Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

# BINARY COUNTER WITH COUNTING PERIOD OF ONE HALF ADDER INDEPENDENT OF COUNTER SIZE

Milos Ercegovac
Tomas Lang

October 1988
CSD-880081

# Binary Counter with Counting Period of One Half Adder Independent of Counter Size

Milos Ercegovac and Tomas Lang

Computer Science Depatment
University of California, Los Angeles

September 1988

**Abstract.**

The properties that make a fast counter suitable for many applications are: i) a high counting rate, independent of the counter size, ii) a binary output that can be read on-the-fly, iii) a sampling rate equal to the counting rate, and iv) a regualr implementation suitable for VLSI. We describe the implementation of a counter having these properties. The minimum period of the counter is equal to the delay of one half adder plus the delay of loading a flip-flop. Both a modulo-$2^n$ and the more general modulo-$p$ cases are considered.

## 1. Introduction

Fast counters are of importance in many applications in communication and measuring devices. The basic properties that such a counter should have are summarized by David Chu in a recent paper [1]. They are: i) a high counting rate, preferably independent of the counter size; ii) a binary output that can be read on-the-fly; iii) a sampling rate equal to the counting rate; and v) a regular implementation suitable for VLSI.

As also stated in [1], an especially simple implementation is a ripple counter. However, the asynchronous operation of such a counter makes reading the value difficult (because the whole counter has to be stable) and reduces the maximum counting rate, especially for large counters. The alternative conventional synchronous counters can be read without problems but the complexity of the logic when the size of the counter increases limits the counting rate. Asynchronous and synchronous counters are discussed in several texbooks, see for example [2].

We present an implementation of a modulo-$p$ counter that has the desired properties. Its minimum period is independent of the counter size and equal to the delay of one half adder plus the delay of loading a flip-flop. The counter is synchronous and produces the count in a radix-2 representation so that it can be read on-the-fly. Moreover, the implementation is very regular and with localized connections, so that it can be implemented efficiently in VLSI.

## 2. Modulo-$2^N$ Counter

We first describe the implementation of a modulo-$2^N$ counter; in the next section we consider the more general modulo-$p$ case.

Assume, for simplicity in the description, that $N = 2^n$, that is, the counter is a $2^n$-bit binary counter. An example is a 64-bit counter used as a modulo-$2^{64}$ counter (that is, counting from 0 to $2^{64}-1$). The $2^n$-bit counter is implemented recursively by two modules $M_1$ and $M_2$. Module $M_1$ is a $n$-bit counter and module $M_2$ is a $(2^n - n)$-bit counter, as shown in Figure 1(a). Module $M_2$ is implemented by an $(2^n - n)$-bit incrementer, a ring counter modulo-$2^n$, and a $(2^n - n)$-bit register. The register, which contains the corresponding portion of the count, is loaded when the ring counter reaches the value $n-1$.

If the output value and the ring counter are both initialized to 0, the operation is that of a $2^n$-bit counter, since the register of module $M_2$ is loaded with the incremented value when module $M_1$ changes from $2^n-1$ to 0. This is illustrated in Figure 1 (b) for a 5-bit counter.

Module $M_1$ is implemented by dividing it into two modules in the same manner as the original counter, that is into a ($\lceil \log_2 n \rceil$)-bit counter and a ($n - \lceil \log_2 n \rceil$)-bit counter. This process is continued until the smaller module is a 1-bit counter. The resulting implementation consists of $m$ modules $M_i$, $i = 1,...,m$.

As an example consider the implementation of a 64-bit counter. The recursive division results in a 58-bit counter, a 3-bit counter, a 2-bit counter and a 1-bit counter, as shown in Figure 2.

To analyze the maximum counting rate, assume that each incrementer is implemented as a linear array of half adders (Figure 3). In such a case, the delay of a $k$-bit incrementer is of $k$ half adders. Moreover, observe the following two facts, as illustrated in Figure 3b:

i) The register of the $i^{th}$ module is updated every $k_i$ cycles and the complete subcounter to its right corresponds to a modulo-$k_i$ counter.

