Introducing PLDs *(Programmable Logic Devices)*

The Logical Devices CUPL Design Tools are for devices such as PALs and PROMs. The set contains editors, processors, and transfer tools. This material describes the CUPL language and how to apply it.

COMPONENTS

Programmable logic devices are "blank" chips that you can customize in your own laboratory without elaborate equipment. From modest origins in the 1970s, programmable logic devices have evolved steadily and now occupy a niche in almost every digital design. The compiler creates detailed logic for these devices from your high-level specifications. You choose the best way to describe your design from methods that include Boolean equations, indexed equations, numerical maps, state machine procedures, truth tables, bit streams, and schematics. In many cases, the compiler takes your behavioral definitions and synthesizes the detailed logic for you.

After a device is programmed, it must be tested to make sure the device and the logic are not faulty. On most programming machines, the device is fed a series of test vectors. Making up a complete set of test vectors by hand can be a brutal experience. A logic tester eases the effort. For combinational logic, it automatically generates all input combinations for signals you specify, then simulates the corresponding output signals, and finally writes test vectors to a file ready to be sent to the device programming machine. For clocked sequential logic, it lets you step through clock cycles. You set and clear input signals interactively while the logic tester works out the corresponding test vectors.

HOW TO USE THIS HANDOUT

Throughout, this approach is taken: First a topic is introduced with some basic examples to give you an intuitive feeling for the subject. Next the subject is expanded more systematically. While a systematic description is essential, I think that introductions are usually best with examples. I'm also working hard not to use up quite so many trees.

Please don't think that you have to read all the way through before using the design tools. As you learn something new, you should go to a computer and try it. And if you don't need certain advanced features, you shouldn't feel compelled to study them. (Yet.)

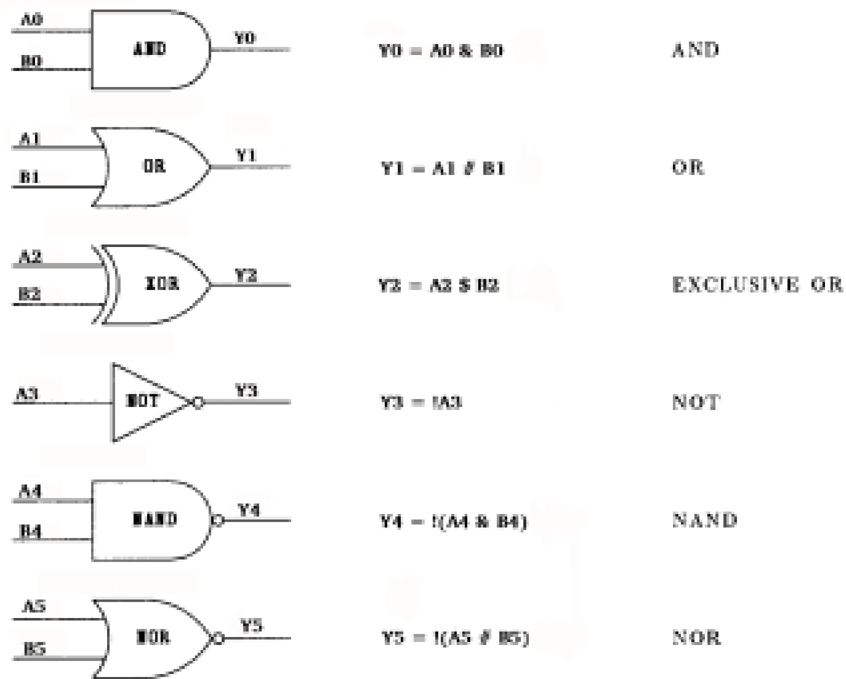
Boolean equations

The starting point for hardware definitions in the CUPL language involves Boolean equations. These are named in honor of Georges Boole, who, in the nineteenth century undertook an analysis of logic, exposing the rules for the now familiar ANDs and ORs of computing machinery. Boolean equations are the simplest and most universal format in the CUPL language, though they are by no means the best for all problems.

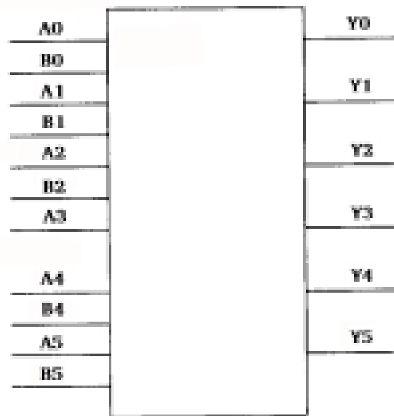
LOGIC GATES

To provide a sound starting point, some of the material introduced in the user's guide is reviewed here. On the left below are schematic symbols, on the right are the corresponding equations written in the CUPL language. In schematics, signals are represented by lines, gates are represented by graphic symbols. In the CUPL language, signals are represented by names, gates are represented by special characters. An AND gate is represented by an ampersand (&), an OR gate is represented by a number sign (#), and EXCLUSIVE-OR by a dollar sign ($). Logical NOT is shown by a exclamation point (!) preceding the signal to be inverted. The exclamation point is analogous to the small circle following a gate in schematics to show inversion. These two terms can be used almost interchangably. Parentheses are used for grouping, as in the NAND and NOR gates shown below.

Sometimes it is more conventient to refer to symbols by their computer science shorthand. For instance, an exclamation point is referred to as "bang". Similarly, the number sign is often called "pound". These shorter terms will be used interchangably with the proper names.



Suppose we want to make a single device that contains each of these basic gates. In other words, we



want a chip with the following schematic symbol.

We start by collecting all six equations together.

Y0 = A0 & B0
Y1 = A1 # B1
Y2 = A2 $ B2

Y3 = !A3
Y4 = !(A4 & B4)
Y5 = !(A5 # B5)

These equations define the logic, but they do not describe what device to use, nor do they describe how to connect the signals to pins. If we had a specific device in mind, we would write the name of the device before the equations, then describe how the signal names relate to pins on the device. At this point in the discussion, we do not care about a specific device, so we just list the signals, showing which are inputs and which are outputs. Since there always needs to be a header segment for CUPL to compile, we substitue the device "virtual".

The header segment is the first nine lines of any CUPL file. The only fields we will worry about are the Name and the Device. These fields determine the output file name and the device to be used. They can however be overridden by picking differed options from the menu in WinCUPL.

