Einführung in Low-Level-Bit-Hacks

Ich habe mich entschlossen, einen Artikel über eine Sache zu schreiben, die für Programmierer von eingebetteten Systemen zur zweiten Natur geworden ist - Low-Level-Bit-Hacks. Bit-Hacks sind geniale kleine Programmiertricks, die Ganzzahlen auf intelligente und effiziente Weise manipulieren. Anstatt Operationen (wie z. B. das Zählen der Anzahl der 1-Bits in einer ganzen Zahl) in einer Schleife über einzelne Bits auszuführen, erledigen diese Programmiertricks das Gleiche mit ein oder zwei sorgfältig ausgewählten bitweisen Operationen.

Um die Sache in Gang zu bringen, gehe ich davon aus, dass Sie wissen, was die binäre Zweierkomplement-Darstellung einer ganzen Zahl ist, und dass Sie auch alle bitweisen Operationen kennen. Ich werde in diesem Artikel die folgende Notation für bitweise Operationen verwenden:

```
&    -  bitwise and
|    -  bitwise or
^    -  bitwise xor
~    -  bitwise not
<<   -  bitwise shift left
>>   -  bitwise shift right
```

Bei den Zahlen in diesem Artikel handelt es sich um vorzeichenbehaftete 8-Bit-Ganzzahlen (obwohl die Operationen mit vorzeichenbehafteten Ganzzahlen beliebiger Länge funktionieren), die im Zweierkomplement dargestellt werden und in der Regel mit „x“ bezeichnet werden. Das Ergebnis ist in der Regel „y“. Die einzelnen Bits von „x“ heißen b7, b6, b5, b4, b3, b3, b2, b1 und b0. Das Bit b7 ist das höchstwertige Bit (oder in der vorzeichenbehafteten Arithmetik - Vorzeichenbit), und b0 ist das niedrigstwertige.

Ich beginne mit den einfachsten Bit-Hacks und gehe allmählich zu schwierigeren über. Ich werde anhand von Beispielen erklären, wie jeder Bit-Hack funktioniert.

Wenn Ihnen dieses Thema gefällt, können Sie meinen Blog abonnieren oder einfach mitlesen. Es wird auch einen zweiten Teil dieses Artikels geben, in dem ich fortgeschrittener Bit-Hacks behandeln werde, und ich werde auch einen Spickzettel mit all diesen Bit-Tricks veröffentlichen.

Los geht's!

Bit Hack #1. Prüfen Sie, ob die Ganzzahl gerade oder ungerade ist.

```
if ((x & 1) == 0) {
  x is even
}
else {
  x is odd
}
```

Ich bin mir ziemlich sicher, dass jeder diesen Trick schon einmal gesehen hat. Die Idee dahinter ist, dass eine ganze Zahl nur dann ungerade ist, wenn das niederwertigste Bit b0 1 ist. Das ergibt sich aus der binären Darstellung von 'x', bei der Bit b0 entweder 1 oder 0 ist. Durch UND-Verknüpfung von 'x' mit 1 eliminieren wir alle anderen Bits außer b0. Wenn das Ergebnis nach dieser Operation 0 ist, dann war 'x' gerade, weil Bit b0 0 war, ansonsten war 'x' ungerade.

Schauen wir uns einige Beispiele an. Nehmen wir die ganze Zahl 43, die ungerade ist. Im Binärformat ist 43 00101011. Beachten Sie, dass das niederwertige Bit b0 1 ist (fett gedruckt). Jetzt wollen wir es mit 1 UND verknüpfen:

```
    00101011
&   00000001   (note: 1 is the same as 00000001)
    --------
    00000001
```

Sehen Sie, wie die UND-Verknüpfung alle höherwertigen Bits b1-b7 löscht, aber das Bit b0 unverändert lässt? Das Ergebnis ist also 1, was uns sagt, dass die ganze Zahl ungerade war.

Schauen wir uns nun -43 an. Zur Erinnerung: Ein schneller Weg, das Negativ einer gegebenen Zahl in Zweierkomplement-Darstellung zu finden, besteht darin, alle Bits zu invertieren und eins zu addieren. Also ist -43 11010101 in binärer Darstellung. Beachten Sie auch hier, dass das letzte Bit 1 ist und die Zahl ungerade ist. (Beachten Sie, dass dies bei Verwendung des Einerkomplements nicht stimmen würde!)

Schauen wir uns nun die gerade ganze Zahl 98 an. In binärer Darstellung ist 98 1100010.

```
    01100010
&   00000001
    --------
    00000000
```

Nach der UND-Verknüpfung ist das Ergebnis 0. Das bedeutet, dass das Bit b0 der ursprünglichen ganzen Zahl 98 0 war. Somit ist die gegebene ganze Zahl gerade.

