5.4.6 Decoders in ABEL and PLDs

Nothing in logic design is much easier than writing the PLD equations for a decoder. Since the logic expression for each output is typically just a single product term, decoders are very easily targeted to PLDs and use few product-term resources.

For example, Table 5-8 is an ABEL program for a 74x138-like 3-to-8 binary decoder as realized in a PAL16L8. Note that some of the input pins and all of the output pins have active-low names ("_L" suffix) in the pin declarations, corresponding to the logic diagram in Figure 5-37 on page 320. However, the program also defines a corresponding active-high name for each signal so that the equations can all be written "naturally," in terms of active-high signals. An alternate way to achieve the same effect is described in the box on page 325.

Table 5-8 An ABEL program for a 74x138-like 3-to-8 binary decoder.

```
module Z74X138
title '74x138 Decoder PLD
J. Wakerly, Stanford University'
Z74X138 device 'P16L8';

" Input and output pins
A, B, C, G2A_L, G2B_L, G1                       pin 1, 2, 3, 4, 5, 6;
Y0_L, Y1_L, Y2_L, Y3_L, Y4_L, Y5_L, Y6_L, Y7_L  pin 19..12 istype 'com';

" Active-high signal names for readability
G2A = !G2A_L;
G2B = !G2B_L;
Y0 = !Y0_L;
Y1 = !Y1_L;
Y2 = !Y2_L;
Y3 = !Y3_L;
Y4 = !Y4_L;
Y5 = !Y5_L;
Y6 = !Y6_L;
Y7 = !Y7_L;

" Constant expression
ENB = G1 & G2A & G2B;

equations
Y0 = ENB & !C & !B & !A;
Y1 = ENB & !C & !B &  A;
Y2 = ENB & !C &  B & !A;
Y3 = ENB & !C &  B &  A;
Y4 = ENB &  C & !B & !A;
Y5 = ENB &  C & !B &  A;
Y6 = ENB &  C &  B & !A;
Y7 = ENB &  C &  B &  A;

end Z74X138
```

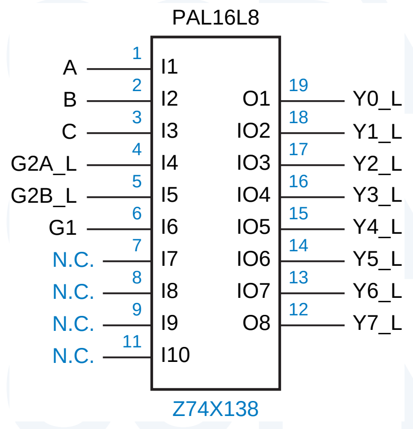


Figure 5-40
Logic diagram for the PAL16L8 used as a 74x138 decoder.

Also note that the ABEL program defines a constant expression for `ENB`. Here, `ENB` is not an input or output signal, but merely a user-defined name. In the `equations` section, the compiler substitutes the expression `(G1 & G2A & G2B)` everywhere that "`ENB`" appears. Assigning the constant expression to a user-defined name improves this program's readability and maintainability.

If all you needed was a '138, you'd be better off using a real '138 than a more expensive PLD. However, if you need nonstandard functionality, then the PLD can usually achieve it much more cheaply and easily than an MSI/SSI-based solution. For example, if you need the functionality of a '138 but with active-high outputs, you need only to change one line in the pin declarations of Table 5-8:

```
Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7          pin 19..12 istype 'com';
```

(Also, the original definitions of `Y0`–`Y7` in Table 5-8 must be deleted.) Since each of the equations required a single product of six variables (including the three in the `ENB` expression), each complemented equation requires a *sum* of six product terms, less than the seven available in a PAL16L8. If you use a PAL16V8 or other device with output polarity selection, then the compiler can select non-inverted output polarity to use only one product term per output.

Table 5-9 Alternate declarations for a 74x138-like 3-to-8 binary decoder.

```
" Input and output pins
A, B, C, !G2A, !G2B, G1                  pin 1, 2, 3, 4, 5, 6;
!Y0, !Y1, !Y2, !Y3, !Y4, !Y5, !Y6, !Y7   pin 19..12 istype 'com';

" Constant expression
ENB = G1 & G2A & G2B;
```

ACTIVE-LOW PIN DEFINITIONS

ABEL allows you to use an inversion prefix (!) on signal names in the pin definitions of a program. When a pin name is defined with the inversion prefix, the compiler automatically prepends an inversion prefix to the signal name anywhere it appears elsewhere in the program. If it's already inverted, this results in a double inversion.

