(Lab 11) Basics of Assembly Programming and Bomb Lab

CS2013 Systems Programming

Department of CSE, IIT Palakkad

IIT Palakkad

Oct 23, 2025

Quiz 10 (15 minutes, **Do not copy the question**)

1. **The command `git add *` always adds files to a repo correctly.** Say `Agree/Disagree`. Give a short reason.
2. Consider the makefile on the left side

Question 2

```
main.o: main.c
    gcc -Wall main.c -o main.o

main: main.o
    gcc main.o -o main
```

Question 3

```
int  n    = 42;
int  a[] = {n};
int *p    = a;
```

Fix errors (if any) in the makefile so that the executable `main` is produced correctly. If the makefile looks OK, say `No errors`.

3. Consider the C code on right side. For each expression (1), (2) and (3), say true or false along with a short explanation.
   (1) `*p == n`, (2) `*p == a[1]` , (3) `&n == &a[0]`

Plan (Demo)

- A curious program
- Assembly Basics
- Common Instructions
- Arithmetic Instructions
- Conditionals and Loops
- Activation Stack and Functions
- Putting it together: An explanation
- Lab exercise

A curious program

- Demo
- High level abstraction hides details
- Why assembly ? Necessity to handwritten assembly
  - Too less memory
  - Critical parts (Context switching in OS, Networking)
  - Necessary to detect vulnerability in code
- Aim: Explain the anomaly !
- First: Understand assembly !

Assembly basics

- Learn Intel x86-64 instructions - syntax and usage
- Demo (`objdump`)
- `%rsp` - points to stack top
- `%rax` - stores return value
- `%rip` - stores **instruction pointer** (no direct modification)

Registers (16)

- `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rdi`, `%rsi`
- `%rsp`, `%rbp`, `%r8` to `%r15`

Instruction Structure

- `opcode operands`. Example: `add $0x02, %rax`

Address	Value
0x804	0xCA
0x808	0xFD
0x80c	0x12
0x810	0x1E

Register	Value
%rax	0x804
%rbx	0x10
%rcx	0x4
%rdx	0x1

Operand	Form	Translation	Value
%rcx	Register	%rcx	0x4
(%rax)	Memory	M[%rax] or M[0x804]	0xCA
$0x808	Constant	0x808	0x808
(0x808)	Memory	M[0x808]	0xFD

Instruction Structure (More examples)

Address	Value
0x804	0xCA
0x808	0xFD
0x80c	0x12
0x810	0x1E

Register	Value
%rax	0x804
%rbx	0x10
%rcx	0x4
%rdx	0x1

Operand	Form	Translation	Value
0x8(%rax)	Memory	M[%rax + 8]	M[0x80c] 0x12
(%rax, %rcx)	Memory	M[%rax + %rcx]	M[0x808] 0xFD
0x4(%rax, %rcx)	Memory	M[%rax + %rcx + 4]	M[0x80c] 0x12
0x800(,%rdx,4)	Memory	M[0x800 + %rdx × 4]	M[0x804] 0xCA
(%rax, %rdx, 8)	Memory	M[%rax + %rdx × 8]	M[0x80c] 0x12

Instruction Structure (Restrictions)

- `opcode S, D` - source to destination (S $\rightarrow$ D)
- Constants cannot be destination. `addr %rax, $0x10` $\times$
- Memory cannot be source and destination

  `mov (%rax), (%rbx)` $\times$
- Scaling allowed 1, 2, 4 and 8 (memory access must be aligned)

Instruction Suffix

Suffix	C Type	Size (bytes)
b	char	1
w	short	2
l	int or unsigned	4
s	float	4
q	long, unsigned long, all pointers	8
d	double	8

Example

- `movs %eax, (%rsi)`
- treats data as 4 bytes (32 bits), move to M[`%rsi`].

What is %eax ?

Register memory access mechanism

- %rax - 64 bit register
- %eax - (lower) 32 bits
- %ax - (lower) 16 bits
- %al - (lower) 8 bits

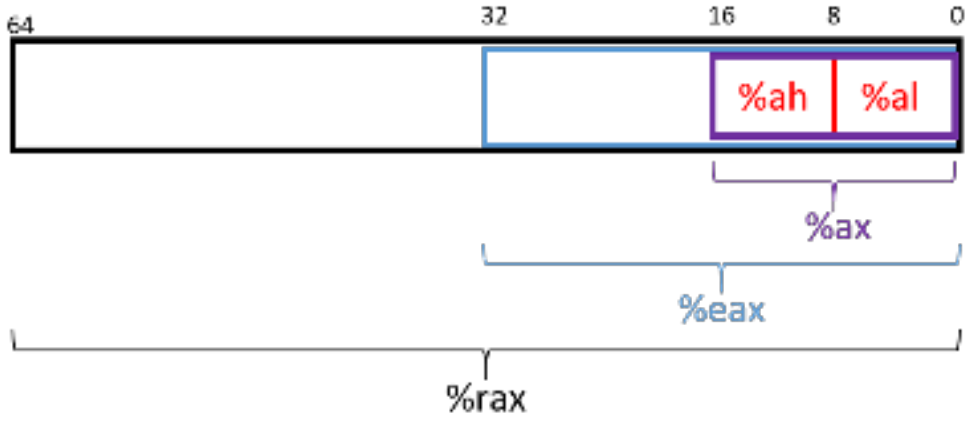


Figure 1: Register access

Common Instructions

Instruction	Translation	Meaning
`mov S, D`	S $\rightarrow$ D	copies value of S to D
`add S, D`	S + D $\rightarrow$ D	adds S to D, stores result in D
`sub S, D`	D $-$ S $\rightarrow$ D	subtracts S from D, stores result in D
`imul S, D`	S $\times$ D $\rightarrow$ D	multiply S and D, stores result in D
`idiv S`	%rax / S	quotient $\rightarrow$ %rax, remainder $\rightarrow$ %rdx

Example

- `mov -0x4(%rbp),%eax` $\rightarrow$?
- . . . Value of %rbp + -0x4 in memory (M[%rbp - 0x4]) to register %eax
- `add $0x2,%eax` $\rightarrow$ Add 0x2 to %eax, store in %eax

Stack (Recall)

- Pay attention to memory address !

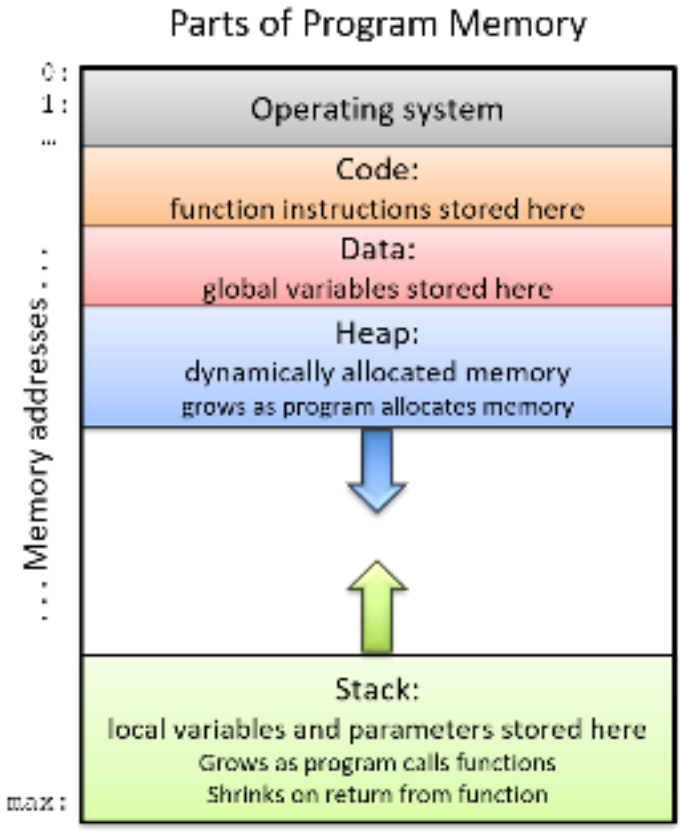


Figure 2: Program Address space

What does adder2() do in assembly

```
0x526   push  %rbp
0x527   mov   %rsp, %rbp
0x52a   mov   %edi, -0x4(%rbp)
0x52d   mov   -0x4(%rbp), %eax
0x530   add   $0x2, %eax
0x533   pop   %rbp
0x534   retq
```

