CS 107
Lecture 11:
Assembly Part II

Friday, February 11, 2022

Computer Systems
Winter 2022
Stanford University
Computer Science Department

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

Lecturer: Chris Gregg

```c
// Type your code here, or load an example.
void while_loop()
{
    int i=100;
    int total;
    while (i >=0) {
        total += i;
        i--;
    }
}
```

11010 .LX0: .text //

```asm
while_loop():
  mov eax, 100
  jmp .L2
.L3:
  sub eax, 1
.L2:
  test eax, eax
  jns .L3
  rep ret
```

Today's Topics

- Reading: Course Reader: x86-64 Assembly Language, Textbook: Chapter 3.5-3.6
- Logistics
  - Midterm Comments
- Programs from class: `/afs/ir/class/cs107/samples/lect11`
- More x86 Assembly Language
  - Review of what we know so far
  - The `lea` instruction
  - `push`ing and `pop`ping from the stack
  - Unary operations, Binary operations, Shift operations
  - Special multiplication and division
  - Control
    - Condition codes
    - Conditional branches

What did we cover last Monday?

- Registers:
  - 16 regular integer registers, **`%rax`**, **`%rbx`**, ...
  - naming is historical, and a register has four nested parts:

`%rax`	`%eax`	`%ax`	`%al`	return value

- Operand forms: lots of ways we can refer to immediate values, register values, or memory:

Address	Value	Register	Value
`0x100`	`0xFF`	`%rax`	`0x100`
`0x104`	`0xAB`	`%rcx`	`0x1`
`0x108`	`0x13`	`%rdx`	`0x3`
`0x10C`	`0x11`		

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`

What did we cover last Monday? (continued)

- Data movement instructions:

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

- Examples:

```
movl $0x4050,%eax        Immediate—Register,  4 bytes
movw %cx,%dx             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
```

What did we cover last Monday? (continued)

```
movl $0x4050, %eax
movq %rsp, %rax
movw %ax, %dx
movl $1,%ecx
movb (%rdi, %rcx), %al
movb $0x61, (%rsp)
movb $0x61,(%rdi, %rcx)
movq %rax, 12(%rsp)
```

```
./mov_examples_main
Before function call: hello
After function call: hallo
```

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*.
- It's 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
```

The `lea` instruction

- Take a look at the following C code (left) and assembly generated by gcc (right):

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

int main() {
    int a = 42;
    int *aptr = &a;
    printf("a: %d, aptr: %p", a, aptr);
}
```

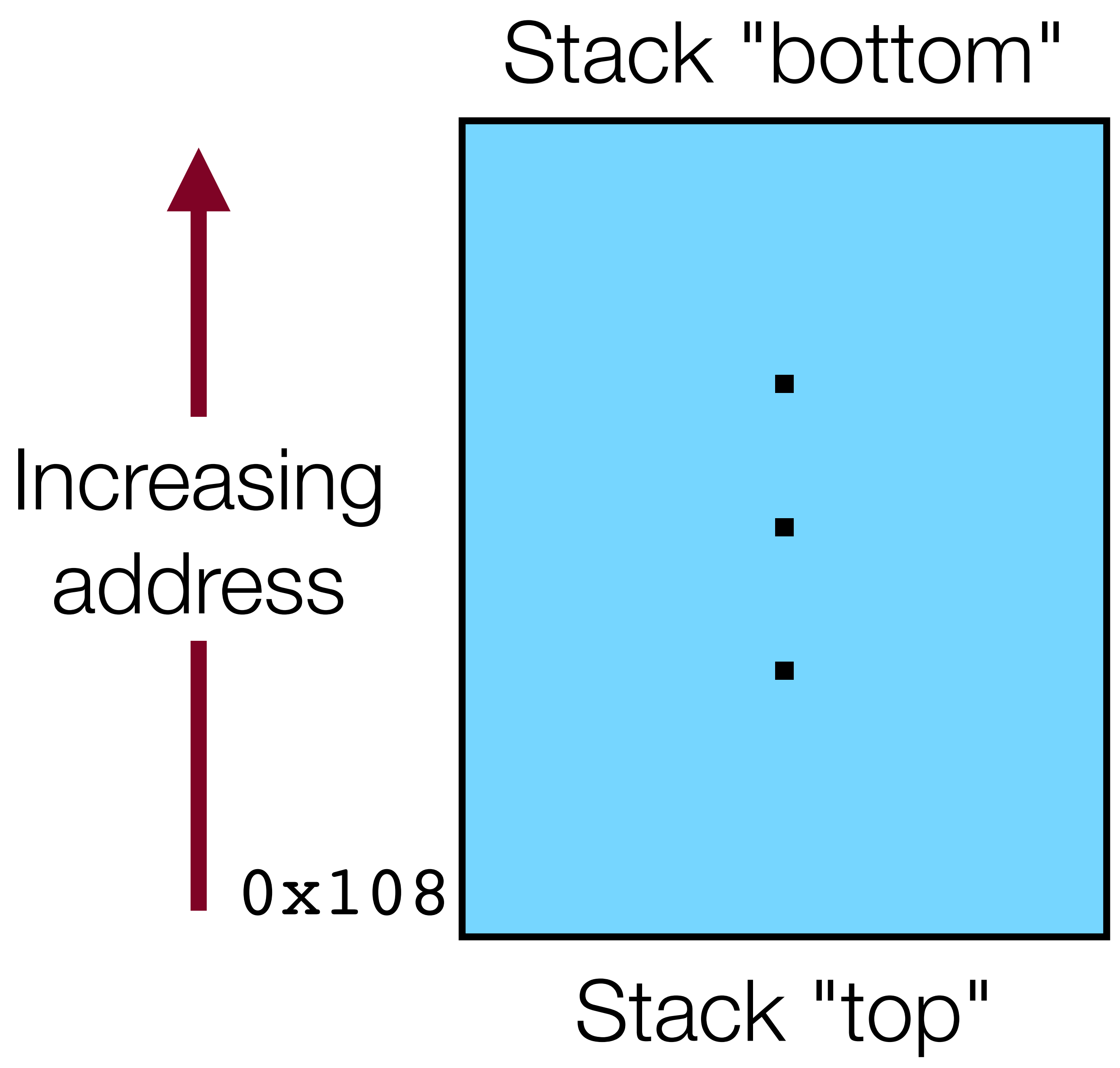
```
.LC0:
        .string "a: %d, aptr: %p"
main:
        subq    $24, %rsp
        movl    $42, 12(%rsp)
        leaq    12(%rsp), %rdx
        movl    $42, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        addq    $24, %rsp
        ret
```

- Notice that the value `42` is moved into memory via a `mov` instruction, while the address of that value is moved into `%rdx` using the `lea` instruction. In both cases, `12(%rsp)` is the location in memory that is referred to, but mov moves data to that address, and lea moves the address into the register.

- Note: we often use the lea instruction for calculations that aren't memory addresses! This is because we get a nifty linear equation from lea, and it really doesn't matter if it isn't an address.

