Efficient Hardware Algorithms for n Choose k Counters

Yasuaki Ito Koji Nakano Youhei Yamagishi*
Graduate School of Engineering
Hiroshima University

Abstract

An "n choose k" counter ($\mathrm{C}(n,k)$ *counter for short) is a counter which lists all* n*-bit numbers with* $(n-k)$ *0's and* k *1's. The "n choose k" counter has applications to solving combinatorial optimization problems and image processing. The main contribution of this work is to present an efficient hardware implementation of the* $\mathrm{C}(n,k)$ *counter. In some applications,* $\mathrm{C}(n,k)$ *counters are used only for small* k*. The second contribution is to show more efficient implementations that support* $\mathrm{C}(n,k)$ *counters only for small* k*. We evaluate the performance of our new implementation and known implementations in terms of the number of used slices and the clock frequency for the Xilinx VirtexII family FPGA XC2V3000-4. Although the theoretical analysis shows that our implementation is not the best, it runs in higher clock frequency using fewer number of slices than the other implementations.*

1 Introduction

An FPGA (Field Programmable Gate Array) is a programmable VLSI in which a hardware design can be embedded quickly. Typical FPGAs consist of an array of programmable logic elements, distributed memory blocks, and programmable interconnections between them. The logic block usually contains either a four-input logic function or a multiplexer and several flip-flops. The distributed memory block is usually a dual-port RAM on which a word of data for possibly distinct addresses can be read/written at the same time. Design tools are available to the users to embed their hardware logic designs into the FPGAs. Our goal is to use FPGAs to accelerate useful computations. In particular, it is very challenging to develop FPGA-based solutions that are faster and more efficient than traditional software solutions.

Let $\mathrm{C}(n,k)$ denote a set of all n-bit binary numbers that has $(n-k)$ 0's and k 1's. For example, $\mathrm{C}(6,3)$ is

$$\begin{aligned}\mathrm{C}(6,3) = \{&000111, 001011, 001101, 001110, 010011,\\ &010101, 010110, 011001, 011010, 011100, 100011,\\ &100101, 100110, 101001, 101010, 101100, 110001,\\ &110010, 110100, 111000\}. \end{aligned} \quad (1)$$

An *"n choose k" counter* ($\mathrm{C}(n,k)$ counter for short) is a counter that lists all numbers in $\mathrm{C}(n,k)$. It has shown in [7] that several computations including combinatorial optimization and image processing can be accelerated using $\mathrm{C}(n,k)$ counters. For example, suppose that we have a function $f : \{0,1\}^n \rightarrow \{0,1,\ldots,m\}$ for some positive integer m, and we need to find an n-bit binary number $\boldsymbol{r}$ such that $f(\boldsymbol{r})$ takes the minimum value over all possible 2^n n-bit binary numbers $\boldsymbol{x}$. In other words, our task is to compute

$$\boldsymbol{r} = \arg \min_{\boldsymbol{x} \in \{0,1\}^n} f(\boldsymbol{x}). \quad (2)$$

This task is a kind of combinatorial optimization, which has many practical applications. A fast and efficient solution for this task is to design an *instance-specific solution* using an FPGA as follows. We design a circuit that computes $f(\boldsymbol{x})$ for any given n-bit binary numbers $\boldsymbol{x}$. The output $\boldsymbol{x}$ of the n-bit counter is given to this circuit computing $f(\boldsymbol{x})$. A comparator is used to compare the current value of $f(\boldsymbol{x})$ and the minimum value obtained so far. If the current value $f(\boldsymbol{x})$ is smaller, then the current minimum $f(\boldsymbol{x})$ and $\boldsymbol{x}$ are updated. This hardware approach is promising if there exists an efficient (i.e. compact and of small depth) circuit computing f. An example of function f for which this approach works efficiently is the MAX-SAT problem. An input instance of the MAX-SAT problem is a set of m Boolean formulas $f_1, f_2, \ldots, f_m$ of n Boolean variables. MAX-SAT problem is a combinatorial optimization problem to find an assignment of Boolean variable values that maximizes the number of satisfied formulas (or minimizes the number of unsatisfied formulas). To solve the MAX-SAT problem using the above approach, we define function $f : \{0,1\}^n \rightarrow \{0,1,...,m\}$ such that

$$f(\boldsymbol{x}) = |\{f_i | f_i(\boldsymbol{x}) \text{ is not satisfied}\}|. \quad (3)$$

*Currently with Paltek Corporation, Japan.

1-4244-0054-6/06/$20.00

It should be clear that, $\boldsymbol{r}$ in formula (2) for function f in (3) is an optimal solution of the MAX-SAT problem. Also, Boolean formulas can be implemented in the FPGA by a combinational circuit in an obvious way. For example, an AND binary operator in a Boolean formula can be implemented using an AND gate with fan-in 2. Thus, the circuit computing $f(\boldsymbol{x})$ above can be implemented in the FPGA very efficiently and the above approach works for the MAX-SAT problem. This approach is an instance-specific solution [1, 6, 8], because the circuit embedded in the FPGA depends on the input instance (i.e. m Boolean formulas) of the problem.

