Efficient Arithmetic Designs Targeting FLASH370™ CPLDs

Introduction

The design of fast and efficient arithmetic elements is imperative because of its applications in the many areas of science and engineering. It is important for designers to be aware of the choices available to them in selecting an efficient algorithm for their application. Even the seemingly simple arithmetic operations turn out to be more complex than one expects, when attempting to implement them. There is a lot of literature available in the field, but very little provides the level of detail required to go all the way from a concept to a final implementation.

This application note is intended to help designers create efficient arithmetic designs targeting a FLASH370™ complex programmable logic device (CPLD). The designer has many alternatives in choosing between arithmetic implementations for a given design. The decision on the final choice is typically based on issues like resource availability, speed of operation, and modularity. Creating designs in view of the target device's architecture will definitely yield better results than implementing a generic design on the same device. The discussion in this application note addresses arithmetic algorithms, design methodologies, and implementations tailored to the features and resources offered in the FLASH370 family of CPLDs. These specialized arithmetic designs achieve a balanced tradeoff between speed/area requirements for a given application. In this application note the user is offered a wide variety of algorithms and implementations to choose from. This variety provides the designer with the flexibility to choose the model best suited for the target application. This choice is absolutely necessary, since design requirements and constraints vary from application to application.

The discussion assumes that the designer has a good feel for the features and resources available in the FLASH370 family of CPLDs. The implementation details and design tradeoffs in building adders, subtracters, equality and magnitude comparators are addressed in this application note. This application note includes many VHDL (VHSIC Hardware Design Language) examples to illustrate the working and implementation of the algorithms presented. Block diagrams are also presented wherever necessary to help the designer understand the design better.

All algorithms in this application note are described within the same framework, so that the similarities between different algorithms become evident and consequently, the basic principle behind these algorithms can be easily identified. This application note is also intended to create a solid foundation from which designers can pick up ideas and concepts and create their own algorithms/implementations.

The VHDL code presented in this application note are intentionally presented in a simple style. The intent of this application note is to allow a designer to visualize and implement arithmetic models efficiently and not to explain how to code them. All VHDL keywords are presented in italics. This application note also assumes that the reader has a good grasp of the fundamentals of VHDL. Some of the LPM (library of parameterized modules) elements for CPLDs provided in the *Warp*™ software are built using the concepts and final implementations discussed here. This provides the user with an excellent opportunity to choose the best algorithm

and implementation tailored to the target application.

Adders

The addition of two operands is the most frequent operation in almost any arithmetic unit. The two-operand adder is commonly used in performing additions and subtractions. It is also used when executing complex arithmetic functions like multiplication and division.

ADD : 1-Bit Full Adder

The basic component used in adding two operands is called a *Full Adder*. The full adder element will be henceforth referred to as the 'ADD' component. The block diagram and functionality of ADD is shown in *Figure 1*. A and B are the two operands to be added and CI is the Carry-in to the component. SUM and CO are the Sum and Carry-out from the component.

The VHDL code describing the functionality of the ADD component is shown here. This design takes one pass through the Logic (AND-OR) array to fit into a FLASH370 CPLD. The ADD component instantiated in the VHDL code shown has exactly the same functionality shown in *Figure 1*.

```
-- This VHDL code invokes the im-
plementation of the MATH PKG ele-
ment ADD

USE WORK.CYPRESS.ALL;
USE WORK.MATHPKG.ALL;

ENTITY add IS
 PORT (CI: IN BIT;
   A, B: IN BIT;
   SUM: OUT BIT;
   CO: OUT BIT);
END add;

ARCHITECTURE archadd OF add IS

BEGIN

 i1: add PORT MAP(CI,A,B,SUM,CO);

END archadd;
```

RADD12 : 12-Bit Ripple Carry Adder

An *n*-bit two-operand ripple carry adder can be built using *n* ADD components. All the 2*n* input bits are available to the adder at the same time. However the carries have to propagate from the LSB position to the MSB. In other words, we need to wait until the carries ripple through *n* ADD components to claim that the SUM outputs are correct. Because of this rippling effect, the adder is referred to as the *Ripple Carry* Adder. This is the simplest form of adding any two operands. It uses the least amount of area compared to all other implementations but, on the negative side, is the slowest implementation. This is typically the implementation provided with a synthesis tool when it recognizes the '+' operator in a VHDL code. The block diagram of a 12-bit Ripple Carry Adder (RADD12) is shown in *Figure 2*.

The VHDL code describing the functionality of the RADD12 component is shown here. This design takes 12 passes through the logic array to fit into a FLASH370 CPLD. The outputs of the LSB ADD

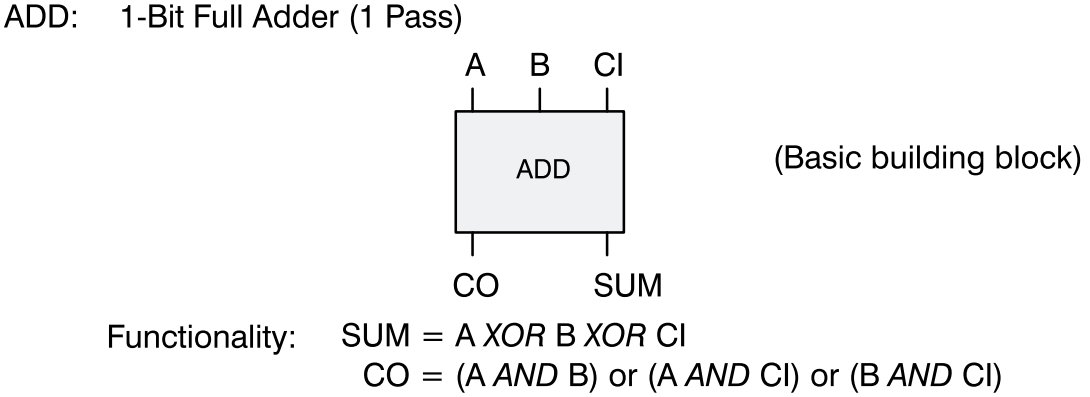

Figure 1. Block Diagram and Functionality of a Full Adder

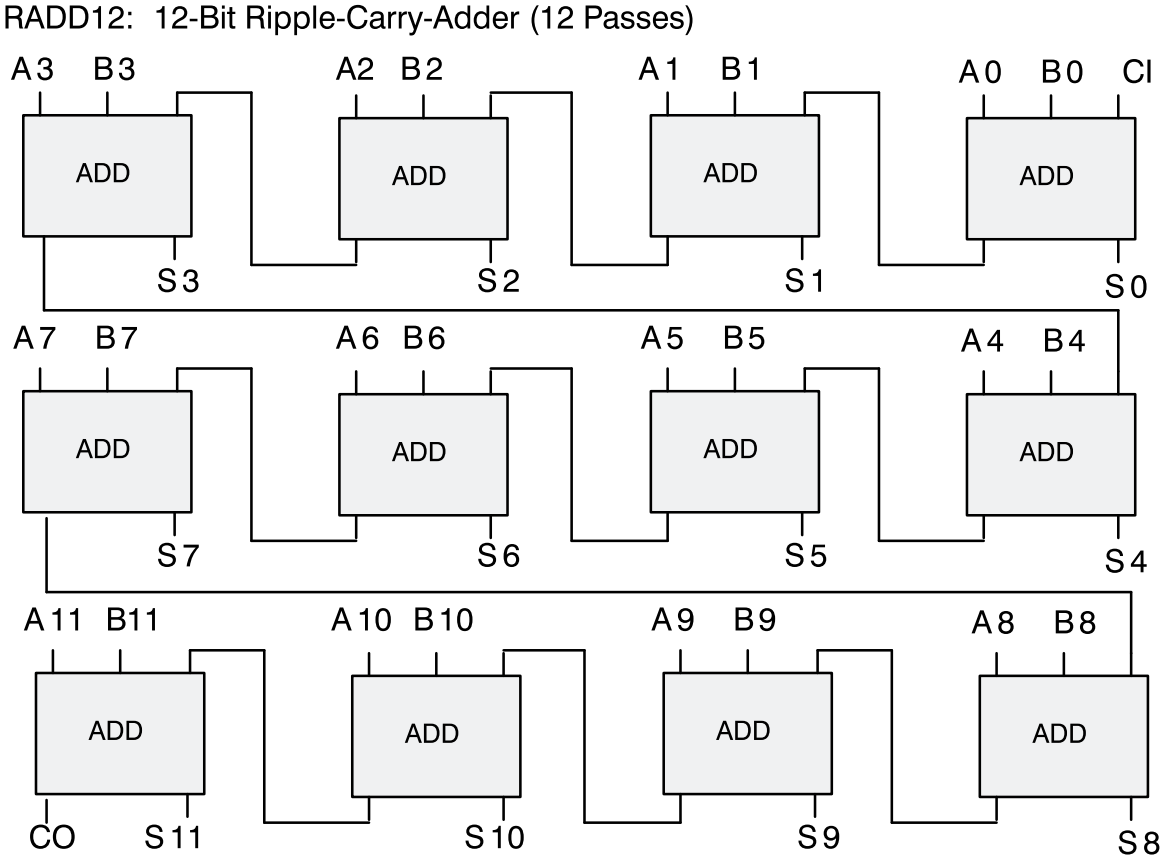

Figure 2. Block Diagram of a 12-Bit Ripple Carry Adder

component are produced in the first pass. The outputs of the succeeding ADD components are produced with every alternate pass through the logic array. Each pass through the logic array has a time penalty associated with it. It is recommended that the reader understand the timing issues associated with the FLASH370 CPLD (refer to the "CY7C37x Timing Parameters" application note).

```
--This VHDL code describes the implementation of a generic
--12 bit ripple carry adder.

USE WORK.CYPRESS.ALL;
USE WORK.MATHPKG.ALL;

ENTITY rippleadd12 IS
  PORT (CI: IN BIT;
    A11, A10, A9, A8, A7, A6, A5, A4, A3, A2, A1, A0 : IN BIT;
    B11, B10, B9, B8, B7, B6, B5, B4, B3, B2, B1, B0 : IN BIT;
    SUM11, SUM10, SUM9, SUM8, SUM7, SUM6, SUM5, SUM4,
    SUM3, SUM2, SUM1, SUM0 : OUT BIT;
    CO: OUT BIT);
END rippleadd12;

ARCHITECTURE archripple12add OF rippleadd12 IS
```

```
SIGNAL C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11 : BIT;

attribute synthesis_off of C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11 :
signal is true;

BEGIN

  i1: add PORT MAP(CI,A0,B0,SUM0,C1);
  i2: add PORT MAP(C1,A1,B1,SUM1,C2);
  i3: add PORT MAP(C2,A2,B2,SUM2,C3);
  i4: add PORT MAP(C3,A3,B3,SUM3,C4);
  i5: add PORT MAP(C4,A4,B4,SUM4,C5);
  i6: add PORT MAP(C5,A5,B5,SUM5,C6);
  i7: add PORT MAP(C6,A6,B6,SUM6,C7);
  i8: add PORT MAP(C7,A7,B7,SUM7,C8);
  i9: add PORT MAP(C8,A8,B8,SUM8,C9);
  i10: add PORT MAP(C9,A9,B9,SUM9,C10);
  i11: add PORT MAP(C10,A10,B10,SUM10,C11);
  i12: add PORT MAP(C11,A11,B11,SUM11,CO);

END archripple12add;
```

The need and use for the 'Synthesis_off' attribute used in the VHDL code will be discussed a little later.

