



Using the Dedicated Carry Logic in XC4000E

XAPP 013 July 4, 1996 (Version 2.0)

Application Note By BERNIE NEW

Summary

This Application Note describes the operation of the XC4000E dedicated carry logic, the standard configurations provided for its use, and how these are combined into arithmetic functions and counters.

Xilinx Family

XC4000E, XC4000L

Demonstrates

Dedicated Carry Logic

Introduction

XC4000E CLBs contain dedicated, hard-wired carry logic to both accelerate and condense arithmetic functions such as adders and counters. Adders achieve ripple-carry delays as low as 350 ps per bit, while utilizing only half a CLB per bit. This is certainly denser than any other approach, and in most cases, faster.

As shown in [Figure 1](#), the carry logic shares operand and control inputs with the function generators. The carry outputs connect to the function generators, where they are combined with the operands to form the sums. A conceptual diagram of a typical addition is shown in [Figure 2](#).

Only the shared and carry inputs to the function generators are predetermined. Any function of these and the remaining inputs may be implemented. For example, in a loadable counter, the function generator may be used to both invert the counter bit, under control of the carry path, and multiplex a load value into the flip-flop. The H function generator also remains available, and the CLB flip-flops may be used in counters or accumulators.

The ripple-carry outputs are routed between CLBs on high-speed dedicated paths. As shown in [Figure 3](#), carries may be propagated either up or down a column of CLBs. At the top and bottom of the columns where there are no CLBs above and below, the carry is propagated to the right. This enables U-shaped adders and counters to be constructed when they cannot be fitted in a single column.

The carry logic may be configured to implement add, subtract and add/subtract functions. Increment, decrement, increment/decrement and 2's-complement functions are also available.

These functions may be implemented using pre-defined CLB configurations provided in XDE. The mnemonics for these configurations, e.g., ADD-FG-CI, describe the arithmetic function supported, the CLB function generators used and the source of the carry input. While these configurations permit the dedicated carry logic to be used without

detailed knowledge of its operation, the following description is provided.

Operation of the Carry Logic

A detailed and rather complex schematic of the dedicated carry logic is shown in [Figure 4](#). [Figure 5](#), however, is much simpler; it shows the same carry logic once it has been configured for an addition and redundant gates have been removed.

Both bits of the carry logic operate in the same way: First, the A and B inputs are compared. If they are equal, C_{OUT} is well-defined without reference to C_{IN} . When both inputs are zero, carry is not propagated and no carry is generated. Consequently, C_{OUT} must be zero. When they are both one, a carry is generated, and C_{OUT} must also be a one. In either case, C_{OUT} is equal to the A input.

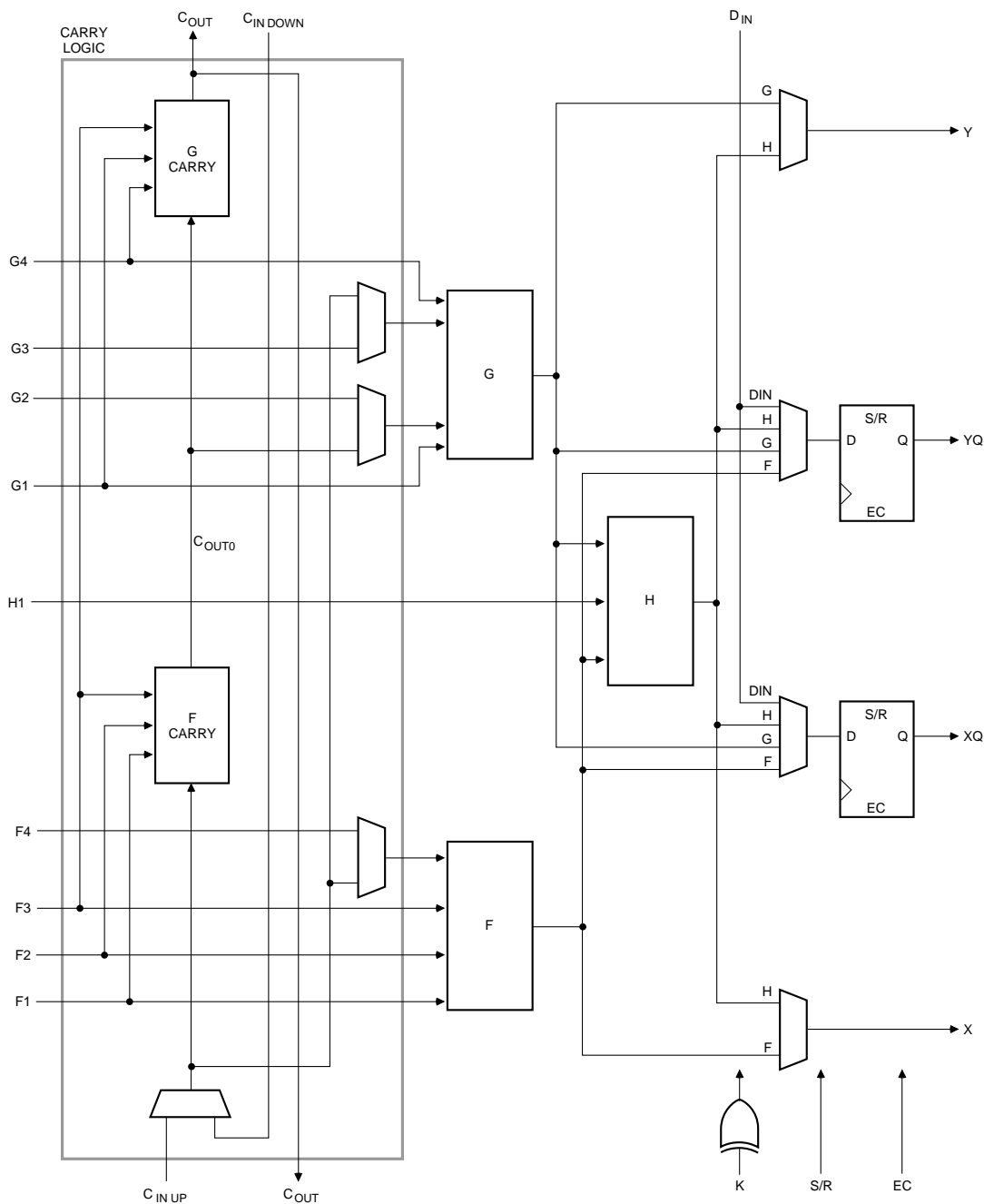
If the A and B inputs are different, the carry is propagated, and C_{OUT} is equal to C_{IN} . C_{OUT} can, therefore, be created by multiplexing between the A input and C_{IN} .