Jetzt die negative -98. Es ist 10011110. Auch hier ist das Bit b0 0, nach der UND-Verknüpfung ist das Ergebnis 0, was bedeutet, dass -98 gerade ist, was ja auch stimmt.

Bit-Hack #2. Testen Sie, ob das n-te Bit gesetzt ist.

```
if (x & (1<<n)) {
  n-th bit is set
}
else {
  n-th bit is not set
}
```

Im vorherigen Bit-Hack haben wir gesehen, dass (x & 1) prüft, ob das erste Bit gesetzt ist. Dieser Bit-Hack verbessert dieses Ergebnis und prüft, ob das n-te Bit gesetzt ist. Dazu wird das erste 1-Bit um n Positionen nach links verschoben und dann die gleiche UND-Verknüpfung durchgeführt, die alle Bits außer dem n-ten eliminiert.

So sieht es aus, wenn man 1 um mehrere Positionen nach links verschiebt:

```
1         00000001    (same as 1<<0)
1<<1      00000010
1<<2      00000100
1<<3      00001000
1<<4      00010000
1<<5      00100000
1<<6      01000000
1<<7      10000000
```

Wenn wir nun 'x' mit einer um n Positionen nach links verschobenen 1 UND-verknüpfen, eliminieren wir effektiv alle Bits außer dem n-ten Bit in 'x'. Wenn das Ergebnis nach der UND-Verknüpfung 0 ist, dann muss dieses Bit 0 gewesen sein, andernfalls war das Bit gesetzt.

Schauen wir uns einige Beispiele an.

Ist bei 122 das 3. Bit gesetzt? Die Operation, die wir durchführen, um das herauszufinden, ist:

```
122 & (1<<3)
```

Nun, 122 ist 01111010 in binärer Form. Und (1<<3) ist 00001000.

```
    01111010
&   00001000
    --------
    00001000
```

Wir sehen, dass das Ergebnis nicht 0 ist, also ja, 122 hat das 3.

Hinweis: In meinem Artikel beginnt die Bit-Zählung mit 0. Es ist also das 0.

Was ist mit -33? Ist bei ihr das 5. Bit gesetzt?

```
    11011111      (-33 in binary)
&   00100000     (1<<5)
    --------
    00000000
```

Das Ergebnis ist 0, also ist das 5. Bit nicht gesetzt.

Bit-Hack #3. Setze das n-te Bit.

```
y = x | (1<<n)
```

Dieser Bit-Hack kombiniert den gleichen (1<<n) Trick des Setzens des n-ten Bits durch Verschieben mit der ODER-Operation. Das Ergebnis der ODER-Verknüpfung einer Variablen mit einem Wert, bei dem das n-te Bit gesetzt ist, ist das Einschalten des n-ten Bits. Das liegt daran, dass die ODER-Verknüpfung eines beliebigen Wertes mit 0 den Wert unverändert lässt; aber die ODER-Verknüpfung mit 1 ändert ihn in 1 (wenn er es nicht schon war). Schauen wir uns an, wie das in der Praxis funktioniert:

Angenommen, wir haben den Wert 120 und wollen das 2. Bit einschalten.

```
    01111000    (120 in binary)
|   00000100    (1<<2)
    --------
    01111100
```

Was ist mit -120 und 6. Bit?

```
    10001000   (-120 in binary)
|   01000000   (1<<6)
    --------
    11001000
```

Bit-Hack #4. Löse das n-te Bit.

```
y = x & ~(1<<n)
```

Der wichtigste Teil dieses Bithacks ist der ~(1<<n) Trick. Er schaltet alle Bits außer dem n-ten ein.

So sieht es aus:

```
 ~1        11111110  (same as ~(1<<0))
~(1<<1)   11111101
~(1<<2)   11111011
~(1<<3)   11110111
~(1<<4)   11101111
~(1<<5)   11011111
~(1<<6)   10111111
~(1<<7)   01111111
```

Die Wirkung der UND-Verknüpfung der Variablen 'x' mit dieser Menge ist die Eliminierung des n-ten Bits. Es spielt keine Rolle, ob das n-te Bit 0 oder 1 war, die UND-Verknüpfung mit 0 setzt es auf 0.

