Hacker's Delight

Lucas Forster*

Seminar *"Crazy Ideas in Data Structures and Algorithms"*
Theoretical Computer Science, RWTH Aachen University†

February 28, 2019

Abstract

This essay serves as an in-depth introduction to the art of bit-manipulating operations. The introduction covers bitwords (bitvectors) as well as logical operators and their representation as bitwise operands in a (high-level) programming language.

The focus lies on establishing a visual approach to familiar operations like incrementing a value. After covering manipulation of rightmost bits, those operators will be combined to form more powerful formulas. They will then be wrapped up as the "right to left computability class", which contains all bitword manipulations that can be expressed with a few elementary operations.

Finally, a method that reverses bitwords will be discussed. It has been selected to augment the possibilities of "right to left"-operations.

*lucas.forster@rwth-aachen.de
†tcs.rwth-aachen.de

1 Hacker's Delight

> "There is nothing here about hacking in the sense of breaking into computers."
>
> *Hacker's Delight* Website [1]

The Book

Henry Warren's book *Hacker's Delight* is described by himself as "a collection of small programming tricks" [2] (p. `xv`). In fact, the table of contents reads like an encyclopedia of formulas for bit level manipulation:

It starts out with detecting, counting and changing specific bits in a bitword. From there on, everything imaginable seems to be part of this collection. This also includes bit-level manipulations that result in a mathematical meaning (interpreting the bitword as a number).

The bypassing of common algebraic steps to achieve performance gains is the typical example of what should be considered a "hack". Concretely, like the book this paper exploits bit-level parallelism of instructions (e.g. a logical function is calculated for all bits of a bitword at the same time).

The Book's Title

As addressed in the epigraph, "hacker" refers to someone enjoying clever ways of coding, not necessarily being a professional or creating useful work. The "delight" is developed when faced with the elegance of some solutions.

This Paper

Inspired by and named after *Hacker's Delight*, this paper complements it with explanations that have been left out by the author (intentionally):

"The presentation is informal. Proofs are given only when the algorithm is not obvious, and sometimes not even then." [2]

Instead of simply explaining given formulas, this essay assembles them from a few elementary operations in a stepwise manner. Besides a fundamental understanding of the covered topics, the aim is to provide the reader with methods that can be adapted and thereby used to create new formulas.

2 Introduction

> "It is amazing what can be done with just [...] some bitwise operations."
>
> *Hacker's Delight*, page `xiii` [2]

Crazy Ideas in Data Structures and Algorithms

The seminar this contribution is part of covers ideas in computer science with a certain "craziness". This might not obviously apply to this paper:

The underlying data structure is a bitword – the simplest interpretation of binary values possible. Furthermore, all formulas are composed of simple logical operators, without any complex (e.g. recursive) nesting involved.

Where the "craziness" starts however, is in the rejection of abstract data types (like boolean-arrays). Addressing single bits isn't directly possible. Instead "hacks" will be used that involve decoupling operations (e.g. increment) from their mathematical meaning (see section 3). Additionally, instruction-level parallelism will allow for "crazy" fast computation: everything fitting in a bitword is processed by the CPU in a single time step.

Bitwords

The common term *bitword* (or *bitvector*). It is simply defined as a sequence of bits with fixed length that is indexed (starting with 0) from right to left (e.g. in Table 1 "↑" refers to x_4). There isn't any further interpretation (e.g. as a numerical value) to it. This holds true when applying operations with such a meaning (e.g. incrementing).

A good way to think of a bitvector is a series of *on* (`1`) and *off* (`0`) values. Terms like "rightmost" and "trailing" are used in their conventional meaning (see Table 1) – others will be introduced when needed.

$$\texttt{x = 1100 1011 0011 } \underbrace{\texttt{0000}}_{\text{trailing 0s}}$$
$$\uparrow$$
$$\text{rightmost 1}$$

Table 1: Bitword

Logical Operators

The mathematical logical operators serve as a basis for all bit manipulation. They are interpreted in the well-known way (see Table 2).

