PLD Design Basics

INTRODUCTION

This section is intended as a beginner's introduction to PLD design, although experienced users may find it a good review. We will take a step-by-step approach through two very simple designs to demonstrate the basic PLD design implementation process. Through this effort, you will be introduced to the concept of device programming.

By "beginner," we mean a logic designer who is just beginning to use programmable logic. You may have a lot of experience with discrete digital logic, or you may have just graduated from college. We assume a basic understanding of digital logic. Some computer experience is helpful, but not essential.

We will take no significant shortcuts for these examples, even though there may be times when we could. In this way, you can gain a better understanding of exactly what is happening as you implement your design.

We will talk about device programming, describing all of the steps that are necessary to program a PLD. However, due to the wide variety of programmers available, we will not get down to the level of detail that tells you exactly which buttons to push. Although we will get as close as we can, we must defer the details to your programmer manual.

Constructing a Combinatorial Design—Basic Gates

The first example we will try is a very simple combinatorial circuit consisting of all of the basic logic gates, as shown in Figure 1. This will be helpful for those designs where you are integrating random logic into a PAL device to save space and money.

As can be seen from the figure, there will be six separate functions involving a total of twelve inputs. It is important to bear in mind that programmable logic provides a convenient means of *implementing* designs. With a real design, some work would be required before this point to conceptualize the design, but due to the simplicity of these circuits, we are already in a position to start the implementation.

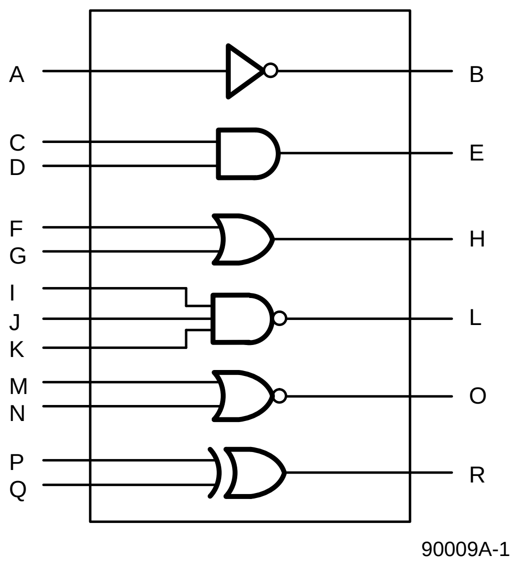


Figure 1. The Basic Logic Gates

Building the Equations

We will start by generating Boolean equations. The first function to be generated is an inverter. This is specified according to Figure 1 as:

```
B = /A
```

Here the "equal" sign (=) is used to assign a function to output B. The slash (/) is used to indicate negation. Thus, this equation may be read:

```
B is TRUE if NOT A is TRUE
```

The next function is a simple AND gate. As shown in Figure 1, we can write:

```
E = C*D
```

Here we use the "equal" sign again, but this time we have introduced the asterisk (*) to indicate the AND operation. This equation may be read:

```
E is TRUE if C AND D are TRUE
```

Publication# **90009** Rev. **A** Amendment **/0**
Issue Date: **February 1996**

The third function is an OR gate, which may be written:

```
H = F + G
```

The “plus” sign (+) is used to specify the OR operation here. Because of the sum-of-products nature of logic as implemented in PLDs, it is often easy to place product terms on separate lines, which improves the readability. We may rewrite this equation as:

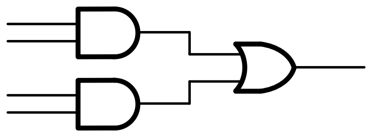
```
H = F
  + G
```

This equation may be read:

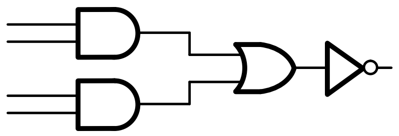
```
H is TRUE if F OR G is TRUE
```

For the moment, we will assume that we have active-HIGH outputs on our device. The functions we have generated so far have essentially been active-HIGH functions. At times we wish to generate active-LOW functions; the next two functions are active-LOW functions that we wish to implement in an active-HIGH device.

