Function Calls Run-time Steps and Memory Layout in x86 Architecture

By: Humayun Kabir

When I learned functions, I also learned that if a function calls another function the caller function loses the control of execution to the called function and a new stack frame, inside the stack segment of the run-time memory, gets wounded for the called function. When the called function returns, the stack frame gets unwounded and the caller function gets the control back. For many years, I took these transitions between the caller and the called functions as granted and did not dive deep to understand how these are actually happening. There are two main reasons for my ignorance. One, I was still a good coder without knowing the details of the behind the scene activities. And the other, I did not get a good resource that elaborates the behind the scene activities in a lay man's terms. In this article, I am going to fill the gap by elaborating how the transitions between the caller and the called functions happen behind the scene. I will use an example C code and walk through its step by step execution to explain the transitions. In order to add visuals to my explanation, I will show related stack frames and their contents at various transition levels.

In our everyday code, both the function call and the return from a function are a single statement or a single step. That's a very simplistic view of a function call and a function return. In reality, both need to take many steps and use 3 CPU registers in x86 architecture at run-time. These CPU registers are **Index Pointer** (**%eip**), **Stack Base Pointer** (**%ebp**), and **Stack Pointer** (**%esp**). Register **%eip** points to the next instruction in the code to be executed. Register **%eip**, is also called instruction pointer. Execution control of the code jumps from one function to another function when **%eip** register is loaded with the address of the instruction of the target function. This is happening both at function call and at function return. Register **%ebp** points to the base or start address of the current stack frame in the stack segment. For this reason, **%ebp** register is also called frame base register. Register **%esp** points to the end of the current stack frame, which is also the end of the stack segment. Stack segment starts from the higher memory address and grows downward as the new stack frames are wounded one after another. A new stack frame is wounded into the stack segment at every function call. Stack segment shrinks upwards as the stack frames are unwounded one after another. A stack frame is unwounded from the stack segment at every function return. **Figure 1** shows the run-time memory layout and the associated CPU registers of an executing code.

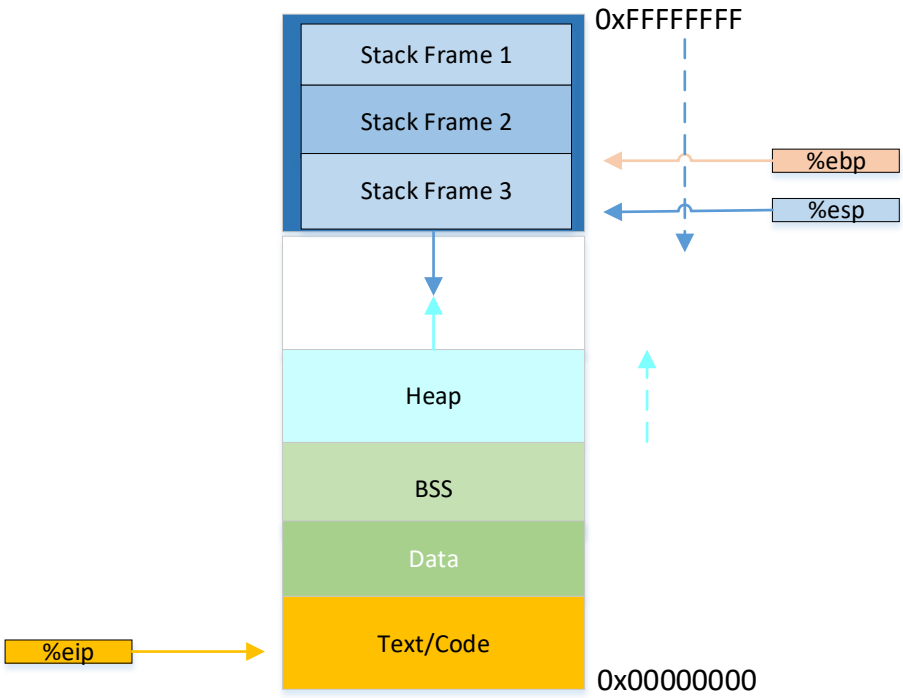

Figure 1

The **caller** and the **called functions** together need to perform **12 behind the scene run-time steps** to get the transitions of execution control between them. For C and C++ code, compiler inserts the assembly instructions of these steps into the caller and the called functions.

1. **Caller** function **pushes** the **function parameters** of the called function, from right to left, onto the stack. As the stack grows downward, the stack pointer (**%esp**) is decremented accordingly.

2. **Caller** function **pushes** the **current %eip** value, which points to the next instruction of the caller function to be executed, onto the stack (**%esp** is decremented). Later, this is used as the **return address** when the called function returns. Caller updates the **%eip** to point to the beginning of the called function code. After step 2, **caller function loses the control to the called function**.

3. **Called** function **pushes** the **current %ebp** onto the stack (**%esp** is decremented) and updates the **%ebp** to the current value of **%esp** to point to the base of its own stack frame. **A new stack frame for the called function gets wounded** at this point.

4. **Called** function **allocates memory** from the stack for its **local variables** one after another. Each allocation decrements **%esp** appropriately.

5. If the **called** function needs to use any CPU register, it **saves** the current value of **that CPU register** onto the stack and decrements **%esp** accordingly. Steps 4 and 5 can happen intermittently based on the called function's code.

6. **Called** function **finishes executing** itself.

7. Before return the **called** function **releases memory** from its stack frame that was allocated for its **local variables**. This operation shrinks the current stack frame by incrementing **%esp** value. This step starts unwinding of current stack frame.

8. Before return the **called** function **restores the CPU registers** that were saved onto its stack frame and releases the corresponding memory by incrementing **%esp** value. This operation also shrinks the current stack frame. Steps 7 and 8 can happen intermittently in the order that took place when the memory was allocated in steps 4 and 5. After the completion of steps 7 and 8 **%esp** points to the location of the **saved %ebp** of the previous stack frame.

9. Before return **called** function **restores the saved %ebp** (**%ebp** of the previous stack frame) of the caller function to **%ebp** register and releases the corresponding stack memory by incrementing **%esp** value appropriately. **This step completes the unwinding of the called function's stack frame**. After this step **%esp** points to the **saved %eip** (**return address**) of the caller function.

10. Before return the **called** function **restores the saved %eip** (**return address**) of the caller function to **%eip** register and releases the corresponding stack memory by incrementing **%esp** value appropriately. After this step **%esp** points to the called function's parameters that were pushed by the caller function, if there were any. At this point **called function returns** or loses the control to the caller function.