This scheme is used because the multiplexers in the ripple path may be implemented using pass transistors; these introduce the least cumulative delay into this critical path.

Referring back to [Figure 4](#), the various configuration options can now be explained. XOR-gates are provided as polarity controls for the B operands. According to a configuration bit, B may be inverted for a subtracter, or not inverted for an adder. Alternatively, the polarity may be controlled by F3 (ADD/SUBTRACT) for an adder/subtracter.

The B operands may be gated out using a configuration bit in conjunction with two AND gates so that add and subtract can become increment and decrement.

To determine whether carry is propagated up or down the column of CLBs, a multiplexer selected the carry output of the CLB below or the CLB above.



X1997

Figure 1: XC4000E Dedicated Carry Logic

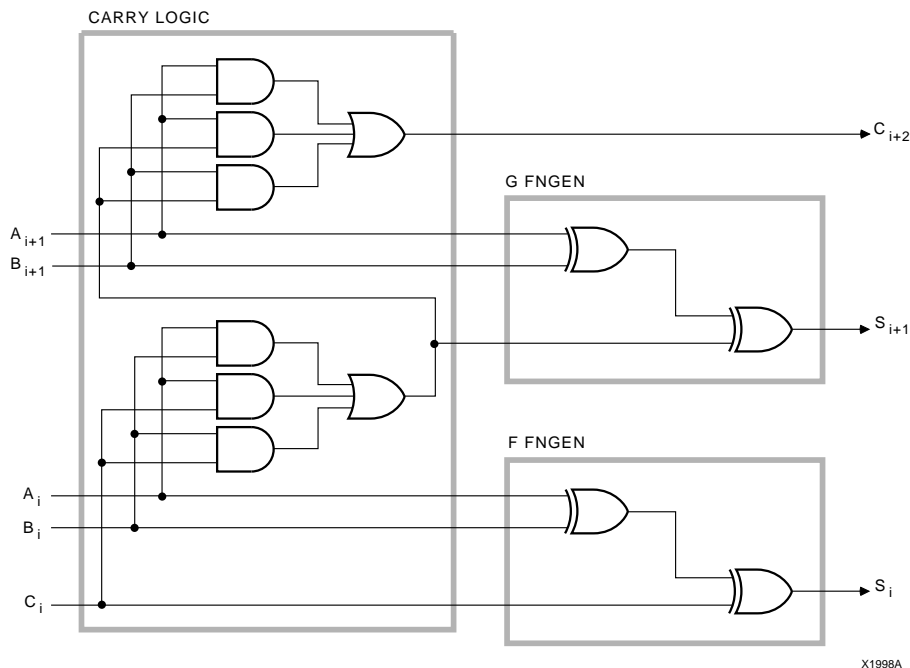


Figure 2: Conceptual Diagram of a Typical Addition (2 Bits/CLB)

If only one adder bit is to be implemented per CLB, the selected carry may be forced to skip the first stage of carry logic. To do this, a configuration bit is set to one and selected to replace the output of the comparator. If the bit is selected and set to zero, an initial value is forced into the carry chain.

