A High-Level Design Language For Programmable Logic Devices

Kyu Y. Lee, Michael J. Holley, Mary L. Bailey, and Walter Bright, Data I/O Corp., Redmond, WA

This article describes ABEL, a design tool for programmable logic devices such as PALs, FPLAs, and PROMs. ABEL (advanced Boolean expression language) consists of a high-level hardware design language and a language processor that provides syntax checking, logic reduction, and design simulation. In this paper the structure and operation of the language processor are explained with the help of two practical examples. Actual execution times for converting a high-level description of a logic design to fuse states are given. Finally, a recently developed PLD test program generator is discussed.

Programmable logic devices such as PALs, FPLAs, and PROMs are replacing random and custom logic in a variety of applications. New devices introduced by the semiconductor manufacturers have shown increasing complexity and flexibility, with more product terms, larger numbers of input and output pins, internal feedback, internal registers, and programmable device characteristics. The complexity of many new devices approaches that of low-end gate arrays. Unfortunately, automated design tools for programmable logic have been rare and often limited to one device type. One popular tool is PALASM (Birkner and Coli, 1983), a simple language in which Boolean equations are converted to fuse maps that are used to program PALs. One drawback of PALASM, however, is that the designer must use simple Boolean equations to describe the full function of the PAL. These equations must be reduced manually using Karnaugh maps and De Morgan's theorem for the most efficient use of the product terms in the PAL. Another disadvantage to using PALASM is that it can describe designs only for PALs, and not for FPLAs or PROMs.

The Language Processor

ABEL's language processor contains six program modules that process designs described in the ABEL design language (Data I/O, 1984). The language processor takes test files created with any general-purpose editor and produces a programmer load file that can be used with a logic or PROM programmer to program a device. The ABEL language processor performs automatic logic reduction, design simulation and verification, syntax checking, and logical checking, as well as generating design documentation. The structure and operation of the language processor are discussed in detail later in this article.

ABEL is a high-level language for describing logic designs for a variety of programmable logic devices. Any combination of state diagrams, truth tables, and Boolean equations can be used to describe a design. The designer chooses the construct that best fits the logic design.

The Design Language

The syntax resembles the C programming language. As such, it includes IF-THEN-ELSE and CASE statements; logical, relational, and arithmetic operators; sets; and macros. ABEL designs are, in general, easier to read and to understand, compared to some other notations. For example, Figure 1 shows a file containing an ABEL design for a BCD to seven-segment display decoder. A truth table is used to describe the decoding function. Figure 2 shows the same design described with PALASM for comparison.

The language processor is made up of six program modules that can be executed independently or sequentially to process a logic design. The programs are PARSE, TRANSFOR, REDUCE, FUSEMAP, SIMULATE, and DOCUMENT. Each program creates an intermediate file that passes information to the next program, as shown in Figure 3.

PARSE

The PARSE program reads the input file containing a logic design and

- Checks for and reports syntax errors in the input file
- Converts state diagrams to Boolean equations
- Converts truth tables to Boolean equations
- Translates test vectors
- Expands set references to actual sets
- Expands macros
- Creates a listing file containing error messages and expanded macros, if desired
- Creates an intermediate output file used by TRANSFOR

PARSE makes sure that the input is in the correct format and prepares that input for processing by the remaining program modules. If syntax errors are found in the input file, PARSE reports the errors to the designer's terminal screen and writes error messages to the listing file. Error messages are specific and indicate the approximate location of the errors.

PARSE also processes macros. Macros let the designer specify text substitutions in a compact form. Macros can be used to describe large sets of test vectors or equations with a

```
module  bcd7          flag  -r3
title  bcd to seven segment display decoder
Data I/O Corp          Redmond WA   24 Feb 1984
"
"          ________________         ________________
"         |                |       |       a        |
"         |              a |-------|    ---------   |
"  -------| D3           b |-------|   |f       |b  |
"  -------| D2           c |-------|   |    g   |   |
"  -------| D1           d |-------|    ---------   |
"  -------| D0           e |-------|   |e       |c  |
"         |              f |-------|   |    d   |   |
"  -------| ena          g |-------|    ---------   |
"         |________________|       |________________|
"

           U6            device   P16L8 ;
           D3,D2,D1,D0   pin 1,2,3,4;
           a,b,c,d,e,f,g    pin 12,13,14,15,16,17,18;
           ena              pin9;
           bcd           = [D3,D2,D1,D0];
           led           = [a,b,c,d,e,f,g];

           ON,OFF        = 0,1; ' for common anode LEDs
           L,H,X,Z       = 0,1,.X.,.Z.;

equations
           enable led = !ena;

truth-table
    ( bcd   ->              [ a, b, c, d, e, f, g ])
       0    ->              [ ON, ON, ON, ON, ON, ON, OFF];
       1    ->              [OFF, ON, ON, OFF, OFF, OFF, OFF];
       2    ->              [ ON, ON, OFF, ON, ON, OFF, ON];
       3    ->              [ ON, ON, ON, ON, OFF, OFF, ON];
       4    ->              [OFF, ON, ON, OFF, OFF, ON, ON];
       5    ->              [ ON, OFF, ON, ON, OFF, ON, ON];
       6    ->              [ ON, OFF, ON, ON, ON, ON, ON];
       7    ->              [ ON, ON, ON, OFF, OFF, OFF, OFF];
       8    ->              [ ON, ON, ON, ON, ON, ON, ON];
       9    ->              [ ON, ON, ON, ON, OFF, ON, ON];
end    bcd7
```