11. **Caller** function **clears up** the **pushed parameters** from the stack (increments **%esp** accordingly), if there is any, before resuming its execution.

12. **Caller** function **resumes** its execution at the code pointed by **%eip** (return address).

I am using **example C code** shown in **Figure 2** to demonstrate above behind the scene run-time steps. To make the discussion simple, I am assuming the operating system is not running any other process while running this code's process. The OS does not use virtual memory and the running process gets full physical memories in its address space. I am also ignoring the segments registers that point to the beginning of the segments (Code, Data, and Stack etc.). The example code has three functions: **main()**, **absSum()**, and **abs()**. Execution starts with **main()** function. Function **main() calls** local function **absSum()** once and the C library function **printf()** once. Function **absSum() calls** function **abs()** twice. Function **abs()** does not call any other function. **Figure 3** shows the call sequence of the example code.


```c
#include <stdio.h>

int abs(int g) {
    int h = g;
    if(g<0) {
        h = -g;
    }
    return h;
}

int absSum(int d, int e) {
    int f = abs(d) + abs(e);
    return f;
}

int main() {
    int a = -5;
    int b = 10;
    int c = absSum(a, b);
    printf("Absolute Sum: %d", c);
    return 0;
}
```

Figure 2

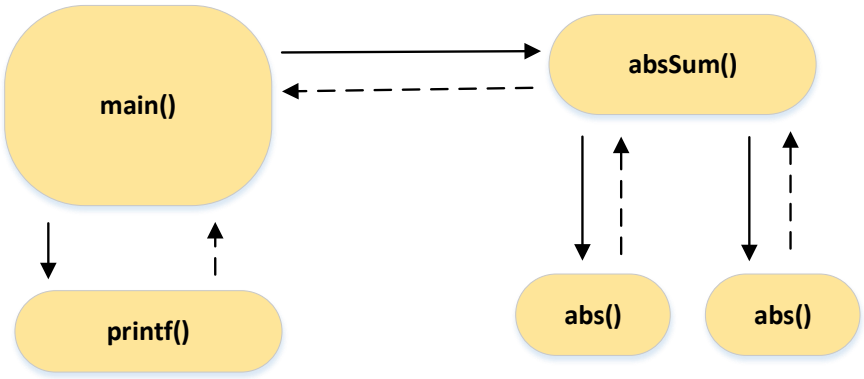

Figure 3

Figure 4 shows the run-time memory layout when the code has been just loaded into the memory.

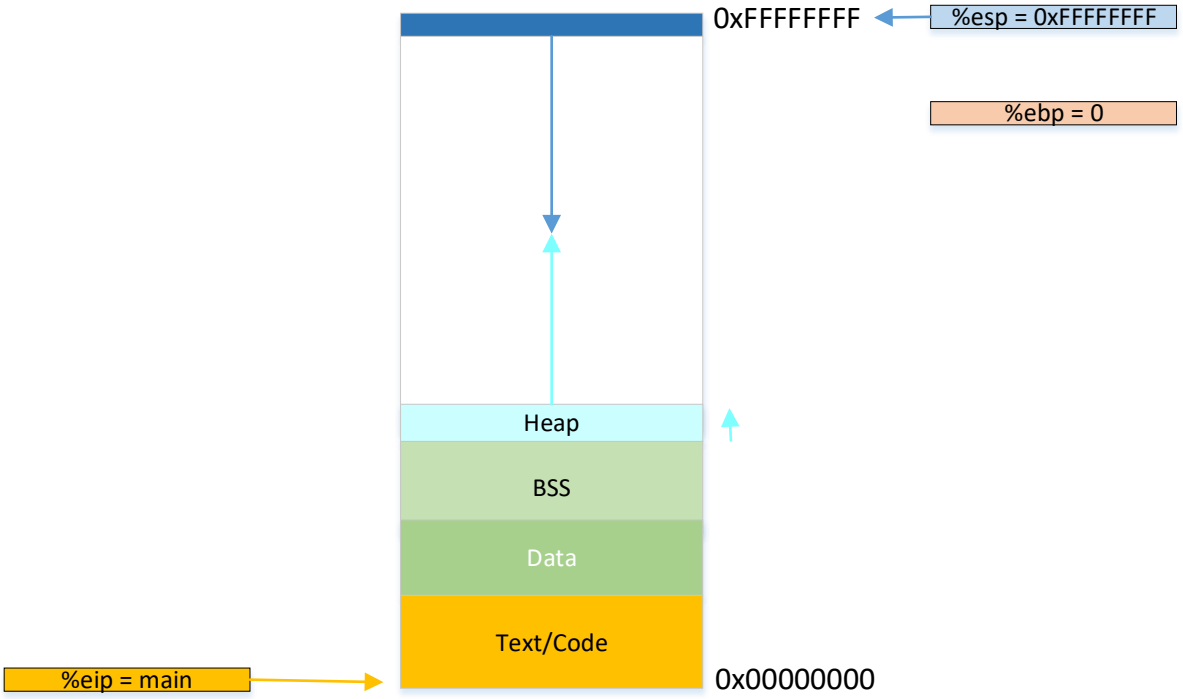

Figure 4

I am assuming that the code has been loaded into **Text/Code segment** starting at 0x00000000. As the example code has no global or static variable, initialized or uninitialized, both **Data** and **BSS** segments are empty. As this code does not allocate memory dynamically, the **Heap** segment is also empty. I am showing Heap, Data and BSS segments in Figure 3 for sanity reason. Initially, the **Stack segment** is also empty. Am assuming that the stack segment started from 0xFFFFFFFF. Initially, **%esp** register points to 0xFFFFFFFF address, **%eip** register points to the memory address in the Code segment corresponds to function main(). In the real world, **%eip** register points to the memory address of an assembly code instruction (C compiler translates C code into assembly code). In order to make the discussion simple to follow for C programmers, **%eip** register in this article points to the imaginary addresses of C statements.

Figure 5 shows the run-time memory layout after **main()** function has been **called** and it is about to call absSum() function.

