GET GOING WITH ...

PLDS

Revision 1.2

K.M. Parnell

ISBN 1 902179 013

PREFACE

Welcome to the second book in the 'Get Going With...' series of books and the exciting world of Programmable Logic. Thank you for buying this book or receiving this book in your Kanda Starter Kit. I hope you find both the book and the Starter Kit useful and enable you to utilise all of the many benefits of programmable logic devices.

This book has been written for the professional engineer who has never used programmable logic devices before and for the engineer who is new to logic design. It is also intended for students or hobbyists who want an easy and understandable way to realise their designs in a straight forward and flexible way. This useful book also is ideal for the designer who knows about the benefits of SPLDs but needs a gentle pointer in the right direction.

The book goes from basic logic design, the history of programmable logic, through basic PLD exercises to more in-depth useful design illustrations. Examples of both combinational and sequential designs are discussed. Useful application examples in Section 8 have been chosen to give an indication of the flexibility and versatility of PLDs and may be used or modified to suit you application.

Included in the back of the book is a copy of CUPL - a design description language and compiler limited to Atmel Programmable Logic Devices. Also included on the disk is an enhancement to the CUPL software courtesy of Atmel Corp. for design template generation.

I am most grateful to my friends at Kanda for giving me the opportunity to write and publish a book, as the adage goes; there is a book inside everyone!

I wish also to thank Atmel for allowing us access to their information and letting us re-produce device data sheets and text from their Configurable Logic Data Book.

This book is dedicated to my children Adam and James who did not have my undivided attention when writing this book. Thank you also to Sean - you know who you are.

K.M. Parnell

NAVIGATING THE BOOK

This book was written for both the professional engineers who has never designed using programmable logic devices and for the new engineer embarking on their exiting career in electronics design. To accommodate this the following navigation section has been written to help the reader decide in advance which sections he/she wishes to read.

This chapter gives an overview of how and where PLDs might be used. It also gives a brief history of the programmable logic device.

Chapter 2 cover's logic and logic design principles, many of these principals you may not have used for some time. Please feel free to skip this section or use it only as a reference guide when you ‘get going’ with your PLD design. This section is intended to give you a quick review and reference of the basic principles of digital logic.

This chapter outlines some basic information about the benefits of using PLDs essential to those who are unfamiliar with Progrmmable Logic Devices (PLDs).The information may also be useful to those who are current users of programmable logic.

Chapter 4 is a step by step approach to your first two simple designs and is intended to demonstrate the basic PLD design implementation process. It covers the use of Atmel CUPL and the Kanda System PLD Starter kit to implement your design. This section includes a simple sequential and combinational design.

This chapter covers a combinational design 7-Segment LED Display decoder. It takes you from design conception, through the defining of logic equations to producing a JEDEC file to programme you device. Full instructions on using CUPL and a device programmer are given.

NAVIGATING THE BOOK (Continued)

In this chapter describes a sequential design - Decade Up/Down Counter from definition to device programming. All the steps necessary to specify , describe and test a sequential PLD design are covered.

In this section we discuss the features of Atmel's Low Power PLD products and have see that these devices can offer the system designer many benefits for applications where power consumption is a critical requirement.

This section gives two application examples which will hopefully give you a good indication of how PLDs can be used in the real world. They have been chosen as examples that may help you with your first design whether it be a sequential or combinational design.

This section contains three Atmel PLD Data Sheets that you will find invaluable when you start your first PLD design. The data sheets are the ATF16V8C, ATF22V10B and ATF20V8B flash based SPLDs.

The appendices are a selection of useful user guides and manuals.

CONTENTS

CONTENTS (Continued)

CONTENTS (Continued)

ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
CMOS	Complementary Metal Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CUPL	Universal Compiler for Programmable Logic
FPGA	Field Programmable Gate Array
GAL	Generic Array Logic
LSB	Least Significant Bit
MSB	Most Significant Bit
NRE	Non Recurring Engineering cost
PAL	Programmable Array Logic device
PCB	Printed Circuit Board
PLA	Programmable Array Logic
PLD	Programmable Logic Device
PROM/EPROM	Programmable Read Only Memory / Erasable
RAM	Random Access Memory
ROM	Read Only Memory
SPLD	Simple Programmable Logic Device
SRAM	Static Random Access Memory
TTL	Transistor Transistor Logic
UV	Ultra Violet
ZIF	Zero Insertion Force

1

INTRODUCTION

1.1 GENERAL OVERVIEW

Many designers have often thought about using PLDs but may have been put off by the thought of software design, expensive programmers and time wasted by using UV erasers to erase devices - well things have moved on and it has never been easier to design using PLDs. This book is intended to show you how easy it is to start a design with PLDs in mind. It will also show you how to save time and money on your designs both in the design phase and when upgrading your products.

The Programmable Array Logic device, commonly known as the PAL device was invented at Monolithic Memories in 1978. The concept for this revolutionary type of device sprang forth as a simple solution to the short comings of discrete TTL logic.

The successfully proven PROM technology that allowed the end user to ‘write on silicon’ provided the technological basis that made this kind of device not only possible, but very popular as well.

Programmable Logic Device (PLD) technology has moved on from the early days with such companies as Atmel producing very low power CMOS devices based on Flash technology. Flash PLDs provide the ability to program the device's time and time again electrically programming and ERASING the device! Gone are the days of erasing taking in excess of twenty minutes under a UV eraser.

The availability of design software such as CUPL has made it much easier to design with programmable logic. Designs can be described easily and quickly using either a description language such as CUPL or a schematic capture package such as View Logic - Work View Office.

As user-programmable semicustom circuits, PLDs provide a valuable compromise that combines many of the benefits of discrete logic with many of the benefits of other semicustom circuits.

1.2 WHY USE PLDS?

Maybe you have heard of all of the wonderful reasons for using PLDs, well they are all true!

The main reasons are:

- **Increased design integration**. You can reduce the amount of devices on your designs while simultaneously increasing the features offered by your product.
- **Lower Power**. CMOS and fewer devices needed in a design combine to reduce power consumption.
- **Improved Reliability**. Lower power plus fewer interconnections and devices translate into greatly improved system reliability.
- **Lower Cost**. PLDs reduce inventory costs.
- **Easier to use**. Yes, believe it or not, once you get past the initial learning period, PLDs are easier to use than discrete logic functions.
- **Easier to change**. OOP's! Need to make a change? You won't need to re-lay out your board, with a PLD all connections are internal and can be changed quickly.

2

LOGIC REFRESHER

2.1 INTRODUCTION

Throughout this book we will be using logic and logic design principles, many of these principles you may not have used for some time. Please feel free to skip this section or use it only as a reference guide when you 'get going' with your PLD design.

This section is intended to give you a quick review and reference of the basic principles of digital logic. We will cover the following areas:

- Basic Logic Elements
- Basic Storage Elements
- Binary Numbers

Digital logic can be divided into *combinational (sometimes known as combinatorial)* and *sequential* . Combinational circuits are those in which the output state depends on the present input states in some predetermined fashion, whereas in sequential circuits the output state depends both on the input states and on the previous history. Combinational circuits can be constructed with gates alone, whereas sequential circuits require some form of memory (flip-flops).

2.1 BASIC LOGIC ELEMENTS

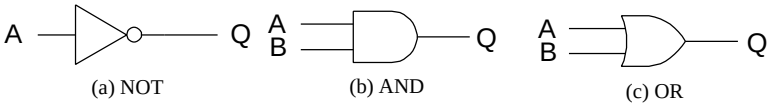
The Three Basic Gates

There are three basic gates from which all other combinational logic functions can be generated: *NOT, AND* and *OR.* The truth table overleaf shows these functions. Since they can be used to generate ANY function they are said to be functionally complete.

A	B	/A	A*B	A+B

0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

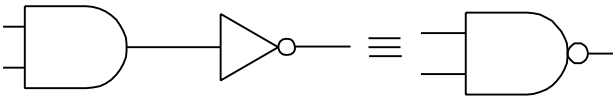
Where '/' is NOT, '*' is AND and '+' is OR.
The logic functions NOT, AND & OR are shown below in schematic form:



(a) NOT (b) AND (c) OR

Figure 1 Logic Functions

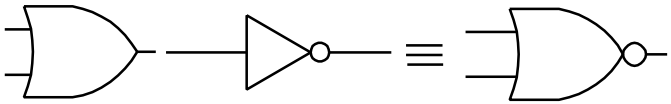
The AND & NOT functions can be combined into the NAND function. This function is equivalent to an AND gate followed by and inverter as shown below in figure 2.



THE NAND FUNCTION

Figure 2

Likewise the OR and NOT gates can be combined into the NOR function as shown in figure 3. Again the NAND and NOR functions are functionally complete; any logic function can be expressed solely as a function of NAND and NOR gates.



THE NOR FUNCTION

Figure 3

Figure 1 shows symbols for the three most important kinds of gates. A 2-input AND gate, shown in (b), produces a 1 output if both of its inputs are 1; otherwise it produces a 0 output. The figure shows the same gate four times with the for possible combinations that may be applied to it and the resulting outputs. A gate is called a combinational circuit because its output depends only on the current input combination.

A two input OR gate shown in (c) produces a 1 output if one or both of its inputs are 1; it produces a 0 output only if both inputs are 0. Once again, there are four possible input combinations, which result in the outputs shown in the figure.

A NOT gate, more commonly called an inverter produces an output value that is the opposite of the input value as shown in (a).

2.3 LOGIC IDENTITIES

ABC = (AB)C = A(BC)
AB = BA
AA = A
A1 = A
A0 = A
A(B+C) = AB + AC
A + AB = A
A + BC = (A+B)(A+C)
A + B + C = (A + B) + C = A + (B + C)
A + B = B + A
A + A = A
A + 1 = 1
A + 0 = A
1' = 0
0' = 1
A + A' = 1
AA' = 0
(A')' = A
A + A'B = A + B
(A + B)' = A'B'
(AB)' = A' + B'
Logic identities can be used to simplify a logic realisation into and AND-OR logic equation ready for easy design implementation.
e.g. $A \oplus B = /AB + A/B$ using logic identities transforms to $(A+B)(\overline{AB})$.

2.4 MINIMISATION AND KARNAUGH MAPS

Since realisation of a logic function isn't unique it is often desirable to find the simplest or perhaps most conveniently constructed circuit for a given function. This can be done using such techniques as Karnaugh maps but this is time consuming and for designs with more than four inputs, extremely tricky. We recommend using the software minimisation package which is included in the Kanda PLD Starter Kit as part of the many functions of Enhanced CUPL.

2.5 BASIC STORAGE ELEMENTS

Storage elements provide circuits with the capability of 'remembering' past conditions or events. You will probably recognise the typical storage element in figure 4. This is just a pair of cross-coupled NAND gates - normally called a 'flip-flop'

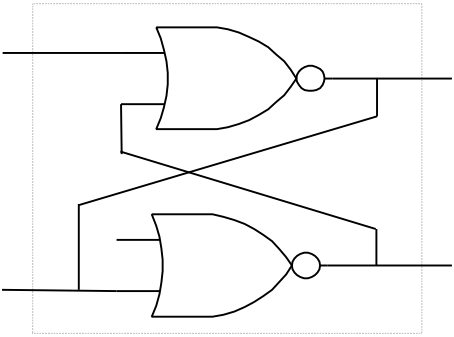


Figure 4 Basic storage element - Flip-Flop

In general there are two classes of flip-flops:

- *Unclocked* flip-flops or latches
- *Clocked* flip-flops

Clocked flip-flops are sometimes referred to as 'registers' although technically speaking, a register is a bank of several flip-flops with a common clock signal.

Flip-flops can also be characterised by their control scheme. There are four types of flip-flops, each of which can be unclocked or clocked:

- S-R Type
- J-K Type
- D- Type
- T- Type

2.5.1 Unclocked Flip-Flops - Latches

2.5.1.1 S-R Latches

An S-R latch can be built out of NOR gates as shown in figure 5, and behaves according to the truth table in table 1. The ‘S’ stands for ‘set’ and the ‘R’ for ‘reset’ as can be seen in the truth table.

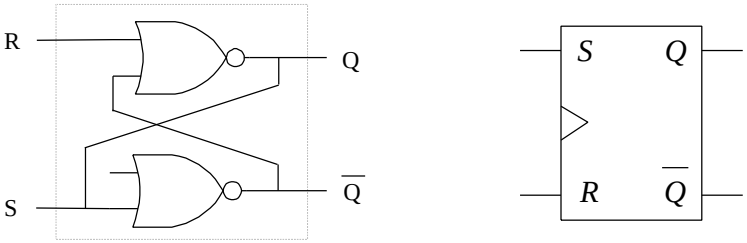


Figure 5 S-R Latch implementation using NOR gates.

The latch has two outputs which are complementary. These outputs are referred to as Q and /Q. If both S and R are raised at the same time then both Q and /Q will be HIGH; although this is physically possible, it does not make sense if Q and /Q are to be complementary. Thus, this condition is not allowed.

The S-R latch is somewhat restrictive, since both inputs cannot be HIGH at the same time. The other latch types are therefore based on the S-R latch but have additional logic which removes the input restrictions.

S	R	Q+
0	0	Q

0	1	0
1	0	1
1	0	Not Allowed

Table 1. S-R Latch Truth Table

2.5.1.2 D-Type Latches (Transparent Latches)

A single-input latch can be formed by adding some logic to the controlled S-R latch in Figure 6; this gives rise to the D-type latch in Figure 6. This latch is often called a *transparent* latch, since data on the input passes right through to the output as long as the control input is HIGH. If the control input is set LOW, then the latch holds whatever data was present when the control went LOW. With this type of latch, the control is usually called a *gate*.

The behaviour of the D-type latch is shown below in table 2.

D	G	Q+
X	0	Q
0	1	0
1	1	1

Table 2 Truth Table for a D-Type Latch

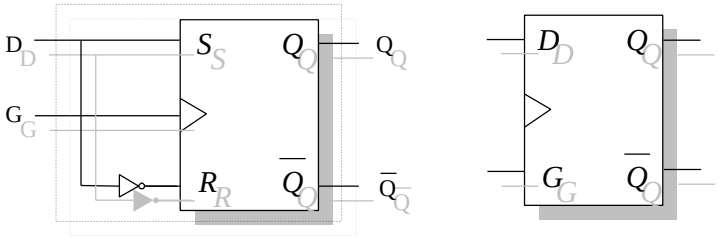


Figure 6 S-R Latch implementation of a D-Type Latch

2.5.1.3 J-K Latches

Another two-input latch can be derived from the S-R latch as shown in Figure 7. This is called a J-K Latch and operates in the same manner as an S-R latch, except that the condition where both inputs are HIGH is now allowed. The behaviour of the J-K-type latch is shown below in table 3.

J	K	Q+
0	0	Q
0	1	0
1	0	1
1	1	/Q

Table 3 Truth Table for a J-K Latch

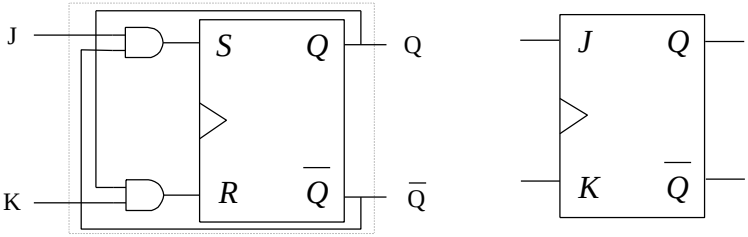


Figure 7 J-K Latch Implementation using an S-R Latch

There are still some potential problems here for the case where J and K are both HIGH. If J and K are left HIGH for too long, the output may change more than one time; if left HIGH forever the output will oscillate. Thus, J and K should not be asserted for a time longer than the propagation delay of the latch. There are also some potential race conditions if J and K are not asserted and removed at exactly the same time. If one of the inputs is raised slightly ahead of the other, it may give the output time to react, giving the wrong output once the second input is raised.

There are several ways to derive transfer functions for J-K latches. Two can be derived directly from Karnaugh maps. The basic transfer functions are listed below in table 4.

Q+ = J * /Q + /K * Q	/Q+ = /J * /Q + K * Q
Q+ = Q	/Q+ = /Q

:+: (J * /Q + K * Q)	:+: (J * /Q + K * Q)
Q+ = /Q :+: (/J * /Q + /K * Q)	/Q+ = Q :+: (/J */Q + /K * Q)

Table 4 Basic Transfer Functions

Where :+: is taken to mead ‘Unless’. The XOR gate can be used as an ‘UNLESS’ operator. In other words the function, A = X :+: Y can be interpreted as:
‘ A will have the same value as X UNLESS Y is true’.

2.5.1.4 T-Type Latches

T-type latches are formed by connecting the J and K inputs of a J-K latch together to form a single input, as shown in figure 8. This latch has two possible functions:

1. Hold the present state
2. Invert the output

The ‘T’ stands for ‘trigger’ or ‘toggle’ depending on who you talk to. When the ‘T’ is HIGH a change at the output is triggered; or to put it another way, raising T causes the output to TOGGLE. This useful output also has a down side - if the T is left high for too long the output will oscillate! However since there is a significant advantage to this latch it only has one input therefore the race condition problems of the J-K latch have been eliminated. There is now no way to get the output into a fixed state without knowing what the previous state was. This device is therefore not very useful without some kind of initialisation circuit - this device typifies the ups and downs of electronic design!

T	Q+
0	Q
1	/Q

Table 5. The Truth Table for a T-Type Latch

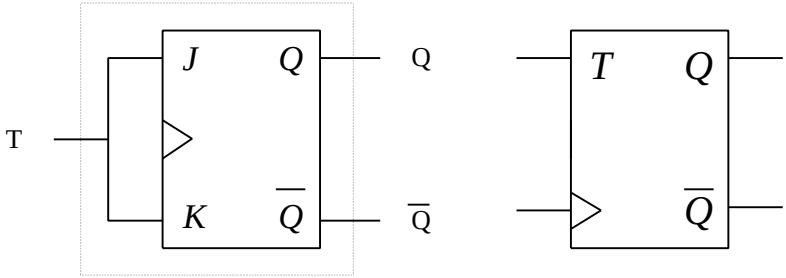


Figure 8 T-Type Latch implemented using a J-K Latch

2.5.2 Clocked Flip-Flops

Latches can be modified by adding a *clock* input. The purpose of the clock is to delay any output changes until the clock signal changes. Latch control inputs (such as the gate) are *level-sensitive* , clock inputs are generally edge-sensitive (or edge triggered), meaning that output transitions can occur only when a clock transition is detected. A device can be classified as positive edge triggered of negative edge triggered, depending on whether it responds to the rising or falling edge of the clock signal.

The clock provides two basic advantages:

1. Removes the hazards inherent in J-K and T flip flops, since all inputs will have settled by the time the clock edge arrives.
2. Only one transition is possible for each clock edge.

The clock also allows the design of synchronous systems. The entire system is then regulated by the clock.

The basic operation of the four types of flip-flop does not change with the addition of the clock - the output changes are merely made to wait for the clock edge. Thus, the basic transfer equations for most of the flip-flops are the same.

2.5.2.1 D-Type Filp-Flops

This is the only flip-flop- type whose basic transfer characteristic changes because the clock input replaces the gate input. Thus the equation becomes:

Q+ := D/Q+ := /D

That is, whatever data appears on the input will be transferred to the output after the next clock edge. The input is not changed in any way.

The simplicity of the D- flip-flop makes it the most popular and widely used flip-flop.
However functions are sometimes more conveniently expressed using J-K flip-flops or by using T-type flip-flops.

If we replace the D signal with the transfer function for one of the other flip-flop types we can emulate that flip-flop type in the D-type flip-flop. This is equivalent to taking a latch and placing a clocked D-type flip-flop after the latch output for synchronisation. Figure 9 illustrates how each flip-flop can be emulated in a D-type flip-flop.

D	G	Q+
X	0	Q
0	1	0
1	1	1

Table 6. Truth Table for a D-Type Latch.

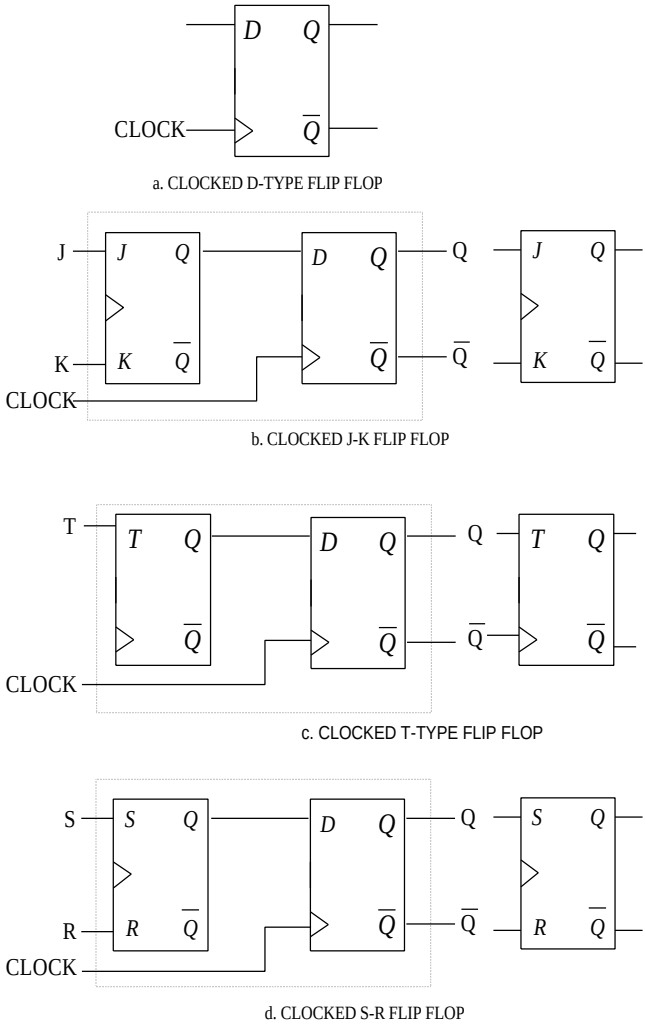


Figure 9 Flip-Flop emulation using D-Types

2.5.3 Binary Numbers

Digital circuits usually only have two states HIGH or LOW. The two states can represent any of a variety of 'bits' (binary digits) of information such as one bit of a number or whether a switch is open or closed etc. How a digital level can represent part of a number can be handled using a variety of numbering systems.

