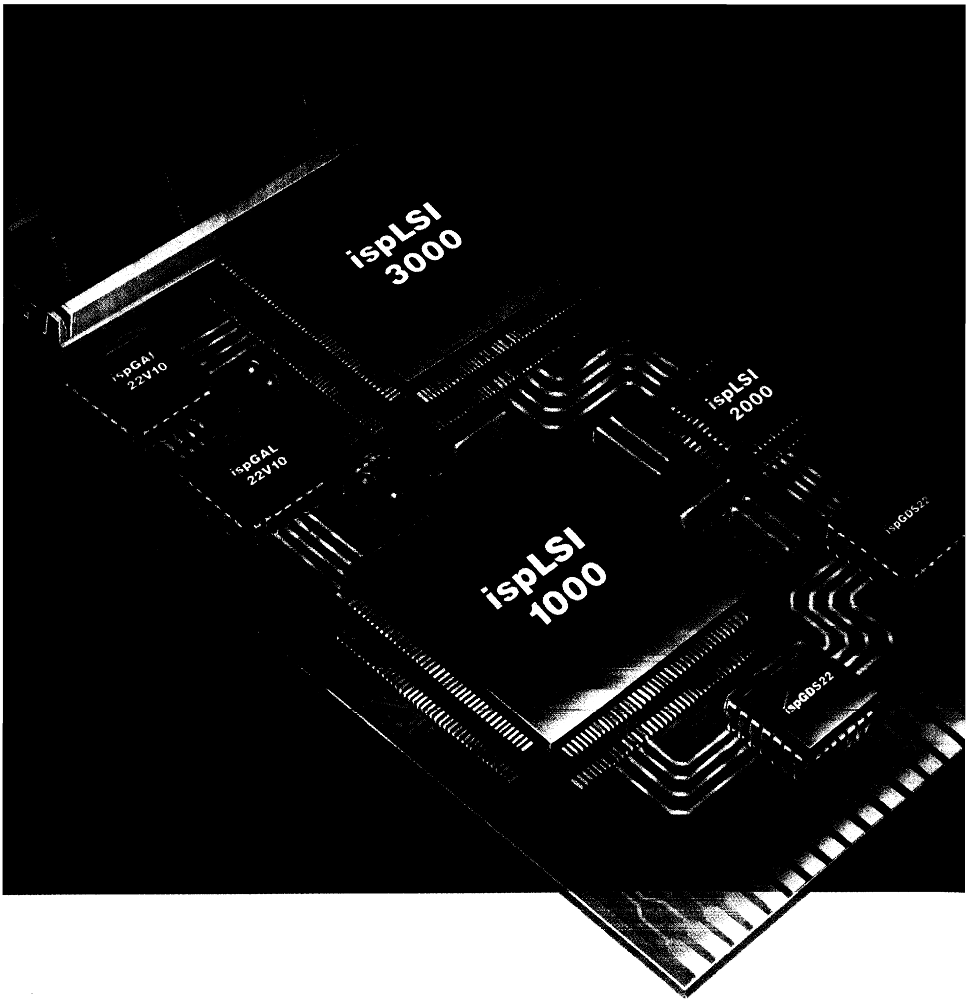
ISP™ Manual 1994



Lattice™

Lattice
In-System Programmability
Manual
1994

Lattice isp™ On Board

Lattice™

Lattice™

Lattice Semiconductor Corporation products are made under one or more of the following U.S. and international patents: 4,761,768 US, 4,766,569 US, 4,833,646 US, 4,852,044 US, 4,855,954 US, 4,879,688 US, 4,887,239 US, 4,896,296 US, 5,130,574 US, 5,138,198 US, 5,162,679 US, 5,191,243 US, 5,204,556 US, 5,231,315 US, 5,231,316 US, 5,237,218 US, 5,245,226 US, 5,251,169 US, 0194091 EP, 0196771B1 EP, 0196771 UK, 0196771 WG.

LATTICE SEMICONDUCTOR CORP.
5555 Northeast Moore Court
Hillsboro, Oregon 97124 U.S.A.
Tel.: (503) 681-0118
FAX: (503) 681-3037

About This Manual

Background

Lattice Semiconductor Corporation, founded in 1983 and based in Hillsboro, Oregon, has been providing innovative solutions to the manufacturers of high-performance systems for over a decade. Lattice pioneered nonvolatile, reprogrammable logic with its UltraMOS® E^2CMOS® technology. This technology, combined with the Lattice GAL® architectures, has established Lattice products as the industry-standard in low-density programmable logic. Lattice's ispLSI® and pLSI® families of high-density PLDs combine leadership performance and density with in-system programmability to establish the high-density programmable logic standard of the 1990s.

What This Manual Contains

This manual provides a comprehensive guide to using Lattice's In-System Programmable (ISP™) devices and design tools. Compiled from information gathered from Lattice's customers and applications engineers, this manual demonstrates how easy it is to design-in and go to full production with Lattice ISP silicon and software solutions. This manual not only describes the theory behind ISP, but also shows the practical, "how-to" implementation of ISP. By combining this manual with the latest Lattice Data Book and Handbook, the reader will have a comprehensive reference library to efficiently design and implement ISP solutions.

