A Non-Oblivious Reduction of Counting Ones to Multiplication

Holger Petersen
Reinsburgstr. 75
70197 Stuttgart
Germany
June 12, 2015

Abstract
An algorithm counting the number of ones in a binary word is presented running in time $O(\log \log b)$ where $b$ is the number of ones. The operations available include bit-wise logical operations and multiplication.

1 Introduction

The operation of counting the number of ones in a binary word consisting of $n$ bits has received considerable attention in quite different fields such as Cryptography [5] and Chess Programming [1], where the operation is used to evaluate the legal moves a player has in a given position. Counting ones is also known under the names sideways addition [4], bit count [2], or population count [1].

An early reference describing a non-trivial method for counting ones is the article by Wegner [6]: Instead of looping through all $n$ bits of a machine word, the right-most one of an operand $x > 0$ is repeatedly deleted by the operation $x$ and $(x - 1)$, where “and” denotes a bit-wise operation. In this way complexity $O(\nu_x)$ is achieved, where $\nu_x$ is the number of ones in the input $x$. This technique is also suggested in Exercise 2-9 of [3].

By forming growing blocks of bits, complexity $O(\log n)$ can be achieved with the help of constant time shift operations [1]. Under unit cost measure for multiplication or division, algorithms of asymptotical time complexity $O(\log \log n)$ are the Gillies-Miller method [7] [1] and Item 169 of [2].

In contrast to Wegner’s approach, the asymptotically more efficient solutions are oblivious in the sense that their complexity is independent of the input value. A sparse input (containing few ones) is not processed more efficiently than an input with many ones. If, e.g., the input is known to contain at most a constant number of ones, then Wegner’s method has time complexity $O(1)$.

In this note we show that the Gillies-Miller method can be modified to work in a non-oblivious way.
2 Result

Theorem 1 Counting ones can be done in $O(\log \log b)$ steps under unit cost measure for logical and arithmetical operations including multiplication, where $b = \nu x$ is the number of ones in the input $x$.

Proof: We describe an algorithm that uses several families of constants ("magic masks" in the sense of [4]) that potentially extend infinitely towards higher order bits. In concrete implementations these constants can be truncated to the word length of the processor architecture and only a finite number of values is required.

The first family $m$ of masks selects blocks of bits depending on parameter $k$:

$$m[k] = \cdots 11 \cdots 10 \cdots 00 \cdots 11 \cdots 1$$

The second family $h$ selects the most significant bits from each block:

$$h[k] = \cdots 10 \cdots 00 \cdots 10 \cdots 00 \cdots 00$$

The masks $h$ are also used in a modified form as multipliers for adding up blocks of bits of the current value of $x$.

Finally we make use of the table $e[k]$ with $e[k] = 2^k$.

Each iteration of the while-loop in code that follows starts with $x$ consisting of a sequence of blocks of length $\ell = 2^k$, where each block contains the number of ones of the input in the corresponding bit positions. Variable $p$ holds the product of $x$ and $2h[k+1] + 1$. The test concerning $p$ and $h[k]$ determines if $\ell - 1$ bits suffice to hold the count of all ones in the input. In fact, the test could be a little less strict with respect to the most significant bit, which may be a 1 in the block containing the count of all blocks.

```
function bitcount(x: integer): integer;
var k, p: integer;
begin
  k := 0;
  p := -x;
  while (p and h[k]) <> 0 do
    begin
      x := (x and m[k]) + ((x div e[k+1]) and m[k]);
      p := x * (2*h[k+1] + 1);
      k := k+1
    end;
  bitcount := (p div e[(n div 2) - e[k]]) and (e[e[k]]-1)
end;
```

We now argue that the above algorithm determines whether an overflow to the most significant bit of the blocks being added up occurs. Let $\ell = 2^k \geq 1$ be a block-length and consider the current $x$ as a sequence of blocks $x_{n/\ell} \cdots x_1$ of length $\ell$. We claim that if $\sum_{i=1}^{n/\ell} \geq 2^{\ell}$ then there is a $j \leq n/\ell$ such that $2^{\ell-1} \leq \sum_{i=1}^{j} \leq 2^\ell - 1$. If $x_1 \geq 2^{\ell-1}$ then we can take $j = 1$ since $\ell \leq 2^{\ell-1} \leq 2^\ell - 1$
for \( \ell \geq 1 \). Otherwise there is a maximum \( s \) such that \( \sum_{i=1}^s \leq 2^{\ell-1} - 1 \). Then \( \sum_{i=1}^{s+1} \leq 2^{\ell-1} - 1 + 2^{\ell-1} = 2^{\ell} - 1 \) and we can set \( j = s + 1 \). Now \( \sum_j \) has bit \( \ell \) set such that “\( p \) and \( h[k] \)” will not be 0.

When the loop terminates, the Gillies-Miller method will have produced the sum of all blocks in the “middle” block of \( p \). This block is extracted by the last assignment.

Since the loop terminates with the smallest \( k \) such that \( 2^{\ell-1} = 2^{2^k-1} > b \) we have \( k = O(\log \log b) \).

\[ \square \]

References

[1] Page “Population Count”. https://chessprogramming.wikispaces.com/Population+Count (download June 8, 2015).

[2] M. Beeler, R. W. Gosper, and R. Schroeppel. HAKMEM. Technical Report AIM-239, Massachusetts Institute of Technology, Cambridge, MA, USA, 1972.

[3] B. W. Kernighan and D. Ritchie. The C Programming Language. Prentice-Hall, 1978.

[4] D. Knuth. The Art of Computer Programming: Vol 4, Fascicles 0-4. Prentice Hall, 2009.

[5] H. Lipmaa and S. Moriai. Efficient algorithms for computing differential properties of addition. In FSE 2001, volume 2355 of LNCS, pages 336–350. Springer-Verlag, Berlin Heidelberg, 2002.

[6] P. Wegner. A technique for counting ones in a binary computer. Commun. ACM, 3(5):322, 1960.

[7] M. V. Wilkes, D. J. Wheeler, and S. Gill. The Preparation of Programs for an Electronic Digital Computer. Addison-Wesley, 2nd edition, 1957.