XCabl: Combinational ABEL Examples

This section shows combinational design examples using ABEL and PLDs.

XCabl.1 Barrel Shifter

barrel shifter

A *barrel shifter* is 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. A barrel shifter that is part of a microprocessor CPU can typically specify the direction of shift (left or right), the type of shift (circular, arithmetic, or logical), and the amount of shift (typically 0 to $n-1$ bits, but sometimes 1 to n bits).

In this subsection we'll look at the design of a simple 16-bit barrel shifter that does left circular shifts only, using a 4-bit control input S[3:0] to specify the amount of shift. For example, if the input word is ABCDEFGHIJKLMNOP (where each letter represents one bit), and the control input is 0101 (5), then the output word is FGHIJKLMNOPABCDE.

A barrel shifter is good example of something *not* to design using PLDs. However, we can put ABEL to good use to describe a barrel shifter's function, and we'll also see why a typical barrel shifter is not a good fit for a PLD.

Table XCabl-1 shows the equations for a 16-bit barrel shifter with the functionality described above—it does left circular shifts only, using a 4-bit control input S[3..0] to specify the amount of shift. ABEL makes it easy to specify the functionality of the overall circuit without worrying about how the circuit might be partitioned into multiple chips. Also, ABEL dutifully generates a minimal sum-of-products expression for each output bit. In this case, each output requires 16 product terms.

Table XCabl-1 ABEL program for a 16-bit barrel shifter.

```
module barrel16
title '16-bit Barrel Shifter'

" Inputs and Outputs
DIN15..DIN0, S3..S0                pin;
DOUT15..DOUT0                      pin istype 'com';

S = [S3..S0];

equations
[DOUT15..DOUT0] = (S==0) & [DIN15..DIN0]
                # (S==1) & [DIN14..DIN0,DIN15]
                # (S==2) & [DIN13..DIN0,DIN15..DIN14]
                # (S==3) & [DIN12..DIN0,DIN15..DIN13]
                ...
                # (S==12) & [DIN3..DIN0,DIN15..DIN4]
                # (S==13) & [DIN2..DIN0,DIN15..DIN3]
                # (S==14) & [DIN1..DIN0,DIN15..DIN2]
                # (S==15) & [DIN0,DIN15..DIN1];
end barrel16
```

Partitioning the 16-bit barrel shifter into multiple PLDs is a difficult task in two different ways. First, it should be obvious that the nature of the function is such that every output bit depends on every input bit. A PLD that produces, say, the DOUT0 output must have all 16 DIN inputs and all four S inputs available to it. So, a GAL16V8 definitely cannot be used; it has only 16 inputs.

The GAL20V8 is similar to the GAL16V8, with the addition of four input-only pins. If we use all 20 available inputs, we are left with two output-only pins (corresponding to the top and bottom outputs in Figure 6-27 on page 377). Thus, it seems possible that we could realize the barrel shifter using eight 20V8 chips, producing two output bits per chip.

No, we still have a problem. The second dimension of difficulty in a PLD-based barrel shifter is the number of product terms per output. The barrel shifter requires 16, and the 20V8 provides only 7. We're stuck—any realization of the barrel shifter using 20V8s is going to require multiple-pass logic. At this point, we would be best advised to look at partitioning options along the lines that we do in Section XCbb.1 at DDPPonline.

The 16-bit barrel shifter can be realized without much difficulty in a larger programmable device, that is, in a CPLD or an FPGA with enough I/O pins. However, imagine that we were trying to design a 32-bit or 64-bit barrel shifter. Clearly, we would need to use a device with even more I/O pins, but that's not all. The number of product terms and the large amount of connectivity (all the inputs connect to all the outputs) would still be challenging.

Indeed, a typical CPLD or FPGA fitter could have difficulty realizing a large barrel shifter with small delay or even at all. There is a critical resource that we took for granted in the partitioned, building-block barrel-shifter designs of Section XCbb.1—wires! An FPGA is somewhat limited in its internal connectivity, and a CPLD is even more so. Thus, even with modern FPGA and CPLD design tools, you may still have to "use your head" to partition the design in a way that helps the tools do their job.

Barrel shifters can be even more complex than what we've shown so far. Just for fun, Table XCabl-2 shows the design for a barrel shifter that supports six different kinds of shifting. This requires even more product terms, up to 40 per output! Although you'd never build this device in a PLD, CPLD, or small FPGA, the minimized ABEL equations are useful because they can help you understand the effects of some of your design choices. For example, by changing the coding of `SLA` and `SRA` to `[1,.X.,0]` and `[1,.X.,1]`, you can reduce the total number of product terms in the design from 624 to 608. You can save more product terms by changing the coding of the shift amount for some shifts (see Exercise XCabl.1). The savings from these changes may carry over to other design approaches.

