24-Bit ADDER Implementation in a CPLD

Introduction

High-speed DSP and arithmetic functions are in high demand. There is an ever-increasing need for speed. The purpose of this application note is to illustrate how to optimize a 24-bit adder in a Lattice Complex Programmable Logic Device, or CPLD. It is possible to implement a full 24-bit adder in just three levels of logic, allowing the adder to run at slightly over one-third the maximum operating frequency of the device.

Lattice produces the highest speed In-System Programmable™ CPLDs in the industry. Since the implementation of an adder in a CPLD requires multiple levels of logic, it makes sense to use Lattice's high-speed ispLSI® devices to implement an adder, particularly a high-performance adder. In this application note, the basic theory for adders is presented, followed by an actual implementation into a Lattice device. Using this implementation for a Lattice ispLSI 2096-125, this adder can operate at speeds of up to 46.3MHz.

One-Bit Arithmetic

A one-bit adder is the basic building block for understanding how to implement arithmetic functions in a CPLD. Throughout this application note, equations will be given to explain the theory behind the 24-bit adder. In order to make the equations more readable, the following symbol convention is used:

Logic Element	Symbol
AND	&
OR	#
XOR	$
NOT	!

The following equations add two bits, A and B, along with a carry-in, CIN, to get the SUM and a carry-out, COUT.

```
SUM  = A $ B $ CIN

COUT = (A & B) # (A & CIN) # (B & CIN)
```

The SUM equation states that SUM is a logical one when there is an odd number of ones being added. The COUT equation states that COUT is a logical one if any pair of the inputs are a logical one. Each of these one-bit adder stages is called a full adder since each stage has a carry-in. In contrast, half adders do not have the carry-in.

To build an adder of any size, simply cascade any number of these one-bit full adders with the carry-out of each stage feeding the carry-in of the next higher order stage. However, such an adder incurs an additional propagation delay for each stage as the carry-out from one stage feeds the carry-in of the succeeding stage. This method makes large counters very slow, since each additional stage adds a level of logic and another propagation delay. To create large and fast counters, use the following propagate-generate technique to limit the propagation delay to three levels.

To see how the propagate-generate technique works, note that the equation for COUT has all the information needed to determine if the one-bit adder stage either generates a carry out or propagates a carry-in.

```
COUT = (A & B) # (A & CIN) # (B & CIN)
     = (A & B) # ((A # B) & CIN)
```

As the equation shows, COUT is generated within the stage if both A and B are a logical one. A CIN is propagated to COUT if either A or B is a logical '1'.

If a generate term, G, and a propagate term, P, are substituted in the equation for COUT, the following equation results:

```
COUT = G # (P & CIN)
   where G = A & B
   and   P = A # B
```

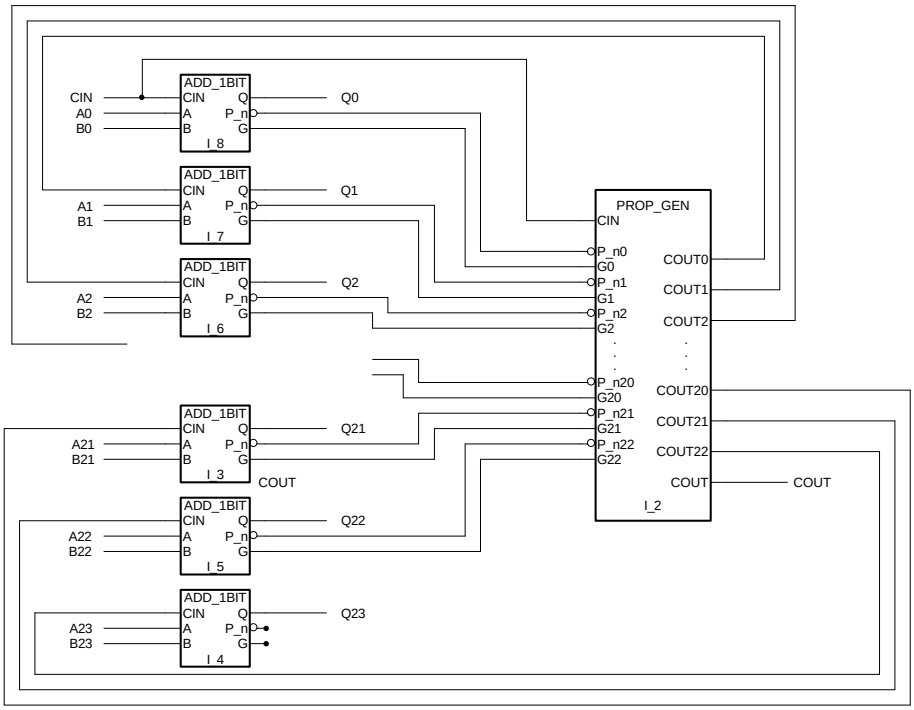
COUT is either generated within the full adder if G is a logical '1' or CIN is propagated to COUT if P is a logical '1'. Thus, for a counter of multiple stages, each stage can generate an output for G and P without any regard to CIN since the equations for G and P involve only the A and B inputs for that stage.

