x86-64 Programming III & The Stack

CSE 351 Autumn 2017

Instructor:

Justin Hsia

Teaching Assistants:

Lucas Wotton

Michael Zhang

Parker DeWilde

Ryan Wong

Sam Gehman

Sam Wolfson

Savanna Yee

Vinny Palaniappan

Administrivia

- Homework 2 due Friday (10/20)
- Lab 2 due next Friday (10/27)

- Section tomorrow on Assembly and GDB
  - Bring your laptops!

- Midterm: 10/30, 5pm in KNE 120
  - You will be provided a fresh reference sheet
  - You get 1 *handwritten*, double-sided cheat sheet (letter)
  - <u>Midterm Clobber Policy</u>: replace midterm score with score on midterm portion of the final if you "do better"

x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- **Loops**
- Switches

Expressing with Goto Code

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

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

- C allows `goto` as means of transferring control (`jump`)
  - Closer to assembly programming style
  - Generally considered bad coding style

Compiling Loops

C/Java code:

```
while ( sum != 0 ) {
    <loop body>
}
```

Assembly code:

```
loopTop:     testq %rax, %rax
             je    loopDone
             <loop body code>
             jmp   loopTop
loopDone:
```

- Other loops compiled similarly
  - Will show variations and complications in coming slides, but may skip a few examples in the interest of time
- Most important to consider:
  - When should conditionals be evaluated? (*while* vs. *do-while*)
  - How much jumping is involved?

Compiling Loops

C/Java code:

```
while ( Test ) {
    Body
}
```

Goto version

```
Loop: if ( !Test )  goto Exit;
      Body
      goto Loop;
Exit:
```

- What are the Goto versions of the following?
  - Do…while: Test and Body
  - For loop: Init, Test, Update, and Body

Compiling Loops

While Loop:

C:
```
while ( sum != 0 ) {
    <loop body>
}
```

x86-64:
```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

Do-while Loop:

C:
```
do {
    <loop body>
} while ( sum != 0 )
```

x86-64:
```
loopTop:
            <loop body code>
            testq  %rax, %rax
            jne    loopTop
loopDone:
```

While Loop (ver. 2):

C:
```
while ( sum != 0 ) {
    <loop body>
}
```

x86-64:
```
            testq %rax, %rax
            je    loopDone
loopTop:
            <loop body code>
            testq %rax, %rax
            jne   loopTop
loopDone:
```

For Loop → While Loop

For Version

```
for (Init; Test; Update)

      Body
```

While Version

```
Init;

while (Test) {

      Body

      Update;

}
```

Caveat: C and Java have `break` *and* `continue`

- *Conversion works fine for* `break`
  - *Jump to same label as loop exit condition*
- *But not* `continue`*: would skip doing Update, which it should do with for-loops*
  - *Introduce new label at Update*

x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- **Switches**

```c
long switch_ex
   (long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Switch Statement Example

