UNDERSTANDING PROGRAMMABLE LOGIC

Understanding GAL devices within the context of the various N8VEM projects.

THE GAL 22V10

The PAL and GAL – An introduction

This paper contains original material plus content I copied from information found on the internet. I've edited some of the found content to remove unnecessary references; to reformat it and to clean it up a bit. Where possible I've tried to give credit to the original authors or sources.

I'm very new to the world of programmable logic; I can't guarantee the accuracy of the content although I've tried to make it as accurate as possible.

The intent of the document is to help those who are new to programmable logic come up to speed in the context of the N8VEM efforts. As the N8VEM boards become more complex we will need to start using programmable logic to help the designs fit on a single board.

Most likely one or more of the members of the N8VEM work will be able to provide programmed GAL devices to those who don't have a GAL programmer. I've recently purchased a GAL programmer and will be able to help provide programmed GALs.

Please feel free to send feedback, corrections etc. to me at nbreeden@me.com

Thanks.
Neil B. Breeden II

TABLE OF CONTENTS

INTRODUCTION

Traditional logic such as the TTL ICs used in most of the N8VEM designs are dedicated to the function they provide. The 74LS00 has four 2-input AND Gates; the 74LS04 has six inverters (NOT gates).

Some N8VEM boards already use programmable devices such as EPROMs and FLASH memories. Typically these contain software (firmware) that the microprocessor requires for the system to operate. EPROM and FLASH can also be used to store look up tables; character sets for video boards and many other things.

The use of GALs will expand the range of boards that can be designed as part of the N8VEM project.

Programmable logic allows you to customize the IC to provide the exact function you need. They can have tens or hundreds of logic gates in a single IC. By programming the device you customize which gates and combinations of gates are to be used. It might take 5 regular TTL ICs to create an address decoder; the same functionality can be defined in a single programmable device. Instead of 5 ICs on the printed circuit board we only need one; this frees up space one the board for other functionality.

THE 22V10 – A GREAT PLACE TO START

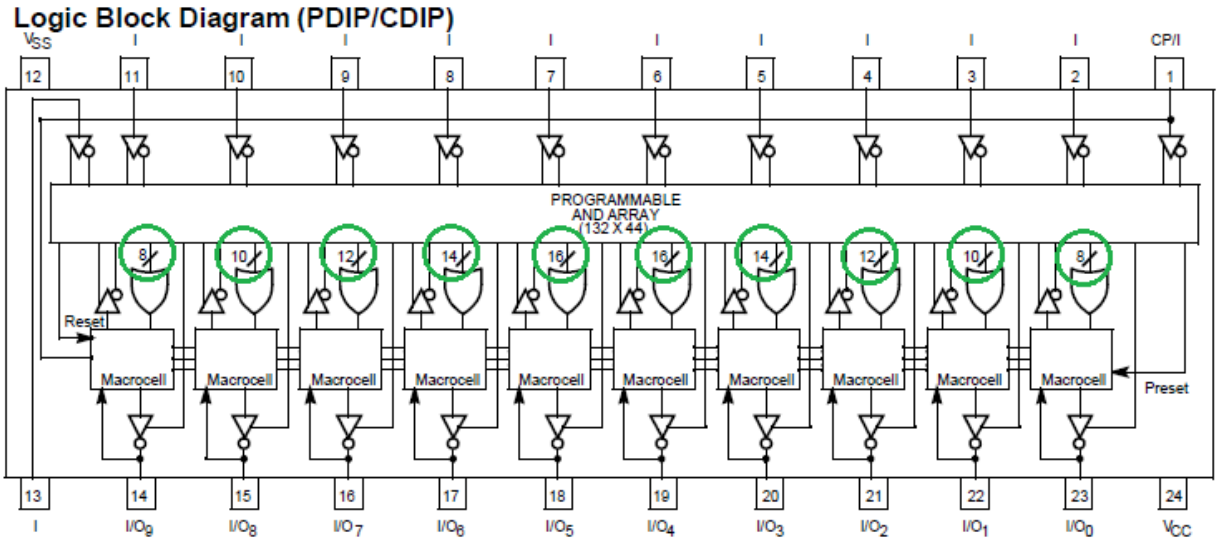
The 22V10 comes in a few different packages. The 24-pin DIP package is the focus of this part of the discussion.

There are 10 potential output pins on pin 14 to 24; this is where the 10 comes from in 22V10. Note that any of these 10 pins can also function as an input pin.

There are 12 pins that are fixed inputs; pin 1 to 11 and pin 13.

The GAL has 22 total I/O pins; this is where the 22 in 22V10 comes from. The number of input pins is then 22 – 10 = 12; there are 12 dedicated input pins.

Each of the I/O pins (pins 14 to 23) supports a differing number of Product Terms. Think of a Product Term as the number of overall OR operations the output can support. I've circled in green the number of Product Terms (PTerm) each particular output supports (From the CYPRESS PALCE22V10 datasheet).



A logic statement that won't compile on Pin 15 (10 PTerms are supported) certainly won't compile on Pin 14 (8 PTerms supported) but may compile on Pins 16, 17 or 18.

There are a number of different manufactures making this GAL; after reviewing data sheets for several different manufactures it must be noted that the programming requirements differ. When programming parts be sensitive to the manufacturer and to the specific part number.

PALS, GALS, PALASM AND THE DEVICE PROGRAMMER

PALs are one time programmable devices; GALs can be erased and reprogrammed a limited number of times.

The GAL version of a device is functionally equivalent to the PAL version.

PALASM is a program that inputs the definition of the function to be implemented and outputs a file that the device programmer can use to 'program the device'.

PALASM compiles .PDS files; it outputs a .JED file.

The .JED file is the programming definition for a generic type of device. Example: You use PALASM to compile for a 'PAL22V10' device, a .JED file is created. You can then program a 'GAL22V10' device using the .JED file from PALASM. The .JED file is the generic programming map for a '22V10' – it doesn't care if the device is a PAL22V10 or a GAL22V10.

My programmer is a Wellon VP-390; it allows me to test the GAL after programming it at +/- 10 percent of VCC. I highly recommend enabling these tests as part of your programming cycle.

The Wellon VP-290 also appears to be a good choice for a general purpose programmer.

If you make the decision to buy a programmer consider buying a higher quality unit. There are many cheap programmers on eBay; my experience with these has been pretty poor. I would strongly suggest consulting the N8VEM forums for advice before you commit to a programmer.

PALASM SUPPORTED DEVICES

After reviewing the PALASM help files I believe the following list covers all devices supported in the version of PALASM linked from the http://www.S100computers.com web site. Please remember that although you specified a PAL in PALASM you can still program a GAL of the same general device specification.

PAL10H20EG8	PAL16R8	PAL22P10
PAL10H20EV8	PAL16RA8	PAL22RX8
PAL10H20G8	PAL16RP4PAL16RP6	PAL22V10
PAL10H20P8	PAL16RP8	PAL23S8
PAL10H20P8	PAL18L4	PAL24L10
PAL10H8	PAL18P8	PAL24R10
PAL10L8	PAL20C1	PAL24R4
PAL12H6	PAL20L10	PAL24R8
PAL12L10	PAL20L2	PAL29M16
PAL12L6	PAL20L8	PAL29MA16
PAL14H4	PAL20R4	PAL32R16
PAL14L4	PAL20R6	PAL32VX10
PAL14L8	PAL20R8	PAL64R32
PAL16C1	PAL20RA10	PAL6L16
PAL16H2	PAL20RS10	PAL8L14
PAL16L2	PAL20RS4	PALC18U8
PAL16L6	PAL20RS8	PALCE16V8
PAL16L6	PAL20S10	PALCE16V8HD
PAL16L8	PAL20X10	PALCE20V8
PAL16P8	PAL20X4	PALCE24V10
PAL16R4	PAL20X8	PALCE26V12
PAL16R6	PAL22IP6	PALCE610

DEFINING THE LOGIC FOR THE GAL

Being comfortable with the following will go a long way towards helping you define the required logic for your design.

The AND function (written as *) for a 2 input AND gate defines the output as Y = A * B

Input A	Symbol	Input B		Output Y
0	*	0	=	0
0	*	1	=	0
1	*	0	=	0
1	*	1	=	1

The OR function (written as +) for a 2 input OR gate defines the output as Y = A + B

Input A	Symbol	Input **A**		Output Y
0	+	0	=	0
0	+	1	=	1
1	+	0	=	1
1	+	1	=	1

The NOT function (written as /) defines the output as Y = /A

Symbol	Input A		Output Y
/	0	=	1
/	1	=	0

By NOTing the output of the AND and OR gates you get the NAND (Not AND) and NOR (Not OR) gates respectively; the little circle indicates that the output pin is NOTed.

/Y = A * B

/Y = A + B

Some examples of how this logic is defined to PALASM follow; note that any `text highlighted in blue` are comments:

I want the output Y to be high when all 3 inputs A, B, C are all high:

```
Y = A * B * C        ; Y=A AND B AND C
```

I want the output Y to be high when inputs A, C are high and input B is low:

```
Y = A * /B * C       ; Y=A AND [B NOT] AND C
```

I want the output Y to be high when any of the inputs A, B, C, D are high:

```
Y = A + B + C + D    ; Y=A OR B OR C OR D
```

I want the output Y to be high when inputs A OR B are high or C AND D are low:

```
Y = (A + B) + (/C * /D) ; Y=A OR B OR ([C NOT] AND [D NOT])
```

Notice that we can use parenthesizes to indicate order of operation precedence. Later in this document you will find a section describing the general order of operation precedence. I prefer to use parenthesizes to clearly indicate the precedence however this is a personal preference only.

Let’s introduce the XOR (eXclusive OR) function (written as :+:). A 2 input XOR gate defines the output as Y = A :+: B

Input B	Symbol	Input A		Output Y	
0	:+:	0	=	0	
0	:+:	1	=	1	
1	:+:	0	=	1	
1	:+:	1	=	0	<- this is where it differs from the OR function

As an example I want the output (Y) to be high when (inputs A, B are both low) or (either input C, D are low but not both low (XOR)) or (input E, F, G are all high):

```
  Y = (/A * /B) + (C :+: D) + (E * F * G)

; Y = ([A NOT] AND [B NOT]) OR (C XOR D) OR (E AND F AND G)
```

If I wanted Y to be low (instead of high) using the logic above you would simply NOT Y as follows:

```
  /Y = (/A * /B) + (C :+: D) + (E * F * G)

;[Y NOT] = ([A NOT] AND [B NOT]) OR (C XOR D) OR (E AND F AND G)
```

READING THE EQUATIONS

Practicing reading the equations out loud will help you learn the logic and how to read them; for example:

The equation `AND_Out = A_In * B_In`

Is read as:

The output `AND_Out` is defined as input `A_In AND` input `B_In`

The equation `AND_Out = A_In + B_In`

Is read as:

The output `AND_Out` is defined as input `A_In OR` input `B_In`

Practice reading the remaining equatons.

```
XOR_Out  = A_IN :+: B_In      ; Pin 18 = Pin 1 XOR Pin 2

ANOT_Out = /A                 ; Pin 20 = NOT Pin 1

BNOT_Out = /B                 ; Pin 22 = NOT Pin 2
```

LET'S LOOK AN EXAMPLE PDS FILE.

The text in green is the contents of the file; the text in blue describes the contents and is not part of the actual PDS file.

Standard header entries:

```
TITLE    Simple logic statements
PATTERN  TEST1.PDS
REVISION 0
AUTHOR   Neil Breeden
COMPANY  N8VEM
DATE     05/30/14
```

Device Definition:

```
CHIP TEST_GAL PAL22V10   ; Defines a generic 22V10
```

The device pins are given names, this helps make the logic definitions easier to read. Further it provides documentation to help explain the design:

```
;------------------ PIN Declarations ---------------
PIN 1  A_In              ; A Input
PIN 2  B_In              ; B Input
PIN 14 AND_Out           ; Output to demonstrate AND
PIN 15 OR_Out            ; Output to demonstrate OR
PIN 16 XOR_Out           ; Output to Demonstrate XOR
PIN 17 ANOT_Out          ; Output to demonstrate A NOT
PIN 18 BNOT_Out          ; Output to demonstrate B NOT
```

The actual logic equations, this is where we define how the GAL will be configured to perform the work we need done.

```
EQUATIONS ;--------- Boolean Equation Segment -----------
AND_Out  = A_In  *  B_In      ; Pin 14 = Pin 1 AND Pin 2
OR_Out   = A_In  +  B_In      ; Pin 16 = Pin 1 OR  Pin 2
```

```
XOR_Out  = A_IN :+: B_In      ; Pin 18 = Pin 1 XOR Pin 2
ANOT_Out = /A_In              ; Pin 20 = NOT Pin 1
BNOT_Out = /B_In              ; Pin 22 = NOT Pin 2

SIMULATION
TRACE_ON   A_In  B_In AND_Out OR_Out XOR_Out ANOT_Out BNOT_Out
SETF      /A_IN /B_IN    ; 0 0
SETF      /A_IN  B_IN    ; 0 1
SETF       A_IN /B_IN    ; 1 0
SETF       A_IN  B_IN    ; 1 1
SETF      /A_IN /B_IN    ; 0 0
TRACE_OFF
```

End of the .PDS file.

Based on the design I would expect the simulation output to look as follows.

```
          ggggg
A_In      LLHHL
B_In      LHLHL
AND_Out   LLLHL
OR_Out    LHHHL
XOR_Out   LHHLL
ANOT_Out  HHLLH
BNOT_Out  HLHLH
```

Each column with 'g' as a header indicates the state of the pins for one SETF in the simulation data. There are five SETFs so there are five 'g' columns.

In the first SETF we set both A_In and B_In low; As both are low both the AND, OR and XOR outputs are low. The two NOTed outputs are both high as they are the NOTed values of the inputs.

The next SETF sets B_In high. The AND output remains low; the OR and XOR outputs are now high; the BNOT output is now low.

Can you explain the remaining columns results?

On pin 19 add a NOR PTerm along with the simulation data.

Add a more complex PTerm such as Y = ((A * B) + (C * D) * E)

What would the simulation data look like?

What would the simulation output look like?

SOME BEST PRACTICES, NOTES, RAMBLINGS AND COMMENTS….

Different manufactures of the same device type can have very different programming requirements. I damaged 4 Lattice parts when I programmed them using the definition for a NS device. The Lattice part wanted 9 volts for VPP; the NS part wanted 14 volts for VPP. This is also true for EPROMS, FLASH etc. An Intel 2764 has a different programming configuration then an Intel 27C64. VPP refers to the programming voltage applied to the device to erase or program it.

An erased GAL has all PTerms functionality enabled; this will result in all of the output pins always being high; keep this in mind.

When erased the output is basically defined as "Output = (A * B) + (/A * B) + (A * /B) etc….", this results in the output always being true (high). By programming you disable the unneeded terms.

I was confused several times thinking my design wasn't working when in reality the GAL was erased due to the fact that the software my VP-390 uses clears the data buffer each time the GAL type is changed; this includes changing the manufacture for the same type of GAL. With the buffer cleared a programming pass will leave the device erased. You may need to reload the .JED file after changing devices.

Using LEDs, a breadboard and jumper wires you can build a test rig to allow you to do a basic verify test to see if the programmed device works as expected. This is HIGHLY recommended as trying to debug a design in circuit is very difficult.

Don't be afraid to fail; you may destroy a few parts; your designs may not work initially or they may not represent what you were trying to do. This is OK, you should stay with it and try to understand why it failed; this is where the BEST learning comes from.

Reach out and ask for help when needed. This is why we have the N8VEM forums. For me there are times when the act of typing the question; of trying to explain the problem will suddenly give me insight that allows me to fix it or to try something different.

Keep notes; keep notes; be sure to keep notes. I keep detailed notes as I work on a design; when I solder up a board; when I write code etc. I go so far as to scan 200dpi and 600dpi images of both side of the blank PCB before I begin to assemble it. Again, get in the practice of keeping notes.

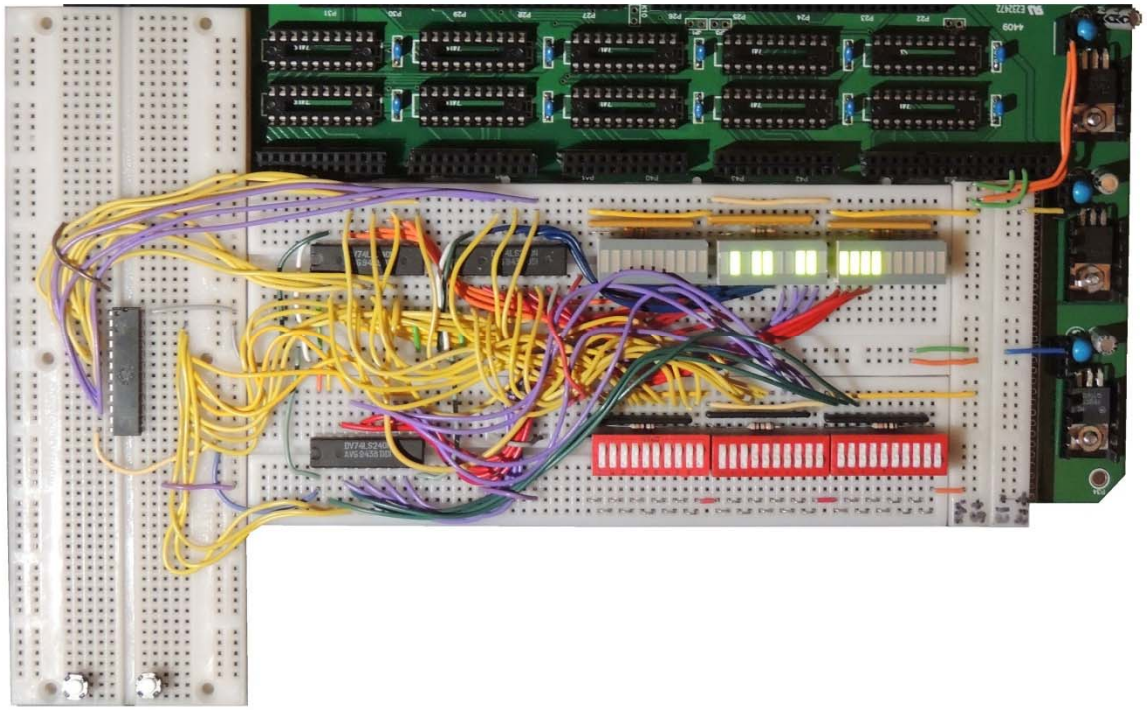
Review the forums first. I'm guilty of not always doing this; it is however a practice I'm working on getting into. Share your experiences, share your successes, and share your failures. As a community we learn together.

MY TESTING RIG FOR 22V10 GALS

The GAL under test is on the left side of the breadboard.

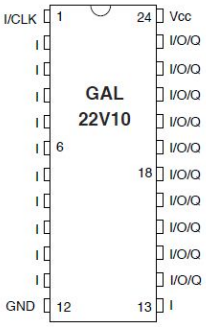
The DIP switches are for pins 1 to 24; the extra 6 on the right end are not used.

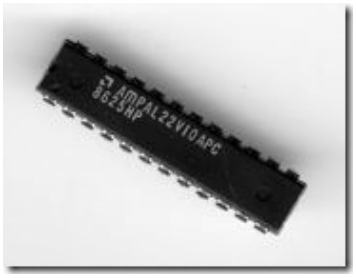
The LEDs represent pins 1 to 24; again the last 6 are unused.

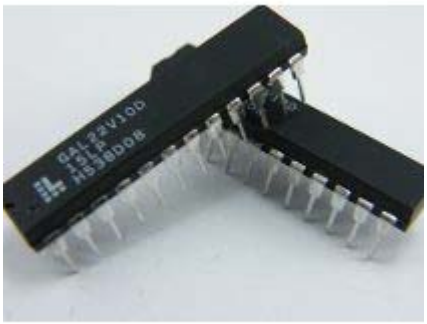


This design includes buffer LED drivers; current limiting resistors for the LEDs and pull-up resistors for the DIP switches.

TYPICAL 22V10S AND THE GENERIC GAL 22V10 PINOUT:







RESOURCES

S100 Computers discussion of GALs
Includes instructions on downloading PALASM and getting it to run:

http://www.s100computers.com/My%20System%20Pages/ISA%20to%20S100%20Bus/Intro%20To%20GALs.htm

Examples of various logic implementations:

http://orion.ipt.pt/~fmbarros/ed/palasmex.pdf

A lot of good reviews and insights into good tools along with a lot of good material on electronics:

www.eevblog.com

Videos lectures on electronics and GALs:

http://www.allaboutcircuits.com/videos/index.html
http://www.allaboutcircuits.com/videos/89.html

Interesting information - the preview doesn't show all pages but there is a lot of interesting material presented:

An interesting book on Google Books

Other links:

http://mazsola.iit.uni-miskolc.hu/cae/docs/theor00.html

http://ee.sharif.edu/~logic_circuits_t/readings/PLD.pdf

http://sourceforge.net/projects/logiccircuitd/?source=directory

THE DESIGN CYCLE

Plan, Document, Execute, Document, Test, Document – repeat as required.

1. Define the requirements – what is the problem to be solved?
    a. Document it – if you can't clearly and completely describe it then you are not ready to move to the next step.
2. Decide how you want to solve it; in this example I am assuming you will be using a PLD.
    a. Document the pro's and con's to the various approaches you consider.
3. Define the inputs and outputs for the projects.
    a. Document it – if you can't clearly and completely describe them then you are not ready to move to the next step.
4. You should be able to define a simulation data set for PALASM; if you can't then you most likely don't understand your design sufficiently to test the programmed part.
    a. Document it.
5. Create or update the PALASM PDS file.
6. Using PALASM compile the PSD file; fix any bugs and compile again until you get a clean compile.
7. Simulate the design in PASASM; repeat steps 5 thru 7 until the simulation is successful.
8. Program a GAL using the JED file produced by PALASM.
    a. Be sure to select the correct manufacture and device in the programming software.
    b. Be sure to load / reload the .JED file as required; changing the device or manufacturer of a device will require you to reload the JED file.
9. Test the design on a breadboard; repeat steps 5 thru 9 until it works on the breadboard.
10. Make a backup copy the PDS file. Rename it to include the time and date.
    a. You may end up with multiple copies; I believe it's safest to keep them all as you can revert back to an older one if needed.
11. Test in the application, repeat steps 5 thru 11 until it works in the application.

All content below this point was copied from resources found on the internet.

PROGRAMMABLE LOGIC DEVICES

Programmable Logic Devices (PLDs) (also known as PALs) are popular devices for implementing digital designs. These devices can be used where earlier systems used TTL or CMOS logic ICs. The PLDASM is a tool that allows Boolean equations to be programmed into a PLD in order to perform a user-defined logic function. Boolean equations make it possible to describe a function in an efficient manner, and this assures that the designer achieves the most compact solution with the fastest propagation delays. Furthermore, with Boolean equations the PLD can function as an address decoder, state machine or counter, and perform any number of other tasks ranging from the simple to the complex. While initially PLDs provided a savings in the amount of space used on a PC board, recent high speed PLDs are often significantly faster than the equivalent circuit implemented in TTL logic. Another recent development in PLDs is the complexity of the macrocells used for I/O. PLDASM automatically configures these macrocells, according to a set of simple rules which apply to all the PLDs supported by PLDASM. This allows substitution of one device for another, and reduces the amount of time required to 'learn' a new PLD.

A PLDs internal structure is built as an AND/OR matrix. A programmable input AND array can generate any AND function of all device inputs (with or without inversion). These AND functions are called 'Product Terms'. Product terms feed a multiple input OR gate. Since the AND/OR matrix can express any Boolean transfer function, the flexibility and functionality of a PLD is limited only by the number of terms available in the AND - OR arrays. PLD devices are available in different sizes, some with over 40 inputs, and some with up to 19 Product Terms per output. The outputs range from simple tri-state drivers to complex registered macrocells with programmable inverters.

BOOLEAN FUNCTIONS

In an un-programmed PLD, all fuses are intact. In other words, every input line is 'ANDed' with all other input lines (including any feedback terms available in the device) The output of these AND functions is fed into an OR gate and is then either fed onto more complex functions or presented directly on the output pins of the device.

For example, let us assume that we have a simple PLD with two input terms (A and B) and two output terms (X and Y). Internally, the device also makes the inverse of the input terms available (/A and /B). In the un-programmed state, the logical function of the device can be represented by the following Boolean equations.

```
X = A*B + /A*/B + A*/B + /A*B
Y = A*B + /A*/B + A*/B + /A*B
```

In this state clearly the device has little use; X and Y are always equal to 1, regardless of the inputs A and B. However, when some of the terms in each of the AND functions are removed, the power of the device becomes obvious. For example, let us assume that the following fuses are 'blown':

```
From X, /A*/B, A*/B
From Y,  A*B, /A*/B, /A*B
```

In the example given, the fuses were 'blown' so that no connection remained. The equations that remain after programming of the device are shown below.

```
X = A*B + /A*B
Y = A*/B
```

As can be seen, very quickly it becomes possible to provide complicated logic functions in a single package. The other main advantage of PLDs is that their precise function can be adapted by the individual designer to meet the application needs, even if the design specification changes after PC boards have been built, (or if bugs are found during system testing and production).

The above equations are usually entered into a disk file using a text editor or the editor built into PALASM. The disk file is passed through PALASM to create a JEDEC file. The JEDEC file can then be loaded into the PAL program for programming a device.

In order to program a PLD, it is necessary to address each fuse in the device individually and to program it. PALASM compiles the equations in the .PLS file into a fuse file formatted as JEDEC data; this file has a .JED extension.

For each input signal, there are two input line numbers, one for the actual input signal and one for its inverse. So, for this device there will be four input lines (1 = A, 2 = /A, 3 = B, 4 = /B).

Additionally, there will be eight product line numbers as there were eight OR combinations in the un-programmed device (4 for each output term). Therefore, for this device, the fuse map needed by the programming utility to create the Boolean functions described is shown below.

	Input Line				
	A	/A	B	/B	
Product Line Number	1	2	3	4	Output
1	X	-	X	-	X
2	-	X	X	-	
3	-	-	-	-	
4	-	-	-	-	
5	X	-	-	X	Y
6	-	-	-	-	
7	-	-	-	-	
8	-	-	-	-	

The fuse map is stored in a JEDEC file where each fuse location represented by an 'X' is stored as a '0' (zero) and will be unaffected by the programming utility. Each location represented by a '-' is stored as a '1' and will be blown by the programming utility.

An example JEDEC file:

```
PALASM4  PAL ASSEMBLER   - MARKET RELEASE 1.5a (8-20-92)
 (C) - COPYRIGHT ADVANCED MICRO DEVICES INC., 1992

TITLE   :Simple logic example     AUTHOR :Neil Breeden
PATTERN :TEST1.PDS                COMPANY:N8VEM
REVISION:0                        DATE   :05/30/14

PAL22V10
TEST_GAL*
QP24*
QF5828*
G0*F0*
L0000 00000000000000000000000000000000000000000000*
L0044 00000000000000000000000000000000000000000000*
L0088 00000000000000000000000000000000000000000000*
…
L2904 11111111111111111111111111111111111111111111*
…
L5720 00000000000000000000000000000000000000000000*
L5764 00000000000000000000000000000000000000000000*
L5808 01010101011111110111*
C3D10*
16B0
```

Content copied from http://orion.ipt.pt/~fmbarros/ed/PALASM%20Language%20Guide.htm#Top

DESIGN WITH BOOLEAN EQUATIONS

A .PDS file using Boolean equations to specify a design consists of two or three sections: declarations, equations, and optional simulation specifications. The declaration section is used to identify the design, list target device data, and define string constants. The equation section defines the outputs in terms of inputs and feedback paths. It also supports device-specific configuration.