This feature can be used to define a different but consistent convention for defining active-low inputs and outputs—give each active-low signal an active-*high* name, but precede it with the inversion prefix in its pin definition. For the 3-to-8 decoder in Table 5-8, we replace the first part of the program with the code shown in Table 5-9; the equations section of the program stays exactly the same.

Which convention to use may be a matter of personal taste, but it can also depend on the capabilities of the CAD tools that you use to draw schematics. Many tools allow you to automatically create a schematic symbol from a logic block that is defined by an ABEL program. If the tool allows you to place inversion bubbles on selected inputs and outputs of the symbol, then the convention in Table 5-9 yields a symbol with active-high signal names inside the function outline. You can then specify external bubbles on the active-low signals to obtain a symbol that matches the conventions described in Section 5.4.2 and shown in Figure 5-41(a).

On the other hand, you may not be able or want to provide inversion bubbles on CAD-created symbols. In that case, you should use the convention in Table 5-8; this yields a CAD-created symbol in which the active level is indicated by the signal name inside the function outline; no external inversion bubbles are needed. This is shown in Figure 5-41(b). Note that unlike Figure 5-36(a) on page 318, we use a text-based convention (_L) rather than an overbar on the signal name to indicate active level. A properly chosen text-based convention provides portability among different CAD tools.

We'll somewhat arbitrarily select one convention or the other in each of the ABEL examples in the rest of this book, just to help you get comfortable with both approaches.

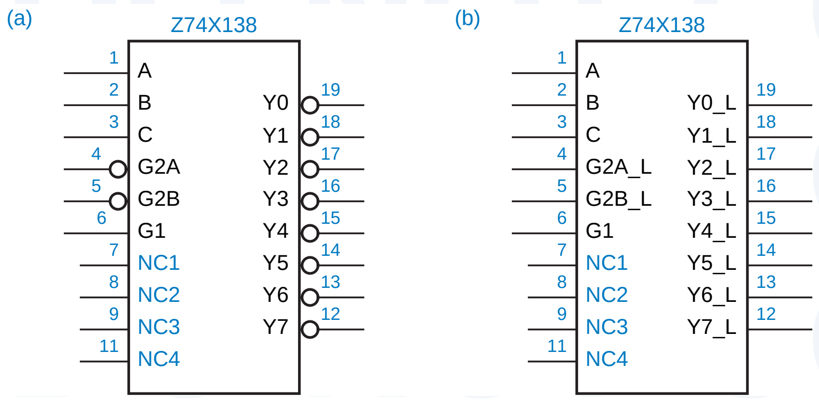


Figure 5-41 Possible CAD-created symbols for the PLD-based, 74x138-like decoder: (a) based on Table 5-9, after manual insertion of inversion bubbles; (b) based on Table 5-8.

Another easy change is to provide alternate enable inputs that are ORed with the main enable inputs. To do this, you need only define additional pins and modify the definition of ENB:

```
EN1, EN2_L                        pin 7, 8;
...
EN2 = !EN2_L;
...
ENB = G1 & G2A & G2B # EN1 # EN2;
```

This change expands the number of product terms per output to three, each having a form similar to

```
Y0 = G1 & G2A & G2B & !C & !B &!A
   # EN1 & !C & !B & !A
   # EN2 & !C & !B & !A;
```

(Remember that the PAL16L8 has a fixed inverter and the PAL16V8 has a selectable inverter between the AND-OR array and the output of the PLD, so the actual output is active low as desired.)

If you add the extra enables to the version of the program with active-high outputs, then the PLD must realize the complement of the sum-of-products expression above. It's not immediately obvious how many product terms this expression will have, and whether it will fit in a PAL16L8, but we can use the ABEL compiler to get the answer for us:

```
!Y0 = C # B # A # !G2B & !EN1 & !EN2
                # !G2A & !EN1 & !EN2
                # !G1  & !EN1 & !EN2;
```

The expression has a total of six product terms, so it fits in a PAL16L8.

