Ассемблер в Linux для программистов С

Введение

> Premature optimization is the root of all evil.
>
> Donald Knuth

Эта книга ориентирована на программистов, которые уже знают Си на достаточном уровне. Почему так? Вряд ли, зная только несколько интерпретируемых языков вроде Perl или Python, кто-то захочет сразу изучать ассемблер. Используя Си и ассемблер вместе, применяя каждый язык для определённых целей, можно добиться очень хороших результатов. К тому же программисты Си уже имеют некоторые знания об архитектуре процессора, особенностях машинных вычислений, способе организации памяти и других вещах, которые новичку в программировании понять не так просто. Поэтому изучать ассемблер после Си несомненно легче, чем после других языков высокого уровня. В Си есть понятие «указатель», программист должен сам управлять выделением памяти в куче, и так далее — все эти знания пригодятся при изучении ассемблера, они помогут получить более целостную картину об архитектуре, а также иметь более полное представление о том, как выполняются их программы на Си. Но эти знания требуют углубления и структурирования.

Хочу подчеркнуть, что для чтения этой книги никаких знаний о Linux не требуется (кроме, разумеется, знаний о том, «как создать текстовый файл» и «как запустить программу в консоли»). Да и вообще, единственное, в чём выражается ориентированность на Linux, — это используемые синтаксис ассемблера и ABI. Программисты на ассемблере в DOS и Windows используют синтаксис Intel, но в системах *nix принято использовать синтаксис AT&T. Именно синтаксисом AT&T написаны ассемблерные части ядра Linux, в синтаксисе AT&T компилятор GCC выводит ассемблерные листинги и так далее.

Большую часть информации из этой книги можно использовать для программирования не только в *nix, но и в Windows, нужно только уточнить некоторые системно-зависимые особенности (например, ABI).

А стоит ли?

При написании кода на ассемблере всегда следует отдавать себе отчёт в том, действительно ли данный кусок кода должен быть написан на ассемблере. Нужно взвесить все «за» и «против», так как современные компиляторы в подавляющем большинстве случаев оптимизируют код гораздо лучше, чем это может сделать программист вручную на ассемблере.

Как править этот викиучебник

Так как изначально этот учебник писался не в вики-формате, автор допускал повествование от первого лица. В вики такое не приветствуется, поэтому такие обороты нужно вычистить.

При внесении первых правок насчёт архитектуры x86_64 (сейчас эта тема не освещена вообще) нужно разграничить и чётко отметить все архитектурно-зависимые абзацы: что относится к IA-32, а что к x86_64, так как ABI (application binary interface) i386 и x86_64 отличаются.

Архитектура

x86 или IA-32?

Вы, вероятно, уже слышали такое понятие, как «архитектура x86». Вообще оно довольно размыто, и вот почему. Само название x86 или 80x86 происходит от принципа, по которому Intel давала названия своим процессорам:

- Intel 8086 — 16 бит;
- Intel 80186 — 16 бит;
- Intel 80286 — 16 бит;
- Intel 80386 — 32 бита;
- Intel 80486 — 32 бита.

Этот список можно продолжить. Принцип наименования, где каждому поколению процессоров давалось имя, заканчивающееся на 86, создал термин «x86». Но, если посмотреть внимательнее, можно увидеть, что «процессором x86» можно назвать и древний 16-битный 8086, и новый Core i7. Поэтому 32-битные расширения были названы архитектурой IA-32 (сокращение от Intel Architecture, 32-bit). Конечно же, возможность запуска 16-битных программ осталась, и она успешно (и не очень) используется в 32-битных версиях Windows. Так как Linux является полностью 32-битной операционной системой, мы будем рассматривать только 32-битный режим.

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. Более подробно см. [1].

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». Например, к младшим 16 битам регистра `%eax` можно обратиться как `%ax`. `%ax`, в свою очередь, содержит две однобайтовых половинки, которые могут использоваться как самостоятельные регистры: старший `%ah` и младший `%al`. Аналогично можно обращаться к `%ebx`/`%bx`/`%bh`/`%bl`, `%ecx`/`%cx`/`%ch`/`%cl`, `%edx`/`%dx`/`%dh`/`%dl`, `%esi`/`%si`, `%edi`/`%di`.

Не следует бояться такого жёсткого закрепления назначения использования регистров. Большая их часть может использоваться для хранения совершенно произвольных данных. Единственный случай, когда нужно учитывать, в какой регистр помещать данные — использование неявно обращающихся к регистрам команд. Такое поведение всегда чётко документировано.

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.

В ОС Linux используется плоская модель памяти (flat memory model), в которой все сегменты описаны как использующие всё адресное пространство процессора и, как правило, явно не используются, а все адреса представлены в виде 32-битных смещений. В большинстве случаев программисту можно даже и не задумываться об их существовании, однако операционная система предоставляет специальные средства (системный вызов `modify_ldt()`), позволяющие описывать нестандартные сегменты и работать с ними. Однако такая потребность возникает редко, поэтому тут подробно не рассматривается.

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.

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.

Стек

Мы полагаем, что читатель имеет опыт программирования на Си и знаком со структурами данных типа стек. В микропроцессоре стек работает похожим образом: это область памяти, у которой определена вершина (на неё указывает `%esp`). Поместить новый элемент можно только на вершину стека, при этом новый элемент становится вершиной. Достать из стека можно только верхний элемент, при этом вершиной становится следующий элемент. У вас наверняка была в детстве игрушка-пирамидка, где нужно было разноцветные кольца надевать на общий стержень. Так вот, эта пирамидка — отличный пример стека. Также можно провести аналогию с составленными стопкой тарелками.

```
Содержимое стека  Адреса в памяти


.                 .
.                 .
.                 .
+-----------------+ 0x0000F040
|                 |
+-----------------+ 0x0000F044 <-- вершина стека (на неё указывает %esp)
|     данные      |
+-----------------+ 0x0000F048
|     данные      |
+-----------------+ 0x0000F04C
.                 .
.                 .
.                 .
+-----------------+ 0x0000FFF8
|     данные      |
+-----------------+ 0x0000FFFC
|     данные      |
+-----------------+ 0x00010000 <-- дно стека
```

Стек растёт в сторону младших адресов. Это значит, что последний записанный в стек элемент будет расположен по адресу младше остальных элементов стека.

При помещении нового элемента в стек происходит следующее (принцип работы команды `push`):

- значение `%esp` уменьшается на размер элемента в байтах (4 или 2);
- новый элемент записывается по адресу, на который указывает `%esp`.

```
.                 .
.                 .
.                 .
+-----------------+ 0x0000F040 <-- новая вершина стека (%esp)
|  новый элемент  |
+-----------------+ 0x0000F044 <-- старая вершина стека
|     данные      |
+-----------------+ 0x0000F048
```

```
.                .
.                .
.                .
+----------------+ 0x0000FFFC
|     данные     |
+----------------+ 0x00010000 <-- дно стека
```

При выталкивании элемента из стека эти действия совершаются в обратном порядке(принцип работы команды `pop`):

- элемент, на который указывает `%esp`, записывается в регистр;
- значение `%esp` увеличивается на размер элемента в байтах (4 или 2).

```
.                .
.                .
.                .
+----------------+ 0x0000F040 <-- старая вершина стека
| верхний элемент | -------------> записывается в регистр
+----------------+ 0x0000F044 <-- новая вершина стека (%esp)
|      данные     |
+----------------+ 0x0000F048
.                .
.                .
.                .
+----------------+ 0x0000FFFC
|      данные     |
+----------------+ 0x00010000 <-- дно стека
```

Память

В Си после вызова `malloc(3)` программе выделяется блок памяти, и к нему можно получить доступ при помощи указателя, содержащего адрес этого блока. В ассемблере то же самое: после того, как программе выделили блок памяти, появляется возможность использовать указывающий на неё адрес для всевозможных манипуляций. Наименьший по размеру элемент памяти, на который может указать адрес, — байт. Говорят, что память адресуется побайтово, или гранулярность адресации памяти — один байт. Отдельный бит можно указать как адрес байта, содержащего этот бит, и номер этого бита в байте.

Правда, нужно отметить ещё одну деталь. Программный код расположен в памяти, поэтому получить его адрес также возможно. Стек — это тоже блок памяти, и разработчик может получить указатель на любой элемент стека, находящийся под вершиной. Таким образом организовывают доступ к произвольным элементам стека.

Порядок байтов. Little-endian и big-endian

Оперативная память — это массив битовых значений, 0 и 1. Не будем говорить о порядке битов в байте, так как указать адрес отдельного бита невозможно; можно указать только адрес байта, содержащего этот бит. А как в памяти располагаются байты в слове? Предположим, что у нас есть число `0x01020304`. Его можно записать в виде байтовой последовательности:

начиная со старшего байта: `0x01 0x02 0x03 0x04` — big-endian

начиная с младшего байта: `0x04 0x03 0x02 0x01` — little-endian

Вот эта байтовая последовательность располагается в оперативной памяти, адрес всего слова в памяти — адрес первого байта последовательности.

Если первым располагается младший байт (запись начинается с «меньшего конца») — такой порядок байт называется little-endian, или «интеловским». Именно он используется в процессорах x86.

Если первым располагается старший байт (запись начинается с «большего конца») — такой порядок байт называется big-endian.

У порядка little-endian есть одно важное достоинство. Посмотрите на запись числа `0x00000033`:

```
0x33 0x00 0x00 0x00
```

Если прочесть его как двухбайтовое значение, получим `0x0033`. Если прочесть как однобайтовое, получим `0x33`. При записи этот трюк тоже работает. Конечно же, мы не можем прочитать число `0x11223344` как байт, потому что получим `0x44`, что неверно. Поэтому считываемое число должно помещаться в целевой диапазон значений.

Hello, world!

При изучении нового языка принято писать самой первой программу, выводящую на экран строку `Hello, world!`. Сейчас мы не ставим перед собой задачу понять всё написанное. Главное — посмотреть, как оформляются программы на ассемблере, и научиться их компилировать.

