THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

CS330
C to Assembly

Spring 2022

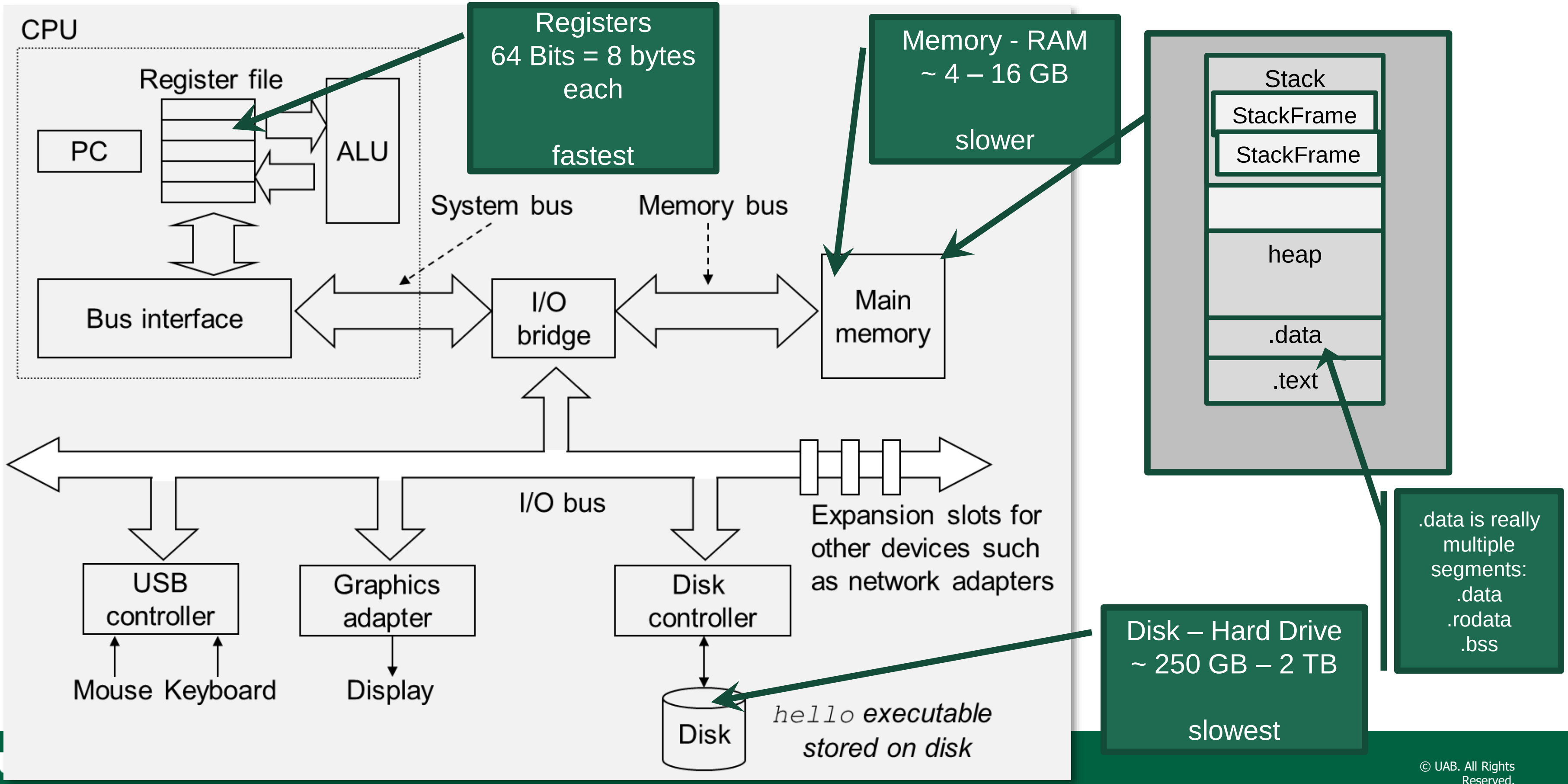
Lab 8

Or Higher Order Language Appreciation Week

Helpful Hints

- Assembly is just like procedural programming in other languages, but the building blocks are smaller, and there are a lot less helper tools
  - The process is the same - Break the problem down into smaller pieces
  - Use the tools in our toolkit to solve the problem, build up the tools we need
- It’s a manual drive, nothing is automatic -- We need to keep a lot of program state info in our head, or better, write it down, e.g. what’s in memory, and where. Or “where did I leave my variable?”
  - A quick sketch of the Registers and Stack can be extremely helpful
- The computer is a shared space (shared resources), following the rules / contract is key to harmony
  - Register management: pass/receive args from functions in certain registers, caller and callee save responsibilities
  - Stack management: leave it like you found it, check stack alignment
- Code documentation is key. It’s not uncommon to have more comment lines than code.
  - # single line comment
  - /* block comment */
- We’ll use x86 AT&T syntax in this course
  - Be careful, there is a lot of Intel syntax (and other syntax, and bad info) on the interwebs

High Level (just for today) Computer Architecture



Registers
64 Bits = 8 bytes each

fastest

Memory - RAM
~ 4 – 16 GB

slower

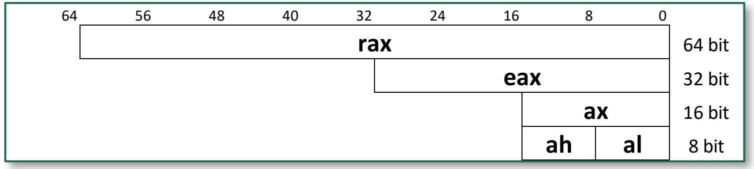
Disk – Hard Drive
~ 250 GB – 2 TB

slowest

.data is really multiple segments:
.data
.rodata
.bss

Registers

- 16 General Purpose Registers
- Register names per
  AT&T syntax
- Will not use floating, vector registers in this course
- Can also access subsets



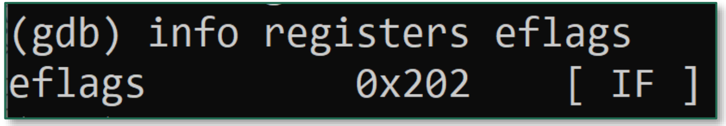
Register	Usage	Old Names	Args	Saved by	Preserved Across Function Calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; **1st return register**	accumulator		**Caller**	No
%rbx	callee-saved register; optionally used as base pointer	base		**Callee**	Yes
%rcx	used to pass 4th integer argument to functions	counter, loop counter	4	**Caller**	No
%rdx	used to pass 3rd argument to functions; **2nd return register**	data	3	**Caller**	No
%rsp	stack pointer	stack pointer		**Callee**	Yes
%rbp	callee-aved register, optionally used as frame pointer	base pointer		**Callee**	Yes
%rsi	used to pass 2nd argument to functions	source index	**2**	**Caller**	No
%rdi	used to pass 1st argument to functions	destination index	**1**	**Caller**	No
%r8	used to pass 5th argument to functions		5	**Caller**	No
%r9	used to pass 6th argument to functions		6	**Caller**	No
%r10	temporary register, used for passing a function's static chain pointer			**Caller**	No
%r11	temporary register			**Caller**	No
%r12 - r15	callee-saved registers			**Callee**	Yes

GNU Assembler (AS) Manual: https://sourceware.org/binutils/docs/as/index.html#SEC_Contents

eflags

(and how to view registers)

- In GDB
  (i)nfo (r)egisters eflags

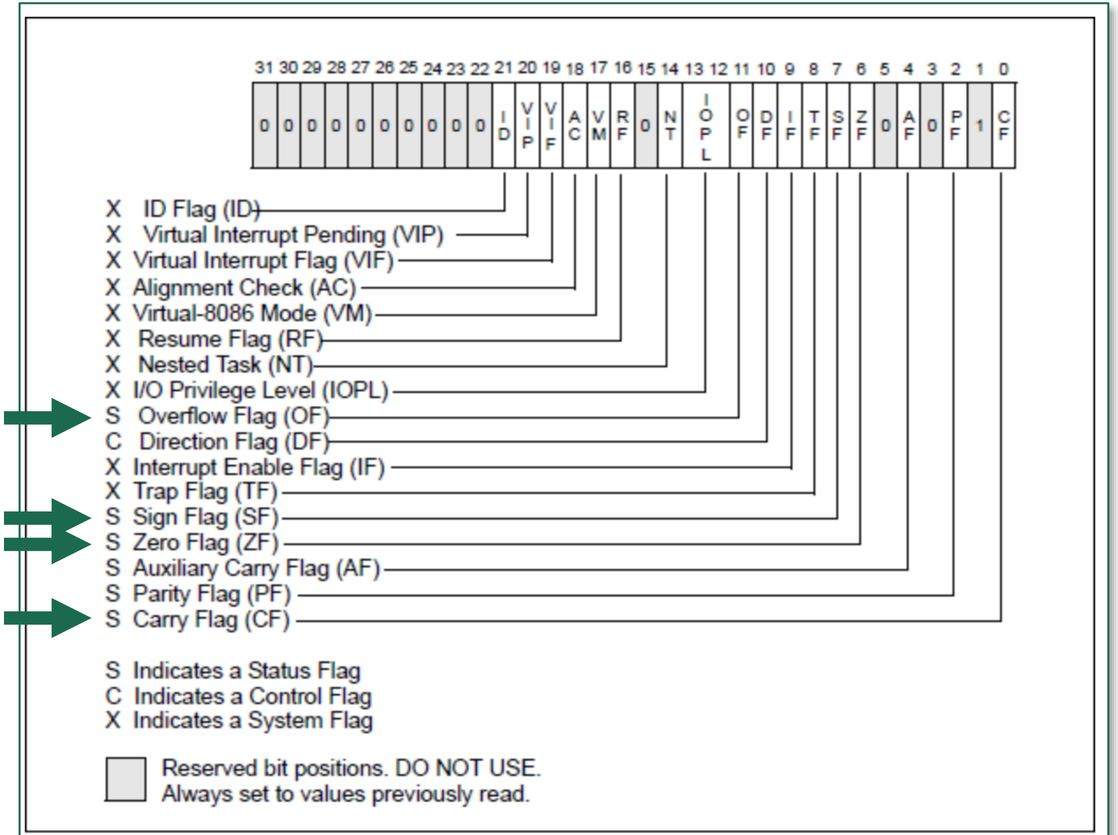


```
(gdb) info registers eflags
eflags         0x202     [ IF ]
```