ADD2WC: 2-Bit Adder with Carry-Out

The concept of the ADD component can be extended to create a 2-bit adder which takes in two 2-bit operands with a carry-in and produces a 2-bit SUM and a carry-out as outputs. This component will be referred to as the ADD2WC (2-bit adder with a carry-out). This also takes just one pass through the logic array to yield results. The block diagram of ADD2WC is shown in *Figure 3*. A0, A1 and B0, B1 are the two operands to be added and CI is the Carry-in to the component. S0, S1 and CO are the Sums and Carry-outs from the component.

The VHDL code describing the functionality of the ADD2WC component is shown here. This design takes one pass through the logic array to fit into a FLASH370 CPLD.

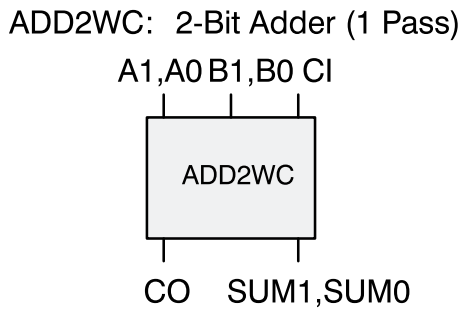

Figure 3. A 2-Bit Full Adder with a Carry-Out

```
--VHDL code describing a 2-bit adder with carry-out.

USE WORK.RTLPKG.ALL;

PACKAGE add2wc_pkg IS
  COMPONENT add2wc PORT(
    CI : IN BIT;
    A1,A0: IN BIT;
    B1,B0: IN BIT;
    SUM1,SUM0 : OUT BIT;
```

```
    CO: OUT BIT);
  END COMPONENT;
END add2wc_pkg;

ENTITY add2wc IS
  PORT(CI : IN BIT;
       A1,A0: IN BIT;
       B1,B0: IN BIT;
       SUM1,SUM0 : OUT BIT;
       CO: OUT BIT);
END add2wc;

ARCHITECTURE archadd2wc OF add2wc IS

BEGIN

SUM0 <= A0 XOR B0 XOR CI;
SUM1 <= A1 XOR B1 XOR ((A0 AND B0) OR (A0 AND CI) OR (B0 AND CI));
CO   <= (A0 AND B0 AND B1)
  OR (A0 AND B0 AND A1)
  OR (CI AND B0 AND B1)
  OR (CI AND B0 AND A1)
  OR (CI AND A0 AND B1)
  OR (CI AND A0 AND A1)
  OR (A1 AND B1);

END archadd2wc;
```

The concept of ADD2WC can be extended to describe the ADD2NC component. The ADD2NC component is a cut-down version of the ADD2WC component, and does *not* have a carry-out. The VHDL code and block diagram for the ADD2NC component is easy to extrapolate and is not shown here.

R2ADD12: 12-Bit Ripple Carry Adder using the ADD2WC as a Basic Block

A 12-bit adder using the ADD2WC component is shown here. This adder takes 6 passes to produce all results, as opposed to the 12 passes needed for the 12-bit adder using the ADD component. The outputs of the LSB ADD2WC component are produced in the first pass. The outputs of the succeeding ADD2WC components are produced with every alternate pass through the logic array. The number of macrocells used by this scheme is less than RADD12, but the product term count is higher. A comparison of different schemes is presented later. The block diagram of R2ADD12 is shown in *Figure 4*. The VHDL code describing the functionality is also attached.

R2ADD12: 12-Bit Adder using ADD2WC (6 Passes)

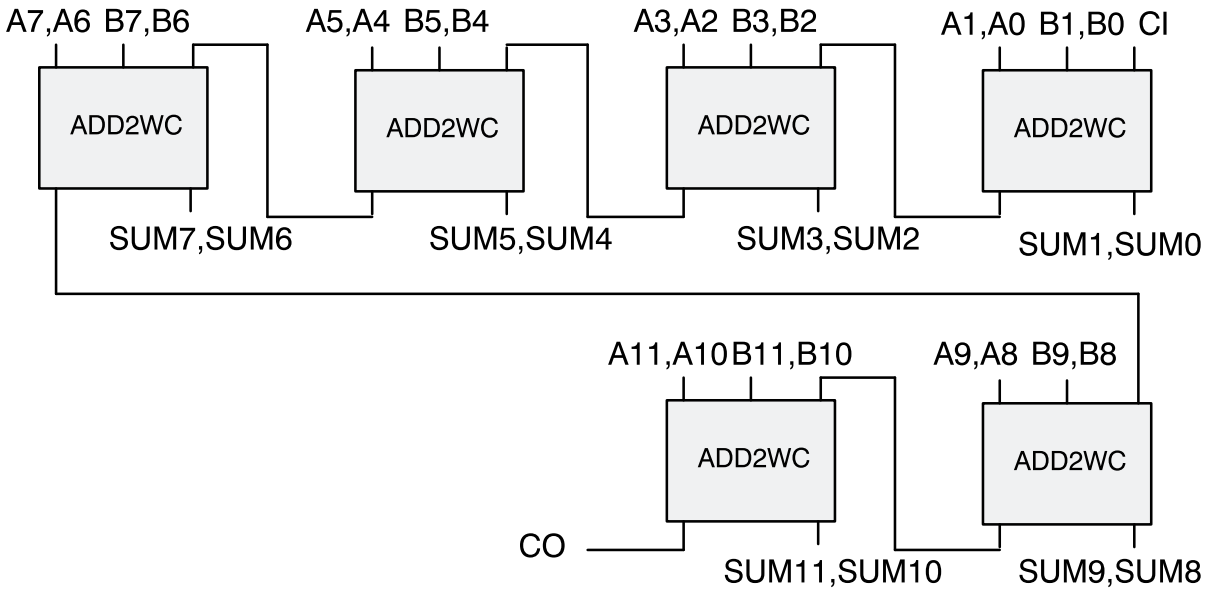

Figure 4. Block Diagram of a 12-Bit Ripple Carry Adder Using 2-Bit Adders

```
--A 12-bit Ripple carry adder built using the ADD2WC element as a basic
--building block

USE WORK.RTLPKG.ALL;
USE WORK.ADD2WC.ALL;

ENTITY add12 IS
  PORT(CI : IN BIT;
       A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0: IN BIT;
       B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0: IN BIT;
       SUM11,SUM10,SUM9,SUM8,SUM7,SUM6,SUM5,SUM4,
       SUM3,SUM2,SUM1,SUM0 : OUT BIT;
       CO: OUT BIT);
END add12;

ARCHITECTURE archadd12 OF add12 IS

  SIGNAL C2, C4, C6, C8, C10 : BIT;

  attribute synthesis_off of C2, C4, C6, C8, C10 : signal is true;

BEGIN

  i1: add2wc PORT MAP(CI,A1,A0,B1,B0,SUM1,SUM0,C2);
  i2: add2wc PORT MAP(C2,A3,A2,B3,B2,SUM3,SUM2,C4);
  i3: add2wc PORT MAP(C4,A5,A4,B5,B4,SUM5,SUM4,C6);
```

```
  i4: add2wc PORT MAP(C6,A7,A6,B7,B6,SUM7,SUM6,C8);
  i5: add2wc PORT MAP(C8,A9,A8,B9,B8,SUM9,SUM8,C10);
  i6: add2wc PORT MAP(C10,A11,A10,B11,B10,SUM11,SUM10,CO);

END archadd12;
```

ADD3WC: The 3-Bit Ripple Carry Adder

There is yet another way we could implement an *n*-bit ripple carry adder targeting the FLASH370 CPLDs. We can implement the *n*-bit adder using the 3-bit group adder (ADD3WC) as opposed to a 2-bit group adder (ADD2WC). The problem with a 3-bit group adder is the sum-splitting of the functionality of the MSB Sum bit (SUM2). This takes more than 16 product terms (PTs) and takes 2 passes through the logic array to produce the result. All other results, including the carry-out, take less than 16 PTs and take just one pass to produce results. To control sum-splitting the functionality of SUM2, the intermediate carry C2 is created and assigned to a node. C2 is then used to create the functionality of SUM2. Note that the functionality of CO takes less than 16 PTs and is generated at the first pass, so the carry rippling is faster. This makes this component a faster building block. This scheme still takes two passes to create the functionality of SUM2, but without getting sum-split. The resource utilization of a 12-bit adder using the 3-bit group adder is presented later. The block diagram of the ADD3WC component is shown in *Figure 5*.

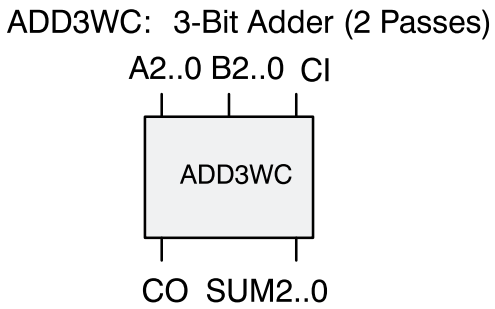

Figure 5. A 3-Bit Full Adder with a Carry-Out

```
-- 3-Bit Adder with Carry-out

PACKAGE add3wc_pkg IS
  COMPONENT add3wC
    PORT (CI : IN BIT;
      A2,A1,A0: IN BIT;
      B2,B1,B0: IN BIT;
      SUM2,SUM1,SUM0 : OUT BIT;
      CO: OUT BIT);
  END COMPONENT;
END add3wc_pkg;

ENTITY add3wc IS
  PORT (CI : IN BIT;
    A2,A1,A0: IN BIT;
    B2,B1,B0: IN BIT;
    SUM2,SUM1,SUM0 : OUT BIT;
    CO: OUT BIT);
END add3wc;

ARCHITECTURE archadd3wc OF add3wc IS

SIGNAL C2: BIT;

  attribute synthesis_off of C2: signal is true;
```

```
BEGIN

  SUM0 <= A0 XOR B0 XOR CI;
  SUM1 <= A1 XOR B1 XOR ((A0 AND B0) or (A0 AND CI) or (B0 AND CI));
  SUM2 <= A2 XOR B2 XOR C2;

  C2   <= (A0 AND B0 AND B1)
    OR (A0 AND B0 AND A1)
    OR (CI AND B0 AND B1)
    OR (CI AND B0 AND A1)
    OR (CI AND A0 AND B1)
    OR (CI AND A0 AND A1)
    OR (A1 AND B1);

  CO <= (A2 AND B2) OR ((A1 AND B1) AND (A2 OR B2))
    OR ((A0 AND B0) AND (A1 OR B1) AND (A2 OR B2))
    OR (CI AND (A0 OR B0) AND (A1 OR B1) AND (A2 OR B2));

END archadd3wc;
```

Function and Use of the *Synthesis_off* Attribute

The Synthesis_off attribute causes a signal to be made into a factoring point for logic equations and keeps the signal from being minimized out during optimization.

The attribute is useful for the following reasons:

1. It gives the user control over which equations or sub-expressions need to be factored into a node.
2. It helps in cutting down on compile time for designs that have a lot of 'signal redirection' (signals getting inverted/reassigned to other signals). This attribute provides the Logic optimizer a better control over the optimization process by reducing the number of signals it needs to deal with.
3. It provides better results for designs where a signal with a large functionality is being used by many other signals. If left alone, the fitter would collapse all the internal signals (which is desirable in many cases) and may drive the design's resource requirements beyond the available limits.

By using the Synthesis_off attribute, the user can assign the commonly-used signal to a node and bring down the resource utilization.

A side effect of using the Synthesis_off attribute is that the design will now take an extra pass through the array to achieve the same functionality. The extra pass may be required anyway, if more than 16 PTs are required.

This attribute is only recommended for use on combinatorial signals. Registered signals are assigned to a node by natural factoring and the Synthesis_off attribute on these signals is redundant.

