# Using de Bruijn Sequences to Index a 1 in a Computer Word

Charles E. Leiserson
Harald Prokop
Keith H. Randall
MIT Laboratory for Computer Science, Cambridge, MA 02139, USA
{cel,prokop,randall}@lcs.mit.edu

July 7, 1998

**Abstract**

Some computers provide an instruction to find the index of a 1 in a computer word, but many do not. This paper provides a fast and novel algorithm based on de Bruijn sequences to solve this problem. The algorithm involves little more than an integer multiply and a lookup in a small table. We compare the performance of our algorithm with other popular strategies that use table lookups or floating-point conversion.

## 1 Introduction

Many applications that use one-word bit vectors require the ability to find the binary index of a 1 in a word. For example, some computers set a bit in an interrupt mask when an interrupt occurs, and the interrupt handler must determine which bit is set in order to properly vector the interrupt. Many chess programs represent the pieces of a given type as a 64-bit word, each bit of which indicates the presence or absence of the piece type on a particular square of the chessboard [5]. To determine which square a piece occupies as a row/column index, the index of a 1 bit must be determined.

To illustrate this indexing problem, suppose an 8-bit word contains `00100010`. This word contains 1's in positions 1 and 5, where we count from the low-order (rightmost) bit. The problem is to provide a fast algorithm that given such a word, outputs either 1 or 5 in binary. Some computers provide instructions FFO (Find First One) or FLO (Find Last One) to find the index of the high-order or low-order 1 in a computer word [4]. Many instruction sets do not contain such instructions, however, and thus it is up to a compiler writer or assembly-language programmer to synthesize the indexing operation.[1]

---

This research is supported by the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-97-1-0150.

[1]The opposite problem, that of taking an index of a bit and outputting a word with a 1 set in that position, can be done with a left shift operation.

The remainder of this paper is organized as follows. Section 2 describes the deBruijn algorithm. Section 3 compares the deBruijn algorithm with other strategies for indexing a `1` in a computer word. Section 4 proposes an extension of the deBruijn algorithm to indexing multiple `1` bits. Finally, Section 5 provides some concluding remarks.

## 2 The deBruijn algorithm

This section describes our deBruijn strategy for indexing a `1` in a computer word. We give three key ideas which, taken together, provide an efficient implementation for this problem.

### Idea #1: Isolate a `1`.

We first reduce the problem of finding a `1` in an arbitrary word to that of finding a `1` in a word that contains exactly one `1`. If the (nonzero) input word is `x`, we compute `y = x & (-x)` ("`&`" is C syntax for bitwise AND), which produces a word `y` having a `1` in the position of the low-order `1` of `x`. For example, if `x` = `01101000`, then the two's complement of `x` is `-x` = `10011000`, and thus `y` = `00001000`. To index all the `1`'s in a word, we can compute `x - y`, thereby removing the bit already indexed, and repeat the process.

This strategy for isolating a `1` appears to be folk knowledge. It is fast on contemporary machines, because the bitwise AND and two's complement are register operations that can be accomplished in a single machine cycle.

### Idea #2: Hashing.

We are now left with the problem of indexing the `1` in a word that contains exactly one `1`. For an $n$-bit word, there are exactly $n$ possible words containing exactly one `1`. Since $n$ is small, we can use another trick from algorithms: hashing. We use a perfect hash function $h$ to map each of the single-`1` words to a hash table. Then, given a single-`1` word `x`, we look up $h$(`x`) in the hash table where the index of the `1` bit is stored. For this strategy to work efficiently, however, we need

- the hash table to be small,
- the hash function to be easily computable, and
- the hash function to produce no collisions, i.e., no two single-`1` words `x` and `y` should produce hash values such that $h(\mathtt{x}) = h(\mathtt{y})$.

### Idea #3: de Bruijn sequences.

The final idea uses de Bruijn sequences [3] to satisfy all three criteria. The size of the hash table is exactly $n$, the minimum possible. The hash function is a typical multiplicative hash function [2, pp. 228–229] that involves a single unsigned integer multiplication of the key by a "de Bruijn" constant. Surprisingly, no two single-`1` words hash to the same location.

Let us briefly review de Bruijn sequences before seeing how they are used in this application. A length-$n$ ***de Bruijn sequence***, where $n$ is an exact power of 2, is a cyclic sequence of $n$ `0`'s and `1`'s such that every `0`-`1` sequence of length $\lg n$ occurs exactly once as a contiguous substring.[2] For example, a length-8 de Bruijn sequence is `00011101`. Each 3-bit number occurs exactly once as a contiguous substring: starting from the leftmost 3 bits and moving a 3-bit window right one bit at a time, we have `000`, `001`, `011`, `111`, `110`, `101`, `010` (wrapping around), and finally, `100` (also wrapping around).