To create a new file, open WinCUPL and Select a New Design File from the File menu. This will give you a dialog box for the header information. After filling in the relevant information, hit enter. WinCUPL will then prompt you for the number of inputs, outputs, and pinnodes (named spots inside the device which are not accessible from pins but allow internal connections). In this example we choose 12 inputs, 6 outputsl and 0 pinnodes. The code WinCUPL generates is shown below.

```
Name     handout1 ;
PartNo   01 ;
Date     11/7/99 ;
Revision 01 ;
Designer Engineer ;
Company  Cooper Union ;
Assembly None ;
Location  ;
Device   virtual ;

/* *************** INPUT PINS *********************/
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */

/* *************** OUTPUT PINS *********************/
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
PIN     =                         ; /*                                  */
```

We then fill in the appropriate fields to create the logical functions we require. Note that comments are inserted in CUPL the same way as in C. They open with a /* and close with a */ They cannot however be nested. Also note that the sections which specify Input pins and Output pins are defined by comments. What this means is that the Input Pin section and the Output Pins sections are defined for the user only. CUPL actually determines which pin is an output or input based on its usage.

```
Name     handout1 ;
PartNo   01 ;
Date     11/7/99 ;
Revision 01 ;
Designer Engineer ;
Company  Cooper Union ;
Assembly None ;
Location  ;
Device   virtual ;

/* *************** INPUT PINS *********************/
PIN     =  A0                    ; /*                                  */
PIN     =  A1                    ; /*                                  */
PIN     =  A2                    ; /*                                  */
PIN     =  A4                    ; /*                                  */
PIN     =  A5                    ; /*                                  */
PIN     =  [B0..B5]        ; /*                                  */

/* *************** OUTPUT PINS *********************/
PIN     =    Y0                  ; /*                                  */
PIN     =    Y1                  ; /*                                  */
PIN     =    Y2                  ; /*                                  */
PIN     =    Y3                  ; /*                                  */
PIN     =    Y4                  ; /*                                  */
PIN     =    Y5                  ; /*                                  */

/* ***** Logic Equations ***** */

Y0 = A0 & B0;
Y1 = A1 # B1;
Y2 = A2 $ B2;
Y3 = !B3;
Y4 = !(A4 & B4);
Y5 = !(A5 # B5);
```

This specification is referred to as source code, a term borrowed from software technology. There are certain minor things you must be careful about when writing source code. For example, every line (to be interpretted as a CUPL command - macros are different) must end in a semi colon.

The format for declaring pins is the keyword PIN, followed by the pin # (which is omitted here since we are using the virtual device) then then equals sign and finally the variable by which the pin will be referred to. Pin names can be specified on single lines (the A inputs) or in a set (the B inputs). Signal names in brackets must be separated by commas or elipses. This is known list notation in CUPL. Commas are used for indexed variables which are no continuous (i.e. [A0,A2,A5]) and elipses (only two dots - ..) are used to specify continous ranges. (i.e. [A0..A3]). Also note that CUPL does not care about the order in which the indexed variables are specified. It always uses them in increasing order. (i.e. [A0,A1,A2] is equivalent to [A2,A0,A1]).

Following these and other syntax rules will become natural as you use the language. You will see more short cuts including lists in later examples.

GENERATED OUTPUT FILES

When we submit the specification from the previous section to the CUPL compiler, we get several reports. Which output files are generated are configurable from the Compile Options dialog box. It is necessary to select the list output file in order to generate the .doc file which we will examine next.

```
 *******************************************************************************
                                   handout1
*******************************************************************************

CUPL(WM)        5.0a Serial# 70044995
Device          virtual  Library DLIB-h-40-1
Created         Sun Nov 07 20:44:43 1999
Name            handout1
Partno          01
Revision        01
Date            11/7/99
Designer        Engineer
Company         Cooper Union
Assembly        None
Location

===============================================================================
                            Expanded Product Terms
===============================================================================

Y0 =>
    A0 & B0

Y1 =>
    A1
  # B1

Y2 =>
    A2 & !B2
  # !A2 & B2

Y3 =>
    !B3

Y4 =>
    !A4
  # !B4

Y5 =>
    !A5 & !B5


===============================================================================
                                 Symbol Table
===============================================================================

Pin Variable                                    Pterms   Max     Min
Pol   Name              Ext     Pin     Type     Used   Pterms  Level
--- --------            ---     ---     ----    ------  ------  -----

    A0                          0        V        -       -       -
    A1                          0        V        -       -       -
    A2                          0        V        -       -       -
    A4                          0        V        -       -       -
    A5                          0        V        -       -       -
    B0                          0        V        -       -       -
    B1                          0        V        -       -       -
    B2                          0        V        -       -       -
    B3                          0        V        -       -       -
    B4                          0        V        -       -       -
    B5                          0        V        -       -       -
    Y0                          0        V        1       0       1
    Y1                          0        V        2       0       1
    Y2                          0        V        2       0       1
    Y3                          0        V        1       0       1
    Y4                          0        V        2       0       1
    Y5                          0        V        1       0       1


LEGEND    D : default variable         F : field      G : group
```

```
I : intermediate variable    N : node        M : extended node
U : undefined                V : variable    X : extended variable
T : function
```

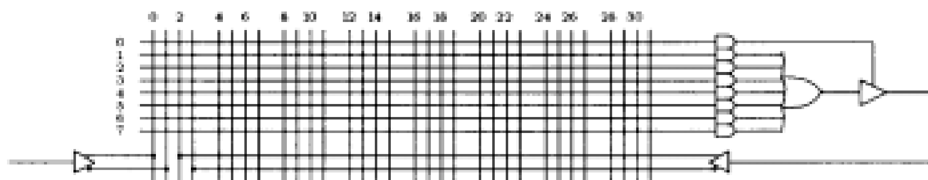
The original equations have been converted by the compiler to sum-of-products form, which is the form used by most programmable logic devices. A typical sum-of-products equation has several rows of signals. In each row, all the signals are multiplied together logically (AND gates) to form a "product," then the results for every row are added together logically (OR gates) to form a "sum." Hence the term "sum-of-products." *All* digital functions can be written in sum-of-products form, given enough terms.

And gate

Let's take the equations one at a time, starting with the AND gate Y0 = A0 & B0. At the beginning of the report, we see what the compiler does with Y0.

```
Y0 =>
    A0 & B0
```

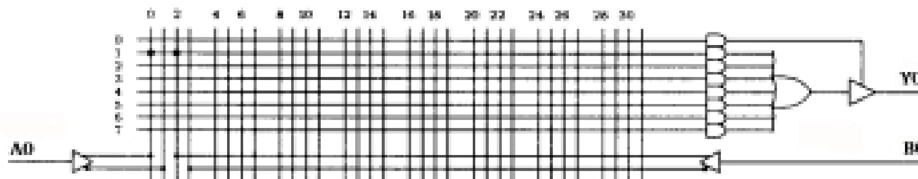
In this simple case, there is only one line for the equation, and it contains the signals A0 and B0. This single line can be taken to represent junctions in a logic diagram of the programmable device. As shown in the manufacturers' data books, such diagrams typically have input signals fanning out to an array along a series of vertical columns, while output signals are connected to AND and OR gates fed by a series of horizontal rows. Connections among the inputs and outputs are made with junctions between columns and rows. Here is an excerpt showing an unprogrammed output signal from the diagram of a hypothetical device.



Two input signals are shown in this portion of the diagram, one at the left permanently connected to columns 0 and 1, the other at the right permanently connected to columns 2 and 3. Typical of programmable devices, an inverted form of each signal is connected to the odd numbered column. Each horizontal row in this diagram is connected to a 32-bit wide AND gate, represented by small AND symbols at the right end of the row. The output of seven of these AND gates feed a seven-input

OR gate, connected in turn to a three-state driver. The output of the remaining AND gate enables the driver. This diagram, while not representing any specific device, is typical of programmable devices like PALs. Further information on the diagrams and examples for actual devices are found in manufacturers' data books.

Now, the single line printed by the compiler for Y0 = A0 & B0 is represented in the diagram by junctions connecting signals A0 and B0 to row 1, as below. The three-state enable on row 0 has no junctions, making it permanently enabled. In this way Y0 becomes A0 ANDed with B0, as specified in the source.

