Bitscans and De Bruijn Sequences

Linus Arver
linus@ucla.edu

December 26, 2014

Abstract

This informal paper is an attempt to describe how De Bruijn sequences can be used to find the index of a single bit (a.k.a., "bitscan") in a computer word. It also goes into great depth in describing a recursive, combinatorial algorithm written by Gerd Isenberg for the generation of these sequences.

Contents

1 The Problem

Consider the following 64-bit word with only 1 bit turned on:[1]

Listing 1: A 64-bit word with 1 bit set at binary index 53.

```
1 00000000 00100000 00000000 00000000 00000000 00000000 00000000 00000000
2           ^ bit index 53 (counting from 0..63)
```

[1] The leftmost bits are the most significant bits, and the rightmost bits are the least significant bits.

This word has just 1 bit set at binary index 53, or b_{53}. ***Our task is to determine the binary index of such a word as quickly as possible.*** This is called "ffs" for "find first (least significant) set bit" in POSIX, or just "bitscan."[2]

In C, a bitwise trick to isolate the LSB (least significant bit) of a word with more than 1 bit set is to use the expression **`(word & -word)`**.[3]

1.1 Native Hardware Instructions

On recent x86-64 processors, you can use assembly code that takes advantage of a native hardware bitscan forward instruction to find the binary index of the LSB.[4] These should be faster than any software implementation, including the topic of this paper (De Bruijn sequences).

2 Non-De Bruijn Solutions

2.1 Loop

This solution takes the word and right-shifts it until a bit is found. For this and all other C examples that follow, we use the keyword **`u64`** to represent an unsigned 64-bit type.[5]

Listing 2: A loop-based bitscan.

```
inline int bitscan_forward_loop(u64 word)
{
        int idx;
        for (idx = 0; idx < 64; idx++) {
                if ((word >> idx) & 1)
                    break;
        }
        return idx;
}
```

`bitscan_forward_loop` works by right-shifting the word repeatedly, increasing the size of the shift one bit at a time. If a binary AND operation with the value 1 is successful, the loop breaks. Listing 3 shows how this would work on our word from Listing 1. For legibility, we use a period (.) to represent a 0 bit.

Listing 3: The iterations of Listing 2.

```
Shift word right idx (0) times.
((....... ..1..... ....... ....... ....... ....... ....... .......) & 1) is FALSE
Shift word right idx (1) times.
((....... ...1.... ....... ....... ....... ....... ....... .......) & 1) is FALSE
                                        *
                                        *
Shift word right idx (52) times.
((....... ....... ....... ....... ....... ....... ....... ......1.) & 1) is FALSE
```

[2] On POSIX systems, see the manpages for ffs(3).

[3] This expression stems from what is known as *two's complement arithmetic*. Since we are dealing with C, it should be noted that the expression **`(word & -word)`** only works if the word is an unsigned type.

[4] For these processors, there is also the corresponding **bitscan reverse** (find index of MSB) instruction.

[5] On POSIX systems, the type is **`uint64_t`**.

```
Shift word right idx (53) times.
((....... ....... ....... ....... ....... ....... ....... .......1) & 1) is TRUE
Return idx (53).
```

The loop-based nature still makes it slow, since a bit on the 63rd (highest) index would result in 63 iterations. The speed of this solution is $O(n)$, where n is the binary index.

2.2 Switch Statement

This solution is perhaps the most straightforward of all: it merely checks the word against a 64-condition switch statement.

Listing 4: A switch statement bitscan.

```
inline int bitscan_forward_switch(u64 word)
{
        switch (word) {
        case 0x0000000000000001ULL: return 0;
        case 0x0000000000000002ULL: return 1;
                        *
                        *
        case 0x4000000000000000ULL: return 62;
        case 0x8000000000000000ULL: return 62;
        }
}
```

The number **0x0000000000000001ULL** represents an unsigned 64-bit word with the first bit set (index 0) in hexadecimal; number **0x0000000000000002ULL** represents the second bit set (index 1), and so on.[6]

3 De Bruijn Method

The idea behind this method is to encode 64 distinct values into a single 64-bit number, and then to use this number as the basis for a hashing algorithm.[7] A De Bruijn sequence meets this requirement, and is faster than either the loop or switch statement approaches shown above.

