Machine-Level Programming II

15-213/15-513: Introduction to Computer Systems
4th Lecture, September 4, 2025

Announcements

- **Recordings**
  - Created and will be updating a page on Canvas with old recordings
- **Activities**
  - There are optional activities to support understanding of the machine programming lectures
- **Deadlines**
  - Written assignments are due by 11:59pm + 12 hours of grace time (no penalty)
- **Bomb Lab**
  - Releases tonight

Today

- **Review of a few tricky bits from last time**
- Loops
- Switch statements
- Procedures

Reminder: Machine Instructions

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:  48 89 03
```

- **C**
  - Store value **t** where designated by **dest**
- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - **t:** Register **%rax**
    - **dest:** Register **%rbx**
    - ***dest:** Memory **M[%rbx]**
- **Machine**
  - 3 bytes at address **0x40059e**
  - Compact representation of the assembly instruction
  - (Relatively) easy for hardware to interpret

Reminder: Machine Instructions

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e: 48 89 03
```

```
0100 1 0 0 0  10001011  00 000 011
REX  W R X B  MOV r->x  Mod R   M
```

- **C**
  - Store value **t** where designated by **dest**
- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - **t:** Register **%rax**
    - **dest:** Register **%rbx**
    - ***dest:** Memory **M[%rbx]**
- **Machine**
  - 3 bytes at address **0x40059e**
  - Compact representation of the assembly instruction
  - (Relatively) easy for hardware to interpret

Assembly Syntax

Intel versus AT&T

In this class we will be using the AT&T syntax

Feature	AT&T Syntax	Intel Syntax
Operand Order	source, destination	destination, source
Register Prefix	% (e.g., %eax)	None (e.g., eax)
Immediate Value Prefix	$ (e.g., $10)	None (e.g., 10)
Memory Addressing	displacement(base, index, scale)	[base + index*scale + displacement]
Operand Size Suffix	b, w, l, q (e.g., movl)	Inferred or ptr prefixes (e.g., dword ptr)

Reminder: Address Modes

- **Most General Form**

  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases**

  (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]

  D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]

  (Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]

Memory operands and LEA

- In most instructions, a memory operand accesses memory

Assembly	C equivalent
mov 6(%rbx,%rdi,8), %ax	ax = *(rbx + rdi*8 + 6)
add 6(%rbx,%rdi,8), %ax	ax += *(rbx + rdi*8 + 6)
xor %ax, 6(%rbx,%rdi,8)	*(rbx + rdi*8 + 6) ^= ax

- LEA is special: it *doesn't* access memory

Assembly	C equivalent
lea 6(%rbx,%rdi,8), %rax	rax = rbx + rdi*8 + 6

Why use LEA?

- **CPU designers' intended use: calculate a pointer to an object**
  - An array element, perhaps
  - For instance, to pass just one array element to another function

Assembly	C equivalent
lea (%rbx,%rdi,8), %rax	**rax = &rbx[rdi]**

- **Compiler authors like to use it for ordinary arithmetic**
  - It can do complex calculations in one instruction
  - It's one of the only three-operand instructions the x86 has
  - It doesn't touch the condition codes (we'll come back to this)

Assembly	C equivalent
lea (%rbx,%rbx,2), %rax	**rax = rbx * 3**

Which numbers are pointers?

- They aren't labeled
- You have to figure it out from context

```
(gdb) info registers
rax      0x40057d         4195709
rbx      0x0              0
rcx      0x4005e0         4195808
rdx      0x7fffffffdc28   140737488346152
rsi      0x7fffffffdc18   140737488346136
rdi      0x1              1
rbp      0x0              0x0
rsp      0x7fffffffdb38   0x7fffffffdb38
r8       0x7ffff7dd5e80   140737351868032
r9       0x0              0
r10      0x7fffffffd7c0   140737488345024
r11      0x7ffff7a2f460   140737348039776
r12      0x400490         4195472
r13      0x7fffffffdc10   140737488346128
r14      0x0              0
r15      0x0              0
rip      0x40057d         0x40057d
```

Which numbers are pointers?

- They aren't labeled
- You have to figure it out from context
- %rsp and %rip always hold pointers

```
(gdb) info registers
rax      0x40057d          4195709
rbx      0x0               0
rcx      0x4005e0          4195808
rdx      0x7fffffffdc28    140737488346152
rsi      0x7fffffffdc18    140737488346136
rdi      0x1               1
rbp      0x0               0x0
rsp      0x7fffffffdb38    0x7fffffffdb38
r8       0x7ffff7dd5e80    140737351868032
r9       0x0               0
r10      0x7fffffffd7c0    140737488345024
r11      0x7ffff7a2f460    140737348039776
r12      0x400490          4195472
r13      0x7fffffffdc10    140737488346128
r14      0x0               0
r15      0x0               0
rip      0x40057d          0x40057d
```

Which numbers are pointers?

- They aren't labeled
- You have to figure it out from context

- %rsp and %rip always hold pointers
  - Register values that are "close" to %rsp or %rip are *probably* also pointers

```
(gdb) info registers
rax      0x40057d         4195709
rbx      0x0              0
rcx      0x4005e0         4195808
rdx      0x7fffffffdc28   140737488346152
rsi      0x7fffffffdc18   140737488346136
rdi      0x1              1
rbp      0x0              0x0
rsp      0x7fffffffdb38   0x7fffffffdb38
r8       0x7ffff7dd5e80   140737351868032
r9       0x0              0
r10      0x7fffffffd7c0   140737488345024
r11      0x7ffff7a2f460   140737348039776
r12      0x400490         4195472
r13      0x7fffffffdc10   140737488346128
r14      0x0              0
r15      0x0              0
rip      0x40057d         0x40057d
```

Which numbers are pointers?

- If a register is being *used* as a pointer…

```
Dump of assembler code for function main:
=> 0x40057d <+0>:   sub    $0x8,%rsp
   0x400581 <+4>:   mov    (%rsi),%rsi
   0x400584 <+7>:   mov    $0x400670,%edi
   0x400589 <+12>:  mov    $0x0,%eax
   0x40058e <+17>:  call   0x400460
```

Which numbers are pointers?