- To show all general purpose registers, including %rip (instruction pointer), eflags
   (i)nfo (r)egisters all

  or individually via
    (i)nfo (r)egisters $<name>
  e.g. (i)nfo (r)egisters $rax

  or
    tui reg general



X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

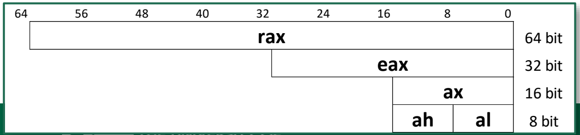
□ Reserved bit positions. DO NOT USE.
Always set to values previously read.

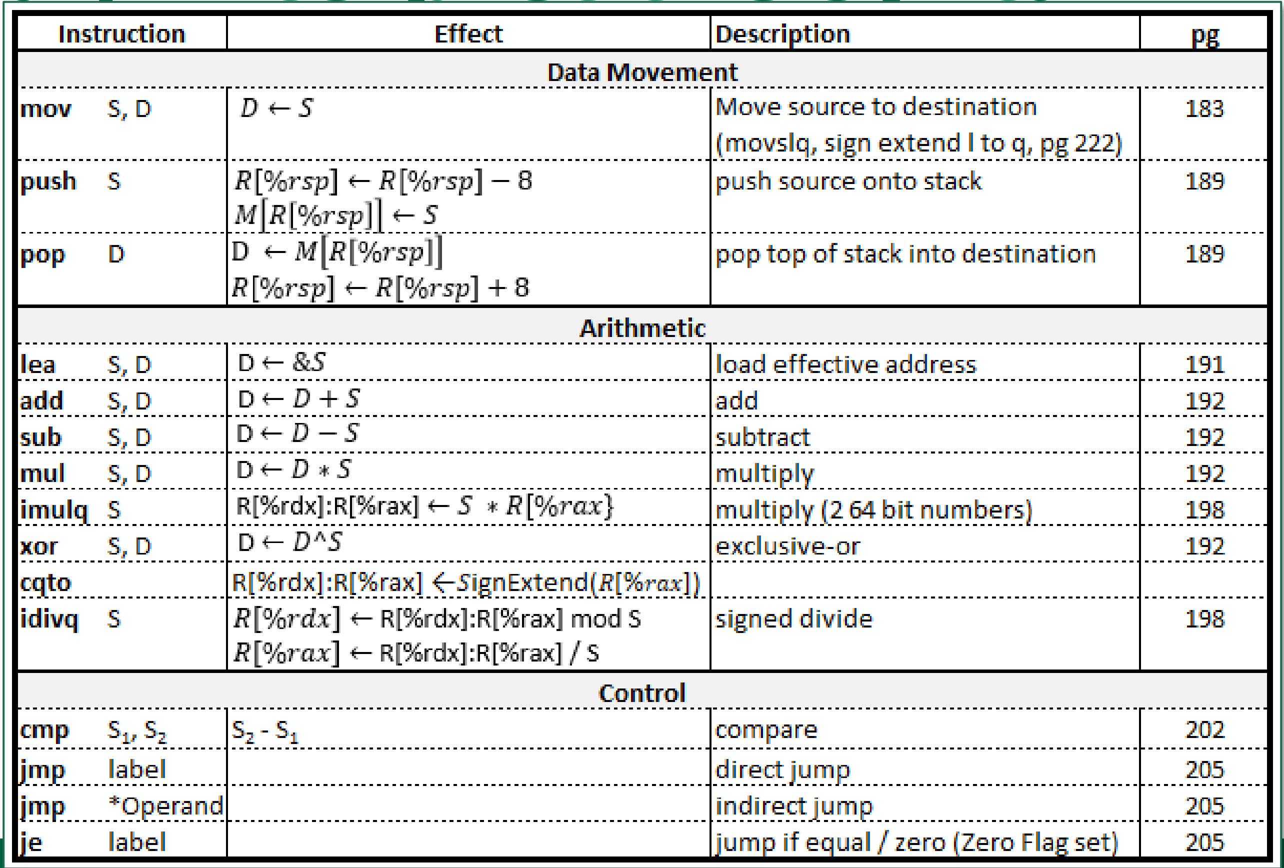
EFLAGS Register

How to read / interpret the syntax

- Typical AT&T mnemonics use three letter instructions with a one letter suffix to represent the size

Suffix		
b	byte	1 byte
w	word	2 bytes
l	doubleword	4 bytes
q	quadword	8 bytes





Instruction	Effect	Description	pg
Data Movement			
mov S, D	$D \leftarrow S$	Move source to destination (movslq, sign extend l to q, pg 222)	183
push S	$R[\%rsp] \leftarrow R[\%rsp] - 8$ $M[R[\%rsp]] \leftarrow S$	push source onto stack	189
pop D	D $\leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	pop top of stack into destination	189
Arithmetic			
lea S, D	D $\leftarrow \&S$	load effective address	191
add S, D	D $\leftarrow D + S$	add	192
sub S, D	D $\leftarrow D - S$	subtract	192
mul S, D	D $\leftarrow D * S$	multiply	192
imulq S	R[%rdx]:R[%rax] $\leftarrow S \; * R[\%rax\}$	multiply (2 64 bit numbers)	198
xor S, D	D $\leftarrow D$^S	exclusive-or	192
cqto	R[%rdx]:R[%rax] $\leftarrow$SignExtend($R[\%rax]$)		
idivq S	$R[\%rdx] \leftarrow$ R[%rdx]:R[%rax] mod S $R[\%rax] \leftarrow$ R[%rdx]:R[%rax] / S	signed divide	198
Control			
cmp S_1, S_2	S_2 - S_1	compare	202
jmp label		direct jump	205
jmp *Operand		indirect jump	205
je label		jump if equal / zero (Zero Flag set)	205

S = Source, D = Destination

Operands take one of these three forms

1. Immediate / Literal: $4
2. Register: %rax
3. Memory

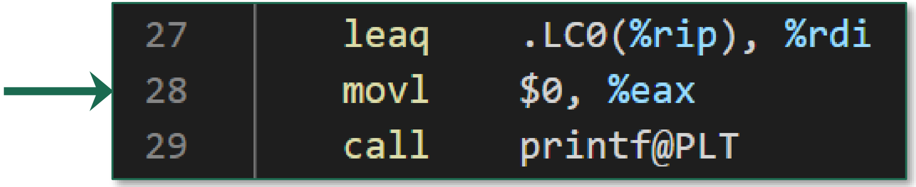
Type	From	Operand Value	Name
Immediate	\$Imm	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	M[Imm]	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + (R[r_i] * s)]$	Scaled Indexed

(see Book, pg 181 for more)

- Imm refers to a constant value, e.g. 0x8048d8e, 48
- r_a refers to a register
- $R[r_a]$ refers to the value stored in register r_a
- M[x] refers to the value stored at memory address x

Note: can’t move (mov) from Memory to Memory

- multiple levels of understanding
  1. The Syntax, what does the specific word mean? e.g movl
  2. The Semantics, what does the word mean in context, ie. what does the sentence mean? E.g. movl $0, %eax
     what does the paragraph mean?

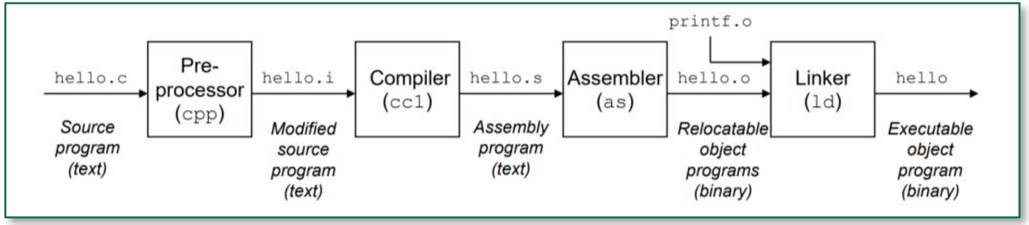


```
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
```

UAB THE UNIVERSITY OF ALABAMA AT BIRMINGHAM.