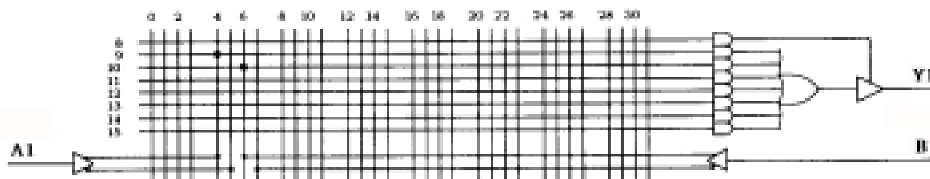


Or gate

Next we specified an OR gate by writing Y1 = A1 # B1.

```
Y1 =>
    A1
  # B1
```

Now only one signal appears on each row, but there are two rows. Since all rows are ORed together, we see that Y1 is equal to A1 ORed with B1, again as specified:



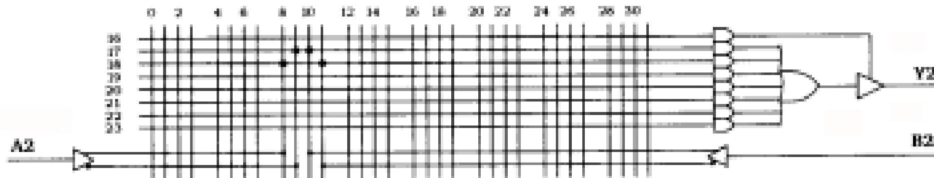
Exclusive-or gate

Third we specified an EXCLUSIVE-OR gate by writing Y2 = A2 $ B2.

```
Y2 =>
```

```
   A2 & !B2
 # !A2 & B2
```

Here are two rows with two signals per row. The first row says that the complement of A2 should be ANDed with B2, while the second row says that A2 should be ANDed with the complement of B2.



Since all rows are ORed together, the two lines from the report are the same as the following equation.

Y2 = (!A2 & B2) # (A2 & !B2)

The compiler has created an equivalent AND/OR form. The table below shows individual product terms that make up the result. The rightmost column, which corresponds to the result, is the truth table for EXCLUSIVE-OR.

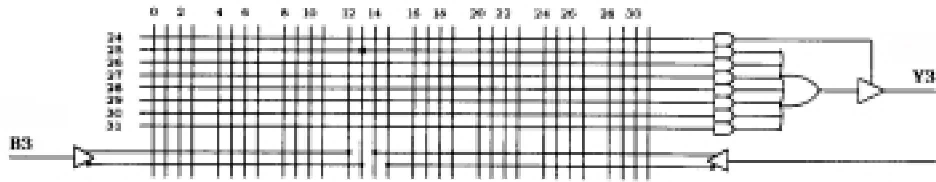
A2	B2	!A2&B2	A2&!B2	(!A2&B2)#(A2&!B2)
0	0	0	0	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

Inverter

Fourth we specified an inverter by writing Y3 = !A3.

```
Y3 =>
    !B3
```

Here the result is just a simpler version of AND and OR, with only one signal in one product term connected.

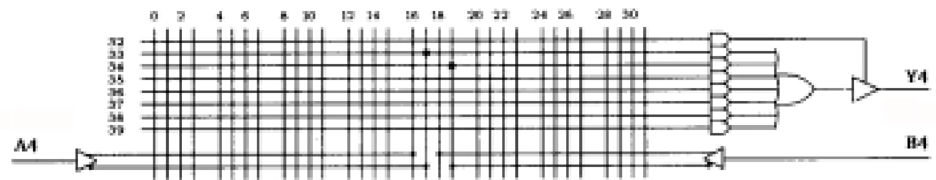


Nand gate

Fifth we specified a NAND gate by writing Y4 = !(A4 & B4).

```
Y4 =>
    !A4
  # !B4
```

The operation NAND is simply the complement of AND. First both input signals are ANDed together, then the result is complemented, as written originally. However, the result is a complement of products, not a sum of products and does not match the typical structure of programmable devices. Here we see that the compiler has first complemented both signals, then ORed the results together. It as applied a De'Morgan equivalence rule, recognizing that !(A4 & B4) is the same as (!A4 # !B4).

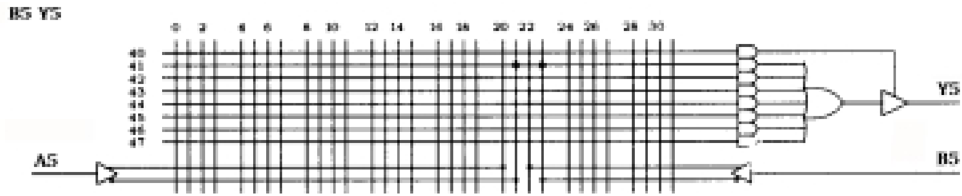


Nor gate

Finally we specified a NOR gate by writing Y5 = !(A5 # B5)

```
Y5 =>
    !A5 & !B5
```

The OR operation has disappeared and has been replaced with an AND operation, since !(A5 # B5) is the same as (!A5 & !B5), again by De'Morgan equivalence .



Symbol Table

The Symbol Table is at the end of the .doc file. It gives information about the resources used. The interresting thing to note in this example is the Column "Pterms used". This is a counter for the number of products used in generating each sum term. Pterm is short for partial terms.

AN ACTUAL DEVICE

Before creating an actual part, we name the device. The equations have six outputs and eleven inputs, with at most two product terms per output, so a device as simple as a PAL12H6 can be used. That device has six outputs and twelve inputs, with two or more product terms on every output. We add the device name at the beginning of the source code.

To find out which devices are available click on Compiler under the Options menu and select the device tab. The p12h6 is under the device type "PAL 6 - 14". To find out the pinouts and a description, click on the "Find Mneumonic" button. This will pop up a new window. Press the Find button and type "p12h6" in the window. This will lead you to the section of the file which describes the PAL 12H6. For instance, the mneumonic file gives the following for the p12h6.

```
%P12H6
P12H6 Architecture
Mnemonic: P12H6              PLCC Mnemonic: NO
DIP Pin Count: 20            Total Product Terms:   16
Extensions:
============================================================
Manufacturer                 Device Name
------------                 -----------
AMD/MMI                      PAL12H6/-2
LATTICE                      RAL12H6
NATIONAL                     PAL12H6/16V8
NATIONAL                     PAL12H6/A/A2
SGS-THOM.                    RAL12H6
============================================================
Clock Pin(s):                Common OE(s):
VCC(s): 20                   GND(s): 10
Input Only: 1  2  3  4  5  6  7  8  9  11  12  19
Output Only: 13  14  15  16  17  18
Input/Output:
%%
```

Based on this knowledge we can edit our source code to match the selected device.

```
Name     handout1 ;
PartNo   01 ;
Date     11/7/99 ;
Revision 01 ;
Designer Engineer ;
Company  Cooper Union ;
Assembly None ;
Location  ;
Device   p12h6 ;

/* *************** INPUT PINS *********************/
PIN 7   = A0                   ; /*                                 */
PIN 8   = A1                   ; /*                                 */
PIN 9   = A2                   ; /*                                 */
PIN 12  = A4                   ; /*                                 */
PIN 19  = A5                   ; /*                                 */
PIN [1..6] = [B0..B5]          ; /*                                 */

/* *************** OUTPUT PINS *********************/
PIN 13  = Y0                   ; /*                                 */
PIN 14  = Y1                   ; /*                                 */
PIN 15  = Y2                   ; /*                                 */
PIN 16  = Y3                   ; /*                                 */
PIN 17  = Y4                   ; /*                                 */
PIN 18  = Y5                   ; /*                                 */

/* ***** Logic Equations ***** */

Y0 = A0 & B0;
Y1 = A1 # B1;
Y2 = A2 $ B2;
Y3 = !B3;
Y4 = !(A4 & B4);
Y5 = !(A5 # B5);
```