Additional Information

For information on product availability and pricing, please contact your Lattice Sales Representative or Distributor. A listing of all Lattice Sales Offices, Sales Representatives, and Distributors is included at the end of this manual.

For immediate help with technical questions or access to selected applications described inside, please call:

Applications Hotline

ispGAL®, ispGDS™, and GAL Products:	Tel. 1-800-FASTGAL (327-8425) FAX (503) 681-3037
ispLSI and pLSI Products:	Tel. 1-800-LATTICE (528-8423) FAX (408) 944-8450

Electronic Bulletin Board

ispGAL, ispGDS, and GAL Products:	(503) 693-0215
ispLSI and pLSI Products:	(408) 980-9814

Table of Contents

Section 1: ISP Overview

Section 2: The Basics of ISP

Section 3: ISP Programming Options

Section 4: Application Notes and Article Reprints

Section 5: General Information

Section 1: ISP Overview

Section 2: The Basics of ISP

Section 3: ISP Programming Options

Section 4: Application Notes and Article Reprints

Section 5: General Information

Index

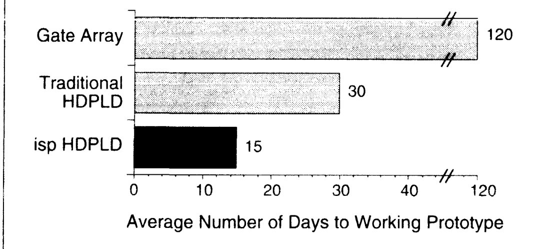
ISP Overview

Introduction

ISP™ (In-System Programmability): The ability to reconfigure the logic and functionality of a device, board, or complete electronic system before, during, and after its manufacture and shipment to the end user.

ISP is the new standard in programmable device technology. ISP eliminates traditional PLD limitations and delivers benefits in board and system-level design, manufacturing, and programming. Since ISP hardware is as flexible and easy to modify as software, design upgrades are simple. Because ISP devices can be treated like any other device on the PCB, no special manufacturing flow is required to program ISP devices; standard 5-volt logic level programming signals are easily generated by a PC, Sun Workstation®, ATE (Automatic Test Equipment), or system embedded microprocessor. In pioneering ISP, Lattice has developed an integrated solution of silicon, software and applications know-how that makes ISP a practical technology.

Figure 1. In-System Programmability: Time-To-Market Advantage



Source: Independent survey of over 300 PLD users

Q: What's Driving ISP Momentum?

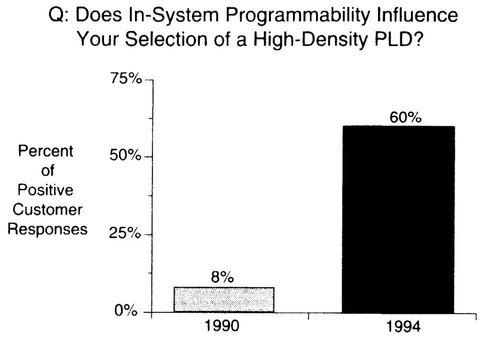
A: Time-To-Market

The drive for a shorter time-to-market has fueled explosive growth in the use of PLDs.

Based on user responses, ISP provides an additional 50% reduction in time-to-market over traditional HDPLDs and a more than 85% reduction in time-to-market compared to gate array implementations (Figure 1).

Another indicator of ISP's momentum is the percentage of designers who say that ISP capability will influence their selection of an HDPLD (Figure 2). Just four years ago, when asked, only 8% of system designers said that ISP would influence their HDPLD decision. Today, that percentage has leaped to 60%!

Figure 2. In-System Programmability: An Emerging Standard



Source: Independent survey of over 300 PLD users.

This overview presents the benefits of ISP and summarizes the ISP solutions available from Lattice. The outcome is convincing—ISP drives dramatic savings in design cycle time, manufacturing costs, and time-to-market.

ISP Design Benefits

ISP allows design, test, and manufacturing engineers to reconfigure system features while the devices remain soldered on the circuit board. This capability revolutionizes design prototyping, board-level debug, system manufacturing, and system upgrades.

The Superior Prototyping Solution

During most system design cycles, major board building blocks such as the microprocessor and RAM are selected first, well before system logic decisions are made. When using ispLSI devices, the designer can fully populate his prototype board with the major building blocks, interconnecting all functions with programmable logic and switch devices. Design changes, whether they require added or modified logic, can be made in minutes using Lattice's pDS or pDS+ software design tools. A 5-wire download cable from a PC or workstation to the prototype board downloads the new logic into the device(s). This ability to modify system functionality without changing components or printed circuit board (PCB) layout is only the first of many advantages afforded by Lattice's ISP technology.

