Machine-Level Representation of Programs: Loops, Procedures

Yipeng Huang

Rutgers University

March 28, 2024
Table of contents

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

Architecture support for recursive programming
Announcements

Class session plan

- Tuesday, 4/2: Arrays and data structures in assembly. (Book chapter 3.8). Bomblab phase_5, phase_6.
Table of contents

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

Architecture support for recursive programming
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.

```c
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;
}
```

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

```
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;
}
```

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. # 0b1 & number>>shift
addq %rdx, %rax # tally += 0b1 & number>>shift
cmpl $64, %ecx # shift<8*sizeof(unsigned long)
jne .LOOP # goto LOOP;
ret # return tally;
Table of contents

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

Architecture support for recursive programming
Table of contents

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

Architecture support for recursive programming
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`

Initially

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
<td></td>
</tr>
</tbody>
</table>

`pushq %rax`

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

`popq %rdx`

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>0x123</td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Architecture support for recursive programming
CPU and memory state in support of procedures and functions

### 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
  - Used for conditional branching

**Memory**
- **Code Data Stack**

**Relevant state in CPU:**
- `%rip` register / instruction pointer / program counter
- `%rsp` register / stack pointer

**Relevant state in Memory:**
- Stack
Procedure call and return: \texttt{call} and \texttt{ret}

Figure: Effect of \texttt{call} 0x400540 instruction and subsequent return. \texttt{call} and \texttt{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;
}
```

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) next
- (gdb) info stack
Table of contents

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

Architecture support for recursive programming
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register / stack</th>
<th>Subset registers</th>
<th>Mnemonic¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>%rdi</td>
<td>%edi, %di</td>
<td>Diane’s</td>
</tr>
<tr>
<td>2nd</td>
<td>%rsi</td>
<td>%esi, %si</td>
<td>silk</td>
</tr>
<tr>
<td>3rd</td>
<td>%rdx</td>
<td>%edx, %dx, %dl</td>
<td>dress</td>
</tr>
<tr>
<td>4th</td>
<td>%rcx</td>
<td>%ecx, %cx, %cl</td>
<td>cost</td>
</tr>
<tr>
<td>5th</td>
<td>%r8</td>
<td>%r8d</td>
<td>$8</td>
</tr>
<tr>
<td>6th</td>
<td>%r9</td>
<td>%r9d</td>
<td>9</td>
</tr>
<tr>
<td>7th and beyond</td>
<td>Stack</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹http://csappbook.blogspot.com/2015/08/dianes-silk-dress-costs-89.html
PA4 Defusing a Binary Bomb: `sscanf()`;

```c
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

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

Architecture support for recursive programming
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.