The hash function is computed by

$$h(\texttt{x}) = (\texttt{x * deBruijn}) \texttt{ >> } (n - \lg n)$$

where "`>>`" denotes a logical right shift; multiplication is performed modulo $2^n$, meaning that the high-order bits of the product are thrown away; and `debruijn` is a computer word whose bit pattern contains a length-$n$ de Bruijn sequence beginning with $\lg n$ `0`'s. For example, for an 8-bit word, we might have `debruijn` = `00011101`, although any other de Bruijn sequence starting with 3 `0`'s, such as `00010111`, would work equally well.[3] For the hash function using `debruijn` = `00011101`, the table indexed by the hash function is the following:

| $h(\texttt{x})$ | Index |
|---|---|
| `000` | 0 |
| `001` | 1 |
| `010` | 6 |
| `011` | 2 |
| `100` | 7 |
| `101` | 5 |
| `110` | 4 |
| `111` | 3 |

Here, we have used the 3-bit binary number produced by the hash function on the left and the decimal number stored in the table on the right.

To illustrate how the hash function works, consider the input `00010000`. We multiply by `00011101`, producing the 16-bit product `0000000111010000`, of which only the low-order 8 bits `11010000` are retained. We shift this word right by $8 - 3 = 5$ bit positions, producing the value `110`. We index the table, producing the value 4 as the index of the `1`.

What is going on? The single-`1` words are all powers of 2. Multiplying by a power of 2 is equivalent to a shift. If the input to the hash function has a bit on in position $i$, then the multiplication causes `debruijn` to be shifted left by $i$ positions. Each of the $n$ possible shifts causes the top $\lg n$ bits of the resulting $n$-bit word to take on a distinct value. Shifting these $\lg n$ bits into the low-order bits of the word allows us to index the table mapping the "de Bruijn index" into the normal index. Indeed, for some applications, any index of bit position works as well as the normal numbering scheme, in which case the de Bruijn index $h(\texttt{x})$ itself can be used, saving the additional expense of indexing a table.

Figure 1 gives a 32-bit C implementation of the DEBRUIJN strategy.

[2] We use the notation $\lg n$ to mean $\log_2 n$.

[3] Guy L. Steele, Jr. of Sun Microsystems has noticed that the de Bruijn sequence need only begin with $(\lg n) - 1$ `0`'s.

```
#define debruijn32 0x077CB531UL
/* debruijn32 = 0000 0111 0111 1100 1011 0101 0011 0001 */

/* table to convert debruijn index to standard index */
int index32[32];

/* routine to initialize index32 */
void setup( void )
{
  int i;
  for(i=0; i<32; i++)
    index32[ (debruijn32 << i) >> 27 ] = i;
}

/* compute index of rightmost 1 */
int rightmost_index( unsigned long b )
{
  b &= -b;
  b *= debruijn32;
  b >>= 27;
  return index32[b];
}
```

Figure 1: 32-bit C implementation of the deBruijn strategy.

## 3 Empirical results

The expensive operations in the deBruijn scheme are the integer multiplication, which is surprisingly slow on many contemporary machines, and the table lookup, because operations on memory are slow compared with register operations. Nevertheless, our experiments have determined that the deBruijn scheme is often faster than or competitive with other indexing methods on many machines. This section provides an empirical comparison of the deBruijn scheme with other common methods for indexing a `1` in a computer word.

The fastest indexing method, which we call the Native method, computes the index in hardware. This method is not available on many computer, however. Thus, most of our comparisons are with software-based algorithms.

One popular software algorithm, which we call float, treats the input word as an unsigned integer and converts it into a floating-point number. Then, the index of the first `1` can be found by masking off the exponent. Since many machines provide integer-to-floating-point conversion as a single instruction, this method can be quite fast.

Another common strategy is to use a table lookup. Unfortunately, a naive table-driven strategy does not work well because there must be $2^n$ entries in the table to handle every possible configuration of bit settings in a word. For $n = 64$, the table would be prohibitively large. By using the divide-and-conquer paradigm, however, this method can be made to work effectively.

