INTRODUCTION

In this section we will take a detailed look at several aspects of combinatorial logic design. Most combinatorial design applications can be easily segmented into five major fields.

- Encoders and Decoders
- Multiplexers
- Comparators
- Adders and Arithmetic Logic
- Latches

We will not only focus on the design methodology for these functions, but will also explore further function-specific PLD selection requirements. Generalized designs will be developed, which can be customized later to suit specific system applications. Ways of optimizing the design will also be discussed.

Encoders and Decoders

Two of the most important functions required in digital design are encoding and decoding. The encoding and decoding of data are used extensively in digital communications as well as in peripherals. Both these areas use various complex encoding and decoding techniques. Most of these techniques are extensions of the simple encoding and decoding techniques often used in other digital designs. In this discussion we will focus on simple encoding and decoding techniques. More complex techniques will be discussed later.

Encoders

A binary code of n bits can be used to represent 2n distinct pieces of coded data. A simple combinatorial encoder is a circuit which generates n bits of output information based upon one of the 2n unique pieces of input data information. This encoding of information is controlled by other independent control signals in a typical digital circuit.

An illustration of a typical encoder is shown in Figure 1. The design methodology typically followed is based on truth tables (Table 1), from which the Boolean equations are directly derived for the design. The same generic device selection considerations discussed in the section on PAL device design methodology apply for encoder and decoder designs.

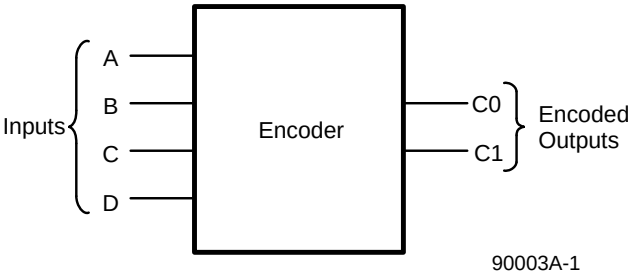


Figure 1. A Block Diagram of an Encoder

Table 1. Truth Table of a Typical Encoder

Inputs				Outputs	
A	**B**	**C**	**D**	**C0**	**C1**
1	0	0	0	L	L
0	1	0	0	L	H
0	0	1	0	H	L
0	0	0	1	H	H

The Boolean equations can then be optimized using Karnaugh maps or the software minimizer.

The resulting Boolean equations are:

```
C1   =  /A  *   B  *  /C  *  /D
     +  /A  *  /B  *  /C  *   D

C0   =  /A  *  /B  *   C  *  /D
     +  /A  *  /B  *  /C  *   D
```

A Priority Encoder

Let us take another look at the encoder example of Table 1. In this example it is assumed that only one of the inputs A, B, C or D is asserted HIGH at any one time. If two of the inputs are asserted HIGH simultaneously, a conflict would be created. To resolve this, a priority needs to be assigned to each of the inputs. Such a priority assignment is used to select a particular element when several inputs are asserted simultaneously. Each input is assigned a priority with respect to the other inputs. The output code generated is the code assigned to the highest priority input asserted.

Thus, a priority encoder is a combinatorial circuit block similar to a general encoder, except that the inputs are assigned a priority. Such priority encoders are used often in state machine applications, where they detect

the occurrence of the highest priority event. They are also used for microprocessor interrupt controllers, where they detect the highest priority interrupt. Another use for priority encoders is in bus control, where they are used in arbitration schemes for allowing selective access to the bus.

The model of a priority encoder is shown in Figure 2. The four input signals are A, B, C and D. These are to be encoded as LL, LH, HL and HH outputs. Let us assign priority to D over C, C over B, and B over A. The next design step would be to modify the truth table (Table 2) to reflect these priorities.

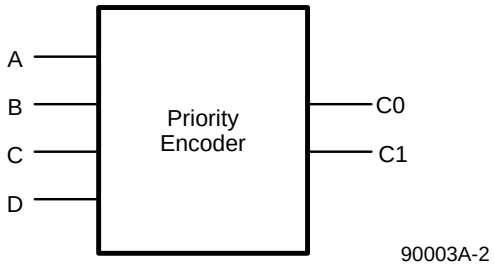


Figure 2. A Four-Input Priority Encoder Block Diagram

Table 2. Priority Encoder Truth Table

Inputs				Outputs	
A	**B**	**C**	**D**	**C0**	**C1**
1	0	0	0	L	L
0	1	0	0	L	H
0	0	1	0	H	L
0	0	0	1	H	H
X	1	0	0	L	H
X	X	1	0	H	L
X	X	X	1	H	H

Priority Assignments

The Boolean equations, directly derived from the truth table, are:

```
C1    =  /A   *    B   *  /C   *     /D
      +  /A   *   /B   *  /C   *      D
      +           B   *  /C   *     /D
      +                              D

C0    =  /A   *   /B   *   C   *     /D
      +  /A   *   /B   *  /C   *      D
      +                   C   *     /D
      +                              D
```

These equations can be further optimized by the design software to the following:

```
C1    =   D   +  /C   *   B
C1    =   D   +   C
```

Although a priority encoder is a purely combinatorial function, output registers are frequently used to hold the output signal stable for longer durations.

Decoders

A decoder performs the reverse function of an encoder. It converts an n-bit code to one of its 2^n unique items. It is a combinatorial circuit designed such that at most one of its several outputs will be asserted based upon the unique input codes.

A decoder may have as many outputs as there are possible binary input selection combinations. As shown in the truth table (Table 3), only one output may be asserted at any time. When a new combination is applied, another output is asserted and the original output is returned to its non-asserted state.