Table XCabl-2 ABEL program for a multi-mode 16-bit barrel shifter.

```
module barrl16f
Title 'Multi-mode 16-bit Barrel Shifter'

" Inputs and Outputs
DIN15..DIN0, S3..S0, C2..C0         pin;
DOUT15..DOUT0                       pin istype 'com';

S = [S3..S0]; C = [C2..C0]; " Shift amount and mode
L = DIN15; R = DIN0;        " MSB and LSB

ROL = (C == [0,0,0]); " Rotate (circular shift) left
ROR = (C == [0,0,1]); " Rotate (circular shift) right
SLL = (C == [0,1,0]); " Shift logical left (shift in 0s)
SRL = (C == [0,1,1]); " Shift logical right (shift in 0s)
SLA = (C == [1,0,0]); " Shift left arithmetic (replicate LSB)
SRA = (C == [1,0,1]); " Shift right arithmetic (replicate MSB)

equations
[DOUT15..DOUT0] = ROL & (S==0) & [DIN15..DIN0]
                # ROL & (S==1) & [DIN14..DIN0,DIN15]
                # ROL & (S==2) & [DIN13..DIN0,DIN15..DIN14]
                ...
                # ROL & (S==15) & [DIN0,DIN15..DIN1]
                # ROR & (S==0) & [DIN15..DIN0]
                # ROR & (S==1) & [DIN0,DIN15..DIN1]
                ...
                # ROR & (S==14) & [DIN13..DIN0,DIN15..DIN14]
                # ROR & (S==15) & [DIN14..DIN0,DIN15]
                # SLL & (S==0) & [DIN15..DIN0]
                # SLL & (S==1) & [DIN14..DIN0,0]
                ...
                # SLL & (S==14) & [DIN1..DIN0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
                # SLL & (S==15) & [DIN0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
                # SRL & (S==0) & [DIN15..DIN0]
                # SRL & (S==1) & [0,DIN15..DIN1]
                ...
                # SRL & (S==14) & [0,0,0,0,0,0,0,0,0,0,0,0,0,0,DIN15..DIN14]
                # SRL & (S==15) & [0,0,0,0,0,0,0,0,0,0,0,0,0,0,DIN15]
                # SLA & (S==0) & [DIN15..DIN0]
                # SLA & (S==1) & [DIN14..DIN0,R]
                ...
                # SLA & (S==14) & [DIN1..DIN0,R,R,R,R,R,R,R,R,R,R,R,R,R,R]
                # SLA & (S==15) & [DIN0,R,R,R,R,R,R,R,R,R,R,R,R,R,R]
                # SRA & (S==0) & [DIN15..DIN0]
                # SRA & (S==1) & [L,DIN15..DIN1]
                ...
                # SRA & (S==14) & [L,L,L,L,L,L,L,L,L,L,L,L,L,L,DIN15..DIN14]
                # SRA & (S==15) & [L,L,L,L,L,L,L,L,L,L,L,L,L,L,DIN15];
end barrl16f
```

XCabl.2 Simple Floating-Point Encoder

An unsigned binary integer B in the range $0 \le B < 2^{11}$ can be represented by 11 bits in "fixed-point" format, $B = b_{10}b_9 \ldots b_1b_0$. We can represent numbers in the same range with less precision using only 7 bits in a floating-point notation, $F = M \cdot 2^E$, where M is a 4-bit mantissa $m_3m_2m_1m_0$ and E is a 3-bit exponent $e_2e_1e_0$. The smallest integer in this format is $0 \cdot 2^0$ and the largest is $(2^4–1) \cdot 2^7$.

Given an 11-bit fixed-point integer B, we can convert it to our 7-bit floating-point notation by "picking off" four high-order bits beginning with the most significant 1, for example,

$$\begin{aligned}
11010110100 &= 1101 \cdot 2^7 + 0110100 \\
00100101111 &= 1001 \cdot 2^5 + 01111 \\
00000111110 &= 1111 \cdot 2^2 + 10 \\
00000001011 &= 1011 \cdot 2^0 + 0 \\
00000000010 &= 0010 \cdot 2^0 + 0
\end{aligned}$$

The last term in each equation is a truncation error that results from the loss of precision in the conversion. Corresponding to this conversion operation, we can write the specification for a fixed-point to floating-point encoder circuit:

- A combinational circuit is to convert an 11-bit unsigned binary integer B into a 7-bit floating-point number M,E, where M and E have 4 and 3 bits, respectively. The numbers have the relationship $B = M \cdot 2^E + T$, where T is the truncation error, $0 \le T < 2^E$.

The I/O-pin requirements of this design are limited—11 inputs and 7 outputs—so we can potentially use a single PLD to replace the four chips that were used in an MSI solution to the same probem in Section XCbb.2.

An ABEL program for the fixed-to-floating-point converter is given in Table XCabl-3. The `when` statement expresses the operation of determining the exponent value `E` in a very natural way. Then `E` is used to select the appropriate bits of `B` to use as the mantissa `M`.

Despite the deep nesting of the `when` statement, only four product terms are needed in the minimal sum for each bit of `E`. The equations for the `M` bits are not too bad either, requiring only eight product terms each. Unfortunately, the GAL20V8 has available only seven product terms per output. However, the GAL22V10 (Figure 8-19 on page 705) has more product terms available, so we can use that if we like.

