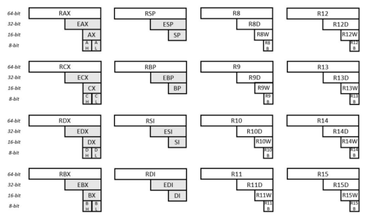
ЧИТАТЬ СВЕРХУ ВНИЗ!!!!!
К ОТВЕТВЛЕНИЯМ ВОЗВРАЩАТЬСЯ КОГДА ВОЗНИКНУТ ВОПРОСЫ ПО КОНКРЕТНЫМ ВЕЩАМ (если не сказано обратное)

Размеры регистров и их вложенное устройство



ВИДЫ РЕГИСТРОВ

Регистр	Обозначение	Назначение
%rax	Accumulator	Регистр аккумулятора. Используется для хранения результатов арифметических и логических операций. Также может хранить возвращаемое значение функции в конвенции вызова.
%rbx	Base	Регистр базы. Обычно используется в качестве базового указателя для доступа к данным. В некоторых ситуациях также может использоваться как общий регистр.
%rcx	Counter	Регистр счетчика. Используется как счетчик цикла в циклах и может также быть использован для операций сдвига и поворота.
%rdx	Data	Регистр данных. Часто используется в качестве вторичного регистра для операций расширения и деления. Также используется для передачи параметров в системные вызовы.
%rsi	Source Index	Регистр индекса источника. Используется для работы с исходными данными в операциях ввода-вывода и операциях обработки строк.
%rdi	Destination Index	Регистр индекса назначения. Используется для работы с целевыми данными в операциях ввода-вывода и операциях обработки строк.
%rsp	Stack Pointer	Регистр указателя стека. Указывает на вершину стека и используется для управления вызовами функций и операциями со стеком.
%rbp	Base Pointer	Регистр указателя базы. Обычно используется для доступа к данным в стеке, в рамках текущей функции.
%rip	Instruction Pointer	Хранит адрес следующей инструкции, используется для управления потоком выполнения.

Регистр	Назначение
%r8	Регистр общего назначения. Используется для хранения данных и участия в арифметических операциях.
%r9	Регистр общего назначения. Аналогично %r8, используется для хранения данных и арифметических операций.
%r10	Регистр общего назначения. Используется для хранения данных и участия в арифметических операциях.
%r11	Регистр общего назначения. Аналогично %r10, используется для хранения данных и арифметических операций.
%r12	Регистр общего назначения. Может использоваться как базовый регистр для доступа к данным.
%r13	Регистр общего назначения. Может использоваться для адресации данных и хранения указателей.
%r14	Регистр общего назначения. Аналогично %r13, используется для адресации данных и хранения указателей.
%r15	Регистр общего назначения. Может использоваться для хранения данных и адресации памяти.

Эти регистры (r8-r15) предназначены для общего использования и могут использоваться для хранения данных, участия в арифметических операциях, адресации памяти и других целей, в зависимости от конкретных потребностей программы. Они предоставляют дополнительные ресурсы для работы с данными в ассемблерных программах.

Регистр	Назначение
CF	Флаг переноса (Carry Flag). Устанавливается, если произошло переполнение при выполнении арифметических операций.
PF	Флаг четности (Parity Flag). Устанавливается, если количество установленных бит в результате операции четное.
AF	Флаг вспомогательного переноса (Auxiliary Carry Flag). Используется в операциях с двоичными числами.
ZF	Флаг нуля (Zero Flag). Устанавливается, если результат операции равен нулю.
SF	Флаг знака (Sign Flag). Устанавливается в соответствии с наивысшим битом результата операции.
OF	Флаг переполнения (Overflow Flag). Устанавливается, если произошло знаковое переполнение при выполнении арифметических операций.
IF	Флаг разрешения прерываний (Interrupt Enable Flag). Управляет разрешением и запретом прерываний.
TF	Флаг трассировки (Trap Flag). Используется в режиме отладки для пошагового выполнения инструкций.
DF	Флаг направления (Direction Flag). Используется в строковых операциях для определения направления обработки данных.
OF	Флаг переполнения (Overflow Flag). Устанавливается, если произошло знаковое переполнение при выполнении арифметических операций.

Эти регистры флагов предоставляют информацию о результате операций и состоянии процессора. Они используются для принятия решений и управления выполнением программы, особенно при выполнении условных операторов и циклов.

