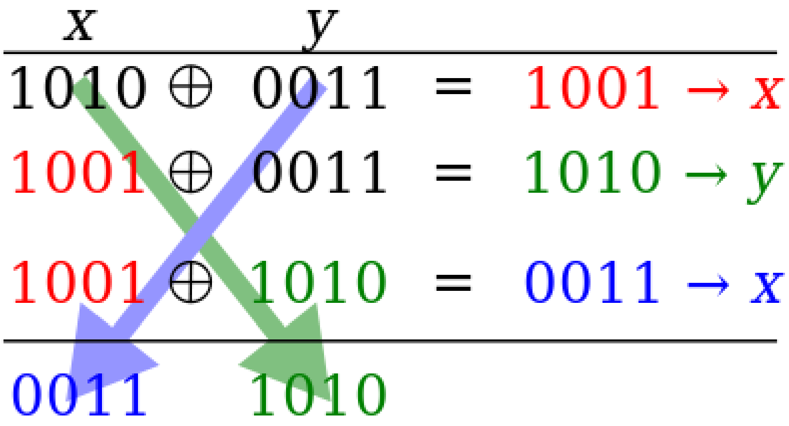
XOR swap algorithm

In computer programming, the **XOR swap** is an algorithm that uses the XOR bitwise operation to swap values of distinct variables having the same data type without using a temporary variable. "Distinct" means that the variables are stored at different memory addresses; the actual values of the variables do not have to be different.



Using the XOR swap algorithm to exchange nibbles between variables without the use of temporary storage

Contents

The algorithm

Conventional swapping requires the use of a temporary storage variable. Using the XOR swap algorithm, however, no temporary storage is needed. The algorithm is as follows:[1][2]

```
X := X XOR Y
Y := Y XOR X
X := X XOR Y
```

The algorithm typically corresponds to three machine code instructions. Since XOR is a commutative operation, X XOR Y can be replaced with Y XOR X in any of the lines. When coded in assembly language, this commutativity is often exercised in the second line:

Pseudocode	IBM System/370 assembly	x86 assembly
`X := X XOR Y`	`XR R1,R2`	`xor eax, ebx`
`Y := Y XOR X`	`XR R2,R1`	`xor ebx, eax`
`X := X XOR Y`	`XR R1,R2`	`xor eax, ebx`

In the above System/370 assembly code sample, R1 and R2 are distinct registers, and each XR operation leaves its result in the register named in the first argument. Using x86 assembly, values X and Y are in registers eax and ebx (respectively), and `xor` places the result of the operation in the first register.

However, the algorithm fails if *x* and *y* use the same storage location, since the value stored in that location will be zeroed out by the first XOR instruction, and then remain zero; it will not be "swapped with itself". Note that this is *not* the same as if *x* and *y* have the same values. The trouble only comes when *x* and *y* use the same storage location, in which case their values must already be equal. That is, if *x* and *y* use the same storage location, then the line:

```
X := X XOR Y
```

sets *x* to zero (because *x* = *y* so X XOR Y is zero) *and* sets *y* to zero (since it uses the same storage location), causing *x* and *y* to lose their original values.

Proof of correctness

The binary operation XOR over bit strings of length N exhibits the following properties (where ⊕ denotes XOR):[a]

- **L1.** Commutativity: $A \oplus B = B \oplus A$
- **L2.** Associativity: $(A \oplus B) \oplus C = A \oplus (B \oplus C)$
- **L3.** Identity exists: there is a bit string, 0, (of length *N*) such that $A \oplus 0 = A$ for any A
- **L4.** Each element is its own inverse: for each A, $A \oplus A = 0$.

Suppose that we have two distinct registers `R1` and `R2` as in the table below, with initial values *A* and *B* respectively. We perform the operations below in sequence, and reduce our results using the properties listed above.