The vocabulary of .PDS files is given separately. The grammar and syntax for .PDS files with Boolean equations is also given separately. General comments about grammar and syntax issues follow.

Note that the first several lines of the grammar (TITLE, PATTERN, ..., DATE) are all optional. If the optional lines are omitted, warning messages will be generated. The information following these optional lines is limited to 24 characters.

The reserved word CHIP is required. The description is limited to 13 alphanumeric characters. The device name must designate a device supported by the software. The on-line databook (in PALASM2) shows which devices are supported. The names of the pins as they are used in the program follow. Traditional style dictates that the pin numbers be placed by the names using comment lines.

Some PLDs have internal global preset or reset lines which affect all the registers in the device. If the device being programmed has this feature (e.g., PAL22V10s), PALASM requires the definition of a phantom pin at the end of the pin definitions. For a 24 pin device, the phantom pin would be defined as a 25th pin. Typical names for the pin are global.rst or global.set which can then be used in the equations section, if desired. Omitting a phantom pin results in the warning "Not enough pins defined".

The STRING section is optional. It permits frequently used patterns to be replaced by a name. For example, a four-literal expression for the numeric value three could be declared as "ONE '/I3 * /I2 * I1 * I0'". Strings can contain other strings, but the references must not be recursive.

The section starting with the reserved word EQUATIONS is required. What follows is a set of Boolean equations which define the functions implemented by the PLD. The results can be combinational (designated with "="), synchronously registered (designated with ":="), or asynchronously latched (designated with "*="). The equations can span more than one line, but no single line may span more than 128 columns.

The permitted operations are the standard Boolean operators with normal precedence: NOT ("/"), AND ("*"), OR ("+"), and XOR (":+:"). Parentheses may be used to group terms.

The output can also be specified as being asserted low or asserted high. Outputs which are to be asserted low are preceded by a slash. For example, /Q2 = I2 + /I1 + I0 would be low when I2=1, I1=0, and I0=1. It would be high otherwise.

DESIGN WITH STATE MACHINES

PALASM allows state machine circuits to be descibed as either Mealy machines (outputs depend on both current state and current inputs) or as Moore machines (outputs depend on just the current state). The specific syntax and semantics for .PDS files with state machine design is given separately. Specific comments about the syntax and semantics follow.

The declaration section follows the same rules as for a Boolean equation design.

The type of state machine to be implemented is specified by using the reserved word MEALY_MACHINE or MOORE_MACHINE.

The global defaults provide a concise way of specifying circuit behavior for cases not explicitely defined in later parts of the design specification. Default state transitions can be specified in one of three ways:

```
DEFAULT_BRANCH <state name>

DEFAULT_BRANCH HOLD_STATE

DEFAULT_BRANCH NEXT_STATE
```

The first defaults to the specified state, the second to the same state, and the third to the next state appearing in the design description.

Default outputs can also be specified as shown below.

```
OUTPUT_HOLD <output pin list>

DEFAULT_OUTPUT <output pin values>
```

In the first case, the list specifies output pins which do not change. In the second case, the output pins go to the specified values. The character '%' preceding a pin name in a pin list denotes a "don't care" output while a '/' preceding a pin name indicates a low output value.

The optional state assignment section equates state names with a unique set of state variable values. The variable values are stored in registers. The syntax of state assignments is

```
<state name> = <var1val> * <var2val> * .... * <varNval>
```

The character '/' precedes variables which are low. State names must be unique and may contain up to 14 characters. By assigning your states values, you may get a better design than by allowing Palasm to do the assignment for you.

The state equations define the state transitions of the state machine. The syntax of each state equation is

```
<current state name> := <condition1> -> <next state 1>
                      +  <condition2> -> <next state 2>
                      ...
                      +  <conditionN> -> <next state N>
                      +-> <local default state>
```

The current state and next state names are those defined in the state assignment section. The conditions are defined in the condition section. The local default state line is optional. When present, the local default state overrides any global default state definition. When absent, the global default is used. An unconditional state transition should use the reserved word VCC as the condition.

An output equation for each state equation is required if OUTPUT_HOLD (in PLS and PROSE designs) or DEFAULT_OUTPUT is specified in the design. Otherwise, the output equations are optional. If the outputs are the same as the state, do not specify output equations. Registered Mealy outputs take on new values one clock cycle after a new state is reached. All others are valid when the new state is reached. For Mealy machines, the output syntax is

```
<state name>.OUTF <OP> <condition1> -> <output list 1>
                  +   <condition2> -> <output list 2>
                  ...
                  +   <conditionN> -> <output list N>
                  +-> <local output defaults>
```

For Moore machines, the syntax is

```
<state name>.OUTF <OP> <output list>
```

where <OP> is again either := for registered outputs or = for combinational outputs. The syntax of the output list is

```
<pin label> * <pin label> * ... * <pin label>
```

where the number of labels in the list is one or more.

The condition section is used to define unique input value combinations. These conditions are then used in the state transition section. The condition section begins with the reserved word CONDITIONS and is followed by a list of definitions with the following syntax:

```
<condition name> = <input Boolean expression>
```

The condition name can contain up to 14 characters and must be unique. The input Boolean expression must use input names as defined in the pin list or string section and it must be unique. Conditions involving only one input do not need to be explicitely defined. Care should be taken to define conditions so that only one is true at any given time.

SIMULATION

PALASM provides an event-driven simulator for PLD design. The simulation is specified as an optional part of the .PDS design file. It begins with the reserved word SIMULATION and is followed by simulation commands. The results of the simulation are stored in two files: the .HST file which contains a complete history of the simulation and the .TRF file which contains a trace of signals specified by the TRACE_ON command.

The next section covers the syntax and meaning of the simulation commands as well as the interpretation of simulation results. A brief simulation command summary is also available.

SIMULATION COMMANDS

The simulation commands can be divided into three categories: value, control, and verification.

VALUE SIMULATION COMMANDS

The value commands set simulation values. The general syntax is

```
COMMAND <List of pin names and values>
```

The list of pin names consists of the name of one or more pins, possibly qualified by the '/' character. Names are separated by blank spaces. The '/' indicates the signal is low or complemented. Its absence indicates the signal is high or un-complemented. A '/' in the pin list will complement a '/' in the CHIP declaration section.

The PRLDF command is used to initialize the values of registers which can be loaded with a value at power-up (preloaded). For example, let P1, P2, and P3 be the output pins associated with registers which are to be preloaded with 1, 0, and 1, respectively. This would be stated in the simulation as

```
PRLDF  P1 /P2 P3
```

If the device cannot be preloaded, the command simply initializes the registers. The Xeltek programmer in the EE department does not support preloading.

The SETF command specifies input signal values. For example, let I1, I2, and I3 be pins associated with input signals which are to be set to 0, 1, and 1, respectively. This would be stated in the simulation as

```
SETF  /I1 I2 I3
```

The inputs will retain the values until explicitly changed. Until a value is specified, input values default to 'undefined'. SETF can be used with clock input pins.

The CLOCKF command generates a clock pulse signal on the specified clock input pins. The pulse goes low-high-low. For example, consider the clock signal CLK0. It would be pulsed by

```
CLOCKF CLK0
```

CONTROL SIMULATION COMMANDS

The control commands permit repetitive and selective execution of commands based on condition evaluation. The conditions for the WHILE and IF commands make use of the relational operators< , >, =, <=, and >=. The conditional expressions may not contain nested parentheses.

The syntax of the FOR command is

```
FOR <index var> := <start> TO <end> DO
  BEGIN
```

```
    <command list>
  END
```

An example of the FOR command follows:

```
FOR J:=1 TO 8 DO
  BEGIN
    SETF /I0 I1
    CLOCKF CLK0
  END
```

FOR loops may be nested. The value of <start> must be less than that of <end> and both must be non-negative. If the limits are equal, the loop is NOT executed.

The syntax of the WHILE command is

```
WHILE <condition> DO
    BEGIN
      <command list>
    END
```

An example WHILE statement:

```
WHILE (J <= 7) DO
  BEGIN
    SETF I0 I1
    CLOCKF CLK1
    J := J + 1
  END
```

WHILE loops may be nested. The <condition> may be either a numeric comparison or Boolean evaluation.

The syntax of the IF...THEN...ELSE command is

```
IF <condition> THEN
```

```
    BEGIN
      <command list>
    END
ELSE
    BEGIN
      <command list>
    END
```

An example of an IF ... THEN ... ELSE command:

```
IF (/Q0 * Q1) THEN
  BEGIN
    SETF I0 /I1
    CHECK Q1 Q2 /Q3
  END
ELSE
  BEGIN
    SETF /I0 I3
    CLOCKF CLK
    CHECK /Q1 /Q2 /Q3
  END
```

The ELSE part is optional. As with the WHILE command, the <condition> may be either a numeric comparison or Boolean evaluation

VERIFICATION SIMULATION COMMANDS

The verification commands allow the correctness of the design to be checked.

The TRACE_ON command commences the writing of specified signal values to the .TRF trace file. The syntax of the command is

```
TRACE_ON  <list of pin names>
```

The signal values will be put in the file in the order they occur in the pin list and with the same polarity. The values will be recorded in the file until a TRACE_OFF command is encountered or

the simulation ends. TRACE_OFF has no arguments. Different signals can be traced by specifying them in a TRACE_ON command which follows a TRACE_OFF.

The CHECK command compares simulation values with expected values. The syntax of the command is

```
CHECK  <list of pin names and values>
```

The list of pin names consists of the name of one or more pins, possibly qualified by the '/' character. Names are separated by blank spaces. The '/' indicates the signal is low or complemented. Its absence indicates the signal is high or un-complemented.

For example, suppose that at a given point in a simulation, the pins P1, P2, and P3 are to have the values 0, 1, and 0, respectively. This would be specified as

```
CHECK  /P1 P2 /P3
```

As with the pin list, a '/' in the pin list will complement a '/' in the CHIP declaration section. The following table shows the relationship between pin declarations in the CHIP section and pin names in the CHECK command.

Pin Names used in CHECK		
	Definition in CHIP	
Test Level	**P1**	**/P1**
High	P1	/P1
Low	/P1	P1
	Form in CHECK Name List	

WRITING SIMULATIONS

General flow: set registers, set input signals

INTERPRETING SIMULATION RESULTS

oscillatory conditions

differences between expected and simulation results

.PDS FILE VOCABULARY

CHARACTERS

LEGAL

Upper and lower case alphanumeric, space, tab, underscore

ILLEGAL

` ~ ! @ # $ % ^ & - { } [ ] " ? < >

LINES

Maximum of 128 characters per line

RESERVED WORDS

```
AUTHOR
BEGIN
CHECK
CHIP
CLKF
CLOCKF
CMBF
COMPANY
CONDITIONS
DATE
DEFAULT_BRANCH
DEFAULT_OUTPUT
DO
ELSE
END
EQUATIONS
FOR
GND
HOLD_STATE
IF
MASTER_RESET
MEALY_MACHINE
MOORE_MACHINE
NC
NEXT_STATE
OR
OUTPUT_ENABLE
OUTPUT_HOLD
```

```
PATTERN
POWER_UP
PRLDF
R
REVISION
RSTF
S
SETF
SIMULATION
STATE
STRING
THEN
TITLE
TRACE_OFF
TRACE_ON
TRST
VCC
WHILE
```

SPECIAL SYMBOLS

```
''   (Single quotes) Delimit strings
,    Pin list separator (comma)
()   Enclose pins in logic expressions
;    Precede comments, which run to end of line
/    NOT or active-low
*    AND
+    OR
:+:  XOR
=    Combinational output
*=   Latched output
:=   Registered output
```

OPERATOR PRECEDENCE

```
/ * + :+: for NOT, AND, OR, and EXCLUSIVE OR, respectively
```

STATE MACHINE SYMBOLS

```
->   State transition (go to state ...)
+->  Local default state transition (otherwise, go to state ...)
%    Don't care value for output (used like '/')
```

```
TITLE      <Design title>
PATTERN    <Identification such as file name>
REVISION   <Version or other ID>
AUTHOR     <Name of designer>
COMPANY    <Organization name>
DATE       <Relevant date>

CHIP  <Description>  <Device name>

; <Pin numbers, eg 1   2   3   4  5  6  7  8>
  <pin names,   eg Clk Clr Pre I1 I2 I3 I4 GND>

; <Pin numbers, eg 9   10  11  12  13  14  15  16>
  <pin names,   eg NC  NC  Q1  Q2  Q3  Q4  NC  Vcc>

STRING  <Name>  '<Characters to substitute>'
        <more string definitions>

EQUATIONS

  <combinatorial equations of the form
    OutName = Name1 Op1 Name2 .... OpN NameM>

  <registered equations of the form
    OutName := Name1 Op1 Name2 .... OpN NameM>

  <latched equations of the form
    OutName *= Name1 Op1 Name2 .... OpN NameM>
```

NOTE: <text> designates text which is supplied by the designer.

```
TITLE      <Design title>
PATTERN    <Identification such as file name>
REVISION   <Version or other ID>
AUTHOR     <Name of designer>
COMPANY    <Organization name>
DATE       <Relevant date>

CHIP  <Description>  <Device name>

; <Pin numbers, eg 1   2   3   4  5  6  7  8>
  <pin names,   eg Clk Clr Pre I1 I2 I3 I4 GND>

; <Pin numbers, eg 9   10  11  12  13  14  15  16  phantom>
  <pin names,   eg NC  NC  Q1  Q2  Q3  Q4  NC  Vcc  global>

STRING  <Name>  '<Characters to substitute>'
        <more string definitions>

STATE

  <kind of state machine>

  <global defaults for when behavior is not defined
    by the state equations>

  <state assignment definitions>

  <state transition and output definitions>

CONDITIONS
    <Name> = <Boolean equations specifying condition>

NOTE: <text> designates text which is supplied by the designer.
```

SIMULATION COMMANDS SUMMARY

VALUE COMMANDS

```
PRLDF    Initialize preloadable register outputs

SETF     Specify input values

CLOCKF   Generate clock signal for clock input
```

CONTROL COMMANDS

```
FOR...TO...DO

WHILE...DO

IF...THEN...ELSE
```

The syntax and use of these three commands is comparable to computer languages like BASIC, Modula-2, etc.

VERIFICATION COMMANDS

```
CHECK       Compare expected and simulated signal values

TRACE_ON    Specifies signals for .TRF file and recording interval

TRACE_OFF
```

Content copied from http://orion.ipt.pt/~fmbarros/ed/PALASM%20Language%20Guide.htm#Top

CYPRESS

PALC22V10D

Flash Erasable, Reprogrammable CMOS PAL® Device

Features

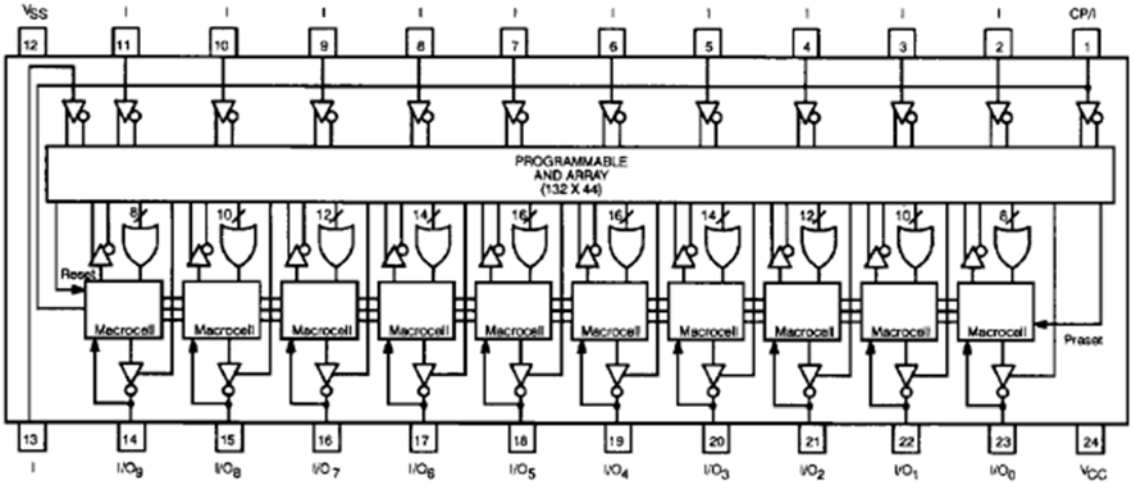
- Advanced second-generation PAL architecture
- Low power
  - 90 mA max. commercial (10 ns)
  - 130 mA max. commercial (7.5 ns)
- CMOS Flash EPROM technology for electrical erasability and reprogrammability
- Variable product terms
  - 2 x (8 through 16) product terms
- User-programmable macrocell
  - Output polarity control
  - Individually selectable for registered or combinatorial operation
- Up to 22 input terms and 10 outputs
- DIP, LCC, and PLCC available
  - 7.5 ns commercial version
    5 ns t_{CO}
    5 ns t_S
    7.5 ns t_{PD}
    133-MHz state machine
  - 10 ns military and industrial versions
    6 ns t_{CO}
    6 ns t_S
    10 ns t_{PD}
    110-MHz state machine
  - 15-ns commercial and military versions
  - 25-ns commercial and military versions
- High reliability
  - Proven Flash EPROM technology
  - 100% programming and functional testing

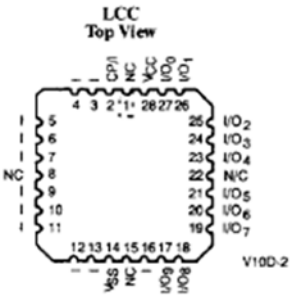
Functional Description

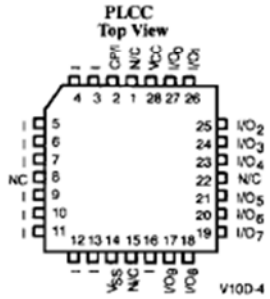
The Cypress PALC22V10D is a CMOS Flash Erasable second-generation programmable array logic device. It is implemented with the familiar sum-of-products (AND-OR) logic structure and the programmable macrocell.

The PALC22V10D is executed in a 24-pin 300-mil molded DIP, a 300-mil cerDIP, a 28-lead square ceramic leadless chip carrier, a 28-lead square plastic leaded chip carrier, and provides up to 22 inputs and 10 outputs. The 22V10D can be electrically

Logic Block Diagram (PDIP/CDIP) and Pin Configurations







PAL is a registered trademark of Advanced Micro Devices.

Functional Description (continued)

erased and reprogrammed. The programmable macrocell provides the capability of defining the architecture of each output individually. Each of the 10 potential outputs may be specified as "registered" or "combinatorial." Polarity of each output may also be individually selected, allowing complete flexibility of output configuration. Further configurability is provided through "array" configurable "output enable" for each potential output. This feature allows the 10 outputs to be reconfigured as inputs on an individual basis, or alternately used as a combination I/O controlled by the programmable array.

PALC22V10D features a variable product term architecture. There are 5 pairs of product term sums beginning at 8 product terms per output and incrementing by 2 to 16 product terms per output. By providing this variable structure, the PALC22V10D is optimized to the configurations found in a majority of applications without creating devices that burden the product term structures with unusable product terms and lower performance.

Additional features of the Cypress PALC22V10D include a synchronous preset and an asynchronous reset product term. These product terms are common to all macrocells, eliminating the need to dedicate standard product terms for initialization functions. The device automatically resets upon power-up.

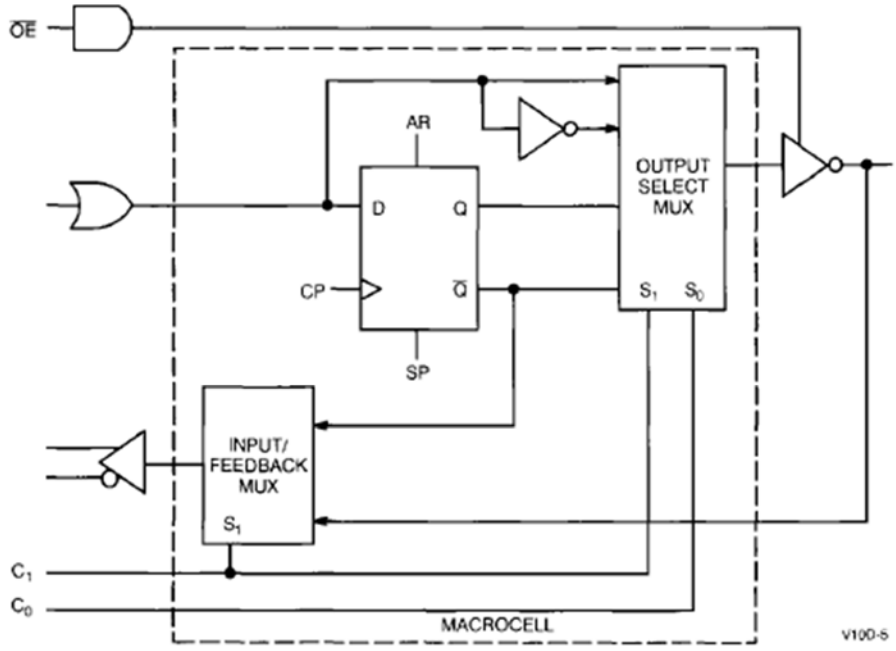
The PALC22V10D, featuring programmable macrocells and variable product terms, provides a device with the flexibility to implement logic functions in the 500- to 800-gate-array complexity. Since each of the 10 output pins may be individually configured as inputs on a temporary or permanent basis, functions requiring up to 21 inputs and only a single output and down to 12 inputs and 10 outputs are possible. The 10 potential outputs are enabled using product terms. Any output pin may be permanently selected as an output or arbitrarily enabled as an output and an input through the selective use of individual product terms associated with each output. Each of these outputs is achieved through an individual programmable macrocell. These macrocells are programmable to provide a combinatorial or registered inverting or non-inverting output. In a registered mode of operation, the output of the register is fed back into the array, providing current status information to the array. This information is available for establishing the next result in applications such as control state machines. In a combinatorial configuration, the combinatorial output or, if the output is disabled, the signal present on the I/O pin is made available to the array. The flexibility provided by both programmable product term control of the outputs and variable product terms allows a significant gain in functional density through the use of programmable logic.

Along with this increase in functional density, the Cypress PALC22V10D provides lower-power operation through the use of CMOS technology, and increased testability with Flash reprogrammability.

Configuration Table

Registered/Combinatorial		
C_1	C_0	Configuration
0	0	Registered/Active LOW
0	1	Registered/Active HIGH
1	0	Combinatorial/Active LOW
1	1	Combinatorial/Active HIGH

Macrocell



Maximum Ratings

(Above which the useful life may be impaired. For user guidelines, not tested.)

Storage Temperature ..... −65°C to +150°C

Ambient Temperature with Power Applied ..... −55°C to +125°C

Supply Voltage to Ground Potential (Pin 24 to Pin 12) ..... −0.5V to +7.0V

DC Voltage Applied to Outputs in High Z State ..... −0.5V to +7.0V

DC Input Voltage ..... −0.5V to +7.0V

Output Current into Outputs (LOW) ..... 16 mA

DC Programming Voltage 12.5V

Latch-Up Current ..... >200 mA

Static Discharge Voltage (per MIL−STD−883, Method 3015) ..... >2001V

Operating Range

Range	Ambient Temperature	V_{CC}
Commercial	0°C to +75°C	5V ±5%
Military[1]	−55°C to +125°C	5V ±10%
Industrial	−40°C to +85°C	5V ±10%

Electrical Characteristics Over the Operating Range[2]

Parameter	Description	Test Conditions			Min.	Max.	Unit
V_{OH}	Output HIGH Voltage	V_{CC} = Min., V_{IN} = V_{IH} or V_{IL}	I_{OH} = −3.2 mA	Com'l	2.4		V
			I_{OH} = −2 mA	Mil/Ind			
V_{OL}	Output LOW Voltage	V_{CC} = Min., V_{IN} = V_{IH} or V_{IL}	I_{OL} = 16 mA	Com'l		0.5	V
			I_{OL} = 12 mA	Mil/Ind			
V_{IH}	Input HIGH Level	Guaranteed Input Logical HIGH Voltage for All Inputs[3]			2.0		V
V_{IL}[4]	Input LOW Level	Guaranteed Input Logical LOW Voltage for All Inputs[3]			−0.5	0.8	V
I_{IX}	Input Leakage Current	$V_{SS} \le V_{IN} \le V_{CC}$, V_{CC} = Max.			−10	10	µA
I_{OZ}	Output Leakage Current	V_{CC} = Max., $V_{SS} \le V_{OUT} \le V_{CC}$			−40	40	µA
I_{SC}	Output Short Circuit Current	V_{CC} = Max., V_{OUT} = 0.5V[5, 6]			−30	−90	mA
I_{CC1}	Standby Power Supply Current	V_{CC} = Max., V_{IN} = GND, Outputs Open in Unprogrammed Device	10, 15, 25 ns	Com'l		90	mA
			7.5 ns			130	mA
			15, 25 ns	Mil/Ind		120	mA
			10 ns			120	mA
I_{CC2}[6]	Operating Power Supply Current	V_{CC} = Max., V_{IL} = 0V, V_{IH} = 3V, Output Open, Device Programmed as a 10-Bit Counter, f = 25 MHz	10, 15, 25 ns	Com'l		110	mA
			7.5 ns			140	mA
			15, 25 ns	Mil/Ind		130	mA
			10 ns			130	mA

Capacitance[6]

Parameter	Description	Test Conditions	Min.	Max.	Unit
C_{IN}	Input Capacitance	V_{IN} = 2.0V @ f = 1 MHz		10	pF
C_{OUT}	Output Capacitance	V_{OUT} = 2.0V @ f = 1 MHz		10	pF

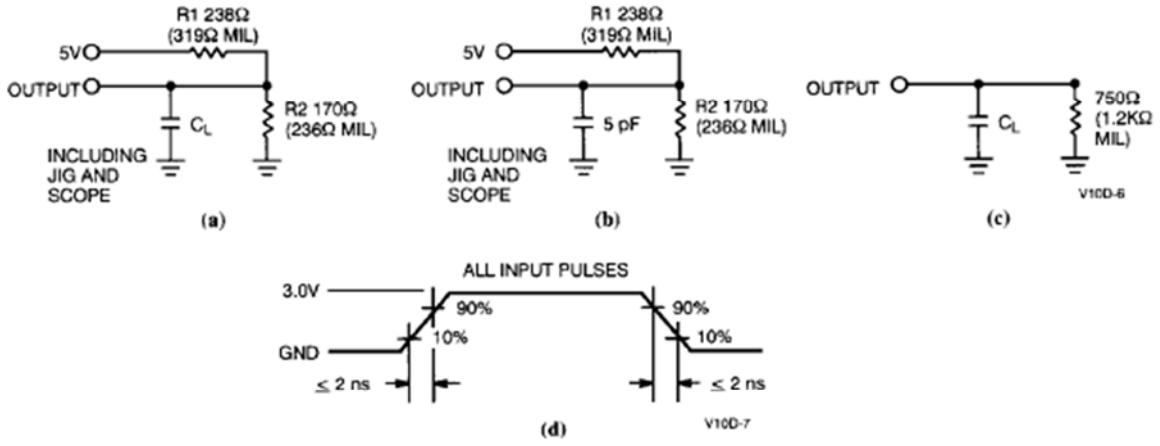
Endurance Characteristics[6]

Parameter	Description	Test Conditions	Min.	Max.	Unit
N	Minimum Reprogramming Cycles	Normal Programming Conditions	100		Cycles

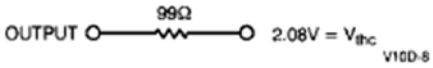
Notes:

1. T_A is the "instant on" case temperature.
2. See the last page of this specification for Group A subgroup testing information.
3. These are absolute values with respect to device ground. All overshoots due to system or tester noise are included.
4. V_{IL} (Min.) is equal to −3.0V for pulse durations less than 20 ns.
5. Not more than one output should be tested at a time. Duration of the short circuit should not be more than one second. V_{OUT} = 0.5V has been chosen to avoid test problems caused by tester ground degradation.
6. Tested initially and after any design or process changes that may affect these parameters.