This initial value has three sources, determined by the configuration bits. The first source is the configuration bit used to gate out the B operand. When this bit is a one, a 2-oper- and function is performed, and a one at the carry input provides add-with-carry or subtract-without-borrow (borrow is active Low). When the bit is a zero, a 1-operand function is performed, and the carry chain is initialized with a zero.

The second source is $\overline{F3}$. If F3 is not selected as the add/subtract control, it is a free input to the carry chain. If it is used to control addition and subtraction, it provides a zero or one such that the initial carry /borrow is un-asserted in both cases.

The final source is F1. When initialization is selected, this is a free input to the carry chain.

The second stage of the carry logic may also be skipped, in the same way as the first stage. However, there is no initialization function in the second stage.

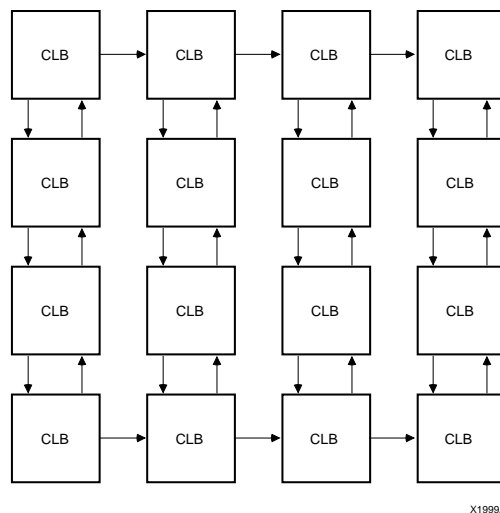


Figure 3: Carry Propagation Paths

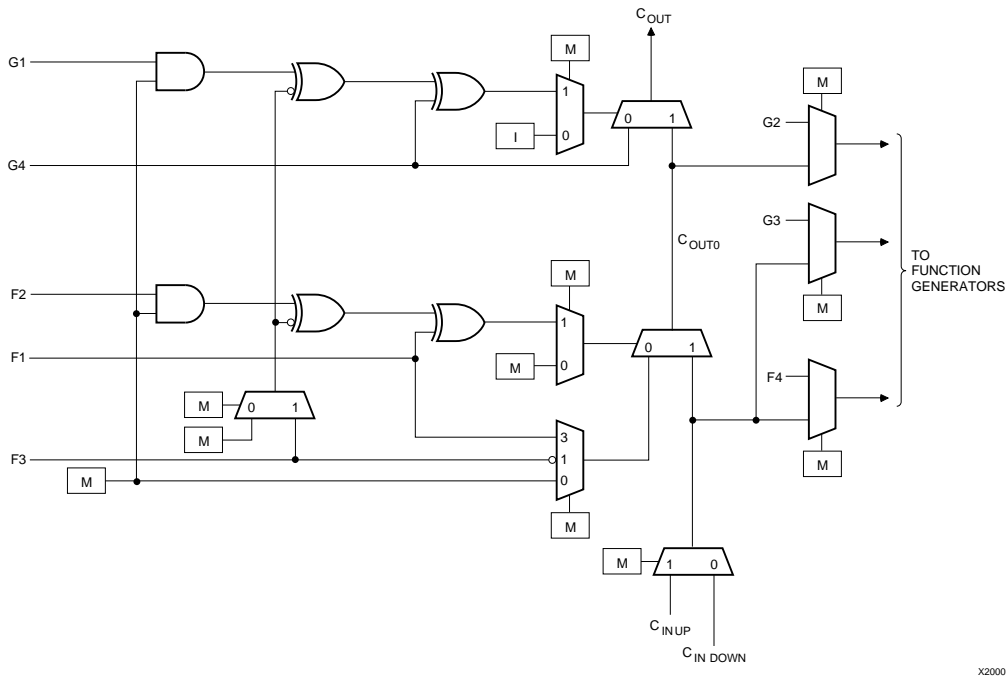


Figure 4: Detail of Dedicated Carry Logic in XC4000E

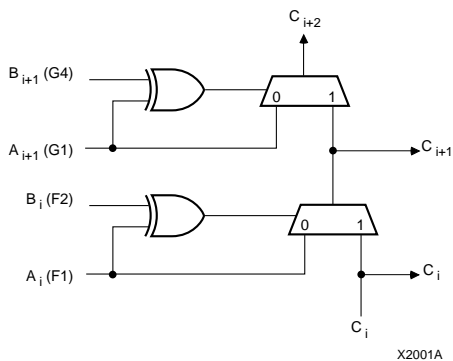


Figure 5: Effective Carry Logic for a Typical Addition

A	B	C	C _{OUT}	
0	0	0	0	A = B, C _{OUT} = A
0	0	1	0	
0	1	0	0	A ≠ B, C _{OUT} = C _{IN}
0	1	1	1	
1	0	0	0	
1	0	1	1	
1	1	0	1	A = B, C _{OUT} = A
1	1	1	1	