A decimal (base 10) number is simply a string of integers that are understood to multiply successive powers of 10, the individual products then being added together.

For example:

$146.05 = 1 \times 10^{2} + 4 \times 10^{1} + 6 \times 10^{0} + 0 \times 10^{-1} + 5 \times 10^{-2}$

Ten symbols (0 to 9) are needed, and the power of 10 each multiplies is determined by its position relative to the decimal point. If we want to represent a number using two symbols only (0 and 1), we use the *binary*, or base-2, number system. Each 1 or 0 then multiplies a successive power of 2.

For example:

$1100_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 12_{10}$

The individual 1's and 0's are called 'bits' (binary digits). The subscript (always given in base 10) tells you what number system we are using, and often it is essential in order to avoid confusion, since the symbols all look the same.

We can convert a number from binary to decimal by the method just described. To convert the other way, we keep dividing the number by 2, and write down the remainders. To convert 12_{10} to binary:

12/2	= 6	remainder	0
6/2	= 3	remainder	0
3/2	= 1	remainder	1
1/2	= 0	remainder	1

from which we can get $12_{10} = 1100_2$. Note that the answer comes out in the order LSB to MSB.

2.5.3.1 Binary Coded Decimal (BCD)

Another way to represent a number is to encode each decimal digit into binary coded decimal (BCD). This system requires a 4-bit group for each digit, for example:

146_{10} = 0001 0100 0110 (BCD)

This representation is NOT the same as binary representation, which in this case would be: $146_{10} = 10010010_2$. You can think of the bit positions (starting from the right) as representing 1,2,4,8,10,20,40,80,100,200,400,800 etc. It can be seen that BCD is very wasteful of bits - so why use it? BCD is ideal if you want to display a number in decimal because all you need to do is converts each BCD character to the appropriate decimal and display it. For this reason BCD is commonly used for input and output of numeric information.

2.5.3.2 2's Complement

Sooner or later it becomes necessary to represent negative numbers in binary, particularly when computation is involved. There are a number of ways of doing this:

Sign-magnitude, Offset binary or 2's complement.

2's complement is by far the most popular way of representing negative numbers. In this system positive numbers are represented as simple unsigned binary. The system is rigged up so that a negative number is then simply represented as the binary number you add to a positive number of the same magnitude to get zero. To form a negative number, first complement each of the bits of the positive number (i.e. write 1 for 0 and vice versa, this is called 1's complement and then add 1 - 2's complement. The table 7 shows sign-magnitude, offset binary and 2's complement representation.

Integer	Sign-Magnitude	Offset Binary	2's Complement
+7	0111	1111	0111
+6	0110	1110	0110
+5	0101	1101	0101
+4	0100	1100	0100
+3	0011	1011	0011
+2	0010	1010	0010
+1	0001	1001	0001
0	0000	1000	0000
-1	1001	0111	1111
-2	1010	0110	1110
-3	1011	0101	1101
-4	1100	0100	1100
-5	1101	0011	1011
-6	1110	0010	1010
-7	1111	0001	1001
-8	-	0000	1000
(-0)	1000	-	-

Table 7 Sign magnitude, Offset binary & 2's Complement representation.

3

BENEFITS OF USING PLDS

3.1 INTRODUCTION

This chapter outlines some basic information about the benefits of using PLDs essential to those who are unfamiliar with Programmable Logic Devices (PLDs).The information may also be useful to those who are current users of programmable logic.

The specific issues which need to be addressed are:

- What is a PLD?
- What other implementations are possible?
- What advantages do PLDs have over other implementations?
- What are the industry standard devices?

3.2 WHAT IS A PLD?

In general a programmable logic device is a circuit which can be configured by the user to perform a logic function. Most ‘standard’ PLDs consist of an AND array followed by an OR array, either (or both) of which is programmable. Inputs are fed into the AND array which performs the desired AND functions and generates product terms. The product terms are then fed into the OR array. In the OR array, the outputs of the various product terms are combined to produce the desired outputs.

PAL Devices

The PAL device has a programmable AND array followed by a fixed OR array. The fact that the AND array is programmable makes it possible for the devices to have many inputs. The OR array is fixed so this ensures that the devices are small (which means low cost) and fast.

3.3 WHAT OTHER IMPLEMENTATIONS ARE POSSIBLE?

There are essentially five alternatives to programmable logic:

- Discrete Logic
- FPGAs
- Gate Arrays
- Standard Cell Circuits
- Full Custom Circuits

3.3.1 Discrete Logic

Discrete logic, or conventional TTL logic (e.g. 7400 series logic) has the advantage of familiarity, hence its popularity. The major drawback with these devices is the form the packaged parts take with many gates in one package, this can lead to device wastage. For example your design may need two AND gates and only one OR gate, with discrete logic the you will need one 7408 and one 7432. This leads to wastage of two of the four AND gates in the 7408 package and three of the four AND gates in the 7432 package.

Designing with discrete chips can also be very tedious. Each design decision affects the layout of the board & changes are difficult to make. The design is also more difficult to document, making it harder to debug and maintain later. These items all contribute to a long design cycle when discrete chips are used extensively.

3.3.2 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs), as the name suggests these devices are a uniform array of gates that can be updated by the designer on the board as and when required. In some cases these devices are known as ASICs (Application Specific Integrated Circuits), that is each device is configured by the designer to perform a function particular to his application. In most ASIC technologies, gate level interconnections are established when the device is manufactured (i.e. custom and semi-custom ASICs).

FPGAs stand apart in that they contain ‘fuses’ that allow the interconnection pattern to be loaded and changed after the device is manufactured. Being a

standard, in stock part, FPGAs can be programmed and tested in an application as soon as the design is completed, compared with 4 - 6 weeks to manufacture a gate array. If the design does not work as anticipated modifications to the design can be made quickly and easily, saving both time and money.

The main disadvantage of these devices is that they are generally slower and more costly compared to custom Gate Arrays. The main disadvantage over PALs and GALs is that the delay through the device is design specific and cannot be determined easily until the design is complete.

There are two main FPGA architectures:

1. SRAM based
2. Anti-fuse based

SRAM based FPGAs are configured every time the device is powered on. The 'fuse' configuration is held in a configuration Serial Memory along side the FPGA or is held in a co-processor and is loaded in to the FPGA on power up using an algorithm held within the FPGA. This type of architecture is very flexible and lends itself to clever methodologies like configuration of the FPGA on the fly (e.g. 40K series of FPGAs from Atmel that can load in FIR filter coefficients without effecting processing in other parts of the device). The main advantage of this type of device is design flexibility, the device can be changed time and time again.

The main disadvantage of this architecture is security, i.e. the configuration bit stream can be read as it configures the FPGA and designs can be copied.

Anti-fuse devices on the other hand are single shot devices and are very secure. Once the design is complete and has been simulated thoroughly the device can be programmed by 'blowing' the device internal fuses. Once the device is programmed it cannot be reprogrammed. This has the obvious disadvantage that there may be a certain amount of device wastage in the design phase if the design is not right first time.

If an FPGA design is mature, not likely to change and being used in quite large volumes it may be worth considering converting the design to a Gate Array. This conversion service is offered by most of the silicon vendors. This conversion process will save money if production quantities are sufficient the offset the

NRE (None Recurring Engineering charge). This process also has the added benefit of making the design secure.

3.3.3 Gate Arrays

Gate Arrays can be split into two main groups:

Standard Cells

Standard Cell devices are classified as semi-custom ASICs. Gate Arrays consist of pre-processed wafers of logic elements (gates) that require only between one and three mask steps of metal interconnect to complete the fabrication process. The GA structure usually consists of columns of transistor arrays that will be configured to form basic logic functions chosen from a cell library, and are surrounded by I/O pads. The designer simply chooses which logic function will be connected on the chip, the ASIC vendor will process the chip by using layers of metal to connect the logic gates with each other and the output pads to complete the array.

Full Custom Circuits

Full Custom ASICs are built up by the designer layer by layer, producing a truly on-of-a-kind integrated circuit. This type of device is very costly and time consuming process.

3.4 Why Do We Use PLDs?

Any logic design can be done using PLDs. If you normally begin your design by:

- Using AND or OR functions
- Thinking of 7400 series components
- Using truth tables
- State diagrams

you are already on the path to using PLDs.

Designing a microprocessor based system, with memory and I/O?
How about all that “glue” logic you use to interface with the bus , provide chip selects, and any unusual signals required by special chips.
Most of these functions are currently done with 7400 series TTL.

How about using a PLD instead?

Designing a standalone PC board which uses a state machine to control multiple output signals?
Using latches to synchronise signals?
Using counters to divide down master clock frequencies?
Converting parallel-to-serial and back again?
All of these functions fit easily in modern PLDs. Almost anything found in your **TTL** Databook can be replaced with your own, PERSONALISED, programmable logic device.

The four main reasons for using PLDs over other implementations are:

- Ease of Design
- Performance
- Reliability
- Cost Savings

Ease of Design

The support tools available such as CUPL and Kanda PLD Starter Kit for use in designing with PLDs greatly simplify the design process by making the lower level implementation detail transparent. In a matter of one or two hours, a first time PLD user can learn to design using a PAL device, program it and implement the design in system.

The design support tools consist of design software e.g. CUPL and a programmer e.g. Kanda PLD Programmer. The design software is used to generate the design; the programmer is used to configure the device. The software provides the link between the higher level design and the low level programming details.

All of the available design software packages perform essentially the same tasks. The design is specified with relatively high-level constructs; the software takes the design and converts it into a form which the programmer uses to configure the PLD. Most software packages provide logic simulation, which allows one to debug the design before actually programming the device. Now though with the emergence of flash based PLDs the design can be tested in system quickly and easily, if the design is not quite right just modify the design

and re-program in a matter of seconds. The high level design file also serves as ready written and complete documentation of the design. This documentation can be even easier to understand than traditional schematics and is effectively as self documenting design!
The convenience of programmable logic lies in the ability to customise a standard, off-the-shelf product. PLDs can be found in stock to suit a wide range of speed and power requirements.

Board layout is vastly simplified with the use of programmable logic. PLDs offer great flexibility in the location of inputs and outputs on the device. Since larger functions are implemented inside the PLD, board layout can begin once the inputs and outputs are known. The details of what will actually be inside the PLD can be worked out independently of the layout. Any needed design changes can be taken care of entirely within the PLD without re-laying out or re-tracking the board.

Performance

Speed is one of the main reasons that designers use PAL devices. The PAL devices can provide equal or better performance than the fastest discrete logic available. The fastest PAL devices are being developed on the newest technologies to gain every extra nanosecond of performance.

Reliability

Reliability is an area of increasing concern. As systems get larger and more complex, the increase in the amount of circuitry tends to reduce system reliability - there are more things to go wrong. PLDs can therefore increase system reliability by reducing the amount of components on the board.

With the reduction in units and board space PCBs can be laid out less densely, which also increases the reliability of the board. This also reduces cross talk between devices and other potential sources of noise.

Cost

For any design to be practical and viable cost is a major concern. Cost influences the decision to redesign or produce a completely new product. But the calculation of total system cost can be misleading if not all aspects are considered. Many costs can appear to be hidden or difficult to measure. For

example, it is difficult to quantify cost of lost market share if a product is late to market. The greatest visible saving of PLDs over discrete solutions are derived from the fact that a single PLD can replace many discrete devices. Board space saved can be in excess of 25% when PLDs are used.

Another cost benefit when using PLDs is that one PAL device can be used in many designs, the user has not committed that device to any one design until the device has been programmed. This means that inventory can be stocked for several different designs in the form of one device.

3.5 INTRODUCTION TO PLD APPLICATIONS

PLDs can be used in a variety of different ways and in many applications, here are just a few:

- Glue Logic
- State machines
- Synchronisation
- Decoders
- Counters
- Bus Interfaces
- Parallel-to-serial
- Serial-to-parallel
- Subsystems

3.6 What Are The Industry Standard Devices

You may have heard the term ‘industry standard’ PALs and GALs - but what does it mean? For the answer we need to look at the history of the Programmable Logic Device.

Historically, the first PLDs were Programmable Logic Arrays (PLAs). A PLA is a combinatorial, two-level AND-OR device that can be programmed to realise 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).

We might describe such a device as ‘an n*m PLA with p product terms.’ In general p is far less than the number of n-variable minterms (2^n).

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 is a fixed OR array.

PAL devices were introduced by Monolithic Memories, now part of AMD, in the late 1970s. Key innovations in the first PAL devices were the use of a fixed OR array and bi-directional input/output pins. Even though PLAs were more flexible the PAL devices are faster (the signal passes through one array of fuses) and cheaper, hence PALs are more popular.

These ideas are well illustrated by the PAL16L8, its programmable AND array has 64 rows and 32 columns. 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 16L8. The PAL16L8 has up to 16 inputs and up to 8 outputs, it is housed in a package with only 20 pins, including two for power and ground (the corner pins 10 and 20). This magic is achieved through the use of six bi-directional pins (13-18) that may be used as inputs or outputs or both.

From there several types of Bipolar PALs emerged:

16L8	Active LOW device
16R8	Registered device
16RP8	Registered & Programmable Polarity device
16H8	Active HIGH
16P8	Programmable Polarity
16C8	Complementary Outputs
16V8	Versatile device using output macrocells

The part number is derived as shown below:

No. of inputs	**output type**	**No. of outputs**
16	V	8

From these family of devices super PALs or GALs have emerged which can mimic any of the common 20-pin and 24-pin PALs with a single 16V8, 20V8 or 22V10 devices. The ‘V’ in the part number signifies a generic device.

The generic GALs available from Atmel are flash based which means they can be reprogrammed time and time again and are erased electrically.

The Atmel range of PALs can emulate any device. Thus it is an industry standard part, i.e. it can emulate the same pinouts and functionality as any device on the market.

There are three emulation modes (these modes will be auto-selected by CUPL):

Registered Mode: Pin 1 and pin 11are permanently configured as clock and output enable, respectively. These pins cannot be configured as dedicated inputs in the registered mode.

Any register usage will make the compiler select this mode.
The following registered devices can be emulated using this mode:

16R8, 16RP8, 16R6, 16RP6, 16R4 & 16RP4.
Complex Mode: Pin 1 and pin 11 become dedicated inputs and use the feedback paths of pin 19 and pin 12 respectively. Because of this feedback path usage, pin 19 and pin 12 do not have the feedback option in this mode.

The compiler selects this mode when applications are combinational with OE required.

The following devices can be emulated using this mode:

16L8, 16H8 & 16P8

Simple Mode: All feedback paths of the output pins are routed via the adjacent pins. In doing so, the two inner most pins (pins 15 and 16) will not have the feedback option as these pins are always configured as dedicated combinatorial output.
The compiler selects this mode when all outputs are combinational without OE control.
The following devices can be emulated using this mode:

10L8, 10H8, 10P8, 12L6, 12H6, 12P6, 14L4, 14H4, 14P4, 16L2, 16H2, 16P2.

Part Number	Package pins	AND-gate inputs	Primary Inputs	Bi-directional comb. outputs	Registered outputs	Combinational outputs
PAL16L8	20	16	10	6	0	2
PAL16R4	20	16	8	4	4	0
PAL16R6	20	16	8	2	6	0
PAL16R8	20	16	8	0	8	0
PAL20L8	24	20	14	6	0	2
PAL20R4	24	20	12	4	4	0
PAL20R6	24	20	12	2	6	0
PAL20R8	24	20	12	0	8	0

Table 8 Characteristics of standard bipolar PLDs

4

INTRODUCTION TO THE PLD DESIGN CYCLE

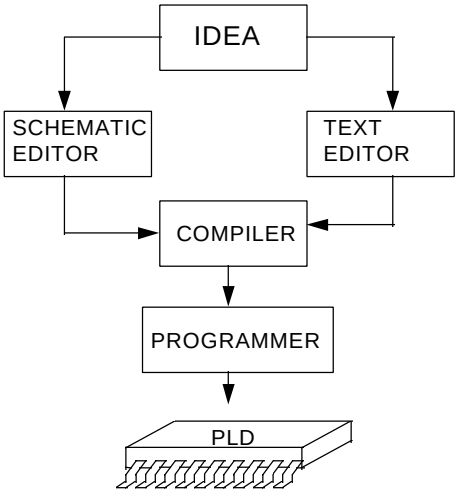
4.1 INTRODUCTION

This section is a step by step approach to your first two simple designs and are intended to demonstrate the basic PLD design implementation process.

We will talk about device programming, describing all of the steps that are necessary to program a PLD. When we talk about device programming we mean all of the steps from building equations to actually physically programming a device using a device programmer e.g. Kanda PLD programmer.

Before we embark on any design examples we need to understand the PLD design process flow, a simplistic design process diagram is shown below:

PLD DESIGN PROCESS



How do you translate your idea into a working prototype?

First you need a computer with a text editor or schematic editor. Either describe your design in text or draw a schematic of your design.

Next, turn the Logic Compiler loose on your design (e.g. **CUPL**). First it will check for typographical errors and any inconsistencies in your specification. It will then attempt to reduce your logic using standard logic reduction theory. Then a simulator will check the test vectors you input, comparing your logic description against the predicted responses. This is an excellent way to verify your design.

At the end of the compilation process, a JEDEC file is output. This file is a standard format accepted by most programming hardware (e.g. the device programmer in the **Kanda PLD developers kit**). Next download this file to you chosen programmer and programme your device.

Atmel devices are all **FLASH** based so they can be programmed time and time again!.

Take your configured PLD and plug it into your system. If it does not work as anticipated simply modify you description and repeat the process. Its easy!

THE RIGHT TOOLS FOR THE JOB!

Development tools are a very important part of designing with PLDs that's why we have put together in one package everything you need to 'get going with PLDs'.

The **Kanda** Developers kit includes:

- Programmer for Atmel 16V8,20V8, 22V10 reprogrammable PLDs with professional ZIF socket
- Training module and Logic Tester
- Enhanced CUPL PLD compiler software with on-line tutorial
- Kanda Template generator
- Kanda Programming software
- Complete training package including applications examples

- 16V8 Device (estimated 100,000 programming lives)

This comprehensive package includes everything required to describe your design, test, simulate and then program your devices. It costs less than traditional PLD programmers and also gives an excellent introduction to using PLDs.

The training is extensive and covers enabling principles such as Boolean logic, using CUPL and lots of application examples that can be tested on the training module. Once up to speed, the user has a complete development system for PLDs.

Includes everything needed to take a programmable logic design from conception to working design.

We have now seen an overview of the design process, we are now ready to complete our first design! This section will cover a simple combinational design, Basic Gates, and a basic sequential design, flip-flop emulation using D-types.

4.2 CONSTRUCTING A COMBINATIONAL DESIGN

4.2.1 Basic Gates Design Example

The first example we will try is a very simple combinational circuit consisting of the basic logic gates shown in figure 10. This will be useful for those designs where you are integrating random logic into a PAL device to save space and money.

As you can see from the diagram, 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 conceptualise and design, but due to the simplicity of these circuits we are already in a position to start the implementation.

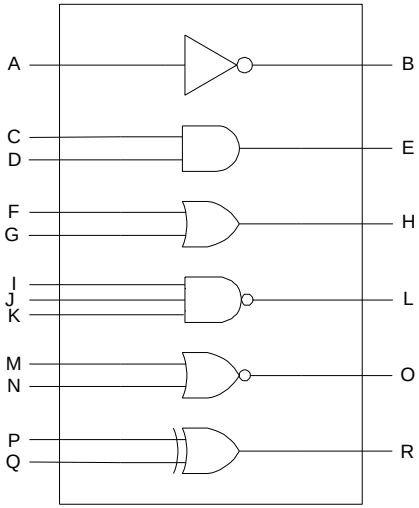


Figure 10 Basic Logic Gates

4.2.1.1 Building the Equations

We will start by generating Boolean equations. If we are using CUPL to describe our designs we must convert our boolean equations into a form that can be recognised by the CUPL compiler (see Appendix C on how this can be done quickly and simply by using the Kanda Enhanced CUPL templates).

CUPL Logical Operators

Operator	**Description**	**Example**
!	Logical negation	!A
&	Logical AND	A & B
#	Logical OR	A # B
$	Logical XOR	A $ B

The first function to be generated is an inverter. This is specified according to Figure 10 as:

B = !A

Here the ‘equal’ sign (=) is used to assign a function to output B. The exclamation point (!) 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 10 we can write:

E = C & D

Here we use the ‘equal’ sign again, but this time we have introduced the ampersand (&) to indicate the AND operation. This equation may be read:

E is TRUE if C AND D are TRUE

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

H = F # G

The ‘hash’ 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 may 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; and active-LOW device has an AND-OR-INVERT structure, which inverts the function at the output.