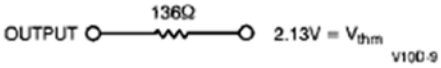
AC Test Loads and Waveforms



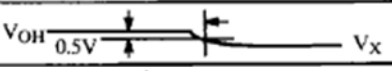
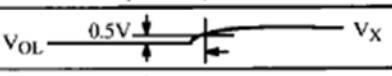
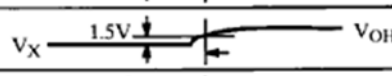
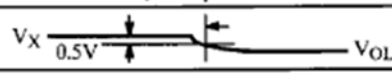
Equivalent to: THÉVENIN EQUIVALENT (Commercial)



Equivalent to: THÉVENIN EQUIVALENT (Military)



Load Speed	C_L	Package
7.5, 10, 15, 25 ns	50 pF	PDIP, CDIP, PLCC, LCC

Parameter	V_X	Output Waveform—Measurement Level
$t_{ER(-)}$	1.5V	
$t_{ER(+)}$	2.6V	
$t_{EA(+)}$	0V	
$t_{EA(-)}$	V_{thc}	

(e) Test Waveforms

Commercial Switching Characteristics (PALC22V10D)[2, 7]

Parameter	Description	22V10D–7		22V10D–10		22V10D–15		Unit
		Min.	Max.	Min.	Max.	Min.	Max.	
t_{PD}	Input to Output Propagation Delay[8, 9]	3	7.5	3	10	3	15	ns
t_{EA}	Input to Output Enable Delay[10]		8		10		15	ns
t_{ER}	Input to Output Disable Delay[11]		8		10		15	ns
t_{CO}	Clock to Output Delay[8, 9]	2	5	2	7	2	8	ns
t_{S1}	Input or Feedback Set-Up Time	5		6		10		ns
t_{S2}	Synchronous Preset Set-Up Time	6		7		10		ns
t_H	Input Hold Time	0		0		0		ns
t_P	External Clock Period ($t_{CO} + t_S$)	10		12		20		ns
t_{WH}	Clock Width HIGH[6]	3		3		6		ns
t_{WL}	Clock Width LOW[6]	3		3		6		ns
f_{MAX1}	External Maximum Frequency ($1/(t_{CO} + t_S)$)[12]	100		76.9		55.5		MHz
f_{MAX2}	Data Path Maximum Frequency ($1/(t_{WH} + t_{WL})$)[6, 13]	166		142		83.3		MHz
f_{MAX3}	Internal Feedback Maximum Frequency ($1/(t_{CF} + t_S)$)[6, 14]	133		111		68.9		MHz
t_{CF}	Register Clock to Feedback Input[6, 15]		2.5		3		4.5	ns
t_{AW}	Asynchronous Reset Width	8		10		15		ns
t_{AR}	Asynchronous Reset Recovery Time	5		6		10		ns
t_{AP}	Asynchronous Reset to Registered Output Delay		12		13		20	ns
t_{SPR}	Synchronous Preset Recovery Time	6		8		10		ns
t_{PR}	Power-Up Reset Time[6, 16]	1		1		1		µs

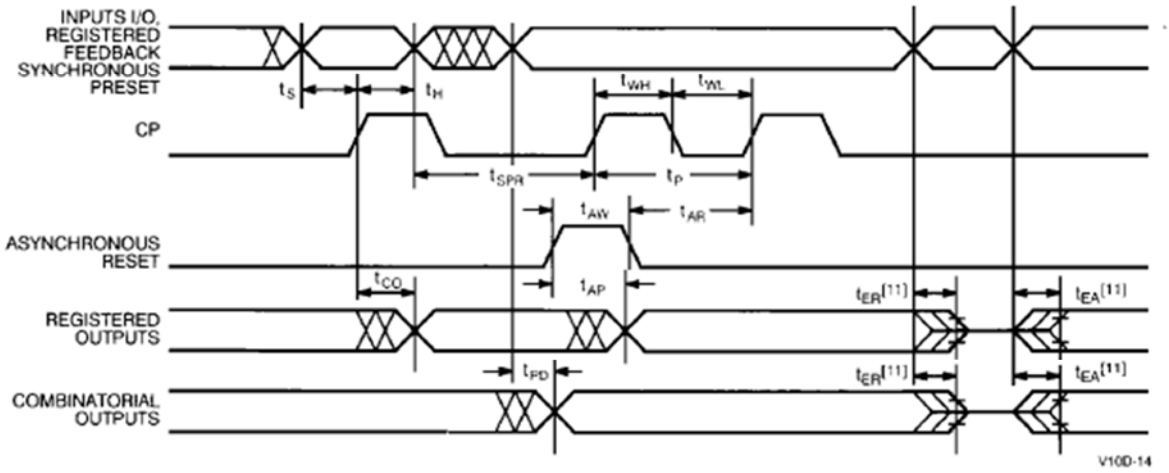
Notes:

7. Part (a) of AC Test Loads and Waveforms is used for all parameters except t_{ER} and $t_{EA(+)}$. Part (b) of AC Test Loads and Waveforms is used for t_{ER}. Part (c) of AC Test Loads and Waveforms is used for $t_{EA(+)}$.
8. Min. times are tested initially and after any design or process changes that may affect these parameters.
9. This specification is guaranteed for all device outputs changing state in a given access cycle.
10. The test load of part (a) of AC Test Loads and Waveforms is used for measuring $t_{EA(-)}$. The test load of part (c) of AC Test Loads and Waveforms is used for measuring $t_{EA(+)}$ only. Please see part (e) of AC Test Loads and Waveforms for enable and disable test waveforms and measurement reference levels.
11. This parameter is measured as the time after output disable input that the previous output data state remains stable on the output. This delay is measured to the point at which a previous HIGH level has fallen to 0.5 volts below V_{OH} min. or a previous LOW level has risen to 0.5 volts above V_{OL} max. Please see part (e) of AC Test Loads and Waveforms for enable and disable test waveforms and measurement reference levels.
12. This specification indicates the guaranteed maximum frequency at which a state machine configuration with external feedback can operate.
13. This specification indicates the guaranteed maximum frequency at which the device can operate in data path mode.
14. This specification indicates the guaranteed maximum frequency at which a state machine configuration with internal only feedback can operate.
15. This parameter is calculated from the clock period at f_{MAX} internal ($1/f_{MAX3}$) as measured (see Note 11 above) minus t_S.
16. The registers in the PALC22V10D have been designed with the capability to reset during system power-up. Following power-up, all registers will be reset to a logic LOW state. The output state will depend on the polarity of the output buffer. This feature is useful in establishing state machine initialization. To insure proper operation, the rise in V_{CC} must be monotonic and the timing constraints depicted in Power-Up Reset Waveform must be satisfied.

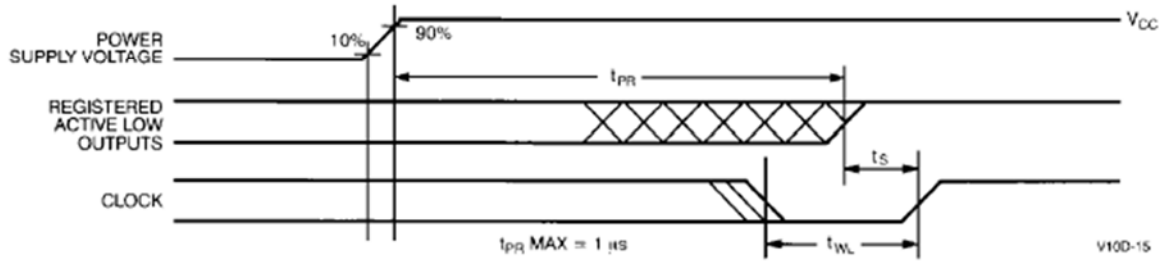
Military and Industrial Switching Characteristics (PALC22V10D)[2, 7]

Parameter	Description	22V10D–10		22V10D–15		22V10D–25		Unit
		Min.	Max.	Min.	Max.	Min.	Max.	
t_{PD}	Input to Output Propagation Delay[8, 9]	3	10	3	15	3	25	ns
t_{EA}	Input to Output Enable Delay[10]		10		15		25	ns
t_{ER}	Input to Output Disable Delay[11]		10		15		25	ns
t_{CO}	Clock to Output Delay[8, 9]	2	7	2	8	2	15	ns
t_{S1}	Input or Feedback Set-Up Time	6		10		18		ns
t_{S2}	Synchronous Preset Set-Up Time	7		10		18		ns
t_H	Input Hold Time	0		0		0		ns
t_P	External Clock Period ($t_{CO} + t_S$)	12		20		33		ns
t_{WH}	Clock Width HIGH[6]	3		6		14		ns
t_{WL}	Clock Width LOW[6]	3		6		14		ns
f_{MAX1}	External Maximum Frequency ($1/(t_{CO} + t_S)$)[12]	76.9		50.0		30.3		MHz
f_{MAX2}	Data Path Maximum Frequency ($1/(t_{WH} + t_{WL})$)[6, 13]	142		83.3		35.7		MHz
f_{MAX3}	Internal Feedback Maximum Frequency ($1/(t_{CF} + t_S)$)[6, 14]	111		68.9		32.2		MHz
t_{CF}	Register Clock to Feedback Input[6, 15]		3		4.5		13	ns
t_{AW}	Asynchronous Reset Width	10		15		25		ns
t_{AR}	Asynchronous Reset Recovery Time	6		12		25		ns
t_{AP}	Asynchronous Reset to Registered Output Delay		12		20		25	ns
t_{SPR}	Synchronous Preset Recovery Time	8		20		25		ns
t_{PR}	Power-Up Reset Time[6, 16]	1		1		1		µs

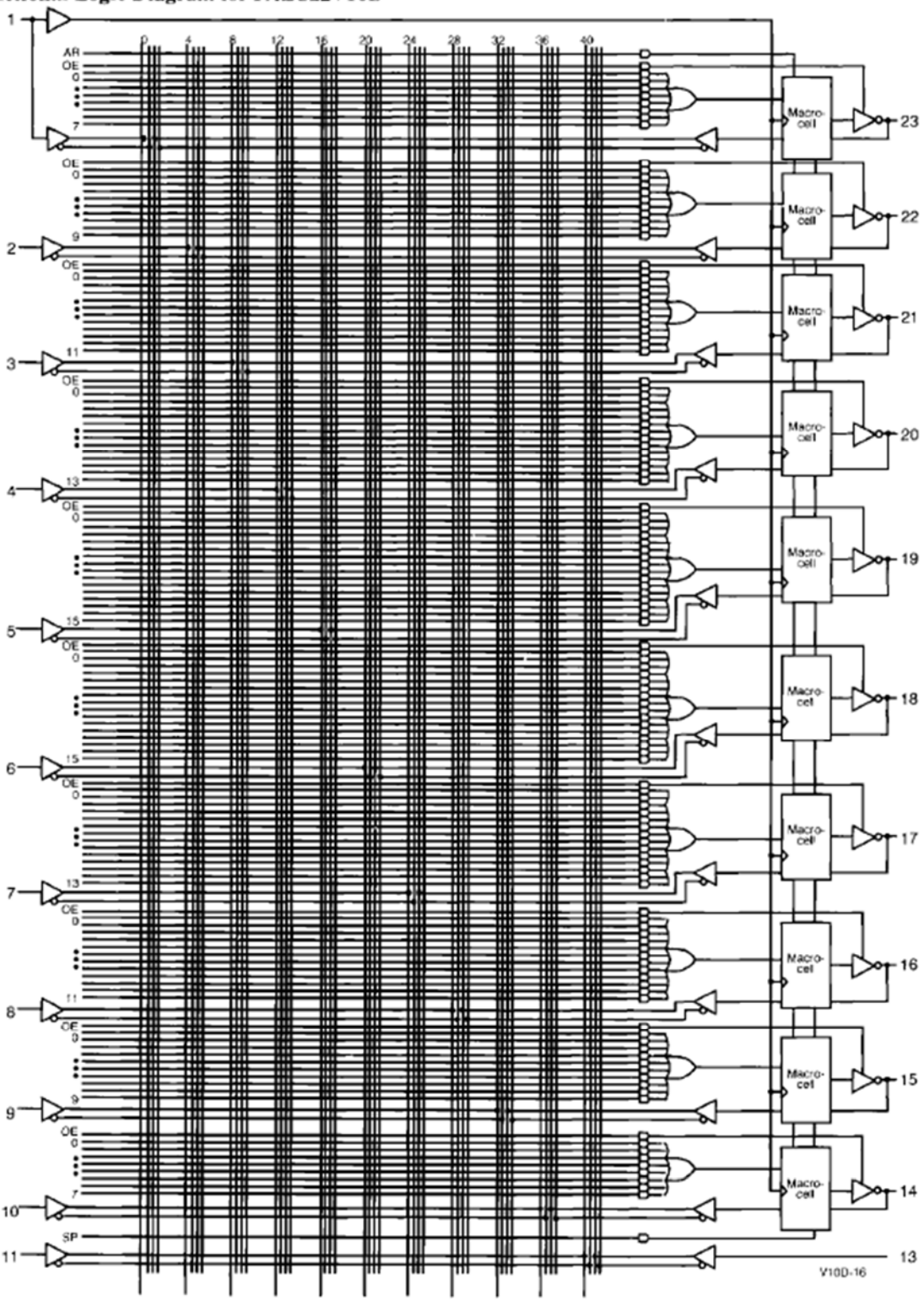
Switching Waveform



Power-Up Reset Waveform[11]



Functional Logic Diagram for PALC22V10D



PLD Design Methodology

Programmable logic devices (PLDs) are used in digital systems design for implementing a wide variety of logic functions. These logic functions range from simple random logic replacement to complex control sequencers. Programmable logic devices offer the multiple advantages of low cost, high integration, ease of use, and easier design debugging capability not available in other systems design options. In the following discussion we will detail the PLD design process.

Most PLDs have an AND-OR array structure with programmable connections in either or both of the arrays. A programmable array implies that the connections can be programmed by the user. The popular PAL (Programmable Array Logic) devices have a programmable AND array and a fixed OR array. PAL devices are used for a wide variety of combinatorial and registered logic functions. In this discussion we will also examine the various design constraints to be considered when selecting the correct architecture for a given application.

All digital logic can be efficiently reduced to two fundamental gates, AND and OR, provided both true and complement versions of all input signals are available. Such logic is generally built around what is known as the sum-of-products (AND-OR) form. Programmable logic devices are ideal for implementing such two-stage logic in the AND and OR arrays.

Various process technologies offer many design options for PLDs. The connections in the programmable arrays can be fuse-based, commonly used in both ECL and TTL bipolar technologies, E/EEPROM cell based in UV-EPROM and EEPROM CMOS technologies, and RAM cell-based in CMOS RAM technology. The selection of technology is mostly dependent upon the system speed and power constraints. Most design engineers are familiar with these constraints, which not only dictate the technology of PLDs but also all of the other logic used in a system.

Designing with PLDs involves the use of design software and a device programmer (Figure 1). The design software eliminates the need to identify every connection to be programmed for implementing the desired sum-of-products logic. The design process begins with the creation of a design file which specifies the desired function. The function is typically represented by its sum-of-products form and can be derived directly from the timing diagram and/or truth tables. Occasionally Karnaugh maps and state diagrams are also used. The design file is then assembled to produce the "JEDEC" file. The JEDEC file gets its name from the fact that it is an approved JEDEC standard for specifying the state of every connection on the device. Simulation can then be performed. If the design is correct, the JEDEC file is downloaded into a device programmer for programming the connections on the device. The device can then be plugged into the PC board where it will function. The entire procedure can often be performed with the designer never having to leave the desk. Most programmers interface to personal computers, so that the design file can be edited, assembled, simulated, and downloaded, and the device programmed, all in one place.

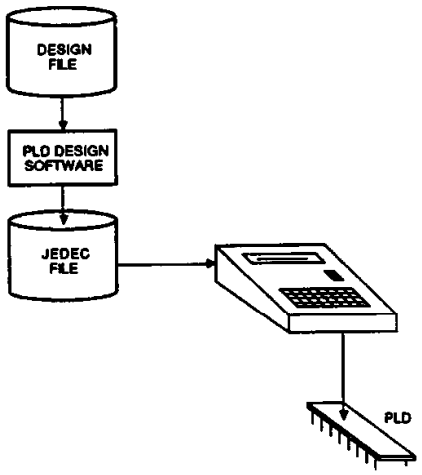


Figure 1. PLDs are Designed Using Software and a Device Programmer

The first stage in a PLD design process (Figure 2) is the conceptualization of a design problem; the second is the selection of the correct device; the third is the implementation of the design, which also includes simulating the design with test vectors; and finally, the

Publication# 90002 Rev. A Amendment/0
Issue Date: June 1993

actual programming and testing on a system board. We will take a simple design example and go through the various stages of this design process.

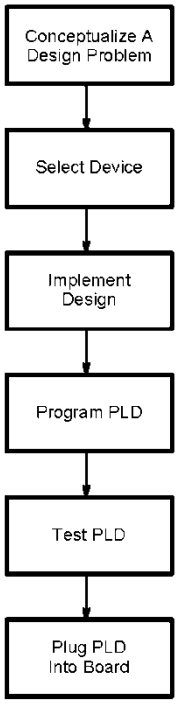


Figure 2. Programmable Logic Device Design Process

Conceptualizing a Design

The first step in the PLD design process is also required for any SSI/MSI design. An advantage of PLDs is that at this stage the designer needs to be concerned only with the required logic function. With SSI or MSI, various device logic limitations must be accounted for before the design can be started. Clearly a designer needs to develop a brief and complete functional description, based upon the system design requirements.

We will take the example of a simple address decoder circuit required for a 68000 microprocessor. The microprocessor has 24 address lines along with separate read and write signals. It requires some ROM to store the boot-up code as well as some RAM for storing and executing programs. The purpose of the address decoder circuitry is to select one of the memory addresses at a time. The RAMs and ROMs are assigned addresses on the 68000 microprocessor address space. The Address decoder circuit has to select one of the RAMs or ROMs for a specific range of addresses, called the address space. This selection is accomplished by asserting the specific chip-select signal for the RAM or ROM when the microprocessor accesses one of the addresses in the address space. There is additional circuitry in a typical microprocessor system for addressing I/O devices (such as disk controllers). These devices also require that chip-select signals be asserted when the microprocessor addresses them. Figure 3 shows an example address map for a 68000 microprocessor.

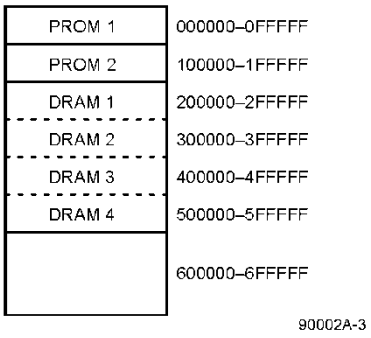


Figure 3. Memory Address Map

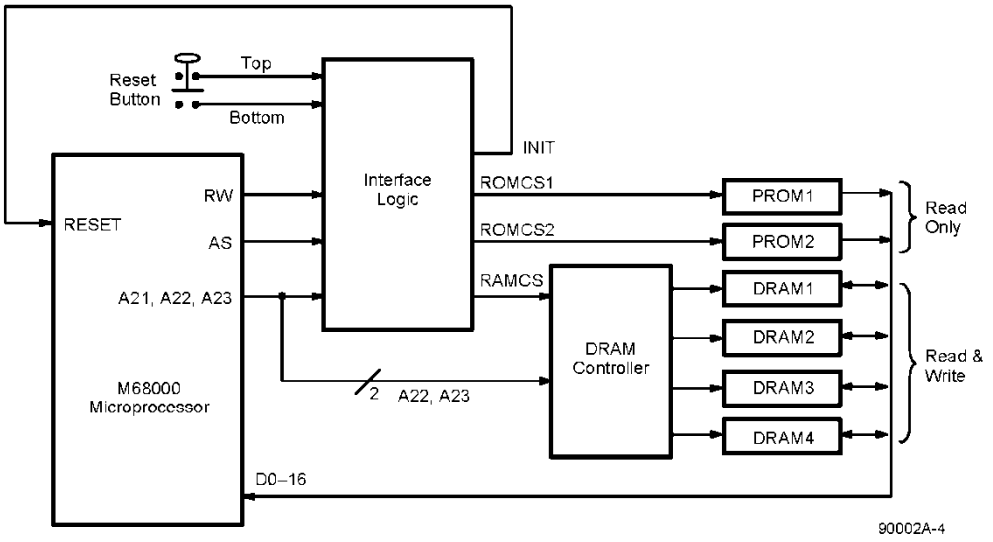


Figure 4. Microprocessor to Memory Interface

Figure 4 show the circuit diagram. The address signals from the 68000 microprocessor are inputs to the interface logic block. The outputs generated are ROMCS1, ROMCS2 and RAMCS. The generation of signals for selecting device I/Os is similar and is not shown here for the sake of simplicity. Other system inputs to the interface are the address strobe signal generated by the 68000 microprocessor as well as the read/write signal. The truth table for generating the outputs is shown in Table 1. This truth table is derived from the memory address map and the functional description of the design.

Table 1. Truth Table for Chip-Select Signals

Addresses Hex	Size	A23	A22	A21	Signal
000000–0FFFFF	1 MB	0	0	0	$\overline{\text{ROMCS1}}$
100000–1FFFFF	1 MB	0	0	1	$\overline{\text{ROMCS2}}$
200000–2FFFFF	1 MB	0	1	0	$\overline{\text{RAMCS}}$
300000–3FFFFF	1 MB	0	1	1	$\overline{\text{RAMCS}}$
400000–4FFFFF	1 MB	1	0	0	$\overline{\text{RAMCS}}$
500000–5FFFFF	1 MB	1	0	1	$\overline{\text{RAMCS}}$

Device Selection Considerations

The first task for the designer is to identify the design problem and classify it as a combinatorial function or a registered function, depending upon whether or not registers are required. In most cases, this decision depends upon the functional nature of the problem. Sometimes timing and logic considerations can also dictate the use of registers; this will be discussed later. Registers are usually not required for such simple combinatorial functions such as encoders, decoders, multiplexers, demultiplexers, adders, and comparators. However, registers are required for functions such as counters, timers, control signal generation, and state machines. No registers are required for this simple address decoding example.

The best choice for our combinatorial design would be a PAL device. The task now is to select a PAL device for implementing the desired function. General device selection considerations are listed below. These items are applicable to most designs.

- Number of input pins
- Number of output pins
- Number of I/O pins
- Device speed
- Device power requirements
- Number of registers (if any)
- Number of product terms
- Output polarity control

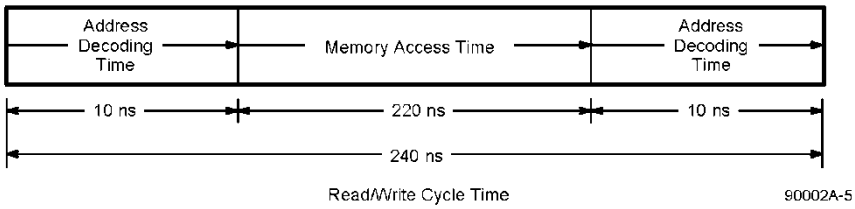


Figure 5. System Timing Requirements

The first resource that must be provided in a PLD is the number of pins needed for the basic logic function. This consists of the number of input and output pins. Many PLDs have internal feedback, which allows the generated output signal to be reused as an input. The same feedback also allows the pin to be used as a dedicated input, if required. This is especially useful for fitting various designs with different input/output requirements on the same device. The I/O pin capability of certain PLDs can also be very useful for certain bus applications.

The task is as simple as counting the number of input, output and I/O pins required by the design and picking a PLD which has the requisite number of pins.

The next selection issue is the device speed. The most important timing consideration for combinatorial PLDs is the propagation delay (t_{PD}) of signals from the input to the output of the device. For registered PLDs, the important timing consideration is the device clocking frequency. This clocking frequency is in turn determined by sum of the register setup time (t_S), and clock-to-output propagation delay (t_{CO}). Most systems impose some timing restrictions on the internal logic functions. These restrictions will determine the necessary t_{PD} (for combinatorial devices) or f_{MAX} (for registered devices).

In our design example, the PLD will primarily perform address decoding. The critical system timing constraint is determined by the read/write cycle time of the microprocessor and the memory access time available (Figure 5). Most microprocessors allow anywhere from 10 to 35 ns for address decoding. That is, 10 ns – 35 ns after the address is available, the correct memory chip-select signal should be asserted. In our design example, the available cycle time of 240 ns and memory access time of 220 ns leaves barely 10 ns for address decode time. We can check the propagation delay and select the appropriate speed device for our design, which is t_{PD} = 10 ns.

We have already briefly discussed the types of applications where registers are needed. Sometimes the consideration of system timing can affect whether or not registers are needed. Devices with registers can hold a signal stable for the long durations required by the addressed peripheral or memory. However, this slows the initial response or access time of the device since the chip select must wait for the setup time before the rising edge of the clock cycle. Devices without registers provide fast access time but hold the signal valid only as long as the input conditions are valid. In most address decoders, the address signals are kept asserted by the microprocessor until the read/write cycle is completed. In this case, the registers are not required for holding the signals asserted.

The remaining two general design considerations are the number of product terms and output polarity. We will discuss these two as we implement the design in the next section.

Implementing a Design

Implementing a design (Figure 6) requires the creation of a design file. The design file contains three types of information.

- Basic bookkeeping information
- Design syntax
- Simulation syntax

Once the design file is complete, it is then assembled and simulated. Once it passes assembly and simulation, the resultant JEDEC file is downloaded to a device programmer for configuring the device.

Design Syntax

In this example, as shown in Figure 6, there are two options available to the designer for expressing the design. The first is through traditional Boolean logic equations; the second is through a state machine syntax. The Boolean logic equations are the only option for combinatorial designs and can also be efficient for some registered designs. The Boolean equations can be derived from a combination of the functional description, the truth table and/or the timing diagrams (Figure 7). The state machine approach is ideal for large registered control designs, and can be derived from the functional description, state table, state diagram and/or the timing diagram (Figure 8).