X2002A

Figure 6: Effective Carry Logic for a Typical Addition

2-Operand Functions

Adders

An adder implemented with the dedicated carry logic must have at least two sections: a main section and an initialization section. In the main section, shown in [Figure 7](#), one or two bits of the adder are implemented in each CLB, and C_{IN} is taken from the dedicated interconnect. Three standard CLB configurations are provided for this purpose: ADD-FG-CI is a two-bit adder, while ADD-F-CI and ADD-G-CI are one-bit adders with the add occurring in F or G, respectively.

C_{IN} can only be driven by other carry logic. At the least significant end of the adder, special attention must be paid to ensure that the carry path is initialized correctly. This is the function of the initialization section.

The design of the carry logic does not provide for the implementation of two adder bits in the initializing CLB. However, a CLB may be used to initialize the carry path and implement the LSB of the adder. The standard CLB configurations for this are ADD-G-F1 and ADD-G-F3-. In both cases, the addition occurs in G, and the carry input is F1 or F3, respectively.

The use of this technique may create bussing difficulties if other parts of the LCA device have the two LSBs implemented in the same CLB. A second approach that avoids this problem uses a CLB to initialize the carry path without implementing part of the adder.

Four standard CLB configurations are provided for this purpose: FORCE-F1 and FORCE-F3- allow F1 and $\overline{F3}$, respectively, to be used as the carry input, while FORCE-0 and FORCE-1 initialize the carry path with a fixed zero or one, respectively. FORCE-0 and FORCE-1 only involve the carry logic, and all the non-carry resources of the CLB are available for other uses.

Optionally, the adder may have a third section at the most significant end, used to create a carry output (other than on the dedicated interconnect) or to detect overflow. Two situations must be considered: where the most significant CLB contains two bits of the adder, and where it contains only one.

If it contains only one bit of the adder, the standard CLB configuration, ADD-F-CI, in [Figure 8](#) should be used. Both C_{IN} and the most significant carry are available as inputs to the G function generator. The most significant carry may be passed through this, or XOR-ed with C_{IN} to detect two's-complement overflow.

Where both carry and overflow are required, overflow should be generated in the same CLB as the most significant bit. The most significant carry is passed to C_{OUT} , and an additional CLB may be configured to route it to either the F or G output. The EXAMINE-CI configuration is provided for this purpose.

If the most significant CLB contains two bits of the adder, the situation is more complex. As shown in [Figure 9](#), the ADD-F-CI configuration should again be used, despite the need for a 2-bit adder. The most significant bits of the operands should be connected to the G1 and G4 inputs, C_{OUT0} selected as the G2 input, and the G function generator manually programmed as if the configuration were ADD-FG-CI. This causes the most significant sum to be generated at the G output. However, the second stage of carry logic will be bypassed.

An additional CLB can then be used to generate the carry and the overflow. This should be configured as ADD-F-CI and the most significant bits of the operands connected to F1 and F2 in addition to the previous connection. This causes the carry stage, bypassed in the previous CLB, to be implemented in the first stage of this additional CLB. In this way, the necessary carries are available in the G function generator for overflow detection as described above.

The F function generator may be manually programmed to create the most significant carry from the operand bits and C_{IN} . This is permissible as the operation of the carry logic is independent of the function generators.

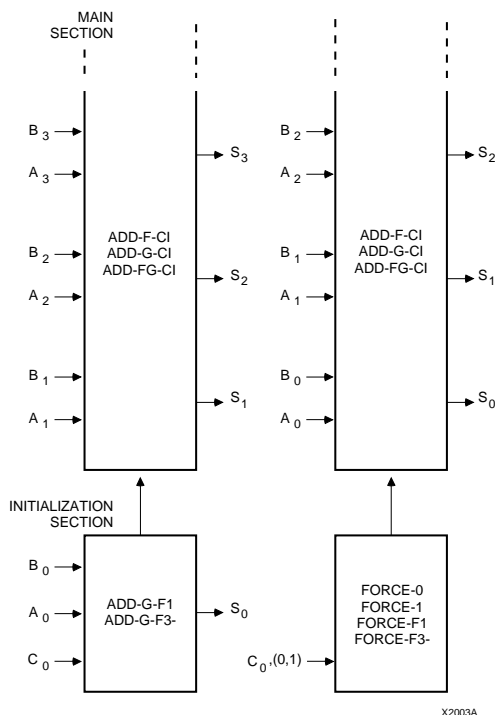
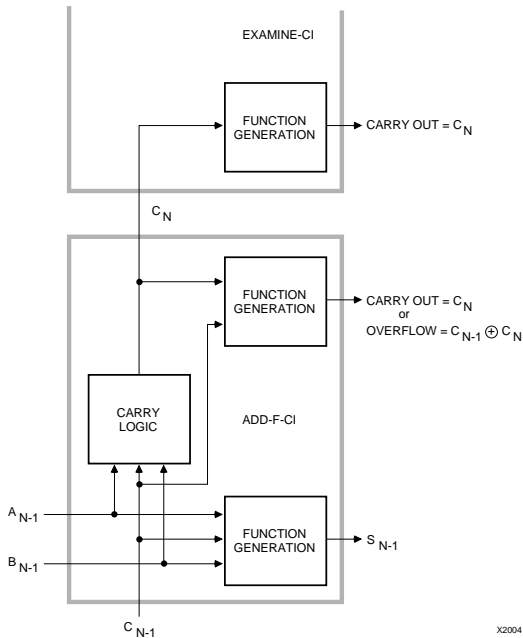