Finally, each stage can compute the sum for that stage provided it has CIN.

```
SUM = A $ B $ CIN
```

Since CIN for a given stage is the COUT of the previous stage, the incoming CIN is a logical '1' if either the previous stage generates a carry or if the previous stage propagates its own incoming carry. For example, the equations for CIN for the second through fifth bits of an adder are shown below:

Figure 1. 24-Bit Adder Using 1-Bit Stages




```
CIN1 = COUT0 = G0 # (P0 & CIN0)
CIN2 = COUT1 = G1 # (P1 & COUT0)
CIN3 = COUT2 = G2 # (P2 & COUT1)
CIN4 = COUT3 = G3 # (P3 & COUT2)
```

By substituting in each stage's equation the value for COUT coming from the previous stage, the following equations result:

```
CIN1 = COUT0
     = G0
     # (P0 & CIN0)
CIN2 = COUT1
     = G1
     # (P1 & G0)
     # (P1 & P0 & CIN0)
CIN3 = COUT2
     = G2
     # (P2 & G1)
     # (P2 & P1 & G0)
     # (P2 & P1 & P0 & CIN0)
CIN4 = COUT3
     = G3
     # (P3 & G2)
     # (P3 & P2 & G1)
     # (P3 & P2 & P1 & G0)
     # (P3 & P2 & P1 & P0 & CIN0)
```

Now each stage of the multiple-bit adder shown in Figure 1 can compute the sum by substituting the COUT developed from the propagate/generate outputs of the previous stage for its CIN.

```
SUM = A $ B $ CIN
```

Note that there are three levels of logic. At the first level, each stage computes its P and G. The second level computes the COUT from each stage for use as the CIN for the next higher order stage. Finally, at the third level, each stage of the adder computes the sum of A, B and CIN.

Two-Bit Arithmetic

Note that the equation for COUT uses one more product term at each stage than the previous stage used. Since CPLDs have a limited number of product terms per macrocell, too many product terms in large adders create a problem. Breaking the equation for COUT into two levels would work but slow down the addition. The solution is to use adder stages that consist of two-bit full adders instead of one-bit full adders. Propagate/generate pairs are generated from each two-bit stage to form the carry to the next higher two-bit stage. As shown in Figure 2, using two-bit adders involves half as many stages. Half as many stages results in half as many propagate/generate pairs from stage to stage for a given size adder. As a result, the highest order CIN equation requires half as many product terms.

The sum equations for the two-bit adder are:

```
SUM0 = A0 $ B0 $ CIN0
     = A0 $ ((B0 & !CIN0) # (!B0 & CIN0))
```

```
SUM1 = A1 $ B1 $ CIN1
     = A1 $ ((B1 & !CIN1) # (!B1 & CIN1))
```