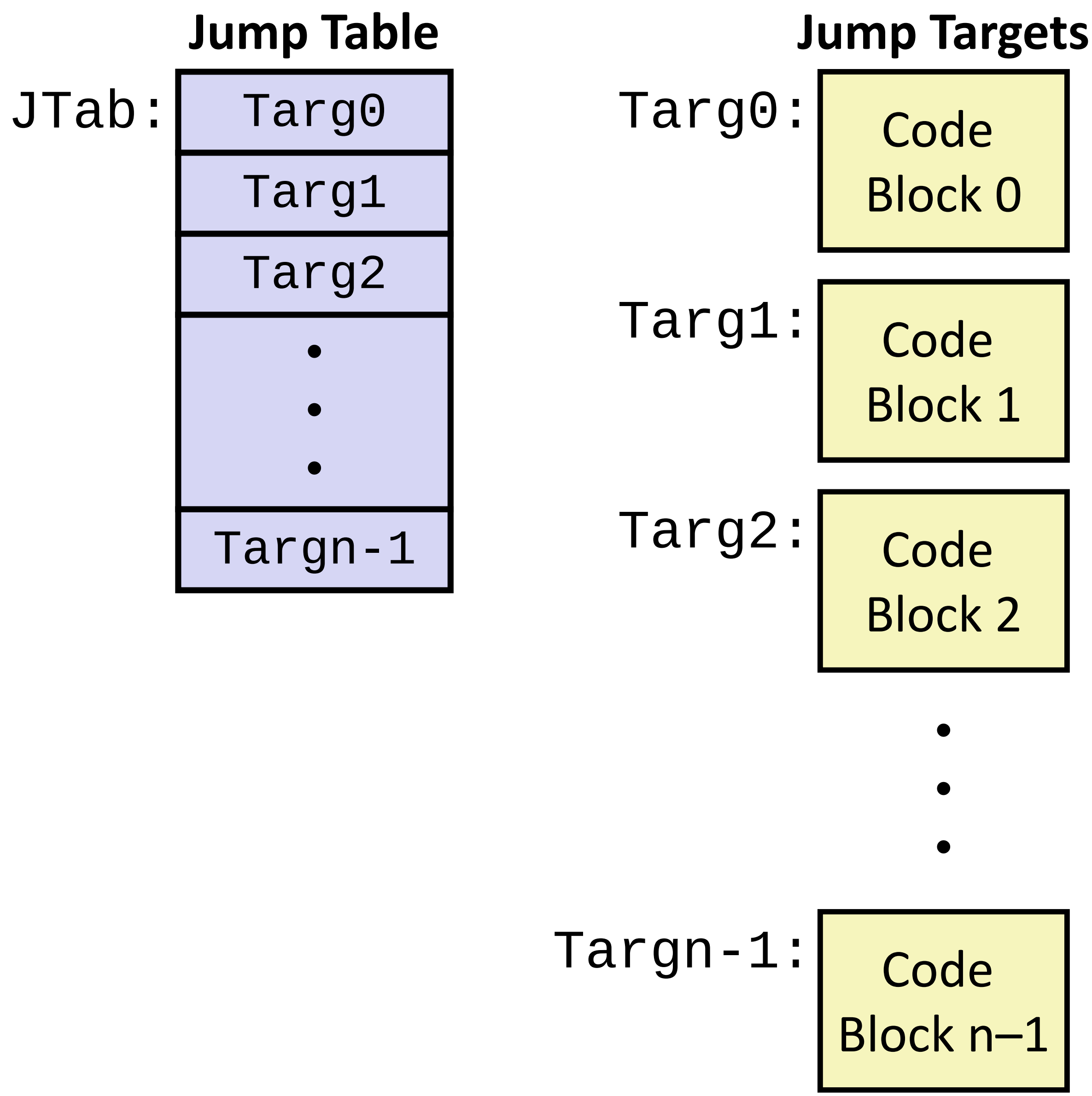
- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

- Implemented with:
  - *Jump table*
  - *Indirect jump instruction*

Jump Table Structure

Switch Form

```
switch (x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```



Approximate Translation

```
target = JTab[x];
goto target;
```

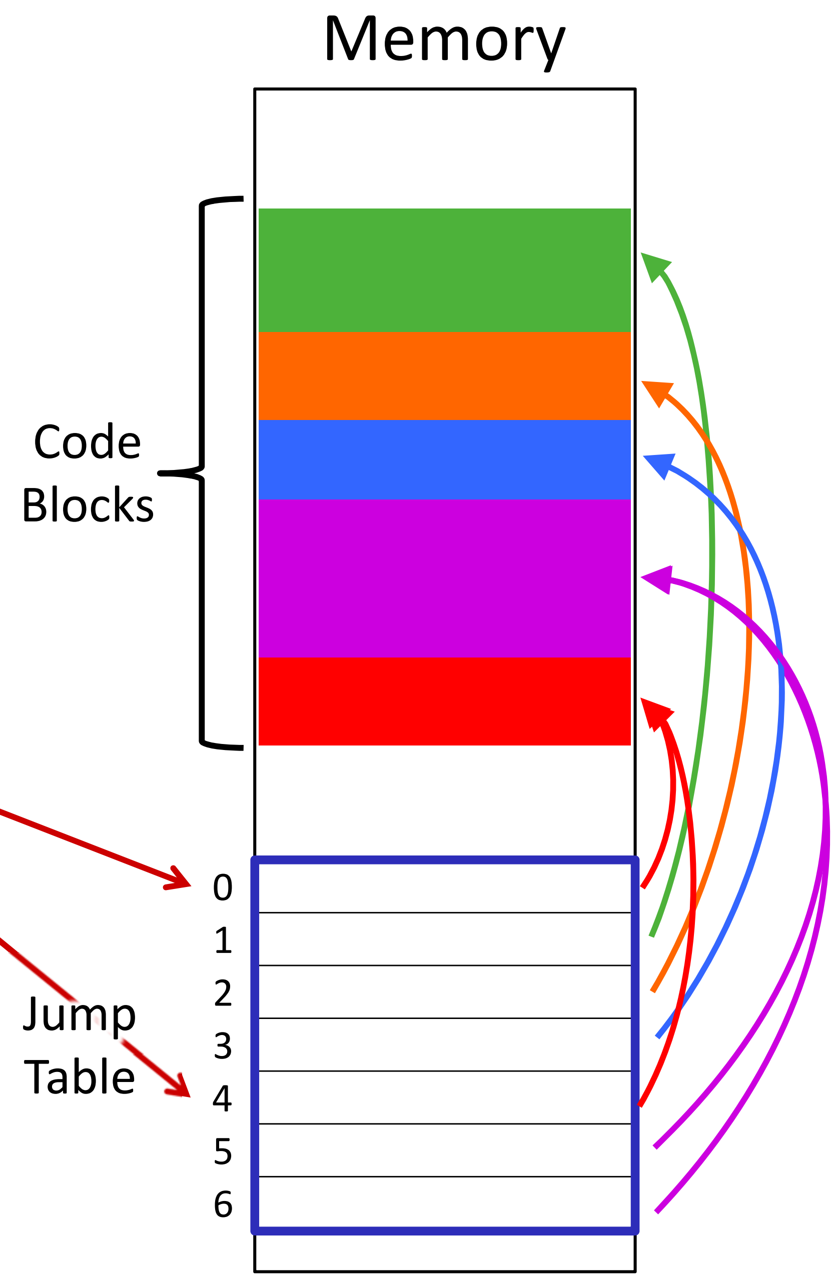
Jump Table Structure

C code:

```
switch (x) {
  case 1: <some code>
          break;
  case 2: <some code>
  case 3: <some code>
          break;
  case 5:
  case 6: <some code>
          break;
  default: <some code>
}
```

Use the jump table when x ≤ 6:

```
if (x <= 6)
  target = JTab[x];
  goto target;
else
  goto default;
```



Switch Statement Example

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

```
long switch_ex(long x, long y, long z)
{
    long w = 1;
    switch (x) {
      . . .
    }
    return w;
}
```

Note compiler chose to not initialize w

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi          # x:6
    ja      .L8               # default
    jmp     *.L4(,%rdi,8)     # jump table
```

Take a look!
https://godbolt.org/g/DnOmXb

jump above – unsigned > catches negative default cases

Switch Statement Example

```
long switch_ex(long x, long y, long z)
{
    long w = 1;
    switch (x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # default
    jmp     *.L4(,%rdi,8)   # jump table
```

Indirect jump

Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes (address)
  - Base address at `.L4`

- **Direct jump:** `jmp .L8`
  - Jump target is denoted by label `.L8`

- **Indirect jump:** `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective address `.L4 + x*8`
    - Only for $0 \leq x \leq 6$

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Jump Table

declaring data, not instructions

8-byte memory alignment

Jump table

```
.section .rodata
  .align 8
.L4:
  .quad   .L8 # x = 0
  .quad   .L3 # x = 1
  .quad   .L5 # x = 2
  .quad   .L9 # x = 3
  .quad   .L8 # x = 4
  .quad   .L7 # x = 5
  .quad   .L7 # x = 6
```

this data is 64-bits wide

```
switch(x) {
case 1:        // .L3
    w = y*z;
    break;
case 2:        // .L5
    w = y/z;
    /* Fall Through */
case 3:        // .L9
    w += z;
    break;
case 5:
case 6:        // .L7
    w -= z;
    break;
default:       // .L8
    w = 2;
}
```

Code Blocks (x == 1)

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

```
switch(x) {
  case 1:     // .L3
    w = y*z;
    break;
   . . .
}
```

```
.L3:
   movq    %rsi, %rax  # y
   imulq   %rdx, %rax  # y*z
   ret
```

Handling Fall-Through

```
long w = 1;
   . . .
switch (x) {
   . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
   . . .
}
```

```
case 2:
       w = y/z;
       goto merge;
```

```
case 3:
       w = 1;
```

```
merge:
       w += z;
```

More complicated choice than "just fall-through" forced by "migration" of `w = 1;`

- *Example compilation trade-off*

Code Blocks (x == 2, x == 3)