In some application, $\mathrm{C}(n,k)$ counters are used only for small k. For example, suppose that an input instance of the MAX-SAT is given as a CNF(Conjunctive Normal Form) and most of the literals in the input formula are negative. For such instance of MAX-SAT, it is expected that the optimal solution has few 1 (or true) assignments. Hence, we can omit the evaluation of the value of $f(\boldsymbol{x})$ for input $\boldsymbol{x}$ that has many 1's. If this is the case, it is sufficient for $\mathrm{C}(n,k)$ counter to support only small k and it is possible to increase the clock frequency and reduce the number of used slices.

The first contribution of this work is to present an efficient hardware implementation of $\mathrm{C}(n,k)$ counters, that we call *bitonic shift implementation*. The second contribution is to show more efficient implementations that supports $\mathrm{C}(n,k)$ counters only for small k. We evaluate the performance of our new implementation in terms of the number of used slices and the clock frequency for the Xilinx VirtexII family FPGA XC2V3000-4. Although our implementation is not the best from the theoretical point of view, it runs in higher clock frequency using fewer number of slices than known implementations.

This paper is organized as follows. In Section 2, we show basic ideas for efficient implementation of $\mathrm{C}(n,k)$ counters. Section 3 shows known implementations of $\mathrm{C}(n,k)$ counters. Section 4 presents a new implementation of $\mathrm{C}(n,k)$ counters. In Section 5, we modify known implementations and our new implementation to list $\mathrm{C}(n,k)$ numbers only for small k. In Section 6, we evaluate the performance of these implementations for Xilinx VirtexII FPGA, XC2V3000-4. Section 7 offers concluding remarks.

2 Basic ideas for implementing $\mathrm{C}(n,k)$ counters

The main purpose of this section is to show basic ideas for implementing $\mathrm{C}(n,k)$ counters.

We can list all numbers in $\mathrm{C}(n,k)$ using the following five rules:

Rule 0: (initialization) Let the current number be $0^{n-k}1^k$.

Rule 1: If the current number is $(0+1)^*010^i$ for some $i \geq 0$, then the next number is $(0+1)^*100^i$.

Rule 2: If the current number is $(0+1)^*011^i$ for some $i \geq 1$, then the next number is $(0+1)^*101^i$.

Rule 3: If the current number is $(0+1)^*011^j0^i$ for some $i \geq 1$ and $j \geq 1$, then the next number is $(0+1)^*100^i1^j$.

Rule 4: (termination) If the current number is 1^k0^{n-k}, then terminate the listing.

Note that, as used in regular expressions, $(0+1)^*$ represents any sequence over $\{0,1\}$ of length zero or longer, and 1^k represents a sequence of consecutive k 1's.

The key rules are Rules 1, 2, and 3. Let us see how the next number is determined. Let $x_nx_{n-1}\cdots x_1$ be the current number. Further, let p ($1 \leq p \leq n-1$)be the smallest index of x such that $x_{p+1} = 0$ and $x_p = 1$. The next number can be obtained using the following two operations:

swap operation swap the values of x_{p+1} and x_p.

shift operation shift $x_{p-1}x_{p-2}\cdots x_1$ to the right until $x_1 = 1$.

In Rules 1 and 2, the swap operation is performed to find the next number. Both the swap and the shift operations are performed when Rule 3 is applied.

First, we show how we implement the swap operation which is performed in Rules 1, 2, and 3. For this purpose, we determine index p above. Let $y_i = \overline{x_{i+1}} \wedge x_i$ for every i ($1 \leq i \leq n-1$). Further, let $z_i = y_i \vee y_{i-1} \vee \cdots \vee y_1$, for every i ($2 \leq i \leq n$). In other words, $z_1 = x_1$ and $z_i = x_i \vee z_{i-1}$ for every i ($2 \leq i \leq n-1$). Thus, every z_i can be simply computed using the cascade of $n-2$ OR gates. Since z is the prefix OR of y, z can be obtained using the parallel prefix circuit [2, 3], which has $O(n)$ gates of depth $O(\log n)$. Let $u_1 = z_1$, and $u_i = z_i \wedge \overline{z_{i-1}}$ for each ($2 \leq i \leq n-1$). It should be clear that, $u_i = 1$ iff $p = i$. We refer the reader to Table 1 for examples of x, y, z, and u. The swap operation can be simply done by

$$x_i \quad \leftarrow \quad x_i \oplus (u_i \vee u_{i-1}) \qquad (1 \leq i \leq n), \qquad (4)$$

where $u_n = u_0 = 0$ and $\oplus$ denotes the XOR operator.

