Function Calls and Stack

A function stack also known as call stack or execution stack is a data structure used in computer systems to represent the continous memory available for the programs to run. When a function is called a new stack frame will be created.

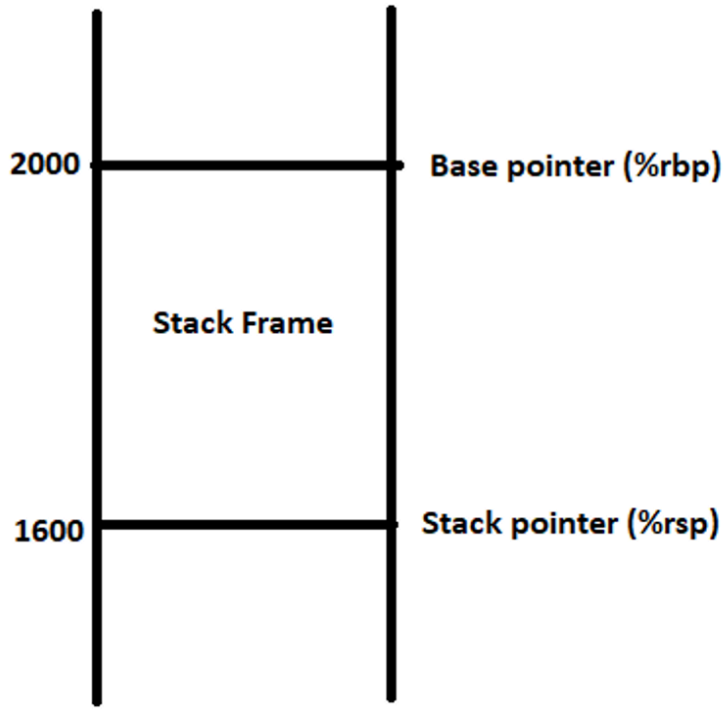
A stack frame is essentially a subset or part of the stack which is defined by starting point and an ending point. Stack frame starts from a particular address and grows downwards i.e. lets say the stack frame of a particular function is starting from address 2000. Then as more things are pushed to the stack the top most point of the stack keeps going down that is when 400 bytes of data is pushed to stack, top pointer of stack will point to 1600.

Since a stack frame is represented by a starting point and an ending point, it is clear that we need to have two variables to denote the same. One for the bottom of stack and another for the top of stack. In assembly it is generally done by using **rbp** and **rsp** registers.

%rbp - points to the base of the stack (starting point of the stack)

%rsp - points to the top of the stack (ending point of the stack)

The following diagram shows a simple representation of how stack frame looks like :



How to insert data to stack and how to retreive data from stack ?

→ Inorder to insert data to the stack in assembly we can use **push** instruction.

```
pushq %rbx # push the data present in %rbx to stack pushq $2
```

→ What pushq essentially does is, it subtract 8 from %rsp to get the new stack top. And writes the
data given in the address from rsp to (rsp + 8).
→ So in the above example after executing pushq %rbx, rsp will be updated to hold the value 1592.
→ The exact same result can be obtained by following two instructions :

```
subq $8, %rsp movq %rbx, (%rsp)
```

→ Similarly inorder to obtain the data from top of the stack we can use **pop** instruction. It will store
the data at the top of the stack in the register passed as argument.

```
popq %rax # data stored at the top of stack is copied to %rax
```

→ popq is exact opposite of pushq, so what it essentially does is, copy the data present at the top
of the stack to given argument then add 8 to the value stored in %rsp
→ So similar result can be obtained by following two instructions :

```
movq (%rsp), %rax addq $8, %rsp
```

Now lets write a simple program to add 3 numbers using stack for storing values :

```
.global main main: pushq $4 # x = 4 pushq $8 # y = 8 pushq $20 # z = 20 movq
(%rsp), %rax # rax = z addq 8(%rsp), %rax # rax = rax + y addq 16(%rsp),
%rax # rax = rax + x popq %rbx popq %rbx popq %rbx # you must use pop the
same number of times as you used push. (restore the rsp value) ret
```

→ Now if we add one more variable in the stack, by `pushq $4`, the relative position of all the
previous variables from rsp will change. i.e. `x` will be `24(%rsp)` instead of `16(%rsp)` Therefore,
its easier if we can have a base pointer which remains constant throughout the program.

→ The above program can be re-written as follows :

```
.global main main: pushq %rbp # saving callee saved registers in stack movq %rsp, %rbp # initialising the base pointer pushq $4 # x = 4 pushq $8 # y = 8 pushq $20 # z = 20 movq -24(%rbp), %rax # rax = z addq -16(%rbp), %rax # rax = rax + y addq -8(%rbp), %rax # rax = rax + x movq %rbp, %rsp popq %rbp ret
```

→ Now even if we push one more variable to the stack the relative address of previous variables
from `%rbp` remain the same as `%rbp` itself is constant throught out the function.

Calling functions in Assembly :

```
.global main main: pushq %rbp movq %rsp, %rbp movq $4, %rdi movq $5, %rsi # setting up arguments for calling the function call add # calling add function similar to add(4, 5) in c movq %rbp, %rsp # restoring rsp popq %rbp ret add: pushq %rbp movq %rsp, %rbp movq %rdi, %rax addq %rsi, %rax movq %rbp, %rsp popq %rbp ret
```

→ Remember that the registers are similar to global variables and if you modify register in any part
of the program it gets modified everywhere.