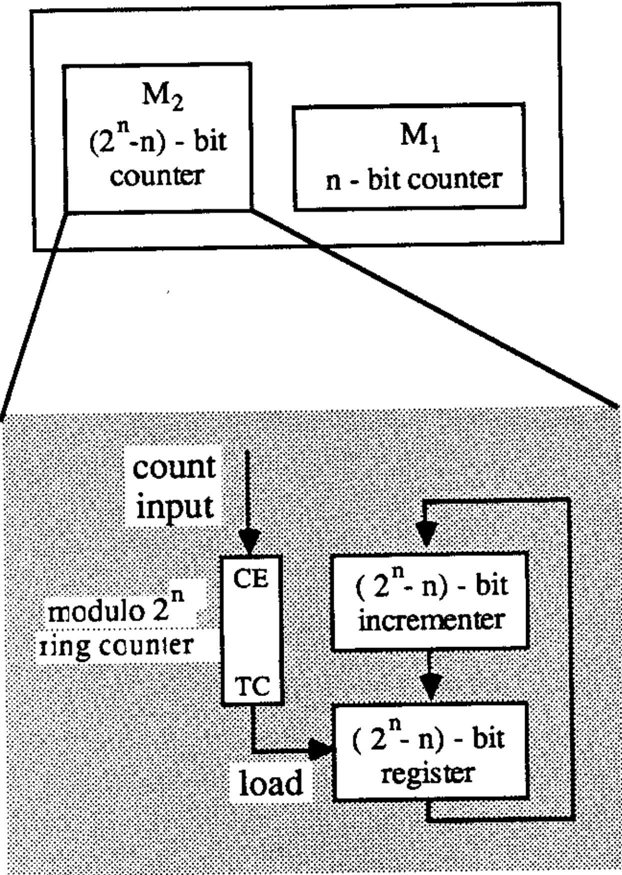
ii) The incrementer of the $i^{th}$ module is at most $k_i$ bits wide.

Because of fact i) the propagation of the incrementation in module $i$ should have a delay of at most $k_i$ cycles. Since the width is at most $k_i$ bits, this results in a minimum cycle of one half adder (plus the loading of one flip-flop).

Note that since the delay in the rightmost 1-bit counter is also of one half adder, it is not necessary to use in the longer incrementers a faster implementation, such as a carry lookahead, for example.

### 3. Modulo-$p$ counter

The counter described in Section 2 can be used for a modulo-$p$ counter, with $p \leq 2^N$. To achieve this type of operation, the counter register has to be reset when the count corresponds to $p-1$. To detect the value $p-1$ several comparators are used, one per module. We now consider the effect of the delay of these comparators in the counting rate and develop a mode of operation

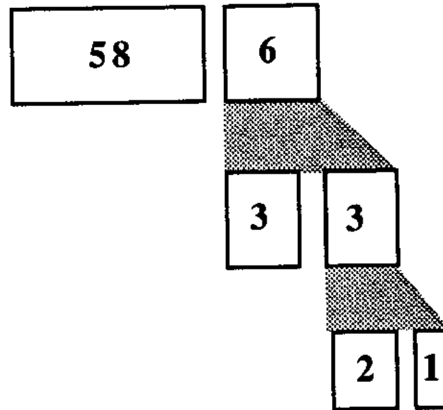



(a)

