CHAPTER 3:
Combinational Logic Design with PLDs

LSI chips that can be programmed to perform a specific function have largely supplanted discrete SSI and MSI chips in board-level designs. A programmable logic device (PLD), is an LSI chip that contains a “regular” circuit structure, but that allows the designer to customize it for a specific application. PLDs sold in the market is not customized with specific functions. Instead, it is programmed by the purchaser to perform a function required by a particular application.

PLD-based board-level designs often cost less than SSI/MSI designs for a number of reasons. Since PLDs provide more functionality per chip, the total chip and printed-circuit-board (PCB) area are less. Manufacturing costs are reduced in other ways too. A PLD-based board manufacturer needs to keep samples of few, “generic” PLD types, instead of many different MSI part types. This reduces overall inventory requirements and simplifies handling.

PLD-type structures also appear as logic elements embedded in LSI chips, where chip count and board areas are not an issue. Despite the fact that a PLD may “waste” a certain number of gates, a PLD structure can actually reduce circuit cost because its “regular” physical structure may use less chip area than a “random logic” circuit. More importantly, the logic function performed by the PLD structure can often be “tweaked” in successive chip revisions by changing just one or a few metal mask layers that define signal connections in the array, instead of requiring a wholesale addition of gates and gate inputs and subsequent re-layout of a “random logic” design.

PLDs are “structured”, logic devices, not only in their internal circuit designs but also in the software tools that are used to specify their functions. A software tool called PLD compiler can convert the functional description into a set of interconnections that can be programmed into the PLD to make it perform the desired function.

Many modern PLDs are “erasable”. They can be programmed to perform different functions. This capability is especially useful during circuit debugging; it makes digital design and debugging a lot more like software programming and debugging, where different design ideas can be tried with meager investment.

The first PLDs discussed in this chapter are combinational PLD. These PLDs contain certain number of gates and programmable interconnections, but no sequential elements like flip-flops or latches. Each logic block can be programmed to perform any combinational logic function that “fits” within the logic block’s inputs and outputs and its type of gates and interconnections.

Read-only memory (ROM) is sometimes considered to be a programmable logic device, since a ROM can be programmed to perform any combinational logic function with a corresponding number of inputs and outputs.
3.1 Programmable Logic Arrays

The first PLDs were programmable logic arrays (PLAs). A PLA is a combinational, two-level AND-OR device that can be programmed to realize any sum-of-products logic expression, subject to the size limitations of the device.

Limitations are the number of inputs (n), the number of outputs (m) and the number of product terms (p).

Such a device is described as “an n x m PLA with p product terms.” In general, p is far less than the number of n-variable min-terms (2n). Thus a PLA cannot perform arbitrary n-input, m-output logic functions; its usefulness is limited to functions that can be expressed in sum-of-products form using p or fewer product terms.

An n x m PLA with p product terms can contains p 2n-input AND gates and m p-input OR gates. Figure 3.1 below shows a small PLA with four inputs, six AND gates and three OR gates and outputs. Each input is connected to a buffer that produces both a true and complemented version of the signal for use within the array. Potential connections within the array are indicated by X’s; the device is programmed by establishing only the connections that are actually needed. Thus, each AND gate can be connected to any subset of the primary input signals and their complements. Similarly each OR gate can be connected to any subset of the AND-gate outputs.
As shown in Figure 3.2, a more compact diagram can be used to represent a PLA. Moreover, the layout of this diagram more closely resembles the actual internal layout of a PLA chip. The PLA in Figure 3.2 can perform any three 4-input combinational logic function that can be written as sums of products using a total of six or fewer distinct product terms, for example:

\[
\begin{align*}
O_1 &= I_1 . I_2 + I_1'. I_2'. I_3' . I_4' \\
O_2 &= I_1 . I_3' + I_1' . I_3 . I_4 + I_2 \\
O_3 &= I_1 . I_2 + I_1 . I_3' + I_1' . I_2' . I_4' 
\end{align*}
\]

These equations have a total of eight product terms, but the first two terms in the \(O_3\) equation are the same as the first terms in the \(O_1\) and \(O_2\) equations. The programmed connection pattern in Figure 3.3 matches these logic equations.
Figure 3.2: Compact representation of a 4x3 PLA with six product terms.
Sometimes a PLA output must be programmed to be a constant 1 or a constant 0. That can be done as shown in Figure 3.4. Product term P1 is always 1 because its product lines is connected to no inputs and is therefore always pulled HIGH; this constant-1 term drives the O1 output.