3.1 The Meaning Behind a De Bruijn sequence

A k-ary De Bruijn sequence $B(k, n)$ of order n is a cyclic sequence of an alphabet A with size k for which every possible subsequence of length n appears exactly once. Each $B(k, n)$ has length

$$k^n \tag{1}$$

and there are

$$\frac{k!^{k^{n-1}}}{k^n} \tag{2}$$

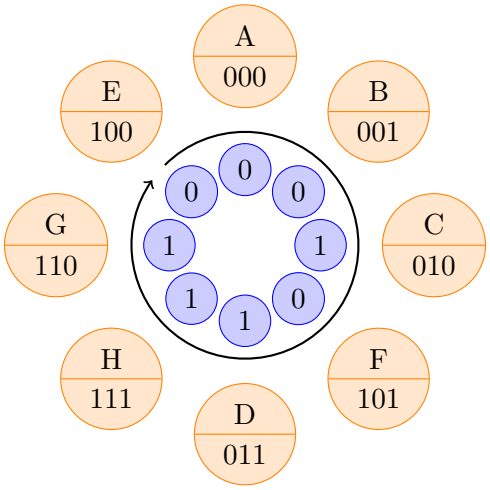
[6] The use of hexadecimal notation for our numbers allows us to write very clean-looking constants (e.g., **0x8000000000000000ULL**, which represents the word with just the 64th (highest) bit set, is 9223372036854775808 in decimal).

[7] Charles E. Leiserson, Harald Prokop, Keith H. Randall. *Using de Bruijn sequences to Index a 1 in a Computer Word*. MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1998. Available from http://supertech.csail.mit.edu/papers/debruijn.pdf

Figure 1: Some examples of De Bruijn sequences of the form $k = 2$.

$B(k,n)$	*Uniques*	*Length*	*Example (in binary and hexadecimal)*
$B(2,1)$	1	2	⟨**.1**⟩ or 0x1
$B(2,2)$	1	4	⟨**..11**⟩ or 0x3
$B(2,3)$	2	8	⟨**...1.111**⟩ or 0x17
$B(2,4)$	16	16	⟨**....1..11.1.1111**⟩ or 0x9af
$B(2,5)$	2,048	32	⟨**.....1...11..1.1..111.1.11.11111**⟩ or 0x4653adf
$B(2,6)$	67,108,864	64	⟨**......1....11..111...111111..1..1.1.11.1111.1.1..11.1...1.111.11**⟩ or 0x0219c7e4adea68bb

Figure 2: The $B(2,3)$ sequence ⟨**...1.111**⟩ with each subsequence named with a letter for legibility.



distinct De Bruijn sequences of $B(k,n)$.[8] In short, there are two essential characteristics of a De Bruijn sequence:

1. It is a cyclic sequence, and
2. every *subsequence* in it is unique.

For the scope of this paper, $k = 2$ because we are concerned with binary sequences, where A is just **1** and **0**. Some examples of genuine De Bruijn sequences are shown in Figure 1. The extra examples in there are there to dispel the intuitive notion that the longest run of consecutive 0's are required to be adjacent to the longest run of consecutive 1's.

Let us consider the two properties of De Bruijn sequences again, in the context of Figure 2. First, a true De Bruijn sequence is cyclic. We can confirm the cyclic property because it does not matter which subsequence (A, B, C, etc.) is the "first" letter. That is, ⟨**...1.111**⟩ (ABCFDHGE), ⟨**..1.111.**⟩ (BCFDHGEA), and ⟨**.1.111..**⟩ (CFDHGEAB) are just cyclic permutations of each other. Second, the subsequences of a De Bruijn sequence occur exactly once. We confirm this

[8] See http://en.wikipedia.org/wiki/De_Bruijn_sequence.

uniqueness property simply by observing that the each of the subsequences ⟨**...**⟩ to ⟨**111**⟩ in the diagram appears just once. Viewed another way, a De Bruijn sequence is a kind of *perfect compression* of information (the subsequences): the whole sequence is the shortest possible sequence that can encode all of the subsequences.