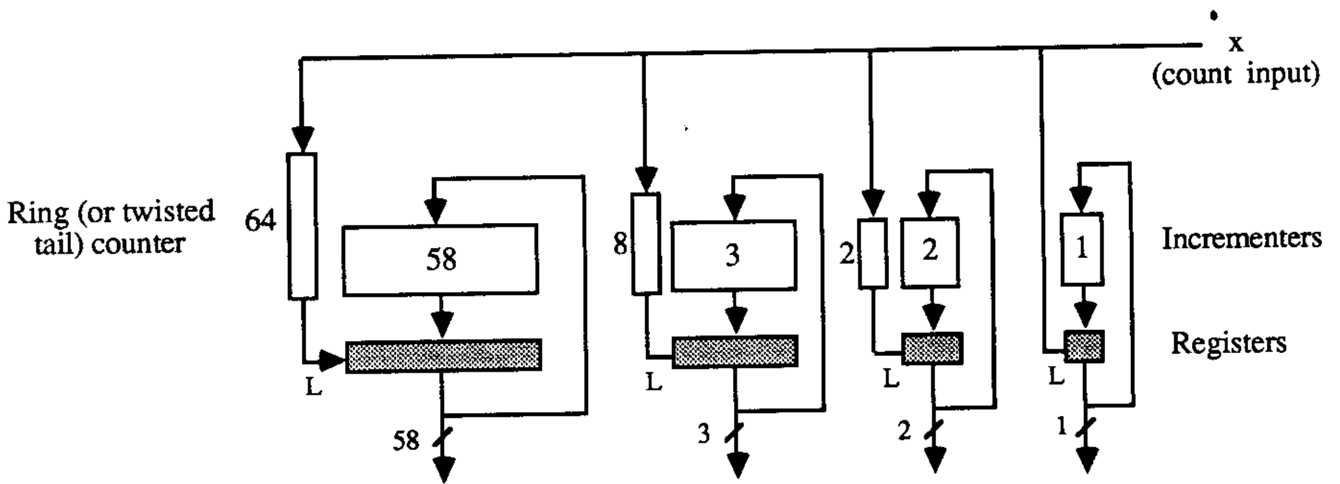
| Ring Counter | $M_2$ | $M_1$ |
|---|---|---|
| 0 | 00 | 000 |
| 1 | 00 | 001 |
| 2 | 00 | 010 |
| 3 | 00 | 011 |
| 4 | 00 | 100 |
| 5 | 00 | 101 |
| 6 | 00 | 110 |
| 7 | 00 | 111 |
| 0 | 01 | 000 |
| 1 | 01 | 001 |
| 2 | 01 | 010 |
| 3 | 01 | 011 |
| 4 | 01 | 100 |
| 5 | 01 | 101 |
| 6 | 01 | 101 |
| 7 | 01 | 111 |
| 0 | 10 | 000 |
| . | . | . |
| . | . | . |
| . | . | . |

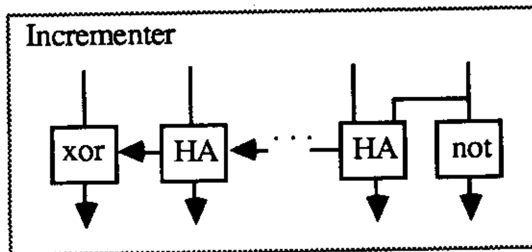
(b)

Figure 1. (a) Modulo - $2^n$ Counter Scheme, (b) Example



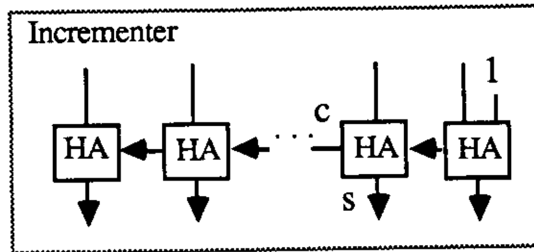

(a)