This attribute can be associated with signals declared both in VHDL and schematics. The 'BUF' component can also be used in schematics and VHDL to achieve the same results as the Synthesis_off attribute. Please refer to the *Warp* Synthesis manual for more details.

Carry-Lookahead Principle

The predominant delay in adders is due to carry propagation. The carry-lookahead principle aims at minimizing this delay. The sum and carry equations for each bit position in an adder is given by:

$$S_i = A_i \text{ xor } B_i \text{ xor } C_i$$
$$C_{i+1} = (A_i \text{ and } B_i) \text{ or } (A_i \text{ and } C_i) \text{ or } (B_i \text{ and } C_i)$$

A carry is *generated* whenever A_i and B_i are both '1' and a carry is *propagated* whenever either A_i or B_i are '1'.

Generate term : $(G_i = A_i \text{ and } B_i)$
Propagate term : $(P_i = A_i \text{ or } B_i)$

Note: P_i can be (A_i xor B_i), but 'OR' is easier to implement than an 'XOR' in CPLDs.

Rewriting the equation for C_{i+1}, we get

$$C_{i+1} = G_i \text{ or } (P_i \text{ and } C_i)$$

Writing the equations for a 4-bit carry-lookahead adder:

$$C_1 = G_0 \text{ or } (P_0 \text{ and } C_0)$$
$$C_2 = G_1 \text{ or } (P_1 \text{ and } C_1)$$
$$C_3 = G_2 \text{ or } (P_2 \text{ and } C_2)$$
$$C_4 = G_3 \text{ or } (P_3 \text{ and } C_3)$$

where $G_i = (A_i \text{ and } B_i)$ and $P_i = (A_i \text{ or } B_i)$. The values of G_i and P_i can be generated in a single pass through the PIM array. The carry-in to any of the bit positions can be computed in a second pass through the array, based upon the values of the various G_is and P_is generated in the first pass.

The generalized carry-lookahead equation to compute the different carry-in signals is shown here:

$$C_{i+1} = G_i \text{ or } (P_i \text{ and } G_{i-1}) \text{ or } (P_i \text{ and } P_{i-1} \text{ and } G_{i-1}) \text{ or } \ldots \text{ or } (P_i \text{ and } P_{i-1} \text{ and } \ldots \text{ and } P_0 \text{ and } C_0)$$

We can further speed up the addition by providing a carry-lookahead over groups in addition to the internal lookahead within the group. We define a *group-generated* carry *E* and a *group-propagated* carry *R*, for a group of size 4 as follows: *E* = '1' if a carry-out (of the group) is generated internally and *R* = '1' if a carry-in (to the group) is propagated internally to produce a carry-out (of the group). The boolean equations for these carries are:

$$E = G_3 \text{ or } (P_3 \text{ and } G_2) \text{ or } (P_3 \text{ and } P_2 \text{ and } G_1) \text{ or } (P_3 \text{ and } P_2 \text{ and } P_1 \text{ and } G_0)$$
$$R = (P_3 \text{ and } P_2 \text{ and } P_1 \text{ and } P_0)$$

The group-generated and group-propagated carries for several groups can now be used to generate group carry-ins in a manner similar to single-bit carry-ins.

The selection of the group size plays an important role in obtaining the best possible implementation for a carry-lookahead adder in a CPLD. Some of the different possible implementations for a 12-bit carry-lookahead adder are shown in *Figure 6*.

The number of passes each of these implementations take and the number of product terms (PTs) and macrocells (MCs) used vary for each scheme (see *Table 2* in the "Comparison of Resource Utilization for Different Schemes in Building a 12-Bit Adder" section). Each scheme has its own advantage over the other. The user needs to judiciously choose between the different schemes based on the application, bit-size, and the CPLD chosen and its architectural constraints. The number of passes taken through the logic is a direct representation of the total time taken for producing final results. Each extra pass results in a time penalty. The rule to follow is, "The smaller the number of passes through the logic array, the faster your application runs." The implementation of a 12-bit carry-lookahead adder with different group-sizes is presented next.

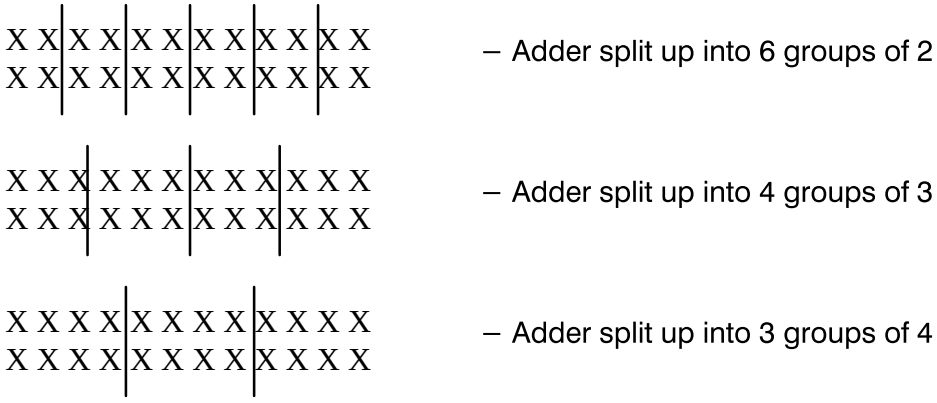

Figure 6. Some Possible Implementations for 12-Bit Carry-Lookahead Adder

FC2ADD12: 12-Bit Full Carry-Lookahead Adder Using a Group-Size of 2 Bits

The FLASH370 CPLD can access up to 16 PTs for each macrocell. The functionality of any signal that has more than 16 PTs is sum-split to fit it into multiple MCs. The number of PTs utilized for signals that sum-split is large and is an undesirable option. With the 2-bit group-size implementation we can accommodate the entire functionality of a 32-bit full carry-lookahead adder without any of the signals getting sum-split. The scheme takes a maximum of three passes through the logic array for all adder sizes up to 32 bits to generate outputs. The various values of Es and Rs, SUM1, SUM0, and C2 are generated in the first pass. All the other intermediate carries are generated in the second pass and the various SUM results are generated in the third pass. A key point to note is that the value of CO is produced in the second pass, even though the various SUM outputs are generated in the third pass only. This makes the component cascadable and modular. Refer to *Table 2* for details on the resource utilization of different 12-bit adder implementations. The FC2ADD12 is built using the ADD2WC and ADD2NC as basic building blocks. The block diagram of a FC2ADD12 is shown in *Figure 7*. The VHDL code for the design is also presented.

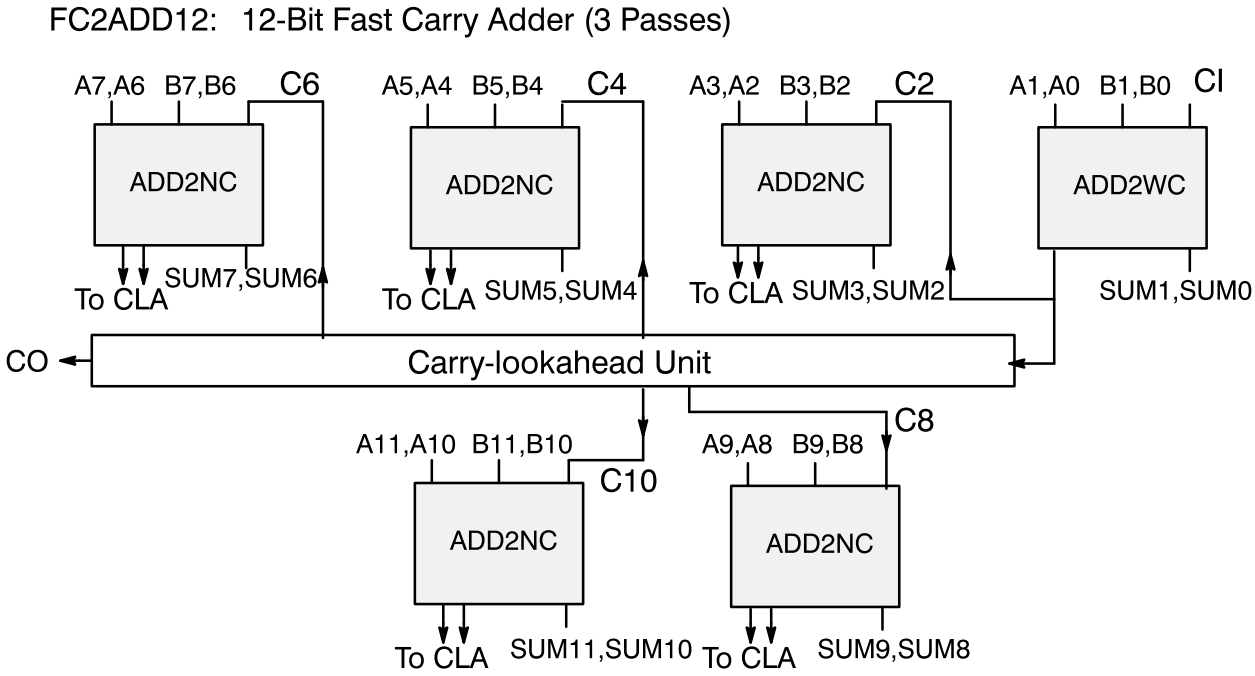

Figure 7. 12-Bit Full Carry-Lookahead Adder Using ADD2WC and ADD2NC

```
--A 12-bit Full carry-lookahead adder built using the ADD2WC and ADD2NC
--elements

USE WORK.RTLPKG.ALL;
USE WORK.ADD2WC.ALL;
USE WORK.ADD2NC.ALL;

ENTITY fc2add12 IS
  PORT(CI : IN BIT;
       A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0: IN BIT;
       B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0: IN BIT;
       SUM11,SUM10,SUM9,SUM8,SUM7,SUM6,SUM5,SUM4,
```

```
      SUM3,SUM2,SUM1,SUM0 : OUT BIT;
      CO: OUT BIT);
END fc2add12;

ARCHITECTURE archfc2add12 OF fc2add12 IS

  SIGNAL C2, C4, C6, C8, C10 : BIT;
  SIGNAL E1,E2,E3,E4,E5  : BIT;
  SIGNAL R1,R2,R3,R4,R5  : BIT;
  attribute synthesis_off of E1,E2,E3,E4,E5 : signal is true;
  attribute synthesis_off of R1,R2,R3,R4,R5 : signal is true;
  attribute synthesis_off of C2, C4, C6, C8, C10 : signal is true;

BEGIN

  i1: add2wc PORT MAP(CI,A1,A0,B1,B0,SUM1,SUM0,C2);
  i2: add2nc PORT MAP(C2,A3,A2,B3,B2,SUM3,SUM2);
  i3: add2nc PORT MAP(C4,A5,A4,B5,B4,SUM5,SUM4);
  i4: add2nc PORT MAP(C6,A7,A6,B7,B6,SUM7,SUM6);
  i5: add2nc PORT MAP(C8,A9,A8,B9,B8,SUM9,SUM8);
  i6: add2nc PORT MAP(C10,A11,A10,B11,B10,SUM11,SUM10);

  E1 <= (A3 AND B3) OR ((A3 OR B3) AND (A2 AND B2));
  R1 <= (A3 OR B3) AND (A2 OR B2);

  C4 <= E1 OR (C2 AND R1);

  E2 <= (A5 AND B5) OR ((A5 OR B5) AND (A4 AND B4));
  R2 <= (A5 OR B5) AND (A4 OR B4);

  C6 <= E2 OR ((E1 OR (C2 AND R1)) AND R2);

  E3 <= (A7 AND B7) OR ((A7 OR B7) AND (A6 AND B6));
  R3 <= (A7 OR B7) AND (A6 OR B6);

  C8 <= E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2)) AND R3);

  E4 <= (A9 AND B9) OR ((A9 OR B9) AND (A8 AND B8));
  R4 <= (A9 OR B9) AND (A8 OR B8);

  C10 <= E4 OR ((E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2)) AND R3)) AND
        R4);

  E5 <= (A11 AND B11) OR ((A11 OR B11) AND (A10 AND B10));
  R5 <= (A11 OR B11) AND (A10 OR B10);

  CO <= E5 OR ((E4 OR ((E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2)) AND
       R3)) AND R4)) AND R5);

END archfc2add12;
```