Pushing and Popping from the Stack

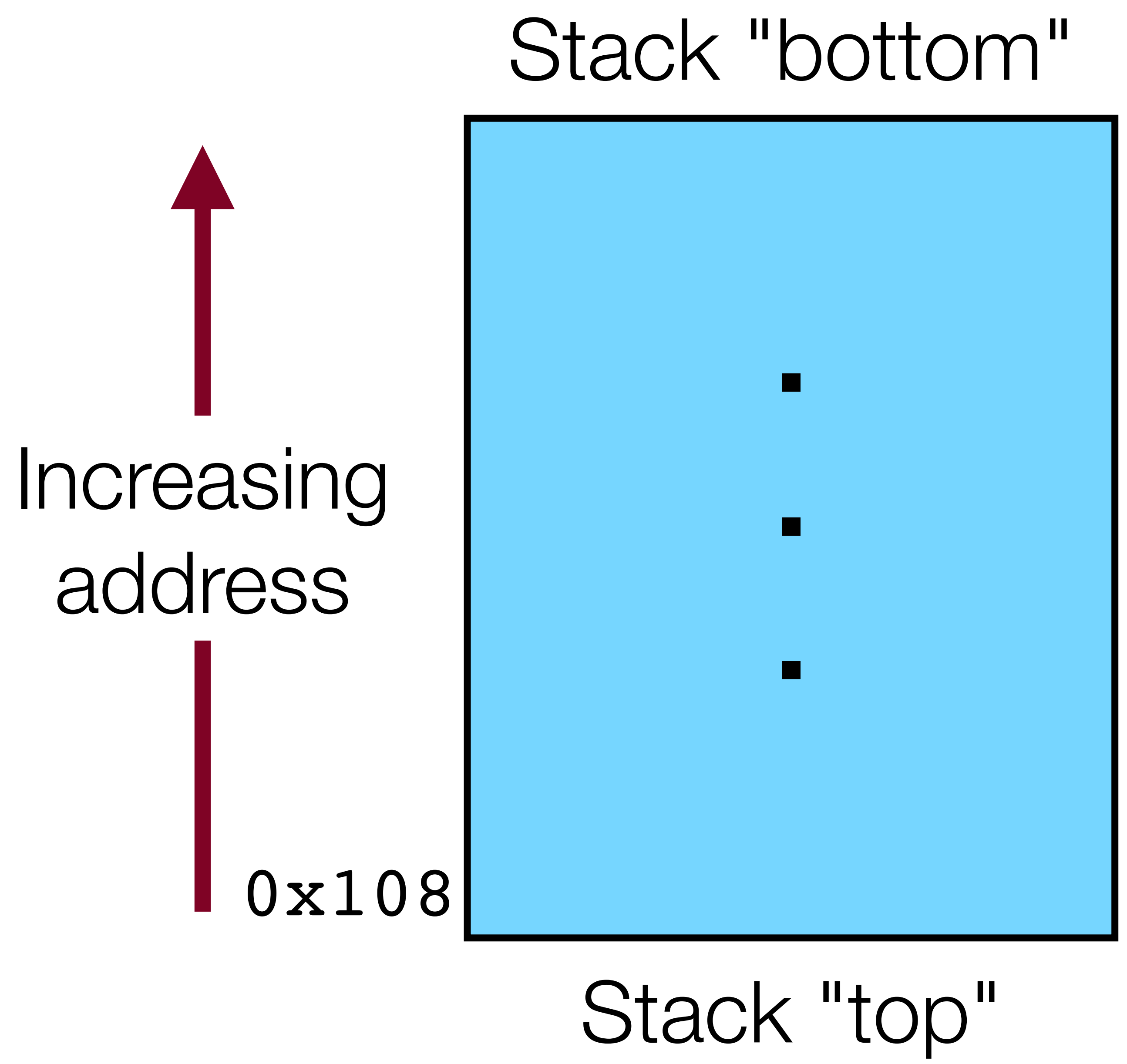
- As we have seen from stack-based memory allocation in C, the stack is an important part of our program, and assembly language has two built-in operations to use the stack.
- Just like the stack ADT, they have a first-in, first-out discipline.
- By convention, we draw stacks upside down, and the stack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

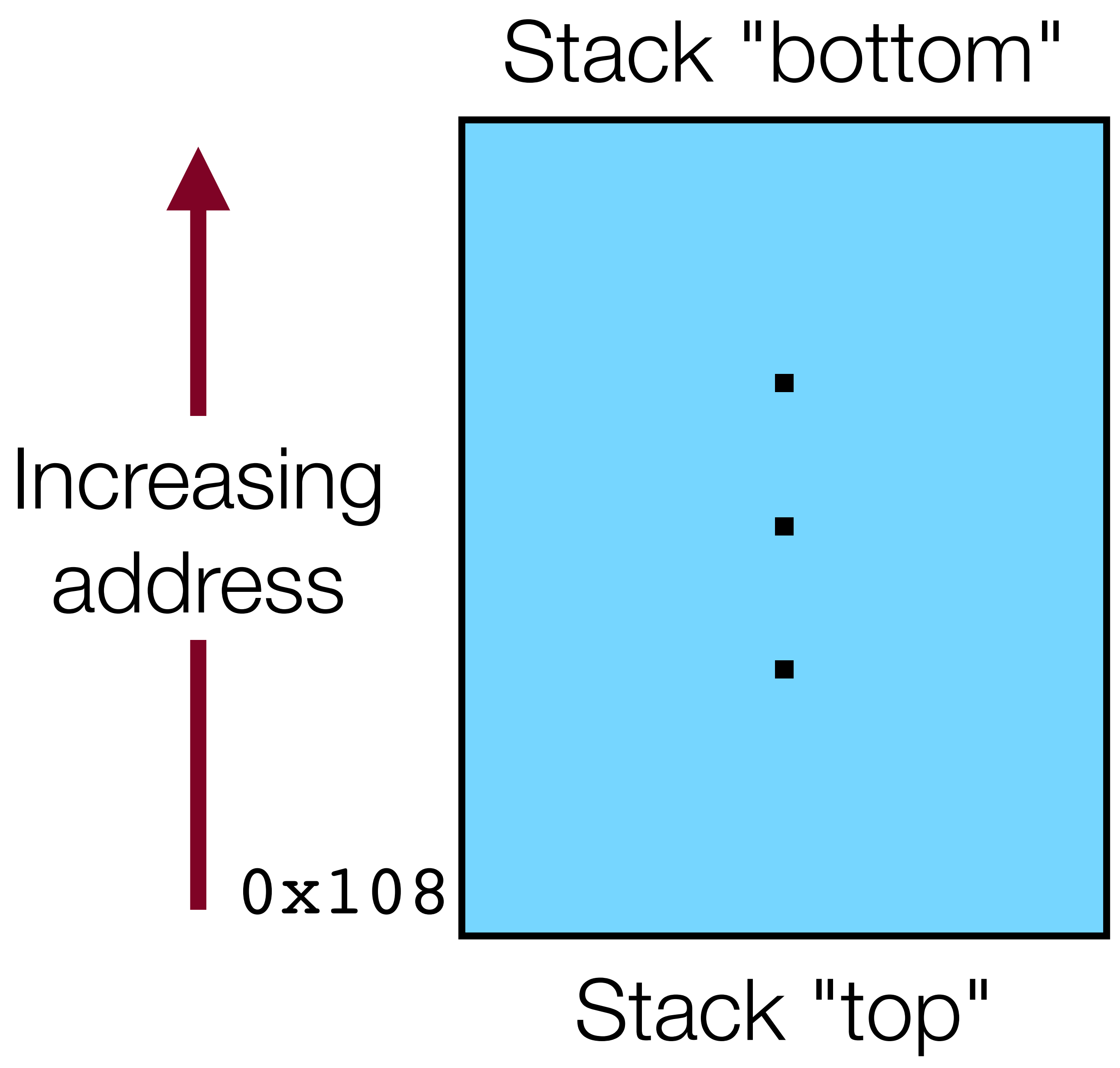
Instruct	Effect	Description
`pushq` *S*	R[`%rsp`] ⟵ R[`%rsp`]-8;	Push quad word
	M[R[%rsp]] ⟵ *S*	
`popq` *D*	D ⟵ M[R[`%rsp`]];	Push quad word
	R[`%rsp`] ⟵ R[`%rsp`]+8	