No product term drives the O2 output, which is therefore always zero. Another method of obtaining a constant-0 output is shown for O3. Product term, P2 is connected to each input variable and its complement; therefore it’s always 0 ($X \cdot X' = 0$). An n-input, PLA could conceivably use as many as $2^n$ product terms to realize all possible n-variable min-terms. The actual number of product terms in typical commercial PLAs is far fewer, on the order of 4 to 16 per output, regardless of the value of n.

The Signetics 82S100 is a typical example of the PLAs that were introduced in mid-1970s. It has 16 inputs, 48 AND gates, and 8 outputs. Thus, it has $2 \times 16 \times 48 = 1536$ fusible links in the AND array and $8 \times 48 = 384$ in the OR array.
3.2 Programmable Array Logic Devices

A special case of a PLA, and today’s most commonly used type of PLD, is the programmable array logic (PAL) device. Unlike a PLA, in which both the AND and OR arrays are programmable; a PAL device has a fixed OR array.

Key innovations in the first PAL devices were the use of fixed OR array and bidirectional input/output pins. These ideas are well illustrated by the PAL16L8 shown in figure 3.5, and probably today’s most commonly used combinational PLD structure. Its programmable AND array has 64 rows and 32 columns, identified for programming purposes by the small numbers in the logic diagram, and $64 \times 32 = 2048$ fusible links. Each of the 64 AND gates in the array has 32 inputs, accommodating 16 variables and their complements (hence, the “16” in “PAL16L8”)

Eight AND gates are associated with each output pin of the PAL16L8. Seven of them provide inputs to a fixed 7-input OR gate. The eighth output-enable gate is connected to the three stable enable input of the output buffer; the buffer is enabled only when the output-enable gate has 1 output.
Thus, an output of the PAL16L8 can perform only logic functions that can be written as sums of seven or fewer product terms. Each product term can be a function of any or all 16 inputs, but only seven such product terms are available.

Although the PAL16L8 has up to 16 inputs and up to 8 outputs, it is housed only in a package with only 20 pins, including two for power and ground. This is made possible by the use of six bi-directional pins (13-18) that may be used as inputs or outputs or both. This and other differences between the PAL16L8 and a PLA are summarized next:

- The PAL16L8 has a fixed OR array, with seven AND gates permanently connected to each OR gate. AND-gate outputs cannot be shared; if a product term is needed by two OR gates, it must be generated twice.

- Each output of the PAL16L8 has an individual three-state output enable signal, controlled by a dedicated AND gate (the output-enable gate). Thus outputs may be programmed as always enabled, always disabled, or enabled by a product term involving the device inputs.

- There is an inverter between the outputs of each OR gate and the external pin of the device.

- Six of the output pins called the I/O pins may also be used as inputs. This provides many possibilities for using each I/O pin, depending how the device is programmed.
If an I/O pin’s output-control gate produces a constant 0, then the output is always disabled and the pin is used strictly as input. If the input signal on an I/O pin is not used by any gate in the AND array, then the pins may be used strictly as an output. Depending on the programming of the output-enable gate, the output may always be enabled, or it may be enabled only for certain input conditions.

If an I/O-control gate produces a constant 1, then the output is always enabled, but the pin may be used as input too. In this way, outputs can be used to generate first-pass “helper terms” for logic functions that cannot be performed in a single pass with the limited number of AND terms available for a single output.

In another case, with an I/O pin always output-enabled, the output may be used as an input to AND gates that affect the very same output. That is a feedback sequential circuit can be embedded in a PAL16L8.

3.3 PLD programming language

A PLD is ultimately programmed by specifying a diode or a fuse pattern. Usually, designers don’t specify a fuse pattern directly by using a hexadecimal text file. Instead, most designers use a PLD programming language to specify logic functions symbolically. One of them is ABEL programming language.

A PLD programming language is supported by a language processor called compiler. The compiler’s job is to translate a text file written in the language into a fuse pattern for a physical PLD. Even though most PLDs can be physically programmed with sum-of-products expressions, languages like ABEL allow PLD equations to be written in any format. The compiler algebraically manipulates and minimizes the equations to fit, if possible into the available PLD structure. Some compilers for sequential PLDs, allow state machines to be defined in a high-level language (Verilog or VHDL) and automatically select an appropriate PLD, perform state assignments, and develop logic equations.

3.3.1 ABEL Program structure

Table 3.1 below shows the typical structure of a PLD program in the ABEL language, and Table 3.2 shows an actual program. This example exhibits the following language features:

Module: module name
Title: string
DeviceID device: deviceType
Pin: declarations
Other: declarations
Equations equations
end module name
Table 3.1 Typical structure of ABEL program.

- An **Identifier** must begin with a letter or underscore. It may contain up to 31 letters, digits, and are case sensitive.