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 the dollar sign ($) 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 summarised in the figure below:

```
B = !A                  ; Inverter

E = C & D               ; AND gate
H = F
  # G                   ; OR gate

L = !I
  # !J
   # !K                 ; NAND gate

O  = !M & !N            ; NOR gate

R  = P & !Q
   # !P & Q             ; XOR gate
```

Basic Gates Equations

Building the Design File

Once the design has been conceptualised, 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, in this case we are using CUPL (please note the design file is more easily generated using Kanda Enhanced CUPL as this package has template generators and on-line help facility see Appendix B).

The CUPL description or design file has a specific set-up or template. The file is case sensitive and care must be taken when entering the data. Any standard text editor can be used to enter the text e.g. WordPad. Below is the example CUPL description file for this design example.

Note: We need to use Clock (Pin 1) and OE (Pin 11) pins to get enough I/O resources on a 16V8 device. It would be fine on a 20V8. As the training board with the Kanda kit is wired with 8 outputs to LEDs, then this design would not work well on it and a second example is given at the end of the section.

GATES.PLD

```
---------------------------------------------------------------------------------------------
Name                    GATES EXAMPLE;
Partno                  KP0001;
Date                    6/11/97;
Rev                     01;
Designer                Karen Parnell;
Company                         Kanda;
Assembly                None;
Location                None;
Device                  G16V8A;
/*************************************************************************************/
/*          Gates Example CUPL Description File                                      */
/*************************************************************************************/
/* Inputs :  define inputs to build simple gates */
Pin  1 = A;
Pin  2 = C;
```

```
Pin  3 = D;
Pin 4  = F;
Pin 5  = G;
Pin 6  = I;
Pin 7  = J;
Pin 8  = K;
Pin 9  = M;
Pin 11 = N;
Pin 12 = P;
Pin 13 = Q;

/* Outputs:    define outputs as active HIGH levels*/
Pin 14 = B;
Pin 15 = E;
Pin 16 = H;
Pin 17 = L;
Pin 18 = O;
Pin 19 = R;

/* Logic Equations */

B = !A;

E = C & D;          /* AND gate */

H = F
   # G;             /* OR gate */

L = !I
   # !J
    # !K;           /* NAND gate */

O  = !M & !N;       /* NOR gate */

R  = P & !Q
    # !P & Q;      /* XOR gate */
```

Next, turn the Logic Compiler loose on your design (e.g. **CUPL**). First it will check for typographical errors and any inconsistencies in your specification. It will then attempt to reduce your logic using standard logic reduction theory.

In this example CUPL was used. The CUPL compiler must be able to access the device library file (CUPL.DL), which contains a description of each of the target Atmel devices supported in this version of CUPL. The library describes the physical characteristics of each device, including internal architecture, number of pins and valid output pins, and also describes the logical characteristics, including registered and non-registered pins, number of product terms, fuse map information and down load format information.

The target device is referenced using device mnemonics. The mnemonic is composed of a device family prefix and industry standard part number suffix. For example the device mnemonic for PAL16V8 is P16V8.

CUPL can output the following files:

A JEDEC- compatible ASCII download file (filename. JED) for input to a device programmer.

An ASCII Hex download file (filename.HEX) available for PROMs.

An HL download file (filename.HL) available for Signetics IFL devices.

An absolute file (filename. ABS) for use by CSIM, the CUPL logic simulation program.

An error listing file (filename.LST) that lists errors in the original source file.

A documentation file (filename.DOC) that contains expanded logic equations, a variable symbol table, product term utilisation and fusemap information.

P-CAD PDF file (filename.PDF) that can be translated by PDIFIN into PC-CAPS symbol representing the pinouts of the programmable logic device.

A Berkeley PLA (filename.PLA) for use by the Berkeley PLA tools.

An Open PLA file (filename.PLA) for use by various back end fitters.

A simulation Input File can then be written to exercise your design (you don't need to do this but it is advisable and could save you re-designing if your finished design does not work!).

GATES.SI
Simulation Input File for Gates Example

```
Name            GATES;
Partno          XXXX;
Date            6/11/97;
Designer        Karen Parnell;
Device          G16V8A;
/*************************************************************************** */
/* Simulation Input File for Gates Example                                  */
/*************************************************************************** */
/* Order: define order, polarity, and output spacing                        */
/*  of stimulus and response values                                         */

Order:          /* Inputs */
/* Outputs */
/*  Vectors: define stimulus and response values with                       */
/* header and intermediate messages for the simulator listing               */
/* Note: Don't care state (x) on inputs is reflected in outputs where       */
/* appropriate*/

Vectors:
0 1 HL HH               /* 0,1 = Input values */
1 0 HL HL               /* L,H = Output values */
```

Then a simulator (e.g. CSIM available from Kanda) will check the test vectors you input, comparing your logic description against the predicted responses. This is an excellent way to verify your design.

Second Gates example for Kanda board

This example is configured for the 10 inputs and 8 outputs available on this board. The LEDs on this board are wired for LOW (0) = ON. This is standard industry practice, but it means that a 1 at the output pin of the PLD will switch the LED OFF.

Therefore you need to either get used to this way of thinking – 0 = LED ON, 1 = LED OFF) or invert the outputs in the design file. This is simple to do e.g.

E = C & D; becomes !E = C & D;

GATES2.PLD

```
---------------------------------------------------------------------------------------------
Name                    GATES EXAMPLE;
Partno                  KP0001;
Date                    6/11/97;
Rev                     01;
Designer                Karen Parnell;
Company                         Kanda;
Assembly                None;
Location                None;
Device                  G16V8A;
/*************************************************************************************/
/*          Gates Example CUPL Description File                                      */
/*************************************************************************************/
/* Inputs :  define inputs to build simple gates */
Pin 1 = A;
Pin 2 = C;
Pin 3 = D;
Pin 4  = F;
Pin 5  = G;
Pin 6  = I;
Pin 7  = J;
Pin 8  = K;
Pin 9  = P;
Pin 11 = Q;


/* Outputs:    define outputs as active HIGH levels*/
Pin 14 = B;
Pin 15 = E;
Pin 16 = H;
Pin 17 = L;
Pin 18 = R;
```

```
/* Logic Equations */
/*Note : Outputs are inverted so LED is ON when result of equation is 1 */
/* as LEDS are active low */
!B = !A;

!E = C & D;        /* AND gate */

!H = F
   # G;           /* OR gate */

!L = !I
   # !J
    # !K;         /* NAND gate */



!R  = P & !Q
    # !P & Q;     /* XOR gate */
```

At the end of the compilation process, a JEDEC file is created. This file is a standard format accepted by most programming hardware (e.g. the **Kanda PLD developers kit**). More information about JEDEC files can be found on the CD or install folder. Next download this file to your PLD programmer.

4.3 CONSTRUCTING A REGISTERED DESIGN

4.3.1 Basic Flip-Flops Design Exercise

Next we will do a very simple registered design: we will be designing all of the basic flip-flop types (figure 11). We will start the design by reviewing briefly the behaviour 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.

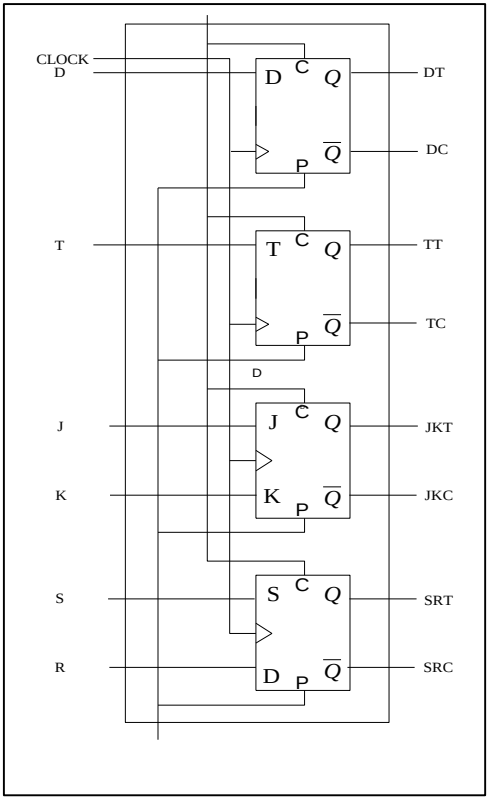


Figure 11 Basic Flip-flops

4.3.3.1 Building the D-type Flip-flop Equations

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

DT : = D

Where the use of ':=' here instead of '='. This indicates that the output is registered for this equation. The difference is illustrated in figure 12. We can also generate the complement signal (named DC) with the statement:

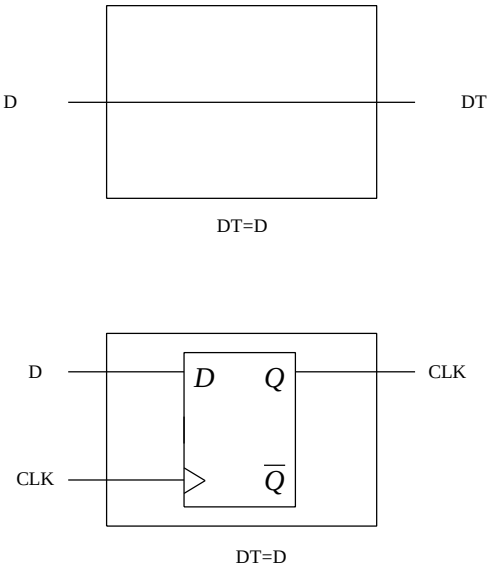


Figure 12 D-Type Flip Flop

DC := /D

As shown in Figure 11 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 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 look 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 omission from this design: what happens if we preset and clear at the same time? As the design stands 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 below:

```
EQUATIONS

DT = D * /CLR          /* D-type - active HIGH -output is D if not clear */
     + PR * /CLR       /* or if preset and not clear at the same time */

DC = /D * /PR          /* D-type - active LOW - output is /D if not preset */
     + CLR             /* or 1 if clear */

TT = T * /TT * /CLR    /* T-type - active HIGH -go HI if toggle and not*/
     + /T * TT * /CLR  /* clear 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     /* T-type - active LOW - go HI if toggle and not*/
     + /T * TC * /PR   /* preset stay HI if not toggle and not preset */
```

```
        + CLR           /* go high if clearing */

JKT = J * /JKT * /CLR   /* JK flip-flop - Active HIGH - go HI if J and not*/
        + /K * /JKT * /PR       /* clear stay HI if not K and not clear*/
        + PR * /CLR     /* go HI if preset and not clear at the same time

JKC = /J * /JKC * /PR   /* JK flip-flop - Active LOW -go HI if not J and not*/
        + K * /JKC * /PR        /* preset stay HI if not K and not preset*/
        + CLR                   /* go HI if clear*/

SRT = S * /CLR          /* SR flip-flop - Active HIGH - go HI if set and not*/
        + /R * SRT * /CLR       /* clear stay HI if not reset and not clear*/
        + PR * /CLR             /* go HI if preset and not clear at the same
time*/

SRC = R * /PR           /*SR flip-flop - Active LOW - go HI if reset and not*/
        + /S * SRC * /PR/* preset stay HI if reset and not preset*/
        + CLR                   /* go HI if clear*/

/* Emulating all flip-flops with D-type Flip-Flops */
```

4.3.1.2 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.

We are now in a position to complete the design file.

The quickest and easiest way to write a CUPL file in the correct format first item every time is to use the Kanda Template Generator included in the PLD Starter Kit.

It is assumed you have followed the installation instructions in Appendix B of this book and your software is installed correctly.

Open the Kanda PLD Editor and Programmer Window and click on ‘File’ at the top tool bar, Select ‘New from Template’ from the pull down menu.

Fill in the File Header boxes as follows to produce a CUPL design header:

Name:	Flip-Flops
Part No:	Your product part number
Rev:	Design Revision
Designer:	Your Name
Company:	Your Company Name for proper documentation practice and because specifications may be sent to sub-contractors for high volume PLD orders.
Assembly:	Enter the assembly name or number or the number of the PCB on which the PLD will be used.
Location:	Enter the PCB reference or co-ordinate where the PLD is located.
Filename:	Enter the filename of your design

Select the target device from the device list, in this case select 16V8. Now click on OK - this has now generated your CUPL design header!

Next we need to enter the 16V8 pin assignments in the Window that has appeared automatically. The Window will automatically pre-define certain 16V8 pins such as: Clk, Gnd and OE (Clock, Ground and Output Enable).

You are now ready to enter your specific design pin assignments as follows:

Pin 2 is an input, enter D for your D-type register input.
Pin 3 is an input, enter T for you T-type register input.
Pin 4 is an input, enter J for one input of your JK flip-flop.
Pin 5 is an input, enter K for one input of your JK flip-flop.
Pin 6 is an input, enter S for one input of your SR flip-flop.
Pin 7 is an input, enter R for one input of your SR flip-flop.
Pin 8 is an input, enter CLR to assign this pin as the Clear pin.
Pin 9 is an input, enter PR to assign this pin as the Preset pin.

All of the input pins have now been assigned, we will now move onto the I/O pins.

Pin 19 is an I/O (input or output pin), check the box to define as an output and enter DT in the relevant box.

Continue the same procedure for the following:

Pin 18 , enter DC
Pin 17, enter TT
Pin 16, enter TC
Pin 15, enter JKT
Pin 14, enter JKC
Pin 13, enter SRT
Pin 12, enter SRC.

Now select the OK button and you input and output pins will be automatically generated.

You can now enter the equations already written to the design file and produce a file similar to the one below.

Note: In this file we are using a new CUPL directive called $DEFINE. This lets us replace one symbol with another with the syntax
$DEFINE new symbol old symbol

For example, $DEFINE AND &
Now you can use AND instead of & in your equations.

```
-------------------------------------------------------------------------------------------------
Name            Flip-Flops;
    Partno          KP0002;
    Revision        01;
    Date            05/1/98;
    Designer        K.Parnell;
    Company         Kanda;
    Location        None;
    Assembly        None;
    Device          G16V8A;
    /*************************************************************************************/
    /*                                                                                   */
    /*              Flip-Flops                                                           */
    /* All of the basic flip-flop types using D-type flip flops.              */
    /*  T, J-K and S-R flip-flops.    */
    /*************************************************************************************/
/* To make logic easier to follow, define alternative symbols that can be used */

$DEFINE         NOT     !     /*Alternate Negation;*/
```

```
$DEFINE         AND     &       /*Alternate AND*/
$DEFINE OR      #       /*Alternate OR;*/
$DEFINE         :+:     $       /*Alternate XOR;*/

/*  Inputs  */

Pin 1 = CLK;
Pin 2 = CLR;
Pin 3 = PR;
Pin 4 = T;
Pin 5 = D;
Pin 6 = J;
Pin 7 = K;
Pin 8 = S;
Pin 9 = R;

Pin 11 = OE;

/**      Outputs       **/
Pin 12 = SRC;
Pin 13 = SRT;
Pin 14 = JKT;
Pin 15 = DC;
Pin 16 = DT;
Pin 17 = JKC;
Pin 18 = TC;
Pin 19 = TT;

/**      Logic Equations       **/

!DT = D AND  NOT CLR            /* D-type - active HIGH -output is D if not
clear */
          OR PR AND NOT CLR;    /* or if preset and not clear at the same time */

!DC = !D AND !PR                /* D-type - active LOW - output is /D if not
preset */
          OR CLR;                       /* or 1 if clear */

!TT = T AND !TT AND !CLR        /* T-type - active HIGH -go HI if toggle and
not*/
```

```
        OR !T AND TT AND !CLR  /* clear stay HI if not toggle and not clear */
        OR PR AND !CLR;        /* go HI if preset and not clear at the same
time */

!TC = T AND !TC AND !PR        /* T-type - active LOW - go HI if toggle and
not*/
        OR !T AND TC AND !PR /* preset stay HI if not toggle and not preset */
        OR CLR;                        /* go high if clearing */

!JKT = J AND !JKT AND !CLR     /* JK flip-flop - Active HIGH - go HI if J and
not*/
        OR !K AND !JKT AND !PR         /* clear stay HI if not K and not
clear*/
        OR PR AND !CLR;         /* go HI if preset and not clear at the same
time*/

 !JKC=!J&!JKC&!PR#K&!JKC&!PR#CLR;
/* JK flip-flop - Active LOW -go HI if not J and not*/
/* preset stay HI if not K and not preset*/
/* go HI if clear*/

!SRT = S & !CLR                /* SR flip-flop - Active HIGH - go HI if set and
not*/
        OR !R & SRT & !CLR     /* clear stay HI if not reset and not clear*/
        OR PR & !CLR;  /* go HI if preset and not clear at the same time*/

!SRC = R & !PR     /*SR flip-flop - Active LOW - go HI if reset and not*/
        OR !S & SRC & !PR      /* preset stay HI if reset and not preset*/
        OR CLR;                        /* go HI if clear*/

/*END;   Flip-Flops;*/
```

Now you can save your file by selecting File and save as from the pull down menu. For this example we will save the file as flipflop.pld in the default sub-directory.

You can now compile your design by selecting Compile from the top tool bar.

The window gives you the following options:

Compile Options:

Generate Document File: This generates a document file containing fully expanded product terms for both intermediate and output pin variables and a fuse plot and chip diagram.

Generate Fuse Plot: This generates the complete device fuse plot in the listing file

Remove Unused OR-gates

Blow Security Fuse - This option programs the security fuse on the device.

Minimisation Options:

Level 1 Minimisation - Quick Minimisation
Level 2 Minimisation - Quine McClusky
Level 3 Minimisation - Presto
Level 4 Minimisation - Expresso

Select the required options from the list above and hit the OK button. For this design example I have selected, generate document file, generate fuse plot and Level 1 Minimisation.

If the syntax and format is correct you will have compiled your file successfully.

You will have the following files, which can be viewed in the same sub-directory:

flipflop.doc - This contains fully expanded product terms for both intermediate and output pin variables and a fuse plot and chip diagram.

flipflop.jed - This is a JEDEC file for downloading to a device programmer. It contains a fuse pattern but no test vectors.
flipflop.lst - This is the list file which is a recreation of the description file, except line numbers have been added and any error messages generated during compilation are appended at the end of the file.

These files are shown in figure 13 and figure 14 below.

