CS107 Lecture 17: Two-Minute Etudes

Problem: Reverse Compilation, Take II

Each function below is shown with one or two lines of C replaced by blanks, alongside the complete x86-64 assembly **`gcc`** generated for the entire function, including the instructions for the missing lines. Work out what each missing line of C must be.

These problems exercise the arithmetic, logic, and shift instructions, along with lea and the **`movz`** and **`movs`** families. Right shifts only ever involve unsigned or nonnegative values. If a problem relies on something you haven't seen yet, we explain it.

Are these really two-minute etudes? Most of them aren't. But each half of b, part c, part e, and each half of part f should be doable in two minutes. The others? You'd be given more time.

a. The C code doesn't include any addition, even though the assembly has an **`addl`** instruction. And recall that writing to a 32-bit register zeroes the upper 32 bits.

C Code:

```
void brooke(int *p) {
  p[0] = -p[0];
  ____________________;
  ____________________;
}
```

x86:

```
brooke:
    movl    (%rdi), %edx
    movl    %edx, %eax
    negl    %eax
    movl    %eax, (%rdi)
    movl    4(%rdi), %eax
    notl    %eax
    movl    %eax, 4(%rdi)
    addl    %edx, 12(%rdi)
    ret
```

b. Each function multiplies **`x`** by a constant, but **`gcc`** avoids **`imul`** like the plague when it can, because shifts, subtractions, and **`lea`** are much cheaper. What are the constants?

C Code:

```
long mult_by(long x) {
  return ______ * x;
}

long times_by(long x) {
  return ______ * x;
}
```

x86:

```
mult_by:
    leaq    (%rdi,%rdi,2), %rax
    salq    $3, %rax
    ret

times_by:
    movq    %rdi, %rax
    salq    $4, %rax
    subq    %rdi, %rax
    leaq    (%rdi,%rax,4), %rax
    ret
```

c. Remember that the two-operand **`imul S, D`** computes **`D ← D * S`**.

C Code:

```
long aryan(long marcus) {
  return ____________________;
}
```

x86:

```
aryan:
    movq    %rdi, %rdx
    imulq   %rdi, %rdx
    leaq    (%rdi,%rdi,8), %rax
    leaq    (%rax,%rdx,4), %rax
    ret
```

d. **`%dil`** is the lowest byte of **`%rdi`**, just as **`%al`** is the lowest byte of **`%rax`**. And as a one-byte immediate, **`$-128`** has the bit pattern **`0x80`**.

C Code:

```
int dahnia(int x, int m) {
  ____________________;
  ____________________;
  return x ^ 0x01;
}
```

x86:

```
dahnia:
    orb     $-128, %dil
    movl    %esi, %eax
    notl    %eax
    andl    %edi, %eax
    xorl    $1, %eax
    ret
```

e. As with part a, remember that writing to a 32-bit register zeroes the upper 32 bits of the full register. That means the second assembly code instruction actually does something meaningful.

C Code:

```
unsigned long
patrick(unsigned long y,
        unsigned long z) {
  return ____________________;
}
```

x86:

```
patrick:
    salq    $32, %rdi
    movl    %esi, %esi
    movq    %rdi, %rax
    orq     %rsi, %rax
    ret
```

f. Each of **`tween`** and **`tail`** returns a single expression. And while it's not obvious, that **`movzbl`** in **`tween`** is functioning as a mask. And that **`-8`** is really just **`~0x07`**.

C Code:

```
unsigned long
tween(unsigned long x) {
  return ____________________;
}

unsigned long
tail(unsigned long n) {
  return ____________________;
}
```

x86:

```
tween:
    shrq    $16, %rdi
    movzbl  %dil, %eax
    ret

tail:
    leaq    7(%rdi), %rax
    andq    $-8, %rax
    ret
```

g. The following swap function (called **sw**) really does exchange the two values reachable through its two parameters using a well-known algorithm. How is it being implemented here?

C Code:

```
void sw(long *a, long *b) {
  *a ^= *b;
  _____________________;
  _____________________;
}
```

x86:

```
sw:
    movq    (%rsi), %rax
    xorq    (%rdi), %rax
    movq    %rax, (%rdi)
    xorq    (%rsi), %rax
    movq    %rax, (%rsi)
    xorq    %rax, (%rdi)
    ret
```

h. The final example relies on the following **struct** definition:

```
typedef struct {
    unsigned int doris; unsigned int evelyn; unsigned char thomas;
} group;
```

doris is at an offset of 0, **evelyn** is at an offset of 4 bytes, and **thomas** is at an offset of 8.

C Code:

```
void
bffs(group *saanvi,
     unsigned int delta) {
  saanvi->doris += delta;
  _____________________;
  _____________________;
}
```

x86:

```
bffs:
    movl    %esi, %edx
    addl    (%rdi), %edx
    movl    %edx, (%rdi)
    movl    4(%rdi), %eax
    subl    %esi, %eax
    addl    $1, %eax
    movl    %eax, 4(%rdi)
    shrl    $4, %edx
    movb    %dl, 8(%rdi)
```

Answers:

a. p[1] = ~p[1]; p[3] -= p[0];
b. 24 and 61
c. 4 * marcus * marcus + 9 * marcus
d. x |= 0x80; x &= ~m;
e. (y << 32) | (z & 0xFFFFFFFF)
f. (x >> 16) & 0xff and (n + 7) & ~0x07
g. *b ^= *a; *a ^= *b;
h. saanvi->evelyn += 1 - delta; saanvi->thomas = saanvi->doris >> 4;