Internal Test

Once the ISP logic has been stabilized, the designer may use the ISP devices to debug other portions of the board. For example, a circuit board frequently operates in a system where it is supplied with stimulus from other boards. The designer can use in-system programmability to debug system-level operation more quickly by reconfiguring the ISP devices to force or redirect signals (e.g. clocks or control signals) into various portions of the board design. This ability to thoroughly check board designs saves precious time during system-level debug and translates directly into a competitive time-to-market advantage.

Board Reconfiguration and Field Upgrades

ISP devices provide an ideal way to reconfigure boards and/or upgrade product features in the field. With conventional logic technology, a system installed at a customer site is very expensive and difficult to upgrade to the latest hardware revision, to fix hardware bugs, or to enable hardware options. With ISP devices, however, if a subsequent reconfiguration, upgrade or repair is required, a simple upgrade disk can be used, either in the field or the factory, to reconfigure the logic (e.g. to modify memory refresh or control logic or to operate with a faster microprocessor). Updates via modem, serial link, or a special ISP programming interface are possible depending on the system environment or needs.

ISP Manufacturing Benefits

ISP is not only revolutionizing the world of logic design but is also dramatically transforming the world of manufacturing. The ISP devices support multi-function hardware designs that reduce system part count and cost. ISP also supports reconfigurability for test which enhances board-level testability and, ultimately, system reliability. Finally, ISP allows the "standard PLD manufacturing flow" to be simplified (Figure 3), reducing cost and enhancing system quality.

Multi-Function Hardware

ISP can be used to exploit the concept of multi-function hardware: a single hardware design able to implement a variety of system-level functions via in-system programming. Multi-function hardware allows manufacturers to reduce the number of unique board designs used in a system, further simplifying the manufacturing flow.

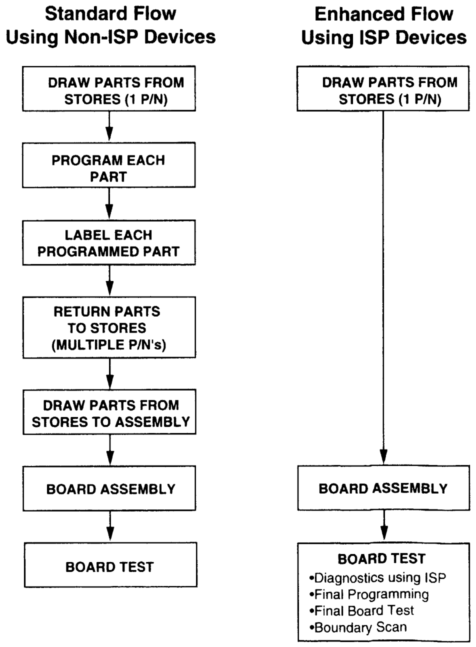
Multi-function hardware dramatically lowers system-level costs by reducing the component count on the boards as well as reducing the number of different boards required to implement various system-level options.

A dual-processor board, intended to interface with several bus interface standards, illustrates these benefits. The traditional solution calls for dedicated logic for each of the bus interface standards, requiring either a unique board dedicated for each standard or a single board with additional logic. ISP devices allow the design of a single generic bus interface, which can be configured in-system to interface with each of the bus standards, saving components, and cost.

Reconfigurability for Test

The ISP approach facilitates board-level testing and increases system fault coverage without sacrificing board resources or real estate. A diagnostic test pattern can be temporarily programmed into the ISP devices to exhaustively exercise board-level functions. Additionally, with ispGDS, programmable signal routing can be exploited in the test environment to perform enhanced board-level

Figure 3. ISP Manufacturing Flow vs. Standard Manufacturing Flow



test. For example, certain ispLSI devices may be configured by the tester to force test sequences into other portions of the board logic. The tester then monitors the response of this action and determines if the board passes or fails. This ability to detect board-level failures early in the manufacturing cycle reduces overall system cost. Once these detailed diagnostics are complete, the ISP devices can be reprogrammed to their normal logic configurations for final functional testing.

Boundary Scan

Complementing the ISP approach to board-level testing, IEEE Standard 1149.1 Boundary Scan technology (available with the 3000 series) enhances overall system quality. As component densities on the system boards increase, along with greater chip density and I/O, the ability to access and test critical nodes is impaired. With Boundary Scan Test, a serial interface through the test access port (TAP) simplifies field diagnostics and testing while costs are reduced. And because the same Boundary Scan serial path and control pins are used for implementing ISP programming, overall manufacturing costs are reduced as well.

Simplified Manufacturing Flow / No Bent Leads

At present, there are no automatic handlers capable of handling the programming of high lead-count, high-density Quad Flat Pack PLDs. As a result, all non-ISP high lead-count devices must be programmed by hand using a standard logic programmer.

It is a difficult task to insert a high lead-count, small lead-pitch device into a programming socket adapter, program, label (or mark) and reinventory the device without bending the delicate package leads. These bent leads can result in poor coplanarity and bad solder connections, increasing the amount of board and system-level troubleshooting required.