Figure 7: Main and Initialization Sections of Adder



X2004

Figure 8: Carry-Out and Overflow Generation

Subtracters

Subtraction is, in most respects, identical to addition. The subtraction may be written in terms of an addition as follows:

$$A - B = A + (-B)$$

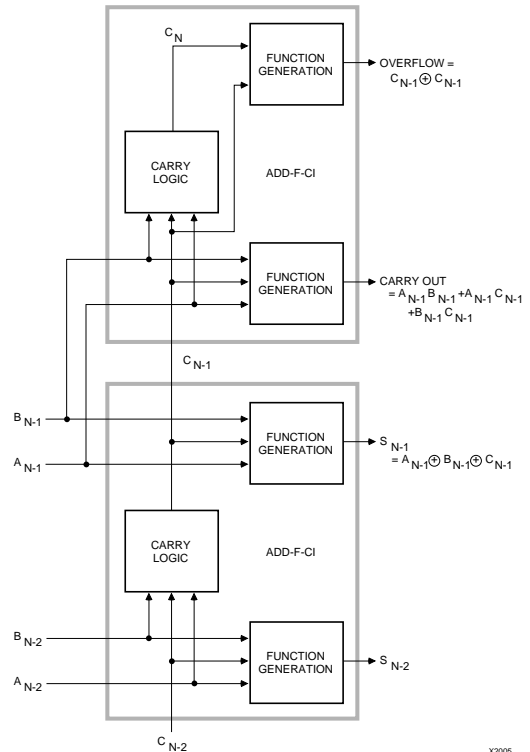
Multiplication by -1, or two's complementing, is performed by logically inverting the operand and adding one. The final form of the subtraction becomes:

$$A - B = A + \bar{B} + 1$$

Using CLB configurations with a SUB prefix, in place of ADD, causes the B operand to be inverted both into the carry logic and within the function generator. The one can be added by forcing the carry into the adder to be High.

An alternative interpretation is that the inversion changes the adder into a subtracter, with the carry becoming an active-Low borrow. Consistent with the first interpretation, the carry input must be High for borrow not to be asserted. If the carry input is Low, the operation is

$$A - B - 1.$$



X2005

Figure 9: Carry-Out and Overflow Generation with Duplicated MSB

Apart from using CLB configurations with the SUB prefix and ensuring that carry-in has the right polarity, subtracters may be constructed in the same way as adders. Equivalent configurations exist for all three sections of the subtracter. The only point to remember is that, when manually configuring function generators for the most significant output or carry output, the B operand must be inverted. The definition of overflow does not change.

One configuration that exists for subtraction, but not for addition, is SUB-G-1. In this configuration, the least significant bit of the subtraction takes place in G with the carry input internally forced to a one (no borrow).

Adder/Subtracters

The adder may be converted to an adder/subtractor by making the inversion of the B operand programmable. This is accomplished using CLB configurations with an ADDSUB prefix.

The ADD/SUBTRACT control is connected to F3, and controls the operation of both the carry logic and the F function generator. If the configuration uses the G function generator, ADD/SUBTRACT must also be connected to G3.

The carry input to the adder/subtractor must be determined by the operation being performed. When an add is in progress, it must be Low for a carry not to be asserted, and it must be High for a borrow not to be asserted during a subtraction.

This will generally preclude the use of FORCE-0 and FORCE-1 to initialize the carry chain. Otherwise, the adder/subtractor is constructed in the same way as the adder, but using CLB configurations with the ADDSUB prefix.

As in the subtracter, the programmable operand inversion must be remembered in any function generators that are manually configured

1-Operand Functions

Incrementers

Essentially, an incrementer is an adder with one operand zero, and the carry input asserted. Consequently, incrementers are constructed in the same way as adders, but using CLB configurations with an INC prefix. These gate out the B operand.