What call does.?

→ It pushes the present address to the stack which is essentially the return address for the function

being called once its done executing. Then it updates the program counter to point to the

address of the function being called. (Remember this is done by processor and you need not do

these manually, you just need to call the function).

```
# Internals of call instruction pushq <present address> pc = <address of function being called>
```

What ret does.?

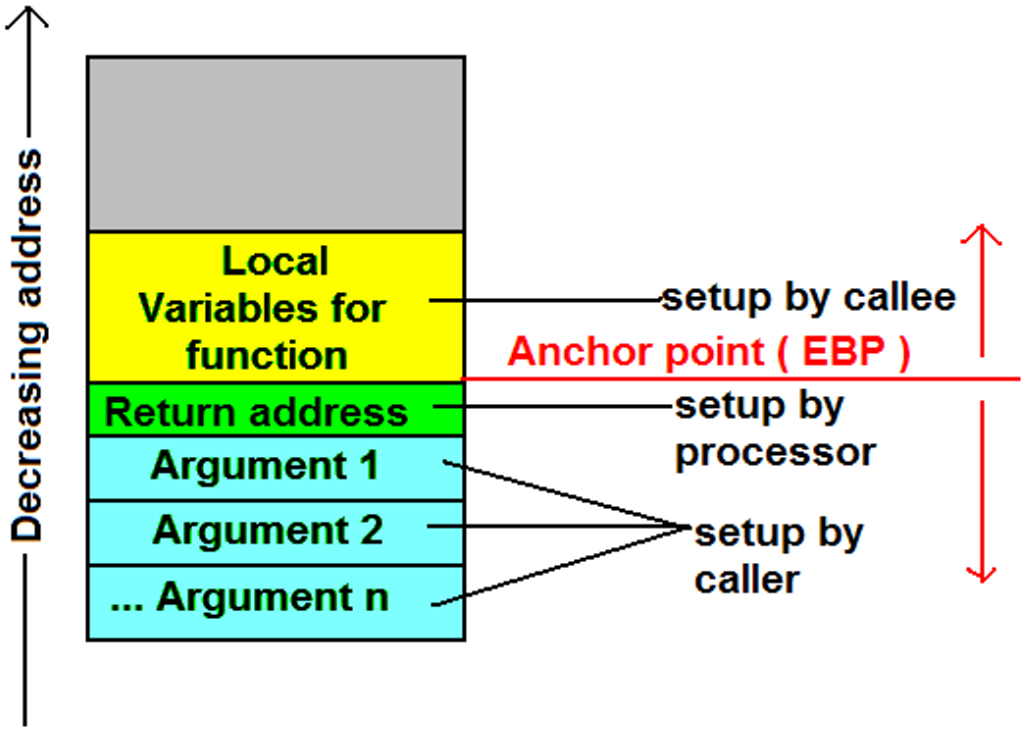
→ It pops the value present at the address pointed by rsp to the program counter.

```
# Internals of ret instrucion popq PC # PC is updated to hold the address pointed by %rsp
```

→ So while executing ret instruction if rsp is not at correct position i.e. the position at which it was, when program started execution your program will try to return to some other address which might result in segmentation fault.

How to handle more than 6 arguments of function .?

→ This is part of the assignment and for the reference purposes you can use the following diagram.



→ This was explained during the tutorial with a clear example.

Caller saved registers and Callee saved registers

→ This is just a convention followed while using multiple functions in a single program to avoid
unintended overwriting of registers
→ Caller saved registers are those registers which caller itself should store before calling any other
function and there is no guarantee that the register values will remain the same once the control
returns back to the caller
→ Caller saved registers : `%rax, %rcx, %rdx, %rdi, %rsi, %r8, %r9,  %r10, %r11`

→ Callee saved registers are those registers which are guaranteed to retain their value even if the
control goes to other functions. That means no function should modify the values present in
these registers. Even if it does modify, it should revert the registers to their original values before
returning.
→ Callee saved registers : `%rbx, %rbp, %r12, %r13, %r14, %r15`

How to write a proper function in assembly accounting all the caller and callee saved registers

```
.global main main: pushq %rbp pushq %rbx pushq %r12 pushq %r13 pushq %r14
pushq %r15 # all callee saved registers are pushed to stack to preserve
their values movq %rsp, %rbp # base pointer is initialised # function body
starts ... some instructions ... # now before calling any function save all
the caller saved registers. pushq %rax pushq %rcx pushq %rdx pushq %rdi
pushq %rsi pushq %r8 pushq %r9 pushq %r10 pushq %r11 # all caller saved
registers are preserved # now call any function you want movq $4, %rdi movq
$5, %rsi call add # before continuing with the function retain values of
caller saved registers ... pop caller saved registers .... ... other
instructions of function ... # function body ends movq %rbp, %rsp #
restoring the stack pointer popq %r15 popq %r14 popq %r13 popq %r12 popq
%rbx popq %rbp # all callee saved registers retain their value. ret # please
note the order of pushing and popping add: ... push callee saved registers
... ... function instructions .... ... pop callee saved registers ... ...
```