When this is compiled, the results are similar, except that the actual device characteristics appear in the listings. For example, the maximum number of partial terms available for each output are listed in the symbol table.

```
*******************************************************************************
                                   handout1
*******************************************************************************

CUPL(WM)        5.0a Serial# 70044995
Device          p12h6  Library DLIB-h-40-9
Created         Mon Nov 08 16:10:34 1999
Name            handout1
Partno          01
Revision        01
Date            11/7/99
Designer        Engineer
Company         Cooper Union
Assembly        None
Location

===============================================================================
                            Expanded Product Terms
===============================================================================

Y0 =>
    A0 & B0

Y1 =>
    A1
  # B1

Y2 =>
    A2 & !B2
  # !A2 & B2

Y3 =>
    !B3

Y4 =>
    !A4
  # !B4

Y5 =>
    !A5 & !B5


===============================================================================
                                 Symbol Table
===============================================================================

Pin Variable                                    Pterms   Max     Min
Pol   Name              Ext     Pin     Type     Used   Pterms  Level
--- --------            ---     ---     ----    ------  ------  -----

    A0                          7        V        -       -       -
    A1                          8        V        -       -       -
    A2                          9        V        -       -       -
    A4                          12       V        -       -       -
    A5                          19       V        -       -       -
    B0                          1        V        -       -       -
    B1                          2        V        -       -       -
    B2                          3        V        -       -       -
    B3                          4        V        -       -       -
    B4                          5        V        -       -       -
    B5                          6        V        -       -       -
    Y0                          13       V        1       4       1
    Y1                          14       V        2       2       1
    Y2                          15       V        2       2       1
    Y3                          16       V        1       2       1
    Y4                          17       V        2       2       1
    Y5                          18       V        1       4       1


LEGEND    D : default variable         F : field      G : group
```

```
I : intermediate variable    N : node       M : extended node
U : undefined                V : variable   X : extended variable
T : function
```

INVERTED POLARITY

The PAL12L6 is identical to the PAL12H6, except that all signals leaving the chip are complemented on the PAL12L6. That is, in the PAL12L6, NOR gates are used instead of OR gates. It is instructive to see what happens when we submit the basic equations to the *PLD* compiler and tell it to use a PAL12L6. Only one letter in the specification must change (the h in the device must become an l).

```
Name      handout1 ;
PartNo    01 ;
Date      11/7/99 ;
Revision 01 ;
Designer Engineer ;
Company  Cooper Union ;
Assembly None ;
Location  ;
Device   p12l6 ;

/* *************** INPUT PINS *********************/
PIN 7    = A0                   ; /*                                  */
PIN 8    = A1                   ; /*                                  */
PIN 9    = A2                   ; /*                                  */
PIN 12   = A4                   ; /*                                  */
PIN 19   = A5                   ; /*                                  */
PIN [1..6] = [B0..B5]           ; /*                                  */

/* *************** OUTPUT PINS *********************/
PIN 13   = Y0                   ; /*                                  */
PIN 14   = Y1                   ; /*                                  */
PIN 15   = Y2                   ; /*                                  */
PIN 16   = Y3                   ; /*                                  */
PIN 17   = Y4                   ; /*                                  */
PIN 18   = Y5                   ; /*                                  */

/* ***** Logic Equations ***** */

Y0 = A0 & B0;
Y1 = A1 # B1;
Y2 = A2 $ B2;
Y3 = !B3;
Y4 = !(A4 & B4);
Y5 = !(A5 # B5);
```

Now the results look a little different (only the Expanded Product Terms section is given in an attempt to save some paper). On the left, each output signal is marked with a bang (!), indicating that the output is complemented, while on the right, the sum of products has changed.

```
===============================================================================
                            Expanded Product Terms
===============================================================================

!Y0 =>
    !A0
```

```
  # !B0
!Y1 =>
    !A1 & !B1

!Y2 =>
    A2 & B2
  # !A2 & !B2

!Y3 =>
    B3

!Y4 =>
    A4 & B4

!Y5 =>
    A5
  # B5
```

The first equation shows what has happened. Since the output pins on the PAL12L6 are connected to inverters, the output is always complemented. So instead of sending Y0 to the output pin, the device sends the inverted signal Y0. To compensate for the inversion of the signal, the *PLD* compiler **inverts** the entire equation. First, the compiler converts from:

Y0 = A0 & B0
to:
!Y0 = !(A0 & B0)

These two equations are equivalent, since complementing both the input and the output does not change the result. (In other words, performing the same operation on both sides of an equation does not change the equality.) Now the compiler applies a De Morgan equivalence rule to the right hand side and obtains

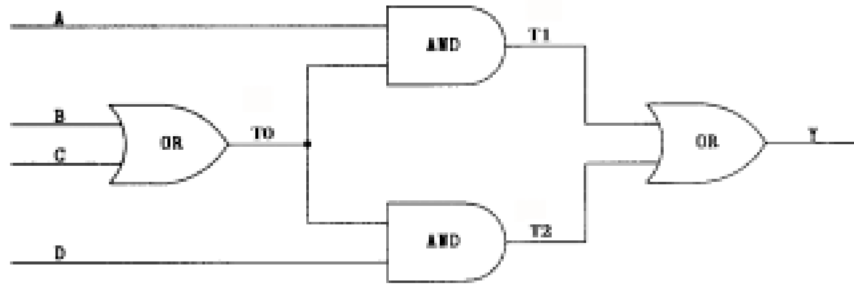
!Y0 = !A0 # !B0

which is what appeared in the sum-of-product terms. The other equations are similar.

Related to this kind of inverted polarity is the topic of *active-high* and *active-low signals.* The *PLD* compiler lets you declare which signals are active-high and which are active-low, eliminating the need for you to work out the details of where signals must be complemented. The topic is fundamental to the proper use of hardware languages. To declare an input or output as active low, simply define the PIN declaration statement with a bang before the symbol name.

COMBINATIONS OF GATES

The basic gates combine to form more complicated functions. Here is a simple schematic showing three levels of logic



We can write this as a set of basic gates by starting at the left and using the temporary signal names T0, T1, and T2.

```
Device   virtual ;

/* *************** INPUT PINS *********************/
PIN      =  A                        ; /*                                   */
PIN      =  B                        ; /*                                   */
PIN      =  C                        ; /*                                   */
PIN      =  D                        ; /*                                   */

/* *************** OUTPUT PINS *********************/
PIN      =  Y                        ; /*                                   */

/* ****Declarations and Intermediate Variables**** */
T0= B # C;              /* First Intermediate Variable */
T1= A & T0;             /* Second */
T2= D & T0;             /* Third */

/* Logic Functions */

Y= T1 # T2;
```

Please note that only the relevant lines of the source file are included. This practice will be adopted for the rest of the handout in order to save paper.