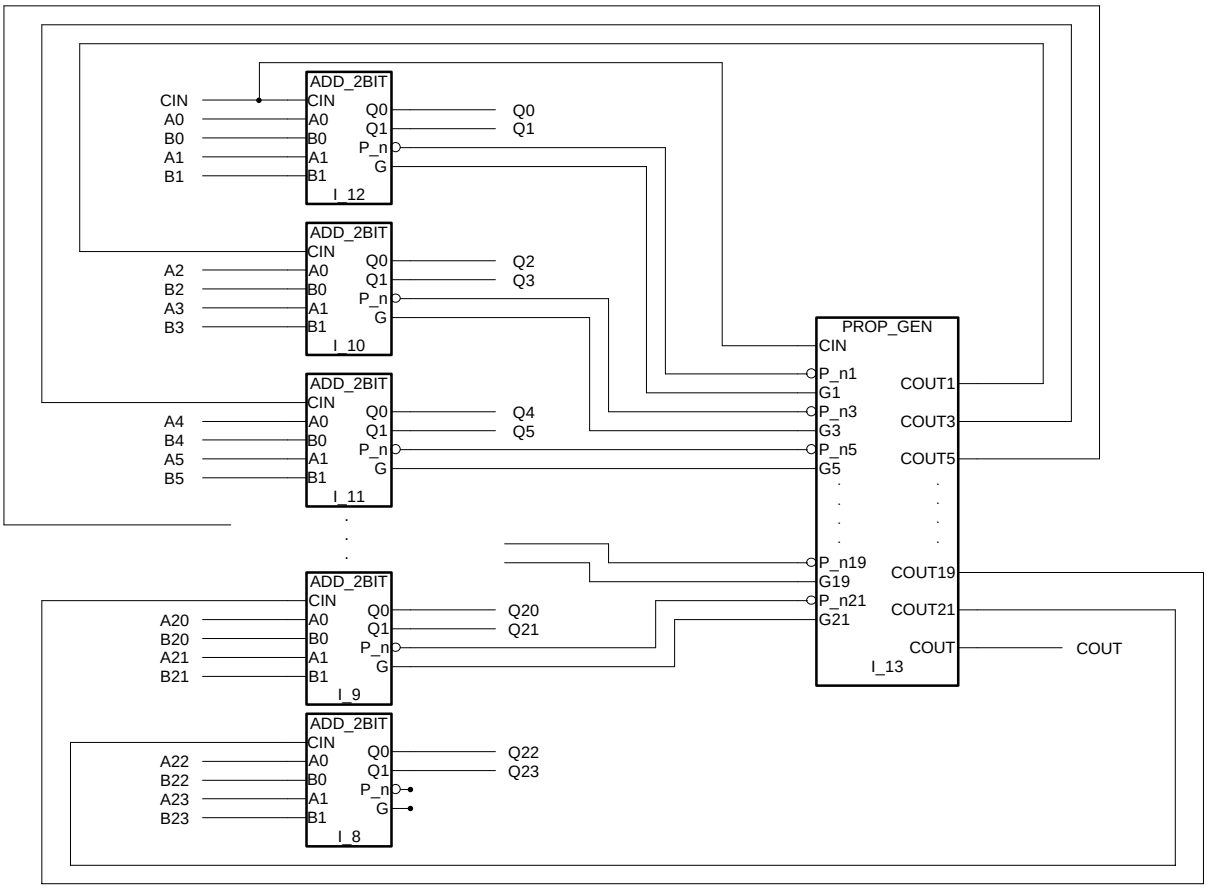
As shown, one of the XORs has been expanded out, since CPLDs have at most one XOR per macrocell. By replacing CIN1 with the stage's internal carry out from the previous stage, the following equation results:

```
CIN1 = COUT0
     = (A0 & B0) # (A0 & CIN0) # (B0 & CIN0)
```

After simplification, the sum equations become:

```
SUM0 = A0 $ ((B0 & !CIN0) # (!B0 & CIN0))
SUM1 = A1 $ ((!A0   & !B0 &  B1)
       #(!CIN0 & !B0 &  B1)
       #(!CIN0 & !A0 &  B1)
       #(A0    & B0  & !B1)
       #(CIN0  & B0  & !B1)
       #(CIN0  & A0  & !B1))
```

Figure 2. 24-Bit Adder Using Two-Bit Stages



Although the equation for SUM1 is complicated, it is easier to understand with knowledge of how the one-bit full adder works and from the definition of XOR. Remember that for the XOR operation:

```
Z = X $ Y
```

Z is the opposite of X if and only if Y is a logical one. Thus to get SUM0, A0 needs to be toggled (i.e. XORd) when either B0 is a logical one and there is no carry coming in (CIN0 a logical '0') or when B0 is a logical '0' and there is a carry. Similarly, to get SUM1, A1 needs to be toggled (XORd) either when B1 is a logical '1' and there is no possibility of a carry coming in (i.e. any pair of A0, B0 and CIN0 is a logical '0') or when B1 is a logical '0' and there is a carry coming in (i.e. any pair of A0, B0 and CIN0 is a logical '1').