Pushing and Popping from the Stack

- Example:

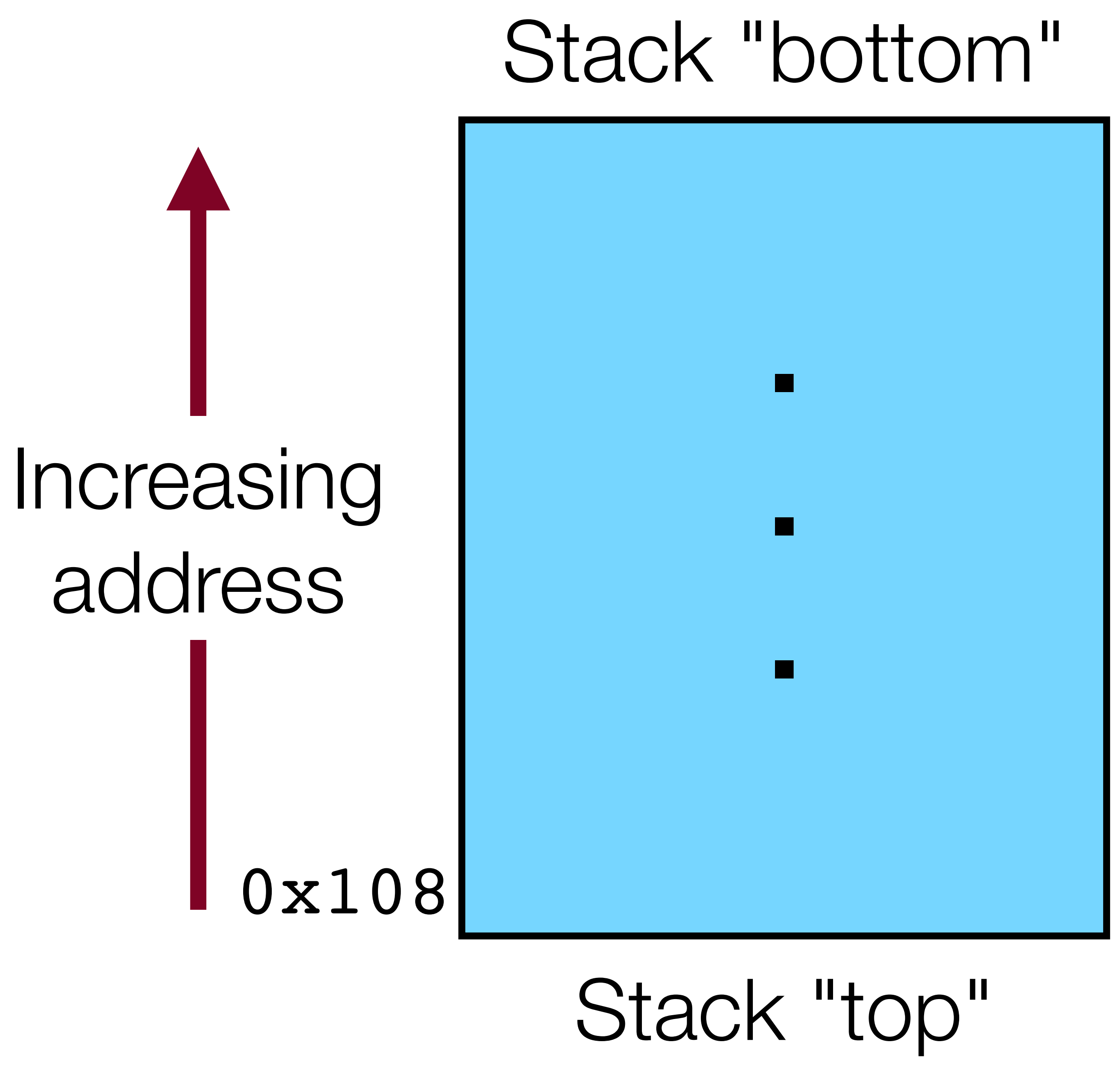
Initially	
`%rax`	`0x123`
`%rdx`	`0`
`%rsp`	`0x108`