Указатель исполняемой инструкции (регистр RIP)

Регистр указателя инструкции (RIP) содержит смещение в текущем сегменте кода для следующей инструкции, которая будет выполняться. Его назначение в перемещении от одной инструкции к другой в прямолинейном коде или в перемещении вперёд или назад на несколько инструкций при выполнении инструкций JMP, Jcc, CALL, RET и IRET.

Регистр RIP не может быть изменён программой. Он управляется исключительно инструкциями перемещения (такими как JMP, Jcc, CALL и RET), прерываниями и исключениями. Единственным способом прочитать значение регистра является выполнение инструкции CALL с последующим чтением значения указателя возврата из стека процедуры. Регистр RIP может быть изменён косвенно, путём изменения значения указателя возврата в стеке процедуры и выполнения инструкции возврата (RET или IRET).

ТИПЫ ОПЕРАНДОВ

- `Imm` означает константное значение, например, 0x0048d8e или 48.
- `Ex` относится к регистру, например, %rax.
- `M[x]` относится к значению, хранящемуся по адресу памяти x.

Эти типы операндов используются в ассемблерных инструкциях для определения источников данных и мест назначения в инструкциях перемещения данных и арифметических операциях.

Тип	Что представляет	Значение	Название
Немедленное (Immediate)	$Imm	Imm	Немедленное (Immediate)
Регистр (Register)	Ea	Значение, хранящееся в регистре Ea	Регистр (Register)
Абсолютная память (Absolute Memory)	Imm	Значение, хранящееся по адресу Imm	Абсолютная память (Absolute Memory)
Абсолютная память (Absolute Memory)	(Ea)	Значение, хранящееся по адресу, хранящееся в регистре Eb	Абсолютная память (Absolute Memory)
Индексированная память (Scaled Indexed)	Imm(Eb, Ei, s)	Значение, хранящееся по адресу, вычисленному как Imm + Значение, хранящееся в регистре Eb + (значение, хранящееся в регистре Ei) умноженное на s)	Индексированная память (Scaled Indexed)

Примеры операндов

Запись операнда в GAS
$0x0048d8e, $48
%rax
0x0048d8e
(%rbx)
0x0048d8e(%rbx, %rcx, 4)

СУФФИКСЫ ИНСТРУКЦИЙ

- "Байт" (byte) означает однобайтовое целое число (суффикс "b"),
- "Слово" (word) означает двубайтовое целое число (суффикс "w"),
- "Двойное слово" (doubleword) означает четырехбайтовое целое число (суффикс "l"),
- "Квадрословд" (quadword) означает восьмибайтовое значение (суффикс "q").

Большинство инструкций, таких как "mov" (перемещение), используют суффикс для указания размера операндов. Например, перемещение квадрослова из %rax в %rbx приводит к инструкции "movq %rax, %rbx".

Некоторые инструкции, такие как "ret" (возврат), не используют суффиксы, потому что это не требуется. Другие, такие как "movs" и "movz", используют два суффикса, так как они преобразуют операнды из типа первого суффикса в тип второго. Таким образом, ассемблерная инструкция для преобразования байта в %al в двойное слово в %ebx с расширением нулей будет "movzbl %al, %ebx".

ОБЯЗАТЕЛЬНО ЧИТАТЬ

Инструкция mov

Не особо важно, чисто для ознакомления и лучшего понимания

```
mov [размер операнда] операнд_назначения, операнд_источника
```

Где:

- `[размер операнда]` может быть `b` (байт), `w` (слово), `l` (двойное слово), или `q` (квадра-слово), и определяет размер операнда.
- `операнд_назначения` указывает, куда будет перемещено значение.
- `операнд_источника` указывает, откуда будет взято значение для перемещения.

Инструкция	Описание
mov %ebx, %eax	Переместить значение из регистра %ebx в %eax
mov (%rdi), %rax	Загрузить значение из памяти по адресу, хранящемуся в регистре %rdi, в регистр %rax
mov $42, %eax	Загрузить константу 42 в регистр %eax
mov %eax, my_variable	Сохранить значение из регистра %eax в памяти по адресу my_variable
mov $0x1234, %eax	Загрузить константу 0x1234 в регистр %eax
mov $100, my_variable	Сохранить константу 100 в памяти по адресу my_variable
movsbw %ax, %ebx	Расширить значение регистра %ax (16 бит) в регистр %ebx (32 бита) с расширением знака
movzwl %ai, %edx	Обрезать значение регистра %al (8 бит) в регистр %edx (32 бита) с обрезкой нулей

Для определения размера операнда и его типа в инструкции `mov`, обычно используется следующий синтаксис:

- `mov[b/w/l/q] операнд1, операнд2`

Где:

- `b` - байт (8 бит),
- `w` - слово (16 бит),
- `l` - двойное слово (32 бита),
- `q` - квадра-слово (64 бита).

