Machine-Level Representation of Programs: Loops, Procedures

Yipeng Huang

Rutgers University

March 30, 2023

Announcements

Loop statements

Compiling for loops to while loops Compiling while loops to do-while loops Compiling do-while loops to goto statements

Compiling goto statements to assembly conditional jump instructions

Switch statements

Procedures and function calls

Memory stack frames

Procedures and function calls: Transferring control

Procedure call and return: call and ret Example in GDB

Procedures and function calls: Transferring data

Data transferred via registers

Data transferred via memory

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)

(ロ)、(部)、(目)、(目)、(目)、(3/24)

Announcements

Loop statements

Compiling for loops to while loops Compiling while loops to do-while loops Compiling do-while loops to goto statements

Compling do-while loops to goto statements

Compiling goto statements to assembly conditional jump instructions

Switch statements

Procedures and function calls

Memory stack frames

Procedures and function calls: Transferring control

Procedure call and return: call and ret Example in GDB

Procedures and function calls: Transferring data

Data transferred via registers

Data transferred via memory

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.

```
1 unsigned long count_bits_for (
                                            1 unsigned long count_bits_while (
    unsigned long number
                                                unsigned long number
                                            2
2
3)
                                            3)
    unsigned long tally = 0;
                                            4
                                               unsigned long tally = 0;
    for (
                                            5
                                               int shift=0; // init
5
6
      int shift=0; // init
                                            6
                                              while (
      shift<8*sizeof(unsigned long);</pre>
                                            7
                                                  shift<8*sizeof(unsigned long) ↔</pre>
7
           // test
                                                      // test
      shift++ // update
8
                                            8
                                                  // bodv
9
                                            9
                                                  tallv += 0b1 & number>>shift;
      // bodv
10
                                           10
      tally += Ob1 & number>>shift;
                                                  shift++; // update
11
                                           11
12
                                           12
    return tally;
                                                return tally;
13
                                           13
14 }
                                           14 }
```

Compiling while loops to do-while loops

```
1 unsigned long count_bits_while (
                                             1 unsigned long count_bits_do_while ↔
    unsigned long number
2
3)
                                                unsigned long number
                                            2
    unsigned long tally = 0;
4
                                            (3) {
    int shift=0; // init
5
                                                unsigned long tally = 0;
                                            4
    while (
6
                                                int shift=0; // init
                                            5
      shift < 8 \times size of (unsigned long) \leftrightarrow
7
                                                do {
                                            6
          // test
                                                // body
                                            7
8
                                                  tally += 0b1 & number>>shift;
                                            8
      // bodv
9
                                            9
                                               shift++; // update
      tallv += 0b1 & number>>shift;
10
                                            10
                                                } while (shift<8*sizeof(unsigned↔
      shift++; // update
11
                                                      long)); // test
12
                                            11
                                                return tally;
    return tally;
13
                                            12 }
14
```

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

Compiling do-while loops to goto statements

```
1 unsigned long count_bits_do_while ↔
    unsigned long number
2
                                           4
3)
                                           5
    unsigned long tally = 0;
4
    int shift=0; // init
5
                                           7
    do {
6
                                           8
    // body
7
                                           9
      tally += 0b1 & number>>shift;
8
                                           10
   shift++; // update
9
10
    } while (shift<8*sizeof(unsigned↔
                                          11
         long)); // test
                                           12
11
    return tally;
                                           13
12
                                           14
                                             }
```

```
1 unsigned long count_bits_goto (
2 unsigned long number
3) \{
  unsigned long tally = 0;
  int shift=0; // init
6 LOOP .
  // bodv
  tally += 0b1 & number>>shift;
  shift++; // update
   if (shift<8*sizeof(unsigned long↔
       )) { // test
     goto LOOP;
   return tallv;
```

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

Compiling goto statements to assembly conditional jump instructions

```
1 unsigned long count bits goto (
   unsigned long number
2
3) {
    unsigned long tally = 0;
4
    int shift=0: // init
5
6 LOOP:
   // body
7
    tally += 0b1 & number>>shift;
8
    shift++; // update
9
10
    if (shift<8*sizeof(unsigned long↔
        )) { // test
    goto LOOP;
11
12
    return tally;
13
14
  }
```

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

```
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. # Obl & number>>shift
  addg %rdx, %rax # tally += 0b1 & number>>sh:
  cmpl $64, %ecx # shift<8*sizeof(unsigned lo</pre>
  jne .LOOP # goto LOOP;
  ret
                 # return tally;
```

Announcements

Loop statements

Compiling for loops to while loops Compiling while loops to do-while loops Compiling do-while loops to goto statements Compiling goto statements to assembly conditional jump instructions

Switch statements

Procedures and function calls

Memory stack frames

Procedures and function calls: Transferring control

Procedure call and return: call and ret Example in GDB

Procedures and function calls: Transferring data

Data transferred via registers

Data transferred via memory

Announcements

Loop statements

Compiling for loops to while loops Compiling while loops to do-while loops Compiling do-while loops to goto statements