With the ISP devices, the parts go directly from the receiving dock to the manufacturing floor for placement on the PCB, entirely eliminating the stand-alone programming and mark operations and avoiding bent leads associated with misalignment of the device in the programmer socket. Unprogrammed ISP devices can be loaded into auto-insertion equipment and then placed directly onto the PCB without sockets or regard for the specific logic configurations. Individual device configurations can be downloaded from Automatic Test Equipment, PC, or workstation platforms at final board test. Programming of high-density PLDs containing thousands of gates takes only seconds.

System Upgrades and Repair

Lasting benefits from the use of ISP can be realized even after systems are shipped. In-system reprogramming can reduce field maintenance costs through enhanced field diagnostic capability, less costly product feature upgrades, and simpler maintenance procedures. Training, documentation, and on-going support can also be simplified by using the ISP approach to build in maintainability.

ISP Applications

Lattice's breadth of ISP device options, together with their leading-edge performance and features, have resulted in the design-in of ISP devices into a wide range of electronic systems. These applications include:

- Multimedia Video Editing
- Electronic Test Equipment
- Network Routers and Bridges
- Cellular Telephone Base Stations
- Telephone Switching Systems
- Hardware Accelerators
- Memory Subsystems
- Multi-Standard Video Frame Grabber
- Data Acquisition
- Image Processing

Why have designers embraced the ISP concept? For many, the manufacturing cost benefits, faster logic design and prototyping, and ability to reliably program high-pin count devices have been the most obvious benefits of employing ISP. However, in addition, ISP's ability to reconfigure systems immediately prior to and after shipment has begun to open up new possibilities.

For example, a very common but practical application for ISP comes from a company manufacturing traffic signal controllers. These controllers support priority "green lights" for emergency vehicles and buses through strobe light sensors that detect coded strobe sequences from the vehicles. The authorized sequences vary from city to city. ISP allows the sequence detector to be reprogrammed easily at the time the signal controller is shipped to a particular area or after it is installed. The alternative of custom-coded, traditional PLDs would result in significant additional effort and expense to customize the hardware of each system.

ISP Overview

E^{2}CMOS Technology

Lattice was the pioneer in electrically erasable CMOS (E^{2}CMOS) technology with the invention of the GAL device. With over 200 million devices shipped to customers, Lattice has the most CMOS PLD manufacturing experience in the industry. Merging E^{2}CMOS technology with ISP, Lattice conceived the in-system programmable, electrically erasable CMOS process. All ispLSI, ispGAL, and ispGDS devices are manufactured using Lattice's proprietary high-speed UltraMOS E^{2}CMOS technology. Lattice is unique among "fab-less" companies in that it performs its own process technology development. UltraMOS technology successfully combines the best features of CMOS and NMOS process technology to yield PLDs with the following key features:

- Industry Leading Performance
- High Logic Densities
- Low Power Consumption
- Fast Erase and Reprogram Times (Seconds)
- 100% Full Parametric Testability
- 100% Programming and Functional Yields

Lattice's experience in E^{2}CMOS manufacturing allows it to specify the best ISP parameters in the industry, including ISP programming over the full Commercial temperature range (0˚ to 70˚ C), a minimum of 1000 program / erase cycles for ispLSI devices (10,000 for ispGAL and ispGDS devices), and 20 year guaranteed program retention.

Unlike SRAM-based programmable devices, the non-volatility of E^{2}CMOS means there is never a need to reprogram ISP devices after a power-down and power-up sequence. In addition, there is no need for a separate memory component to store the logic program. A "security cell" feature is also available, allowing the device to be programmed, verified, and then secured. After the device has been secured, the binary pattern cannot be read from the device. However, even a secured device can be identified via the User Electronic Signature (UES). This field is reserved for the user to record product data such as code revisions and device functions.

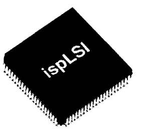
ISP Solutions from Lattice

Lattice offers five families of In-System Programmable devices (Table 1).

The ispLSI® Families of High-Density Programmable Logic from Lattice

The ispLSI families merge ISP technology with Lattice's high-performance, high-density pLSI® (programmable Large Scale Integration) architecture. The ispLSI devices are the first in-system programmable logic devices to combine the performance and ease of use of PLDs with the density and flexibility of FPGAs. Their powerful architecture can implement a wide range of logic functions including registers, counters, multiplexers, decoders, and complex state machines. With 135MHz system speed and logic densities ranging from 1,000 to 14,000 gates, they're the most powerful programmable logic components available today.