We could submit this specification to the *PLD* compiler just as it appears above. Since the signals T0, T1, and T2 are not named in the input and output lists, the compiler knows that they represent intermediate gates and not real signals. It is typical, however, and usually better, to combine the gates and eliminate the temporary signals. We can proceed step by step, starting with temporary signal T0.

```
/* ****Declarations and Intermediate Variables**** */
T0= B # C;              /* First Intermediate Variable */
T1= A & T0;             /* Second */
T2= D & T0;             /* Third */
/* Logic Functions */
Y= T1 # T2;
```

We take the right-hand side of the first equation, B#C, place it in parentheses, and substitute it in the two places it occurs. This substitution removes one line.

```
/* ****Declarations and Intermediate Variables**** */
T1= A & (B # C);               /* Second */
T2= D & (B # C);               /* Third */
/* Logic Functions */
Y= T1 # T2;
```

Next we take the right-hand side of the next equation, A&(B#C), enclose it in parentheses in the same way, and substitute it in the one place it occurs. The form

```
/* ****Declarations and Intermediate Variables**** */
T1= A & (B # C);               /* Second */
T2= D & (B # C);               /* Third */
/* Logic Functions */
Y= T1 # T2;
```

leads to

```
/* ****Declarations and Intermediate Variables**** */
T2= D & (B # C);               /* Third */
/* Logic Functions */
Y= (A & (B # C)) # T2;
```

Another line has been removed. Finally we do the same for T3 and get a single-line equation. The form:

```
/* ****Declarations and Intermediate Variables**** */
T2= D & (B # C);               /* Third */
/* Logic Functions */
Y= (A & (B # C)) # T2;
```

leads to

```
/* Logic Functions */
Y= (A & (B # C)) # (D & (B # C));
```

Then dropping the bolded portion, the final form becomes

```
/* Logic Functions */
Y= (A & (B # C)) # (D & (B # C));
```

Whether the shorter form is clearer depends somewhat on personal taste. In this case, it may be better to leave T0 as a temporary signal, since T0 is used in two places. If we keep T0, we can shorten its name to T and write the specification a little differently.

```
/* Declarations and Intermediate Variables */
T = B # C;
/* Logic Functions */
Y= (A & T) # (D & T);
```

However we write the specification, the *PLD* compiler produces the same result. By submitting this

specification, we find that the sum of products has four rows, which you can verify at your leisure.

```
===============================================================================
                             Expanded Product Terms
===============================================================================

T =>
    B
  # C

Y =>
    A & B
  # A & C
  # B & D
  # C & D
```

.

OPERANDS

To state the obvious, an *operand is* something that is operated on, whereas an *operator is* what does the operating. For example, if two signals are ANDed together, the signals are the operands and the AND gate is the operator. This terminology is common in math and computer languages, and it applies to the *CUPL* hardware language *too*.

Names and numbers

In the *CUPL* language, signal names can start with letter A through z (CUPL is case sensitive), a numeric digit or an underscore, but it must contain one alpanumerica character (so that it is not interpretted as a number). Variables can contain up to 32 characters, but a space cannot be one of them. Variables cannot contain any CUPL reserved symbols or be the same as any CUPL reserved keywords.

A	*Perhaps too short*
A0	*The first signal in a set*
A1B2C3	*A strange name, but legitimate*
ACKNOWLEDGEMENT	*Inconveniently long*
ACK	*A better abbreviation*
CATCH22	*The 22nd signal of a 32-bit bus?*
CATCH22A	
ZEN	
_and_the_art_of	
MOTORCYCLE	
Maintenance	

Here are some examples that are *not* signal names.

ACKNOWLEDG$MENT	*Contains a CUPL reserved symbol*
A 1	*Contains a blank in the middle*
123	*A number*
PIN	*A CUPL reserved keyword*

Simple hexadecimal numbers contain only digits' like the last example above. CUPL uses hexadecimal as the default for all numbers except device pin numbers and indexed variables which are always decimal. Binary, octal, decimal and hexadecimal numbers are specified by begin with a letter to indicate their base. All four of the numbers below are equivalent. The letter describing the type of number may be either upper or lower case.

'D'13	Decimal
'b'1101	Binary
'o'15	Octal
'H'0D	Hex

Numbers in binary, octal or hexadecimal can have Xs embedded to indicate so-called "don't care" bits. The two numbers below are equivalent.

'h'3XB	*Hexadecimal*
'b'0011XXXX1011	*Binary*

You will see one use for the X's shortly when comparison operators are introduced. Other uses appear later in this handout in truth tables and testing.

Either upper or lower case letters can be used, but there is a convention you should follow to make your work match the work of others. The convention calls for upper case letters in signal names, upper case letters for the hexadecimal digits A through F, and lower case letters for the binary, octal, and hexadecimal suffixes on numbers. The Xs are also usually written in upper case. You are expected to follow this convention regardless of what CUPL allows.

Signal sets - Indexed Variables

Sets of signals with similar names can be written in a shorthand notation. For the basic gates on page 5, we had six signals beginning with Y, namely Y0, Y1, Y2,Y3,Y4, and Y5. When a numeric suffix to a signal name is written in brackets, it is called an *index;* several such suffixes are called a set of *indices.* Grouping variables together is useful for creating address or data lines.

```
/* *************** OUTPUT PINS ********************/
PIN 13  = Y0                  ; /*                          */
PIN 14  = Y1                  ; /*                          */
PIN 15  = Y2                  ; /*                          */
PIN 16  = Y3                  ; /*                          */
PIN 17  = Y4                  ; /*                          */
PIN 18  = Y5                  ; /*                          */
```

Now, using indices, we can write this shorter form.

```
/* *************** OUTPUT PINS ********************/
PIN [13,14,15,16,17,18]  = [Y0,Y1,Y2,Y3,Y4,Y5]                  ; /*              */
```

Writing a list of values in brackets is the same as writing all signals that begin with the name and end with the numeric values. It is not possible to change the order of the signals in CUPL. CUPL will always order the numbers in the set from lease to greatest. CUPL will also allow indexed variables that do not begin with 0. However, these are assumed to be part of a larger number and will probably not be used in this class. You should start all your indexed varables with zeros.

Now, reconsider the example above. Whenever we have a list of indices that are all in order, with no omissions, we can use a list symbol (..) to shorten the list. The notation "0..5" means all numbers from 0 to 5, including both 0 and 5. Therefore, when we write

```
/* *************** OUTPUT PINS ********************/
PIN [13..18]  = [Y0..5]                      ; /*                            */
```

we obtain the same results as we did earlier by writing

```
/* *************** OUTPUT PINS ********************/
PIN [13,14,15,16,17,18]  = [Y0,Y1,Y2,Y3,Y4,Y5]                  ; /*              */
```

The notation [Y0..5] is read "Y zero-through-five" The beginning of the list does not have to be a smaller number than the end. We can write:

```
PIN [13..18]  = [Y5..0]                      ; /*                */
```

when we mean

```
PIN [13..18]  = [Y0..5]                      ; /*                */
```