Примеры:

- `movb` - перемещение байта (8 бит).
- `movw` - перемещение слова (16 бит).
- `movl` - перемещение двойного слова (32 бита).
- `movq` - перемещение квадра-слова (64 бита).

- `movl src, dst` - это формат инструкции `mov`, где `src` представляет источник данных, а `dst` - место, куда данные будут перемещены.

Источник (`src`) или место назначения (`dst`) могут быть одним из следующих типов:

- **Немедленное значение (immediate)**, например, `$0x10` или `$4`. Это фиксированная константа.
- **Регистр (register)**, например, `%eax`. Это регистр процессора, который содержит данные.
- **Память (memory)**, например, адрес в памяти. Это место, где хранятся данные в памяти.

Существуют ограничения:

- Место назначения (`dst`) не может быть немедленным значением (immediate).
- Источник (`src`) или место назначения (`dst`) может быть памятью, но не одновременно.

Инструкция `mov` также имеет общую форму адресации памяти:

```
%[register], [register2], C]
```

Эта форма позволяет вычислить адрес памяти, который будет использоваться для чтения или записи данных. Где:

- `N` может быть большим числом и представлять смещение (offset) относительно базового адреса.
- `[register1]` - это регистр, добавляемый к итоговому адресу.
- `[register2]` - еще один регистр, который участвует в вычислении адреса.
- `C` может быть равно 1,2, 4 или 8 и является множителем для `[register2]`.

Итоговый адрес памяти вычисляется следующим образом:

```
% + ([содержимое register1] + (C * содержимое [register2]))
```

Для более краткого форм адресации памяти существуют следующие варианты:

- `N absolute (reg1=0, reg2=0)`: Абсолютный адрес, где `N` - это адрес.
- `(%eax) register indirect (N=0, reg2=0)`: Чтение значения из адреса, содержащегося в регистре `%eax`.
- `N(%eax) base + displacement (reg2=0)`: Адрес с базовым смещением. `N(%eax, %ebx) indexed (C=1)`: Индексированный адрес с учетом обоих регистров `%eax` и `%ebx`.

Пример:

```
movl C(%eax), %ebx
```

Это инструкция берет значение из регистра `%eax`, добавляет C к нему, а затем извлекает содержимое памяти по этому адресу и помещает результат в регистр `%ebx`. Это часто называется "загрузкой" (load) данных, так как она перемещает данные из памяти в регистр.

Инструкция lea

Инструкция `lea` (Load Effective Address) в языке ассемблера x86/x86-64 не загружает фактическое значение в регистр, как это делает, например, инструкция `mov`. Вместо этого она загружает в регистр адрес вычисляемого операнда.

Главное назначение инструкции `lea` - вычисление адреса операнда и загрузка этого адреса в регистр. Она полезна, например, для выполнения арифметических операций с адресами и вычисления адресов в таблицах массивов и структур данных, а также для создания сложных адресных режимов.

Преимущества использования `lea` включают:

1. Эффективное использование инструкции `lea` может быть использована для выполнения сложных вычислений адресов без необходимости использования временных регистров.
2. Гибкость: `lea` позволяет выполнять арифметические операции над адресами и регистрами, что может быть полезно для реализации сложных алгоритмов.
3. Отсутствие побочных эффектов поскольку `lea` не изменяет фактические данные в памяти, она не влияет на содержимое памяти и флаги процессора.

Пример использования `lea`:

```
lea (%eax,%ebx,4), %edx  # Загрузить адрес (%eax + 4*%ebx) в %edx
```

Здесь инструкция `lea` вычисляет адрес, равный сумме регистров `%eax` и `%ebx`, умноженной на 4, и загружает этот адрес в регистр `%edx`, не изменяя фактические данные в памяти.

Конечно, вот несколько примеров использования инструкции `lea`:

1. Вычисление адреса элемента массива:

```
data_array:
    .long 10, 20, 30, 40, 50
    ...
lea (data_array,%ebx,4), %eax  # Загрузить адрес data_array[%ebx*4] в %eax
```

Здесь `lea` используется для вычисления адреса элемента массива `data_array` с индексом, хранящимся в регистре `%ebx`. Полученный адрес загружается в регистр `%eax`.

