Compiler Construction

Theoretical Exercise 5: x86-64 assembler

Michael Engel

TE5.1 x86-64 code analysis

Consider the following x86-64 assembler code function, compiled from C:

a. How many parameters does the function take?
   Which instructions indicate this (give the instruction address)?

Values are passed to functions in registers (in that order):
rdi, rsi, rdx, rcx, r8, and r9
(or edi, esi… for 32-bit values)

The function uses the value in esi (address 0) as well as edi and edx (addresses 6 and 0xc)

So we can assume that the function uses 3 parameters

```
Disassembly of section .text:

0000000000000000 <foo>:
   0: 89 f0                 mov    %esi,%eax
   2: 85 f6                 test   %esi,%esi
   4: 7e 0f                 jle    15 <foo+0x15>
   6: 85 d2                 test   %edx,%edx
   8: 74 0c                 je     16 <foo+0x16>
   a: 31 d2                 xor    %edx,%edx
   c: 01 f8                 add    %edi,%eax
   e: 83 c2 01              add    $0x1,%edx
  11: 39 d0                 cmp    %edx,%eax
  13: 7f f7                 jg     c <foo+0xc>
  15: c3                    retq
  16: 29 f8                 sub    %edi,%eax
  18: 83 c2 01              add    $0x1,%edx
  1b: 39 d0                 cmp    %edx,%eax
  1d: 7f f7                 jg     16 <foo+0x16>
  1f: c3                    retq
```

TE5.1 x86-64 code analysis

Consider the following x86-64 assembler code function, compiled from C:

b. Does the code of the function include an if statement?
How did you find this out?

The instructions `test %esi,%esi` and `test %edx,%edx` look a bit strange:

"The TEST instruction performs a bitwise AND on two operands. The flags SF, ZF, PF are modified while the result of the AND is discarded"

When both operands are identical, it works as a test for zero and sign of the operand => if instructions use conditional jumps to check the zero and sign flags (here: jle/je)

```
Disassembly of section .text:

0000000000000000 <foo>:
   0: 89 f0                    mov    %esi,%eax
   2: 85 f6                    test   %esi,%esi
   4: 7e 0f                    jle    15 <foo+0x15>
   6: 85 d2                    test   %edx,%edx
   8: 74 0c                    je     16 <foo+0x16>
   a: 31 d2                    xor    %edx,%edx
   c: 01 f8                    add    %edi,%eax
   e: 83 c2 01                 add    $0x1,%edx
  11: 39 d0                    cmp    %edx,%eax
  13: 7f f7                    jg     c <foo+0xc>
  15: c3                       retq
  16: 29 f8                    sub    %edi,%eax
  18: 83 c2 01                 add    $0x1,%edx
  1b: 39 d0                    cmp    %edx,%eax
  1d: 7f f7                    jg     16 <foo+0x16>
  1f: c3                       retq
```

TE5.1 x86-64 code analysis

Consider the following x86-64 assembler code function, compiled from C:

c. Does the code of the function include a loop?
How did you find this out?

Loops in the code usually jump *backwards* with a conditional jump, so there are two loops here: at addresses 0xc-0x14 and at 0x16-0x1e

We cannot find out which loop this was originally (do–while or for), since the compiler can transform loops to other forms

```
Disassembly of section .text:

0000000000000000 <foo>:
   0: 89 f0                    mov    %esi,%eax
   2: 85 f6                    test   %esi,%esi
   4: 7e 0f                    jle    15 <foo+0x15>
   6: 85 d2                    test   %edx,%edx
   8: 74 0c                    je     16 <foo+0x16>
   a: 31 d2                    xor    %edx,%edx
   c: 01 f8                    add    %edi,%eax
   e: 83 c2 01                 add    $0x1,%edx
  11: 39 d0                    cmp    %edx,%eax
  13: 7f f7                    jg     c <foo+0xc>
  15: c3                       retq
  16: 29 f8                    sub    %edi,%eax
  18: 83 c2 01                 add    $0x1,%edx
  1b: 39 d0                    cmp    %edx,%eax
  1d: 7f f7                    jg     16 <foo+0x16>
  1f: c3                       retq
```

TE5.1 x86-64 code analysis

Consider the following x86-64 assembler code function, compiled from C:

d. Does the function return a value?

The x86-64 ABI (System V, used in Linux) requires the return value of a function to be passed in the %eax register.

