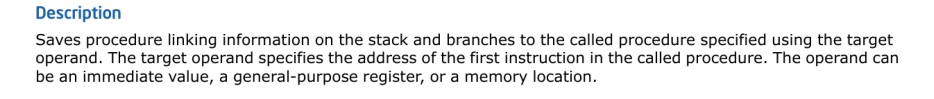
**Вопрос** Выполните эту программу по шагам в gdb и проследите за указателем на стек (регистром rsp). Как на него влияют инструкции call и ret?

при вызове call -> значение rsp уменьшается на 8 (байт)

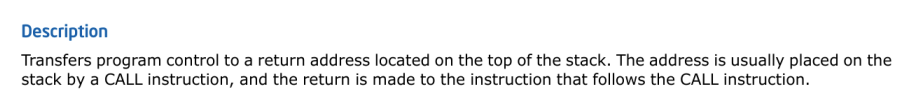
при вызове ret -> значение rsp увеличивается на 8 (байт)

**Вопрос** Обратитесь к Intel software developer manual и прочитайте страницы, соответствующие инструкциям call и ret. Опишите действие этих инструкций на память и регистры. страница 712

страница 712



т.е. call уменьшает rsp на 8 и записывает адрес текущей исполняемой инструкции (значение регистра rip) в стек (по адресу rsp ибо rsp - stack pointer) и переходит по адресу которого мы указали в качестве аргумента



т.е. ret совершает обратное действие: пихает в указатель текущей исполняемой инструкции(rip) адрес с вершины стека и увеличивает его(rsp) на 8

таким образом реализуется вызов функций и возврат из них

**Вопрос** Что, если функция f вызывает функцию g? Напишите программу, где есть такие функции и вызовы; выполните её по шагам в gdb и проследите за указателем на стек (регистром rsp).

- происходит логика описанная выше.

пример программы:

global \_start

section .text

f:

call g

ret

g:

ret

exit:

mov rax, 60

xor rdi, rdi

syscall

\_start:

call f

call exit

**Вопрос** Почему мы не написали инструкцию ret в конце функции exit?

потому что мы исполнили системный вызов sys\_exit(0) т.е. сказали операционной системе что корректно завершаем программу. После завершения проги возвращаться нам уже и некуда

mov rax, 60

xor rdi, rdi

syscall

в rax кладется номер системного вызова 60=exit а с помощь rdi мы передали первый и единственный аргумент данной функции. Она принимает код возврата, ну а так как мы совершили операцию xor rdi с самой rdi, мы по сути ее обнулили (просто xor rdi, rdi работает чуть быстрее чем mov rdi, 0). По итогу мы передали системному вызову exit код возврата 0, что означает корректное завершение работы программы

**Вопрос** Выделите из следующего кода (взято со стр. 22 книги “Low-level programming”) функцию print\_hex, которая примет аргумент в правильном регистре (не rax) и выведет его на экран. Выведите с её помощью любые три числа.

section .text

global\_start

exit:

mov rax, 60 ; invoke 'exit' system call

xor rdi, rdi

syscall

print\_hex:

mov rax, rdi

mov rdi, 1

mov rdx, 1

mov rcx, 8 одна 16ричная цифра однозначно кодируется 4мя битами так что тут мы должны указать число равное 4\*кол-во символов в строке. в данном примере я вывожу двухсимвольное число AB. Если захочешь поменять его длину меняй и это значение

.loop:

push rax

sub rcx, 4

sar rax, cl

and rax, 0xf

lea rsi, [codes + rax]

mov rax, 1

push rcx

syscall

pop rcx

pop rax

test rcx, rcx

jnz .loop

ret

\_start:

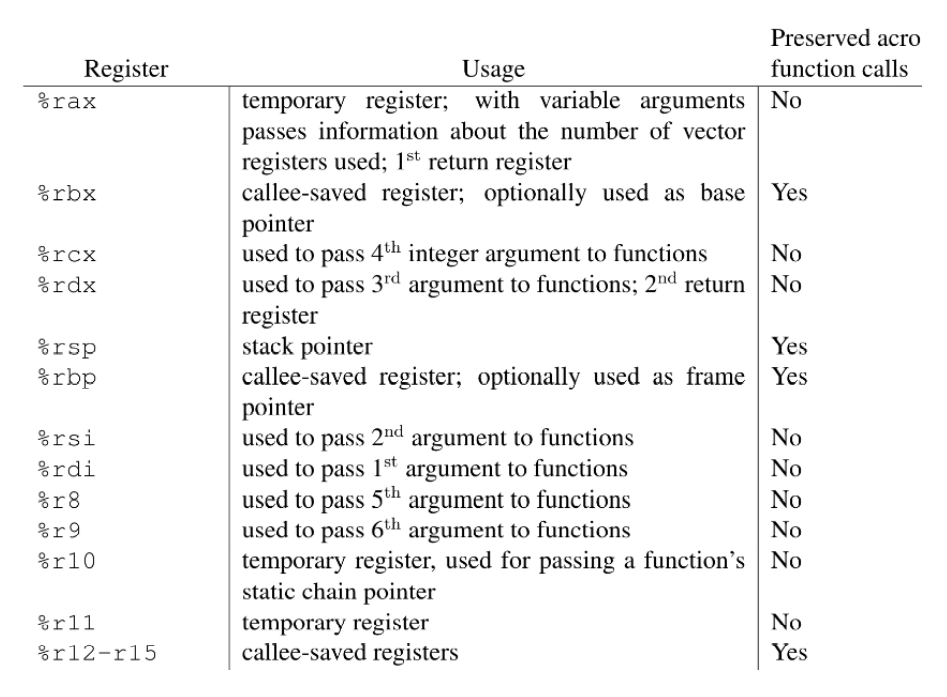
mov rdi, 0xab а вот и само число

call print\_hex

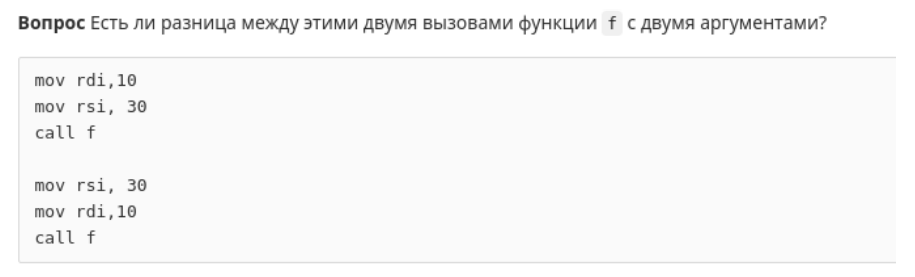
call exit

Функции могут иметь множество аргументов. Первые шесть они получают через регистры.

**Вопрос** Какие регистры для этого используются? Ищите ответ в секции 2.4 “Low-level programming”.



это супер-пупер полезная табличка! вот они регистры для передачи аргументов слева направо: rdi rsi rdx rcx r8 r9 😊



Нет никакой разницы. Мы распихали одни и те же значения в регистры перед вызовом f, просто в разном порядке.

**Вопрос** Сколько аргументов нужно передать функции print\_string, чтобы она могла вывести любую строку с помощью системного вызова write? Хватит ли одного? Да, хватит одного. Это будет указатель на начало строки (правда sys\_write( , , ) принимает 3 аргумента: сначала дескриптор потока [sysout имеет дескриптор 1] далее указатель на начало строки далее количество символов в ней. Т.е. придется сначала посчитать кол-во символов)

**Вопрос** Прочитайте секцию 2.5.2 из книги “Low-level programming”. Перепишите следующую программу так, чтобы функция print\_string принимала нуль-терминированную строку в единственном аргументе строки.

global \_start

section .data

msg: db "Hello!", 0

section .text

\_start:

mov rdi, msg

call print\_string

call exit

; Принимает код возврата и завершает текущий процесс

exit:

mov rax, 60

syscall

ret

; Принимает указатель на нуль-терминированную строку, возвращает её длину

string\_length:

xor rax, rax

.loop:

cmp byte[rdi+rax], 0

je .end

inc rax

jmp .loop

.end:

ret

; Принимает указатель на нуль-терминированную строку, выводит её в stdout

print\_string:

call string\_length

mov r11, rax

mov rax, 1

mov rdx, r11

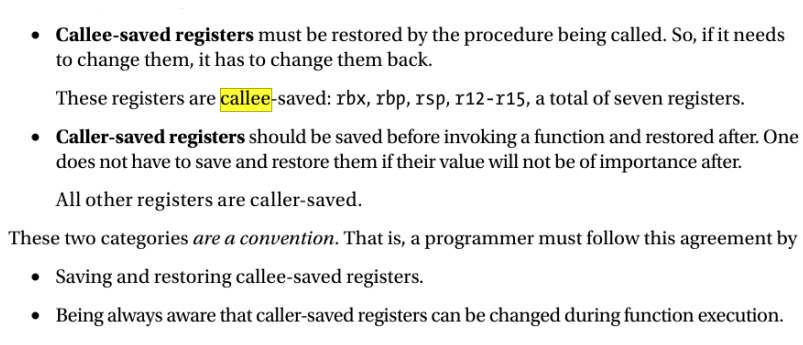
mov rsi, rdi

mov rdi, 1

syscall

ret

**Вопрос** Что такое callee-saved и caller-saved регистры? В чём разница между их использованием при вызове функций?



если изменяем callee-saved регистр где-либо в функции, то мы обязаны перед возвратом из нее положить в регистры изначальные их значения. Все потому, что эти регистры задействованы в работе с памятью и их изменение приведет к ошибкам в дальнейшем.