- **If a register is being *used* as a pointer…**
  - mov (%rsi), %rsi
  - …Then its value is *expected* to be a pointer.
    - There might be a bug that makes its value incorrect.

```
Dump of assembler code for function main:
=> 0x40057d <+0>:   sub    $0x8,%rsp
   0x400581 <+4>:   mov    (%rsi),%rsi
   0x400584 <+7>:   mov    $0x400670,%edi
   0x400589 <+12>:  mov    $0x0,%eax
   0x40058e <+17>:  call   0x400460
```

Which numbers are pointers?

- **If a register is being *used* as a pointer…**
  - mov (%rsi), %rsi
  - …Then its value is *expected* to be a pointer.
    - There might be a bug that makes its value incorrect.

```
Dump of assembler code for function main:
=> 0x40057d <+0>:   sub    $0x8,%rsp
   0x400581 <+4>:   mov    (%rsi),%rsi
   0x400584 <+7>:   mov    $0x400670,%edi
   0x400589 <+12>:  mov    $0x0,%eax
   0x40058e <+17>:  call   0x400460
```

- **Not as obvious with complicated address "modes"**
  - (%rsi, %rbx) – *One* of these is a pointer, we don't know which.
  - (%rsi, %rbx, 2) – %rsi is a pointer, %rbx isn't (why?)
  - 0x400570(, %rbx, 2) – 0x400570 is a pointer, %rbx isn't (why?)
  - lea (anything), %rax – (anything) *may or may not* be a pointer

Conditional reminder slide

- **Condition codes**
  - Usually set by cmp or test
  - Other instructions may also implicitly set condition codes
- **Types of Condition codes**
  - Zero Flag (ZF) , Carry Flag(CF), Sign Flag (SF).Overflow Flag (OF)
  - Exist inside a special FLAGS register
- **Jumps**
  - Two kinds of jump instructions
  - Unconditional jump - jmp
  - Conditional j**X**

Today

- Review of a few tricky bits from yesterday
- **Loops**
- Switch statements
- Procedures

Loops

- What are the parts of a loop?

Assembly has Goto

- **Loops need to be conceptualized with goto and conditionals**

- **What distinguishes a loop's goto from an if's goto?**
  - Since loops repeat code, which way will the goto go?

"Do-While" Loop Example

C Code

```
long pcount_do
   (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
   (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

- Count number of 1's in argument `x` ("popcount")
- Use conditional branch to either continue looping or to exit loop

"Do-While" Loop Compilation

Goto Version

```
long pcount_goto
   (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
        movl    $0, %eax            #  result = 0
        .L2:                        # loop:
           movq    %rdi, %rdx
           andl    $1, %edx         #  t = x & 0x1
           addq    %rdx, %rax       #  result += t
           shrq    %rdi             #  x >>= 1
           jne     .L2              #  if (x) goto
        loop
           rep; ret
```

General "Do-While" Translation

C Code

```
do
   Body
   while (Test);
```

Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

- **Body:**

```
{
   Statement1;
   Statement2;
      …
   Statementn;
}
```

General "While" Translation #1

- "Jump-to-middle" translation
- Used with –Og

While version

```
while (Test)
  Body
```

Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

While Loop Example #1

C Code

```
long pcount_while
   (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Jump to Middle

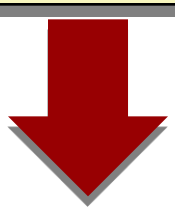
```
long pcount_goto_jtm
   (unsigned long x) {
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

General "While" Translation #2

While version

```
while (Test)
  Body
```



Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

- "Do-while" conversion
- Used with –O1

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

While Loop Example #2

C Code

```
long pcount_while
   (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Do-While Version

```
long pcount_goto_dw
   (unsigned long x) {
  long result = 0;
  if (!x) goto done;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
 done:
  return result;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

"For" Loop Form

General Form

```
for (Init; Test; Update)
    Body
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

"For" Loop → While Loop

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update;
}
```

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
       (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

"For" Loop Do-While Conversion

C Code

```
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  if (!(i < WSIZE))
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  i++;
  if (i < WSIZE)
    goto loop;
 done:
  return result;
}
```

Ini

! Test

Body

Update

Test

- Initial test can be optimized away

Today

- Review of a few tricky bits from yesterday
- Loops
- **Switch statements**
- Procedures