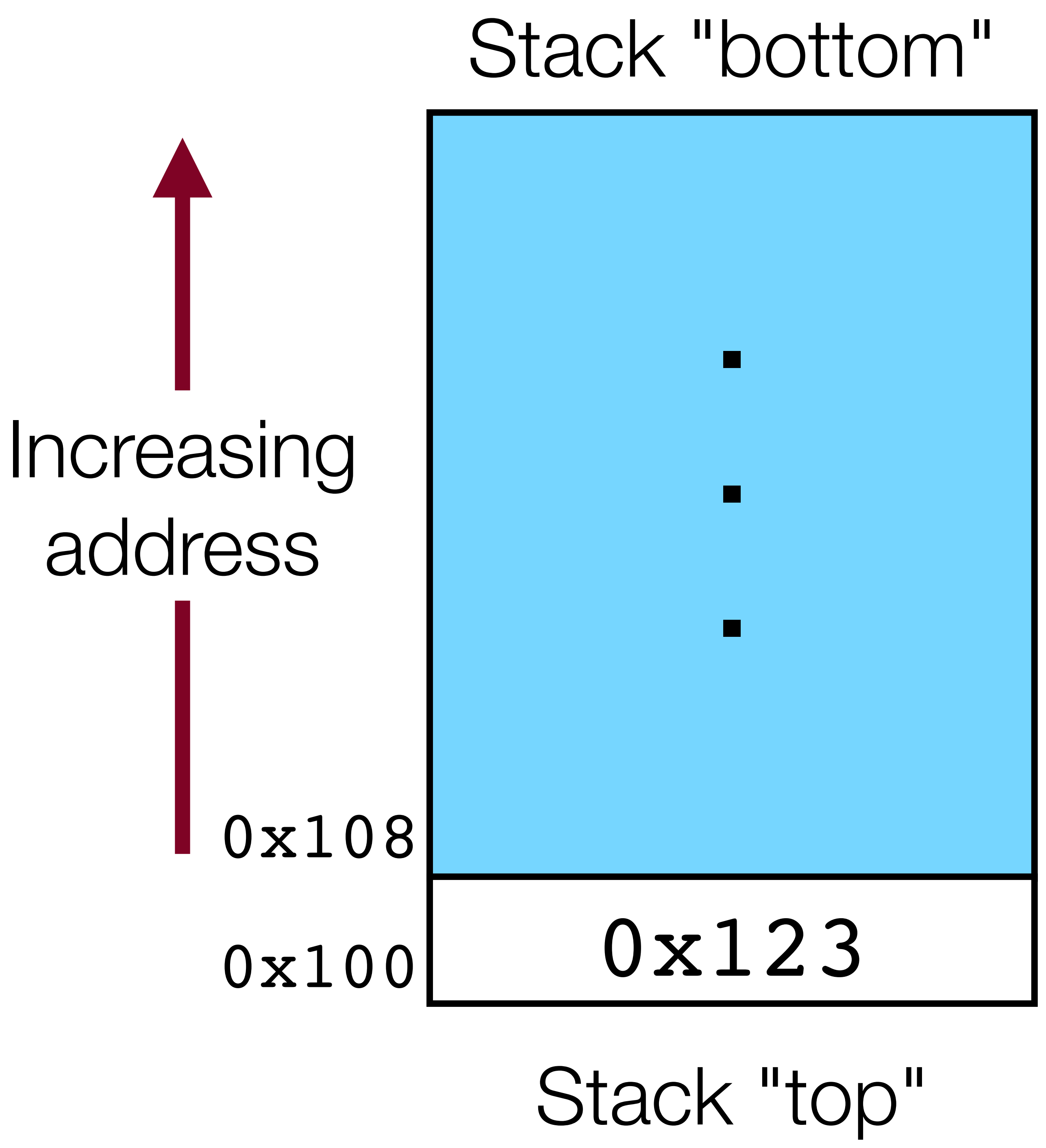
Pushing and Popping from the Stack

- Example:

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



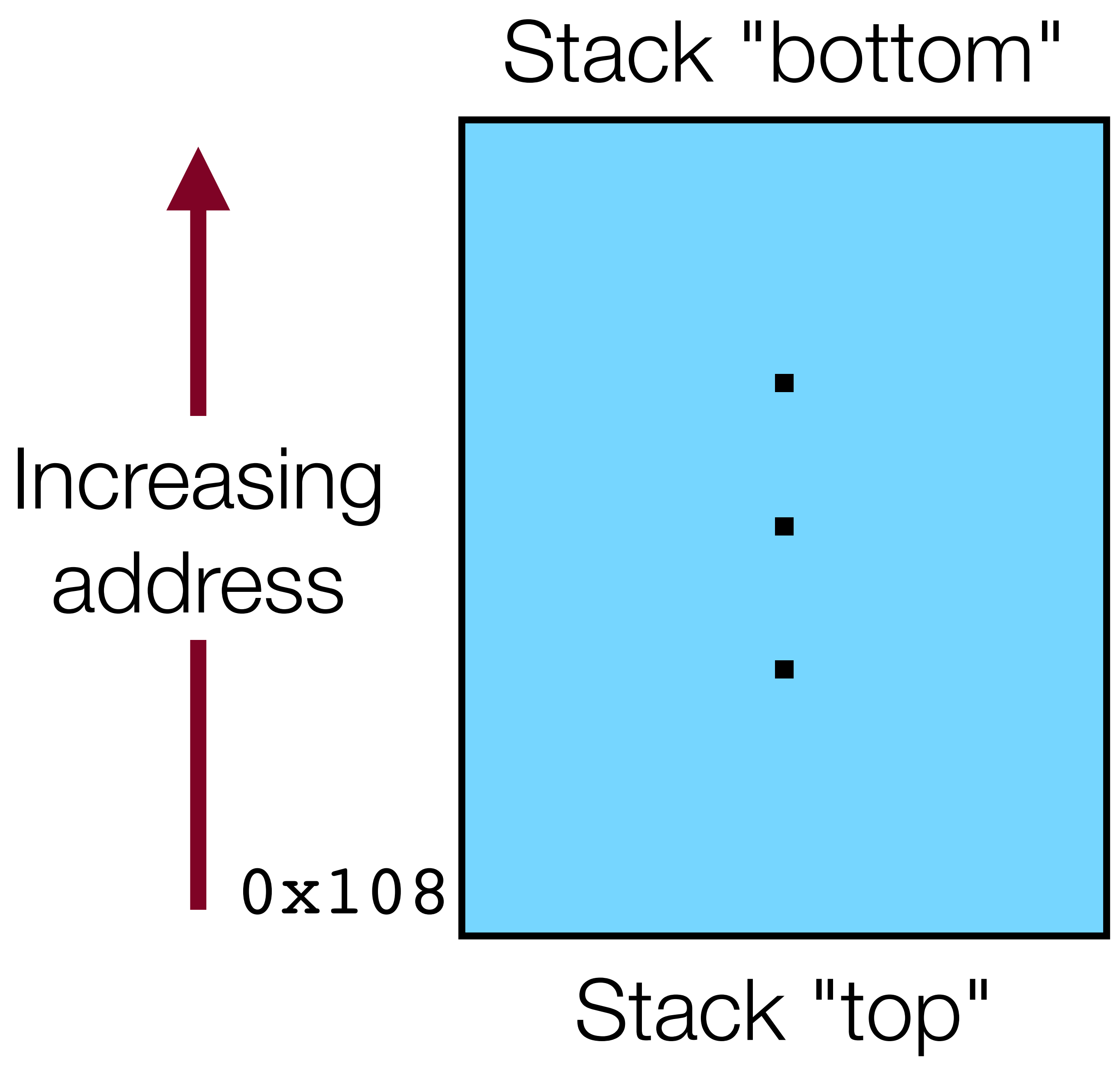
`pushq %rax`	
`%rax`	`0x123`
`%rdx`	`0`
`%rsp`	`0x100`