Вспомним, как вы писали `Hello, world!` на Си. Скорее всего, приблизительно так:

```c
#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`.

```
.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 С. Теперь запускаем нашу программу и убеждаемся, что она корректно завершилась с кодом возврата 0.

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

Теперь было бы хорошо прочитать главу про отладчик GDB. Он вам понадобится для исследования работы ваших программ. Возможно, сейчас вы не всё поймёте, но эта глава специально расположена в конце, так как задумана больше как справочная, нежели обучающая. Для того, чтобы научиться работать с отладчиком, с ним нужно просто работать.

Синтаксис ассемблера

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
```

Для того, чтобы привести пример команды, я, немного забегая наперед, расскажу об одной операции. Команда `mov` *`источник, назначение`* производит копирование *источника* в *назначение*. Возьмем строку из `hello.s`:

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

Важной особенностью всех команд является то, что они не могут работать с двумя операндами, находящимися в памяти. Хотя бы один из них следует сначала загрузить в регистр, а затем выполнять необходимую операцию.

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
        ...
```

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"
```

Приведём небольшую таблицу, в которой сопоставляются типы данных в Си на IA-32 и в ассемблере. Нужно заметить, что размер этих типов в языке Си на других архитектурах (или даже компиляторах) может отличаться.

Тип данных в Си	Размер (sizeof), байт	Выравнивание, байт	Название
char signed char	1	1	signed byte (байт со знаком)
unsigned char	1	1	unsigned byte (байт без знака)
short signed short	2	2	signed halfword (полуслово со знаком)
unsigned short	2	2	unsigned halfword (полуслово без знака)
int signed int long signed long enum	4	4	signed word (слово со знаком)
unsigned int unsigned long	4	4	unsigned word (слово без знака)

Отдельных объяснений требует колонка «Выравнивание». Выравнивание задано у каждого фундаментального типа данных (типа данных, которым процессор может оперировать непосредственно). Например, выравнивание word — 4 байта. Это значит, что данные типа word должны располагаться по адресу, кратному 4 (например, 0x00000100, 0x03284478). Архитектура рекомендует, но не требует выравнивания: доступ к невыровненным данным может быть медленнее, но принципиальной разницы нет и ошибки это не вызовет.

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
```

Метки и прочие символы

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).

Значение метки как константы — это всегда адрес. А если вам нужна константа с каким-то другим значением? Тогда мы приходим к более общему понятию «символ». Символ — это просто некоторая константа. Причём он может быть определён в одном файле, а использован в других.

Возьмём `hello.s` и скомпилируем его так:

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

Обратите внимание на параметр `-c`. Мы компилируем исходный код не в исполняемый файл, а лишь только в отдельный объектный файл `hello.o`. Теперь воспользуемся программой `nm(1)`:

```
[user@host:~]$ nm hello.o
00000000 d hello_str
0000000e a hello_str_length
00000000 T main
```

`nm(1)` выводит список символов в объектном файле. В первой колонке выводится значение символа, во второй — его тип, в третьей — имя. Посмотрим на символ `hello_str_length`. Это длина строки `Hello, world!\n`. Значение символа чётко определено и равно `0xe`, об этом говорит тип `a` — absolute value. А вот символ `hello_str` имеет тип `d` — значит, он находится в секции данных (data). Символ `main` находится в секции кода (text section, тип `T`). А почему `a` представлено строчной буквой, а `T` — прописной? Если тип символа обозначен строчной буквой, значит это локальный символ, который видно только в пределах данного файла. Заглавная буква говорит о том, что символ глобальный и доступен другим модулям. Символ `main` мы сделали глобальным при помощи директивы `.globl main`.

Для создания нового символа используется директива `.set`. Синтаксис:

```
 .set    символ, выражение
```

Например, определим символ `foo` = 42:

```
.set    foo, 42
```

Ещё пример из `hello.s`:

```
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).

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
```

Методы адресации

Пространство памяти предназначено для хранения кодов команд и данных, для доступа к которым имеется богатый выбор методов адресации (около 24). Операнды могут находиться во внутренних регистрах процессора (наиболее удобный и быстрый вариант). Они могут располагаться в системной памяти (самый распространенный вариант). Наконец, они могут находиться в устройствах ввода/вывода (наиболее редкий случай). Определение местоположения операндов производится кодом команды. Причем существуют разные методы, с помощью которых код команды может определить, откуда брать входной операнд и куда помещать выходной операнд. Эти методы называются методами адресации. Эффективность выбранных методов адресации во многом определяет эффективность работы всего процессора в целом.

Прямая или абсолютная адресация

Физический адрес операнда содержится в адресной части команды. Формальное обозначение:

$Операнд_i = (A_i)$

где A_i – код, содержащийся в i-м адресном поле команды.

```
.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

В команде содержится не адрес операнда, а непосредственно сам операнд.

$Операнд_i = A_i$.

Непосредственная адресация позволяет повысить скорость выполнения операции, так как в этом случае вся команда, включая операнд, считывается из памяти одновременно и на время выполнения команды хранится в процессоре в специальном регистре команд (РК). Однако при использовании непосредственной адресации появляется зависимость кодов команд от данных, что требует изменения программы при каждом изменении непосредственного операнда.

Пример:

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

Косвенная (базовая) адресация

Адресная часть команды указывает адрес ячейки памяти или регистр, в котором содержится адрес операнда:

$Операнд_i = ((A_i))$

Применение косвенной адресации операнда из оперативной памяти при хранении его адреса в регистровой памяти существенно сокращает длину поля адреса, одновременно сохраняя возможность использовать для указания физического адреса полную разрядность регистра. Недостаток этого способа – необходимо дополнительное время для чтения адреса операнда. Вместе с тем он существенно повышает гибкость программирования. Изменяя содержимое ячейки памяти или регистра, через которые осуществляется адресация, можно, не меняя команды в программе, обрабатывать операнды, хранящиеся по разным адресам.

Косвенная адресация не применяется по отношению к операндам, находящимся в регистровой памяти.

Пример:

```
.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  */
```

Предоставляемые косвенной адресацией возможности могут быть расширены, если в системе команд ЭВМ предусмотреть определенные арифметические и логические операции над ячейкой памяти или регистром, через которые выполняется адресация, например увеличение или уменьшение их значения.

Автоинкрементная и автодекрементная адресация

Иногда, адресация, при которой после каждого обращения по заданному адресу с использованием механизма косвенной адресации, значение адресной ячейки автоматически увеличивается на длину считываемого операнда, называется автоинкрементной. Адресация с автоматическим уменьшением значения адресной ячейки называется автодекрементной.

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:

$Операнд_i = (база_i + смещение_i)$

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:

1. 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.
2. 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.

Главный недостаток относительной адресации – большое время вычисления физического адреса операнда. Но существенное преимущество этого способа адресации заключается в возможности создания "перемещаемых" программ – программ, которые можно размещать в различных частях памяти без изменения команд программы. То же относится к программам, обрабатывающим по единому алгоритму информацию, расположенную в различных областях ЗУ. В этих случаях достаточно изменить содержимое базового адреса начала команд программы или массива данных, а не модифицировать сами команды. По этой причине относительная адресация облегчает распределение памяти при составлении сложных программ и широко используется при автоматическом распределении памяти в мультипрограммных вычислительных системах.

Команды ассемблера

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 её адресу. Например:

```
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 [2] 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                     */
```

Давайте подумаем, каким будет результат выполнения следующего кода на Си:

```
char x, y;
x = 250;
y = 14;
x = x + y;
printf("%d", (int) x);
```

Большинство сразу скажет, что результат (250 + 14 = 264) больше, чем может поместиться в одном байте. И что же напечатает программа? 8. Давайте рассмотрим, что происходит при сложении в двоичной системе.

```
  11111010       250
+ 00001110      + 14
----------       ---
1 00001000       264
 |        |
 |<------>|
    8 бит
```

Получается, что результат занимает 9 бит, а в переменную может поместиться только 8 бит. Это называется переполнением — перенос из старшего бита результата. В Си переполнение не может быть перехвачено, но в микропроцессоре эта ситуация регистрируется, и её можно обработать. Когда происходит переполнение, устанавливается флаг `cf`. Команды условного перехода `jc` и `jnc` анализируют состояние этого флага. Команды условного перехода будут рассмотрены далее, здесь эта информация приводится для полноты описания команд.

```
        movb  $0,   %ah         /* %ah = 0                              */
        movb  $250, %al         /* %al = 250                            */
        addb  $14,  %al         /* %al = %al + 14
                                   происходит переполнение,
                                   устанавливается флаг cf;
                                   в %al число 8                        */
        jnc   no_carry          /* если переполнения не было, перейти
                                   на метку                             */
        movb  $1,   %ah         /* %ah = 1                              */
no_carry:
        /* %ax = 264 = 0x0108 */
```

Этот код выдаёт правильную сумму в регистре `%ax` с учётом переполнения, если оно произошло. Попробуйте поменять числа в строках 2 и 3.

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.

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  $0, %eax          /* в %eax будет храниться результат;
                                   в начале наибольшее значение — 0  */
        movl  $array, %ebx      /* в %ebx находится адрес текущего
                                   элемента массива                  */
```

```
loop_start:                     /* начало цикла                        */
        cmpl  %eax, (%ebx)      /* сравнить текущий элемент массива с
                                   текущим наибольшим значением из %eax
                                                                       */
        jbe   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           */
        cmpl  $array_end, %ebx  /* сравнить адрес текущего элемента и
                                   адрес конца массива                 */
        je    loop_end          /* если они равны, выйти из цикла      */
        jmp   loop_start        /* иначе повторить цикл снова          */
loop_end:

/*
 * 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 0. После этого мы сравниваем каждый элемент массива с текущим наибольшим значением из `%eax`, и, если этот элемент больше, он становится текущим наибольшим. После просмотра всего массива в `%eax` находится наибольший элемент.

Этот код соответствует приблизительно следующему на Си:

```
#include <stdio.h>

int main()
{
  int array[] = { 10, 15, 148, 12, 151, 3, 72 };
  int *array_end = &array[sizeof(array) / sizeof(int)];
  int max = 0;
  int *p = array;

  do
  {
    if(*p > max)
```

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

  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:

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

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

array_size:
        .long (array_end - array)/4  /* количество элементов массива */

.text
.globl main
main:
        movl  $0, %eax          /* в %eax будет храниться результат;
                                   в начале наибольшее значение — 0  */
        movl  $0, %ecx          /* начать просмотр с нулевого элемента
                                                                      */

loop_start:                     /* начало цикла                      */
        cmpl  %eax, array(,%ecx,4)  /* сравнить текущий элемент
                                   массива с текущим наибольшим
                                   значением из %eax                 */
        jbe   less              /* если текущий элемент массива меньше
                                   или равен наибольшему, пропустить
                                   следующий код                     */
        movl  array(,%ecx,4), %eax  /* а вот если элемент массива
                                   превосходит наибольший, значит, его
                                   значение и есть новый максимум    */
less:
        incl  %ecx              /* увеличить на 1 номер текущего
                                   элемента                          */
        cmpl  array_size, %ecx  /* сравнить номер текущего элемента с
                                   общим числом элементов            */
        je    loop_end          /* если они равны, выйти из цикла    */
        jmp   loop_start        /* иначе повторить цикл снова        */
```

```
loop_end:

/*
 * 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
```

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 */
        je    loop_end          /* подобрать соответствующую команду
                                   условного перехода для выхода из
                                   цикла                                */
        jmp   loop_start        /* иначе повторить цикл снова           */
loop_end:
```

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:

```
loop_start:                     /* начало цикла                         */
        cmpl  ...               /* что-то с чем-то сравнить для
                                   принятия решения о выходе из цикла */
        je    loop_end          /* подобрать соответствующую команду
                                   условного перехода для выхода из
                                   цикла                                */

        /* вот тут находится тело цикла */

        jmp   loop_start        /* перейти к проверке условия цикла    */
loop_end:
```

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

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

эквивалентен такой конструкции:

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

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 `&`, `|`, `^`. Эти команды устанавливают флаги согласно результату.

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[4].

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`[5].

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.

Принцип работы команды `shl`:

```
                          До сдвига:
          +---+     +----------------------------------+
          | ? |     | 10001000100010001000100010001011 |
          +---+     +----------------------------------+
         Флаг CF    Операнд

                       Сдвиг влево на 1 бит:
          +---+     +----------------------------------+
          | 1 | <-- | 00010001000100010001000100010110 | <-- 0
          +---+     +----------------------------------+
         Флаг CF    Операнд

                      Сдвиг влево на 3 бита:
 +----+   +---+     +----------------------------------+
 | 10 |   | 0 | <-- | 01000100010001000100010001011000 | <-- 000
 +----+   +---+     +----------------------------------+
Улетели  Флаг CF    Операнд
в никуда
```

Принцип работы команды `shr`:

```
                  До сдвига:
        +----------------------------------+     +---+
        | 10001000100010001000100010001011 |     | ? |
        +----------------------------------+     +---+
        Операнд                                  Флаг CF
```

```
          Логический сдвиг вправо на 1 бит:
        +----------------------------------+     +---+
  0 --> | 01000100010001000100010001000101 | --> | 1 |
        +----------------------------------+     +---+
        Операнд                                  Флаг CF

          Логический сдвиг вправо на 3 бита:
        +----------------------------------+     +---+  +----+
000 --> | 00010001000100010001000100010001 | --> | 0 |  | 11 |
        +----------------------------------+     +---+  +----+
        Операнд                                  Флаг CF Улетели
                                                         в никуда
```

Эти две команды называются командами логического сдвига, потому что они работают с операндом как с массивом бит. Каждый «выдвигаемый» бит попадает в флаг `cf`, причём с другой стороны операнда «вдвигается» бит 0. Таким образом, в флаге `cf` оказывается самый последний «выдвинутый» бит. Такое поведение вполне допустимо для работы с беззнаковыми числами, но числа со знаком будут обработаны неверно из-за того, что знаковый бит может быть потерян.

Для работы с числами со знаком существуют команды арифметического сдвига. Команды `shl` и `sal` выполняют полностью идентичные действия, так как при сдвиге влево знаковый бит не теряется (расширение знакового бита влево становится новым знаковым битом). Для сдвига вправо применяется команда `sar`. Она «вдвигает» слева знаковый бит исходного значения, таким образом сохраняя знак числа:

```
                 До сдвига:
        +----------------------------------+     +---+
        | 10001000100010001000100010001011 |     | ? |
        +----------------------------------+     +---+
        Операнд                                  Флаг CF
        старший бит равен 1 ==>
          ==> значение отрицательное ==>
          ==> "вдвинуть" бит 1 ---+
                                  |
  +-------------------------------+
  |
  V       Арифметический сдвиг вправо на 1 бит:
        +----------------------------------+     +---+
  1 --> | 11000100010001000100010001000101 | --> | 1 |
        +----------------------------------+     +---+
        Операнд                                  Флаг CF

          Арифметический сдвиг вправо на 3 бита:
        +----------------------------------+     +---+  +----+
111 --> | 11110001000100010001000100010001 | --> | 0 |  | 11 |
        +----------------------------------+     +---+  +----+
        Операнд                                  Флаг CF Улетели
                                                         в никуда
```

Многие программисты Си знают об умножении и делении на степени двойки (2, 4, 8…) при помощи сдвигов. Этот трюк отлично работает и в ассемблере, используйте его для оптимизации.

Кроме сдвигов обычных, существуют циклические сдвиги:

```
/* ROtate Right */
ror   количество_сдвигов, назначение

/* ROtate Left */
rol   количество_сдвигов, назначение
```

Объясню на примере циклического сдвига влево на три бита: три старших («левых») бита «выдвигаются» из регистра влево и «вдвигаются» в него справа. При этом в флаг `cf` записывается самый последний «выдвинутый» бит.

Принцип работы команды `rol`:

```
                          До сдвига:
     +---+         +---------------------------------+
     | ? |         | 10001000100010001000100010001011 |
     +---+         +---------------------------------+
    Флаг CF        Операнд

                    Циклический сдвиг влево на 1 бит:
     +---+  1    1 +---------------------------------+
     | 1 | <--+--- | 00010001000100010001000100010111 | ---+
     +---+    |    +---------------------------------+    |
    Флаг CF   V    Операнд                                 ^
              |                                            |
              +------------------->--->--->----------------+
               1

                    Циклический сдвиг влево на 3 бита:
     +---+  0  100 +---------------------------------+
     | 0 | <--+--- | 01000100010001000100010001011100 | ---+
     +---+    |    +---------------------------------+    |
    Флаг CF   V    Операнд                                 ^
              |                                            |
              +------------------->--->--->----------------+
               100
```

Принцип работы команды `ror`:

```
                 До сдвига:
       +---------------------------------+         +---+
       | 10001000100010001000100010001011 |         | ? |
       +---------------------------------+         +---+
       Операнд                                   Флаг CF

         Циклический сдвиг вправо на 1 бит:
       +---------------------------------+ 1    1  +---+
```

```
  +--- | 11000100010001000100010001000101 | ---+--> | 1 |
  |    +-----------------------------------+    |    +---+
  ^    Операнд                                  V   Флаг CF
  |                                             |
  +--------------------<---<---<----------------+
                                               1

        Циклический сдвиг вправо на 3 бита:
       +-----------------------------------+ 011  0  +---+
  +--- | 01110001000100010001000100010001 | ---+--> | 0 |
  |    +-----------------------------------+    |    +---+
  ^    Операнд                                  V   Флаг CF
  |                                             |
  +--------------------<---<---<----------------+
                                            011
```

Существует ещё один вид сдвигов — циклический сдвиг через флаг `cf`. Эти команды рассматривают флаг `cf` как продолжение операнда.

```
/* Rotate through Carry Right */
rcr   количество_сдвигов, назначение

/* Rotate through Carry Left */
rcl   количество_сдвигов, назначение
```

Принцип работы команды `rcl`:

```
                          До сдвига:
        +---+      +-----------------------------------+
        | X |      | 10001000100010001000100010001011 |
        +---+      +-----------------------------------+
       Флаг CF     Операнд

                    Циклический сдвиг влево через CF на 1 бит:
     X  +---+      +-----------------------------------+
   +-<- | 1 | <--- | 00010001000100010001000100010111X | ---+
   |    +---+      +-----------------------------------+    |
   V   Флаг CF     Операнд                                  ^
   |                                                        |
   +------------------------------>--->--->----------------+

                    Циклический сдвиг влево через CF на 3 бита:
    X10 +---+      +-----------------------------------+
   +-<- | 0 | <--- | 01000100010001000100010001011X10 | ---+
   |    +---+      +-----------------------------------+    |
   V   Флаг CF     Операнд                                  ^
   |                                                        |
   +------------------------------>--->--->----------------+
```

Принцип работы команды `rcr`:

```
                  До сдвига:
        +----------------------------------+      +---+
        | 10001000100010001000100010001011 |      | X |
        +----------------------------------+      +---+
        Операнд                                 Флаг CF

          Циклический сдвиг вправо через CF на 1 бит:
        +----------------------------------+      +---+  X
    +---| X10001000100010001000100010001000101 | ---> | 1 | ->-+
    |   +----------------------------------+      +---+    |
    ^   Операнд                                 Флаг CF   V
    |                                                     |
    +-------------------<---<---<---------------------------+

          Циклический сдвиг вправо через CF на 3 бита:
        +----------------------------------+      +---+ 11X
    +---| 11X10001000100010001000100010001 | ---> | 0 | ->-+
    |   +----------------------------------+      +---+    |
    ^   Операнд                                 Флаг CF   V
    |                                                     |
    +-------------------<---<---<---------------------------+
```

Эти сложные циклические сдвиги вам редко понадобятся в реальной работе, но уже сейчас нужно знать, что такие инструкции существуют, чтобы не изобретать велосипед потом. Ведь в языке Си циклический сдвиг производится приблизительно так:

```
int main()
{
  int a = 0x11223344;
  int shift_count = 8;

  a = (a << shift_count) | (a >> (32 - shift_count));

  printf("%x\n", a);
  return 0;
}
```

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.

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.

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:
$0! = 1, \quad n! = n \cdot (n-1)!$

```
.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: $n! = 1 \cdot 2 \cdot \ldots \cdot n$

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.

