??????? ? ???????????

Функция - набор инструкций под некоторой меткой (имя функции). Функции завершается ret. Вызываемые функции могут вызывать другие функции.

```
sum:
    mov rdi, 7
    mov rsi, 5
    add rdi, rsi
    ret
```

Вызов функции:

```
call название_функции
```

Call помещает в стек 64-битный адрес инструкции, которая идет сразу после вызова. Это называется адресом возврата. Когда процедура завершает выполнение, для возвращения к вызывающему коду она выполняет инструкцию ret. Команда ret извлекает 64-битный адрес возврата из стека и косвенно передает управление на этот адрес.

```
global _start

section .text
_start:
    call sum
    mov rax, 60
    syscall

sum:
    mov rdi, 7
    mov rsi, 5
    add rdi, rsi
    ret
```

Стек и функции

При вызове ret на верхушке стека должен быть адрес возврата. Иначе скорее всего будет ошибка "Segmentation fault":

Поэтому процедура должна извлекать из стека все ранее сохраненные в ней данные и извлекать ровно столько, сколько было сохранено, чтобы адрес возврата сохранялся в стеке и к концу программы оказался в верхушке стека.

Можно использовать этот адрес для выхода из функции:

```
global _start

section .text
_start:
    mov rdi, 5
    mov rsi, 20
    call sum

    add rdi, 10      ; RDI = 15
    mov rax, 60
    syscall

; определяем функцию sum
sum:
    jmp [rsp]        ; переходим по адресу, который храниться в RSP
    add rdi, rsi        ; эта строка НЕ выполняется
    ret
```

Функции могут использовать регистры. Поэтому нужно сохранять нужные регистры перед вызовом функций.

Передача и возврат параметров

Для передачи параметров применяются регистры, стек или через глобальные переменные. Если параметров немного, то через регистры. Наиболее удобным местом для возврата результатов функции в архитектуре x86-64 являются регистры.

Как правило, результат в регистр RAX, хотя можно любой регистр общего назначения. В RAX большинство языков высокого уровня помещают результат функции. Согласно интерфейсам System ABI и Microsoft Windows ABI целочисленный результат помещается в регистр RAX.

Соглашения о вызовах (Calling Conventions) Определяют, как передавать аргументы и кто очищает стек.

Стандартные соглашения (x86)

Соглашение	Передача аргументов	Очистка стека	Используемые регистры

cdecl	Через стек (справа налево)	Вызывающий	`EAX`, `ECX`, `EDX` — не сохраняются
stdcall	Через стек (справа налево)	Вызываемая	`EAX`, `ECX`, `EDX` — не сохраняются
fastcall	Первые 2 — `ECX`, `EDX`, остальные — стек	Вызываемая	`EAX`, `ECX`, `EDX` — не сохраняются

В случае большого объекта можно вместо значения возвратить его адрес (который занимает 8 байт).

При вызове функции доступен весь стек, выделенный в программе. Но функция может иметь свои локальные переменные. Для этого определяется фрейм стека (stack frame) - некоторая область в стеке, которая предназначена для текущей функции, включая адрес возврата, параметры и локальные переменные. Для доступа к фрейму стека предназначен регистр RBP (BP - base pointer или базовый указатель), который представляет указатель на базовый адрес фрейма стека.

```
global _start

section .data
nums dq 10, 20, 30, 15, 15
count equ ($-nums)/numSize    ; количество элементов
numSize equ 8   ; размер каждого элемента

section .text
_start:
    mov rdi, 11       ; в RDI параметр для функции sum
    call sum            ; после вызова в RAX - результат сложения
    mov rdi, rax     ; помещаем результат в RDI
    mov rax, 60
    syscall

sum:
    ; добавляем в стек число 5 - условная безымянная локальная переменная
    push 5          ; RSP указывает на адрес числа 5
    mov rax, rdi    ; в RAX значение параметра из RDI
    add rax, [rsp]  ; rax = rax + [rsp] = rax + 5
    add rsp, 8      ; особождаем стек
    ret
```

Нередко значения параметров, которые передаются через регистры, также помещаются в локальные переменные. Благодаря этому мы сможем высвободить регистры для

вычислений.

```
global _start

section .text
_start:
    mov rdi, 11       ; в RDI параметр для функции sum
    call sum            ; после вызова в RAX - результат сложения
    mov rdi, rax     ; помещаем результат в RDI
    mov rax, 60
    syscall

sum:
    sub rsp, 8         ; резервируем для двух переменных в стеке 8 байт

    mov dword [rsp+4], 5       ; По адресу [rsp+4] первая локальная переменная, которая равна
5
    mov dword [rsp], edi     ; По адресу [rsp] вторая локальная переменная, которая равна EDI

    mov eax, [rsp+4]    ; в EAX значение первой переменной (5)
    add eax, [rsp]     ; EAX = EAX + вторая переменная (edi)

    add rsp, 8           ; особождаем стек
    ret
```

