Unfortunately, when I compiled this program, my computer just sat there, CPU-bound, for an hour without producing any results. That gave me time to use my brain, a good exercise for those of us who have become too dependent on CAD tools. I then realized that I could write the logic function for the SUM0 output by hand in just a few seconds,

$$SUM0 = D0 \oplus D1 \oplus D2 \oplus D3 \oplus D4 \oplus D5 \oplus D6 \oplus D7 \oplus \ldots \oplus D13 \oplus D14$$

The Karnaugh map for this function is a checkerboard, and the minimal sum-of-products expression has 2^{14} product terms. Obviously this is not going to fit in one or a few passes through a 22V10! So, anyway, I killed the ABEL compiler process, and rebooted Windows just in case the compiler had gone awry.

Obviously, a partitioning into smaller chunks is required to design the ones-counting circuit. Although we could pursue this further using ABEL and PLDs, it's more interesting to do a structural design using VHDL, as we will in Section 6.3.6. The ABEL and PLD version is left as an exercise (6.6).

6.2.7 Tic-Tac-Toe

In this example, we'll design a combinational circuit that picks a player's next move in the game of Tic-Tac-Toe. The first thing we'll do is decide on a strategy for picking the next move. Let us try to emulate the typical human's strategy, by following the decision steps below:

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.

TIC-TAC-TOE, IN CASE YOU DIDN'T KNOW

The game of Tic-Tac-Toe is played by two players on a 3×3 grid of cells that are initially empty. One player is "X" and the other is "O". The players alternate in placing their mark in an empty cell; "X" always goes first. The first player to get three of his or her own marks in the same row, column, or diagonal wins. Although the first player to move (X) has a slight advantage, it can be shown that a game between two intelligent players will always end in a draw; neither player will get three in a row before the grid fills up.

row \ column	1	2	3
1	X11,Y11	X13,Y12	X13,Y13
2	X21,Y21	X23,Y22	X23,Y23
3	X21,Y21	X23,Y22	X23,Y23

Figure 6-11
Tic-Tac-Toe grid and ABEL signal names.

Planning ahead, we'll call the second player "Y" to avoid confusion between "O" and "0" in our programs. The next thing to think about is how we might encode the inputs and outputs of the circuit. There are only nine possible moves that a player can make, so the output can be encoded in just four bits. The circuit's input is 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).

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

00 Cell is empty.

10 Cell is occupied by X.

01 Cell is occupied by Y.

So, we can encode the 3×3 grid's state in 18 bits. As shown in Figure 6-11, we'll number the grid with row and column numbers, and use ABEL signals `Xij` and `Yij` to denote the presence of X or Y in cell `i,j`. We'll look at the output coding later.

With a total of 18 inputs and 4 outputs, the Tic-Tac-Toe circuit could conceivably fit in just one 22V10. However, experience suggests that there's just no way. We're going to have to find a partitioning of the function, and partitioning along the lines of the decision steps on the preceding page seems like a good idea.

In fact, steps 1 and 2 are very similar; they differ only in reversing the roles of the player and the opponent. Here's where our symmetric encoding can pay

COMPACT ENCODING

Since each cell in the Tic-Tac-Toe grid can have only three states, not four, the total number of board configurations is 3^9, or 19,683. This is less than 2^{15}, so the board state can be encoded in only 15 bits. However, such an encoding would lead to much larger circuits for picking a move, unless the move-picking circuit was a read-only memory (see Exercise 11.26).