3.1.1 Verifying a De Bruijn sequence for $k = 2$

Here is a program that checks by brute force whether a given 8-bit sequence (of the range ⟨**.....**⟩ to ⟨**11111111**⟩) is indeed a $B(2, 3)$ sequence. The only drawback with this specific implementation is that it only looks for sequences with leading zeroes and does not check for cyclic variations; however, this sacrifice in accuracy is not a significant drawback, because no one cyclic variation is "better" than another.

Listing 5: Find 8-bit De Bruijn sequences by brute force.

```
#include <stdio.h>
#include <inttypes.h>
typedef uint8_t u8;

const int B111 = 7;
u8 pow2[8] = {
    0x1,  /* .......1 or 1 << 0 */
    0x2,  /* ......1. or 1 << 1*/
    0x4,  /* .....1.. or 1 << 2*/
    0x8,  /* ....1... or 1 << 3*/
    0x10, /* ...1.... or 1 << 4*/
    0x20, /* ..1..... or 1 << 5*/
    0x40, /* .1...... or 1 << 6*/
    0x80  /* 1....... or 1 << 7*/
};

void print_binary(u8 seq)
{
    int i;
    for (i = 7; i >= 0; i--)
        printf((seq & ((u8)1 << i)) ? "1" : ".");
}

int main()
{
    int i, j;
    u8 seq, subseq, seen;
    for (i = 0; i < 256; i++) {
        seq = i;
        subseq = 0;
        seen = 0;
        printf("%03d (0x%02x) [", seq, seq);
        print_binary(seq);
        printf("] ");
```

```
        for (j = 0; j < 8; j++) {
            subseq = ((seq << j) >> 5) & B111;
            if (seen & pow2[subseq]) {
                printf("subsequence collision\n");
                break;
            } else
                seen ^= pow2[subseq];
        }
        if (j == 8)
            printf("De Bruijn sequence verified!\n");
    }

    return 0;
}
```

Notice how the **seen** variable acts a sort of checklist, so to speak, where the checkboxes are the **1**-bits that are encoded with **subseq**, through the use of **pow2[]**. I.e., the **pow2[]** array is used to generate a single unique bit for a particular given subsequence **subseq**. This is an extremely compact way of detecting whether a particular subsequence has already been generated or not. This same technique will be used when we generate De Bruijn sequences in Section 3.2.

3.2 Generating a De Bruijn sequence for $k = 2$

Now that we understand what a De Bruijn sequence is, let's try to create one ourselves. The following is a complete program that generates a De Bruijn sequence $B(2, 3)$ (an 8-bit long binary sequence):