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

```
long w = 1;
   . . .
switch (x) {
   . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
   . . .
}
```

```
.L5:                       # Case 2:
    movq    %rsi, %rax     #   y in rax
    cqto                   #   Div prep
    idivq   %rcx           #   y/z
    jmp     .L6            #   goto merge
.L9:                       # Case 3:
    movl    $1, %eax       #   w = 1
.L6:                       # merge:
    addq    %rcx, %rax     #   w += z
    ret
```

Code Blocks (rest)

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

```
switch (x) {
   . . .
   case 5:  // .L7
   case 6:  // .L7
     w -= z;
     break;
   default: // .L8
     w = 2;
}
```

```
.L7:                     # Case 5,6:
  movl   $1, %eax        #   w = 1
  subq   %rdx, %rax      #   w -= z
  ret
.L8:                     # Default:
  movl   $2, %eax        #   2
  ret
```

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

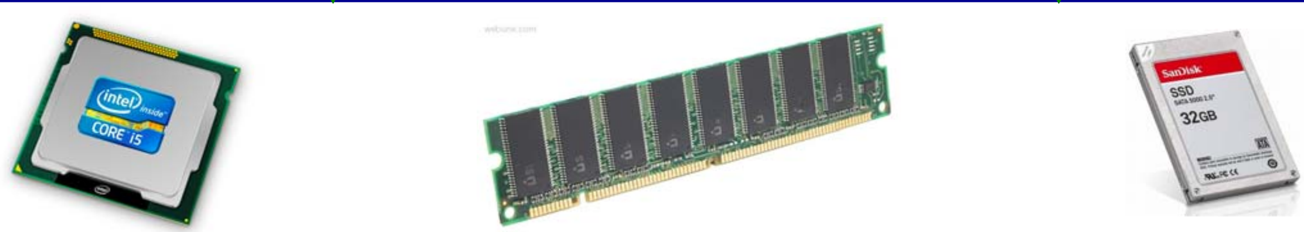
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 10

OS X Yosemite

Computer system:

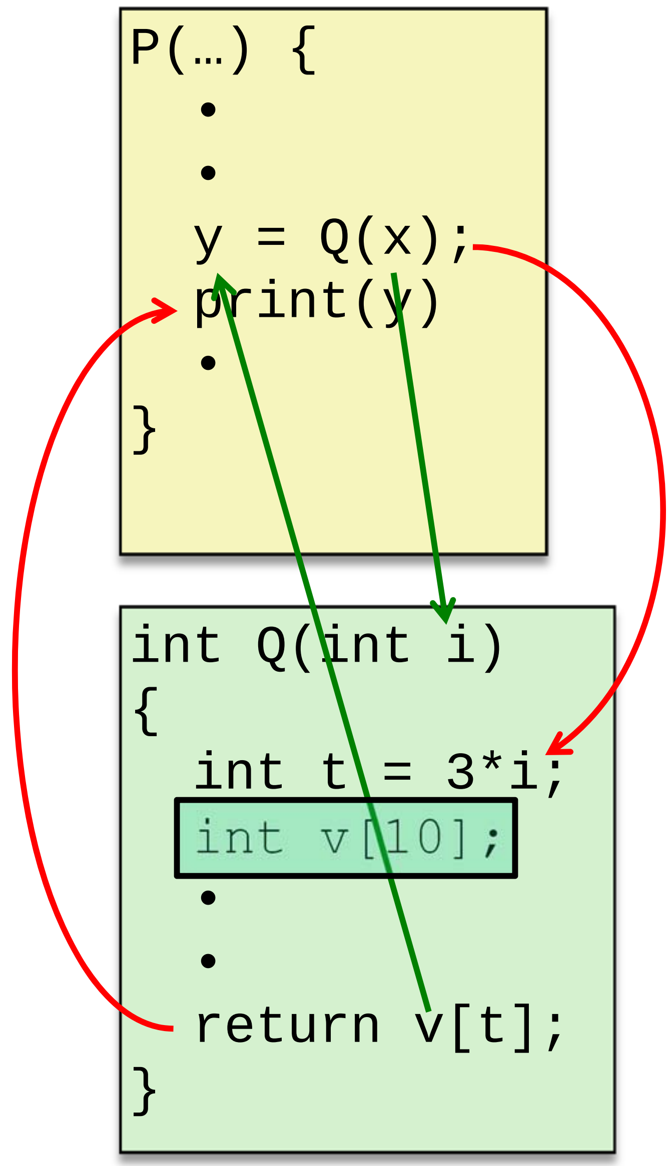


- Memory & data
- Integers & floats
- x86 assembly
- **Procedures & stacks**
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Mechanisms required for *procedures*