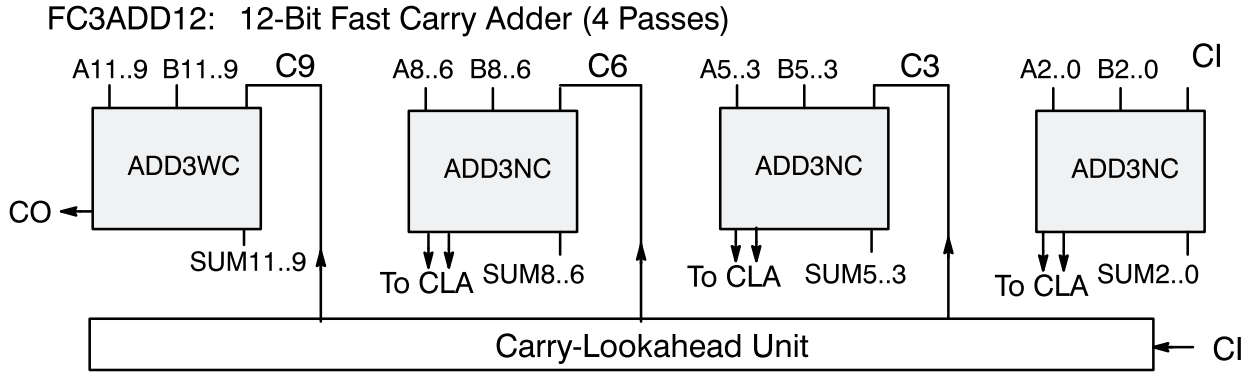

Figure 8. 12-Bit Full Carry-Lookahead Adder using ADD3WC and ADD3NC

FC3ADD12: 12-Bit Full Carry-Lookahead Adder using a Group-Size of 3 Bits

This is very similar to the FC2ADD12, differing in the group-size of the adder used as the basic building block. The basic building blocks in this scheme are the ADD3WC and the ADD3NC components. The VHDL code attached and the block diagram in *Figure 8* illustrate the design. This scheme takes four passes through the logic array to yield all the results. The Es and the Rs are generated in the first pass. The intermediate carries C3, C6, and C9 are generated in the second pass. The carries internal to the group are generated in the third pass and the final SUM outputs in the fourth pass. As a different approach, the CO is generated by the MSB ADD3WC as opposed to the Carry-lookahead unit. This results in CO being generated in the third pass as opposed to the second pass. The VHDL code clearly indicates the manner in which the model is built.

For some bit-sizes, given that the 3-bit group-size is odd-numbered, the designer will have to choose a non-modular structure in building the adder. For example, a 32-bit adder cannot be built using just ADD3NCs and can be built using 10 ADD3NCs and one ADD2NC. The designer needs to choose the final implementation based on the constraints of the application.

```
--12-Bit Fast carry-Lookahead adder with 3-bit groups

USE WORK.ADD3WC.ALL;
USE WORK.ADD3NC.ALL;

ENTITY fc3add12 IS
  PORT (
    A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0 : IN BIT;
    B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0 : IN BIT;
    CI : IN BIT;
    CO : OUT BIT;
    SUM11,SUM10,SUM9,SUM8,SUM7,SUM6,SUM5,SUM4,SUM3,
    SUM2,SUM1,SUM0 : OUT BIT);
END fc3add12;

ARCHITECTURE fc3add12arch OF fc3add12 IS

SIGNAL  E1,E2,E3 : BIT;
SIGNAL  R1,R2,R3  : BIT;
SIGNAL  C3,C6,C9 : BIT;
```

```
attribute synthesis_off of C3,C6,C9 : signal is true;
attribute synthesis_off of E1,E2,E3 : signal is true;
attribute synthesis_off of R1,R2,R3 : signal is true;

BEGIN

i1: add3nc PORT MAP(CI,A2,A1,A0,B2,B1,B0,SUM2,SUM1,SUM0);
i2: add3nc PORT MAP(C3,A5,A4,A3,B5,B4,B3,SUM5,SUM4,SUM3);

i3: add3nc PORT MAP(C6,A8,A7,A6,B8,B7,B6,SUM8,SUM7,SUM6);
i4: add3wc PORT MAP(C9,A11,A10,A9,B11,B10,B9,SUM11,SUM10,SUM9,CO);

E1 <=  (A2 AND B2)
  OR ((A1 AND B1) AND (A2 OR B2))
  OR ((A0 AND B0) AND (A1 OR B1) AND (A2 OR B2));

R1 <=  (A2 OR B2) AND (A1 OR B1) AND (A0 AND B0);

C3 <= E1 OR (R1 AND CI);

E2 <=  (A5 AND B5)
  OR ((A4 AND B4) AND (A5 OR B5))
  OR ((A3 AND B3) AND (A4 OR B4) AND (A5 OR B5));

R2 <=  (A5 OR B5) AND (A4 OR B4) AND (A3 AND B3);

C6 <= E2 OR (E1 AND R2) OR (R2 AND R1 AND CI);

E3 <=  (A8 AND B8)
  OR ((A7 AND B7) AND (A8 OR B8))
  OR ((A6 AND B6) AND (A7 OR B7) AND (A8 OR B8));

R3 <=  (A8 OR B8) AND (A7 OR B7) AND (A6 AND B6);

C9 <= E3 OR (E2 AND R3) OR (E1 AND R3 AND R2) OR (R3 AND R2 AND R1 AND CI);

END fc3add12arch;
```

FC4ADD12: 12-Bit Full Carry-Lookahead Adder using a Group-Size of 4 Bits

This is very similar to the FC2ADD12 and, again, differs in the group-size of the adder used as the basic building block. The basic building block in this scheme is the ADD4NC component. The ADD4NC component is built using a combination of ADD2WC and ADD2NC in the same order. This component is replicated to create the adder of the desired size. In the very last stage, two ADD2WCs are used instead of an ADD2WC and an ADD2NC.

The VHDL code attached and the block diagram in *Figure 9* illustrate the design's functionality. This scheme takes four passes through the logic array to yield results. The various Es and Rs are generated in the first pass, the values of C4 and C8 in the second pass, the outputs from all the ADD2WCs in the third pass, and the outputs from ADD2NC in the fourth pass. Note that the value of CO is generated in the second pass. This scheme uses fewer MCs and more PTs than the previously mentioned schemes. The resource utilization of this model is shown in *Table 2*.

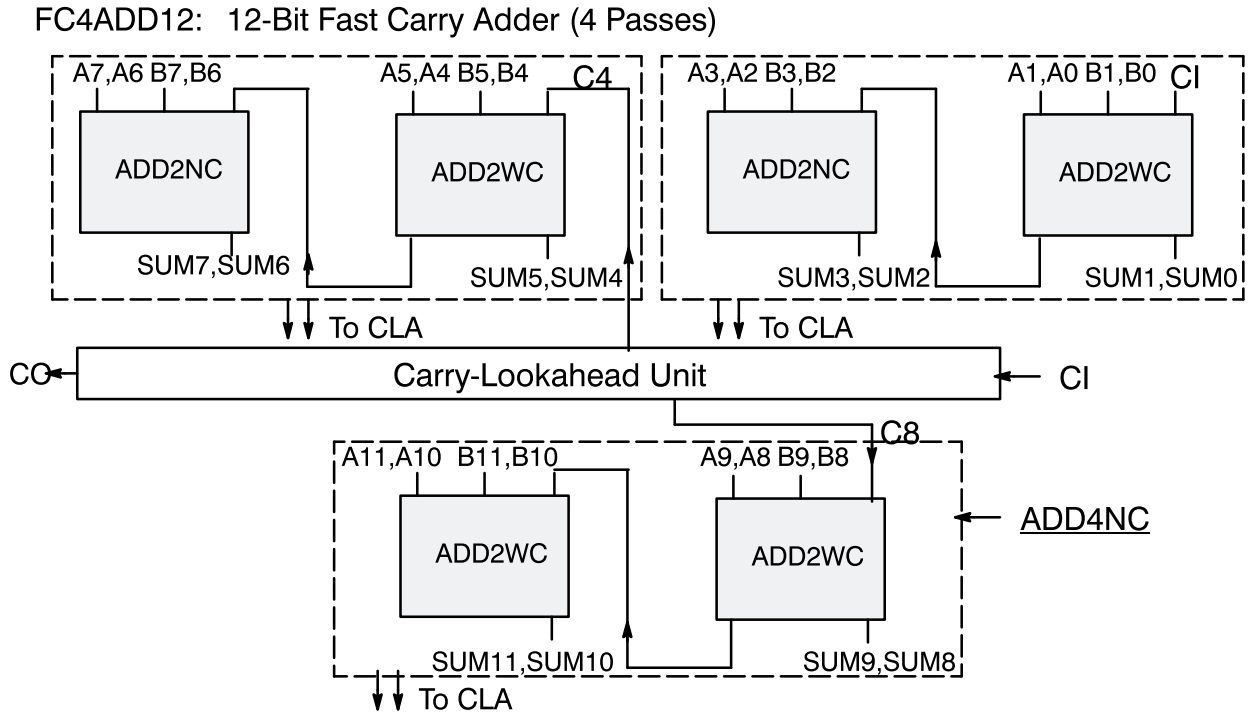

Figure 9. 12-Bit Full Carry-Lookahead Adder using ADD4NC

```
--A 12-bit Full carry-lookahead adder built using the ADD2WC and ADD2NC
--elements. The  ADD2WC and ADD2NC elements are part of the ADD4NC in the
--same order

USE WORK.RTLPKG.ALL;
USE WORK.ADD2WC.ALL;
USE WORK.ADD2NC.ALL;

ENTITY fc4add12 IS
  PORT (
    A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0 : IN BIT;
    B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0 : IN BIT;
    CI : IN BIT;
    CO : OUT BIT;
    SUM11,SUM10,SUM9,SUM8,SUM7,SUM6,SUM5,SUM4,SUM3,
    SUM2,SUM1,SUM0 : OUT BIT);
END fc4add12;

ARCHITECTURE fc4add12arch OF fc4add12 IS

 SIGNAL E1,E2 : BIT;
 SIGNAL R1,R2  : BIT;
 SIGNAL C2,C4,C6,C8,C10 : BIT;

attribute synthesis_off of C2,C4,C6,C8,C10 : signal is true;
attribute synthesis_off of E1,E2 : signal is true;
attribute synthesis_off of R1,R2 : signal is true;
```

```
BEGIN

i1: add2wc PORT MAP(CI,A1,A0,B1,B0,SUM1,SUM0,C2);
i2: add2nc PORT MAP(C2,A3,A2,B3,B2,SUM3,SUM2);

i3: add2wc PORT MAP(C4,A5,A4,B5,B4,SUM5,SUM4,C6);
i4: add2nc PORT MAP(C6,A7,A6,B7,B6,SUM7,SUM6);

i5: add2wc PORT MAP(C8,A9,A8,B9,B8,SUM9,SUM8,C10);
i6: add2wc PORT MAP(C10,A11,A10,B11,B10,SUM11,SUM10,CO);

E1 <=  (A3 AND B3)
  OR ((A2 AND B2) AND (A3 OR B3))
  OR ((A1 AND B1) AND (A2 OR B2) AND (A3 OR B3))
  OR ((A0 AND B0) AND (A1 OR B1) AND (A2 OR B2) AND (A3 OR B3));

R1 <= (A3 OR B3) AND (A2 OR B2) AND (A1 OR B1) AND (A0 AND B0);

C4 <= E1 OR (R1 AND CI);

E2 <=  (A7 AND B7)
  OR ((A6 AND B6) AND (A7 OR B7))
  OR ((A5 AND B5) AND (A6 OR B6) AND (A7 OR B7))
  OR ((A4 AND B4) AND (A5 OR B5) AND (A6 OR B6) AND (A7 OR B7));

R2 <= (A7 OR B7) AND (A6 OR B6) AND (A5 OR B5) AND (A4 AND B4);

C8 <= E2 OR (E1 AND R2) OR (R2 AND R1 AND CI);

END fc4add12arch;
```

Subtracters

Subtracters are just a modified form of adders. The discussion presented for the adders can be easily extended to the subtracters. For any given sized adder or subtracter, the resource utilization is exactly the same for both in all respects.