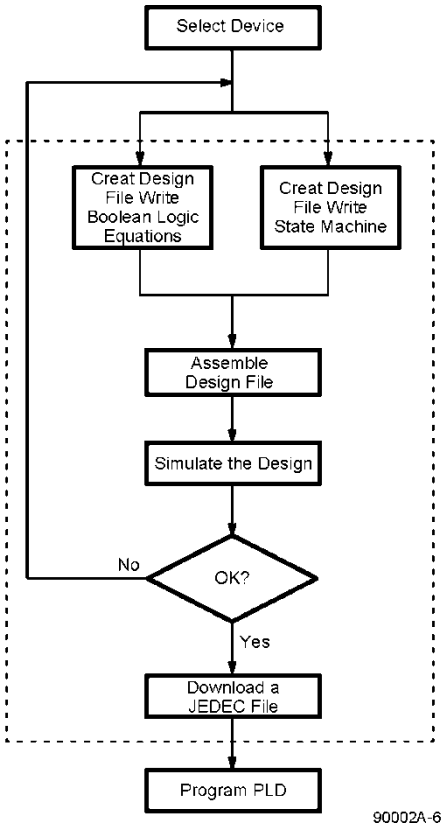


Figure 6. Implementing a Design

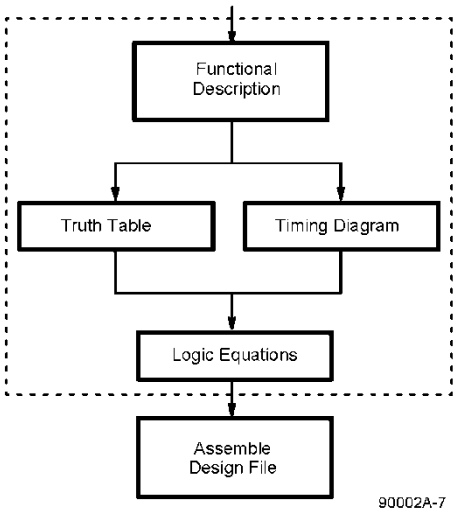


Figure 7. Writing Boolean Logic Equations

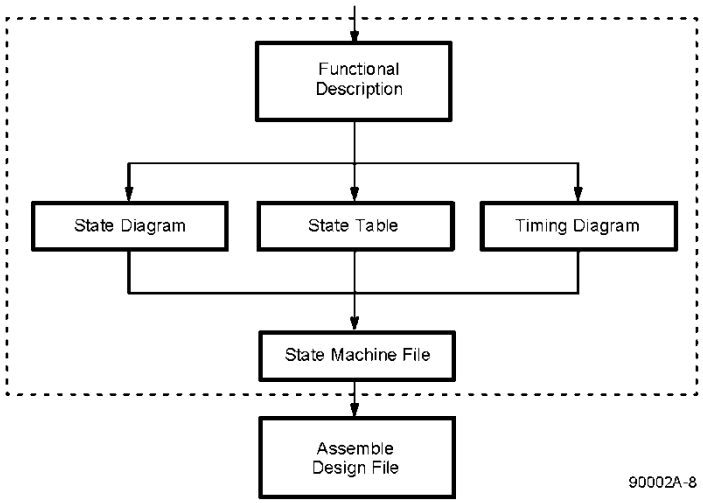


Figure 8. State Machine Description

Boolean Logic Equations

Boolean equations are used to represent the sum-of-products logic form. The Boolean equations are ideally suited for representing the two-level AND-OR logic available in most PLDs.

A conventional approach to the design is to convert the design problem to its discrete logic implementation. Such random SSI and MSI logic can be easily implemented in PLDs. This usually involves converting to sum-of-products Boolean logic form. This approach can be a chore, and much effort can be saved by implementing a design with PLDs in a sum-of-products form right from the start. This essentially means that the designer does not have to design around the limitations of fixed SSI and MSI functions. A direct implementation of a design in sum-of-products form in a PLD can also yield a faster circuit.

Boolean equations can be directly derived from the truth table or timing diagram (Figure 7). The truth table is used more often in simple combinatorial designs. The timing diagram method is used more often in registered control designs. We will first discuss the truth table method and then discuss the details of the timing diagram method.

In addition to specifying the logic function, the Boolean equations in the design file help document the design. There is no need to draw out an equivalent schematic. This allows design modularity; the schematic can just show a block for a particular PLD. Separate supporting documentation (the design file) provides the details without cluttering the drawing.

Truth-Table-Based Design

The requirements for our particular design example can be easily converted to a truth table format (Table 2). This truth table is based upon the functional description of the design, and is derived from the address map (Figure 3) and the truth table (Table 1).

Table 2. Truth Table for the Address Decoder

						Output Generated		
A23	A22	A21	INIT	AS	RW	ROMCS1	ROMCS2	RAMCS
0	0	0	1	0	1	0	1	1
0	0	1	1	0	1	1	0	1
0	1	0	1	0	X	1	1	0
0	1	1	1	0	X	1	1	0
1	0	0	1	0	X	1	1	0
1	0	1	1	0	X	1	1	0

There are three additional input signals in this design example. The first, RW, is generated by the microprocessor, and distinguishes between read and write cycles. Since the ROM data is only for reading, the ROMCS1 and ROMCS2 signals are asserted only when RW is high (when the microprocessor attempts to read the ROM) and are not asserted for the write cycle. On the other hand, RAMCS is generated for both read and write cycles and the state of signal RW is "don't care."

The second additional signal, AS, is the address strobe signal generated by the microprocessor, and is asserted only when the address lines carry a valid address. All of the chip select signals need to be gated with the AS signal to ensure that they are only generated for valid addresses, and no spurious chip selects are generated.

The last signal is the INIT signal, which is a system initialization signal. This signal is used to initialize the microprocessor for a "warm boot," and none of the chip selects is allowed when this INIT signal is asserted.

Writing Boolean equations from the above logic is very straight forward. The output signal names, along with their polarity, are assigned to sum-of-product equations, which are based upon inputs and their polarities.

```
/ROMCS1 = /A23 * /A22 * /A21 * INIT * /AS * RW
/ROMCS2 = /A23 * /A22 *  A21 * INIT * /AS * RW
/RAMCS  = /A23 *  A22 * /A21 * INIT * /AS
        + /A23 *  A22 *  A21 * INIT * /AS
        +  A23 * /A22 * /A21 * INIT * /AS
        +  A23 * /A22 *  A21 * INIT * /AS
```

Figure 9. The Implementation In Boolean Equations

The equations are derived directly from the truth tables. Each one of the AND equations uses up one product term of the device as shown in Figure 9. One device selection consideration is to ensure that all the outputs have sufficient product terms to accommodate the desired function.

This brings us to the issue of output polarity. Suppose we had to generate active-HIGH outputs. In that case the output equations for the ROMCS1 signal would be:

```
ROMCS1 = /A23 + /A22 * /A21 * INIT */AS = RW
```

If the device has active-LOW outputs only, this equation's output polarity needs to be inverted to be able to fit the device. Using DeMorgan's theorem for Boolean logic we get:

```
/ROMCS1 = A23 - A22 + A21 + /INIT + AS + /RW
```

This equation requires a large number of product terms (six). Some signals are efficient and use fewer product terms in their true form, while others are more efficient in their inverted form. The device selection issues of product terms and output polarity also apply to registered designs.

Timing-Diagram-Based Design

Until now, we have discussed a PLD design using truth tables as the primary design vehicle. In this section we will attempt a design using a timing diagram as a design vehicle.

Earlier in the address decoder design we mentioned the INIT signal. This INIT signal essentially an initialization signal for the entire system. The INIT signal is used internally (via feedback) for disabling the chip selects during initialization. Externally it can be used to initialize

other system signals. This INIT signal is generated from a RESET switch connected to the inputs of the device as shown in Figure 10.

Most experienced designers understand the tradeoffs for device selection. They implicitly go through the steps of design conceptualization and device selection, explained earlier. They typically draw a block around the logic being designed, with the previous knowledge that it would fit a PLD which has sufficient inputs, outputs, IOs and product terms.

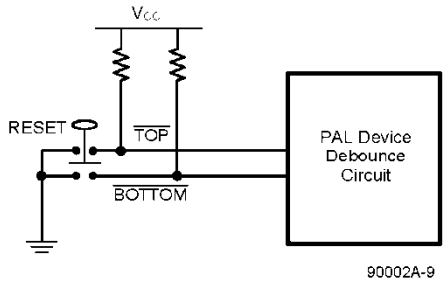


Figure 10. RESET Switch for System Initialization

To avoid unwanted initialization, the RESET switch must be debounced. That is, we want the INIT signal to remain HIGH until the switch actually contacts the bottom side. Once the bottom side is hit, INIT should be asserted active LOW. Once asserted, it should stay LOW and not change until the top side is hit again. The timing requirements of the debounce circuitry are shown in Figure 11. Signals TOP and BOTTOM are inputs to the programmable logic device. These signals are activated when the RESET switch touches the top and the bottom contacts, respectively.

We can formulate the equations by looking at the timing requirements of the debounce circuitry shown in Figure 11. The idea is to identify the key elements of this timing diagram. The arrows in Figure 11 show the critical events. The first arrow shows the normal state of all the pins when the RESET switch is not asserted. Subsequent arrows show each event in the timing of the INIT signal, depending upon the movement of the switch.

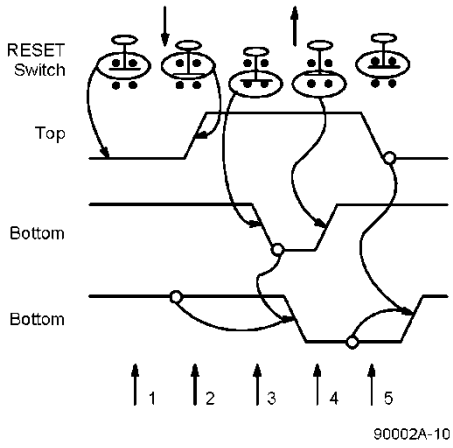


Figure 11. Timing Diagram for the Debounce Switch

The logic level of the signals at each critical event carries useful logic information for deriving Boolean equations. This logic information for each event is converted into direct Boolean equations as shown in below. For example, at instant 1 the INIT signal remains HIGH as long as the TOP signal remains LOW; this is converted to INIT = /TOP * BOTTOM.

1. Normal state	`INIT = /TOP`
2. Switch travels from TOP to BOTTOM	`INIT = TOP * BOTTOM * INIT`
3. Switch contacts BOTTOM	`/INIT = /BOTTOM`
4. Switch travels from BOTTOM to TOP	`/INIT = /INIT * BOTTOM * TOP`
5. Normal State Again	

We can combine the two active-LOW events into one equation:

```
/INIT = /BOTTOM
      + /INIT * BOTTOM * TOP
```

Minimizing, this becomes:

```
/INIT = /BOTTOM
      - / INIT = TOP
```

This can also be done by way of a truth table and Karnaugh map.

Table 3. Truth Table of INIT Logic

TOP	BOTTOM	INIT–	INIT+
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	1
0	0	1	X
0	0	0	X

Here TOP or BOTTOM will be LOW if contacted. Note that both TOP and BOTTOM can not be contacted at the same time. The truth table of Table 3 yields the Karnaugh map shown in Figure 12. Grouping the zeros (because we are using active-LOW outputs) yields the Boolean equation identical to the one derived from the timing diagram.

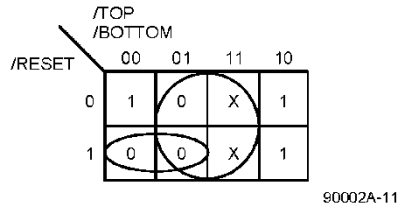


Figure 12. Karnaugh Map of INIT Signal Logic

There is essentially no difference between the truth table and timing diagram techniques for writing Boolean logic. Also, a careful analysis will indicate that we implicitly assumed a truth table in the timing diagram example. Some designers prefer to make a separate truth table (at least in the first few PLD designs), while others prefer to design directly from timing diagrams. While the truth table method allows a more optimal utilization of product terms, the timing diagram method is easier to visualize as it retains the design perspective. In both cases the logic should be minimized by the design software to ensure that the design is testable.

Most experienced designers understand the tradeoffs for device selection. They implicitly go through the steps of design conceptualization and device selection, explained earlier. They typically draw a block around the logic being designed, with the previous knowledge that it would fit a PLD which has sufficient inputs, outputs, IOs and product terms.

Simulation

Design simulation is an integral part of the design process, as shown in Figure 13. The purpose is to exercise all of the inputs and test the response of outputs to verify that they will work as desired in the system. These are essentially test vectors which designate the state of every input on the device; the outputs are then checked for an appropriate response. The simulation test vectors identify any flaws in the design equations which could affect the logical operation of the devices programmed. Thus, the simulation vectors serve as a design debugging tool.

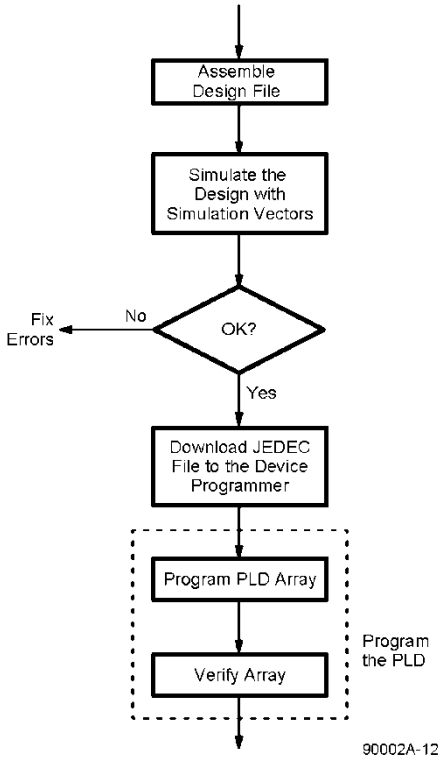


Figure 13. Device Simulation and Programming

Simulation test vectors will eventually make up part of a larger set of test vectors called "functional test vectors". These functional test vectors are used to exercise a real device after programming to identify any individual devices which are defective. Other means of identifying defective devices, such as signature analysis, are also available. In this section we will strictly focus on simulation vectors.

Simulation is included in the design file along with the logic equations. There is little standardization in these simulation expressions among various PLD design software packages, although most of them rely on test vectors to exercise the logic.

The simulation vectors or events can be directly derived from the truth table and the timing diagram of the design. The logic level and functions of all signals can be expanded and rewritten in a test vector form by the software. For example, the truth table for the address decoder example discussed earlier can be easily rewritten as shown in Table 4.

Table 4. Truth Table Used to Derive Simulation Vectors

A23	A22	A21	TOP	BOTTOM	AS	RW	ROMCS1	ROMCS2	RAMCS	INIT
0	0	0	0	1	1	1	H	H	H	H
0	0	0	0	1	0	1	L	H	H	H
0	0	1	0	1	1	1	H	H	H	H
0	0	1	0	1	0	1	H	L	H	H
0	1	0	0	1	1	X	H	H	H	H
0	1	0	0	1	0	X	H	H	L	H
0	1	1	0	1	1	X	H	H	H	H
0	1	1	0	1	0	X	H	H	L	H
1	0	0	0	1	1	X	H	H	H	H
1	0	0	0	1	0	X	H	H	L	H
1	0	1	0	1	1	X	H	H	H	H
1	0	1	0	1	0	X	H	H	L	H
1	0	1	0	1	X	X	H	H	H	H
1	0	1	1	1	X	X	H	H	H	H
1	0	1	1	0	1	X	H	H	H	L
1	0	1	1	1	1	X	H	H	H	L
1	0	1	0	1	1	X	H	H	H	H

These are essentially the simulation vectors which will allow us to define the inputs to the device and check the outputs of the device.

The simulator then interprets the design file and generates the output logic levels and/or waveforms, which can be checked by the designer.

Once the simulation is complete, the design file can be assembled to generate the JEDEC file. In the proceeding discussions we have assumed prior knowledge of the design file assembly. The procedure for assembly varies with different software packages.

Device Programming and Testing

Once the design simulation is completed, the final step is device programming and testing (Figure 14). Programmers are available from a variety of vendors. It is important to note that Advanced Micro Devices, Inc., qualifies programmers upon verifying that the algorithms used by the programmers are correct and that other basic criteria are met. When purchasing a programmer, check that the programmer is qualified for the devices you intend to use.

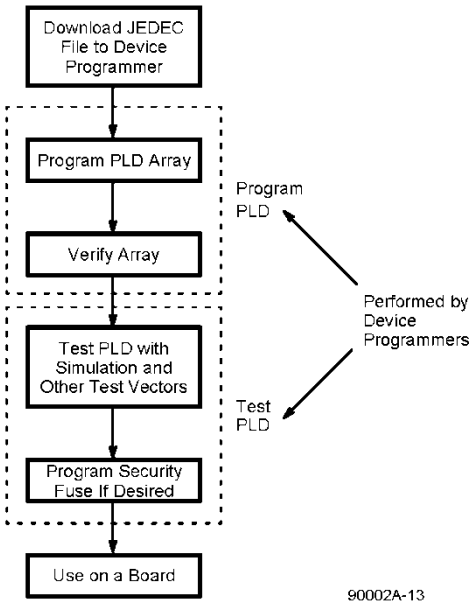


Figure 14. Device Programming and Testing

There are two types of programmers available: menu-driven or device code based. The menu-driven programmer directly indicates the part type being programmed, whereas the latter type requires the user to enter the device code before programming.

Once the JEDEC fuse file has been downloaded, the programmer can program the device; the PLD is then ready for use. The programmer also verifies the connections after the programming cycle. Programmers also provide the capability of reading a previously programmed device and creating duplicates of that device.

Testing PLDs

The testing of PLDs can be performed by the device programmer or by other test equipment. For a manufacturing environment, where high yields are required, device testing is critical. After testing is complete, the device security bit may be programmed, if desired, to secure the design from copying.

State Machine Design

INTRODUCTION

State machine designs are widely used for sequential control logic, which forms the core of many digital systems. State machines are required in a variety of applications covering a broad range of performance and complexity; low-level controls of microprocessor-to-VLSI-peripheral interfaces, bus arbitration and timing generation in conventional microprocessors, custom bit-slice microprocessors, data encryption and decryption, and transmission protocols are but a few examples.

Typically, the details of control logic are the last to be settled in the design cycle, since they are continuously affected by changing system requirements and feature enhancements. Programmable logic is a forgiving solution for control logic design because it allows easy modifications to be made without disturbing PC board layout. Its flexibility provides an escape valve that permits design changes without impacting time-to-market.

A majority of registered PAL device applications are sequential control designs where state machine design techniques are employed. As technology advances, new high-speed and high-functionality devices are being introduced which simplify the task of state machine design. A broad range of different functionality-and-performance solutions are available for state machine design. In this discussion we will examine the functions performed by state machines, their implementation on various devices, and their selection.

What Is a State Machine?

A state machine is a digital device that traverses through a predetermined sequence of states in an orderly fashion. A state is a set of values measured at different parts of the circuit. A simple state machine can consist of PAL-device based combinatorial logic, output registers, and buried (state) registers. The state in such a sequencer is determined by the values stored in the buried and/or output registers.

A general form of a state machine can be depicted as a device shown in Figure 1. In addition to the device inputs and outputs, a state machine consists of two essential elements: combinatorial logic and memory (registers). This is similar to the registered counter designs discussed previously, which are essentially simple state machines. The memory is used to store the state of the machine. The combinatorial logic can be viewed as two distinct functional blocks: the next state decoder and the output decoder (Figure 2). The next state decoder determines the next state of the state machine while the output decoder generates the actual outputs. Although they perform two distinct functions, these are usually combined into one combinatorial logic array as in Figure 1.

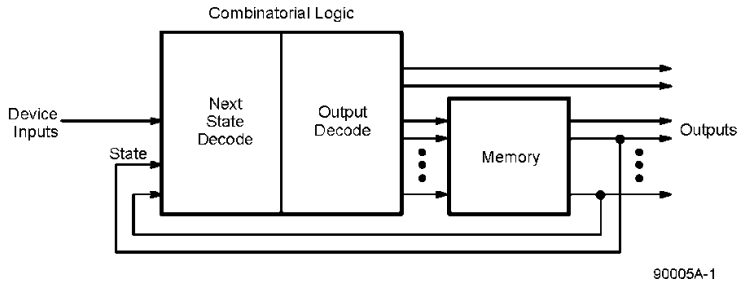


Figure 1. Block Diagram of a Simple State Machine

Publication# 90005 Rev. A Amendment/0
Issue Date: June 1993

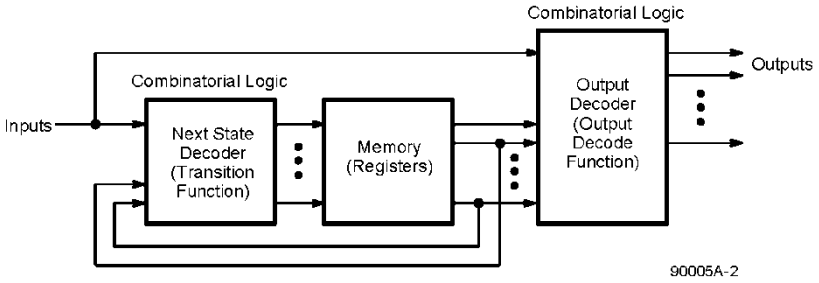


Figure 2. State Machine, with Separate Output and Next State Decoders

The basic operation of a state machine is twofold:

1. It traverses through a sequence of states, where the next state is determined by next state decoder, depending upon the present state and input conditions.
2. It provides sequences of output signals based upon state transitions. The outputs are generated by the output decoder based upon present state and input conditions.

Using input signals for deciding the next state is also known as branching. In addition to branching, complex sequencers provide the capability of repeating sequences (looping) and subroutines. The transitions from one state to another are called *control sequencing* and the logic required for deciding the next states is called the *transition function* (Figure 2).

The use of input signals in the decision-making process for *output generation* determines the type of a state machine. There are two widely known types of state machines: Mealy and Moore (Figure 3). Moore state machine outputs are a function of the present state only. In the more general Mealy-type state machines, the outputs are functions of both the state and the input signals. The logic required is known as the *output function*. For either type, the control sequencing depends upon both states and input signals.

Most practical state machines are synchronous sequential circuits that rely on clock signals to trigger the state transitions. A single clock is connected to all of the state and output edge-triggered flip-flops, which allows a state change to occur on the rising edge of the clock. Asynchronous state machines are also possible, which utilize the propagation delay in combinatorial logic for the memory function of the state machine. Such machines are highly susceptible to hazards, hard to design and are seldom used. In our discussion we will focus solely on sequential state machines.

State Machine Applications

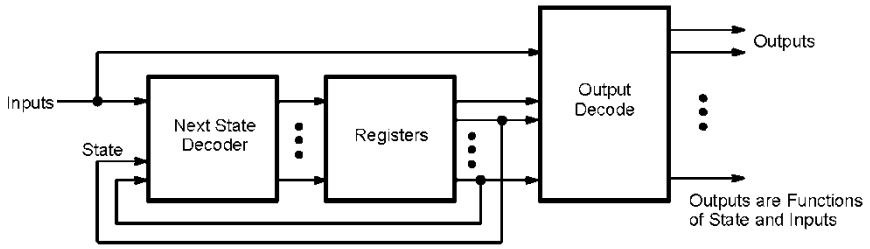
State machines are used in a number of system control applications. A sampling of a few of the applications, and how state machines are applied, is described below.

As sequencers for digital signal processing (DSP) applications, state machines offer speed and sufficient functionality without the overkill of complex microprocessors. For simple algorithms, such as those involved in performing a Fast Fourier Transform (FFT), a state machine can control the set of vectors that are multiplied and added in the process. For complex DSP operations, a programmable DSP may be better. On the other hand, the programmable DSP solution is not likely to be as fast as the dedicated hardware approach.

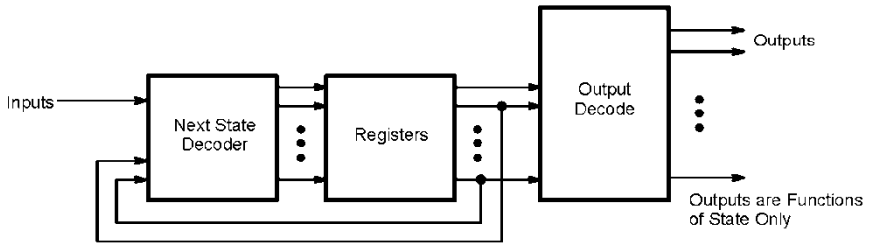
Consider the case of a video controller. It generates addresses for scanning purposes, using counters with various sequences and lengths. Instead of implementing these as actual counters, the sequences involved can be "unlocked" and implemented, instead, as state machine transitions. There is an advantage beyond mere economy of parts. A count can be set or initiated, then left to take care of itself, freeing the microprocessor for other operations.

In peripheral control the simple state machine approach can be very efficient. Consider the case of run-length-limited (RLL) code. Both encoding and decoding can be translated into state machines, which examine the serial data stream as it is read, and generate the output data.

Industrial control and robotics offer further areas where simple control functions are required. Such tasks as mechanical positioning of a robot arm, simple decision making, and calculation of a trigonometric function, usually does not require the high-power solution of microprocessors with stacks and pointers. Rather, what is required is a device that is capable of storing a limited number of states and allows simple branching upon conditions.



a. Mealy State Machines



b. Moore State Machines

90005A-3

Figure 3. The Two Standard State Machine Models

Data encryption and decryption present similar problems to those encountered in encoding and decoding for mass media, only here it is desirable to make the scheme not so obvious. A programmable state machine device with a security Bit is ideal for this because memory is internally programmed and cannot be accessed by someone tampering with the system.

Functions Performed

All the system design functions performed by controllers can be categorized as one of the following state machine functions:

- Arbitration
- Event monitoring
- Multiple condition testing
- Timing delays
- Control signal generation

Later we will take a design example and illustrate how these functions can be used when designing a state machine.

State Machine Theory

Let us take a brief look at the underlying theory for all sequential logic systems, the *finite state machine* (FSM), or simply state machine.

Those parts of digital systems whose outputs depend on their past inputs as well as their current ones can be modeled as finite state machines. The "history" of the machine is summed up in the value of its internal state. When a new input is presented to the FSM, an output is generated which depends on this input and the present state of the FSM, and the machine is caused to move into new state, referred to as the next state. This new state also depends on both the input and present state. The structure of an FSM is shown pictorially in Figure 2. The internal state is stored in a block labeled "memory." As discussed earlier, two combinatorial functions are required: the transition function, which generates the value of the next state, and the output function, which generates the state machine output.

State Diagram Representation

The behavior of an FSM may be specified in graphical form as shown in Figure 4. This is called a state diagram, or state transition diagram. Each bubble represents a state, and each arrow represents a transition between states. Inputs that cause the transitions are shown next to each transition arrow.

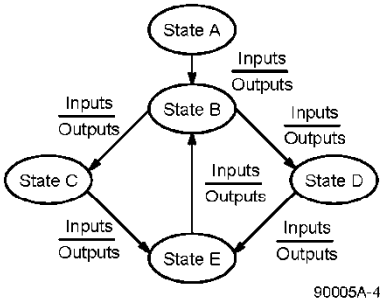


Figure 4. State Machine Representation

Control sequencing is represented in the state transition diagram as shown in Figure 5. Direct control sequencing requires an unconditional transition from state A to state B. Similarly conditional control sequencing shows a conditional transition from state C to either state D or state E, depending upon input signal I1.