When we talk in terms of an active-HIGH or an active-LOW device, the real question is whether there is an extra inverter at the output. An active-HIGH device has an AND-OR structure; an active-LOW device has an AND-OR-INVERT structure which inverts the function at the output (see Figure 2).



a. AND-OR Structure



b. AND-OR-INVERT Structure

90009A-2

Figure 2. Active HIGH vs. Active LOW

NAND and NOR gates could be generated very simply in an active-LOW device, because we would just have to generate AND and OR functions, and let the output inverter generate their complements. However, given that we wish to implement these functions in an active-HIGH device, we must invoke DeMorgan’s theorem, as follows:

```
/(X * Y) = /X + /Y
/(X + Y) = /X * /Y
```

We may generate our NAND function by writing:

```
L = / (I * J * K)
```

or, if preferred,

```
L = /I
  + /J
  + /K
```

Likewise the NOR function may be specified as:

```
O = /(M
  +  N)
```

or

```
O = /M * /N
```

Finally, an exclusive-OR (XOR) gate may be specified either as:

```
R = P :+: Q
```

where :+: represents the XOR operation, or more explicitly as:

```
R = P * /Q
  + /P * Q
```

We have now specified all of the functions in terms of their Boolean equations. The equations are summarized in Figure 3.

```
B = /A      ; inverter

E = C * D   ; AND gate

H = F       ; OR gate
  + G

L = /I      ; NAND gate
  + /J
  + /K

O = /M * /N ; NOR gate

R = P * /Q  ; XOR gate
  + /P * Q
```

Figure 3. Basic Gates Equations

Understanding the Logic Diagram

A portion of a logic diagram is shown in Figure 4.

The logic diagram shows all of the logic resources available in a particular device. In each device, inputs are provided in true and complement versions, as shown in Figure 4. These drive what are often called “input lines,” which are the vertical lines in the logic diagram. These input lines can then be connected to product terms. The name “product term” is really just a fancy name for an AND gate. However, PLDs provide very wide gates, which can be cumbersome to draw. To save space, the product terms are drawn as horizontal lines with a small AND gate symbol at one end to indicate the function being performed.

Although you really do not need to be concerned with the actual implementation of these functions inside the PAL device, you may be curious. Figure 5 shows how the inverter and the AND gate are implemented. An ‘X’ indicates a connection. A product term that is not used is indicated by an ‘X’ in the small AND gate.