2. Вычисление адреса структуры:

```
struct_size = 12
...
lea (struct_size,%edx,2), %ebx  # Загрузить адрес структуры в %ebx
```

В этом примере `lea` вычисляет адрес структуры с размером `struct_size` и умноженным на 2 в регистре `%edx` и загружает этот адрес в регистр `%ebx`.

3. Сложение адресов:

```
lea (data_ptr1,%eax,1), %edx  # Загрузить адрес data_ptr1 + %eax в %edx
```

Здесь `lea` используется для сложения адреса, хранящегося в `data_ptr1`, с содержимым регистра `%eax`, и результат загружается в регистр `%edx`.

4. Вычисление адреса вложенных структур:

```
lea (struct_ptr,%ecx,8), %edx  # Загрузить адрес struct_ptr.field1.field2 в %edx
```

В этом примере `lea` вычисляет адрес вложенного поля в структуре, на которую указывает `struct_ptr`, и результат загружается в регистр `%edx`.

Инструкция `lea` предоставляет большую гибкость при работе с адресами и позволяет выполнять сложные вычисления без необходимости влиять на фактические данные в памяти.

Строка хранится так:
в переменной хранится указатель на первый символ в памяти
строка обязательно должна заканчиваться терминальным нулем - '\0'

```
mov ... $message, %rsi    # address of string to output
rsi     0x402000     4202496     source index (source for data copies) (64-bit)
0x402000  48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 0a 00 00 00   Hello, world....
movb (message), %al       # get first symbol of string and it
rax     0x48         72          register a extended (64-bit)
```

Тут мы сохраняем указатель на строку

Тут мы сохраняем первый символ
используем movb чтобы сохранить один символ, так как один символ в ascii - 1 байт

С обычными числами работает не так

```
number:
    .int  10
...
lea number, %rax
rax     0x402000     4202209
0x402000:   0a 00 00 00 00 00 00 00
```

Тут мы положили адрес на переменную с помощью инструкции lea в регистр, в котором хранится значение 10

```
mov (number), %rax        # move value to rax register
rax     0xa          10          r
```

Тут мы просто положили значение 10 в регистр

Пример Программы

```
    .section .text # code section
    ...
    .globl _start
_start:
    # write(1, message, 13)
    mov     $1, %rax            # system call 1 is write
    mov     $1, %rdi            # file handle 1 is stdout
    mov     $message, %rsi      # address of string to output
    mov     $13, %rdx           # number of bytes
    syscall                     # invoke operating system to do the write

    # exit(0)
    mov     $60, %rax           # system call 60 is exit
    xor     %rdi, %rdi          # we want return code 0
    syscall                     # invoke operating system to exit

    .section .data  # data section
message:
    .ascii  "Hello, world\n"
```

инструкция syscall

Системные вызовы

Системные вызовы (Syscall) представляют собой механизм, который позволяет пользовательским программам взаимодействовать с операционной системой для выполнения различных задач. Эти вызовы обеспечивают интерфейс между прикладным программным обеспечением и ядром операционной системы, позволяя программам выполнять операции, которые требуют привилегий ядра.

Вот некоторые общие аспекты системных вызовов:

1. **Примеры операций:** Системные вызовы могут быть использованы для разнообразных задач, таких как чтение и запись файлов, управление процессами, создание сетевых сокетов, выделение памяти, управление устройствами и другие операции, которые требуют доступа к ядру операционной системы.
2. **Интерфейс ядра:** Для выполнения системного вызова, пользовательская программа обращается к определенной функции, предоставляемой библиотекой Cи (libc) или другими средствами, которая, в свою очередь, вызывает соответствующую системную функцию ядра. Системные вызовы имеют числовой идентификатор (называемый номером вызова), который уникален для каждой операции.
3. **Привилегии и безопасность:** Системные вызовы позволяют управлять доступом к ресурсам и операциям, требующим привилегий. Ядро операционной системы контролирует и проверяет запросы, обеспечивая безопасность выполнения системных вызовов.
4. **Кросс-платформенность:** Системные вызовы могут варьироваться в зависимости от операционной системы. Например, системные вызовы для Linux отличаются от системных вызовов для Windows. Поэтому код, использующий системные вызовы, может потребовать адаптации для работы на разных платформах.
5. **Примеры системных вызовов:** Ниже приведены некоторые общие примеры системных вызовов, которые могут выполняться в программе:
   - `open`: открытие файла или устройства для чтения/записи.
   - `read` и `write`: чтение и запись данных в файл или устройство.
   - `fork` и `exec`: создание новых процессов и выполнение других программ.
   - `socket`: создание сетевого сокета для сетевого взаимодействия.
   - `malloc` и `free`: выделение и освобождение памяти.