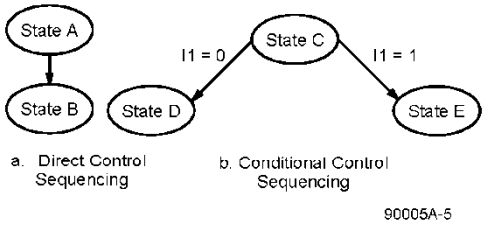


Figure 5. Control Sequencing

For Moore machines the output generation is represented by assigning outputs with states (bubbles) as shown in Figure 6. Similarly, for Mealy machines conditional output generation is represented by assigning outputs to transitions (arrows), as was shown in Figure 4. More detail on Mealy and Moore output generation is given later.

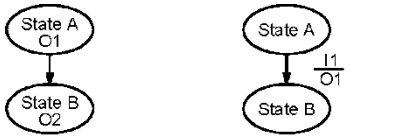


Figure 6. Output Generation

For this notation, there is a specification uncertainty as to which signals are outputs or inputs, as they both occur on the drawing next to the arrow in which they are active. This is usually resolved by separating the input and output signals names with a line (Figures 4 and 6). Sometimes an auxiliary pin list detailing the logic polarity and input or output designations is also used.

State transition diagrams can be made more compact by writing on the transitions not the input values that cause the transition, as in Figure 4, but a Boolean expression defining the input combination or combinations that cause this transition. For example, in Figure 7, some transitions have been shown for a machine with inputs START, X1, and X2. In the transition between states 1 and 2, the inputs X1 and X2 are ignored (that is, they are "don't cares") and thus do not appear on the diagram. This saves space and makes the function more obvious.

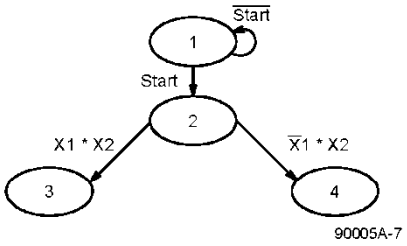


Figure 7. State Transition Diagram with Mnemonics

There can be a problem with this method if one is careless. The state transitions in figure 8 show what can happen. There are three input combinations, (I0, I1, I2, I3) = {1011}, {1101} and {1111}, which make both (/I0 * /I2 + I3) and (I0 * I1 + I0 * I2) true. Since a transition to two next states is impossible, this is an error in the

specification. It must either be guaranteed that these input combinations never occur, or the transition conditions must be modified. In this example, changing (I0 * I1 + I0 * I2) to (I0 * I1 + I0 * I2) * /I3 would solve the problem.

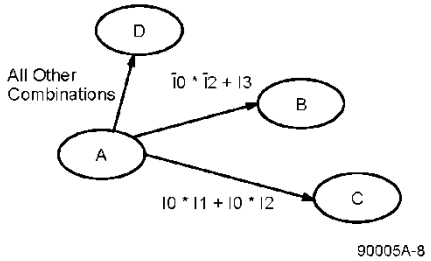


Figure 8. State Diagram with Conflicting Branch Conditions

State Transition Table Representation

A second method for state machine representation is the tabular form known as the state transition table, which has the format shown in Table 1. Listed along the top are all the possible input bit combinations and internal states. Each row gives the next state and the next output; thus, the table specifies the transition and output functions. However, this type of table is not suitable for specifying practical machines in which there is a large number of inputs, since each input combination defines a row of the table. For example, with 10 inputs, 1024 rows would be required!

Table 1. A State Transition Table

Present State	Inputs	Next State	Outputs Generated
S0 – Sn	I0 – Im	S0 – Sn	O0 – Op

Flowcharts

Another popular notation is based on flowcharts. In this notation, states are represented by rectangular boxes, and alternative state transitions are determined by strings of diamond-shaped boxes. The elements may have multiple entry points, but in general have only one exit. The state name is written as the first entry in the rectangular state box. Any Moore outputs present are written next in the state box, with a caret (^) following those that are unregistered. The state code assignment, if it is known, is written next to the upper right corner of the state box. Decision boxes are diamond or hexagonal shaped boxes containing either an input signal or a logic expression. Two exits labeled "0" and "1" lead to either another decision box, a state box, or a Mealy output. The rounded oval is used for Mealy machine outputs. Again, a caret follows those outputs that are unregistered. All the boxes may need to be expanded to accommodate a number of output signals or a larger expression.

The use of these symbols is shown in Figure 9. Each path, through the decision boxes from one state to another defines a particular combination or set of combinations of the input variables. A path does not have to include all input variables; thus, it accommodates "don't cares." These decision trees take more space than the expressions would, but in many practical cases, state machine controllers only test a small subset of the input variables in each state and the trees are quite manageable. Also, the chain of decisions often mirrors the designer's way of thinking about the actions of the controller. It is important to note that these tests are not performed sequentially in the FSM; all are performed in parallel by the FSM's state transition logic.

A benefit of this method of specifying transitions is that the problem of Figure 8 can be avoided. Such a conflict would be impossible as one path cannot diverge to define paths to two states.

This flowchart notation can be compacted by allowing more complex decisions, when there is no danger of conflicts due to multiple next states being defined. Expressions can be tested, as shown in Figure 10a, or multiple branches can extend from a decoding box, as in Figure 10b. In the second case, it is convenient to group the set of binary inputs into a vector, and branch on different values of this vector.

The three methods of state machine representation state diagrams, state tables, and flowcharts

are all equivalent and interchangeable, since they all describe the same hardware structure. Each style has its own particular advantages. Although most popular, the state transition diagrams are more complex for problems where state transitions depend on many inputs, since the transition conditions are written directly on the transition arrows. Although cumbersome, the state tables allow the designer tight control over signal logic. Flowcharts are convenient for small problems where there are not more than about ten states and where up to two or three inputs or input expressions are tested in each state. For larger problems, they can become ungainly.

Once a state machine is defined, it must be implemented on a device. Software packages are then used to implement the design on a device. The task is to convert the state machine description into transition and output functions. Software packages also account for device-specific architectural variations and limitations, to provide a uniform user interface.

Some software packages accept all three different state machine representations directly as design inputs. However, the most prevalent design methodology is to convert the three state machine design representations to a simple textual representation. Textual representations are accepted by most software packages although the syntax varies.

Since the most common of all state machine representations is the state transition diagram representation, we will use it in all subsequent discussions. Transition table and flowchart representation implementations will be very similar.

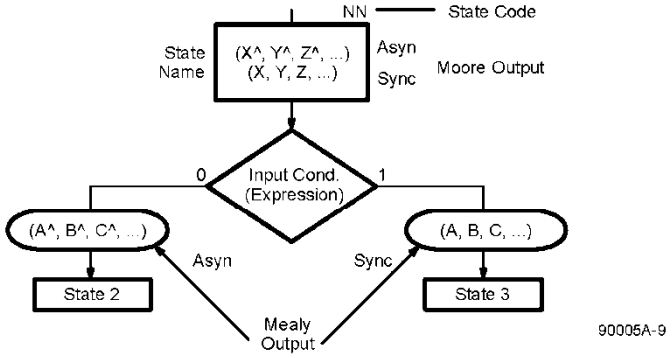
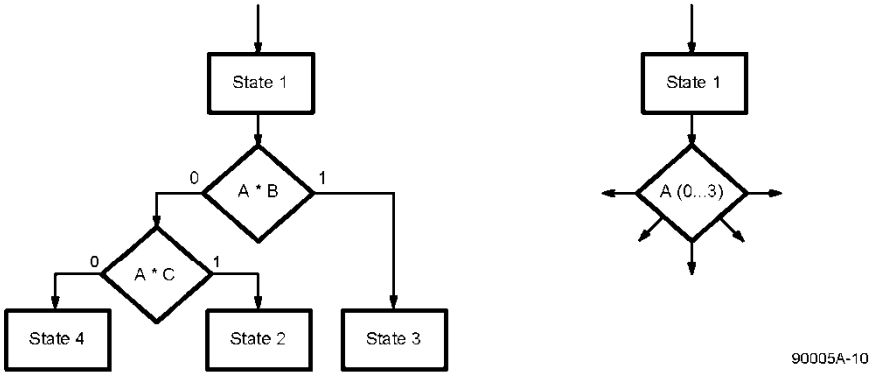


Figure 9. Flowchart Notation



a. Testing Expressions **b. Multiway Branch**

Figure 10. Using Flowcharts

State Machine Types: Mealy & Moore

With the state machine representation clarified, we can now return to the generic sequencer model of Figure 1, which has been labeled (Figure 11) to show the present state (PS), next state (NS), and output (OB, OA). This will illustrate how Mealy and Moore machines are implemented with most sequencer devices that provide a single combinatorial logic array for both next state and output decode functions. There are four ways of using the sequencer, two of which implement Moore machines and two Mealy. First, let us look at the Mealy forms.

The standard Mealy form is shown in Figure 12, where the signals are labeled as in Figure 11 to indicate which registers and outputs are used. The register outputs PS are fed back into the array and define the present state. The combinatorial logic implements the transition function, which produces the next state flip-flop inputs NS, and the output function, which produces the machine output OB. This is the asynchronous Mealy form.

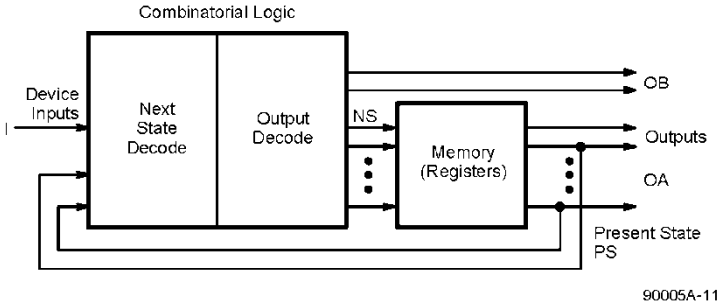


Figure 11. Generic Model of an FSM

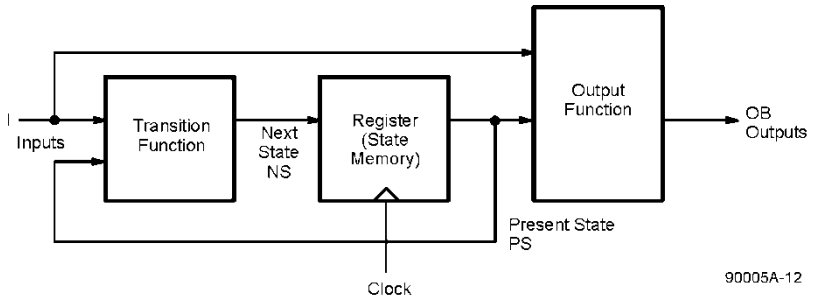


Figure 12. Asynchronous Mealy Form

An alternative Mealy form is shown in Figure 13. Here the outputs are passed through an extra output register (OA) and thus, do not respond immediately to input changes. This is the synchronous Mealy form.

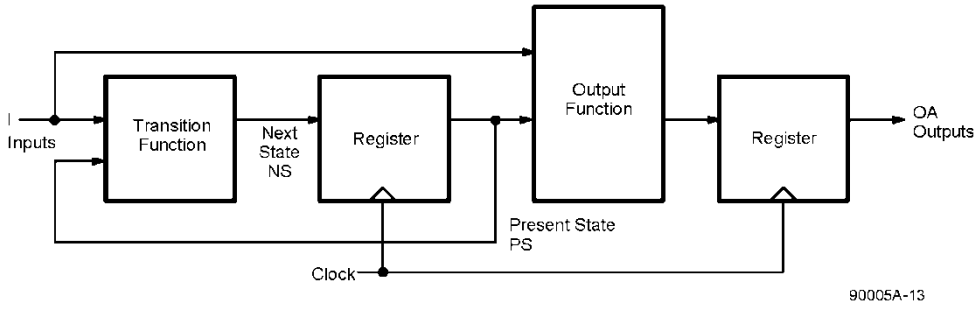


Figure 13. Synchronous Mealy Form

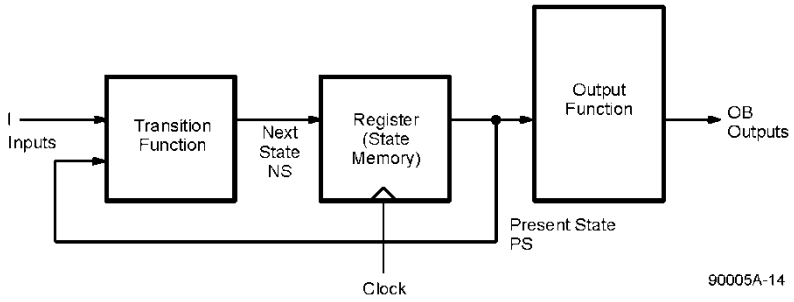


Figure 14. Asynchronous Moore Form

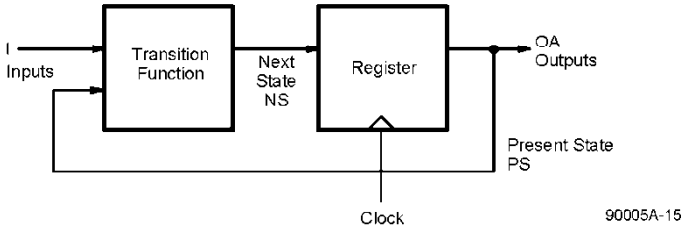


Figure 15. Synchronous Moore Form

The standard Moore form is given in Figure 14. Here the outputs OB depend only on the present state PS. This is the asynchronous Moore form. The synchronous Moore form is shown in Figure 15. In this case the combinatorial logic can be assumed to be the unity function. The outputs (OB) can be generated directly along with the present state (PS). Although these forms have been described separately, a single sequencer is able to realize a machine that combines them, provided that the required paths exist in the device.

In the synchronous Moore form, the outputs occur in the state in which they are named in the state transition diagram. Similarly, in the asynchronous Mealy and Moore forms the outputs occur in the state in which they are named, although delayed a little by the propagation delay of the output decoder. This is because they are combinatorial functions of the state (and inputs in the Mealy case).

However, the synchronous Mealy machine is different. Here an output does not appear in the state in which it is named, since it goes into another register first. It appears when the machine is in the next state, and is thus delayed by one clock cycle. The state diagram in Figure 16 illustrates all the possibilities on a state transition diagram.

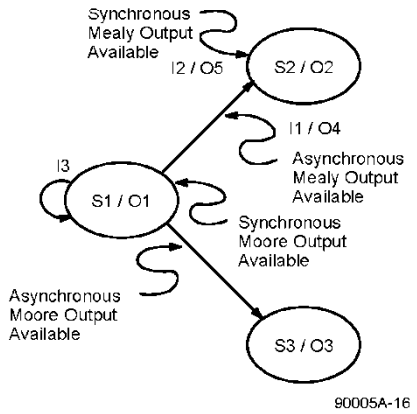


Figure 16. State Diagram Labelling for Different Output Types

As a matter of notation, Moore outputs are often placed within the state bubble and Mealy outputs are placed next to the path or arrow that activates them.

The relationship of Mealy and Moore, synchronous and asynchronous outputs to the states is shown in Figure 17.

Device Selection Considerations

There are three major criteria for selecting the correct state machine device for a design:

- Number of inputs/outputs
  - I/O flexibility
  - Number of output registers
- Speed
- Intelligence/functionality
  - Number of product terms
  - Type of flip-flops
  - Number of state registers

Number of I/Os

The number of inputs, outputs and I/O pins determine the signals that can be sampled or generated by a state machine.

Timing and Speed

The timing considerations for sequencer design are similar to those for registered logic design. A system clock cycle forms the basic kernel for evaluating control function behavior. For the most part, all input and output functions are specified in relationship to the positive edge. Registered outputs are available after a period of time t_{CO}, the clock-to-output propagation delay. Asynchronous outputs require an additional propagation delay (t_{PD}) before they are valid.

For the circuit to operate reliably, all the flip-flop inputs must be stable at the flip-flop by the minimum set-up time (t_S) of the flip-flops before the next active clock edge. If one of the inputs changes after this threshold, then the next state or synchronous output could be stored incorrectly; the circuit may even malfunction. To avoid this, the clock period (t_A) must be greater than the sum of the set-up time of the flip-flops and the clock to output time ($t_S + t_{CO}$). This determines the minimum clock period and hence the maximum clock frequency, fMAX, of the circuit. Metastability and erroneous system operation may occur if these specifications are violated.

The timing relationships are shown in Figure 18. In each cycle there are two regions: the stable region, when all signals are steady, and the transition region, when the machine is changing state and signals are unstable. The active clock edge causes the flip-flops to load the value of the new state that has been set up at their inputs.

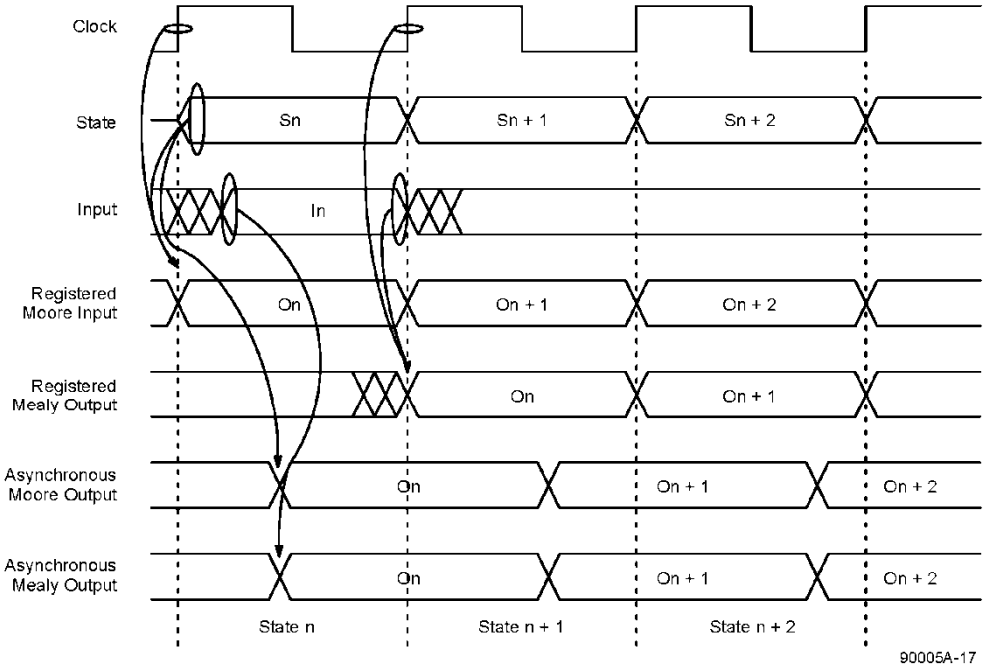


Figure 17. State Machine Timing Diagram

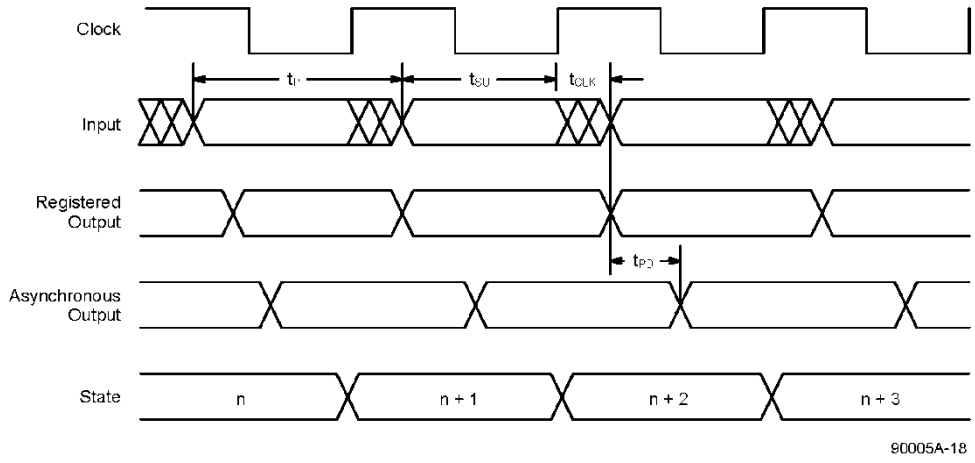


Figure 18. Timing Diagram for Maximum Operating Frequency

At a time after this, the present state and output flip-flop outputs will start to change to their new values. After a time has elapsed, the slowest flip-flop output will be stable at its new value. Ignoring input changes for the moment, the changes in the state register cause the combinatorial logic to start generating new values for the asynchronous outputs and the inputs to the flip-flops. If the propagation delay of the logic is t_{PD}, then the stable period will start at a time equal to the sum of the maximum values of t_{CO}, and t_{PD}.

Asynchronous Inputs

The timing of the inputs to an asynchronous state machine is often beyond the control of the designer and may be random, such as sensor or keyboard inputs, or they may come from another synchronous system that has an unrelated clock. In either case no assumptions can be made about the times when inputs can or cannot arrive. This fact causes reliability problems that cannot be completely eliminated, but only reduced to acceptable levels.

Figure 19 shows two possible transitions from state "S1" (code 00) either back to itself, or to state "S2" (code 11). Which transition is taken depends on input variable "A" which is asynchronous to the clock. The transition function logic for both state bits B1 and B2 include this input. The input A can appear in any part of the clock cycle. For the flip-flops to function correctly, the logic for B1 and B2 must stabilize correctly before the clock. The input should be stable in a window t_S (setup time) before the clock and t_H (hold time) after the clock. If the input changes within this window, both the flip-flops may not switch, causing the sequence to jump to states 01 or 10, which are both undefined transitions. This type of erroneous behavior is called an input race.

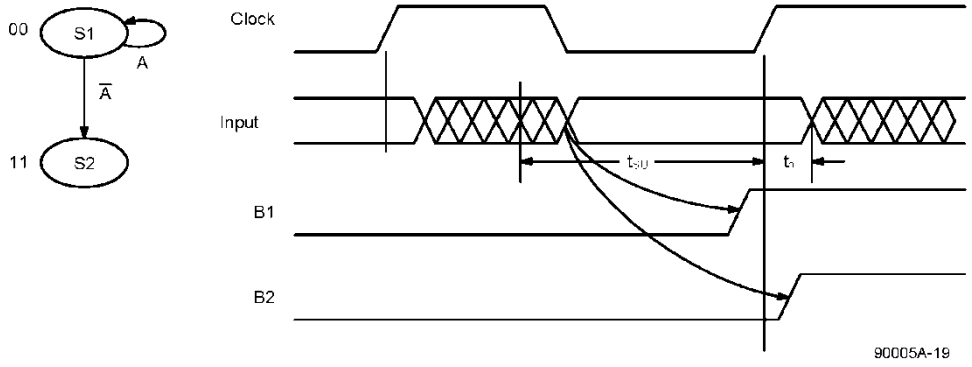


Figure 19. Asynchronous Input Cascading Race

A solution to this problem is to change the state assignment so that only one state variable depends on the asynchronous input. Thus, the 11 code must be changed to 01 or 10. Now, with only one unsynchronized flip-flop input, either the input occurs in time to cause the transition, or it does not, in which case no transition occurs. In the case of a late input, the machine will respond to it one cycle later, provided that the input is of sufficient duration.

There is still the possibility of an input change violating the setup time of the internal flip-flop, driving it into a metastable state. This can produce system failures that can be minimized, but never eliminated. The same problem arises when outputs depend on an asynchronous input.

Very little can be done to handle asynchronous inputs without severely constraining the design of the state machine. The only way to have complete freedom in the use of inputs is to convert them into synchronous inputs. This can be done by allocating a flip-flop to each input as shown in Figure 20. These synchronizing flip-flops are clocked by the sequencer clock, and may even be the sequencer's own internal flip-flops. This method is not foolproof, but significantly reduces the chance of metastability occurring.

Functionality

The functionality of different devices is difficult to compare since different device architectures are available. The number of registers in a device determines the number of state combinations possible. However, all the possible state combinations are not necessarily usable, since other device constraints may be reached. The number of registers does give an idea of the functionality achievable in a device. Other functionality measures include the number of product terms and type of flip-flop. One device may be stronger than another in one of these measures, but overall may be less useful due to other shortcomings. Choosing the best device involves both skill and experience.

In order to give an idea of device functionality, we will consider each of the architecture options available to the designer and evaluate its functionality.

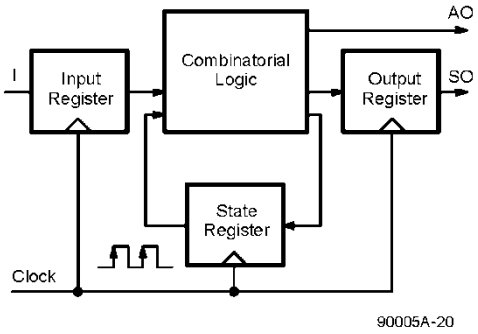


Figure 20. Input Synchronizing Register

PAL Devices as Sequencers

A vast majority of state machine designs are implemented with PAL devices. Early versions of software required the user to manually write the sum-of-products Boolean equations for using PAL devices. Second generation software allows one to specify the design in "state machine syntax," and handles the translation to sum-of-products logic automatically. PAL devices implement the output and transition functions in sum-of-products form through a user-programmable AND array and a fixed OR array.

PAL devices deliver the fastest speed of any sequencer and are ideally suited for simple control applications characterized by few input and output signals interacting within a dedicated controller in a sequential manner. The number of flip-flops in a typical PAL device range from 8 to 12, which offer potentially more than one thousand state values. Since some of the flip-flops are used for outputs, and the number of product terms is limited, the usable number of states is reduced drastically. Generally, up to about 35 states can be utilized.

PAL Device Flip-Flops

PAL device based sequencers implement small state machine designs, which have a relatively large number of output transitions. Since the output registers change with most state transitions, they can be used simultaneously as state registers, once the state values are carefully selected. Most PAL devices are used for small state machines, and efficiently share the same register for output and state functions. High-functionality PAL device based sequencers provide dedicated buried state registers when sharing is difficult.

As a state machine traverses from one state to another, every output either makes a transition (changes logic level) or holds (stays at the same logic level). Small state machine designs require relatively more transitions and fewer holds. As designs get larger, state machines statistically require relatively fewer transitions and more holds.

Most PAL devices provide D-type output registers. D-type flip-flops use up product terms only for active transitions from logic LOW to HIGH level, and for holds for logic HIGH level only. J-K, S-R, and T-type flip-flops use up product terms for both LOW-to-HIGH and HIGH-to-LOW transitions, but eliminate hold terms. Thus, D-type flip-flops are more efficient for small state machine designs. Some PAL devices offer the capability of configuring the flip-flops as J-K, S-R or T-types, which are more efficient for large state machine designs since they require no hold terms.

Many examples of PAL-device-based sequencers can be found in system time base functions, special counters, interrupt controllers, and certain types of video display hardware.

PAL devices are produced in a variety of technologies for multiple applications, and provide a broad range of speed-power options.

Registered Logic Design

INTRODUCTION

In the previous section we discussed combinatorial designs, circuits whose outputs are totally independent of any system clock. In this section we will discuss sequential circuits, where outputs store their previous values until a new clock is applied. The storage elements which retain the previous output values are called flip-flops. A bank of these flip-flops forms a register, although individual flip-flops are often called registers.

Before we discuss purely registered designs, let us take a look at designs which combine both registered and combinatorial portions. Registered and combinatorial outputs are often mixed on a single device. There can be two distinct designs, one registered and one combinatorial (often glue logic) combined on a single device for higher integration. There may also be a design requirement where registered outputs need to be decoded using combinatorial logic.

There are a number of devices which provide both registered and combinatorial outputs. Most devices provide programmable register bypass, which allows outputs to be programmed as registered or combinatorial.