Next, we will show how the shift operation is implemented. Recall that the shift operation is performed for Rule 3. Let $s_i = \overline{z_i} \wedge x_i$ for each i ($1 \leq i \leq n-2$). Clearly, s is a sequence of bits to be shifted to the right. Let $t_{n-2}t_{n-3}\cdots t_1$ be a sequence of bits that can be obtained by repeating the shift of $s_{n-2}s_{n-3}\cdots s_1$ until the rightmost bit is 1. We refer the reader to Table 1 for examples of s and t. Once t is obtained, we can perform the shift operation by the following formula:

$$x_i \quad \leftarrow \quad (x_i \wedge z_i) \vee t_i \qquad (1 \leq i \leq n), \qquad (5)$$

Table 1. Examples of x, y, z, u, s, and t

i	10	9	8	7	6	5	4	3	2	1
x(current)	0	1	1	0	1	1	1	0	0	0
y	–	1	0	0	1	0	0	0	0	0
z	–	1	1	1	1	0	0	0	0	0
u	–	0	0	0	1	0	0	0	0	0
s	–	–	0	0	0	1	1	0	0	0
t	–	–	0	0	0	0	0	0	1	1
x(next)	0	1	1	1	0	0	0	0	1	1

where $s_n = s_{n-1} = t_n = t_{n-1} = 0$ for simplicity. We assume that every bit of t_i is 0 when all bits of s_i are 0. Then, when Rules 1 or 2 are applied, $s_i = t_i = 0$ for all i. Thus, from formulas (4) and (5) combined, regardless of the applied rules, the next number x can be obtained by a single formula as follows:

$$x_i \quad \leftarrow \quad ((x_i \oplus (u_{i-1} \vee u_i)) \wedge z_i) \vee t_i. \qquad (6)$$

Now we can conclude a basic algorithm for listing all $\mathrm{C}(n,k)$ numbers.

$x_n x_{n-1} \cdots x_1 \leftarrow 0^{n-k}1^k$
while $z_{n-1} = 1$ **do**
  foreach i **in** $[1,n]$ **do in parallel**
    $y_i \leftarrow x_i \wedge \overline{x_{i+1}}$
    $ty_i \leftarrow \overline{y_{i-1}} \wedge y_i$
    $x_i \leftarrow ((x_i \oplus (ty_{i-1} \vee ty_i)) \wedge z_i) \vee t_i \quad (1 \le i \le n)$.

Note that if $z_{n-1} = 0$ then $y_{n-1} = y_{n-2} = \cdots = y_1 = 0$. If this is the case, there exists no p such that $x_{p+1} = 0$ and $x_p = 1$. In other words, $x_n x_{n-1} \cdots x_1 = 1^k 0^{n-k}$ and Rule 4 (termination) should be applied.

As we have seen, y can be obtained by $n-1$ NOT gates and $n-1$ AND gates. The prefix OR circuit, which can be implemented using $O(n)$ gates of depth $O(\log n)$ [2], is used to compute z. Once z is obtained, u and s can be computed using $n-2$ NOT gates and $n-2$ AND gates, each. After that, if t is obtained, each x_i can be computed using two OR gates, one AND gate, and one XOR gate. Thus, a $\mathrm{C}(n,k)$ counter can be implemented using $O(n)$ gates of depth $O(\log n)$ excluding the circuit for computing t from s. However, it is not easy to obtain t. In what follows, we will show how we obtain t from s. For later reference, let $s = 0^{N-l-m}1^l0^m$, where $N = n-2$. Clearly, we need to compute $t = 0^{N-l}1^l$.

3 Known implementations of $\mathrm{C}(n,k)$ counters

This section shows two known implementations [7] *the simple shift* and *the binary shift implementations* that compute *t* from *s*. The simple shift implementation runs in high frequency for small n although it uses so many gates that it does not fit in the FPGA for large n. The binary shift implementation uses much smaller number of gates, but it runs in low frequency.

3.1 The simple shift implementation

The simple shift implementation uses all the shifted sequences of s. For each i and j ($1 \le i \le N; 0 \le j \le N-1$), let

$$\begin{aligned} s_i^{[j]} &= s_{i+j} \qquad \text{if } i+j \le N \\ &= 0 \qquad \text{if } i+j > N. \end{aligned} \qquad (7)$$

In other words, $s^{[j]}$ is a sequence obtained by shifting s by j bits to the right. Then, t can be obtained by

$$\begin{aligned} t_i \quad = \quad & (s_1^{[0]} \wedge s_i^{[0]}) \vee (s_1^{[1]} \wedge s_i^{[1]}) \vee \\ & \cdots \vee (s_1^{[N-1]} \wedge s_i^{[N-1]}). \end{aligned} \qquad (8)$$