Listing 6: Generate De Bruijn sequences via a recursive, combinatorial search.

```
/*
 * Code adopted from Gerd Isenberg's C++ implementation for finding 64-bit long
 * (B(2,6)) sequences, available at
 * http://chessprogramming.wikispaces.com/De+Bruijn+Sequence+Generator
 * (retrieved September 12, 2011).
 */
#include <stdio.h> /* printf */
#include <stdlib.h> /* atoi */
#include <inttypes.h> /* uint8_t */
typedef uint8_t u8;

① /* constants */
const int B111 = 7; /* a mask to grab exactly 3 bits */
const int B110 = 6; /* 011, 111, then _110_ (see its use in gen_bruijn()) */
const int B100 = 4; /* the last, wrapped subsequence */
const int B011 = 3; /* the subsequence with a 0 followed by all 1's */

② /* globals */
u8 lock = 0; /* locks each bit used */
int cnt = 0; /* counter (nth generated De Bruijn sequence) */
int cnt_desired; /* the sequence to get */
```

```
u8 seq_desired = 0; /* the nth generated De Bruijn sequence itself */
/* pre-calculated array of powers of 2 */
u8 pow2[8] = {
    0x1,  /* .......1 or 1 << 0 */
    0x2,  /* ......1. or 1 << 1*/
    0x4,  /* .....1.. or 1 << 2*/
    0x8,  /* ....1... or 1 << 3*/
    0x10, /* ...1.... or 1 << 4*/
    0x20, /* ..1..... or 1 << 5*/
    0x40, /* .1...... or 1 << 6*/
    0x80  /* 1....... or 1 << 7*/
};

/* prototypes */
void gen_bruijn(u8 seq, int bitidx, int subseq);
void print_binary(u8 seq);

int main(int argc, char **argv)
{
    printf("generating 8-bit De Bruijn sequence...");
    if (argc < 2 || atoi(argv[1]) < 1 || atoi(argv[1]) > (2))
        printf("\nusage: mcg 1 .. %d\n", 2);
    else {
        cnt_desired = atoi(argv[1]);
        gen_bruijn(0, 4, 0);
        printf("OK\n");
        printf("binary form: ");
        print_binary(seq_desired);
        printf("\n");
    }

    return 0;
}

/* recursive search (generates every unique De Bruijn sequence!) */
void gen_bruijn(u8 seq, int bitidx, int subseq)
{
    if (seq_desired)
        return;
    if ((lock & pow2[subseq]) == 0 && subseq != B100) {
        if (bitidx < 0) {
            if (++cnt == cnt_desired)
                seq_desired = seq;
        } else {
            lock ^= pow2[subseq]; /* toggle ON; same as lock |= pow2[subseq]; */
            if (bitidx > 1 && subseq == B011)
                gen_bruijn(seq | pow2[bitidx], bitidx - 2, B110);
            else {
```

```
⑪ 70                if (bitidx > 0)
⑫ 71                    gen_bruijn(seq, bitidx - 1, (subseq << 1) & B111);
⑬ 72                gen_bruijn(seq | pow2[bitidx], bitidx - 1, ((subseq << 1) + 1) & B111);
   73            }
⑭ 74            lock ^= pow2[subseq]; /* toggle OFF; same as lock &= ~pow2[subseq]; */
   75        }
   76    }
   77 }
   78
   79 /* print binary representation of seq */
   80 void print_binary(u8 seq)
   81 {
   82     int i;
   83     for (i = 7; i >= 0; i--)
   84         printf((seq & ((u8)1 << i)) ? "1" : ".");
   85 }
```

The big picture in Listing 6 is that we add a 0 or 1 bit to a word, sort of like a binary tree, in a recursive fashion with **gen_bruijn()**. The result is that we try out every possible combination of 0 or 1 in a 8-bit word. The recursive function terminates branches that violate the unique subsequences rule with ⑥, so that we are left only with valid De Bruijn sequences.

Now let us break down the components of the **gen_bruijn()** function. This function is called with arguments **seq**, **bitidx**, and **subseq**. The **seq** variable merely holds the incrementally-generated sequence of 1's and 0's (such that the end result is the complete sequence). The **bitidx** and **subseq** variables are the bit index of **seq**, and the (hopefully) unique subsequence, respectively, that are currently being considered by **gen_bruijn()**.

The **pow2[]** array is the cornerstone of the algorithm. Its function is identical to the **pow2[]** array found in Listing 5. This array simply houses 8 numbers, that all have 1 bit turned on ③. The **pow2[]** array is used in conjunction with the **lock** variable at ⑥, ⑧, and ⑭ to toggle a unique bit in **lock**. The **lock** variable here functions in the same way as the **seen** variable from Listing 5. That is, **subseq** at these lines acts as an index to the particular single bit in **pow2[]**, which is then used to turn on a bit in **lock**. The expression **lock & pow2[subseq] == 0** checks if subseq was already encountered (i.e., the bit from **pow2[subseq]** was already seen before). The **XOR** statements at ⑧ and ⑭ act to ensure that the current **subseq** is toggled on for all child calls at ⑩, ⑫, and ⑬, and toggled off when we backtrack out of the current recursive depth.