Автор увлёкся процессом и написал 64-битную версию этой функции. Она возвращает результат в паре `%eax:%edx` и может вычислить $20! = 2432902008176640000$.

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

.text
.type   factorial, @function    /* long long int factorial(int)      */
factorial:
        movl  4(%esp), %ecx

        cmpl  $0, %ecx
        jne   not_zero

        movl  $1, %eax
        ret

not_zero:

        movl  $1, %esi          /* младшие 32 бита                   */
        movl  $0, %edi          /* старшие 32 бита                   */
loop_start:
        movl  %esi, %eax        /* загрузить младшие биты для
                                   умножения                         */
        mull  %ecx              /* %eax:%edx = младшие биты * %ecx   */
        movl  %eax, %esi        /* записать младшие биты
                                   обратно в %esi                    */

        movl  %edi, %eax        /* загрузить старшие биты            */
        movl  %edx, %edi        /* записать в %edi старшие биты
                                   предыдущего умножения; теперь
                                   результат умножения младших битов
                                   находится в %esi:%edi, а старшие
                                   биты — в %eax для следующего
                                   умножения                         */
        mull  %ecx              /* %eax:%edx = старшие биты * %ecx   */
        addl  %eax, %edi        /* сложить полученный результат со
                                   старшими битами предыдущего
                                   умножения                         */

        loop  loop_start

        movl  %esi, %eax        /* результат вернуть в паре          */
        movl  %edi, %edx        /* %eax:%edx                         */

        ret
```

```
.size   factorial, .-factorial

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

        pushl $20
        call  factorial

        pushl %edx
        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
```

Умножение 64-битного числа на 32-битное делается как при умножении «в столбик»:

```
     %edi       %esi
×               %ecx
--------------------
%edi×%ecx  %esi×%ecx
    A           |
   /|\          |
    +--<--<--<--+
   старшие 32 бита
```

Но произведение `%esi` × `%ecx` не поместится в 32 бита, останутся ещё старшие 32 бита. Их мы должны прибавить к старшим 32-м битам результата. Приблизительно так вы это делаете на бумаге в десятичной системе:

```
  2  5      25 × 3 = 75
×    3
  ----
    15
+ 6
  ----
  7  5
```

Задание: напишите программу-считалочку. Есть числа от 0 до m, которые располагаются по кругу. Счёт начинается с элемента 0. Каждый n-й элемент удаляют. Счёт продолжается с элемента, следующего за удалённым. Напишите программу, выводящую список вычеркнутых элементов. Подсказка: используйте

`malloc(3)` для получения $m+1$байт памяти и занесите в каждый байт число 1 при помощи `memset(3)`. Значение 1 означает, что элемент существует, значением 0 отмечайте удалённые элементы. При счете пропускайте удалённые элементы.

Системные вызовы

Программа, которая не взаимодействует с внешним миром, вряд ли может сделать что-то полезное. Вывести сообщение на экран, почитать 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:

1. первый — в `%ebx`;
2. второй — в `%ecx`;
3. третий — в `%edx`;
4. четвертый — в `%esi`;
5. пятый — в `%edi`;
6. 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 отличается, 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`.

Я сделал так:

```
[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                   */
```

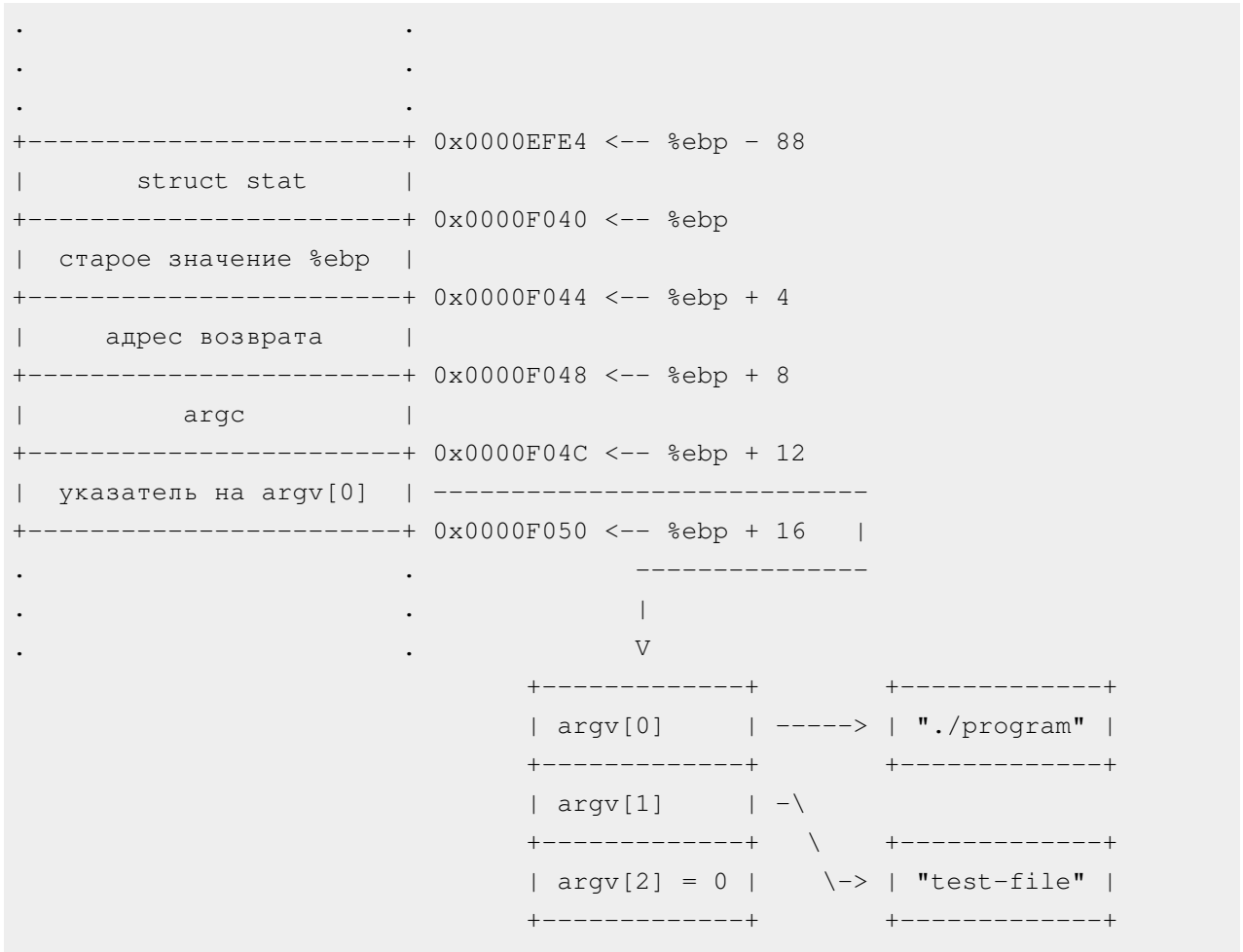
```
return:
        movl  %ebp, %esp
        popl  %ebp
        ret
```

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:



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.

```
.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
```

Для того, чтобы послать с клавиатуры сигнал о конце файла, нажмите Ctrl-D.

```
[user@host:~]$ gcc print.S -o print
[user@host:~]$ ./print
aaa
bbbb
ccccc
^D<>
<ccccc>
<bbbb>
<aaa>
```

Обратите внимание, что мы ввели 4 строки: "aaa", "bbbb", "ccccc", "".

В этой программе был использован некоторый новый синтаксис. Во-первых, вы видите директиву препроцессора `#define`. Препроцессор Си (cpp) может быть использован для обработки исходного кода на ассемблере: нужно всего лишь использовать расширение `.S` для файла с исходным кодом. Файлы с таким расширением gcc предварительно обрабатывает препроцессором cpp, после чего компилирует как обычно.

Во-вторых, были использованы метки-числа, причём некоторые из них повторяются в двух функциях. Почему бы не использовать текстовые метки, как в предыдущих примерах? Можно, но они должны быть уникальными. Например, если бы мы определили метку `read_start` и в функции `read_str`, и в `main`, GCC бы выдал ошибку при компиляции:

```
[user@host:~]$ gcc print.S
print.S: Assembler messages:
print.S:85: Error: symbol `read_start' is already defined
```

Поэтому, используя текстовые метки, приходится каждый раз придумывать уникальное имя. А можно использовать метки-числа, компилятор преобразует их в уникальные имена сам. Чтобы поставить метку, просто используйте любое положительное число в качестве имени. Чтобы сослаться на метку, которая определена ранее, используйте `Nb` (мнемоническое значение — backward), а чтобы сослаться на метку, которая определена дальше в коде, используйте `Nf` (мнемоническое значение — forward).

Операции с цепочками данных

При обработке данных часто приходится иметь дело с цепочками данных. Цепочка, как подсказывает название, представляет собой массив данных — несколько переменных одного размера, расположенных друг за другом в памяти. В Си вы использовали массив и индексную переменную, например, `argv[i]`. Но в ассемблере для последовательной обработки цепочек есть специализированные команды. Синтаксис:

```
lods
stos
```

«Странно», — скажет кто-то, — «откуда эти команды знают, где брать данные и куда их записывать? Ведь у них и аргументов-то нет!» Вспомните про регистры `%esi` и `%edi` и про их немного странные имена: «индекс источника» (англ. source index) и «индекс приёмника» (англ. destination index). Так вот, все цепочечные команды подразумевают, что в регистре `%esi` находится указатель на следующий необработанный элемент цепочки-источника, а в регистре `%edi` — указатель на следующий элемент цепочки-приёмника.

Направление просмотра цепочки задаётся флагом `df`: 0 — просмотр вперед, 1 — просмотр назад.

Итак, команда `lods` загружает элемент из цепочки-источника в регистр `%eax`/`%ax`/`%al` (размер регистра выбирается в зависимости от суффикса команды). После этого значение регистра `%esi` увеличивается или уменьшается (в зависимости от направления просмотра) на значение, равное размеру элемента цепочки.

Команда stos записывает содержимое регистра `%eax`/`%ax`/`%al` в цепочку-приёмник. После этого значение регистра `%edi` увеличивается или уменьшается (в зависимости от направления просмотра) на значение, равное размеру элемента цепочки.

Вот пример программы, которая работает с цепочечными командами. Конечно же, она занимается бестолковым делом, но в противном случае она была бы гораздо сложнее. Она увеличивает каждый байт строки `str_in` на 1, то есть заменяет a на b, b на c, и так далее.

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

str_in:
        .string "abc123()!@!777"
        .set str_in_length, .-str_in

.bss
str_out:
        .space str_in_length

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

        movl  $str_in, %esi     /* цепочка-источник                  */
        movl  $str_out, %edi    /* цепочка-приёмник                  */

        movl  $str_in_length - 1, %ecx  /* длина строки без нулевого
                                   байта (нулевой байт не обрабатываем)
```

```
                                                                    */
1:
        lodsb                   /* загрузить байт из источника в %al */
        incb  %al               /* произвести какую-то операцию с %al
                                                                    */
        stosb                   /* сохранить %al в приёмнике         */
        loop  1b

        movsb                   /* копировать нулевой байт           */

        /* важно: сейчас %edi указывает на конец цепочки-приёмника */

        pushl $str_out
        pushl $printf_format
        call  printf            /* вывести на печать                 */

        movl  $0, %eax

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

```
[user@host:~]$ ./stringop
bcd234)*"A"888
[user@host:~]$
```

Но с цепочками мы часто выполняем довольно стандартные действия. Например, при копировании блоков памяти мы просто пересылаем байты из одной цепочки в другую, без обработки. При сравнении строк мы сравниваем элементы двух цепочек. При вычислении длины строки в Си мы считаем байты до тех пор, пока не встретим нулевой байт. Эти действия очень просты, но, в тоже время, используются очень часто, поэтому были введены следующие команды:

```
 movs
 cmps
 scas
```

Размер элементов цепочки, которые обрабатывают эти команды, зависит от использованного суффикса команды.

Команда `movs` выполняет копирование одного элемента из цепочки-источника в цепочку-приёмник.

Команда `cmps` выполняет сравнение элемента из цепочки-источника и цепочки-приёмника (фактически, как и `cmp`, выполняет вычитание, источник — приёмник, результат никуда не записывается, но флаги устанавливаются).

Команда `scas` предназначена для поиска определённого элемента в цепочке. Она сравнивает содержимое регистра `%eax`/`%ax`/`%al` и содержимое элемента цепочки (выполняется вычитание `%eax`/`%ax`/`%al` — элемент_цепочки, результат не записывается, но флаги устанавливаются). Адрес цепочки должен быть помещён в регистр `%edi`.

После того, как эти команды выполнили своё основное действие, они увеличивают/уменьшают индексные регистры на размер элемента цепочки.

Подчеркну тот факт, что эти команды обрабатывают только один элемент цепочки. Таким образом, нужно организовать что-то вроде цикла для обработки всей цепочки. Для этих целей существуют префиксы команд:

```
rep
repe/repz
repne/repnz
```

Эти префиксы ставятся перед командой, например: `repe scas`. Префикс организовывает как бы цикл из одной команды, при этом с каждым шагом цикла значение регистра `%ecx` автоматически уменьшается на 1.

- `rep` повторяет команду, пока `%ecx` не равен нулю.
- `repe` (или `repz` — то же самое) повторяет команду, пока `%ecx` не равен нулю и установлен флаг `zf`. Анализируя значение регистра `%ecx`, можно установить точную причину выхода из цикла: если `%ecx` равен нулю, значит, `zf` всегда был установлен, и вся цепочка пройдена до конца, если `%ecx` больше нуля — значит, флаг `zf` в какой-то момент был сброшен.
- `repne` (или `repnz` — то же самое) повторяет команду, пока `%ecx` не равен нулю и не установлен флаг `zf`.

Также следует указать команды для управления флагом `df`:

```
cld
std
```

`cld` (CLear Direction flag) сбрасывает флаг `df`.

`std` (SeT Direction flag) устанавливает флаг `df`.

Пример: memcpy

Вооружившись новыми знаниями, попробуем заново изобрести функцию `memcpy(3)`:

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

str_in:
        .string "abc123()!@!777"
        .set str_in_length, .-str_in

.bss
str_out:
        .space str_in_length

.text

/* void *my_memcpy(void *dest, const void *src, size_t n); */

my_memcpy:
        pushl %ebp
        movl  %esp, %ebp

        pushl %esi
        pushl %edi
```

```
        movl  8(%ebp), %edi     /* цепочка-назначение                */
        movl  12(%ebp), %esi    /* цепочка-источник                  */
        movl  16(%ebp), %ecx    /* длина                             */

        rep movsb

        movl  8(%ebp), %eax     /* вернуть dest                      */

        popl  %edi
        popl  %esi

        movl  %ebp, %esp
        popl  %ebp
        ret

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

        pushl $str_in_length
        pushl $str_in
        pushl $str_out
        call  my_memcpy

        pushl $str_out
        pushl $printf_format
        call  printf

        movl  $0, %eax

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

Вы, наверно, будете удивлены, если я вам скажу, что эта реализация `memcpy` всё равно не самая быстрая. «Что ещё можно сделать?» — спросите вы. Ведь мы можем копировать данные не по одному байту, а по целых 4 байта за раз при помощи `movsl`. Тогда у нас получается приблизительно такой алгоритм: копируем как можно больше данных блоками по 4 байта, после этого остаётся хвостик в 0, 1, 2 или 3 байта; этот остаток можно скопировать при помощи `movsb`. Поэтому нашу `memcpy` лучше переписать вот так:

```
/* void *my_memcpy(void *dest, const void *src, size_t n); */

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

```
        pushl %esi
        pushl %edi

        movl  8(%ebp), %edi     /* цепочка-назначение                */
        movl  12(%ebp), %esi    /* цепочка-источник                  */
        movl  16(%ebp), %edx    /* длина                             */

        movl  %edx, %ecx
        shrl  $2, %ecx          /* делить на 2^2 = 4; теперь в
                                   находится %ecx количество 4-байтных
                                   кусочков                          */
        rep movsl

        movl  %edx, %ecx
        andl  $3, %ecx          /* $3 == $0b11, оставить только два
                                   младших бита, то есть остаток от
                                   деления на 4                      */
        jz    1f                /* если результат 0, пропустить
                                   цепочечную команду                */
        rep movsb
1:

        movl  8(%ebp), %eax     /* вернуть dest                      */

        popl  %edi
        popl  %esi

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

Пример: strlen

Теперь `strlen`: нам нужно сравнить каждый байт цепочки с 0, остановиться, когда найдём 0, и вернуть количество ненулевых байт. Как счетчик мы будем использовать регистр `%ecx`, который автоматически изменяют все префиксы. Но префиксы уменьшают счетчик и прекращают выполнение команды, когда `%ecx` равен 0. Поэтому перед цепочечной командой мы поместим в `%ecx` число `0xffffffff`, и этот регистр будет уменьшатся в ходе выполнения цепочечной команды. Результат получится в обратном коде, поэтому мы используем команду `not` для инвертирования всех битов. И после этого ещё уменьшим результат на 1, так как нулевой байт тоже был посчитан.

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

str_in:
        .string "abc123()!@!777"

.text
```

```
/* size_t my_strlen(const char *s); */

my_strlen:
        pushl %ebp
        movl  %esp, %ebp

        pushl %edi

        movl  8(%ebp), %edi             /* цепочка */

        movl  $0xffffffff, %ecx
        xorl  %eax, %eax                /* %eax = 0 */

        repne scasb

        notl  %ecx
        decl  %ecx

        movl  %ecx, %eax

        popl  %edi

        movl  %ebp, %esp
        popl  %ebp
        ret

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

        pushl $str_in
        call  my_strlen

        pushl %eax
        pushl $printf_format
        call  printf

        movl  $0, %eax

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

Как реализованы другие стандартные цепочечные функции, можно посмотреть, например, в исходных кодах ядра Linux в файлах `/usr/src/linux/arch/x86/include/asm/string_*.h`, `/usr/src/linux/arch/x86/lib/{mem*,str*}`. Оттуда взяты все примеры для этого раздела.

В заключение обсуждения цепочечных команд нужно сказать следующее: не следует заново изобретать стандартные функции, как мы это только что сделали. Это всего лишь пример и объяснение принципов их работы. В реальных программах используйте цепочечные команды, только когда они реально смогут помочь при нестандартной обработке цепочек, а для стандартных операций лучше вызывать библиотечные функции.

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.

В этом примере, как и в предыдущих, имеет смысл переставить некоторые части этого кода в начало, если вы заранее знаете, какие значения вам придётся обрабатывать чаще всего.

Пример: интерпретатор языка Brainfuck

Brainfuck — это эзотерический язык программирования, то есть язык, предназначенный не для практического применения, а придуманный как головоломка, как задача, которая заставляет программиста думать нестандартно. Команды Brainfuck управляют массивом целых чисел с неограниченным набором ячеек, есть понятие текущей ячейки.

- Команды `<` и `>` дают возможность перемещаться по массиву на одну ячейку влево и, соответственно, вправо.
- Команды `+` и `-` увеличивают и, соответственно, уменьшают содержимое текущей ячейки на 1.
- Команда `.` выводит содержимое текущей ячейки на экран как один символ; команда `,` читает один символ и помещает его в текущую ячейку.
- Команды циклов `[` и `]` должны всегда находиться в парах и соблюдать правила вложенности (как скобки в математических выражениях). Команда `[` сравнивает значение текущей ячейки с 0: если оно равно 0, то выполняется команда, следующая за соответствующей `]`, если не равно, то просто выполняется следующая команда. Команда `]` передаёт управление на соответствующую `[`.
- Остальные символы в коде программы являются комментариями, и их следует пропускать.

В начальном состоянии все ячейки содержат значение 0, а текущей является крайняя левая ячейка.

Вот несколько программ с объяснениями:

```
+>+>+        Устанавливает первые три ячейки в 1
[-]          Обнуляет текущую ячейку
[>>>+<<<-]   Перемещает значение текущей ячейки в ячейку, расположенную
             "тремя шагами правее"
```

Интерпретация программы состоит из двух шагов: загрузка программы и собственно исполнение. Во время загрузки следует проверить корректность программы (соответствие `[]`) и расположить код программы в памяти в удобном для выполнения виде. Для этого каждой команде присваивается номер операции, начиная с 0, — для того, чтобы можно было выполнять операции при помощи помощи перехода по массиву адресов, как в `switch`.

Большинство программ на Brainfuck содержат последовательности одинаковых команд `< > + -`, которые можно выполнять не по одной, а все сразу. Например, выполняя код `+++++`, можно выполнить пять раз увеличение на 1, или один раз увеличение на 5. Таким образом, довольно простыми средствами можно сильно оптимизировать выполнение программы.

Вот программы, которые вызовут ошибки загрузки:

```
[       No matching ']' found for a '['
]       No matching '[' found for a ']'
```

А эти программы вызовут ошибки выполнения:

```
<       Memory underflow
+[>+]   Memory overflow
```

Исходный код:

```
#define BF_PROGRAM_SIZE 1024
#define BF_MEMORY_CELLS 32768
#define BF_MEMORY_SIZE BF_MEMORY_CELLS*4

#define BF_OP_LOOP_START 0
#define BF_OP_LOOP_END   1
#define BF_OP_MOVE_LEFT  2
#define BF_OP_MOVE_RIGHT 3
#define BF_OP_INC        4
#define BF_OP_DEC        5
#define BF_OP_PUTC       6
#define BF_OP_GETC       7
#define BF_OP_EXIT       8

.section .rodata
str_memory_underflow:
        .string "Memory underflow\n"

str_memory_overflow:
        .string "Memory overflow\n"

str_loop_start_not_found:
        .string "No matching '[' found for a ']'\n"

str_loop_end_not_found:
        .string "No matching ']' found for a '['\n"

.data
bf_program_ptr:
        .long 0

bf_program_size:
        .long 0

/*
 * Программа загружается в память вот так:
 * ============================
 * код_операции, операнд,
 * код_операции, операнд,
 * код_операции, операнд, ...
 * ============================
 * И код_операции, и операнд занимают по 4 байта.
 * Таким образом, одна команда занимает в памяти 8 байт.
```

```
 *
 * Для команды [ (начало цикла) операндом является номер команды,
 * следующий за концом цикла.
 *
 * Для команды ] (конец цикла) операндом является номер команды-начала
 * цикла ].
 *
 * Для остальных команд (< > + - . ,) операнд задаёт количество
 * повторений этой команды.  Например, для кода +++++ должен быть
 * сгенерирован код операции BF_OP_INC с операндом 5, который при
 * выполнении увеличит текущую ячейку на 5.
 */

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

/* ****************************************** */
/* загрузка программы                         */
/* ****************************************** */

        movl  $BF_PROGRAM_SIZE, %ecx
        movl  %ecx, bf_program_size
        pushl %ecx
        call  malloc
        movl  %eax, bf_program_ptr

        movl  %eax, %ebx        /* %ebx — указатель на блок памяти,
                                   содержащий внутреннее представление
                                   программы                          */
        xorl  %ecx, %ecx        /* %ecx — номер текущей команды       */
        xorl  %esi, %esi        /* %esi — предыдущая команда, символ */

bf_read_loop:
        pushl %ecx
        pushl stdin
        call  fgetc
        addl  $4, %esp
        popl  %ecx

        cmpl  $-1, %eax
        je    bf_read_end

        cmpl  $'[, %eax         /* команды, которые всегда
                                   обрабатываются по одной: [ и ]     */
```

```
        je    bf_read_loop_start

        cmpl  $'], %eax
        je    bf_read_loop_end

        cmpl  %esi, %eax        /* текущая команда такая же, как и
                                   предыдущая?                      */
        jne   not_dupe

        incl  -4(%ebx,%ecx,8)   /* такая же. Но %ecx указывает на
                                   следующую команду, поэтому
                                   используем отрицательное смещение -4
                                                                    */
        jmp   bf_read_loop

not_dupe:                       /* другая                           */
        cmpl  $'<, %eax
        je    bf_read_move_left

        cmpl  $'>, %eax
        je    bf_read_move_right

        cmpl  $'+, %eax
        je    bf_read_inc

        cmpl  $'-, %eax
        je    bf_read_dec

        cmpl  $'., %eax
        je    bf_read_putc

        cmpl  $',, %eax
        je    bf_read_getc

        jmp   bf_read_loop

bf_read_loop_start:
        movl  $BF_OP_LOOP_START, (%ebx,%ecx,8)
        movl  $0, 4(%ebx,%ecx,8)
        jmp   bf_read_switch_end

bf_read_loop_end:
        movl  $BF_OP_LOOP_END, (%ebx,%ecx,8)
        movl  %ecx, %edx
bf_read_loop_end_find:
        testl %edx, %edx
        jz    bf_read_loop_end_not_found
```

```
        decl  %edx
        cmpl  $0, 4(%ebx,%edx,8)
        je    bf_read_loop_end_found
        jmp   bf_read_loop_end_find
bf_read_loop_end_not_found:
        jmp   loop_start_not_found
bf_read_loop_end_found:
        leal  1(%ecx), %edi
        movl  %edi, 4(%ebx,%edx,8)
        movl  %edx, 4(%ebx,%ecx,8)
        jmp   bf_read_switch_end

bf_read_move_left:
        movl  $BF_OP_MOVE_LEFT, (%ebx,%ecx,8)
        jmp   bf_read_switch_end_1

bf_read_move_right:
        movl  $BF_OP_MOVE_RIGHT, (%ebx,%ecx,8)
        jmp   bf_read_switch_end_1

bf_read_inc:
        movl  $BF_OP_INC, (%ebx,%ecx,8)
        jmp   bf_read_switch_end_1

bf_read_dec:
        movl  $BF_OP_DEC, (%ebx,%ecx,8)
        jmp   bf_read_switch_end_1

bf_read_putc:
        movl  $BF_OP_PUTC, (%ebx,%ecx,8)
        jmp   bf_read_switch_end_1

bf_read_getc:
        movl  $BF_OP_GETC, (%ebx,%ecx,8)

bf_read_switch_end_1:
        movl  $1, 4(%ebx,%ecx,8)

bf_read_switch_end:

        movl  %eax, %esi        /* сохранить текущую команду для
                                   сравнения                        */

        incl  %ecx

        leal  (,%ecx,8), %edx   /* блок памяти закончился?          */
        cmpl  bf_program_size, %edx
```

```
        jne   bf_read_loop

        addl  $BF_PROGRAM_SIZE, %edx  /* увеличить размер блока памяти
                                                                    */
        movl  %edx, bf_program_size
        pushl %ecx
        pushl %edx
        pushl %ebx
        call  realloc
        addl  $8, %esp
        popl  %ecx
        movl  %eax, bf_program_ptr
        movl  %eax, %ebx

        jmp   bf_read_loop

bf_read_end:

        movl  $BF_OP_EXIT, (%ebx,%ecx,8)  /* последней добавить
                                             команду выхода          */
        movl  $1, 4(%ebx,%ecx,8)

/*
 * Ищем незакрытые '[':
 * Ищем 0 в поле операнда.  Саму команду не проверяем, так как 0 может
 * быть операндом только у '['.
 */

        xorl  %edx, %edx
1:
        cmpl  $0, 4(%ebx,%ecx,8)
        je    loop_end_not_found
        incl  %ecx
        testl %edx, %ecx
        je    2f
        jmp   1b
2:

/* ****************************************** */
/* выполнение программы                       */
/* ****************************************** */


        pushl $BF_MEMORY_SIZE   /* выделить блок памяти для памяти
                                   программы                        */
        call  malloc
        addl  $4, %esp
```

```
        movl  %eax, %esi

        xorl  %ecx, %ecx        /* %ecx — номер текущей команды      */
        xorl  %edi, %edi        /* %edi — номер текущей ячейки памяти
                                                                      */

interpreter_loop:
        movl  (%ebx,%ecx,8), %eax     /* %eax — команда              */
        movl  4(%ebx,%ecx,8), %edx    /* %edx — операнд              */

        jmp   *interpreter_jump_table(,%eax,4)
.section .rodata
interpreter_jump_table:
        .long bf_op_loop_start
        .long bf_op_loop_end
        .long bf_op_move_left
        .long bf_op_move_right
        .long bf_op_inc
        .long bf_op_dec
        .long bf_op_putc
        .long bf_op_getc
        .long bf_op_exit
.text

bf_op_loop_start:
        cmpl  $0, (%esi,%edi,4)
        je    bf_op_loop_start_jump
        incl  %ecx
        jmp   interpreter_loop
bf_op_loop_start_jump:
        movl  %edx, %ecx
        jmp   interpreter_loop

bf_op_loop_end:
        movl  %edx, %ecx
        jmp   interpreter_loop

bf_op_move_left:
        movl  %edi, %eax
        subl  %edx, %eax        /* если номер новой ячейки
                                   памяти < 0 ...                     */
        js    memory_underflow
        movl  %eax, %edi
        incl  %ecx
        jmp   interpreter_loop

bf_op_move_right:
```

```
        movl  %edi, %eax
        addl  %edx, %eax        /* если номер новой ячейки памяти
                                   больше допустимого...            */
        cmpl  $BF_MEMORY_CELLS, %eax
        jae   memory_overflow
        movl  %eax, %edi
        incl  %ecx
        jmp   interpreter_loop

bf_op_inc:
        addl  %edx, (%esi,%edi,4)
        incl  %ecx
        jmp   interpreter_loop

bf_op_dec:
        subl  %edx, (%esi,%edi,4)
        incl  %ecx
        jmp   interpreter_loop

bf_op_putc:
        xorl  %eax, %eax
        movb  (%esi,%edi,4), %al
        pushl %ecx
        pushl %edi
        movl  %edx, %edi
        pushl stdout
        pushl %eax
bf_op_putc_loop:
        call  fputc
        decl  %edi
        testl %edi, %edi
        jne   bf_op_putc_loop
        addl  $4, %esp
        call  fflush
        addl  $4, %esp
        popl  %edi
        popl  %ecx
        incl  %ecx
        jmp   interpreter_loop

bf_op_getc:
        pushl %ecx
        pushl %edi
        movl  %edx, %edi
        pushl stdin
bf_op_getc_loop:
        call  getc
```