Table 3. The Truth Table of an Active-LOW 4-to-16 Decoder

Input Select Lines				Output Lines														
A	B	C	D	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15
0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

The Boolean logic equations can be directly derived from the truth table shown in Figure 5. The procedure is the same as explained in the previous section on PLD design methodology. The Boolean equations derived are shown in Figure 3.

```
/Q0   =  /D  *  /C  *  /B  *   /A
/Q1   =  /D  *  /C  *  /B  *    A
/Q2   =  /D  *  /C  *   B  *   /A
/Q3   =  /D  *  /C  *   B  *    A
/Q4   =  /D  *   C  *  /B  *   /A
/Q5   =  /D  *   C  *  /B  *    A
/Q6   =  /D  *   C  *   B  *   /A
/Q7   =  /D  *   C  *   B  *    A
/Q8   =   D  *  /C  *  /B  *   /A
/Q9   =   D  *  /C  *  /B  *    A
/Q10  =   D  *  /C  *   B  *   /A
/Q11  =   D  *  /C  *   B  *    A
/Q12  =   D  *   C  *  /B  *   /A
/Q13  =   D  *   C  *  /B  *    A
/Q14  =   D  *   C  *   B  *   /A
/Q15  =   D  *   C  *   B  *    A
```

Figure 3. Decoder Boolean Logic Equations

Probably the most commonly used decoders are the address decoders required by most microprocessors and bus interfaces. These also constitute the most common application of PLDs in digital designs. The design considerations for address decoders have been covered earlier in the PLD Design Methodology section. Later we will develop a general Boolean equation for an address decoder circuit when we discuss range decoders.

Encoder/Decoder Device Selection Considerations

The general device selection considerations are listed below. Based upon the number of inputs and outputs required, a device can be selected.

- Number of Input Pins
- Number of Output Pins
- Number of I/O Pins
- Device Speed
- Device Power Requirements
- Number of Registers
- Number of Product Terms
- Output Polarity Control

Encoders typically require a large number of inputs and fewer outputs, whereas decoders typically require a large number of outputs and fewer inputs.

Notice from the truth table that there is no combination of inputs that will send all the outputs to their non-asserted state. Many designs actually need to be able to make all outputs inactive. This can be done simply by putting enable lines in all of the output AND gates. Many such design modifications can be easily added once the basic Boolean equations have been derived, instead of redoing the truth table.

Another important device selection consideration for encoders and decoders is the number of product terms required for a design. A careful selection of code values

(and priority assignments in priority encoders) can often reduce the required number of product terms. This can sometimes determine whether or not a design fits a device successfully. Figure 4 shows the truth tables of two simple partial 3-to-2 encoders. The product terms required for the two designs are different due to the different assignment of encoded bits.

Inputs			Outputs	
A	B	C	X1	X0
1	0	0	0	0
0	1	0	0	1
0	0	1	1	0

```
X1 =  /A * /B * C

X0 =  /A * /B * /C
```

Inputs			Outputs	
A	B	C	X1	X0
1	0	0	0	1
0	1	0	1	0
0	0	1	1	1

```
X1 = /A * B * /C
   + /A */B *  C

X0 =  A * /B * /C
   + /A * /B *  C
```

10173D-26

Figure 4. Two Encoders with Different Product Term Requirements

Another way of looking at a decoder is as a logic function which, depending upon the select code applied, connects one data input to the selected outputs. Also known as a demultiplexer, a decoder essentially connects an input to one of 2^n outputs based upon n select code bits. The reverse logic function, which combines data from multiple sources to an output signal, is called a multiplexer and is discussed next.

Multiplexers

A multiplexer (sometimes referred to as a data selector) is a special combinatorial circuit, widely used in digital design. It is designed to gate one of several inputs to a single output. The input selected for connection to the output is controlled by a separate set of select inputs.

The traditional use of a multiplexer is for “time division multiplexing” in data communication, when gating several data lines to a single data transmission line for short intervals of time. The data received is then demultiplexed by using a demultiplexer.

The design methodology employed for multiplexer design is the truth-table approach. As an example, we can look at a three in put-to-one-output (3:1) multiplexer, which uses two select signals A and B. Based on these two select bits, the data on one of the three inputs is sent to the output. The truth table is shown in Table 4.

Table 4. Truth Table for a Three-to-One Multiplexer

Select		Inputs			Output
B	A	I1C0	I1C1	I1C2	O1Y
0	0	0	X	X	0
0	0	1	X	X	1
0	1	X	0	X	0
0	1	X	1	X	1
1	0	X	X	0	0
1	0	X	X	1	1

Deriving the Boolean equation from this truth table is a straight forward task. In this case no further minimization is possible. The Boolean equation is:

```
/01Y  =  /B   *   /A   *    /I1C0
      +  /B   *    A   *    /I1C1
      +   B   *   /A   *    /I1C2
```

The equations derived in the above example can be easily generalized for other multiplexers. The symbol for a general 2^n-inputs-to-one-output multiplexer is shown in Figure 5 where n select lines are used.

