# Machine-Level Representation of Programs: Loops, Procedures

Yipeng Huang

Rutgers University

March 30, 2023

# Table of contents

# Announcements

## Class session plan

- Thursday, 3/30: Function calls in assembly. (Book chapter 3.7)
- Monday, 4/3: Arrays and data structures in assembly. (Book chapter 3.8)

# Table of contents

# Compiling for loops to while loops

C loop statements such as for loops, while loops, and do-while loops do not exist in assembly. They are instead constructed from conditional jump statements.

```
unsigned long count_bits_for (
  unsigned long number
) {
  unsigned long tally = 0;
  for (
    int shift=0; // init
    shift<8*sizeof(unsigned long); // test
    shift++ // update
  ) {
    // body
    tally += 0b1 & number>>shift;
  }
  return tally;
}
```

```
unsigned long count_bits_while (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
  while (
    shift<8*sizeof(unsigned long) // test
  ) {
    // body
    tally += 0b1 & number>>shift;
    shift++; // update
  }
  return tally;
}
```

# Compiling while loops to do-while loops

```c
unsigned long count_bits_while (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
  while (
    shift<8*sizeof(unsigned long) // test
  ) {
    // body
    tally += 0b1 & number>>shift;
    shift++; // update
  }
  return tally;
}
```

```c
unsigned long count_bits_do_while (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
  do {
    // body
    tally += 0b1 & number>>shift;
    shift++; // update
  } while (shift<8*sizeof(unsigned long)); // test
  return tally;
}
```

If initial iteration is guaranteed to run, then do one fewer test.

# Compiling do-while loops to goto statements

```c
unsigned long count_bits_do_while (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
  do {
    // body
    tally += 0b1 & number>>shift;
    shift++; // update
  } while (shift<8*sizeof(unsigned long)); // test
  return tally;
}
```

```c
unsigned long count_bits_goto (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
LOOP:
  // body
  tally += 0b1 & number>>shift;
  shift++; // update
  if (shift<8*sizeof(unsigned long)) { // test
    goto LOOP;
  }
  return tally;
}
```

Loops get compiled into goto statements which are readily translated to assembly.

# Compiling goto statements to assembly conditional jump instructions

```c
unsigned long count_bits_goto (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
LOOP:
  // body
  tally += 0b1 & number>>shift;
  shift++; // update
  if (shift<8*sizeof(unsigned long)) { // test
    goto LOOP;
  }
  return tally;
}
```

```
count_bits_for:
count_bits_while:
count_bits_do_while:
count_bits_goto:
  xorl %ecx, %ecx # int shift=0; // init
  xorl %eax, %eax # unsigned long tally = 0;
.LOOP:
  movq %rdi, %rdx # number
  shrq %cl, %rdx  # number>>shift
  incl %ecx       # shift++; // update
  andl $1, %edx.  # 0b1 & number>>shift
  addq %rdx, %rax # tally += 0b1 & number>>shi
  cmpl $64, %ecx  # shift<8*sizeof(unsigned lo
  jne .LOOP       # goto LOOP;
  ret             # return tally;
```

All C loop statements so far translate to assembly at right.

# Table of contents

# Table of contents

# Procedures and function calls

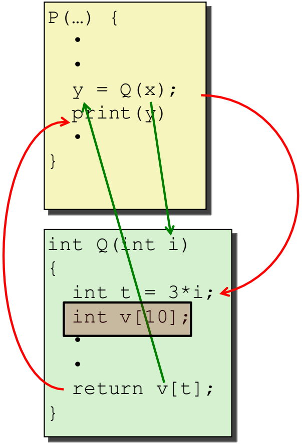
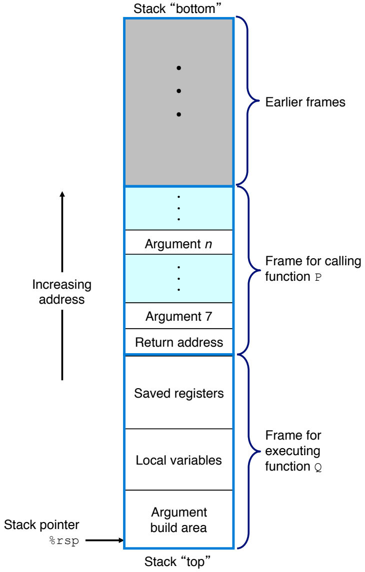



Figure: Steps of a C function call. Image credit CS:APP

To create the abstraction of functions, need to:

- Transfer control to function and back
- Transfer data to function (parameters)
- transfer data from function (return type)

# Memory stack frames




## Structure of stack for currently executing function Q()

- P() calls Q(). P() is the caller function. Q() is the callee function.