Table 1. ISP Solutions from Lattice



	The Premier High-Density PLD 1K, 2K, and 3K Families	**World's First In-System Programmable 22V10**	**In-System Programmable Switch Matrix**
Density (PLD Gates)	1-14K Gates	500 Gates	7x7-11x11 Matrix
Speed: f_{max} (MHz)	135	111	50
Speed: t_{pd} (ns)	7.5	7.5	7.5
Macrocells	32-320	10	N/A
Registers	32-480	10	N/A
Inputs + I/Os	34-160	22	14-22
Packages	44-, 68-, 84-Pin PLCC 44-, 100-, 176-Pin TQFP 120-, 128-Pin PQFP 160-, 208-Pin MQUAD 167-, 207-Pin CPGA	28-Pin PLCC	20-, 28-Pin PLCC 20-, 24-, 28-Pin PDIP

The ispGAL® Family of Low-Density Programmable Logic from Lattice

The ispGAL family brings ISP technology to Lattice's industry standard GAL® (Generic Array Logic) family of Low-Density PLDs. The ispGAL22V10 combines the full functionality of the popular GAL22V10 architecture together with ISP technology, while maintaining the GAL22V10 standard 28-pin PLCC package and footprint.

At 7.5ns Tpd and 111MHz system speed, the ispGAL22V10 is ideally suited for high-speed, small- to medium-scale logic functions typically found at the heart of today's microprocessor based systems.

The ispGDS™ Family of Programmable Digital Switches from Lattice

The ispGDS (in-system programmable Generic Digital Switch) family represents the expansion of ISP technology beyond system logic to system interconnect. This merger of ISP and a switch matrix architecture provides the ability to quickly implement and change p.c. board connections without changing mechanical switches or other system hardware. These high-performance, low-power, programmable digital switch devices are offered in a variety of matrix sizes and packages adding system flexibility to users.

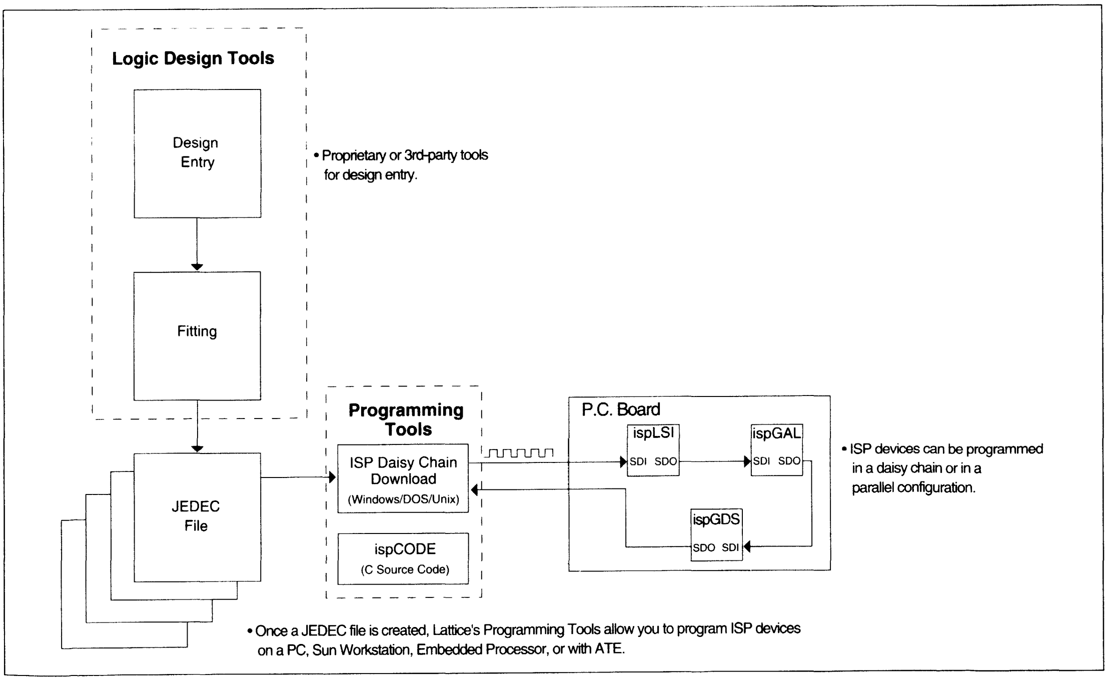
The ispGDS family is an ideal solution for easy, end-system feature configuration. With its 7.5ns performance, the ispGDS family also supports high-performance signal routing applications. The result is system hardware that can be reconfigured under software control without manual intervention.

ISP Implementation

ISP Interface

Programming and reprogramming ISP devices is simple and straightforward (Figure 4), requiring only a 5-volt power supply and a simple 4- or 5-wire serial interface depending on which ISP device is used. In-system programming operations such as PROGRAM, VERIFY, and ERASE are performed by passing commands and data to the ISP devices over the serial in-system programming interface (Figure 5). The basic programming signals consist of Serial Data In (SDI), MODE select (MODE), Serial Data Out (SDO) and Serial Clock (SCLK). In addition, $\overline{\text{ispEN}}$ is used to enable or disable the other four programming control signals on ispLSI devices, allowing these four pins to also function as dedicated inputs during normal operation.