The carry input should be High to increment the A operand, and Low to pass it unchanged. Alternatively, it may be fixed High for permanent incrementation. This may be accomplished using CLB configurations equivalent to those used to initialize adders. In addition, INC-G-1 and INC-FG-1 allow the carry chain to be initialized with the carry asserted, along with one or two bits of the function.

Decrementers

These are subtracters with the B operand zero and a borrow asserted. CLB configurations with a DEC prefix gate out the B operand before it is inverted. The carry input should be Low to decrement the A operand, and High to pass it.

Alternatively, a fixed Low may be used. DEC-G-0 and DEC-FG-0 provide this, along with one or two bits of the function. FORCE-0 may also be used.

Incrementer/Decrementers

Not surprisingly, these are constructed in the same way as adder/subtracters, but using cells with an INCDEC prefix that gate out the B operand. When increment is selected,

the carry input should be High to increment or Low to pass. When decrement is selected, the carry should be Low to decrement or High to pass. INCDEC-FG-1 implements two least significant bits of the incrementer/decrementer with the carry or borrow input permanently asserted.

2's Complementers

The traditional two's-complement procedure, invert-and-add-one, is not appropriate for use with the dedicated carry logic. In the increment configuration, the A operand cannot be inverted at the input to the carry logic, and using a subtracter for $0 - B$ consumes unnecessary resources routing the zero operand.

The answer is to replace invert-and-increment with decrement-and-invert, which produces the same result. A conventional decrementer is constructed, and an additional output inversion is programmed into the function generators.

The use of a function generator input allows this inversion to become programmable. In conjunction with control of the carry input, this programmable inversion may be used to two's complement a number or pass it, as required.

Counters

Up Counters

An up counter is constructed by combining an incrementer with a register, as shown in [Figure 10](#). Typically, the register in the same CLBs as the incrementer is used, and the sum outputs should be routed to this register. The output of the register is fed back as the input to the incrementer. Each clock, the register is loaded with a value one greater than its previous value.

Any incrementer may be used. If it has the ability to increment or pass the operand, this feature may be used as a count enable.

As shown in [Figure 11](#), counters may easily be made loadable by adding a multiplexer into the function generators. This multiplexer selects between the incrementer output and the value to be loaded as the source for the register.

Down Counters

Down counters are constructed in the same way as up counters, but using decrementers in place of incrementers.

Up/Down Counters

Incrementer/decrementers are used for up/down counters. The only significant difference comes in the loadable counter. Because the INC/DEC control is an input to the function generators, there are not enough inputs available for the load function. One CLB must be used for each bit of the counter, and there are several ways in which this can be organized.

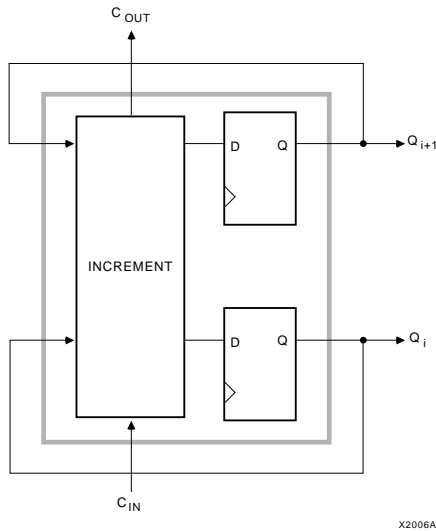


Figure 10: Typical Counter CLB

One possibility is to use a CLB configuration that only implements one bit of the incrementer/decrementer function, as shown in Figure 12. The H function generator can then be used as the load multiplexer. The H1 input acts as the Parallel Enable, and the value to be loaded is passed through the second function generator.

A better choice is to construct the incrementer and decrementer separately in two columns of CLBs with two bits per CLB, as shown in Figure 13. The decrementer is connected as a conventional loadable down counter. In the incrementer, the function generators are modified with a multiplexer, as is it were to be a loadable up-counter. However, the register is not connected, and data is not fed back.

Instead, the input to the incrementer is taken from the output of the down counter, and the incrementer output is routed to what would have been the down-counter load input. The value to be loaded is input to the multiplexer attached to the incrementer.

The load control of the down counter becomes the up/down control, selecting the output of either the incrementer or the decrementer. Data is loaded by replacing the incrementer output with the value to be loaded, and selecting count up. An external gate may be required to force the up/down control.