The statements at ⑨ and ⑩ is just an optimization and is not strictly necessary. Let us examine the recursive calls at ⑫ and ⑬ first. These calls simply add a **0** or **1** bit to **seq** at **bitidx**. They also make sure to advance to the next lower **bitidx** with the argument **bitidix - 1**. The statements **(subseq « 1) & B111)** and **((subseq « 1) + 1) & B111)** are very straightforward: they replicate action of adding of **0** or **1** bit to **seq** to the **subseq** variable (masking with **B111** ensures that the new **subseq** is also only 3 bits wide). In this way, we end up with recursive calls that try to add all possible combinations of adding a **0** or **1** bit. When the child call starts up, it first tries to check that the currently-proposed **subseq**, from the parent's **(subseq « 1) & B111)** or **((subseq « 1) + 1) & B111)**, was not seen before with ⑥. And so the process continues, until bitidx is **-1**, at which point we can safely say that **seq** has passed the uniqueness test, that all of its subsequences are unique ⑦.

The **if** statement at ⑪ is required. This is because we initially start our search by manually adding the maximum number of consecutive **0** bits to the topmost bits of **seq**, by calling the first

gen_bruijn() with 0 as **seq** and **bitidx** as 4, not 7 (7th bit is the topmost bit). That is, we start out with **seq** as **000?????**. Because a De Bruijn sequence is cyclic, this implies that we cannot, by definition, add another **0** at the 0th bit (**000????0**), because the wrapping would mean that we get 4 consecutive 0 bits, which would result in 2 instances of the **000** subsequence (**000????0** and **000?????0**). Hence, we require that **0** bits may be added only if **bitidx** is greater than 0.

The optimization at ⑩ only comes into play if the subsequence being looked at is **0** followed by all **1**'s, or **011**. The call to **gen_bruijn()** at ⑩ skips over 1 **bitidx**, so that it calls **bitidx - 2** instead of **bitidx - 1**. That is, instead of essentially calling **gen_bruijn(seq, bitidx - 1, B010);** and **gen_bruijn(seq | pow2[bitidx], bitidx - 1, B011);** recursively as usual to try out adding both a **0** bit and **1** bit, we bypass this possible bifurcation by adding a 1 bit manually and skipping over two bits. I.e., if we're able to add the subsequence **B011** (recall that we're inside the **if** block from ⑥), then instead of bifurcating with **110** and **111**, we manually add **1** with **seq | pow2[bitidx]**, and then also add a **0** bit, resulting in the subsequence **B110** by skipping over two bits with **bitidx - 2** (recall that **seq** is zeroed-out). In short, it adds the bits **10** when we encounter the special subsequence **B011**. This subsequence is special because it is always followed by the sequence with the most consecutive 1 bits, or **111**. And since the longest consecutive run of **1** bits is followed by a **0** bit, we skip over the confirmation of the 111 subsequence by skipping over it (**bitidx - 2**), and instead test the next subsequence, **110**. The **bitidx > 1** test in ⑨ is yet another optimization; practically, this test ensures that we only skip over a bit if our current **bitidx** is at least 2. Empirically, this prevents ⑩ from being executed too greedily, which results in more dead-end nodes that do not yield a valid De Bruijn sequence.

Lastly, the test **subseq != B100** at ⑥ is another optimization. Recall that **seq** begins with its topmost bits set with the longest run of consecutive **0** bits allowed (**000?????**). This means that the last bit must be a 1, to prevent the double occurrence of **000**. Thus, all generated sequences have the form **000????1**. This implies that the subsequence 100 occurs via *wrapping* (**000????1**) at the very last node, when we add a 1 bit with either ⑩ or ⑬. Those child calls result in **subseq** being 6 ⑩, or some odd number ⑬. This means that the only time we encounter the **subseq 100** is when we are in the middle of generating our De Bruijn sequence **seq**. This would mean that we are attempting to generate a 100 subsequence *before* the tail end of our sequence, which would make all further recursive calls futile since the subsequence 100 must only occur as the very last, wrapped subsequence (as we have discussed above). To eliminate such futile searches, we aggressively cut all such branches in our recursive search.