Step	Operation	Register 1	Register 2	Reduction
0	Initial value	A	B	—
1	`R1 := R1 XOR R2`	$A \oplus B$	B	—
2	`R2 := R1 XOR R2`	$A \oplus B$	$(A \oplus B) \oplus B = A \oplus (B \oplus B)$ $= A \oplus 0$ $= A$	**L2** **L4** **L3**
3	`R1 := R1 XOR R2`	$(A \oplus B) \oplus A = A \oplus (A \oplus B)$ $= (A \oplus A) \oplus B$ $= 0 \oplus B$ $= B \oplus 0$ $= B$	A	**L1** **L2** **L4** **L1** **L3**

Linear algebra interpretation

As XOR can be interpreted as binary addition and a pair of values can be interpreted as a point in two-dimensional space, the steps in the algorithm can be interpreted as 2×2 matrices with binary values. For simplicity, assume initially that *x* and *y* are each single bits, not bit vectors.

For example, the step:

```
X := X XOR Y
```

which also has the implicit:

```
Y := Y
```

corresponds to the matrix $\left(\begin{smallmatrix}1 & 1\\ 0 & 1\end{smallmatrix}\right)$ as

$$\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}\begin{pmatrix}x\\ y\end{pmatrix} = \begin{pmatrix}x+y\\ y\end{pmatrix}.$$

The sequence of operations is then expressed as:

$$\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}\begin{pmatrix}1 & 0\\ 1 & 1\end{pmatrix}\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$$

(working with binary values, so $1 + 1 = 0$), which expresses the elementary matrix of switching two rows (or columns) in terms of the transvections (shears) of adding one element to the other.

To generalize to where X and Y are not single bits, but instead bit vectors of length *n*, these 2×2 matrices are replaced by 2*n*×2*n* block matrices such as $\left(\begin{smallmatrix}I_n & I_n\\ 0 & I_n\end{smallmatrix}\right)$.

Note that these matrices are operating on *values,* not on *variables* (with storage locations), hence this interpretation abstracts away from issues of storage location and the problem of both variables sharing the same storage location.

Code example

A C function that implements the XOR swap algorithm:

```c
void xorSwap (int *x, int *y) {
    if (x != y) {
        *x ^= *y;
        *y ^= *x;
        *x ^= *y;
    }
}
```

Note that the code does not swap the integers passed immediately, but first checks if their addresses are distinct. This is because, if the addresses are equal, the algorithm will fold to a triple *x ^= *x resulting in zero.

The code below is a nice example of overly concise C code. An anti-pattern nowadays. The behavior of the code is undefined. Writing overly concise C code has become unnecessary as modern optimizing compilers will eliminate intermediate results and temporary variables.

```c
/* kcc: "Undefined behavior - Error: EIO8
   Description: unsequenced side effect on scalar object with value computation of same object."
   See: https://github.com/kframework/c-semantics and https://runtimeverification.com/match/1.0-SNAPSHOT/docs/
   There are no sequence points to force the execution of (*y=*x) last - see https://en.wikipedia.org/wiki/Seq
*/
void xorSwap (int *x, int *y) {
    *x^=*y^(*y^=*x);
}
```

The XOR swap algorithm can also be defined with a macro:

```
#define XORSWAP_UNSAFE(a, b)    ((a)^=(b),(b)^=(a),(a)^=(b)) /* Doesn't work when a and b are the same object -
#define XORSWAP(a, b)   ((&(a) == &(b)) ? (a) : ((a)^=(b),(b)^=(a),(a)^=(b))) /* checks that the addresses of a
```

Reasons for use in practice

In most practical scenarios, the trivial swap algorithm using a temporary register is more efficient. Limited situations in which XOR swapping may be practical include:

- on a processor where the instruction set encoding permits the XOR swap to be encoded in a smaller number of bytes
- in a region with high register pressure, it may allow the register allocator to avoid spilling a register
- in microcontrollers where available RAM is very limited.
- in cryptographic applications which need constant time functions to prevent time-based side-channel attacks[3]

Because these situations are rare, most optimizing compilers do not generate XOR swap code.

Reasons for avoidance in practice

Most modern compilers can optimize away the temporary variable in the native swap, in which case the native swap uses the same amount of memory and the same number of registers as the XOR swap and is at least as fast, and often faster. The XOR swap is also much less readable and completely opaque to anyone unfamiliar with the technique.