```
long switch_eg
   (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

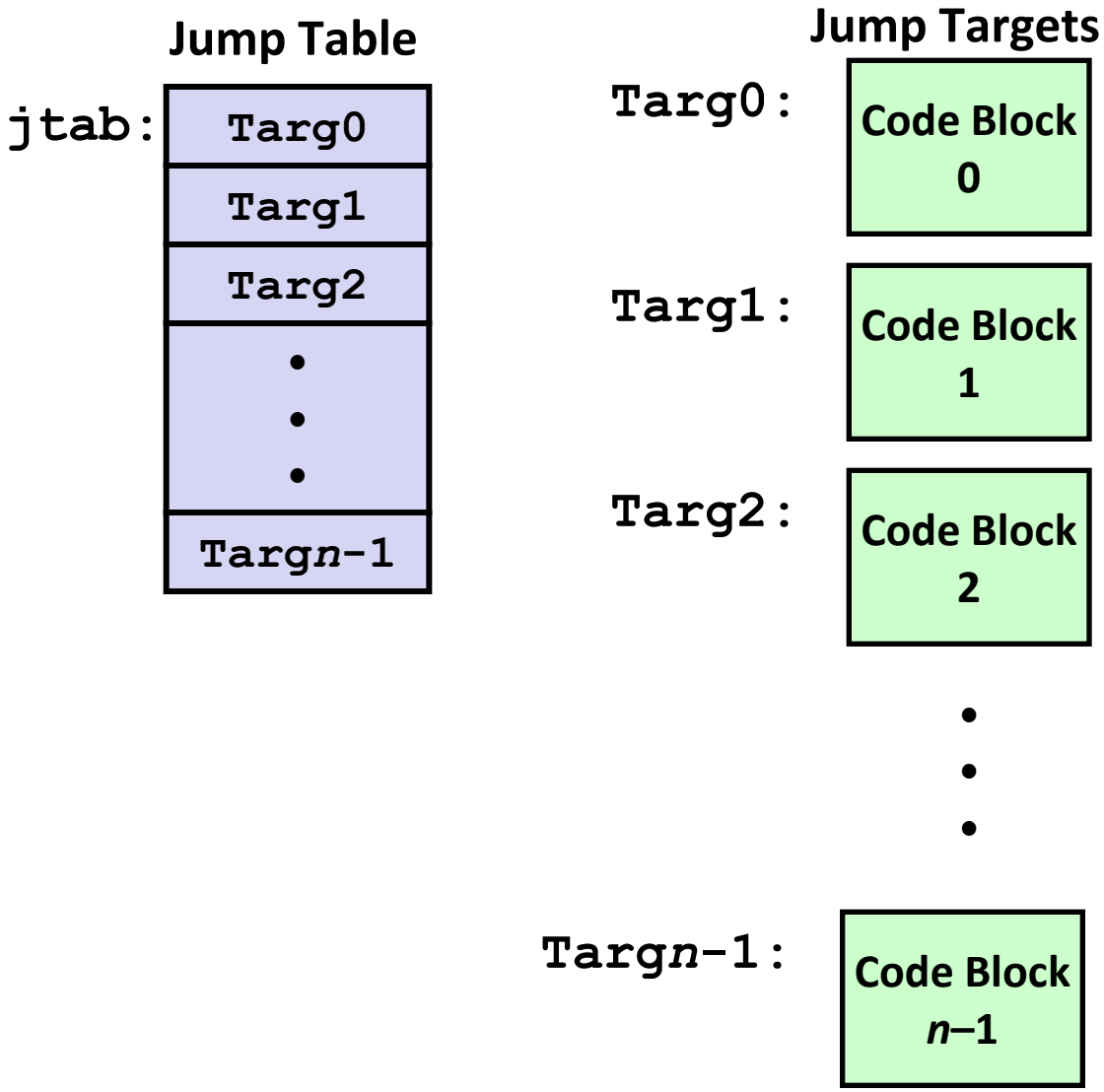
Jump Table Structure

Switch Form

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

Translation (Extended C)

```
goto *JTab[x];
```

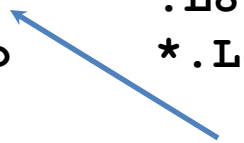


Switch Statement Example

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

Setup:

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



What range of values takes default?

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Note that **w** not initialized here

Switch Statement Example

```
long switch_eg(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
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Indirect jump

Assembly Setup Explanation

Table Structure

- Each target requires 8 bytes
- Base address at `.L4`

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
```

Jumping

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

- **Indirect:** `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 $0 \le x \le 6$

Jump Table

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(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)

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

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

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Handling Fall-Through

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

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

```
case 3:
    w = 1;
merge:
    w += z;
```

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

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

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

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Code Blocks (x == 5, x == 6, default)

```
  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
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Finding Jump Table in Binary

```
 00000000004005e0 <switch_eg>:
4005e0:       48 89 d1                mov    %rdx,%rcx
4005e3:       48 83 ff 06             cmp    $0x6,%rdi
4005e7:       77 2b                   ja     400614 <switch_eg+0x34>
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
4005f0:       48 89 f0                mov    %rsi,%rax
4005f3:       48 0f af c2             imul   %rdx,%rax
4005f7:       c3                      retq
4005f8:       48 89 f0                mov    %rsi,%rax
4005fb:       48 99                   cqto
4005fd:       48 f7 f9                idiv   %rcx
400600:       eb 05                   jmp    400607 <switch_eg+0x27>
400602:       b8 01 00 00 00          mov    $0x1,%eax
400607:       48 01 c8                add    %rcx,%rax
40060a:       c3                      retq
40060b:       b8 01 00 00 00          mov    $0x1,%eax
400610:       48 29 d0                sub    %rdx,%rax
400613:       c3                      retq
400614:       b8 02 00 00 00          mov    $0x2,%eax
400619:       c3                      retq
```

Finding Jump Table in Binary (cont.)

```
 00000000004005e0 <switch_eg>:
. . .
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
. . .
```

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
(gdb)
```

Finding Jump Table in Binary (cont.)

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
```

```
  . . .
  4005f0:       48 89 f0                mov    %rsi,%rax
  4005f3:       48 0f af c2             imul   %rdx,%rax
  4005f7:       c3                      retq
  4005f8:       48 89 f0                mov    %rsi,%rax
  4005fb:       48 99                   cqto
  4005fd:       48 f7 f9                idiv   %rcx
  400600:       eb 05                   jmp    400607 <switch_eg+0x27>
  400602:       b8 01 00 00 00          mov    $0x1,%eax
  400607:       48 01 c8                add    %rcx,%rax
  40060a:       c3                      retq
  40060b:       b8 01 00 00 00          mov    $0x1,%eax
  400610:       48 29 d0                sub    %rdx,%rax
  400613:       c3                      retq
  400614:       b8 02 00 00 00          mov    $0x2,%eax
  400619:       c3                      retq
```

Jump Tables (alternate version)

- **Recently the compiler developers worked out how to make the jump tables smaller**

- **Instead of addresses, jump tables can contain 4-byte offsets**
  - Then rip + jump table address + offset => location of target

- **Content will be covered further in recitation**

Quiz

- Please complete the quiz on Cavnas
- https://canvas.cmu.edu/courses/49105/quizzes/150037

Transition

We need other control flow beyond conditionals and loops.

Good design has us reused computation via functions.

Mechanisms in Procedures

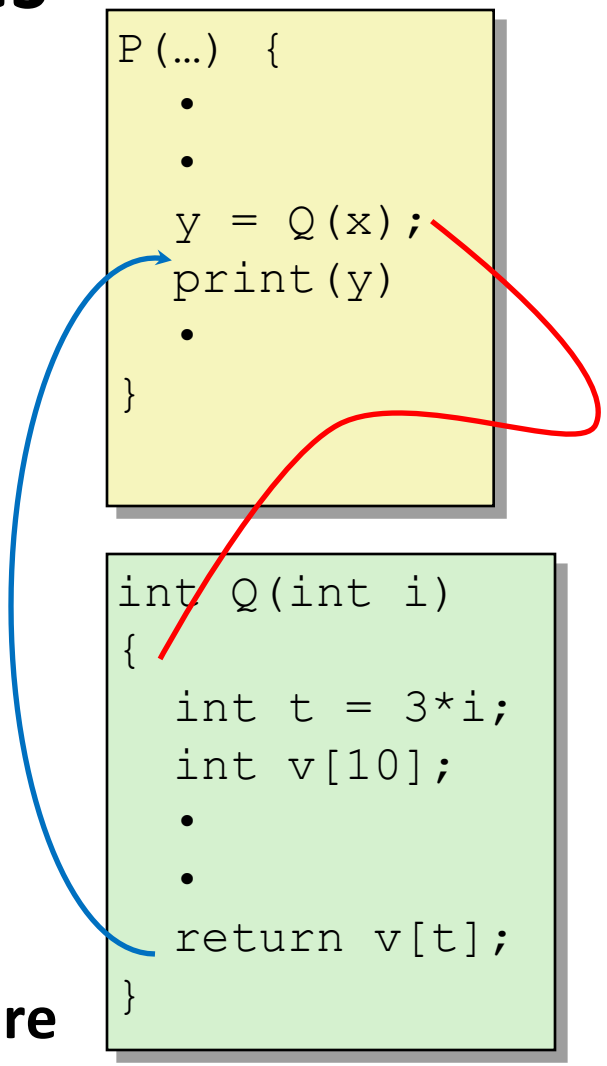
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**