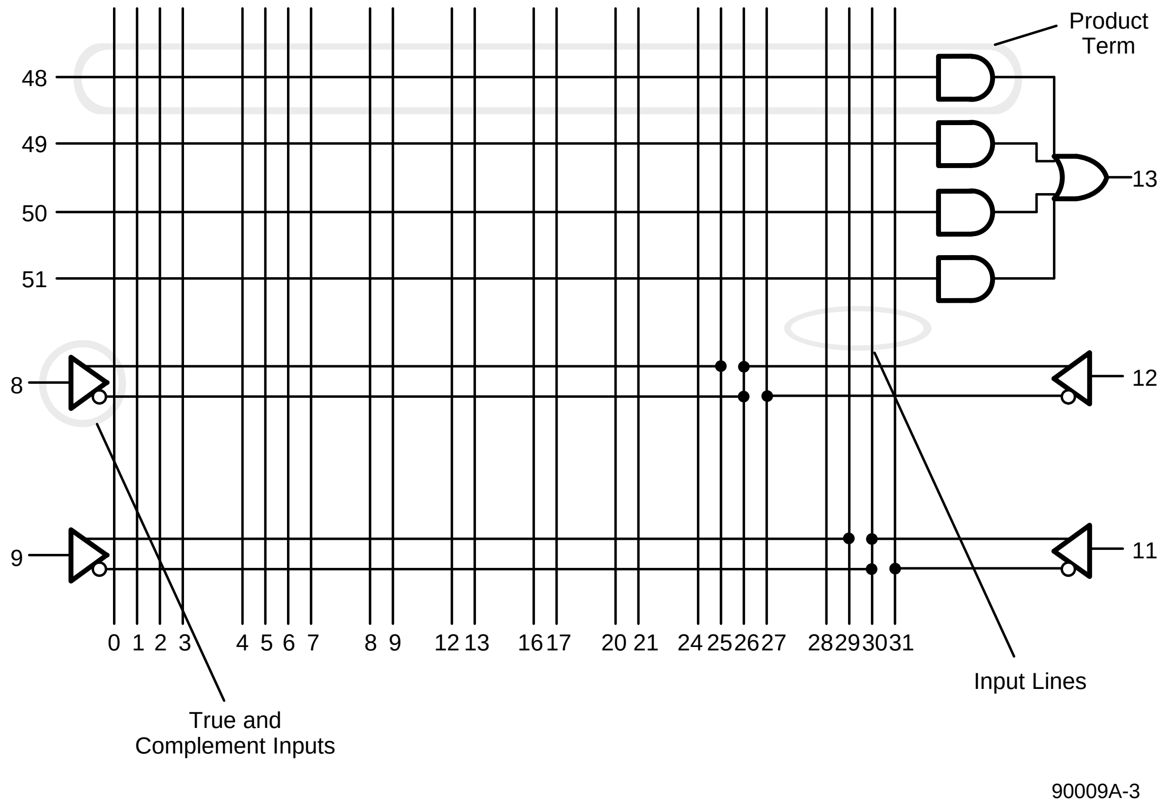


Figure 4. A Portion of a Logic Diagram

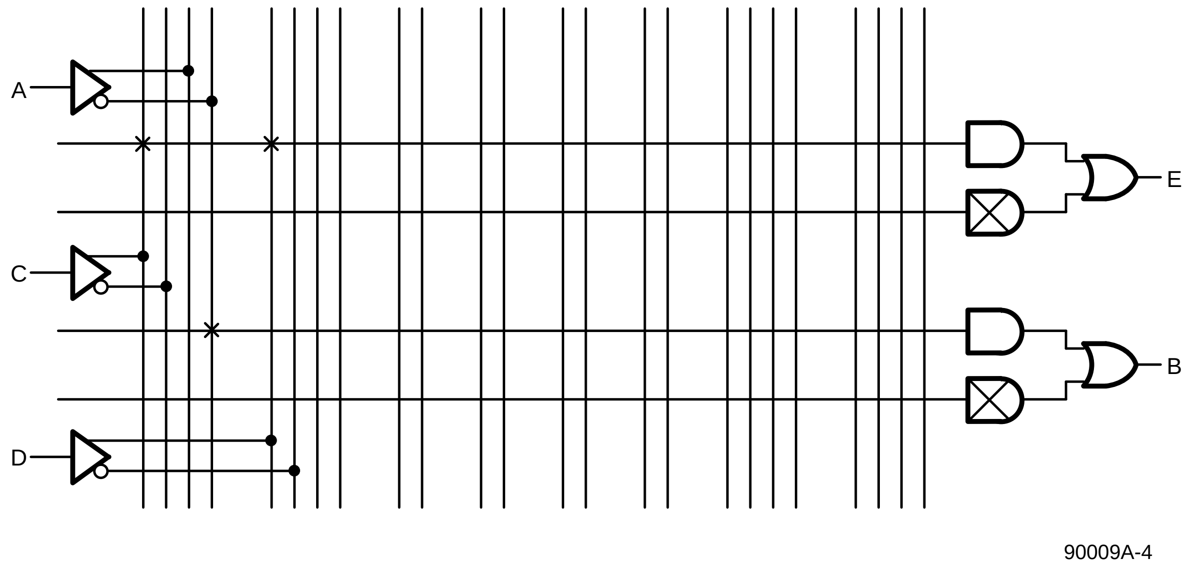


Figure 5. Implementation of NOT, AND Gates

Building the Design File

Once the design has been conceptualized, the design file must be generated.