Registers	
%eax	0x123
%edi	0x28
%rsp	0xd28
%rbp	0xd40
%rip	0x526

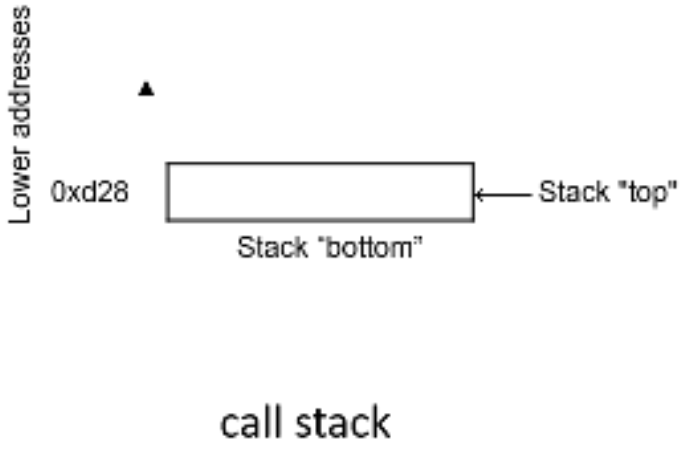


Figure 3: Prior to execution

What does `adder2()` do in assembly

```
0x526   push  %rbp
0x527   mov   %rsp, %rbp
0x52a   mov   %edi, -0x4(%rbp)
0x52d   mov   -0x4(%rbp), %eax
0x530   add   $0x2, %eax
0x533   pop   %rbp
0x534   retq
```

Registers	
%eax	0x123
%edi	0x28
%rsp	**0xd20**
%rbp	0xd40
%rip	**0x527**

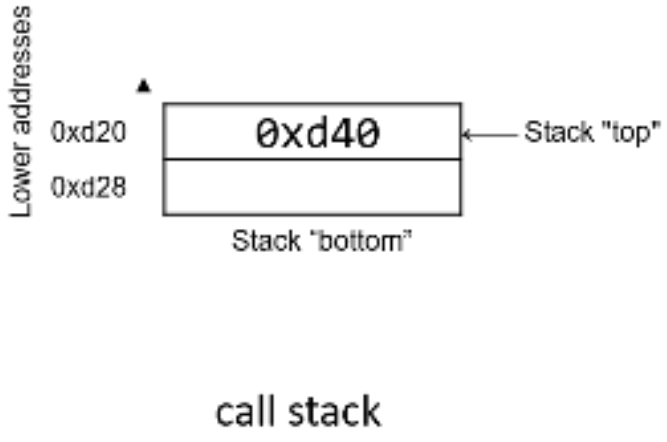


Figure 4: First instruction execution

What does `adder2()` do in assembly

```
0x526   push  %rbp
0x527   mov   %rsp, %rbp
0x52a   mov   %edi, -0x4(%rbp)
0x52d   mov   -0x4(%rbp), %eax
0x530   add   $0x2, %eax
0x533   pop   %rbp
0x534   retq
```

Registers	
%eax	0x123
%edi	0x28
%rsp	0xd20
%rbp	**0xd20**
%rip	**0x52a**

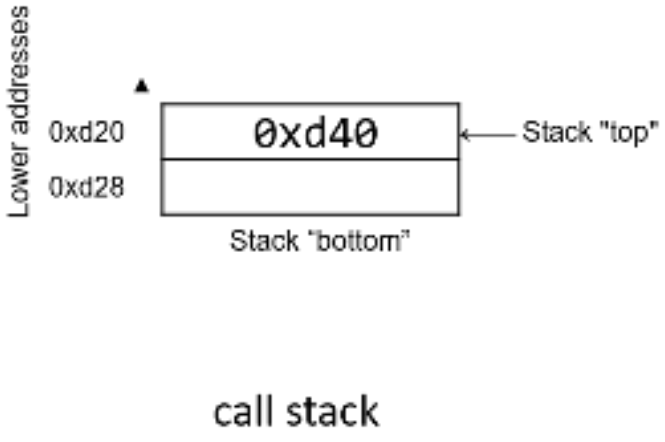


Figure 5: Second instruction execution

What does `adder2()` do in assembly

```
0x526   push  %rbp
0x527   mov   %rsp, %rbp
0x52a   mov   %edi, -0x4(%rbp)
0x52d   mov   -0x4(%rbp), %eax
0x530   add   $0x2, %eax
0x533   pop   %rbp
0x534   retq
```

Registers	
%eax	0x123
%edi	0x28
%rsp	0xd20
%rbp	0xd20
%rip	**0x52d**

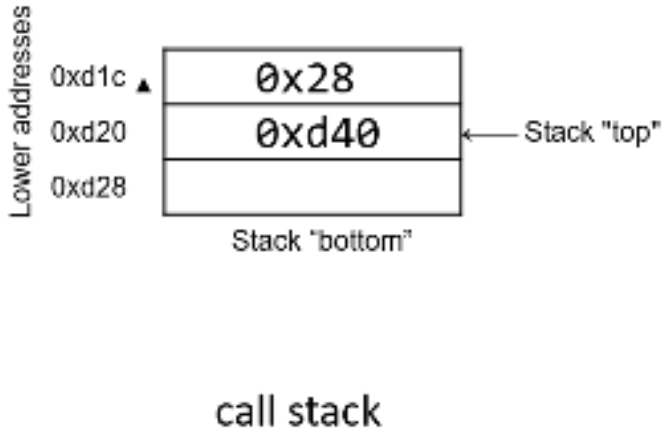


Figure 6: Third instruction execution

What does adder2() do in assembly

```
0x526   push   %rbp
0x527   mov    %rsp, %rbp
0x52a   mov    %edi, -0x4(%rbp)
0x52d   mov    -0x4(%rbp), %eax
0x530   add    $0x2, %eax
0x533   pop    %rbp
0x534   retq
```

Registers	
%eax	**0x28**
%edi	0x28
%rsp	0xd20
%rbp	0xd20
%rip	**0x530**

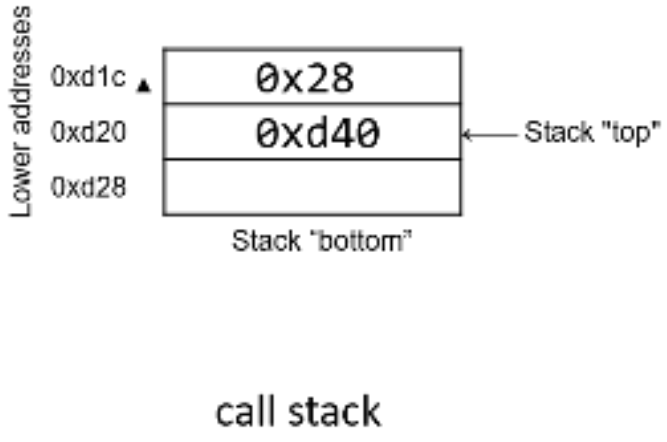


Figure 7: Fourth instruction execution

What does adder2() do in assembly

```
0x526   push  %rbp
0x527   mov   %rsp, %rbp
0x52a   mov   %edi, -0x4(%rbp)
0x52d   mov   -0x4(%rbp), %eax
0x530   add   $0x2, %eax
0x533   pop   %rbp
0x534   retq
```