```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

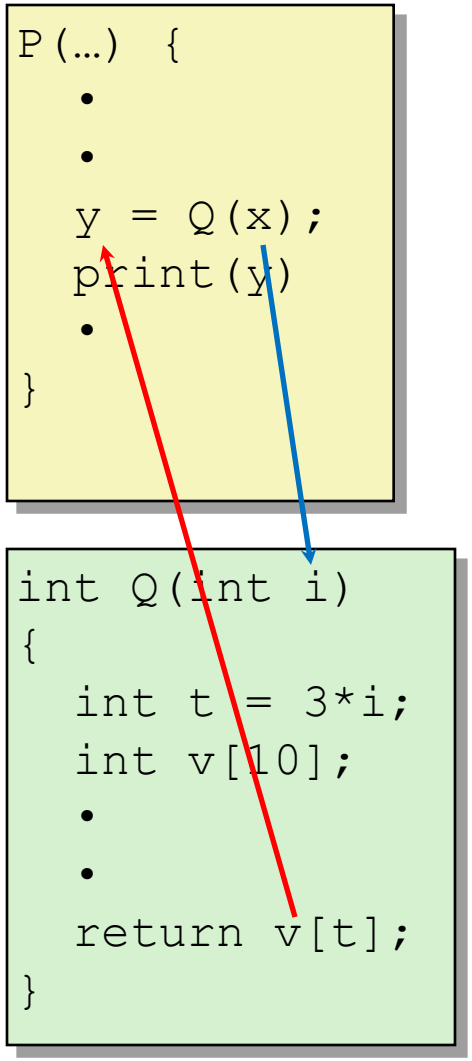
Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**



Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**



Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**

```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

Mechanisms in Procedures

Machine instructions implement the mechanisms, but the choices are determined by designers. These choices make up the **Application Binary Interface (ABI)**.

- Deallocate upon return

- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**

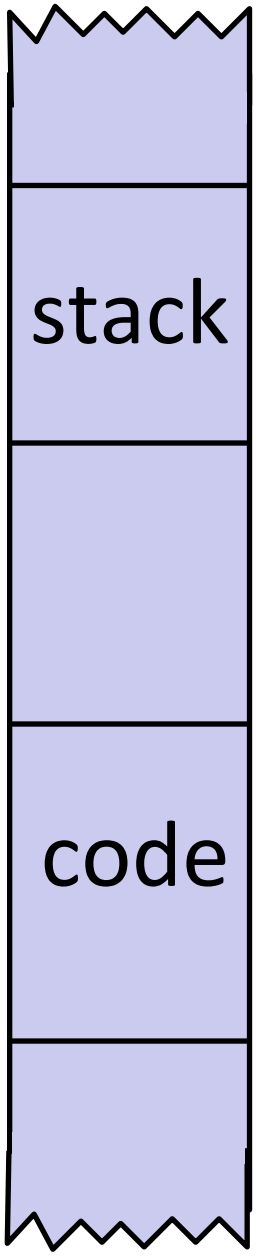
```
P( ) {
   int v[10];
   •
   •
   return v[t];
}
```

Today

- **Procedures**
  - **Stack Structure**