On modern CPU architectures, the XOR technique can be slower than using a temporary variable to do swapping. One reason is that modern CPUs strive to execute instructions in parallel via instruction pipelines. In the XOR technique, the inputs to each operation depend on the results of the previous operation, so they must be executed in strictly sequential order, negating any benefits of instruction-level parallelism.[4]

A historical reason was that it used to be patented (US4197590). Even then, this was only for computer graphics.

Aliasing

The XOR swap is also complicated in practice by aliasing. As noted above, if an attempt is made to XOR-swap the contents of some location with itself, the result is that the location is zeroed out and its value lost. Therefore, XOR swapping must not be used blindly in a high-level language if aliasing is possible.

Similar problems occur with call by name, as in Jensen's Device, where swapping `i` and `A[i]` via a temporary variable yields incorrect results due to the arguments being related: swapping via `temp = i; i = A[i]; A[i] = temp` changes the value for `i` in the second statement, which then results in the incorrect i value for `A[i]` in the third statement.

Variations

The underlying principle of the XOR swap algorithm can be applied to any operation meeting criteria L1 through L4 above. Replacing XOR by addition and subtraction gives a slightly different, but largely equivalent, formulation:

```
void addSwap (unsigned int *x, unsigned int *y)
{
    if (x != y) {
        *x = *x + *y;
        *y = *x - *y;
        *x = *x - *y;
    }
}
```

Unlike the XOR swap, this variation requires that the underlying processor or programming language uses a method such as modular arithmetic or bignums to guarantee that the computation of `X + Y` cannot cause an error due to integer overflow. Therefore, it is seen even more rarely in practice than the XOR swap.

Note, however, that the implementation of `addSwap` above in the C programming language always works even in case of integer overflow, since, according to the C standard, addition and subtraction of unsigned integers follow the rules of modular arithmetic, i. e. are done in the cyclic group $\mathbb{Z}/2^s\mathbb{Z}$ where s is the number of bits of `unsigned int`. Indeed, the correctness of the algorithm follows from the fact that the formulas $(x+y)-y=x$ and $(x+y)-((x+y)-y)=y$ hold in any abelian group. This is actually a generalization of the proof for the XOR swap algorithm: XOR is both the addition and subtraction in the abelian group $(\mathbb{Z}/2\mathbb{Z})^s$.

Please note that the above doesn't hold when dealing with the `signed int` type (the default for `int`). Signed integer overflow is an undefined behavior in C and thus modular arithmetic is not guaranteed by the standard (a standard-conforming compiler might optimize out such code, which leads to incorrect results).

See also

- Symmetric difference
- XOR linked list
- Feistel cipher (the XOR swap algorithm is a degenerate form of a Feistel cypher)

Notes

a. The first three properties, along with the existence of an inverse for each element, are the definition of an abelian group. The last property is the statement that every element is an involution, that is, having order 2, which is not true of all abelian groups.

References

1. "The Magic of XOR" (http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/xor.html). Cs.umd.edu. Retrieved 2014-04-02.
2. "Swapping Values with XOR" (http://graphics.stanford.edu/~seander/bithacks.html#SwappingValuesXOR). graphics.stanford.edu. Retrieved 2014-05-02.
3. Schneier, Tadayoshi Kohno, Niels Ferguson, Bruce (2010). *Cryptography engineering : design principles and practical applications*. Indianapolis, IN: Wiley Pub., inc. p. 251 ff. ISBN 978-0-470-47424-2.
4. Amarasinghe, Saman; Leiserson, Charles (2010). "6.172 Performance Engineering of Software Systems, Lecture 2" (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2010/video-lectures/lecture-2-bit-hacks/). *MIT OpenCourseWare*. Massachusetts Institute of Technology. Retrieved 27 January 2015.

Retrieved from "https://en.wikipedia.org/w/index.php?title=XOR_swap_algorithm&oldid=803867060"

- This page was last edited on 5 October 2017, at 04:12.
-