CS 107 Lecture 16: Assembly Part II

Wed., February 14th, 2024

Computer Systems
Winter 2024
Stanford University
Computer Science Department

Reading: Course Reader: x86-64 Assembly Language, Textbook: Chapter 3.1-3.4

Lecturer: Chris Gregg

```
(gdb)
    ──looper.c──────────────────────────────────────────────────────────────
    3
    4       void count_from_offset(int offset)
    5       {
B+> 6           for (int i=0; i < 10; i++) {
    7               printf("count: %d\n", offset);
    8           }
    9       }
    10
    ────────────────────────────────────────────────────────────────────────
    0x400531 <count_from_offset+4>  sub    $0x20,%rsp
    0x400535 <count_from_offset+8>  mov    %edi,-0x14(%rbp)
B+> 0x400538 <count_from_offset+11> movl   $0x0,-0x4(%rbp)
    0x40053f <count_from_offset+18> jmp    0x40055e <count_fro
    0x400538 <count_from_offset+11> movl   $0x0,-0x4(%rbp)
    0x400541 <count_from_offset+20> mov    -0x14(%rbp),%edx
    0x400544 <count_from_offset+23> mov    -0x4(%rbp),%edx
    0x400547 <count_from_offset+26> add    %edx,%eax
    0x400549 <count_from_offset+28> mov    %eax,%esi
child process 8824 In: count_from_off* Line: 6    PC: 0x400538

(gdb)
```

Today's Topics

- Logistics
- Reading: Course Reader: x86-64 Assembly Language;Textbook, Chapter 3.1-3.4
- Programs from class: **`/afs/ir/class/cs107/samples/lect16`**
- PDF handout in class: https://web.stanford.edu/class/cs107/lectures/15/asm-intro.pdf
- Introduction to x86 Assembly Language
  - Overview of assembly code and the weirdness of x86 (primarily historical)
    - First example: HelloWorld, gcc -S, gdbtui
    - Second Example: Looper
  - Registers
  - Data formats
  - Addressing Modes
  - The **`mov`** instruction
  - Access to variables of various types

Looper

```c
#include<stdio.h>
#include<stdlib.h>

void count_from_offset(int offset)
{
    for (int i=0; i < 10; i++) {
        printf("Count: %d\n",i+offset);
    }
}

int main()
{
    count_from_offset(5);
    return 0;
}
```

```
.LC0:
        .string "Count: %d\n"
count_from_offset(int):
        pushq   %rbp
        pushq   %rbx
        subq    $8, %rsp
        movl    %edi, %ebp
        movl    $0, %ebx
.L3:
        cmpl    $9, %ebx
        jg      .L1
        leal    (%rbx,%rbp), %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        addl    $1, %ebx
        jmp     .L3
.L1:
        addq    $8, %rsp
        popq    %rbx
        popq    %rbp
        ret
main:
        subq    $8, %rsp
        movl    $5, %edi
        call    count_from_offset(int)
        movl    $0, %eax
        addq    $8, %rsp
        ret
```

https://godbolt.org/z/WG8Ye4ofs

Practice with Operand Forms

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
`0x100`	`0xFF`
`0x104`	`0xAB`
`0x108`	`0x13`
`0x10C`	`0x11`

Register	Value
`%rax`	`0x100`
`%rcx`	`0x1`
`%rdx`	`0x3`

Fill in the table to the right showing the values for the indicated operands.

Operand	Value
`%rax`	_____
`0x104`	_____
`$0x108`	_____
`(%rax)`	_____
`4(%rax)`	_____
`9(%rax,%rdx)`	_____
`260(%rcx,%rdx)`	_____
`0xFC(,%rcx,4)`	_____
`(%rax,%rdx,4)`	_____

Reminder:

Most general form: *Imm*(r_b,r_i,*S*)

Imm* + R[r_b] + R[r_i] * *s

Also: 260d = 0x104

Practice with Operand Forms

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
`0x100`	`0xFF`
`0x104`	`0xAB`
`0x108`	`0x13`
`0x10C`	`0x11`

Register	Value
`%rax`	`0x100`
`%rcx`	`0x1`
`%rdx`	`0x3`

Fill in the table to the right showing the values for the indicated operands.