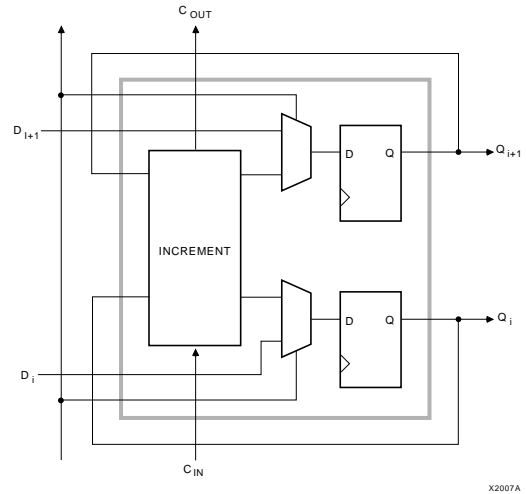
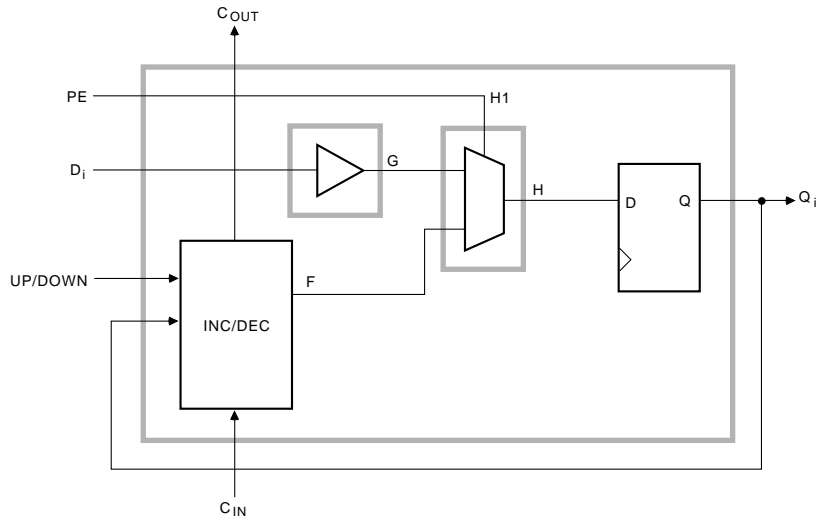


Figure 11: Typical Loadable Counter CLB

This second approach has the advantage that its layout is compatible with other functions that implement two bits per CLB. More importantly, however, it is faster. The incremental carry delay is incurred per CLB, not per bit, and implementing two bits per CLB halves the number of carry delays. Also, the set-up time on the up/down control is much shorter. The up/down control need only select the output of the incrementer or decrementer, instead of selecting the increment or decrement function before carry/borrow propagation can begin. Both the incrementer and decrementer operate in parallel, starting immediately after the clock.

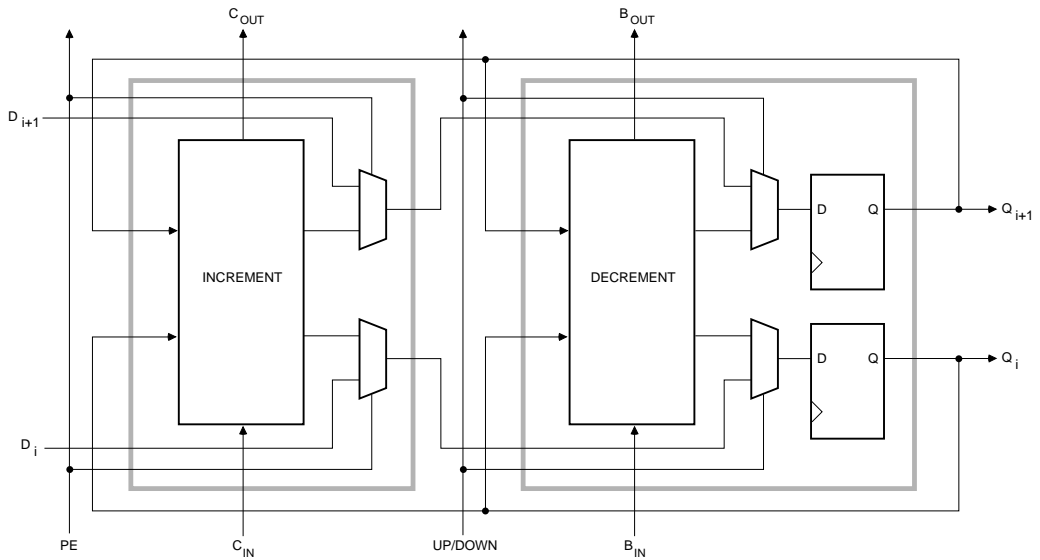
Alternatively, an incrementer/decrementer may be implemented in one column of CLBs, with the register and load multiplexers implemented in a second column. A count-enable multiplexer can be built into the same function generator as the load multiplexer. If this is placed logically in front of the load multiplexer, the load control takes precedence over the Count Enable.

This scheme eliminates the additional gating required to ensure that the counter is enabled and counting up during a load. The Load and Count Enable controls are both fast, but the set-up time for the up/down control is similar to the carry-propagation delay.



