Table 6-35 Structural VHDL program for a full adder.

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

entity FA is
    port ( A, B, CI: in  STD_LOGIC;
           S, CO:    out STD_LOGIC );
end FA;

architecture FA_arch of FA is
begin
  S <= A xor B xor CI;
  CO <= (A and B) or (A and CI) or (B and CI);
end FA_arch;
```

6.3.7 Tic-Tac-Toe

Our last example is the design of a combinational circuit that picks a player's next move in the game of Tic-Tac-Toe. In case you're not familiar with the game, the rules are explained in the box on page 488. We'll repeat here our strategy for playing and winning the game:

1. Look for a row, column, or diagonal that has two of my marks (X or O, depending on which player I am) and one empty cell. If one exists, place my mark in the empty cell; I win!
2. Else, look for a row, column, or diagonal that has two of my opponent's marks and one empty cell. If one exists, place my mark in the empty cell to block a potential win by my opponent.
3. Else, pick a cell based on experience. For example, if the middle cell is open, it's usually a good bet to take it. Otherwise, the corner cells are good bets. Intelligent players can also notice and block a developing pattern by the opponent or "look ahead" to pick a good move.

To avoid confusion between "O" and "0" in our programs, we'll call the second player "Y". Now we can think about how to encode the inputs and outputs of the circuit. The inputs represent the current state of the playing grid. There are nine cells, and each cell has one of three possible states (empty, occupied by X, occupied by Y). The outputs represent the move to make, assuming that it is X's turn. There are only nine possible moves that a player can make, so the output can be encoded in just four bits.

There are several choices of how to code the state of one cell. Because the game is symmetric, we choose a symmetric encoding that can help us later:

00 Cell is empty.

10 Cell is occupied by X.

01 Cell is occupied by Y.

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

package TTTdefs is

type TTTgrid is array (1 to 3) of STD_LOGIC_VECTOR(1 to 3);
subtype TTTmove is STD_LOGIC_VECTOR (3 downto 0);

constant MOVE11: TTTmove := "1000";
constant MOVE12: TTTmove := "0100";
constant MOVE13: TTTmove := "0010";
constant MOVE21: TTTmove := "0001";
constant MOVE22: TTTmove := "1100";
constant MOVE23: TTTmove := "0111";
constant MOVE31: TTTmove := "1011";
constant MOVE32: TTTmove := "1101";
constant MOVE33: TTTmove := "1110";
constant NONE:   TTTmove := "0000";

end TTTdefs;
```

Table 6-36 VHDL package with definitions for the Tic-Tac-Toe project.

So, we can encode the 3×3 grid's state in 18 bits. Since VHDL supports arrays, it is useful to define an array type, `TTTgrid`, that contains elements corresponding to the cells in the grid. Since this type will be used throughout our Tic-Tac-Toe project, it is convenient to put this definition, along with several others that we'll come to, in a VHDL package, as shown in Table 6-36.

It would be natural to define `TTTgrid` as a two-dimensional array of `STD_LOGIC`, but not all VHDL tools support two-dimensional arrays. Instead, we define it as an array of 3-bit `STD_LOGIC_VECTOR`s, which is almost the same thing. To represent the Tic-Tac-Toe grid, we'll use two signals `X` and `Y` of this type, where an element of a variable is 1 if the like-named player has a mark in the corresponding cell. Figure 6-11 shows the correspondence between signal names and cells in the grid.

row \ column	1	2	3
1	X(1)(1) Y(1)(1)	X(1)(2) Y(1)(2)	X(1)(2) Y(1)(2)
2	X(2)(1) Y(2)(1)	X(2)(2) Y(2)(2)	X(2)(2) Y(2)(2)
3	X(3)(1) Y(3)(1)	X(3)(2) Y(3)(2)	X(3)(2) Y(3)(2)

Figure 6-16 Tic-Tac-Toe grid and VHDL signal names.

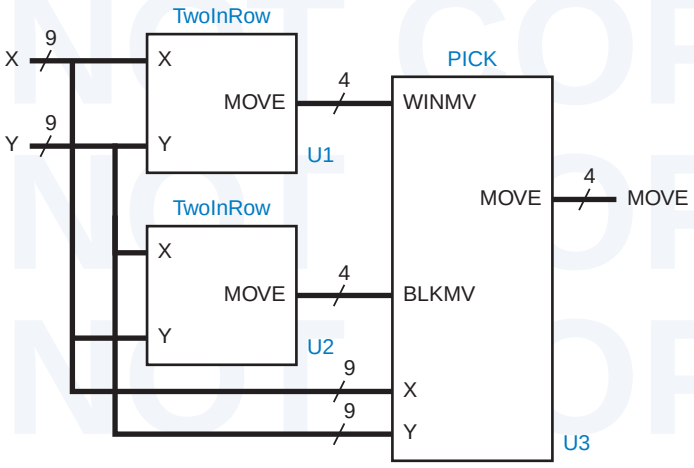

Figure 6-17
Entity partitioning for the Tic-Tac-Toe game.