Operand	Value	Comment
`%rax`	`0x100`	Register
`0x104`	`0xAB`	Absolute address
`$0x108`	`0x108`	Immediate
`(%rax)`	`0xFF`	Address `0x100`
`4(%rax)`	`0xAB`	Address `0x104`
`9(%rax,%rdx)`	`0x11`	Address `0x10C`
`260(%rcx,%rdx)`	`0x13`	Address `0x108`
`0xFC(,%rcx,4)`	`0xFF`	Address `0x100`
`(%rax,%rdx,4)`	`0x11`	Address `0x10C`
`2(%rax,%rdx,2)`	`0x13`	Address `0x108`

Reminder:

Most general form: *Imm*(r_b,r_i,*S*)

Imm* + R[r_b] + R[r_i] * *s

Also: 260d = 0x104

Data Movement Instructions

- Copying data from location to another is one of the most common instructions in assembly code.
- The x86 processors have a "Complex Instruction Set Architecture" (CISC), as opposed to some other processors—like the ARM that is most likely in your phone—called "Reduced Instruction Set Architecture" (RISC). The many ways to copy data is a hallmark of a CISC processor.
- We will discuss many different types of data movement instructions, starting with the **`mov`** instruction. The simple data movement instructions are as follows:

Instruction		Effect	Description
`MOV`	S, D	$D \leftarrow S$	Move
`movb`			Move byte
`movw`			Move word
`movl`			Move double word
`movq`			Move quad word
`movabsq`	I, R	$R \leftarrow I$	Move absolute quad word

mov

- The **`mov`** instruction has a source and a destination, but only one can potentially be a memory location (you need two instructions to copy from memory to memory: first copy to a register from memory, then copy to memory from the register)
- For most cases, the **`mov`** instruction only updates the specific register bytes or memory locations indicated by the destination operand.
- The exception is for the **`movl`** instruction: if it has a register as a destination, it will also set the high order 4 bytes of the register to 0.
- Examples:

```
movl $0x4050,%eax        Immediate—Register, 4 bytes
movw %bp,%sp             Register—Register,  2 bytes
movb (%rdi,%rcx),%al     Memory—Register,    1 byte
movb $-17,(%rsp)         Immediate—Memory,   1 byte
movq %rax,-12(%rbp)      Register—Memory,    8 bytes
```

movabsq

- The **movabsq** instruction is used when a 64-bit immediate (constant) value is needed in a register. The regular **movq** instruction can only take a 32-bit immediate value (because of the way the instruction is represented in memory).
- The **movabsq** instruction can have a 64-bit immediate as a source, and only a register as a destination.
- Example:

```
movabsq $0x0011223344556677, %rax
```

`movz` and `movs`

- There are two **`mov`** instructions that can be used to copy a smaller source to a larger destination: **`movz`** and **`movs`**.
- **`movz`** fills the remaining bytes with zeros
- **`movs`** fills the remaining bytes by sign-extending the most significant bit in the source.
- The source must be from memory or a register, and the destination is a register.
- There are six ways to move a 1- or 2-byte size to a 2-, 4- or 8-byte size, for each case:

Instruction		Effect	Description
`MOVZ`	*S*, *R*	*R* ← *ZeroExtend(S)*	Move with zero extension
`movzbw`			Move zero-extended byte to word
`movzbl`			Move zero-extended byte to double word
`movzwl`			Move zero-extended word to double word
`movzbq`			Move zero-extended byte to quad word
`movzwq`			Move zero-extended word to quad word

- There isn't a 4-byte source to 8-byte destination, as it is already covered by the **`movl`** instruction with a register destination, which always populates the upper 4 bytes with 0s.