The generate equations for the two-bit adder are:

```
G0 = A0 & B0
G1 = A1 & B1
     # (A1 & A0 & B0)
     # (B1 & A0 & B0)
```

G1 is generated either when both A1 and B1 are a logical '1' or when at least one of the addends in the second stage is a logical '1' and there is also a carry being generated within the first stage of the two-bit adder.

The propagate equation is simpler in the deMorganized form. Propagate is inactive when there is no possibility for the two-bit adder to propagate a carry. No possibility of propagating occurs when both A and B are a logical '0' in *either* stage of the two-bit adder.

```
!P1 = (!A0 & !B0) # (!A1 & !B1)
```

Although the sum equations and the propagate-generate equations are more complex for the two-bit full adder than for the one-bit adder, there are less product terms needed for the COUTs since there is only one propagate-generate pair for every two bits being summed.

The CIN/COUT equations are the same when using two-bit adders as when using one-bit adders, except now CIN is the carry-in and COUT is the carry-out for the two-bit adder stage. Again, for a given size adder, using two-bit adders involves half as many stages, half as many carries and propagate/generate pairs, and half as many product terms for the highest order equation for the carry-in, CIN.

Three-Bit Arithmetic

Figure 3 shows the same counter using three-bit adder stages. The reasoning used to develop the equations for the two-bit full adder can be used to develop the equations for the three-bit adder. While the equations for SUM0 and SUM1 are the same as the two-bit adder, one can gain an understanding for the SUM2 equation from knowledge of how the one-bit adder works and the definition of XOR. To get SUM2, A2 needs to be toggled (i.e. XORd) when either:

B2 is a logical '1' and there is no possibility of a carry coming in from the middle stage of the three-bit adder (Case 1); or

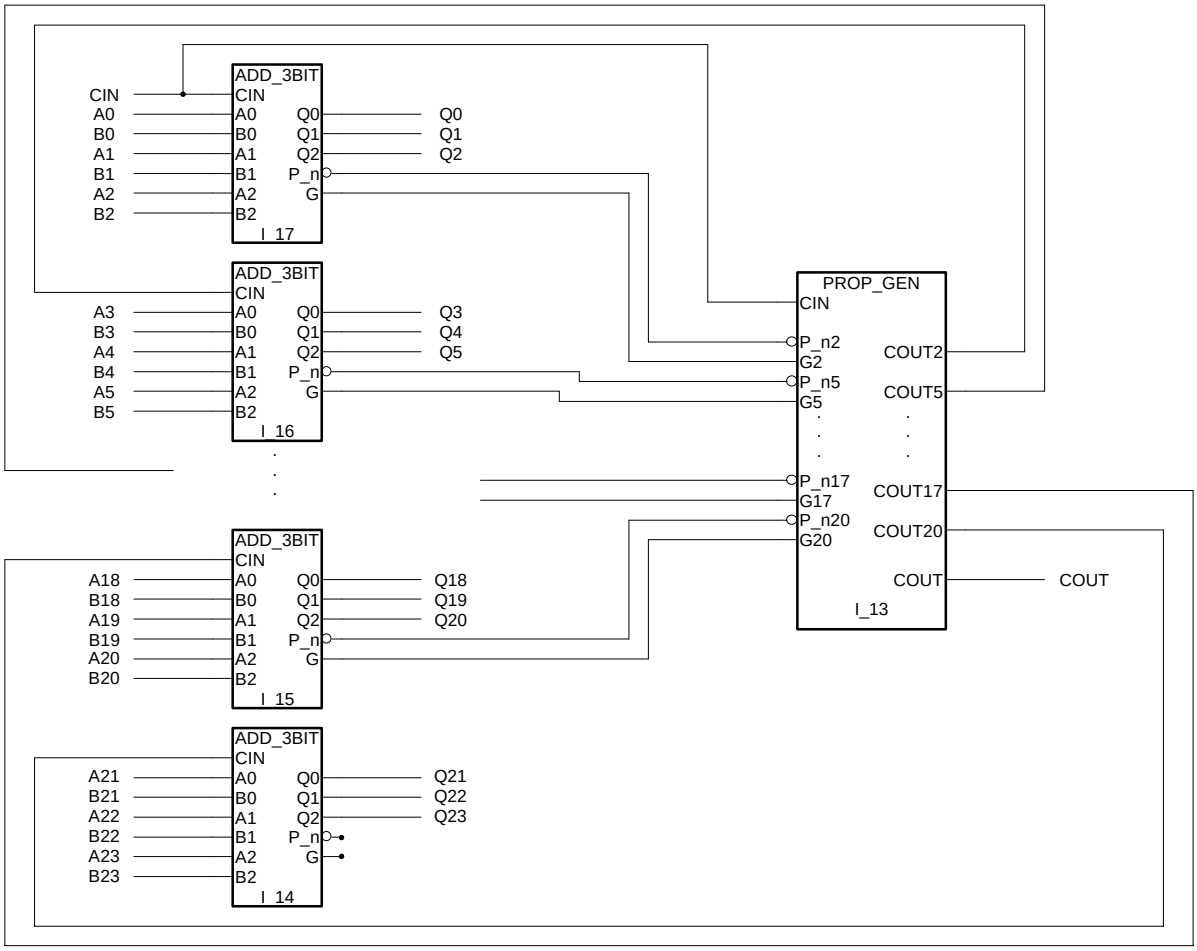
B2 is a logical '0' and there is a carry coming in from the middle stage (Case 2).

