
```

Относительная адресация:

Этот способ используется тогда, когда память логически разбивается на блоки, называемые сегментами. В этом случае адрес ячейки памяти содержит две составляющих: адрес начала сегмента (базовый адрес) и смещение адреса операнда в сегменте. Адрес операнда определяется как сумма базового адреса и смещения относительно этой базы:

Операндi = (базаi + смещениеi)

Для задания базового адреса и смещения могут применяться ранее рассмотренные способы адресации. Как правило, базовый адрес находится в одном из регистров регистровой памяти, а смещение может быть задано в самой команде или регистре.

Рассмотрим два примера:

Адресное поле команды состоит из двух частей, в одной указывается номер регистра, хранящего базовое значение адреса (начальный адрес сегмента), а в другом адресном поле задается смещение, определяющее положение ячейки относительно начала сегмента. Именно такой способ представления адреса обычно и называют относительной адресацией. Первая часть адресного поля команды также определяет номер базового регистра, а вторая содержит номер регистра, в котором находится смещение. Такой способ адресации чаще всего называют базово-индексным.

Команда mov

Синтаксис:

mov источник, назначение

Команда mov производит копирование источника в назначение. Рассмотрим примеры:

/* * Это просто примеры использования команды mov, * ничего толкового этот код не делает */

```
.data
some_var:
        .long 0x00000072

other_var:
        .long 0x00000001, 0x00000002, 0x00000003

.text
.globl main
main:
        movl  $0x48, %eax       /* поместить число 0x00000048 в %eax */

        movl  $some_var, %eax   /* поместить в %eax значение метки
                                   some_var, то есть адрес числа в
                                   памяти; например, у автора
                                   содержимое %eax равно 0x08049589  */

        movl  some_var, %eax    /* обратиться к содержимому переменной;
                                   в %eax теперь 0x00000072          */

        movl  other_var + 4, %eax  /* other_var указывает на 0x00000001
                                   размер одного значения типа long - 4
                                   байта; значит, other_var + 4
                                   указывает на 0x00000002;
                                   в %eax теперь 0x00000002          */

        movl  $1, %ecx          /* поместить число 1 в %ecx          */

        movl  other_var(,%ecx,4), %eax  /* поместить в %eax первый
                                   (нумерация с нуля) элемент массива
                                   other_var, пользуясь %ecx как
                                   индексным регистром               */

        movl  $other_var, %ebx  /* поместить в %ebx адрес массива
                                   other_var                         */

        movl  4(%ebx), %eax     /* обратиться по адресу %ebx + 4;
                                   в %eax снова 0x00000002           */
```

```
        movl  $other_var + 4, %eax  /* поместить в %eax адрес, по
                                    которому расположен 0x00000002
                                    (адрес массива плюс 4 байта --
                                    пропустить нулевой элемент)       */

        movl  $0x15, (%eax)     /* записать по адресу "то, что записано
                                    в %eax" число 0x00000015          */
```

Внимательно следите, когда вы загружаете адрес переменной, а когда обращаетесь к значению переменной по

```
movl  other_var + 4, %eax       /* забыли знак $, в результате в %eax
                                   находится число 0x00000002        */

movl  $0x15, (%eax)             /* пытаемся записать по адресу
                                   0x00000002 -> получаем segmentation
                                   fault                             */

movl  0x48, %eax                /* забыли $, и пытаемся обратиться по
                                   адресу 0x00000048 -> segmentation
                                   fault                             */
```

Команда lea

lea — мнемоническое от англ. Load Effective Address. Синтаксис:

lea источник, назначение

Команда lea помещает адрес источника в назначение. Источник должен находиться в памяти (не может быть непосредственным значением — константой или регистром). Например:

```
.data
some_var:
        .long 0x00000072

.text
        leal  0x32, %eax        /* аналогично movl $0x32, %eax       */
        leal  some_var, %eax    /* аналогично movl $some_var, %eax   */

        leal  $0x32, %eax       /* вызовет ошибку при компиляции,
                                   так как $0x32 - непосредственное
                                   значение                          */
        leal  $some_var, %eax   /* аналогично, ошибка компиляции:
                                   $some_var - это непосредственное
                                   значение, адрес                   */

        leal  4(%esp), %eax     /* поместить в %eax адрес предыдущего
                                   элемента в стеке;
                                   фактически, %eax = %esp + 4       */
```

Команды для работы со стеком

Предусмотрено две специальные команды для работы со стеком: push (поместить в стек) и pop (извлечь из стека). Синтаксис:

push источник pop назначение

При описании работы стека мы уже обсуждали принцип работы команд push и pop. Важный нюанс: push и pop работают только с операндами размером 4 или 2 байта. Если вы попробуете скомпилировать что-то вроде

pushb 0x10

GCC вернёт следующее:

[user@host:]$ gcc test.s test.s: Assembler messages: test.s:14: Error: suffix or operands invalid for ‘push’ [user@host:]$

Согласно ABI, в Linux стек выровнен по long. Сама архитектура этого не требует, это только соглашение между программами, но не рассчитывайте, что другие библиотеки подпрограмм или операционная

система захотят работать с невыровненным стеком. Что всё это значит? Если вы резервируете место в стеке, количество байт должно быть кратно размеру long, то есть 4. Например, вам нужно всего 2 байта в стеке для short, но вам всё равно придётся резервировать 4 байта, чтобы соблюдать выравнивание.

А теперь примеры:

```
.text
        pushl $0x10             /* поместить в стек число 0x10       */
        pushl $0x20             /* поместить в стек число 0x20       */
        popl  %eax              /* извлечь 0x20 из стека и записать в
                                   %eax                              */
        popl  %ebx              /* извлечь 0x10 из стека и записать в
                                   %ebx                              */

        pushl %eax              /* странный способ сделать           */
        popl  %ebx              /* movl %eax, %ebx                   */

        movl  $0x00000010, %eax
        pushl %eax              /* поместить в стек содержимое %eax  */
        popw  %ax               /* извлечь 2 байта из стека и
                                   записать в %ax                    */
        popw  %bx               /* и ещё 2 байта и записать в %bx    */
                                /* в %ax находится 0x0010, в %bx
                                   находится 0x0000; такой код сложен
                                   для понимания, его следует избегать
                                                                     */

        pushl %eax              /* поместить %eax в стек; %esp
                                   уменьшится на 4                   */
        addl  $4, %esp          /* увеличить %esp на 4; таким образом,
                                   стек будет приведён в исходное
                                   состояние                         */
```

Интересный вопрос: какое значение помещает в стек вот эта команда
pushl %esp

Если ещё раз взглянуть на алгоритм работы команды push, кажется очевидным, что в данном случае она должна поместить уже уменьшенное значение %esp. Однако в документации Intel сказано, что в стек помещается такое значение %esp, каким оно было до выполнения команды — и она действительно работает именно так. Арифметика

Арифметических команд в нашем распоряжении довольно много. Синтаксис:
inc операнд dec операнд
add источник, приёмник sub источник, приёмник
mul множитель_1
Принцип работы:
inc: увеличивает операнд на 1. dec: уменьшает операнд на 1.

add: приёмник = приёмник + источник (то есть, увеличивает приёмник на источник). sub: приёмник = приёмник - источник (то есть, уменьшает приёмник на источник).

Команда mul имеет только один операнд. Второй сомножитель задаётся неявно. Он находится в регистре %eax, и его размер выбирается в зависимости от суффикса команды (b, w или l). Место размещения результата также зависит от суффикса команды. Нужно отметить, что результат умножения двух n-разрядных чисел может уместиться только в 2n-разрядном регистре результата. В следующей таблице описано, в какие регистры попадает результат при той или иной разрядности операндов. Команда Второй сомножитель Результат mulb %al 16 бит: %ax mulw %ax 32 бита: младшая часть в %ax, старшая в %dx mull %eax 64 бита: младшая часть в %eax, старшая в %edx

Примеры:

```
.text
        movl  $72, %eax
        incl  %eax              /* в %eax число 73                   */
        decl  %eax              /* в %eax число 72                   */
```

```
        movl  $48, %eax
        addl  $16, %eax         /* в %eax число 64                  */

        movb  $5, %al
        movb  $5, %bl
        mulb  %bl               /* в регистре %ax произведение
                                   %al  %bl = 25                    */
```

Команда lea для арифметики

Для выполнения некоторых арифметических операций можно использовать команду lea[3]. Она вычисляет адрес своего операнда-источника и помещает этот адрес в операнд-назначение. Ведь она не производит чтение памяти по этому адресу, верно? А значит, всё равно, что она будет вычислять: адрес или какие-то другие числа.

Вспомним, как формируется адрес операнда:

смещение(база, индекс, множитель)

Вычисленный адрес будет равен база + индекс множитель + смещение.

Чем это нам удобно? Так мы можем получить команду с двумя операндами-источниками и одним результатом:

```
movl  $10, %eax
movl  $7, %ebx

leal  5(%eax)       ,%ecx  /* %ecx = %eax + 5 = 15                  */
leal  -3(%eax)      ,%ecx  /* %ecx = %eax - 3 = 7                   */
leal  (%eax,%ebx)   ,%ecx  /* %ecx = %eax + %ebx  1 = 17           */
leal  (%eax,%ebx,2) ,%ecx  /* %ecx = %eax + %ebx  2 = 24           */
leal  1(%eax,%ebx,2),%ecx  /* %ecx = %eax + %ebx  2 + 1 = 25       */
leal  (,%eax,8)     ,%ecx  /* %ecx = %eax  8 = 80                  */
leal  (%eax,%eax,2) ,%ecx  /* %ecx = %eax + %eax  2 = %eax  3 = 30 */
leal  (%eax,%eax,4) ,%ecx  /* %ecx = %eax + %eax  4 = %eax  5 = 50 */
leal  (%eax,%eax,8) ,%ecx  /* %ecx = %eax + %eax  8 = %eax  9 = 90 */
```

Вспомните, что при сложении командой add результат записывается на место одного из слагаемых. Теперь, наверно, стало ясно главное преимущество lea в тех случаях, где её можно применить: она не перезаписывает операнды-источники. Как вы это сможете использовать, зависит только от вашей фантазии: прибавить константу к регистру и записать в другой регистр, сложить два регистра и записать в третий... Также lea можно применять для умножения регистра на 3, 5 и 9, как показано выше. Команда loop

Синтаксис:

loop метка

Принцип работы:

уменьшить значение регистра %ecx на 1; если %ecx = 0, передать управление следующей за loop команде; если %ecx 0, передать управление на метку.

Напишем программу для вычисления суммы чисел от 1 до 10 (конечно же, воспользовавшись формулой суммы арифметической прогрессии, можно переписать этот код и без цикла — но ведь это только пример).

```
.data
printf_format:
        .string "%d\n"

.text
.globl main
main:
        movl  $0, %eax          /* в %eax будет результат, поэтому в
                                   начале его нужно обнулить        */
        movl  $10, %ecx         /* 10 шагов цикла                   */

sum:
```

```
        addl  %ecx, %eax        /* %eax = %eax + %ecx                */
        loop  sum

        /* %eax = 55, %ecx = 0 */

/*
 * следующий код выводит число в %eax на экран и завершает программу
 */
        pushl %eax
        pushl $printf_format
        call  printf
        addl  $8, %esp

        movl  $0, %eax
        ret
```

На Си это выглядело бы так:

```
#include <stdio.h>

int main()
{
  int eax, ecx;
  eax = 0;
  ecx = 10;
  do
  {
    eax += ecx;
  } while(--ecx);
  printf("%d\n", eax);
  return 0;
}
```

Команды сравнения и условные переходы. Безусловный переход

Команда loop неявно сравнивает регистр %ecx с нулём. Это довольно удобно для организации циклов, но часто циклы бывают намного сложнее, чем те, что можно записать при помощи loop. К тому же нужен эквивалент конструкции if(). Вот команды, позволяющие выполнять произвольные сравнения операндов:

cmp операнд_2, операнд_1

Команда cmp выполняет вычитание операнд_1 – операнд_2 и устанавливает флаги. Результат вычитания нигде не запоминается.

Внимание! Обратите внимание на порядок операндов в записи команды: сначала второй, потом первый.

Сравнили, установили флаги, — и что дальше? А у нас есть целое семейство jump-команд, которые передают управление другим командам. Эти команды называются командами условного перехода. Каждой из них поставлено в соответствие условие, которое она проверяет. Синтаксис:

jcc метка

Команды jcc не существует, вместо cc нужно подставить мнемоническое обозначение условия. Мнемоника Английское слово Смысл Тип операндов

- e equal равенство любые
- n not инверсия условия любые
- g greater больше со знаком
- l less меньше со знаком
- a above больше без знака
- b below меньше без знака

Таким образом, je проверяет равенство операндов команды сравнения, jl проверяет условие операнд_1 < операнд_2 и так далее. У каждой команды есть противоположная: просто добавляем букву n:

je — jne: равно — не равно; jg — jng: больше — не больше.

Теперь пример использования этих команд:

```
.text
        /* Тут пропущен код, который получает некоторое значение в %eax.
           Пусть нас интересует случай, когда %eax = 15 */

        cmpl  $15, %eax         /* сравнение                          */
        jne   not_equal        /* если операнды не равны, перейти на
                                  метку not_equal                    */

        /* сюда управление перейдёт только в случае, когда переход не
           сработал, а значит, %eax = 15 */

not_equal:
        /* а сюда управление перейдёт в любом случае */
```

Сравните с кодом на Си:

```
if(eax == 15)
{
  /* сюда управление перейдёт только в случае, когда переход не сработал,
     а значит, %eax = 15 */
}
/* а сюда управление перейдёт в любом случае */
```

Кроме команд условного перехода, область применения которых ясна сразу, также существует команда безусловного перехода. Эта команда чем-то похожа на оператор goto языка Си. Синтаксис:

jmp адрес

Эта команда передаёт управление на адрес, не проверяя никаких условий. Заметьте, что адрес может быть задан в виде непосредственного значения (метки), регистра или обращения к памяти. Произвольные циклы

Все инструкции для написания произвольных циклов мы уже рассмотрели, осталось лишь собрать всё воедино. Лучше сначала посмотрите код программы, а потом объяснение к ней. Прочитайте её код и комментарии и попытайтесь разобраться, что она делает. Если сразу что-то непонятно — не страшно, сразу после исходного кода находится более подробное объяснение. Программа: поиск наибольшего элемента в массиве

```
.data
printf_format:
        .string "%d\n"

array:
        .long -10, -15, -148, 12, -151, -3, -72
array_end:

.text
.globl main
main:
        movl  array, %eax         /* в %eax будет храниться результат;
                                     в начале наибольшее значение - array[0]*/
        movl  $array+4, %ebx      /* в %ebx находится адрес текущего
                                     элемента массива                   */
        jmp   ch_bound           /* проверить границы массива */
loop_start:                      /* начало цикла                       */
        cmpl  %eax, (%ebx)       /* сравнить текущий элемент массива с
                                    текущим наибольшим значением из %eax
                                                                        */
```