`movz` and `movs`

- **`movs`** fills the remaining bytes by sign-extending the most significant bit in the source.
- There is also a **`cltq`** instruction, which is a more compact encoding of

```
movslq %eax,%rax
```

Instruction		Effect	Description
`MOVZ`	*S*, *R*	*R* ← *SignExtend(S)*	Move with sign extension
`movsbw`			Move sign-extended byte to word
`movsbl`			Move sign-extended byte to double word
`movswl`			Move sign-extended word to double word
`movsbq`			Move sign-extended byte to quad word
`movswq`			Move sign-extended word to quad word
`movslq`			Move sign-extended double word to quad word
`cltq`		`%rax` ← *SignExtend(`%eax`)*	Sign-extend `%eax` to `%rax`

Practice with `mov`

- For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands (e.g., **`mov`** can be **`movb`**, **`movw`**, **`movl`**, **`movq`**)

```
mov____ %eax, (%rsp)

mov____ (%rax), %dx

mov____ $0xFF, %bl

mov____ (%rsp,%rdx,4), %dl

mov____ (%rdx), %rax

mov____ %dx, (%rax)
```

Practice with `mov`

- For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands (e.g., **`mov`** can be **`movb`**, **`movw`**, **`movl`**, **`movq`**)

Instruction	Description
`movl %eax, (%rsp)`	copy 4 bytes from %eax into memory at (%rsp)
`movw (%rax), %dx`	copy 2 bytes from memory at (%rax) to %dx
`movb $0xFF, %bl`	set %bl to hold the 1-byte value $0xFF
`movb (%rsp,%rdx,4), %dl`	copy 1 byte from memory at (%rsp + 4*%rdx) to %dl
`movq (%rdx), %rax`	copy 8 bytes from memory at (%rdx) to %rax
`movw %dx, (%rax)`	copy 2 bytes from %dx to memory at (%rax)

Practice with `mov`

- Each of the following lines of code generate an error message if we use the assembler.

```
movl %rax, (%rsp)

movw (%rax),4(%rsp)

movb %al, %sl

movq %rax,$0x123

movl %eax,%dx

movb %si, 8(%rbp)
```

Practice with `mov`

- Each of the following lines of code generate an error message if we use the assembler.

`movl %rax, (%rsp)`	Mismatch between instruction suffix and register ID
`movw (%rax), 4(%rsp)`	Cannot have both source and destination be memory registers
`movb %al, %sl`	No register named `%sl`
`movq %rax, $0x123`	Cannot have immediate destination
`movl %eax, %dx`	Destination operand incorrect size
`movb %si, 8(%rbp)`	Mismatch between instruction suffix and register ID (`%si` is a word)

- The good news: you won't be *writing* any assembly

More C to Assembly

exchange.c

```
#include<stdio.h>
#include<stdlib.h>

long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}

int main()
{
    long x = 1000;
    long y = 42;

    printf("Before exchange: x:%lu, y:%lu\n", x, y);
    y = exchange(&x, y);
    printf("After exchange: x:%lu, y:%lu\n", x, y);
    return 0;
}
```

Compiler Explorer:

https://gcc.godbolt.org

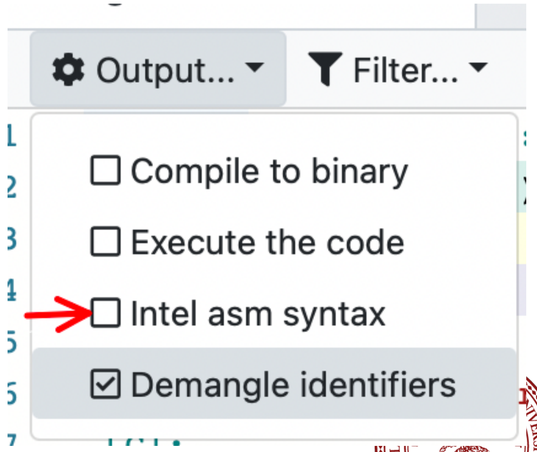
Use compiler:

x86-64 gcc 9.3

Flags:

-g -Og -std=gnu99

uncheck Intel asm syntax in settings



Compile line:

```
gcc -g -Og -std=gnu99 -Wall $warnflags  exchange.c  -o exchange
```