Compiling goto statements to assembly conditional jump instructions

Switch statements

Procedures and function calls

Memory stack frames

Procedures and function calls: Transferring control

Procedure call and return: call and ret Example in GDB

Procedures and function calls: Transferring data

Data transferred via registers

Data transferred via memory

Procedures and function calls

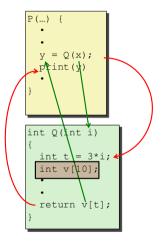


Figure: Steps of a C function call. Image credit CS:APP

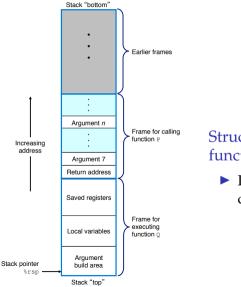
To create the abstraction of functions, need to:

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

(ロ)、(型)、(E)、(E)、(E)、(O)()

11/24

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

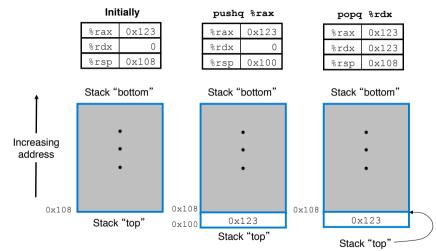


Figure: x86-64 offers dedicated instructions to work with stack in memory. In addition to moving data, the updating of %rsp is implied. Image credit: CS:APP.

< □ ▶ < □ ▶ < ■ ▶ < ■ ▶ < ■ ♪ ■ のへで 13/24

Announcements

Loop statements

Compiling for loops to while loops Compiling while loops to do-while loops Compiling do-while loops to goto statements

Compiling goto statements to assembly conditional jump instructions

Switch statements

Procedures and function calls

Memory stack frames

Procedures and function calls: Transferring control

Procedure call and return: call and ret Example in GDB

Procedures and function calls: Transferring data

Data transferred via registers

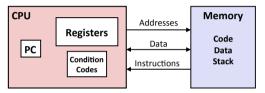
Data transferred via memory

CPU and memory state in support of procedures and functions

Carnegie Mellon

12

Assembly/Machine Code View



Programmer-Visible State

- PC: Program counter
 - Address of next instruction
 - Called "RIP" (x86-64)

Register file

- Heavily used program data
- Condition codes
 - Store status information about most recent arithmetic or logical operation
- Bryant and O'Hallaron, Computer systems: A Programmer's Perspective, Third Edition

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures

Relevant state in CPU:

 %rip register / instruction pointer / program counter

<ロト < 回 ト < 目 ト < 目 ト 目 の 9 0 15/24

- %rsp register / stack pointer
- Relevant state in Memory:



Procedure call and return: call and ret

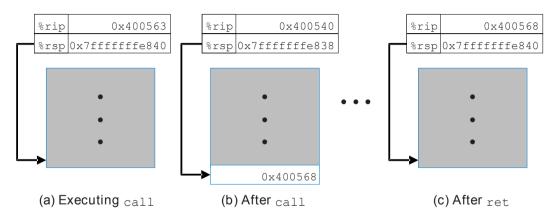


Figure: Effect of call 0x400540 instruction and subsequent return. call and ret instructions update the instruction pointer, the stack pointer, and the stack to create the procedure / function call abstraction. Image credit: CS:APP.

Example in GDB

```
1 #include <stdio.h>
2
3 int return_neg_one() {
4   return -1;
5 }
6
7 int main() {
8   int num = return_neg_one();
9   printf("%d", num);
10   return 0;
11 }
```

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

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 17/24

Announcements

Loop statements

Compiling for loops to while loops Compiling while loops to do-while loops Compiling do-while loops to goto statements

Compiling goto statements to assembly conditional jump instructions

Switch statements

Procedures and function calls

Memory stack frames

Procedures and function calls: Transferring control

Procedure call and return: call and ret Example in GDB

Procedures and function calls: Transferring data

Data transferred via registers

Data transferred via memory

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		

¹http://csappbook.blogspot.com/2015/08/dianes-silk-dress-costs-89.htmbace 19/24

PA4 Defusing a Binary Bomb: sscanf();

```
1 int sscanf (
2 const char *str, // lst arg, %rdi
3 const char *format, // 2nd arg, %rsi
4 ...
5 )
```

Procedures and function calls: Transferring data

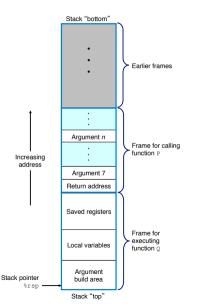
Passing function return data

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.

◆□▶ ◆□▶ ◆ ■▶ ◆ ■ ● ○ へ ○ 22/24

Example from textbook slides on assembly procedures

Slides 40 through 44.

Announcements

Loop statements

Compiling for loops to while loops Compiling while loops to do-while loops Compiling do-while loops to goto statements Compiling goto statements to assembly conditional jump instructions

Switch statements

Procedures and function calls

Memory stack frames

Procedures and function calls: Transferring control

Procedure call and return: call and ret Example in GDB

Procedures and function calls: Transferring data

Data transferred via registers

Data transferred via memory

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.