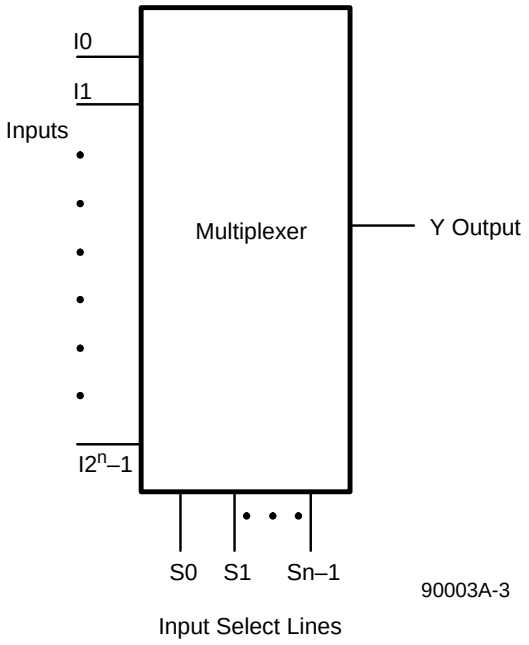


Figure 5. General Model of a 2^n-to-1 Multiplexer

The Boolean equations are:

n=2

```
Y  = /S1  * /S0  *   (I0)
   + /S1  *  S0  *   (I1)
   +  S1  * /S0  *   (I2)
   +  S1  *  S0  *   (I3)
```

n=3

```
Y  = /S2  *   /S1   * /S0  *  (I0)
   + /S2  *   /S1   *  S0  *  (I1)
   + /S2  *    S1   * /S0  *  (I2)
   + /S2  *    S1   *  S0  *  (I3)
   +  S2  *   /S1   * /S0  *  (I4)
   +  S2  *   /S1   *  S0  *  (I5)
   +  S2  *    S1   * /S0  *  (I6)
   +  S2  *    S1   *  S0  *  (I7)
```

Multiplexer Device Selection Considerations

Multiplexers typically require more inputs than outputs, so the devices with a large number of inputs and I/Os are usually more useful. Careful consideration must also be given to the number of product terms available on each output.

Several multiplexers are often used simultaneously to route multiple address and data bits, under the control of the same select lines. In such cases, multiple devices can be cascaded when the number of inputs and outputs exceeds device limits. Cascading is also possible for large multiplexers that do not fit in a single device. In such cases, the select bits should also be judiciously selected for each PLD, to minimize the number of product terms.

Another common trick for designing a multiplexer is to connect a number of outputs together and control the output enables using the select bits to multiplex data. Timing considerations for such designs include the output enable and disable times, which should be carefully selected to avoid output contentions.

Comparators

A comparator is a combinatorial circuit designed primarily to compare the relative magnitude of two binary numbers. Table 5 shows the truth table for a two-bit comparator.

Table 5. Truth Table for a Comparator

Inputs				Outputs		
A		B		EQL	LES	GTR
A2	A1	B2	B1	A=B	A<B	A>B
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

A basic comparator compares two numbers only for equality, and generates the EQL signal (indicating A=B). An extension, called a magnitude comparator, also generates the LES signal (indicating A<B) and GTR signal (indicating A>B). Based on this truth table, the equations for the three output signals EQL, LES and GTR can be easily derived. These equations can then be optimized by using Boolean algebra, Karnaugh maps, or the minimization routine available with the software.

The final Boolean equations are:

```
EQL   = /A2* /A1   * /B2   * /B1
      + /A2   *   A1* /B2   *  B1
      +  A2* /A1       *  B2   *    /B1
      +  A2   *   A1   *  B2   *     B1

LES   = /A2   * /A1   * /B2   *     B1
      + /A2   * /A1   *  B2   *    /B1
      + /A2   * /A1   *  B2   *     B1
      + /A2   *    A  *  B2   *    /B1
      +/A2    *   A1   *  B2   *     B1
      + A2 */A1    *  B2    *  B1

      = /A1   * /B2   *  B1
      + /A2   * /A1   *  B1
      + /A2   *   B2

GTR   = /A2   *   A1   * /B2   *    /B1
      +  A2   * /A1   * /B2   *    /B1
      +  A2   *   A1   * /B2   *    /B1
      +  A2   * /A1   * /B2   *     B1
      +  A2   *   A1   * /B2   *     B1
      +  A2   *   A1   *  B2   *    /B1

      = /A1   * /B2   * /B1
      + /A2   *   A1   * /B1
      + /A2   * /B2
```

Comparator Device Selection Considerations

The number of product terms needed is directly related to the number of bits compared. For LES (less than) and GTR (greater than) functions, the number of product terms required depends upon the number of bits in the two operands compared, as well as their value. The LES and GTR equations can be written as follows:

```
LES   =  B2   * /A2
      + (B2 :+:/A2)   *  B1*/A1

GTR   =  A2   * /B2
      + (A2 :+:/B2)   *  A1*/B1
```

These equations can then be extended for a general comparison of n-bit comparands as follows:

```
LES   =  Bn   * /An
      + (Bn :+:/An)   *Bn-1  *  /An-1
      + (Bn :+:/An)   *(Bn-1:+:/An-1)
        *Bn-2 */An-2

      + ...
      + ...
      + ...

      + (Bn :+:/An)   *(Bn-1:+:/An-1)...
        (B2 :+:/A2)   *  B1   *    /A1

GTR   =  An   * /Bn
      + (An :+:/Bn)   *An-1  *  /Bn-1
      + (An :+:/Bn)   *(An-1:+:/Bn-1) *
        An-2 */Bn-2

      + ...
      + ...
      + ...

      + (An :+:/Bn)   *(An-1:+:/Bn-1)...
        (A2 :+:/B2)   *  A1* /B1
```

The total number of product terms required for an n-bit comparison is 2^n–1. Comparators required a large number of product terms so, devices that offer many product terms can be used very effectively.

As is obvious from these equations, comparators require exclusive-OR functions. They can be efficiently implemented in devices that offers exclusive-OR functions but, can still be implemented in those devices that do not.