As a final tweak, we can add an input to dynamically control whether the output is active high or active low, and modify all of the equations as follows:

```
POL                 pin 9;
...
Y0 = POL $ (ENB & !C & !B & !A);
Y1 = POL $ (ENB & !C & !B &  A);
...
Y7 = POL $ (ENB &  C &  B &  A);
```

As a result of the XOR operation, the number of product terms needed per output increases to 9, in either output-pin polarity. Thus, even a PAL16V8 cannot implement the function as written.

helper output

The function can still be realized if we create a *helper output* to reduce the product term explosion. As shown in Table 5-10, we allocate an output pin for

Table 5-10 ABEL program fragment showing two-pass logic.

```
...
" Output pins
Y0_L, Y1_L, Y2_L, Y3_L         pin 19, 18, 17, 16 istype 'com';
Y4_L, Y5_L, Y6_L, ENB          pin 15, 14, 13, 12 istype 'com';

equations
ENB = G1 & G2A & G2B # EN1 # EN2;
Y0 = POL $ (ENB & !C & !B & !A);
...
```

the ENB expression (losing the Y7_L output) , and move the ENB equation into the equations section of the program. This reduces the product-term requirement to five in either polarity.

helper output

Besides sacrificing a pin for the helper output, this realization has the disadvantage of being slower. Any changes in the inputs to the helper expression must propagate through the PLD twice before reaching the final output. This is called *two-pass logic*. Many PLD and FPGA synthesis tools can automatically generate logic with two or more passes if a required expression cannot be realized in just one pass through the logic array.

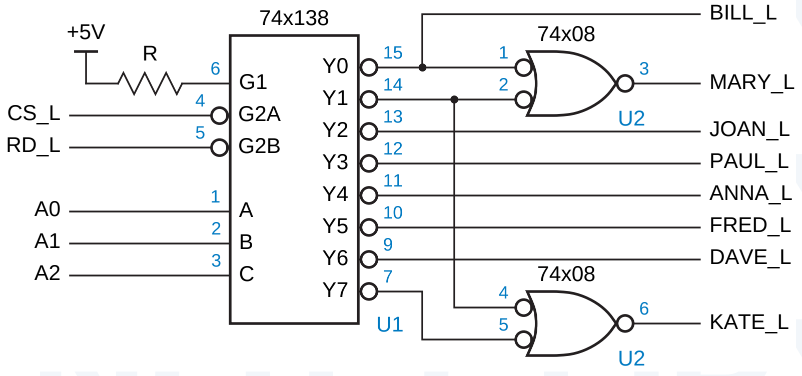
two-pass logic

Decoders can be customized in other ways. A common customization is for a single output to decode more than one input combination. For example, suppose you needed to generate a set of enable signals according to Table 5-11. A 74x138 MSI decoder can be augmented as shown in Figure 5-42 to perform the required function. This approach, while potentially less expensive than a PLD, has the disadvantages that it requires extra components and delay to create the required outputs, and it is not easily modified.

CS_L	RD_L	A2	A1	A0	***Output(s) to Assert***
1	x	x	x	x	none
x	1	x	x	x	none
0	0	0	0	0	BILL_L, MARY_L
0	0	0	0	1	MARY_L, KATE_L
0	0	0	1	0	JOAN_L
0	0	0	1	1	PAUL_L
0	0	1	0	0	ANNA_L
0	0	1	0	1	FRED_L
0	0	1	1	0	DAVE_L
0	0	1	1	1	KATE_L

Table 5-11 Truth table for a customized decoder function.

Figure 5-42 Customized decoder circuit.



A PLD solution to the same problem is shown in Table 5-12. Note that this program uses the pin active-low pin-naming convention described in the box on page 325 (you should be comfortable with either convention). Each of the last six equations uses a single AND gate in the PLD. The ABEL compiler will also minimize the MARY equation to use just one AND gate. Active-high output signals could be obtained just by changing two lines in the declaration section:

```
BILL, MARY, JOAN, PAUL       pin 19, 18, 17, 16 istype ’com’;
ANNA, FRED, DAVE, KATE       pin 15, 14, 13, 12 istype ’com’;
```