```
        jle   less              /* если текущий элемент массива меньше
                                   или равен наибольшему, пропустить
                                   следующий код                    */
        movl  (%ebx), %eax      /* а вот если элемент массива
                                   превосходит наибольший, значит, его
                                   значение и есть новый максимум    */
less:
        addl  $4, %ebx          /* увеличить %ebx на размер одного
                                   элемента массива, 4 байта         */
ch_bound:
        cmpl  $array_end, %ebx  /* сравнить адрес текущего элемента и
                                   адрес конца массива               */
        jne    loop_start        /* если они не равны, повторить цикл снова*
/*
 * следующий код выводит число из %eax на экран и завершает программу
 */
        pushl %eax
        pushl $printf_format
        call  printf
        addl  $8, %esp

        movl  $0, %eax
        ret
```

Сначала мы заносим в регистр %eax число array[0]. После этого мы сравниваем каждый элемент массива, начиная со следующего (нам незачем сранивать нулевой элемент с самим собой), с текущим наибольшим значением из %eax, и, если этот элемент больше, он становится текущим наибольшим. После просмотра всего массива в %eax находится наибольший элемент. Отметим, что если массив состоит из 1 элемента, то следующий после нулевого элемента будет находиться за границей массива, поэтому перед циклом стоит безусловный переход на проверку границы.

Этот код соответствует приблизительно следующему на Си:

```
#include<stdio.h>
int main()
{
  static const int array[] = { -10, -15, -148, 12, -151, -3, -72 };
  static const int *array_end = &array[sizeof(array) / sizeof(int) - 1];
  int max = array[0];
  int * p = (void*)array;

  while (p != array_end)
  {
    if(*p > max)
    {
      max = *p;
    }
    p++;
  }

  printf("%d\n", max);
  return 0;
}
```

Возможно, такой способ обхода массива не очень привычен для вас. В Си принято использовать переменную с номером текущего элемента, а не указатель на него. Никто не запрещает пойти этим же путём и на ассемблере:

```
.section .rodata /* Сегмент read-only data */
str_d:
```

```
        .asciz "%d\n"
array_start:
        .long 1,2,32,6,8,-100
.set count_el, (.-array_start)/4


.globl main
.type main, @function
.text /* cs -- code segment */

main:

        movl $0, %ecx /* запишем константу 0 в %ecx */
        movl array_start, %ebx /* Записать элемент (1) в %ebx */
        jmp is_last /* перепрыгнуть сразу на проверку is_last */



search:
        movl array_start(,%ecx,4), %eax /* Запишем array_start+%ecx*4 в %eax, заметьте мы не берем адре
        cmpl %eax, %ebx /* Проверим %ebx == %eax ? */
        jge above /* если %ebx >= %eax (если ebx уже больше eax то пропустит присваивание)   */
        movl %eax, %ebx /* переместить %eax в %ebx (значит оно уже наибольшее) */
above:
        inc %ecx /* Увеличить %ecx на 1*/

is_last:
        cmpl $count_el, %ecx /* %ecx == константе count_el? */
        jl search /* if( %ecx < $count_el) goto search; */
/* Если уже не меньше, продолжаем*/

pushl %ebx /* Поместить в стек число из %ebx */
pushl $str_d /* Поместить адрес строки в стек */
call printf /* printf(&str_d, edx); */
addl $2*4, %esp /* Переместить стек на 2 ячейки выше(для intel) */
```

Рассматривая код этой программы, вы, наверно, уже поняли, как создавать произвольные циклы с постусловием на ассемблере, наподобие do while(); в Си. Ещё раз повторю эту конструкцию, выкинув весь код, не относящийся к циклу:

```
loop_start:                     /* начало цикла                        */

        /* вот тут находится тело цикла */

        cmpl  ...               /* что-то с чем-то сравнить для
                                   принятия решения о выходе из цикла */
        jne    loop_start       /* подобрать соответствующую команду
                                   условного перехода для повторения цикла     */
```

В Си есть ещё один вид цикла, с проверкой условия перед входом в тело цикла (цикл с предусловием): while(). Немного изменив предыдущий код, получаем следующее:

```
        jmp    check
loop_start:                     /* начало цикла                        */

        /* вот тут находится тело цикла */

check:
        cmpl  ...               /* что-то с чем-то сравнить для
```

```
                                принятия решения о выходе из цикла */
        jne    loop_start      /* подобрать соответствующую команду
                                условного перехода для повторения цикла     */
```

Кто-то скажет: а ещё есть цикл for()! Но цикл

```
for(init; cond; incr)
{
  body;
}
```

эквивалентен такой конструкции:

```
init;
while(cond)
{
  body;
  incr;
}
```

И так-же:

```
for(init; decr; )
{
  body;
}
```

Эквивалетен

```
  movl init, %ecx
  body:
    loop body
```

Таким образом, нам достаточно и уже рассмотренных двух видов циклов. Логическая арифметика

Кроме выполнения обычных арифметических вычислений, можно проводить и логические, то есть битовые.

- and источник, приёмник
- or источник, приёмник
- xor источник, приёмник
- not операнд
- test операнд_1, операнд_2

Команды and, or и xor ведут себя так же, как и операторы языка Си , |, ˙.

Команда not инвертирует каждый бит операнда (изменяет на противоположный), так же как и оператор языка Си .

Команда test выполняет побитовое И над операндами, как и команда and, но, в отличие от неё, операнды не изменяет, а только устанавливает флаги. Её также называют командой логического сравнения, потому что с её помощью удобно проверять, установлены ли определённые биты. Например, так:

```
        testb $0b00001000, %al  /* установлен ли 3-й (с нуля) бит?   */
        je    not_set
        /* нужные биты установлены */
not_set:
        /* биты не установлены */
```

Обратите внимание на запись константы в двоичной системе счисления: используется префикс 0b.

Команду test можно применять для сравнения значения регистра с нулём:

```
        testl %eax, %eax
        je    is_zero
        /* %eax != 0 */
is_zero:
        /* %eax == 0 */
```

Intel Optimization Manual рекомендует использовать test вместо cmp для сравнения регистра с нулём.

Ещё следует упомянуть об одном трюке с xor. Как вы знаете, a XOR a = 0. Пользуясь этой особенностью, xor часто применяют для обнуления регистров:

```
xorl  %eax, %eax
/* теперь %eax == 0 */
```

Почему применяют xor вместо mov? Команда xor короче, а значит, занимает меньше места в процессорном кэше, меньше времени тратится на декодирование, и программа выполняется быстрее. Но эта команда устанавливает флаги. Поэтому, если вам нужно сохранить состояние флагов, применяйте mov.

Иногда для обнуления регистра применяют команду sub. Помните, она тоже устанавливает флаги.

```
subl  %eax, %eax
/* теперь %eax == 0 */
```

К логическим командам также можно отнести команды сдвигов:

```
 /* Shift Arithmetic Left/SHift logical Left */
 sal/shl количество_сдвигов, назначение