  - **Calling Conventions**
    - **Passing control**

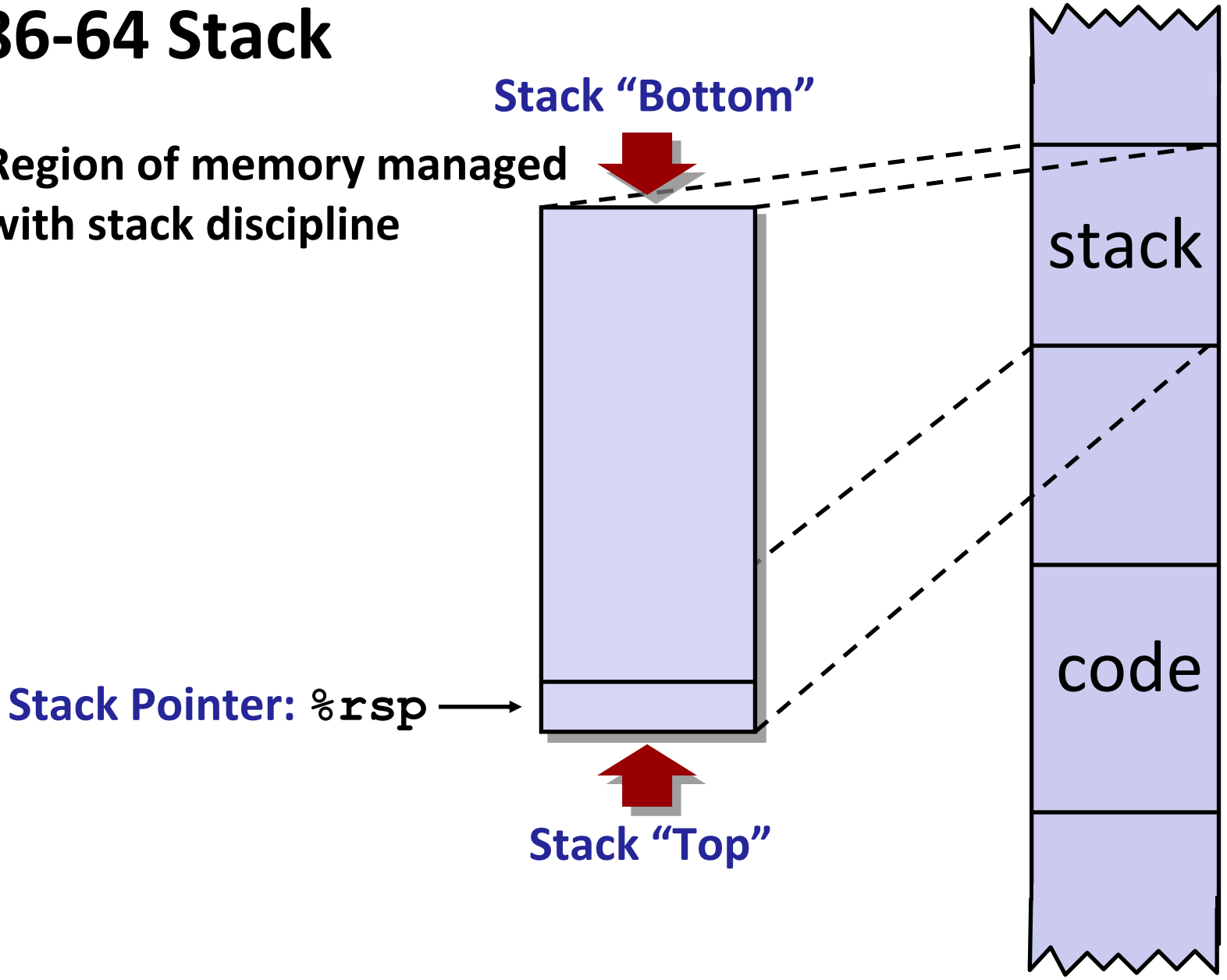
x86-64 Stack

- **Region of memory managed with stack discipline**
  - Memory viewed as array of bytes.
  - Different regions have different purposes.
  - (Like ABI, a policy decision)



x86-64 Stack

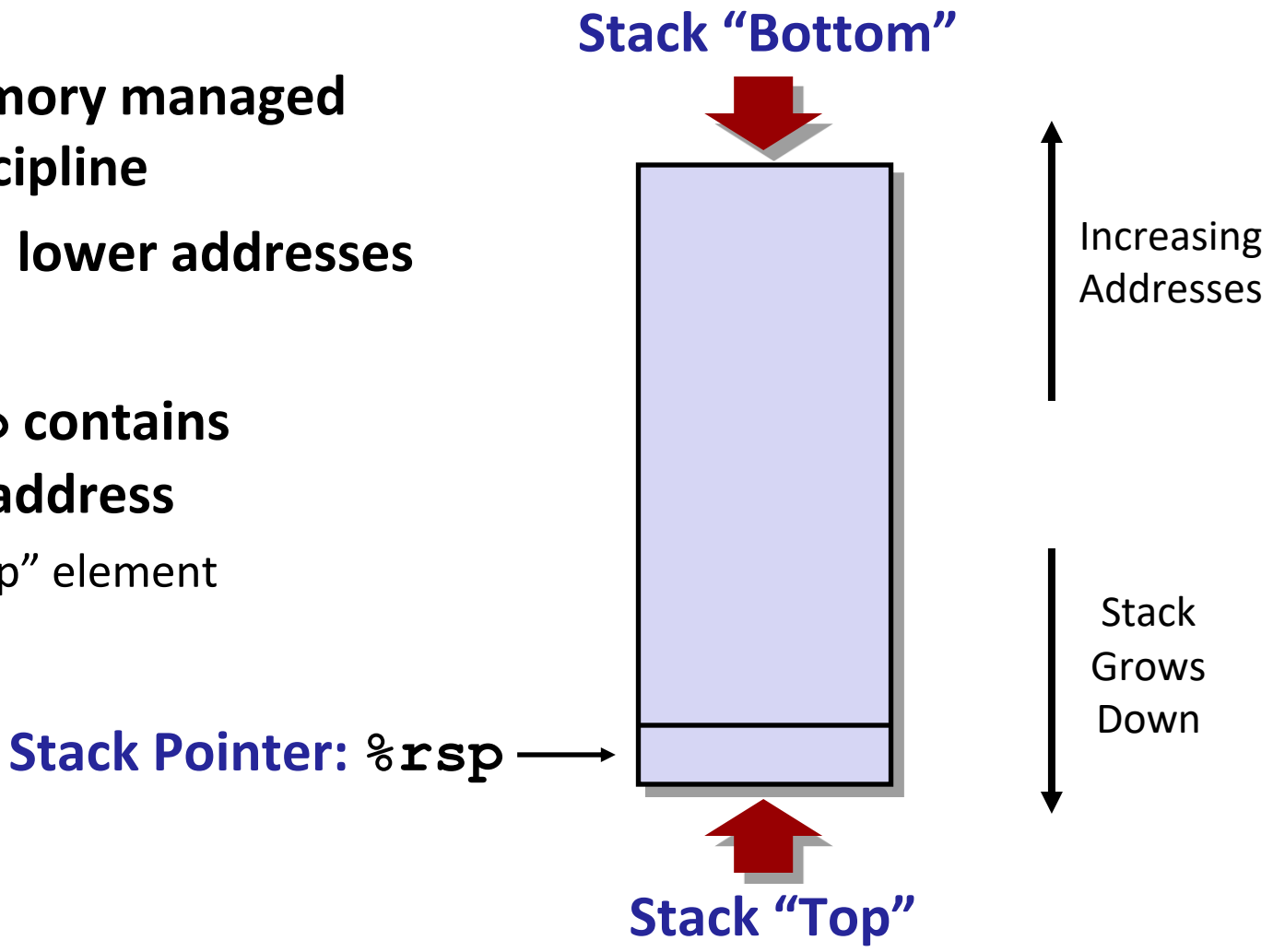
- Region of memory managed with stack discipline



x86-64 Stack

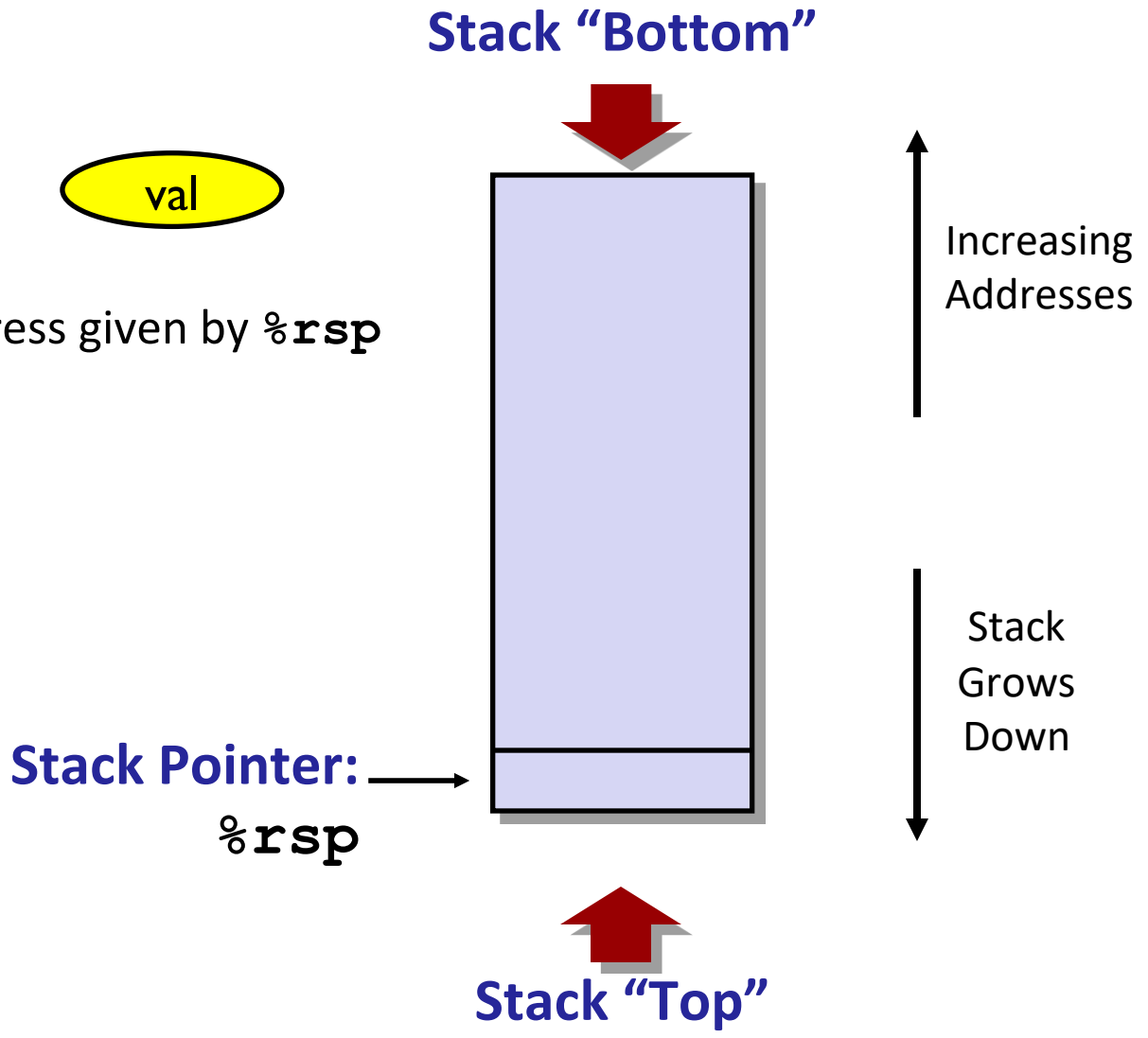
- Region of memory managed with stack discipline
- Grows toward lower addresses

- Register `%rsp` contains lowest stack address
  - address of "top" element