Hier ist ein Beispiel. Wir wollen das 4. Bit in 127 löschen:

```
    01111111    (127 in binary)
&   11101111    (~(1<<4))
    --------
    01101111
```

Bit-Hack #5. Schalten Sie das n-te Bit um.

```
y = x ^ (1<<n)
```

Dieser Bit-Hack verwendet ebenfalls den wunderbaren „set n-th bit shift hack“, aber dieses Mal wird er mit der Variablen 'x' XOR-verknüpft. Das Ergebnis der XOR-Verknüpfung von etwas mit etwas anderem ist, dass, wenn beide Bits gleich sind, das Ergebnis 0 ist, andernfalls ist es 1. Wie wird das n-te Bit umgeschaltet? Nun, wenn das n-te Bit 1 war, dann wird es durch XOR-Verknüpfung mit 1 zu 0; umgekehrt, wenn es 0 war, dann wird es durch XOR-Verknüpfung mit 1 zu 1. Sehen Sie, das Bit wurde umgedreht.

Hier ist ein Beispiel. Angenommen, Sie möchten das 5. Bit im Wert 01110101 umschalten:

```
    01110101
^   00100000
    --------
    01010101
```

Was ist mit dem gleichen Wert, aber das 5. Bit war ursprünglich 0?

```
    01010101
^   00100000
    --------
    01110101
```

Ist Ihnen etwas aufgefallen? Die XOR-Verknüpfung desselben Bits ergab zweimal denselben Wert. Diese raffinierte XOR-Eigenschaft wird bei der Berechnung der Parität in RAID-Arrays und in einfachen Kryptographie-Zyphern verwendet, aber dazu mehr in einem anderen Artikel.

Bit-Hack #6. Schalten Sie das ganz rechte 1-Bit aus.

```
y = x & (x-1)
```

Jetzt wird es endlich interessanter!!! Die Bit-Hacks #1 - #5 waren ehrlich gesagt ziemlich langweilig.

Dieser Bit-Hack schaltet das ganz rechte 1-Bit aus. Wenn man zum Beispiel eine ganze Zahl 00101010 (das ganz rechte 1-Bit in Fettdruck) gibt, wird sie zu 00101000. Oder bei 00010000 wird daraus eine 0, da es nur ein einziges 1-Bit gibt.

Hier sind weitere Beispiele:

```
    01010111    (x)
&   01010110    (x-1)
    --------
    01010110

    01011000    (x)
&   01010111    (x-1)
    --------
    01010000

    10000000    (x = -128)
&   01111111    (x-1 = 127 (with overflow))
    --------
    00000000

    11111111    (x = all bits 1)
&   11111110    (x-1)
    --------
    11111110

    00000000    (x = no rightmost 1-bits)
&   11111111    (x-1)
    --------
    00000000
```

Warum funktioniert das?

Wenn Sie sich die Beispiele ansehen und eine Weile darüber nachdenken, werden Sie feststellen, dass es zwei mögliche Szenarien gibt:

Der Wert hat das Bit ganz rechts 1. In diesem Fall werden durch die Subtraktion von 1 alle niedrigeren Bits auf 1 gesetzt und das Bit ganz rechts wird auf 0 gesetzt (so dass man, wenn man jetzt 1 addiert, wieder den ursprünglichen Wert erhält). Durch diesen Schritt wurde das äußerste rechte 1-Bit ausgeblendet, und durch UND-Verknüpfung mit dem ursprünglichen Wert wird dieses äußerste rechte 1-Bit nun auf Null gesetzt.

Der Wert hat kein 1-Bit ganz rechts (alle 0). In diesem Fall führt die Subtraktion von 1 zu einem Unterlauf des Wertes (da er vorzeichenbehaftet ist) und setzt alle Bits auf 1. Die UND-Verknüpfung aller Nullen mit allen Einsen ergibt 0.

Bit-Hack #7. Isoliere das ganz rechte 1-Bit.

```
y = x & (-x)
```

Dieser Bit-Hack findet das ganz rechte 1-Bit und setzt alle anderen Bits auf 0. Das Endergebnis hat nur dieses eine ganz rechte 1-Bit gesetzt. So wird z. B. 01010100 (ganz rechts in Fettdruck) in 00000100 umgewandelt.

Hier sind einige weitere Beispiele:

```
    10111100  (x)
&   01000100  (-x)
    --------
    00000100

    01110000  (x)
&   10010000  (-x)
    --------
    00010000

    00000001  (x)
&   11111111  (-x)
    --------
    00000001

    10000000  (x = -128)
&   10000000  (-x = -128)
    --------
    10000000

    11111111  (x = all bits one)
&   00000001  (-x)
    --------
    00000001

    00000000  (x = all bits 0, no rightmost 1-bit)
&   00000000  (-x)
    --------
    00000000
```

Dieser Bit-Hack funktioniert wegen des Zweierkomplements. Im Zweierkomplement-System ist -x dasselbe wie ~x+1. Betrachten wir nun die beiden möglichen Fälle:

1. Es gibt ein 1-Bit-Bi ganz rechts. In diesem Fall konzentrieren wir uns auf dieses Bit und teilen alle anderen Bits in zwei Flanken auf - Bits rechts und Bits links. Erinnere dich daran, dass alle Bits rechts

von bi-1, bi-2 ... b0 0 sind (weil bi das äußerste rechte 1-Bit war). Und die Bits auf der linken Seite sind so, wie sie sind. Nennen wir sie bi+1, ..., bn.