One drawback of the design in Table XCabl-3 is that the `[M3..M0]` outputs are slow; since they use `[E2..E0]`, they take two passes through the PLD. A faster approach, if it fits, would be to rewrite the "select" terms (`E==0`, etc.) as intermediate equations before the equations section, and let ABEL expand the

Table XCabl-3 An ABEL program for the fixed-point to floating-point PLD.

```
module fpenc
title 'Fixed-point to Floating-point Encoder'
FPENC device 'P20V8C';

" Input and output pins
B10..B0               pin 1..11;
E2..E0, M3..M0        pin 21..15 istype 'com';

" Constant expressions
B = [B10..B0];
E = [E2..E0];
M = [M3..M0];

equations

when B < 16 then E = 0;
else when B < 32 then E = 1;
else when B < 64 then E = 2;
else when B < 128 then E = 3;
else when B < 256 then E = 4;
else when B < 512 then E = 5;
else when B < 1024 then E = 6;
else E = 7;

M = (E==0) & [B3..B0]
  # (E==1) & [B4..B1]
  # (E==2) & [B5..B2]
  # (E==3) & [B6..B3]
  # (E==4) & [B7..B4]
  # (E==5) & [B8..B5]
  # (E==6) & [B9..B6]
  # (E==7) & [B10..B7];

end fpenc
```

resulting `M` equations in a single level of logic. Unfortunately, ABEL does not allow `when` statements outside of the `equations` section, so we'll have to roll up our sleeves and write our own logic expressions in the intermediate equations.

Table XCabl-4 on the next page shows the modified approach. The expressions for `S7`–`S0` are just mutually exclusive AND-terms that indicate exponent values of 7–0 depending on the location of the most significant 1 bit in the fixed-point input number. The exponent `[E2..E0]` is a binary encoding of the select terms, and the mantissa bits `[M3..M0]` are generated using a select term for each case. It turns out that these `M` equations still require 8 product terms per output bit, but at least they're a lot faster, since they use just one level of logic.

Table XCabl-4 Alternative ABEL program for the fixed-point to floating-point PLD.

```
module fpence
title 'Fixed-point to Floating-point Encoder'
FPENCE device 'P20V8C';

" Input and output pins
B10..B0                    pin 1..11;
E2..E0, M3..M0             pin 21..15 istype 'com';

" Intermediate equations
S7 = B10;
S6 = !B10 & B9;
S5 = !B10 & !B9 & B8;
S4 = !B10 & !B9 & !B8 & B7;
S3 = !B10 & !B9 & !B8 & !B7 & B6;
S2 = !B10 & !B9 & !B8 & !B7 & !B6 & B5;
S1 = !B10 & !B9 & !B8 & !B7 & !B6 & !B5 & B4;
S0 = !B10 & !B9 & !B8 & !B7 & !B6 & !B5 & !B4;

equations

E2 = S7 # S6 # S5 # S4;
E1 = S7 # S6 # S3 # S2;
E0 = S7 # S5 # S3 # S1;

[M3..M0] = S0 & [B3..B0] # S1 & [B4..B1] # S2 & [B5..B2]
         # S3 & [B6..B3] # S4 & [B7..B4] # S5 & [B8..B5]
         # S6 & [B9..B6] # S7 & [B10..B7];

end fpenc
```

XCabl.3 Dual-Priority Encoder

In this example, we'll design a PLD-based priority encoder that identifies both the highest-priority and the second-highest-priority asserted signal among a set of eight active-high request inputs named `[R0..R7]`, where `R0` has the highest priority. We'll use `[A2..A0]` and `AVALID` to identify the highest-priority request, asserting `AVALID` only if a highest-priority request is present. Similarly, we'll use `[B2:B0]` and `BVALID` to identify the second-highest-priority request.

Table XCabl-5 shows an ABEL program for the priority encoder. As usual, a nested `when` statement is perfect for expressing priority behavior. To find the second-highest priority input, we exclude an input if its input number matches the highest-priority input number, which is `A`. Thus, we're using two-pass logic to compute the `B` outputs. The equation for `AVALID` is easy; `AVALID` is 1 if the request inputs are not all 0. To compute `BVALID`, we OR all of the conditions that set `B` in the `when` statement.

Table XCabl-5
ABEL program for a dual-priority encoder.

```
title 'Dual Priority Encoder'
PRIORTWO device 'P16V8';

" Input and output pins
R7..R0                          pin 1..8;
AVALID, A2..A0, BVALID, B2..B0  pin 19..12 istype 'com';

" Set definitions
A = [A2..A0]; B = [B2..B0];

equations

when R0==1 then A=0;
else when R1==1 then A=1;
else when R2==1 then A=2;
else when R3==1 then A=3;
else when R4==1 then A=4;
else when R5==1 then A=5;
else when R6==1 then A=6;
else when R7==1 then A=7;

AVALID = ([R7..R0] != 0);

when (R0==1) & (A!=0) then B=0;
else when (R1==1) & (A!=1) then B=1;
else when (R2==1) & (A!=2) then B=2;
else when (R3==1) & (A!=3) then B=3;
else when (R4==1) & (A!=4) then B=4;
else when (R5==1) & (A!=5) then B=5;
else when (R6==1) & (A!=6) then B=6;
else when (R7==1) & (A!=7) then B=7;

BVALID = (R0==1) & (A!=0) # (R1==1) & (A!=1)
       # (R2==1) & (A!=2) # (R3==1) & (A!=3)
       # (R4==1) & (A!=4) # (R5==1) & (A!=5)
       # (R6==1) & (A!=6) # (R7==1) & (A!=7);

end priortwo
```

Even with two-pass logic, the B outputs use too many product terms to fit in a 16V8; Table XCabl-6 on the next page shows the product-term usage. The B outputs use too many terms even for a 22V10, which has 16 terms for two of its output pins and 8–14 for the others. Sometimes you just have to work harder to make things fit!