The values of the comparands themselves affect the number of product terms used. When the comparison is made with comparands which are power-of-two numbers, the number of product terms required can be reduced drastically. This essentially relies on the fact that when the lower bits of a comparand are all zeros only the highest bit needs to be compared, requiring only one product term. For example, in a two-bit comparator, if A1 is zero and A2 is one, the equation for the greater-than function becomes very simple and requires only one product term:

```
GTR = /B2
```

The general equation for the GTR signal can also be simplified when comparing a number B to a fixed power-of-two comparand A with p least significant zeros.

```
A  =  000010000 ...  00
      n   p           1

GTR = /Bn */Bn-1...*/Bp+1 */Bp
```

This general GTR equation can also be considered as an equation for comparing a number to a range of numbers extending from zero to number A. In fact, this trick is used very often by many system designers for address decoder functions. In the PLD design methodology section the ROMCS1 signal is one such signal that is generated for the address range from (000000) hex to (0FFFFF) hex. For this design n=23, the comparand A=(0FFFFF + 1)=100000, and p=21. Substituting in the general equation we get the same address decoder Boolean logic equation.

```
ROMCS1 = /A23 * /A22* /A21
```

As such designs require few product terms and no XOR gates, they are efficiently implemented on standard combinatorial PLDs. A general form of range comparators with two boundary comparands will be discussed later.

The third output signal is the EQL signal. The EQL Boolean equation tells us whether the two numbers are identical. Such information is useful not only in address decoders, but also in digital signal processing designs. This equation requires a large number of product terms. A closer examination reveals that it is essentially an exclusive-OR function.

```
EQL = /A2 * /B2 * (/A1 * /B1 + A1 * B1)
    + A2 *  B2 * (/A1 * /B1 + A1 * B1)

EQL = (A1:*:B1)* (A2:*:B2);Exclusive-NOR
                           ;function
```

Inverting this:

```
/EQL = (A1:+:B1) + (A2:+:B2); Exclusive–OR
                            ; function
```

This equation can be extended to give a general equation for equal-to comparison for two n-bit comparands.

```
/EQL  = (An :+: Bn)
      + (An–1 :+: Bn–1)
      + (An–2 :+: Bn–2)
      + (An–3 :+: Bn–3)
      + ...
      + ...
      + (A1 :+: B1)
```

This inverted equation is implementable in the sum-of-products form of the exclusive-OR functions, and can be easily expanded to the following:

```
/EQL  = A1 * /B1 + /A1 * B1
      + A2 * /B2 + /A2 * B2
      + A3 * /B3 + /A3 * B3
      + ...
      + ...
      + An * /Bn + /An * Bn
```

This gives us a general sum-of-products form of a comparator equation which is easily implemented in PAL devices. A n-bit comparator requires 2n product terms.

Note that the EQL equation, as well as GTR and LES equations, rely upon the XOR function. Often the logic represented by the equations is implemented in two or more devices.

Let us analyze these equations further. The LES and GTR outputs indicate whether one number is greater than or less that another. In fact, these equations can also be judiciously combined to get a comparison of a range of numbers such as A>X>B. Such range comparisons are very useful for address decoder circuits.

Range Decoders

Range decoders implemented as address decoders are one of the most commonly used applications of PLDs in digital systems. A good example is the address decoder illustrated earlier. Range decoders compare a number (address) to a given range of comparands (addresses). One way to arrive at the range decoder Boolean equations is to use the traditional truth table approach. Another way is to use the Boolean equations generated earlier in the comparator section for greater-than and less-than functions. To decode a range of three-bit numbers from B to A, we must compare another number X such that A>X>B. The Boolean equations for the GTR (A>X) and LES (B<X) functions are illustrated below:

```
GTR   =  A3 */X3
      + (A3 :+:/X3)   *  A2* /X2
      + (A3 :+:/X3)   * (A2:+: /X2 *A1 *
        /X1

LES   =  X3   * /B3
      + (X3 :+:/B3)   * X2 * /B2
      + (X3 :+:/B3)   * (X2:+: /B2)* X1
        */B1
```

Combining these two equations can give us a range signal which will be asserted only when A is greater than X and X is greater than B. The combined Boolean equation follows:

```
RANG  = (A3   * /X3
      + (A3 :+:/X3)   *  A2 * /X2
      + (A3 :+:/X3)   * (A2 :+: /X2)* A1 *
        /X1) * (X3*/B3
      + (X3 :+:/B3)   *  X2 * /B2
      + (X3 :+: /B3   * (X2 :+:/B2) * X1
        */B1)
```

Using Boolean algebra we get the following equation:

```
RANG   =
          (A3  :+:/X3)  *  (X3  :+: /B3)   *  (A2  :+: /X2)*  A1   *   /X1 *  X2  *  /B2
       +  (A3  :+:/X3)  *  (X3  :+: /B3)   *  (X2  :+: /B2)*  A2   *   /X2 *  X1  *  /B1
       +  (A3  :+:/X3)  *  (A2  :+: /X2)   *  A1    *  /X1 *  X3   *   /B3
       +  (X3  :+:/B3)  *  (X2  :+: /B2)   *  A3    *  /X3 *  X1   *   /B1
       +  (A3  :+:/X3)  *  A2    *   /X2   *  X3    *  /B3
       +  (X3  :+:/B3)  *  A3    *   /X3   *  X2    *  /B2
```