Registers	
%eax	**0x2A**
%edi	0x28
%rsp	0xd20
%rbp	0xd20
%rip	**0x533**

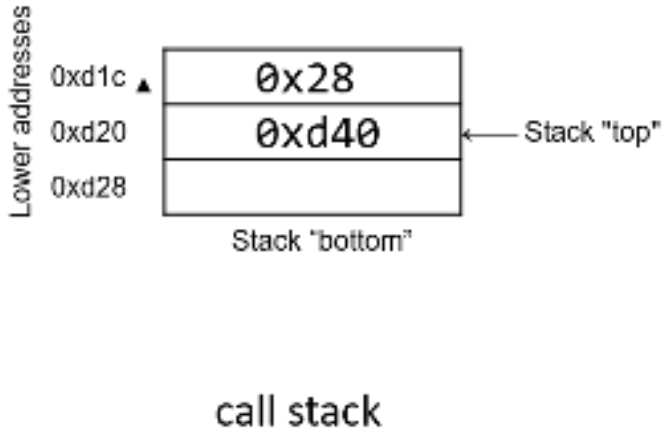


Figure 8: Fifth instruction execution

What does adder2() do in assembly

```
0x526   push  %rbp
0x527   mov   %rsp, %rbp
0x52a   mov   %edi, -0x4(%rbp)
0x52d   mov   -0x4(%rbp), %eax
0x530   add   $0x2, %eax
0x533   pop   %rbp
0x534   retq
```

Registers	
%eax	0x2A
%edi	0x28
%rsp	**0xd28**
%rbp	**0xd40**
%rip	**0x534**

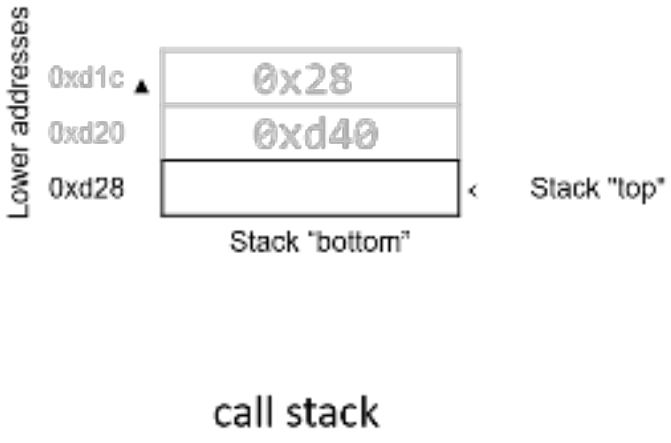


Figure 9: Sixth instruction execution

Load Effective Address (LEA)

- Compute **address** of a loc. `lea S, D`, effect : S $\rightarrow$ D

Register	Value
%rax	0x5
%rcx	0x808
%rdx	0x4

Instruction	Translation
`lea 8(%rax), %rax`	8 + %rax $\rightarrow$ %rax (0xd = 13)
`lea (%rax, %rdx), %rax`	%rax + %rdx $\rightarrow$ %rax (0x9 = 9)
`lea (,%rax,4), %rax`	%rax $\times$ 4 $\rightarrow$ %rax (0x14 = 20)
`lea -0x8(%rcx), %rax`	%rcx $-$ 8 $\rightarrow$ %rax (0x800)
`lea -0x4(%rcx, %rdx, 2),%rax`	%rcx + %rdx $\times$ 2 $-$ 4 $\rightarrow$ %rax (0x80c)

Plan (Progress so far)

- A curious program ✓
- Assembly Basics ✓
- Common Instructions ✓
- Arithmetic Instructions ✓
- Conditionals and Loops
- Activation Stack and Functions
- Putting it together: An explanation
- Lab exercise

Conditionals (Instructions and Flags)

Instruction	Translation
`cmp R1, R2`	Evaluates R2 - R1 and sets **conditional code flags**
`test R1, R2`	Computes R1 & R2 and sets **conditional code flags**

Conditional code flags

Flag	Meaning	Value
ZF	Is zero ?	1: yes; 0: no
SF	Is negative ?	1: yes; 0: no
OF	Overflow has occurred ?	1: yes; 0: no
CF	Arithmetic carry has occurred ?	1: yes; 0: no

Compare and Test

- `cmp R1, R2` does the following
  - ZF flag set to 1 if R1 = R2
  - SF flag set to 1 if R2 < R1 (R2 - R1 results in a negative value)
  - OF flag set to 1 if R2 - R1 results in an integer overflow
    - (useful for signed comparisons)
  - CF flag set to 1 if R2 - R1 results in a carry operation
    - (useful for unsigned comparisons)
- Example: Say, %rax = 0x2 and %rbx = 0x1
  - `cmp %rax, %rbx` sets ZF to ? 0,
  - sets SF to ? 1,
  - sets OF to ? 0,
  - sets CF to ? 0

Unconditional and Conditional Jump Instructions

Direct Jump

Instruction	Description
`jmp L`	Jump to location specified by L
`jmp *addr`	Jump to specified address

Conditional Jump

Letter	Word	Comparison
`j`	jump	
`n`	not	
`e`	equal	
`s`	signed	
`g`	greater	signed
`l`	less	signed
`a`	above	unsigned
`b`	below	unsigned

Conditional Jump Instructions (examples)

- `jge` - ? jump if greater than equal to (signed comparison)
- `jb` - ? jump if less than (unsigned comparison)
- `jae` - ? jump if greater than equal to (unsigned comparison)
- `jl` is same as `jnge`
- `js` - ? jump if signed (that is, negative)
- Trick : try spelling what each letter means

Conditional Jump

- Original code

```
int getSmallest(int x, int y) {
    int smallest;
    if (x <= y) {
        smallest = x;
    }
    else {
        smallest = y;
    }
    return smallest;
}
```

- Goto variant

```
int getSmallest(int x, int y)
    int smallest;
    if (x <= y) {
        goto assign_x;
    }
    smallest = y;
    goto done;

assign_x:
    smallest = x;

done:
    return smallest;
}
```

Conditional Jump

```
<+4>:  mov %edi,-0x14(%rbp)     # copy x to %rbp-0x14
<+7>:  mov %esi,-0x18(%rbp)     # copy y to %rbp-0x18
<+10>: mov -0x14(%rbp),%eax     # copy x to %eax
<+13>: cmp -0x18(%rbp),%eax     # compare x with y
<+16>: jle 0x4005b0 <get..+26> # if x<=y goto <get..+26>
<+18>: mov -0x18(%rbp),%eax     # copy y to %eax
<+24>: jmp 0x4005b9 <get..+35> # goto <get..+35>
<+26>: mov -0x14(%rbp),%eax     # copy x to %eax
<+35>: pop %rbp                 # restore %rbp
<+36>: ret                      # exit function
```

- Loops can be implemented in a similar manner

Functions (Activation Stack)