1) Passing control
   - To beginning of procedure code
   - Back to return point
2) Passing data
   - Procedure arguments
   - Return value
3) Memory management
   - Allocate during procedure execution
   - Deallocate upon return

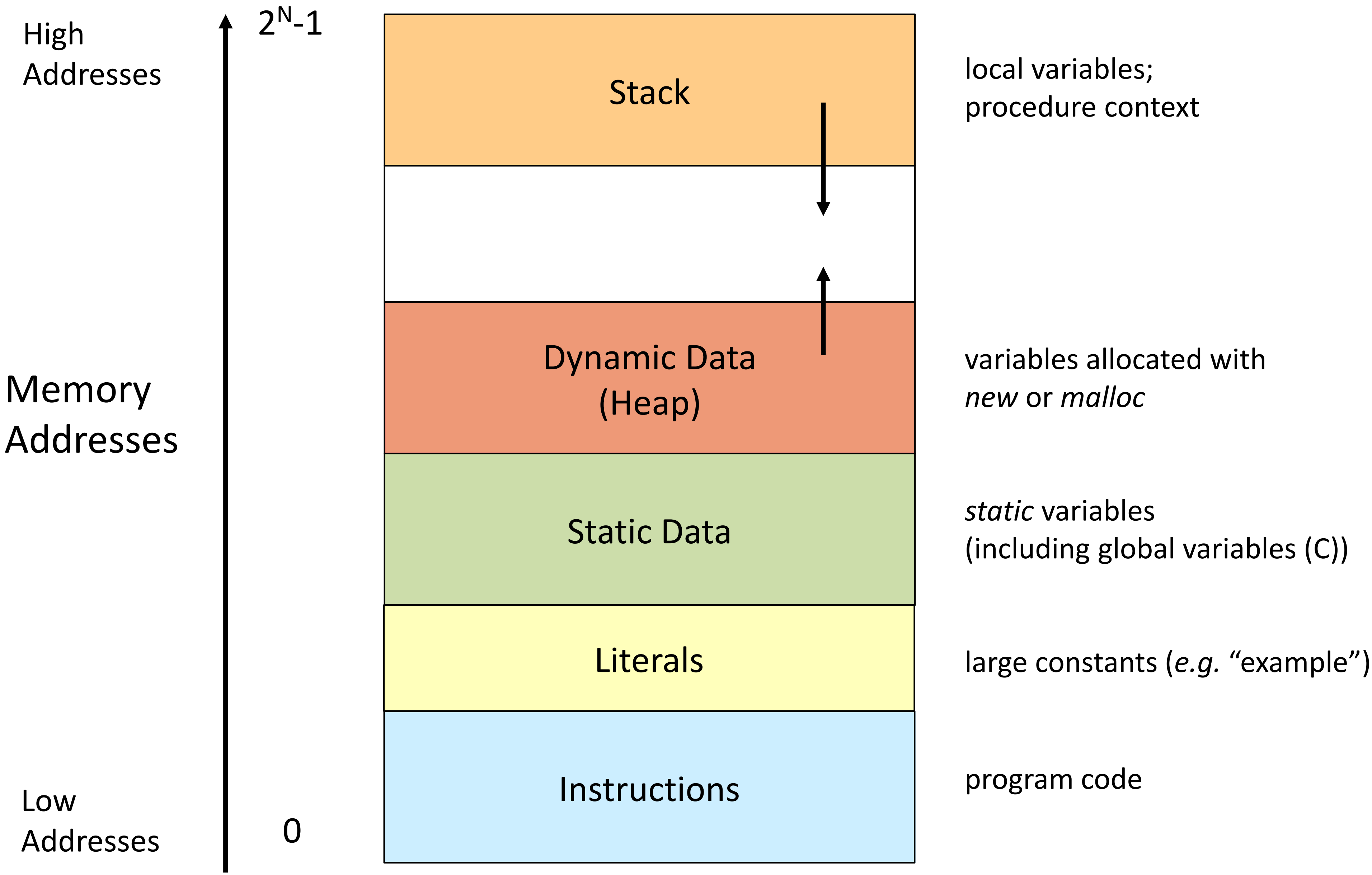
- All implemented with machine instructions!
   - An x86-64 procedure uses only those mechanisms required for that procedure