 /* SHift logical Right */
 shr     количество_сдвигов, назначение

 /* Shift Arithmetic Right */
 sar     количество_сдвигов, назначение
```

количество_сдвигов может быть задано непосредственным значением или находиться в регистре %cl. Учитываются только младшие 5 бит регистра %cl, так что количество сдвигов может варьироваться в пределах от 0 до 31.

8 Подпрограммы

Термином «подпрограмма» будем называть и функции, которые возвращают значение, и функции, не возвращающие значение (void proc(...)). Подпрограммы нужны для достижения одной простой цели — избежать дублирования кода. В ассемблере есть две команды для организации работы подпрограмм.

call метка

Используется для вызова подпрограммы, код которой находится по адресу метка. Принцип работы:

Поместить в стек адрес следующей за call команды. Этот адрес называется адресом возврата. Передать управление на метку.

Для возврата из подпрограммы используется команда ret.

ret ret число

Принцип работы:

Извлечь из стека новое значение регистра %eip (то есть передать управление на команду, расположенную по адресу из стека). Если команде передан операнд число, %esp увеличивается на это число. Это необходимо для того, чтобы подпрограмма могла убрать из стека свои параметры.

Существует несколько способов передачи аргументов в подпрограмму.

При помощи регистров. Перед вызовом подпрограммы вызывающий код помещает необходимые данные в регистры. У этого способа есть явный недостаток: число регистров ограничено, соответственно, ограничено и максимальное число передаваемых параметров. Также, если передать параметры почти во всех регистрах, подпрограмма будет вынуждена сохранять их в стек или память, так как ей может не хватить регистров для собственной работы. Несомненно, у этого способа есть и преимущество: доступ к регистрам очень быстрый. При помощи общей области памяти. Это похоже на глобальные переменные в Си. Современные рекомендации написания кода (а часто и стандарты написания кода в больших проектах)

запрещают этот метод. Он не поддерживает многопоточное выполнение кода. Он использует глобальные переменные неявным образом — смотря на определение функции типа void func(void) невозможно сказать, какие глобальные переменные она изменяет и где ожидает свои параметры. Вряд ли у этого метода есть преимущества. Не используйте его без крайней необходимости. При помощи стека. Это самый популярный способ. Вызывающий код помещает аргументы в стек, а затем вызывает подпрограмму.

Рассмотрим передачу аргументов через стек подробнее. Предположим, нам нужно написать подпрограмму, принимающую три аргумента типа long (4 байта). Код:

```
sub:
        pushl %ebp              /* запоминаем текущее значение
                                   регистра %ebp, при этом %esp -= 4 */
        movl  %esp, %ebp        /* записываем текущее положение
                                   вершины стека в %ebp              */

        /* пролог закончен, можно начинать работу */

        subl  $8, %esp          /* зарезервировать место для локальных
                                   переменных                        */


        movl  8(%ebp),  %eax    /* что-то сделать с параметрами      */
        movl  12(%ebp), %eax
        movl  16(%ebp), %eax


        /* эпилог */

        movl  %ebp, %esp        /* возвращем вершину стека в исходное
                                   положение                         */
        popl  %ebp              /* восстанавливаем старое значение
                                   %ebp, при этом %esp += 4          */
        ret

main:
        pushl $0x00000010       /* поместить параметры в стек        */
        pushl $0x00000020
        pushl $0x00000030
        call  sub               /* вызвать подпрограмму              */
        addl  $12, %esp
```

С вызовом всё ясно: помещаем аргументы в стек и даём команду call. А вот как в подпрограмме удобно достать параметры из стека? Вспомним про регистр %ebp.

Мы сохраняем предыдущее значение регистра %ebp, а затем записываем в него указатель на текущую вершину стека. Теперь у нас есть указатель на стек в известном состоянии. Сверху в стек можно помещать сколько угодно данных, %esp поменяется, но у нас останется доступ к параметрам через %ebp. Часто эта последовательность команд в начале подпрограммы называется «прологом».

. . . . . +————————-+ 0x0000F040 <– новое значение %ebp | старое значение %ebp | +————————-+ 0x0000F044 <– %ebp + 4 | адрес возврата | +————————-+ 0x0000F048 <– %ebp + 8 | 0x00000030 | +————————-+ 0x0000F04C <– %ebp + 12 | 0x00000020 | +————————-+ 0x0000F050 <– %ebp + 16 | 0x00000010 | +————————-+ 0x0000F054 . . . . . .

Используя адрес из %ebp, мы можем ссылаться на параметры:

8(%ebp) = 0x00000030 12(%ebp) = 0x00000020 16(%ebp) = 0x00000010

Как видите, если идти от вершины стека в сторону аргументов, то мы будем встречать аргументы в обратном порядке по отношению к тому, как их туда поместили. Нужно сделать одно из двух: или помещать аргументы в обратном порядке (чтобы доставать их в прямом порядке), или учитывать обратный порядок аргументов в подпрограмме. В Си принято при вызове помещать аргументы в обратном порядке. Так как операционная система Linux и большинство библиотек для неё написаны именно на Си, для обеспечения переносимости и совместимости лучше использовать «сишный» способ передачи аргументов и в ваших ассемблерных программах.

Подпрограмме могут понадобится собственные локальные переменные. Их принято держать в стеке, так как в этом случае легко обеспечить необходимое время жизни локальных переменных: достаточно в конце подпрограммы вытолкнуть их из стека. Для того, чтобы зарезервировать для них место, мы просто уменьшим содержимое регистра %esp на размер наших переменных. Это действие эквивалентно использованию соответствующего количества команд push, только быстрее, так как не требует записи в память. Предположим, что нам нужно 2 переменные типа long (4 байта), итого 2 4 = 8 байт. Таким образом, регистр %esp нужно уменьшить на 8. Теперь стек выглядит так:

. . . . . +————————+ 0x0000F038 <– %ebp - 8 | локальная переменная 2 | +————————+ 0x0000F03C <– %ebp - 4 | локальная переменная 1 | +————————+ 0x0000F040 <– %ebp | старое значение %ebp | +————————+ 0x0000F044 <– %ebp + 4 | адрес возврата | +————————+ 0x0000F048 <– %ebp + 8 | 0x00000030 | +————————+ 0x0000F04C <– %ebp + 12 | 0x00000020 | +————————+ 0x0000F050 <– %ebp + 16 | 0x00000010 | +————————+ 0x0000F054 . . . . . .

Вы не можете делать никаких предположений о содержимом локальных переменных. Никто их для вас не инициализировал нулём. Можете для себя считать, что там находятся случайные значения.

При возврате из процедуры мы восстанавливаем старое значение %ebp из стека, потому что после возврата вызывающая функция вряд ли будет рада найти в регистре %ebp неизвестно что (а если серьёзно, этого требует ABI). Для этого необходимо, чтобы старое значение %ebp было на вершине стека. Если подпрограмма что-то поместила в стек после старого %ebp, она должна это убрать. К счастью, мы не должны считать, сколько байт мы поместили, сколько достали и сколько ещё осталось. Мы можем просто поместить значение регистра %ebp в регистр %esp, и стек станет точно таким же, как и после сохранения старого %ebp в начале подпрограммы. После этого команда ret возвращает управление вызывающему коду. Эта последовательность команд часто называется «эпилогом» подпрограммы.

Внимание! Сразу после того, как вы восстановили значение %esp в эпилоге, вы должны считать, что локальные переменные уничтожены. Хотя они ещё не перезаписаны, они, несомненно, будут затёрты последующими командами push, поэтому вы не должны сохранять указатели на локальные переменные дальше эпилога своей функции.

Остаётся одна маленькая проблема: в стеке всё ещё находятся аргументы для подпрограммы. Это можно решить одним из следующих способов:

использовать команду ret с аргументом; использовать необходимое число раз команду pop и выбросить результат; увеличить %esp на размер всех помещенных в стек параметров.

В Си используется последний способ. Так как мы поместили в стек 3 значения типа long по 4 байта каждый, мы должны увеличить %esp на 12, что и делает команда addl сразу после call.

Заметьте, что не всегда обязательно выравнивать стек. Если вы вызываете несколько подпрограмм подряд (но не в цикле!), то можно разрешить аргументам «накопиться» в стеке, а потом убрать их всех одной командой. Если ваша подпрограмма не содержит вызовов других подпрограмм в цикле и вы уверены, что оставшиеся аргументы в стеке не вызовут проблем переполнения стека, то аргументы можно не убирать вообще. Всё равно это сделает команда эпилога, которая восстанавливает %esp из %ebp. С другой стороны, если не уверены — лучше уберите аргументы, от одной лишней команды программа медленнее не станет.

Строго говоря, все эти действия с %ebp не требуются. Вы можете использовать %ebp для хранения своих значений, никак не связанных со стеком, но тогда вам придётся обращаться к аргументам и локальным переменным через %esp или другие регистры, в которые вы поместите указатели. Трюк состоит в том, чтобы не изменять %esp после резервирования места для локальных переменных и до конца функции: так вы сможете использовать %esp на манер %ebp, как было показано выше. Не изменять %esp значит, что вы не сможете использовать push и pop (иначе все смещения переменных в стеке относительно %esp «поплывут»); вам понадобится создать необходимое число локальных переменных для хранения этих временных значений. С одной стороны, этот способ доступа к переменным немного сложнее, так как вы должны заранее просчитать, сколько места в стеке вам понадобится. С другой стороны, у вас появляется еще один свободный регистр %ebp. Так что если вы решите пойти этой дорогой, вы должны заранее продумать, сколько места для локальных переменных вам понадобится, и дальше обращаться к ним через смещения относительно %esp.

И последнее: если вы хотите использовать вашу подпрограмму за пределами данного файла, не забудьте сделать её глобальной с помощью директивы .globl.

Посмотрим на код, который выводил содержимое регистра %eax на экран, вызывая функцию стандартной библиотеки Си printf(3). Вы его уже видели в предыдущих программах, но там он был приведен без объяснений. Для справки привожу цитату из man:

```
PRINTF(3)            Linux Programmer's Manual            PRINTF(3)
```

```
NAME
       printf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
```

```
.data
printf_format:
        .string "%d\n"

.text
        /* printf(printf_format, %eax); */
        pushl %eax              /* аргумент, подлежащий печати       */
        pushl $printf_format    /* аргумент format                   */
        call  printf            /* вызов printf()                    */
        addl  $8, %esp          /* выровнять стек                    */
```

Обратите внимание на обратный порядок аргументов и очистку стека от аргументов.

Внимание! Значения регистров глобальны, вызывающая и вызываемая подпрограммы видят одни и те же регистры. Конечно же, подпрограмма может изменять значения любых пользовательских регистров, но она обязана при возврате восстановить значения регистров %ebp, %ebx, %esi, %edi и %esp. Сохранение остальных регистров перед вызовом подпрограммы — задача программиста. Даже если вы заметили, что подпрограмма не изменяет какой-то регистр, это не повод его не сохранять. Ведь неизвестно, как будут обстоять дела в следующей версии подпрограммы. Вы не должны делать каких-либо предположений о состоянии регистров на момент выхода из подпрограммы. Можете считать, что они содержат случайные значения.

Также внимания требует флаг df. При вызове подпрограмм флаг должен быть равен 0. Подпрограмма при возврате также должна установить флаг в 0. Коротко: если вам вдруг нужно установить этот флаг для какой-то операции, сбросьте его сразу, как только надобность в нём исчезнет.

До этого момента мы обходились общим термином «подпрограмма». Но если подпрограмма — функция, она должна как-то передать возвращаемое значение. Это принято делать при помощи регистра %eax. Перед началом эпилога функция должна поместить в %eax возвращаемое значение.

Программа: печать таблицы умножения

Рассмотрим программу посложнее. Итак, программа для печати таблицы умножения. Размер таблицы умножения вводит пользователь. Нам понадобится вызвать функцию scanf(3) для ввода, printf(3) для вывода и организовать два вложенных цикла для вычислений.

```
.data
input_prompt:
        .string "enter size (1-255): "

scanf_format:
        .string "%d"

printf_format:
        .string "%5d "

printf_newline:
        .string "\n"

size:
        .long 0

.text
.globl main
main:
        /* запросить у пользователя размер таблицы */
```

```
        pushl $input_prompt     /* format                              */
        call  printf            /* вызов printf                        */

        /* считать размер таблицы в переменную size */
        pushl $size             /* указатель на переменную size        */
        pushl $scanf_format     /* format                              */
        call  scanf             /* вызов scanf                         */

        addl  $12, %esp         /* выровнять стек одной командой сразу
                                   после двух функций                  */

        movl  $0, %eax          /* в регистре %ax команда mulb будет
                                   выдавать результат, но мы печатаем
                                   всё содержимое %eax, поэтому два
                                   старших байта %eax должны быть
                                   нулевыми                            */

        movl  $0, %ebx          /* номер строки                        */

print_line:
        incl  %ebx              /* увеличить номер строки на 1         */
        cmpl  size, %ebx
        ja    print_line_end    /* если номер строки больше
                                   запрошенного размера, завершить цикл
                                                                       */

        movl  $0, %ecx          /* номер колонки                       */

print_num:
        incl  %ecx              /* увеличить номер колонки на 1        */
        cmpl  size, %ecx
        ja    print_num_end     /* если номер колонки больше
                                   запрошенного размера, завершить цикл
                                                                       */

        movb  %bl, %al          /* команда mulb ожидает второй
                                   операнд в %al                       */
        mulb  %cl               /* вычислить %ax = %cl * %al           */

        pushl %ebx              /* сохранить используемые регистры
                                   перед вызовом printf                */
        pushl %ecx

        pushl %eax              /* данные для печати                   */
        pushl $printf_format    /* format                              */
        call  printf            /* вызов printf                        */
        addl  $8, %esp          /* выровнять стек                      */

        popl  %ecx              /* восстановить регистры               */
        popl  %ebx

        jmp   print_num         /* перейти в начало цикла              */
print_num_end:

        pushl %ebx              /* сохранить регистр                   */

        pushl $printf_newline   /* напечатать символ новой строки      */
        call  printf
```

```
        addl  $4, %esp

        popl  %ebx              /* восстановить регистр             */

        jmp   print_line        /* перейти в начало цикла           */
print_line_end:

        movl  $0, %eax          /* завершить программу              */
        ret
```

Программа: вычисление факториала

Теперь напишем рекурсивную функцию для вычисления факториала. Она основана на следующей формуле: 0! = 1, n! = n (n 1)!

```
.data
printf_format:
        .string "%d\n"

.text
/* int factorial(int) */
factorial:
        pushl %ebp
        movl  %esp, %ebp

        /* извлечь аргумент в %eax */
        movl  8(%ebp), %eax

        /* факториал 0 равен 1 */
        cmpl  $0, %eax
        jne   not_zero

        movl  $1, %eax
        jmp   return
not_zero:

        /* следующие 4 строки вычисляют выражение
           %eax = factorial(%eax - 1) */

        decl  %eax
        pushl %eax
        call  factorial
        addl  $4, %esp

        /* извлечь в %ebx аргумент и вычислить %eax = %eax * %ebx */

        movl  8(%ebp), %ebx
        mull  %ebx

        /* результат в паре %edx:%eax, но старшие 32 бита нужно
           отбросить, так как они не помещаются в int */

return:
        movl  %ebp, %esp
        popl  %ebp
        ret

.globl main
main:
        pushl %ebp
```

```
        movl  %esp, %ebp

        pushl $5
        call  factorial

        pushl %eax
        pushl $printf_format
        call  printf

        /* стек можно не выравнивать, это будет сделано
           во время выполнения эпилога */

        movl  $0, %eax                  /* завершить программу */

        movl  %ebp, %esp
        popl  %ebp
        ret
```

Любой программист знает, что если существует очевидное итеративное (реализуемое при помощи циклов) решение задачи, то именно ему следует отдавать предпочтение перед рекурсивным. Итеративный алгоритм нахождения факториала даже проще, чем рекурсивный; он следует из определения факториала: n ! = 1 2 . . . n

Говоря проще, нужно перемножить все числа от 1 до n.

Функция — на то и функция, что её можно заменить, при этом не изменяя вызывающий код. Для запуска следующего кода просто замените функцию из предыдущей программы вот этой новой версией:

```
factorial:
        movl  4(%esp), %ecx

        cmpl  $0, %ecx
        jne   not_zero

        movl  $1, %eax
        ret

not_zero:

        movl  $1, %eax
loop_start:
        mull  %ecx
        loop  loop_start
        ret
```

Что же здесь изменено? Рекурсия переписана в виде цикла. Кадр стека больше не нужен, так как в стек ничего не перемещается и другие функции не вызываются. Пролог и эпилог поэтому убраны, при этом регистр %ebp не используется вообще. Но если бы он использовался, сначала нужно было бы сохранить его значение, а перед возвратом восстановить.

Системные вызовы

Программа, которая не взаимодействует с внешним миром, вряд ли может сделать что-то полезное. Вывести сообщение на экран, прочитать данные из файла, установить сетевое соединение — это всё примеры действий, которые программа не может совершить без помощи операционной системы. В Linux пользовательский интерфейс ядра организован через системные вызовы. Системный вызов можно рассматривать как функцию, которую для вас выполняет операционная система.

Теперь наша задача состоит в том, чтобы разобраться, как происходит системный вызов. Каждый системный вызов имеет свой номер. Все они перечислены в файле /usr/include/asm-i386/unistd.h.

Системные вызовы считывают свои параметры из регистров. Номер системного вызова нужно поместить в регистр %eax. Параметры помещаются в остальные регистры в таком порядке:

первый — в %ebx; второй — в %ecx; третий — в %edx; четвертый — в %esi; пятый — в %edi; шестой — в %ebp.

Таким образом, используя все регистры общего назначения, можно передать максимум 6 параметров. Системный вызов производится вызовом прерывания 0x80. Такой способ вызова (с передачей параметров через регистры) называется fastcall. В других системах (например, *BSD) могут применяться другие способы вызова.

Следует отметить, что не следует использовать системные вызовы везде, где только можно, без особой необходимости. В разных версиях ядра порядок аргументов у некоторых системных вызовов может отличаться, и это приводит к ошибкам, которые довольно трудно найти. Поэтому стоит использовать функции стандартной библиотеки Си, ведь их сигнатуры не изменяются, что обеспечивает переносимость кода на Си. Почему бы нам не воспользоваться этим и не «заложить фундамент» переносимости наших ассемблерных программ? Только если вы пишете маленький участок самого нагруженного кода и для вас недопустимы накладные расходы, вносимые вызовом стандартной библиотеки Си, — только тогда стоит использовать системные вызовы напрямую.

В качестве примера можете посмотреть код программы Hello world. Структуры

Объявляя структуры в Си, вы не задумывались о том, как располагаются в памяти её элементы. В ассемблере понятия «структура» нет, зато есть «блок памяти», его адрес и смещение в этом блоке. Объясню на примере: 0x23 0x72 0x45 0x17

Пусть этот блок памяти размером 4 байта расположен по адресу 0x00010000. Это значит, что адрес байта 0x23 равен 0x00010000. Соответственно, адрес байта 0x72 равен 0x00010001. Говорят, что байт 0x72 расположен по смещению 1 от начала блока памяти. Тогда байт 0x45 расположен по смещению 2, а байт 0x17 — по смещению 3. Таким образом, адрес элемента = базовый адрес + смещение.

Приблизительно так в ассемблере организована работа со структурами: к базовому адресу структуры прибавляется смещение, по которому находится нужный элемент. Теперь вопрос: как определить смещение? В Си компилятор руководствуется следующими правилами:

Вся структура должна быть выровнена так, как выровнен её элемент с наибольшим выравниванием. Каждый элемент находится по наименьшему следующему адресу с подходящим выравниванием. Если необходимо, для этого в структуру включается нужное число байт-заполнителей. Размер структуры должен быть кратен её выравниванию. Если необходимо, для этого в конец структуры включается нужное число байт-заполнителей.

Примеры (внизу указано смещение элементов в байтах; заполнители обозначены XX):

struct Выравнивание структуры: 1, размер: 1 +—-+ char c; | c | ; +—-+ 0

struct Выравнивание структуры: 2, размер: 4 +—-+—-+—-+—-+ char c; | c | XX | s | short s; +—-+—-+—-+—-+ ; 0 2

struct Выравнивание структуры: 4, размер: 8 +—-+—-+—-+—-+—-+—-+—-+—-+ char c; | c | XX XX XX | i | int i; +—-+—-+—-+—-+—-+—-+—-+—-+ ; 0 4

struct Выравнивание структуры: 4, размер: 8 +—-+—-+—-+—-+—-+—-+—-+—-+ int i; | i | c | XX XX XX | char c; +—-+—-+—-+—-+—-+—-+—-+—-+ ; 0 4

struct Выравнивание структуры: 4, размер: 12 +—-+—-+—-+—-+—-+—-+—-+—-+—-+—-+—-+—-+ char c; | c | XX XX XX | i | s | XX XX | int i; +—-+—-+—-+—-+—-+—-+—-+—-+—-+—-+—-+—-+ short s; 0 4 8 ;

struct Выравнивание структуры: 4, размер: 8 +—-+—-+—-+—-+—-+—-+—-+—-+ int i; | i | c | XX | s | char c; +—-+—-+—-+—-+—-+—-+—-+—-+ short s; 0 4 6 ;

Обратите внимание на два последних примера: элементы структур одни и те же, только расположены в разном порядке. Но размер структур получился разный! Программа: вывод размера файла

Напишем программу, которая выводит размер файла. Для этого потребуется вызвать функцию stat(2) и прочитать данные из структуры, которую она заполнит. man 2 stat:

```
STAT(2)                  Системные вызовы                  STAT(2)

ИМЯ
       stat, fstat, lstat - получить статус файла

КРАТКАЯ СВОДКА
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <unistd.h>

       int stat(const char *file_name, struct stat *buf);

ОПИСАНИЕ
```

```
stat возвращает информацию о файле, заданном с помощью
file_name, и заполняет буфер buf.

Все эти функции возвращают структуру stat, которая содержит
такие поля:

   struct stat {
       dev_t         st_dev;     /* устройство                */
       ino_t         st_ino;     /* индексный дескриптор      */
       mode_t        st_mode;    /* режим доступа             */
       nlink_t       st_nlink;   /* количество жестких ссылок */
       uid_t         st_uid;     /* идентификатор
                                    пользователя-владельца    */
       gid_t         st_gid;     /* идентификатор
                                    группы-владельца          */
       dev_t         st_rdev;    /* тип устройства (если это
                                    устройство)               */
       off_t         st_size;    /* общий размер в байтах     */
       unsigned long st_blksize; /* размер блока ввода-вывода */
                                 /* в файловой системе        */
       unsigned long st_blocks;  /* количество выделенных
                                    блоков                    */
       time_t        st_atime;   /* время последнего доступа  */
       time_t        st_mtime;   /* время последнего
                                    изменения                 */
       time_t        st_ctime;   /* время последней смены
                                    состояния                 */
   };
```

Так, теперь осталось только вычислить смещение поля st_size... Но что это за типы — dev_t, ino_t? Какого они размера? Следует заглянуть в заголовочный файл и узнать, что обозначено при помощи typedef.

[user@host:]$ cpp /usr/include/sys/types.h | less

Далее, ищу в выводе препроцессора определение dev_t, нахожу:

typedef __dev_t dev_t;

Ищу __dev_t:

__extension__ typedef __u_quad_t __dev_t;

Ищу __u_quad_t:

__extension__ typedef unsigned long long int __u_quad_t;

Значит, sizeof(dev_t) = 8.

Мы бы могли и дальше продолжать искать, но в реальности всё немного по-другому. Если вы посмотрите на определение struct stat (cpp /usr/include/sys/stat.h | less), вы увидите поля с именами __pad1, __pad2, __unused4 и другие (зависит от системы). Эти поля не используются, они нужны для совместимости, и поэтому в man они не описаны. Так что самый верный способ не ошибиться — это просто попросить компилятор Си посчитать это смещение для нас (вычитаем из адреса поля адрес структуры, получаем смещение):

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  struct stat t;
  printf("sizeof = %zu, offset = %td\n",
         sizeof(t),
         ((void *) &t.st_size) - ((void *) &t));
  return 0;
```

```
}
```