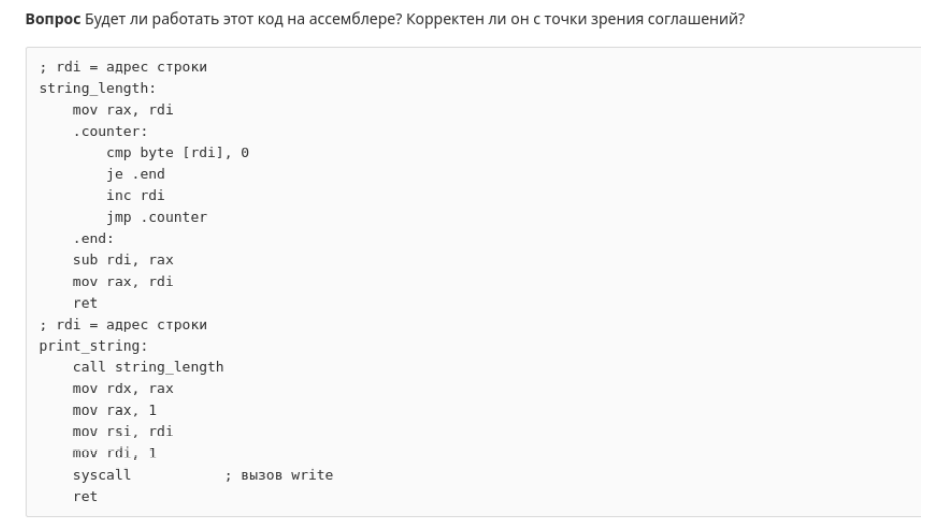
за caller-saved регистрами тебе нужно следить, т.е. если меняем в одной функции их значение, а в другой ожидаем взять из них какое-нибудь старое значение, надо прежде позаботиться о том, чтоб заранее где-либо сохранить из значение (например, в стеке с помощью инструкции push). Как видно на скрине, rbx rbp rsp r12-15 являются callee-saved, остальные caller-saved

**Вопрос** Почему не сделать все регистры callee-saved?

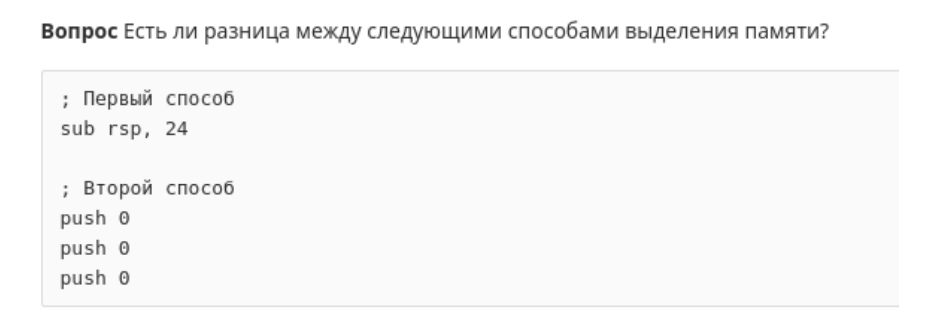
Большинство регистров не имеют таких важных обязанностей как, например, rsp, поэтому не имеет смысла так ограничивать работу с ними, более того, тогда нам было бы крайне проблематично возвращать какие-либо значения из функций и т.п.



Тут всё шик-блеск



Работать будет, но мы тут изменили значение rdi, не сохранив его перед вызовом функции. При том этот регистр являлся указателем на начало строки, так что больше мы не сможем использовать его по назначению.



Нет, просто в первом случае в выделенном месте будет мусор, а во втором нули.

**Вопрос** Напишите функцию, которая выделит место под три локальные переменные, запишет туда aa, bb и ff и выведет их на экран. Затем она должна корректно завершиться. Протестируйте её. Для вывода на экран используйте уже написанную в начале семинара print\_hex.

section .data

codes:

db '0123456789ABCDEF'

section .text

global \_start

exit:

mov rax, 60 ; invoke 'exit' system call

xor rdi, rdi

syscall

print\_hex:

; number 1122... in hexadecimal format

mov rax, rdi

mov rdi, 1

mov rdx, 1

mov rcx, 8

; Each 4 bits should be output as one hexadecimal digit

; Use shift and bitwise AND to isolate them

; the result is the offset in 'codes' array

.loop:

push rax

sub rcx, 4

; cl is a register, smallest part of rcx

; rax -- eax -- ax -- ah + al

; rcx -- ecx -- cx -- ch + cl

sar rax, cl

and rax, 0xf

lea rsi, [codes + rax]

mov rax, 1

; syscall leaves rcx and r11 changed

push rcx

syscall

pop rcx

pop rax

; test can be used for the fastest 'is it a zero?' check

; see docs for 'test' command

test rcx, rcx

jnz .loop

ret

f:

push rdi

push rsi

push rdx

pop rdi

call print\_hex

pop rdi

call print\_hex

pop rdi

call print\_hex

ret

\_start:

mov rdi, 0xaa

mov rsi, 0xbb

mov rdx, 0xcc

call f

call exit