Let us confirm that t is correctly computed by formulas (7) and (8). Recall that $s = 0^{N-l-m}1^l0^m$. Thus, $s_i^{[j]} = 1$ iff $m+1 \le i+j \le m+l$. Since $s_1^{[0]} = s_1^{[1]} = \cdots = s_1^{[m-1]} = 0$, $s_1^{[m]} = s_1^{[m+1]} = \cdots = s_1^{[m+l-1]} = 1$, and $s_1^{[m+l]} = s_1^{[m+l+1]} = \cdots = s_1^{[N-1]} = 0$, we have $t_i = s_i^{[m]} \vee s_i^{[m+1]} \vee \cdots \vee s_i^{[m+l-1]}$. Hence, $t_i = 1$ iff $[m+i, m+l-1+i] \cap [m+1, m+l]$ is not empty, that is $1 \le i \le l$. Therefore, $t_1 = t_2 = \cdots = t_l = 1$ and $t_{l+1} = t_{l+2} = \cdots = t_N = 0$, and thus *t* is computed correctly. Let us evaluate the number of gates used to compute *t*. Since $s_i^{[N-i+1]} = s_i^{[N-i+2]} = \cdots = s_i^{[N-1]} = 0$ ($i \ge 2$) always holds, t_i can be computed using $N-i+1$ AND gates and $N-i$ OR gates. Thus, t can be computed using at most $N + (N-1) + \cdots + 1 < \frac{N(N+1)}{2} < \frac{n^2}{2}$ AND gates and at most $(N-1) + (N-2) + \cdots + 1 < \frac{N^2}{2} < \frac{n^2}{2}$ OR gates. Since each t_i can be computed by a tree of $N-1$ OR gates with fan-in 2, the depth of the circuit is at most $\log N < \log n$.

3.2 The binary shift implementation

The binary shift implementation computes the binary representation of the number of 1's in s and generates the same number of 1's by exponential shifting. For simplicity, we assume that $N = 2^u - 1$ for some integer u. Let l be the

number of 1's in s and $l_u l_{u-1} \cdots l_1$ be the binary representation of l, that is $l = l_u \cdot 2^{u-1} + l_{u-1} \cdot 2^{u-2} + \cdots + l_1 \cdot 2^0$. The binary representation $l_u l_{u-1} \cdots l_1$ can be computed by the Muller-Preparata's adder tree circuit[5]. Let $s^{\langle j \rangle}$ $(0 \leq j \leq u)$ be a sequence of length $2^j - 1$ determined by the following procedure.

for $j \leftarrow 1$ **to** u **do**
  if $l_j = 0$ **then** $s^{\langle j \rangle} \leftarrow 0^{2^{j-1}} s^{\langle j-1 \rangle}$
  else $s^{\langle j \rangle} \leftarrow s^{\langle j-1 \rangle} 1^{2^{j-1}}$

If $l_j = 1$ then 2^{j-1} 1's are added to the sequence. Thus, it is not difficulty to see that $t = s^{\langle u \rangle}$ holds. Further, each $s^{\langle j \rangle}$ can be computed from $s^{\langle j-1 \rangle}$ using $2^j - 1$ multiplexers whose output is determined by l_j. Thus, t can be computed using at most $2^1 - 1 + 2^2 - 1 + \cdots + 2^u - 1 < 2N < 2n$ multiplexers. Also, it is easy to confirm that the depth of the circuit is $O(u) = O(\log n)$.

4 New implementations of $\mathrm{C}(n, k)$ counters

Our new idea is to use the bitonic merging [3] for implementing the $\mathrm{C}(n, k)$ counters. A sequence of bits is *bitonic* if

(1) it has consecutive 0's of length at least 0 followed by consecutive 1's of length at least 0, and

(2) it satisfies (1) by performing a cyclic shift.

For example, 00000111 is bitonic from (1). Thus, all of sequences 00001110, 00011100, 00111000, 01110000, 11100000, 11000001, 10000011 are bitonic from (2). Also, both 00000000 and 1111111 are bitonic from (1).

Let $A = a_1 a_2 \cdots a_N$ be a bitonic sequence of length N Further, let $B = b_1 b_2 \cdots b_{N/2}$ and $C = c_1 c_2 \ldots c_{N/2}$ be two sequences of length N each defined as follows:

$$b_i = a_i \wedge a_{i+N/2} \qquad 1 \leq i \leq N/2 \qquad (9)$$
$$c_i = a_i \vee a_{i+N/2} \qquad 1 \leq i \leq N/2 \qquad (10)$$

For example, if $A = 00111000$, then $B = 0000$ and $C = 10111$. For sequences B as C thus obtained, we have the following lemma:

Lemma 1 **(1)** *The total number of 1's in B and C is the same as those in A.*

(2) *If A has at least $N/2$ 1's, then C has no 0's. Similarly, if A has at most $N/2$ 1's, then B has no 1's.*

