Bit array

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(December 2010)*

A **bit array** (also known as **bitmap**, **bitset**, **bit string**, or **bit vector**) is an array data structure that compactly stores bits. It can be used to implement a simple set data structure. A bit array is effective at exploiting bit-level parallelism in hardware to perform operations quickly. A typical bit array stores *kw* bits, where *w* is the number of bits in the unit of storage, such as a byte or word, and *k* is some nonnegative integer. If *w* does not divide the number of bits to be stored, some space is wasted due to internal fragmentation.

Contents [hide]

1 Definition
2 Basic operations
3 More complex operations
3.1 Population / Hamming weight
3.2 Inversion
3.3 Find first one
4 Compression
5 Advantages and disadvantages
6 Applications
7 Language support
8 See also
9 References
10 External links

Definition [edit]

A bit array is a mapping from some domain (almost always a range of integers) to values in the set {0, 1}. The values can be interpreted as dark/light, absent/present, locked/unlocked, valid/invalid, et cetera. The point is that there are only two possible values, so they can be stored in one bit. As with other arrays, the access to a single bit can be managed by applying an index to the array. Assuming its size (or length) to be *n* bits, the array can be used to specify a subset of the domain (e.g. {0, 1, 2, ..., *n*−1}), where a 1-bit indicates the presence and a 0-bit the absence of a number in the set. This set data structure uses about *n*/*w* words of space, where *w* is the number of bits in each machine word. Whether the least significant bit (of the word) or the most significant bit indicates the smallest-index number is largely irrelevant, but the former tends to be preferred (on little-endian machines).

Basic operations [edit]

Although most machines are not able to address individual bits in memory, nor have instructions to manipulate single bits, each bit in a word can be singled out and manipulated using bitwise operations. In particular:

- OR can be used to set a bit to one: 11101010 OR 00000100 = 11101110
- AND can be used to set a bit to zero: 11101010 AND 11111101 = 11101000
- AND together with zero-testing can be used to determine if a bit is set:

  11101010 AND 00000001 = 00000000 = 0

  11101010 AND 00000010 = 00000010 ≠ 0

- XOR can be used to invert or toggle a bit:

  11101010 XOR 00000100 = 11101110

  11101110 XOR 00000100 = 11101010

- NOT can be used to invert all bits.

  NOT 10110010 = 01001101

To obtain the bit mask needed for these operations, we can use a bit shift operator to shift the number 1 to the left by the appropriate number of places, as well as bitwise negation if necessary.

Given two bit arrays of the same size representing sets, we can compute their union, intersection, and set-theoretic difference using *n*/*w* simple bit operations each (2*n*/*w* for difference), as well as the complement of either:

```
for i from 0 to n/w-1
    complement_a[i] := not a[i]
    union[i]        := a[i] or b[i]
    intersection[i] := a[i] and b[i]
    difference[i]   := a[i] and (not b[i])
```

If we wish to iterate through the bits of a bit array, we can do this efficiently using a doubly nested loop that loops through each word, one at a time. Only *n*/*w* memory accesses are required:

```
for i from 0 to n/w-1
    index := 0    // if needed
    word := a[i]
    for b from 0 to w-1
        value := word and 1 ≠ 0
        word := word shift right 1
        // do something with value
        index := index + 1    // if needed
```

Both of these code samples exhibit ideal locality of reference, which will subsequently receive large performance boost from a data cache. If a cache line is *k* words, only about *n*/*wk* cache misses will occur.

More complex operations [edit]

As with character strings it is straightforward to define *length*, *substring*, lexicographical *compare*, *concatenation*, *reverse* operations. The implementation of some of these operations is sensitive to endianness.

Population / Hamming weight [edit]

If we wish to find the number of 1 bits in a bit array, sometimes called the population count or Hamming weight, there are efficient branch-free algorithms that can compute the number of bits in a word using a series of simple bit operations. We simply run such an algorithm on each word and keep a running total. Counting zeros is similar. See the Hamming weight article for examples of an efficient implementation.