Compile C to Assembly

C to Assembly



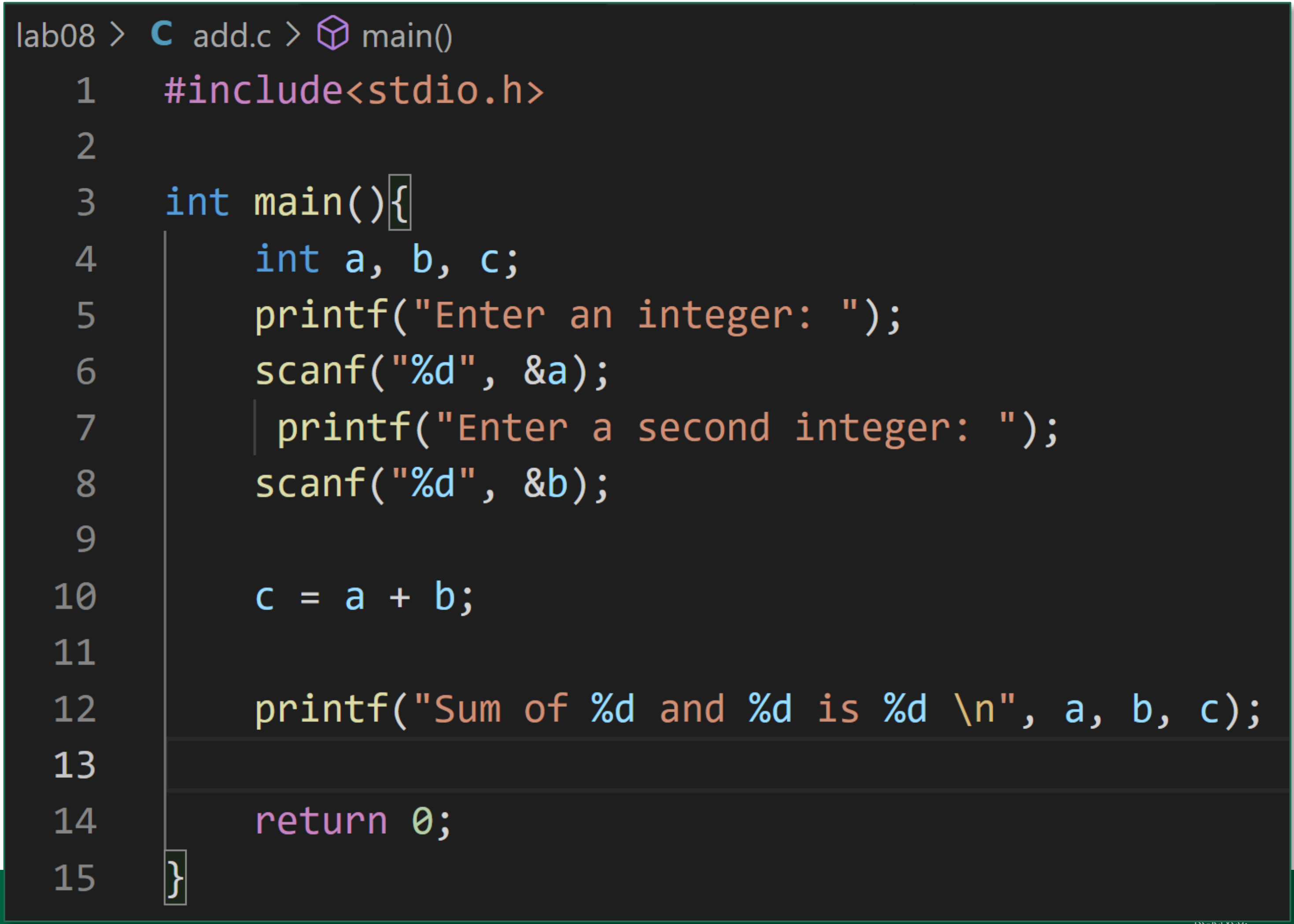
- Using **-S** will run the compiler, generating an assembly file *add.s,* but will NOT call the assembler, and so will not produce an executable file.
- Instead of our usual gcc command, run this instead:

```
gcc add.c –S
```

Or better (turn off optimization): `gcc add.c –S –O0`

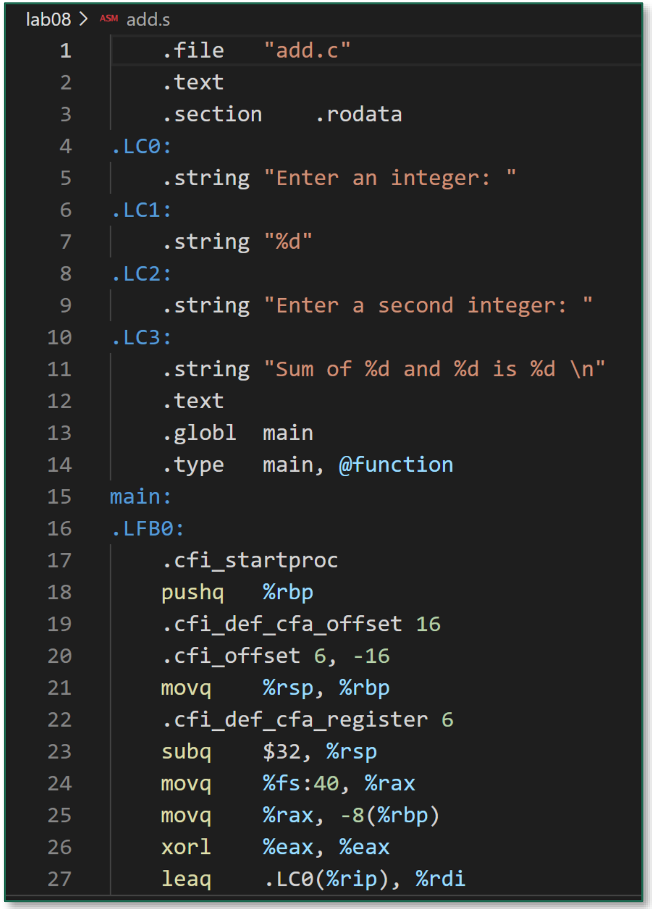
- If you have done this correctly, you should now see a new file, *add.s*

add.c

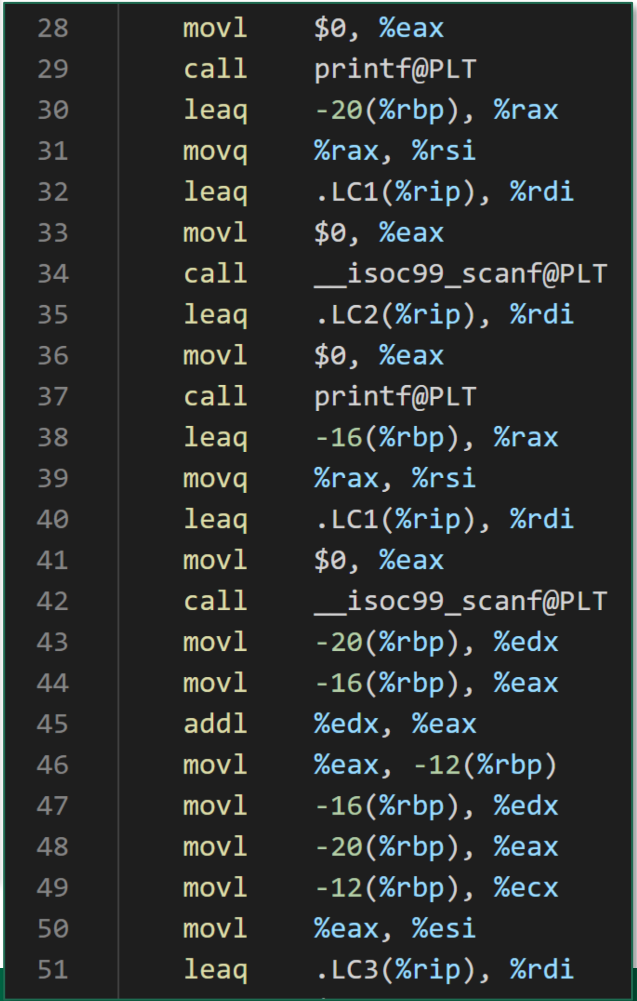


```c
#include<stdio.h>

int main(){
    int a, b, c;
    printf("Enter an integer: ");
    scanf("%d", &a);
     printf("Enter a second integer: ");
    scanf("%d", &b);

    c = a + b;

    printf("Sum of %d and %d is %d \n", a, b, c);

    return 0;
}
```

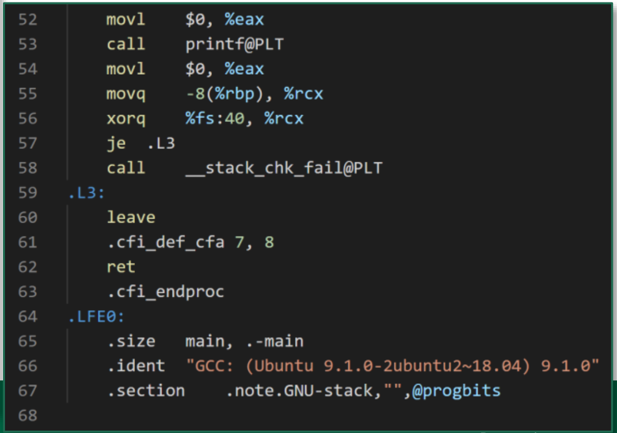
add.s



```
        .file   "add.c"
        .text
        .section        .rodata
.LC0:
        .string "Enter an integer: "
.LC1:
        .string "%d"
.LC2:
        .string "Enter a second integer: "
.LC3:
        .string "Sum of %d and %d is %d \n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        leaq    .LC0(%rip), %rdi
```