(3) *Both B and C are bitonic.*

Proof: If $a_i = a_{i+N/2} = 1$, then $b_i = c_i = 1$. If one of a_i and $a_{i+N/2}$ is 0 and the other one is 1, then $b_i = 0$ and $c_i = 1$. If $a_i = a_{i+N/2} = 0$, then $b_i = c_i = 0$. Hence (1) holds. If A has at least $N/2$ 1's then either a_i or $a_{i+N/2}$ is 1 and thus, C has no 0's. If A has at most $N/2$ 1's, then either a_i or $a_{i+N/2}$ is 0 and B has no 1's. Therefore, (2) holds. For any non-negative integer x, y, and z such that $x + y + z = N$, let $A = 0^x 1^y 0^z$. If $y \geq N/2$ then, C has no 0's and thus C is bitonic. Also, since $B = 0^x 1^{y-N/2} 0^z$, B is bitonic. If $y < N/2$ then, B has no 1's and thus B is bitonic. Similarly, since $C = 0^x 1^y 0^{N/2-x-y}$ if $x + y \leq N/2$ and $C = 1^{x+y-N/2} 0^{N/2-y} 1^{N/2-x}$ if $x + y > N/2$. Thus, B is bitonic. Consequently (3) holds. **Q.E.D.**

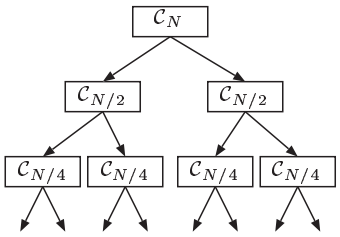

Figure 1. The tree structure of bitonic circuit

Let $\mathcal{C}_N$ denote the circuit that computes formulas (9) and (10). Since both B and C are bitonic, we can recursively use this circuit for each of B and C. The resulting circuit has a binary tree structure as illustrated in Figure 1. It should be clear that, From Lemma 1, this circuit computes $0^{x+z}1^y$ from $0^x 1^y 0^z$. We call this circuit `bitonic circuit`. See Figure 2 for illustrating the bitonic circuit for $N = 8$.

The bitonic circuit can be used to compute $t = 0^{N-l}1^l$ from $s = 0^{N-l-m}1^l 0^m$. Thus, we can obtain an implementation of $\mathrm{C}(n, k)$ counter using the bitonic circuit. We call this implementation `bitonic shift` implementation. The bitonic circuit has 1 $\mathcal{C}_N$, 2 $\mathcal{C}_{N/2}$s, 4 $\mathcal{C}_{N/4}$s, ..., $N/2$ $\mathcal{C}_2$s. Since $\mathcal{C}_N$ has $N/2$ AND gates and $N/2$ OR gates, the bitonic circuit has $1 \times N + 2 \times N/2 + \cdots + N/2 \times 2 = N \log N$ gates of depth $\log N$. Thus, the bitonic shift implementation has $N = \log N = O(n \log n)$ gates of depth $\log N = O(\log n)$.

Table 2 summarizes theoretical analysis of the performance of three implementations for $\mathrm{C}(n, k)$ counters. All of the implementations has depth $O(\log n)$. The binary shift uses only $O(n)$ gates, while the other implementations needs more than $O(n)$ gates. From the theoretical analysis, the binary shift is the best implementations. However, the constant factor hidden in big-O notation is very large. The binary shift implementation has tree of adders and three of selectors. Although both trees has $O(n)$ gates of depth $O(\log n)$, the constant factor is not small. On the other hands, constant factor in big-O notation in the simple shift and the bitonic shift is small. To compute t from s using AND and OR gates with fan-in 2, the simple shift uses less than N^2 gates of depth $\log N$. The bitonic shift uses

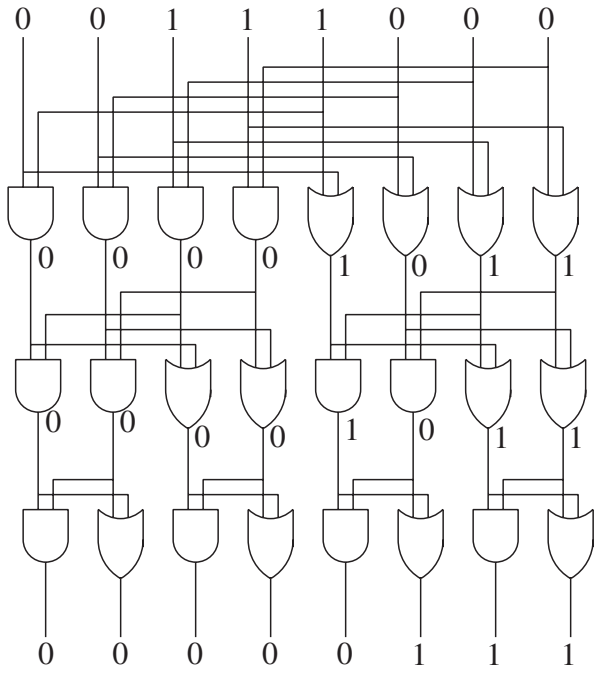

Figure 2. The bitonic circuit for $n = 8$

Table 2. Theoretical analysis of the performance of implementations of $\mathrm{C}(n,k)$ counters

implementations	gates	delay
simple shift	$O(n^2)$	$O(\log n)$
binary shift	$O(n)$	$O(\log n)$
bitonic shift	$O(n \log n)$	$O(\log n)$

$N \log N$ gates of depth $\log N$. Thus, it is possible that the simple shift and the bitonic shift implementations outperforms the binary shift implementation for practically small n.