In most design software packages, the output registers are signified by the ":=" assignment symbol, as opposed to the "=" sign for a combinatorial output. This helps to easily identify registers in each equation. In devices which provide outputs configurable as either registered or combinatorial, this sign is also used by the software to configure the outputs.

General Device Selection Considerations

The same set of general device selection considerations discussed in the PLD design methodology section apply to registered designs. The list of items which must be considered is repeated in Figure 1 for convenience. A device can be conveniently selected based upon the specific input and output requirements.

- Number of input pins
- Number of output pins
- Number of I/O pins
- Device speed
- Device power requirements
- Number of registers
- Number of product terms
- Output polarity control

Figure 1. General Device Selection Considerations

Maximum Frequency

For registered designs, speed is a parameter which needs careful consideration. Most combinatorial designs use the propagation delay (t_{PD}) for ensuring that enough time is allowed for the data from the inputs to appear at the outputs. In registered designs the effects of the clock must be taken into account. This is reflected in the maximum frequency (f_{MAX}) parameter. The flexibility inherent in PLD design provides a choice of configurations from which different f_{MAX} parameters can be calculated.

In the first type of design, the PLD is used for a standalone registered design. In order to decide the next logic level of the registers, the present logic level needs to be available at the inputs of the registers before they are clocked (Figure 2.) Under these conditions the clock period is limited by the internal delay from the flip-flop outputs through the internal feedback and logic to the flip-flops inputs. This f_{MAX} is designated "f_{MAX} *internal*." A simple internal counter is a good example of this type of design, therefore, this parameter is sometimes called "f_{CNT}."

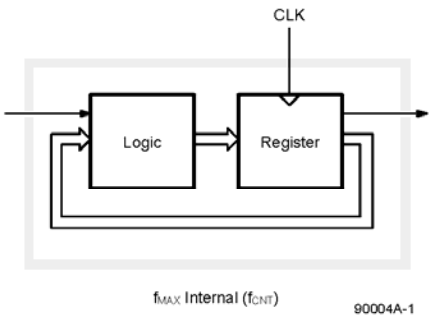


Figure 2. Internal f_{MAX}

Publication# 90004 Rev. A Amendment/0
Issue Date: February 1996

The second type of system configuration is when a number of logic devices with registers, including PLDs, are clocked with a common clock. This is probably the most prevalent configuration. In this case, the registered outputs are sent off-chip back to the device inputs or to the inputs of a second device. The slowest path defining the period (Figure 3) is the sum of the clock-to-output time and the input setup time for the external signals (t_S+t_{CO}). The reciprocal, f_{MAX}, is the maximum frequency with external feedback or in conjunction with an equivalent speed device. This f_{MAX} is designated "f_{MAX} *external.*"

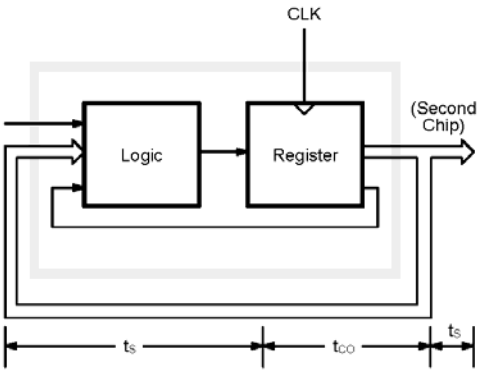


Figure 3. External f_{MAX}

The third type of design is a simple data path application. In this case, input data is presented, to the flip-flop and clocked; no feedback is employed (Figure 4). In this case, the period is limited by the sum of data setup time and data hold time (t_S+t_H). However, the minimum clock period (t_{WH} + t_{WL}) is usually a stricter limit. Thus, the third f_{MAX} designated "f_{MAX} *no feedback*" will be the lesser of $1/(t_S + t_H)$ or $1/(t_{WH} + t_{WL})$.

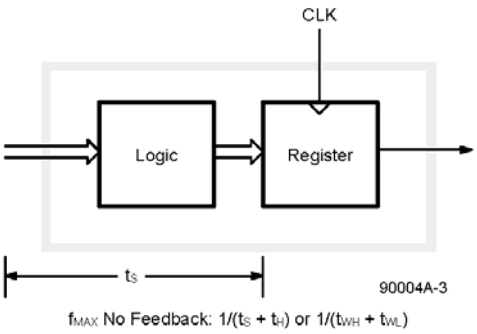


Figure 4. f_{MAX} with No Feedback

f_{MAX} external and f_{MAX} no feedback are calculated parameters. f_{MAX} internal is measured.

Flip-Flop Types

There are four basic types of flip-flops; S-R, J-K, T and the popular D-type. These flip-flops are described in the "PLD Design Basics" section of this data book.

Almost all registered PLDs provide the basic D-type flip-flops. D-type flip-flops are the simplest to design with and will be used throughout this section. Some PLDs provide the capability of configuring output registers as either D, T, J-K or S-R. Configurable flip-flops in some cases can reduce the number of product terms required for certain designs. The effect of the configurable flip-flops will be discussed wherever relevant.

Synchronous vs. Asynchronous

Registered designs can be easily classified into two categories; synchronous and asynchronous. In synchronous designs the clock inputs of all the registers are tied together to a common clock. With asynchronous designs, the flip-flops' clock inputs may not be tied together, and the clocks may be gated or even driven by other flip-flops. We will first discuss synchronous registered designs and then asynchronous registered designs.

Synchronous Registered Designs

Synchronous registered designs are used for two major functions: data handling and control. Registered synchronous designs for data handling include counters and shift registers. There are various types of counters. Some are; binary counters, modulo counters, Johnson counters, and Gray-code counters. These counters are differentiated by the sequence of values through which the counter travels. A binary counter is the simplest form of a counter, and is used most often for data functions. Any system requiring a regular count uses a binary counter. Modulo, Gray-code, and Johnson counters are also used for control.

All counters are actually subsets of a larger class of digital designs called state machines. State machines are discussed in detail in the next chapter of this handbook.

Counters

Counters are the most commonly used sequential circuits. A set of registers, that cycles through a predetermined, unvarying sequence, is called a counter. A general model of a synchronous counter is illustrated in Figure 5. This shows a common clock to all the flip-flops, whose outputs are fed back to a combinatorial logic array called the next-state (count) decoder. The next count is generated by this logic based upon the present count and control inputs. Most PLDs use the standard sum-of-products form of array for this logic.

The relationship between a four-bit counter and its signal timing diagram is illustrated in Figure 6. The counters can also be represented by state diagrams (Figure 7). The state diagrams are bubble-and-arrow diagrams. Each bubble represents a count value and each arrow a transition from one count to the next. More detail on state diagrams is given in the next chapter on state machine design. For counters, the state diagrams are a convenient representation tool and will be used in the discussion when necessary.

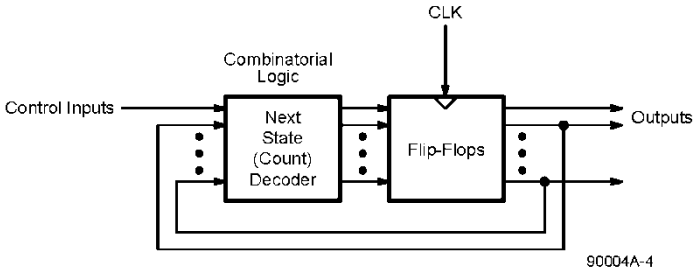


Figure 5. General Model of a Counter

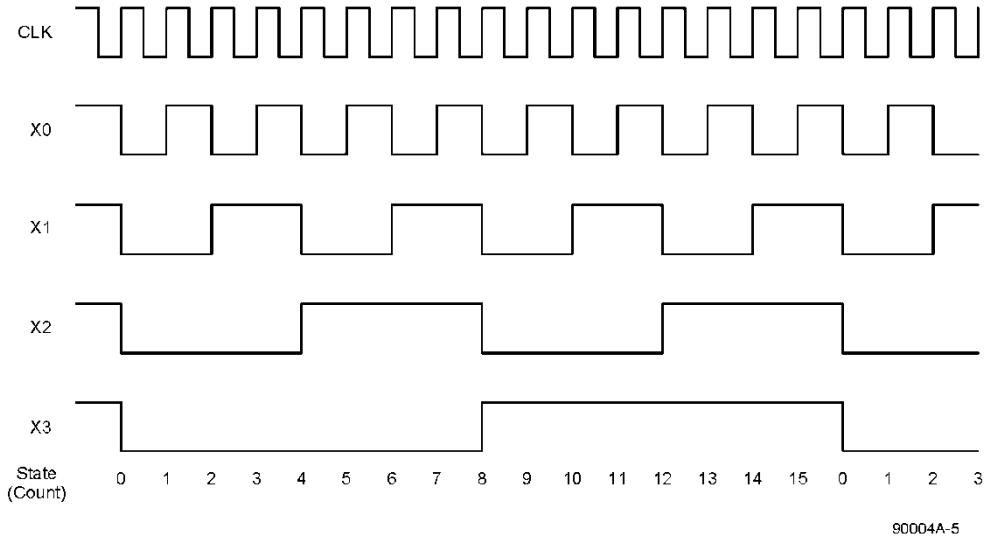


Figure 6. Timing Diagram of a Four-Bit Binary Counter

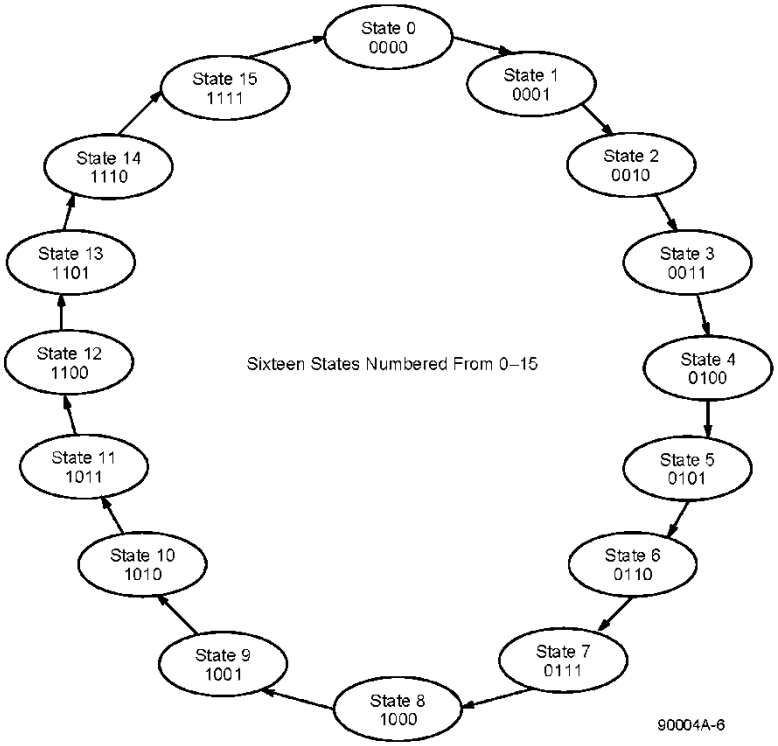


Figure 7. State Diagram of a Four-Bit Binary Counter

Binary Counters

Let us examine a four-bit binary counter. The truth table (also called the transition table) for such a counter is given in Table 1. The table lists the next state values of all the output registers based upon their present values.

Table 1. The Truth Table for a Four-Bit Binary Counter

Present State				Next State			
X3	X2	X1	X0	X3	X2	X1	X0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	0	0	0	0

We derive Boolean equations for each bit directly from the above truth table by collecting all the product terms where outputs are asserted HIGH (ones). This yields:

```
X3  :=     /X3 *  X2 *  X1 *  X0
       +    X3 * /X2 * /X1 * /X0
       +    X3 * /X2 * /X1 *  X0
       +    X3 * /X2 *  X1 * /X0
       +    X3 * /X2 *  X1 *  X0
       +    X3 *  X2 * /X1 * /X0
       +    X3 *  X2 * /X1 *  X0
       +    X3 *  X2 *  X1 * /X0

X2  :=     /X3 * /X2 *  X1 *  X0
       +   /X3 *  X2 * /X1 * /X0
       +   /X3 *  X2 * /X1 *  X0
       +   /X3 *  X2 *  X1 * /X0
       +    X3 * /X2 *  X1 *  X0
       +    X3 *  X2 * /X1 * /X0
       +    X3 *  X2 * /X1 *  X0
       +    X3 *  X2 *  X1 * /X0

X1  :=     /X3 * /X2 * /X1 *  X0
       +   /X3 * /X2 *  X1 * /X0
       +   /X3 *  X2 * /X1 *  X0
       +   /X3 *  X2 *  X1 * /X0
       +    X3 * /X2 * /X1 *  X0
       +    X3 * /X2 *  X1 * /X0
       +    X3 *  X2 * /X1 *  X0
       +    X3 *  X2 *  X1 * /X0

X0  :=     /X3 * /X2 * /X1 * /X0
       +   /X3 * /X2 *  X1 *  X0
       +   /X3 * /X2 * /X1 *  X0
       +   /X3 *  X2 *  X1 *  X0
       +    X3 * /X2 * /X1 * /X0
       +    X3 * /X2 *  X1 *  X0
       +    X3 *  X2 * /X1 *  X0
       +    X3 *  X2 *  X1 *  X0
```

These Boolean equations are for devices with active-HIGH outputs. These equations can be inverted for devices with active LOW outputs. The Boolean equations for active-LOW devices can also be directly derived from the truth table by collecting all the product terms where the active-LOW outputs (zeros) are asserted.

Manipulating the equations with Boolean algebra, we obtain the Boolean logic equations:

```
X0  :- /X0
X1  :- X1  :+: X0
X2  :=  X2  :+: (X1   * X0)
X3  :-  X3  :+: (X2   * X1   * X0)
```

Similarly, for active-LOW output devices (since /(A :+: B) = /A :+: B):

```
/X0 :=  X0
/X1 :- /X1 :+: X0
/X2 :- /X2 :+: (X1   * X0)
/X3 :=  /X2 :+: (X2   * X1   * X0)
```

These equations could also be obtained from the Boolean equations developed for an adder in the combinatorial design section.

Rewriting the equations for an adder:

```
X0  =  A0  :+: B0 :+: Cin
X1  -  A1  :+: B1 :-: C0
```

where

```
C0  =  A0  *  B0   +  (A0 + B0)  *  Cin
```

```
X2  -  A2  :+: B2 :-: C1
```

where

```
C1  =  A1  *  B1   +  (A1 + B1) * (A0 * B0)
    +  (A1 + B1)  *  (A0 + B0)  *  Cin
```

```
X3  =  A3  :+: B3 :+: C2
```

where

```
C2  -  A2  *  B2   + (A2 - B2) *(A1 * B1)
    +  (A2 + B2)  *  (A1 + B1)  *(A0 * B0)
    +  (A2 + B2)  *  (A1 + B1)  *(A0 * B0)
       *  Cin
```

Assuming one of the operands in the adder is the number itself and the second operand is one (X3–X0 = A3–A0, B3–B0 = 0001 and Cin = 0) we get the following equations for a counter:

```
X0  := /X0
X1  := X1  :+: X0
X2  :- X2  :+: (X1   * X0)
X3  :- X3  :+: (X2   * X1   * X0)
```

These are, of course, the same equations as the ones derived directly from the truth table. The equations for a binary counter are very regular. The general equation for an n-bit binary counter can be directly expressed:

```
Xn := Xn :+: (Xn-1* Xn-2 ... X0)
```

For devices with active-LOW outputs, the general Boolean equations can be derived by inverting both sides of the equation:

```
/Xn :- /Xn :-: (Xn-1* Xn-2 ... X0)
```

These equations represent a binary UP counter. Counting backwards for a DOWN counter, the Boolean equations can be similarly generated, either from the truth table or from the adder Boolean equations. The general equation for a DOWN counter is:

```
Xn := Xn :+: (/Xn-1* /Xn-2 ... /X0)
```

This equation is for active-HIGH outputs. For active-LOW output devices the Boolean equation for a DOWN counter is:

```
/Xn := /Xn :-: (/Xn-1* /Xn-2 ... /X0)
```

Further control functions can be added to these counter equations directly either at the truth-table stage or in the equations. For example, a load data function is required in most counters. This allows registers to be loaded with a count under the control of another input signal (LOAD). When the LOAD signal is HIGH the counter is loaded with the input data, and when the LOAD signal is LOW the counting is resumed.

Binary Counter Device Selection Considerations

One major device selection consideration is the logic requirement.

The binary counter Boolean equations make use of exclusive-OR functions in the output. In most of the registered PLDs, the XOR functions are implemented in their sum-of-products logic form. This usually requires a large number of product terms. Most standard PAL devices provide eight product terms per output. However, for larger counters, a greater number of product terms is required.

Some PLDs provide a dedicated XOR gate on the outputs. This allows an AND-OR-XOR implementation of the Boolean logic, and consequently requires fewer product terms.

Cascading Binary Counters

Situations are occasionally encountered in digital system designs where very long counters are required. Binary counters can be easily cascaded into two or more devices to construct such large counters. The design of long counters is very simple. These are designed as simple binary counters with a count enable control. The less significant counters generate an extra output signal at the penultimate count. These signals are ANDed together to form the count enable signal for the higher-order counter. For a down counter the reverse scheme is implemented.

Cascading counters is a lot easier than cascading adders because the carry-look-ahead circuitry is not required. The only thing to remember is that the more significant counter toggles only when the penultimate count of all of the less significant counters is reached.

Flip-Flop Selection

Until now, all the designs have been implemented in devices with D-type flip-flops. What happens if the counter design is implemented in a device that allows both J-K and T-type registers? The Boolean logic equations for such a design can be derived from the truth table. This requires advanced knowledge of the functionality of the J-K and T-type registers. For the J-K register the output is asserted when the J input goes HIGH and the output is unasserted when the K input goes HIGH. Toggle type registers require the T input to be asserted for every change in the output level.

Table 2. Truth Table for D, J-K and T-Type Flip-Flops

Present State				Next State															
				X3				X2				X1				X0			
X3	X2	X1	X0	D	J	K	T	D	J	K	T	D	J	K	T	D	J	K	T
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	1	0	0	1	1
0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1
0	0	1	1	0	0	0	0	1	1	0	1	0	0	1	1	0	0	1	1
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	1
0	1	0	1	0	0	0	0	1	0	0	0	1	1	0	1	0	0	1	1
0	1	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	1	0	1
0	1	1	1	1	1	0	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
1	0	0	1	1	0	0	0	0	0	0	0	1	1	0	1	0	0	1	1
1	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1
1	0	1	1	1	0	0	0	1	1	0	1	0	0	1	1	0	0	1	1
1	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0	1	0	0	0	1	1	0	1	0	0	1	1
1	1	1	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	0	1
1	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1

Table 2 shows the truth table for both a J-K and a T-type register implementation for a binary counter. Deriving and optimizing the equations from the table, we get the following results:

```
X3-J   :=  /X3  *   X2  *   X1   *   X0
X3-K   :=   X3  *   X2  *   X1   *   X0
X2-J   :=  /X2  *   X1 *X0
X2-K   :=   X2  *   X1 X0
X1-J   :=  /X1  *   X0
X1 K   :=   X1  *   X0
X0-J   :=  /X0
X0-K   :=   X0
X3-T   :=   X2  *   X1  *   X0
X2 T   :=   X1  *   X0
X1-T   :=   X0
X0-T   :=    1
```

As we can see from these equations, the number of product terms used for J-K and T-type implementations are smaller than the number of product terms required for a D-type implementation.

Which flip-flop is most efficient depends on the relative number of transitions or holds required. As a counter traverses from one count (state) to another, every output either makes a "transition" (changes logic level) or "holds" (stays at the same logic level). Small counters in general require more transitions and fewer holds. As the designs get larger, the higher-order bits require fewer transitions and more holds.

D-type flip-flops use up product terms only for active transitions from logic LOW level to HIGH level, and for logic HIGH level holds only. J-K and T-type flip-flops use up product terms for both LOW-to-HIGH and HIGH-to-LOW transitions, but eliminate hold terms. Generally, the requirements of transition and hold terms depends upon the count sequence selection. D-type flip-flops are more efficient for small designs. Conversely J-K and T-type flip-flops can be more efficient for large designs, which require more hold terms.

A comparison of product term requirements of 2-, 3-, 4- and 5-bit binary counters can be representative for other types of counters and state machines. Table 3 shows the transition terms and the hold terms required for these counters. For a J-K type flip-flop implementation, after optimizing, total product terms required are 4, 6, 8, and 10 respectively. The D-type implementation requires 3, 6, 10, and 15 respectively, and is relatively less efficient for large counters.

Table 3. Product Term Requirements for Configurable Flip-Flops

Binary Counter	Transitions	Holds	D Product Terms	J-K Product Terms	T Product Terms
2-Bit	6	2	3	4	1
3-Bit	14	10	6	6	1
4-Bit	30	34	10	8	1
5-Bit	62	98	15	10	1

Modulo Counters

The number of unique states a counter traverses is generally referred to as the modulus. A typical n-bit binary counter has a maximum modulus of 2n. It is often necessary to introduce signal delays into the logic design to meet timing requirements. This makes it possible to allow for bus-skew, access time, or differential propagation delays between devices along two different signal paths. A typical example of this is the introduction of wait states to allow for access times of different memory elements. Counters and delay lines are commonly used to introduce the delay. Counters in PLDs have the added advantage of programmability to select the required delay. Such applications where precise timing duration control is required usually use modulo counters with a non-power-of-two modulus. Other applications of modulo counters include waveform generators and arbiters.

Table 4. Truth Table for a BCD Counter

Present State					Next State			
Q3	Q2	Q1	Q0		Q3	Q2	Q1	Q0
0	0	0	0	0 -> 1	0	0	0	1
0	0	0	1	1 -> 2	0	0	1	0
0	0	1	0	2 -> 3	0	0	1	1
0	0	1	1	3 -> 4	0	1	0	0
0	1	0	0	4 -> 5	0	1	0	1
0	1	0	1	5 -> 6	0	1	1	0
0	1	1	0	6 -> 7	0	1	1	1
0	1	1	1	7 -> 8	1	0	0	0
1	0	0	0	8 -> 9	1	0	0	1
1	0	0	1	9 -> 0	0	0	0	0

A good example of a modulo counter is a BCD counter. Such a counter is useful in applications where the computer's outputs are generated using a decimal system. While a four-bit binary counter can count to sixteen, the BCD counter terminates the count at the modulus of 10.

Modulo counters can be designed in a variety of ways. One direct way is to use the truth table to implement a count to a modulus and directly derive the equations from it. The truth table for a BCD count (from zero to nine) is shown in Table 4.

Now let us consider what happens if the device accidentally powers up in one of the count values from ten to fifteen. These are illegal counts (states) and, for a good design, a mechanism must be built into the equations to allow it to recover back into a legal state. What we actually need is to consider the truth table in Table 5 in conjunction with the one in Table 4 for deriving the Boolean equations.

Table 5. Truth Table for Illegal State Recovery to Count Zero

Present State					Next State			
Q3	Q2	Q1	Q0		Q3	Q2	Q1	Q0
1	0	1	0	10 -> 0	0	0	0	0
1	0	1	1	11 -> 0	0	0	0	0
1	1	0	0	12 -> 0	0	0	0	0
1	1	0	1	13 -> 0	0	0	0	0
1	1	1	0	14 -> 0	0	0	0	0
1	1	1	1	15 -> 0	0	0	0	0

A state diagram for the BCD counter is shown in Figure 8. For active-LOW outputs, the Boolean equations can be derived directly from the truth table and optimized using Karnaugh maps or the software minimizer.

The Boolean equation for Q3 is:

```
/Q3 :=    /Q3 * /Q2 * /Q1 * /Q0
        + /Q3 * /Q2 * /Q1 *  Q0
        + /Q3 * /Q2 *  Q1 * /Q0
        + /Q3 * /Q2 *  Q1 *  Q0
        + /Q3 *  Q2 * /Q1 * /Q0
        + /Q3 *  Q2 * /Q1 *  Q0
        + /Q3 *  Q2 *  Q1 * /Q0
        +  Q3 * /Q2 * /Q1 *  Q0
        +  Q3 * /Q2 *  Q1 * /Q0
        +  Q3 * /Q2 *  Q1 *  Q0
        +  Q3 *  Q2 * /Q1 * /Q0
        +  Q3 *  Q2 * /Q1 *  Q0
        +  Q3 *  Q2 *  Q1 * /Q0
        +  Q3 *  Q2 *  Q1 *  Q0
```

The equation can be reduced to the following:

```
/Q3 :=    /Q3 * /Q2
        + /Q3 * /Q1
        + /Q2 *  Q0
        +  Q3 *  Q1
        +  Q3 *  Q2
```

Similar Boolean equations can be generated for Q2, Q12 and Q0.

Figure 9 shows the circuit diagram of a loadable dual BCD counter.

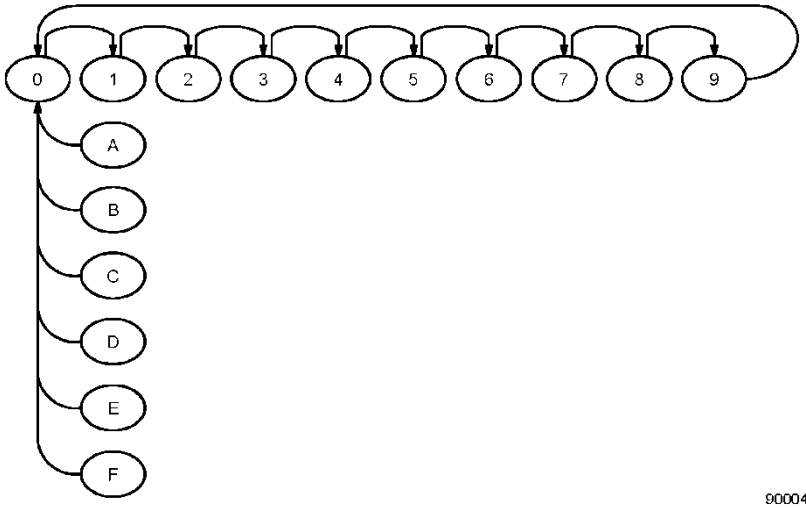


Figure 8. State Sequence of a BCD Counter Showing Illegal State Recovery

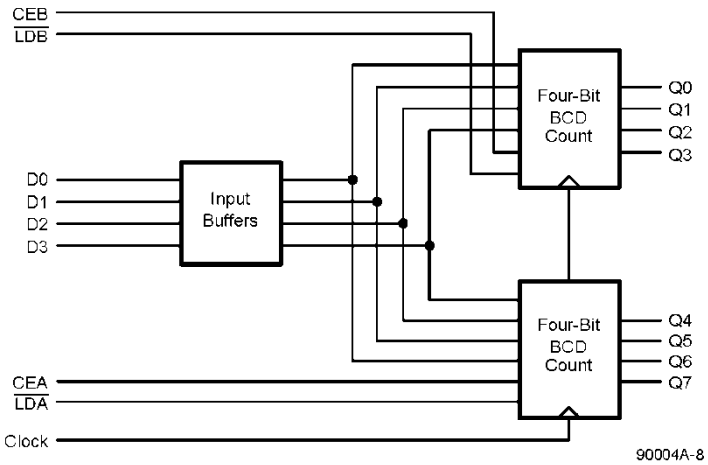


Figure 9. Circuit of a Dual BCD Counter

Modulo Counter Device Selection Considerations

We have illustrated a counter that counts from zero to a fixed modulus. The same technique can be applied for a counter which counts down from a maximum power-of-two number to a fixed modulus, or even a counter which counts from one modulus to another. The important considerations will be the number of product terms used.