Figure 4. ISP Design and Implementation Flow


Logic Design Tools
Design Entry
Fitting
• Proprietary or 3rd-party tools for design entry.
JEDEC File
Programming Tools
ISP Daisy Chain Download (Windows/DOS/Unix)
ispCODE (C Source Code)
P.C. Board
ispLSI
SDI SDO
ispGAL
SDI SDO
ispGDS
SDO SDI
• ISP devices can be programmed in a daisy chain or in a parallel configuration.
• Once a JEDEC file is created, Lattice's Programming Tools allow you to program ISP devices on a PC, Sun Workstation, Embedded Processor, or with ATE.

Device Programming Configurations

Single or multiple In-System Programmable (ISP) devices can be programmed in several configurations. Each ISP device can be programmed individually, through an independent ISP interface, or multiple devices can share a parallel multiplexed or serial daisy chained interface. The serial daisy chain configuration, shown in Figure 5, is the most efficient and easiest to implement, as it utilizes a simple hardware interface and programming procedures.

Multiple Programming Platforms

Lattice's ISP devices can be easily programmed on a wide variety of platforms:

PC and Workstation

Today, every engineer has a PC or workstation. With ISP, they save time and money by designing and prototyping logic on a single platform. Designers can enter and simulate designs using popular third-party CAE tools, automatically place-and-route the logic using Lattice's pDS or pDS+ tools, and then download the programming files to ISP devices without leaving their seat. With ISP, device programming is easy; Lattice's ispDOWNLOAD cable connects to the PC parallel port, or to the workstation's serial port (via the isp Engineering Kit Model 200), to give quick, easy and inexpensive programming of one or more ISP devices.

ATE

Lattice's ISP devices can also be programmed at final board test on an ATE, completely eliminating the need for a third-party device programmer for production. This streamlines the manufacturing flow and allows programmable devices for the first time to be treated like any other components on the board. To make the task easy, Lattice's ispCODE and tester programming utilities can be used to generate programming test vectors, or programming routines based on ispCODE can be written in the tester's high-level language.

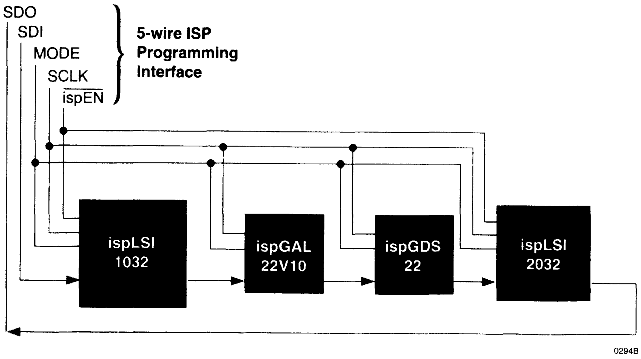
Embedded Processor

System Designers who want to be able to modify product features or upgrade their system hardware after the product has shipped to their customers will be interested in in-system programming using their product's own embedded processor. The product's embedded processor can be used to directly supply the ISP programming signals through a simple 4- or 5-bit port. Logic fuse maps and code can be stored in EPROM or other available system memory element. If the system has a modem or network link, remote download of new configurations from a central point is even possible. As result, systems no longer become obsolete as soon as they leave the factory, but can adapt and change to meet customers' growing needs for years to come.

Third-Party Programmer

Finally, ISP devices can be programmed by a number of popular third-party programmers. If a user already owns a third-party device programmer and is interested in the long-term benefits of ISP (like future field upgradability), but isn't ready to make the change yet, they can still program ISP devices like any other PLD on their device programmer. If they do need to change the device programming pattern at some future date, ISP will certainly make the process a lot easier: no device desoldering, no board swapping, no wasted effort.

Figure 5. In-System Programming Interface



ISP Development Tools

Design Entry and Fitter Software

To provide design support for in-system programmable devices, compiler support for ispGDS and ispGAL devices is available from numerous third-party compilers. ABEL, CUPL, LOG/iC, MINC, ORCAD PLD and others produce JEDEC files for these devices. Lattice also provides a compiler for the ispGDS family.

ispLSI designs can be quickly implemented using Lattice's low-cost pDS® development system or pDS+™ Fitter software that interfaces with third-party development software packages such as ABEL, Viewlogic, LOG/iC, Cadence, Synopsys, Mentor Graphics, Synario, and ORCAD (Figure 6), and produces standard JEDEC programming files.

ispStarter™ Kits

Lattice has also introduced products called ispStarter Kits that contain all the software, download cables, samples, and data sheets needed to implement ISP for the first time. These ispStarter Kits, available in pDS- and ABEL-compatible versions, allow system designers to try out ISP for themselves at a price of only $99.

Programming Tools

Once the JEDEC file has been generated for a given design, the design information must be programmed into the proper device. Lattice supports programming the ISP devices on a PC, workstation, embedded processor, ATE or third-party programmer with several programming tools.