- A program file begins with a module statement, which associates an identifier (e.g., Memory Decoder) with the program module. Large programs can have multiple modules, each with its own local title, declarations and equations.

- The title statement specifies a title string that will be inserted into the documentation files that are created by the compiler.

- A **string** is a series of characters enclosed by single quotes.

- The device declaration includes a device identifier (e.g., MEMDEC) and a string that denotes the device type (e.g., ‘P16L8’ for a PAL16L8). The compiler uses the device identifier in the names of the documentation files that it generates, and it uses the device type to determine whether the device can really perform the logic functions requested in the program.

- The @ALTERNATE directive tells the compiler to recognize an alternative set of symbols to denote logic operations:
  
  - AND, + OR, / NOT (used as a prefix), :+: XOR, :*: XNOR.

  The default symbols in ABEL for these operations are &, #, !, $, and !$.

- **Comments** begin with a double quote and end with another double quotes or the end of the line, whichever comes first.

- A **pin declaration** tells the compiler about symbolic names associated with the devices external pins. Signals whose names have the NOT prefix (/) the active low at the external pin; others are active high.

- **Other declarations** allow the designer to determine to define constants and expressions to simplify program design and improve program readability.

- The equation statement indicates that logic equation defining output signals, as functions of input signals will follow.

- **Equations** are written like assignment statements in a programming language. A semicolon terminates each equation.

- The end statement marks the end of the module.
Equations for combinational PLD outputs use the *unclocked assignment operator*, =. The left-hand side of an equation normally contains a signal name. The right-hand side is a logic expression, not necessarily in sum-of-products form. The signal name on the left-hand side of an equation may be optionally preceded by the NOT operator /; this is equivalent to complementing the right-hand side. The compiler’s job is to generate a fuse pattern such that the signal named on the left-hand side realizes the logic expression on the right-hand side.

### 3.3.2 ABEL Documentation

The example ABEL program in Table 3.3 defines the logic function to be performed by a PAL16L8 device, but the logic diagram for a system using devices does not list the ABEL program. Instead a logic diagram contains a generic symbol for the PAL16L8 and uses signal names that correspond to the names in the program as shown in Fig 3.6.

```ABEL
module Memory_Decoder
title ‘Memory Decoder PLDMEMDEC device ‘P16L8’;
@ALTERNATE

“Input pins
LARGE
A16, A17, A18, A19
A20, A21, A22, A23
/RTEST, IOSEL

“Output pins
/ROMCS
LOCAL

equations

ROMCS = LARGE * A23 * A22 * A21 * A20 * A19 * A18 * A17 * A16 + /LARGE * A19 * A18 * A17 * A16 + RTEST;
LOCAL = LARGE * A23 * A22 * A21 * A20 * A19 * A18 * A17 * A16 + /LARGE * A19 * A18 * A17 * A16 + RTEST + IOSEL;
end Memory_Decoder
```

Table 3.2: An ABEL program for a memory-decoder PLD

### 3.3.3 Fuse Patterns

The ABEL compiler generates the complete fuse pattern for a PLD from its program. This fuse pattern is stored in a special file whose name usually has an extension of .jed.
This file is then downloaded into a PROM programmer to customize an un-programmed PLD for the desired functions. The compiler also produces a documentation file .doc that shows the final equations resulting from Boolean manipulations on the original ABEL equations.

Figure 3.6: Logic diagram for decoder using a PAL16L8

3.4 Combinational PLD applications

Combinational PLDs can easily duplicate the functions of most standard combinational MSI devices, such as decoders, encoders, multiplexers and the like. On one hand, if a logic design project involves a function that is performed by a standard MSI device, then this will be the smallest, fastest and cheapest solution. On the other hand, if a special version of one of these functions is needed or if the design presented by an available MSI device needs to be tweaked at a later time then PLD is the best choice.

3.4.1 Generic Array Logic Devices

Sequential PLDs are those that can provide flip-flops at some or all OR-gate outputs. These devices can be programmed to perform a variety of useful sequential-circuit functions.

One type of sequential PLD, originally developed by Lattice Semiconductor is called a generic array logic or a GAL device. A single GAL device type the GAL16V8 can be configured to emulate the AND-OR, flip-flop, and output structure of any variety
of combinational and sequential PAL devices, including the PAL16L8. In addition the GAL device can be erased electrically and reprogrammed to perform functions that are superset of those supported by standard PAL devices.

3.4.2 Complex programmable logic devices (CPLD)

These are multiple PAL type devices on a single chip. These can be programmed to perform complex logic designs consisting of decoders, multiplexers and flip-flops. CPLD has predictable timing and capacity. They can be programmed using readily available design software.