Note: This compiled design does not use the $DEFINE statements to define the logical operators but instead using the standard CUPL operators.

```
******************************************************************************
                              Flip-Flops
******************************************************************************

CUPL(WM)   4.7b Serial# MW-65999997
Device     g16v8ma  Library DLIB-h-36-8
Created    Fri Jan 09 14:04:39 1998
Name       Flip-Flops
Partno     KP0002
Revision   01
Date       05/1/98
Designer   K.Parnell
Company    Kanda
Assembly   None
Location   None
================================================================
                        Expanded Product Terms
================================================================
DC =>
   !D & !PR
  # CLR

DT =>
   !CLR & D
  # !CLR & PR

END =>

JKT =>
   !CLR & J & !JKT
  # !JKT & !K & !PR
  # !CLR & PR

SRT =>
   !CLR & S
  # !CLR & !R & SRT
  # !CLR & PR
TC =>
   !PR & T & !TC
  # !PR & !T & TC
```

```
  # CLR

TT =>
   !CLR & T & !TT
  # !CLR & !T & TT
  # !CLR & PR

DC.oe  =>
   1

DT.oe  =>
   1

JKT.oe  =>
   1

SRT.oe  =>
   1

TC.oe  =>
   1

TT.oe  =>
   1
```

Symbol Table

Pin Pol	Variable Name	Ext	Pin	Pterms Type	Max Used	Min Pterms	Level
	CLK		1	V	-	-	-
	CLR		8	V	-	-	-
	D		2	V	-	-	-
	DC		18	V	2	7	1
	DT		19	V	2	7	1
	END		0	I	1	-	-
	J		4	V	-	-	-
	JKC		14	V	-	-	-

```
JKT              15   V   3   7   1
K                5    V   -   -   -
OE               11   V   -   -   -
PR               9    V   -   -   -
R                7    V   -   -   -
S                6    V   -   -   -
SRT              13   V   3   7   1
T                3    V   -   -   -
TC               16   V   3   7   1
TT               17   V   3   7   1
DC          oe   18   D   1   1   0
DT          oe   19   D   1   1   0
JKT         oe   15   D   1   1   0
SRT         oe   13   D   1   1   0
TC          oe   16   D   1   1   0
TT          oe   17   D   1   1   0

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
=========================================================
                    Fuse Plot
=========================================================

Syn   02192 - Ac0   02193 -

Pin #19  02048  Pol -  02120  Ac1 -
 00000 --------------------------------
 00032 x-----------------------x------
 00064 -------------------------x--x---
 00096 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00128 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00160 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00192 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00224 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #18  02049  Pol -  02121  Ac1 -
 00256 --------------------------------
 00288 -x----------------------------x--
```

```
 00320 ------------------------x-------
 00352 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00384 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00416 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00448 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00480 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #17  02050  Pol -  02122  Ac1 -
 00512 --------------------------------
 00544 ----x------x-------------x------
 00576 -----x----x--------------x------
 00608 ------------------------x--x---
 00640 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00672 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00704 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00736 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #16  02051  Pol -  02123  Ac1 -
 00768 --------------------------------
 00800 ----x----------x-------------x--
 00832 -----x--------x--------------x--
 00864 ------------------------x-------
 00896 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00928 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00960 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00992 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #15  02052  Pol -  02124  Ac1 -
 01024 --------------------------------
 01056 --------x----------x-----x------
 01088 -------------x-----x---------x--
 01120 ------------------------x--x---
 01152 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01184 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01216 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01248 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #14  02053  Pol x  02125  Ac1 -
 01280 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01312 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

LEGEND X : fuse not blown

- : fuse blown

==========

Chip Diagram

==========

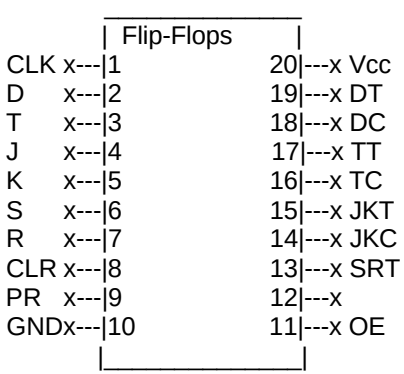


Figure 13 Flip-Flop.doc

```
CUPL(WM)        4.7b  Serial# MW-65999997
Device          g16v8ma  Library DLIB-h-36-8
Created         Fri Jan 09 14:04:39 1998
Name            Flip-Flops
Partno          KP0002
Revision        01
Date            05/1/98
Designer        K.Parnell
Company         Kanda
Assembly        None
Location        None
*QP20
*QF2194
*G0
*F0
*L00000 11111111111111111111111111111111
*L00032 01111111111111111111111110111111
*L00064 11111111111111111111111110110111
*L00256 11111111111111111111111111111111
*L00288 10111111111111111111111111111011
*L00320 11111111111111111111111101111111
*L00512 11111111111111111111111111111111
*L00544 11110111111011111111111110111111
*L00576 11110111110111111111111110111111
*L00608 11111111111111111111111110110111
*L00768 11111111111111111111111111111111
*L00800 11110111111111101111111111111011
*L00832 11111011111111011111111111111011
*L00864 11111111111111111111111101111111
*L01024 11111111111111111111111111111111
*L01056 11111111011111111110111110111111
*L01088 11111111111110111110111111111011
*L01120 11111111111111111111111110110111
*L01536 11111111111111111111111111111111
*L01568 11111111111111101111111110111111
*L01600 11111111111111111111101110011111
*L01632 11111111111111111111111110110111
*L02048 11111010010010110101000000110000
*L02080 00110000000110000001100100000000
*L02112 00000000111111111111111111111111
```

```
*L02144 11111111111111111111111111111111
*L02176 111111111111111111
*C5FF2
*_2669
```

Figure 14 Flip-Flop.jed

The flip-flops may also be defined thus:

```
Q0.D = Q1 & Q2 & Q3;    /* output pin w/ D flip flip*/
Q1.J  = Q2 # Q3;        /* output pin w/ JK flip-flop*/
Q1.K = Q2 & !Q3;
```

The design can now be simulated or tested using the Kanda application board after the device has been programmed.

5

COMBINATIONAL DESIGN EXAMPLE

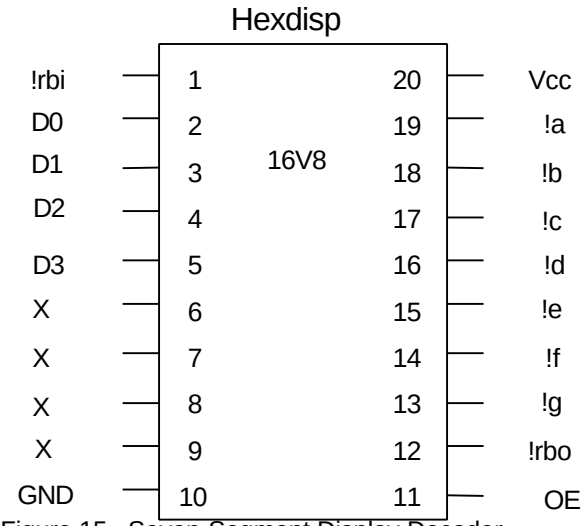
5.1 INTRODUCTION

Programmable logic devices are ideal for use as encoders or decoders. The following design is an example of a commonly required Seven-Segment LED display decoder. By providing the appropriate binary input on the four input pins the relevant seven segment equivalent is shown on the display. The inputs and outputs are shown in the following table:

Binary Input D3 D2 D1 D0	Display Output (HEX)
0 0 0 0	**0**
0 0 0 1	**1**
0 0 1 0	**2**
0 0 1 1	**3**
0 1 0 0	**4**
0 1 0 1	**5**
0 1 1 0	**6**
0 1 1 1	**7**
1 0 0 0	**8**
1 0 0 1	**9**
1 0 1 0	**A**
1 0 1 1	**B**
1 1 0 0	**C**
1 1 0 1	**D**
1 1 1 0	**E**
1 1 1 1	**F**

5.2 Seven-Segment Display Decoder

This application example is a hexadecimal-to-seven-segment decoder capable of driving common-anode LEDs. It incorporates both a ripple-blanking input (to inhibit displaying leading zeroes) and a ripple blanking output for easy cascading of digits.



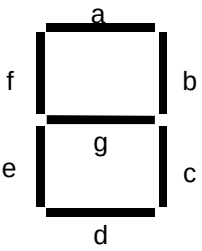


Figure 15. Seven-Segment Display Decoder

The segments in the display, labelled a-g, correspond to the outputs in the diagram. Figure 16 shows the source file HEXDISP.PLD.

```
---------------------------------------------------------------
                HEXDISP.PLD
      Name            Hexdisp;
      Partno          KMP003;
      Revision         01;
      Date            05/1/98;
      Designer        K. Parnell;
      Company         Kanda;
      Location        None;
      Assembly        None;
      Device          G16V8A;
      /*************************************************************************/
     /*                                                    a                  */
     /* This is a hexadecimal-to-seven-segment           -----                */
      /* decoder capable of driving common-anode        |     |              */
      /* LEDs. It incorporates both a ripple-         f |     | b            */
      /* blanking input (to inhibit displaying          | g   |              */
      /* leading zeroes) and a ripple blanking output     -----              */
      /* to allow for easy cascading of digits          |     |              */
      /*                                              e |     | c            */
      /*                                                |     |              */
      /*                                                -----                */
      /*                                                  d                  */
       /*************************************************************************/
/** Input group   */
      pin [2..5] = [D0..3];     /*  data input lines to display  */
      pin 1      = !rbi;       /* ripple blanking input        */
      pin 11     = OE;

     /** Output Group **/
      pin [19..13] = ![a,b,c,d,e,f,g];  /* Segment output lines  */
      pin 12       = !rbo;              /* Ripple Blanking output  */

      /** Declarations and Intermediate Variable Definitions          */
      FIELD data      = [D3..0];           /* hexadecimal input field  */
      FIELD segment =[a,b,c,d,e,f,g];        /* Display segment field   */

$DEFINE ON 'b'1                  /* segment lit when logically"ON"   */
$DEFINE OFF 'b'0                /* segment dark when logically "OFF"  */
$DEFINE  OR  #              /* To use OR instead of # symbol */
```

Figure 16. Display Decoder Source File (HEXDISP.PLD)

The first part of the file provides archival information and a description of the intended function of the design.

Pin declarations are made corresponding to the inputs and outputs in the design diagram. In the "Declarations and Intermediate Variables" section, field assignments are made to group the input pins into a set named data and the output pins into a set named segment. ON and OFF are defined respectively as binary 1 and binary 0.

The '$FIELD' range operation has a constant field with a range of values. Using this format greatly reduces the logic equations used.

```
-------------------------------------------------------------------
/** Logic Equations  **/
/*                a       b       c       d       e       f       g       */
segment =
 /* 0 */      [   ON,    ON,    ON,    ON,    ON,    ON,    OFF ] & data:0 & !rbi
 /* 1 */   #  [   OFF,  ON,    ON,    OFF,   OFF,   OFF,   OFF ] & data:1
 /* 2 */   #  [   ON,    ON,    OFF,  ON,    ON,    OFF,  ON   ] & data:2
 /* 3 */   #  [   ON,    ON,    ON,    ON,    OFF,   OFF,  ON   ] & data:3
 /* 4 */   #  [   OFF,  ON,    ON,    OFF,   OFF,  ON,    ON   ] & data:4
 /* 5 */   #  [   ON,    OFF,  ON,    ON,    OFF,  ON,    ON   ] & data:5
 /* 6 */   #  [   ON,    OFF,  ON,    ON,    ON,    ON,    ON   ] & data:6
 /* 7 */   #  [   ON,    ON,    ON,    OFF,   OFF,   OFF,  OFF ] & data:7
 /* 8 */   #  [   ON,    ON,    ON,    ON,    ON,    ON,    ON   ] & data:8
 /* 9 */   #  [   ON,    ON,    ON,    ON,    OFF,  ON,    ON   ] & data:9
 /* A */   #  [   ON,    ON,    ON,    OFF,  ON,    ON,    ON   ] & data:A
 /* b */   #  [   OFF,   OFF,  ON,    ON,    ON,    ON,    ON   ] & data:B
 /* C */   #  [   ON,    OFF,   OFF,  ON,    ON,    ON,    OFF ] & data:C
 /* d */   #  [   OFF,  ON,    ON,    ON,    ON,    OFF,  ON   ] & data:D
 /* E */   #  [   ON,    OFF,   OFF,  ON,    ON,    ON,    ON   ] & data:E
 /* F */   #  [   ON,    OFF,   OFF,   OFF,  ON,    ON,    ON   ] & data:F;

rbo = rbi & data:0;
-------------------------------------------------------------------
```

The logic equations are set up as a function table to describe the segments that are lit up by each input pattern. Comments create a header for the function

table, listing the output segments across the top and the input numbers vertically down the side. Each line of the table describes a decoded hex value and the segments of the display that the hex value turns on or off.

For example, the line for an input value of 4 is written as follows:

[OFF, ON, ON, OFF, OFF, ON, ON] & data:4

The function table format expresses the intent of the design more clearly than equations; that is, the example above shows that an input value of 4 turns segment a off, segment b on, segment c on, and so on.

When the design has been entered into the text editor a compiler can be set use on the design. Most design compilers have built-in minimisation which can be selected by the designer, e.g. Quine McClusky (if you are using the Kanda PLD Starter Kit select minimisation level 1 or above). When the compile option has been selected the compiler will check your design for typographical and syntax errors and then it will then attempt to reduce your logic using standard logic reduction theory.

After successful compilation the CUPL software produces HEXDISP.DOC and is shown overleaf, this document includes expanded product terms, fuse map and device diagram.

```
*******************************************************************************
                              Hexdisp
*******************************************************************************

CUPL(WM)        4.7b Serial# MW-65999997
Device          g16v8as  Library DLIB-h-36-2
Created         Wed Feb 25 14:17:52 1998
Name            Hexdisp
Partno          KMP003
Revision        01
Date            05/1/98
Designer        K.Parnell
Company         Kanda
Assembly        None
Location        None
```

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

a =>
    !D0 & !D1 & !D2 & !D3 & !rbi
  # D0 & D1 & D2 & D3
  # D1 & !D2 & !D3
  # D0 & D2 & !D3
  # !D0 & D1 & D2
  # !D1 & !D2 & D3
  # !D0 & D1 & !D2 & D3
  # !D0 & !D1 & D2 & D3

b =>
    !D0 & !D1 & !D2 & !D3 & !rbi
  # D0 & !D2 & !D3
  # !D0 & D1 & !D2
  # !D0 & !D1 & D2 & !D3
  # D0 & !D1 & D2 & D3
  # D0 & D1 & D2 & !D3
  # !D1 & !D2 & D3

c =>
    !D0 & !D1 & !D2 & !D3 & !rbi
  # D0 & !D2 & !D3
  # D0 & !D1 & D2 & D3
  # !D2 & D3
  # D2 & !D3

d =>
    !D0 & !D1 & !D2 & !D3 & !rbi
  # !D0 & !D1 & D2 & D3
  # D1 & !D2 & !D3
  # D0 & !D1 & D2
  # !D0 & D1 & D2
  # !D1 & !D2 & D3
  # D0 & D1 & !D2 & D3

data =>
    D3 , D2 , D1 , D0
```

```
e =>
    !D0 & !D1 & !D2 & !D3 & !rbi
  # !D0 & D1 & D2 & D3
  # !D0 & D1 & !D3
  # !D0 & !D2 & D3
  # D0 & D1 & D3
  # !D1 & D2 & D3

f =>
    !D0 & !D1 & !D2 & !D3 & !rbi
  # D0 & D1 & D2 & D3
  # !D1 & D2 & !D3
  # !D0 & D1 & D2
  # !D0 & !D1 & D2 & D3
  # !D2 & D3

g =>
    D0 & D2 & D3
  # D1 & !D2
  # !D1 & D2 & !D3
  # !D0 & D1 & D2
  # !D1 & !D2 & D3

rbo =>
    !D0 & !D1 & !D2 & !D3 & rbi

segment =>
    a , b , c , d , e , f , g

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

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

        D0                2        V        -      -       -
        D1                3        V        -      -       -
        D2                4        V        -      -       -
        D3                5        V        -      -       -
        OE               11        V        -      -       -
```

```
!       a               19      V       8       8       1
!       b               18      V       7       8       1
!       c               17      V       5       8       1
!       d               16      V       7       8       1
        data             0      F       -       -       -
!       e               15      V       6       8       1
!       f               14      V       6       8       1
!       g               13      V       5       8       1
!       rbi              1      V       -       -       -
!       rbo             12      V       1       8       1
        segment          0      F       -       -       -


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

================================================================
                          Fuse Plot
================================================================

Syn   02192 - Ac0   02193 x

Pin #19  02048  Pol x  02120  Ac1 x
 00000 -xx--x---x---x-----------------
 00032 x---x---x---x-----------------
 00064 ----x----x---x-----------------
 00096 x-------x----x-----------------
 00128 -x--x---x-----------------------
 00160 -----x---x--x-----------------
 00192 -x--x----x--x-----------------
 00224 -x---x--x---x-----------------
Pin #18  02049  Pol x  02121  Ac1 x
 00256 -xx--x---x---x-----------------
 00288 x--------x---x-----------------
 00320 -x--x----x-----------------------
 00352 -x---x--x----x-----------------
 00384 x----x--x---x-----------------
 00416 x---x---x----x-----------------
 00448 -----x---x--x-----------------
 00480 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
Pin #17  02050  Pol x  02122  Ac1 x
 00512 -xx--x---x---x-----------------
 00544 x--------x---x-----------------
 00576 x----x--x---x-----------------
 00608 ---------x--x-----------------
 00640 --------x----x----------------
 00672 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00704 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00736 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #16  02051  Pol x  02123  Ac1 x
 00768 -xx--x---x---x-----------------
 00800 -x---x--x---x-----------------
 00832 ----x----x---x----------------
 00864 x----x--x----------------------
 00896 -x--x---x----------------------
 00928 -----x---x--x-----------------
 00960 x---x----x--x-----------------
 00992 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #15  02052  Pol x  02124  Ac1 x
 01024 -xx--x---x---x-----------------
 01056 -x--x---x---x-----------------
 01088 -x--x--------x----------------
 01120 -x-------x--x-----------------
 01152 x---x-------x-----------------
 01184 -----x--x---x-----------------
 01216 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01248 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #14  02053  Pol x  02125  Ac1 x
 01280 -xx--x---x---x-----------------
 01312 x---x---x---x-----------------
 01344 -----x--x----x----------------
 01376 -x--x---x----------------------
 01408 -x---x--x---x-----------------
 01440 ---------x--x-----------------
 01472 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01504 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #13  02054  Pol x  02126  Ac1 x
 01536 x-------x---x-----------------
 01568 ----x----x----------------------
 01600 -----x--x----x----------------
 01632 -x--x---x----------------------
 01664 -----x---x--x-----------------
```

```
 01696 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01728 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01760 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #12  02055  Pol x  02127  Ac1 x
 01792 -x-x-x---x---x-----------------
 01824 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01856 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01888 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01920 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01952 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01984 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 02016 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx


LEGEND    X : fuse not blown
        - : fuse blown
============================================================
                      Chip Diagram
============================================================

                    _____________
                   |  Hexdisp    |
          !rbi x---|1          20|---x Vcc
            D0 x---|2          19|---x !a
            D1 x---|3          18|---x !b
            D2 x---|4          17|---x !c
            D3 x---|5          16|---x !d
               x---|6          15|---x !e
               x---|7          14|---x !f
               x---|8          13|---x !g
               x---|9          12|---x !rbo
           GND x---|10         11|---x OE
                   |_____________|
```

The expanded product terms show how the compiler expanded the product terms in the design. The fuse map is used by the device programmer to set the fuses in the target device and finally the device diagram shows you how the input and output pins have been assigned.

The design can now be tested, first we need to program the device. The device can be programmed using the Kanda PLD programmer (instructions of how to use the programmer are detailed in Appendix B).

Once your device is programmed either put you device in you target system or place the device into the Kanda application board; ensuring the device is the correct orientation. You can now test your design by using the onboard 7-segment display or LEDs and the appropriate switches.

6

SEQUENTIAL DESIGN EXAMPLE

6.1 INTRODUCTION

PLDs are ideal for implementing both combinational and sequential logic designs. In this section we will discover how to implement a Decade up/down counter using a '16V8' PAL. This is a very useful example and may be modified to suit many applications. It shows how registers may be described using CUPL both quickly and simply.

6.2 DECADE UP/DOWN COUNTER

This example describes a four-bit up/down decade counter with a synchronous clear capacity. The counter also provides an asynchronous ripple carry output for cascading multiple devices.

The source file to implement the counter uses CUPL state machine syntax. Figure 17 shows the counter design and its states.

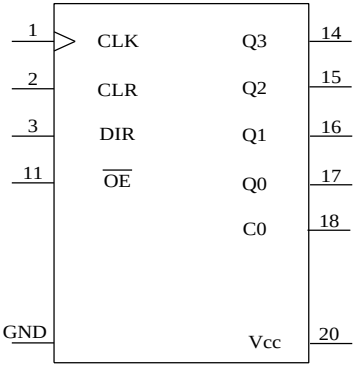


Figure 17. Up/Down Counter Diagram

The input signal ‘dir’ determines the direction of the count. When ‘dir’ is high, the count goes down one on each clock; when ‘dir’ is low, the count goes up one on each clock. The ‘clr’ signal performs a synchronous reset.

Figure 18 shows the CUPL source file (COUNT10.PLD, provided in the CUPL package) that implements the design.

```
-----------------------------------------------------------------
            /*    COUNT10.PLD */
    Name            Count10;
    Partno          KMP0004;
    Revision        02;
    Date            05/1/98;
    Designer        K.Parnell;
    Company         Kanda;
    Location        None;
    Assembly        None;
    Device          G16V8;
    /*****************************************************************/
    /*                                                               */
    /*              Decade Counter                                   */
    /* This is a 4-bit up/down decade counter with                   */
    /* synchronous clear capability. An asynchronous                 */
    /* ripple carry output is provided for cascading                 */
    /* multiple devices. CUPL state machine syntax                   */
    /* is used                                                       */
    /*****************************************************************/
    /** Inputs **/
    Pin 1 = clk;            /* counter clock                        */
    Pin 2 = clr;            /* counter clear input                  */
    Pin 3 = dir;            /* counter direction input              */
    Pin 11 = !oe;           /* Register output enable               */

    /** Outputs    **/
    Pin [17..14] = ![Q3..0];     /* counter outputs  inverted 0= LED ON
        */
    Pin 18 = !carry;            /* ripple carry out inverted */
                                        /*      as LED ON = 0           */

/* Declarations and Intermediate Variable Definitions          */
```

```
      field count = [Q3..0];      /* declare counter bit field        */
$define S0 'b'0000
$define S1 'b'0001
$define S2 'b'0010
$define S3 'b'0011
$define S4 'b'0100
$define S5 'b'0101
$define S6 'b'0110
$define S7 'b'0111
$define S8 'b'1000
$define S9 'b'1001

field mode = [clr,dir];    /* declare field mode control           */
up = mode:0;               /* define count up mode                 */
down = mode:1;             /* define count down mode               */
clear = mode:[2..3];       /* define count clear mode              */
/*---------------------------------------------------------------------*/

            /* Logic Equations */
sequence count {     /*    free running counter */
present S0        if up        next S1;
                  if down      next S9;
                  if clear     next S0;
                  default      next S0;
present S1        if up        next S2;
                  if down      next S0;
                  if clear     next S0;
present S2        if up        next S3;
                  if down      next S1;
                  if clear     next S0;
present S3        if up        next S4;
                  if down      next S2;
                  if clear     next S0;
present S4        if up        next S5;
                  if down      next S3;
                  if clear     next S0;
present S5        if up        next S6;
                  if down      next S4;
                  if clear     next S0;
present S6        if up        next S7;
```

```
                if down     next S5;
                if clear    next S0;
present S7      if up       next S8;
                if down     next S6;
                if clear    next S0;
present S8      if up       next S9;
                if down     next S7;
                if clear    next S0;
present S9      if up       next S0;
                if down     next S8;
                if clear    next S0;
                out         carry;      /*next carry;*/
        /* assert carry output */
}
```

Figure 19 Logic Equations

The first part of the file provides archival information and a description of the intended function of the design, including compatible PLDs.

Pin declarations are made corresponding to the inputs and outputs in the design diagram.