The general equation for n-bit comparands can also be obtained by extending these equations.

```
RANG   =  (An   *  /Xn
       +  (An  :+:/Xn)  *  An–1   * /Xn–1
       +  (An  :+:/Xn)  *  (An–1:+:/Xn–1) *  An–2  *  /Xn–2
       +  ...
       +  ...
       +  (An  :+:/Xn)  *  (An–1:+:/Xn–1)... (A2  :+: /X2)*  A1   *   /X1)
       *
          (Xn   *  /Bn
       +  (Xn  :+:/Bn)  *  Xn–1   * /Bn–1
       +  (Xn  :+:/Bn)  *  (Xn–1:+:/Bn–1) *  Xn–2  *  /Bn–2
       +  ...
       +  ...
       +  ...
       +  (Xn  :+:/Bn)  *  (Xn–1:+:/Bn–1)... (X2  :+: /B2)*  X1   *   /B1)
```

The number of product terms required is clearly very large and can easily exceed one hundred for an eight-bit range comparator. Most microprocessors have addresses which exceed 16 bits. In order to fit the design on a PAL device, one commonly used technique is to select the address range defined by A and B such that the range extends from address B+1 to A–1, where A and B+1 are power-of-two numbers. Because the address space is aligned on the power-of-two boundaries, a number of bits of the address comparands will be zero. When implemented in Boolean equations, this substantially reduces the number of product terms required.

The maximum number of product terms required for a three-bit range decoder shown above, with any comparand values, is 28. If the address chosen is from 2 to 3, resulting in A=4 and B=1, then only one product term will be required.

```
RANG = /X3 * X2
```

Similarly, for a range from one to five, B =1 and A = 6 (a multiple-of-two), and the number of product terms required is only two.

```
RANG  = /X3 * X2
      + X3 * /X2
```

Thus, a careful selection of range boundaries allows such logic functions to be implemented easily in PLDs. Such reduction in logic obviously also holds true for discrete implementations. Most address decoders are designed with address ranges with boundaries that are power-of-two numbers, and require few product terms for implementation.

For a power-of-two address range, the comparand A would be a power-of-two; 2, 4 or 8. These are numbers whose least significant bits are all zeros. Similarly, comparand B will be a power-of-two number (minus one); 1, 3, and 7. These are numbers whose least significant bits are all ones. Substituting these in the general equation for range comparators we arrive at:

```
A   =  0000100000000000
       n   p       1

B   =  0000000000011111
       n   q       1

RANG  =    /Xn * /Xn–1 *...* /Xp *
           (Xp–1 + Xp–2 + ... +Xq)
```

This is a general equation for a power-of-two range comparison. In the address decoder example, the ROMCS2 signal addresses the range from 100000 to 1FFFFF, in which case B=0FFFFF and A=2FFFFF. Here n=23, p=22, and q=21. The address decode equation for the ROMCS2 signal can be arrived at by substituting:

```
ROMCS2 = /A23 * /A22 * A21
```

This is the same equation that was found from the truth table.

Such designs are very common for address decoder applications. These do not require any XOR gates, and can be implemented in standard combinatorial PLDs with only sum-of-products logic.

Adders/Arithmetic Circuits

Digital systems are designed to carry out a variety of arithmetic instructions on binary numerical data. A good example is the ALU (Arithmetic Logic Unit) used in digital computers. The basic function of an ALU is that of an adder performing addition on two binary numbers. A binary adder takes two inputs, adds them, and generates the binary sum. A full adder is a one-bit adder with carry-in and carry-out; this is the basic building block of any adding circuit. The truth table of such an adder is shown in Table 6.

Table 6. Truth Table for a Full Adder

Inputs			Outputs	
A	**B**	**C_{IN}**	**Y**	**C_{OUT}**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

This truth table is then used to form the Boolean equations in the manner described earlier.

```
Y     =   A   *  /B   */Cin
      +  /A   *  /B   * Cin
      +   A   *   B   * Cin
      +  /A   *   B   */Cin

Cout  =   A   * Cin
      +   A   * B
      +   B   * Cin
```

Larger binary adders can be made by cascading these full adders. Each carry-out is directed to carry-in for the next stage. Such adders are known as Ripple Adders.

Combinatorial PAL devices are ideal for this purpose, since they also provide internal feedback. Thus, one strong consideration in such designs is the internal feedback capability of the device, in addition to other general device selection considerations.

These ripple adders have the advantage that they can be cascaded to any length. However, since the carry-out from the least significant bit has to travel all the way to the highest significant bit, which can take a long time, such large adders are inefficient. Adders with built in carry-look-ahead circuitry can save time by simultaneously generating the carry-in signal for all of the bits.

Rewriting the equations for the full adder from above gives:

```
Y0 =  A0:+: B0:+:Cin
```

where the carry-out signal is:

```
C0 =  A0  *  B0   + (A0   +B0) * Cin
```

Extending these equations for an n-bit carry-look-ahead adder, we can directly get the following equations:

```
Y0 =  A0:+:  B0 :+: Cin

Y1 =  A1:+:  B1 :+:  C0
```

where

```
C0 =  A0  *  B0
   +  (A0 + B0)   * Cin

Y2 =  A2 :+: B2 :+:  C1
```

where

```
C1 =  A1  *  B1
   +  (A1 + B1)   * (A0   * B0)
   +  (A1 + B1)   * (A0   +B0) * Cin

Y3 +  A3:+:  B3 :+:  C2
```

where

```
C2 =  A2  *  B2
   +  (A2 + B2)   * (A1   * B1)
   +  (A2 + B2)   * (A1   +B1) * (A0 * B0)
   +  (A2 + B2)   * (A1   +B1) * (A0+ B0)
      * Cin
```

In general,

```
Yn =  An:+:  Bn :+:Cn–1
and Cn–1  =An–1   *Bn–1
   +  (An–1   +Bn–1)  *(An–2 *  Bn–2)
   +  ...
   +  ...
   +  (An–1   +Bn–1)  * ...   *    (A0
   +  B0) * Cin
```

and finally the carry-out is:

```
Cn =  An  *  Bn
   +  (An + Bn)    *(An–1 *Bn–1)
   +  (An + Bn)    *(An–1 +Bn–1) *
      (An–2* Bn–2)

   +  ...
   +  ...
   +  (An + Bn)    * ...   *(A0 + B0) * Cin
```

These equations are essentially a combination of the traditional generate and propagate logic for ALU design.