So, how can we save some product terms? One important thing to notice is that R0 can never be the second-highest-priority asserted input, and therefore B

```
 P-Terms   Fan-in  Fan-out  Type  Name
---------  ------  -------  ----  ----------
   8/1        8        1    Pin   AVALID
   4/5        8        1    Pin   A2
   4/5        8        1    Pin   A1
   4/5        8        1    Pin   A0
  24/8       11        1    Pin   BVALID
  24/17      11        1    Pin   B2
  20/21      11        1    Pin   B1
  18/22      11        1    Pin   B0
=========
 106/84          Best P-Term Total: 76
                        Total Pins: 16
              Average P-Term/Output: 9
```

Table XCabl-6 Product-term usage in the dual-priority encoder PLD.

can never be valid and 0. Thus, we can eliminate the `when` clause for the `R0==1` case. Making this change reduces the minimum number of terms for `B2`–`B0` to 14, 17, and 15, respectively. We can almost fit the design in a 22V10, if we can just take one term out of the `B1` equation.

Well, let's try something else. The second `when` clause, for the `R1==1` case, also fails to make use of everything we know. We don't need the full generality of `A!=1`; this case is only important when `R0` is 1. So, let us replace the first two lines of the original `when` statement with

```
when (R1==1) & (R0==1) then B=1;
```

This subtle change reduces the minimum number of terms for `B2`–`B0` to 12, 16, and 13, respectively. We made it! Can the number of product terms be reduced further, enough to fit into a 16V8 while maintaining the same functionality? It's not likely, but we'll leave that as an exercise (XCabl.2) for the reader!

SUMS OF PRODUCTS AND PRODUCTS OF SUMS (SAY THAT FIVE TIMES FAST)

As shown at DDPPonline in Section Min.1, the minimal sum-of-products expression for the complement of a function can be manipulated through DeMorgan's theorem to obtain a minimal product-of-sums expression for the original function. The number of product terms in the minimal sum of products may differ from the number of sum terms in the minimal product of sums. The "P-Terms" column in Table XCabl-6 lists the number of terms in both minimal forms (product/sum terms). If *either* minimal form has less than or equal to the number of product terms available in a 22V10's AND-OR array, then the function can be made to fit.

HAVE IT YOUR WAY

Early PLDs such as the PAL16L8s did not have output-polarity control. Designers who used these devices were forced to choose a particular polarity, active high or active low, for some outputs in order to obtain reduced equations that would fit. When a 16V8, 20V8, 22V10, or any of a plethora of modern CPLDs is used, no such restriction exists. If an equation *or its complement* can be reduced to the number of product terms available, then the corresponding output can be made active high or active low by programming the output-polarity fuse appropriately.

XCabl.4 Cascading Comparators

We showed in Section 6.9.6 that equality comparisons are easy to realize in PLDs but that magnitude comparisons (greater-than or less-than) of more than a few bits are not good candidates for PLD realization, owing to the large number of product terms required. Thus, comparators are best realized using discrete MSI comparator components or as specialized cells within an FPGA or ASIC library. However, PLDs are quite suitable for realizing the combinational logic used in "parallel expansion" schemes that construct wider comparators from smaller ones, as we'll show here.

In Section 6.9.4 we showed how to connect 74x85 4-bit comparators in series to create larger comparators. Although a serial cascading scheme requires no extra logic to build arbitrarily large comparators, it has the major drawback that the delay increases linearly with the length of the cascade.

In Section XCbb.4, on the other hand, we showed how multiple copies of the 74x682 8-bit comparator could be used in parallel along with combinational logic to perform a 24-bit comparison. This scheme can be generalized for comparisons of arbitrary width.

Table XCabl-7 on the next page uses a GAL22V10 to perform a 64-bit comparison, combining the equal (EQ) and greater-than (GT) outputs of eight individual 74x682s to produce all six possible relations of the two 64-bit input values ($=, \neq, >, \geq, <, \leq$).

In this program, the `PEQQ` and `PNEQ` outputs can be realized with one product term each. The remaining eight outputs use eight product terms each. As we've mentioned previously, the 22V10 provides 8–16 product terms per output, so the design fits.