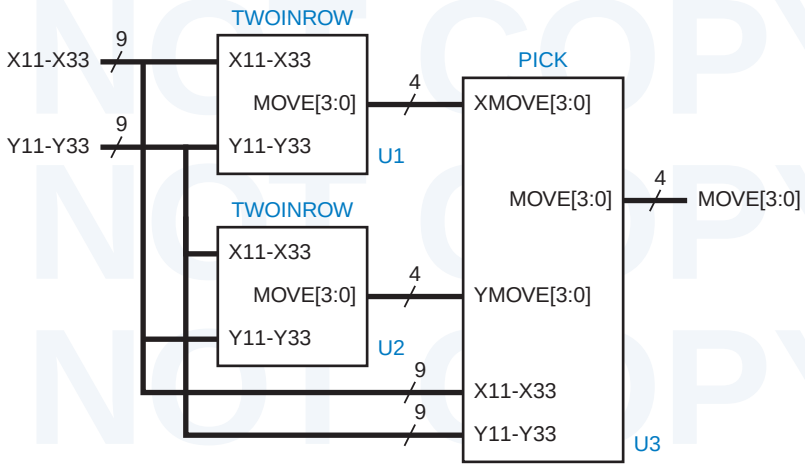

Figure 6-12 Preliminary PLD partitioning for the Tic-Tac-Toe game.

off. A PLD that finds me two of my marks in a row along with one empty cell for a winning move (step 1) can find two of my opponent's marks in a row plus an empty for a blocking move (step 2). All we have to do is swap the encodings for X and Y. With out selected coding, that doesn't require any logic, just physically swapping the `Xij` and `Yij` signals for each cell. With this in mind, we can use two copies of the same PLD, TWOINROW, to perform steps 1 and 2 as shown in Figure 6-12. Notice that the `X11–X33` signals are connected to the top inputs of the first TWOINROW PLD, but to the bottom inputs of the second.

The moves from the two TWOINROW PLDs can be examined in another PLD, PICK. This device picks a move from the first two PLDs if either found one; else it performs step 3. It looks like PICK has too many inputs and outputs to fit in a 22V10, but we'll come back to that later.

Table 6-13 is a program for the TWOINROW PLD. It looks at the grid's state from the point of view of X, that is, it looks for a move where X can get three in a row. The program makes extensive use of intermediate equations to define

Table 6-13 ABEL program to find two in a row in Tic-Tac-Toe.

```
module twoinrow
Title 'Find Two Xs and an empty cell in a row, column, or diagonal'
TWOINROW device 'P22V10';

" Inputs and Outputs
X11, X12, X13, X21, X22, X23, X31, X32, X33 pin 1..9;
Y11, Y12, Y13, Y21, Y22, Y23, Y31, Y32, Y33 pin 10,11,13..15,20..23;
MOVE3..MOVE0                                pin 16..19 istype 'com';

" MOVE output encodings
MOVE   = [MOVE3..MOVE0];
MOVE11 = [1,0,0,0]; MOVE12 = [0,1,0,0]; MOVE13 = [0,0,1,0];
MOVE21 = [0,0,0,1]; MOVE22 = [1,1,0,0]; MOVE23 = [0,1,1,1];
MOVE31 = [1,0,1,1]; MOVE32 = [1,1,0,1]; MOVE33 = [1,1,1,0];
NONE   = [0,0,0,0];
```