In Case 1, no carry can come from the middle stage of the three-bit adder if both A1 and B1 are a logical '0'. In other words, a carry is stopped from propagating if both A1 and B1 are a logical '0'. If neither A1 or B1 are a logical '0', a propagating carry has to be stopped at the first stage.

In Case 2 where B2 is a logical '0', A2 is toggled when there is a carry coming in from the middle stage. A carry comes from the middle stage if A1 and B1 are both a logical '1'. If only one of the A1, B1 pair is a logical '1', then the middle stage will still send a carry to the third stage if the middle stage gets a carry from the first stage.

```
SUM0 = A0 $ ((B0 & !CIN0) # (!B0 & CIN0))
SUM1 = A1 $ ((!A0   & !B0 &  B1)
       #(!CIN0 & !B0 &  B1)
       #(!CIN0 & !A0 &  B1)
       #(A0    & B0  & !B1)
       #(CIN0  & B0  & !B1)
       #(CIN0  & A0  & !B1))
SUM2 = A2 $ ((!A1   & !B1 &  B2)
       #(!A0   & !B0 & !B1 &  B2)
       #(!CIN0 & !A0 & !B1 &  B2)
       #(!CIN0 & !B0 & !B1 &  B2)
       #(!A0   & !A1 & !B0 &  B2)
       #(!CIN0 & !A0 & !A1 &  B2)
       #(!CIN0 & !A1 & !B0 &  B2)
       #(A1    & B1  & !B2)
       #(A0    & B0  &  B1 & !B2)
```

Figure 3. 24-Bit Adder Using 3-Bit Stages




```
#(CIN0  & A0  & B1  & !B2)
#(CIN0  & B0  & B1  & !B2)
#(A0    & A1  & B0  & !B2)
#(CIN0  & A0  & A1  & !B2)
#(CIN0  & A1  & B0  & !B2))
```

As shown next, G0 and G1 are as before. G2 is seen to be generated when:

(1) both A2 and B2 are a logical ‘1’; or

(2) at least one of the addends is a logical ‘1’ and there is a carry being generated in the middle stage; or

(3) at least one of the addends is a logical ‘1’ and there is a carry being generated in the first stage and the middle stage propagates the carry to the third stage.

```
G0 = A0 & B0
G1 = A1 & B1
     # (A1 & A0 & B0)
     # (B1 & A0 & B0)
G2 = A2 & B2
     # (A2 & A1 & B1)
     # (A2 & A1 & A0 & B0)
     # (A2 & B1 & A0 & B0)
     # (B2 & A1 & B1)
     # (B2 & A1 & A0 & B0)
     # (B2 & B1 & A0 & B0)
```

The propagate equation is again simpler in the deMorganized form. Propagate is inactive when there is no possibility for the three-bit adder to propagate a carry.

No possibility of propagating occurs when both A and B are a logical '0' in one of the stages of the two-bit adder.

```
!P1 = (!A0 & !B0)
#(!A1 & !B1)
#(!A2 & !B2)
```

Since the SUM2 term for the three-bit full adder already uses 14 product terms in conjunction with the XOR operation, we will not bother implementing the four-bit adder equations, since the SUM3 term for the four-bit adder would need an excessive number of product terms for what is available in a CPLD.