Table XCabl-7 ABEL program for combining eight 74x682s into a 64-bit comparator.

```
module compexp
title 'Expansion logic for 64-bit comparator'
COMPEXP device 'P22V10';

" Inputs from the individual comparators, active-low, 7 = MSByte
EQ_L7..EQ_L0, GT_L7..GT_L0              pin 1..11, 13..14, 21..23;

" Comparison outputs
PEQQ, PNEQ, PGTQ, PGEQ, PLTQ, PLEQ      pin 15..20 istype 'com';

" Active-level conversions
EQ7 = !EQ_L7; EQ6 = !EQ_L6; EQ5 = !EQ_L5; EQ4 = !EQ_L4;
EQ3 = !EQ_L3; EQ2 = !EQ_L2; EQ1 = !EQ_L1; EQ0 = !EQ_L0;
GT7 = !GT_L7; GT6 = !GT_L6; GT5 = !GT_L5; GT4 = !GT_L4;
GT3 = !GT_L3; GT2 = !GT_L2; GT1 = !GT_L1; GT0 = !GT_L0;

" Less-than terms
LT7 = !(EQ7 # GT7); LT6 = !(EQ6 # GT6); LT5 = !(EQ5 # GT5);
LT4 = !(EQ4 # GT4); LT3 = !(EQ3 # GT3); LT2 = !(EQ2 # GT2);
LT1 = !(EQ1 # GT1); LT0 = !(EQ0 # GT0);

equations

PEQQ = EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & EQ0;

PNEQ = !(EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & EQ0);

PGTQ = GT7  #  EQ7 & GT6  #  EQ7 & EQ6 & GT5
     # EQ7 & EQ6 & EQ5 & GT4  #  EQ7 & EQ6 & EQ5 & EQ4 & GT3
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & GT2
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & GT1
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & GT0;

PLEQ = !(GT7  #  EQ7 & GT6  #  EQ7 & EQ6 & GT5
       # EQ7 & EQ6 & EQ5 & GT4  #  EQ7 & EQ6 & EQ5 & EQ4 & GT3
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & GT2
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & GT1
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & GT0);

PLTQ = LT7  #  EQ7 & LT6  #  EQ7 & EQ6 & LT5
     # EQ7 & EQ6 & EQ5 & LT4  #  EQ7 & EQ6 & EQ5 & EQ4 & LT3
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & LT2
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & LT1
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & LT0;

PGEQ = !(LT7  #  EQ7 & LT6  #  EQ7 & EQ6 & LT5
       # EQ7 & EQ6 & EQ5 & LT4  #  EQ7 & EQ6 & EQ5 & EQ4 & LT3
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & LT2
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & LT1
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & LT0);

end compexp
```

XCabl.5 Mode-Dependent Comparator

For the next example, let us suppose we have a system in which we need to compare two 32-bit words under normal circumstances, but where we must sometimes ignore one or two low-order bits of the input words. The operating mode of the comparator is specified by two mode-control bits, M1 and M0, as shown in Table XCabl-8.

M1	*M0*	*Comparison*
0	0	32-bit
0	1	31-bit
1	0	30-bit
1	1	not used

Table XCabl-8 Mode-control bits for the mode-dependent comparator.

As we've noted previously, comparing, adding, and other "iterative" operations are usually poor candidates for PLD-based design, because an equivalent two-level sum-of-products expression has far too many product terms. In Section 6.9.6 we calculated how many product terms are needed for an *n*-bit comparator. Based on these results, we certainly wouldn't be able to build the 32-bit mode-dependent comparator or even an 8-bit slice of it with a PLD; the 74x682 8-bit comparator is just about the most efficient possible single chip we can use to perform an 8-bit comparison. However, a PLD-based design is quite reasonable for handling the mode-control logic and the part of the comparison that is dependent on mode (the two low-order bits).

Figure XCabl-1 on the next page shows a complete circuit design resulting from this idea, and Table XCabl-9 is the ABEL program for a 16V8 MODE-COMP PLD that handles the "random logic." Four '682s are used to compare most of the bits, and the 16V8 combines the '682 outputs and handles the two low-order bits as a function of the mode. Intermediate expressions `EQ30` and `GT30` are defined to save typing in the `equations` section of the program.

As shown in Table XCabl-10, the `XEQY` and `XGTY` outputs use 7 and 11 product terms, respectively. Thus, `XGTY` does not fit into the 7 product terms available on a 16V8 output. However, this is another example where we have some flexibility in our coding choices. By changing the coding of `MODE30` to `[1,.X.]`, we can reduce the product-term requirements for `XGTY` to 7/12 and thereby fit the design into a 16V8.