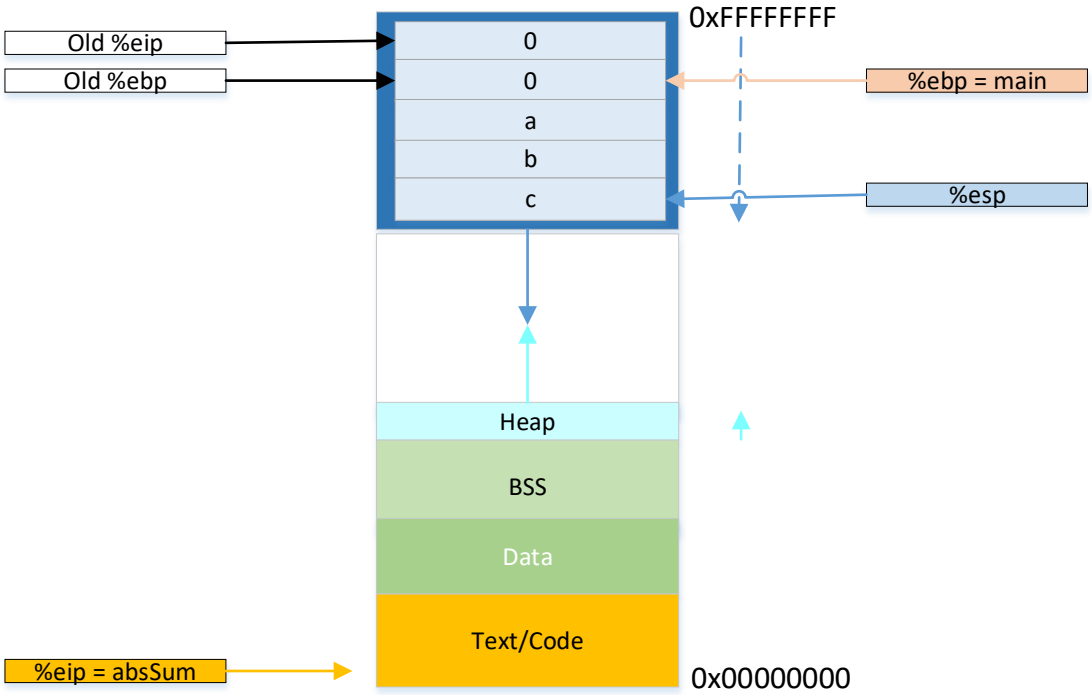

Figure 5

In order to make the discussion simple, I assume initial value of both **%eip** and **%ebp** are zero (not pointing to anything). The **caller** of function main() skipped **step 1** as no parameter has been passed into main(). The **caller** of main() pushed current **%eip** value (0) onto the stack and loaded the address of the first instruction of function main() into **%eip** register (**step 2**). Function **main()** function got the control of execution and pushed the current **%ebp** value (0) onto the stack and updated **%ebp** to the base of its own frame stack (**step 3**). Function **main()** allocated memory for its local variables a, b, and c (**step 4**). After step 4, register **%ebp** points to the base or the start of the frame stack related to main() function, register **%eip** points to the memory address in the Code segment that corresponds to absSum() function call inside function main(), and register **%esp** points to the end of the current stack segment (i.e., the address in the stack segment corresponds to main() function's local variable c).

Figure 6 shows the run-time memory layout after function **absSum()** has been **called** and it is about to call function abs(d).

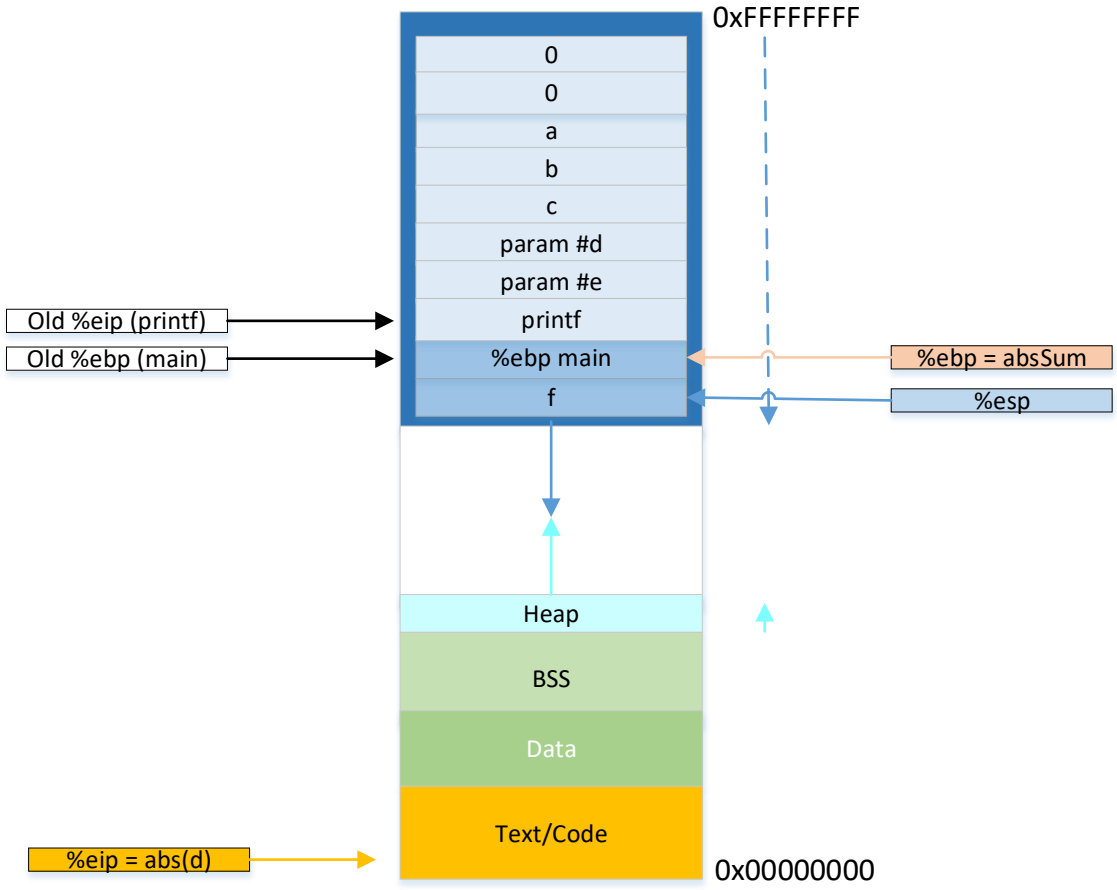

Figure 6

Function **main()** pushed parameters d and e as it has called function absSum() **(step 1**). Function **main()** also pushed the current **%eip** (printf) onto the stack and loaded the address of the first instruction of function absSum() into **%eip** register (**step 2**). Function **absSum()** got the control of execution and pushed main() function's **%ebp** onto the stack and updated **%ebp** to the base of its own frame stack (**step 3**). Function **absSum()** allocated memory for its local variable f (**step 4**). After step 4, register **%ebp** points to the base of the frame stack related to absSum() function, register **%eip** points to the memory address in the Code segment that corresponds to abs(d) function call inside absSum() function, and register **%esp** points to the end of the current stack segment (i.e., the address corresponds to absSum() function's local variable f).