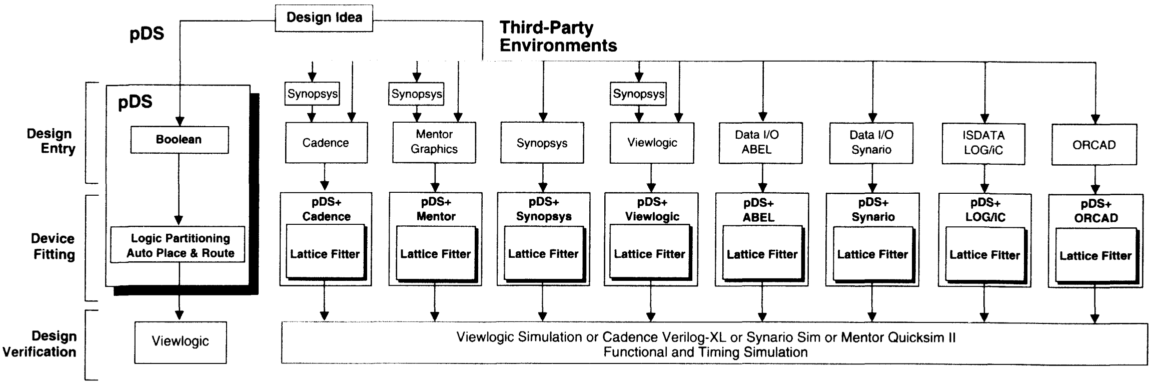
ispCODE

Lattice provides a library of programming routines written in ANSI-standard C++ language (called ispCODE™) which can be easily incorporated into a system or tester software to support programming of the ISP devices. These routines include such common operations as Program, Read, Verify, Erase, and Secure. After completion of the logic design and creation of a JEDEC file, in-system programming can be accomplished on customer-specific hardware: UNIX systems, PCs, testers, or embedded systems. The ispCODE software package supplies specific routines, with extensively commented source code, for incorporation into application programs. These routines provide flexible, easy-to-use program modules which support the programming of a single device or multiple devices on a board. Example programs are included to demonstrate the use of each routine. One of these programs, when compiled, produces a Windows application called ISP Serial Programmer, which not only programs, reads, and verifies ISP devices using the parallel port of a PC, but also supports ATE programming.

ISP Daisy Chain Download Software

Lattice also provides ISP Daisy Chain Download software, a Windows-based executable utility which supports

Figure 6. Development Tools Available from Lattice



programming of all Lattice ISP devices in a serial daisy chain configuration in a PC environment.

isp Engineering Kits and Download Cable

Lattice's isp Engineering Kits for ispLSI devices function as device programmers in conjunction with a PC or workstation. Or, they can be used for direct download to an ispLSI device on your board. The isp Engineering Kits interface with either a PC or workstation and consist of a programming module, download cables and socket adapters which are used to program any of the Lattice ispLSI devices. The ispDOWNLOAD™ cable supports programming of any ISP device directly on the PCB.

Future of ISP

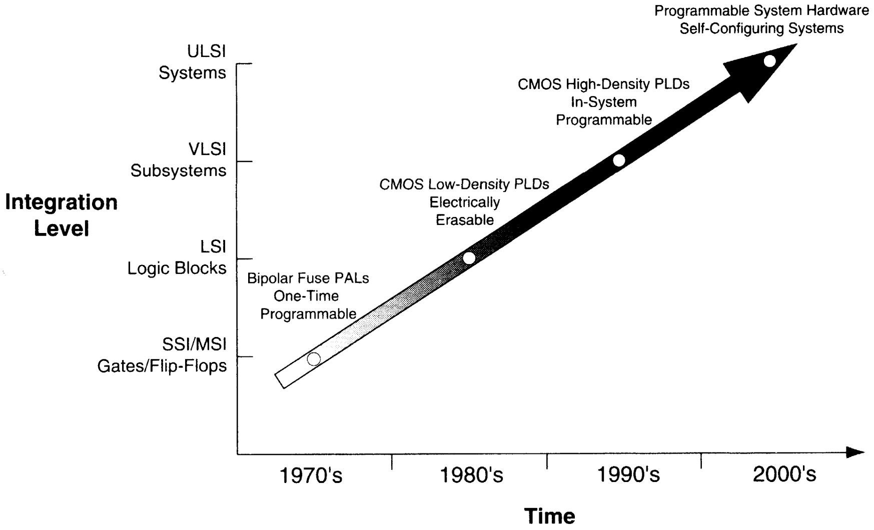
In-system programmability is the logical evolution of programmable device technology. PLDs have evolved from the fuse-based, one-time programmable devices invented in the 70's, to the electrically erasable components of the 80's, and now the in-system programmable devices of the 90's. The time and expense benefits of employing ISP make its widespread use inevitable.

But what about the future applications that ISP opens up?

To begin with, system designers will need to adopt a new mindset to exploit hardware that can evolve after the product is shipped. No longer will a hardware design be "frozen" as it is today; features will change based on updates transmitted from a central site or the system microprocessor can reconfigure peripheral functions in response to application needs. The use of ISP will drive hardware designs to more generic configurations that are given their "personality" through in-system programmable logic and interconnect components. Entire boards will be able to reconfigure microprocessor, memory and peripheral functions for any application. Just as a PC can run CAE tools, financial spreadsheets, games and multimedia software depending on the need, ISP hardware will be able to solve a broad range of problems just by downloading a new personality.