FIGURE 1. BCD to seven-segment decoder written in the ABEL design language.

```
PAL16L8              PAL DESIGN SPECIFICATION
BCD7
BINARY TO 7 SEGMENT DECODER
DATA I/O REDMOND WA
D3 D2 D1 D0 NC NC NC NC ENA GND
NC A B C D E F G NC VCC

IF(/ENA)     /G =     /D3  *   /D2  *  D1
                    + /D3  *    D2  * /D1  *   D0
                    +  D3  *   /D2  * /D1
                    + /D3  *    D2  *  D1  *  /D0
                    + /D3  *    D2  * /D1  *  /D0

IF(/ENA)     /F =     /D2  *   /D1  * /D0
                    + /D3  *    D2  * /D1  *   D0
                    +  D3  *   /D2  * /D1
                    + /D3  *    D2  *  D1  *  /D0
                    + /D3  *    D2  * /D1  *  /D0

IF(/ENA)     /E =     /D2  *   /D1  * /D0
                    + /D3  *    D2  *  D1  *  /D0
                    + /D3  *   /D2  * /D0

IF(/ENA)     /D =     /D2  *   /D1  * /D0
                    + /D3  *   /D2  *  D1
                    + /D3  *    D2  * /D1  *  /D0
                    +  D3  *   /D2  * /D1
                    + /D3  *    D2  *  D1  *  /D0

IF(/ENA)     /C =     /D3  *    D1  *  D0
                    + /D2  *   /D1
                    + /D3  *    D2  * /D1  *   D0
                    + /D3  *    D2  *  D1  *  /D0
                    + /D3  *    D2  * /D1  *  /D0

IF(/ENA)     /B =     /D3  *    D1  *  D0
                    + /D2  *   /D1
                    + /D3  *   /D2  * /D0
                    + /D3  *    D2  * /D1  *  /D0

IF(/ENA)     /A =     /D3  *    D1  *  D0
                    + /D3  *    D2  * /D1  *  /D0
                    +  D3  *   /D2  * /D1
                    + /D3  *    D2  *  D1  *  /D0
                    + /D3  *   /D2  * /D0
```

FIGURE 2. BCD to seven-segment decoder written in PALASM.

few simple expressions. PARSE expands macros into the full ABEL text that they represent.

TRANSFOR

The TRANSFOR program reads an intermediate file created by PARSE and

- Replaces sets by equivalent equations without sets
- Replaces all operators with NOT, AND, OR, and XOR operators
- ORs together equations for the same output
- Performs basic logic reduction
- Creates an intermediate output file used by REDUCE
- Resolves all references to "don't cares"

A set is a group of signals defined by the designer in the original input file. TRANSFOR expands equations containing sets into equations for each signal in the set so that the equations can be reduced. Similarly, TRANSFOR replaces all logical, arithmetic, and relational operators in the equations with just four logical operators: NOT (!), AND (&), OR (#), and XOR ($).

REDUCE

The reduce program takes the Boolean equations from TRANSFOR, applies DeMorgan's theorem to convert them to sum-of-products form, and reduces the equations. Logic reduction is performed according to one of three reduction algorithms selected by the designer. The three types of reduction are simple reduction, PRESTO reduction, and PRESTO-by-pin reduction.

Simple logic reduction is performed according to the following rules of Boolean algebra:

$$!0 = 1$$
$$!1 = 0$$
$$A \,\&\, 0 = 0$$
$$A \,\&\, 1 = A$$
$$A \# 0 = A$$
$$A \# 1 = 1$$
$$A \# A = A$$
$$A \,\&\, A = A$$
$$A \,\&\, !A = 0$$
$$A \# !A = 1$$

Simple logic reduction is performed as a preliminary step to both PRESTO and PRESTO-by-pin.