Figure 7 shows the run-time memory layout after **abs(d)** function has been **called** and it is about to return.

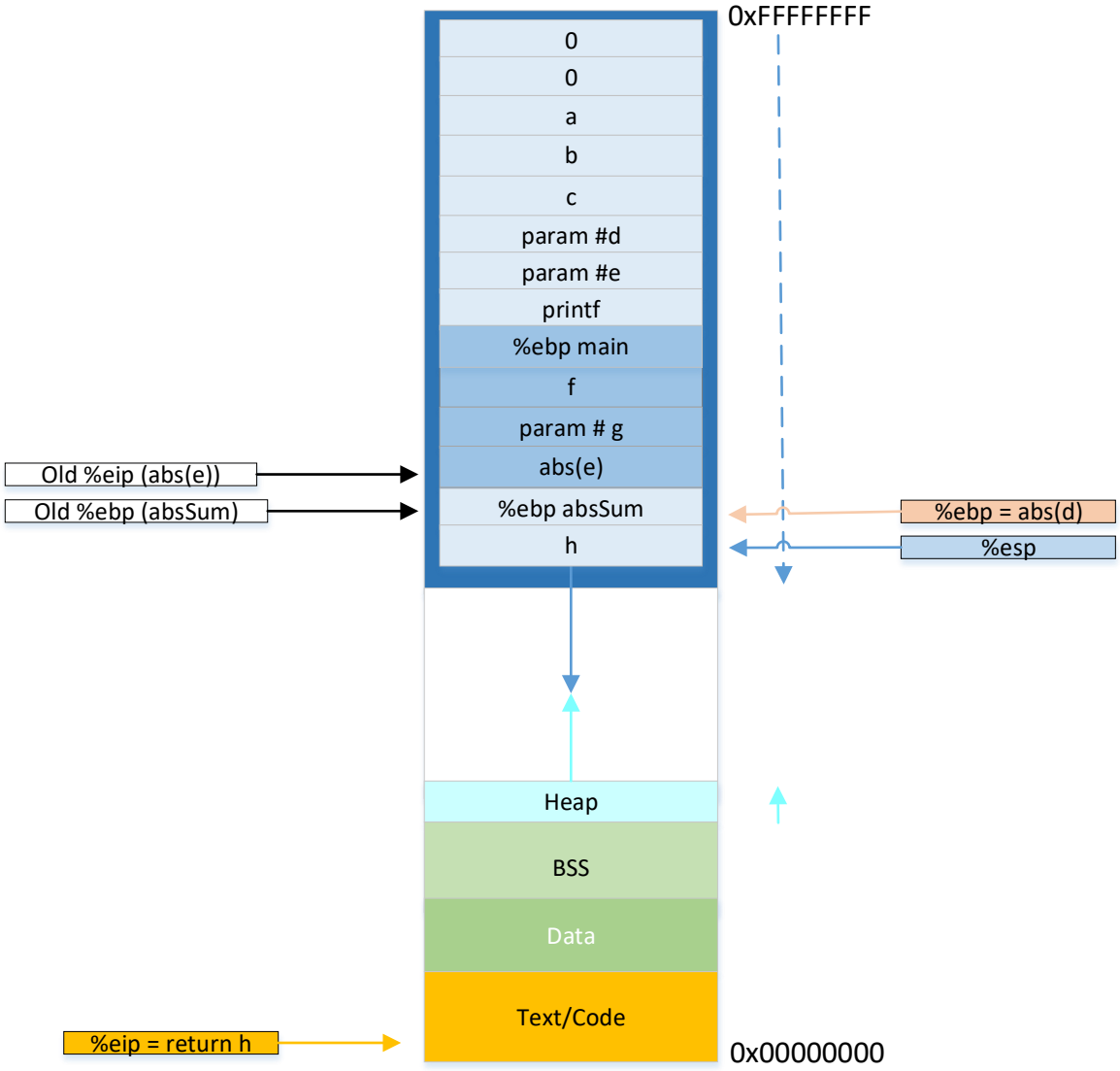

Figure 7

Function **absSum()** pushed parameter g, as it has called function abs(d) (**step 1**). Function **absSum()** also pushed the current **%eip** (abs(e)) onto the stack and loaded the address of the first instruction of function abs(d) into **%eip** register (**step 2**). Function **abs(d)** got the control of execution and pushed absSum() functions **%ebp** onto the stack and updated **%ebp** to the base of its own frame stack (**step 3**). Function **abs(d)** allocated memory for its local variable h (**step 4**). After step 4, register **%ebp** points to the base of the frame stack related to abs(d) function, register **%eip** points to the memory address in the Code segment that corresponds to return h inside abs(d) function, and register **%esp** points to the end of the current stack segment (i.e., the address corresponds to abs(d) function's local variable h).

Figure 8 shows the run-time memory layout after function **abs(d)** has **returned** and function absSum() is about to call function abs(e).