Inversion [edit]

Vertical flipping of a one-bit-per-pixel image, or some FFT algorithms, require to flip the bits of individual words (so `b31 b30 ... b0` becomes `b0 ... b30 b31`). When this operation is not available on the processor, it's still possible to proceed by successive passes, in this example on 32 bits:

```
exchange two 16bit halfwords
exchange bytes by pairs (0xddccbbaa -> 0xccddaabb)
...
swap bits by pairs
swap bits (b31 b30 ... b1 b0 -> b30 b31 ... b0 b1)

The last operation can be written ((x&0x55555555)<<1) | (x&0xaaaaaaaa)>>1)).
```

Find first one [edit]

The find first set or *find first one* operation identifies the index or position of the 1-bit with the smallest index in an array, and has widespread hardware support (for arrays not larger than a word) and efficient algorithms for its computation. When a priority queue is stored in a bit array, find first one can be used to identify the highest priority element in the queue. To expand a word-size *find first one* to longer arrays, one can find the first nonzero word and then run *find first one* on that word. The related operations *find first zero*, *count leading zeros*, *count leading ones*, *count trailing zeros*, *count trailing ones*, and *log base 2* (see find first set) can also be extended to a bit array in a straightforward manner.

Compression [edit]

A bit array is the densest storage for "random" bits, that is, where each bit is equally likely to be 0 or 1, and each one is independent. But most data is not random, so it may be possible to store it more compactly. For example, the data of a typical fax image is not random and can be compressed. Run-length encoding is commonly used to compress these long streams. However, most compressed data formats are not so easy to access randomly; also by compressing bit arrays too aggressively we run the risk of losing the benefits due to bit-level parallelism (vectorization). Thus, instead of compressing bit arrays as streams of bits, we might compress them as streams of bytes or words (see Bitmap index (compression)).

Advantages and disadvantages [edit]

Bit arrays, despite their simplicity, have a number of marked advantages over other data structures for the same problems:

- They are extremely compact; few other data structures can store n independent pieces of data in n/w words.
- They allow small arrays of bits to be stored and manipulated in the register set for long periods of time with no memory accesses.
- Because of their ability to exploit bit-level parallelism, limit memory access, and maximally use the data cache, they often outperform many other data structures on practical data sets, even those that are more asymptotically efficient.

However, bit arrays aren't the solution to everything. In particular:

- Without compression, they are wasteful set data structures for sparse sets (those with few elements compared to their range) in both time and space. For such applications, compressed bit arrays, Judy arrays, tries, or even Bloom filters should be considered instead.
- Accessing individual elements can be expensive and difficult to express in some languages. If random access is more common than sequential and the array is relatively small, a byte array may be preferable on a machine with byte addressing. A word array, however, is probably not justified due to the huge space overhead and additional cache misses it causes, unless the machine only has word addressing.

Applications [edit]

Because of their compactness, bit arrays have a number of applications in areas where space or efficiency is at a premium. Most commonly, they are used to represent a simple group of boolean flags or an ordered sequence of boolean values.

Bit arrays are used for priority queues, where the bit at index k is set if and only if k is in the queue; this data structure is used, for example, by the Linux kernel, and benefits strongly from a find-first-zero operation in hardware.

Bit arrays can be used for the allocation of memory pages, inodes, disk sectors, etc. In such cases, the term *bitmap* may be used. However, this term is frequently used to refer to raster images, which may use multiple bits per pixel.

Another application of bit arrays is the Bloom filter, a probabilistic set data structure that can store large sets in a small space in exchange for a small probability of error. It is also possible to build probabilistic hash tables based on bit arrays that accept either false positives or false negatives.

Bit arrays and the operations on them are also important for constructing succinct data structures, which use close to the minimum possible space. In this context, operations like finding the nth 1 bit or counting the number of 1 bits up to a certain position become important.

Bit arrays are also a useful abstraction for examining streams of compressed data, which often contain elements that occupy portions of bytes or are not byte-aligned. For example, the compressed Huffman coding representation of a single 8-bit character can be anywhere from 1 to 255 bits long.

In information retrieval, bit arrays are a good representation for the posting lists of very frequent terms. If we compute the gaps between adjacent values in a list of strictly increasing integers and encode them using unary coding, the result is a bit array with a 1 bit in the nth position if and only if n is in the list. The implied probability of a gap of n is $1/2^n$. This is also the special case of Golomb coding where the parameter M is 1; this parameter is only normally selected when $-\log(2-p)/\log(1-p) \leq 1$, or roughly the term occurs in at least 38% of documents.

Language support [edit]

The C programming language's *bitfields*, pseudo-objects found in structs with size equal to some number of bits, are in fact small bit arrays; they are limited in that they cannot span words. Although they give a convenient syntax, the bits are still accessed using bitwise operators on most machines, and they can only be defined statically (like C's static arrays, their sizes are fixed at compile-time). It is also a common idiom for C programmers to use words as small bit arrays and access bits of them using bit operators. A widely available header file included in the X11 system, xtrapbits.h, is “a portable way for systems to define bit field manipulation of arrays of bits.” A more explanatory description of aforementioned approach can be found in the comp.lang.c faq.