Adder Device Selection Considerations

The number of product terms required for implementing a carry-look-ahead adder is enormous. The carry-out function alone for a four-bit carry-look-ahead adder requires over 36 product terms in the sum-of-products form. For a single-level AND-OR implementation the number of product terms required for the most significant bit Y3 is 28.

A logic trick lies in the bit-pair decoding function. All of the bits of the first operand in the registers (A) and the second operand at the inputs (B) are bit-pair decoded. As illustrated in Figure 6, the results of this bit-pair decoding are A + B, A + /B, /A + B, and /A + /B. These outputs are then fed to the AND array as inputs.

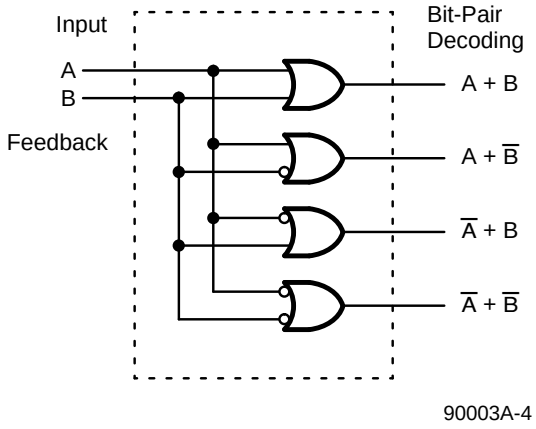


Figure 6. Bit-Pair-Decoding Function

Sixteen AND combinations of these four inputs can then be formed on every product term of the AND-OR array. These are shown in Figure 7, and include the standard true and complements of both the bits as well as XOR< XNOR and various other combinations. This bit pair decoding essentially provides an extra two-level AND-OR logic level before the AND-OR array. The cost as well as extra propagation delay of the extra logic level is minimal, since the array size does not increase.

The equations for the adder can obviously benefit from multi-level logic. The bit-pair decoding can be used to implement the first two levels of logic. The next level of logic can be implemented in the standard AND-OR array. Every product term of the AND-OR array can combine one of the sixteen possible functions of different inputs/feedbacks of the device.

The product terms are then combined together through an OR gate to implement the CARRY-OUT function, shown in Figure 8. For adder outputs Y0, Y1, Y2, and Y3, the product terms are combined through an XOR gate, as shown in Figure 9.

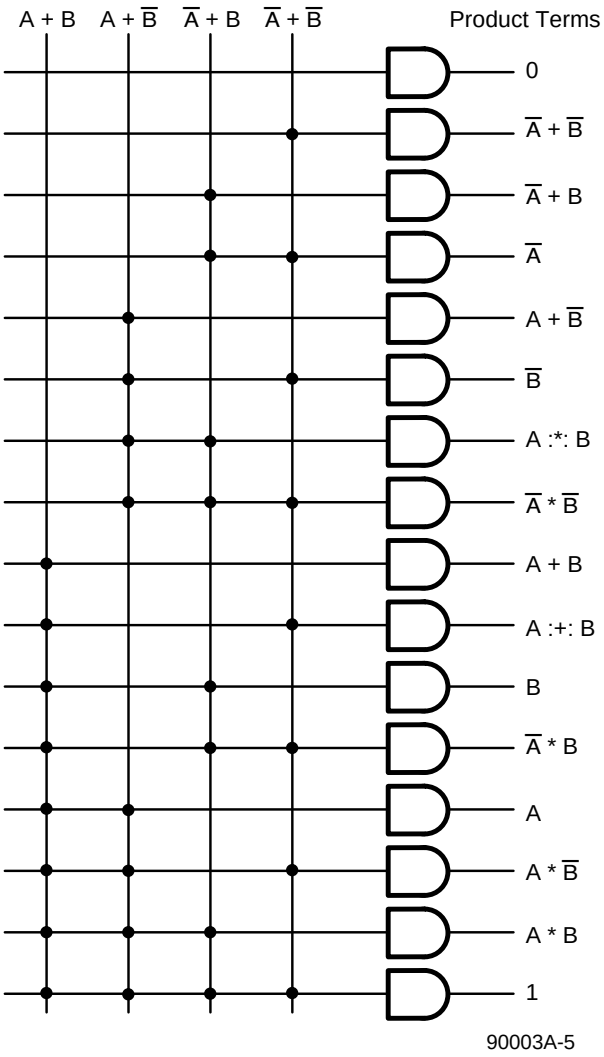
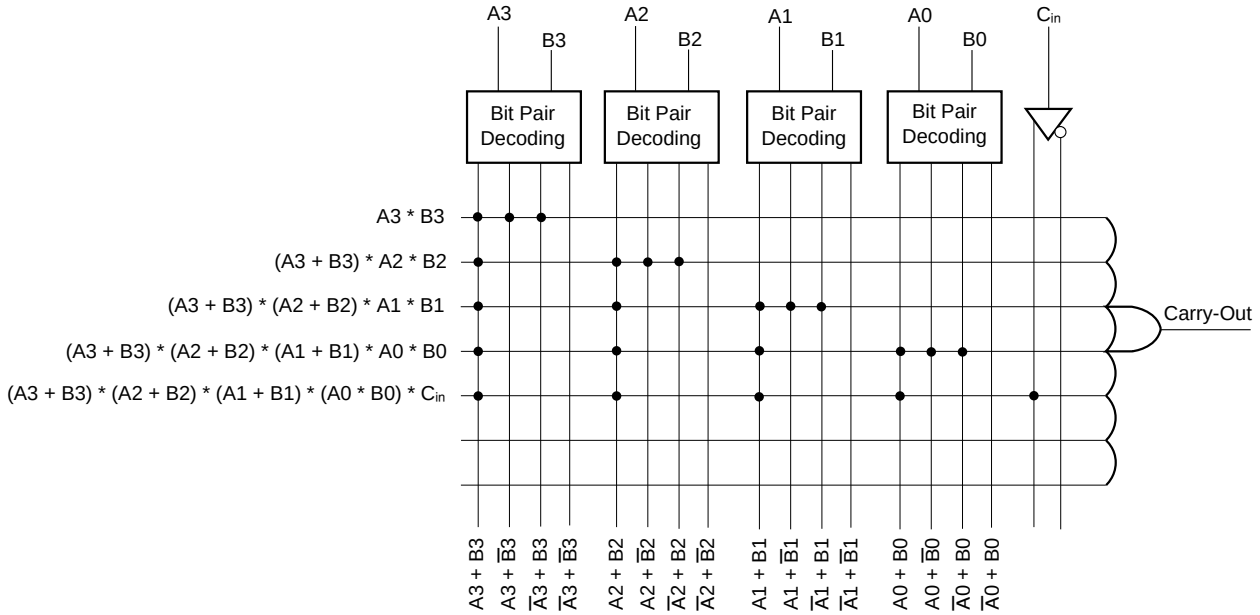