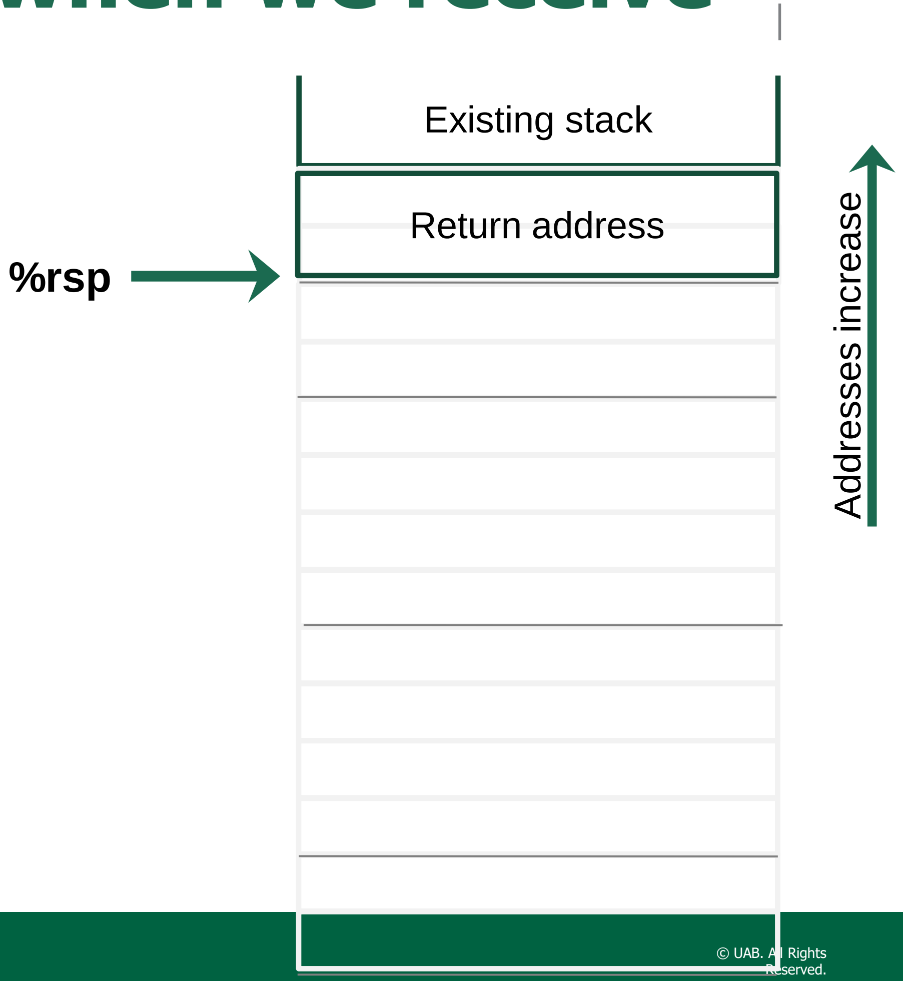
```
        movl    $0, %eax
        call    printf@PLT
        leaq    -20(%rbp), %rax
        movq    %rax, %rsi
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    __isoc99_scanf@PLT
        leaq    .LC2(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        leaq    -16(%rbp), %rax
        movq    %rax, %rsi
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    __isoc99_scanf@PLT
        movl    -20(%rbp), %edx
        movl    -16(%rbp), %eax
        addl    %edx, %eax
        movl    %eax, -12(%rbp)
        movl    -16(%rbp), %edx
        movl    -20(%rbp), %eax
        movl    -12(%rbp), %ecx
        movl    %eax, %esi
        leaq    .LC3(%rip), %rdi
```



```
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        movq    -8(%rbp), %rcx
        xorq    %fs:40, %rcx
        je  .L3
        call    __stack_chk_fail@PLT
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.1.0-2ubuntu2~18.04) 9.1.0"
        .section        .note.GNU-stack,"",@progbits

```

State of the machine when we receive it

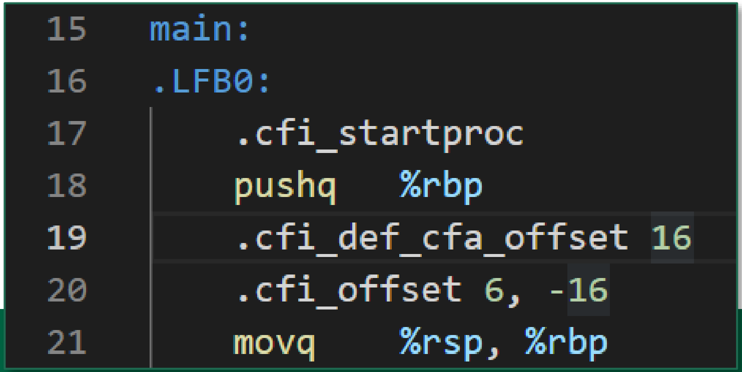
Registers	
%rax	
%rcx	
%rdx	
%rsi	
%rdi	
%rsp	
%rbp	



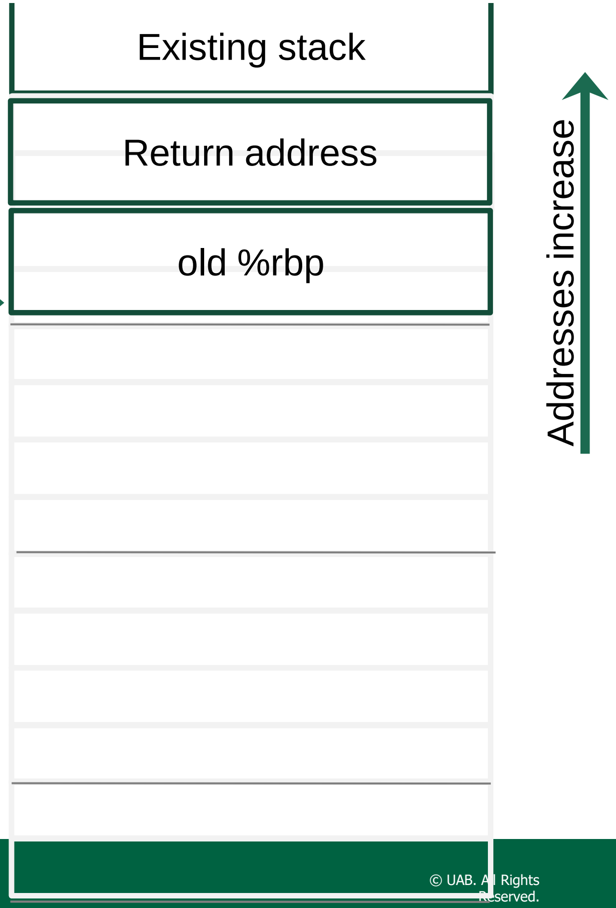
Preamble

Registers	
%rax	
%rcx	
%rdx	
%rsi	
%rdi	
%rsp	
%rbp	%rsp

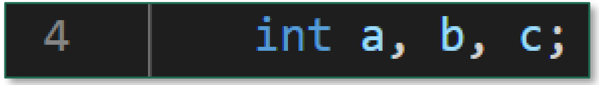
%rsp



```
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
```