The "Declarations and Intermediate Variable Definitions" section contains declarations that simplify the notation.

The name "count" is assigned to the output variables Q3, Q2, Q1, and Q0.

The ‘$DEFINE’ command is used to assign names to ten binary states representing the state machine output. The state name can then be used in the logic equations to represent the corresponding binary number.

The ‘FIELD’ keyword is used to combine the ‘clr’ and ‘dir’ inputs into a set called mode. Mode is defined by the following equations:

```
up = mode:0;
down = mode:1;
clear = mode[2..3];
```

Mode represents the inputs ‘clr’ and ‘dir’, so the three equations above are equivalent to the following equations:

```
up = !clr & !dir ;
down = !clr & dir ;
clear = (clr & !dir) # (clr & dir) ;
```

The three modes are defined as follows:

up - Both the dir and clr inputs are not asserted.

down - The dir input is asserted and clr is not asserted.

clear - The clr input is asserted and dir is either asserted or not asserted.

The "Logic Equations" section contains the state machine syntax that specifies the states in the counter. In the first line, the SEQUENCE keyword identifies count (that is, Q3, Q2, Q1, and Q0) as the outputs to which the state values apply.

Conditional statements have been written to specify the transition from each possible present state to a next state, for each of the three modes. For example, when the present state is S4, if the mode is up, the counter goes to S5; if the mode is down the counter goes to S3; or if the mode is clear, the counter goes to S0. As this example shows, one advantage of the state machine syntax is that it clearly documents the operation of the design. In this example, state 0 (binary value 0000) is defined, because it is the result of the ‘clr’ signal. It is recommended that all designs have a valid 0000 defined to avoid being stuck at state 0. For example, in this design, if a state that hasn't been defined occurs at power-on, such as hexadecimal A-F, none of the conditions described in the equations is met, so the state goes to state 0 (hex value 0000). If 0000 has not been defined as a valid state, the counter stays at state 0.

Figure 20 shows how this example could have been written as a virtual design. It is the same file, but it has been modified where necessary to show the difference between a virtual design and a device specific design.

```
            COUNT10.PLD
Name            Count10;
Partno          KMP0004;
Revision        01;
Date            05/1/98;
Designer        K. Parnell;
Company         Kanda;
Location        None;
Assembly        None;
Device          VIRTUAL;
/*****************************************************************/
/*                                                               */
/*              Decade Counter                                   */
/* This is a 4-bit up/down decade counter  with                  */
/* synchronous clear capability. An asynchronous                 */
/* ripple carry output is provided for cascading                 */
/* multiple devices. CUPL state machine syntax                   */
/* is used                                                       */
/*****************************************************************/
     /** Inputs **/
Pin = clk;          /* counter clock                             */
Pin = clr;          /* counter clear input                       */
Pin = dir;          /* counter direction input                   */
Pin = !oe;          /* Register  output enable                   */

/* Outputs */
Pin = ![Q3..0];     /* counter outputs   inverted                */
Pin = !carry;       /* ripple carry out inverted LED on = 0 */

/* Declarations and Intermediate Variable Definitions            */
field count = [Q3..0];    /* declare counter bit field           */

$define S0 'b'0000
$define S1 'b'0001
$define S2 'b'0010
$define S3 'b'0011
$define S4 'b'0100
$define S5 'b'0101
$define S6 'b'0110
$define S7 'b'0111
```

```
    $define S8 'b'1000
    $define S9 'b'1001

    field mode = [clr,dir];    /* declare field mode control      */
    up = mode:0;               /* define count up mode            */
    down = mode:1;             /* define count down mode          */
    clear = mode:[2..3];       /* define count clear mode         */

 ---------------------------------------------------------------------
  Up/Down Counter Source File (virtual)

 ---------------------------------------------------------------------
          /* Logic Equations */
 sequence count {     /*    free running counter       */
present S0        if up       next S1;
                  if down     next S9;
                  if clear    next S0;
present S1        if up       next S2;
                  if down     next S0;
                  if clear    next S0;
present S2        if up       next S3;
                  if down     next S1;
                  if clear    next S0;
present S3        if up       next S4;
                  if down     next S2;
                  if clear    next S0;
present S4        if up       next S5;
                  if down     next S3;
                  if clear    next S0;
present S5        if up       next S6;
                  if down     next S4;
                  if clear    next S0;
present S6        if up       next S7;
                  if down     next S5;
                  if clear    next S0;
present S7        if up       next S8;
                  if down     next S6;
                  if clear    next S0;
present S8        if up       next S9;
                  if down     next S7;
```

```
                if clear     next S0;
  present S9        if up        next S0;
                if down      next S8;
                if clear     next S0;
                out          carry;       /* assert carry output */
```

Figure 20 Virtual Design

It is possible to use some of the features of the CUPL pre-processor to considerably shorten this PLD file. Figure 21 will show how this same file could be written with a .i.repeat;$REPEAT structure which reduces the file size considerably.

COUNT2

```
Name            Count2;
Partno          KMP0005;
Revision        01;
Date            05/1/98;
Designer        K. Parnell;
Company         Kanda;
Location        None;
Assembly        None;
Device          G16V8;
/*****************************************************************/
/*                                                               */
/*              Decade Counter                                   */
/* This is a 4-bit up/down decade counter with                   */
/* synchronous clear capability. An asynchronous                 */
/* ripple carry output is provided for cascading                 */
/* multiple devices. CUPL state machine syntax                   */
/* is used                                                       */
/*****************************************************************/
/** Inputs **/
Pin 1 = clk;            /* counter clock              */
Pin 2 = clr;            /* counter clear input        */
Pin 3 = dir;            /* counter  direction input   */
Pin 11 = !oe;           /* Register output enable     */

/* Outputs      */
Pin [17..14] = ![Q3..0];    /* counter outputs inverted LED ON = 0  */
```

```
/* Declarations and Intermediate Variable Definitions    */
field count = [Q3..0];      /* declare counter bit field      */
field mode = [clr,dir];    /* declare field mode control                */
up = mode:0;               /* define count up mode           */
down = mode:1;              /* define count down mode        */
clear = mode:[2..3];        /* define count clear mode       */

 /* state machine description */
 sequence count {
   present 0
        if up & !clear  next 1;
        if down & !clear next 9;
        if clear        next  0;
 $REPEAT i=[1..9]
   present {i}
        if up & !clear  next {(i+1)%10};
        if down & !clear next {(i-1)%10};
        if clear next 0;
 $REPEND
      }
```

Figure 21. Up/Down Counter Source File

In this variation, we removed the $DEFINE statements because we use the raw numbers instead. The most significant change is that we used a $REPEAT loop to define most of the states instead of defining each state individually. It is possible to do this because all the states are identical in the sense that all the next states can be calculated from the present state. State 0 is done separately because when counting down the next state is -1 modulo 10, which cannot be handled by the compiler. We, therefore, define state 0 by itself and then all the other states are defined in one $REPEAT loop. The $REPEAT loop expands upon compilation to give a definition for each state. Notice that the statement indicating the next state is given as a calculation from the repeat variable 'i'.

In the loop, 'i' represents the number of the current state. The next state is therefore 'i+1'. This will work for all states except the last state. In the last state, the state machine must go back to state 0. To accomplish this, the formula to calculate the next state is given as '(i+1)%10'. This means 'i+1' `modulo 10. The number 10 represents the number of states. Therefore, when in state 9 the next state is calculated as 9+1 = 10 then modulo 10 which gives 0. A similar condition occurs in calculating the previous state except that we subtract 1

instead of adding it. You might have noticed that we defined state 0 separately. This was done because the $REPEAT variables can only handle positive numbers. If we had defined state 0 in the $REPEAT loop this would result in evaluating to next state -1 and the compiler would produce an unexpected result.

If we want to add a Carry on 9, then define the carry pin in outputs as Count2.pld

```
Pin 18 = !carry;            /* ripple carry  out  0= LED ON  */
```

Now change the repeat loop from 1..8 and add a new section for 9:

```
present 9
            if up & !clear  next 0;
            if down & !clear next 8;
            if clear        next  0;
             out carry;
```

7

POWER SAVING WITH PLDS

7.1 INTRODUCTION

Many designers today are looking for ways to reduce power consumption in their designs, to meet expanding market demands for portability and battery operated products. With the push to add more features to these products while maintaining the same board size and power budget, Programmable Logic Devices with the low power consumption are playing a larger role in design of these type of products. These devices allow the designer to add more features in the same board space, yet be able to maintain or decrease the overall power consumption of the system. Atmel offers a variety of PLDs in several densities with pin counts ranging from 20 to 160 pins that have this low power consumption or 'L' feature.

7.2 POWER CONSUMPTION FOR PLDS

Before discussing the detailed features of Atmel's low power PLDs, it is important to point out the two components for PLD power consumption.

1. Paverage = nCload Fop $(Vsupply)^2$
2. Pstandby = Icc, SB * Vsupply = Standby Power when the device is powered down

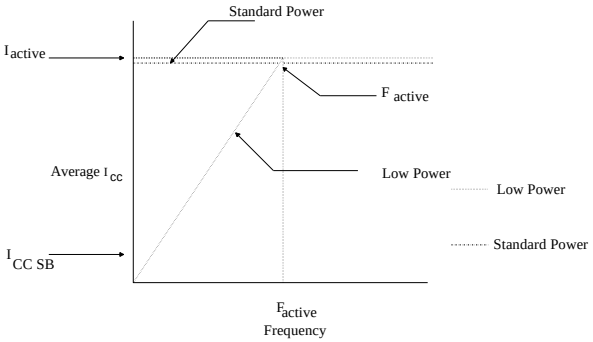
Where:
Paverage = average power consumed while the outputs are switching
Clod = load capacitance on each output pin
n = number of output pins switching
Fop = Frequency of operation
Vsupply = supply voltage.

As shown in equation 1 , a designer may be able to reduce the overall power consumption of his or her design by reducing the supply voltage, pin capacitive loading, or operating frequency. However, these alternatives are usually not practical. Another alternative a designer can choose, as equation 2 shows, is to use parts that consume as little standby power as possible during idle periods

when the device is not responding to input stimulus. Atmel's low power PLDs are ideally suited to this purpose.

7.3 POWER CONSUMPTION SAVINGS WITH ATMEL LOW POWER ("L") PLDS

Atmel Low Power PLDs save power by powering down automatically to a 'standby' of 'sleep' mode when no signal transitions occur on the inputs of the internal feedback's of the device. When an input signal transition occurs, the device responds by "waking up" to an active mode. Figure 1 shows the average Icc vs. frequency characteristics for Atmel's Standard and Low Power PLDs. For frequencies less than Factive, a low power device will automatically go through active and standby cycles to reduce the average current consumption. Compared with a standard power device , which always remains active, Low Power PLDs offer significant power savings. As the input signal frequency increases, the percentage of time a Low Power device is active will also increase proportionally until Factive is approached. For frequencies greater than or equal to Factive, a Low Power device will consume about the same amount of current as a standard power device.



Where: Icc,SB = in standby mode
I active = Icc at cuttoff frequency
Fop = Frequency of operation
F active = Cutoff frequency

Figure 1, chapter 7, Average Icc vs. Frequency for Atmel Standard and Low Power Devices

7.3.1 Standby Mode

When a Low Power device is in the standby mode, the internal fuse array powers down and the device draws standby current, Icc, SB from the system's supply. The device will enter the 'standby' mode automatically when no inputs or internal or internal feedback's have switched within a time period of Tactive.

Equation 3 shows how to calculate Tactive for a ATF16V8BL PLD device (Full data sheet can be found in appendix 1).

3. Tactive = 1/(2* Factive)

Where:

Tactive = Active time period
Factive = Cutoff Frequency

For Example, on the ATF16V8BL,

Iactive = 50 mA, Factive = 40 MHz, and Icc, SB = 5 mA.

So, Tactive = 1/(2 * 40 MHz) = 12.5 nsec.

Therefore, if no inputs or feedbacks have switched for 12.5 ns, the ATF16V8B will power down to the standby mode and only draw 5 mA from the supply.

During the standby mode all logic signals are latched so all outputs and internal feedbacks will remain valid. Since the device powers down to this mode automatically, separate power down pin is required.

7.3.2 The Active Mode

An Atmel Low Power PLD automatically wakes up to the active mode when it senses an input change from either Low to High or High to Low. When waking up, the internal fuse array is powered up and transient current increases from Icc,SB to a peak value of Iactive. The current remains at the peak value while the device is awake. The time required for the device to change current from Icc,Sb to Iactive during wake-up or vice versa during power down is called the 'wake-up' time or Twake. The wake-up time is already included within the Propagation Delay (Tpd) specification in the Atmel data sheet. Figure 2 shows what happens while the device is awake. In Figure 2 (I), the device awoke by a single input transition and saw no additional transitions. Hence, the device will stay awake for Tactive before entering the standby mode. In figure 2(ii), the

device sees several input transitions after the first transition that woke the device up. Therefore, it will stay awake for Tactive after the last input transition occurs before powering down.

The slopes on Figure 2 show the transient current slew rate required for an Atmel Low Power device to change current from Icc,SB to Iactive during the wake up time for both the wake-up and power down process. This slew rate can be calculated by equation 4.

4. Icc Slew Rate during wake up = dI/dt = iactive - Icc,SB / Twake

during power down = - dl/dt = Iactive - Icc,SB / Twake

For example, for a ATF16V8BL device,

Iactive = 50 mA, Icc,SB = 5 mA, and Twake = 3ns

Therefore, during wake up,

dl/dt = (50 mA - 5mA)/(3 ns) = 15 mA/ns.

During power down the slew rate is negative or -15 mA/ns.

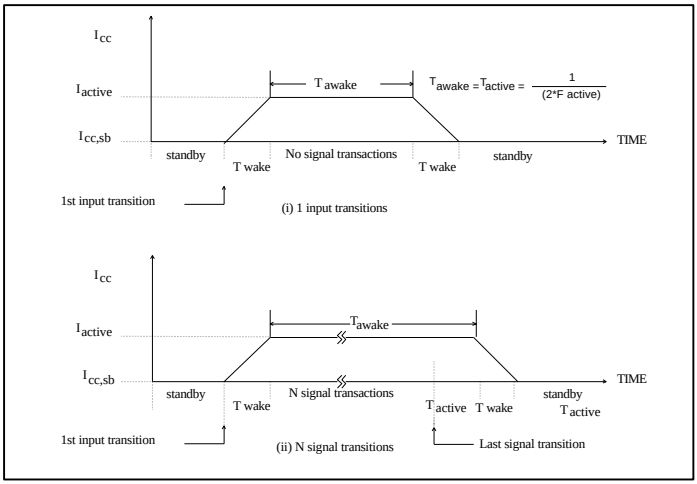


Figure 2, Chapter 7, Active Cycle

7.3.3 Average Icc vs. Peak Icc

The total current required for a low power device to complete an active cycle is the sum of transient and peak currents. Since the transient current cancels during the wake up and power down portions of the active cycle, its total contribution is zero. Therefore, the total current required is the peak current.

The average current required by the device when operating at a particular operating frequency can be approximately related to the peak current in the active cycle by,

5. For Fop = 0 Average Icc = Icc,SB
For Fop < Factive Average Icc ~ [Fop / Factive] * Ipeak + [1 - (Fop / Factive)] * Icc,SB

For Fop >= Factive Average Icc ~ Iactive,

Where:

Fop = Frequency of Operation
Ipeak = Iactive = Peak Current or current at the cutoff frequency
Factive = Cutoff frequency

For example, with an ATF22V10BL device (full data sheet can be found in Chapter 9),

For Fop = 30 MHz; Factive = 60 MHz,

Ipeak = Iactive = 70 mA, and Icc,SB = 8 mA.

Factive , Ipeak, and Icc,SB are determined from the Icc vs. frequency curves in the databook.

Average Icc ~ (30 MHz/60MHz) * (70 mA) + (1 - (30 MHz/60 MHz)) * 8 mA

= 39 mA.

Therefore, at an operating frequency of 30 MHz, the average current drawn by a ATF22V10BL device should be about 39 mA.

7.3.4 Supplying Transient and Peak Currents

In many applications with limited power sources, such as battery supplied or portable systems, the transient and peak currents required by the active cycle can be generated by the existing decoupling capacitiors on the board. Decoupling capacitors act as a temporary power source to the PLDs supply, supplying a Low Power device with the necessary amount of current so it can undergo active cycles automatically to save power.

To calculate the minimum decoupling capacitance needed, we first need to compute the total amount of charge required during an active cycle and use this amount to derive the capacitance. The total charge needed is,

6. Qtotal = Qactive + Qtransient

Itransient cancels during the active mode so Itransient = 0. Therefore, Qtransient = 0 as well. Substituting this result into equation 6 gives,

7. Qtotal = Qactive = (iactive) * [1/(2*Factive)]

The decoupling capacitance required can then be calculated as qtotal/dV and is,

8. Creq = Iactive * [1/(2*Factive)] / dV

Where dV is the maximum droop allowed in the supply voltage caused by draining this charge from the capacitors.

For example, with an ATF16V8BL device,

Assume dV = 100 mV maximum, Iactive = 50 mA, and Tactive = 12.5 ns.

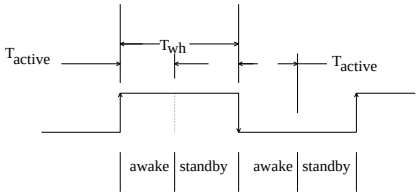
Therefore Qactive = (50 mA) * (12 ns) = .63nC and
Creq = (.63nC)/(100mV)= 6.3nF.

This is the minimum decoupling capacitance needed to supply the peak and transient currents required for the active time period, in this case 12.5ns.

A 0.22 uF ceramic or tantalum decoupling capacitor placed as close to the supply pin(s) as possible is adequate for supplying both the transient and peak current needs of a Low Power device.

7.3.5 How Duty Cycle Affects Power Consumption

A Low Power device normally wakes up twice for each clock input cycle. For example, with a 50% duty cycle input (Case 1, Figure 3), the device wakes up on the rising edge and falling edges of the input. If the input signal width Twh, is less than or equal to Tactive, as shown in Case 2 Figure 3, the device will wake up once during each input cycle. From Case 2 Figure 3, we see that the input duty cycle affects the power consumption of a Low Power device by reducing the time it is awake. If the input duty cycle is greater than 50% the device will consume the same power as Case 2 Figure 3.



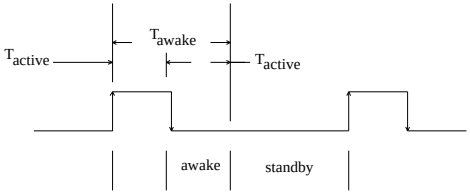
CASE 1

T_{wh} => T_{active}

T_{awake} = 2* T_{active}

2 "Wake-up" cycles per second

Figure 3 (i) 50% Duty Cycle



CASE 2

$T_{active} > T_{wh} => T_{w}$,

$T_{awake} = T_{wh} + T_{active}$

1 "Wake-up" cycles per second

T_{w} = Minimum input width (specified in the Atmel datasheet)

Figure 3 (ii) < 50% Duty Cycle

Figure 3, Chapter 7, Awake Time vs. Input Duty Cycle

7.4 ATMEL PLD PRODUCT SELECTIONS

In addition to Standard and Low Power PLDs, Atmel also offers Quarter Power and Low Voltage products. These versions are also available with the Low Power feature, Figure 4 shows the Average Icc vs. Frequency characteristics for all Atmel PLDs.

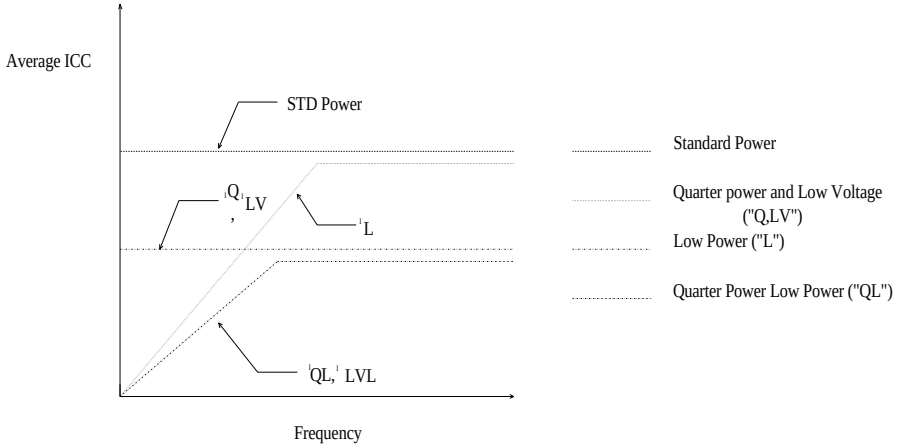


Figure 4, Chapter 7, Average Icc vs. Frequency for all Atmel PLDs

7.4.1 Atmel Standard Power and Low Power PLDs

Atmel Standard Power PLDs are high performance devices whose power consumption remains about the same with frequency. Low Power products approach Standard Power PLD power consumption at higher frequencies but save power at lower frequencies when the device is idle.

7.4.2 Quarter Power PLDs

Atmel Quarter Power PLDs are available in two versions, a Standard Power Quarter Power part (‘Q’ Suffix) and a Quarter Power device, which has Low Power feature (‘QL’ suffix). Both versions have approximately one half the average Icc of a Standard Power PLD and consume a quarter of the power of a

compatible Bipolar part. The ‘QL’ devices have the Quarter Power active current, plus the Low Power feature.