5 $\mathrm{C}(n,k)$ counters for small k

The main purpose of this section is to design several implementation of $\mathrm{C}(n,k)$ counters for small k. More specifically, for some small fixed k, we design counters that can list $\mathrm{C}(n,i)$ numbers ($1 \leq i \leq k$). In some applications, this restriction is possible. Since the counter does not have to list $\mathrm{C}(n,k)$ numbers for large k, we may reduce the hardware resources and increase the clock frequency.

Let us modify the simple shift, the binary shift, and the bitonic shift implementations to support $\mathrm{C}(n,k)$ counters for small k. For this purpose, we will modify circuits to compute $t = 0^{N-l}1^l$ from $s = 0^{N-l-m}1^l0^m$, where $N = n-2$. Note that to implement a $\mathrm{C}(n,k)$ counter, the subcircuits need to compute t from s for $l = 1, 2, \ldots k-1$.

We first modify the simple shift implementation of a $\mathrm{C}(n,k)$ counter for small k. Since s and t has at most $k-1$ 1's, $t_n = t_{n-1} = \cdots = t_k = 0$ and $t_1, t_2, \ldots, t_{k-1}$ can be either 0 or 1. Thus, it is sufficient to compute $t_1, t_2, \ldots, t_{k-1}$ and we can omit the circuit to compute $t_k, t_{k+1}, \ldots, t_n$. Recall that each t_i ($1 \leq i \leq n$) needs $N-i+1$ AND gates and $N-i$ OR gates. Hence, $N + (N-1) + \cdots + N - k + 2 = O(Nk) = O(nk)$ AND gates and $(N-1) + (N-2) + \cdots + N - k + 1 = O(Nk) = O(nk)$ OR gates. It should be clear that the depth of the circuit is still $\log N < \log n$.

We next modify the binary shift implementation. In the binary shift implementation, the binary representation $l_u l_{u-1} \cdots l_1$ of l is computed by the Muller-Preparata's adder tree circuit. Since $l \leq k-1$, we need only $\log(k-1)$-bit binary representation. Thus, adders of the Muller-Preparata's adder tree circuit can be limited to $\log(k-1)$ bits. However, the Muller-Preparata's adder tree circuit still has $O(n)$ gates with depth $O(\log n)$. Using the binary representation, $s = s^{\langle \log k - 1 \rangle}$ is computed by the following procedure:

for $j \leftarrow 1$ **to** $\log k - 1$ **do**
  if $l_j = 0$ **then** $s^{\langle j \rangle} \leftarrow 0^{2^{j-1}} s^{\langle j-1 \rangle}$
  else $s^{\langle j \rangle} \leftarrow s^{\langle j-1 \rangle} 1^{2^{j-1}}$

Thus, from the $\log(k-1)$-bit binary representation of l, s can be computed using at most $2^1 - 1 + 2^2 - 1 + \cdots + 2^{\log k - 1} - 1 < k$ multiplexers of depth $O(\log k)$. So, the Muller-Preparata's adder circuit is dominant in the binary shift implementation. Hence, it still has $O(n)$ gates with depth $O(\log n)$.

Finally, we modify bitonic shift implementation. Suppose that $0^x 1^y 0^z$ ($x + y + z = N$) is given to the bitonic circuit. If b is small, the output of the most of $\mathcal{C}_i$ ($i = 1, 2, 4, \ldots, N/2$) has no 1's. We can remove such $\mathcal{C}_i$ from the bitonic circuit. We assume that $y \leq k$ for some small k always holds. Then, the left $\mathcal{C}_{N/2}$ in Figure 1 can be removed if $k \leq N/2$. Similarly, if $k \leq N/4$, only the rightmost $\mathcal{C}_{N/4}$ is necessary. In general, if $k \leq N/2^i$, then the rightmost $\mathcal{C}_{N/2^i}$ is necessary and the other $\mathcal{C}_{N/2^i}$'s in the bitonic circuit can be removed.

Let j be the largest integer satisfying $k > 2^j$. The resulting circuit has 1 $\mathcal{C}_N$, 1 $\mathcal{C}_{N/2}$, ..., 1 $\mathcal{C}_{N/2^{j-1}}$, 2 $\mathcal{C}_{N/2^j}$s, 4 $\mathcal{C}_{N/2^{j+1}}$s, ..., 2^j $\mathcal{C}_2$s. Hence the total number of gates is $N + N/2 + N/4 + \cdots + n/2^{j-1} + 2 \cdot N/2^j + 4 \cdot N/2^{j+1} + \cdots N/2^j \cdot 2 < 2N + j \cdot N/2^{j+1} = O(N + k \log k)$.