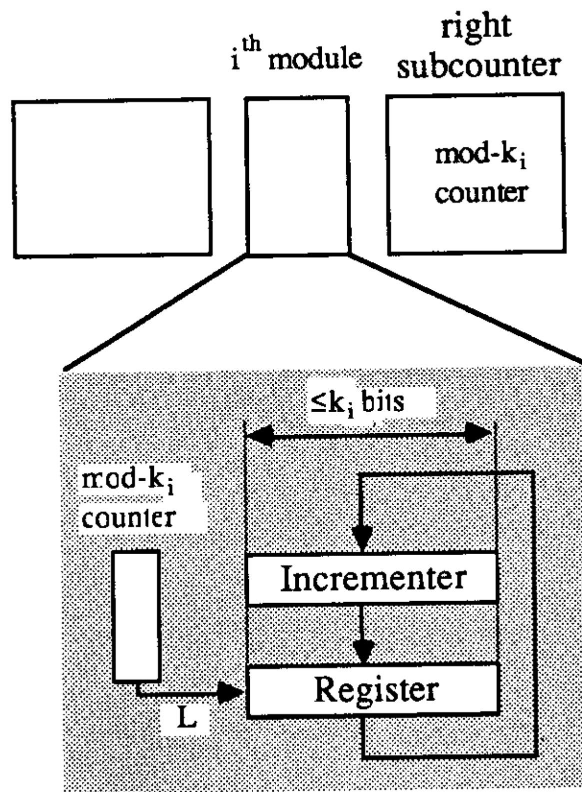




(b)

Figure 2. 64-bit Counter




(a)




(b)

Figure 3. (a) Incrementer, (b) Critical Path

that minimizes this effect.

Suppose that the value $(p-1)$ is represented by the digit vector $P$ such that

$$P = (p_m, p_{m-1}, \ldots, p_2, p_1)$$

so that the value is

$$p-1 = \sum_{i=1}^{m} p_i k_i$$

where

$$K = (k_m, k_{m-1}, \ldots, k_2, k_1)$$

are the weights of this representation and $k_i$ is, as in Section 2, the number of cycles between consecutive updatings of the register of module $i$. Consequently, the comparator of the $j^{th}$ module has $\sum_{i=1}^{j-1} p_i k_i + 1$ cycles for the comparison, since this is the number of cycles between the last updating of the $j^{th}$ module and the time when equality has to be detected.

*Example.* Consider the case where $p-1$ is represented as follows:

$$(p-1) = 511,5,3,1$$

and

$$K = 64,8,2,1 \quad (\textit{for a 64-bit counter})$$

Then, when the left-most module is updated to 511 (and the others to 0), the comparator of the left-most module has $5\times8 + 3\times2 + 1\times1 + 1$ cycles to finish the comparison.

Since the values of these $p_i$'s can be 0, in the worst case there is only one cycle for the comparison. This would require an unacceptably fast comparator to achieve the maximum counting rate discussed in the previous section. Therefore, a modified mode of operation has to be developed.

To avoid the before mentioned problem, we make the comparator of module $j$ detect the value $(p_j - 1)\ mod\ k_j$ (instead of $p_j$). This allows the comparator to have a full $k_j$ cycles to perform the comparison. Since module $j$ is at most $k_j$ bits wide, a simple iterative comparator has enough time to perform the comparison (since the delay per bit is not larger than that of the incrementer).

Now, the condition "count equal to $p_j$" is obtained when the output of the comparator is 1 AND the count value is to be updated (from $p_j-1$ to $p_j$). When this condition is detected, a flip-flop $FF_j$ is set. The value $p-1$ is obtained when all flip-flops are set.

*Example.* Suppose again that $p-1 = 511, 5, 3, 1$ and $K = 64,8,2,1$ (for a 64-bit counter).Then the comparators are set to $510, 4, 2, 0$, respectively. The sequence of setting of the flip-flops is described by the following table.

| Module | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| | 510 | x | x | x |
| | 510 | 7 | 3 | 1 |
| | 511 FF | 0 | 0 | 0 |
| | ... | | | |
| | 511 FF | 4 | 3 | 1 |
| | 511 FF | 5 FF | 0 | 0 |
| | ... | | | |
| | 511 FF | 5 FF | 1 | 1 |
| | 511 FF | 5 FF | 2 FF | 0 |
| | 511 FF | 5 FF | 2 FF | 1 FF |