The package in Table 6-36 also defines a 4-bit type `TTTmove` for encoded moves. A player has nine possible moves, and one more code is used for the case where no move is possible. This particular coding was chosen and used in the package for no other reason than that it's the same coding that was used in the ABEL version of this example in Section 6.2.7. By defining the coding in the package, we can easily change the definition later without having to change the entities that use it (for example, see Exercise 6.23).

Rather than try to design the Tic-Tac-Toe move-finding circuit as a single monolithic entity, it makes sense for us to try to partition it into smaller pieces. In fact, partitioning it along the lines of the three-step strategy at the beginning of this section seems like a good idea.

We note that steps 1 and 2 of our strategy are very similar; they differ only in reversing the roles of the player and the opponent. An entity that finds a winning move for me can also find a blocking move for my opponent. Looking at this characteristic from another point of view, an entity that finds a winning move for me can find a blocking move for me if the encodings for me and my opponent are swapped. Here's where our symmetric encoding pays off—we can swap players merely by swapping signals `X` and `Y`.

With this in mind, we can use two copies of the same entity, TwoInRow, to perform steps 1 and 2 as shown in Figure 6-17. Notice that signal X is connected to the top input of the first TwoInRow entity, but to the bottom input of the second. A third entity, `PICK`, picks a winning move if one is available from U1, else it picks a blocking move if available from U2, else it uses "experience" (step 3) to pick a move.

Table 6-37 is a structural VHDL program for the top-level entity, `GETMOVE`. In addition to the IEEE `std_logic_1164` package, it uses our `TTTdefs` package. Notice that the "`use`" clause for the `TTTdefs` packages specifies that it is stored in the "`work`" library, which is automatically created for our project.

Table 6-37 Top-level structural VHDL entity for picking a move in Tic-Tac-Toe.

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

entity GETMOVE is
    port ( X, Y: in  TTTgrid;
           MOVE: out TTTmove );
end GETMOVE;

architecture GETMOVE_arch of GETMOVE is

component TwoInRow port ( X, Y: in  TTTgrid;
                          MOVE: out STD_LOGIC_VECTOR(3 downto 0) );
end component;

component PICK port ( X, Y:         in  TTTgrid;
                      WINMV, BLKMV: in  STD_LOGIC_VECTOR(3 downto 0);
                      MOVE:         out STD_LOGIC_VECTOR(3 downto 0) );
end component;

signal WIN, BLK: STD_LOGIC_VECTOR(3 downto 0);

begin
  U1: TwoInRow port map (X, Y, WIN);
  U2: TwoInRow port map (Y, X, BLK);
  U3: PICK port map (X, Y, WIN, BLK, MOVE);
end GETMOVE_arch;
```

The architecture in Table 6-37 declares and uses just two components, `TwoInRow` and `PICK`, which will be defined shortly. The only internal signals are `WIN` and `BLK`, which pass winning and blocking moves from the two instances of `TwoInRow` to `PICK`, as in Figure 6-17. The statement part of the architecture has just three statements to instantiate the three blocks in the figure.

Now comes the interesting part, the design of the individual entities in Figure 6-17. We'll start with `TwoInRow` since it accounts for two-thirds of the design. Its entity definition is very simple, as shown in Table 6-38. But there's plenty to discuss about its architecture, shown in Table 6-39.

Table 6-38 Declaration of `TwoInRow` entity.

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

entity TwoInRow is
    port ( X, Y: in  TTTgrid;
           MOVE: out TTTmove );
end TwoInRow;
```

Table 6-39 Architecture of TwoInRow entity.

```
architecture TwoInRow_arch of TwoInRow is

function R(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;
  begin                          -- Find 2-in-row with empty cell i,j
    result := TRUE;
    for jj in 1 to 3 loop
      if jj = j then result := result and X(i)(jj)='0' and Y(i)(jj)='0';
      else result := result and X(i)(jj)='1'; end if;
    end loop;
    return result;
  end R;

function C(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;
  begin                          -- Find 2-in-column with empty cell i,j
    result := TRUE;
    for ii in 1 to 3 loop
      if ii = i then result := result and X(ii)(j)='0' and Y(ii)(j)='0';
      else result := result and X(ii)(j)='1'; end if;
    end loop;
    return result;
  end C;

function D(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;        -- Find 2-in-diagonal with empty cell i,j.
  begin                            -- This is for 11, 22, 33 diagonal.
    result := TRUE;
    for ii in 1 to 3 loop
      if ii = i then result := result and X(ii)(ii)='0' and Y(ii)(ii)='0';
      else result := result and X(ii)(ii)='1'; end if;
    end loop;
    return result;
  end D;

function E(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;        -- Find 2-in-diagonal with empty cell i,j.
  begin                            -- This is for 13, 22, 31 diagonal.
    result := TRUE;
    for ii in 1 to 3 loop
      if ii = i then result := result and X(ii)(4-ii)='0' and Y(ii)(4-ii)='0';
      else result := result and X(ii)(4-ii)='1'; end if;
    end loop;
    return result;
  end E;
```