```
        decl  %edi
        testl %edi, %edi
        jne   bf_op_getc_loop
        addl  $4, %esp
        movl  %eax, (%esi,%edi,4)
        popl  %edi
        popl  %ecx
        incl  %ecx
        jmp   interpreter_loop

bf_op_exit:
        xorl  %eax, %eax
        jmp   interpreter_exit

/* ************************************** */
/* обработчики ошибок                     */
/* ************************************** */

memory_underflow:
        pushl $str_memory_underflow
        call  printf
        movl  $1, %eax
        jmp   interpreter_exit

memory_overflow:
        pushl $str_memory_overflow
        call  printf
        movl  $1, %eax
        jmp   interpreter_exit

loop_start_not_found:
        pushl $str_loop_start_not_found
        call  printf
        movl  $1, %eax
        jmp   interpreter_exit

loop_end_not_found:
        pushl $str_loop_end_not_found
        call  printf
        movl  $1, %eax

interpreter_exit:
        movl  %ebp, %esp
        popl  %ebp
        .size main, .-main
```

Булевы выражения

Рассмотрим такой код на языке Си:

```
if(((a > 5) && (b < 10)) || (c == 0))
{
  do_something();
}
```

В принципе, булево выражение можно вычислять как обычное арифметическое, то есть в такой последовательности:

- `a > 5`
- `b < 10`
- `(a > 5) && (b < 10)`
- `c == 0`
- `((a > 5) && (b < 10)) || (c == 0)`

Такой способ вычисления называется полным. Можем ли мы вычислить значение этого выражения быстрее? Смотрите, если `c == 0`, то всё выражение будет иметь значение true в любом случае, независимо от `a` и `b`. А вот если `c != 0`, то приходится проверять значения `a` и `b`. Таким образом, наш код (фактически) превращается в такой:

```
if(c == 0)
{
  goto do_it;
}
if((a > 5) && (b < 10))
{
  goto do_it;
}
goto dont_do_it;

do_it:
  do_something();

dont_do_it:
```

В принципе, можно пойти дальше: если `a <= 5`, нас не интересует сравнение `b < 10`: всё равно выражение равно false.

```
if(c == 0)
{
  goto do_it;
}
if(a > 5)
{
  if(b < 10)
  {
    goto do_it;
  }
}
```

```
goto dont_do_it;

do_it:
  do_something();

dont_do_it:
```

Такой способ вычисления выражений называется сокращённым (от англ. short-circuit evaluation), потому что позволяет вычислить выражение, не проверяя всех входящих в него подвыражений. Можно вывести такие формальные правила:

- если у оператора OR хотя бы один операнд имеет значение true, всё выражение имеет значение true;
- если у оператора AND хотя бы один операнд имеет значение false, всё выражение имеет значение false.

В принципе, сокращённое вычисление булевых выражений помогает написать более быстрый (а часто и более простой) код. С другой стороны, возникают проблемы, если одно из подвыражений при вычислении вызывает побочные эффекты (англ. side effects), например вызов функции:

```
if((c == 0) || foo())
{
  do_something();
}
```

Если мы используем сокращённое вычисление и оказывается, что `c == 0`, то функция `foo()` вызвана не будет, потому что от её результата значение выражения уже не зависит. Хорошо это или плохо, зависит от конкретной ситуации, но, без сомнения, способ выполнения такого кода становится не очевидным.

Во многих языках высокого уровня сокращённое вычисление выражений требуется от компилятора стандартом языка (например, в Си). Однако, обычно задаются более строгие правила вычислений. В большинстве стандартов языков требуется, чтобы выражения соединённые оператором OR (или AND) вычислялись строго слева направо, и если очередное значение будет true (соответственно, false для AND), то вычисление данной цепочки OR-ов (AND-ов) прекращается. Но нужно отметить, что первый пример в этой главе всё равно является корректным с точки зрения стандарта Си (хотя `c == 0` стоит в конце выражения, а вычисляется первым), так как сравнение локальных переменных не вызывает побочных эффектов и компилятор вправе реорганизовать код таким образом.

Теперь перейдём к тому, как это реализовывается на ассемблере. Начнём с полного вычисления:

```
cmpl  $5, a
/* так, а что дальше? */
```

Действительно, нам нужно сохранить результат сравнения в переменную. Из команд, анализирующих флаги, мы знаем только семейство `jcc`, но они нам не подходят. Кроме `jcc`, существует семейство `setcc`. Они проверяют состояние флагов точно так же, как и `jcc`. На основе флагов операнд устанавливается в 1, если проверяемое условие `cc` истинно, и в 0, если условие ложно.

```
setcc операнд
```

Требуется заметить, что команды `setcc` работают только с операндами (хранящимися в регистрах и памяти) размером один байт.

Тогда полное вычисление будет выглядеть так:

```
        cmpl  $5, a
        seta  %al
```

```
        cmpl  $10, b
        setb  %bl
        andb  %bl, %al
        cmpl  $0, c
        sete  %bl
        orb   %bl, %al
        jz    is_false
is_true:
        ...
is_false:
        ...
```

Обратите внимание, что команда `or` устанавливает флаги, и нам не нужно отдельно сравнивать `%al` с нулём.

Сокращённое вычисление:

```
        cmpl  $0, c
        je    is_true
        cmpl  $5, a
        jbe   is_false
        cmpl  $10, b
        jae   is_false
is_true:
        ...
is_false:
        ...
```

Как видите, этот код является не только более коротким, но и завершает своё исполнение, как только результат становится известен. Таким образом, сокращённое вычисление намного быстрее полного.

Отладчик GDB

Цель отладки программы — устранение ошибок в её коде. Для этого вам, скорее всего, придётся исследовать состояние переменных во время выполнения, равно как и сам процесс выполнения (например, отслеживать условные переходы). Тут отладчик — наш первый помощник. Конечно же, в Си достаточно много возможностей отладки без непосредственной остановки программы: от простого `printf(3)` до специальных систем ведения логов по сети и `syslog`. В ассемблере такие методы тоже применимы, но вам может понадобиться наблюдение за состоянием регистров, образ (dump) оперативной памяти и другие вещи, которые гораздо удобнее сделать в интерактивном отладчике. В общем, если вы пишете на ассемблере, то без отладчика вы вряд ли обойдётесь.

Начать отладку можно с определения точки останова (breakpoint), если вы уже приблизительно знаете, какой участок кода нужно исследовать. Этот способ используется чаще всего: ставим точку останова, запускам программу и проходим её выполнение по шагам, попутно наблюдая за необходимыми переменными и регистрами. Вы также можете просто запустить программу под отладчиком и поймать момент, когда она аварийно завершается из-за segmentation fault, — так можно узнать, какая инструкция пытается получить доступ к памяти, подробнее рассмотреть приводящую к ошибке переменную и так далее. Теперь можно исследовать этот код ещё раз, пройти его по шагам, поставив точку останова чуть раньше момента сбоя.

Начнём с простого. Возьмём программу Hello world и скомпилируем её с отладочной информацией при помощи ключа компилятора `-g`:

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

Запускаем gdb:

```
[user@host:~]$ gdb ./hello
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

(gdb)
```

GDB запустился, загрузил исследуемую программу, вывел на экран приглашение `(gdb)` и ждёт команд. Мы хотим пройти программу «по шагам» (single-step mode). Для этого нужно указать команду, на которой программа должна остановиться. Можно указать подпрограмму — тогда остановка будет осуществлена перед началом исполнения инструкций этой подпрограммы. Ещё можно указать имя файла и номер строки.

```
(gdb) b main
Breakpoint 1 at 0x8048324: file hello.s, line 17.
(gdb)
```

`b` — сокращение от `break`. Все команды в GDB можно сокращать, если это не создаёт двусмысленных расшифровок. Запускам программу командой `run`. Эта же команда используется для перезапуска ранее запущенной программы.

```
(gdb) r
Starting program: /tmp/hello

Breakpoint 1, main () at hello.s:17
17      movl  $4, %eax   /* поместить номер системного вызова write = 4
Current language:  auto; currently asm
(gdb)
```

GDB остановил программу и ждёт команд. Вы видите команду вашей программы, которая будет выполнена следующей, имя функции, которая сейчас исполняется, имя файла и номер строки. Для пошагового исполнения у нас есть две команды: `step` (сокращённо `s`) и `next` (сокращённо `n`). Команда `step` производит выполнение программы с заходом в тела подпрограмм. Команда `next` выполняет пошагово только инструкции текущей подпрограммы.

```
(gdb) n
20      movl  $1, %ebx          /* первый параметр - в регистр %ebx  */
(gdb)
```

Итак, инструкция на строке 17 выполнена, и мы ожидаем, что в регистре `%eax` находится число 4. Для вывода на экран различных выражений используется команда `print` (сокращённо `p`). В отличие от команд

ассемблера, GDB в записи регистров использует знак `$` вместо `%`. Посмотрим, что в регистре `%eax`:

```
(gdb) p $eax
$1 = 4
(gdb)
```

Действительно 4! GDB нумерует все выведенные выражения. Сейчас мы видим первое выражение (`$1`), которое равно 4. Теперь к этому выражению можно обращаться по имени. Также можно производить простые вычисления:

```
(gdb) p $1
$2 = 4
(gdb) p $1 + 10
$3 = 14
(gdb) p 0x10 + 0x1f
$4 = 47
(gdb)
```

Пока мы играли с командой `print`, мы уже забыли, какая инструкция исполняется следующей. Команда `info line` выводит информацию об указанной строке кода. Без аргументов выводит информацию о текущей строке.