PRESTO is a logic reduction algorithm developed by Antonin Svoboda that eliminates input terms and product terms (Brown, 1981). PRESTO, as implemented in ABEL, takes into account possible product term sharing among outputs. Thus, for devices such as FPLAs that can share

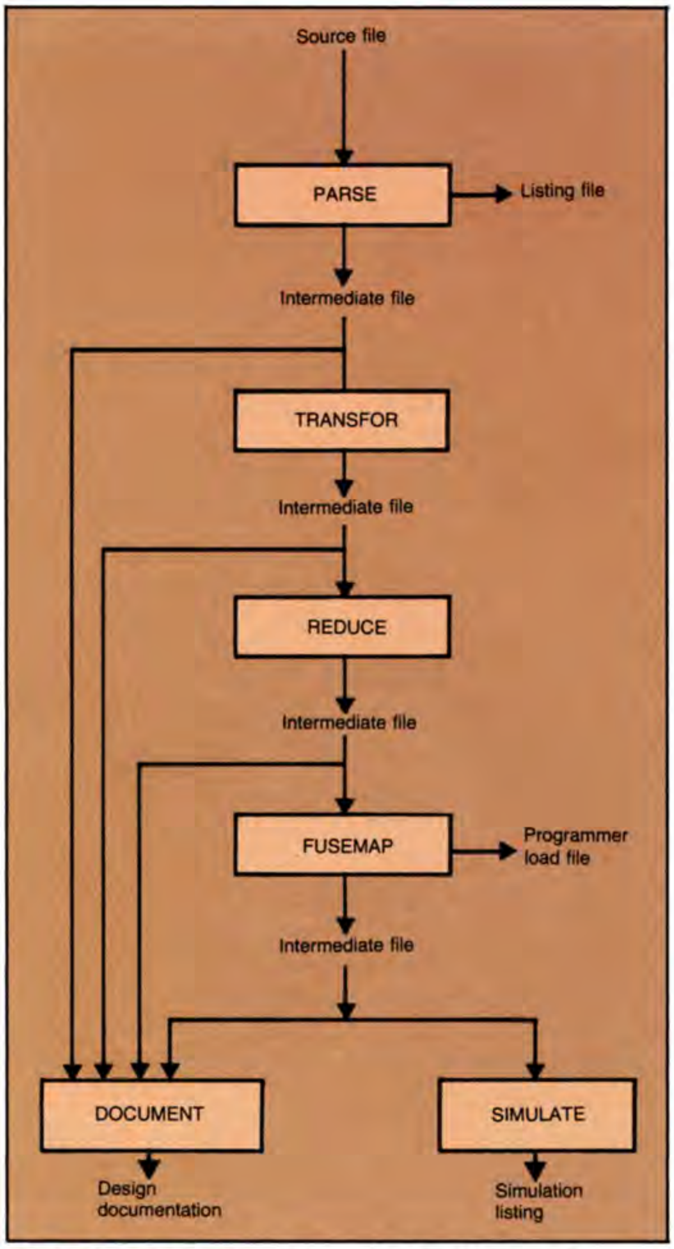

FIGURE 3. Data flow through the language processor.

product terms, ABEL can produce a set of equations that as a group use fewer product terms than if each equation were reduced independently.

PRESTO-by-pin reduction is a variation on straight PRESTO in that it deliberately does not consider product term sharing among the outputs. Each equation is reduced independent of the others to a near-minimal form. PRESTO-by-pin is faster than PRESTO and is the preferred reduction for devices such as PALs that cannot share product terms.

FUSEMAP

The FUSEMAP module produces a programmer load file from the reduced equations. These load files are produced in JEDEC (JEDEC, 1983), Motorola Exorciser, or Intellec 8/MDS formats to support a variety of logic and PROM programmers.

The programmer load file contains the required instructions to disconnect (blow) or leave connected fuses in the programmable device. This file may also contain test vectors used by the programmer to test the programmed device for correct operation. These test vectors are a translated version of those described in the ABEL source file.

FUSEMAP uses device specification files provided with ABEL to translate the reduced equations and test vectors to the fuse states that will correctly program a device. FUSEMAP also determines whether the device contains enough terms to implement the design.

One device specification file is provided for each device type supported by ABEL. Each device specification file typically supports devices from several different manufacturers. For example, the device specification file named P16R8 supports MMI, AMD, TI, and National Semiconductor 16R8 parts and the Harris 77212. ABEL currently supports over 95 devices. Simple PAL and PROM specification files are 200 to 400 bytes long (device files for more complicated devices can be up to 1.5K bytes long). All processor modules except TRANSFOR uses the specification files.

SIMULATE

The SIMULATE program uses the intermediate file produced by FUSEMAP and a device specification file to simulate programmable logic designs. SIMULATE uses an iterative solution process to simulate synchronous and asynchronous designs, and designs with feedback. The iterative process also allows the detection of unstable designs.

SIMULATE does not use the original equations contained in the source file to simulate device operation. Instead, it uses output from the FUSEMAP program, which takes device characteristics into account, and further device-specific information from the device specification file. In this way, SIMULATE can simulate operation of a specific device programmed with the design.

DOCUMENT

The DOCUMENT program uses intermediate output files from PARSE, TRANSFOR, REDUCE, FUSEMAP, and SIMULATE to create a test file containing design documentation. The design documentation includes (if the user desires) original Boolean equations and Boolean equations produced from truth tables and state diagrams; transformed equations produced by TRANSFOR; reduced equations; the number of terms used; a pictorial fusemap showing blown and connected fuses; a symbol table listing names assigned to pins and internal nodes; test vectors; and a chip diagram showing pinouts. The documentation file can be retained on disk or printed as a permanent record of the design.

