Computation of the least significant set bit

Andrej Brodnik*
University of Waterloo
Department of Computer Science
Waterloo, Ontario, Canada
ABrodnik@UWaterloo.CA

Abstract

We investigate the problem of computing of the least significant set bit in a word. We describe a constant time algorithm for the problem assuming a cell probe model of computation.

1 Introduction and Definitions

The problem of computing an index of the least significant set bit in a word arises in different issues of data organization such as bit representation of ordered sets etc. In this paper we describe a novel algorithm to compute the index.

Our algorithm runs on a cell probe model of computation (c.f. [2, 4, 5]), which is a generalization of a random access machine model. We assume that the memory registers are of bounded size. The bits in the word (register) are enumerated from 0, which is the least significant bit, to $m-1$, which is the most significant bit – the word is m bits wide. Using the terminology of Fredman and Saks [2] we are dealing with CPROB(m) model. Furthermore, we assume that we can perform in one unit of time arithmetic operations of multiplication, addition and subtraction; logical operations of bitwise and, bitwise or and bitwise negation; and shifting to the right a specified number of bits. In the last operation we assume that shifting to the right is the same operation as division by the power of 2, where the least significant bits are lost while the most significant ones become 0.

In the next section we will give a brief background on the algorithm and explain the general ideas behind it. This is followed by a detail explanation and analysis of the complete algorithm. The paper is concluded with a short discussion.

*This research was supported by the Natural Sciences and Engineering Council of Canada under grant A-8237 and by the Information Technology Research Centre of Ontario.

2 Overview of the algorithm

The described algorithm is derived from the one due to Fredman and Willard [3] to compute $\lfloor \log_2(x) \rfloor$, which is essentially the index of the most significant set bit in a word. They developed the algorithm as a partial result used for "Fusion trees". As does theirs, our algorithm also performs in one operation (on a sequential machine) several "parallel" operations. At the end the results of these operations are quickly combined in the final result. To achieve this parallelism, the size of operands must be small enough. In general, the sum of sizes of operands of all operations has to be $O(m)$. In our case all operands are of equal size, what brings the best performance when we have sizes of about $\sqrt{m}$ and about the same number of parallel operations.

This parallelism is used first in a procedure `Lsb` (Algorithm 1) which takes an argument y of size about $\sqrt{m}$ bits (the size will be precisely defined in § 3) and returns the index of the least significant set bit in it. The procedure computes for each bit i ($0 \leq i \leq \sqrt{m}$) if it is smaller than the index of the least significant set bit in y in parallel, and at the end counts affirmative answers. For the final result the sum is subtracted from $\sqrt{m}$.

The second idea used in the algorithm is a *bit compression.* Fredman and Willard [3] showed how to compress d equally spaced bits into d least significant bits of a word in constant time using only two constants. They proved the following lemma:

Lemma 1 (Lemma 3 of [3]) *We say that a number x is d-sparse provided that the positions of all its 1 bits belong to a set of the form, $Y = \{a + di \mid 0 \leq i < d\}$, which consists of d consecutive terms of an arithmetic progression with common difference d. (Not all of these positions have to be occupied by 1's however.) If x is d-sparse, then there exist constants y_1 and y_2 such that for $z = (y_1 x) \wedge y_2$, the i^{th} bit of the significant part of z equals the bit in the position $a + di$ of x, $0 \leq i < d$.*

From the lemma we immediately get a function `Compress`.

The complete algorithm (see Algorithm 2) works in two phases. In the first phase it splits the argument (m bits wide) into $\sqrt{m}$ blocks of about $\sqrt{m}$ bits each. Using sequential parallelism it computes the representatives for individual blocks. The representatives are $\sqrt{m}$-sparse bits which are set if at least one bit in their corresponding blocks is set. After bit compression we apply to them function `Lsb`. The result is an index of the least significant block in the original parameter with set at least one bit. This concludes the first phase of the algorithm.

In the second phase we shift the proper block of the original parameter to right and apply to it function `Lsb`. Combining this result and the result at the end of the first phase, we get the final result.

For the sake of bravity we intentionally omitted in the explanation all details such as exact size of parameters, the problem of masking out the unwanted bits etc. They are explained in the next section.