x	y	NOT: $\neg$ x	OR: x $\vee$ y	AND: x $\wedge$ y
0	0	1	0	0
0	1	1	1	0
1	0	0	1	0
1	1	0	1	1

Table 2: Operators NOT, OR, AND

The set $\{\neg, \vee, \wedge\}$ presented in Table 2 is considered a *complete set of logical operators*, which means that any logical formula can be expressed only using elements from this set. There is an important distinction to be made on how to apply a logical operator to a bitword (see Table 3 for *C*):

byte-level: This common interpretation defines the whole variable (at least 1 byte in size) as a single boolean value being false iff all bits are 0.

bitwise: This lesser used version is made for bit-level manipulation. As the name suggests every bit of the argument is treated independently.

function	mathematical	*C*: byte-level	*C*: bitwise
NOT	$\neg$	!	
OR	$\vee$	||	|
AND	$\wedge$	&&	&

Table 3: Byte-level and bitwise operators in *C*

Mathematical Operators

In addition to manipulating bits independently, operators that interpret sequence as a whole a required. Increment (INC, (x+1) in *C*) and decrement (DEC, (x-1) in *C*) do so by treating the bitwords in a mathematical sense. The assigning notations x++ and x-- are ignored to avoid side effects.

When executing these operation, *carry* and *borrow* refer to the values that get passed to adjacent bits (see section 3).

3 A Visual Approach to Operations

> "The fact that the carry chain allows a single bit to affect all the bits to its left makes addition a peculiarly powerful data manipulation operation"
>
> *Hacker's Delight*, page `xiv` [2]

As mentioned in section 1, the goal is to complement *Hacker's Delight*: chapter 2 of the book simply lists formulas. While it is possible to understand why they work as they do, it is a completely different task to create these on your own – this needs a systematic approach.

In order to develop the interpretations described by Henry Warren, a *visual approach* to operations has to be obtained first. Consider the `INC` operator as an example: Similar to performing addition by hand, the value `1` gets added onto the least significant bit. In the case of performing `1+1`, the overflow gets carried over to the next bit.

This repeats until the first `0` is reached. So starting from the right, every bit gets inverted until the first `0`-bit is found. This latter description is considered the visual interpretation of `INC`. `DEC` follows an analog pattern to the presented `INC` operator. This allows for the visual interpretations shown in Table 4 and Table 5.

`x`	left of rightmost `0`	rightmost `0`	trailings `1`s
`(x+1)`	unchanged	`1`	`0...0`
	invert all bits up to the rightmost `0`		

Table 4: `INC` (visual interpretation)

`x`	left of rightmost `1`	rightmost `1`	trailings `0`s
`(x-1)`	unchanged	`0`	`1...1`
	invert all bits up to the rightmost `1`		

Table 5: `DEC` (visual interpretation)

After the now covered mathematical operators from section 2, the bitwise logical operators are of interest. `NOT` is the trivial one to visually interpret, because it simply inverts all bits of a given bitword.

Regarding `OR` and `AND`, the approach is to reduce the original definition of calculating $\texttt{r}_i$ = $\texttt{x}_i \vee \texttt{y}_i$ and $\texttt{r}_i$ = $\texttt{x}_i \wedge \texttt{y}_i$ respectively for each bit i.

Instead, one might think of the second argument manipulating the first one to become the result: Set `r` to `x`, then for every `1` in `y` set the corresponding bit in `r` to `1`. Set to `0` for every `0` in `y` when executing `AND`. This is summarised in Table 6 and Table 7.

y	bits that are 0	bits that are 1
(x|y)	x (unchanged)	1
	for each 1 in y, set to 1 in x	

Table 6: `OR` (visual interpretation)

y	bits that are 0	bits that are 1
(x&y)	0	x (unchanged)
	for each 0 in y, set to 0 in x	

Table 7: `AND` (visual interpretation)

Notice that there was made a specific choice in Table 6 and Table 7 to let the second argument `y` manipulate the first argument `x` instead of the other way round. This will enhance readability in formulas presented in section 4.