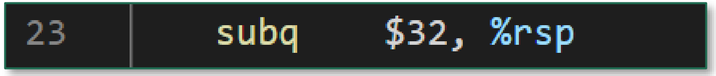


Prep for local variables

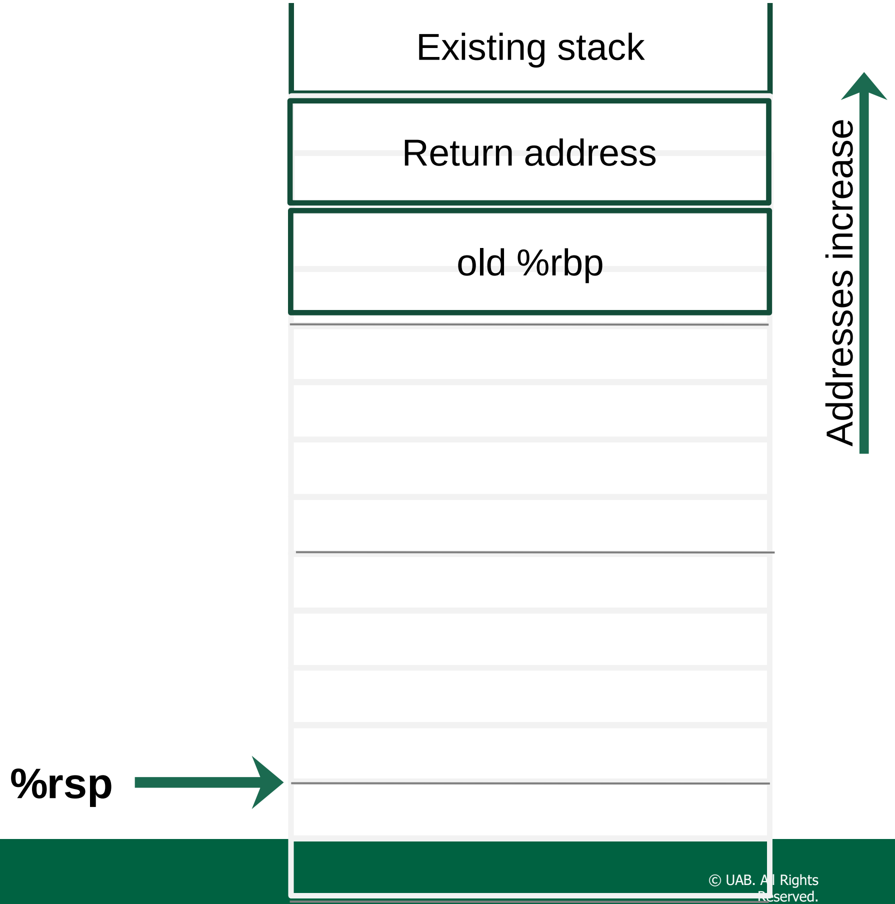


```
int a, b, c;
```

Registers	
%rax	
%rcx	
%rdx	
%rsi	
%rdi	
%rsp	
%rbp	%rsp



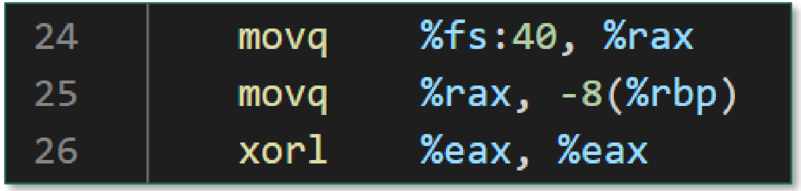
```
subq     $32, %rsp
```



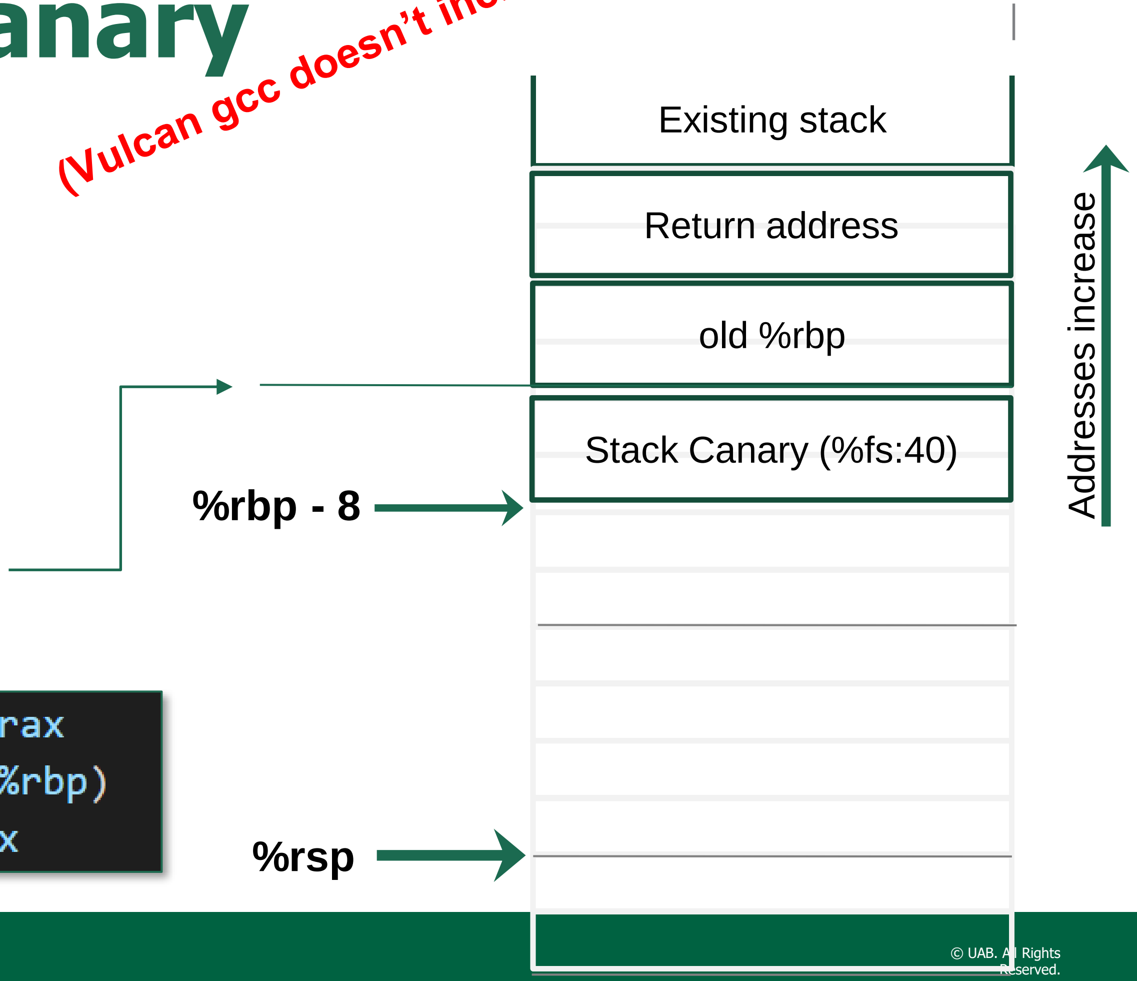
Setup Stack Canary

(Vulcan gcc doesn't include)

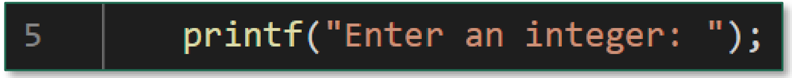
Registers	
%rax	~~%fs:40~~ 0
%rcx	
%rdx	
%rsi	
%rdi	
%rsp	
%rbp	%rsp



```
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
```

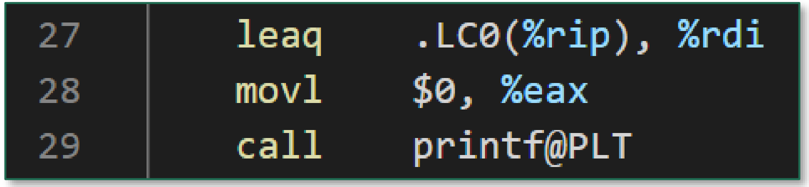


Let’s print

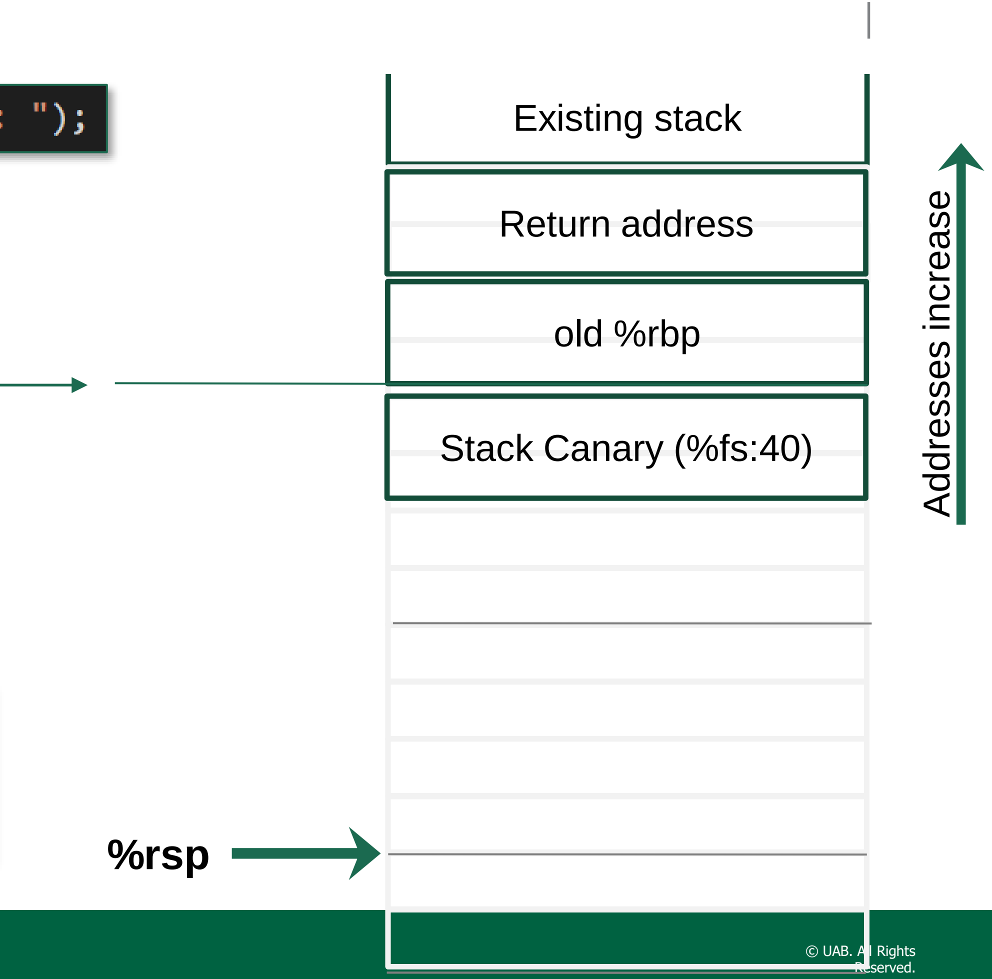


```c
    printf("Enter an integer: ");
```