# Stack instructions: `push src` and `pop dest`

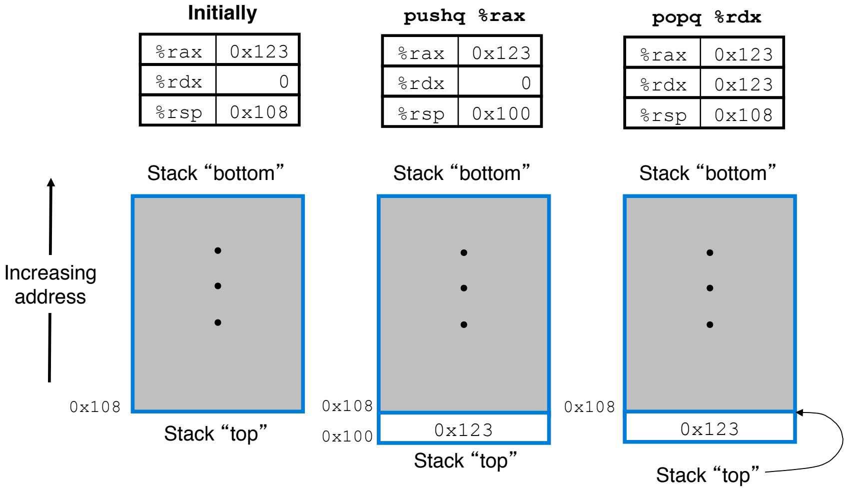
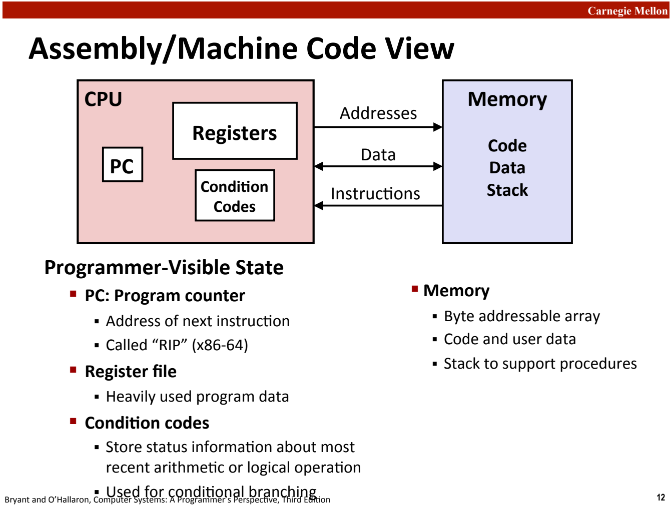



Figure: x86-64 offers dedicated instructions to work with stack in memory. In addition to moving data, the updating of %rsp is implied. Image credit: CS:APP.

# Table of contents

# CPU and memory state in support of procedures and functions




## Relevant state in CPU:

- %rip register / instruction pointer / program counter
- %rsp register / stack pointer