SUB: 1-Bit Full Subtracter

The basic component used in subtracting two operands is called a *Full subtracter*. The full subtracter element will be referred to as the 'SUB' component.

The block diagram and functionality of SUB is shown in *Figure 10*. A (minuend) and B (subtrahend) are the two operands to be subtracted and Bin is the Borrow-in to the component. DIF and Bout are the Difference and Borrow-out from the component.

The VHDL code describing the functionality of the SUB component is shown here. This design takes one pass through the Product Term Matrix logic array to fit into a FLASH370 CPLD. The SUB component instantiated in the VHDL code has the exact same functionality shown in *Figure 10*.

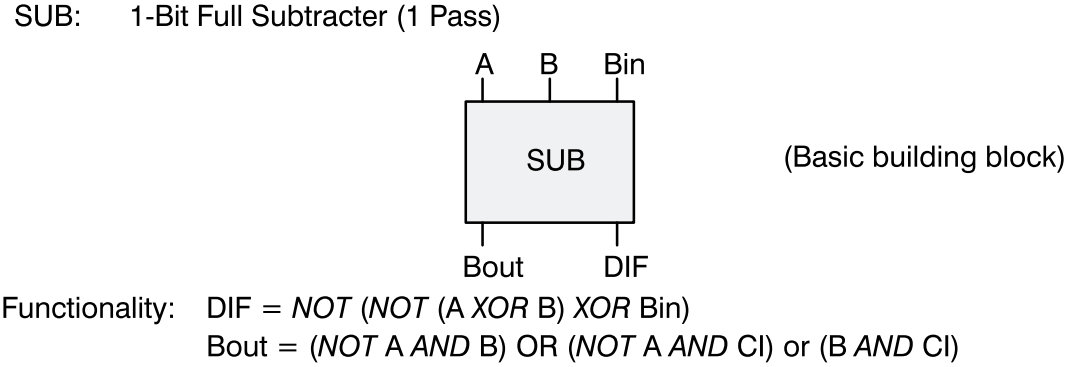

Figure 10. Block Diagram and Functionality of a Full Subtracter

```
-- This VHDL code invokes the
MATHPKG element SUB

USE WORK.CYPRESS.ALL;
USE WORK.MATHPKG.ALL;

ENTITY sub IS
 PORT (Bin: IN BIT;
   A, B: IN BIT;
   DIF: OUT BIT;
    Bout: OUT BIT);
END sub;

ARCHITECTURE archsub OF sub IS

BEGIN

 i1: sub PORT
MAP(Bin,A,B,DIF,Bout);

END archsub;
```

SUB2WB: A 2-Bit Subtracter with a Borrow-Out

The structure of a 2-bit group subtracter (SUB2WB) is very similar to that of the ADD2WC and is shown here. This component can be used as a building block to build larger sized subtracters, exactly like ADD2WC was used to build larger sized adders. The block diagram of the SUB2WB is shown in *Figure 11*. The corresponding VHDL code used to describe the functionality of the SUB2WB is also attached. As in the case of ADD2WC, the functionality for SUB2WB is realized in one pass through the logic array.

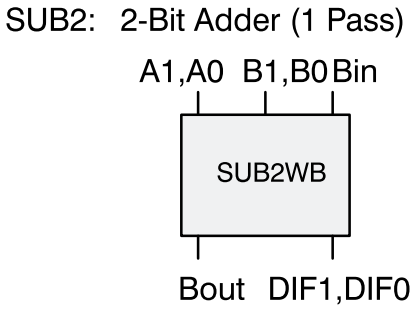

Figure 11. Block Diagram of a 2-Bit Subtracter with a Borrow-Out

```
USE WORK.RTLPKG.ALL;

PACKAGE sub2wb_pkg IS
  COMPONENT sub2wb PORT(
    Bin : IN BIT;
      A1,A0: IN BIT;
      B1,B0: IN BIT;
      DIF1,DIF0 : OUT BIT;
      Bout: OUT BIT);
  END COMPONENT;
END sub2wb_pkg;
```

```
ENTITY sub2wb IS
  PORT(Bin : IN BIT;
    A1,A0: IN BIT;
    B1,B0: IN BIT;
    DIF1,DIF0 : OUT BIT;
    Bout: OUT BIT);
END sub2wb;

ARCHITECTURE archsub2wb OF sub2wb IS

BEGIN

DIF0 <= NOT (NOT (A0 XOR B0) XOR Bin);
DIF1 <= NOT (NOT (A1 XOR B1) XOR ((NOT A0 AND B0) OR (NOT A0 AND Bin) OR
           (B0 AND Bin)));

Bout   <= (NOT A0 AND B0 AND B1)
  OR (NOT A0 AND B0 AND NOT A1)
  OR (BI AND B0 AND B1)
  OR (BI AND B0 AND NOT A1)
  OR (BI AND NOT A0 AND B1)
  OR (BI AND NOT A0 AND NOT A1)
  OR (NOT A1 AND B1);

END archsub2wb;
```

FB2SUB12 : 12-Bit Full Borrow-Lookahead Subtracter using 2-Bit Subtracters

It was mentioned before that we can build equivalent subtracter models for all the adder models discussed earlier. The functionality and the implementation of an FB2SUB12 (subtracter equivalent of an FC2ADD12) is shown here as an example. The implementation of all the possible subtracter elements is not discussed in this application note, since the concept involved in building them is identical to that of the adders.

The block diagram of the FB2SUB12 is very similar to that of the adder element FC2ADD12 and is shown in *Figure 12*. The FB2SUB12 is built using the basic elements SUB2WB and SUB2NC (2-bit subtracter with no borrow-out). This takes three passes through the logic array. The values of the various Es and Rs are generated in the first pass, the intermediate carries (borrows) in the second pass, and the various DIFs in the third pass. Note that the value of BO is generated in the second pass. The VHDL code for FB2SUB12 is also attached.

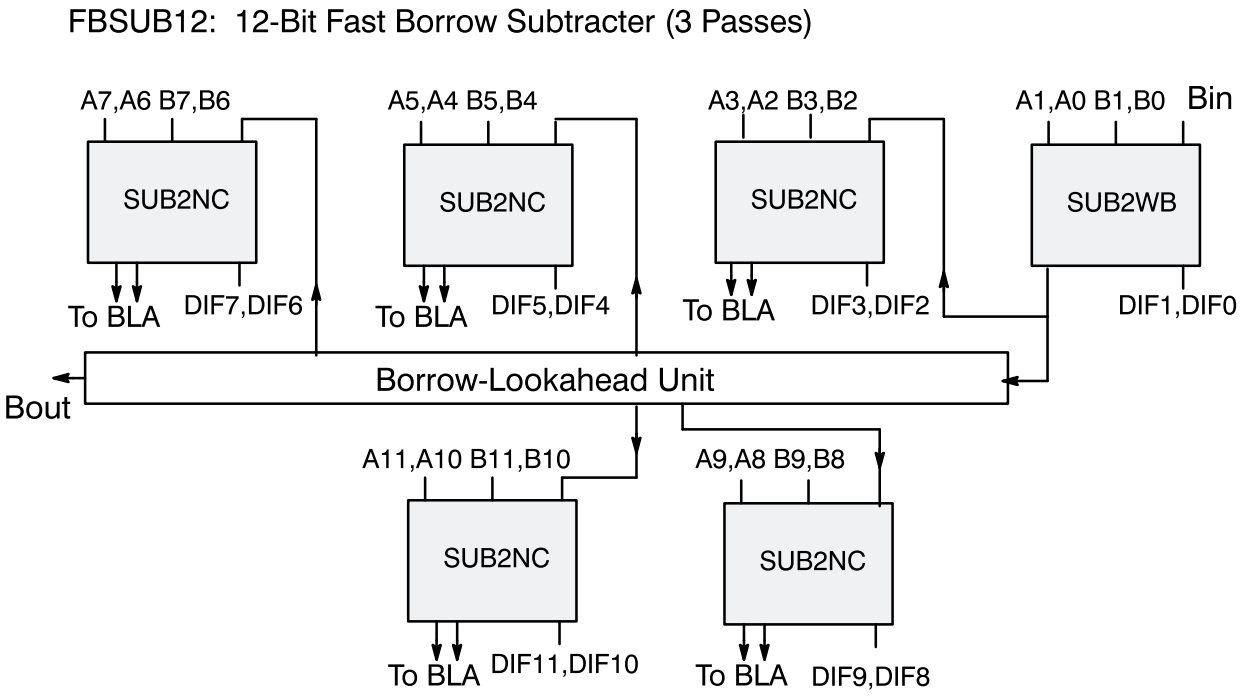

Figure 12. 12-Bit Fast Borrow Subtracter Built using SUB2WB and SUB2NC

```
--A 12-bit Full borrow-lookahead subtracter built using the SUB2WC and
--SUB2NC elements

USE WORK.RTLPKG.ALL;
USE WORK.SUB2WB.ALL;
USE WORK.SUB2NC.ALL;

ENTITY fb2sub12 IS
  PORT(Bin : IN BIT;
       A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0: IN BIT;
       B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0: IN BIT;
       DIF11,DIF10,DIF9,DIF8,DIF7,DIF6,DIF5,DIF4,
       DIF3,DIF2,DIF1,DIF0 : OUT BIT;
    Bout: OUT BIT);
END fb2sub12;

ARCHITECTURE archfb2sb12 OF fb2sub12 IS

  SIGNAL C2, C4, C6, C8, C10 : BIT;
  SIGNAL E1,E2,E3,E4,E5  : BIT;
  SIGNAL R1,R2,R3,R4,R5  : BIT;

--The internal carries are referred to as C's to distinguish between
--borrow-out's and the operands
  attribute synthesis_off of E1,E2,E3,E4,E5 : signal is true;
  attribute synthesis_off of R1,R2,R3,R4,R5 : signal is true;
```

```
  attribute synthesis_off of C2, C4, C6, C8, C10 : signal is true;

BEGIN

  i1: sub2wb PORT MAP(Bin,A1,A0,B1,B0,DIF1,DIF0,C2);
  i2: sub2nc PORT MAP(C2,A3,A2,B3,B2,DIF3,DIF2);
  i3: sub2nc PORT MAP(C4,A5,A4,B5,B4,DIF5,DIF4);
  i4: sub2nc PORT MAP(C6,A7,A6,B7,B6,DIF7,DIF6);
  i5: sub2nc PORT MAP(C8,A9,A8,B9,B8,DIF9,DIF8);
  i6: sub2nc PORT MAP(C10,A11,A10,B11,B10,DIF11,DIF10);

    E1 <= (NOT A3 AND B3) OR ((NOT A3 OR B3) AND (NOT A2 AND B2));
    R1 <= (NOT A3 OR B3) AND (NOT A2 OR B2);

    C4 <= E1 OR (C2 AND R1);

    E2 <= (NOT A5 AND B5) OR ((NOT A5 OR B5) AND (NOT A4 AND B4));
    R2 <= (NOT A5 OR B5) AND (NOT A4 OR B4);

    C6 <= E2 OR ((E1 OR (C2 AND R1)) AND R2);

    E3 <= (NOT A7 AND B7) OR ((NOT A7 OR B7) AND (NOT A6 AND B6));
    R3 <= (NOT A7 OR B7) AND (NOT A6 OR B6);

    C8 <= E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2)) AND R3);

    E4 <= (NOT A9 AND B9) OR ((NOT A9 OR B9) AND (NOT A8 AND B8));
    R4 <= (NOT A9 OR B9) AND (NOT A8 OR B8);

    C10 <= E4 OR ((E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2)) AND R3))
              AND R4);

    E5 <= (NOT A11 AND B11) OR ((NOT A11 OR B11) AND (NOT A10 AND B10));
    R5 <= (NOT A11 OR B11) AND (NOT A10 OR B10);

    Bouy <= E5 OR ((E4 OR ((E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2))
            AND R3)) AND R4)) AND R5);

END archfb2sub12;
```