We now know exactly what our functions are going to be. We have twelve inputs, six outputs, and the NAND function requires three product terms. Note that if we had specified:

```
L = / (I * J * K)
```

instead of:

```
L = /I
  + /J
  + /K
```

for the NAND gate, it would not be as obvious how many product terms would be needed.

We are now in a position to create the design file. The design entry varies with the software package used. You must consult the manuals supplied with the software for design entry format.

Generating a JEDEC File

Once the design file has been entered, you can assemble the design to get a JEDEC file. We have two purposes here: to make sure there are no basic mistakes in the file, and to generate a JEDEC file for programming. Again, how this is done is determined by the software.

Simulating the Gates

After you have verified that your design file is correct, it is time to verify that the design itself is correct. This is done by simulating the design. Simulation provides a way for you to see whether your design is working as you expect it to. You provide a series of commands, or events, which are then simulated by the software. If requested, the software can tell you if the simulation matches what you expect, and, if not, where the problems are.

The simulation section is the last part of the design file. It is not required, but is invariably helpful both in debugging the design, and in generating what can eventually be used as a portion of a test vector sequence.

The simulator also converts the simulation results into test vectors, and appends the vectors to the JEDEC file. This file can be used with programmers that provide functional tests.

Constructing a Registered Design—Basic Flip-Flops

Next we will do a very simple registered design: we will be designing all of the basic flip-flop types (Figure 6). We will conceptualize the design by reviewing briefly the behavior of the D-type flip-flop. We will then present the results for T, J-K, and S-R flip-flops.

The devices we will be using in the examples only have D-type flip-flops. Thus, we will be emulating the other flip-flops with D-type flip-flops.

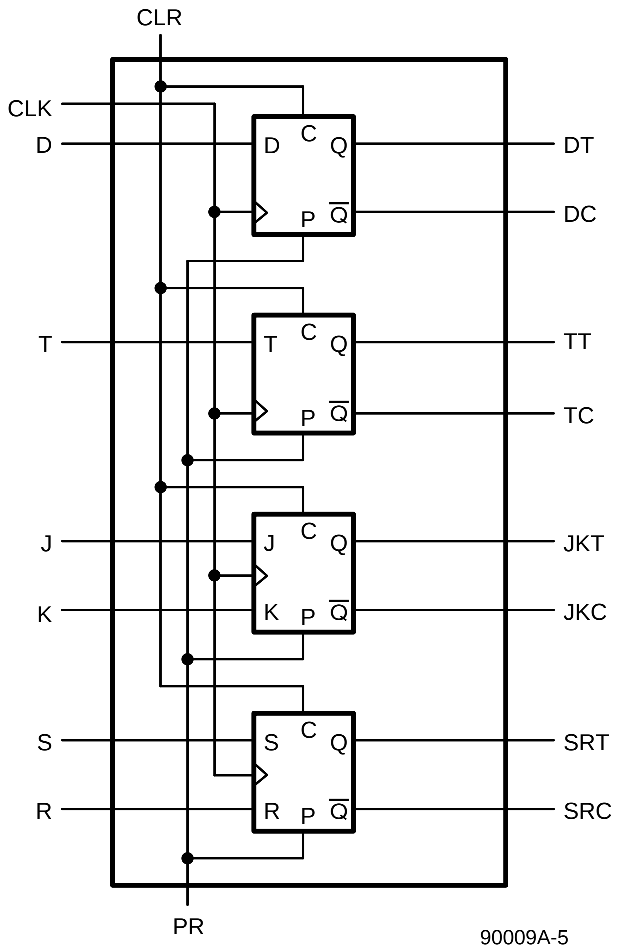


Figure 6. Basic Flip-Flops

Building the D-Type Flip-Flop Equations

A D-type flip-flop merely presents the input data at the output after being clocked. Its basic transfer function can be expressed as:

```
DT : = D
```

where we have used pins DT (D True) and D as shown in Figure 6.

Note the use of ':=' here instead of '='. This indicates that the output is registered for this equation. The difference is illustrated in Figure 7. (PLD design syntax may vary. Consult the appropriate language reference manual.)