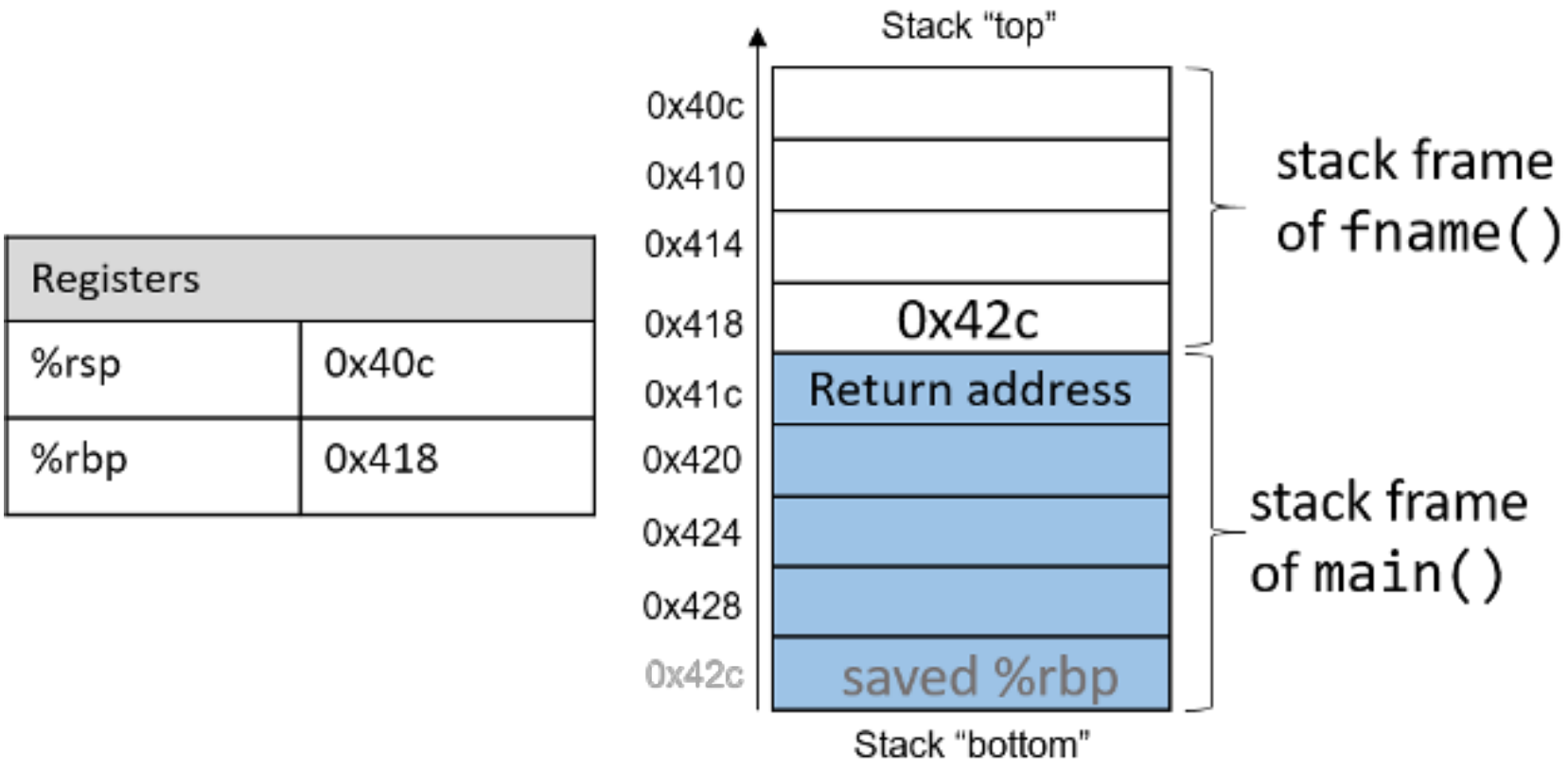


Figure 10: Stack frame

- `main()` is **caller**
- `fname()` is **callee**

Function argument locations

Parameter	Location
Parameter 1	`%rdi`
Parameter 2	`%rsi`
Parameter 3	`%rdx`
Parameter 4	`%rcx`
Parameter 5	`%r8`
Parameter 6	`%r9`
Parameter 7+	`on call stack`
Return value	`%rax`

Functions (Program's address space)

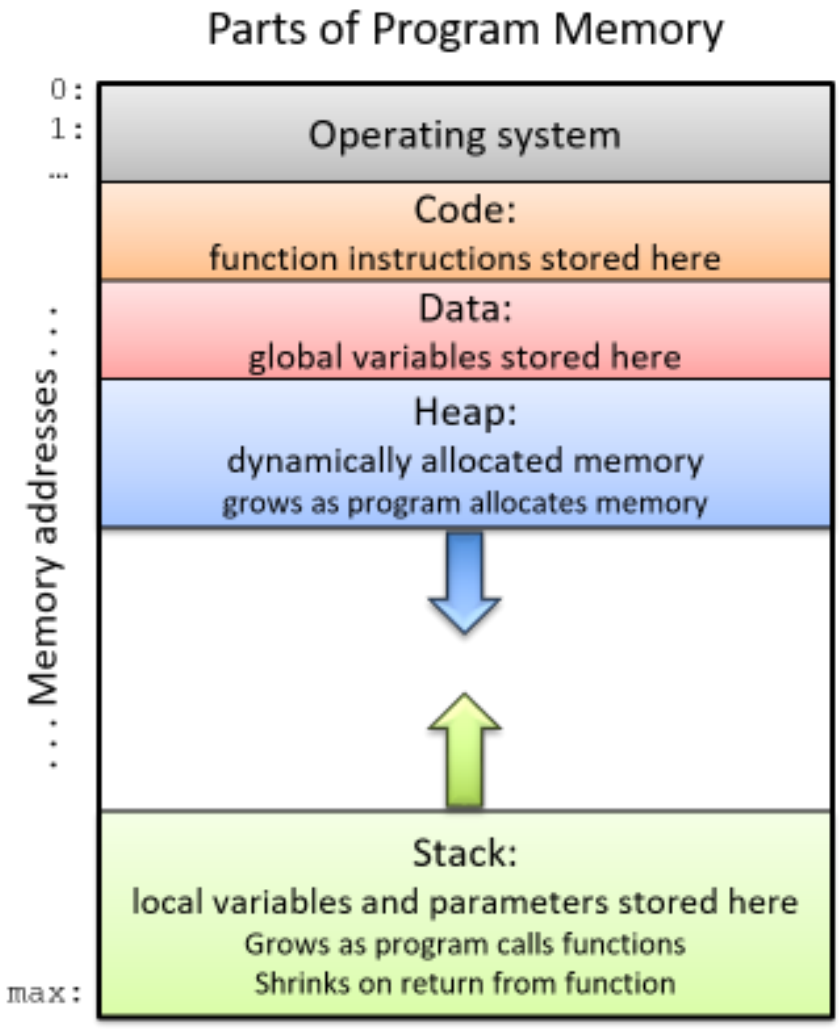


Figure 11: Program Address space

Functions (Additional instructions)

Instruction	Translation
`leaveq`	Prepares the stack for leaving a function.
`callq addr <fname>`	Switches active frame to callee function
`retq`	Restores active frame to caller function.

`leaveq`, `leave`

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

`callq`, `call`

```
push %rip
mov addr, %rip
```

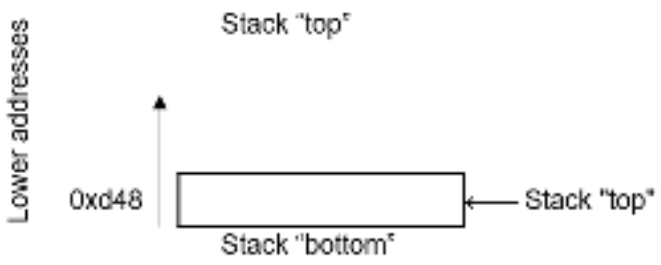
`retq`, `ret`

```
pop %rip
```

Understanding strange code (1)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	650
%edi	1
%rsp	0xd48
%rbp	0x830
%rip	0x542



Terminal:

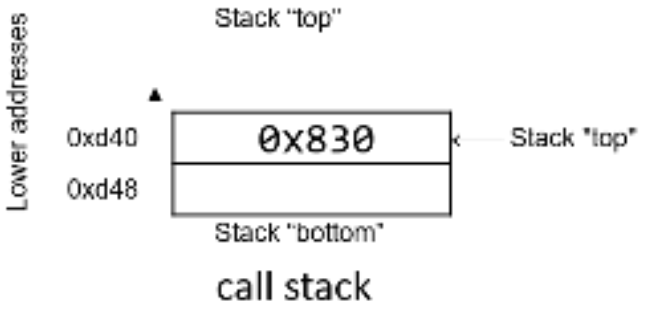
```
$ ./prog
```

Figure 12: State of execution

Understanding strange code (2)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	650
%edi	1
%rsp	**0xd40**
%rbp	0x830
%rip	**0x543**



Terminal:

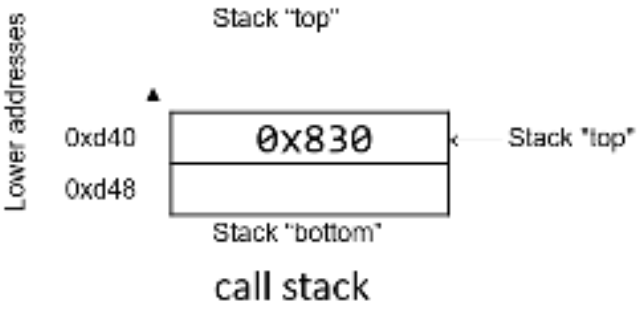
```
$ ./prog
```

Figure 13: State of execution

Understanding strange code (3)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	650
%edi	1
%rsp	0xd40
%rbp	**0xd40**
%rip	**0x546**



Terminal:

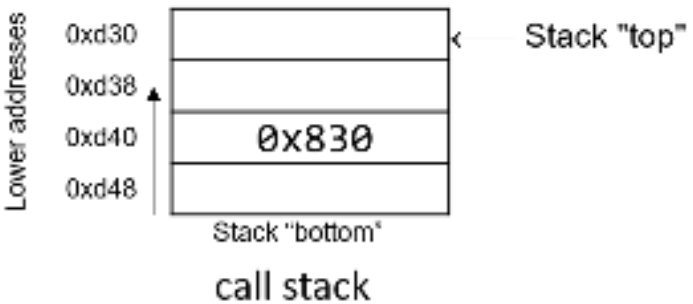
```
$ ./prog
```

Figure 14: State of execution

Understanding strange code (4)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	650
%edi	1
%rsp	**0xd30**
%rbp	0xd40
%rip	**0x54a**



Terminal:

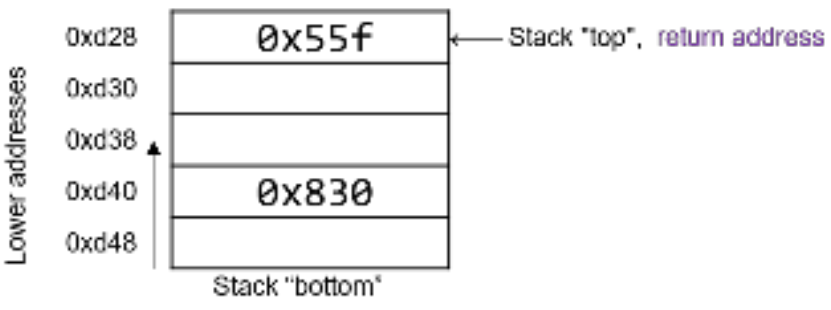
```
$ ./prog
```

Figure 15: State of execution

Understanding strange code (5)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x0
%edi	1
%rsp	**0xd28**
%rbp	0xd40
%rip	**0x526**



Terminal:

```
$ ./prog
```

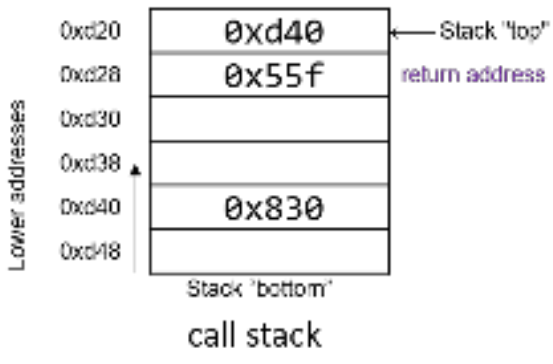
Equivalent to:
push %rip
mov 0x526, %rip

Figure 16: State of execution

Understanding strange code (6)

```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x0
%edi	1
%rsp	**0xd20**
%rbp	0xd40
%rip	**0x527**



Terminal:

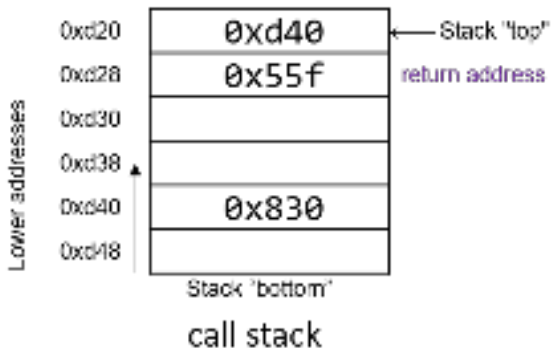
```
$ ./prog
```

Figure 17: State of execution

Understanding strange code (7)

```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	**0xd20**
%rip	**0x52a**



Terminal:

```
$ ./prog
```

Figure 18: State of execution

Understanding strange code (8)

```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	0x531

Terminal:

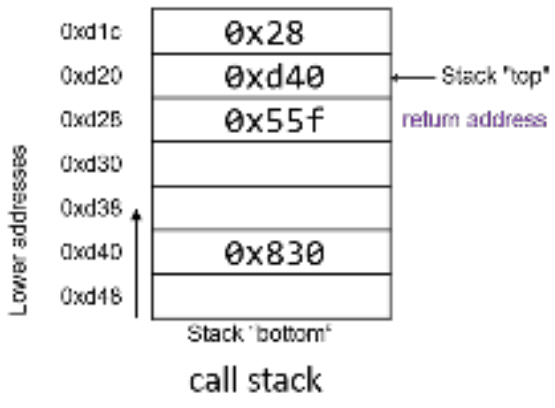
```
$ ./prog
```

Figure 19: State of execution

Understanding strange code (9)

```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x28
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	0x534



Terminal:

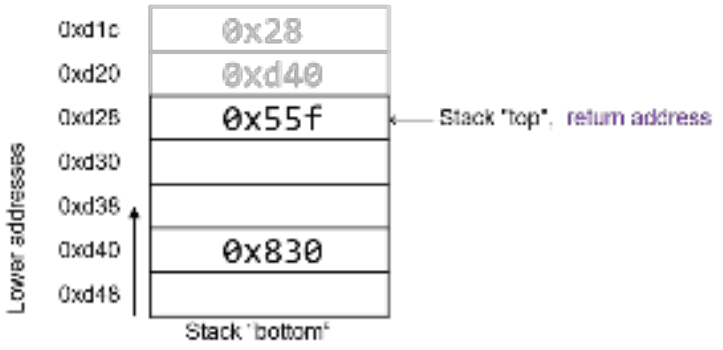
```
$ ./prog
```

Figure 20: State of execution

Understanding strange code (10)

```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x28
%edi	1
%rsp	0xd28
%rbp	0xd40
%rip	0x535



call stack

Terminal:

```
$ ./prog
```

Figure 21: State of execution

Understanding strange code (11)

```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x28
%edi	1
%rsp	0xd30
%rbp	0xd40
%rip	0x55f

call stack

Terminal:

```
$ ./prog
```

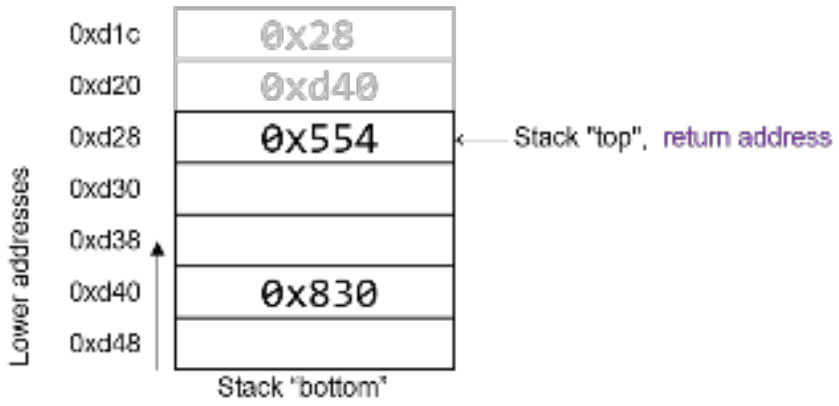
Equivalent to:
pop %rip

Figure 22: State of execution

Understanding strange code (12)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x0
%edi	1
%rsp	**0xd28**
%rbp	0xd40
%rip	**0x536**