Table 5-12 ABEL equations for a customized decoder.

```
module CUSTMDEC
title 'Customized Decoder PLD
J. Wakerly, Stanford University'
CUSTMDEC device ’P16L8’;

" Input pins
!CS, !RD, A0, A1, A2         pin 1, 2, 3, 4, 5;
" Output pins
!BILL, !MARY, !JOAN, !PAUL   pin 19, 18, 17, 16 istype 'com';
!ANNA, !FRED, !DAVE, !KATE   pin 15, 14, 13, 12 istype 'com';

equations
BILL = CS & RD & (!A2 & !A1 & !A0);
MARY = CS & RD & (!A2 & !A1 & !A0 # !A2 & !A1 &  A0);
KATE = CS & RD & (!A2 & !A1 &  A0 #  A2 &  A1 &  A0);
JOAN = CS & RD & (!A2 &  A1 & !A0);
PAUL = CS & RD & (!A2 &  A1 &  A0);
ANNA = CS & RD & ( A2 & !A1 & !A0);
FRED = CS & RD & ( A2 & !A1 &  A0);
DAVE = CS & RD & ( A2 &  A1 & !A0);

end CUSTMDEC
```

Another way of writing the equations is shown in Table 5-13. In most applications, this style is more clear, especially if the select inputs have numeric significance.

Table 5-13 Equivalent ABEL equations for a customized decoder.

```
ADDR = [A2,A1,A0];

equations
BILL = CS & RD & (ADDR == 0);
MARY = CS & RD & (ADDR == 0) # (ADDR == 1);
KATE = CS & RD & (ADDR == 1) # (ADDR == 7);
JOAN = CS & RD & (ADDR == 2);
PAUL = CS & RD & (ADDR == 3);
ANNA = CS & RD & (ADDR == 4);
FRED = CS & RD & (ADDR == 5);
DAVE = CS & RD & (ADDR == 6);
```

5.4.7 Decoders in VHDL

There are several ways to approach the design of decoders in VHDL. The most primitive approach would be to write a structural equivalent of a decoder logic circuit, as Table 5-14 does for the 2-to-4 binary decoder of Figure 5-32 on page 314. Of course, this mechanical conversion of an existing design into the equivalent of a netlist defeats the purpose of using VHDL in the first place.

Instead, we would like to write a program that uses VHDL to make our decoder design more understandable and maintainable. Table 5-15 shows one approach to writing code for a 3-to-8 binary decoder equivalent to the 74x138, using the dataflow style of VHDL. The address inputs `A(2 downto 0)` and the active-low decoded outputs `Y_L(0 to 7)` are declared using vectors to improve readability. A `select` statement enumerates the eight decoding cases and assigns the appropriate active-low output pattern to an 8-bit internal signal `Y_L_i`. This value is assigned to the actual circuit output `Y_L` only if all of the enable inputs are asserted.

This design is a good start, and it works, but it does have a potential pitfall. The adjustments that handle the fact that two inputs and all the outputs are active-low happen to be buried in the final assignment statement. While it’s true that most VHDL programs are written almost entirely with active-high signals, if we’re defining a device with active-low external pins, we really should handle them in a more systematic and easily maintainable way.

Table 5-14 VHDL structural program for the decoder in Figure 5-32.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V2to4dec is
  port (I0, I1, EN: in STD_LOGIC;
        Y0, Y1, Y2, Y3: out STD_LOGIC );
end V2to4dec;

architecture V2to4dec_s of V2to4dec is
  signal NOTI0, NOTI1: STD_LOGIC;
  component inv port (I: in STD_LOGIC; O: out STD_LOGIC ); end component;
  component and3 port (I0, I1, I2: in STD_LOGIC; O: out STD_LOGIC ); end component;
begin
  U1: inv port map (I0,NOTI0);
  U2: inv port map (I1,NOTI1);
  U3: and3 port map (NOTI0,NOTI1,EN,Y0);
  U4: and3 port map (   I0,NOTI1,EN,Y1);
  U5: and3 port map (NOTI0,   I1,EN,Y2);
  U6: and3 port map (   I0,   I1,EN,Y3);
end V2to4dec_s;
```

Table 5-15 Dataflow-style VHDL program for a 74x138-like 3-to-8 binary decoder.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V74x138 is
    port (G1, G2A_L, G2B_L: in STD_LOGIC;      -- enable inputs
          A: in STD_LOGIC_VECTOR (2 downto 0);  -- select inputs
          Y_L: out STD_LOGIC_VECTOR (0 to 7) ); -- decoded outputs
 end V74x138;

architecture V74x138_a of V74x138 is
  signal Y_L_i: STD_LOGIC_VECTOR (0 to 7);
begin
  with A select Y_L_i <=
    "01111111" when "000",
    "10111111" when "001",
    "11011111" when "010",
    "11101111" when "011",
    "11110111" when "100",
    "11111011" when "101",
    "11111101" when "110",
    "11111110" when "111",
    "11111111" when others;
  Y_L <= Y_L_i when (G1 and not G2A_L and not G2B_L)='1' else "11111111";
end V74x138_a;
```