Procedures

- **Stack Structure**
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Simplified Memory Layout



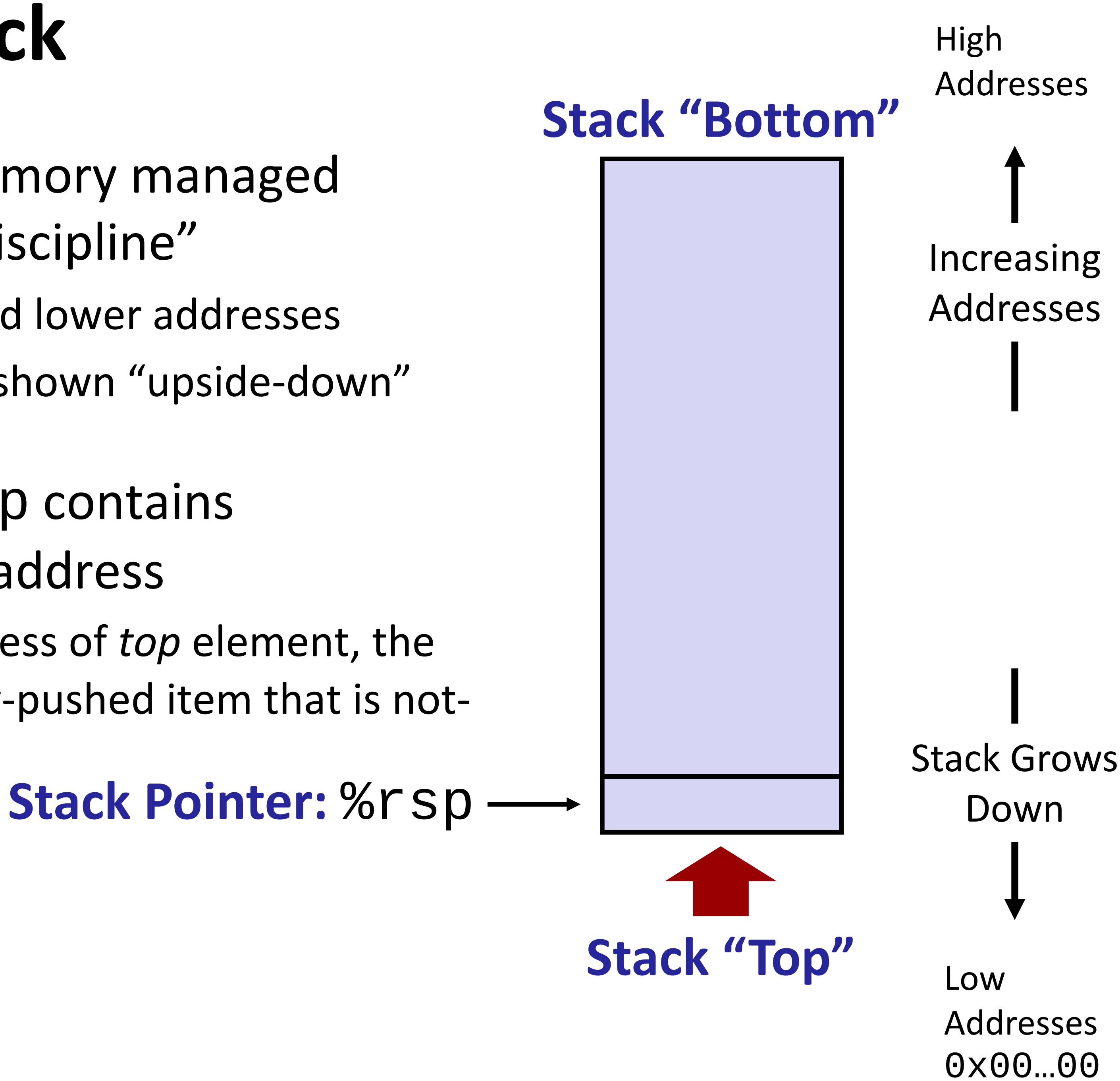
segmentation faults?