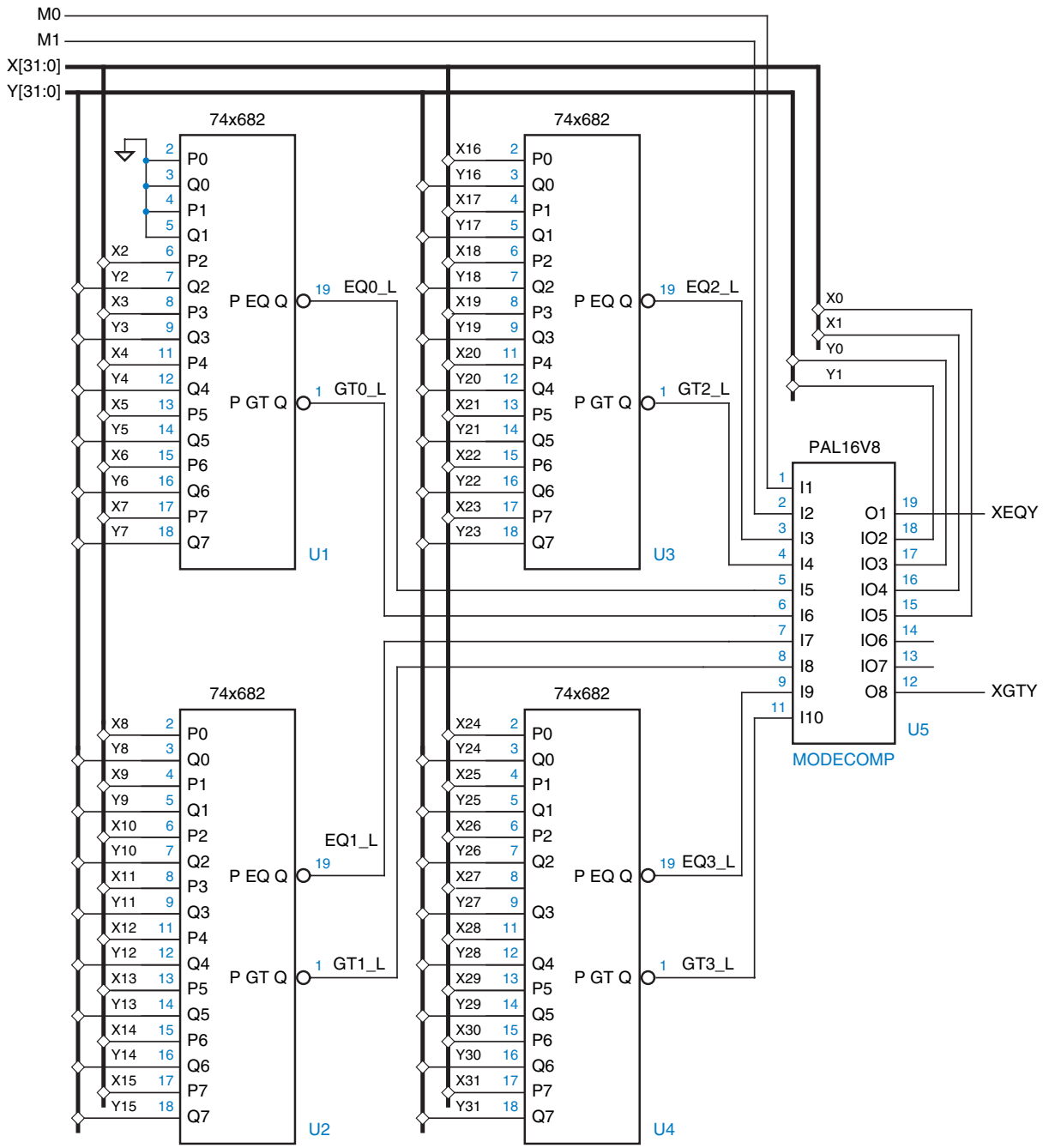

Figure XCabl-1 A 32-bit mode-dependent comparator.

Table XCabl-9
ABEL program for a mode-dependent comparator.

```
module modecomp
title 'Control PLD for Mode-Dependent Comparator'
MODECOMP device 'P16V8';

" Input and output pins
M0, M1, EQ2_L, GT2_L, EQ0_L, GT0_L          pin 1..6;
EQ1_L, GT1_L, EQ3_L, GT3_L, X0, X1, Y0, Y1  pin 7..9, 10, 15..18;
XEQY, XGTY                                  pin 19, 12 istype 'com';

" Active-level conversions
EQ3 = !EQ3_L; EQ2 = !EQ2_L; EQ1 = !EQ1_L; EQ0 = !EQ0_L;
GT3 = !GT3_L; GT2 = !GT2_L; GT1 = !GT1_L; GT0 = !GT0_L;

" Mode definitions
MODE32 = ([M1,M0] == [0,0]); " 32-bit comparison
MODE31 = ([M1,M0] == [0,1]); " 31-bit comparison
MODE30 = ([M1,M0] == [1,0]); " 30-bit comparison
MODEXX = ([M1,M0] == [1,1]); " Unused

" Expressions for 30-bit equal and greater-than
EQ30 = EQ3 & EQ2 & EQ1 & EQ0;
GT30 = GT3 # (EQ3 & GT2) # (EQ3 & EQ2 & GT1) # (EQ3 & EQ2 & EQ1 & GT0);

equations

when MODE32 then {
  XEQY = EQ30 & (X1==Y1) & (X0==Y0);
  XGTY = GT30 # (EQ30 & (X1>Y1)) # (EQ30 & (X1==Y1) & (X0>Y0));
  }
else when MODE31 then {
  XEQY = EQ30 & (X1==Y1);
  XGTY = GT30 # (EQ30 & (X1>Y1));
  }
else when MODE30 then {
  XEQY = EQ30;
  XGTY = GT30;
  }

end modecomp
```

Table XCabl-10
Product-term usage for the MODECOMP PLD.

```
 P-Terms   Fan-in  Fan-out  Type  Name
---------  ------  -------  ----  --------
   7/9       10        1    Pin   XEQY
  11/13      14        1    Pin   XGTY
=========
  18/22          Best P-Term Total: 18
                       Total Pins: 16
              Average P-Term/Output: 9
```

XCabl.6 Ones Counter

There are several important algorithms that include the step of counting the number of "1" bits in a data word. In fact, some microprocessor instruction sets have been extended recently to include ones counting as a basic instruction.

Counting the ones in a data word can be done easily as an iterative process, where you scan the word from one end to the other and increment a counter each time a "1" is encountered. However, this operation must be done more quickly inside the arithmetic and logic unit of a microprocessor. Ideally, we would like ones counting to run as fast as any other arithmetic operation, such as adding two words. Therefore, a combinational circuit is required.

For this example, suppose that we have a requirement to build a 32-bit ones counter as part of a larger system. Based on the number of inputs and outputs required, we obviously can't fit the design into a single 22V10-class PLD, but we might be able to partition the design into a reasonably small number of PLDs.