The registered PLDs used for modulo counters are similar to the ones selected for other counters. Since the counts used are binary, devices with J-K, T-type flip-flops, or XOR gates will help optimize the number of product terms used. The product term usage also depends upon the modulus selected. Generally, a power-of two or a multiple-of-two modulus will require fewer product terms.

Another factor for flip-flop selection is the illegal states. D-type flip-flops are generally better suited for illegal state recovery than the J-K or T-type flip-flops. This is because when no product term is asserted, the D-type flip-flops reset to zero. Designers using J-K or T-type flip-flops must design-in illegal state recovery.

Certain devices allow the use of a synchronous RESET product term for modulo counters. The idea is to use the minimal number of product terms to build a binary counter that counts up to a power-of-two number. However, this counter is RESET to zero using the synchronous RESET product term when the desired modulus is reached. It then begins counting afresh from zero, and the procedure is repeated. Similar operation can also be achieved with a synchronous PRESET product term for a down counter.

Using synchronous RESET and PRESET product terms allows the counter to recover from illegal states. Notice that the logic product terms in the counter are designed for a complete binary count. If the counter powers up in any illegal state (as shown in Figure 10), it will continue the count until the terminal count and then, return to zero, where the correct modulo count will begin. This illegal state recovery will take an unpredictable number of clock cycles, and you may wish to design a more systematic recovery system.

Cascading Modulo Counters

For large modulo counters, the technique of generating Boolean equations from the truth tables is very tedious and time consuming. Another approach for designing modulo counters is to divide it into two smaller modulo counters. In addition to simplifying the design, this approach usually helps optimize the number of product terms.

As an example, a modulo-360 counter can be directly implemented with nine register bits. However, instead of implementing this as a straight 9-bit counter, we can implement this as two counters: one four-bit counter (counting from zero to 14) and another five-bit counter (counting from zero to 23). Together, the two counters count up to 360. The terminal count output, MOUT, is asserted when the count reaches 360, as shown in Figure 11.

The design requires nine inputs, nine outputs, one clock pin, one LOAD pin, one RESET and one MOUT (module output signal) pin. Note that no extra flip-flops or pins were needed. Obviously, the count values of this counter are not the same as a straight modulo-360 counter. Actually, this is what contributes to the optimization of the number of product terms used.

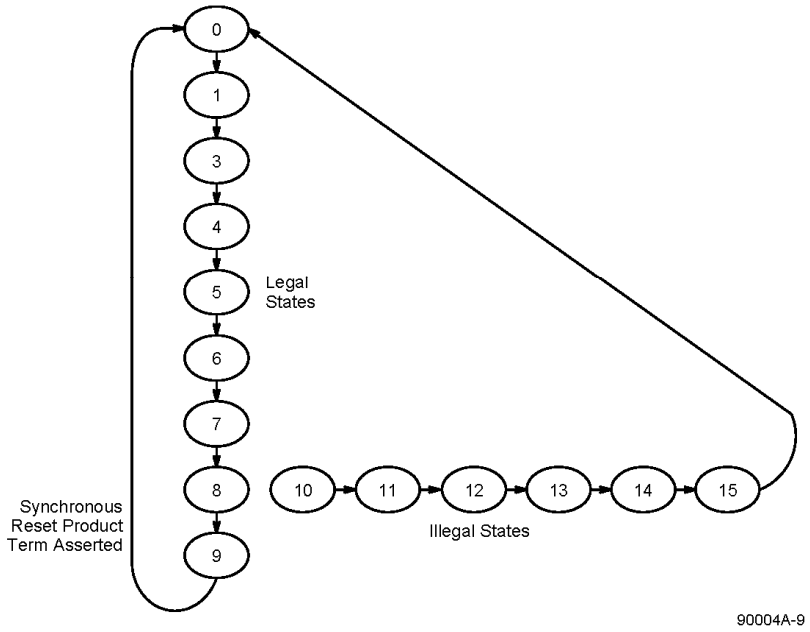


Figure 10. A BCD Counter Using Synchronous RESET Product Term

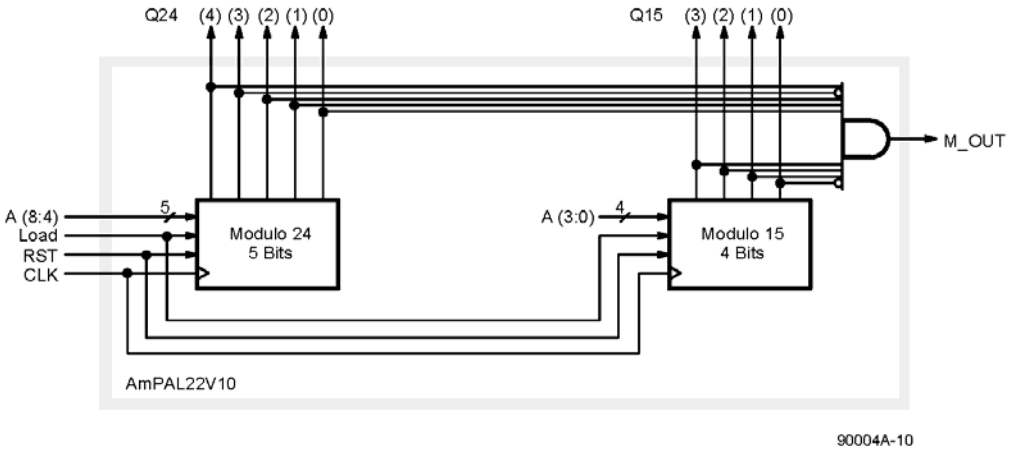


Figure 11. A Modulo-360 Counter

Counters with Encoding

Until now, we have discussed counters that generate binary output sequences. Most peripherals require a predetermined sequence of control signals. Custom control sequences can be generated by decoding the binary sequence with combinatorial logic. Figure 12 shows a general model of a counter with combinatorial output decoding circuitry. This combinatorial circuit modifies the counter bits and generates output signals in the manner required for peripheral timing and control. Since these circuits require extra combinatorial logic, they are not very efficient. They are also more susceptible to hazards and output glitches.

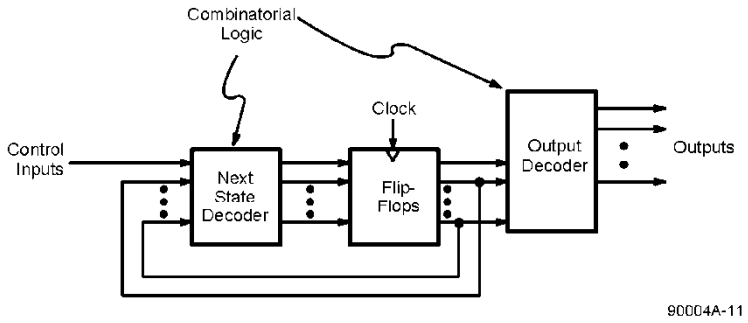


Figure 12. Counter with an Output Decoder

It is possible to have a different output coding for a four-bit counter, as shown in Table 6. This code, called Gray code, allows only one output bit to toggle for each new count value. This code can be easily derived from a four-bit binary counter code (also shown in Table 6) using an output decoder.

Table 6. Generating Gray Code from a Binary Code

Binary Code				Gray Code			
X3	X2	X1	X0	G3	G2	G1	G0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

We can derive the Boolean equations for the combinatorial output decoder from the truth table. The equations are:

```
G3 = X3
G2 = X3 :+: X2
G1 = X2 :+: X1
G0 = X1 :+: X0
```

A more efficient and easier technique for generating control signals is to implement the decode circuitry before the registers. This alternative is shown in Figure 13. This essentially generates a non-standard counter with state values that are not a binary progression. It can be considered as a counter where the product terms for a binary count and encoding the outputs have been combined.

Many different codes can be generated using such techniques. We will limit ourselves to the ones that are most commonly used: Gray-code counters and Johnson counters.

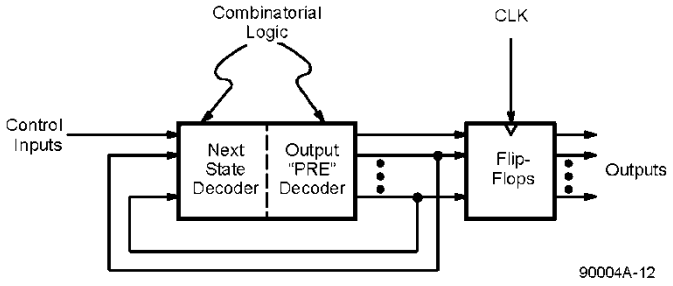


Figure 13. Counter with Combined Next State Generation and Output Encoding Circuit

Gray-Code Counters

Gray-code counters are often used in digital designs for control timing functions. The primary advantage of Gray-code counters stems from the characteristic that only one output bit changes value for every clock cycle. These output signals can be easily decoded using a combinatorial decoder without any risk of hazards. Gray-code counters are used extensively as system clocks, since the different output bits provide different clock pulses, without the risks of hazards. Gray-code is also used in high-speed data communication applications, where data is transmitted from one part of the system to another, and where the error susceptibility increases with the number of bit changes between adjacent numbers in a sequence. These are also used for such specialized applications as shaft encoders and real-time process control.

The implementation of a Gray-code counter is very simple. A truth table can be derived from the transition table as is done for a binary counter. The Boolean equations can then be directly derived from the truth table. The truth table for the Gray-code counter is shown in Table 7.

Table 7. Truth Table for a Four-Bit Gray-Code Counter

Present State				Next State			
X3	X2	X1	X0	X3	X2	X1	X0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	1
0	0	1	1	0	0	1	0
0	0	1	0	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	0	1	0	1
0	1	0	1	0	1	0	0
0	1	0	0	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	0
1	1	1	0	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	0	0	1
1	0	0	1	1	0	0	0
1	0	0	0	0	0	0	0

The Boolean logic equations for a Gray-code counter are:

```
X3  :=    /X3 *  X2 * /X1  * /X0
        +  X3 *  X2 * /X1  * /X0
        +  X3 *  X2 * /X1  *  X0
        +  X3 *  X2 *  X1  *  X0
        +  X3 *  X2 *  X1  * /X0
        +  X3 * /X2 *  X1  * /X0
        +  X3 * /X2 *  X1  *  X0
        +  X3 * /X2 * /X1  *  X0
X2  :=    /X3 * /X2 *  X1  * /X0
        +  /X3 *  X2 *  X1  * /X0
        +  /X3 *  X2 *  X1  *  X0
        +  /X3 *  X2 * /X1  *  X0
        +  /X3 *  X2 * /X1  * /X0
        +   X3 *  X2 * /X1  * /X0
        +   X3 *  X2 * /X1  *  X0
        +   X3 *  X2 *  X1  *  X0
X1  :=    /X3 * /X2 * /X1  *  X0
        +  /X3 * /X2 *  X1  *  X0
        +  /X3 *  X2 *  X1  * /X0
        +  /X3 *  X2 *  X1  *  X0
        +   X3 *  X2 * /X1  *  X0
        +   X3 *  X2 *  X1  *  X0
        +   X3 *  X2 *  X1  * /X0
        +   X3 * /X2 *  X1  * /X0
X0  :=    /X3 * /X2 * /X1  * /X0
        +  /X3 * /X2 * /X1  *  X0
        +  /X3 *  X2 *  X1  * /X0
        +  /X3 *  X2 *  X1  *  X0
        +   X3 *  X2 * /X1  * /X0
        +   X3 *  X2 * /X1  *  X0
        +   X3 * /X2 *  X1  * /X0
        +   X3 * /X2 *  X1  *  X0
```

Johnson Counters

A Johnson counter is part of a family of counters known as "ring counters." These counters are used for special applications where code symmetry is desired. Ring counters are also often used for timing purposes, since all the outputs are essentially a series of pulses. This code symmetry also allows use of the fewest possible product terms with a D-type register. Devices that provide a small amount of logic per cell, can implement Johnson counters very easily.

Johnson counters are also known as circular-shift counters. The sequence for a five-stage Johnson counter is shown in Table 8. As can be seen in the truth table, the counter first fills up with 1's from left to right and then it fills up with zeros again. Note from the output sequence that only one of the Johnson counter bits changes for every clock period, like the Gray-code counter. One major advantage of the Johnson counter is that it can be readily decoded with small two-input NAND gates and hence is suitable for high-speed applications.

Note that the five-stage sequence has a table of 10 legal states and 22 illegal states (Table 9). In general, an n-bit Johnson counter will produce a modulus of 2n. Figure 14 shows the state diagram of the five-bit counter.

Table 8. Five-Bit Johnson Counter Truth Table

Legal States

Present State					Next State				
Q4	Q3	Q2	Q1	Q0	Q4	Q3	Q2	Q1	Q0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	1	1	0	0	0	0	1
0	0	1	1	1	0	0	0	1	1
0	1	1	1	1	0	0	1	1	1
1	1	1	1	1	0	1	1	1	1
1	1	1	1	0	1	1	1	1	1
1	1	1	0	0	1	1	1	1	0
1	1	0	0	0	1	1	1	0	0
1	0	0	0	0	1	1	0	0	0

The implementation of a Johnson counter is relatively straight-forward, and is the same regardless of the number of stages. When D-type flip-flops are used, the Q output of each flip-flop is connected to the D input of the following stage. The single exception is the Q output of the last stage, which is complemented and connected to the D input of the first stage.

Table 9. Illegal States for a Five-Bit Johnson Counter

Illegal States

Present State					Next State				
Q4	Q3	Q2	Q1	Q0	Q4	Q3	Q2	Q1	Q0
0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0
0	1	0	1	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0
1	1	1	0	1	0	0	0	0	0

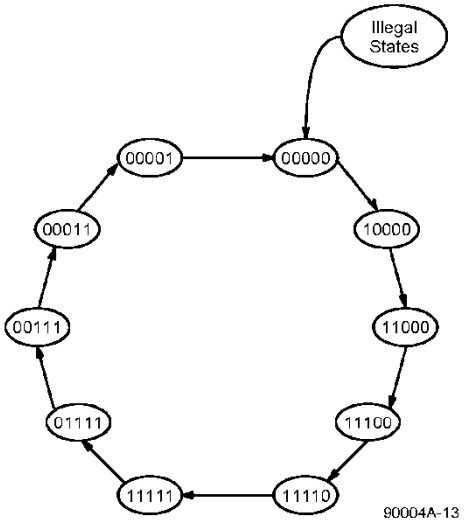


Figure 14. State Diagram of a Five-Bit Johnson Counter

One disadvantage of the counter is the number of invalid (or illegal) states. The invalid states increase exponentially with the length of the counter. The bigger the counter becomes, the greater are its chances of entering an illegal state. Johnson counters are very susceptible to illegal states, and can "hang up" very easily. Noise or improper use can cause this counter to end up in an illegal state. Therefore, a design with illegal state recovery circuitry is always recommended.

Figure 15 shows a nine-bit Johnson counter that can be derived by directly extending the design of a five-bit Johnson counter.

Shift Registers

A Shift Register is a special digital circuit often used as a primary building block in digital computer systems. It is closely related to a ring counter. Its fundamental usage is for temporary data storage and bit-wise data manipulation for advanced arithmetic and multiplication operations. Shift registers are also frequently used in communications, for converting parallel byte-wide data from the microprocessor to a serial data bit-stream for transmission. Shift registers are also used in graphics systems for serializing parallel data for use by the display monitor. A number of examples of video shift registers are included in the graphics section.

The fundamental purpose of a shift register (Figure 16) is to shift data from one flip-flop to another. There are several types of shift registers. They are classified by the way in which incoming data is received (parallel or serial), and how outgoing data is transmitted (parallel or serial).

In the following example, we will discuss a simple universal shifter that provides both serial and parallel input and output functions. Depending upon the control signals I0 and I1, the data is shifted from one flip-flop to another in the left or the right direction. These inputs also control when the new parallel data is loaded onto the registers. When shifting left or right, serial data can be received and transmitted on serial pins LIRO and RILO. Since the flip-flop outputs appear on the output pins at all times, the parallel output data is always available. The truth table is shown in Table 10.

The Boolean logic equations can be directly derived from the truth table, and are shown Figure 17.

Shift registers can be modified to suit various system design requirements. This universal shift register can be used for serial in/serial out, parallel in/parallel out, serial in/parallel out and parallel in/serial out functions.

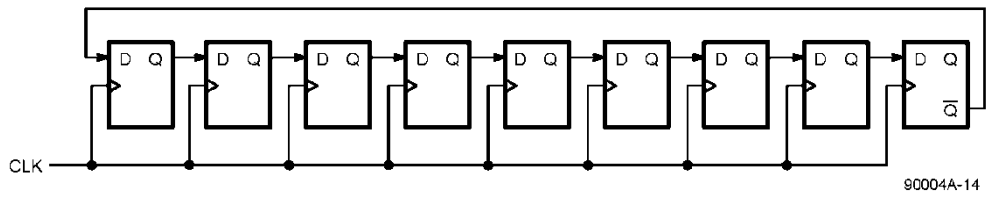


Figure 15. Block Diagram of a Nine-Bit Johnson Counter

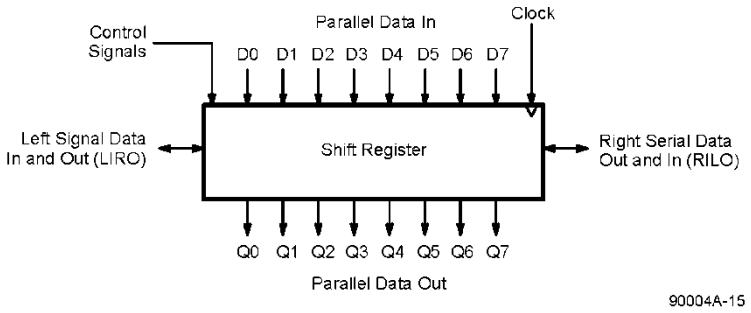


Figure 16. A Shift Register Block Diagram

Table 10. The Truth Table for a Universal Shift Register

Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0	I1	I0	
Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0	0	0	;Retain Data
RILO	Q7	Q6	Q5	Q4	Q3	Q2	Q1	0	1	;Shift Right
Q6	Q5	Q4	Q3	Q2	Q1	Q0	LIRO	1	0	;Shift Left
D7	D6	D5	D4	D3	D2	D1	D0	1	1	;Load Data

```
Equations
/Q0   :=     /I1*/I0*/Q0              ;HOLD Q0
       +     /I1*I0*Q1                ;SHIFT RIGHT
     :+:      I1*/I0*/LIRO            ;SHIFT LEFT
       +      I1*I0*/D0               ;LOAD D0
/Q1   :=     /I1*/I0*/Q1              ;HOLD Q1
       +     /I1*I0*/Q2               ;SHIFT RIGHT
     :+:      I1*/I0*/Q0              ;SHIFT LEFT
       +      I1*I0*/D1               ;LOAD D1
/Q2   :=     /I1*/I0*/Q2              ;HOLD Q2
       +     /I1*I0*/Q3               ;SHIFT RIGHT
     :+:      I1*/I0*/Q1              ;SHIFT LEFT
       +      I1*I0*/D2               ;LOAD D2
/Q3   :=     /I1*/I0*/Q3              ;HOLD Q3
       +     /I1*I0*/Q4               ;SHIFT RIGHT
     :+:      I1*/I0*/Q2              ;SHIFT LEFT
       +      I1*I0*/D3               ;LOAD D3
/Q4   :=     /I1*/I0*/Q4              ;HOLD Q4
       +     /I1*IO*/Q5               ;SHIFT RIGHT
     :+:      I1*/I0*/Q3              ;SHIFT LEFT
       +      I1*I0*/D4               ;LOAD D4
/Q5   :=     /I1*/I0*/Q5              ;HOLD Q5
       +     /I1* I0*/Q6              ;SHIFT RIGHT
     :+:      I1*/I0*/Q4              ;SHIFT LEFT
       +      I1* I0*/D5              ;LOAD D5
/Q6   :=     /I1*/I0*/Q6              ;HOLD Q6
       +     /I1*I0*/Q7               ;SHIFT RIGHT
     :+:     I1*/I0*/Q5               ;SHIFT LEFT
       +     I1*I0*/D6                ;LOAD D6
/Q7   :=     /I1*/I0*/Q7              ;HOLD Q7
       +     /I1*I0*/RILO             ;SHIFT RIGHT
     :+:     I1*/I0*/Q6               ;SHIFT LEFT
       +     I1*I0*/D7                ;LOAD D7
             /LIRO = /Q0              ;LEFT IN RIGHT OUT
             LIRO.TRST = /I1*I0
             /RILO = /Q7              ;RIGHT IN LEFT OUT
             RILO.TRST = I1*/I0
```

Figure 17. Boolean Logic Equations for an Octal Shift Register

Barrel Shifters

In most data processing systems, some form of data shifting or rotation is necessary. In typical computer systems, the shifter is located at the output of the ALU, and usually requires a single-cycle shift and add function (Figure 18). For such applications as floating-point arithmetic or string manipulation, ordinary shift registers are inefficient, since they require n clock cycles for an n-bit shift.

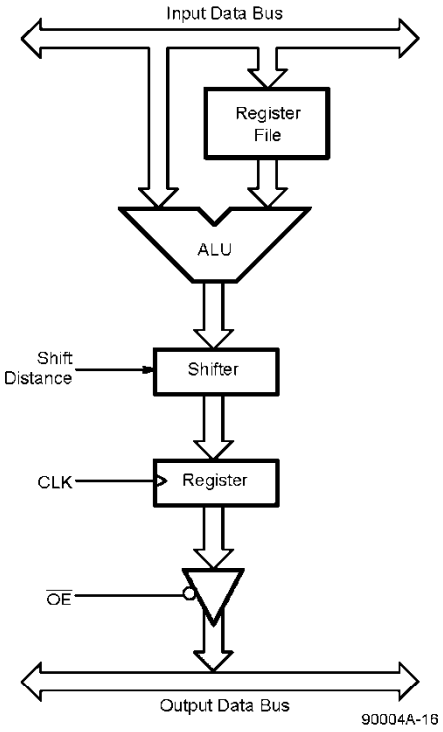


Figure 18. Typical ALU Architecture

A specialized shift register, called a "barrel shifter," is used to shift (or rotate) data by any number of bits in a single clock cycle. The name "barrel shifter" is used because of the circular nature of the shift operation. The storage registers on the output of the shifter are used in this architecture to pipeline the data operation, increasing throughput. The three-state buffer on the output registers is also useful for providing an interface to the data bus.

The design of a barrel shifter proceeds in the same manner as a regular shift register. The truth table is drawn, and the Boolean equations are then written based upon the truth tables. An eight-bit barrel shifter requires at least eight data inputs, eight registered data outputs, three control lines to specify the shift distance, a clock input and an output enable that controls the three-state buffer on the register output.

Figure 19 shows the block diagram for an eight-bit registered barrel shifter, while Table 11 shows the truth table. The registered barrel shifter requires a total of 14 inputs and 8 outputs.

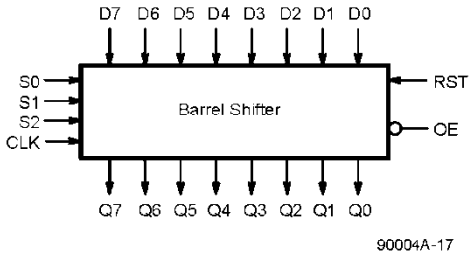


Figure 19. Block Diagram of an Eight-Bit Barrel Shifter

Table 11. Truth Table for an Eight-Bit Barrel Shifter

S2	S1	S0	Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0
0	0	0	D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	D6	D5	D4	D3	D2	D1	D0	D7
0	1	0	D5	D4	D3	D2	D1	D0	D7	D6
0	1	1	D4	D3	D2	D1	D0	D7	D6	D5
1	0	0	D3	D2	D1	D0	D7	D6	D5	D4
1	0	1	D2	D1	D0	D7	D6	D5	D4	D3
1	1	0	D1	D0	D7	D6	D5	D4	D3	D2
1	1	1	D0	D7	D6	D5	D4	D3	D2	D1

Gray-Code, Johnson Counter and Shift Register Device Selection Considerations

Gray-code counters, Johnson counters and shift registers are not very logic-intensive; the number of product terms required is minimal. The D-type flip-flops provide the most efficient implementations, allowing these designs to be easily implemented in most PAL devices.

Since Gray-code counters are often used as system clocks, very high speed PAL devices provide the highest resolution clocks.

Barrel shifters are very logic-intensive and require many product terms, since data from all the inputs needs to be accessible at any output. Registered PLDs with a large number of product terms are ideal for barrel shifters. Large barrel shifters can also be partitioned into a number of PLDs.

Asynchronous Registered Designs

Until now, we have discussed strictly synchronous registered designs, where a common system clock is used. In asynchronous registered designs, a common clock is not used. The register clock may be generated by the output of another register, or by a logical combination of various other signals. Such designs are usually slow for such applications as timing generation, because when the output of one register is used to clock another, multiple delays are encountered before all the register outputs stabilize. On the other hand, designs can be very fast for asynchronous applications such as bus arbitration and control, where a fast response to a bus signal can be provided without waiting for a common system clock.

Although asynchronous designs are easier to visualize, they present larger problems in implementation. Combinatorial hazard conditions can cause false clocking of registers, destroying the logic intended by the designer. The designer also needs to worry about race conditions when clocking a number of register simultaneously. Careful design analysis is strongly recommended before implementing any asynchronous design.

Ripple counters are probably the easiest examples of such asynchronous designs. Figure 31 shows the logic diagram of a five-bit binary ripple counter. These counters clearly have the advantage of design simplicity. The output from one stage is fed as the clock to the next stage. However, this results in a slower counting rate, since the clock signals need to propagate through all five registers before the next count is reached.

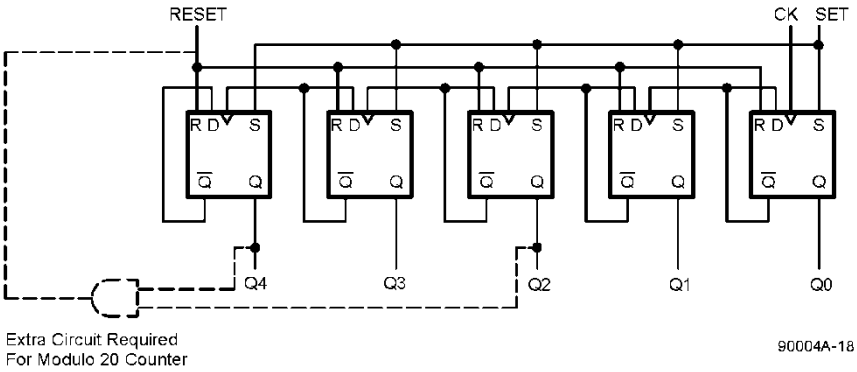


Figure 20. A Five-Bit Ripple Counter

Figure 20 shows the implementation of a modulo-20 counter that is RESET when output bits Q4 and Q2 are both HIGH. Since the RESET is implemented with a product term, the extra AND gate shown can be implemented directly within the PAL device.

Asynchronous Designs Device Selection Considerations

The device selection for asynchronous designs is easy. As the clock signals require logic, only PLDs that allow implementations of Boolean logic on the clock signals are useful.