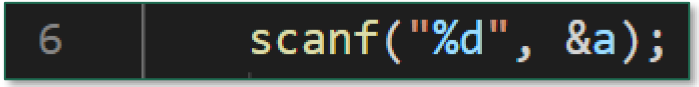
Registers	
%rax	~~0~~ 18
%rcx	
%rdx	
%rsi	
%rdi	Pointer to .LCO “Enter an integer: “**(maybe)**
%rsp	
%rbp	%rsp



```
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
```

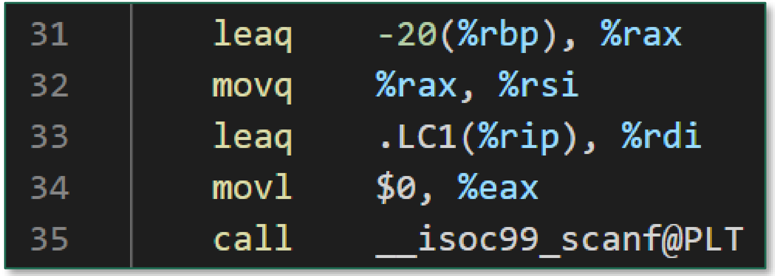


Let’s get some user input, int a aka:

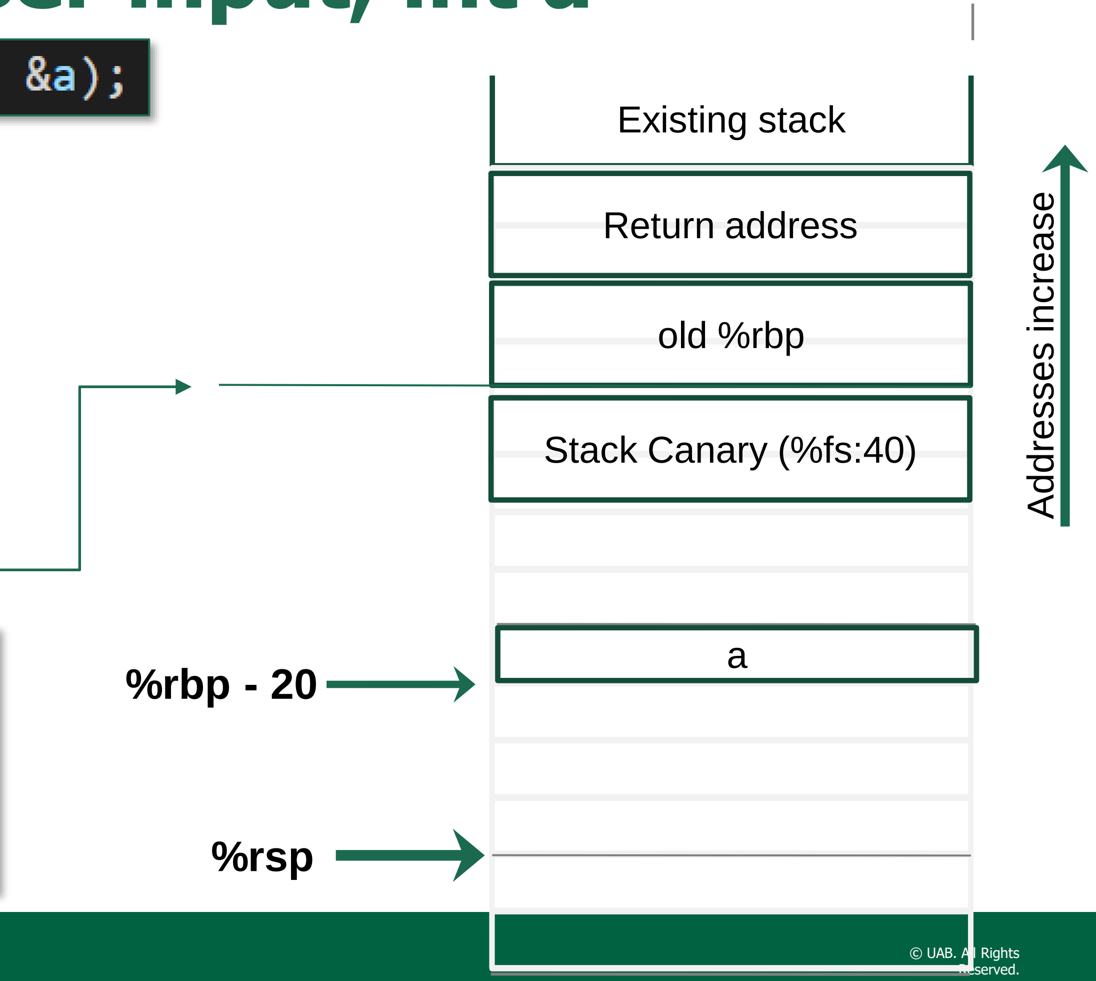


```
scanf("%d", &a);
```

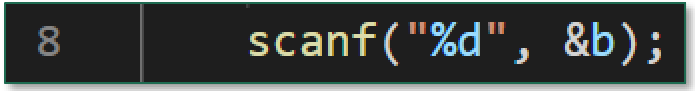
Registers	
%rax	~~Pointer to %rbp - 20~~ ~~0~~ 1
%rcx	
%rdx	
%rsi	Pointer to %rbp - 20
%rdi	Pointer to .LC1 “%d“
%rsp	
%rbp	%rsp



```
        leaq    -20(%rbp), %rax
        movq    %rax, %rsi
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    __isoc99_scanf@PLT
```

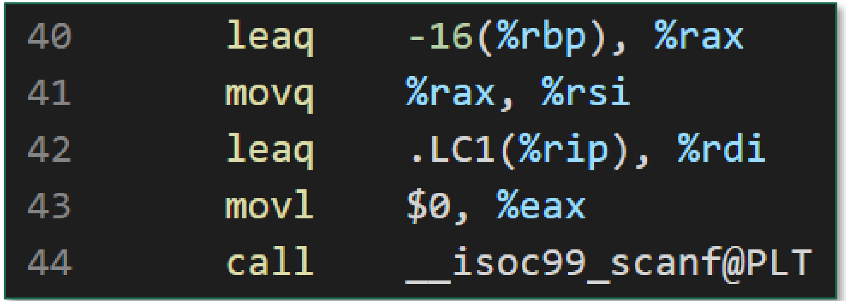


Let’s get some user input, int b aka:

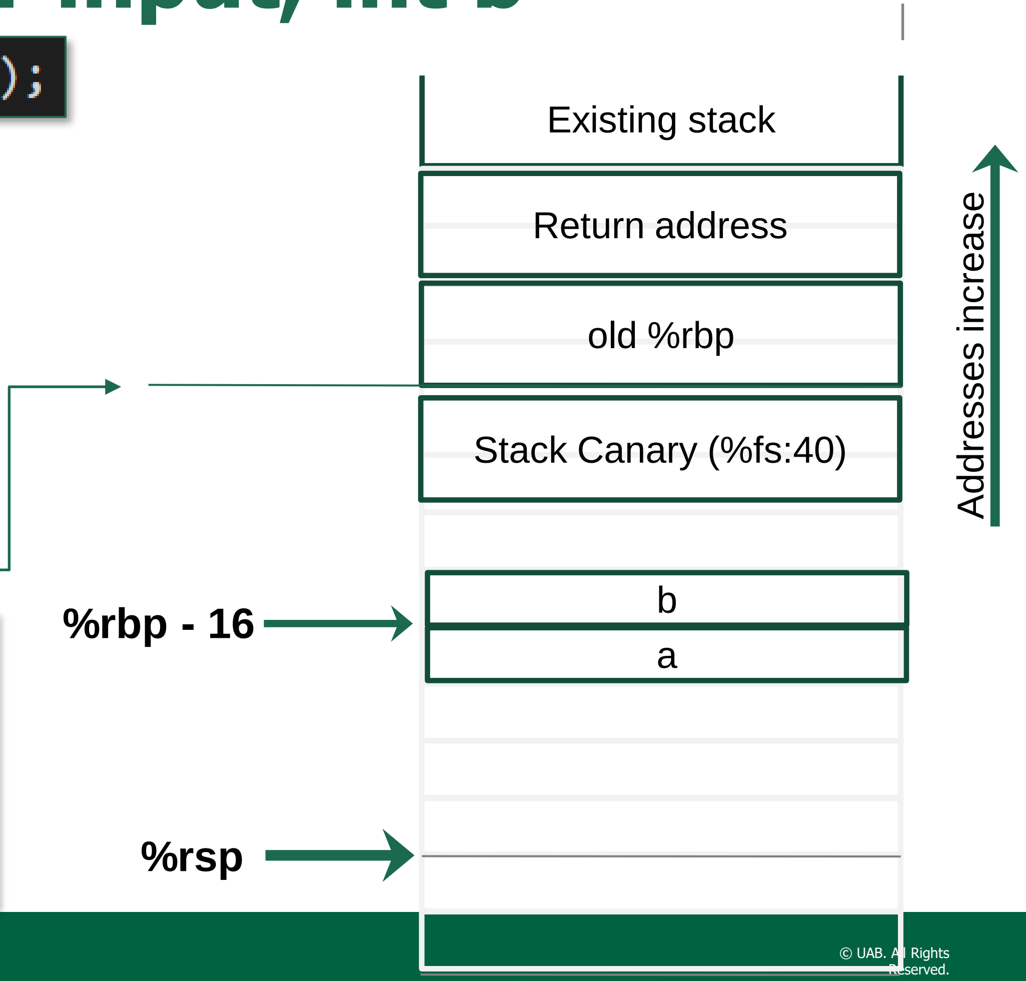


```
    scanf("%d", &b);
```

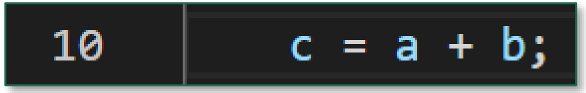
Registers	
%rax	~~Pointer to %rbp - 16~~ ~~0~~ 1
%rcx	
%rdx	
%rsi	Pointer to %rbp - 16
%rdi	Pointer to .LC1 “%d“
%rsp	
%rbp	%rsp



```
        leaq    -16(%rbp), %rax
        movq    %rax, %rsi
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    __isoc99_scanf@PLT
```