Table 6-39 (continued)

```
begin
  process (X, Y)
  variable G11, G12, G13, G21, G22, G23, G31, G32, G33: BOOLEAN;
  begin
    G11 := R(X,Y,1,1) or C(X,Y,1,1) or D(X,Y,1,1);
    G12 := R(X,Y,1,2) or C(X,Y,1,2);
    G13 := R(X,Y,1,3) or C(X,Y,1,3) or E(X,Y,1,3);
    G21 := R(X,Y,2,1) or C(X,Y,2,1);
    G22 := R(X,Y,2,2) or C(X,Y,2,2) or D(X,Y,2,2) or E(X,Y,2,2);
    G23 := R(X,Y,2,3) or C(X,Y,2,3);
    G31 := R(X,Y,3,1) or C(X,Y,3,1) or E(X,Y,3,1);
    G32 := R(X,Y,3,2) or C(X,Y,3,2);
    G33 := R(X,Y,3,3) or C(X,Y,3,3) or D(X,Y,3,3);
    if    G11 then MOVE <= MOVE11;
    elsif G12 then MOVE <= MOVE12;
    elsif G13 then MOVE <= MOVE13;
    elsif G21 then MOVE <= MOVE21;
    elsif G22 then MOVE <= MOVE22;
    elsif G23 then MOVE <= MOVE23;
    elsif G31 then MOVE <= MOVE31;
    elsif G32 then MOVE <= MOVE32;
    elsif G33 then MOVE <= MOVE33;
    else           MOVE <= NONE;
    end if;
  end process;
end TwoInRow_arch;
```

The architecture defines several functions, each of which determines whether there is a winning move (from X's point of view) in a particular cell `i,j`. A winning move exists if cell `i,j` is empty and the other two cells in the same row, column, or diagonal contain an `X`. Functions `R` and `C` look for winning moves in cell `i,j`'s row and column, respectively. Functions `D` and `E` look in the two diagonals.

Within the architecture's single process, nine `BOOLEAN` variables `G11`–`G33` are declared to indicate whether each of the cells has a winning move possible. Assignment statements at the beginning of the process set each variable to `TRUE` if there is such a move, calling and combining all of the appropriate functions for cell `i,j`.

The rest of the process is a deeply nested "`if`" statement that looks for a winning move in all possible cells. Although it typically results in slower synthesized logic nested "`if`" is required rather than some form of "case" statement, because multiple moves may be possible. If no winning move is possible, the value "`NONE`" is assigned.

EXPLICIT IMPURITY

In addition to a cell index `i,j`, the functions `R`, `C`, `D`, and `E` in Table 6-39 are passed the grid state `X` and `Y`. This is necessary because VHDL functions are by default *pure*, which means that signals and variables declared in the function's parents are *not* directly visible within the function. However, you can relax this restriction by explicitly declaring a function to be *impure* by placing the keyword `impure` before the keyword `function` in its definition.

The `PICK` entity combines the results of two `TwoInRow` entities according to the program in Table 6-40. First priority is given to a winning move, followed by a blocking move. Otherwise, function `MT` is called for each cell, starting with the middle and ending with the side cells, to find an available move. This completes the design of the Tic-Tac-Toe circuit.

Table 6-40 VHDL program to pick a winning or blocking Tic-Tac-Toe move, or else use "experience."

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

entity PICK is
  port ( X, Y:          in  TTTgrid;
         WINMV, BLKMV: in  STD_LOGIC_VECTOR(3 downto 0);
         MOVE:          out STD_LOGIC_VECTOR(3 downto 0) );
end PICK;

architecture PICK_arch of PICK is
function MT(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  begin                          -- Determine if cell i,j is empty
    return X(i)(j)='0' and Y(i)(j)='0';
  end MT;
begin
  process (X, Y, WINMV, BLKMV)
  begin                                          -- If available, pick:
    if    WINMV /= NONE then MOVE <= WINMV;  -- winning move
    elsif BLKMV /= NONE then MOVE <= BLKMV;  -- else blocking move
    elsif MT(X,Y,2,2)   then MOVE <= MOVE22; -- else center cell
    elsif MT(X,Y,1,1)   then MOVE <= MOVE11; -- else corner cells
    elsif MT(X,Y,1,3)   then MOVE <= MOVE13;
    elsif MT(X,Y,3,1)   then MOVE <= MOVE31;
    elsif MT(X,Y,3,3)   then MOVE <= MOVE33;
    elsif MT(X,Y,1,2)   then MOVE <= MOVE12; -- else side cells
    elsif MT(X,Y,2,1)   then MOVE <= MOVE21;
    elsif MT(X,Y,2,3)   then MOVE <= MOVE23;
    elsif MT(X,Y,3,2)   then MOVE <= MOVE32;
    else                     MOVE <= NONE;   -- else grid is full
    end if;
  end process;
end PICK_arch;
```