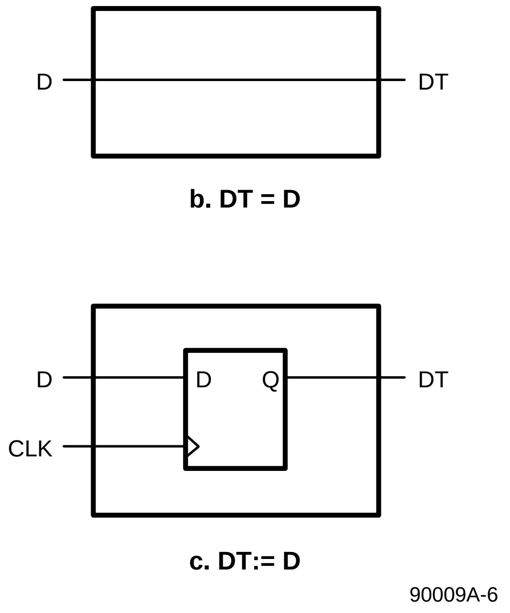


Figure 7. Registered vs. Combinatorial Equations

We can also generate the complement signal (named DC) with the statement:

```
DC : = /D
```

As shown in Figure 6, we want to add synchronous preset and clear functions to the flip-flops. This can be done with two input pins, called PR and CLR. To add these functions to the true flip-flop signal, we add /CLR to every product term and add one product term consisting only of PR. Likewise, for the complement functions, we add /PR to every product term, and add one product term consisting only of CLR. With these changes, the equations now looks like:

```
DT := D * /CLR
    + PR
```

```
DC := /D * /PR
    +  CLR
```

In this way, when clearing the flip-flops, the active-HIGH flip-flops have no product terms true, and go LOW; the active-LOW flip-flops have the last product term true, and will therefore go HIGH. The reverse will occur for the preset function.

There is still one hole in this design: what happens if we preset and clear at the same time? As it is right now, both outputs will go HIGH. This makes no sense since one signal is supposed to be the inverse of the other. To rectify this, we can give the clear function priority over the preset function. We can do this by placing /CLR on every product term for the true flip-flop signal. The results are shown as follows:

```
DT := D * /CLR
    + PR * /CLR

DC := /D * /PR
    + CLR
```

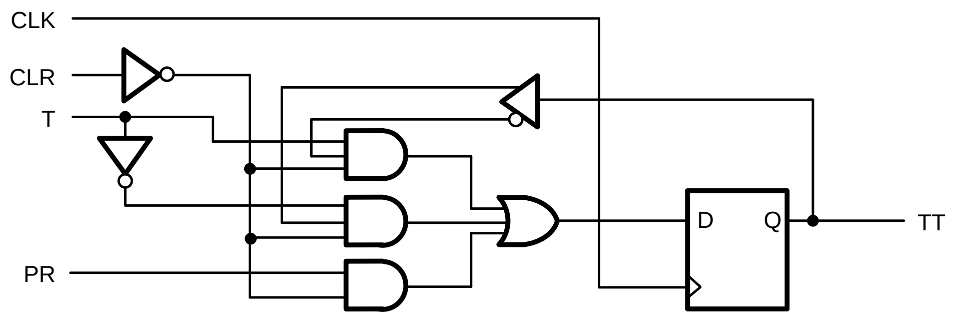
The same basic procedure can be applied to all of the other flip-flops. The equations are shown in Figure 8.

```
EQUATIONS

;emulating all flip-flops with D-type flip-flops

DT := D * /CLR              ;output is D if not clear
   + PR * /CLR              ;or 1 if preset and not clear at the same time

DC := /D * /PR              ;output is /D if not preset
   + CLR                    ;or 1 if clear

TT := T * /TT * /CLR        ;go HI if toggle and not clear
   + /T * TT * /CLR         ;stay HI if not toggle and not clear
   + PR * /CLR              ;go HI if preset and not clear at the same time

TC := T * /TC * /PR         ;go HI if toggle and not preset
   + /T * TC * /PR          ;stay HI if not toggle and not preset
   + CLR                    go HI if clearing

JKT:= J * /JKT * /CLR       ;go HI if J and not clear
   + /K * JKT * /CLR        ;stay HI if not K and not clear
   + PR * /CLR              ;go HI if preset and not clear at the same time

JKC:= /J * /JXC * /PR       ;go HI if not J and not preset
   + K * /JKC * /PR         ;stay HI if not K and not preset
   + CLR                    ;go HI if clear

SRT:= S * /CLR              ;go HI if set and not clear
   + /R * SRT * /CLR        ;stay HI if not reset and not clear
   + PR * /CLR              ;go HI if preset and not clear at the same time

SRC:= R * /PR               ;go HI if reset and not preset
   + /S * SRC * /PR         ;stay HI if not set and not preset
   + CLR                    ;go HI if clear
```