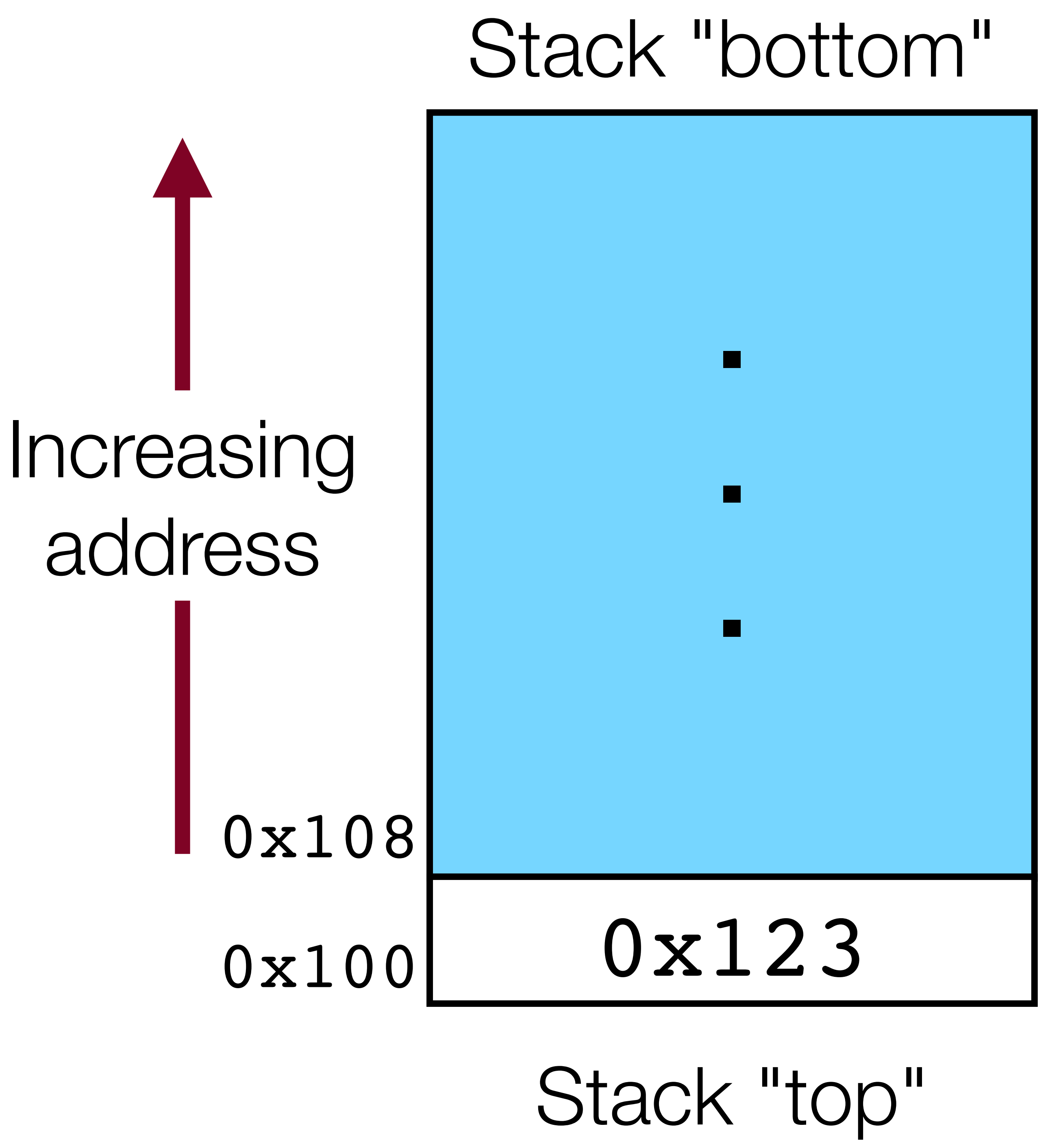
Pushing and Popping from the Stack

- Example:

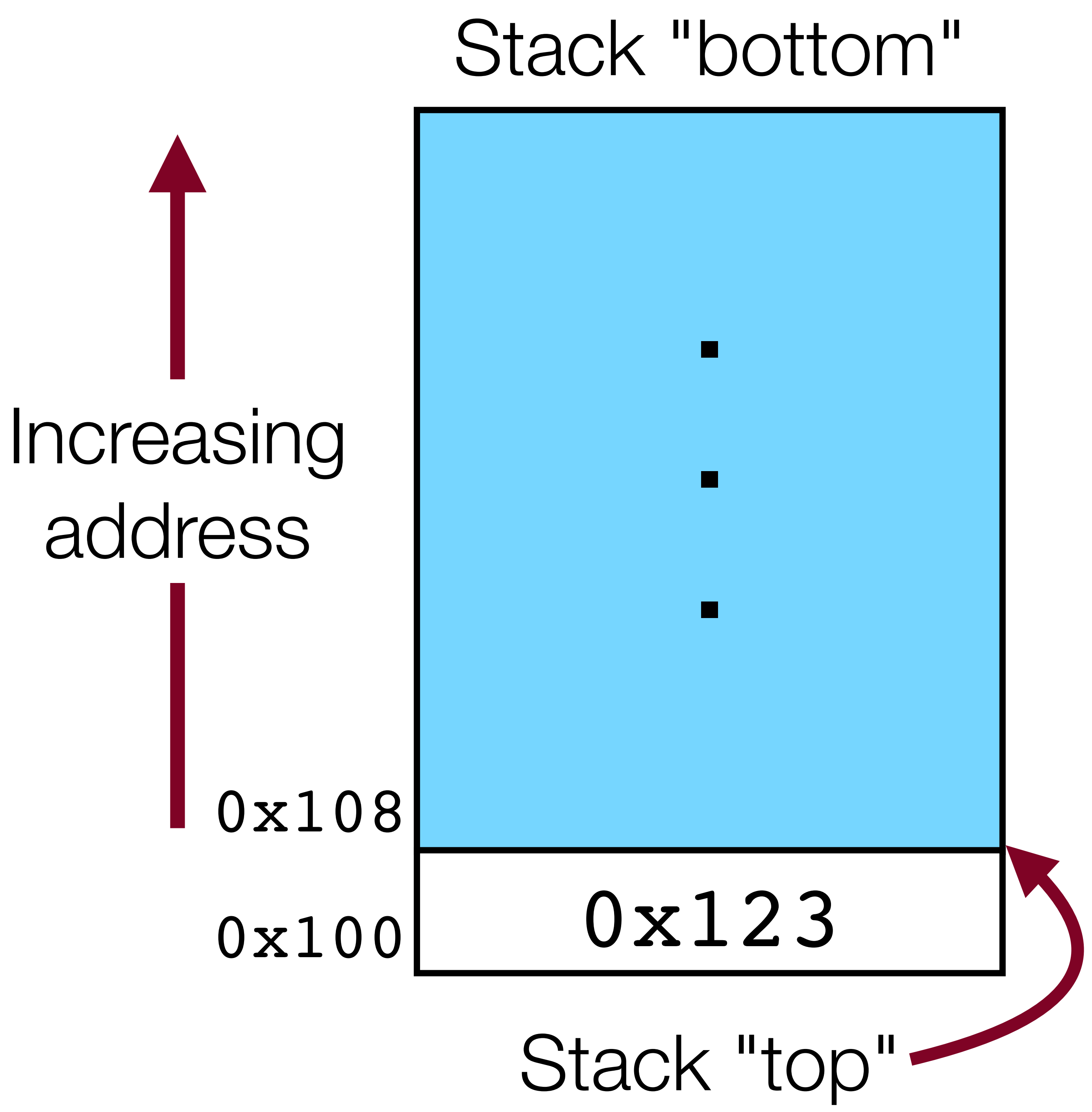
Initially	
`%rax`	`0x123`
`%rdx`	`0`
`%rsp`	`0x108`



`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)
```

Unary Instructions

The following instructions act on a single operand (register or memory):

Instruction	Effect	Description
`inc D`	$D \leftarrow D + 1$	Increment
`dec D`	$D \leftarrow D - 1$	Decrement
`neg D`	$D \leftarrow -D$	Negate
`not D`	$D \leftarrow \sim D$	Complement

`inc D` is reminiscent of C's **`++`** operator, and **`neg D`** is reminiscent of C's **`--`** operator.

Examples:

```
incq 16(%rax)
decq %rdx
not %rcx
```

Binary Instructions

The following instructions act on two operands (register or memory, but not both):

Instruction	Effect	Description
`add S, D`	$D \leftarrow D + S$	Add
`sub S, D`	$D \leftarrow D - S$	Subtract
`imul S, D`	$D \leftarrow D * S$	Multiply
`xor S, D`	$D \leftarrow D \wedge S$	Exclusive-or
`or S, D`	$D \leftarrow D \mid S$	Or
`and S, D`	$D \leftarrow D \& S$	And

Reading the syntax is a bit tricky — e.g., `subq %rax,%rdx` decrements `%rdx` by `%rax`, and can be read as "Subtract `%rax` from `%rdx`"

Examples:

```
addq %rcx,(%rax)
xorq $16,(%rax,%rdx,8)
subq %rdx,8(%rax)
```

Shift Instructions

The following instructions perform shifts. The first operand can be either an immediate value or the byte **`%cl`** (and only that register!)

Instruction	Effect	Description
`sal k, D`	$D \leftarrow D << k$	Left shift
`shl k, D`	$D \leftarrow D << k$	Left shift (same as `sal`)
`sar k, D`	$D \leftarrow D >>_A k$	Arithmetic right shift
`shr k, D`	$D \leftarrow D >>_L k$	Logical right shift

Technically, you could shift up to 255 with **`%cl`**, but the data width operation determines how many bits are shifted, and the high order bits are ignored. For example, if **`%cl`** has a value of 0xFF, then **`shlb`** shifts by 7 (ignoring the upper bits), **`shlw`** shifts by 15, **`shll`** would shift by 31, and **`shlq`** would shift by 63.

Examples:
```
shll $3,(%rax)
shr %cl,(%rax,%rdx,8)
sar $4,8(%rax)
```

Special Multiplication Instructions

Recall that multiplying two 64-bit numbers can produce a 128-bit result. The x86-64 instruction set supports 128-bit numbers with the "oct" (16-byte) size.

Instruction	Effect	Description
`imulq S`	R[`%rdx`]:R[`%rax`] $\leftarrow$ S × R[`%rax`]	Signed full multiply
`mulq S`	R[`%rdx`]:R[`%rax`] $\leftarrow$ S × R[`%rax`]	Unsigned full multiply

The **`imulq`** instruction has two forms. One, shown on **`slide 15`**, takes two operands and leaves the result in a single 64-bit register, truncating if necessary (and acts the same on signed and unsigned numbers). Example: **`imulq %rbx, %rcx`**

The second form (shown above) multiplies the source by **`%rax`**, and puts the product into the 128-bit **`%rdx`** (upper 64 bits) and **`%rax`** (lower 64 bits).

Example: **`mul %rdx`**
This multiplies **`%rdx`** by **`%rax`** and puts the result into the combined **`%rdx:%rax`** registers.

Multiplication Example

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

typedef unsigned __int128 uint128_t;

void store_uprod(uint128_t *dest,
                 uint64_t x, uint64_t y)
{
    *dest = x * (uint128_t) y;
}

int main()
{
    uint64_t x = 2000000000000; // 2 trillion
    uint64_t y = 3000000000000; // 3 trillion

    uint128_t z;
    store_uprod(&z,x,y);

    print_uint128(z); // see lect12 code
                      // for function definition
    return 0;
}
```

```
mov     %rsi,%rax
mul     %rdx
mov     %rax,(%rdi)
mov     %rdx,0x8(%rdi)
retq
```

Note: `%rdi` is 1st argument
`%rsi` is 2nd argument
`%rdx` is 3rd argument

Special Division Operations

Slide 11 did not list a division instruction or modulus instruction. There are single-operand divide instructions (shown below):

Instruction	Effect	Description
`cqto`	R[`%rdx`]:R[`%rax`] ← SignExtend(R[`%rax`])	Convert to oct word
`idivq S`	R[`%rdx`] ← R[`%rdx`]:R[`%rax`] mod S;	Signed divide
	R[`%rax`] ← R[`%rdx`]:R[`%rax`] ÷ S	
`divq S`	R[`%rdx`] ← R[`%rdx`]:R[`%rax`] mod S;	Unsigned divide
	R[`%rax`] ← R[`%rdx`]:R[`%rax`] ÷ S	

The dividend for the **`idivq`** and **`divq`** instructions is the 128-bit quantity in registers **`%rdx`** (high-order 64-bits) and **`%rax`** (low-order 64-bits). The divisor is the operand source. The quotient from the division is stored in **`%rax`**, and the remainder is stored in **`%rdx`**.

For most division, the dividend is just in **`%rax`**, and **`%rdx`** is either all zeros (for unsigned, or the sign bit of %rax (for signed arithmetic). The ctqo instruction can be used to accomplish this.

Division Example

```
void remdiv(long x, long y,
         long *qp, long *rp)
{
      long q = x / y;
      long r = x % y;
      *qp = q;
      *rp = r;
}
```

```
mov     %rdx,%r8      # copy qp
mov     %rdi,%rax     # Move x to lower 8 bytes of dividend
cqto                  # Sign-extend to upper 8 bytes of dividend
idiv    %rsi          # Divide by y
mov     %rax,(%r8)    # Store quotient at qp
mov     %rdx,(%rcx)   # Store remainder at rp
retq
```

Note: `%rdi` is 1st argument
`%rsi` is 2nd argument
`%rdx` is 3rd argument
`%rcx` is 4th argument

`gcc` is clever enough to see that only one division is needed!

3 minute break

Control

- So far, we have only been discussing "straight-line" code, where one instruction happens directly after the previous instruction.
- However, it is often necessary to perform one instruction or another instruction based on the logic in our programs, and assembly code gives us tools to do this.
- We can alter the flow of code using a "jump" instruction, which indicates that the next instruction will be somewhere else in the program (this is called a *branch*)
- We will start by discussing "condition codes" that are set when we do arithmetic (and other operations), and then we will talk about jump instructions to change control flow.

Condition Codes

- Besides the registers we have already discussed, the CPU has a separate set of single-bit *condition code* registers describing attributes of recent operations.
- We can use these registers (by testing them) to perform branches in the code.
- These are the most useful condition code registers:
  - **CF**: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
  - **ZF**: Zero flag. The most recent operation yielded zero.
  - **SF**: Sign flag. The most operation yielded a negative value.
  - **OF**: Overflow flag. The most recent operation caused a two's-complement overflow—either negative or positive.

Condition Codes: Examples

- **CF**: Carry flag. The most recent operation generated a carry out of the most significant b/t. Used to detect overflow for unsigned operations.
- **ZF**: Zero flag. The most recent operation yielded zero.
- **SF**: Sign flag. The most operation yielded a negative value.
- **OF**: Overflow flag. The most recent operation caused a two's-complement overflow—either negative or positive.

```
int a = 5;
int b = -5;
int t = a + b;
```

Which flag above would be set?

The **ZF** flag.

Condition Codes: Examples

- **CF**: Carry flag. The most recent operation generated a carry out of the most significant b/t. Used to detect overflow for unsigned operations.
- **ZF**: Zero flag. The most recent operation yielded zero.
- **SF**: Sign flag. The most operation yielded a negative value.
- **OF**: Overflow flag. The most recent operation caused a two's-complement overflow—either negative or positive.

```
int a = 5;
int b = -5;
int t = a + b;
```

Which flag above would be set?