OTHER APPLICATIONS OF REGISTERED PLDs

Registered PLDs are used for a number of miscellaneous applications that are not covered by the synchronous and asynchronous design applications discussed up to now. One such application is as a frequency divider.

- Frequency dividers
- Addressable Registers

Frequency Dividers

Standard synchronous counters provide the basic capability of dividing an input frequency. A single register of a PAL device will let us divide by two.

If we stack these registers, a binary counter provides symmetrical division by 2, 4, 8, 16, etc. This divider has been a standard for years, and the PAL device has always been on excellent choice for such applications.

One unique application of PAL devices is for dividing input frequencies by odd numbers. This has been done historically by designing a counter that cycles an odd number modulo, and decoding the specific states of the counter. The disadvantage of this approach is that the output is not symmetrical and the duty cycle is not 50%.

Let us examine a simple divide-by-five counter. This counter can be implemented using three flip-flops that start at zero and reset at four, resulting in a five-state counter. Table 12 shows the outputs of the three individual flip-flops.

Table 12. Truth Table for a Five-Bit Counter

Present State			Next State			
Q2	Q1	Q0	Q2	Q1	Q0	
0	0	0	0	0	1	State zero to one.
0	0	1	0	1	0	State one to two.
0	1	0	0	1	1	State two to three.
0	1	1	1	0	0	State three to four.
1	0	0	0	0	0	State four to zero.

The Boolean equations are:

```
Q2  := /Q2   * Q1   * Q0    ;MSB bit
Q1  := /Q1   * Q0   + Q1   * /Q0
Q0  := /Q2   */Q0           ;LSB bit
```

The waveforms for this divider are shown in Figure 21. Notice that the Q2 output goes HIGH for one state and that this output is one fifth of the input frequency, but it is a 20% duty cycle. Q1 is active for two states; it provides the same frequency, but with a 40% duty cycle. If we want a 50% duty cycle, we are going to have to divide a state in half.

To provide the 50% duty cycle, the two edges should be evenly spaced in the count sequence, one edge in the middle of state two and one at the beginning of state zero. The first edge can be formed by logically "ANDing" state_2 with the falling edge of the clock. The second edge can be formed by decoding state zero.

```
edge_1 =/clock */Q2 * Q1*/Q0
                        ;edge between
                        ;states two and
                        ;three
edge_2= /Q2 */Q1 */Q1 ;edge at state
                        ;zero
```

The logical "OR" of these two equations will provide the needed rising edges. To provide a clean output, this signal should clock another output register.

The next step in the design is to pick the appropriate PAL device to fit this design. Our biggest concern is that we need the capability of clocking the counter at one speed and the output flip-flop at another. To do this, we cannot use a PAL device that has a dedicated clock pin; we need an architecture that allows programmable clocks.

The clock signal requires two product terms (one for each edge). Another technique is to use the independent asynchronous SET and asynchronous RESET product terms of the output register. A HIGH on the SET product term asserts the register output, and a HIGH on the RESET product term unasserts the register output. Due to the asynchronous nature of the product terms some adjustment in timing is required. The SET product term is asserted when in state 0 (Q2=0, Q1=0 and Q0=0), and the RESET product term is asserted when between states two and three.

```
OUTPUT.SET = /clock * /Q2 * Q1 * /Q0
                                ;set between
                                ;states 2 & 3
OUTPUT.RESET = /Q2 */Q1 * /Q0
                                ;reset at
                                ;state zero
```

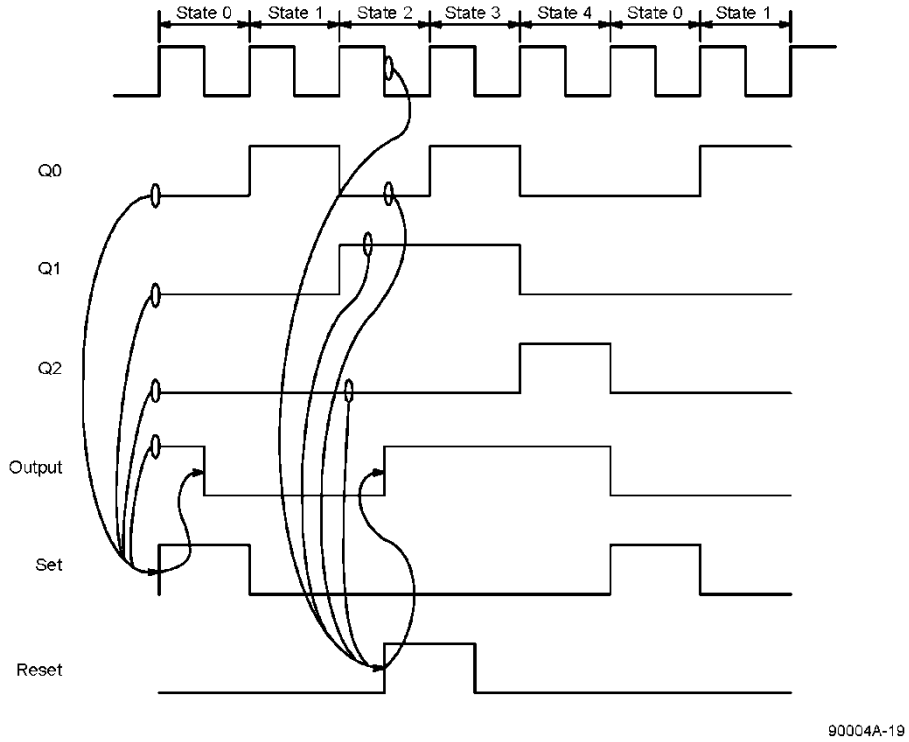
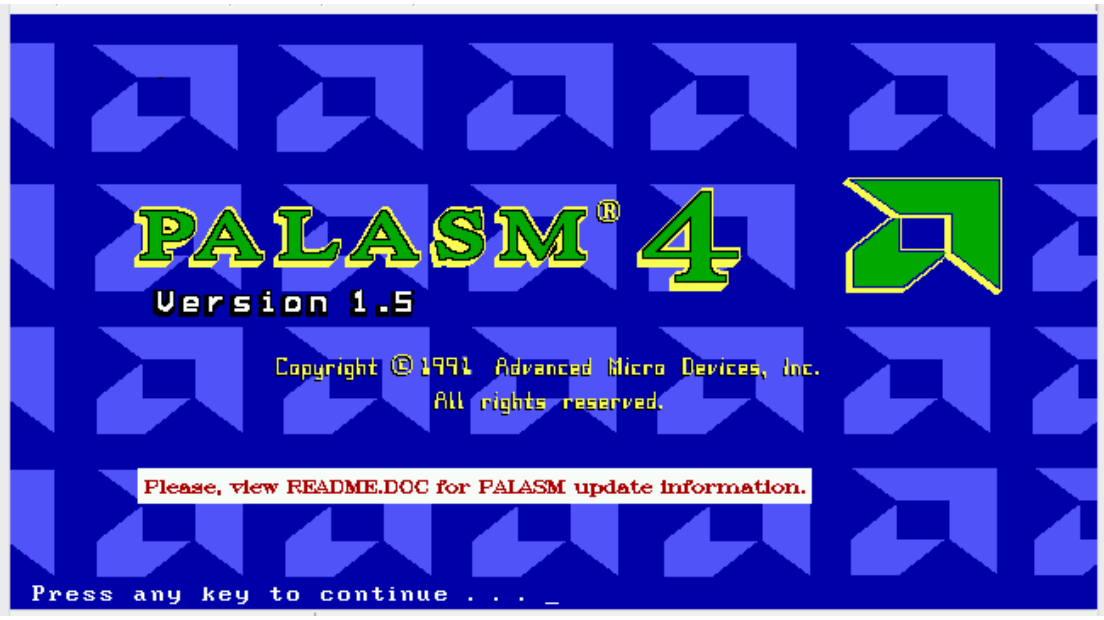


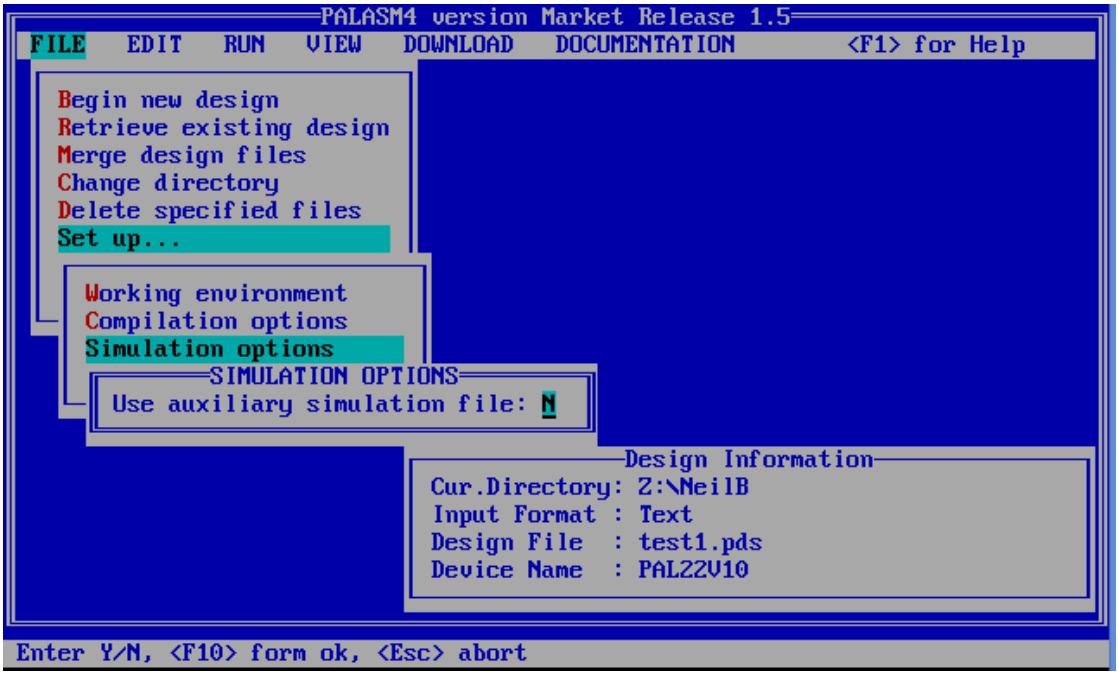
Figure 21. Waveform for a Frequency Divider

Addressable Registers

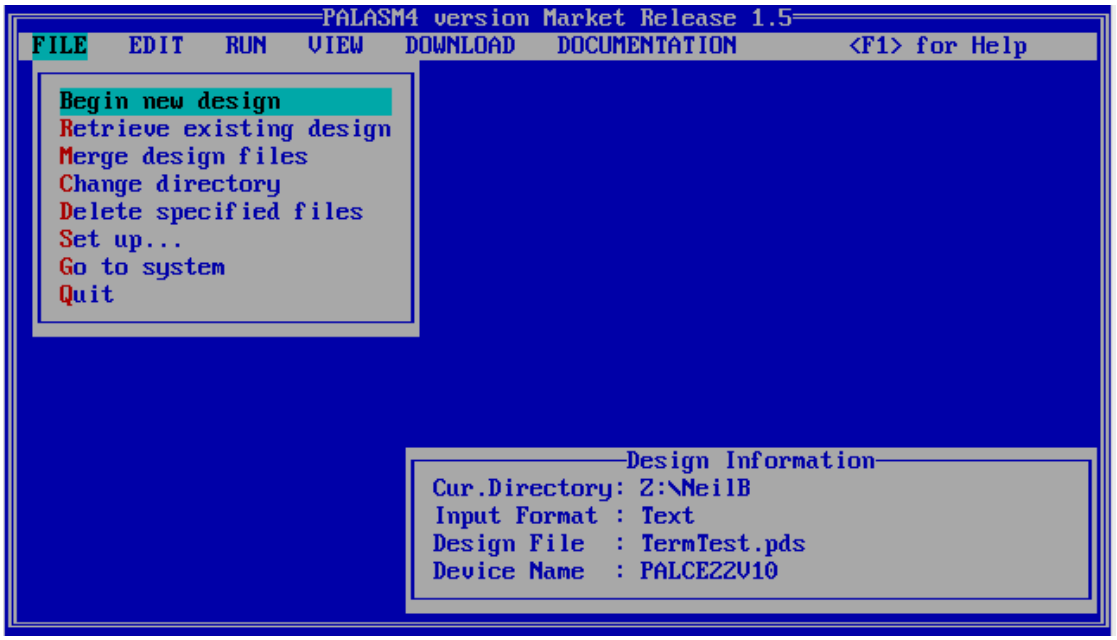
Addressable registers are commonly-used MSI functions, often implemented in PAL devices. Addressable registers are used as building blocks for digital computers. Depending upon the address input one of the many flip-flops in the register retain their previous values.



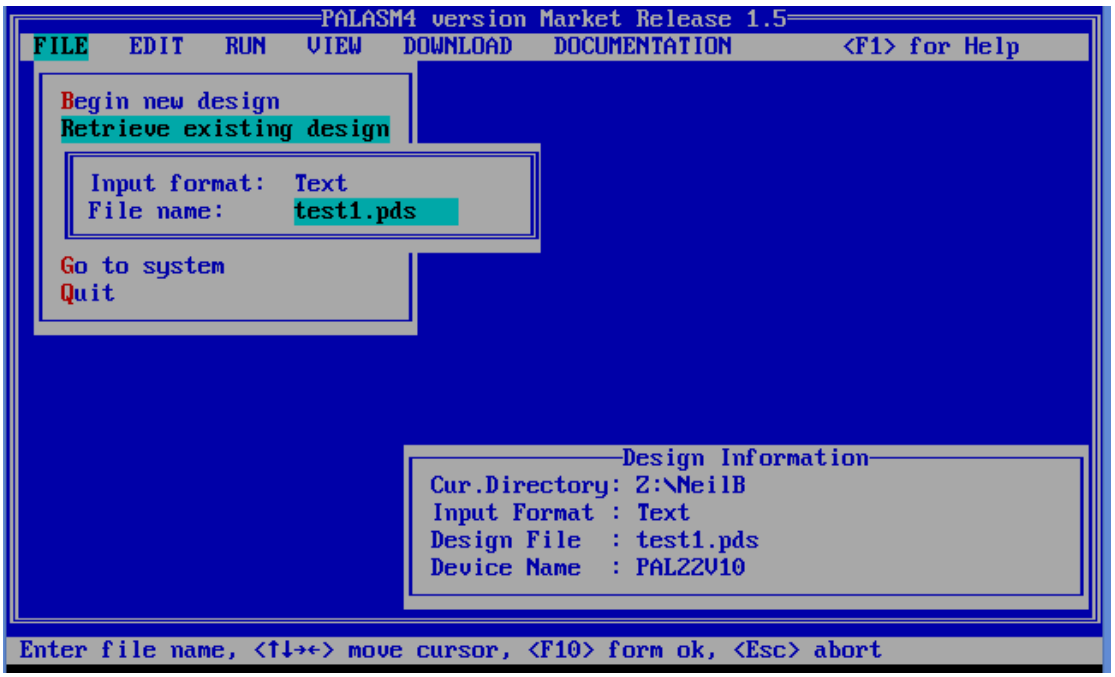
Welcome to the machine….



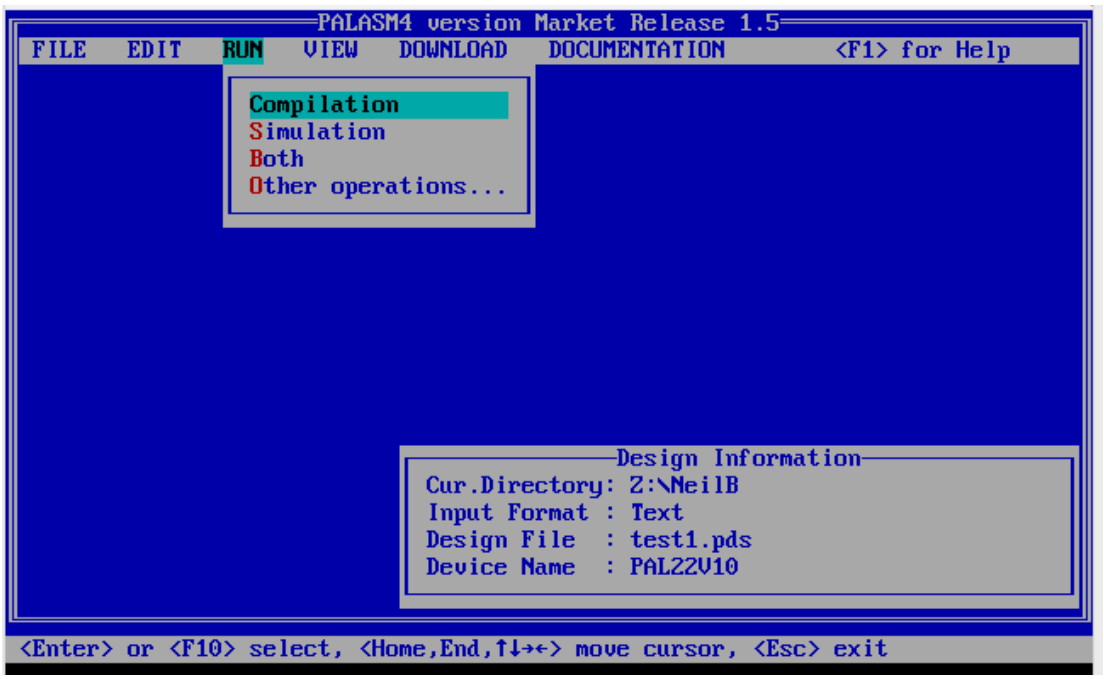
Setting “Use auxiliary simulation file” to ‘N’ directs PALASM to use the simulation data in your .PDS file.



If you want to start a new design you can do so from inside PALASM. I use an external text editor (notepad++) to edit my .PDS files so I don't typically use this option.

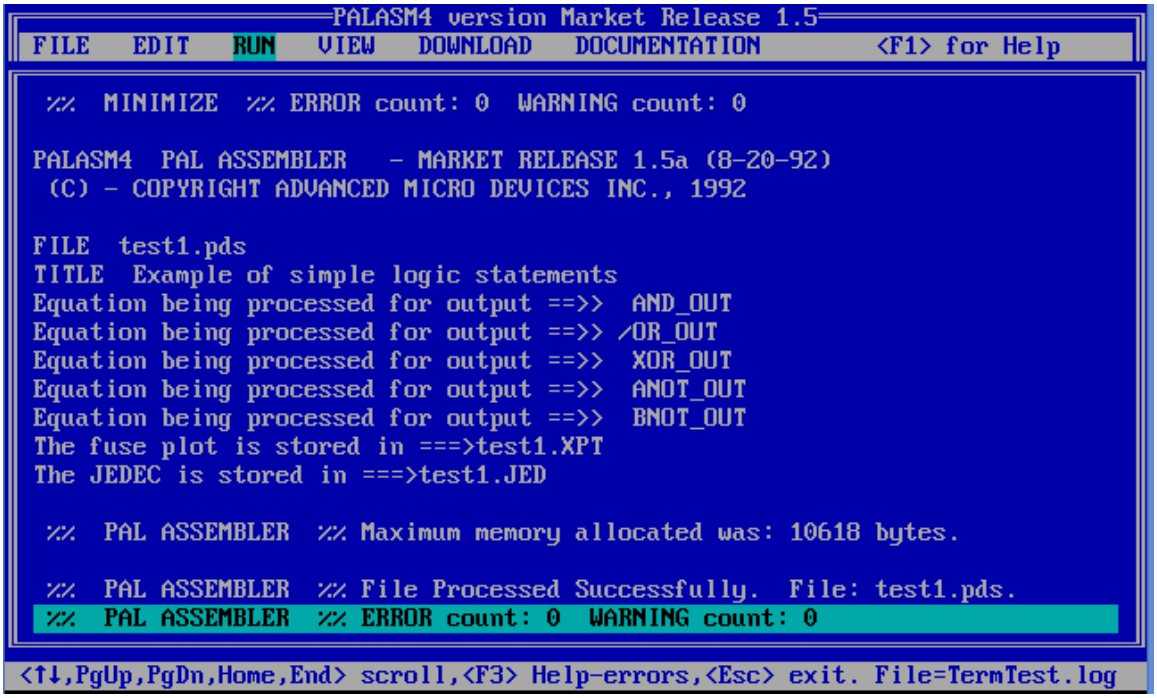


Using the .PDS file edited in an external text editor.

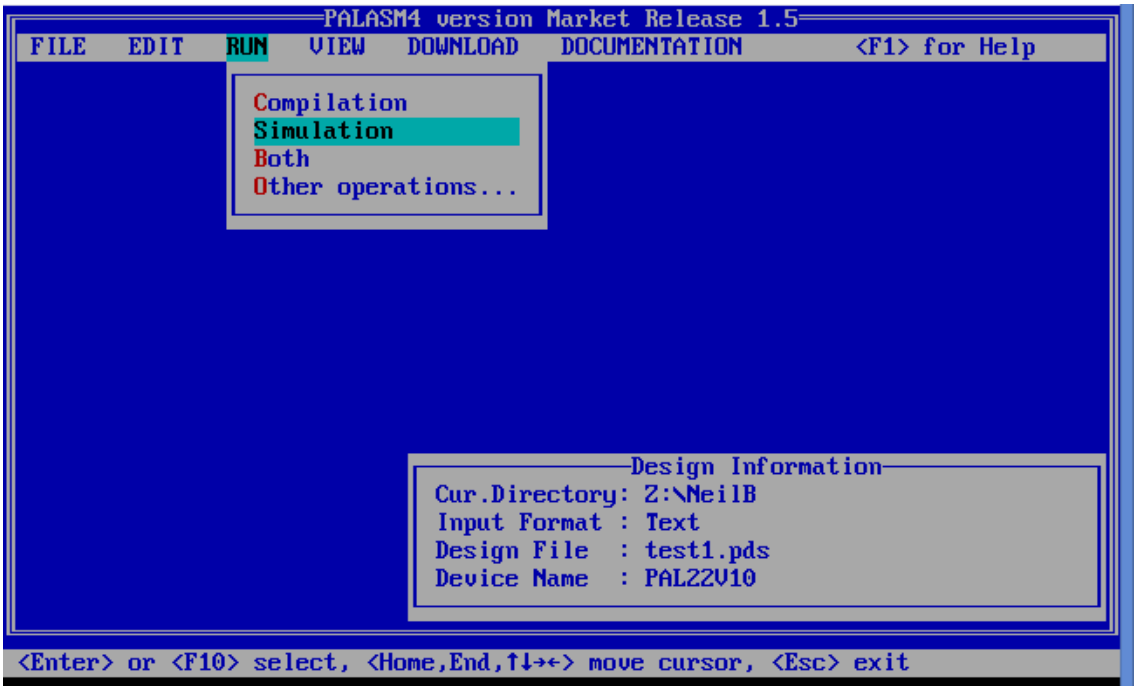


Compiling the .PDS file

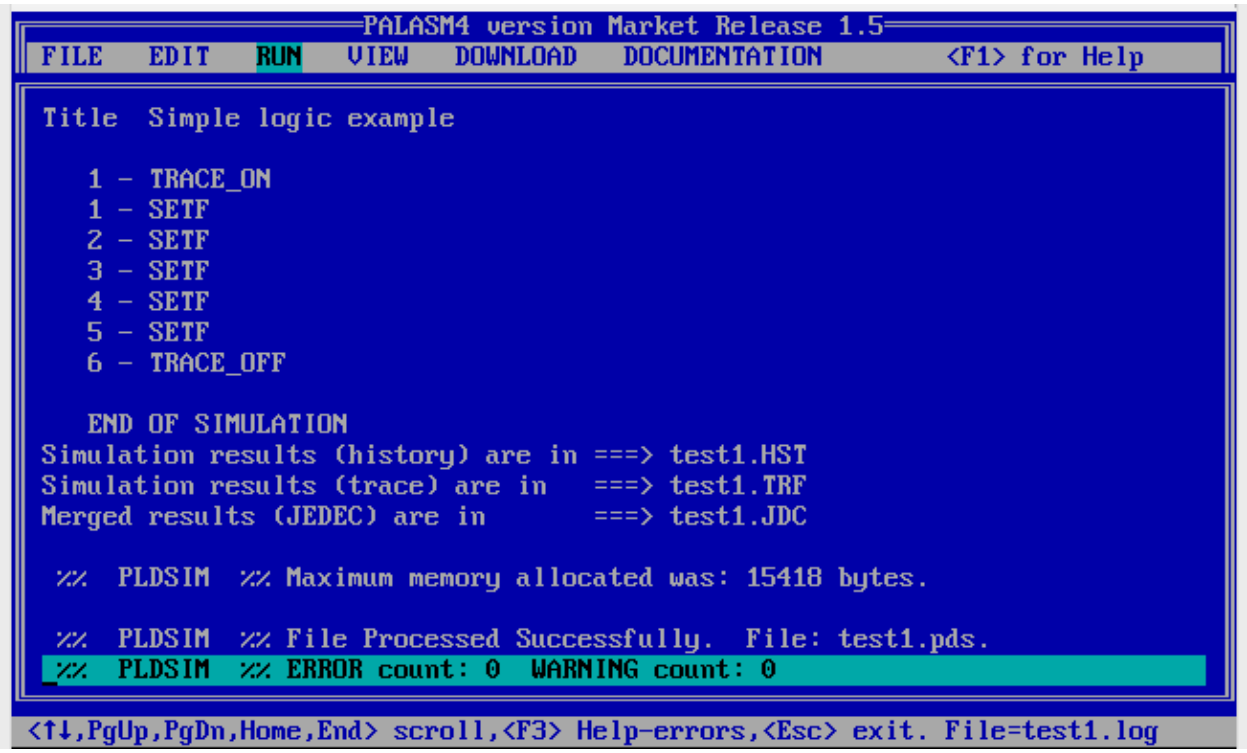
Notice the settings for the compile. You will typically need to turn “Minimize Boolean” on.



It compiled!



Running a simulation of the design.



The simulation run has completed.

Note that if you get error ‘D10003’ one way I have found to resolve it is to turn “Minimize Boolean” off; recompile your project; turn “Minimize Boolean” back on; recompile your design then run the simulation.

The .TRF file has the results of the simulation run. You will need to review the file to determine if the simulation results match your design goals.

```
PALASM4  PLDSIM   - MARKET RELEASE 1.5 (7-10-92)
 (C) - COPYRIGHT ADVANCED MICRO DEVICES INC., 1992
PALASM SIMULATION SELECTIVE TRACE LISTING
Title     : Simple logic example     Author   : Neil Breeden
Pattern   : TEST1.PDS                Company  : N8VEM
Revision  : 0                        Date     : 05/30/14
PAL22V10
Page : 1

           ggggg
 A_IN      LLHHL
 B_IN      LHLHL
 AND_OUT   LLLHL
 OR_OUT    LHHHL
 XOR_OUT   LHHLL
 ANOT_OUT  HHLLH
 BNOT_OUT  HLHLH
```

I can see that AND_OUT (highlighted in yellow) is only high when A_In and B_In are high so I know the equation for AND_Out is working as expected.

The files produced by PALSAM for our example above.

```
TEST1.HST - Complete trace log; includes all pins
TEST1.JED - The JEDEC file your device programmer will use
TEST1.LOG - Compiler log
test1.pds - Our design file
TEST1.TRF - Trace log for the pins defined in TRACE_ON
TEST1.XPT - Fuse map dump file
```

Version History:

V1.5 – Initial Release

V1.6 – Addition content and editiing