7.4.3 Atmel Low Voltage PLDs

Atmel Low Voltage PLDs are capable of operating down to 3.0V. Low Voltage PLDs include the letters ‘LV’ in the device name e.g. ATF16LV8. These devices save power because they can operate at a reduce supply voltage compared to Standard Power PLDs. Low Voltage PLDs are also available with the Low Power feature and include the ‘LV’ in the part name along with a ‘L’ suffix e.g. AT22LV10L.

7.5 SUMMARY

In this section we have discussed the features of Atmel’s Low Power PLD products and have seen that these devices can offer the system designer many benefits for applications where power consumption is a critical requirement. These benefits include:

- *Low Power Feature in ALL PLD and CPLD Products.*

Designers can use any Atmel PLD or CPLD product from 20 up to 68 pins, and still take advantage of the Low Power feature.

- *High Performance AND Power Consumption Savings.*

Designers can save power in their designs yet not sacrifice overall system operation speed. There is no separate delay for the Low Power feature. All delays are included in the Tpd specification in the Atmel data sheet.

- *No Separate Power Down Pin.*

Atmel’s patented ‘Low Power’ feature automatically powers down the device to a low power mode when no inputs or internal feedbacks are switching. Therefore, no separate power down pin is needed, since power down is automatic.

- *Increased System Reliability.*

Atmel Low Power parts consume lees power, and operate cooler. So thermal related issues are less of a concern in the overall system design.

8

PLD DESIGN APPLICATIONS

8.1 INTRODUCTION

The two following applications examples will hopefully give you a good indication of how PLDs can be used in the real world. They have been chosen as examples that may help you with your first design whether it be a sequential of combinational design.

8.2 Application 1: 7-Segment-to-Hex Encoder (Combinational)

There are times when you might like to use an available LSI chip that performs a convenient function (e.g. a stopwatch or calculator) as part of an instrument you are designing. The trouble is that the LSI chips usually provide outputs to drive 7-segment display directly, rather than the hexadecimal (or BCD) outputs that you want. Therefore this design example is an encoder that converts from 7-segment back to 4-bit binary.

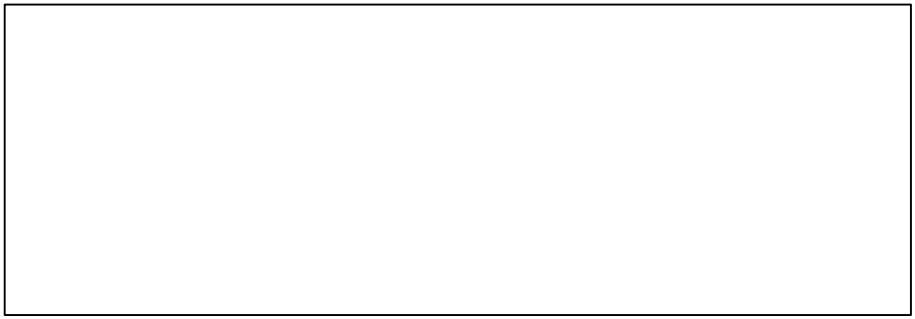


Figure 22 7-segment display codes

The inputs are the individual segment signals, which are always labelled a - f , see figure 22, which also shows how the digits a - f are represented with seven segment displays. Note that 9 and C can be represented in two ways, both of

which should be correctly recognised by the logic. The PAL chosen was an ATF16V8, a 20-pin combinational device.

Figure 23 shows the CUPL file for this design. We have the (positive true) segment derive signals a-g as inputs, and the (negative true) hexadecimal bits D0-D3 as outputs. CUPL lets you define intermediate variables that can be used in later expressions; in this case it is convenient to define the obvious variables zero through hexf, the possible displayed digits in terms of the segment inputs. These are simply large product (AND) terms of the input segment variables, which you can read from the digit shapes in figure XX. Finally, each binary output bit is written as the sum (OR) of the digit variables in which that bit is set. This completes the logic specification to CUPL.

CUPL first uses the intermediate variable definitions to write the D0-D3 expressions directly in terms of the input variables a-f. At this point the logic terms are in the desirable AND-NOR form. However, we're not finished yet, because the 16V8 permit at most 7 product terms in each sum, whereas we have 9,8,9 and 10 respectively, for the outputs D0-D3. One solution would be to string each output through a second OR gate, in order to get enough product terms in the sum; this is generally considered to be not a good idea as this doubles the propagation delay, though it wouldn't matter in a slow application like this. The better solution is to perform a logic minimisation, using logic identities, DeMorgan's formula etc.

We ran CUPL's minimiser (Best For Polarity), producing the product terms shown in figure 24. Luckily all fit within the 7-product constraint. CUPL also draws a fuse map for you as shown in figure 24 and contained within hexseg.DOC file.

The seghex.JED file can be used to program your device via the Kanda PLD programmer. In this example CUPL has made a laborious design problem simple.

```
NAME        SEGHEX;
PARTNo      FFFF;
DATE        12/01/98;
REVISION    1.0;
DESIGNER    KAREN PARNELL;
COMPANY     KANDA ;
ASSEMBLY    NONE;
LOCATION    NONE;
DEVICE      G16V8A;

/**  Inputs  **/

Pin 1 = CLK;
Pin 2 = a;
Pin 3 = b;
Pin 4 = c;
Pin 5 = d;
Pin 6 = e;
Pin 7 = f;
Pin 8 = g;
Pin 11 = OE;


/**     Outputs     **/
Pin 16 = !D0;
Pin 17 = !D1;
Pin 18 = !D2;
Pin 19 = !D3;

/** Declarations and Intermediate Variable Definitions **/

zero    = a & b & c & d & e & f & !g ;
one     = !a & b & c & !d & !e & !f & !g ;
two     = a & b & !c & d & e & !f & g ;
three   = a & b & c & d & !e & !f & g ;
four    = !a & b & c & !d & !e & f & g ;
five    = a & !b & c & d & !e & f & !g ;
six     = a & !b & c & d & e & f & g ;
seven   = a & b & c & !d & !e & !f & !g ;
eight   = a & b & c & d & e & f & g ;
nine    = a & b & c & !d & !e & f & g
```

```
            # a & b & c & d & !e & f & g ;   /* two ways */
hexa      = a & b & c & !d & e & f & g ;
hexb      = !a & !b & c & d & e & f & g ;
hexc      = !a & !b & !c & d & e & !f & g
            # a & !b & !c & d & e & f & !g ;  /* two ways */
hexd      = !a & b & c & d & e & !f & g ;
hexe      = a & !b & !c & d & e & f & g ;
hexf      = a & !b & !c & !d & e & f & g ;

/**      Logic Equations       **/

D3 = eight # nine # hexa # hexb # hexc # hexd # hexe # hexf ;
D2 = four # five # six # seven # hexc # hexd # hexe # hexf ;
D1 = two # three # six # seven # hexa # hexb # hexe # hexf ;
D0 = one # three # five # seven # nine # hexb # hexd # hexf ;

END;     /*seg hex*/
```

Figure 23 Seghex.pld.

```
********************************************************************************
                              SEGHEX
********************************************************************************

CUPL(WM)        4.7b Serial# MW-65999997
Device          g16v8as  Library DLIB-h-36-2
Created         Mon Jan 12 15:02:14 1998
Name            SEGHEX
Partno          FFFF
Revision        1.0
Date            12/01/98
Designer        KAREN PARNELL
Company         KANDA
Assembly        NONE
Location        NONE

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

```
D0 =>
    b & c & !d & !e & !f & !g
  # a & b & c & d & !e & g
  # a & !b & c & d & !e & f & !g
  # a & b & c & !d & !e & f & g
  # !a & !b & c & d & e & f & g
  # !a & b & c & d & e & !f & g
  # a & !b & !c & !d & e & f & g

D1 =>
    a & b & !c & d & e & !f & g
  # a & b & c & d & !e & !f & g
  # !b & c & d & e & f & g
  # a & b & c & !d & !e & !f & !g
  # a & b & c & !d & e & f & g
  # a & !b & !c & e & f & g

D2 =>
    !a & b & c & !d & !e & f & g
  # a & !b & c & d & !e & f & !g
  # a & !b & d & e & f & g
  # a & b & c & !d & !e & !f & !g
  # !a & !b & !c & d & e & !f & g
  # a & !b & !c & d & e & f & !g
  # !a & b & c & d & e & !f & g
  # a & !b & !c & !d & e & f & g

D3 =>
    a & !b & !c & !d & e & f & g
  # a & b & c & f & g
  # !a & !b & c & d & e & f & g
  # !a & !b & !c & d & e & !f & g
  # a & !b & !c & d & e & f
  # !a & b & c & d & e & !f & g
END =>

eight =>
    a & b & c & d & e & f & g

five =>
    a & !b & c & d & !e & f & !g
```

```
four =>
   !a & b & c & !d & !e & f & g

hexa =>
   a & b & c & !d & e & f & g

hexb =>
   !a & !b & c & d & e & f & g

hexc =>
   !a & !b & !c & d & e & !f & g
 # a & !b & !c & d & e & f & !g

hexd =>
   !a & b & c & d & e & !f & g

hexe =>
   a & !b & !c & d & e & f & g

hexf =>
   a & !b & !c & !d & e & f & g

nine =>
   a & b & c & !e & f & g

one =>
   !a & b & c & !d & !e & !f & !g

seven =>
   a & b & c & !d & !e & !f & !g

six =>
   a & !b & c & d & e & f & g

three =>
   a & b & c & d & !e & !f & g

two =>
   a & b & !c & d & e & !f & g
zero =>
```

a & b & c & d & e & f & !g

Symbol Table

Pin Pol	Variable Name	Ext	Pin	Type	Pterms Used	Max Pterms	Min Level
	CLK		1	V	-	-	-
	! D0		16	V	7	8	1
	! D1		17	V	6	8	1
	! D2		18	V	8	8	1
	! D3		19	V	6	8	1
	END		0	I	1	-	-
	OE		11	V	-	-	-
	a		2	V	-	-	-
	b		3	V	-	-	-
	c		4	V	-	-	-
	d		5	V	-	-	-
	e		6	V	-	-	-
	eight		0	I	1	-	-
	f		7	V	-	-	-
	five		0	I	1	-	-
	four		0	I	1	-	-
	g		8	V	-	-	-
	hexa		0	I	1	-	-
	hexb		0	I	1	-	-
	hexc		0	I	2	-	-
	hexd		0	I	1	-	-
	hexe		0	I	1	-	-
	hexf		0	I	1	-	-
	nine		0	I	1	-	-
	one		0	I	1	-	-
	seven		0	I	1	-	-
	six		0	I	1	-	-
	three		0	I	1	-	-
	two		0	I	1	-	-
	zero		0	I	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
==================================
                    Fuse Plot
==================================

Syn   02192 - Ac0   02193 x

Pin #19  02048  Pol x  02120  Ac1 x
 00000 x----x---x---x--x---x---x-------
 00032 x---x---x-----------x---x-------
 00064 -x---x--x---x---x---x---x-------
 00096 -x---x---x--x---x----x--x-------
 00128 x----x---x--x---x---x-----------
 00160 -x--x---x---x---x----x--x-------
 00192 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00224 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #18  02049  Pol x  02121  Ac1 x
 00256 -x--x---x----x---x--x---x-------
 00288 x----x--x---x----x--x----x------
 00320 x----x------x---x---x---x-------
 00352 x---x---x----x---x---x---x------
 00384 -x---x---x--x---x----x--x-------
 00416 x----x---x--x---x---x----x------
 00448 -x--x---x---x---x----x--x-------
 00480 x----x---x---x--x---x---x-------
Pin #17  02050  Pol x  02122  Ac1 x
 00512 x---x----x--x---x----x--x-------
 00544 x---x---x---x----x---x--x-------
 00576 -----x--x---x---x---x---x-------
 00608 x---x---x----x---x---x---x------
 00640 x---x---x----x--x---x---x-------
 00672 x----x---x------x---x---x-------
 00704 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00736 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #16  02051  Pol x  02123  Ac1 x
 00768 ----x---x----x---x---x---x------
 00800 x---x---x---x----x------x-------
 00832 x----x--x---x----x--x----x------
 00864 x---x---x----x---x--x---x-------
```

```
 00896 -x---x--x---x---x---x---x-------
 00928 -x--x---x---x---x----x--x-------
 00960 x----x---x---x--x---x---x-------
 00992 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #15  02052  Pol x  02124  Ac1 -
 01024 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01056 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01088 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01120 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01152 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01184 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01216 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01248 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #14  02053  Pol x  02125  Ac1 -
 01280 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01312 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01344 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01376 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01408 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01440 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01472 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01504 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #13  02054  Pol x  02126  Ac1 -
 01536 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01568 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01600 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01632 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01664 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01696 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01728 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01760 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #12  02055  Pol x  02127  Ac1 -
 01792 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01824 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01856 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01888 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01920 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01952 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01984 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 02016 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

LEGEND X : fuse not blown
- : fuse blown

==========
Chip Diagram
==========

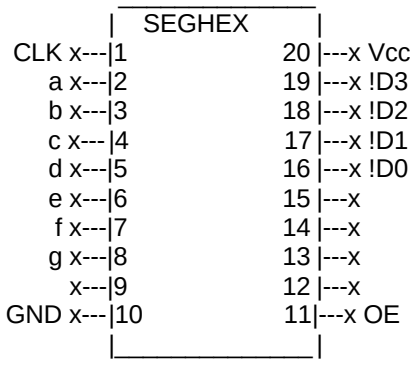


Figure 24: Seghex.doc

8.3 APPLICATION 2: VENDING MACHINE (SEQUENTIAL)

This example is an example of a state machine. You can implement a state machine in programmable logic containing registers if:

a. There are enough register bits to represent all possible states (e.g. with 4 registers you could have upto 16 states)

b. There are enough inputs and logic gates to implement the transition rules.

The design we are about to implement is a vending machine, and it is supposed to provide a bottle of coke when 20pence (or more) has been deposited in the coin slot. There is some sort of coin interface that gobbles up and recognises the money and send our PAL a 2-bit input (C1, C0), valid for one clock edge, indicating the coin just deposited (01 = 5pence, 10 = 10pence, 11 = 20pence, 00 = no coin or duff coin).

The state machines job is to add up the total deposited and generate an output called ‘bottle’ when there’s enough money. Figure 25 shows the specification, in CUPL’s state machine syntax. As before we begin by defining input and output pins. Note that we have a ‘reset’ input so that you can initialise to the state S0 (no money). Next we define the states, then the rules for moving between them. If any output, either registered or combinational, need to be generated during states or transitions between states, they are specified at the same time. In this example, for instance, the output bottle has been specified as a separate output register, so that no output state decoding is needed. In fact, this is the only output needed, and the state-machine bits Q0-2 could be implemented in internal registers that don’t generate outputs directly; some programmable logic devices have such ‘buried’ registers, in addition to the usual output registers.

Note that you have to specify explicitly the transition from state to itself, as we have done here for the input ‘nocoin’. An unspecified condition implicitly resets the state to all zeros. That is because these conditionals are compiled into combinational logic to assert the D inputs of the registers, and thus if the condition is not met, the corresponding D input is not asserted.

Figure 25 shows the output from CUPL. There is nothing obvious or simple about the logic, because both the machine state (S0-S5) and the input variable (C0-1) are specified as binary numbers, whereas the logic operates on individual bits. Thus, the resulting logic does not bear much relation to the

original state description, figure 26. In fact the particular choice of states (ascending binary 0-5) could have been chosen differently, completely changing the resulting logic. In this case, this example fits easily into a 16V8. Note that the ‘reset’ input acts by an over-riding diassertion of all D inputs, which we forced by our definition of the intermediate variables nocoin, 5p etc.

```
NAME            VENDING MACHINE;
PARTNo          FFFF;
DATE            12/01/98;
REVISION        1.0;
DESIGNER        KAREN PARNELL;
COMPANY         KANDA;
ASSEMBLY        NONE;
LOCATION        NONE;
DEVICE          G16V8A;

/**  Inputs  **/

Pin 1 = CLK;   /* clock -- positive edge */
Pin 3 = c0;    /* coin type -- low bit */
Pin 4 = c1;    /* coin type -- high bit */
Pin 6 = reset; /* reset input          */
Pin 11 = OE;

/**      Outputs       **/
Pin 15 = !bottle;   /* bottle release command  */
Pin 16 = !Q2;       /* bit 2                   */
Pin 17 = !Q1;       /* bit 1                   */
Pin 18 = !Q0;       /* bit 0 of state variable */

/* Define machine states with symbolic names; "enough = 20pence or more*/

$define S0     'b'000
$define S5     'b'001
$define S10    'b'010
$define S15    'b'011
$define S20    'b'100
$define ENOUGH 'b'101

/* define intermediate variable */
nocoin    = !c0 & !c1 & !reset ;
```

```
5pence   = c0 & !c1 & !reset ;
10pence = !c0 & c1 & !reset ;
20pence = c0 & c1 & !reset ;

/* define state bit variable field */

field statebit = [Q2..0] ;

/* transition rules for vending machine */
sequence statebit {
   present  S0   if nocoin    next S0;
            if 5pence    next S5;
            if 10pence   next S10;
            if 20pence   next ENOUGH   out bottle;

   present  S5   if nocoin    next S5;
            if    5pence    next S10;
            if   10pence   next S15;
            if   20pence   next ENOUGH   out bottle;

   present  S10  if nocoin    next S10;
            if    5pence    next S15;
            if   10pence   next S20;
            if   20pence   next ENOUGH   out bottle;

   present  S15  if nocoin    next S15;
            if   5pence    next S20;
            if  10pence   next ENOUGH   out bottle;
            if  20pence   next ENOUGH   out bottle;

   present  S20  if nocoin    next S20;
            if   5pence    next ENOUGH   out bottle;
            if  10pence   next ENOUGH   out bottle;
            if  20pence   next ENOUGH   out bottle;

   present ENOUGH            next S0;   }
```

Figure 25 Vend1.pld

```
*******************************************************************************
                          VENDING
```

```
********************************************************************************

CUPL(WM)       4.7b Serial# MW-65999997
Device         g16v8ms  Library DLIB-h-36-11
Created        Mon Jan 12 16:20:22 1998
Name           VENDING MACHINE
Partno         FFFF
Revision       1.0
Date           12/01/98
Designer       KAREN PARNELL
Company        KANDA
Assembly       NONE
Location       NONE

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

10pence =>
   !c0 & c1 & !reset

20pence =>
   c0 & c1 & !reset

5pence =>
   c0 & !c1 & !reset

Q0.d  =>
   !Q0 & !Q1 & Q2 & !c0 & c1 & !reset
  # !Q0 & !Q1 & Q2 & c0 & !reset
  # Q0 & !Q2 & !c0 & !reset
  # Q0 & !Q2 & c0 & c1 & !reset
  # !Q0 & !Q2 & c0 & !reset

Q1.d  =>
   Q0 & Q1 & !Q2 & !c0 & !c1 & !reset
  # !Q1 & !Q2 & !c0 & c1 & !reset
  # Q0 & !Q1 & !Q2 & c0 & !c1 & !reset
  # !Q0 & Q1 & !Q2 & !c1 & !reset

Q2.d  =>
```

```
   !Q0 & !Q1 & Q2 & !reset
  # !Q1 & !Q2 & c0 & c1 & !reset
  # !Q0 & Q1 & !Q2 & c1 & !reset
  # Q0 & Q1 & !Q2 & c0 & !reset
  # Q0 & Q1 & !Q2 & !c0 & c1 & !reset

bottle.d  =>
   !Q0 & !Q1 & Q2 & !c0 & c1 & !reset
  # !Q2 & c0 & c1 & !reset
  # Q0 & Q1 & !Q2 & !c0 & c1 & !reset
  # !Q0 & !Q1 & Q2 & c0 & !reset

nocoin =>
   !c0 & !c1 & !reset

statebit =>
   Q2 , Q1 , Q0

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

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

        10pence               0     I     1       -       -
        20pence               0     I     1       -       -
        5pence                0     I     1       -       -
        CLK                   1     V     -       -       -
        OE                    11    V     -       -       -
        !  Q0                 18    V     -       -       -
        !  Q0         d       18    X     5       8       1
        !  Q1                 17    V     -       -       -
        !  Q1         d       17    X     4       8       1
        !  Q2                 16    V     -       -       -
        !  Q2         d       16    X     5       8       1
        !  bottle             15    V     -       -       -
        !  bottle     d       15    X     4       8       1
        c0                    3     V     -       -       -
        c1                    4     V     -       -       -
```

nocoin	0	I	1	-	-
reset	6	V	-	-	-
statebit	0	F	-	-	-

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

==========

Fuse Plot

==========

Syn 02192 x Ac0 02193 -

Pin #19 02048 Pol x 02120 Ac1 -
00000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00032 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00064 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00096 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00128 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00160 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00192 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00224 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #18 02049 Pol x 02121 Ac1 x
00256 -----xx-x-x----x-x--------------
00288 ----x-x---x----x-x--------------
00320 -----x-x------x--x--------------
00352 ----x--xx-----x--x--------------
00384 ----x-x-------x--x--------------
00416 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00448 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00480 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #17 02050 Pol x 02122 Ac1 x
00512 -----x-x-x-x--x--x--------------
00544 -----x--x-x---x--x--------------
00576 ----x--x-xx---x--x--------------
00608 ------x--x-x--x--x--------------
00640 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00672 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```
 00704 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00736 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #16  02051  Pol x  02123  Ac1 x
 00768 ------x---x----x-x-------------
 00800 ----x---x-x---x--x-------------
 00832 ------x-x--x--x--x-------------
 00864 ----x--x---x--x--x-------------
 00896 -----x-xx--x--x--x-------------
 00928 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00960 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 00992 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #15  02052  Pol x  02124  Ac1 x
 01024 -----xx-x-x----x-x-------------
 01056 ----x---x-----x--x-------------
 01088 -----x-xx--x--x--x-------------
 01120 ----x-x---x----x-x-------------
 01152 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01184 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01216 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01248 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #14  02053  Pol x  02125  Ac1 -
 01280 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01312 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01344 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01376 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01408 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01440 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01472 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01504 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #13  02054  Pol x  02126  Ac1 -
 01536 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01568 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01600 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01632 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01664 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01696 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01728 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 01760 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #12  02055  Pol x  02127  Ac1 -

LEGEND    X : fuse not blown
```