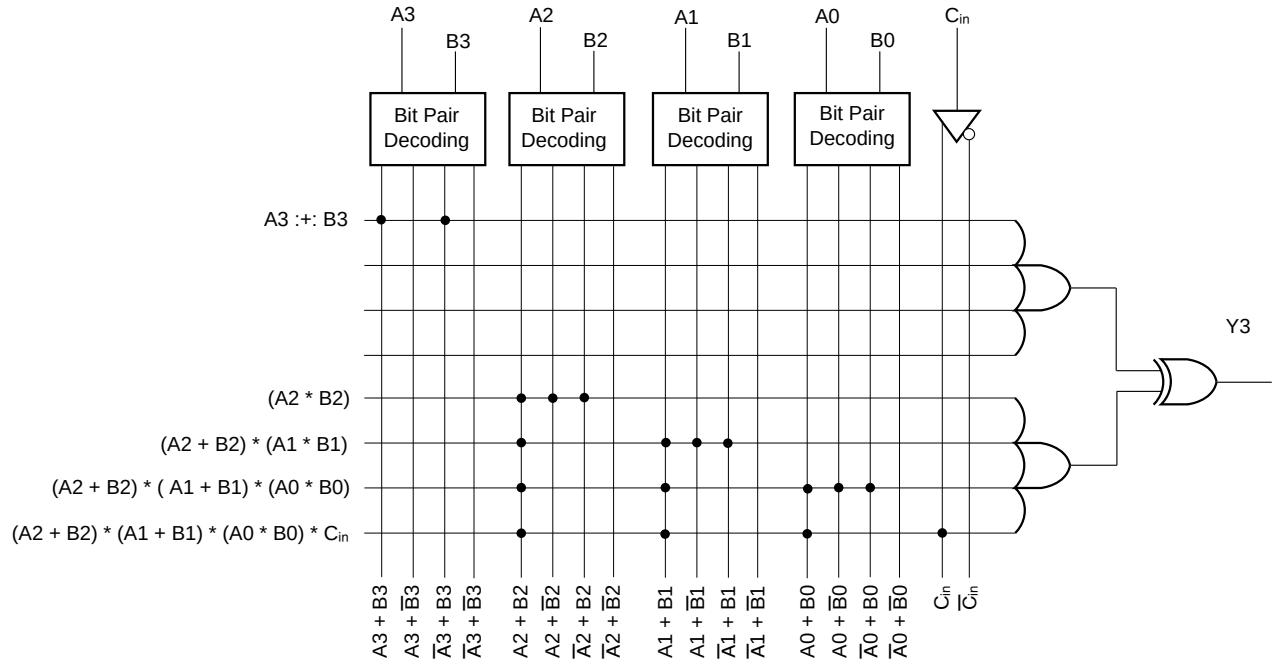
Figure 7. Sixteen Possible Input Logic Combinations



90003A-6

Carry-Out = A3 * B3
(A3 + B3) * A2 * B2
(A3 + B3) * (A2 + B2) * A1 * B1
(A3 + B3) * (A2 + B2) * (A1 + B1) * A0 * B0
(A3 + B3) * (A2 + B2) * (A1 + B1) * (A0 * B0) * C_{in}

Figure 8. Implementation of CARRY-OUT Function



90003A-7

Y3 = A3 :+: B3
:+: (A2 * B2)
+(A2 + B2) * (A1 * B1)
+(A2 + B2) * (A1 + B1) * (A0 * B0)
+(A2 + B2) * (A1 + B1) * (A0 * B0) * C_{in})

Figure 9. Relationship Between Adder Boolean Equation and Device Logic

Latches

PAL devices are often used to implement latches. One of the most common uses for a latch is as a temporary storage for data or addresses. PLD-based latches are often used in address decoders to assert the decoded signal for long durations. These latches are also very useful for asynchronous digital designs, and are used often for control and arbitration functions.