3 A detail explanation of the algorithm

During the explanation we will define some constants and variables as they are needed and/or computed. Initially we assume that we are computing the least significant set bit in a word x and that $x \neq 0$.

The first step we will describe is a computation of representatives x_r. Let $s = \lceil \sqrt{m} + 1 \rceil$ be the size of blocks and their number is $t = \lceil \frac{m}{s} \rceil$. Note also that $t < s$. For each block we perform the following computation in parallel ($x_{r,i}$ is a representative of the i^{th} block x_i)

$$x_{r,i} = (x_i \wedge 2^s) \vee (((x_i \wedge (2^s - 1)) + (2^s - 1)) \wedge 2^s) \quad . \tag{1}$$

It consists of two terms which are ored together. The first term represents possibly set the most significant bit of the block and the second one any of less significant set bits. The result of computation $x_{r,i}$ is stored in the most significant bit of a block.

For parallel computation we define the constants

$$\begin{aligned} C_1 &= \left(\sum_{i=1}^{t-1} 2^{i \cdot s - 1} \right) + 2^{m-1} \\ C_2 &= (2^m - 1) - C_1 \end{aligned}$$

which replace values 2^s and $2^s - 1$ from eq. (1) respectively. The constant C_1 is also an s-sparse word where the set bits are the most significant bits of individual blocks. The constant C_2 is a complement of C_1.[1]

Because of the choice of constants and the computation in eq. (1), which does not interfere for two different blocks, we can compute all representatives in parallel

$$x_r = (x \wedge C_1) \vee (((x \wedge C_2) + C_2) \wedge C_1)$$

or simplified to

$$x_r = (x \vee ((x \wedge C_2) + C_2)) \wedge C_1 \quad . \tag{2}$$

For a more detail description of the function `Compress` is reader referred to the original Fredman and Willard paper [3]. All what we assume is that constants y_1 and y_2 from Lemma 1 are known.

The next procedure we develop is `Lsb` with an argument y ($y \neq 0, \lg y \leq t < s$). The result of the function is k, the index of the least significant set bit in y. The procedure computes for all i, $0 \leq i < s$

$$y_{r,i} = \big((y \wedge (2^{i+1} - 1)) + (2^s - 1)\big) \wedge 2^s \tag{3}$$

where $y_{r,i}$ is set iff there is set any of i less significant bits in y. The parallel computation is performed in non-overlapping blocks in a similar way as in eq. (2). Before description of the parallel part we need to distribute y to all blocks, what is done by a multiplication with

$$P = 2^0 + 2^s + \ldots = \sum_{i=0}^{t-1} 2^i \quad .$$

Next we define a constant

$$B = (2^1 - 1) \cdot 2^0 + (2^2 - 1) \cdot 2^s + \ldots = \sum_{i=0}^{t-1} (2^{i+1} - 1) \cdot 2^{is}$$

which takes a role of the expression $2^{i+1} - 1$ in eq. (3). Finally we compute

$$y_r = (((y \cdot P) \wedge B) + C_2) \wedge C_1 \tag{4}$$

for all indices i. Note that the last and masks out unwanted bits. To sum the remaining bits we first observe ($y_{r,j}$ is the j^{th} bit of y_r)

$$\begin{aligned} y_r \cdot P &= (\sum_{i=0}^{t-1} y_{r,i} 2^{si}) P = (\sum_{i=0}^{t-1} y_{r,i} 2^{si})(\sum_{j=0}^{t-1} 2^{sj}) \\ &= (2^0 y_{r,0}) + \ldots + (2^{(t-2)s} \sum_{i=0}^{t-2} y_{r,i}) + \\ & \quad (2^{(t-1)s} \sum_{i=0}^{t-1} y_{r,i}) + \\ & \quad (2^{ts} \sum_{i=1}^{t-1} y_{r,i}) + \ldots + (2^{2(t-1)s} a_{r,t-1}) \end{aligned}$$

that the t^{th} block exactly represents sum of set bit in y_r. Therefore if we define $S = 2^s - 1$ we can finally compute

$$k = t - \frac{y_r \cdot P}{2^{(t-1)s}} \wedge S \tag{5}$$

[1] Note that if the last block of bits is smaller than s bits, then we also set the most significant bit $m - 1$ in C_1.