- : fuse blown

```
==================================
             Chip Diagram
==================================


                 _____________
                |  VENDING    |
          CLK x---|1          20|---x Vcc
              x---|2          19|---x
           c0 x---|3          18|---x !Q0
           c1 x---|4          17|---x !Q1
              x---|5          16|---x !Q2
        reset x---|6          15|---x !bottle
              x---|7          14|---x
              x---|8          13|---x
              x---|9          12|---x
          GND x---|10         11|---x OE
                  |_____________ |
```

Figure 26 Vend1.doc

9

COMPONENT REFERENCE DATA

9.1 **INTRODUCTION**

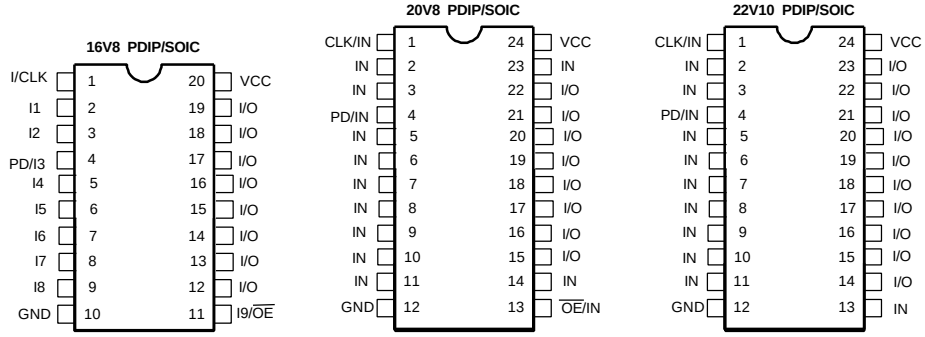
PLDs can be split into two main families:

1. Simple Programmable Logic Devices (SPLDs) and
2. Complex Programmable Logic Devices (CPLDs).

In this book we have used Atmel Flash based SPLDs. There are three main industry standard SPLDs which are usually defined thus:

PLD	PINS (PDIP/PLCC)	Vcc	Technology	TPD nS	Inputs	Outputs
16V8	20/2O	5 OR 3.3	EE CMOS	5 - 25	16	8
20V8	24/28	5 OR 3.3	EE CMOS	5 - 25	20	8
22V10	24/28	5 OR 3.3	EE CMOS	5 - 25	22	10

If this book was obtained as part of the Kanda PLD Starter Kit you will already be in possession of one of the following devices, which have the following pin outs:



On the following pages Atmel have kindly given us permission to reproduce three of their industry standard SPLD data sheets to help with your PLD designs. These data sheets should prove in-valuable to you. If you should require any further information on any of the other PLD products from Atmel or any of the other products such as non-volatile memories, flash based microcontrollers or FPGAs please contact us (contact information can be found at the back of the book).

Kanda also produce and distribute development tools for microcontrollers including device programmers, In-circuit emulators and C-compilers. We also design and develop training tools and books as well as PC based Logic Analysers.

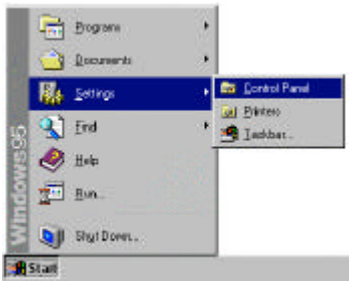
Appendix A

Software Installation Instructions

Windows 98

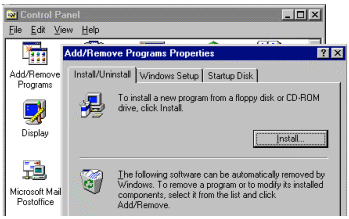
To install the software please insert the CDROM in your computer and perform the following steps:

- Click on your Start button.
- Select Settings.
- Select Control Panel
- Choose Add/Remove Programs.
- Click the Install button.
- Follow On-Screen prompts.



The software will then be installed onto your computer and an Icon will be added to your start menu.

Windows 200/XP



Run the EXE file on the CD to install the software and printer port drivers. You will need Admin privileges.

The software and book will then be installed onto your computer and an Icon and Program Group will be added to Program Manager.

Technical Support

In the unlikely event that you have any problems installing the software or suspect that you have faulty media please contact our technical support department for advice.

Telephone Number: +44 (0) 8707 446 807
Fax Number: +44 (0) 8707 446 807

Appendix B

Using the Kanda PLD Starter Kit

The Kanda PLD Starter Kit is a complete working and training environment. The kit includes a device programmer, applications module (to test your designs), programming software and design software integrating CUPL and the template generator onto a single windows platform.

You will have now installed the software and double-clicked on the icon to enter the software.

On first entering the program there are two menus available these are **File** and **Action.**

The File menu has 4 options:

Option	Meaning
New	Starts a new program from scratch.
New from Template	Starts a new program using the template generator (advisable).
Open	Opens a file saved on disk.
Exit	Exits program.

The Action menu has 3 options:

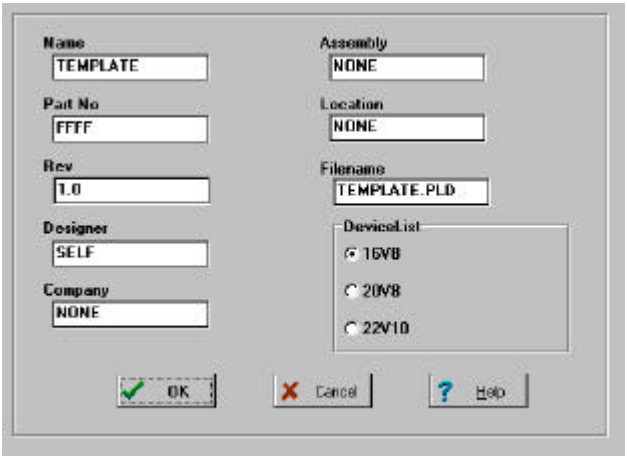
Option	Meaning
Read Device	Reads the selected device.
Erase Device	Erases the selected device
Set Printer Port	Enables the user to select which printer port (1-3) the PLD board will be attached to.

These menu choices will be described in the following section.

Appendix B

Using the template generator

On selecting “New from template” the following window is displayed (figure B1). On this screen you can enter a name for your program, Part No, Revision, Designer, Company, Assembly, Location and the Filename of your ‘.PLD’ file. After you have completed the selection you need to select a device that you want to complete your design with (select a device from the pull-down menu).



FigureB1 Template generator

After completion of this page and pressing OK the next screen that is shown is the (device) Pin Assignment, figure B2.

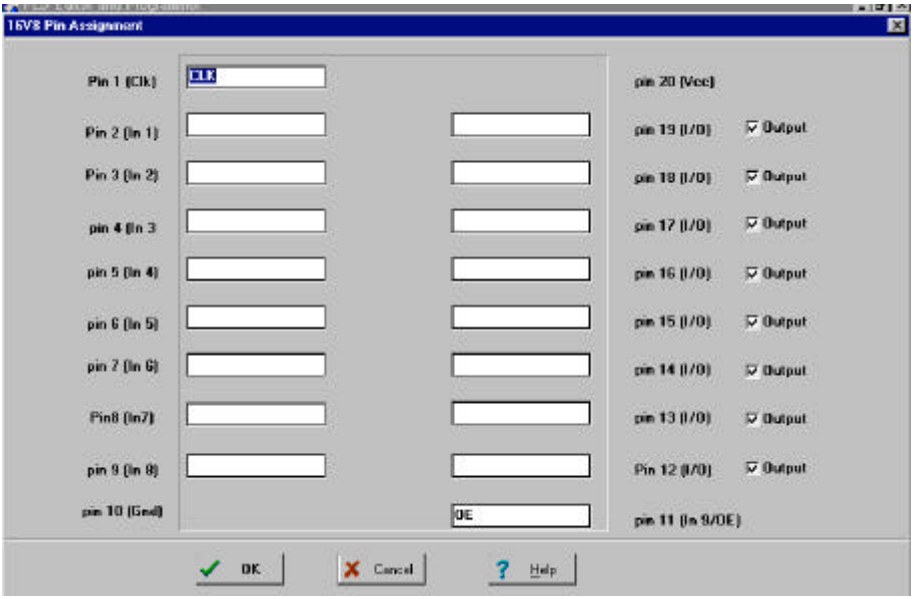


Figure B2 Pin Assignment Screen

On this screen you can select which Input and output pins are called. It is suggested that the I/O pins are called names that will relate to your program i.e. Pin2(In 1) = Input 1. When this screen has been completed and OK has been pressed the header and pin assignments will be automatically written in the correct format. The screen now showing is the main menu. The cursor is placed in the ‘logic equations’ area ready for the rest of the logic design description to be entered. The screen showing is shown overleaf in figure B3.

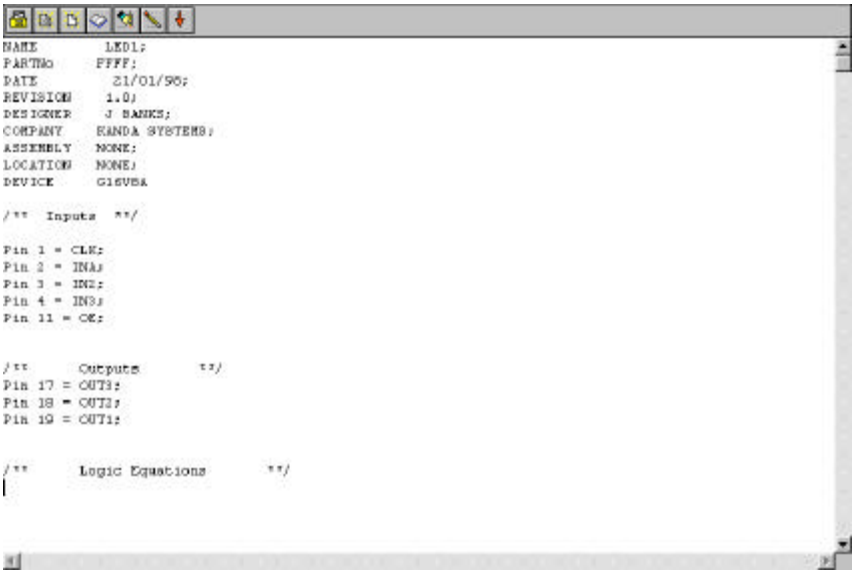


Figure B3 View of main menu after template generator used.

When writing the logic description ensure that the same case is entered as that of the template generator because the compiler is ‘case sensitive’ and also ensure that a semicolon (;) is used as the last character in the code
e.g. OUT1 = INA # INB;
When the logic description is completed the next step is to ‘Compile’ the code into its output format.

Select Compile - the next screen shows the compile options and the minimisation options that can be selected. The compile option screen is shown overleaf in figure B4.

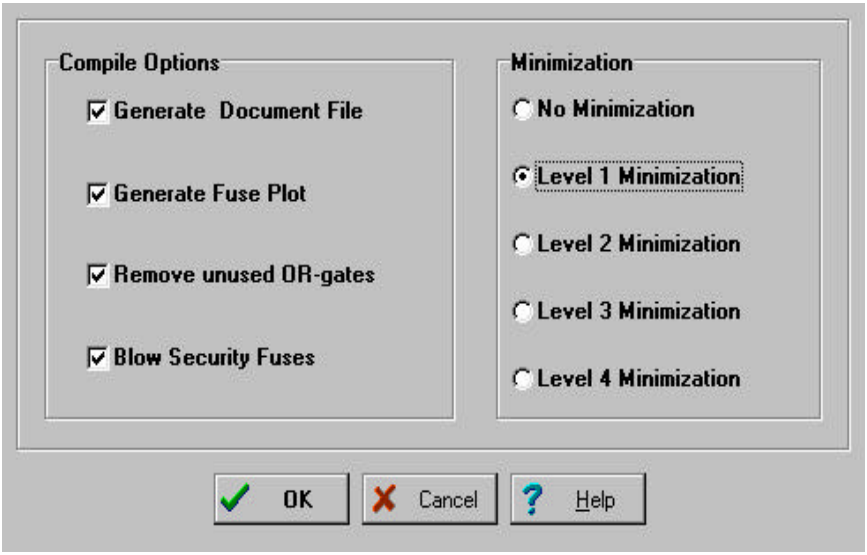


Figure B4 Compile and minimisation option screen

Select the required output files and minimisation required from the list. (please refer to the CUPL manual supplied on the disk included on the Starter Kit for details on the minimisation options). It is advisable to always select at least Level 1 Minimization for best design results.

Note: If there are any problems when compiling it is suggested that you should close down all other applications and try compiling the data again.

If there are any syntax or design errors in the description file the compiler will report information on the errors.

When the description file compiles without errors, the next stage is to program the device.

Programming the Device

Select Program from the menu, the JEDEC file is displayed and the screen below becomes available.

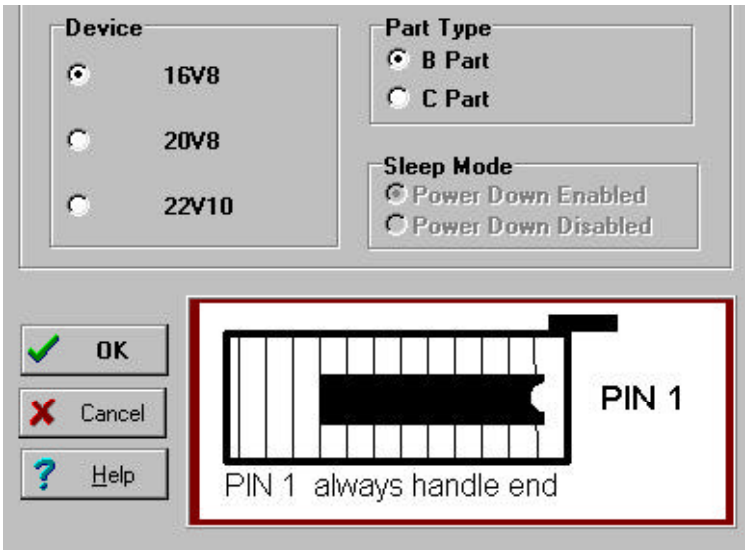


Figure B5 PLD Programming Screen

Put the device into the ZIF socket in the correct orientation.
NOTE : Pin 1 is always at the handle end.

Select the device that it is to be programmed, select which part type is required (part type is the letter after the number e.g. 16V8C). If the part type is ‘C’ then Sleep mode becomes available giving you access to:

Power down enabled or power down disabled, select which one you require then press OK to start programming the device.

When the device has been programmed a box will appear stating that "Programming is complete".

Reading the Device

Select Read and the device selection page is available. Select a device to read and press OK the program will then read the selected device into a file called "result.pld".

Note: In the file result.pld 0 = data & 1 = No data

Erasing the device

Select 'Erase' and the device selection page is available. Select a device to erase and press 'OK' the program will then erase the selected device and inform the user that it has been completed.

PLD Starter Kit Technical Information

PC Requirements
Windows 3.1 or Windows 95, 4MB RAM, 2MB Hard Disk Space, Parallel Port.

PC Connection
Connection to PC Parallel Port, Suitable Cable Supplied.

Power Supply Requirements
>9V, 500ma, 3.5mm Barrel Connector. Centre Positive.

Device Programmer
The device programmer has support for the following devices:

- ATF16V8B & C
- ATF20V8B & C
- ATF22V10

Appendix B

Package Contents
1 x ATF16V8B device (shipped on the application board to prevent damage in transit)
1 x Device Programmer
1 x Application / Training Board
1 x Parallel Connection Board
1 x Get going with PLD's Book
AtmelCUPL Software
Kanda PLD Programming Software & Kanda AtmelCUPL Interface.

Kanda PLD Programming Software
Windows based programming software.

Programming functions available:

- Program
- Erase
- Read
- Verify

Kanda AtmelCUPL Interface
Provides an interface to AtmelCUPL. Includes template generation which allows quick set up of the following:

- Name / Part-number, Revision, Designer, Company Information.
- Pin Name Assignments
- Pin I/O Configuration

Also included is an integrated text editor, which allows most common windows editing functions (Cut\Copy\Paste etc.).

ATF16V8 Included Device

- Industry standard architecture.
- High speed electrically erasable programmable logic device.
- Advanced Flash Technology

Appendix B

For more information on the ATF16V8B. Please see Atmels Web Site (http://www.atmel.com) or Fax on Demand Fax: +1 (408) 441-0732.

Application / Training Board
The Application / Training board is suitable for use with the supplied ATF16V8B. Power Supply required is the same as for the programmer.

Training board contains the following:

- 7 Segment Display
- 10 Switches
- 7 LED's
- ATF16V8 device already on board (the device is placed here to prevent damage during transit. Remove to programme the device)

If you have any further technical questions please contact our technical support department on:

Tel: +44 8707 446 807
Fax: +44 8707 446 807
E-Mail: support@kanda.com

About Kanda

Kanda are a leading supplier of semiconductor tools and development products.

Kanda can help you realise your digital designs by providing everything from software to hardware solutions. We can offer professional support in the following areas:

- *Development Tools*
  - Device **Starter kits, In-Circuit Emulators** and **Programmers**
- *Training*
  - Logic trainer, ADC Trainer, Switched mode PSU designer, technical books and training seminars & courses.
- *Support Services*
  - High volume device programming & design Consultancy.

Here is a selection of products from Kanda that can help you take you designs from the 'drawing board' to the circuit board.

PLD Starter Kit

Programmable Logic Devices (PLDs) provide the digital design engineer with excellent design flexibility and enhanced levels of security for any design. Costing less, in many cases than the equivalent TTL they now come in enhanced flash versions that may be reprogrammed at will. This system takes the engineer from basic logic through the design process and up to the point where you can produce your own designs. The system includes everything you need to get started. Included is a full tutorial book, an applications module, an enhanced copy of the internationally acclaimed CUPL design language, a reprogrammable device (that can emulate a range of industry standard PALs) and a device programmer. All of the design and programming functions are accessed through the integrated Windows desktop so producing a working design is a quick and easy task.

PLD Device Adapters

To increase the types of devices that can be programmed using the Kanda PLD starter kit and PLD Programmer we can provide the following adapters:

- 28 Pin PLCC Adapter
- 20 Pin PLCC Adapter
- 20 Pin SOIC Adapter

All of the adapters fit into a standard ZIF socket.

Serial EEPROM Starter Kit

This starter kit includes everything you need to programme the following Serial EEPROMs:

- Atmel AT25XXX Series (SPI)
- Atmel AT24CXX Series (2-wire)
- Atmel AT93CXX Series (3-wire)
- Atmel AT56CXX Series (4-wire)

The kit comprises:

- Kanda Windows programming software with FREE upgrades
- Programming Module, battery operated for portability
- Programming cable and dongle
- Full Technical Support
- In-system programming via an IDC connector
- FREE software updates via our website: www.kanda.com

AVR Trainer

The AVR Trainer is a complete low cost development system for the Atmel AVR range of **FLASH** 8-bit in-system programmable RISC microprocessors & includes a **JTAG IN Circuit Emulator** to help with design debugging. Everything is included to develop AVR designs in this **fast track** learning environment.

Features include:

- Board with swiches, LEDs, A2D, UART and port headers
- WinAVR C Compiler
- AVRStudio Integrated Windows Development Software
- **Parallel port programmer**
- **In Circuit Emulator using serial port**
- Supports ALL AVR devices via ISP
- **FREE** software upgrades via our Website: www.kanda.com/support
- **Get Going with AVR Book**
- AVR device
- Supports an external clock

The AVR Trainer is an ideal design platform for taking your AVR design from conception to realisation. It incorporates **Switches**, **Lights,** and **RS232** provision, with expansion sockets for external RAM, LCD and keypad.

The development board has a full range of ports for development purposes. The Integrated Development Environment includes **AVR Builder** for fast track development, it allows automatic code generation to set-up on-chip peripherals.

Other 'Get Going With…' books in the Series

Get Going with… the AVR

The Get Going with…the AVR book was written by Peter J. Sharpe who is a specialist in microcontroller technology.

The AVR is the latest 8-bit FLASH RISC microcontroller from Atmel. The AVR is a high speed, low cost microcontroller that utilises a powerful instruction set (most single cycle) and has on-chip flash program memory, EEPROM and SRAM.

This book takes you on a journey through microelectronic systems (including an introduction to logic), introduction to the AVR range, how to plan designs and includes a ream of useful design examples to help you Get Going With… the AVR!