The values in this algorithm can be trivially changed to create 16 $B(2,4)$, 32 $B(2,5)$, or 64-bit $B(2,6)$ sequences. This algorithm, simple as it is, is very fast. It takes less than a minute to generate all 67,108,864 unique sequences for 64-bit sequences. Also, due to its combinatorial, bifurcating approach, it generates *every unique* De Bruijn sequence for $B(2, \langle 3..6 \rangle)$.

3.3 Use As a Bitscan

Now that we understand how a De Bruijn sequence can be generated, we will proceed to discuss its use as a bitscan for 64-bit words. In this context, the De Bruijn sequence of concern is a $B(2,6)$ sequence, composed of 64 subsequences that are each 6 bits long (6 bits are all we need to represent the numeric values 0 through 63 (64 unique values), since $2^6 = 64$). And, the subsequences are from $\langle$**.....**$\rangle$ (decimal 0) to $\langle$**111111**$\rangle$ (decimal 63). The Listing 7 shows a complete working example in the use of a De Bruijn-based bitscan:

Listing 7: A complete De Bruijn-based bitscan.

```
/* ......1....11...1.1...111..1..1.11..11.1..1111.1.1.111.11.111111 */
const u64 BRUIJN_SEQ = 0x0218a392cd3d5dbfULL;

const int BRUIJN_HASH[64] = {
          0,  1,  2,  7,  3, 13,  8, 19,
          4, 25, 14, 28,  9, 34, 20, 40,
          5, 17, 26, 38, 15, 46, 29, 48,
         10, 31, 35, 54, 21, 50, 41, 57,
         63,  6, 12, 18, 24, 27, 33, 39,
         16, 37, 45, 47, 30, 53, 49, 56,
         62, 11, 23, 32, 36, 44, 52, 55,
         61, 22, 43, 51, 60, 42, 59, 58
};

/* b must have only 1 bit turned on */
inline int bitscan_forward_debruijn(u64 b)
{
        return BRUIJN_HASH[(BRUIJN_SEQ * k) >> 58];
}
```

The values for **BRUIJN_HASH[]** can be generated easily enough, as seen in Listing 8.

Listing 8: A loop to populate the values of **BRUIJN_HASH[]**.

```
const int B111111 = 63;
int i, bruijn_hash[64];
for (i = 0; i < 64; i++)
    bruijn_hash[((seq << i) >> 58) & B111111] = i;
```

We will now discuss how the De Bruijn Bitscan works.

3.3.1 The Lookup

Essentially, all that the De Bruijn Bitscan does is look up a table with precomputed values (**BRUIJN_HASH[]**). The operation

(BRUIJN_SEQ * b) » 58

always results in a value that has 6 meaningful bits. This is because we right shift the product 58 places, which erases the 58 rightmost (LSB) bits. These 6 bits represent one of 64 possible subsequences (that are 6-bits long) inside **BRUIJN_SEQ**. The last piece of the puzzle is that the product (**BRUIJN_SEQ * b**) is able to pick out a unique subsequence from **BRUIJN_SEQ** for every possible value of **b** (where b has only 1 bit turned on). We discuss how this could be below.