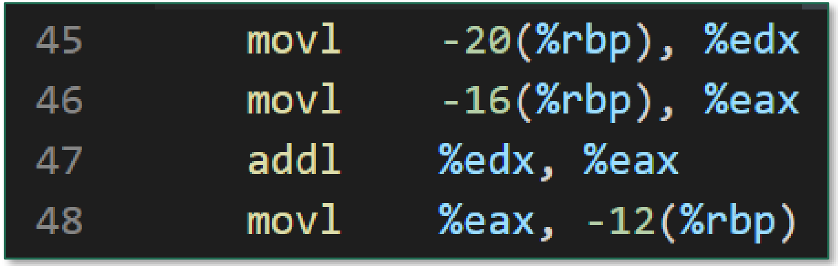


Let's add

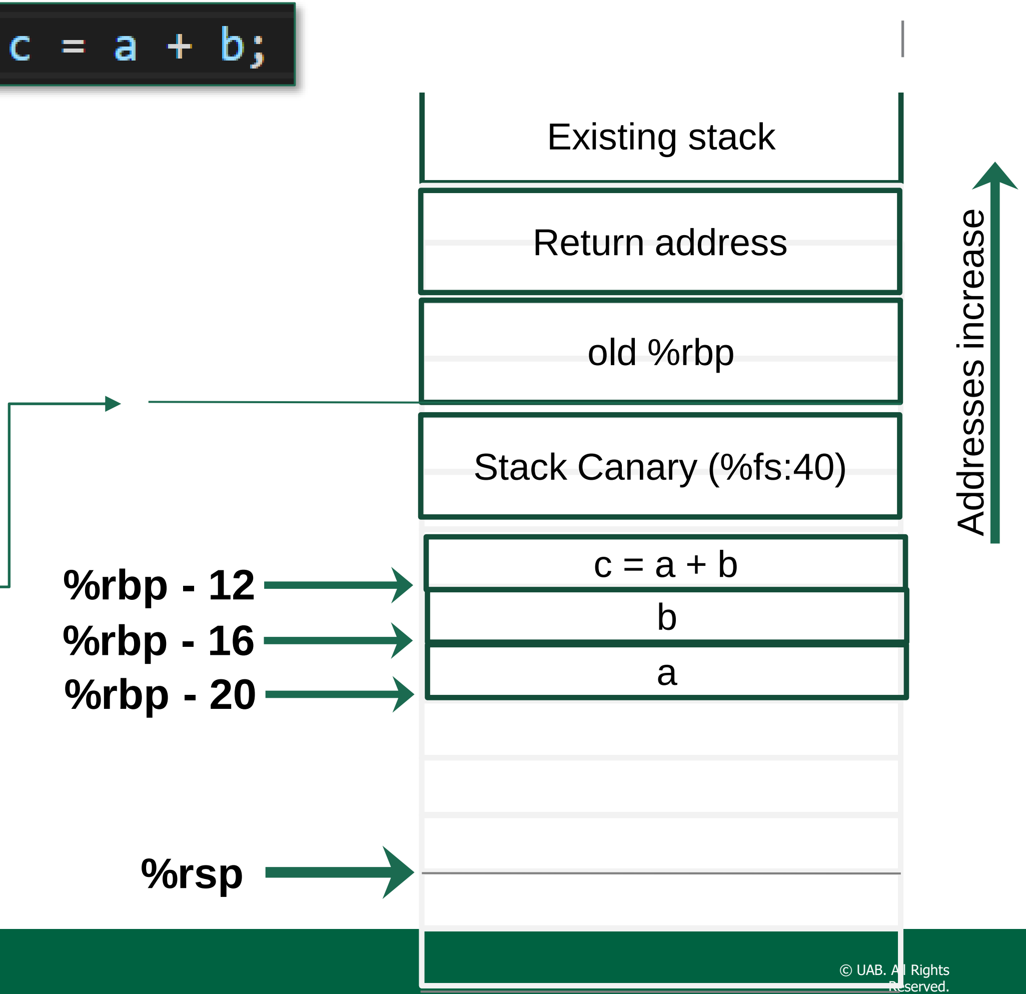


```
c = a + b;
```

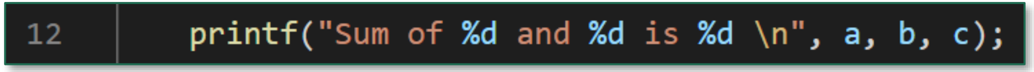
Registers	
%rax	~~b~~ a + b
%rcx	
%rdx	a
%rsi	
%rdi	
%rsp	
%rbp	%rsp



```
        movl    -20(%rbp), %edx
        movl    -16(%rbp), %eax
        addl    %edx, %eax
        movl    %eax, -12(%rbp)
```

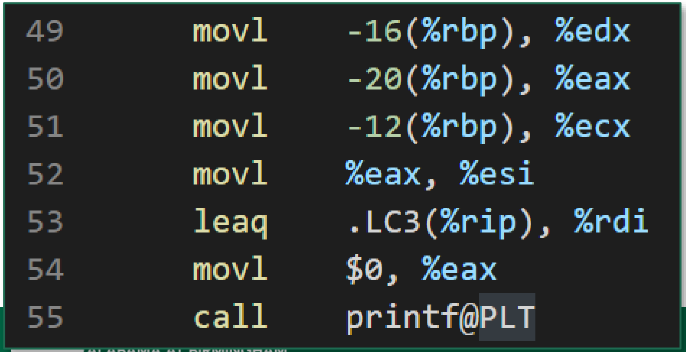


Print

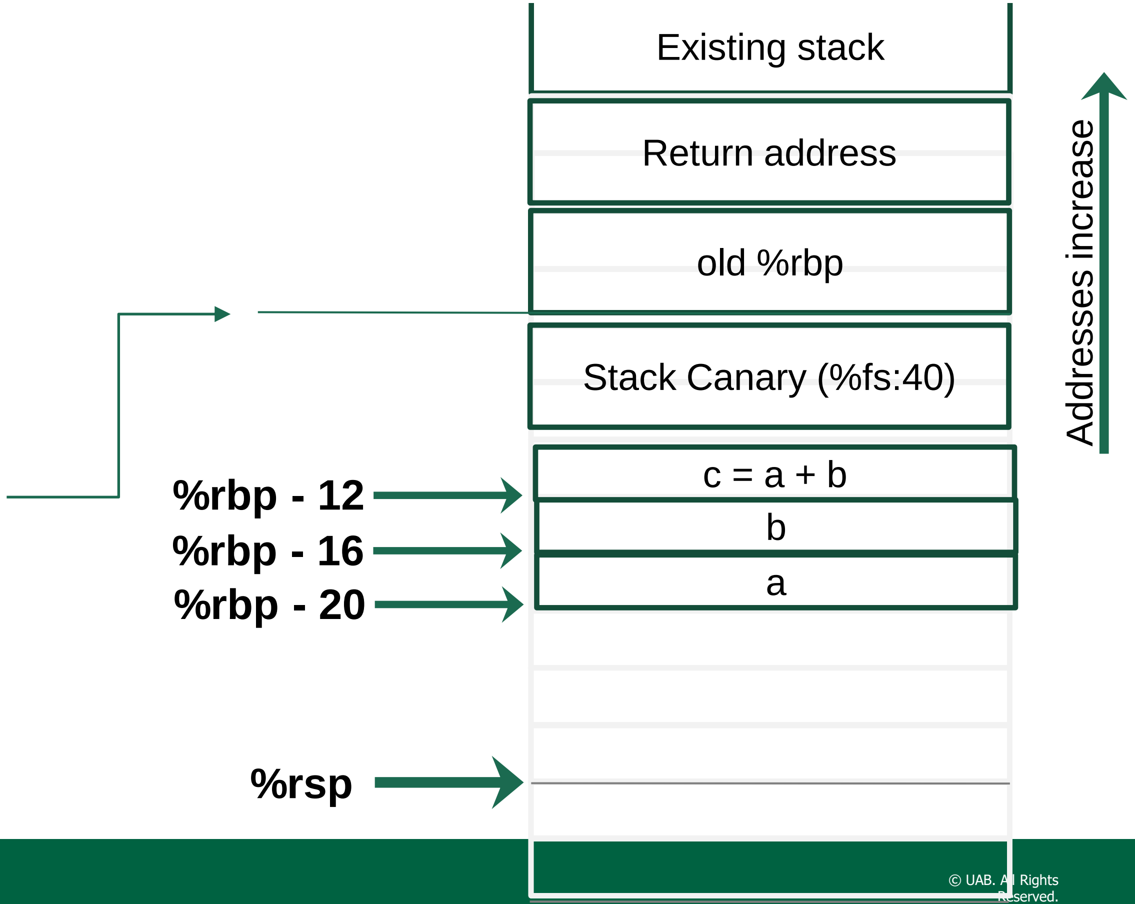


```c
    printf("Sum of %d and %d is %d \n", a, b, c);
```