Системные вызовы - это мощный механизм, который позволяет программам взаимодействовать с операционной системой и выполнять различные задачи. Их использование обычно требует знания API операционной системы и является важным аспектом разработки программного обеспечения.

РАБОТА С ПАМЯТЬЮ

Инструкции с одним суффиксом

Инструкция	Описание	Пример
mov S, D	Переместить источник в назначение	movl %eax, %ebx
push S	Поместить источник в стек	push %rax
pop D	Извлечь вершину стека в назначение	pop %rbx

Инструкции с двумя суффиксами

Инструкция	Описание	Пример
mov S, D	Переместить байт в слово (знаковое расширение)	movsbw %ax, %bx
push S	Переместить байт в слово (расширение нулями)	pushw %al

Инструкции без суффиксов

Инструкция	Описание	Пример
cwtl	Преобразовать слово в %ax в двойное слово в %eax (с расширением знака)	cwtl
cltq	Преобразовать двойное слово в %eax в квадрослов в %rax (с расширением знака)	cltq
cqto	Преобразовать квадрослов в %rax в восьмиворд в %rdx:%rax	cqto

Адрес	Значение	Комментарий
0x7fff0000	42	Верхний элемент стека
0x7fff0008	28	
0x7fff0010	15	^ push кладет элементы вот туда,
0x7fff0018	8	| в обратную сторону
0x7fff0020	4	
0x7fff0028	2	
0x7fff0030	1	Нижний элемент стека
0x7fff0038	2	Количество аргументов <- изначально rsp указывает сюда
0x7fff0040	arg1	тут лежат уже количество аргументов, например 2
0x7fff0048	arg2	сами аргументы
0x7fff0050	/0	<- конец аргументов
0x7fff0058	...	
0x7fff0060	...	
0x7fff0068	...	
0x7fff0070	...	
0x7fff0078	...	
0x7fff0080	...	

```
0x7fffffffd0f8    00 00 00 00 00 00 00 00
0x7fffffffd100    00 00 00 00 00 00 00 00
0x7fffffffd108    00 00 00 00 00 00 00 00
0x7fffffffd110    01 00 00 00 00 00 00 00
0x7fffffffd118    2e 00 ff ff ff ff 7f 00
0x7fffffffd120    01 00 ff ff ff ff 7f 00
0x7fffffffd128    86 00 ff ff ff ff 7f 00
0x7fffffffd130    8b 00 ff ff ff ff 7f 00
0x7fffffffd138    50 00 ff ff ff ff 7f 00
0x7fffffffd140    00 00 00 00 00 00 00 00
....
```

ПРИМЕР

Тут %rsp указывает на 0x7ff...fd110
где и лежит количество аргументов

как устроен стек инструкции pop и push

Изначально при запуске программы указатель на вершину стека хранится в регистре %rsp, поэтому его нужно сначала сохранить в %rbp

После mov сюда

Инструкция push

Инструкция `push` в языке ассемблера x86/x86-64 используется для помещения (размещения) значения на вершину стека. Она часто используется для сохранения значений регистров на стеке перед вызовом функций, передачи параметров функциям и временного хранения данных в стеке. Вот описание инструкции `push` и примеры её использования:

```
# Инструкция push вставляет значение на вершину стека.
# Операнд может быть регистром или адресом в памяти.

# Пример 1: Сохранение значения регистра на стеке
push %eax

# Пример 2: Сохранение значения по адресу в памяти на стеке
push 0x0040000

# Пример 3: Передача параметров функции через стек
push $42
```

Инструкция `push` увеличивает указатель стека (обычно регистр `%rsp` или `%rsp`) на размер данных, помещаемых на стек (обычно 4 байта для 32-битной архитектуры и 8 байт для 64-битной архитектуры). Это обеспечивает помещение данных на вершину стека и перемещение указателя стека вниз.