More C to Assembly

looper.c

```
#include<stdio.h>
#include<stdlib.h>

void count_from_offset(int offset)
{
    for (int i=0; i < 10; i++) {
        printf("Count: %d\n",i+offset);
    }
}

int main()
{
    count_from_offset(5);
    return 0;
}
```

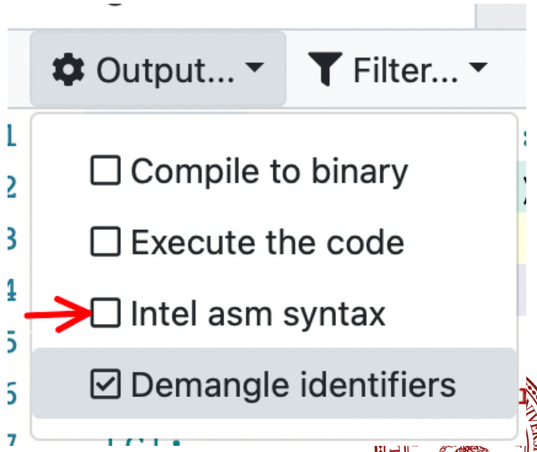
Compiler Explorer:

https://gcc.godbolt.org

Use compiler:
x86-64 gcc 9.3

Flags:
-g -Og -std=gnu99

uncheck Intel asm syntax in settings



Slightly different compile line:

```
gcc -g -Og -std=gnu99 -Wall $warnflags  looper.c  -o looper
```

The `lea` instruction

- The **`lea`** instruction is related to the **`mov`** instruction. It has the form of an instruction that reads from memory to a register, but it *does not reference memory at all*.
- Its first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination.
- You can think of it as the "&" operator in C — it retrieves the address of a memory location:

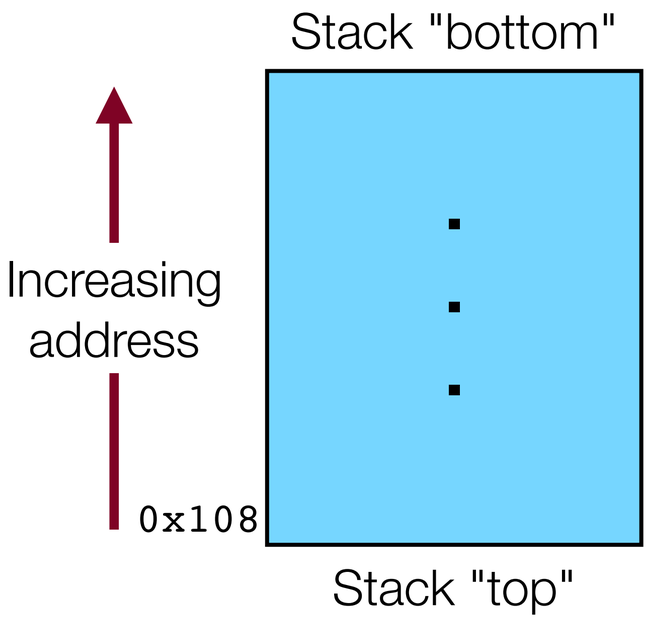
Instruction	Effect	Description
`leaq S,D`	$D \leftarrow \&S$	Load effective address

Examples: if **`%rax`** holds value *x* and **`%rcx`** holds value *y*:

```
leaq 6(%rax), %rdx          : %rdx now holds x + 6
leaq (%rax,%rcx), %rdx      : %rdx now holds x + y
leaq (%rax,%rcx,4), %rdx    : %rdx now holds x + 4*y
leaq 7(%rax,%rax,8), %rdx   : %rdx now holds 7 + 9x
leaq 0xA(,%rcx,4), %rdx     : %rdx now holds 10 + 4y
leaq 9(%rax,%rcx,2), %rdx   : %rdx now holds 9 + x + 2y
```