Example 1

Figure 4 shows a source file for a simple design described by one Boolean equation. The design monitors four input lines, V3–V0, and produces one output Normal. Normal is high if the binary value on the input lines is less than 9 and greater than 3. The design is implemented on a 14H4 PAL. The inputs are assigned to pins 1–4, and, for simplicity, are grouped into the set named Value. The output is assigned to pin 14. Test vectors are included for simulation.

To process the design, the user starts the language proces-

```
module Range  flag "-r3"
title "Relational Operators Example"

      U1                    device "P14H4";

      V3,V2,V1,V0      pin 1,2,3,4;
      Normal                pin 14;

      Value   = [V3,V2,V1,V0];

equations
      Normal = (Value > 3) & (Value < 9);

test_vectors (Value -> Normal)
                          0  ->   0;
                          2  ->   0;
                          3  ->   0;
                          4  ->   1;
                          7  ->   1;
                          8  ->   1;
                          9  ->   0;
                         14  ->   0;
end
```

FIGURE 4. ABEL source file (RANGE) for a value comparator.

sor by typing the command ABEL RANGE. The design is processed as follows.

The source file contains the set name Value, representing V3–V0, in the Boolean equation shown here:

$$\text{Normal} = (\text{Value} > 3)\ \&\ (\text{Value} < 9)$$

PARSE replaces the set name with the elements of the set to create the following equation:

$$\text{Normal} = ([V3,V2,V1,V0] > 3)\ \&\ ([V3,V2,V1,V0] < 9)$$

TRANSFOR converts this equation to one containing no sets and only AND, OR, NOT, and XOR operators. This conversion is performed according to the following rules, stated for general A < B. A and B are sets with k elements. A_n is the nth element of set A (rightmost element is element 1) and B_n is the nth element of set B ('::=' means 'is defined as').

$$A < B ::= C_k$$

C_k is defined by iteratively applying the following two rules for n ranging from 1 to k:

$$C_0 ::= 0 \qquad \text{(used only to calculate } C_1)$$
$$C_n ::= (!A_n\ \&\ (B_n\ \#\ C_{n-1})\ \#\ A_n\ \&\ B_n\ \&\ C_{n-1}$$

Other relational operators are defined in terms of the < operator as follows:

$$A > B ::= B < A$$
$$A \leqslant B ::= !(B < A)$$
$$A \geqslant B ::= !(A < B)$$

In this example, the Boolean equation produced by PARSE is converted to:

$$\text{Normal} = (V3\ \#\ V2)\ \&\ (!V3\ \#\ V3\ \&\ !V2\ \&\ !V1\ \&\ !V0)$$

REDUCE reduces the equation using simple and PRESTO reduction to:

$$\text{Normal} = (V2\ \&\ !V3\ \#\ !V0\ \&\ !V1\ \&\ !V2\ \&\ V3)$$

FUSEMAP reads the intermediate file created by REDUCE and the device specification file associated with the 14H4

```
Vector 4
Input Reg[....................]
                          Output        First
Pin     Name              Enable        Fuse            Product Terms

14 Normal                   Y            336              T F F F
15                          Y            224              F F F F
16                          Y            112              F F F F
17                          Y              0              F F F F

OR gate                                            -> [............HLLL...]
Register                                           -> [....................]
Output                                             -> [............HLLL...]
V 0004   [010000000-000000000-]                    -> [............HNNN...]
```

FIGURE 5. Simulation output file; output for one test vector.

device and creates a programmer load file and an intermediate file. The programmer load file can be loaded into a logic programmer to program a 14H4 device. First, however, the design is simulated, and the results of the simulation should be checked.

The intermediate file created by FUSEMAP contains device information and a translated version of the test vectors contained in the original source file. SIMULATE uses this information along with information in the 14H4 device specification file to simulate operation of the programmed device. This design operates correctly in the chosen device. Figure 5 shows a portion of a simulation output file. The output shows inputs, outputs, and logic levels of internal nodes. If simulation errors occur, the output contains a message indicating the expected output and the actual output from simulation. Figure 5 shows the output for one test vector. A full simulation output file contains similar output for each test vector in the source file.

The final program, DOCUMENT, combines information created by the other five programs into one documentation file.

Example 2

Figure 6 shows a source file for an octal up/down counter. The counter is implemented on an 82S105 fuse programmable logic sequencer (Signetics, 1984) using the ABEL state diagram construct. The state diagram uses CASE statements to describe the state transitions. Each CASE statement corresponds to a state in the state machine and describes the next state for given conditions.

Figure 7 shows a portion of the logic diagram for an 82S105. Equations are written for the device's flip-flop inputs to set or reset the RS flip-flops. Note that these flip-flop inputs are not output pins of the device but are internal nodes. The following ABEL statements define signal names for these nodes so that equations can be written to control the flip-flop states:

```
P0 node 37; RP0 node 43;
P1 node 38; RP1 node 44;
P2 node 39; RP2 node 45;
```

These signals are then grouped into a set named "OUT" for easy reference. The language processes this design as follows.