Memory Permissions

writable; not executable	Stack	Managed "automatically" (by compiler)
writable; not executable	Dynamic Data (Heap)	Managed by programmer
writable; not executable	Static Data	Initialized when process starts
read-only; not executable	Literals	Initialized when process starts
read-only; executable	Instructions	Initialized when process starts

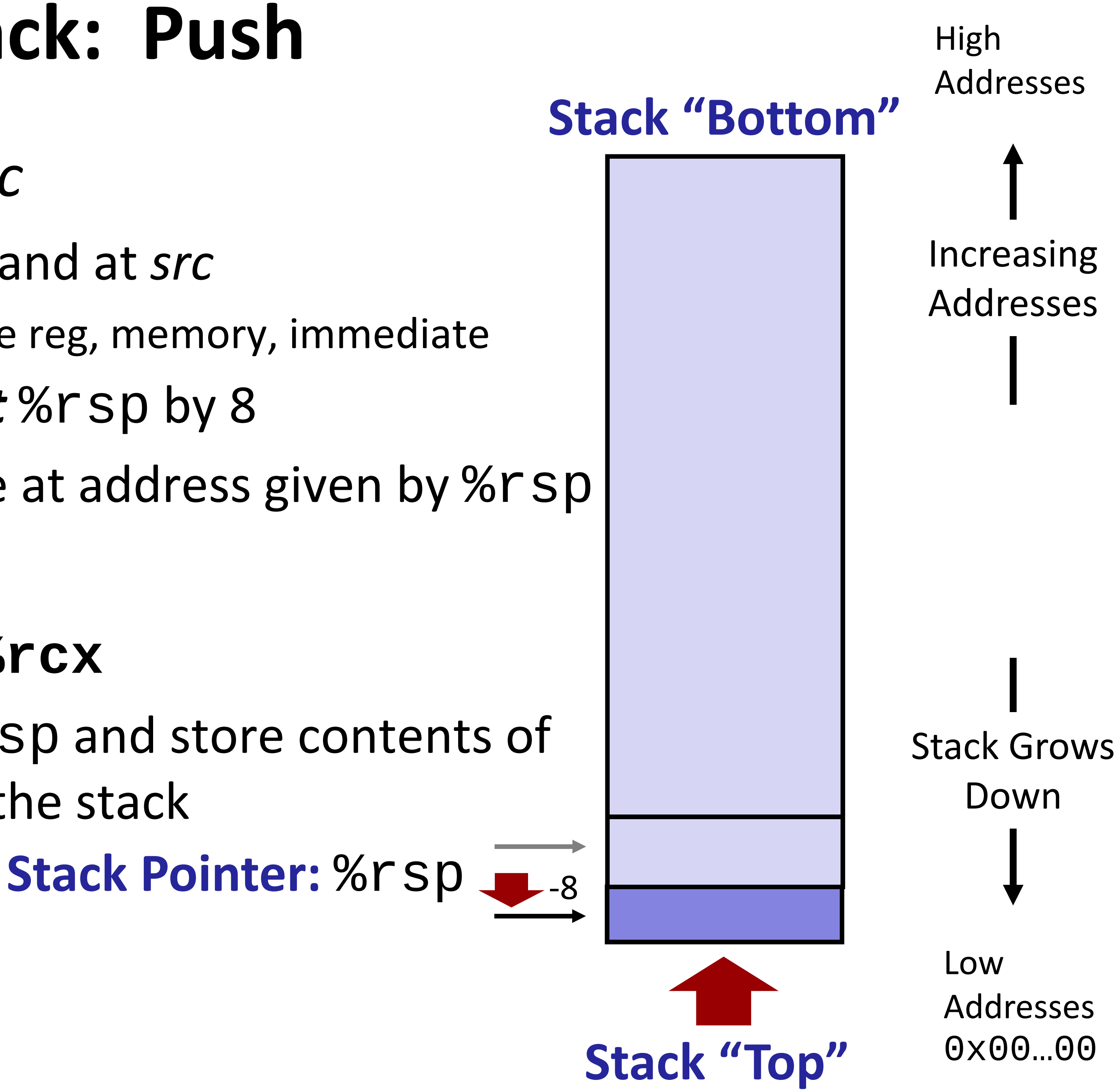
x86-64 Stack

- Region of memory managed with stack "discipline"
  - Grows toward lower addresses
  - Customarily shown "upside-down"
- Register `%rsp` contains *lowest* stack address
  - `%rsp` = address of *top* element, the most-recently-pushed item that is not-yet-popped