Terminal:

```
$ ./prog
```

Figure 23: State of execution

Understanding strange code (13)

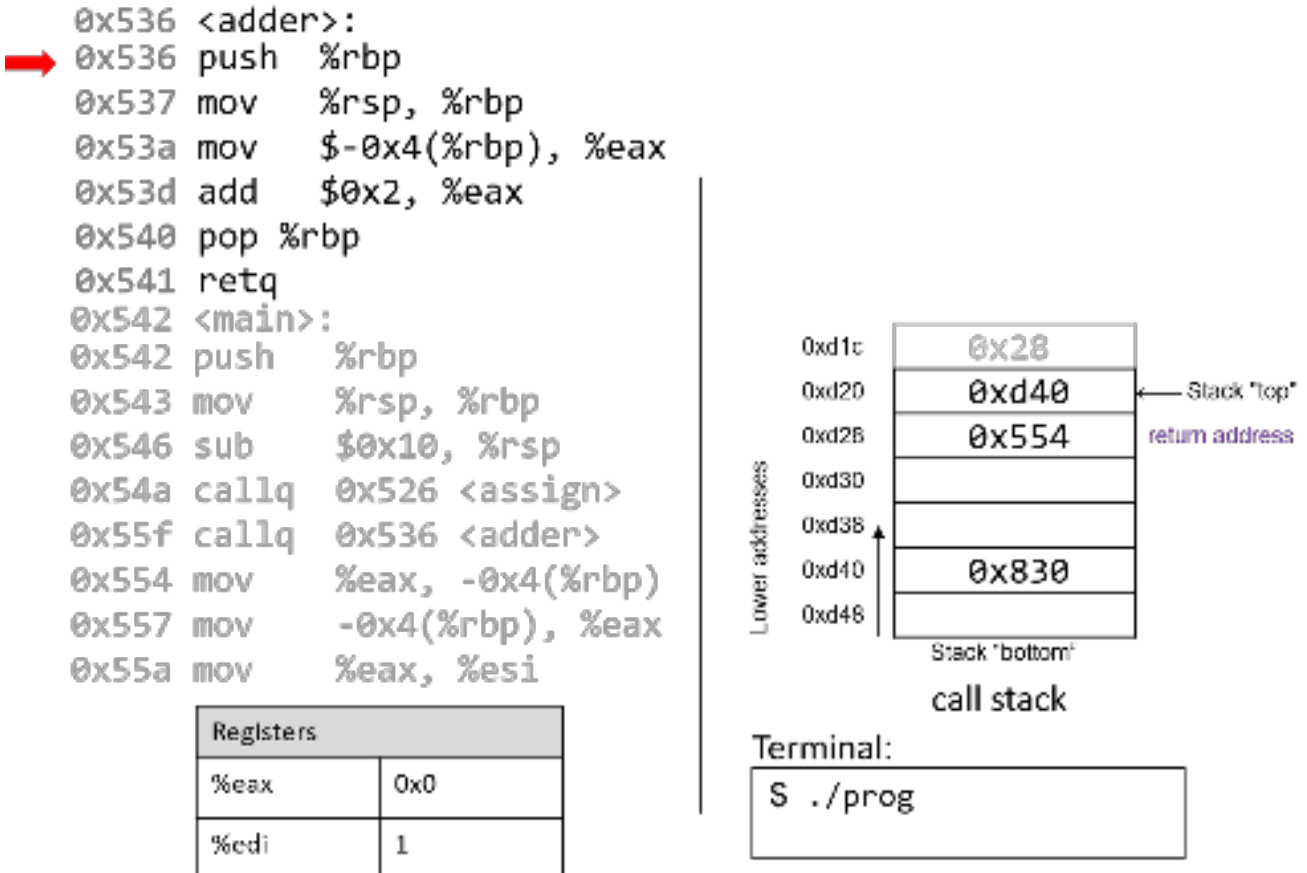
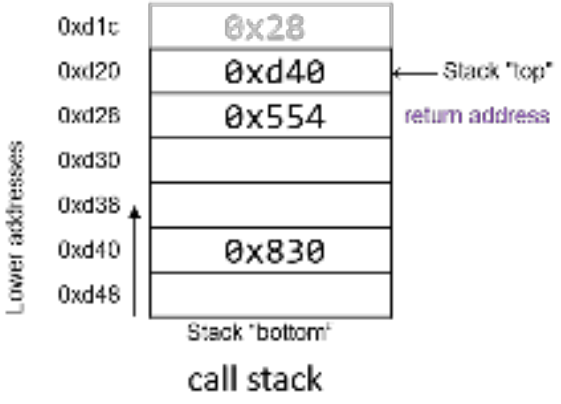


Figure 24: State of execution

Understanding strange code (14)

```
0x536 <adder>:
0x536 push  %rbp
0x537 mov   %rsp, %rbp
0x53a mov   $-0x4(%rbp), %eax
0x53d add   $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	**0xd20**
%rip	**0x53a**



Terminal:

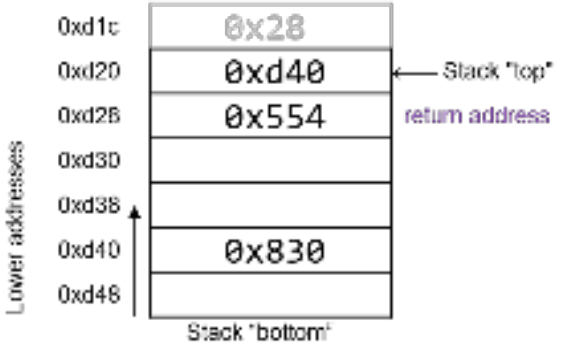
```
S ./prog
```

Figure 25: State of execution

Understanding strange code (15)

```
0x536 <adder>:
0x536 push  %rbp
0x537 mov   %rsp, %rbp
0x53a mov   $-0x4(%rbp), %eax
0x53d add   $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x28
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	0x53d



Terminal:

```
S ./prog
```

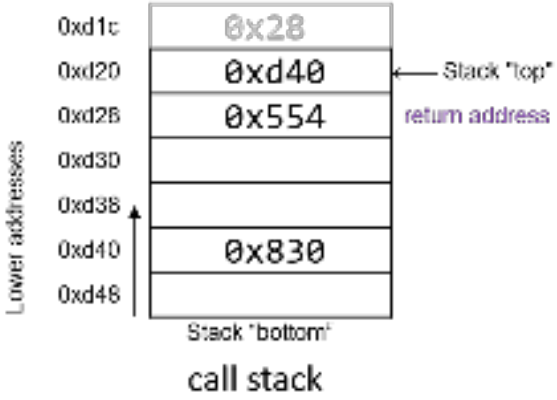
Using an old value on the stack!