PARSE reads the source file and, among the many operations it performs, converts the state diagram to Boolean

```
module cnt8  flag "-r2"
title "Octal Up/Down counter
Data I/O Corp   1   Nov 1984"

            Ck,X,Z = .C. , .X., .Z.;

            SM8       device "F82S105";

" Assign pin and node names
            Clk,Dir,Clr       pin 1,2,3;

" Internal state counter
        P0        node 37; RP0 node 43;
        P1        node 38; RP1 node 44;
        P2        node 39; RP2 node 45;

        out       = [P2,P1,P0];

" Counter modes
        Mode    = [Clr,Dir];
        Up        = [ 1 , 1 ];
        Down    = [ 1 , 0 ];
        Clear    = [ 0, X];

test_vectors (   [Clk , Mode      ]   -> out )
                 [ Ck , Clear     ]   ->   0;
                 [ Ck , Up        ]   ->   1;
                 [ Ck , Up        ]   ->   2;
                 [ Ck , Up        ]   ->   3;
                 [ Ck , Up        ]   ->   4;
                 [ Ck , Up        ]   ->   5;
                 [ Ck , Up        ]   ->   6;
                 [ Ck , Up        ]   ->   7;
                 [ Ck , Up        ]   ->   0;
                 [ Ck , Up        ]   ->   1;
                 [ Ck , Down      ]   ->   0;
                 [ Ck , Down      ]   ->   7;
                 [ Ck , Down      ]   ->   6;
                 [ Ck , Clear     ]   ->   0;

state_diagram out
    State 0:    case          (Mode = = Up)    : 1;
                              (Mode = = Down) : 7;
                              (Mode = = Clear) : 0;
                endcase;

    State 1:    case          (Mode = = Up)    : 2;
                              (Mode = = Down) : 0;
                              (Mode = = Clear) : 0;
                endcase;

    State 2:    case          (Mode = = Up)    : 3;
                              (Mode = = Down) : 1;
                              (Mode = = Clear) : 0;
                endcase;

    State 3:    case          (Mode = = Up)    : 4;
                              (Mode = = Down) : 2;
                              (Mode = = Clear) : 0;
                endcase;
    State 4:    case          (Mode = = Up)    : 5;
                              (Mode = = Down) : 3;
                              (Mode = = Clear) : 0;
                endcase;

    State 5:    case          (Mode = = Up)    : 6;
                              (Mode = = Down) : 4;
                              (Mode = = Clear) : 0;
                endcase;

    State 6:    case          (Mode = = Up)    : 7;
                              (Mode = = Down) : 5;
                              (Mode = = Clear) : 0;
                endcase;

    State 7:    case          (Mode = = Up)    : 0;
                              (Mode = = Down) : 6;
                              (Mode = = Clear) : 0;
                endcase;
end cnt8
```

FIGURE 6. An octal up/down counter source file.

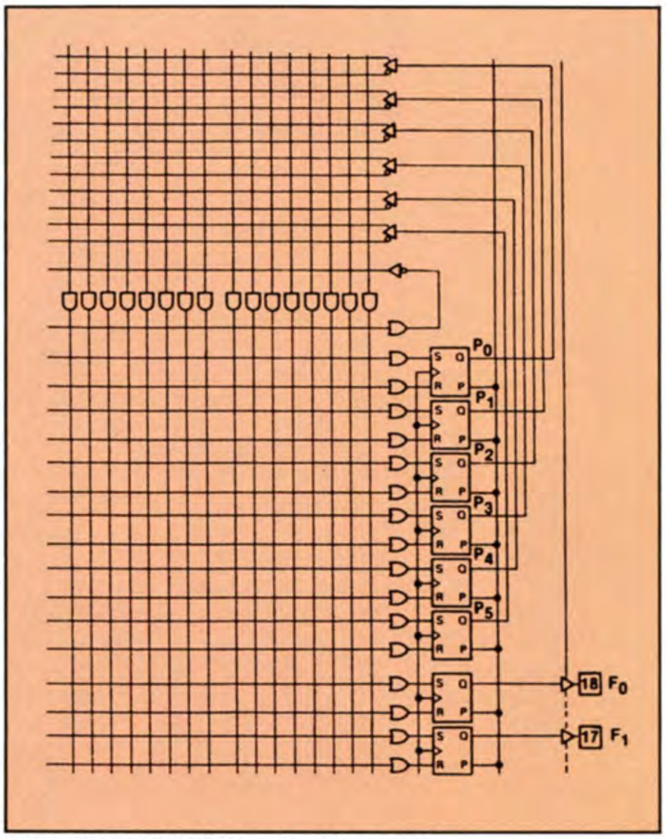

FIGURE 7. 82S105 logic diagram.

equations that contain set elements rather than set names. These equations are shown for the State 3 CASE statement in Figure 8.