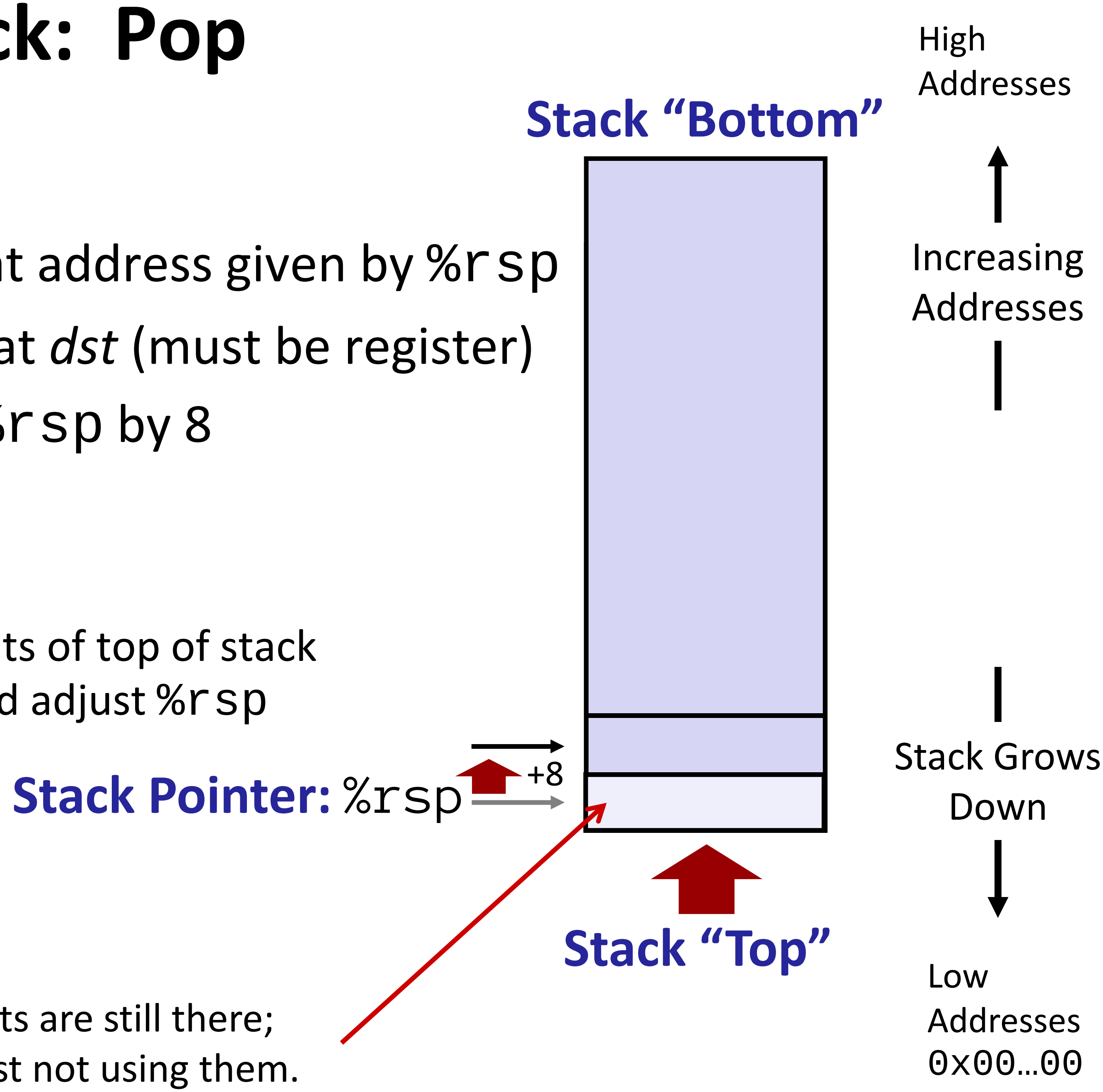
x86-64 Stack: Push

- `pushq` *src*
  - Fetch operand at *src*
    - *Src* can be reg, memory, immediate
  - ***Decrement*** `%rsp` by 8
  - Store value at address given by `%rsp`
- Example:
  - **`pushq %rcx`**
  - Adjust `%rsp` and store contents of `%rcx` on the stack



x86-64 Stack: Pop

- popq *dst*
  - Load value at address given by `%rsp`
  - Store value at *dst* (must be register)
  - ***Increment*** `%rsp` by 8
- Example:
  - **`popq %rcx`**
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`



Those bits are still there; we're just not using them.