Pushing and Popping from the Stack

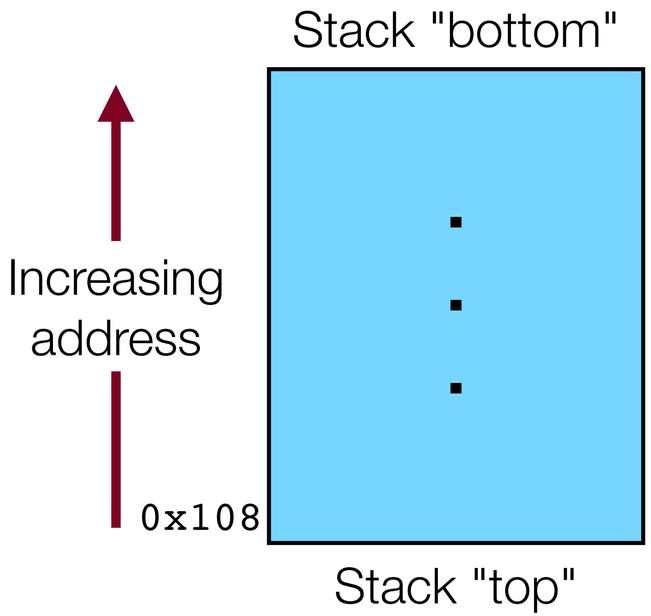
- As we have seen from stack-based memory allocation in C, the stack is an important part of our program, and assembly language has two built-in operations to use the stack.
- Just like the stack ADT, they have a last-in, first-out discipline.
- By convention, we draw stacks upside down, and the stack "grows" downward.



Pushing and Popping from the Stack

- The push and pop operations write and read from the stack, and they also modify the stack pointer, **`%rsp`**:

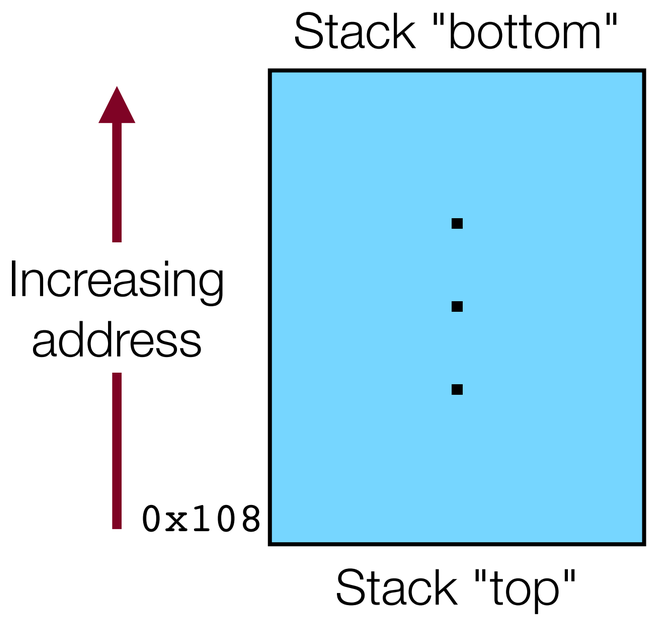
Instruction		Effect	Description
`pushq`	*S*	R[`%rsp`] ⟵ R[`%rsp`]-8;	Push quad word
		M[R[%rsp]] ⟵ *S*	
`popq`	*D*	D ⟵ M[R[`%rsp`]];	Push quad word
		R[`%rsp`] ⟵ R[`%rsp`]+8	



Pushing and Popping from the Stack

- Example:

Initially	
`%rax`	`0x123`
`%rdx`	`0`
`%rsp`	`0x108`

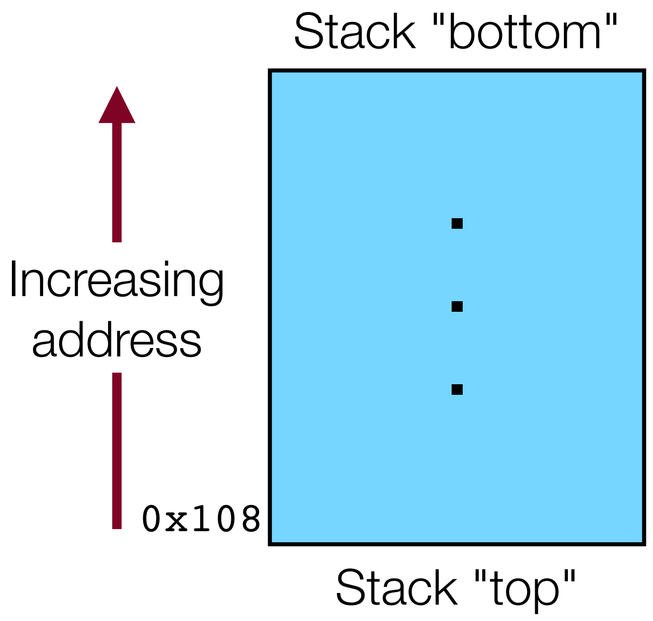


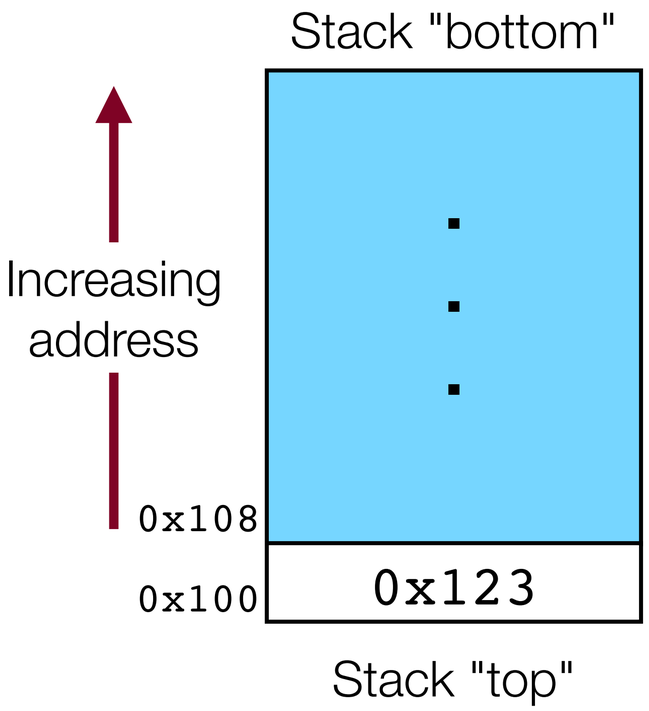
Pushing and Popping from the Stack

- Example:

Initially	
`%rax`	`0x123`
`%rdx`	`0`
`%rsp`	`0x108`

`pushq %rax`	
`%rax`	`0x123`
`%rdx`	`0`
`%rsp`	`0x100`





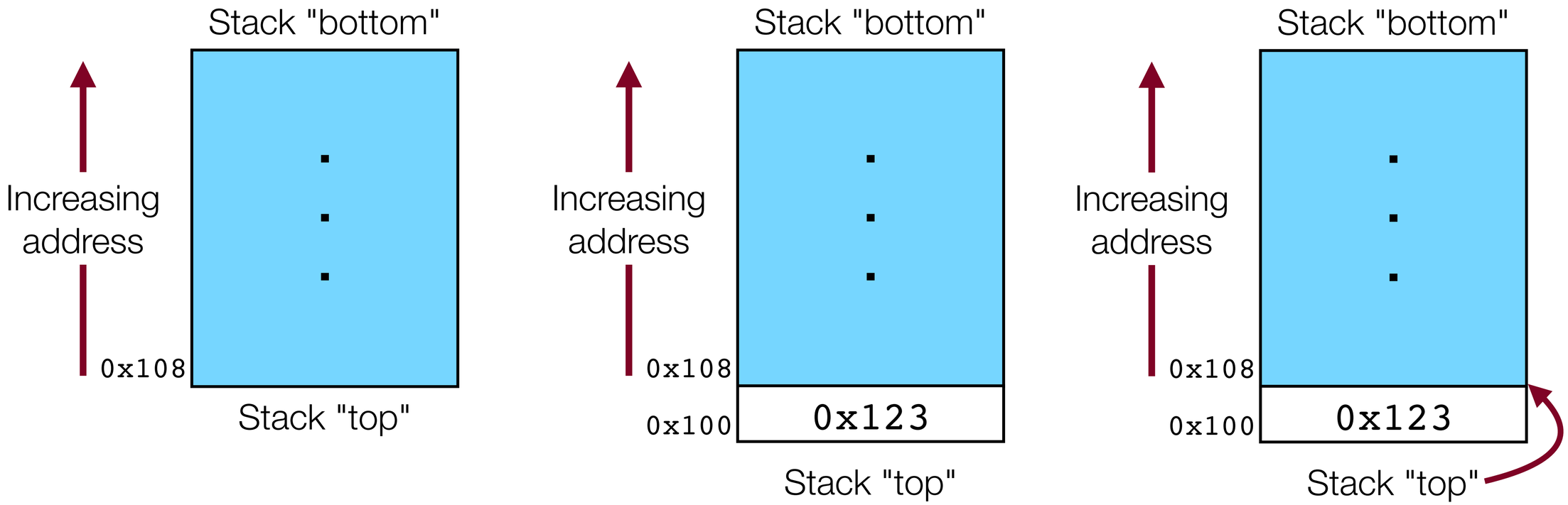
Pushing and Popping from the Stack

- Example:

Initially	
%rax	0x123
%rdx	0
%rsp	0x108

pushq %rax	
%rax	0x123
%rdx	0
%rsp	0x100

popq %rdx	
%rax	0x123
%rdx	0x123
%rsp	0x108



Pushing and Popping from the Stack

- As you can tell, pushing a quad word onto the stack involves first decrementing the stack pointer by 8, and then writing the value at the new top-of-stack address.
- Therefore, the behavior of the instruction **`pushq %rbp`** is equivalent to the pair of instructions:

```
subq $8, %rsp        (subq is a subtraction, and this decrements the stack pointer)
movq $rbp,(%rsp)     (Store %rbp on the stack)
```

- The behavior of the instruction **`popq %rax`** is equivalent to the pair of instructions:

```
movq (%rsp), %rax    (Read %rax from the stack)
addq $8,%rsp         (Increment the stack pointer)
```

References and Advanced Reading

- References:
  - Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  - CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  - gdbtui: https://beej.us/guide/bggdb/
  - More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  - Compiler explorer: https://gcc.godbolt.org
- Advanced Reading:
  - x86-64 Intel Software Developer manual: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
  - history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  - x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64

Extra Slides

Extra Slides

The aside on page 184 of the textbook is interesting: you should understand how data movement changes the destination register:

Aside Understanding how data movement changes a destination register

As described, there are two different conventions regarding whether and how data movement instructions modify the upper bytes of a destination register. This distinction is illustrated by the following code sequence:

```
1    movabsq $0x0011223344556677, %rax    %rax = 0011223344556677
2    movb    $-1, %al                     %rax = 00112233445566FF
3    movw    $-1, %ax                     %rax = 001122334455FFFF
4    movl    $-1, %eax                    %rax = 00000000FFFFFFFF
5    movq    $-1, %rax                    %rax = FFFFFFFFFFFFFFFF
```

In the following discussion, we use hexadecimal notation. In the example, the instruction on line 1 initializes register %rax to the pattern 0011223344556677. The remaining instructions have immediate value −1 as their source values. Recall that the hexadecimal representation of −1 is of the form FF···F, where the number of F's is twice the number of bytes in the representation. The movb instruction (line 2) therefore sets the low-order byte of %rax to FF, while the movw instruction (line 3) sets the low-order 2 bytes to FFFF, with the remaining bytes unchanged. The movl instruction (line 4) sets the low-order 4 bytes to FFFFFFFF, but it also sets the high-order 4 bytes to 00000000. Finally, the movq instruction (line 5) sets the complete register to FFFFFFFFFFFFFFFF.