Установка имен переменных

Выше обе наших локальных переменных были безымянными. Для нас фактически они существуют лишь как смещения относительно указателя стека RSP. Однако манипулировать смещения не очень удобно, в процессе написания программы мы можем перепутать спещения. Но с помощью констант мы можем им назначить переменным определенные имена.

```
global _start

_a equ 4    ; смещение переменной _a относительно rsp
_b equ 0    ; смещение переменной _b относительно rsp

section .text
_start:
    mov rdi, 12       ; в RDI параметр для функции sum
```

```
    call sum            ; после вызова в RAX - результат сложения
    mov rdi, rax     ; помещаем результат в RDI
    mov rax, 60
    syscall

sum:
    sub rsp, 8         ; резервируем для двух переменных в стеке 8 байт

    mov dword [rsp+_a], 5      ; По адресу (rsp+4) первая локальная переменная, которая равна
5
    mov dword [rsp + _b], edi    ; По адресу (rsp) вторая локальная переменная, которая равна
EDI

    mov eax, [rsp+_a]     ; в EAX значение первой переменной
    add eax , [rsp + _b]    ; EAX = EAX + вторая переменная

    add rsp, 8           ; особождаем стек
    ret
```

Регистр RBP

Для управления доступом к различным частям фрейма стека Intel предоставляет специальный регистр - RBP (Base Pointer). А для доступа к объектам во фрейме стека можно использовать смещение до нужного объекта относительно адреса из регистра RBP.

Вызывающий функцию код отвечает за выделение памяти для параметров в стеке и перемещение данных параметра в соответствующее место. Инструкция call помещает адрес возврата в стек. Функция несет ответственность за создание остальной части фрейма, в частности, за добавление локальных переменных. Для этого при вызове функции значение RBP помещается в стек (поскольку при вызове функции в RBP значение вызывающего кода, и это значение надо сохранить), а значение указателя стека RSP копируется в RBP. Затем в стеке освобождается место для локальных переменных.

Для доступа к объектам во фрейме стека необходимо использовать смещение до нужного объекта относительно адреса из регистра RBP. Для обращения к параметрам, которые передаются через стек, применяется положительное смещение относительно значения регистра RBP, а для доступа к локальным переменным - отрицательное смещение. Следует с осторожностью использовать регистр RBP для общих расчетов, потому что если вы произвольно измените значение в регистре RBP, вы можете потерять доступ к параметрам текущей функции и локальным переменным.

```
global _start
```

```
section .text
_start:
    mov rdi, 11       ; в RDI параметр для функции sum
    call sum            ; после вызова в RAX - результат сложения
    mov rdi, rax     ; помещаем результат в RDI
    mov rax, 60
    syscall

sum:
    push rbp              ; сохраняем старое значение RBP в стек
    mov rbp, rsp         ; копируем текущий адрес из RSP в RBP
    sub rsp, 16          ; выделяем место для двух переменных по 8 байт

    mov qword[rbp-8] , 7      ; По адресу [rbp-8] первая локальная переменная, равная 7
    mov qword [rbp-16], rdi    ; По адресу [rbp-16] вторая локальная переменная, равная RDI

    mov rax, [rbp-8]    ; в RAX значение из [rbp-8]  - первая локальная переменная
    add rax, [rbp-16]    ; RAX = RAX + [rbp-16] - вторая локальная переменная

    mov rsp, rbp         ; восстанавливаем ранее сохраненное значение RSP
    pop rbp               ; восстанавливем RBP

    ret
```

Инструкции enter и leave

Поскольку данная схема работа с регистром %rbp довольно распространена, то для упрощения ассемблер NASM предоставляет две дополнительные инструкции. Так, вместо кода:

```
push rbp
mov rbp, rsp
sub rsp, N_байтов
```

Можно применять следующую инструкцию:

```
enter N_байтов, 0
```

Инструкции enter передается выделяемое в стеке количество байт, а второй параметр - число 0. При выполнении эта инструкция сама сохранит старое значение %rbp в стек, скопирует значение rsp в rbp и выделит в стеке N_байтов.

А вместо кода

```
mov rsp, rbp
pop rbp
```

Можно применить

```
leave
```

Revision #6
Created 10 November 2025 07:08:23 by Admin
Updated 11 November 2025 05:34:15 by Admin