Вот эквивалентная реализация инструкции `push` с использованием инструкций `mov` и `sub` для 64-битной архитектуры:

```
# Сохранение значения в регистре на стеке (эквивалент инструкции push)
movq %rax, -8(%rsp)   # Сначала копируем значение в стек
subq $8, %rsp         # Затем уменьшаем указатель стека
```

В этом примере мы сначала используем инструкцию `mov` для копирования значения из регистра `%rax` в ячейку памяти на стеке, находящуюся в `-8(%rsp)` (это эквивалент вершины стека). Затем мы используем `sub` для уменьшения указателя стека `%rsp` на 8 байт, чтобы сделать место для нового значения на вершине стека.

Этот код выполняет те же действия, что и инструкция `push`, сохраняя значение на вершине стека.

ДЛЯ 32 -БИТНОЙ:
адреса в 64 битной архитектуре занимают 8 байт, а в 32 битной 4,
поэтому если мы хотим двигаться по стеку то мы должны прибавлять или убавлять по 4 байта соответственно для 32 битной архитектуры

Инструкция push

Инструкция `pop` в языке ассемблера x86/x86-64 используется для извлечения (удаления) значения из вершины стека и загрузки его в регистр или в память. Она часто используется для доступа к данным, хранящиеся в стеке, и для восстановления значений регистров, сохраненных на стеке в процессе вызова функции. Вот описание инструкции `pop` и примеры её использования:

```
# Инструкция pop извлекает значение из вершины стека и загружает его в регистр
# Операнд может быть регистром или адресом в памяти.

# Пример 1: Загрузка значения из вершины стека в регистр
pop %eax

# Пример 2: Загрузка значения из вершины стека в ячейку памяти
pop 0x0040000

# Пример 3: Восстановление значения регистра после вызова функции
pop %ebx
```

Инструкция `pop` уменьшает указатель стека (обычно регистр `%rsp` или `%rsp`) на размер данных, извлекаемых из стека (обычно 4 байта для 32-битной архитектуры и 8 байт для 64-битной архитектуры). Это обеспечивает извлечение данных из вершины стека и перемещение указателя стека вниз.

Вот эквивалентная реализация инструкции `pop` с использованием инструкций `mov` и `add` для 64-битной архитектуры:

```
# Извлечение значения из вершины стека (эквивалент инструкции pop)
addq $8, %rsp         # Увеличение указателя стека
movq (%rsp), %rax     # Копирование значения из вершины стека в регистр %rax
```

В этом коде мы сначала используем инструкцию `add` чтобы увеличить указатель стека (`%rsp` на 8 байт (размер элемента, извлекаемого из стека). Затем мы используем `movq`, чтобы скопировать значение из вершины стека (которое находится в адресе, указываемом указателем стека `%rsp`) в регистр `%rax`.

Этот код выполняет те же действия, что и инструкция `pop`, извлекая значение из вершины стека и загружая его в регистр.

Структура программы

В GAS (GNU Assembler) секции - это части программы, которые определяют, как размещаются и как обрабатываются различные типы данных и кода в исполняемом файле. В ассемблере x86/x86-64, общие секции, которые используются, включают:

1. **.text:** Эта секция содержит исполняемый код программы. Здесь размещаются инструкции процессора, функции, и другие исполняемые части программы. Эта секция является обязательной и содержит точку входа в программу (обычно `main`).
   Пример объявления секции `.text`:
   ```
   .text
   ```
2. **.data:** В этой секции хранятся и инициализируются глобальные переменные и данные, которые могут быть изменены в процессе выполнения программы. Эта секция также может содержать строки и другие инициализированные данные.
   Пример объявления секции `.data`:
   ```
   .data
   ```
3. **.bss:** Секция `.bss` используется для объявления неинициализированных глобальных переменных и данных, которые будут инициализированы во время выполнения программы. Это место для резервирования памяти.
   Пример объявления секции `.bss`:
   ```
   .bss
   ```
4. **.rodata:** Эта секция содержит только для чтения данные, такие как константные строки и другие данные, которые не могут быть изменены во время выполнения программы.
   Пример объявления секции `.rodata`:
   ```
   .section .rodata
   ```

Как лежит программа в оперативной памяти

Инструкция mov
и инструкция lea

```
| Адрес        | Секция   | Содержание                  |
|--------------|----------|-----------------------------|
| 0x00040000   | .text    | Инструкции программы        |
| 0x00040100   | .text    |                             |
| 0x00040200   | .text    |                             |
| 0x00040300   | .text    |                             |
                                   ...
| 0x00040A00   | .text    |                             |
| 0x00040B00   | .text    |                             |
| 0x00040C00   | .data    | Инициализированные данные   |
| 0x00040C10   | .data    |                             |
| 0x00040C20   | .data    |                             |
| 0x00040C30   | .data    |                             |
                                   ...
| 0x00040CB0   | .data    |                             |
| 0x00040CC0   | .bss     | Неинициализированные данные |
| 0x00040CD0   | .bss     |                             |
| 0x00040CE0   | .bss     |                             |
                                   ...
| 0x00040D40   | .bss     |                             |
                                 КОНЕЦ
```