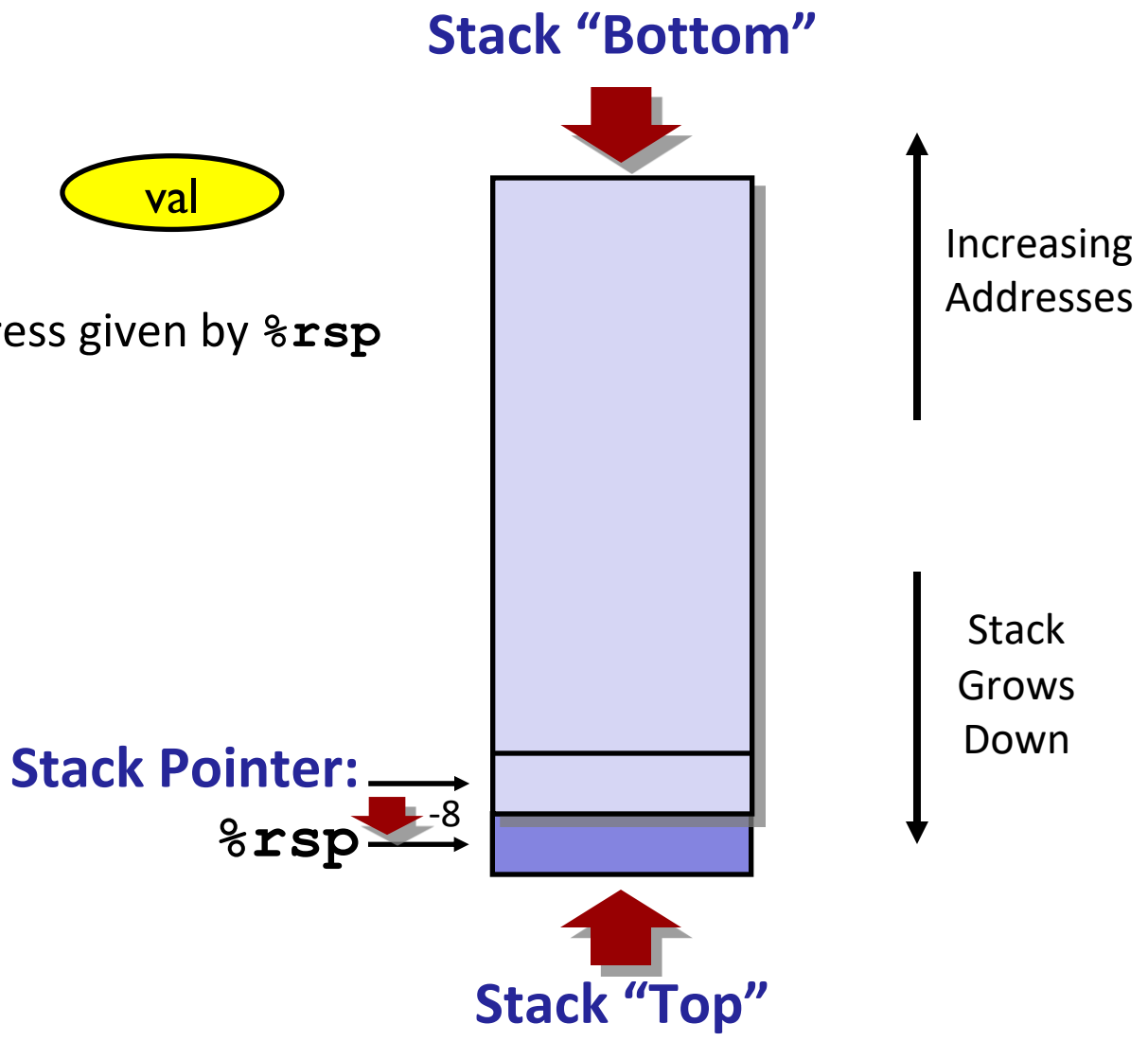
x86-64 Stack: Push

- **`pushq` *Src***
  - Fetch operand at *Src*
  - Decrement **`%rsp`** by 8
  - Write operand at address given by **`%rsp`**



x86-64 Stack: Push

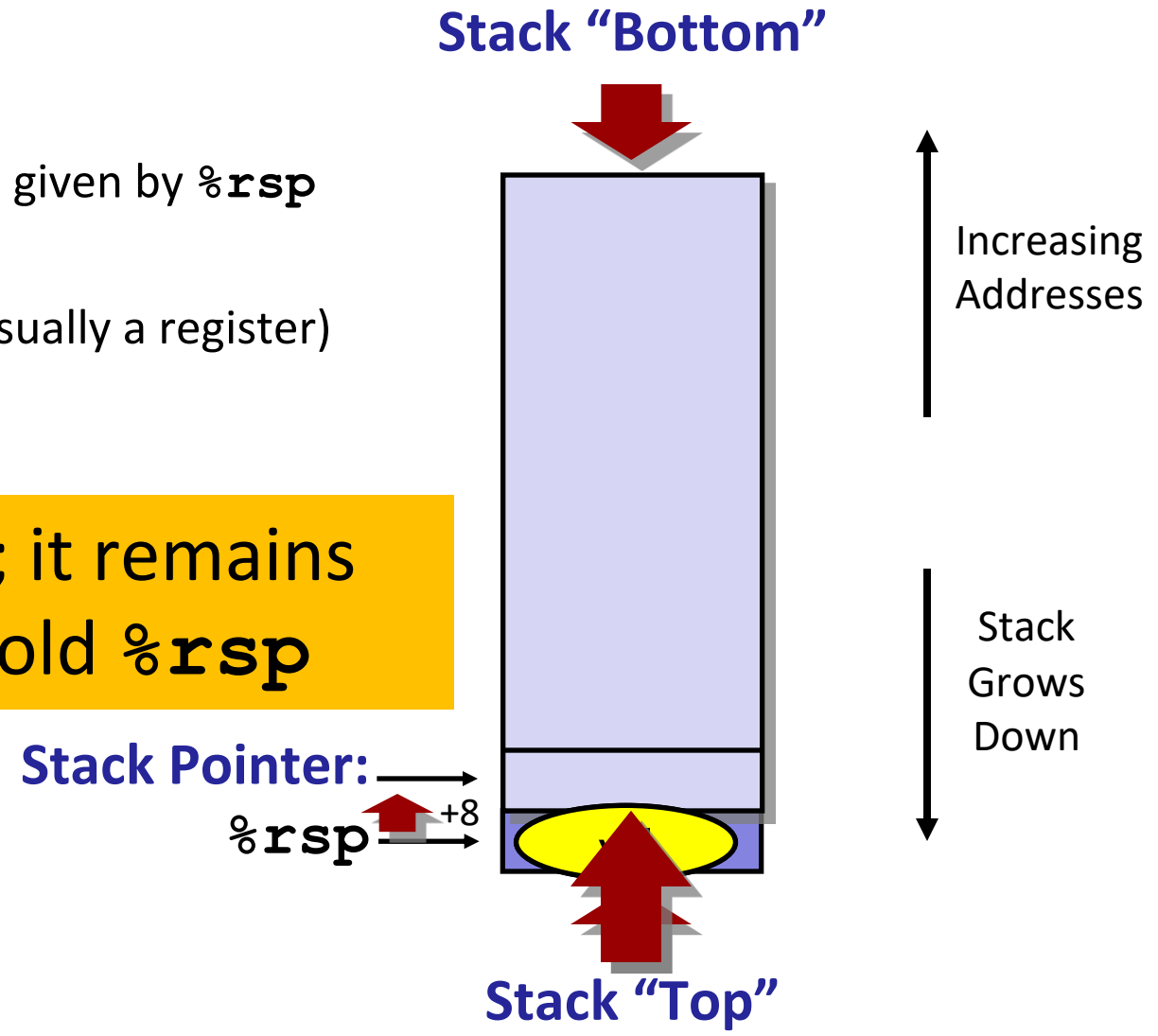
- **`pushq` *Src***
  - Fetch operand at *Src*
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`



x86-64 Stack: Pop

- `popq` *Dest*
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)

Value is **copied**; it remains in memory at old `%rsp`



Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - **Passing control**

Code Examples