X2008A

Figure 12: Typical Up/Down Counter CLB



X2009

Figure 13: Up/Down Counter with Separate Incrementer and Decrementer

Timing Analysis

Typically, the critical delay is from the carry input or operand LSB to the output MSB, carry output or overflow flag. As shown in Figure 14, this delay has three parts: The delay onto the carry chain from the input, the delay from the carry chain to the output and the delay of the intervening CLBs.

If part of the function is performed in the CLB that initializes the carry chain, the delay onto the chain is the greater of the operand-input-to-C_{OUT} (T_{OPCY}) and the initialization-input-to-C_{OUT} (T_{INCY}) delays. If a CLB is used for initialization only, separate delays must be calculated from the least significant operand input and the initialization input, taking into account the different number of intervening CLBs.

The output delay (T_{SUM}) is from C_{IN} to the output. Each intervening CLB introduces a T_{BYP} delay.

To calculate the minimum clock period in a counter, the clock-to-output delay and a routing delay must be added to the operand input delay. Typically, in a -3 part, this routing delay is 1.2 ns; but this must be verified by simulation after the implementation is complete. The output delay must be replaced with the equivalent set-up time (T_{CKK}), and the intervening CLBs must be taken into account, as in the basic delay calculation.

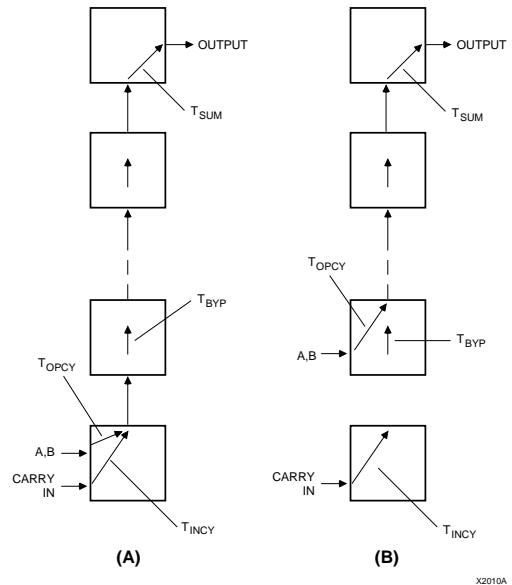
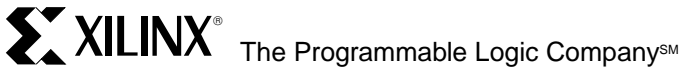


Figure 14: Carry-Logic Delay Paths

X2010A



Headquarters

Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124
U.S.A.
Tel: 1 (800) 255-7778
or 1 (408) 559-7778
Fax: 1 (800) 559-7114
Net: hotline@xilinx.com
Web: http://www.xilinx.com

North America

Irvine, California
(714) 727-0780

Englewood, Colorado
(303) 220-7541

Sunnyvale, California
(408) 245-9850

Schaumburg, Illinois
(847) 605-1972

Nashua, New Hampshire
(603) 891-1098

Raleigh, North Carolina
(919) 846-3922

West Chester, Pennsylvania
(610) 430-3300

Dallas, Texas
(214) 960-1043

Europe

Xilinx Sarl
Jouy en Josas, France
Tel: (33) 1-34-63-01-01
Net: frhelp@xilinx.com

Xilinx GmbH
Aschheim, Germany
Tel: (49) 89-99-1549-01
Net: dlhelp@xilinx.com

Xilinx, Ltd.
Byfleet, United Kingdom
Tel: (44) 1-932-349401
Net: ukhelp@xilinx.com

Japan

Xilinx, K.K.
Tokyo, Japan
Tel: (03) 3297-9191

Asia Pacific

Xilinx Asia Pacific
Hong Kong
Tel: (852) 2424-5200
Net: hongkong@xilinx.com

© 1996 Xilinx, Inc. All rights reserved. The Xilinx name and the Xilinx logo are registered trademarks, all XC-designated products are trademarks, and the Programmable Logic Company is a service mark of Xilinx, Inc. All other trademarks and registered trademarks are the property of their respective owners.

Xilinx, Inc. does not assume any liability arising out of the application or use of any product described herein; nor does it convey any license under its patent, copyright or maskwork rights or any rights of others. Xilinx, Inc. reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx, Inc. cannot assume responsibility for the use of any circuitry described other than circuitry entirely embodied in its products. Products are manufactured under one or more of the following U.S. Patents: (4,847,612; 5,012,135; 4,967,107; 5,023,606; 4,940,909; 5,028,821; 4,870,302; 4,706,216; 4,758,985; 4,642,487; 4,695,740; 4,713,557; 4,750,155; 4,821,233; 4,746,822; 4,820,937; 4,783,607; 4,855,669; 5,047,710; 5,068,603; 4,855,619; 4,835,418; and 4,902,910. Xilinx, Inc. cannot assume responsibility for any circuits shown nor represent that they are free from patent infringement or of any other third party right. Xilinx, Inc. assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made.