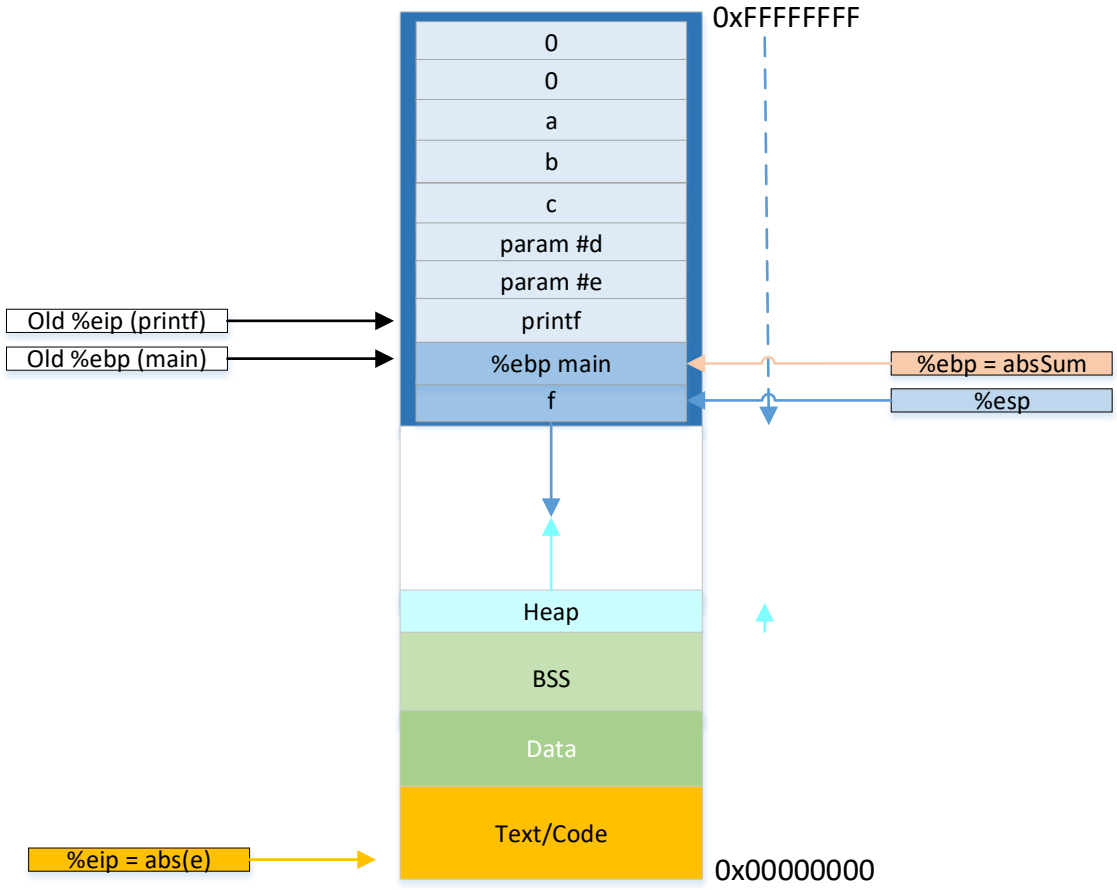

Figure 8

Function **abs(d)** did not use any CPU register and skipped **step 5**. Function **abs(d)** has finished its execution (**step 6**). Function **abs(d)** released memory from its stack frame that was allocated for its local variable h (**step 7**). Function **abs(d)** skipped **step 8**. Function **abs(d)** restored **%ebp** register to point the base of the frame stack of function absSum() (**step 9**). The stack frame related to function abs(d) has been unwounded. Function **abs(d)** restored **%eip** register to point to the saved return address (abs(e)) of function absSum() (**step 10**). Function **absSum()** got the control back and cleared the pushed parameter g from the stack (**step 11**). After step 11, register **%esp** points to the end of the current stack segment (i.e., the address corresponds to absSum() function's local variable f).

Figure 9 shows the run-time memory layout after **abs(e)** function has been **called** by function absSum() and function abs(e) is about to return.

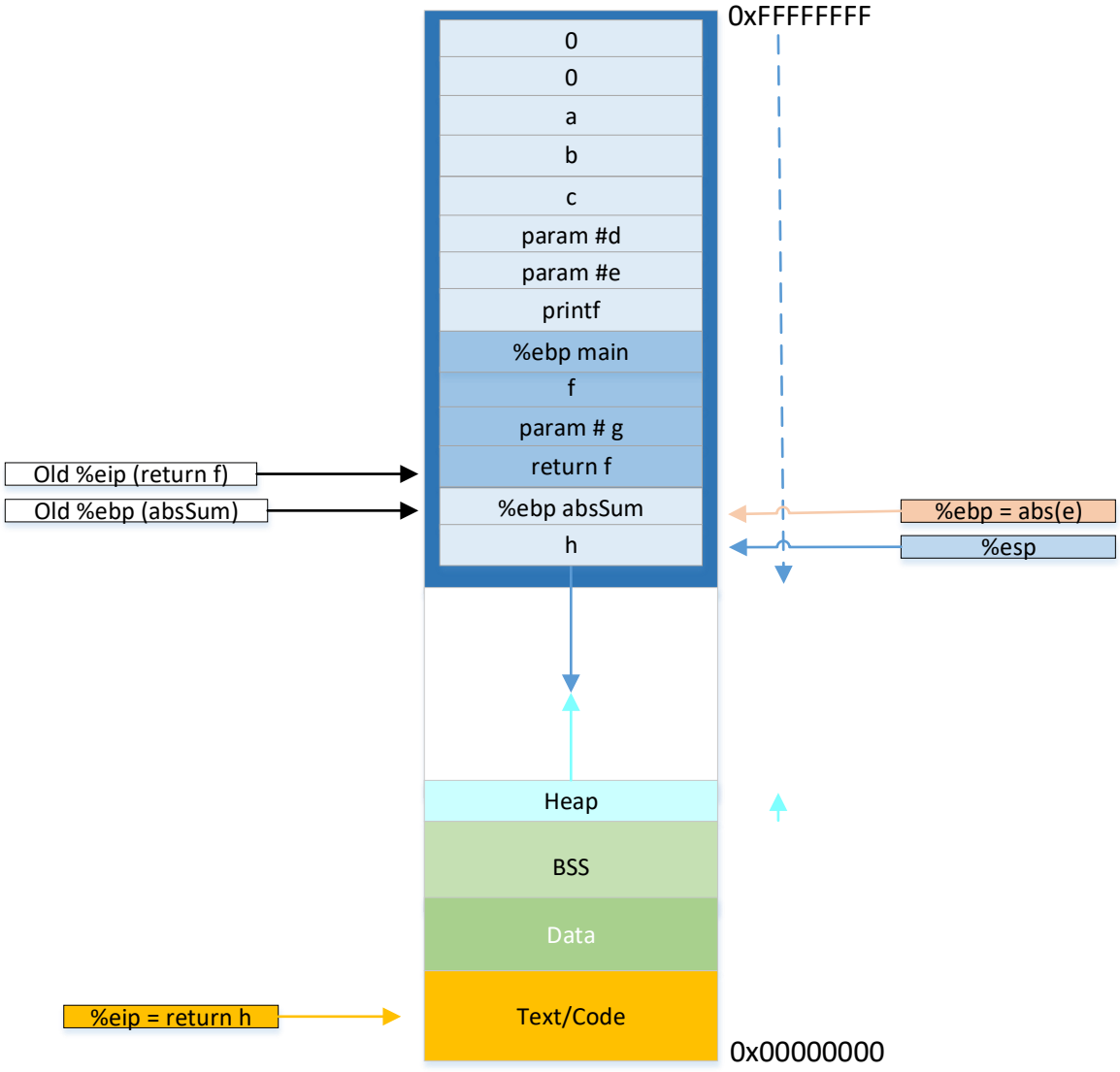

Figure 9

Function **absSum()** pushed parameter g, as it has called function abs(e) (**step 1**). Function **absSum()** also pushed the current **%eip** (return f) onto the stack and loaded the address of the first instruction of function abs(e) into **%eip** register (**step 2**). Function **abs(e)** got the control of execution and pushed absSum() functions **%ebp** onto the stack and updated **%ebp** to the base of its own frame stack (**step 3**). Function **abs(e)** allocated memory for its local variable h (**step 4**). After step 4, register **%ebp** points to the base of the frame stack related to abs(e) function, register **%eip** points to the memory address in the Code segment that corresponds to return h inside abs(e) function, and register **%esp** points to the end of the current stack segment (i.e., the address corresponds to abs(e) function's local variable h).