If the set of indices contains omissions, then two separate lists can be used. In the set of signals beginning with A (on page 5), signal A3 was missing. Therefore, we could not write A[0..5], because we would be referencing signal A3, which was not included in the device. Instead, we write A[0..2,4..5]. Then the set of input signals which we originally wrote as

```
/* *************** INPUT PINS ********************/
```

```
PIN     =  A0                   ; /*                                 */
PIN     =  A1                   ; /*                                 */
PIN     =  A2                   ; /*                                 */
PIN     =  A4                   ; /*                                 */
PIN     =  A5                   ; /*                                 */
PIN     =  [B0..B5]       ; /*                                 */
```

can be rephrased more compactly and more clearly like this.

```
/* *************** INPUT PINS *********************/
PIN     =  [A0..2,A4,A5]        ; /*                                 */
PIN     =  [B0..B5]       ; /*                                 */
```

In this new form we see at a glance which signals are included, more clearly than when each signal name is listed separately. For actual work, when the names are larger and the signals become more numerous, the indexing notation becomes indispensable. Using indexing notation, the example of basic gates looks like this.

```
Device   p12h6 ;

/* *************** INPUT PINS *********************/
PIN [7..9,12,19]   = A0                   ; /*                                 */
PIN 8    = A1                   ; /*                                 */
PIN 9    = A2                   ; /*                                 */
PIN 12   = A4                   ; /*                                 */
PIN 19   = A5                   ; /*                                 */
PIN [1..6] = [B0..B5]           ; /*                                 */

/* *************** OUTPUT PINS *********************/
PIN 13   = Y0                   ; /*                                 */
PIN 14   = Y1                   ; /*                                 */
PIN 15   = Y2                   ; /*                                 */
PIN 16   = Y3                   ; /*                                 */
PIN 17   = Y4                   ; /*                                 */
PIN 18   = Y5                   ; /*                                 */

/* ***** Logic Equations ***** */

Y0 = A0 & B0;
Y1 = A1 # B1;
Y2 = A2 $ B2;
Y3 = !B3;
Y4 = !(A4 & B4);
Y5 = !(A5 # B5);
```

Lists can also be used to specify pin numbers explicitly. If we want to assign signals sequentially by pin number, rather than by geometric location on the chip, we can use ranges on pin numbers. Below is a slightly different assignment.

```
Device   p12h6 ;

/* *************** INPUT PINS *********************/
PIN [7..9,12,19]   = [A0..2,A4,A5] ; /*                                 */
PIN [1..6] = [B0..B5]           ; /*                                 */
/* *************** OUTPUT PINS *********************/
PIN [13..18]  = [Y0..5]              ; /*                   */

/* ***** Logic Equations ***** */
```

```
Y0 = A0 & B0;
Y1 = A1 # B1;
Y2 = A2 $ B2;
Y3 = !B3;
Y4 = !(A4 & B4);
Y5 = !(A5 # B5);
```

To summarize, brackets enclose a list of indices, a range of indices, a list of ranges, or a combination.
So far, indexing has saved space and made lists of signals more readable In later material, you will see how entire equations can be indexed, allowing the *CUPL* compiler to synthesize logic for rather complex devices.

OPERATORS

Boolean operators

In a language like this, operators often represent gates (&, #, $, and so on) Signals are grouped together with operators and parentheses to form *Boolean expressions. A* Boolean expression is the same as a simple collection of multilevel logic. Here are the operators introduced so far.

=	Assignment of an expression to an output signal
!	Logical NOT of a Boolean expression
&	Logical AND of two Boolean expressions
#	Logical OR of two Boolean expressions
$	Logical EXCLUSIVE-OR of two Boolean expressions

All of the symbols shown thus far arise because of the limitations of modern keyboards, printers, and text editors. There is no deep reason that '#' is a good symbol for logical OR, for example. A much better choice for AND and OR are the symbols '^' and 'v'. These two symbols reflect the duality of AND/OR in Boolean algebra and can help prevent errors in algebraic manipulations. But they are not readily available on most keyboards. We will have to stick with what's available.

By the way, some languages use plus (+) for logical OR and asterisk (*) for logical AND, making an analogy with addition and multiplication of ordinary numbers. ORCAD did not do this for two reasons. First, addition and multiplication in ordinary algebra have different distributive properties from logical OR and AND of Boolean algebra, so it is preferable not to inter-mix the notation. Second, the plus sign and asterisk are already used in the language for normal arithmetic operations, as you will see later. Without the arithmetic operators, the *CUPL* language would lose much of its power.

Comparison operators

As you have seen, signals can be grouped using an indexing notation to simplify input and output definitions. Signals can also be grouped to simplify logical operations. One operator is a shorthand notation for combinations of AND and OR.

: Comparison for equal

On the left of the colon (:) we write a signal set and on the right we write a numeric literal. For Instance,

```
SELECT = A:'b'1;
```

The notation is read "SELECT is equal to 1 iff A is equal to 1" or "SELECT is equal to 1 iff A is equivalent to 1" or "SELECT is A is 1". It is true if signal A is active. Likewise,

```
SELECT = A:'b'O;
```

These two examples can be summarized like this.

SELECT = A:'b'1 means the same as SELECT = A
SELECT = A:'b'0 means the same as SELECT = !A

If we were only planning to compare to values 0 and 1, there would be little point to this new notation. However, we can place an entire indexed signal set on the left as in the following two lines

```
SELECT = [A0..5]:'b'000000;
SELECT = [A0..5]:0;          /* remember that constants are in hex.  */
```

This expression is true only if all signals A0,A1,A2,...,A5 are zero. Conceptually, this expression means that the signal set A[5..0] should be treated as a binary number-with A5 at the left and A0 at the right-and compared to see if is zero. Thus, it is the same as writing

```
SELECT = !A5 & !A4 & !A3 & !A2 & !A1 & !A0;
```

Here is how the comparison operator works when a signal set appears on the left and a number on the right. First the number is expanded into binary notation. Then the signals are paired from right to left with the bits of the binary number. In this case, the number 0 expands with every bit zero. The signals are paired as follows.

```
O   O   O   O   O   O
↑   ↑   ↑   ↑   ↑   ↑
A5  A4  A3  A2  A1   AO
```

Now the individual signals are compared with their associated bits and ANDed together, yielding

the following (please note that == is being used to represent equality. This is shorthand for this handout only and is not part of the CUPL language):

A5==0 & A4==0 & A3==0 & A2==0 & A1==0 & A0==0

Since A5==0 means the same as !A5, we arrive at the previous result.

```
SELECT = !A5 & !A4 & !A3 & !A2 & !A1 & !A0;
```

If we use a number other than zero, we get a different result. For example, suppose we write

```
SELECT = A[5..0]:'d'13;
```

The number 13 expands to 1101 in binary. Therefore, the signals are paired like this.

```
0  0  1  1  0  1
↑  ↑  ↑  ↑  ↑  ↑
A5 A4 A3 A2 A1 A0.
```

Notice how the binary number 1101 is padded to the left with zeros. Now when the individual signals are compared with their associated bits and ANDed together, we obtain

A5==0 & A4==0 & A3==1 & A2==1 & A1==O & AO==1

which is the same as

```
SELECT = !A5 & !A4 & A3 & A2 & !A1 & AO;
```

In other words, the expression [A5..0]:'d'13 asks whether the signal set [A5..O] , treated as a binary number, is equal to the decimal value 13. If [A5..0] were a six-bit bus carrying ASCII symbols, the expression [A5..0]:'d'13 would be asking whether the bus contained a carriage return symbol.

Also it is important to remember that [A5..0] is the same as [A0..5] because CUPL reorders the variables.