The value $p-1$ is detected by the AND function of the flip-flop outputs. Since the flip-flops are set in sequence an implementation using an iterative network of 2-input AND gates is suitable; this results in a delay of one 2-input AND gate in the last cycle. A problem arises when the representation of $p-1$ terminates in a sequence of $0's$, that is, $p-1 = x,x,...,0,0,0$, since in this case all flip-flops corresponding to the zeros are set at the same time. Consequently, the delay of the iterative AND gate would increase the cycle time. To solve this, whenever $p-1$ has a representation which ends in more than one zero, the counter is modified to detect $p-2$, whose representation ends in a sequence of $1's$. This detection can use the iterative AND discussed before. To implement the detection of $p-1$, one additional cycle is used. The implementation of a modulo-$p$ counter is shown in Figure 4.

**References**

[1] D. Chu, "Phase Digitizing Sharpens Timing Measurements", IEEE Spectrum, July1988, pp. 28-32.

[2] M. Ercegovac and T. Lang, *Digital Systems and Hardware/Firmware Algorithms*, Wiley and Sons, 1985.

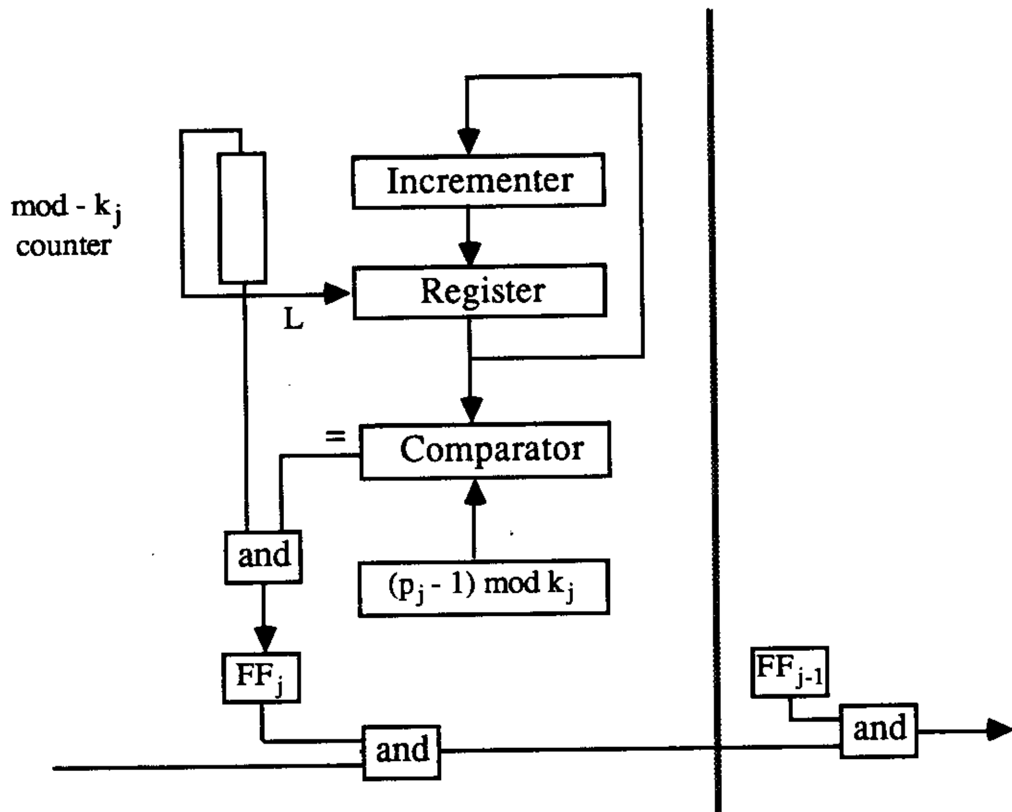



Figure 4. Implementation of Modulo-p Counter with Comparators