Table 6-13 (continued)

```
" Find moves in rows.  Rxy ==> a move exists in cell xy
R11 = X12 & X13 & !X11 & !Y11;
R12 = X11 & X13 & !X12 & !Y12;
R13 = X11 & X12 & !X13 & !Y13;
R21 = X22 & X23 & !X21 & !Y21;
R22 = X21 & X23 & !X22 & !Y22;
R23 = X21 & X22 & !X23 & !Y23;
R31 = X32 & X33 & !X31 & !Y31;
R32 = X31 & X33 & !X32 & !Y32;
R33 = X31 & X32 & !X33 & !Y33;

" Find moves in columns.  Cxy ==> a move exists in cell xy
C11 = X21 & X31 & !X11 & !Y11;
C12 = X22 & X32 & !X12 & !Y12;
C13 = X23 & X33 & !X13 & !Y13;
C21 = X11 & X31 & !X21 & !Y21;
C22 = X12 & X32 & !X22 & !Y22;
C23 = X13 & X33 & !X23 & !Y23;
C31 = X11 & X21 & !X31 & !Y31;
C32 = X12 & X22 & !X32 & !Y32;
C33 = X13 & X23 & !X33 & !Y33;

" Find moves in diagonals.  Dxy or Exy ==> a move exists in cell xy
D11 = X22 & X33 & !X11 & !Y11;
D22 = X11 & X33 & !X22 & !Y22;
D33 = X11 & X22 & !X33 & !Y33;
E13 = X22 & X31 & !X13 & !Y13;
E22 = X13 & X31 & !X22 & !Y22;
E31 = X13 & X22 & !X31 & !Y31;

" Combine moves for each cell.  Gxy ==> a move exists in cell xy
G11 = R11 # C11 # D11;
G12 = R12 # C12;
G13 = R13 # C13 # E13;
G21 = R21 # C21;
G22 = R22 # C22 # D22 # E22;
G23 = R23 # C23;
G31 = R31 # C31 # E31;
G32 = R32 # C32;
G33 = R33 # C33 # D33;

equations

WHEN G22 THEN MOVE= MOVE22;
ELSE WHEN G11 THEN MOVE = MOVE11;
ELSE WHEN G13 THEN MOVE = MOVE13;
ELSE WHEN G31 THEN MOVE = MOVE31;
ELSE WHEN G33 THEN MOVE = MOVE33;
ELSE WHEN G12 THEN MOVE = MOVE12;
ELSE WHEN G21 THEN MOVE = MOVE21;
ELSE WHEN G23 THEN MOVE = MOVE23;
ELSE WHEN G32 THEN MOVE = MOVE32;
ELSE MOVE = NONE;

end twoinrow
```

all possible row, column, and diagonal moves. It combines all of the moves for a cell `i,j` in an expression for `Gij`, and finally the `equations` section uses a `WHEN` statement to select a move.

Note that a nested `WHEN` statement must be used rather than nine parallel `WHEN` statements or assignments, because we can only select one move even if multiple moves are available. Also note that `G22`, the center cell, is checked first, followed by the corners. This was done hoping that we could minimize the number of terms by putting the most common moves early in the nested `WHEN`. Alas, the design still requires a ton of product terms, as shown in Table 6-14.

By the way, we still haven't explained why we chose the output coding that we did (as defined by `MOVE11`, `MOVE22`, etc. in the program). It's pretty clear that changing the encoding is never going to save us enough product terms to fit the design into a 22V10. But there's still method to this madness, as we'll now show.

Clearly we'll have to split TWOINROW into two or more pieces. As in any design problem, several different strategies are possible. The first strategy I tried was to use two different PLDs, one to find moves in all the rows and one of the diagonals, and the other to work on all the columns and the remaining diagonal. That helped, but not nearly enough to fit each half into a 22V10.

With the second strategy, I tried slicing the problem a different way. The first PLD finds all the moves in cells 11, 12, 13, 21, and 22, and the second PLD finds all the moves in the remaining cells. That worked! The first PLD, named TWOINHAF, is obtained from Table 6-13 simply by commenting out the four lines of the `WHEN` statement for the moves to cells 23, 31, 32, and 33.

We could obtain the second PLD from TWOINROW in a similar way, but let's wait a minute. In the manufacture of real digital systems, it is always desirable to minimize the number of distinct parts that are used; this saves on inventory costs and complexity. With programmable parts, it is desirable to minimize the number of distinct programs that are used. Even though the physical parts are identical, a different set of test vectors must be devised at some cost for each different program. Also, it's possible that the product will be successful enough for us to save money by converting the PLDs into hard-coded devices, a different one for each program, again encouraging us to minimize programs.

Table 6-14 Product-term usage for the TWOINROW PLD.

```
 P-Terms   Fan-in  Fan-out  Type  Name
---------  ------  -------  ----  -----------
  61/142     18        1    Pin   MOVE3
 107/129     18        1    Pin   MOVE2
  77/88      17        1    Pin   MOVE1
 133/87      18        1    Pin   MOVE0
=========
 378/446        Best P-Term Total: 332
                  Total Pins: 22
              Average P-Term/Output: 83
```

The Tic-Tac-Toe game is the same game even if we rotate the grid 90° or 180°. Thus, the TWOINHAF PLD can find moves for cells 33, 32, 31, 23, and 22 if we rotate the grid 180°. Because of the way we defined the grid state, with a separate pair of inputs for each cell, we can "rotate the grid" simply by shuffling the input pairs appropriately. That is, we swap 33↔11, 32↔12, 31↔13, and 23↔21.