Figure 26: State of execution

Understanding strange code (16)

```
0x536 <adder>:
0x536 push  %rbp
0x537 mov   %rsp, %rbp
0x53a mov   $-0x4(%rbp), %eax
0x53d add   $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x2A
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	0x540



Terminal:

```
S ./prog
```

Figure 27: State of execution

Understanding strange code (17)

```
0x536 <adder>:
0x536 push  %rbp
0x537 mov   %rsp, %rbp
0x53a mov   $-0x4(%rbp), %eax
0x53d add   $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x2A
%edi	1
%rsp	**0xd28**
%rbp	**0xd40**
%rip	**0x541**

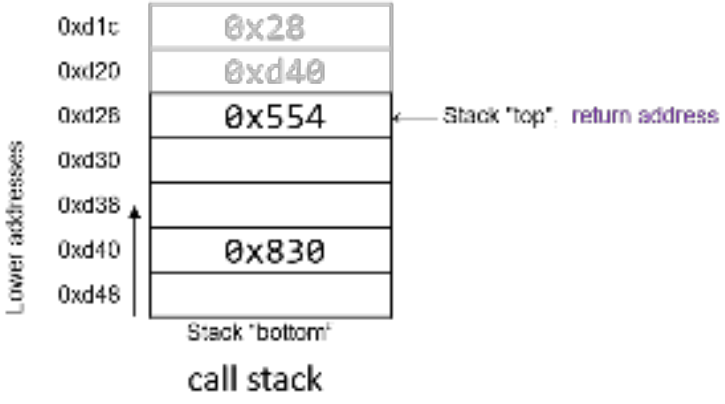
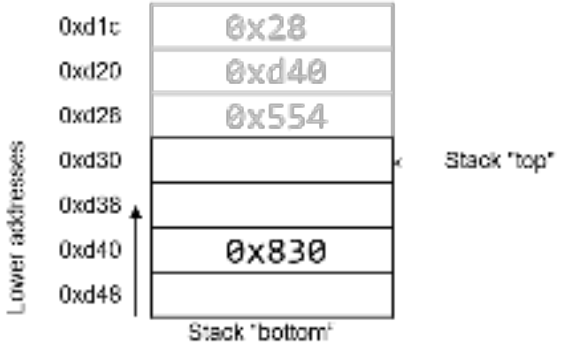


Figure 28: State of execution

Understanding strange code (18)

```
0x536 <adder>:
0x536 push   %rbp
0x537 mov    %rsp, %rbp
0x53a mov    $-0x4(%rbp), %eax
0x53d add    $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x2A
%edi	1
%rsp	0xd30
%rbp	0xd40
%rip	0x554



call stack

Terminal:

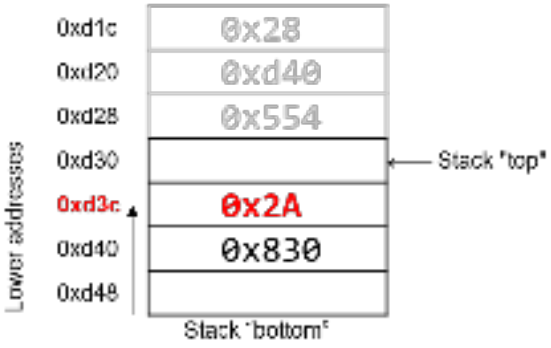
```
S ./prog
```

Figure 29: State of execution

Understanding strange code (19)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```

Registers	
%eax	0x2A
%edi	1
%rsp	0xd30
%rbp	0xd40
%rip	**0x557**



call stack

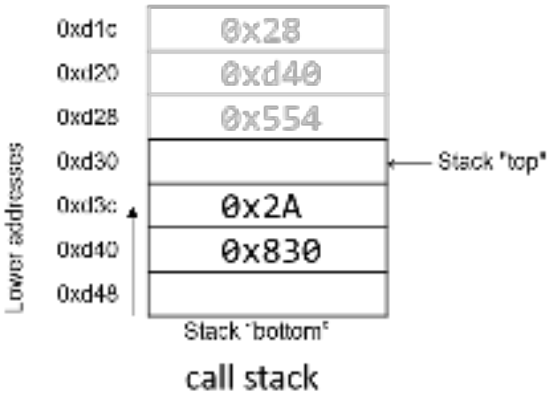
Terminal:

```
$ ./prog
```

Figure 30: State of execution

Understanding strange code (20)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```

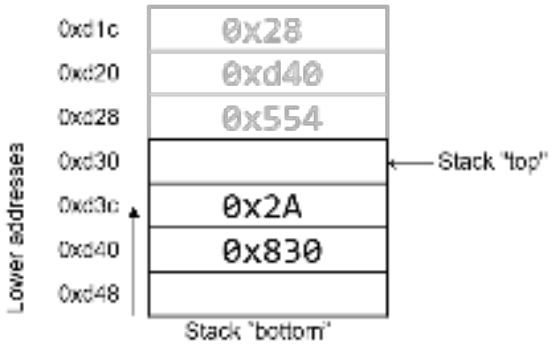


Registers	
%eax	0x2A
%edi	1
%rsp	0xd30
%rbp	0xd40
%rip	0x55a

Figure 31: State of execution

Understanding strange code (21)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```



Registers			
%eax	0x2A		
%edi	1		
%rsp	0xd30		
%rbp	0xd40		
%rip	0x55c	%esi	0x2A

Terminal:

```
$ ./prog
```

Figure 32: State of execution

Understanding strange code (22)

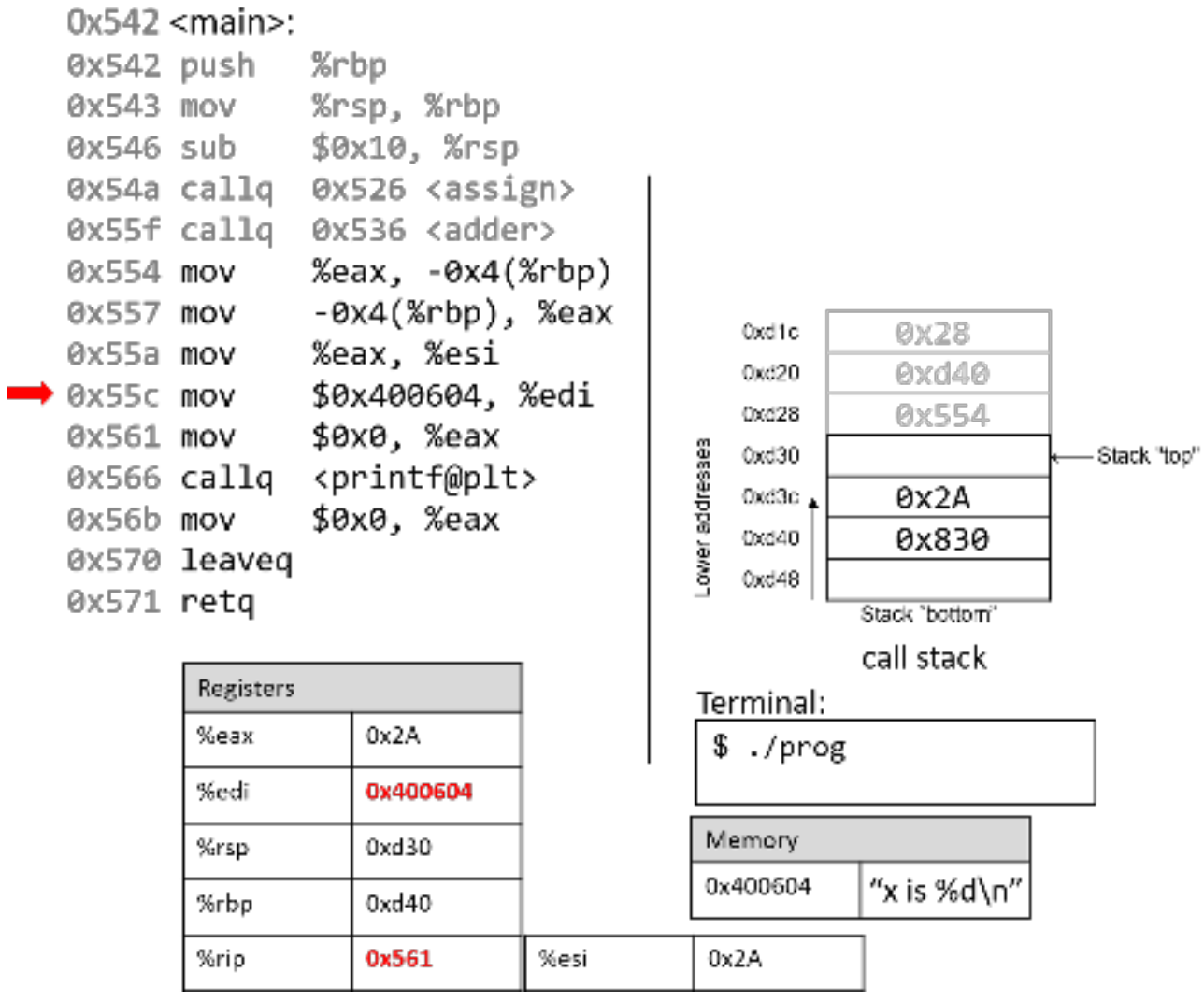
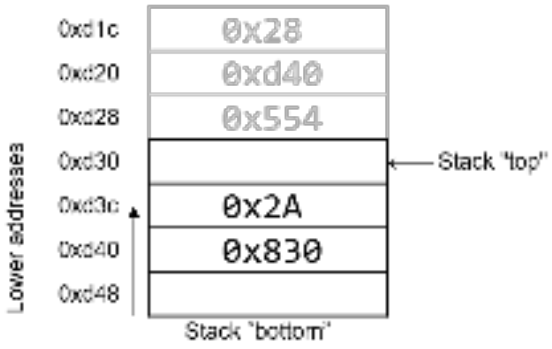


Figure 33: State of execution

Understanding strange code (23)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```



call stack

Registers	
%eax	0x0
%edi	0x400604
%rsp	0xd30
%rbp	0xd40
%rip	0x566

%esi	0x2A

Terminal:

```
$ ./prog
```

Memory	
0x400604	"x is %d\n"

Figure 34: State of execution

Understanding strange code (24)

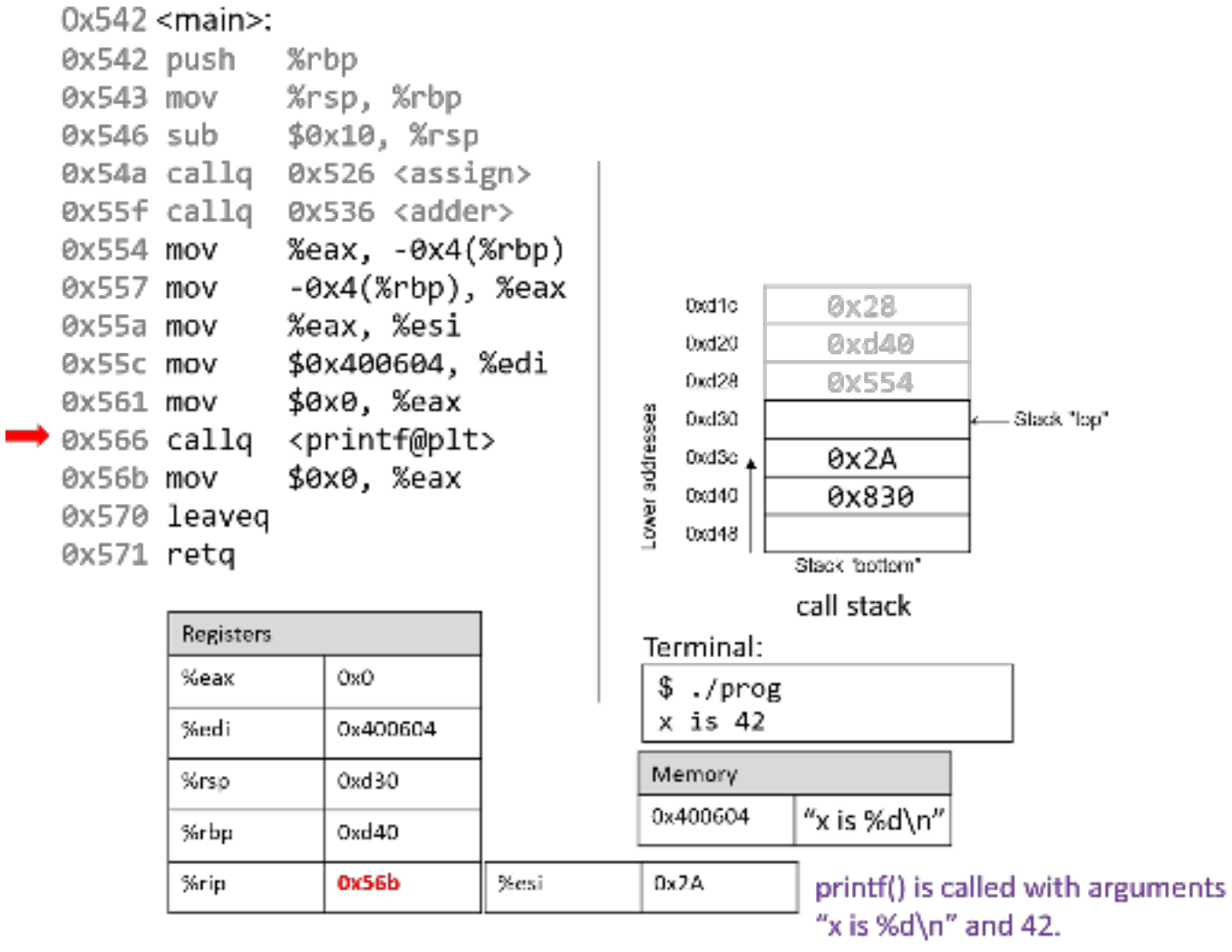
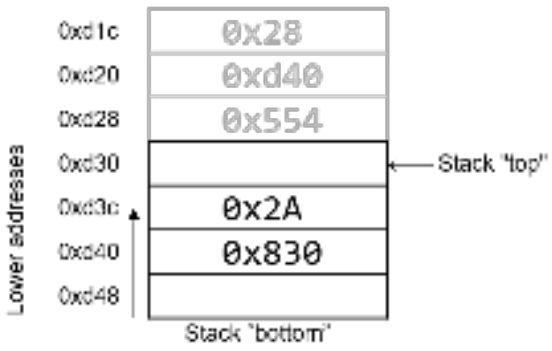


Figure 35: State of execution

Understanding strange code (25)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```

Registers	
%eax	0x0
%edi	0x400604
%rsp	0xd30
%rbp	0xd40
%rip	0x570



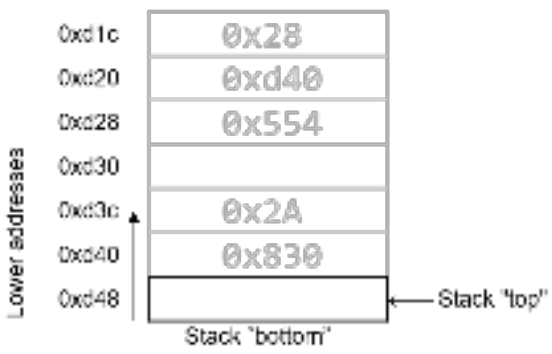
Terminal:

```
$ ./prog
x is 42
```

Figure 36: State of execution

Understanding strange code (26)

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```



call stack

Registers	
%eax	0x0
%edi	0x400604
%rsp	0xd48
%rbp	0x830
%rip	0x571

Terminal:

```
$ ./prog
x is 42
```

Equivalent to:
mov %rbp, %rsp
pop %rbp

Figure 37: State of execution

Why did it happen ?

- Program inadvertently uses old values on the stack.
- More harmful scenarios exists. Source of many vulnerabilities.
- Popular one: **Buffer overflows** (Will be covered in Architecture/OS course)

Push existing code

- Do `$ git switch lab10`. Commit all the changes.
- Do `$ git push -u origin lab10`.

Lab Exercise

Questions (Do the following in your repo)

- Do `$ git fetch && git merge`
- Do `$ git switch lab11` to see the `questions.md` in `lab11` folder.
- **To push changes: do `$ git push -u origin lab11`**

Class repo (for in-class demo)

- Accessible via
  - `git clone git@gitserver:class_repo`
- To see latest changes, cd to the class_repo and do
  - `git fetch && git merge`

Class has ended

- No more pushes to `gitserver`.
- Complete the exercises during off-lab hours.

Humble Request

Please keep the chairs in position before you leave.
(as a token of respect for our CFET staff)