4 Combining Operators

Two types of operators have been introduced in section 2:

- Operators that manipulate bits independently in parallel: `OR`, `AND`
- Operators that chain through multiple bits of the argument: `INC`, `DEC`

Combining pairs from these two sets will allow to create more complex formulas. `NOT` (part of the first group) is left out, since it would simply invert the result and thereby not contributing to interesting results. It will be useful as a third operator later on.

Combining two operators

Working a single bitwords `x`, we can manipulate `x` with `INC`, `DEC` and can combine the result with the original argument to add or remove certain bits of `x` using `OR`, `AND`.

`x AND (DEC x)` for example inverts all bits up to the rightmost `1` and then turns off every bit in `x` that hasn't been inverted. This means every bit up to the rightmost `1` has been turned off. This is equivalent to turning off the rightmost `1` (on the right are only trailing `0`s anyway).

Other combinations follow a similar pattern as demonstrated in Table 8, Table 9, Table 10 and Table 11.

`x`	left of rightmost `1`	rightmost `1`	trailing `0`s
`(x-1)`	unchanged	`0`	`1...1`
`x&(x-1)`	unchanged	`0`	unchanged
	turn off the rightmost `1`		

Table 8: `x AND (DEC x)`

By switching the two operators for their counterparts `INC` and `OR`, the meaning of the formula gets switched as well.

`x`	left of rightmost `0`	rightmost `0`	trailing `1`s
`(x+1)`	unchanged	`1`	`0...0`
`x|(x+1)`	unchanged	`1`	unchanged
	turn on the rightmost `0`		

Table 9: `x OR (INC x)`

Using the defined sets of operators, two other combinations are possible, which handle multiple (trailing) bits instead of a single (rightmost) one.

`x`	left of rightmost `0`	rightmost `0`	trailing `1`s
`(x+1)`	unchanged	`1`	`0...0`
`x&(x+1)`	unchanged	unchanged	`0...0`
	turn off trailing `0`s		

Table 10: `x AND (INC x)`

Again switching the two operators for their counterparts `DEC` and `OR` leads to the meaning of the formula getting switched as well.

`x`	left of rightmost `1`	rightmost `1`	trailing `0`s
`(x-1)`	unchanged	`0`	`1...1`
`x|(x-1)`	unchanged	unchanged	`1...1`
	turn on trailing `0`s		

Table 11: `x OR (DEC x)`

Zero-Testing

The two formulas build with `AND` turn off bits. Therefore their result can be zero-tested (in a meaningful way):

- `x AND (DEC x)`:
  `!(x&(x-1))`≡"is `x` 0 or a power of 2"
- `x AND (INC x)`:
  `!(x&(x+1))`≡"is `x` $2^m - 1, m \in [0, n]$" (for a bitword of length n)

Combining Three Operators

`NOT` has been left out at the beginning. With the four above introduced formulas however, new possibilities arise. There are three options to use the negation on them:

1. invert the full formula:
   `(x&(x-1))`, `(x|(x+1))`, `(x&(x+1))`, `(x|(x-1))`
   this just inverts the original result, meaning that:
   up to the rightmost `0`/`1` the turning on/off gets swapped
   and additionally the left part is inverted
2. invert the incremented/decrement argument:
   `x&( (x-1))`, `x|( (x+1))`, `x&( (x+1))`, `x|( (x-1))`
   this simply turns on/off the section left of the rightmost `0`/`1`
3. invert the original argument:
   `( x)&(x-1)`, `( x)|(x+1)`, `( x)&(x+1)`, `( x)&(x-1)`

While the first two options deliver foreseeable results, the inverting of the unchanged argument is the most complex and promising option, as demonstrated in Table 12, Table 13, Table 14 and Table 15.

x	left of rightmost 0	rightmost 0	trailing 1s
(x)	inverted	1	0...0
(x+1)	unchanged	1	0...0
(x)&(x+1)	0...0	1	0...0
	a single 1 at the rightmost 0		

Table 12: (NOT x) AND (INC x)

x	left of rightmost 1	rightmost 1	trailing 0s
(x)	inverted	1	0...0
(x-1)	unchanged	0	1...1
(x)|(x-1)	1...1	0	1...1
	a single 0 at the rightmost 1		