In addition to simple decimal numbers, we can use binary, hexadecimal, or octal numbers, and these numbers can contain Xs. The comparison

SELECT = [A5..0]:'b'10XX11;

generates this alignment.

```
1  0  X  X  1  1
↑  ↑  ↑  ↑  ↑  ↑
A5 A4 A3 A2 A1 AO
```

Now the two signals A3 and A2 will not enter into the comparison, since it doesn't matter what their values are. Only four signals take part.

```
SELECT = A5 & !A4 & A1 & A0;
```

Using the basic operations of comparing a signal set with a number, and combining these operations with indexed sets of equations (as discussed next) two or more signal sets can be compared in rather complex ways.

Similar to equality, CUPL can also test to see if a number is in range. For example

```
SELECT = [A3..0]:[C..F];
```

This line will check to see if the indexed variables A3,A2,A1,A0 interpretted as a hexadecimal number is in the range of C through F. It is equivalent to saying

```
SELECT = [A3..0]:C #  [A3..0]:D # [A3..0]:E # [A3..0]:F;
```

This equation would expand to

```
SELECT = A3 & A2 & !A1 & !A0
       #  A3 & A2 & !A1 &  A0
       #  A3 & A2 &  A1 & !A0
       #  A3 & A2 &  A1 &  A0;
```

Which could eventuall be minimized to

```
SELECT = A3 & A2;
```

There are more functions available through CUPL that are not discussed here. For instance, the user may specify a FIELD which is to be used in place of a list of variables. Also, it is possible to have a list of variables which does not begin with 0. These can be of use in large projects, but are often more combersome than helpful The user is referred to the CUPL Users guide for further reading if either of these two functions are required.

Precedence among operators

As we have noted above, CUPL supports only four standard logical operators. The precedence for these is as follows:

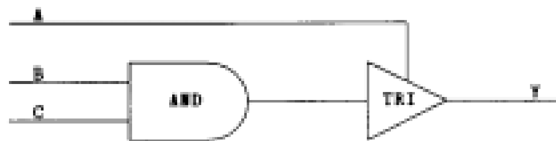
Operator	Example	Description	Precedence
!	!A	NOT	1
&	A & B	AND	2
#	A # B	OR	3
$	A $ B	XOR	4

Hardware operators - Extensions

Additional operators address hardware features. You can use the operators for AND, OR, XOR and NOT rather indiscriminately since all programmable logic supports them. Others, however, require special hardware support. It does no good to specify a tri-state enable for a device not equipped with three-state drivers. CUPL uses extensions on the input and output pins to assign variables and logic to various functions. For instance Y0.OE is the variable used to enable output on a pin which Y0 is mapped to. Some of the common extensions are as follows.

.OE = Three-state (tri state) enable
.CK = clock input to the output flipflop (from a product)
.CKMUX = clock input to the output flipflop (from an input pin)
.AR = asynchronous reset of a flip-flop
.D = D input of a D-type flip-flop

The three-state enable extension requires a second equation which determines enabling condition. The notation has the following schematic.



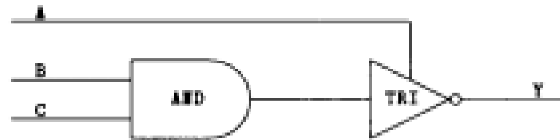
The logical product of B and C will reach the output Y if signal A is active. If A is inactive, output Y will enter a high impedance state. The CUPL code would look like the following

```
Y = B & C;
Y.OE = A;
```

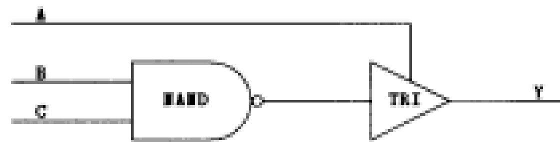
If the rest of the equation is complemented,

```
Y = !(B & C)
Y.OE = A
```

then the corresponding schematic has an inverting driver



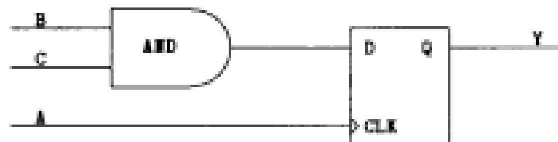
or what is equivalent, an inverting AND gate (NAND).



The clock extension (.CK) is used to attach a product term signal to the clock for the output latch. If an input pin selected from a number of possible input clock pins, then .CKMUX is to be used clock

```
Y =  B & C;
Y.CK = A;      /* If A is an intermediate variable representing a product term
Y.CKMUX = A;   /* If A is an input pin - which pin is dependant on
                  the device chosen */
```

has the following schematic.



On the rising edge of A, the logical product of B and C is latched, provided of course that the setup and hold time for the device have been met.

D-type flip-flops

For a logic chip which has d flip-flops on the output registers, it is easy to map these D flip-flop inputs to product terms.

```
Y.D = HRDY & EN;
Y.CLK = HCLK;
```

In this case, the flip-flop will set (have an output value of 1) whenever the clock HCLK rises and HRDY and EN are active. If HRDY or EN are inactive when HCLK rises, then the flip-flop will reset to zero.

Toggle flip-flops

In addition to D flip-flops, T flip-flops (toggle flip-flops) can be specified for devices which support them. The format of a T flip-flop is similar to that of a D flip-flop. Assume the output varible name is Y.

```
Y.T = HRDY & EN;
Y.CK = HCLK;
```

In this case, the flip-flop will toggle (change from 0 to 1 or from 1 to 0) whenever the clock HCLK rises and HRDY and EN are active. If HRDY or EN are inactive, then the flip-flop will remain unchanged. Some logic requires fewer product terms when defined with T flip-flops.

It is important to understand that toggle flip-flops are simply devices attached to whatever logic is defined, and that when the logic generates an active signal, the flip-flop toggles. Keep in mind that you must not just change D flip-flops to toggle flip-flops without providing for this in the combinational part of the logic.

The List of all the extensions.

What follows is a list of all the extensions supported by CUPL. Not all devices support all extensions. For a more complete exaplanation of the functions of each extension, the user is referred to the CUPL Reference Manual which is available from the WinCUPL help menu.

Extension	Side Used	Description
.AP	L	Asynchronous preset of flip-flop
.AR	L	Asynchronous reset of flip-flop
.APMUX	L	Asynchronous preset multiplexer selection
.ARMUX	L	Asynchronous reset multiplexer selection
.BYP	L	Programmable register bypass
.CA	L	Complement array
.CE	L	CE input of enabled D-CE type flip-flop
.CK	L	Programmable clock of flip-flop
.CKMUX	L	Clock multiplexer selection
.D	L	D nput of D-type flip-flop
.DFB	R	D registered feedback path selection
.DQ	R	Q output of D-type flip-flop
.IMUX	L	Input multiplexer selection of two pins
.INT	R	Internal feedback path for registered macrocell
.IO	R	Pin feedback path selection
.IOAR	L	Asynchronous reset for pin feedback register
.IOAP	L	Asynchronous preset for pin feedback register
.IOCK	L	Clock for pin feedback register
.IOD	R	Pin feedback path through D register
.IOL	R	Pin feedback path through latch
.IOSP	L	Synchronous preset for pin feedback register
.IOSR	L	Synchronous reset for pin feedback register
.J	L	J input of JK-type output flip-flop
.K	L	K input of JK-type output flip-flop
.L	L	D input of transparent latch
.LE	L	Programmable latch enable
.LEMUX	L	Latch enable multiplexer selection
.LFB	R	Latched feedback path selection
.LQ	R	Q output of transparent input latch
.OBS	L	Programmable observability of buried nodes
.OE	L	Programmable output enable
.OEMUX	L	Tri-state multiplexer selection
.PR	L	Programmable preload
.R	L	R input of SR-type output flip-flop
.S	L	S input of SR-type output flip-flop
.SP	L	Synchronous preset of flip-flop
.SR	L	Synchronous reset of flip-flop
.T	L	T input of toggle output flip-flop
.TEC	L	Technology-dependent fuse selection

.TFB	R	T registered feedback path selection
.T1	L	T1 input of 2-T flip-flop
.T2	L	T2 input of 2-T flip-flop

EXAMPLES

The examples so far have been constructed to illustrate principles of the language. To close this section, we should look at some examples that are more like those encountered in real life. The problem with real-life examples, however, is that they are not usually complete in themselves; the rest of the circuit is needed to understand them completely. Nonetheless, it is important to see examples that go beyond usual textbook illustrations.