Table 5-16 VHDL architecture with a maintainable approach to active-level handling.

```
architecture V74x138_b of V74x138 is
  signal G2A, G2B: STD_LOGIC;             -- active-high version of inputs
  signal Y: STD_LOGIC_VECTOR (0 to 7);    -- active-high version of outputs
  signal Y_s: STD_LOGIC_VECTOR (0 to 7);  -- internal signal
begin
  G2A <= not G2A_L; -- convert inputs
  G2B <= not G2B_L; -- convert inputs
  Y_L <= Y;         -- convert outputs
  with A select Y_s <=
    "10000000" when "000",
    "01000000" when "001",
    "00100000" when "010",
    "00010000" when "011",
    "00001000" when "100",
    "00000100" when "101",
    "00000010" when "110",
    "00000001" when "111",
    "00000000" when others;
  Y <= not Y_s when (G1 and G2A and G2B)='1' else "00000000";
end V74x138_b;
```

Table 5-16 shows such an approach. No changes are made to the `entity` declarations. However, active-high versions of the active-low external pins are defined within the `V74x138_a` architecture, and explicit assignment statements are used to convert between the active-high and active-low signals. The decoder function itself is defined in terms of only the active-high signals, probably the biggest advantage of this approach. Another advantage is that the design can be easily modified in just a few well-defined places if changes are required in the external active levels.

OUT-OF-ORDER EXECUTION

In Table 5-16, we've grouped all three of the active-level conversion statements together at the beginning of program, even a value isn't assigned to `Y_L` until *after* a value is assigned to `Y`, later in the program. Remember that this is OK because the assignment statements in the architecture body are executed "concurrently." That is, an assignment to any signal causes all the other statements that use that signal to be re-evaluated, regardless of their position in the architecture body.

You could put the "`Y_L <= Y`" statement at the end of the body if its current position bothers you, but the program is a bit more maintainable in its present form, with all the active-level conversions together.

Table 5-17 Hierarchical definition of 74x138-like decoder with active-level handling.

```
architecture V74x138_c of V74x138 is
  signal G2A, G2B: STD_LOGIC;          -- active-high version of inputs
  signal Y: STD_LOGIC_VECTOR (0 to 7);  -- active-high version of outputs
  component V3to8dec port (G1, G2, G3: in STD_LOGIC;
                           A: in STD_LOGIC_VECTOR (2 downto 0);
                           Y: out STD_LOGIC_VECTOR (0 to 7) ); end component;
begin
  G2A <= not G2A_L;   -- convert inputs
  G2B <= not G2B_L;   -- convert inputs
  Y_L <= not Y;       -- convert outputs
  U1: V3to8dec port map (G1, G2A, G2B, A, Y);
end V74x138_c;
```

Active levels can be handled in an even more structured way. As shown in Table 5-17, the V74x138 architecture can be defined hierarchically, using a fully active-high V3to8dec component that has its own dataflow-style definition in Table 5-18. Once again, no changes are required in the top-level definition of the V74x138 entity. Figure 5-43 shows the relationship between the entities.

Still another approach to decoder design is shown in Table 5-19, which can replace the V3to8dec_a architecture of Table 5-18. Instead of concurrent state-

Table 5-18 Dataflow definition of an active-high 3-to-8 decoder.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V3to8dec is
    port (G1, G2, G3: in STD_LOGIC;
          A: in STD_LOGIC_VECTOR (2 downto 0);
          Y: out STD_LOGIC_VECTOR (0 to 7) );
end V3to8dec;