Figure 10 shows the run-time memory layout after function **abs(e)** has **returned** to function absSum() and function absSum() is about to return to function main().

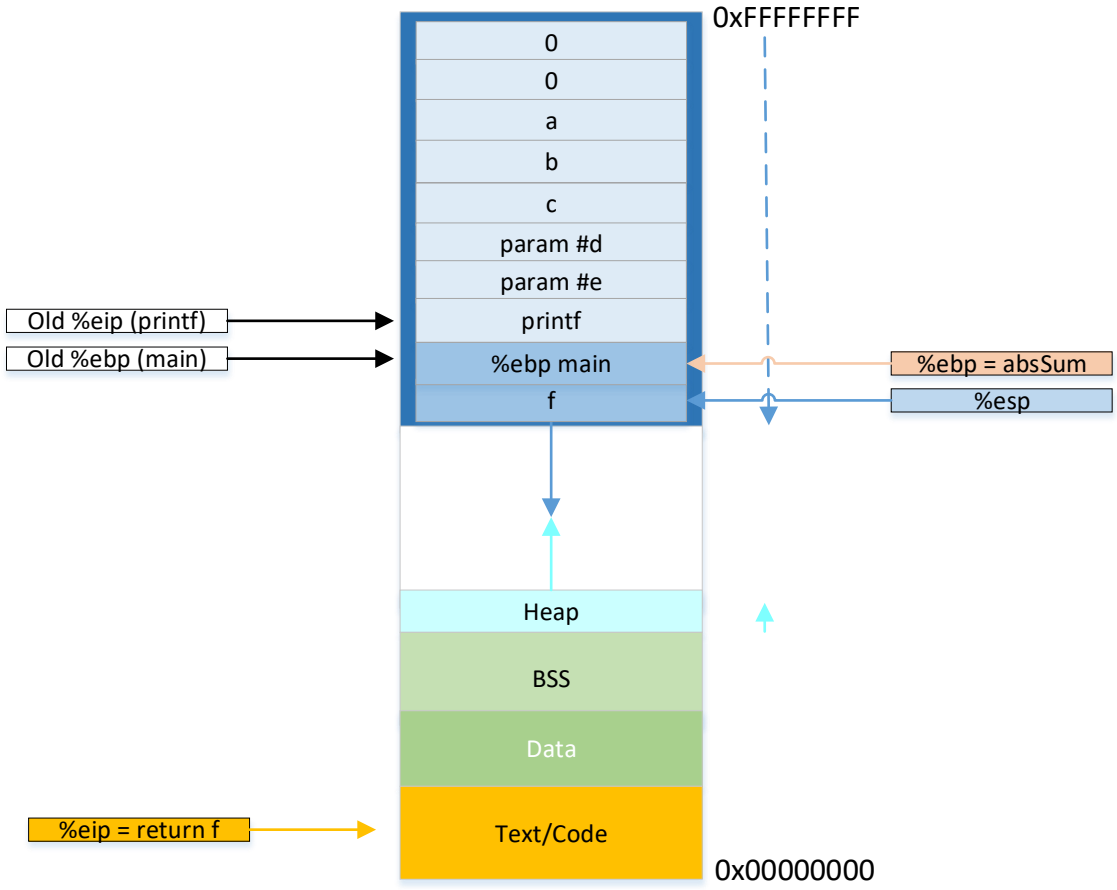

Figure 10

Function **abs(e)** did not use any CPU register and skipped **step 5**. Function **abs(e)** has finished its execution (**step 6**). Function **abs(e)** released memory from its stack frame that was allocated for its local variable h (**step 7**). Function **abs(e)** skipped **step 8**. Function **abs(e)** restored **%ebp** register to point to the base of the frame stack of function absSum() **(step 9**). The stack frame related to function abs(e) has been unwounded. Function **abs(e)** restored **%eip** register to point to the saved return address (return f) of function absSum()(**step 10**). Function **absSum()** got the control of execution and cleared the pushed parameter g from the stack (**step 11**). After step 11, register **%esp** points to the end of the current stack segment (i.e., the address corresponds to absSum() function's local variable f).

Figure 11 shows the run-time memory layout after function **absSum()** has **returned** to function main() and function main() is about to call function printf().