Get Going with… FPGAs

This is the second in the series of ‘fast track’ books to help you learn about and design with new technology. This book takes you from basic logic principles to in-depth design examples based on the latest Atmel FPGA offering - the 40K series.

The book package includes the Figaro design software package and a limited version of a popular schematic design package.

Fax Back Form

Name

Company

Address

Country Postcode/ ZIP

I would like more information on:

Programming products ☐ Network ☐

In-Circuit Emulators ☐ All products ☐

Fax Number: +44 (0) 1970 621040

Telephone Number: +44 (0) 1970 621030

Website: www.kanda.com

Appendix C CUPL - ERROR MESSAGES

CUPL error messages are intended to be self-explanatory. This appendix provides additional information describing them.

Some of the CUPL programs, such as CUPL are composed of individual modules. Error messages are numbered and listed according to the program and module in which they occur. The suffix to the error message number identifies the program and module.

Table A-1. Error Message Module Suffixes

Module	Suffix
CUPL processor	ck
CUPLX preprocessor	cx
CUPLA source file parser	ca
CUPLB equation fitter	cb
CUPLM minimizer	cm
CUPLC fusemap generator	cc

This appendix lists the error messages by modules in the same order as they appear in Table A-1 above. The error messages within each module are listed in numerical order.

CUPL provides three levels of error messages: warnings, errors, and fatals.

WARNINGS - do not prevent CUPL from continuing, but indicate a problem that should be corrected.

ERRORS - allow CUPL to continue but must be corrected before future compiles.

FATALS - prevent CUPL from continuing and must be corrected.

==========

Note Error messages with indexes greater than 1000 are program errors. This section does not individually list program errors. Possible causes for program errors are bad data in a source file caused by disk errors or word processors in document mode; or previous errors continuing to propagate unexpected circumstances. If the cause of a program error cannot be determined, gather as much information as possible on the conditions in effect when the error occurred, then call CUPL support.

Error messages report the line number on which the error was detected; however, the cause of the error may be on a previous line. If the message doesn't seem to apply to the reported line, look at preceding lines for the source of the error.

==========

CUPL Module - Error Messages

0001ck could not open: "filename"
Fatal. CUPL cannot continue because of the failure to open the indicated File. Be sure the file exists if it is an input.

0002ck could not execute program: "program name"
Fatal. CUPL is unable to perform the next step in the compilation. Be sure that all of the CUPL program files exist on the same directory or disk.

0003ck could not find PATH in ENVIRONMENT
Fatal. The PATH assignment has not been made in the ENVIRONMENT.

0004ck could not find LIBCUPL in ENVIRONMENT
Fatal. The LIBCUPL assignment has not been made in the ENVIRONMENT.

0005ck could not find program: "program name"
Fatal. CUPL is unable to locate the CUPL programs using the PATH in the ENVIRONMENT.

0006ck insufficient memory to execute program: "filename"
Fatal. Not enough program storage available to load and execute the program.

0007ck invalid flag: "option flag"
Fatal. The option flag specified is not one of the allowable compilation flags.

0008ck out of memory: "condition"
Fatal. CUPL has used all available RAM memory which has been allocated by the operating system. Check for the existence of print spoolers, RAM disks, or other memory-resident programs which may decrease the amount of memory available to the CUPL application.

10xxck program error: "specifics"
Fatal. An operating system interface problem is suspected.

CUPLX Module - Error Messages

0001cx could not open: "filename"
Fatal. CUPLX cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002cx could not execute program: "program name"
Fatal. CUPLX is unable to perform the next step in the compilation. Be sure that all of the CUPL program files exist on the same directory or disk.

0003cx no label given for command
Error. One of the pre-processor commands, $DEFINE, $UNDEF, $IFDEF, or $IFNDEF, was used without a succeeding label.

0004cx already defined: "label"
Error. The label was previously defined using $DEFINE. To redefine the label, first use $UNDEF to undefine the label, and then use $DEFINE to redefine it.

0005cx string error
Fatal. All pre-processor label string space has been used.

0006cx $else without $ifdef
Error. An $ELSE pre-processor command was used without being preceded by an $IFDEF or $IFNDEF command.

0007cx $endif without $ifdef
Error. An $ENDIF pre-processor command was used without being preceded by an $IFDEF or $IFNDEF command.

0008cx $ifdef nesting too deep
Error. The level of $IFDEF nesting exceeded twelve.

0009cx missing $endif
Error. An $IFDEF pre-processor command was used without being succeeded by an $ENDIF command.

0010cx invalid pre-processor command: "$command"
Error. The pre-processor command is unknown.

0011cx disk write error: "filename"
Fatal. CUPLX encountered an I/O error trying to write the indicated file. This error usually occurs when there is insufficient disk space.

0012cx out of memory: "condition"
Fatal. CUPLX has used all the available RAM memory allocated by the operating system.

0013cx illegal character: "hex value"

Error. CUPLX has encountered an illegal ASCII value in the source file. Make sure the file was created in nondocument mode on the word processor.

0014cx unexpected symbol:"symbol"

Fatal. CUPLX encountered a symbol that it was not expecting. This occurs when certain symbols are expected in a particular order and are either incorrect, misplaced or misspelled.

0015cx Repeat nesting too deep

Fatal. The level of Repeat nesting exceeded two.

0016cx duplicate Macro function name:"function"

Error. The Macro function name has already been previously defined. A duplicate Macro name will cause confusion when they are called.

0017cx missing Macro name

Fatal. A Macro was defined without a name. This macro will never be accessed.

0018cx incorrect number of parameters

Fatal. The number of parameters defined in the Macro function did not equal the number of parameters in the macro call. All parameters defined in the Macro function must be defined in the Macro call.

0019cx out of range

Fatal. The index number exceeded 1023.
Valid index numbers are 0 - 1023.

0020cx internal stack overflow

Fatal. A mathematical expression was too complex for CUPLX to handle. The expression can be reduced by eliminating as many parenthetical expressions as possible. Expressions are evaluated from left to right using standard precedence. The user should take advantage of this.

0021cx expression contains undefined symbol: "symbol"

Fatal. A symbol appearing in the expression has not been defined in the source file or predefined by CUPL.

0022cx invalid library access key

Fatal. The version of CUPLX is not compatible with the version of the device library file. This occurs when either CUPLX or the device library, but not both, has been updated.

0023cx invalid library interface

Fatal. Either the device library was not created using the CUPL library manager, CBLD or CUPLX and the device library are not compatible.

0024cx bad library file: "library"

Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0025cx unexpected end-of-file

Fatal. CUPLX has unexpectedly reached the end-of-file.

0026cx reached end-of-file before ending comment

Fatal. CUPLX detected that a comment was not terminated before reaching the end-of-file. The beginning of the comment can be found by searching for the last occurrence of /* in the PLD file.

0027cx invalid syntax for pre-processor command: "$command"

Fatal. One of the pre-processor commands, $REPEAT or $MACRO, has been used improperly. The command syntax contains unexpected symbols.

10xxcx program error: "specifics"

Fatal. An operating system interface problem is suspected. Contact Logical Devices customer support.

CUPLA Module - Error Messages

0001ca could not open: "filename"
Fatal. CUPLA cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002ca invalid number: "number"
Error. Either the number is used improperly, or a previous syntax error caused the number to be used improperly.

0003ca invalid library access key
Fatal. The version of CUPLA is not compatible with the version of the device library file. This occurs when either CUPLA or the device library, but not both, has been updated.

0004ca invalid library interface
Fatal. Either the device library was not created using the CUPL library manager, or CUPLA and the device library are not compatible.

0005ca bad library file: "library"
Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0006ca device not in library: "device"
Fatal. Either the specified target device does not exist or an entry has not been made in the device library for the device.

0007ca invalid syntax: "symbol"
Error. Either the symbol is used improperly, or a previous syntax error caused the symbol to be used improperly.

0008ca too many errors
Fatal. CUPLA has encountered more than 30 errors.

0009ca missing symbol: "symbol"
Error. The missing symbol is required to make the specified statement valid.

0010ca vector too wide
Fatal. A variable list has more than 50 members.

0011ca expression already assigned to: "variable"
Error. The variable (either an intermediate or output variable) was previously assigned an expression. Use APPEND to make multiple expression assignments for the same variable.

0012ca vector size mismatch

Error. The number of members in the variable list on the left side of the equation does not match the number of variables on the right side.

0013ca undefined function: "function"

Error. The variable name used as a function reference has no corresponding function definition. Functions must be defined before they can be referenced.

0014ca variable already declared: "variable"

Error. The variable which was previously assigned an expression cannot be reassigned.

0015ca out of memory: "condition"

Fatal. CUPLA has used all available RAM memory which has been allocated by the operating system. Decrease the number of intermediate variables, fields, or numbers in order to reduce the size of the symbol t able.

==========

Note : This error is not a result of insufficient product terms in the device to implement a particular expression.

==========

0016ca invalid number of function arguments: "number"

Error. The user has attempted to pass an incorrect number of arguments to the user-defined function. The number of arguments for the function reference does not match the number in the function definition.

0017ca disk write error: "filename"

Fatal. CUPLA encountered an I/O error trying to write the indicated file. This error usually occurs when there is insufficient disk space.

0018ca intermediate var not assigned an expression: "variable"

Error. The intermediate variable was used as an input in an expression without having been assigned an expression. This error often occurs when a pin or intermediate variable in a logic expression is misspelled.

0019ca indexed and non-indexed vars in range or match expression

Warning. A list (or field variable) in a range or match expression contains both indexed (variable names ending in a number) and nonindexed variables. This type of operation cannot produce the expected results because of inability to hold relative bit positions in the field. It is recommended to use all non-indexed variables in a field for portability to future versions of CUPL.

0020ca index too large for range or match operation
Error. The index of a variable in a list or field exceeds the range or match values.

0021ca header item already declared
Error. One of the header statements was duplicated.

0022ca missing header item(s)
Warning. At least one of the header statements is missing.

0023ca invalid range arguments: always true (in range)
Error. A range has been specified which will always be true and is therefore not an actual range. CUPLA attempts to minimise range functions and does not allow a NULL range such as this. This happens with ranges such as [0000..FFFF] for a 16-bit address. This error can also be given if non-indexed list variables are used in a range expression.

0024ca range or match number larger than variable list
Warning. The range or match number exceeds the width of the bit field it is being applied to. Values exceeding the width of the bit field will be ignored.

0025ca range minimisation error
Error. The range reduces to always false, that is, none of the bits in the range are active.

0026ca invalid table statement
Error. Input numbers cannot be mapped into more than one output number.

0027ca invalid present state number
Error. The present state number specified is not valid. This error can occur whenever the present state has not been properly defined as a number using the $DEFINE command.

0028ca invalid next state number
Error. The next state number specified is not valid. This error can occur whenever the next state has not been properly defined as a number using the $DEFINE command.

0029ca invalid flip-flop type for sequence statement: "type"
Error. The flip-flop type for this device cannot be used for building the requested sequential state machine.

0030ca intermediate dependent on itself: "variable"
Error. The intermediate variable was used in the expression defining the same intermediate variable. This error often occurs when an intermediate variable is misspelled or an output pin expression is being defined using feedback without declaring the output variable as a pin.

0031ca invalid minimisation level: "level"
Error. The minimisation level specified is invalid.

0032ca invalid next state: "hex number"
Error. The next state value is invalid. This error can occur whenever the next state has not been properly defined as a number using the $DEFINE command or has not been identified as a present state using the present command.

0033ca multiple asynchronous defaults for state: "hex number"
Error. By definition, only one asynchronous default expression can be assigned for any one state. The resulting expression is the complement of all previous conditional (if) asynchronous expressions.

0034ca multiple synchronous defaults for state: "hex number"
Error. By definition, only one synchronous default expression can be assigned for any one state. The resulting expression is the complement of all previous conditional (if) synchronous expressions.

0035ca multiple unconditional statements for state: "hex number"
Error. By definition, only one unconditional synchronous statement can be given for any one state.

0036ca device does not support synchronous state machines
Fatal. The device specified for compilation cannot be used with the sequence statement since it does not support registered operations.

0037ca duplicate present state: "hex number"
Error. The present state number was identified in more than one PRESENT command. This can occur when symbolic state names are used to refer to states, but the $DEFINE command, used to define states, assigned the same number to more than one symbolic name.

0038ca target device not specified
Fatal. The user did not specify a target device on the command line and the source file did not contain a DEVICE assignment in the header information.

0039ca line exceeds maximum length

Error. The statement is greater than 256 characters long. Break the line up into shorter statements.

0040ca invalid or duplicate header name: "name"

Fatal. The NAME field in the header information must not be NULL. When more than one device is being defined in a logic description file, the NAME field in the header information must be unique.

0041ca don't care(s) not allowed for decimal number, treated as 0

Warning. "Don't-care" values, "X", are valid only for binary, octal, and hexadecimal numbers.

0042ca range or match list completely don't cared, decoded as 0

Warning. The variable list in a range or match operation has been completely "don't-cared," leaving an empty variable list. The empty variable list will be decoded into a 0.

0043ca invalid GROUP name: "variable name"

Fatal. The GROUP name must contain the keyword BLOCK_ followed by "variable name". Ex. GROUP-BLOCK A=[X,Y]; where A is the variable name.

0044ca unexpected end-of-file

Fatal. CUPLA has unexpectedly reached the end-of-file.

0045ca reached end-of-file before ending comment

Fatal. CUPLA detected that a comment was not terminated before reaching the end-f-file. The beginning of the comment can be found by searching for the last occurrence of /* in the PLD file.

10xxca program error: "specifics"

Fatal. An operating system interface problem is suspected.

CUPLB Module - Error Messages

0001cb could not open: "filename"
Fatal. CUPLB cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002cb could not execute program: "program name"
Fatal. CUPLB is unable to perform the next step in the compilation. Be sure that all of the CUPL program files exist on the same directory or disk.

0003cb invalid file:"filename"
Warning. The file was not created by the current version of CUPL.

0004cb missing or mismatched parentheses:
Error. The number of open parentheses [(] and close parentheses [)] in the specified statement does not match.

0005cb invalid library access key
Fatal. The version of CUPLB is not compatible with the version of the device library file. This occurs when either CUPLB or the device library, but not both, has been updated.

0006cb invalid library interface
Fatal. Either the device library was not created using the CUPL library manager, CBLD, or CUPLB and the device library are not compatible.

0007cb bad library file: "library"
Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0008cb device not in library: "device"
Fatal. Either the specified target device does not exist or an entry has not been made in the device library for the device.

0009cb pin/node "number" redeclared: "variable"
Error. The same pin number or variable name was used more than once in a pin declaration statement.

0010cb pin/node "number" invalid output: "variable"
Error. The variable being assigned an output expression was previously declared for an input-only pin.

0011cb unknown extension: "extension"
Error. The extension is unknown or invalid for the particular device. Check to make sure the device has the capability required.

0012cb pin/node "number" invalid usage: "variable"
Fatal. The pin number assigned to the variable is invalid for the target device specified.

0013cb pin/node "number" invalid output extension or usage: "variable"
Error. Either the extension is used improperly or it is not valid for the assigned pin/node.

0014cb invalid input:"var" or pin/node # invalid input:"var"
Error. The variable used as an input was previously assigned to an output that is neither bi-directional nor feeds back into the input array.

0015cb device not yet fully supported: "device"
Fatal. There is an entry for the device in the device library, but the device is not fully supported by the current version of CUPL.

0016cb no expression assigned to: "variable"
Warning. The variable requires an output expression assignment. This warning message is commonly given when all outputs in a bank have the same capability (reset, preset, and so on) and not all the variables have been assigned the same expression. It is given to remind the user that all outputs will be affected.

==========
Note : This warning may be suppressed by assigning the variable to 'b'0 or 'b'1 as appropriate.
==========

0017cb out of memory: "conditions"
Fatal. CUPLB has used all available RAM memory that has been allocated by the operating system, typically as a result of performing a DeMorgan or expansion operation on a large expression. If using fixed polarity devices, check to make sure that the pin variable declaration matches the polarity of the device. Also check whether an intermediate variable which has been expressed in sum-of-product form is being complemented.

==========
Note : This error does not result from insufficient product terms in the device to implement a particular expression.
==========

0018cb missing flip-flop expression for: "variable"
Error. The matching flip-flop expression for a J-K or S-R type flip-flop is missing. Both inputs must have expressions assigned to them. An input may be assigned to 'b'0 or 'b'1 as appropriate.

0019cb DeMorgan's theorem invoked for: "variable"
Warning. DeMorgan's Theorem has been applied to the expression assigned to the variable. Unlike D or T registers, meaningful results are not guaranteed when a DeMorgan equivalent expression is applied to the logic input.

0020cb invalid mix of banked outputs: "variable"
Error. All outputs in a banked group must be used in the same manner. An attempt was made to mix registered and nonregistered output types.

0021cb no expression allowed for: "variable"
Error. Logic expressions are not allowed for reset and preset nodes when the output has been specified as asynchronous. CUPL will generate the proper defaults.

0022cb pin/node "number" conflicting input architectures: "variable"
Error. A fuse-assigned input architecture must be used consistently in all expressions. An attempt was made to specify both fuse options in different expressions.

0023cb disk write error: "filename"
Fatal. CUPLB encountered an I/O error trying to write the indicated file. This error usually occurs when there is insufficient disk space.

0024cb output defined for node which does not exist: "variable"
Error. Variable is defined for a pin or node number which does not exist.

0025cb output mutually excluded by previous output: "variable"
Error. Variable usage is mutually excluded by a previous usage or other output. A shared product term or terms has been defined more than once.

0026cb disk read error, unexpected end of file: "filename"
Fatal. CUPLB encountered an I/O error trying to read the indicated file. This error usually occurs when the file is being read from damaged media.

10xxcb program error: "specifics"
Fatal. An operating system interface problem is suspected.

CUPLM Module - Error Messages

0001cm could not open: "filename"
Fatal. CUPLM cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002cm could not execute program: "program name"
Fatal. CUPLM is unable to perform the next step in the compilation. Be sure that all of the CUPL program files exist on the same directory or disk.

0003cm invalid file: "filename"
Warning. The file was not created by the current version of CUPL.

0004cm out of memory: "conditions"
Fatal. CUPLM has used all available RAM memory which has been allocated by the operating system while performing logic reduction.

==========

Note : This error does not result from insufficient product terms in the device to implement a particular expression.

==========

0005cm disk write error: "filename"
Fatal. CUPLM encountered an I/O error trying to write the indicated file. This error usually occurs when there is insufficient disk space.

0006cm invalid library access key
Fatal. The version of CUPLM is not compatible with the version of the device library. This occurs when either CUPLM or the device library, but not both, has been updated.

0007cm invalid library interface
Fatal. Either the device library was not created using the CUPL library manager, CBLD or CUPLM and the device library are not compatible.

0008cm bad library file: "library"
Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0009cm device is not in library: "device"
Fatal. Either the specified target device does not exist or an entry has not been made in the device library for the device.

00010cm design too complex for this minimisation level

Fatal. CUPLM has exceeded the array size allowed on this machine while reducing a particular expression. Specify a more efficient minimisation level.

00011cm disk read error, unexpected end of file: "filename"

Fatal. CUPLM encountered an I/O error trying to read the indicated file. This error usually occurs when the file is being read from damaged media.

10xxcm program error: "specifics"

Fatal. An operating system interface problem is suspected.

CUPLC Module - Error Messages

0001cc could not open: "filename"
Fatal. CUPLC cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002cc invalid file: "filename"
Warning. The file was not created by the current version of CUPL.

0003cc invalid library access key
Fatal. The version of CUPLC is not compatible with the version of the device library. This occurs when either CUPLC or the device library, but not both, has been updated.

0004cc invalid library interface
Fatal. Either the device library was not created using the CUPL library manager or CUPLC and the device library are not compatible.

0005cc bad library file: "library"
Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0006cc excessive number of product terms: "variable"
Error. The number of product terms needed to implement the logic expression for the given variable exceeds the capacity of the output pin for which it was declared.

0007cc invalid download format(s)
Warning. At least one of the download formats specified is not available for the target device.

0008cc pin can not be used as input: "variable"
Error. The pin to which the variable is assigned provides no input or feedback capability.

0009cc header name undefined, using no_name
Error. The NAME field in the header information is missing. Since CUPLC uses this name to generate download files, the desired file will be created as "no_name" along with the appropriate extension.

0010cc disk write error: "filename"
Fatal. CUPLC encountered an I/O error trying to write the indicated file. This error usually occurs when there is insufficient disk space.

0011cc out of memory: "conditions"

Fatal. CUPLC has used all the available RAM memory allocated by the operating system.

==========

Note This error does not result from insufficient product terms in the device to implement a particular expression.

==========

0012cc disk read error, unexpected end of file: "filename"

Fatal. CUPLC encountered an I/O error trying to read the indicated file. This error usually occurs when the file is being read from damaged media.

0013cc conflicting usage of pinnode:"variable"

Error. Variable usage is mutually excluded by a previous usage of the pin or pinnode. A shared product term or terms has been defined more than once.

0014cc unknown extension encountered: "extension"

Warning. The translation of a CUPL extension into another file format could not be accomplished. The equation is still placed in the new file except the extension has been lost.

0015cc invalid local feedback from "variable name" to "variable name"

Fatal. The local feedback of a macrocell was used outside the quadrant. This means that the feedback of a local macrocell or the internal feedback of a global macrocell was used as input to another macrocell that is located in another quadrant.

10xxcc program error: "specifics"

Fatal. An operating system interface problem is suspected.