Comparison of Resource Utilization for Different Schemes in Building a 12-Bit Adder

A comparison chart showing the resource utilization for the different models that can be used in building a 12-bit adder is shown in *Table 2*. This table summarizes some of the key issues that have been presented in the discussion so far. Some comparisons and comments from the charts and are listed here:

Ripple Carry Adders

1. For a given group-size, the number of passes taken to yield results is *dependent* on the size of the adder being built.

Table 2. Comparison of Different 12-Bit Adder Schemes

Resource	R1ADD12	R2ADD12	R3ADD12	FC2ADD12	FC3ADD12	FC4ADD12
PTs used	84	138	165	148	153	169
MCs used	24	18	16	28	26	22
# of passes	12	6	5	3	4	4

2. As the group-size increases, the number of passes taken through the logic array is *(n/k) −1 + # of passes for final stage*, where *n* is the size of the adder and *k* is the group size. For example, a R2ADD12 takes *(12/2) −1 + 1* = 6 passes to yield the desired result.

3. In the R3ADD12 (ripple carry adder built using 3-bit groups) scheme, the value of the MSB sum bit within a 3-bit group is produced only in the second pass through the array. This, however, does not affect the 12-bit adder yielding results in 5 passes *(12/3) −1 + 2 = 5)* as expected. This is possible because the carry-out from the 3-bit group is produced in the first pass. The implementation of the ADD3WC was discussed in detail earlier. This solution is a very desirable solution for most applications that use small sized adders.

4. The R1ADD12 uses fewer PTs and more MCs among the different versions of ripple-carry adders. The opposite is the case for the R3ADD12. The R2ADD12 provides an intermediate solution between the two extremes.

5. The macrocell count in R1ADD12 can be brought down from 24 to 18, if the attribute 'synthesis_off' is used on the even-numbered carries only. The number of passes is also brought down from 12 to 6. This, however, pushes the product term count from 84 to 138. In either case, none of the equations sum-split. This is, in fact, R2ADD12. The designer can choose the implementation that best chooses the application.

6. The R4ADD12 (ripple carry adder built using 4-bit groups) is not a viable solution, since the carry-out from one of the 4-bit groups would take two passes to be generated. This results in a implementation that takes six passes to yield results as opposed to the expected three passes. This solution is inefficient and is not considered.

Carry-Lookahead Adders

1. For a given group-size, the number of passes taken to yield results is *largely independent* of the size of the adder being built. This is the biggest advantage with carry-lookahead adders.

2. All the *group generate*s (*E*s) and *group propagate*s (*R*s) are generated in the first pass and the carry-ins to all groups in the second pass through the logic array. The Sum outputs are generated in the third or the fourth pass, depending on the group-size being used.

3. The FC2ADD12 takes three passes to complete, and four passes for the FC3ADD12 and FC4ADD12. The number of passes *remains the same* up to 32-bit versions of the adder.

4. Similar to the ripple carry adders, the FC2ADD12 uses fewer PTs and more MCs among the different versions of carry-lookahead adders. The opposite is the case for the FC4ADD12. The FC3ADD12 provides an intermediate solution between the two extremes.

5. The FC5ADD12 (carry-lookahead adder built using 5-bit groups) is not a viable solution, since the extra number of PTs and number of passes (5) taken through the logic array do not justify its usage. The design is also not modular and difficult to deal with. A designer can, however, extend the discussion presented to build his own FC5ADD12 model if the application demands it. This, however, would be an extreme case and is not presented.

Summary

Comparing ripple carry and carry-lookahead adders, it is evident that ripple carry adders are area efficient but have poor speed performance. The carry-lookahead adders on the other hand are faster but utilize more resources. Given the different choices, the user needs to make a careful selection of the scheme best suited for his application.

Large-Sized Adders/Subtracters

As PLDs have grown in size and speed over the past few years, a lot of designers have been pushing towards larger sized adders and subtracters. A lot of focus and time has been dedicated towards building efficient smaller sized elements and it was left to the designer's discretion to build and implement the algorithm. Cypress believes in providing designers with the best possible implementation for their adders/subtracters and relieve them of the problems they would normally face. This provides the customer with the opportunity to get the best implementation for their application without spending a lot of time.

Table 1 talks about the resource utilization for 24-bit and 32-bit adders using 2-bit, 3-bit, and 4-bit group-sizes with carry/borrow-lookahead principle. In the previous sections, different implementation strategies and the VHDL codes for a 12-bit full-carry-lookahead adder were shown as an example. The VHDL codes for most variations of the 24- and 32-bit implementations are not presented here due to space constraints. The codes are provided, however, as a part of the tutorial section in the *Warp* VHDL compiler. *Figure 12* illustrates three schemes used in implementing a 24-bit adder. The VHDL code for a 24-bit carry-lookahead adder with a 4-bit group size is shown here as an example. The code for other models is very similar and can be easily extrapolated.

Table 1. Comparison of Different 24-Bit and 32-Bit Adder Schemes

Resource	FC2ADD24	FC3ADD24	FC4ADD24	FC2ADD32	FC3ADD32	FC4ADD32
PTs used	272	314	359	393	427	488
MCs used	58	54	46	78	73	62
# of passes	3	4	4	3	4	4

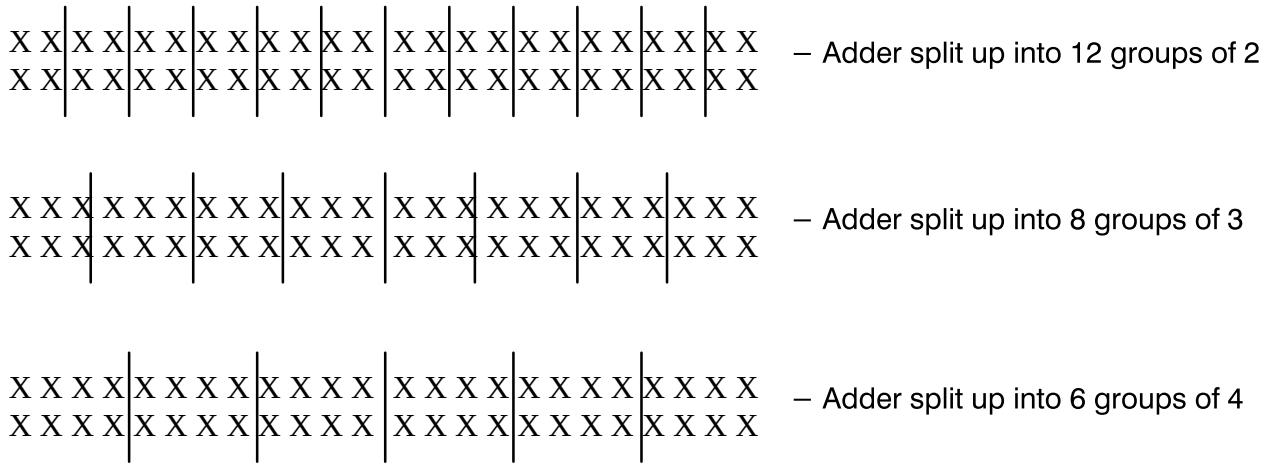

Figure 13. Three Different Carry-Lookahead Schemes to Implement a 24-Bit Adder

```
--24-bit Fast Carry lookahead adder with 4-bit groups

USE work.add2wc_pkg.all;
USE work.add2nc_pkg.all;

ENTITY fc4add24 IS
PORT (

A23,A22,A21,A20,A19,A18,A17,A16,A15,A14,A13,A12,
A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0 : IN BIT;
B23,B22,B21,B20,B19,B18,B17,B16,B15,B14,B13,B12,
B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0 : IN BIT;
CI : IN BIT;
CO : OUT BIT;
SUM23,SUM22,SUM21,SUM20,SUM19,SUM18,SUM17,SUM16,SUM15,SUM14,SUM13,SUM12,SUM
11,SUM10,SUM9,SUM8,SUM7,SUM6,SUM5,SUM4,SUM3,SUM2,SUM1,SUM0: OUT BIT);

END fc4add24;

ARCHITECTURE fc4add24arch OF fc4add24 IS

SIGNAL  E1,E2,E3,E4,E5 : BIT;
SIGNAL  R1,R2,R3,R4,R5  : BIT;
SIGNAL  C2,C4,C6,C8,C10,C12,C14,C16,C18,C20,C22 : BIT;

attribute synthesis_off of C2,C4,C6,C8,C10,C12,C14,C16,C18,C20,C22 : signal
is true;
attribute synthesis_off of E1,E2,E3,E4,E5 : signal is true;
attribute synthesis_off of R1,R2,R3,R4,R5 : signal is true;

BEGIN

i1: add2wc PORT MAP (CI,A1,A0,B1,B0,SUM1,SUM0,C2);
i2: add2nc PORT MAP (C2,A3,A2,B3,B2,SUM3,SUM2);

i3: add2wc PORT MAP (C4,A5,A4,B5,B4,SUM5,SUM4,C6);
i4: add2nc PORT MAP (C6,A7,A6,B7,B6,SUM7,SUM6);

i5: add2wc PORT MAP (C8,A9,A8,B9,B8,SUM9,SUM8,C10);
i6: add2nc PORT MAP (C10,A11,A10,B11,B10,SUM11,SUM10);

i7: add2wc PORT MAP (C12,A13,A12,B13,B12,SUM13,SUM12,C14);
i8: add2nc PORT MAP (C14,A15,A14,B15,B14,SUM15,SUM14);

i9: add2wc PORT MAP (C16,A17,A16,B17,B16,SUM17,SUM16,C18);
i10: add2nc PORT MAP (C18,A19,A18,B19,B18,SUM19,SUM18);

i11: add2wc PORT MAP (C20,A21,A20,B21,B20,SUM21,SUM20,C22);
i12: add2wc PORT MAP (C22,A23,A22,B23,B22,SUM23,SUM22,Co);
```

```
E1 <=  (A3 AND B3)
  OR ((A2 AND B2) AND (A3 OR B3))
  OR ((A1 AND B1) AND (A2 OR B2) AND (A3 OR B3))
  OR ((A0 AND B0) AND (A1 OR B1) AND (A2 OR B2) AND (A3 OR B3));

R1 <= (A3 OR B3) AND (A2 OR B2) AND (A1 OR B1) AND (A0 AND B0);

C4 <= E1 OR (R1 AND CI);

E2 <=  (A7 AND B7)
  OR ((A6 AND B6) AND (A7 OR B7))
  OR ((A5 AND B5) AND (A6 OR B6) AND (A7 OR B7))
  OR ((A4 AND B4) AND (A5 OR B5) AND (A6 OR B6) AND (A7 OR B7));

R2 <= (A7 OR B7) AND (A6 OR B6) AND (A5 OR B5) AND (A4 AND B4);

C8 <= E2 OR (E1 AND R2) OR (R2 AND R1 AND CI);

E3 <=  (A11 AND B11)
  OR ((A10 AND B10) AND (A11 OR B11))
  OR ((A9 AND B9) AND (A10 OR B10) AND (A11 OR B11))
  OR ((A8 AND B8) AND (A9 OR B9) AND (A10 OR B10) AND (A11 OR B11));

R3 <= (A11 OR B11) AND (A10 OR B10) AND (A9 OR B9) AND (A8 AND B8);

C12 <= E3 OR (E2 AND R3) OR (E1 AND R3 AND R2)  OR (R3 AND R2 AND R1 AND
CI);

E4 <=  (A15 AND B15)
  OR ((A14 AND B14) AND (A15 OR B15))
  OR ((A13 AND B13) AND (A14 OR B14) AND (A15 OR B15))
  OR ((A12 AND B12) AND (A13 OR B13) AND (A14 OR B14) AND (A15 OR B15));

R4 <= (A15 OR B15) AND (A14 OR B14) AND (A13 OR B13) AND (A12 AND B12);

C16 <= E4 OR (E3 AND R4) OR (E2 AND R4 AND R3) OR (E1 AND R4 AND R3 AND R2)
  OR (R3 AND R2 AND R1 AND CI);

E5 <=  (A19 AND B19)
  OR ((A18 AND B18) AND (A19 OR B19))
  OR ((A17 AND B17) AND (A18 OR B18) AND (A19 OR B19))
  OR ((A16 AND B16) AND (A17 OR B17) AND (A18 OR B18) AND (A19 OR B19));
R5 <= (A19 OR B19) AND (A18 OR B18) AND (A17 OR B17) AND (A16 AND B16);

C20 <= E5 OR (E4 AND R5) OR (E3 AND R5 AND R4) OR (E2  AND R5  AND R4 AND
  R3) OR (E1 AND R5 AND R4 AND R3 AND R2) OR (R5 AND R4 AND R3 AND R2 AND
  R1 AND CI);

END fc4add24arch;
```