Table 13: (NOT x) OR (DEC x)

x	left of rightmost 1	rightmost 1	trailing 0s
(x)	inverted	1	1...1
(x-1)	unchanged	0	1...1
(x)&(x-1)	0...0	0	1...1
	1s at trailing 0s		

Table 14: (NOT x) AND (DEC x)

x	left of rightmost 0	rightmost 0	trailing 1s
(x)	inverted	1	1...1
(x+1)	unchanged	1	0...0
(x)|(x+1)	1...1	1	0...0
	0s at trailing 1s		

Table 15: (NOT x) OR (INC x)

Inverting

Additionally, every formula can be inverted in total. This is especially useful when the result is "a single `0`" like for `(NOT x) OR (DEC x)`. The result then has the same scheme as `(NOT x) AND (INC x)`.

5 Right to Left Computability Class

After having developed formulas modifying rightmost bits in section 4, the option arises to classify these kind of formulas. *Hacker's Delight* [2] therefore introduces the so called "right to left computability class". Every formula in this class shares the property, that each result bit only depends on input bits at its position or to the right of it (see Table 16).

x:	$\mathtt{x}_n$	...	$\boxed{\mathtt{x}_i\ \mathtt{x}_{i-1} \ldots \mathtt{x}_0}$
y:	$\mathtt{y}_n$	...	$\boxed{\mathtt{y}_i\ \mathtt{y}_{i-1} \ldots \mathtt{y}_0}$
result r:	$\mathtt{x}_n$	...	$\boxed{\mathtt{r}_i}\ \mathtt{r}_{i-} \ldots \mathtt{r}_{i-1}$

$\boxed{\mathtt{r}_i \text{ only depends on } \mathtt{x}_i \ldots \mathtt{x}_0 \text{ and } \mathtt{y}_i \ldots \mathtt{y}_0}$

Table 16: Right to left computability property

Notice that, as it is common for a lot of operators, the result bit $\mathtt{r}_i$ is allowed to depend on $\mathtt{r}_{i-1} \ldots \mathtt{r}_0$. This is already given since every of these result bits to the right of $\mathtt{r}_i$ only depend on bits of `x` and `y`.

Before defining the class, the previously used set of operators will get slightly modified: `INC` and `DEC` get exchanged for `ADD` and `SUB`. This will allow for more flexibility while not using more powerful operations: they can simply be expressed with loops. The instruction set is therefore:

$$\mathtt{NOT}, \underbrace{\mathtt{ADD}, \mathtt{SUB}}_{\mathtt{INC},\ \mathtt{DEC}}, \mathtt{AND}, \mathtt{OR}$$

It should also be mentioned, that this selection is a rational choice. Smaller sets would also be possible, and there are other right to left computable operators – they can be combined from the selected ones (e.g. shifting left as a sequence of `ADD`s).

> "THEOREM. A function mapping words to words can be implemented with world-parallel *add*, *subtract*, *or* and *not* instructions if and only if each bit of the result depends only on bits at and to the right of each input operand."
>
> *Hacker's Delight*, page 12 [2]

Proof.

" $\Longrightarrow$ ": In section 3 `INC` and `DEC` were established as right to left operations. This then also applies to `ADD` and `SUB`. Regarding `NOT`, `AND`, `OR`, they clearly fulfil the right to left computability property, since their result bits do not depend on any adjacent ones at all.

Thereby a formula using only `ADD`, `SUB`, `AND`, `OR` and `NOT` only depends on bits at and to the right of each input operand.

" $\Longleftarrow$ ": The result only depends on bits at and to the right of each input operand. These can be isolated using `AND` with the argument and a mask (a constant containing a single `1` at the respective position).

Once every required bit is isolated, the function can be expressed using only `NOT`, `OR` and `AND`. This is true because (as mentioned in section 2) $\{\neg, \vee, \wedge\}$ is a complete set of logical operators, every logical function can be expressed using only its elements.

□

The obtained class contains all functions that can calculate the nth result bit with only knowing the input operands up to their nth bit. This has applications e.g. in bitstreams when working on partial data.