Implementing the 24-Bit Adder

ABEL-HDL (High-level Design Language) was used to implement the equations for each of the three-bit stages of the 24-bit adder. The first section of the source file contains pLSI® property statements that are ignored by the ABEL compiler but used later by the Lattice Fitter. The first several pLSI property statements specify to the Fitter what part to use, to reserve the ISP™ pins, which clock to use, and what type of simulation file to produce. The 'preserve' property statements specify to the fitter which nodes to preserve, thus controlling where the breaks occur in the levels of logic. The 'LXOR2' property specifies the nodes where the 'hardware' or built-in XOR is to be used. The 'ecp' (End Critical Path) states where product-term-bypass should be attempted to maximize performance. The 'CRIT' property may be used to call for the bypassing of the Output Routing Pool. These optional property statements allow easy use of the Lattice Fitter without sacrificing power. One can leave the property statements out and let the fitter run with the defaults. In this case, one can use knowledge of the function and the power of the pLSI property statements to guide the Fitter in giving the exact fit desired.

After the property statements is the section containing ABEL statements for specifying the inputs and outputs and for assigning pins. Node declarations are made for the propagates and generates from each three-bit full adder stage. Additional node declarations for each stage's carry-in, for the D inputs of the flip-flops in the macrocells, and for the macrocells follow. Note that the node definition for the D inputs calls for XOR as part of the istype. This statement is ensures that the XORs are not reduced to AND-OR when ABEL reduces the logic but are passed intact to the fitter.

Set declarations, defined next, allow ABEL's powerful feature of handling multiple similar signals in a single equation and in the test vector section. The macro definitions reduce the size of the source file. The macros are defined with dummy arguments (arguments preceded with a '?' symbol). Then, the compiler expands the macros by substituting the actual arguments for the dummy arguments. The 24-bit adder uses macros to define most of the equations used during the expansion for the SUM outputs. The XOR part of the SUM equations could have been done inside the macro. Instead, it was done outside of the macros, as can be seen in the EQUATIONS section, where the equations for D-inputs have the XORs explicitly stated. Macros also define the propagate/generate equations for each three-bit adder stage.

The EQUATIONS section enables the macros for the propagate/generates, creates the second level of logic (the carry-ins), and uses the add macros to form the D-inputs for the flip-flops at the third logic level. Finally, the D-inputs are assigned to their respective flip-flops, and the clock and outputs are assigned.

Testing the Design

After compilation, the report file from the Lattice fitter will show three levels of logic with the breaks in the logic levels occurring exactly where they were specified by the pLSI property statements. Functional simulation could be achieved by simulation software. In this application, a judicious choice of test vectors allows a quick confirmation of the design. Since there are three levels of logic, we can expect that a 125MHz PLD should run about one-third of that speed or a little faster. Lattice's static timing analyzer adds up the prop delays, the set-up times and the clock-to-Q times. Then the timing analyzer gives the reciprocal of this time as the maximum frequency of operation. Here the 24-bit adder is calculated to run as fast as 46.3MHz.

Summary

Implementing this design in a Lattice ispLSI device allows the design engineer to easily program the CPLD without the use of a stand-alone programmer. The need is eliminated to remove and install devices in sockets. ISP also allows the use of dense, high-performance CPLDs in small, high pin-count packages without concern about damaging pins from excessive handling.

In addition to facilitating the programming of the PLDs at the design stage, ISP affords significant cost savings during manufacturing, since the blank parts can be assembled onto the PC board just like non-programmable devices. ISP eliminates all special steps in the manufacturing process that used to be required for PLDs.

Programming is done after assembly using ISP Daisy Chain Download software or the PLDs can be programmed in high volume applications on the same ATE (automatic test equipment) already used during the final board test. The ATE test vectors used for programming are automatically generated from the JEDEC file by using Lattice's ispATE™ software. There is no need for a separate PLD programming station, no inventory of programmed parts, and no need for sockets.

References

1 Lattice Semiconductor Corporation. *Lattice System Macro Library, Version 2.00*. 1993.

2. Rangasayee, Krishna. "Complex PLDs Let You Produce Efficient Arithmetic Designs." *EDN* (June 20, 1996): 109.

Source File

The source file for this design is available in text format (file name: an8007.txt) on the ISP Encyclopedia CD-ROM and the Lattice web site. Please note that if you copy or download this file to another directory, this link will not work.