Equality Comparators

Equality comparators are used often to compare the value of two operands. Equality comparators are built using the Exclusive-OR gate as the building block. A bit-wise comparison of the two data streams is done using XOR gates and each of the individual results are OR-ed together to obtain the final result.

EQCOMP4: 4-Bit Equality Comparator

The EQCOMP4 is a 4-bit equality compare element. The model can be described as:

```
EQ = NOT ((A3 XOR B3)
   OR (A2 XOR B2)
   OR (A1 XOR B1)
   OR (A0 XOR B0))
```

This implementation takes 8 PTs. *Figure 14* shows the block diagram for EQCOMP4. NEQCOMP4 is the 4-bit non-equality comparator. The EQCOMP4 is implemented as an inverted version of the NEQCOMP4. The NEQCOMP4 element takes 8 PTs and the EQCOMP4 takes 16 PTs. The FLASH370 CPLD has a polarity control in the macrocell and can create the EQCOMP4 element using the NEQCOMP4 element, resulting in a implementation with a reduced product term count.

The equality comparator for all bit sizes greater than 8 takes more than 16 PTs to produce the result and takes two passes, since the FLASH370 CPLD architecture takes in a maximum of 16 PTs into one macrocell.

EQCOMP24: 24-Bit Equality Comparator

The EQCOMP24 uses three EQCOMP8s in parallel and combines the results of the three components to produce the result. This takes two passes through the logic array, 4 MCs, and 49 PTs. The block diagram of this model is shown in *Figure 15*.

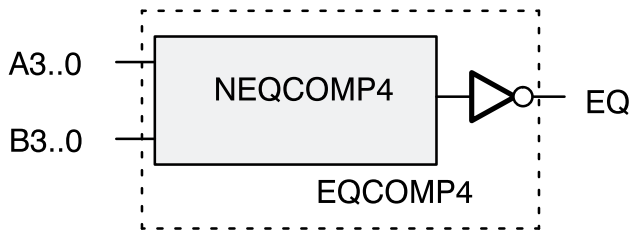

Figure 14. Block Diagram of a 4-Bit Equality Compare

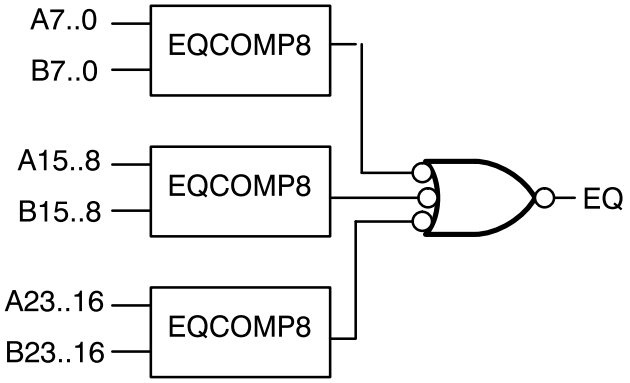

Figure 15. Block Diagram of a 24-Bit Equality Compare

Magnitude Comparators

Magnitude comparators are also widely used in the industry in comparing values of two operands. The magnitude comparators provide information if a signal is greater than (>) , or less than (<) another signal of the same length.

MAGCOMP8: 8-Bit Magnitude Comparator

This is the generic implementation of a magnitude comparator and does a bit-wise comparison, similar to that of the equality comparison. However, in the case of a magnitude comparator the results of a bit-wise comparison are to be retained and passed onto the succeeding set of bits. This passage of information continues and tends to increase the resource utilization of the design exponentially.

The VHDL implementation of an 8-bit magnitude comparator is shown here. The design takes 255 PTs and fits in two passes through the logic array. The block diagram of MAGCOMP8 is shown in *Figure 16*.

Figure 16. Block Diagram of an 8-Bit Magnitude Compare

```
-- Flattened version of the Magni-
tude comparator

USE work.int_math.all;

ENTITY magcomp IS
PORT (
A,B : IN BIT_VECTOR(7 DOWNTO 0);
MAG : OUT BIT);

END magcomp;

ARCHITECTURE magarch OF magcomp IS

BEGIN

MAG <= '1' WHEN (A < B) ELSE '0';

END magarch;
```

A fully flattened implementation of a magnitude comparator would take $(2^n - 1)$ PTs to implement. It is, however, not recommended to use the fully-flattened version of the magnitude comparator for any bit-size greater than 4 bits. This is to ensure that there is no sum-splitting involved in the equations. There are other means to achieve better results and the best scheme is presented next.

FB2MGCMP8: 8-Bit Borrow-Lookahead Magnitude Comparator

The block diagram of a 8-bit magnitude compare is shown in *Figure 17*.

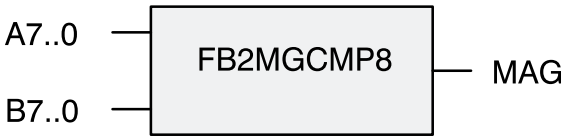

Figure 17. Block Diagram of an 8-Bit Magnitude Compare

This scheme uses a different approach to compare the magnitudes of two binary bit vectors. As an example, the scheme is illustrated for a 8-bit magnitude comparator. The 4 MSB bits of the bit vectors A[7:0] and B[7:0] are called A_M and B_M, respectively. Similarly, the 4 LSB bits are referred to as A_L and B_L respectively. The bit vector A is greater than B if $(A_M > B_M)$ or if $(A_M = B_M)$ and $(A_l > B_L)$.

It is evident from the set of equations in *Figure 18* that the magnitude comparison of two binary bit vectors can be done by evaluating the values of G_M, G_L and P_M. G_M and G_L are the *generate* functions for the MSHalf (most significant half) and the LSHalf (least significant half) for the two bit vectors and P_M is the *propagate* function for the MSHalf. This scheme is a stripped down version of the borrow-lookahead scheme used to build fast subtracters. In this implementation we need to determine the values of the *generate* and *propagate* functions for the bit vectors and need not produce any of the difference results. The borrow-out signal determines the output of the magnitude comparison. If the borrow-out is a '1' then $(A < B)$, else $(A \geqslant B)$.

This scheme allows for a fast and efficient means to do magnitude comparisons. Magnitude Comparators up to 32 bits can be built to produce the result in just 2 passes. The number of PTs used is also substantially less than the 'flattened' implementation of the magnitude comparators.

The discussion presented earlier on group-sizes can also be extended here. The group-size over which the *propagate* and *generate* functions are generated can be varied to be 2, 3 or 4. In *all* cases the design takes 2 passes to produce the desired result. The various values of Es and Rs are generated in the first

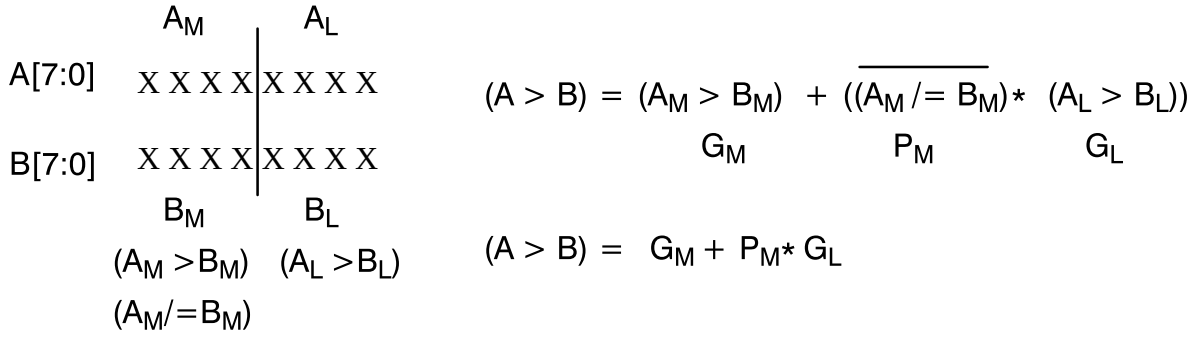

Figure 18. Bit Vector Magnitude Comparison Equations

pass and the value of the borrow-out in the second pass. However, there is a trade-off between the number of PTs and MCs used among the different group-sizes chosen. A comparison between these different implementations is discussed later.

The number of PTs used to implement the P_M (propagate) function can be halved if 'OR' gates are used instead of 'XOR' gates. This was mentioned earlier in the discussion on carry-lookahead. This extension makes the implementation of the borrow-lookahead magnitude comparator fast and efficient.

Comparison of Two Implementations of a 12-Bit Magnitude Compare

Two different implementations of a 12-bit magnitude comparator are shown here. The first implementation is an extension of MAGCOMP4. The second implementation uses the borrow-lookahead scheme and is built using borrow-lookahead over a group-size of 2 bits. This comparison illustrates the advantage of using FB2MGCMP12 over the simple MAGCOMP12.