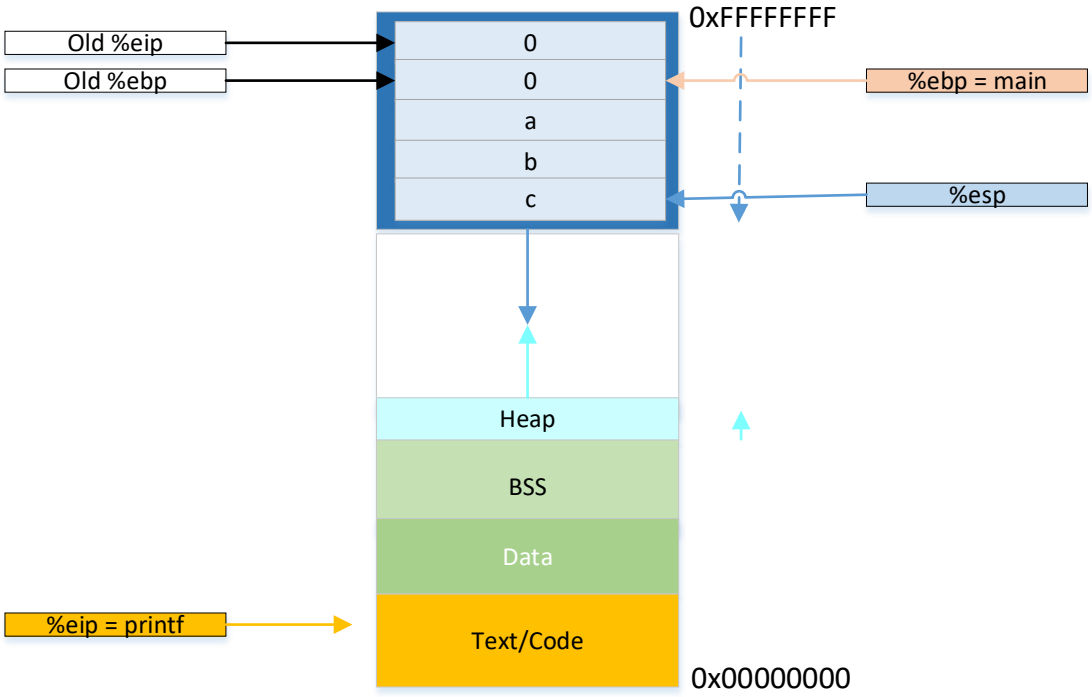

Figure 11

Function **absSum()** did not use any CPU register and skipped **step 5**. Function **absSum()** has finished its execution (**step 6**). Function **absSum()** released memory from its stack frame that was allocated for its local variable f (**step 7**). Function **absSum()** skipped **step 8**. Function **absSum()** restored **%ebp** register to point to the base of the frame stack of function main() (**step 9**). The stack frame related to function absSum() has been unwounded. Function **absSum()** restored **%eip** register to point the saved return address (printf) of function main() (**step 10**). Function **main()** got the control of execution and cleared the pushed parameters d and e from the stack (**step 11**). After step 11, register **%esp** points to the end of the current stack segment (i.e., the address corresponds to main() function's local variable c).

Figure 12 shows the run-time memory layout after **printf()** function has been **called** by function main() and function printf() is about to return.

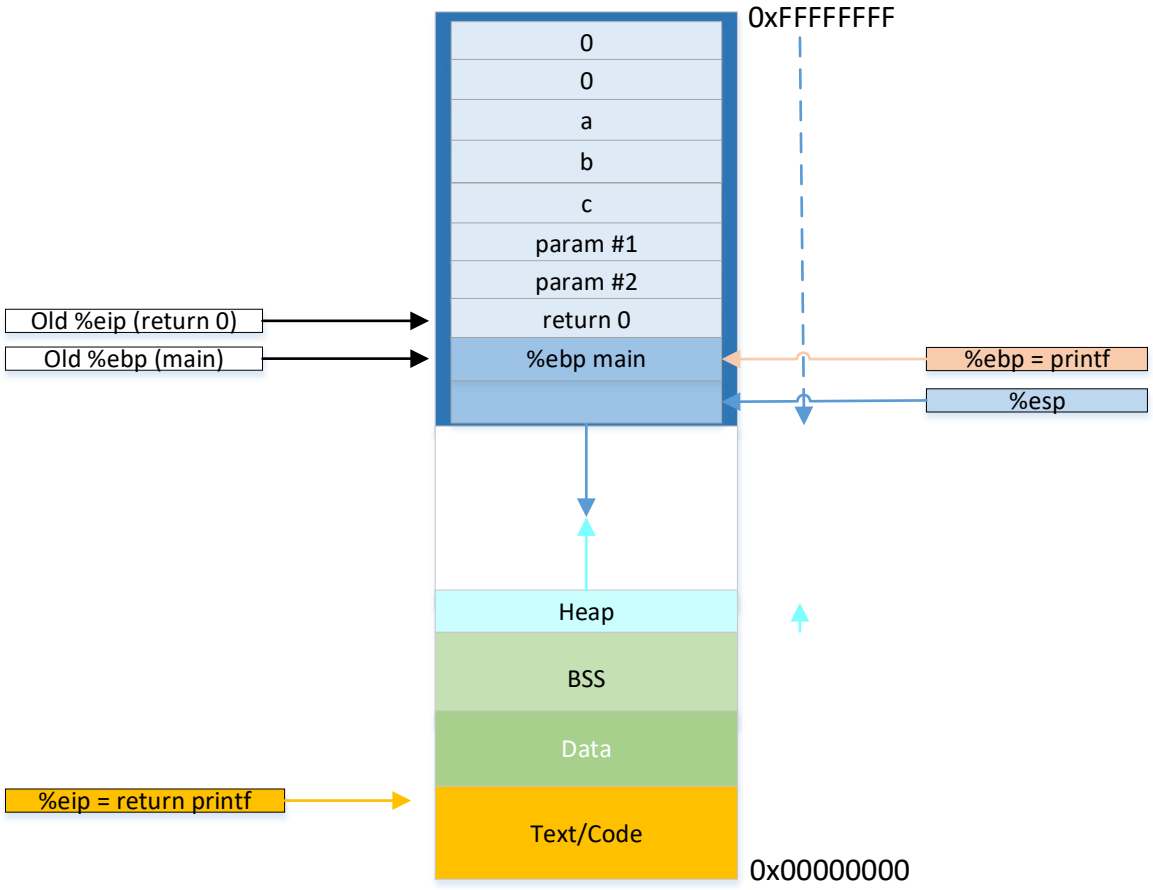

Figure 12

Function **main()** pushed parameters #1 and #2 as it has called function printf() (**step 1**). Function **main()** also pushed the current **%eip** (return 0) onto the stack and loaded the address of the first instruction of function printf() into **%eip** register (**step 2**). Function **printf()** got the control of execution and pushed main() functions **%ebp** onto the stack and updated **%ebp** to the base of its own frame stack (**step 3**). Function **printf()** allocated memory for its local variables (**step 4**). Figure 11 does not show printf() function's local variables. Function **printf()** might have used CPU registers. It saved the current value of the registers onto the stack (**step 5**). Figure 11 does not show printf() function's saved registers either. After step 5, register **%ebp** points to the base of the frame stack related to printf() function, register **%eip** points to the memory address in the Code segment that corresponds to return inside printf() function, and register **%esp** points to the end of the current stack segment (i.e., the address corresponds to printf() function's local variables).

Figure 13 shows the run-time memory layout after function **printf()** has **returned** to function main() and function main() is about to return.