As indicated in the figure, the equations created from the state diagram construct include sets of the state register signals, the current state, the counter mode, the next state, and a mask. The equations assign appropriate values for the next state to the state register signals based on the current state and counter mode. Because only equations for State 3 are shown, the current state in these equation is always 3. The counter mode is either up (Clr = 1, Dir = 1), down (Clr = 1, Dir = 0), or clear (Clr = 0).

PARSE determines which of the state register signals need to change value for a given state transition. The program generates a mask that prevents signals that do not need to change from changing, thereby saving product terms. For example, in the equation for the set flip-flop inputs for a transition from State 3 to State 4, the state register values for State 3 are [0,1,1] and the values for State 4 are [1,0,0]. To change from State 3 to State 4, all three state register signals must change. Thus the mask is [1,1,1]. This mask is ANDed with the next state and the result is that the set inputs [P2,P1,P0] are changed to [1,0,0].

However, for the transition from State 3 ([0,1,1]) to State 2 ([0,1,0]), only one of the state register signals changes value. In both the set and reset equations for this transition, only the low-order signal needs to change, so the mask is [0,0,1]. The set input, P0, for this transition should be 0, and the AND operation performed on the next state and the mask produces the correct value.

The PARSE program creates similar equations with masks

```
STATE REGISTER    CURRENT STATE          MODE                                 NEXT          MASK
SIGNALS                                                                       STATE
[P2,P1,P0]      = ([P2,P1,P0]       ==   3) & ([Clr,Dir]     ==   [1,1])   &  [1,0,0]   &   [1,1,1];
[RP2,RP1,RP0]   = ([P2,P1,P0]       ==   3) & ([Clr,Dir]     ==   [1,1])   &  [0,1,1]   &   [1,1,1];
[P2,P1,P0]      = ([P2,P1,P0]       ==   3) & ([Clr,Dir]     ==   [1,0])   &  [0,1,0]   &   [0,0,1];
[RP2,RP1,RP0]   = ([P2,P1,P0]       ==   3) & ([Clr,Dir]     ==   [1,0])   &  [1,0,1]   &   [0,0,1];
[P2,P1,P0]      = ([P2,P1,P0]       ==   3) & ([Clr,Dir]     ==   [0,.X.]) &  [0,0,0]   &   [0,1,1];
[RP2,RP1,RP0]   = ([P2,P1,P0]       ==   3) & ([Clr,Dir]     ==   [0,.X.]) &  [1,1,1]   &   [0,1,1];
```

FIGURE 8. Boolean equations for State 3 created by PARSE.

for all of the counter's state transitions. TRANSFOR reads these equations and removes the set notation. If the State 3 transitions were the only equations in the state diagram, the transformed equations would be as follows:

```
P2   =   !P2 & P1 & P0 & Clr & Dir
RP2  =   0
P1   =   0
RP1  =   !P2 & P1 & P0 & Clr & Dir
       # !P2 & P1 & P0 & !Clr
P0   =   0
RP0  =   !P2 & P1 & P0 & Clr & Dir
       # !P2 & P1 & P0 & Clr & !Dir
       # !P2 & P1 & P0 & !Clr
```

The Boolean equations produced from the state diagram include equations for every state transition. This means that there will be multiple equations for each signal. TRANSFOR ORs the equations for the same signal together to create one complete equation for each signal. For example, the full equations for P2 and P1, including the effect of all state transitions, are:

```
P2   =   !P2 & !P1 & !P0 & Clr & !Dir
       # !P2 & P1 & P0 & Clr & Dir
P1   =   !P2 & !P1 & !P0 & Clr & !Dir
       # !P2 & !P1 & P0 & Clr & Dir
       # P2 & !P1 & !P0 & Clr & !Dir
       # P2 & !P1 & P0 & Clr & Dir
```

Each OR (#) operation in the above equations combines equations created from different state transitions. Notice that the first product term in the equation for P2 is the product term from the State 3 transitions.

REDUCE reduces these equations to a near-minimal form. Because the design in this example is implemented in an 82S105 that can share product terms, regular PRESTO reduction is used. Figure 9 shows the full set of reduced equations for this design. Ten product terms are used.

Note that the first product term for P2 and P1 in the reduced equations is identical. This product term is shared by those two signals. If PRESTO-by-pin reduction is used to reduce this design, 12 product terms are used, as opposed to the 10 used with straight PRESTO. (If no reduction is performed, 23 terms are used.)

System Portability and Run Times

The ABEL software is written in the C programming language for maximum portability to different computer systems. Currently, it is implemented on the IBM PC, XT, and AT and other MS-DOS based microcomputers; VAX VMS and VAX UNIX systems; the Valid CAE workstation; and the Apollo CAE workstation.