Figure XCabl-2 shows such a partition. Two copies of a first 22V10, ONESCNT1, are used to count the ones in two 15-bit chunks of the 32-bit input word D[31:0], each producing a 4-bit sum output. A second 22V10, ONESCNT2, is used to add the two 4-bit sums and the last 2 input bits.

The program for ONESCNT1 is deceptively simple, as shown in Table XCabl-11. The statement "`@CARRY 1`" is included to limit the carry chain to one stage; as explained in Section 6.10.8, this reduces product-term requirements at the expense of helper outputs and increased delay.

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,

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

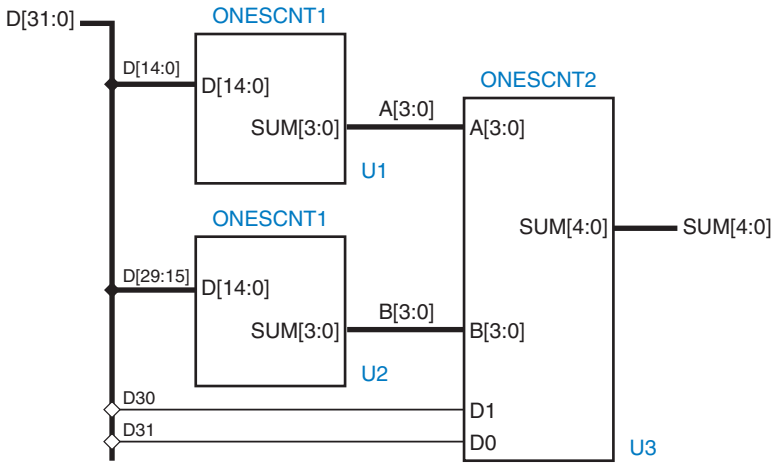

Figure XCabl-2 Possible partitioning for the ones-counting circuit.

```
module onescnt1
title 'Count the ones in a 15-bit word'
ONESCNT1 device 'P22V10';

" Input and output pins
D14..D0              pin 1..11, 13..15, 23;
SUM3..SUM0           pin 17..20 istype 'com';

equations

@CARRY 1;
[SUM3..SUM0] = D0 + D1 + D2 + D3 + D4 + D5 + D6 + D7
             + D8 + D9 + D10 + D11 + D12 + D13 + D14;

end onescnt1
```

Table XCabl-11 ABEL program for counting the 1 bits in a 15-bit word.

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 much interesting to do a structural design using VHDL (Section XCvhd.6) or Verilog (Section XCver.6). The ABEL and PLD version is left as an exercise (XCabl.4).

XCabl.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.

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

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.

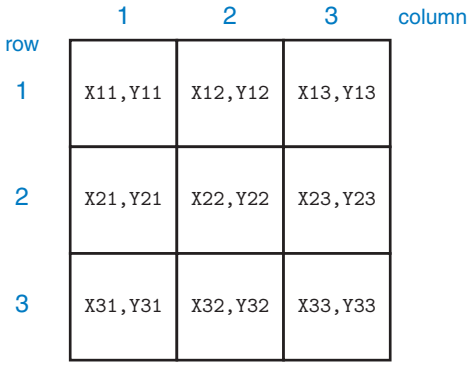

Figure XCabl-3
Tic-Tac-Toe grid and ABEL signal names.

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 using two bits per cell, a total of 18 bits. As shown in Figure XCabl-3, 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.

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 9.26).

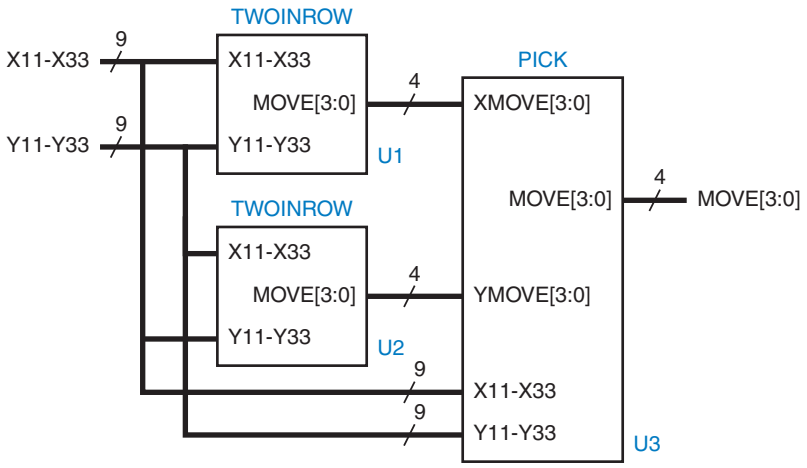

Figure XCabl-4 Preliminary PLD partitioning for the Tic-Tac-Toe game.