Wenn wir nun -x berechnen, machen wir zuerst ~x, was das Bit bi zu 0, die Bits bi-1 ... b0 zu 1en macht und die Bits bi+1, ..., bn invertiert, und dann addieren wir 1 zu diesem Ergebnis.

Da die Bits bi-1 ... b0 alle 1en sind, werden sie durch das Hinzufügen einer 1 bis zum Bit bi übertragen, das das erste Nullbit ist.

Wenn wir alles zusammenzählen, ist das Ergebnis der Berechnung von -x, dass die Bits bi+1, ..., bn invertiert werden, das Bit bi gleich bleibt und die Bits bi-1, ..., b0 alle 0 sind.

Die UND-Verknüpfung von x mit -x führt dazu, dass die Bits bi+1, ..., bn alle 0 sind, das Bit bi bleibt unverändert, und die Bits bi-1, ..., b0 werden auf 0 gesetzt.

2. Es gibt kein 1-Bit ganz rechts. Der Wert ist 0. Das Negativ von 0 im Zweierkomplement ist ebenfalls 0. 0&0 = 0. Keine Bits werden eingeschaltet.

Wir haben eindeutig bewiesen, dass dieser Bithack korrekt ist.

Bit-Hack #8. Propagiere das ganz rechte 1-Bit nach rechts.

```
y = x | (x-1)
```

Dies lässt sich am besten anhand eines Beispiels erklären. Bei einem Wert 01010000 wird er in 01011111 umgewandelt. Alle 0-Bits bis zum ganz rechten 1-Bit werden in Einsen umgewandelt.

Dies ist jedoch kein sauberer Hack, da er alle 1en erzeugt, wenn x = 0 ist.

Schauen wir uns weitere Beispiele an:

```
    10111100  (x)
|   10111011  (x-1)
    --------
    10111111

    01110111  (x)
|   01110110  (x-1)
    --------
    01110111

    00000001  (x)
|   00000000  (x-1)
    --------
    00000001

    10000000  (x = -128)
|   01111111  (x-1 = 127)
    --------
    11111111

    11111111  (x = -1)
|   11111110  (x-1 = -2)
    --------
    11111111

    00000000  (x)
|   11111111  (x-1)
    --------
```

```
    11111111
```

Beweisen wir es, wenn auch nicht so rigoros wie im vorigen Bithack (da es zu zeitaufwendig ist und dies keine wissenschaftliche Veröffentlichung ist). Es gibt wieder zwei Fälle. Beginnen wir mit dem einfachsten Fall.

3. Es gibt kein 1-Bit ganz rechts. In diesem Fall ist x = 0 und x-1 ist -1. -1 im Zweierkomplement ist 11111111. Die ODER-Verknüpfung von 0 mit 11111111 ergibt das gleiche 11111111. (Nicht das gewünschte Ergebnis, aber so ist es nun einmal).
4. Das ist das ganz rechte 1-Bit bi. Teilen wir wieder alle Bits in zwei Gruppen auf (wie im vorherigen Beispiel). Bei der Berechnung von x-1 werden nur die Bits rechts verändert, so dass bi zu 0 wird und alle niedrigeren Bits zu 1en. Die ODER-Verknüpfung von x mit x-1 lässt alle höheren Bits (links) unverändert, lässt das Bit bi so, wie es war: 1, und da die niedrigeren Bits alle niedrige 1en sind, werden auch sie eingeschaltet. Das Ergebnis ist, dass das ganz rechte 1-Bit an die Bits niedrigerer Ordnung weitergegeben wurde.

Bit-Hack #9. Isoliere das äußerste rechte 0-Bit.

```
y = ~x & (x+1)
```

Dieser Bithack macht das Gegenteil von #7. Er findet das 0-Bit ganz rechts, schaltet alle Bits aus und setzt dieses Bit im Ergebnis auf 1. Zum Beispiel findet er die fett gedruckte Null in dieser Zahl 10101011 und erzeugt 00000100.