architecture V3to8dec_a of V3to8dec is
  signal Y_s: STD_LOGIC_VECTOR (0 to 7);
begin
    with A select Y_s <=
      "10000000" when "000",
      "01000000" when "001",
      "00100000" when "010",
      "00010000" when "011",
      "00001000" when "100",
      "00000100" when "101",
      "00000010" when "110",
      "00000001" when "111",
      "00000000" when others;
    Y <= Y_s when (G1 and G2 and G3)='1'
             else "00000000";
end V3to8dec_a;
```

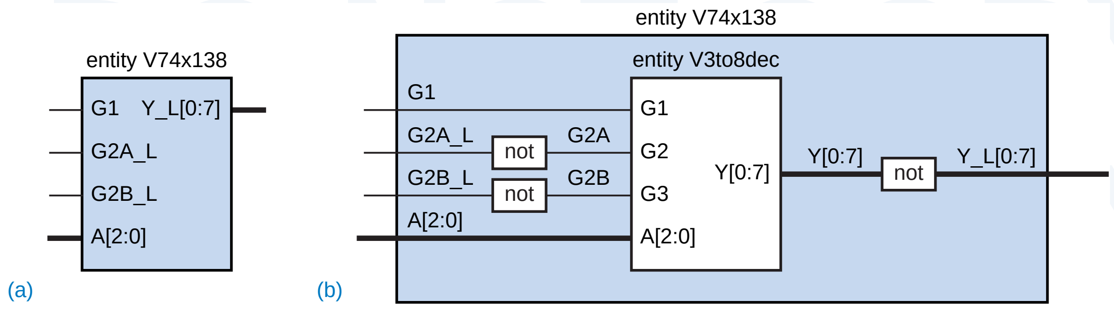


Figure 5-43 VHDL entity V74x138: (a) top level; (b) internal structure using architecture V74x138_c.

NAME MATCHING In Figure 5-43, the port names of an entity are drawn inside the corresponding box. The names of the signals that are connected to the ports when the entity is used are drawn on the signal lines. Notice that the signal names may match, but they don't have to. The VHDL compiler keeps everything straight, associating a scope with each name. The situation is completely analogous to the way variable and parameter names are handled in structured, procedural programming languages like C.

ments, this architecture uses a process and sequential statements to define the decoder's operation in a behavioral style. However, a close comparison of the two architectures shows that they're really not that different except for syntax.

Table 5-19 Behavioral-style architecture definition for a 3-to-8 decoder.

```
architecture V3to8dec_b of V3to8dec is
  signal Y_s: STD_LOGIC_VECTOR (0 to 7);
begin
process(A, G1, G2, G3, Y_s)
  begin
    case A is
      when "000" => Y_s <= "10000000";
      when "001" => Y_s <= "01000000";
      when "010" => Y_s <= "00100000";
      when "011" => Y_s <= "00010000";
      when "100" => Y_s <= "00001000";
      when "101" => Y_s <= "00000100";
      when "110" => Y_s <= "00000010";
      when "111" => Y_s <= "00000001";
      when others => Y_s <= "00000000";
    end case;
    if (G1 and G2 and G3)='1' then Y <= Y_s;
    else Y <= "00000000";
    end if;
  end process;
end V3to8dec_b;
```

Table 5-20 Truly behavioral architecture definition for a 3-to-8 decoder.

```
architecture V3to8dec_c of V3to8dec is
begin
process (G1, G2, G3, A)
  variable i: INTEGER range 0 to 7;
  begin
    Y <= "00000000";
    if (G1 and G2 and G3) = '1' then
      for i in 0 to 7 loop
        if i=CONV_INTEGER(A) then Y(i) <= '1'; end if;
      end loop;
    end if;
  end process;
end V3to8dec_c;
```

As a final example, a more truly behavioral, process-based architecture for the 3-to-8 decoder is shown in Table 5-20. (Recall that the CONV_INTEGER function was defined in \secref{VHDLconv}.) Of the examples we've given, this is the only one that describes the decoder function without essentially embedding a truth table in the VHDL program. In that respect, it is more flexible because it can be easily adapted to make a binary decoder of any size. In another respect, it is less flexible in that it does not have a truth table that can be easily modified to make custom decoders like the one we specified in Table 5-11 on page 327.

*5.4.8 Seven-Segment Decoders

seven-segment display

Look at your wrist and you'll probably see a *seven-segment display*. This type of display, which normally uses light-emitting diodes (LEDs) or liquid-crystal display (LCD) elements, is used in watches, calculators, and instruments to display decimal data. A digit is displayed by illuminating a subset of the seven line segments shown in Figure 5-44(a).

seven-segment decoder

74x49

A *seven-segment decoder* has 4-bit BCD as its input code and the "seven-segment code," which is graphically depicted in Figure 5-44(b), as its output code. Figure 5-45 and Table 5-21 are the logic diagram truth table and for a *74x49* seven-segment decoder. Except for the strange (clever?) connection of the "blanking input" BI_L, each output of the 74x49 is a minimal product-of-sums

Figure 5-44 Seven-segment display: (a) segment identification; (b) decimal digits.