The lookup strategy we used in our comparisons works as follows. We first isolate a `1` in the computer word, and then test whether the upper half is zero. If it is zero, we recursively index the lower half of the word, and if it is nonzero, we recursively index the upper half of

| *Machine* | LOOKUP16 | LOOKUP4 | FLOAT | DEBRUIJN | NATIVE |
|---|---|---|---|---|---|
| UltraSPARC II | 7.2 | 13.5 | 24.1 | 10.5 | N/A |
| Alpha 21164 | 6.6 | 9.9 | 8.9 | 7.9 | N/A |
| Pentium II | 7.7 | 13.7 | 31.4 | 6.4 | 2.2 |
| R10000 | 4.0 | 8.3 | 6.0 | 2.5 | N/A |

Figure 2: Comparison of 32-bit implementations. Times are reported in processor cycles for each of the architectures.

the word. We terminate the recursion when the remaining portion that needs to be indexed is sufficiently small that table lookup can be performed easily. This strategy contains no long-latency instructions, but it can nevertheless be slow, because the branches used in the recursion cannot be easily predicted by the branch-prediction hardware in contemporary computers.

We compared the 32-bit implementation of DEBRUIJN from Figure 1 with good implementations of the LOOKUP and FLOAT strategies. We tested two implementations of the LOOKUP strategy: LOOKUP16, which uses a table with $2^{16}$ entries (256K bytes) and 16-bit keys; and LOOKUP4, which uses a table with 16 entries and 4-bit keys. (The DEBRUIJN strategy itself uses a table with 32 entries and 5-bit keys.) We implemented the algorithms in C, and where necessary in assembly language, on four machines: a 167-MHz Sun UltraSPARC II [10], a 300-Mhz Pentium II [7, 8], a 466-MHz Alpha 21164[4] [4], and a 194-MHz R10000. The results are tabulated in Figure 2.

To compare the strategies we measured the average number of clock cycles to find the index of a `1` in a circular shifted test number with seven `1` bits. We did not use random numbers, since the number of bits set to `1` is likely to be half the word size, thereby making the branches in the LOOKUP strategy more predictable than they would be for sparsely populated input words, which are common in most applications. Although the results tended to vary from one run to the next, the relative numbers reported in Figure 2 are reasonably consistent.

As can be seen from the figure, the DEBRUIJN strategy is a good method on all four platforms. The LOOKUP16 method outperforms it on the UltraSPARC and Alpha, but LOOKUP16 requires a 256K-byte table, which for many applications would be prohibitively large. By choosing shorter keys, such as 4-bit keys for the LOOKUP4 method, this table could be made smaller, but performance would suffer.

We also compared 64-bit implementations. Since 32-bit multipications tend to be fast compared to 64-bit multiplications on some machines, we implemented a modification of the 64-bit DEBRUIJN strategy that we call HALF-DEBRUIJN. After extracting a single `1` bit, we determine which half-word contains the `1`, and then use the 32-bit DEBRUIJN algorithm on that half-word. The expensive operations for this strategy are one 32-bit mutliplication, one table lookup, and one (unpredictable) branch. We compare these strategies with the FLOAT method described above and a LOOKUP strategy that uses a table with $2^{16}$ entries and 16-bit keys. The comparisons of the various strategies are shown in Figure 3.

As can be seen from the figure, either the DEBRUIJN or the HALF-DEBRUIJN algorithm

[4]The Alpha 21264, a later model, implements a native instruction for finding the rightmost bit.

| *Machine* | LOOKUP | FLOAT | DEBRUIJN | HALF-DEBRUIJN |
|---|---|---|---|---|
| UltraSPARC II | 15.6 | 16.9 | 25.5 | 13.7 |
| Alpha 21164 | 14.4 | 19.1 | 21.0 | 18.1 |
| Pentium II | 35.4 | 56.2 | 32.2 | 26.6 |
| R10000 | 8.0 | 8.1 | 6.6 | 8.0 |

Figure 3: Comparison of 64-bit implementations. Times are reported in processor cycles for each of the architectures.

is fastest on three of four architectures for the 64-bit indexing problem. The crucial issue seems to be the speed of 64-bit multiplication on these machines. In the full paper, we shall attempt to break down the performance figures.

## 4 Indexing two 1's

We have investigated whether our indexing strategy can be extended to indexing the 1's in words with at most two 1's. For example, a chess program might use this method to determine the positions of the two White Knights in a 64-bit representation of a chessboard. Although we have found no general theory for indexing multiple bits, we have made some empirical progress. This section describes our algorithm for indexing double-1 words and compares it to a lookup-based strategy.