## Relevant state in Memory:

- Stack

# Procedure call and return: `call` and `ret`

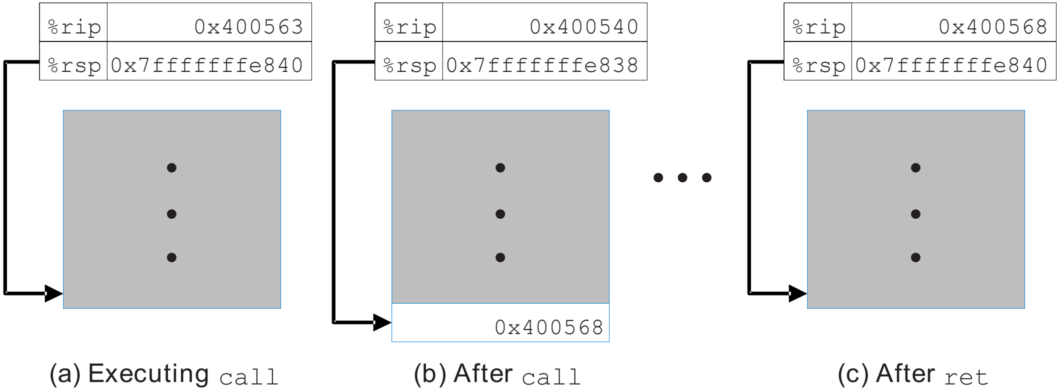



Figure: Effect of `call 0x400540` instruction and subsequent return. `call` and `ret` instructions update the instruction pointer, the stack pointer, and the stack to create the procedure / function call abstraction. Image credit: CS:APP.

# Example in GDB

```c
#include <stdio.h>

int return_neg_one() {
  return -1;
}

int main() {
  int num = return_neg_one();
  printf("%d", num);
  return 0;
}
```

```
return_neg_one:
    movl $-1, %eax
    ret
main:
    subq $8, %rsp
    movl $0, %eax
    call return_neg_one
    movl %eax, %edx
```

## Compile, and then run it in GDB:

`gdb return`

## In GDB, see evolution of %rip, %rsp, and stack:

- `(gdb) layout split`
- `(gdb) break return_neg_one`
- `(gdb) info stack`
- `(gdb) print /a $rip`
- `(gdb) print /a $rsp`
- `(gdb) x /a $rsp`

## Step past return instruction, and inspect again:

- `(gdb) stepi`
- `(gdb) info stack`

# Table of contents

# Procedures and function calls: Transferring data

For purposes of this class, the Bomb Lab, and the CS:APP textbook, we study the x86-64 Linux Application Binary Interface (ABI). Would be different on ARM or in Windows. So, don't memorize this, but it is helpful for PA4 Lab.

## Passing parameters

| Parameter | Register / stack | Subset registers | Mnemonic[1] |
|---:|:---:|:---:|:---|
| 1st | %rdi | %edi, %di | Diane's |
| 2nd | %rsi | %esi, %si | silk |
| 3rd | %rdx | %edx, %dx, %dl | dress |
| 4th | %rcx | %ecx, %cx, %cl | cost |
| 5th | %r8 | %r8d | $8 |
| 6th | %r9 | %r9d | 9 |
| 7th and beyond | Stack | | |

[1] http://csappbook.blogspot.com/2015/08/dianes-silk-dress-costs-89.html

# PA4 Defusing a Binary Bomb: `sscanf();`

```
int sscanf (
    const char *str, // 1st arg, %rdi
    const char *format, // 2nd arg, %rsi
    ...
)
```

# Procedures and function calls: Transferring data

## Passing function return data

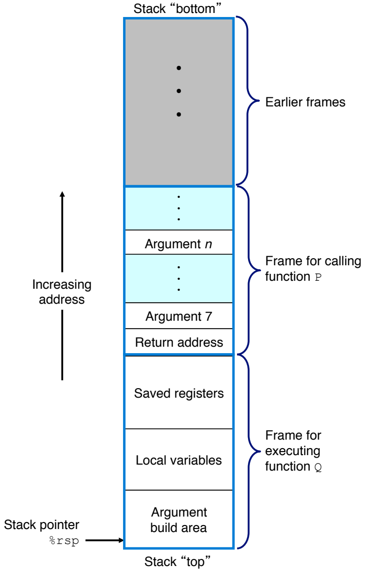
Function return data is passed via:

- the 64-bit %rax register
- the 32-bit subset %eax register

## Example from textbook slides on assembly procedures

Slides 33 through 38.

# Data transferred via memory




## Structure of stack for currently executing function Q()

- P() calls Q(). P() is the caller function. Q() is the callee function.

## Example from textbook slides on assembly procedures

Slides 40 through 44.

# Table of contents

# `3_recursion.c`: Putting it all together to support recursion

## Discussion points

- Use info stack, info args in GDB to see recursion depth
- Difference between compiling with and without -g for debugging information.
- Memory costs of recursion.
- Compilers can recognize tail recursive calls to reduce memory use. Enabled with -foptimize-sibling-calls, -O2, -O3, and -Os.