3.3.2 The Product, or "Turning the Wheel"

Usually, the product of two numbers, if seen from a bitwise perspective, results in a lot of "junk" looking bits. However, this is not so in the case of

BRUIJN_SEQ * b

because of the very special nature of **b** — *it only has a single bit turned on.* In binary, a single bit turned on in a word represents a value that is a power of 2. To illustrate, consider the following 8-bit word:

00000100

To get the value of this word, we compute each place.

$$(2^7 \times 0) + (2^6 \times 0) + (2^5 \times 0) + (2^4 \times 0) + (2^3 \times 0) + (2^2 \times 1) + (2^1 \times 0) + (2^0 \times 0) = 2^2 = 4$$

Thus, if only 1 bit is set, *its value is always a power of 2.* Going back to our example, the expression **BRUIJN_SEQ * b** always results in multiplying the **BRUIJN_SEQ** by a power of 2. But now consider the following identity:

$$(w \cdot 2^n) = (\texttt{w « n}) = (\text{adding } n \text{ significant zeroes to the right})$$

This identity shows us that multiplying a number w with 2^n is the same as moving all the bits in w left n times, which is the same as adding n zeroes to the right.

The reason behind this identity becomes clearer if we think about it in terms of base 10 arithmetic. The following should be of no surprise to you:

$$x \cdot 10^3 = x \text{ plus 3 zeroes}$$

E.g., if we multiply 18,491 by 10^3, you can instantly tell me that it's 18491 plus 3 zeroes, or 18,491,000. This idea of "adding zeroes" is what a binary left-shift operation does, essentially — the only difference being that the computer is working with base 2 (binary) arithmetic instead of base 10.

Combining everything we have learned so far, we can untangle the strange-looking expression **BRUIJN_SEQ * b** to pseudocode:

Left-shift BRUIJN_SEQ by the binary index of the bit set in word b

or

BRUIJN_SEQ « idx_{bit},

where idx_{bit} is our yet-unknown binary index. If we think back to the sequence in Figure 2, the computation **BRUIJN_SEQ « idx_{bit}** is the act of rotating the wheel counter-clockwise; the greater the value of **b**, the greater the turn. Figure 3 illustrates some sample turns.

Figure 3: Some examples of the computation **(BRUIJN_SEQ * b) » 58** in action. **BRUIJN_SEQ** is **0x0218a392cd3d5dbfULL** in all examples. The pink bits represent the added 0 bits after the left shift operation. The second line represents the result of the operation (**BRUIJN_SEQ * b) » 58**. The green bits represent the added 0 bits after the **» 58** operation. The blue bits represent the unique 6-bit subsequence used as an index for the **BRUIJN_HASH** array.

Value of **b**	**(BRUIJN_SEQ * b) » 58**
1 (2^0)	⟨**.....1....11...1.1...111..1..1.11..11.1..1111.1.1.111.11.111111**⟩
	⟨**.....**⟩
2 (2^1)	⟨**.....1....11...1.1...111..1..1.11..11.1..1111.1.1.111.11.111111.**⟩
	⟨**.....1**⟩
8192 (2^{13})	⟨**...1.1...111..1..1.11..11.1..1111.1.1.111.11.111111.....**⟩
	⟨**.....1.1**⟩
1048576 (2^{20})	⟨**..111..1..1.11..11.1..1111.1.1.111.11.111111.....**⟩
	⟨**.....111.**⟩
562949953421312 (2^{49})	⟨**1.111.11.111111.....**⟩
	⟨**.....1.111.**⟩

From Figure 3, we can see that the operation **(BRUIJN_SEQ * b)** results in turning the **BRUIJN_SEQ** left. The right shift of 58 bits merely acts to select the topmost 6 bits (i.e., the topmost subsequence). This way, we can guarantee that the resulting number only had at most 6 bits (the maximum value that 6 bits can generate is 63), which is necessary because the **BRUIJN_HASH** array only has 64 values (indexed from 0 to 63).

4 Conclusion

Hopefully, you've understood everything about the De Bruijn bitscan now. I personally had a difficult time with understanding how Gerd Isenberg's De Bruijn sequence generator worked. The De Bruijn bitscan is very powerful because it runs in constant time in a branchless way. At the same time, it is almost deceptively simple. The secret behind it all is the compact nature of a De Bruijn sequence and its ability to store unique subsequences.