объявление
переменных

Объявление переменных

В GAS (GNU Assembler), объявление переменных и выделение памяти для них может выполняться в различных секциях, таких как `.data`, `.bss` и `.rodata`. Вот как вы можете объявить переменные в разных секциях:

1. **Переменные в секции `.data`:**
   - Секция `.data` предназначена для хранения и инициализации глобальных переменных и данных, которые имеют значение на момент компиляции.
   - Переменные в секции `.data` могут быть инициализированы определенными значениями.
   - Пример объявления переменной в секции `.data`:
   ```
   .section .data
   my_variable:    # Имя переменной
     .int 42       # Инициализация переменной значением 42 (4 байта)
   ```
2. **Переменные в секции `.bss`:**
   - Секция `.bss` используется для объявления неинициализированных глобальных переменных, которые будут инициализированы во время выполнения программы.
   - Переменные в секции `.bss` не имеют начальных значений и занимают память, но инициализируются нулями по умолчанию.
   - Пример объявления переменной в секции `.bss`:
   ```
   .section .bss
   my_buffer:    # Имя переменной
     .skip 1024  # Резервирование 1024 байт памяти
   ```
3. **Константы в секции `.rodata`:**
   - Секция `.rodata` предназначена для хранения данных, которые могут только читаться, но не изменяться.
   - Обычно в этой секции размещаются константы и строки.
   - Пример объявления константы в секции `.rodata`:
   ```
   .section .rodata
   my_message:   # Имя константы
     .string "Hello, World!\n"  # Инициализация строки
   ```

Объявление переменных и констант в GAS выполняется с использованием специфичных директив, таких как `.int` для целых чисел, `.skip` для резервирования памяти и `.string` для строк. После объявления переменных, вы можете использовать их в коде программы, обращаясь к ним по их именам.

Типы данных

Тип данных	Размер (в byte-ах)	Директива в GAS
Целое число (8-бит)	1	`.byte`
Целое число (16-бит)	2	`.short`
Целое число (32-бита)	4	`.int`
Целое число (64-бита)	8	`.quad`

Тип данных	Директива в GAS	Пример
ASCII-строка	`.ascii`	`.ascii "Hello, World!"`
UTF-8 строка	`.utf8string`	`.utf8string "привет, мир!"`
Null-терминированная строка	`.asciz`	`.asciz "Hello"`
Вещественное (float-ы)	`.byte`	`.byte 75, 101, 108, 108, 111`

Вместо `.ascii` можно просто использовать `.string` и не обьять себе мозг постоянно держа в голове, что нужно писать '\0' в конце строки
НО
utf-8 строка состоит из символов, которые занимают НЕ ВСЕГДА 1 БАЙТ

Вкратце алгоритм кодирования символов в UTF8 выглядит так:

1. Если код символа меньше 128, то кодировка так есть.
2. Если больше или равно, то кодировка в следующем формате: в первом байте в ограничение кода зависимости количество используемых байт включая первый (то есть записывается такое число единиц, сколько байт в числе; теоретически формат UTF8 не накладывает ограничений на количество байт используемых для записи чисел, но обычно используется не более 6). Затем записывается 0 (чтобы отделить количество байт числа от значащих разрядов числа), после него идут биты, представляющие число. Каждый следующий байт начинается с последовательности 10 (чтобы не путать однобайтовые и многобайтные символы - все однобайтовые начинаются с нуля).

Например,

111 0 0011 10 010101 10 000100

1. Сначала **3 единицы**: значит, для записи используется 3 байта.
2. Затем 0, отделяющий запись числа байт от записи числа.
3. Дальше значащие разряды числа. Первые 2 бита каждого байта (10) у них не относятся это метка, сообщающая, что это второй (третий, и т.д.) байт записи.

Таким образом, закодированный код - 00110101000100100. Это и вопросу о том, откуда взялись hexdump-таблицы. А вопросу какую функцию использовать - исходите из потребностей. Если вы всегда будете работать с символами, код которых не превышает 255, то используйте hexdump-таблицу, она быстрее. Если же планируете полная поддержка Юникода, то используйте функцию. Только учтите индексирующего, так как она не рассчитана 9- и 8-битные символы.