На моей системе программа напечатала sizeof = 88, offset = 44. На вашей системе это значение может отличаться по описанным причинам. Теперь у нас есть все нужные данные об этой структуре, пишем программу:

```
.data
str_usage:
        .string "usage: %s filename\n"

printf_format:
        .string "%u\n"

.text
.globl main
main:
        pushl %ebp
        movl  %esp, %ebp

        subl  $88, %esp         /* выделить 88 байт под struct stat  */

        cmpl  $2, 8(%ebp)       /* argc == 2?                        */
        je    args_ok
                                /* программе передали не 2 аргумента,
                                   вывести usage                     */
        movl  12(%ebp), %ebx    /* поместить в %ebx адрес массива argv
                                                                      */
        pushl (%ebx)            /* argv[0]                           */
        pushl $str_usage
        call  printf

        movl  $1, %eax          /* выйти с кодом 1                   */
        jmp   return

args_ok:
        leal  -88(%ebp), %ebx   /* поместить адрес структуры в
                                   регистр %ebx                      */
        pushl %ebx

        movl  12(%ebp), %ecx    /* поместить в %ecx адрес массива argv
                                                                      */
        pushl 4(%ecx)           /* argv[1] - имя файла               */
        call  stat

        cmpl  $0, %eax          /* stat() вернул 0?                  */
        je    stat_ok

        /* stat() вернул ошибку, нужно вызвать perror(argv[1]) и
           завершить программу */

        movl  12(%ebp), %ecx
        pushl 4(%ecx)
        call  perror

        movl  $1, %eax
        jmp   return

stat_ok:
        pushl 44(%ebx)          /* нужное нам поле по смещению 44    */
```

```
        pushl $printf_format
        call  printf

        movl  $0, %eax          /* выйти с кодом 0                    */

return:
        movl  %ebp, %esp
        popl  %ebp
        ret
```

Обратите внимание на обработку ошибок: если передано не 2 аргумента — выводим описание использования программы и выходим, если stat(2) вернул ошибку — выводим сообщение об ошибке и выходим.

Наверное, могут возникнуть некоторые сложности с пониманием, как расположены argc и argv в стеке. Допустим, вы запустили программу как

[user@host:]$ ./program test-file

Тогда стек будет выглядеть приблизительно так:

. . . . . +————————+ 0x0000EFE4 <– %ebp - 88 | struct stat | +————————+ 0x0000F040 <– %ebp | старое значение %ebp | +————————+ 0x0000F044 <– %ebp + 4 | адрес возврата | +————————+ 0x0000F048 <– %ebp + 8 | argc | +————————+ 0x0000F04C <– %ebp + 12 | указатель на argv[0] | ————————- +————————+ 0x0000F050 <– %ebp + 16 | . . ——— . . | . . V +———-+ +———-+ | argv[0] | —> | "./program +———-+ +———-+ | argv[1] | - +———-+ +———-+ | argv[2] = 0 | -> | "test-file +———-+ +———-+

Таким образом, в стек помещается два параметра: argc и указатель на первый элемент массива argv[]. Где-то в памяти расположен блок из трёх указателей: указатель на строку "./program указатель на строку "test-file"и указатель NULL. Нам в стеке передали адрес этого блока памяти. Программа: печать файла наоборот

Напишем программу, которая читает со стандартного ввода всё до конца файла, а потом выводит введённые строки в обратном порядке. Для этого мы во время чтения будем помещать строки в связный список, а потом пройдем этот список в обратном порядке и напечатаем строки.

Внимание! Сохраните исходный код этой программы в файл с расширением .S — S в верхнем регистре.

```
.data
printf_format:
        .string "<%s>\n"

#define READ_CHUNK 128

.text

/* char *read_str(int *is_eof) */
read_str:
        pushl %ebp
        movl  %esp, %ebp

        pushl %ebx              /* сохранить регистры                 */
        pushl %esi
        pushl %edi

        movl  $0,   %ebx        /* прочитано байт                     */
        movl  $READ_CHUNK, %edi /* размер буфера                      */
        pushl %edi
        call  malloc
        addl  $4, %esp          /* убрать аргументы                   */
        movl  %eax, %esi        /* указатель на начало буфера         */
        decl  %edi              /* в конце должен быть нулевой байт,
                                   зарезервировать место для него     */
```

```
        pushl stdin             /* fgetc() всегда будет вызываться с
                                   этим аргументом                  */

1: /* read_start */
        call  fgetc             /* прочитать 1 символ               */
        cmpl  $0xa, %eax        /* новая строка '\n'?               */
        je    2f                /* read_end                         */
        cmpl  $-1, %eax         /* конец файла?                     */
        je    4f                /* eof_yes                          */
        movb  %al, (%esi,%ebx,1)  /* записать прочитанный символ в
                                   буфер                            */
        incl  %ebx              /* инкрементировать счётчик
                                   прочитанных байт                 */
        cmpl  %edi, %ebx        /* буфер заполнен?                  */
        jne   1b                /* read_start                       */

        addl  $READ_CHUNK, %edi /* увеличить размер буфера          */
        pushl %edi              /* размер                           */
        pushl %esi              /* указатель на буфер               */
        call  realloc
        addl  $8, %esp          /* убрать аргументы                 */
        movl  %eax, %esi        /* результат в %eax - новый указатель
                                                                    */
        jmp   1b                /* read_start                       */
2: /* read_end */

3: /* eof_no */
        movl  8(%ebp), %eax     /* *is_eof = 0                      */
        movl  $0, (%eax)
        jmp   5f                /* eof_end                          */
4: /* eof_yes */
        movl  8(%ebp), %eax     /* *is_eof = 1                      */
        movl  $1, (%eax)
5: /* eof_end */

        movb  $0, (%esi,%ebx,1) /* записать в конец буфера '\0'     */

        movl  %esi, %eax        /* результат в %eax                 */

        addl  $4, %esp          /* убрать аргумент fgetc()          */

        popl  %edi              /* восстановить регистры            */
        popl  %esi
        popl  %ebx

        movl  %ebp, %esp
        popl  %ebp
        ret

/*
struct list_node
{
  struct list_node *prev;
  char *str;
};
*/

.globl main
```

```
main:
        pushl %ebp
        movl  %esp, %ebp

        subl  $4, %esp          /* int is_eof;                        */

        movl  $0, %edi          /* в %edi будет храниться указатель на
                                   предыдущую структуру              */

1: /* read_start */
        leal  -4(%ebp), %eax    /* %eax = &is_eof;                    */
        pushl %eax
        call  read_str
        movl  %eax, %esi        /* указатель на прочитанную строку
                                   поместить в %esi                   */

        pushl $8                /* выделить 8 байт под структуру      */
        call  malloc

        movl  %edi, (%eax)      /* указатель на предыдущую структуру */
        movl  %esi, 4(%eax)     /* указатель на строку                */

        movl  %eax, %edi        /* теперь эта структура - предыдущая */

        addl  $8, %esp          /* убрать аргументы                   */

        cmpl  $0, -4(%ebp)      /* is_eof == 0?                       */
        jne   2f
        jmp   1b
2: /* read_end */


3: /* print_start */
                                /* просматривать список в обратном
                                   порядке, так что в %edi адрес
                                   текущей структуры                  */
        pushl 4(%edi)           /* указатель на строку из текущей
                                   структуры                          */
        pushl $printf_format
        call  printf            /* вывести на экран                   */

        addl  $4, %esp          /* убрать из стека только
                                   $printf_format                     */
        call  free              /* освободить память, занимаемую
                                   строкой                            */

        pushl %edi              /* указатель на структуру для
                                   освобождения памяти                */
        movl  (%edi), %edi      /* заменить указатель в %edi на
                                   следующий                          */
        call  free              /* освободить память, занимаемую
                                   структурой                         */

        addl  $8, %esp          /* убрать аргументы                   */

        cmpl  $0, %edi          /* адрес новой структуры == NULL?     */
        je    4f
        jmp   3b
```

```
4: /* print_end */

        movl  $0, %eax          /* выйти с кодом 0                    */

return:
        movl  %ebp, %esp
        popl  %ebp
        ret
```

Конструкция switch

Оператор switch языка Си можно переписать на ассемблере разными способами. Рассмотрим несколько вариантов того, какими могут быть значения у case:

значения из определённого маленького промежутка (все или почти все), например, 23, 24, 25, 27, 29, 30; значения, между которыми большие «расстояния» на числовой прямой, например, 5, 15, 80, 3800; комбинированный вариант: 35, 36, 37, 38, 39, 1200, 1600, 7000.