```
(gdb) info line
Line 20 of "hello.s" starts at address 0x8048329 <main+5> and ends at
0x804832e <main+10>.
(gdb)
```

Команда `list` (сокращённо `l`) выводит на экран исходный код вашей программы. В качестве аргументов ей можно передать:

- *номер_строки* — номер строки в текущем файле;
- *файл*:*номер_строки* — номер строки в указанном файле;
- *имя_функции* — имя функции, если нет неоднозначности;
- *файл*:*имя_функции* — имя функции в указанном файле;
- **адрес* — адрес в памяти, по которому расположена необходимая инструкция.

Если передавать один аргумент, команда `list` выведет 10 строк исходного кода вокруг этого места. Передавая два аргумента, вы указываете строку начала и строку конца листинга.

```
(gdb) l main
12                                  за пределами этого файла          */
13      .type  main, @function  /* main — функция (а не данные)      */
14
15
16      main:
17              movl  $4, %eax  /* поместить номер системного вызова
18                                  write = 4 в регистр %eax          */
19
20              movl  $1, %ebx  /* первый параметр поместить в регистр
21                                  %ebx; номер файлового дескриптора
22                                  stdout = 1                        */
(gdb) l *$eip
```

```
0x8048329 is at hello.s:20.
15
16      main:
17              movl  $4, %eax  /* поместить номер системного вызова
18                                 write = 4 в регистр %eax          */
19
20              movl  $1, %ebx  /* первый параметр поместить в регистр
21                                 %ebx; номер файлового дескриптора
22                                 stdout = 1                        */
23              movl  $hello_str, %ecx  /* второй параметр поместить в
24                                 регистр %ecx; указатель на строку */
(gdb) l 20, 25
20              movl  $1, %ebx  /* первый параметр поместить в регистр
21                                 %ebx; номер файлового дескриптора
22                                 stdout = 1                        */
23              movl  $hello_str, %ecx  /* второй параметр поместить в
24                                 регистр %ecx; указатель на строку */
25
(gdb)
```

Запомните эту команду: `list *$eip`. С её помощью вы всегда можете просмотреть исходный код вокруг инструкции, выполняющейся в текущий момент. Выполняем нашу программу дальше:

```
(gdb) n
23              movl  $hello_str, %ecx  /* второй параметр поместить в
                                           регистр %ecx
(gdb) n
26              movl  $hello_str_length, %edx /* третий параметр
                                           поместить в регистр %edx
(gdb)
```

Не правда ли, утомительно каждый раз нажимать `n`? Если просто нажать Enter, GDB повторит последнюю команду:

```
(gdb)
29              int   $0x80     /* вызвать прерывание 0x80           */
(gdb)
Hello, world!
31              movl  $1, %eax  /* номер системного вызова exit = 1  */
(gdb)
```

Ещё одна удобная команда, о которой стоит знать — `info registers`. Конечно же, её можно сократить до `i r`. Ей можно передать параметр — список регистров, которые необходимо напечатать. Например, когда выполнение происходит в защищённом режиме, нам вряд ли будут интересны значения сегментных регистров.

```
(gdb) info registers
eax            0xe      14
ecx            0x804955c        134518108
edx            0xe      14
ebx            0x1      1
```

```
esp            0xbfabb55c       0xbfabb55c
ebp            0xbfabb5a8       0xbfabb5a8
esi            0x0      0
edi            0xb7f6bcc0       -1208566592
eip            0x804833a        0x804833a <main+22>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) info registers eax ecx edx ebx esp ebp esi edi eip eflags
eax            0xe      14
ecx            0x804955c        134518108
edx            0xe      14
ebx            0x1      1
esp            0xbfabb55c       0xbfabb55c
ebp            0xbfabb5a8       0xbfabb5a8
esi            0x0      0
edi            0xb7f6bcc0       -1208566592
eip            0x804833a        0x804833a <main+22>
eflags         0x246    [ PF ZF IF ]
(gdb)
```

Так, а кроме регистров у нас ведь есть ещё и память, и частный случай памяти — стек. Как просмотреть их содержимое? Команда `x/`*`формат адрес`* отображает содержимое памяти, расположенной по адресу в заданном формате. Формат — это (в таком порядке) количество элементов, буква формата и размер элемента. Буквы формата: o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char) и s(string). Размер: b(byte), h(halfword), w(word), g(giant, 8 bytes). Например, напечатаем 14 символов строки `hello_str`:

```
(gdb) x/14c &hello_str
0x804955c <hello_str>: 72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 44 ','
                       32 ' ' 119 'w'
0x8049564 <hello_str+8>: 111 'o' 114 'r' 108 'l' 100 'd' 33 '!'  10 '\n'
(gdb)
```

То же самое, только в шестнадцатеричном виде:

```
(gdb) x/14xb &hello_str
0x804955c <hello_str>:   0x48  0x65  0x6c  0x6c  0x6f  0x2c  0x20  0x77
0x8049564 <hello_str+8>: 0x6f  0x72  0x6c  0x64  0x21  0x0a
(gdb)
```

Напечатаем 8 верхних слов (4 байта) из стека (для «погружения в стек» читаем слева направо и сверху вниз):

```
(gdb) x/8xw $esp
0xbfd8902c:  0xb7e14ea8      0x00000001      0xbfd890a4      0xbfd890ac
0xbfd8903c:  0x00000000      0xb7f2dff4      0x00000000      0xb7f53cc0
```

```
(gdb)
```

Было бы хорошо, если бы GDB отображал значение какого-то выражения автоматически. Это делает команда `display/`*`формат выражение`*. Если в формате будет указан размер, то принцип действия аналогичен `x`. Если размер не указан, команда ведёт себя как `print`.

```
(gdb) display/4xw $esp
1: x/4xw $esp
0xbf8fdb9c:  0xb7e4dea8      0x00000001      0xbf8fdc14      0xbf8fdc1c
(gdb) display/x $eax
2: /x $eax = 0xe
(gdb) n
32              movl  $0, %ebx  /* передать 0 как значение параметра */
2: /x $eax = 0x1
1: x/4xw $esp
0xbf8fdb9c:  0xb7e4dea8      0x00000001      0xbf8fdc14      0xbf8fdc1c
(gdb)
```

Ссылки

Книги и спецификации

- http://www.intel.com/products/processor/manuals/ — Документация от Intel
- http://developer.amd.com/documentation/guides/Pages/default.aspx — Документация от AMD
- http://download.savannah.gnu.org/releases/pgubook/
- http://www.drpaulcarter.com/pcasm/
- http://refspecs.freestandards.org/ — SysV ABI, различные psABI (Processor Suppliment aBI)
- http://www.sco.com/developers/devspecs/ — i386 psABI
- http://www.x86-64.org/documentation.html — x86-64 psABI

Программы

- http://ald.sourceforge.net/
- info gas [6]
- info gdb [7]

Руководства и ответы на часто задаваемые вопросы

- http://gazette.linux.ru.net/lg94/ramankutty.html
- http://lists.canonical.org/pipermail/kragen-fw/2002-April/000226.html
- http://la.kmv.ru/intro/Assembly-Intro.html
- http://web.cecs.pdx.edu/~bjorn/CS200/linux_tutorial/
- http://docs.cs.up.ac.za/programming/asm/derick_tut/
- http://www.unknownroad.com/rtfm/gdbtut/
- http://asm.sourceforge.net/resources.html
- http://urls.net.ru/computer/programming/asm/
- http://en.wikibooks.org/wiki/X86_Assembly
- http://en.wikipedia.org/wiki/X86

Floating-point

- http://www.rsdn.ru/article/alg/fastpow.xml — Возведение числа в действительную степень. Варианты алгоритма возведения в степень: повышение точности и ускорение

Операционные системы и особенности реализации

- http://www.trilithium.com/johan/2005/08/linux-gate/ — Что такое linux-gate.so.1?
- http://hdante.blogspot.com/2007/02/new-style-system-call-in-linux-x86-ref.html
- http://hdante.blogspot.com/2007/02/getting-vsyscall-address-from-elf.html

Inline Assembly

- http://www.ibm.com/developerworks/library/l-ia.html — Inline assembly for x86 in Linux
- http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html — GCC Inline Assembly HOWTO

x86-64 (AMD64 и Intel 64)

- http://en.wikipedia.org/wiki/X86-64 — x86-64: общая информация, терминология, история
- http://www.x86-64.org/documentation/assembly.html

---

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, 3.2 Overview of the basic execution environment

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual, 4.1 Instructions (N-Z), PUSH

[3] Intel® 64 and IA-32 Architectures Optimization Reference Manual, 3.5.1.3 Using LEA

[4] Intel® 64 and IA-32 Architectures Optimization Reference Manual, 3.5.1.7 Compares

[5] Intel® 64 and IA-32 Architectures Optimization Reference Manual, 3.5.1.6 Clearing Registers and Dependency Breaking Idioms

[6] http://www.gnu.org/software/binutils/

[7] http://www.gnu.org/software/gdb/

Источники и основные авторы

Ассемблер в Linux для программистов С *Источник*: http://ru.wikibooks.org/w/index.php?oldid=70087 *Редакторы*: Alex Yu Yug, Fix27, Gribozavr, Igel Sk, Kgvklhjul, SergeyJ, Smails, Trijnstel, Vladimir32, Xaionaro, Yurikoles, Константин Бытенский, 91 анонимных правок

Источники, лицензии и редакторы изображений

Image:Information icon.svg *Источник*: http://ru.wikibooks.org/w/index.php?title=Файл:Information_icon.svg *Лицензия*: Public domain *Редакторы*: El T

Лицензия