The **ZF** flag.

```
int a = 5;
int b = -20;
int t = a + b;
```

Which flag above would be set?

The **SF** flag.

Condition Codes

- The **`leaq`** instruction does not set any condition codes (because it is intended for address computations), but the other arithmetic instructions we talked about do set them (**`inc`**, **`dec`**, **`neg`**, **`not`**, **`add`**, **`sub`**, **`imul`**, **`xor`**, **`or`**, **`and`**, **`shl`**, **`sar`**, **`shr`**, etc.)
- For logical operations (e.g., **`xor`**), the carry and overflow flags are set to 0.
- For shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to zero.
- **`inc`** and **`dec`** set the overflow and zero flags, but leave the carry flag unchanged (see here about a potential reason why).

`cmp` and `test`

- There are two types of instructions we can use that set the condition codes **without altering any other registers**, the **`cmp`** and **`test`** instructions:

Instruction		Based on	Description
`CMP`	S_1, S_2	$S_2 - S_1$	Compare
`cmpb`			Compare byte
`cmpw`			Compare word
`cmpl`			Compare double word
`cmpq`			Compare quad word
`TEST`	S_1, S_2	S_2 & S_1	Test
`testb`			Test byte
`testw`			Test word
`testl`			Test double word
`testq`			Test quad word

- By setting the condition codes, we can set up for a jump or other logic, based on some condition (e.g., whether a register has reached a certain value.
- Be careful! The operands for **`cmp`** are listed in reverse order! (**`cmp`** is based on the **`sub`** instruction)
- Often, we use **`testq %rax, %rax`** to see whether **`%rax`** is negative, zero, or positive.

Accessing the Condition Codes

- There are three common ways to use the condition codes:
  1. We can set a single byte to 0 or 1 depending on some combination of the condition codes.
  2. We can conditionally jump to some other part of the program.
  3. We can conditionally transfer data.

Accessing the Condition Codes

- There are three common ways to use the condition codes:
  1. We can set a single byte to 0 or 1 depending on some combination of the condition codes.
  2. We can conditionally jump to some other part of the program.
  3. We can conditionally transfer data.

Instruction	Synonym	Set Condition
`sete D`	`setz`	Equal / zero
`setne D`	`setnz`	Not equal / not zero
`sets D`		Negative
`setns D`		Nonnegative
`setg D`	`setnle`	Greater (signed >)
`setge D`	`setnl`	Greater or equal (signed >=)
`setl D`	`setnge`	Less (signed <)
`setle D`	`setng`	Less or equal (signed <=)
`seta D`	`setnbe`	Above (unsigned >)
`setae D`	`setnb`	Above or equal (unsigned >=)
`setb D`	`setnae`	Below (unsigned <)
`setbe D`	`setna`	Below or equal (unsigned <=)

Example: `a < b`

```
int comp(data_t a, data_t b)
a in %rdi, b in %rsi

comp:
cmpq %rsi, %rdi   # Compare a:b
setl %al          # Set low-order byte of
                  # %eax to 0 or 1
movzbl %al, %eax  # Clear rest of %eax
                  # (and rest of %rax)
ret
```

Accessing the Condition Codes

- There are three common ways to use the condition codes:
  1. We can set a single byte to 0 or 1 depending on some combination of the condition codes.
  2. We can conditionally jump to some other part of the program.
  3. We can conditionally transfer data.

Instruction	Synonym	Set Condition
`jmp` *Label*		Direct jump
`jmp` **Operand*		Indirect jump
`je` *Label*	`jz`	Equal / zero (ZF=1)
`jne` *Label*	`jnz`	Not equal / not zero (ZF=0)
`js` *Label*		Negative (SF=1)
`jns` *Label*		Nonnegative (SF=0)
`jg` *Label*	`jnle`	Greater (signed >) (SF=0 and SF=OF)
`jge` *Label*	`jnl`	Greater or equal (signed >=) (SF=OF)
`jl` *Label*	`jnge`	Less (signed <) (SF != OF)
`jle` *Label*	`jng`	Less or equal (signed <=) (ZF=1 or SF!=OF)
`ja` *Label*	`jnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`jae` *Label*	`jnb`	Above or equal (unsigned >=) (CF = 0)
`jb Label`	`jnae`	Below (unsigned <) (CF = 1)
`jbe Label`	`jna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

- Jump instructions jump to *labels* in assembly code, and those labels are changed to addresses (most often relative)
- **`jmp`** is an *unconditional* jump, meaning that the jump is always taken.
- Unconditional jumps can be direct *or* indirect
- Conditional jumps must be direct.

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

`gcc -c -Og while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

Set `%eax` to 0

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

`%rax: 0`

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

`gcc -c -Og while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

`%rax: 0`

compare `%eax` to `0x63 (99d)` by subtracting `%eax - 0x63`, setting the Sign Flag (SF) because the result is negative.

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

`gcc -c -Og while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

`%rax: 0`

`jle` is "jump less than or equal." The Sign Flag indicates that the result was negative (less than), so we jump to `0x7`.

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

`gcc -c -Og while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

`%rax: 1`

Add 1 to `%eax`

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

`%rax: 1`

Compare `%eax` to `0x63 (99d)` by subtracting `%eax - 0x63`. When `%rax` is 0, what flags change based on the the comparison? (We care about **Zero Flag**, **Sign Flag**, **Carry Flag**, and **Overflow Flag**): 0 - 99, so **SF** and **CF**

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

`gcc -c -Og while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

`%rax: 1`

Eventually, this will become positive (when `%eax` is 100), and the loop will end.

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

Could the compiler have done better with this loop?

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

~~`gcc -c -Og while_loop.c`~~

`gcc -c -O1 while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x64,%eax
   0x0000000000000005 <+5>:   sub    $0x1,%eax
   0x0000000000000008 <+8>:   jne    0x5 <loop+5>
   0x000000000000000a <+10>: repz retq
End of assembler dump.
```

Fewer lines, less jumping!

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

~~`gcc -c -Og while_loop.c`~~

`gcc -c -O1 while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x64,%eax
   0x0000000000000005 <+5>:   sub    $0x1,%eax
   0x0000000000000008 <+8>:   jne    0x5 <loop+5>
   0x000000000000000a <+10>:  repz retq
End of assembler dump.
```

Could we do better?