Table 3 summarizes the theoretical analysis of the performance. The bitonic shift implementation uses $O(n)$ gates if $k \log k = O(n)$, that is, $k = O(n/\log n)$. Thus, the

Table 3. Theoretical analysis of the performance of implementations of $\mathrm{C}(n,k)$ counters for small k

implementations	gates	delay
simple shift	$O(nk)$	$O(\log n)$
binary shift	$O(n)$	$O(\log n)$
bitonic shift	$O(n + k \log k)$	$O(\log n)$

binary shift and the bitonic shift implementations use the same number of gates if $k = O(n \log n)$. On the other hand, as we are going to show later, the actual implementations to FPGAs show that the bitonic shift implementation uses fewer hardware resources and runs in higher frequency.

6 Performance Evaluation

This section is devoted to show the performance evaluation for for the Xilinx VirtexII family FPGA XC2V3000-4, which has 14336 slices. A slice is a unit block of the VirtexII FPGA, which has two four-input function generators, carry logic, multiplexers, and two storage elements [4]. We have used Xilinx ISE logic design tool (Ver 7.1i) to analyze the timing and the number of slices used. We have wrote the HDL source codes for $\mathrm{C}(n,k)$ counter implementations in RTL (Register Transfer Level) of Verilog HDL. We have used default parameter values, for example "Optimization goal = Speed" and "Optimization effort = Normal", for logic synthesis using Xilinx ISE logic design tool. Also, we gave no user constraints to synthesize our Verilog HDL source codes.

Figure 3 shows the clock frequency and the number of used slices for $n = 8, 16, 32, 64, 128, 256, 512$, and 1024 estimated based on the net list obtained by XST logic synthesis tool, which is a part of Xilinx ISE logic design tool. For $n \geq 512$, the simple shift implementation does not fit in the XC2V3000-4. The simple shift and the bitonic shift implementation runs in almost the same frequency for all n ($n \leq 256$). The binary shift implementation runs in lower frequency than the others. Recall that the binary shift implementation has two circuits: (1) the Muller-Preparata's adder tree circuit to compute the number of 1's and (2) the multiplexer tree to generate consecutive 1's. Although both circuits has $O(\log n)$ depth, the adder-tree is complicated and has large depth. Hence, as long as the Muller-Preparata's adder tree is used, the clock frequency of the binary shift implementation cannot be better than the other two implementation.

As shown in Figure 3, the bitonic and the binary shift implementations use almost the same number of slices in the FPGA. On the other hand, the simple shift uses much more slices and does not fit in the FPGA for $n \geq 512$. Consequently, from the practical point of view, the bitonic shift implementation runs higher frequency and uses fewer slices, and thus it is the best among the tree.

Figure 4 shows the performance of the implementations that lists $\mathrm{C}(n,k)$ numbers for small k. Since each implementation has a sub-circuit that computes t from s with at most 8 1's, it can support $\mathrm{C}(n,k)$ counters for $k \leq 8+1 = 9$. Also, Figure 5 shows the performance of the implementations that lists $\mathrm{C}(n,k)$ numbers for $k \leq 17$. In either case, for large n such that $n \geq 128$, the bitonic shift implementation works in the highest frequencies. Further, in both cases of $k \leq 9$ and $k \leq 17$, the bitonic shift implementation uses fewest slices among the three. Therefore, the bitonic shift implementation is the best implementation.

7 Conclusions

The main contribution of this work is to present an efficient implementation of $\mathrm{C}(n,k)$ counters. We also presented implementations that supports $\mathrm{C}(n,k)$ counters for small k. The performance of the implementations are evaluated in terms of the number of used slices and the clock frequency for the Xilinx VirtexII FPGA XC2V3000-4. The results show that our new implementation called bitonic implementation is the best implementation from the practical point of view.

References

[1] J. L. Bordim, Y. Ito, and K. Nakano. Accelerating the CKY parsing using FPGAs. *IEICE Transactions on Information and Systems*, E86-D(5):803–810, May 2003.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[4] Xilinx Inc. *Virtex-II Platform FPGAs: Complete Data Sheet*, 2003.

[5] D. E. Muller and F. P. Preparata. Bounds to complexityies of network for sorting and for switching. *J. ACM*, 22:195–201, 1975.

[6] K. Nakano and E. Takamichi. An image retrieval system using FPGAs. *IEICE Transactions on Information and Systems*, E86-D(5):811–818, May 2003.

[7] K. Nakano and Y. Yamagishi. Hardware n choose k counters with applications to the partial exhaustive search. *IEICE Trans. on Information & Systems*, 2005.

