Homework 4 - Solution

JUMP INSTRUCTIONS

In the following excerpts from a disassembled binary, some of the information has been replaced by `X`s. Answer the following questions about these instructions.

A. What is the target of the `je` instruction below? (You don't need to know anything about the `callq` instruction here.)

```
4003fa: 74 02                   je     XXXXXX
4003fc: ff d0                   callq  *%rax
```

B. What is the target of the `je` instruction below?

```
40042f: 74 f4                   je     XXXXXX
400431: 5d                      pop    %rbp
```

C. What is the address of the `ja` and `pop` instructions?

```
XXXXXX: 77 02                   ja     400547
XXXXXX: 5d                      pop    %rbp
```

D. In the code that follows, the jump target is encoded in PC-relative form as a 4-byte, two's-complement number. The bytes are listed from least significant to most, reflecting the little-endian byte ordering of x86-64. What is the address of the jump target?

```
4005e8: e9 73 ff ff ff          jmp XXXXXX
4005ed: 90                      nop
```

A. The `je` instruction has as its target `0x4003fc + 0x02`. As the original disassembled code shows, this is `0x4003fe`:

```
4003fa: 74 02                   je     4003fe
4003fc: ff d0                   callq  *%rax
```

B. The je instruction has as its target 0x400431 – 12, this is 0x400425:

```
40042f: 74 f4                   je     400425
400431: 5d                      pop    %rbp
```

C. According to the annotation produced by the disassembler, the jump target is at absolute address `0x400547`. According to the byte encoding, this must be at an address `0x2` bytes beyond that of the `pop` instruction. Subtracting these gives address `0x400545`. Noting that the encoding of the `ja` instruction requires 2 bytes, it must be located at address `0x400543`.

```
400543: 77 02                   ja     400547
400545: 5d                      pop    %rbp
```

D. Reading the bytes in reverse order, we see that the target offset is `0xffffff73`, or decimal −141. Adding this to `0x4005ed` (the address of the nop instruction) gives address `0x400560`:

```
4005e8: e9 73 ff ff ff              jmp 400560
4005ed: 90                          nop
```

CONDITIONAL MOVES

In the following C function, we have left the definition of operation `OP` incomplete:

```
#define OP ____ /* Unknown operator */

long arith(long x) {
    return x OP 8;
}
```

When compiled, GCC generates the following assembly code:

long arith(long x)

x in %rdi

```
arith:
  leaq      7(%rdi), %rax
  testq     %rdi, %rdi
  cmovns    %rdi, %rax
  sarq      $3, %rax
  ret
```

What operation is OP (only one operation) and explain how it works.

The operator is '/'. We see this is an example of dividing a two's-complement by powers of 2 using right shifting. Before shifting by $k = 3$, we must add a bias of $2^k - 1 = 7$ when the dividend is negative (Why? You can checkout textbook Section 2.3.7).

Here is an annotated version of the assembly code:

```
arith:
  leaq      7(%rdi), %rax           temp = x+7
  testq     %rdi, %rdi              Test x
  cmovns    %rdi, %rax              If x >= 0, temp = x
  sarq      $3, %rax                result = temp >> 3 (= x/8)
  ret
```

The program creates a temporary value equal to $x + 7$, in anticipation of x being negative and therefore requiring biasing. The `cmovns` instruction conditionally changes this number to x when $x \geq 0$, and then it is shifted by 3 to generate $x/8$.

LOOPS

Executing a `continue` statement in C causes the program to jump to the end of the current loop iteration. The stated rule for translating a `for` loop into a `while` loop needs some refinement when dealing with `continue` statements. For example, consider the following code:

```
/* Example of for loop using a continue statement */
/* Sum even numbers between 0 and 9 */
long sum = 0;
long i;
for (i = 0; i < 10; i++) {
    if (i & 1)
        continue;
    sum += i;
}
```

A. What would we get if we naively applied our rule for translating the `for` loop into a `while` loop? What would be wrong with this code?
B. How could you replace the `continue` statement with a `goto` statement to ensure that the `while` loop correctly duplicates the behavior of the `for` loop?

A. Applying our translation rule would yield the following code:

```
/* Naive translation of for loop into while loop */
/* WARNING: This is buggy code */
long sum = 0;
long i = 0;
while (i < 10) {
    if (i & 1)
        /* This will cause an infinite loop */
        continue;
    sum += i;
    i++;
}
```

This code has an infinite loop, since the `continue` statement would prevent index variable `i` from being updated.

B. The general solution is to replace the continue statement with a goto statement that skips the rest of the loop body and goes directly to the update portion:

```
/* Correct translation of for loop into while loop */
long sum = 0;
long i = 0;
while (i < 10) {
    if (i & 1)
        goto update;
    sum += i;
update:
    i++;
}
```