The speed at which ABEL processes logic designs depends greatly on the processing power of the system being used. Table 1 shows typical run times on different systems. These run times include full processing by the language processor, including parsing, logic reduction, creation of the programmer load file, simulation, and creation of design documentation. The number of product terms shown for each design in the table is the final number of terms created by ABEL, after logic reduction. Table 2 shows the distribution of time among the different language processor programs for processing on a VAX 11/750.

Typically, logic reduction is the most processing-intensive and time-intensive step in design processing. The correct choice of logic reduction algorithms can have a significant impact on execution time. One startling example of this impact occurred during the design of a waveform controller board for a logic programmer. The design included two state machines implemented in a 16R8 PAL. All outputs of the PAL were used. When straight PRESTO reduction was performed on an unloaded VAX 11/750, the reduction time was one hour. PRESTO-by-pin reduction took only 35 seconds. Only 38 of the 64 available product terms were used.

In general, for PAL designs, PRESTO-by-pin reduction should be used. For FPLA designs, straight PRESTO should be used to minimize the number of product terms used.

PLDtest

After the device programmer has blown the appropriate fuses in the programmable logic device (based on the information provided by ABEL's JEDEC-format fuse map), the device must be tested. This step is necessary both to verify that the programmed device accurately replicates the original design and to make sure that the device is fully functional. Device verification testing is performed by the device programmer after programming is completed.

In testing PLDs, as in designing them, automated design tools make the design engineer's life easier. PLDtest, for example, not only performs the testing but also makes sure that the specified test vectors are sufficient to fully test the device. This step is particularly important in view of the increased complexity of current PLDs, which has made it more difficult to write test vectors that thoroughly exercise each component in a device.

Design	IBM XT MS-DOS	Valid UNIX	VAX 750 UNIX
Memory address decoder Relational equations 6 product-terms	59	33	17
Seven-segment decoder Truth table 33 product-terms	77	50	25
Black Jack machine State diagram 42 test vectors 31 product-terms	180	98	47
Decade counter D flip/flops State diagram 17 product-terms	117	65	32
Octal counter RS flip/flops State diagram 10 product-terms	160	85	43
64 state counter and 8-bit barrel shifter State diagram MEGA PAL 32R16 105 product-terms	3050	720	361

TABLE 1. Run times on various systems.

Counter—Barrel Shifter on DEC VAX 750	
PARSE	20
TRANSFOR	42
REDUCE	138
FUSEMAP	18
SIMULATE	110
DOCUMENT	14

TABLE 2. Execution times for language processor programs.

PLDtest assists the design engineer with the testing process in three ways: testability checks, fault-grading, and supplementary test vector generation. PLDtest specifies whether the design of the device allows it to be fully tested. For example, a design involving oscillating circuits cannot be completely tested because the output for any given input is unstable, oscillating between logic 1 and logic 0. Similarly, a design that includes redundant circuits cannot be fully tested, because the contribution of the redundancy may "fix" the output, so that it shows no change with a change of input. If the original design included oscillating or redundant circuitry, the design engineer will have to remove these features to produce a testable device.

Faults must be detectable at the output pins if a device is to be thoroughly tested. To fault-grade the test vectors, PLDtest first simulates operation of the design and then traces a path backward from each output pin to determine whether potential faults can be detected from that pin. The output of this step tells the designer what percentage of the total design can be verified with the test vectors he generated.

If PLDtest finds the specified test vectors insufficient to fully test the device, it will generate additional test vectors to exercise the unverified components. Note, however, that these can never replace the design-verification test vectors supplied by the design engineer, because PLDtest has no way of knowing the intended function of the device. All the software can do is to make sure that the design, as specified, is correctly implemented in the device, and that the number of test vectors is sufficient to detect all possible faults.

```
P2    =     Clr & !Dir & !P0 & !P1 & !P2
          # Clr &  Dir &  P0 &  P1 & !P2;

P1    =     Clr & !Dir & !P0 & !P1 & !P2
          # Clr & !Dir & !P0 & !P1 &  P2
          # Clr &  Dir &  P0 & !P1;

P0    =     Clr & !P0;

RP2   =     Clr & !Dir & !P0 & !P1 &  P2
          # !Clr &                    P2
          #         Dir &  P0 &  P1 &  P2;

RP1   =     Clr &  Dir &  P0 &  P1 & !P2
          #        !Dir & !P0 &  P1
          #         Dir &  P0 &  P1 &  P2
          # !Clr &               P1;

RP0   =     P0;
```

FIGURE 9. Reduced equations for Example 2.

Running PLDtest consists of specifying the input file (fuse map), the output file (to which the trace output is to be written), the desired format for the output file (e.g., JEDEC), and the type of device being tested. At the end of the run, the software provides a report with the following information:

- The number of iterations performed to test the device. For each iteration, it lists the number of detected and undetected faults (that is, those that were known to exist but could not be detected at the output pins). It also specifies the percentage of fault coverage provided.
- A detailed list of each of the detected faults, specifying, in each case, the test vectors used to discover them.
- A detailed list of each of the known but undetected faults, specifying both the fuse number and the type of fault, so that the designer can go back and correct the problem.
- A list of any faults that are totally undetectable due to the nature of the design (e.g., a design that includes oscillating or redundant circuits). The designer can use this information to correct the design so the device will be fully testable.