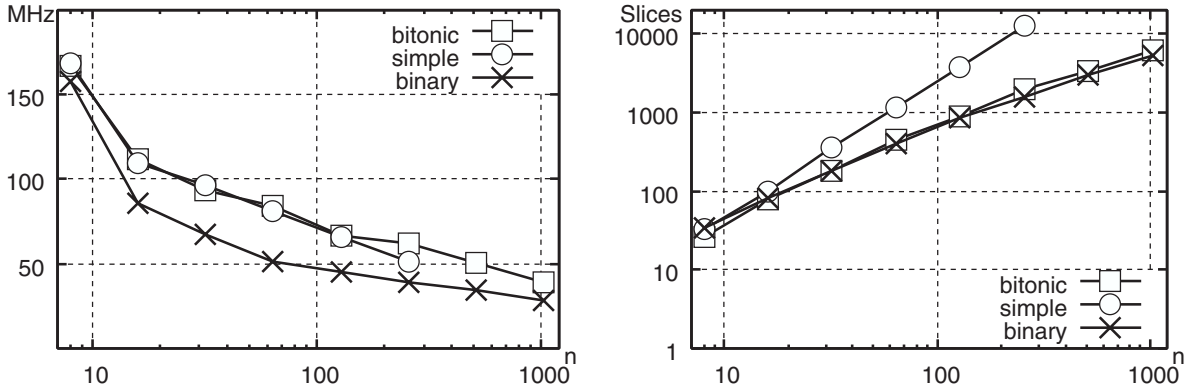

Figure 3. The performance of implementations of $\mathrm{C}(n,k)$ counters

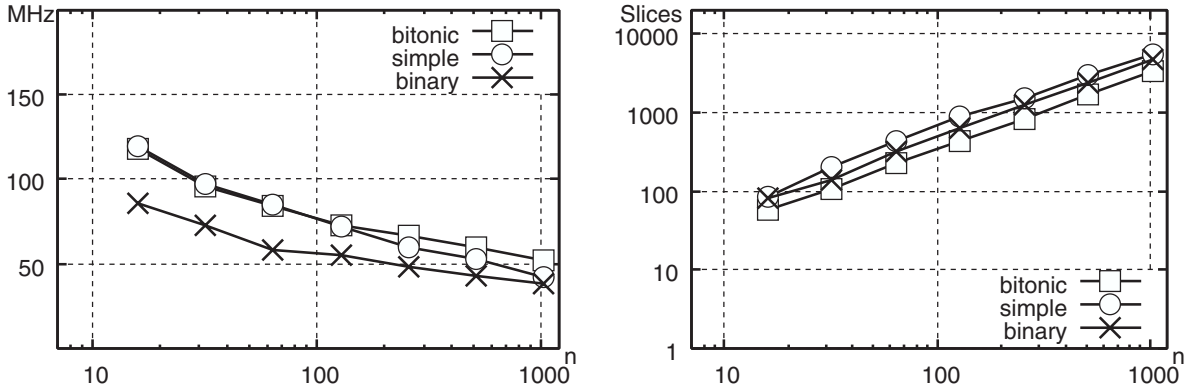

Figure 4. The performance of implementations of $\mathrm{C}(n,k)$ counters for $k \leq 9$

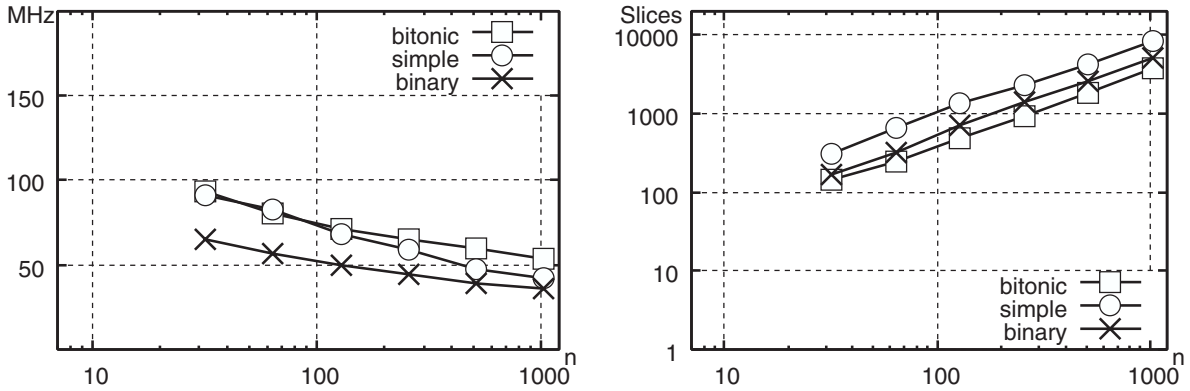

Figure 5. The performance of implementations of $\mathrm{C}(n,k)$ counters for $k \leq 17$

[8] P. Zhong, P. Ashar, S. Malik, and M. Martonosi. Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with boolean satisfiability. In *Design Automation Conference*, pages 194–199, 1998.