Рассмотрим решение для первого случая. Вспомним, что команда jmp принимает адрес не только в виде непосредственного значения (метки), но и как обращение к памяти. Значит, мы можем осуществлять переход на адрес, вычисленный в процессе выполнения. Теперь вопрос: как можно вычислить адрес? А нам не нужно ничего вычислять, мы просто поместим все адреса case-веток в массив. Пользуясь проверяемым значением как индексом массива, выбираем нужный адрес case-ветки. Таким образом, процессор всё вычислит за нас. Посмотрите на следующий код:

```
.data
printf_format:
        .string "%u\n"

.text
.globl main

main:
        pushl %ebp
        movl  %esp, %ebp

        movl  $1, %eax          /* получить в %eax некоторое
                                   интересующее нас значение         */

                                /* мы предусмотрели случаи только для
                                   0, 1, 3, поэтому,                 */
        cmpl  $3, %eax          /* если %eax больше 3
                                   (как беззнаковое),                */
        ja    case_default      /* перейти к default                 */

        jmp   *jump_table(,%eax,4) /* перейти по адресу, содержащемуся
                                   в памяти jump_table + %eax*4      */

.section .rodata
        .p2align 4
jump_table:                     /* массив адресов                    */
        .long case_0            /* адрес этого элемента массива:
                                                        jump_table + 0  */
        .long case_1            /*                     jump_table + 4  */
        .long case_default      /*                     jump_table + 8  */
        .long case_3            /*                     jump_table + 12 */
.text

case_0:
        movl  $5, %ecx          /* тело case-блока                   */
        jmp   switch_end        /* имитация break - переход в конец
                                   switch                            */
```

```
case_1:
        movl  $15, %ecx
        jmp   switch_end

case_3:
        movl  $35, %ecx
        jmp   switch_end

case_default:
        movl  $100, %ecx

switch_end:

        pushl %ecx              /* вывести %ecx на экран, выйти      */
        pushl $printf_format
        call  printf

        movl  $0, %eax

        movl  %ebp, %esp
        popl  %ebp
        ret
```

Этот код эквивалентен следующему коду на Си:

```
#include <stdio.h>

int main()
{
  unsigned int a, c;

  a = 1;
  switch(a)
  {
    case 0:
      c = 5;
      break;

    case 1:
      c = 15;
      break;

    case 3:
      c = 35;
      break;

    default:
      c = 100;
      break;
  }

  printf("%u\n", c);
  return 0;
}
```

Смотрите: в секции .rodata (данные только для чтения) создаётся массив из 4 значений. Мы обращаемся к нему как к обычному массиву, индексируя его по %eax: jump_table(,%eax,4). Но зачем перед этим стоит звёздочка? Она означает, что мы хотим перейти по адресу, содержащемуся в памяти по адресу

jump_table(,%eax,4) (если бы её не было, мы бы перешли по этому адресу и начали исполнять массив jump_table как код).

Заметьте, что тут нам понадобились значения 0, 1, 3, укладывающиеся в маленький промежуток [0; 3]. Так как для значения 2 не предусмотрено особой обработки, в массиве адресов jump_table индексу 2 соответствует case_default. Перед тем, как сделать jmp, нужно обязательно убедиться, что проверяемое значение входит в наш промежуток, и если не входит — перейти на default. Если вы этого не сделаете, то, когда попадётся значение, находящееся за пределами массива, программа, в лучшем случае, получит segmentation fault, а в худшем (если рядом с этим масивом адресов в памяти окажется еще один массив адресов) код продолжит исполнение вообще непонятно где.

Теперь рассмотрим случай, когда значения для веток case находятся на большом расстоянии друг от друга. Очевидно, что способ с массивом адресов не подходит, иначе массив занимал бы большое количество памяти и содержал в основном адреса ветки default. В этом случае лучшее, что может сделать программист, — выразить switch как последовательное сравнение со всеми перечисленными значениями. Если значений довольно много, придётся применить немного логики: приблизительно прикинуть, какие ветки будут исполняться чаще всего, и отсортировать их в таком порядке в коде. Это нужно для того, чтобы наиболее часто исполняемые ветки исполнялись после маленького числа сравнений. Допустим, у нас есть варианты 5, 38, 70 и 1400, причём 70 будет появляться чаще всего:

```
.data
printf_format:
        .string "%u\n"

.text
.globl main

main:
        pushl %ebp
        movl  %esp, %ebp

        movl  $70, %eax         /* получить в %eax некоторое
                                   интересующее нас значение         */

        cmpl  $70, %eax
        je    case_70

        cmpl  $5, %eax
        je    case_5

        cmpl  $38, %eax
        je    case_38

        cmpl  $1400, %eax
        je    case_1400

case_default:
        movl  $100, %ecx
        jmp   switch_end

case_5:
        movl  $5, %ecx
        jmp   switch_end

case_38:
        movl  $15, %ecx
        jmp   switch_end

case_70:
        movl  $25, %ecx
        jmp   switch_end
```

```
case_1400:
        movl  $35, %ecx

switch_end:

        pushl %ecx

        pushl $printf_format
        call  printf

        movl  $0, %eax

        movl  %ebp, %esp
        popl  %ebp
        ret
```

Единственное, на что хочется обратить внимание, — на расположение ветки default: если все сравнения оказались ложными, код default выполняется автоматически.

Наконец, третий, комбинированный, вариант. Путь имеем варианты 35, 36, 37, 39, 1200, 1600 и 7000. Тогда мы видим промежуток [35; 39] и ещё три числа. Код будет выглядеть приблизительно так:

```
        movl  $1, %eax          /* получить в %eax некоторое
                                   интересующее нас значение         */

        cmpl  $35, %eax
        jb    case_default

        cmpl  $39, %eax
        ja    switch_compare

        jmp   *jump_table-140(,%eax,4)

.section .rodata
        .p2align 4
jump_table:
        .long case_35
        .long case_36
        .long case_37
        .long case_default
        .long case_39
.text

switch_compare:
        cmpl  $1200, %eax
        jmp   case_1200

        cmpl  $1600, %eax
        jmp   case_1600

        cmpl  $7000, %eax
        jmp   case_7000

case_default:
        /* ... */
        jmp   switch_end

case_35:
        /* ... */
```

```
        jmp   switch_end

        ... ещё код ...
switch_end:
```

Заметьте, что промежуток начинается с числа 35, а не с 0. Для того, чтобы не производить вычитание 35 отдельной командой и не создавать массив, в котором от 0 до 34 идёт адреса метки default, сначала проверяется принадлежность числа промежутку [35; 39], а затем производится переход, но массив адресов считается размещённым на 35 двойных слов «ниже» в памяти (то есть, на 35 4 = 140 байт). В результате получается, что адрес перехода считывается из памяти по адресу jump_table - 35*4 + %eax*4 = jump_table + (%eax - 35)*4. Выиграли одно вычитание.