The block diagram of MAGCOMP12 is shown in *Figure 19*. The flattened version of MAGCOMP12 takes $(2^{12} - 1)$ PTs. This is a large amount of logic and will not fit into any of the FLASH370 CPLDs.

Figure 19. Block Diagram of a 12-Bit Magnitude Compare

The MAGCOMP12 with the synthesis_off attribute on the intermediate signals uses 44 unique PTs, but is very slow and takes 11 passes through the array.

The block diagram of FB2MGCMP12 is shown in *Figure 20*. The VHDL code for this design is also shown here. This design takes just two passes through the array and uses 36 unique PTs. The various values of Es and Rs are generated in the first pass and the value of the borrow-out in the second pass. Each of the Es uses 3 PTs and Rs 2 PTs and the output MAG takes 6 PTs. This is clearly a much better implementation than the MAGCOMP12.

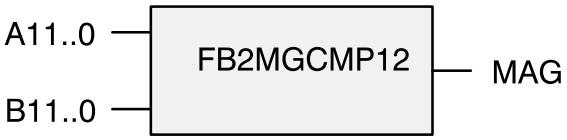

Figure 20. Block Diagram of a 12-Bit Magnitude Compare with Borrow-Lookahead

```
--The borrow-lookahead principle using 2-bit groups was used to build this
--element

USE WORK.RTLPKG.ALL;

ENTITY fb2mgcmp12 IS
  PORT (
    A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0: IN BIT;
    B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0: IN BIT;
      MAG: OUT BIT);
END fb2mgcmp12;

ARCHITECTURE archfb2mgcmp12 OF fb2mgcmp12 IS

  SIGNAL E0,E1,E2,E3,E4,E5  : BIT;
  SIGNAL R0,R1,R2,R3,R4,R5  : BIT;
  SIGNAL BO : BIT;

  attribute synthesis_off of E0,E1,E2,E3,E4,E5 : signal is true;
  attribute synthesis_off of R0,R1,R2,R3,R4,R5 : signal is true;
```

```
BEGIN

    E0 <= (NOT A1 AND B1) OR ((NOT A1 OR B1) AND (NOT A0 AND B0));
    R0 <= (NOT A1 OR B1) AND (NOT A0 OR B0);

    E1 <= (NOT A3 AND B3) OR ((NOT A3 OR B3) AND (NOT A2 AND B2));
    R1 <= (NOT A3 OR B3) AND (NOT A2 OR B2);

    E2 <= (NOT A5 AND B5) OR ((NOT A5 OR B5) and (NOT A4 AND B4));
    R2 <= (NOT A5 OR B5) AND (NOT A4 OR B4);

    E3 <= (NOT A7 AND B7) OR ((NOT A7 OR B7) AND (NOT A6 AND B6));
    R3 <= (NOT A7 OR B7) AND (NOT A6 OR B6);

    E4 <= (NOT A9 AND B9) OR ((NOT A9 OR B9) AND (NOT A8 AND B8));
    R4 <= (NOT A9 OR B9) AND (NOT A8 OR B8);

    E5 <= (NOT A11 AND B11) OR ((NOT A11 OR B11) AND (NOT A10 AND B10));
      R5 <= (NOT A11 OR B11) AND (NOT A10 OR B10);

    BO <= E5 OR
          (R5 AND E4) OR
          (R5 AND R4 AND E3) OR
          (R5 AND R4 AND R3 AND E2) OR
          (R5 AND R4 AND R3 AND R2 AND E1) OR
          (R5 AND R4 AND R3 AND R2 AND R1 AND E0);

MAG <= '1' WHEN (BO = '1') ELSE '0';

--MAG is a '1' if B > A

END archfb2mgcmp12;
```

A comparison between 2-, 3-, and 4-bit group sized implementation of a 12-bit magnitude comparator based on the borrow-lookahead scheme is shown in *Table 3*. As mentioned before, the number of passes through the logic array is the same for all group-bit-sizes. The number of PTs and MCs used vary as shown in the table. The user has a wide choice and needs to choose the right group-size depending on the application.

Table 2. Comparison of a 12-Bit Magnitude Compare between Different Group-Sizes

Group-Bit-Size	2	3	4
# of PTs	34	44	60
# of MCs	13	9	7
# of passes	2	2	2

Three-Output Comparators

The discussion on magnitude comparators has so far been restricted to the values of less than (<) and greater than or equal to (≥) only. The discussion in this section talks about producing all three outputs, namely '<', '>' and '='.

FB2EQMCMP12: 12-Bit Borrow-Lookahead Three-Output Magnitude Comparator Using 2-Bit Groups

This model combines all the concepts discussed in the magnitude comparator section into one design. This uses borrow-lookahead, 2-bit groups, and also produces three outputs. The block diagram of this model is shown in *Figure 21*.

There are two ways in which the Borrow-lookahead principle can be used to achieve the functionality of a three-output comparator.

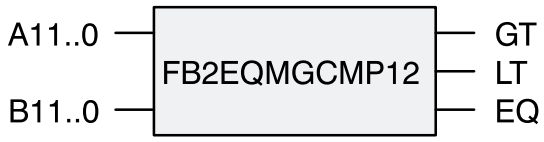

Figure 21. Block Diagram of a 12-Bit Borrow-Lookahead Three-Output Magnitude Compare

1. Use two passes for 'A < B' and 'A = B' each, then use a third pass for A > B using the results from A < B and A = B. This uses 62 PTs. The EQCOMP12 required for this model is built using three EQCOMP4s similar to the block diagram shown in *Figure 15*. The EQCOMP12 can also be built using four EQCOMPs, or two EQCOMP6s, or an EQCOMP8 and an EQCOMP4 or any other combination. As long as the EQCOMP model chosen does not sum-split, the value of EQCOMP12 can be realized in two passes using 25 PTs.

2. Use two passes to generate all three outputs. In this implementation a set of Es and Rs is required to create a value of LT (A − B). A second set of Es and Rs is required to obtain the value of GT (B − A). The value of EQ is also produced in 2 passes along with GT and LT. This scheme uses 97 PTs.

The first scheme is area efficient, but takes three passes though the logic array to generate the final results. The VHDL implementation for the first scheme is presented here. It is very easy to extrapolate the code for the second scheme.

```
--This VHDL code describes the implementation of a 3-output magnitude
--comparator. The borrow-lookahead principle using 2-bit groups was used
--to build this element

USE WORK.RTLPKG.ALL;

ENTITY fb2eqmgcmp12 IS
  PORT (
    A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0: IN BIT;
    B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0: IN BIT;
    EQ,LT,GT: OUT BIT);
END fb2eqmgcmp12;

ARCHITECTURE archfb2eqmgcmp12 OF fb2mgeqcmp12 IS

  SIGNAL E0,E1,E2,E3,E4,E5  : BIT;
  SIGNAL R0,R1,R2,R3,R4,R5  : BIT;
  SIGNAL X11,X10,X9,X8,X7,X6,X5,X4,X3,X2,X1,X0 : BIT;
  SIGNAL INT1, INT2, INT3: BIT;
  SIGNAL BO : BIT;
```

```
  attribute synthesis_off of E0,E1,E2,E3,E4,E5 : signal is true;
  attribute synthesis_off of R0,R1,R2,R3,R4,R5 : signal is true;
  attribute synthesis_off of INT1, INT2, INT3 : signal is true;

BEGIN

    E0 <= (NOT A1 AND B1) OR ((NOT A1 OR B1) AND (NOT A0 AND B0));
      R0 <= (NOT A1 OR B1) AND (NOT A0 OR B0);

    E1 <= (NOT A3 AND B3) OR ((NOT A3 OR B3) AND (NOT A2 AND B2));
    R1 <= (NOT A3 OR B3) AND (NOT A2 OR B2);
    E2 <= (NOT A5 AND B5) OR ((NOT A5 OR B5) and (NOT A4 AND B4));
    R2 <= (NOT A5 OR B5) AND (NOT A4 OR B4);

    E3 <= (NOT A7 AND B7) OR ((NOT A7 OR B7) AND (NOT A6 AND B6));
    R3 <= (NOT A7 OR B7) AND (NOT A6 OR B6);

    E4 <= (NOT A9 AND B9) OR ((NOT A9 OR B9) AND (NOT A8 AND B8));
    R4 <= (NOT A9 OR B9) AND (NOT A8 OR B8);

    E5 <= (NOT A11 AND B11) OR ((NOT A11 OR B11) AND (NOT A10 AND B10));
    R5 <= (NOT A11 OR B11) AND (NOT A10 OR B10);

    BO <= E5 OR
          (E4 AND R5) OR
          (E3 AND R5 AND R4) OR
          (E2 AND R5 AND R4 AND R3) OR
          (E1 AND R5 AND R4 AND R3 AND R2) OR
          (E0 AND R5 AND R4 AND R3 AND R2 AND R1);

LT <= '1' WHEN (BO = '1') ELSE '0';

-- LT is a '1' if A < B

GT <= '1' WHEN (LT = '0' AND EQ = '0' ) ELSE '0';

-- GT is a '1' if A > B

    X11 <= A11 XOR B11;
    X10 <= A10 XOR B10;
    X9 <= A9 XOR B9;
    X8 <= A8 XOR B8;
    X7 <= A7 XOR B7;
    X6 <= A6 XOR B6;
    X5 <= A5 XOR B5;
    X4 <= A4 XOR B4;
    X3 <= A3 XOR B3;
    X2 <= A2 XOR B2;
    X1 <= A1 XOR B1;
    X0 <= A0 XOR B0;
```

```
    INT1 <= (X11 OR X10 OR X9 OR X8);
    INT2 <= (X7 OR X6 OR X5 OR X4);
    INT3 <= (X3 OR X2 OR X1 OR X0);

    EQ <= NOT (INT1 OR INT2 OR INT3);

END archfb2eqmgcmp12;
```

Summary

A number of arithmetic elements frequently used in various applications were presented in this application note. The underlying concepts and the final implementations for all these models were also presented. Designs created with an understanding of the target architecture always perform better than generic designs. The LPM elements available in *Warp* are all geared towards obtaining the best performance, both in speed and area, for CPLDs. The concepts and implementations presented in this application note are used to build the various LPM elements. Understanding this application note will enable the user to understand the LPM elements better and exploit their availability in the best possible manner.

CPLDs are getting to be very popular with the programmable logic industry, and are widely used in DSP applications, PCs, Motherboards, Data Communication equipment, Multimedia, Instrumentation. etc. They have many advantages over other programmable logic devices. A few key advantages are listed here:

- Ease of use—Simple extension of AND-OR structure of small PLDs like 22V10
- Predictable timing model
- No fanout penalty
- Provide high speed of operation
- Off the shelf availability
- Cost effective solution

These advantages make CPLDs an ideal platform to implement high-performance arithmetic circuits in a cost-effective manner.

FPGAs inherently have more useable gates than CPLDs and also provide a very fine grain architecture. The major constraints to deal with FPGAs are I/O utilization, logic utilization, and timing. A particular design can be literally placed in many different places in an FPGA because of its fine grain architecture. In CPLDs the structure is very coarse grained and this pushes the number of constraints higher. The typical constraints to deal with arithmetic designs in CPLDs are product term count, macrocell count, number of inputs into a logic block, product term and macrocell placement, number of passes through logic array, and sum-splits. All of these facts make designing arithmetic operations with CPLDs a tougher task. Understanding the structure and capabilities of CPLDs is absolutely essential in creating efficient designs.

With the background provided in this application note, a designer should be able to create any algorithm or implementation for an arithmetic application. The user is strongly encouraged to read the VHDL textbook written by the PLD applications group to get a good grasp of VHDL and using it to implement efficient designs in CPLDs and FPGAs.