The flow of computation in the algorithm (Algorithm 1) is slightly different then a computation presented in eq. (5). In the algorithm we invert the answers of individual operations and count the negative rather than the affirmative answers in y_r. Note as well that some quantities (y_r and $y_r * P$) are bigger than m bits and we have to use a double precision arithmetic.

```
PROCEDURE Lsb (y)
  y_r := (((y * P) AND B) + C_2) AND C_1;
  y_r := NOT(y_r) AND C_1;
  k := ShiftRight (y_r * P, (t-1) * s) AND S;
  RETURN k
END Lsb;
```

Algorithm 1: Computation of the least significant set bit in domain of $s = \lceil \sqrt{m} + 1 \rceil$ bits

Finally we put pieces together and present the complete algorithm (Algorithm 2) for words from a domain of m bits. The number of instructions used in the algorithm (if the call to function Lsb is unfolded) is presented in Table 1. It does not include the as-

Instruction	number
multiplication	6
addition	4
bitwise AND	13
bitwise OR	1
negation	2
shifting	3
total	29

Table 1: The number of instructions used in the algorithm

signments and assumes that the constant $(t-1)s$ in Algorithm 1 is precomputed. None of the instructions is a branching instruction and therefore they can be efficiently pipelined on modern computer architectures.

```
PROCEDURE LSB (x)
                    (* — PHASE 1 — *)
                                  (* representatives *)
  x_r := (x AND C_1) OR
         (((x AND C_2) + C_2) AND C_1);
  y := (y_1 * x_r) AND y_2;          (* compress them *)
  k_r := Lsb (y);          (* Lsb of representatives *)
  k_r := k_r * s;             (* what is actually bit *)
                    (* — PHASE 2 — *)
                           (* get proper block of x *)
  y := ShiftRight (x, k_r) AND S;
  k_b := Lsb (y);                (* Lsb of the block *)
  RETURN (k_r + k_b);      (* final result is sum *)
END LSB;
```

Algorithm 2: The constant time algorithm to compute the least significant set bit in the word x

4 Discussion

Algorithm 2 assumes that arithmetic operations have double precision. This can be avoided if we assume that the parameter x is from the domain of $\frac{m}{2}$ bits. Note as well, that we assume $x \neq 0$ and this has to be also checked at the beginning of the algorithm.

It is possible to have a restricted set of operations. One restriction, the lack of double precision arithmetic, was mentioned in a previous paragraph. It can be shown that for the purpose of our algorithm the shift operation, double precision arithmetic, swapping of word halves, and addressability of half-words are equivelent (c.f. [1]). Ben-Amram and Galil in the same work also discuss other approaches which can be used when there is available none of the mentioned operations.

Comparing our algorithm with a classical binary search algorithm, we first observe that our algorithm runs in time $O(1)$ and the binary search in time $O(\log m)$. Furthermore, the constant hidden in order notation of our algorithm is relatively small, approximately 30. This makes our algorithm even more preferable for longer words.

On the other hand the algorithm we presented does not use branching instructions, what makes it especially suitable for modern, pipeline architectures. In comparison, the binary search uses a branching instruction at each recursive step.

In conclusion, we developed a practical constant time algorithm to compute the index of the least significant set bit. The algorithm is specially appropriate for modern RISC architectures.

Acknowledgements

The author would like to thank Ian Munro for his help in shaping up this work and pointing out on errors in earlier versions of the paper.

References

[1] A.M. Ben-Amram and Z. Galil. When can we sort in $o(n \log n)$ time? In *IEEE Symposium on Foundations of Computer Science*, Palo Alto, California, 1993.

[2] M.L. Fredman and M.E. Saks. The cell probe complexity of dynamic data structures. In *ACM Symposium on Theory of Computing*, 1989.

[3] M.L. Fredman and D.E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. In *ACM Symposium on Theory of Computing*, pages 1–7, Baltimore, Maryland, 1990.

[4] A.C. Yao. Should tables be sorted? In *IEEE Symposium on Foundations of Computer Science*, pages 22–27, Ann Arbor, Michigan, 1978.

[5] A.C.-C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):614–628, July 1981.