A latch is essentially a simple combinatorial circuit in which the output is a function of inputs and feedback. The most commonly used latch is the D-type latch. When the control signal latch-enable (LEN) is HIGH, the latch is in the "transparent mode" and the input signal /D is available at the outputs. When the LEN signal is LOW, the input data is latched on the outputs and is retained until LEN goes back HIGH. In a typical address decoder, the input will be a combination of various address signals, decoded as explained earlier for range comparators. The latching signal in most microprocessors is called AS (address strobe) or ALE (address latch enable).

The truth table for a latch can be derived directly from this functional description, and is shown in Table 7.

Table 7. Truth Table of a Simple Latch

Inputs		Outputs
/D	LEN	/Q
0	1	0
1	1	1
X	0	/Q (previous)

The Boolean equations for this latch can be directly derived from the truth table:

```
/Q = /D * LEN
   + /Q * /LEN
```

The logic implementation for this latch is shown in Figure 10.

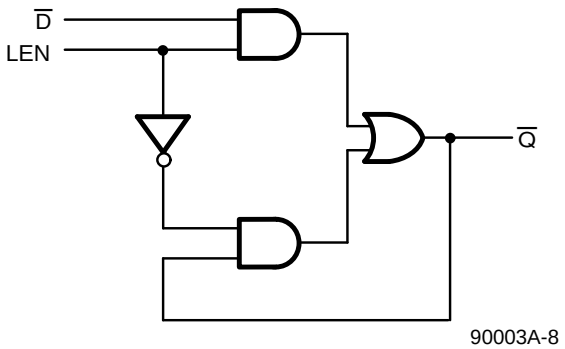


Figure 10. A Transparent Latch

Hazards

Even when a combinatorial circuit has been designed correctly, it may still have erroneous outputs due to "hazards." Hazards exist because physical circuits do not behave ideally. Combinatorial complementary output functions based on the same inputs are prime candidates for such hazards. As the input changes, the two outputs will not respond simultaneously. Although this will not change the steady-state output of the circuit, it may cause a spurious pulse or a "glitch." Such hazards are even more dangerous in latches, where the glitch can cause incorrect data to be latched.

There are two types of hazards, static and dynamic. Static hazards occur when the steady-state output of combinatorial logic is not supposed to change due to an input transition, but a momentary change does occur. Such a glitch can be further classified as a static 1 or a static 0 hazard as shown in Figure 11.

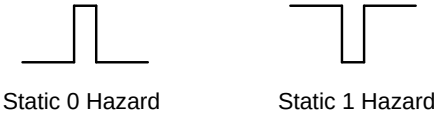


Figure 11. Static Hazards

Dynamic hazards involve situations where the steady-state output is supposed to change due to an input transition. The hazard occurs when the transient output changes several times before settling. Figure 12 shows dynamic hazards.

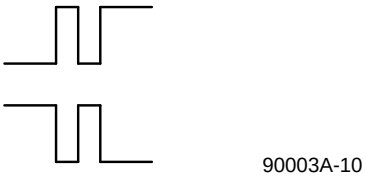


Figure 12. Dynamic Hazards

A Karnaugh map is a very good way of detecting hazard conditions. When trying to detect a static 0 or static 1 hazard, only the mapping of the zeros and the ones, respectively, are required. For example, the latch equations in Figure 10 can be mapped to a Karnaugh map shown in Figure 13. The relationship between the Karnaugh mapping and the Boolean equation product terms is also illustrated.

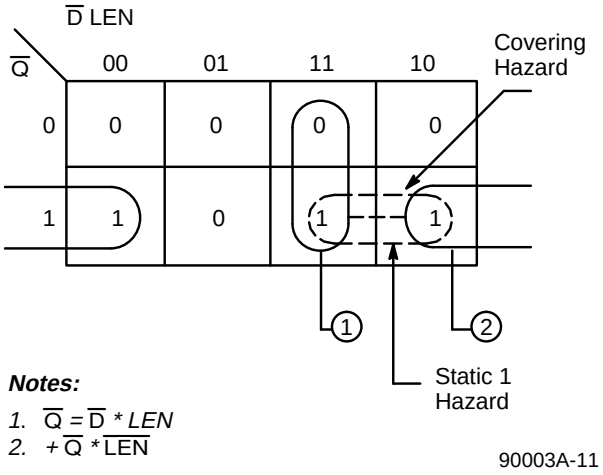


Figure 13. Karnaugh Map for Transparent Latch Design

The possibility of a hazard exists when the signal LEN changes. Initially, when D and LEN are HIGH, the output Q is also HIGH. When LEN switches to LOW, it is possible for the output to go LOW momentarily. This is because when LEN goes LOW the first product term is disabled, and to maintain the output HIGH the second product term should be enabled exactly at the same instant. Due to the uneven gate delays or routing conditions on board, these two events will not take place simultaneously. This is a static 1 hazard. It can also be identified directly in the Karnaugh map by the two adjacent but disjoint sets of ones, grouped together to form a product term each.

The hazard conditions can be easily avoided in the PLDs by providing an extra cover product term. This product term is shown with a dotted line in the Karnaugh map. This third product term will keep the output asserted during the transition of the LEN signal, when the control changes from the first product term to the second. The modified Boolean equation is shown below.

```
/Q = /D * LEN
   + /Q * /LEN
   + /D * /Q    (Cover product term)
```

Devices on which latches are implemented need to provide output feedback. All devices with I/O pins provide this necessary feedback. The only other consideration for selecting a device would be the provision of sufficient number of product terms for addressing the needs of glitch-free and testable design.