Of course, once we rearrange inputs, TWOINHAF will still produce output move codes corresponding to cells in the top half of the grid. To keep things straight, we should transform these into codes for the proper cells in the bottom half of the grid. We would like this transformation to take a minimum of logic. This is where our choice of output code comes in. If you look carefully at the `MOVE` coding defined at the beginning of Table 6-13, you'll see that the code for a given position in the 180° rotated grid is obtained by complementing and reversing the order of the code bits for the same position in the unrotated grid. In other words, the code transformation can be done with four inverters and a rearrangement of wires. This can be done "for free" in the PLD that looks at the TWOINHAF outputs.

You probably never thought that Tic-Tac-Toe could be so tricky. Well, we're halfway there. Figure 6-13 shows the partitioning of the design as we'll now continue it. Each TWOINROW PLD from our original partition is replaced

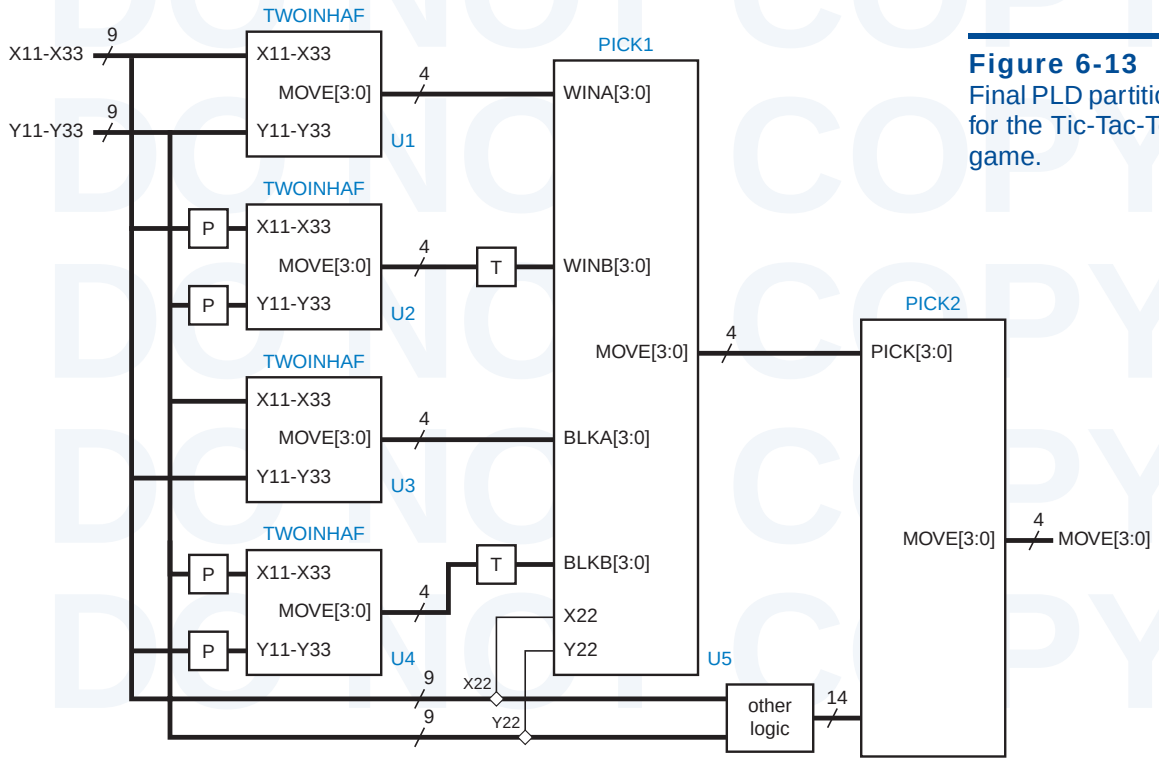

Figure 6-13
Final PLD partitioning for the Tic-Tac-Toe game.

by a pair of TWOINHALF PLDs. The bottom PLD of each pair is preceded by a box labeled "P", which permutes the inputs to rotate the grid 180° as discussed previously. Likewise, it is followed by a box labeled "T", which compensates for the rotation by transforming the output code; this box will actually be absorbed into the PLD that follows it, PICK1.