In C++, although individual `bool`s typically occupy the same space as a byte or an integer, the STL type `vector<bool>` is a partial template specialization in which bits are packed as a space efficiency optimization. Since bytes (and not bits) are the smallest addressable unit in C++, the [] operator does *not* return a reference to an element, but instead returns a proxy reference. This might seem a minor point, but it means that `vector<bool>` is *not* a standard STL container, which is why the use of `vector<bool>` is generally discouraged. Another unique STL class, `bitset`,[1] creates a vector of bits fixed at a particular size at compile-time, and in its interface and syntax more resembles the idiomatic use of words as bit sets by C programmers. It also has some additional power, such as the ability to efficiently count the number of bits that are set. The Boost C++ Libraries provide a `dynamic_bitset` class[2] whose size is specified at run-time.

The D programming language provides bit arrays in both of its competing standard libraries. In Phobos, they are provided in `std.bitmanip`, and in Tango, they are provided in `tango.core.BitArray`. As in C++, the [] operator does not return a reference, since individual bits are not directly addressable on most hardware, but instead returns a `bool`.

In Java, the class `BitSet` creates a bit array that is then manipulated with functions named after bitwise operators familiar to C programmers. Unlike the `bitset` in C++, the Java `BitSet` does not have a "size" state (it has an effectively infinite size, initialized with 0 bits); a bit can be set or tested at any index. In addition, there is a class `EnumSet`, which represents a Set of values of an enumerated type internally as a bit vector, as a safer alternative to bitfields.

The .NET Framework supplies a `BitArray` collection class. It stores boolean values, supports random access and bitwise operators, can be iterated over, and its `Length` property can be changed to grow or truncate it.

Although Standard ML has no support for bit arrays, Standard ML of New Jersey has an extension, the `BitArray` structure, in its SML/NJ Library. It is not fixed in size and supports set operations and bit operations, including, unusually, shift operations.

Haskell likewise currently lacks standard support for bitwise operations, but both GHC and Hugs provide a `Data.Bits` module with assorted bitwise functions and operators, including shift and rotate operations and an "unboxed" array over boolean values may be used to model a Bit array, although this lacks support from the former module.

In Perl, strings can be used as expandable bit arrays. They can be manipulated using the usual bitwise operators (`~ | & ^`),[3] and individual bits can be tested and set using the *vec* function.[4]

In Ruby, you can access (but not set) a bit of an integer (`Fixnum` or `Bignum`) using the bracket operator (`[]`), as if it were an array of bits.

Apple's Core Foundation library contains CFBitVector and CFMutableBitVector structures.

PL/I supports arrays of *bit strings* of arbitrary length, which may be either fixed-length or varying. The array elements may be *aligned*— each element begins on a byte or word boundary— or *unaligned*— elements immediately follow each other with no padding.

Hardware description languages such as VHDL, Verilog, and SystemVerilog natively support bit vectors as these are used to model storage elements like flip-flops, hardware busses and hardware signals in general. In hardware verification languages such as OpenVera, *e* and SystemVerilog, bit vectors are used to sample values from the hardware models, and to represent data that is transferred to hardware during simulations.

See also [edit]

- Bit field
- Bitboard Chess and similar games.
- Bitmap index
- Binary numeral system
- Bitstream
- Judy array

References [edit]

1. ^ std::bitset
2. ^ boost::dynamic_bitset
3. ^ http://perldoc.perl.org/perlop.html#Bitwise-String-Operators
4. ^ http://perldoc.perl.org/functions/vec.html

External links [edit]

- mathematical bases by Pr. D.E.Knuth
- vector<bool> Is Nonconforming, and Forces Optimization Choice
- vector<bool>: More Problems, Better Solutions

v · t · e	Data structures [hide]
Types	Collection · Container
Abstract	Associative array · Double-ended priority queue · Double-ended queue · List · Map · Multimap · Priority queue · Queue · Set (multiset) · Disjoint Sets · Stack
Arrays	**Bit array** · Circular buffer · Dynamic array · Hash table · Hashed array tree · Sparse array
Linked	Association list · Linked list · Skip list · Unrolled linked list · XOR linked list
Trees	B-tree · Binary search tree (AA · AVL · red-black · self-balancing · splay) · Heap (binary · binomial · Fibonacci) · R-tree (R* · R+ · Hilbert) · Trie (Hash tree)
Graphs	Binary decision diagram · Directed acyclic graph · Directed acyclic word graph
	List of data structures

Categories: Arrays | Bit data structures