For a computer word with two 1's, we can apply the DEBRUIJN strategy twice, which approximately doubles the cost. Can we index the two 1's with less work? There are $\binom{n}{2} + n + 1$ possible $n$-bit words with at most two 1's set. Is there a hash function that would allow us to index the 1's with only a single multiplication and table lookup on a table not much larger than $\binom{n}{2} + n + 1$?

We have written a Cilk [1, 6] parallel backtracking program to search for a hashing constant that could be used in a multiplicative hash function to index two or fewer bits. The key observation used by the algorithm is that the hash value of an input word whose $s$ low-order bits are all zero depends only on the low-order $n - s$ bits of the hashing constant. In each step the algorithm tries to extend the $n - s$ low-order bits of the hash constant to $n - s + 1$ bits. If any of the input words with $s - 1$ zero low-order bits collide, the search aborts and backtracks. The smallest table we found with this method for 64 bits contains $2^{15}$ entries, which is 16 times larger than the optimal table with slightly more than $2^{11}$ entries.

Our search yielded the following hash function, which maps all 64-bit numbers with at most two 1's into a table with 32706 entries:

$$h(\mathtt{x}) = (\mathtt{x} \ * \ \mathtt{E50F\,A91B\,E3A2\,5401}_{16}) \ \mathtt{>>} \ 49$$

We implemented the DEBRUIJN2BIT method on the four reference architectures and compared it to a strategy we call LOOKUP2BIT that uses the 64-bit LOOKUP method twice. (Recall that LOOKUP uses a table of $2^{16}$ entries and 16-bit keys.) Our results are tabulated in Figure 4. Compared to the LOOKUP2BIT strategy, which needs two memory lookups and has two unpredictable branches, DEBRUIJN2BIT performs only one multiplication and one

| *Machine* | LOOKUP2BIT | DEBRUIJN2BIT |
|---|---:|---:|
| UltraSPARC II | 52.2 | 38.0 |
| Alpha 21164 | 30.9 | 26.8 |
| Pentium II | 148.7 | 53.1 |
| R10000 | 79.3 | 21.7 |

Figure 4: Comparison of 64-bit implementations for indexing two bits. Times are reported in processor cycles for each of the architectures.

memory lookup. It is not only faster than LOOKUP2BIT on all the architectures tested, it also needs only half the memory to store its hash table.

# 5 Conclusion

The DEBRUIJN method seems to be a good trick for the bit-twiddling arsenal. We expect that technology will drive hardware implementations of integer multiplication to get faster (because of the heavy use of integer multiplication in cryptography and data compression [9]). Consequently, we can expect that the DEBRUIJN method will outperform other software implementations by increasing margins. On the other hand, it is also possible that direct hardware support for indexing `1`'s will increasingly be provided, obviating the need for software implementations.

We were unable to find a good theory for indexing sparse, multiple-`1` bit computer words. Perhaps with more theoretical insight a practical hashing strategy can be found.


# Acknowledgments

Thanks to Chris Joerg from the DEC Cambridge Research Laboratory who originally suggested the idea of hashing. Thanks to Guy L. Steele, Jr. of Sun Microsystems and Don Dailey of MIT for helpful comments.


# References


[1] Cilk-5.2 (Beta 1) Reference Manual. Available on the Internet from `http://theory.lcs.mit.edu/~cilk`.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[3] Nicolaas G. de Bruijn. A combinatorial problem. In *Indagationes Mathematicae*, volume VIII, pages 461–467. Koninklije Nederlandsche Akademie van Wetenschappen, 1946.

[4] Digital Equipment Corporation, Maynard, Massachusetts. *Alpha Architecture Handbook, Version 3*, 1996.

[5] Peter W. Frey and Larry Atkin. *Creating a Chess Player*, pages 226–324. Springer-Verlag, New York, 1998.

[6] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Canada, June 1998.

[7] Intel Corp. *Pentium Pro Family Developer's Manual, Vol. 3*, 1995. Available at `http://developer.intel.com/design/pro/MANUALS/refer1.htm`.

[8] Intel Corp. *Intel Architecture Software Developer's Manual, Vol. 1*, 1997. Available at `http://developer.intel.com/design/pro/MANUALS/243190.htm`.

[9] 1998 ACM SIGPLAN conference on programming language design and implementation (PLDI), 1998. Architecture Panel Session.

[10] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. PTR Prentice Hall, 1994.