The value of %eax is modified, so we can *assume* this is a return value.

We can only be sure about this if we see the code calling the function

```
Disassembly of section .text:

0000000000000000 <foo>:
   0: 89 f0                    mov    %esi,%eax
   2: 85 f6                    test   %esi,%esi
   4: 7e 0f                    jle    15 <foo+0x15>
   6: 85 d2                    test   %edx,%edx
   8: 74 0c                    je     16 <foo+0x16>
   a: 31 d2                    xor    %edx,%edx
   c: 01 f8                    add    %edi,%eax
   e: 83 c2 01                 add    $0x1,%edx
  11: 39 d0                    cmp    %edx,%eax
  13: 7f f7                    jg     c <foo+0xc>
  15: c3                       retq
  16: 29 f8                    sub    %edi,%eax
  18: 83 c2 01                 add    $0x1,%edx
  1b: 39 d0                    cmp    %edx,%eax
  1d: 7f f7                    jg     16 <foo+0x16>
  1f: c3                       retq
```

TE5.1 x86-64 code analysis

(this was not a question, but it helps…) What would the corresponding C code look like?

int a: %edi, int b: %esi, int c: %edx

```
Disassembly of section .text:

0000000000000000 <foo>:
   0: 89 f0                 mov    %esi,%eax
   2: 85 f6                 test   %esi,%esi
   4: 7e 0f                 jle    15 <foo+0x15>
   6: 85 d2                 test   %edx,%edx
   8: 74 0c                 je     16 <foo+0x16>
   a: 31 d2                 xor    %edx,%edx
   c: 01 f8                 add    %edi,%eax
   e: 83 c2 01              add    $0x1,%edx
  11: 39 d0                 cmp    %edx,%eax
  13: 7f f7                 jg     c <foo+0xc>
  15: c3                    retq
  16: 29 f8                 sub    %edi,%eax
  18: 83 c2 01              add    $0x1,%edx
  1b: 39 d0                 cmp    %edx,%eax
  1d: 7f f7                 jg     16 <foo+0x16>
  1f: c3                    retq
```

```
int foo(int a, int b, int c) {
  if (b <= 0) return b;

  if (c != 0) {
    do {
      b = b - a;
      c++;
    } while(c > b);
  } else {
    c = 0;
    do {
      b = b + a;
      c++;
    } while (c > b);
  }
  return b;
}
```

TE5.2 Disassemble!

The following x86-64 assembly code is given:

a. Give equivalent valid C code that would compile without warnings to this assembler code function. Assume the declaration extern unsigned a, b;

```
void f() {
    a -= b & 255;
}
```

```
f:
        movl a, %eax
        movl b, %edx
        andl $255, %edx
        subl %edx, %eax
        movl %eax, a
        retq
```

TE5.2 Disassemble!

The following x86-64 assembly code is given:

b. Find two different versions of C code that compile to the above code. One of these should have a different function signature than the ones you described already.

```
f:
        movl a, %eax
        movl b, %edx
        andl $255, %edx
        subl %edx, %eax
        movl %eax, a
        retq
```

```
void f() {
    a += -(b % 256);
}
```

```
unsigned f() {
    a = a - b % 0x100;
    return a;
}
```

```
unsigned f() {
    a -= (unsigned char) b;
    return a;
}
```

```
char* f(int x, int y, int z[1000]) {
    a -= (unsigned char) b;
    return (char*) a;
}
```

TE5.3 Data types

For each of the following x86-64 assembler instructions, give the type of the data object that is most likely to be accessed by this code. Indicate the reason for your answer.

- `movzbl %al, %eax`
  `unsigned char`: `movzbl` instructs the cpu to fetch a byte from memory, and zero extend it to 32 bits.
- `movl -28(%rbp), %edx`
  `int` or `unsigned`: `movl` copies a 32 bit value, here from the stack frame to edx
- `movsbl -32(%rbp), %eax`
  [`signed`] `char`: `movsbl` means "move with sign extend from byte to longword"
- `movl (%rdx,%rax,4), %eax`
  Array of `int`s or `unsigned int`s: offset in rax multiplied by 4 (=sizeof(int)), base address in rdx
- `movzbl 4(%rax), %eax ; movsbl %al, %eax`
  `char` field from a structure; or the 4th character in a string.