6 Conclusion

The formulas introduced in the beginning of Hacker's Delight [2] form a basis for more complex operations in the book and have many applications in low level programming. This paper not only explained why they work the way they do, but moreover provided an approach to create them from scratch.

Readers who are involved with low level coding should be encouraged to systematically combine the presented operations in order to form a compact implementation of their desired function (instead of searching for it through

precompiled lists). From a theoretical standpoint, the presented formulas have been identified to be part of the right to left computability class, where all included functions can be expressed with a small set of operations.

A Exponentiation by Binary Decomposition

This section covers the computation of x^n for a nonnegative integer n as in section `11-3` of *Hackers's Delight* [2] (p. `288`). The base x can be of any type with an associative operation $\cdot$ defined on it. Therefore the method presented here is not only applicable to multiplication on integers and floating points, but also i.e. concatenation of strings (('ab')^3 = 'abababb').

The goal is to reduce the number of required operations with the most primitive approach taking $(n-1)$ steps. Whatever the operation may be, it will be referred to as "multiplication" below.

Using the identities $x^{(a+b)} = x^a \cdot x^b$ and $(x^a)^b = x^{(a \cdot b)}$, a decomposition of the exponent n can be achieved and thereby the number of required multiplications is reduced up to a minimum. Consider these examples[1]:

$$x^{15} = x^3 \cdot ((x^3)^2)^2 \qquad x^{27} = ((x^3)^3)^3$$

> "Perhaps surprisingly, there is no known simple method that, for all n, finds the optimal sequence of multiplications to compute x^n. The only known methods involve an extensive search."
>
> – Hacker's Delight [2] (p. `289`)

This circumstance leads to the question, which decomposition is simple to obtain and a relatively good approximation of the optimum. The binary representation of n is cheap when already operating in a low level coding environment. Consider the same examples as before:

$$x^{15} = x^8 \cdot x^4 \cdot x^2 \cdot x \qquad x^{27} = x^{16} \cdot x^8 \cdot x^2 \cdot x$$

In comparison, binary decomposition takes for $n := 15$ a total of 6 $(\log_2 8 + 3)$ instead of 5 $(2 + 1 + 1 + 1)$ operations and for $n := 27$ a total of 7 $(\log_2 16 + 3)$ instead of 6 $(2 + 2 + 2)$ multiplications.

[1] Henry Warren [2] mentions $x^{15} = (x^3)^5$ which requires 6 (one more) multiplications. The decomposition shown here was selected as a replacement to demonstrate that simply chaining prime factors isn't necessarily the optimal solution.

The algorithm for this computation[2] scans the binary representation from right to left, by isolating the least significant bit (`n&1`) and right-shifting it (`n = n >> 1`). For every new bit, a factor `p` (that starts as x) is squared and therefore equals $x, x^2, x^4, \ldots$ throughout the cycles. It is multiplied onto the final result `y` (which starts as 1) if and only if the bit in that cycle is `1`.

List of Tables

1 Bitword . . . 3
2 Operators `NOT`, `OR`, `AND` . . . 4
3 Byte-level and bitwise operators in C . . . 4
4 `INC` (visual interpretation) . . . 5
5 `DEC` (visual interpretation) . . . 5
6 `OR` (visual interpretation) . . . 6
7 `AND` (visual interpretation) . . . 6
8 `x AND (DEC x)` . . . 7
9 `x OR (INC x)` . . . 7
10 `x AND (INC x)` . . . 7
11 `x OR (DEC x)` . . . 8
12 `(NOT x) AND (INC x)` . . . 9
13 `(NOT x) OR (DEC x)` . . . 9
14 `(NOT x) AND (DEC x)` . . . 9
15 `(NOT x) OR (INC x)` . . . 9
16 Right to left computability property . . . 10

References

[1] H. S. Warren. *Hacker's Delight (Website).* https://hackersdelight.org/, 12.09.2018.

[2] H. S. Warren. *Hacker's Delight.* Addison-Wesley Professional, 2nd edition, 2012.

[2] See `https://hackersdelight.org/hdcodetxt/iexp.c.txt` [1]