Example 1

Here we have a small device acting as part of an address controller for a graphics display.

```
Name     example1 ;
PartNo   00 ;
Date     11/8/99 ;
Revision 01 ;
Designer Engineer ;
Company  Cooper Union ;
Assembly None ;
Location  ;
Device   p1414 ;

/* *************** INPUT PINS *********************/
PIN  [11,12,13,18]  =   [BIT0..3]      ; /* Bits 0 - 8 - Active High        */
PIN  [1..9]    =   [RW8..0]            ; /* signal from counter - Active High */
PIN  19    =   !ER                       ;/* end of Rectangle Input - Active  Low */

/* *************** OUTPUT PINS *********************/
PIN  14    =   !EOW                    ; /* End of Word - Active Low        */
PIN  15    =   !EOR                    ; /* End of Rectangle - Active Low   */
PIN  16    =   !EOI                    ; /* End of Input - Active Low       */

/* Logic Equations */
EOI = [RW8..0]:0;                              /* End of Input */
EOW = [RW8..0]:0 # [BIT3..0]:'d'12;   /* End of Word */
EOR = [RW8..0]:0 # ER;                /* End of Rectangle */
```

BIT is a four-bit signal that defines which bit of a twelve-bit graphics word is being processed. RW is a nine-bit signal derived from a counter that defines the number of words remaining. The first equation activates the end-of-input signal EOI when all nine bits of RW are zero-that is, when all input words have been processed. The second equation activates the end-of-word signal EOW either when all words have been processed or when the 12th bit of the word is being processed. The third equation activates the end-of-rectangle signal EOR either when

all input words have been processed or when the ER signal is active. This example contains wide AND gates combined with two-input OR gates. Notice that all signals but BIT and RW are active-low. In CUPL an input or output is defined as being active low by putting a NOT symbol (!) in the PIN declaration line.

Example 2

The next example shows several techniques.[TREQ1..2] are created by simple AND functions-just basic gates. In upcoming sections, you will see how these two lines could have been combined into one. Signal MASK uses comparison operations (:) to define a decoder for signal [OP0..1], then ANDs the decoded signals with DOT to form the result. The signal DOT itself is created on the next line and feeds into the line above. It is formed in part by EXCLUSIVE-OR with an inverse video signal INV. Signal ACK is interesting. Notice that ACK is used as part of its own definition. This self-reference is completely correct, but to explain it now would be to jump too far ahead.

```
Name     example2 ;
PartNo   00 ;
Date     11/8/99 ;
Revision 01 ;
Designer Engineer ;
Company  Cooper Union ;
Assembly None ;
Location  ;
Device   p1618 ;

/* *************** INPUT PINS *********************/
PIN  1    =      BREQ              ; /*                                  */
PIN  2    =      BBEG              ; /*                                  */
PIN  [3..4]   =      [OP0..1]      ; /*                                  */
PIN  [5..6]   =      ![DEN0..1]    ; /*                                  */
PIN  7    =      !INV              ; /*                                  */
PIN  8    =      !DREQ             ; /*                                  */
PIN  9    =      !BIT0             ; /*                                  */
PIN  11   =      !BITB             ; /*                                  */

/* *************** OUTPUT PINS *********************/
PIN   [12,19]  =      ![TREQ1..2]  ; /*                                  */

/* *************** I/O PINS    *********************/
PIN    13 = !REQ                   ; /*                                  */
PIN    14 = !ACK                   ; /*                                  */
PIN    15 = !DOT                   ; /*                                  */
PIN    16 = !MASK                  ; /*                                  */
PIN    [17..18] = ![DACK0..1]      ; /*                                  */


TREQ0 = DREQ & DEN0;
TREQ1 = DREQ & DEN1;

MASK = [OP0..1]:'d'1
       # ([OP0..1]:'d'2 & DOT)
       # ([OP0..1]:'d'3 & DOT);

DOT = ((!DEN0#BITB) & (!DEN1#BIT0)) $ INV;
REQ = BREQ $ BBEG;
ACK = ( ((DREQ & !DEN0) # DACK0) &
       ((DREQ & !DEN1) # DACK1) )
       # ( ACK & (DACK0 # DACK1) );
```

Notice that the expressions here are longer and span several lines in the source code. Any amount of white space can be used, even when it wraps from line to line.

Example 3

The next example contains three clocked signals, ABIT, AWORD, and ALINE. The clock extension indicates that these three signals change only when the clock CL3 rises. To become active, AWORD requires that B[3..0] contain the value 12, which means that B[3..0] must be equal to the binary value 1100b. In contrast, for signal ABIT to be active, B[3..0] must not contain the value 12. The other signals are obtained directly from basic gates. Notice that signals LCD1 and LCD2 are complements of one another.

```
Name     example3 ;
PartNo   00 ;
Date     11/8/99 ;
Revision 01 ;
Designer Engineer ;
Company  Cooper Union ;
Assembly None ;
Location  ;
Device   p16r4 ;


/* *************** INPUT PINS *********************/
PIN [2..5]    = [B3..0]            ; /*                            */
PIN 6    = !AREQ                   ; /*                            */
PIN 7    = LOAD                    ; /*                            */
PIN 8    = !CX                     ; /*                            */
PIN 9    = !CY                     ; /*                            */
// PIN 26   = CL3                      ; /* Clock Pin */

/* *************** Output PINS *********************/
PIN [14..15]  = [LCD0..1]          ; /*                            */
PIN 16    = DWIDTH                 ; /*                            */

/* *************** I/O PINS *********************/
PIN  19   =  ABIT                  ; /*                            */
PIN  13   =  AWORD                 ; /*                            */
PIN  18   =  ALINE                 ; /*                            */

ABIT = (AREQ & !CX) & ([B3..0]:['d'0..'d'11] # [B3..0]:['d'13..'d'15]);
AWORD = AREQ & !CX & [B3..0]:'d'12; /* Advance one word */
ALINE = AREQ & CX & !CY; /* Advance one line */

DWIDTH.D = ABIT # AWORD; /* Decrement the Width */

LCD0.D = (ALINE # LOAD); /* Load new coordinate */
LCD1.D = !(ALINE # LOAD); /* and dot numbers */
```

The last two examples used signals identified as "io" in addition to the "in", and "out" signals you have already seen. The type of signals depend upon the device.