The function of PICK1 is pretty straightforward. As shown in Table 6-15, it is simply picks a winning move or a blocking move if one is available. Since there are two extra input pins available on the 22V10, we use them to input the state of the center cell. In this way, we can perform the first part of step 3 of the "human" algorithm on page 488, to pick the center cell if no winning or blocking move is available. The PICK1 PLD uses at most 9 product terms per output.

Table 6-15 ABEL program to pick one move based on four inputs.

```
module pick1
Title 'Pick One Move from Four Possible'
PICK1 device 'P22V10';

" Inputs from TWOINHAF PLDs
WINA3..WINA0      pin 1..4;       "Winning moves in cells 11,12,13,21,22
WINB3..WINB0      pin 5..8;       "Winning moves in cells 11,12,13,21,22 of rotated grid
BLKA3..BLKA0      pin 9..11, 13;  "Blocking moves in cells 11,12,13,21,22
BLKB3..BLKB0      pin 14..16, 21; "Blocking moves in cells 11,12,13,21,22 of rotated grid
" Inputs from grid
X22, Y22          pin 22..23;     "Center cell; pick if no other moves
" Move outputs to PICK2 PLD
MOVE3..MOVE0      pin 17..20 istype 'com';

" Sets
WINA = [WINA3..WINA0];  WINB = [WINB3..WINB0];
BLKA = [BLKA3..BLKA0];  BLKB = [BLKB3..BLKB0];
MOVE = [MOVE3..MOVE0];

" Non-rotated move input and output encoding
MOVE11 = [1,0,0,0]; MOVE12 = [0,1,0,0]; MOVE13 = [0,0,1,0];
MOVE21 = [0,0,0,1]; MOVE22 = [1,1,0,0]; MOVE23 = [0,1,1,1];
MOVE31 = [1,0,1,1]; MOVE32 = [1,1,0,1]; MOVE33 = [1,1,1,0];
NONE   = [0,0,0,0];

equations

WHEN WINA != NONE THEN MOVE = WINA;
ELSE WHEN WINB != NONE THEN MOVE = ![WINB0..WINB3];  " Map rotated coding
ELSE WHEN BLKA != NONE THEN MOVE = BLKA;
ELSE WHEN BLKB != NONE THEN MOVE = ![BLKB0..BLKB3];  " Map rotated coding
ELSE WHEN !X22 & !Y22 THEN MOVE = MOVE22;            " Pick center cell if it's empty
ELSE MOVE = NONE;

end pick1
```

The final part of the design in Figure 6-13 is the PICK2 PLD. This PLD must provide most of the "experience" in step 3 of the human algorithm if PICK1 does not find a move.

We have a little problem with PICK2 in that a 22V10 does not have enough pins to accommodate the 4-bit input from PICK1, its own 4-bit output, and all 18 bits of grid state; it has only 22 I/O pins. Actually, we don't need to connect X22 and Y22, since they were already examined in PICK1, but that still leaves us two pins short. So, the purpose of the "other logic" block in Figure 6-13 is to encode