UPD

Перед прощанием, выражение hexdump таблицы сбило меня с толку и навеяло неверные ассоциации. Да, действительно, эта таблица предназначена для проверки первого байта последовательности и решения о длине кода. Лично я все равно остановлю в функции - она выглядит гораздо лаконичнее. Но lookup-таблица работает быстрее, если это приоритетность, лучше использовать ее.

инструкция cmp

Инструкция `CMP` в x86 устанавливает флаги в регистре флагов в зависимости от результата сравнения двух операндов. Вот какие флаги устанавливаются и при каких условиях:

1. **ZF (Zero Flag):** Устанавливается в 1, если операнды равны, и в 0, если операнды не равны.
2. **SF (Sign Flag):** Устанавливается в значении старшего бита результата (знаковый бит).
3. **PF (Parity Flag):** Устанавливается в 1, если количество установленных бит в младших 8 битах результата четно.
4. **CF (Carry Flag):** Устанавливается в 1, если при сложении операндов произошел перенос (операнды беззнаковые).
5. **OF (Overflow Flag):** Устанавливается в 1, если при сложении операндов произошло переполнение.
6. **AF (Auxiliary Flag):** Используется в операциях BCD (Binary Coded Decimal), для отслеживания переноса из младшей тетрады в старшую.

Summary

Overflow Flag	Signed Addition	Signed Subtraction
0	Result is correct.	Result is correct.
1	Result is incorrect.	Result is incorrect.

Carry Flag	Unsigned Addition	Unsigned Subtraction
0	Result is correct.	Result is incorrect.
1	Result is incorrect.	Result is correct.

Примеры сравнений с разными числами:

```
; Пример 1: Сравнение беззнаковых чисел
mov eax, 5        ; Помещаем значение 5 в регистр eax
mov ebx, 7        ; Помещаем значение 7 в регистр ebx
cmp eax, ebx      ; Сравниваем значения в регистрах eax и ebx

; Флаги после сравнения:
; ZF = 0, SF = 1 (поскольку результат отрицателен), PF = 1 (четное количество установленных бит),
; CF = 1 (поскольку 5 < 7, устанавливается флаг переноса), OF = 0, AF = не определен
------------------------------------------------------------------------------------------

; Пример 2: Сравнение знаковых чисел
mov eax, -10      ; Помещаем значение -10 в регистр eax
mov ebx, 5        ; Помещаем значение 5 в регистр ebx
cmp eax, ebx      ; Сравниваем значения в регистрах eax и ebx

; Флаги после сравнения:
; ZF = 0, SF = 1 (поскольку результат отрицателен), PF = 1 (четное количество установленных бит),
; CF = 1 (поскольку -10 < 5, устанавливается флаг переноса), OF = 0, AF = не определен
------------------------------------------------------------------------------------------

; Пример: Сравнение, когда число слева больше, чем число справа
mov eax, 15       ; Помещаем значение 15 в регистр eax
mov ebx, 7        ; Помещаем значение 7 в регистр ebx
cmp eax, ebx      ; Сравниваем значения в регистрах eax и ebx

; Флаги после сравнения:
; ZF = 0 (поскольку 15 не равно 7), SF = 0 (поскольку результат положителен),
; PF = 0 (нечетное количество установленных бит), CF = 0 (поскольку 15 > 7),
; OF = 0 (поскольку нет переполнения), AF = не определен
------------------------------------------------------------------------------------------

; Пример: Сравнение, когда число слева больше, чем отрицательное число справа
mov eax, 15       ; Помещаем значение 15 в регистр eax
mov ebx, -7       ; Помещаем значение -7 в регистр ebx
cmp eax, ebx      ; Сравниваем значения в регистрах eax и ebx

; Флаги после сравнения:
; ZF = 0 (поскольку 15 не равно -7), SF = 0 (поскольку результат положителен),
; PF = 0 (нечетное количество установленных бит), CF = 0 (поскольку 15 > -7),
; OF = 0 (поскольку нет переполнения), AF = не определен
------------------------------------------------------------------------------------------

; Пример: Сравнение, когда оба числа отрицательные, и одно по абсолютному значению больше другого
mov eax, -15      ; Помещаем значение -15 в регистр eax
mov ebx, -7       ; Помещаем значение -7 в регистр ebx
cmp eax, ebx      ; Сравниваем значения в регистрах eax и ebx

; Флаги после сравнения:
```