Conclusion

ABEL supports design efforts from conception to implementation in programmable logic. Because the same source file can be used to program devices from different families, ABEL offers the designer flexibility. ABEL helps to produce error-free and efficient designs for direct implementation in a device. Automatic logic reduction and simulation eliminate tedious manual methods of reducing the number of logic terms required by a design, and they provide verification that the reduced design will work as expected. □

Acknowledgments

The authors would like to extend thanks to Gerrit Barrere, Bjorn Benson, Brenda French, Ngoc Nicholas, Charles Olivier, and Dave Pellerin for their contribution to the ABEL project, and to the personnel at the Beta test sites, who provided valuable input and suggestions in the early stages of the project. The authors would also

like to thank Daniel Burrier for his assistance in preparing the manuscript.

References

Birkner, J.M., and V.J. Coli. 1983. *PAL Programmable Array Logic Handbook, 3d ed.*, Monolithic Memories Inc., Santa Clara, CA.

Brown, D.W. 1981. "A State Machine Synthesizer—SMS," *18th Design Automation Conference*, Nashville, TN.

Data I/O Corp. 1984. *ABEL User's Manual.*

JEDEC Solid State Products Engineering Council. October 1983. *JEDEC Preparation System and Programmable Logic Device Programmer.*

Kang, S. 1981. *Minimizing, Partitioning, and Synthesis of Programmable Logic Arrays*, Ph.D. dissertation, Stanford University, Stanford, CA.

Kang, S., and W.M. van Cleemput. 1981. "Automatic PLA Synthesis from a DDL-P Description," *18th Design Automation Conference.*

Signetics Corp. 1984. *IFL Integrated Fuse Logic Data Manual.*

About the Authors

Kyu Y. Lee is the vice president of engineering with Butler Controls in Kirkland, WA. Prior to taking this position, he was director of software engineering and manager of design automation products at Data I/O, responsible for the development and marketing of ABEL and other design automation products. Before that, he was director of the software engineering program at Seattle University, software development manager at EG&G, and systems programmer at Fermilab. He received his B.S. from Seoul National University and his Ph.D. from Indiana University.

Michael J. Holley is the project engineering manager for logic tools at Data I/O. Currently he is developing new CAE tools for programmable logic devices and other semicustom integrated circuits. He is also a member of the JEDEC subcommittee for solid state memories and programmable logic. He received his B.S.E.E. from Seattle University.

Mary L. Bailey holds a B.A. in mathematics and physics from Vanderbilt University and a M.A. from the University of Washington. Currently she is a principal software engineer at Data I/O Corp., where she is working on logic design tools, specializing in logic reduction. She is enrolled in the Ph.D. program in computer science at the University of Washington. Her research area is parallel algorithms and architecture.

Walter G. Bright is a senior software engineer at Data I/O. He received his B.S. in engineering and applied science from the California Institute of Technology. His principal interests are in software tools for digital circuit design. His other interests are in compiler design and implementation.

The ***STM 4000*** *is a fully integrated digital function tester. Available at PC prices.* ***Perfect for incoming inspection*** *of digital ICs. Lets you automate inspection and reject bad ICs. Test MICROs, EPROMs, RAMs, ROMs, PALs, PLAs, PLDs, UARTs, 7400s, and custom/semi-custom VLSI parts. Performs pass/fail tests, as well as detailed failure analysis.*

Easy to use. *No special skills required.* ***Powerful menu driven software eliminates programming.*** *The* ***STM 4000*** *can directly drive your autoprobe/autofeed stations with no additional equipment required.*

Complete 7400 test library, vector generators, and adapters are available. Price for 64 channels and software to meet all of your basic needs is $4895.

Available now from ***CADIC Inc.,*** *Beaverton, Oregon. 1 (800) 824-1617 or (503) 626-7902.*

CIRCLE NUMBER 34

Here's the programmer you've been waiting for!

The new OMNI 64 has **10A, 80V** software configured bipolar pin drivers **on every pin.** Pin drivers so fast and accurate that they can program every **EPROM, PROM, PAL, FPLA, or single chip uP** manufactured today without a single family pack or configuration module!

The OMNI 64 is so flexible it can call the factory by modem and update itself with new programming algorithms within minutes!

And best of all, the OMNI 64 is plug compatible with over 300 different computers and operating systems.

Isn't it time to stop throwing good money after bad on a never ending list of configuration modules, pinout adapters and family packs? Call (818) 240-0080 now for a free demonstration of the **fastest** software configured programmer on the market or for a free copy of the "Universal Programmer Buyer's Guide".

OAE

Oliver Advanced Engineering, Inc.
676 West Wilson Avenue
Glendale, California 91203
818-240-0080, Telex: 194773

CIRCLE NUMBER 35