Table 6-16 ABEL program to pick one move using "experience."

```
module pick2
Title 'Pick a move using experience'
PICK2 device 'P22V10';

" Inputs from PICK1 PLD
PICK3..PICK0                          pin 1..4;  " Move, if any, from PICK1 PLD
" Inputs from Tic-Tac-Toe grid corners
X11, Y11, X13, Y13, X31, Y31, X33, Y33   pin 5..11, 13;
" Combined inputs from external NOR gates; 1 ==> corresponding cell is empty
E12, E21, E23, E32                    pin 14..15, 22..23;
" Move output
MOVE3..MOVE0                          pin 17..20 istype 'com';

PICK = [PICK3..PICK0];  " Set definition
" Non-rotated move input and output encoding
MOVE = [MOVE3..MOVE0];
MOVE11 = [1,0,0,0]; MOVE12 = [0,1,0,0]; MOVE13 = [0,0,1,0];
MOVE21 = [0,0,0,1]; MOVE22 = [1,1,0,0]; MOVE23 = [0,1,1,1];
MOVE31 = [1,0,1,1]; MOVE32 = [1,1,0,1]; MOVE33 = [1,1,1,0];
NONE   = [0,0,0,0];

" Intermediate equations for empty corner cells
E11 = !X11 & !Y11;  E13 = !X13 & !Y13;  E31 = !X31 & !Y31;  E33 = !X33 & !Y33;

equations

"Simplest approach -- pick corner if available, else side
WHEN PICK != NONE THEN MOVE = PICK;
ELSE WHEN E11 THEN MOVE = MOVE11;
ELSE WHEN E13 THEN MOVE = MOVE13;
ELSE WHEN E31 THEN MOVE = MOVE31;
ELSE WHEN E33 THEN MOVE = MOVE33;
ELSE WHEN E12 THEN MOVE = MOVE12;
ELSE WHEN E21 THEN MOVE = MOVE21;
ELSE WHEN E23 THEN MOVE = MOVE23;
ELSE WHEN E32 THEN MOVE = MOVE32;
ELSE MOVE = NONE;

end pick2
```

some of the information to save two pins. The method that we'll use here is to combine the signals for the middle edge cells 12, 21, 23, and 32 to produce four signals E12, E21, E23, and E32 that are asserted if and only if the corresponding cells are empty. This can be done with four 2-input NOR gates, and actually leaves two spare inputs or outputs on the 22V10.

Assuming the four NOR gates as "other logic," Table 6-16 on the preceding page gives a program for the PICK2 PLD. When it must pick a move, this program uses the simplest heuristic possible—it picks a corner cell if one is empty, else it picks a middle edge cell. This program could use some improvement, because it will sometimes lose (see Exercise 6.8). Luckily, the equations resulting from Table 6-16 require only 8 to 10 terms per output, so it's possible to put in more intelligence (see Exercises 6.9 and 6.10).

6.3 Design Examples Using VHDL

6.3.1 Barrel Shifter

On page 464, we defined a barrel shifter as a combinational logic circuit with *n* data inputs, *n* data outputs, and a set of control inputs that specify how to shift the data between input and output. We showed in Section 6.1.1 how to build a simple barrel shifter that performs only left circular shifts using MSI building blocks. Later, in Section 6.2.1, we showed how to define a more capable barrel shifter using ABEL, but we also pointed out that PLDs are normally unsuitable for realizing barrel shifters. In this subsection, we'll show how VHDL can be used to describe both the behavior and structure of barrel shifters for FPGA or ASIC realization.

Table 6-17 is a behavioral VHDL program for a 16-bit barrel shifter that performs any of six different combinations of shift type and direction. The shift types are circular, logical, and arithmetic, as defined previously in Table 6-3, and the directions are of course left and right. As shown in the entity declaration, a 4-bit control input `S` gives the shift amount, and a 3-bit control input `C` gives the shift mode (type and direction). We used the IEEE `std_logic_arith` package and defined the shift amount `S` to be type `UNSIGNED` so we could later use the `CONV_INTEGER` function in that package.

Notice that the entity declaration includes six constant definitions that establish the correspondence between shift modes and the value of `C`. Although we didn't discuss it in Section 4.7, VHDL allows you to put constant, type, signal, and other declarations within an entity declaration. It makes sense to define such items within the entity declaration only if they must be the same in any architecture. In this case, we are pinning down the shift-mode encodings, so they should go here. Other items should go in the architecture definition.

In the architecture part of the program, we define six functions, one for each kind of shift on a 16-bit `STD_LOGIC_VECTOR`. We defined the subtype `DATAWORD` to save typing in the function definitions.