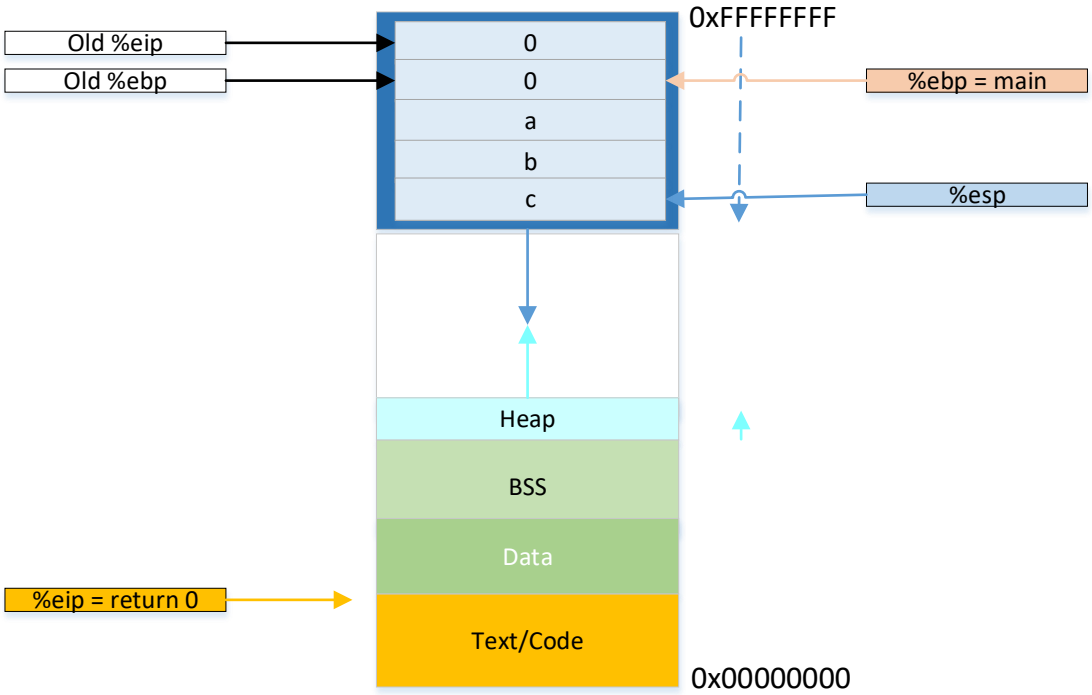

Figure 13

Function **printf()** might have used CPU registers saved their previous value onto the stack in **step 5**. Function **printf()** has finished its execution (**step 6**). Function **printf()** has released memory from its stack frame that was allocated for its local variables (**step 7**). Functions **printf()** restored saved CPU register values and released corresponding stack memory (**step 8**). Function **printf()** restored **%ebp** register to point to the base of the frame stack of function main() (**step 9**). The stack frame related to function printf() has been unwounded. Function **printf()** restored **%eip** register to point to the saved return address (return 0) of function main() (**step 10**). Function **main()** got the control of execution and cleared the pushed parameters #1 and #2 from the stack (**step 11**). After step 11, register **%esp** points to the end of the current stack segment (i.e., the address corresponds to main() function's local variable c).

Figure 14 shows run-time memory layout after function **main()** has **returned**.

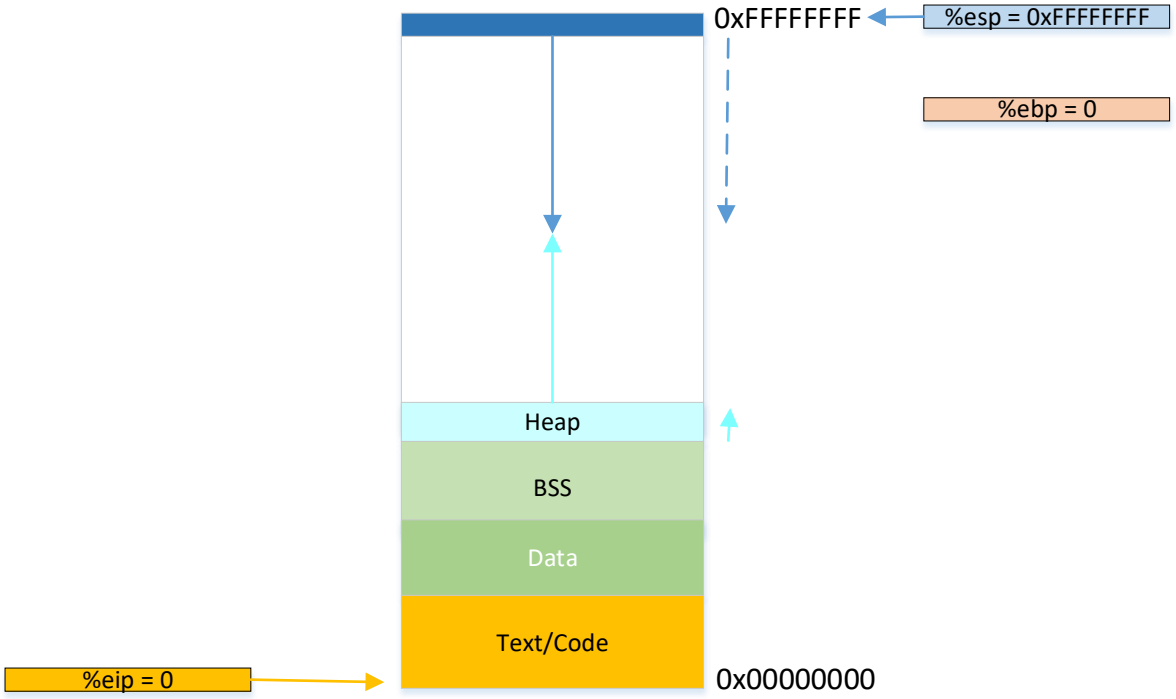

Figure 14

Function **main()** did not use any CPU register and skipped **step 5**. Function **main()** has finished its execution (**step 6**). Function **main()** released memory from its stack frame that was allocated for its local variables a, b, and c (**step 7**). Functions **main()** skipped **step 8**. Function **main()** restored **%ebp** register to point nothing (**step 9**). The stack frame related to function main() has been completely unwounded. Function **main()** restored **%eip** register to point to nothing (**step 10**). The **caller** of function main() skipped **step 11**. After step 10, register **%esp** points to the end of the current stack segment (i.e., the address 0xFFFFFFFF).

Using **Figures 4** to **14**, I have demonstrated how the behind the scene run-time steps are taken place and how the stack segment grows and shrinks along with the execution of the example code. I have demonstrated the new stack frames that are wounded at function calls as well as the contents of these stack frames. I have demonstrated the smooth unwinding of the stack frames at function returns. I have demonstrated how the execution control jumps from the caller functions to the called functions by loading **%eip** register with the address of the instruction of the called functions. I have demonstrated how the execution control comes back from the called functions to the caller functions by loading **%eip** register with the address of the next instruction of the caller functions. I will consider my effort successful only if the readers of this article get a clear understanding of these steps and concepts. Please, feel free to reach me at humayun.kabir@viu.ca for comments and questions.