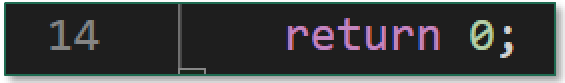
Registers	
%rax	~~a~~ ~~0~~ 20
%rcx	c
%rdx	b
%rsi	a
%rdi	Pointer to .LC3 “Sum of %d and ….”
%rsp	
%rbp	%rsp



```
        movl    -16(%rbp), %edx
        movl    -20(%rbp), %eax
        movl    -12(%rbp), %ecx
        movl    %eax, %esi
        leaq    .LC3(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
```

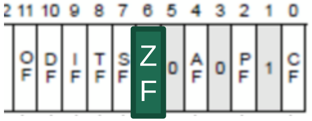


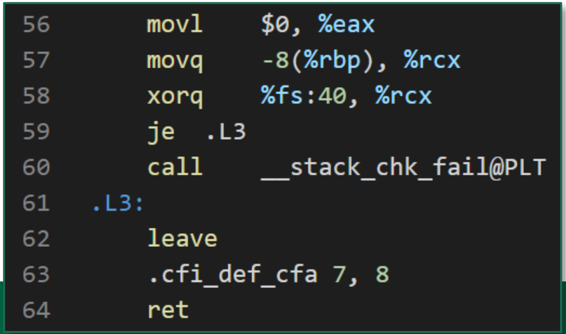
Clean-up and return



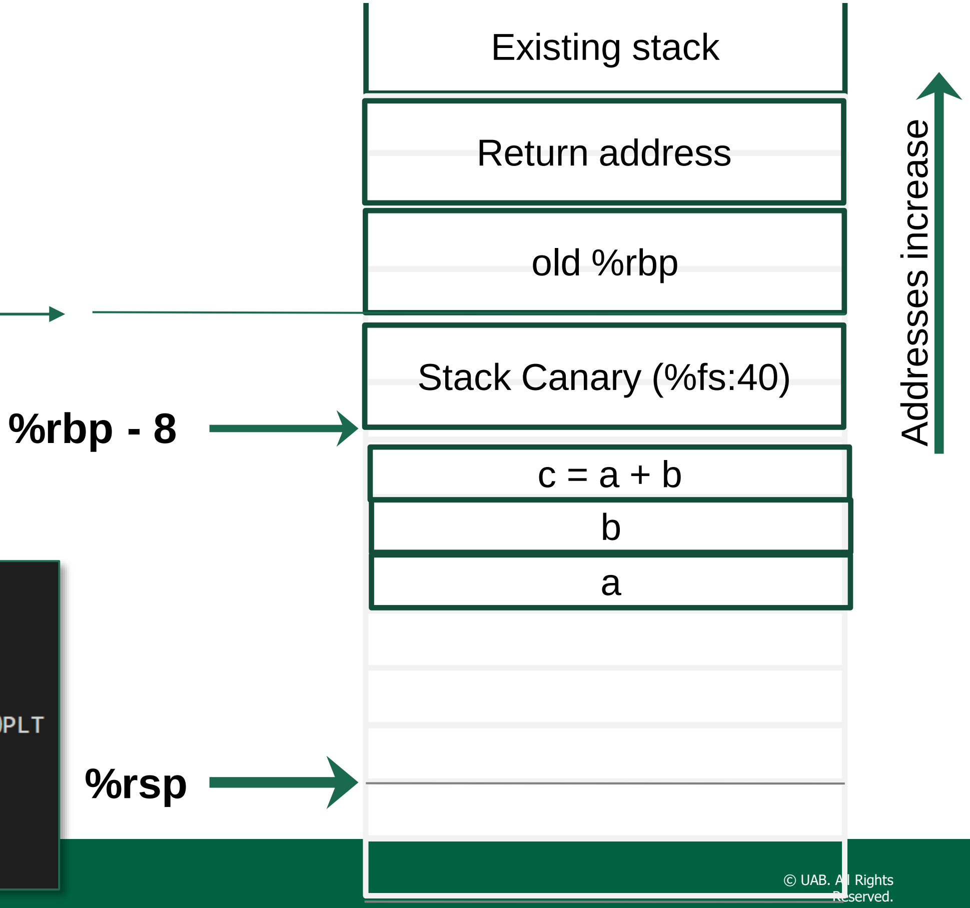
```
return 0;
```

Registers	
%rax	0
%rcx	~~Stack Canary (%fs:40)~~ 0
%rdx	
%rsi	
%rdi	
%rsp	
%rbp	%rsp

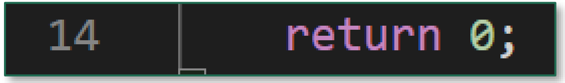




```
        movl    $0, %eax
        movq    -8(%rbp), %rcx
        xorq    %fs:40, %rcx
        je  .L3
        call    __stack_chk_fail@PLT
    .L3:
        leave
        .cfi_def_cfa 7, 8
        ret
```

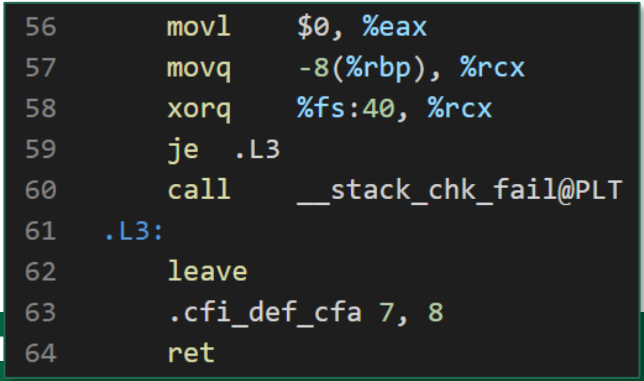


Clean-up and return

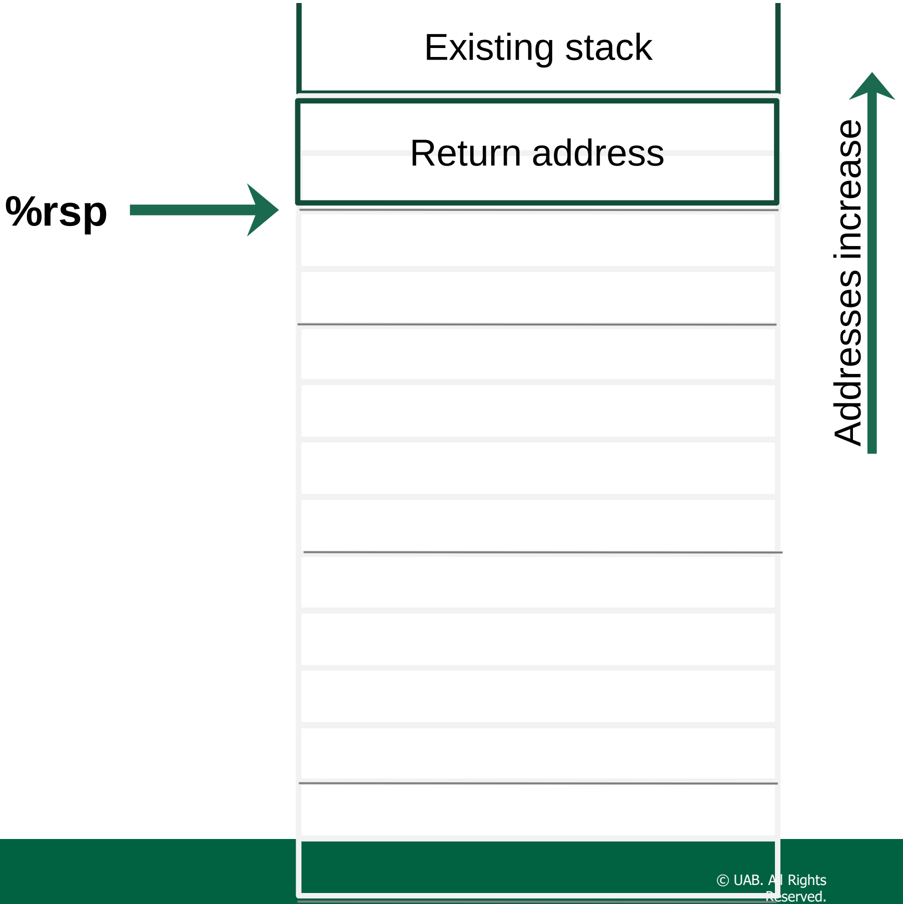


```
        return 0;
```

Registers	
%rax	0
%rcx	0
%rdx	
%rsi	
%rdi	
%rsp	
%rbp	Old %rbp



```
        movl    $0, %eax
        movq    -8(%rbp), %rcx
        xorq    %fs:40, %rcx
        je  .L3
        call    __stack_chk_fail@PLT
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
```


Existing stack
Return address
%rsp
Addresses increase