```
void multstore(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx                  # Save %rbx
  400541: mov    %rdx,%rbx             # Save dest
  400544: call   400550 <mult2>        # mult2(x,y)
  400549: mov    %rax,(%rbx)           # Save at dest
  40054c: pop    %rbx                  # Restore %rbx
  40054d: ret                          # Return
```

```
long mult2(long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax            # a
  400553:  imul   %rsi,%rax            # a * b
  400557:  ret                         # Return
```

Procedure Control Flow

- **Use stack to support procedure call and return**
- **Procedure call: `call label`**
  - Push return address on stack
  - Jump to ***label***
- **Return address:**
  - Address of the next instruction right after call
  - Example from disassembly
- **Procedure return: `ret`**
  - Pop address from stack
  - Jump to address

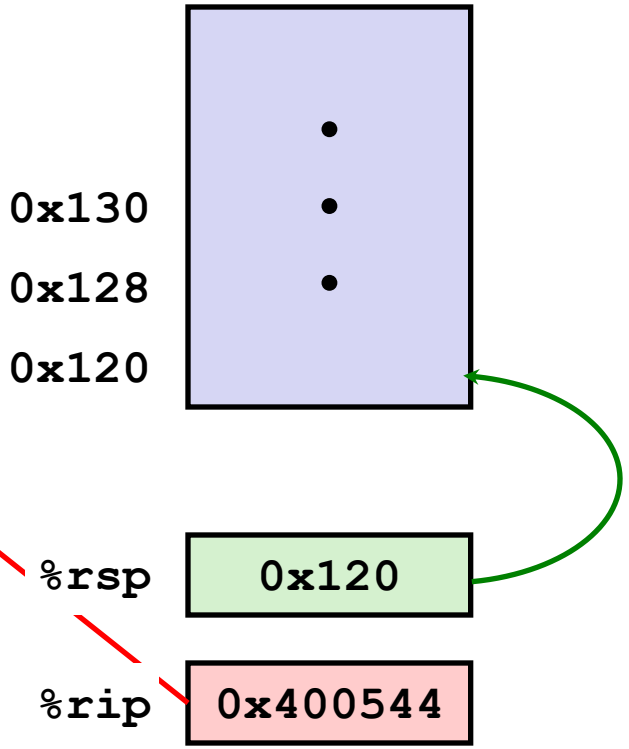
These instructions are sometimes printed with a `q` suffix

- This is just to remind you that you're looking at 64-bit code

Control Flow Example #1

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

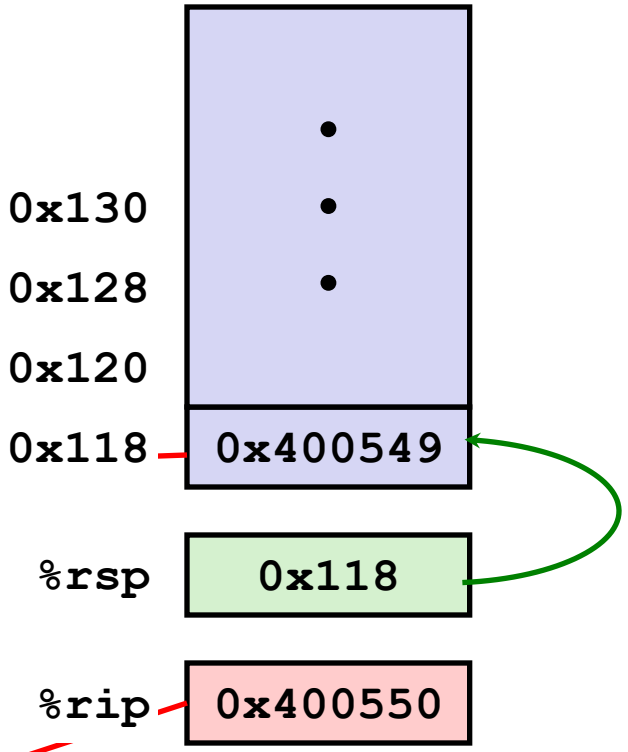
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #2

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

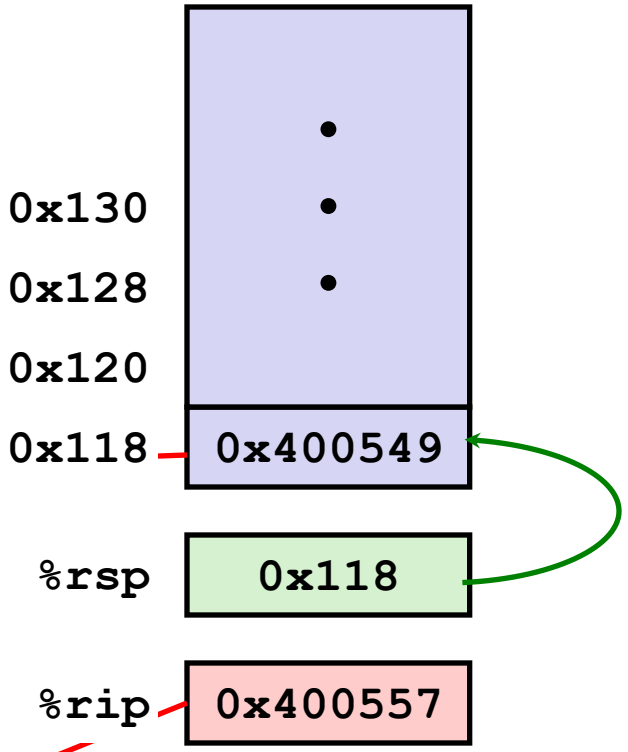
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #3

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #4

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```