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   repz retq
End of assembler dump.
```

Compile to an object file:

~~`gcc -c -Og while_loop.c`~~

~~`gcc -c -O1 while_loop.c`~~

`gcc -c -O2 while_loop.c`

Sure! As the optimization level goes up, **`gcc`** gets smarter! The compiler realized that this loop is not doing anything, so it completely optimized it out!

Digging Deeper: Jump Instruction Encodings

- As we have mentioned before, assembly language is still one step higher than machine code.
- It is instructive in this case to look at the machine code for some jump instructions, just to see how the underlying machine is referencing where to jump.
- Remember, `%rip` is the *instruction pointer*, which has an address of the current instruction.
  - Well…kind of. On older x86 machines, when an instruction was executing, the first thing that happened was that `%rip` is changed to point to the next instruction. The instruction set has retained this behavior.
  - Jump instructions are often encoded to jump *relative to* `%rip`. Let's see what that means in practice…

Digging Deeper: Jump Instruction Encodings

- Let's look at our while loop again:

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

Run the objdump program:

```
objdump -d while_loop.o
```

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00          mov    $0x0,%eax
   5: eb 03                   jmp    a <loop+0xa>
   7: 83 c0 01                add    $0x1,%eax
   a: 83 f8 63                cmp    $0x63,%eax
   d: 7e f8                   jle    7 <loop+0x7>
   f: f3 c3                   repz retq
```

Digging Deeper: Jump Instruction Encodings

- Take the following function:

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

Run the objdump program:

```
objdump -d while_loop.o
```

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00          mov    $0x0,%eax
   5: eb 03                   jmp    a <loop+0xa>
   7: 83 c0 01                add    $0x1,%eax
   a: 83 f8 63                cmp    $0x63,%eax
   d: 7e f8                   jle    7 <loop+0x7>
   f: f3 c3                   repz retq
```

0-based addresses for each instruction (will be replaced with real addresses when a full program is created)

Digging Deeper: Jump Instruction Encodings

- Take the following function:

```
void loop()
{
     int i = 0;
     while (i < 100) {
         ++i;
     }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

Run the objdump program:

```
objdump -d while_loop.o
```

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00          mov    $0x0,%eax
   5: eb 03                   jmp    a <loop+0xa>
   7: 83 c0 01                add    $0x1,%eax
   a: 83 f8 63                cmp    $0x63,%eax
   d: 7e f8                   jle    7 <loop+0x7>
   f: f3 c3                   repz retq
```

Machine code for the instructions. Instructions are "variable length" — the **mov** instruction is 5 bytes, the tmp is 3 bytes, etc.

Digging Deeper: Jump Instruction Encodings

- Take the following function:

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

Run the objdump program:

```
objdump -d while_loop.o
```

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00          mov    $0x0,%eax
   5: eb 03                   jmp    a <loop+0xa>
   7: 83 c0 01                add    $0x1,%eax
   a: 83 f8 63                cmp    $0x63,%eax
   d: 7e f8                   jle    7 <loop+0x7>
   f: f3 c3                   repz retq
```

The `jmp` instruction. "`eb`" means that this is a `jmp`, and `03` is the number of instructions to jump, relative to `%rip`. When the instruction is executing, `%rip` is set to the next instruction (`7` in this case). So…`7 + 3` is 0xa, so this instruction jumps to `0xa`.

Digging Deeper: Jump Instruction Encodings

- Take the following function:

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

Run the objdump program:

```
objdump -d while_loop.o
```

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00          mov    $0x0,%eax
   5: eb 03                   jmp    a <loop+0xa>
   7: 83 c0 01                add    $0x1,%eax
   a: 83 f8 63                cmp    $0x63,%eax
   d: 7e f8                   jle    7 <loop+0x7>
   f: f3 c3                   repz retq
```

The **`cmp`** instruction. Notice that the 0x63 is embedded into the machine code, because it is an immediate value.

Digging Deeper: Jump Instruction Encodings

- Take the following function:

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

Run the objdump program:

```
objdump -d while_loop.o
```

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00          mov    $0x0,%eax
   5: eb 03                   jmp    a <loop+0xa>
   7: 83 c0 01                add    $0x1,%eax
   a: 83 f8 63                cmp    $0x63,%eax
   d: 7e f8                   jle    7 <loop+0x7>
   f: f3 c3                   repz retq
```

The `jle` instruction. "`7e`" means that this is a `jle` (jump if less than), and `f8` is the number of instructions to jump (in two's complement! So, it means -8), relative to `%rip`, which is at `0xf` when the instruction is running. So, `0xf - 8` is 0xa, so this instruction jumps to `0x7`.

Practice: Reverse-engineer Assembly to C

- Take the following function:

```
long test(long x, long y, long z) {
    long val = __________;
    if (__________) {
        if (__________)
            val = __________;
        else
            val = __________;
    } else if (__________)
        val = __________;
    return val;
}
```

```
# x in %rdi, y in %rsi, z in %rdx
test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4:
    rep; ret
```

Practice: Reverse-engineer Assembly to C

- Take the following function:

```
long test(long x, long y, long z) {
    long val = x + y + z;
    if (x < -3) {
        if (y < z)
            val = x * y;
        else
            val = y * z;
    } else if (x > 2)
        val = x * z;
    return val;
}
```

```
# x in %rdi, y in %rsi, z in %rdx
test:
        leaq (%rdi, %rsi), %rax
        addq %rdx, %rax
        cmpq $-3, %rdi
        jge .L2
        cmpq %rdx, %rsi
        jge .L3
        movq %rdi, %rax
        imulq %rsi, %rax
        ret
.L3:
        movq %rsi, %rax
        imulq %rdx, %rax
        ret
.L2:
        cmpq $2, %rdi
        jle .L4
        movq %rdi, %rax
        imulq %rdx, %rax
.L4:
        rep; ret
```

Conditional Moves

- The x86 processor provides a set of "conditional move" instructions that move memory based on the result of the condition codes, and that are completely analogous to the jump instructions:

Instruction	Synonym	Move Condition
`cmove S,R`	`cmovz`	Equal / zero (ZF=1)
`cmovne S,R`	`cmovnz`	Not equal / not zero (ZF=0)
`cmovs S,R`		Negative (SF=1)
`cmovns S,R`		Nonnegative (SF=0)
`cmovg S,R`	`cmovnle`	Greater (signed >) (SF=0 and SF=OF)
`cmovge S,R`	`cmovnl`	Greater or equal (signed >=) (SF=OF)
`cmovl S,R`	`cmovnge`	Less (signed <) (SF != OF)
`cmovle S,R`	`cmovng`	Less or equal (signed <=) (ZF=1 or SF!=OF)
`cmova S,R`	`cmovnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`cmovae S,R`	`cmovnb`	Above or equal (unsigned >=) (CF = 0)
`cmovb S,R`	`cmovnae`	Below (unsigned <) (CF = 1)
`cmovbe S,R`	`cmovna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

- With these instructions, we can sometimes eliminate branches, which are particularly inefficient on modern computer hardware.

Jumps -vs- Conditional Move

```
long absdiff(long x, long y)
{
    long result;
    if (x < y)
        result = y - x;
    else
        result = x - y;
    return result;
}
```

```
long cmovdiff(long x, long y)
{
    long rval = y-x;
    long eval = x-y;
    long ntest = x >= y;
    if (ntest) rval = eval;
    return rval;
}
```

```
# x in %rdi, y in %rsi
absdiff:
    cmpq %rsi, %rdi
    jge .L2
    movq %rsi, %rax
    subq %rdi, %rax
    ret
.L2:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
```

Which is faster?
Let's test!

```
# x in %rdi, y in %rsi
cmovdiff:
    movq %rsi, %rax
    subq %rdi, %rax
    movq %rdi, %rdx
    subq %rsi, %rdx
    cmpq %rsi, %rdi
    cmovge %rdx, %rax
    ret
```

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