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 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 cell for a blocking move (step 2). All we have to do is swap the encodings for X and Y. With our 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 XCabl-4. 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 XCabl-12 on the next page 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 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 select only one move even if multiple moves are available. Also note that `G22`, the center cell, is checked first,

Table XCabl-12
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];

" 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;
```

Table XCabl-12
(continued)

```
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
```

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 XCabl-13.

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 XCabl-12 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.

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

```
 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
```

Table XCabl-13 Product-term usage for the TWOINROW PLD.

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 XCabl-12, 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 XCabl-5 shows the partitioning of the design as we'll now continue it. Each TWOINROW PLD from our original partition is replaced by a pair of TWOINHAF 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 XCabl-14, 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 15, to pick the center cell if no winning or blocking move is available. The PICK1 PLD uses at most 9 product terms per output.

The final part of the design in Figure XCabl-5 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 XCabl-5 is to encode 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 XCabl-15 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 XCabl.6). Luckily, the equations resulting from Table XCabl-15 require only 8 to 10 terms per output, so it's possible to put in more intelligence (see Exercises XCabl.7 and XCabl.8).

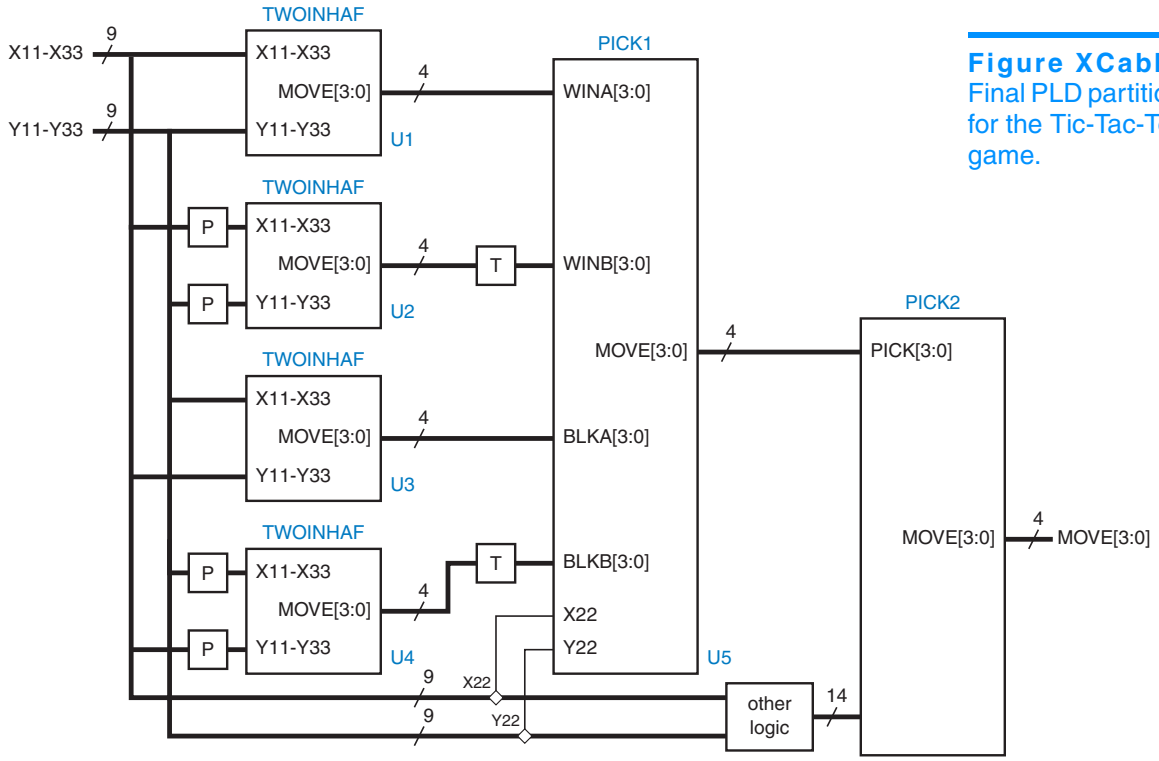

Figure XCabl-5 Final PLD partitioning for the Tic-Tac-Toe game.

Table XCabl-14 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
```

Table XCabl-15 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
```

Exercises

XCabl.1 Find a coding of the shift amounts (`S[3:0]`) and modes (`C[2:0]`) in the barrel shifter of Table XCabl-2 that further reduces the total number of product terms used by the design.

XCabl.2 Make changes to the dual-priority encoder program of Table XCabl-5 to further reduce the number of product terms required. State whether your changes increase the delay of the circuit when realized in a GAL22V10. Can you reduce the product terms enough to fit the design into a GAL16V8?

XCabl.3 Here's an exercise where you can use your brain, like the author had to when figuring out the equation for the SUM0 output in Table XCabl-11. Do each of the SUM1–SUM3 outputs require more terms or fewer terms than SUM0?

XCabl.4 Complete the design of the ABEL and PLD-based ones-counting circuit that was started in Section XCabl.6. Use 22V10 or smaller PLDs and try to minimize the total number of PLDs required. State the total delay of your design in terms of the worst-case number of PLD delays in a signal path from input to output.

XCabl.5 Find another code for the Tic-Tac-Toe moves in Table XCabl-12 that has the same rotation properties as the original code. That is, it should be possible to compensate for a 180° rotation of the grid using just inverters and wire rearrangement. Determine whether the TWOINHAF equations will still fit in a single 22V10 using the new code.

XCabl.6 Using a simulator, demonstrate a sequence of moves in which the PICK2 PLD in Table XCabl-15 will lose a Tic-Tac-Toe game, even if X goes first.

XCabl.7 Modify the program in Table XCabl-15 to give the program a better chance of winning, or at least not losing. Can your new program still lose?

XCabl.8 Modify both the "other logic" in Figure XCabl-5 and the program in Table XCabl-15 to give the program a better chance of winning, or at least not losing. Can your new program still lose?