Weitere Beispiele:

```
    10111100  (x)
    --------
    01000011  (~x)
&   10111101  (x+1)
    --------
    00000001

    01110111  (x)
    --------
    10001000  (~x)
&   01111000  (x+1)
    --------
    00001000

    00000001  (x)
    --------
    11111110  (~x)
&   00000010  (x+1)
    --------
    00000010

    10000000  (x = -128)
    --------
    01111111  (~x)
&   10000001  (x+1)
    --------
    00000001

    11111111  (x = no rightmost 0-bit)
```

```
    --------
    00000000  (~x)
&   00000000  (x+1)
    --------
    00000000

    00000000  (x)
    --------
    11111111  (~x)
&   00000001  (x+1)
    --------
    00000001
```

Der Beweis: Angenommen, es gibt ein 0-Bit ganz rechts. Dann verwandelt ~x dieses äußerste rechte 0-Bit in ein 1-Bit. Das gilt auch für x+1 (denn die Bits weiter rechts vom 0-Bit sind 1en). Durch UND-Verknüpfung von ~x mit x+1 werden nun alle Bits bis zu diesem 0-Bit ganz rechts verdampft. Dies ist das höchstwertige Bit, das im Ergebnis gesetzt wird. Was ist nun mit den Bits niedrigerer Ordnung rechts von der äußersten rechten 0? Sie wurden ebenfalls verdampft, weil x+1 sie in 0s verwandelte (sie waren 1s) und ~x sie in 0s verwandelte. Sie wurden mit 0 UND-verknüpft und verdampft.

Bit-Hack #10. Schalte das äußerste rechte 0-Bit ein.

```
y = x | (x+1)
```

Dieser Hack ändert das ganz rechte 0-Bit in 1. Zum Beispiel wird aus einer ganzen Zahl 10100011 die Zahl 10100111.

Weitere Beispiele:

```
    10111100  (x)
|   10111101  (x+1)
    --------
    10111101

    01110111  (x)
|   01111000  (x+1)
    --------
    01111111

    00000001  (x)
|   00000010  (x+1)
    --------
    00000011

    10000000  (x = -128)
|   10000001  (x+1)
    --------
    10000001

    11111111  (x = no rightmost 0-bit)
|   00000000  (x+1)
    --------
    11111111

    00000000  (x)
```

```
|   00000001  (x+1)
    --------
    00000001
```

Hier ist der Beweis in Form einer Reihe von wahren Aussagen. Bei der ODER-Verknüpfung von x mit x+1 geht keine Information verloren. Das Addieren von 1 zu x füllt die erste 0 ganz rechts. Das Ergebnis ist max{x, x+1}. Wenn x+1 überläuft, ist es x und es gab keine 0-Bits. Wenn nicht, ist es x+1, dessen rechtes Bit gerade mit 1 aufgefüllt wurde.

Hackers Vergnügen

Es gibt ein 300 Seiten langes Buch, das sich ausschließlich mit Bit-Hacks wie diesen beschäftigt. Es heißt Hacker's Delight. Werfen Sie einen Blick hinein. Wenn Ihnen der Inhalt meines Beitrags gefallen hat, dann werden Sie dieses Buch lieben.

Bonusmaterial

Wenn Sie sich entscheiden, mehr mit diesen Hacks zu spielen, finden Sie hier ein paar Dienstprogramme zum Drucken von Binärwerten von 8-Bit-Ganzzahlen mit Vorzeichen in Perl, Python und C.

Binäre Darstellung in Perl drucken:

```
sub int_to_bin {
  my $num = shift;
  print unpack "B8", pack "c", $num;
}
```

Oder Sie können es gleich von der Kommandozeile aus drucken:

```
perl -wle 'print unpack "B8", pack "c", shift' <integer>

# For example:
perl -wle &#39;print unpack &#34;B8&#34;, pack &#34;c&#34;, shift&#39; 113
01110001

perl -wle &#39;print unpack &#34;B8&#34;, pack &#34;c&#34;, shift&#39;
&#45;&#45; -128
10000000
```

Drucken von Binärzahlen in Python:

```
def int_to_bin(num, bits=8):
 r = ''
 while bits:
  r = ('1' if num&1 else '0') + r
  bits = bits - 1
  num = num >> 1
 print r
```

Binäre Darstellung in C drucken:

```
void int_to_bin(int num) {
  char str[9] = {0};
  int i;
  for (i=7; i>=0; i--) {
    str[i] = (num&1)?'1':'0';
    num >>= 1;
  }
  printf("%s\n", str);
}
```

Viel Spaß damit. Ich werde als nächstes über fortgeschrittene Bit-Hacks schreiben. Bis dann!

bithacks.h - bit hacks header file (catonmat.net)

https://catonmat.net/bit-hacks-header-file