Figure 8. Flip-Flop Equation Section

Building the Remaining Equations and Completing the Design File

Notice that in some of the equations above, the output signal itself shows up in the equations. This is the way in which feedback from the flip-flop can be used to determine the next state of the flip-flop. An equivalent logic drawing of the TT equation is shown in Figure 9.



$$\boxed{TT} := T^* \boxed{/TT} * /CLR$$
$$+/T^* \boxed{TT} * /CLR$$
$$+ PR^*/CLR$$

90009A-7

Figure 9. Feedback in the Equation for TT

We are now in a position to complete the design file. You must follow the instructions included with your software package to complete the file.

Simulating the Flip-Flops

After processing the design and correcting any mistakes, we can run the simulation.

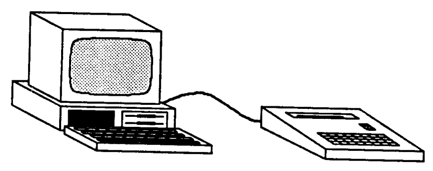
The file can now be simulated in the same manner as the basic gates design.

Programming a Device

After simulating the design, and verifying that it works, it is time to program a device. There are several steps to programming, but the exact operation of the programmer naturally depends on the type of programmer being used. We will be as explicit as we can here, but you will need to refer to your programmer manual for the specifics.

The first thing that must be done after turning the programmer on is to select the device type. This tells the programmer what kind of programming data to expect. The device type is usually selected either from a menu or by entering a device code. Your programmer manual will have the details.

Next a JEDEC file must be downloaded. To transfer the JEDEC file from the computer to your programmer, you will need to provide a connection, as shown in Figure 10.



90009A-8

Figure 10. A Connector Must Be Provided Between the Computer and the Programmer

If your programmer can perform functional tests, and you wish for those tests to be performed, you should download the JEDEC file containing the vectors; otherwise, you should download the JEDEC file without vectors.

To download data, the programmer must first be set up to receive data. The programmer manual will tell you how to do this.

Communication must be set up between the computer and the programmer. Whichever communication program is installed must be invoked. This is used to transmit the JEDEC file to the programmer. Follow the instructions for your program to accomplish the next steps.

Before actually sending the data, you must verify the correct communication protocol. Check to make sure you know what protocol the programmer is expecting; then set up the baud rate, data bits, stop bits, and parity, to match the protocol.

Once the protocol has been set up the JEDEC file must be downloaded.

Enter the name of the JEDEC file you wish to use. The computer will then announce that it is sending the data, and tell you when it is finished. Note that just because it says it has finished sending data does not mean that the data was received. Your programmer will indicate whether or not data was received correctly.

Once the data has been received, the programmer is ready to program a device. Place a device in the appropriate socket, and follow the instructions for your programmer to program the device. This procedure programs and verifies the connections in the device, and, if a JEDEC file containing vectors was used, will perform a functional test.

The programmer will announce when the programming procedure has been completed. You may then take the device and plug it into your application.

If you have actually programmed one of the examples that we created above, you naturally don't have a board into which you can plug the device. If you do have a lab setup, you may wish to play with the devices to verify for yourself that the devices perform just as you expected them to.

You will find much more detail on many issues that were not discussed in this section in the remaining sections of this handbook. This section should have provided you with the basic knowledge you need to understand the remaining design examples in this book, and to start your own designs.