Ultimately, ISP will make the term "hardware" an anachronism, and usher in the era of truly programmable systems.

Programmable Technology Evolution



Section 1: ISP Overview

Section 2: The Basics of ISP

Section 3: ISP Programming Options

Section 4: Application Notes and Article Reprints

Section 5: General Information

Index

The Basics of ISP

Introduction

This section describes the details of programming with Lattice's In-System Programmable (ISP™) devices. It is organized into three sections. The first section summarizes the ISP design flow. The next section describes ISP hardware interface basics, including discussions on both key issues required to get started with ISP quickly, as well as detailed ISP information for those interested in a thorough understanding of ISP at the device level. The final section focuses on ISP software, which summarizes all the development tools available to support easy implementation of the Lattice ISP solutions.

ISP Design Flow

As with other Programmable Logic Devices (PLDs), the ISP design flow includes design entry using CAD software, compiling and fitting the design, generating a JEDEC standard fuse map file, and programming the device (Figure 1).

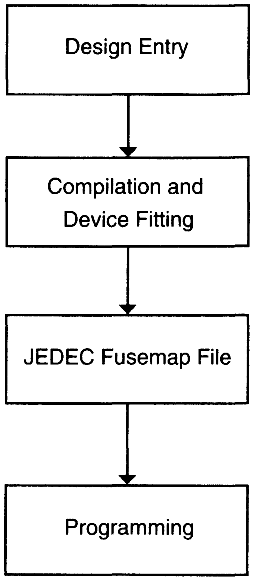
Creating a JEDEC Fuse map File

As part of any PLD design flow, the logic design must be entered through any combination of VHDL, schematic, Boolean equation, state machine, or truth table design entry. Lattice has various third-party and proprietary software packages which support these design methodologies. Each of these design packages will take a design and generate a standard JEDEC fuse map for programming. Up to this point in the logic design process, ISP devices share the same design flow as standard programmable logic devices.

Programming

Programming consists of converting the JEDEC fuse map file into a serial data format and shifting that serial data into the device. The ISP programming software automatically converts the JEDEC fuse map file into the serial data format which is programmed into the ISP device. However, a JEDEC file is made up of ASCII characters which use a relatively large amount of space, especially in environments in which storage space for the fuse map information is limited. In order to support these storage-critical environments, Lattice has defined an ispSTREAM™ data format which represents each fuse location with a single bit instead of an ASCII character. The ISP programming software also accepts this ispSTREAM format for programming.

Figure 1. ISP Design Flow



Once the fuse map is ready, it is just a matter of serially shifting the data into the device along with the appropriate addresses and commands. The basic ISP interface uses four wires to shift the JEDEC fuse map data into the device. An additional fifth wire is used by ispLSI devices, employing an active low ISP Enable ($\overline{\text{ispEN}}$) signal as a mode control to put the device into programming or normal operation mode.

The MODE signal, along with SDI, controls the ISP device's internal state machine to step through the ISP programming process. The entire ISP programming process is controlled by a three-state state machine. The three states are Idle State, Shift State, and Execute State. Transitions between these states are controlled by the MODE and SDI inputs along with SCLK for synchronization. ISP commands such as Program, Bulk Erase, Verify, Data Shift, and Address Shift are executed through the device's instruction registers and ISP interface input pins.

The Basics of ISP

Traditionally, programmable logic devices have been programmed on PLD/PROM programmers which require that all programming signals and algorithms be generated by the programmer. The programmer also generates the external super voltage or high voltage required by non-ISP devices (typically 12-14 volts). This super voltage requirement is one of the main reasons dedicated programmers are used to program conventional PLDs.

However, with ISP devices, the ISP programming super voltage is generated within the device from the 5-volt power supply. This internal super voltage generation teamed with Lattice's unique serial ISP programming interface enables designers to program any ISP device using a simple four- or five-wire interface in which all the programming signals are driven by standard TTL logic levels (5 volts).

The details of device programming are transparent to the user if ISP programming software such as ISP Daisy Chain Download and ispCODE™ C++ Source Code are used. These software tools drive the four or five ISP programming signals in accordance with the programming specifications and the state machine requirements. The ISP Daisy Chain Download software generates the ISP signals with proper timing through the use of the PC parallel port. ispCODE can be ported to any hardware platform required and can be used to generate ISP programming signals using whatever hardware is available.

Lattice also supports the use of other programming interfaces such as Automatic Test Equipment (ATE). Specifically for testers, Lattice provides JEDEC file conversion routines to tester-acceptable formats. In addition, testers which accept high-level languages can be programmed using ispCODE C++ routines as a model for structuring test programs.

These and other topics are covered in the next two sections, "Hardware Basics" and "Software Basics."

Hardware Basics

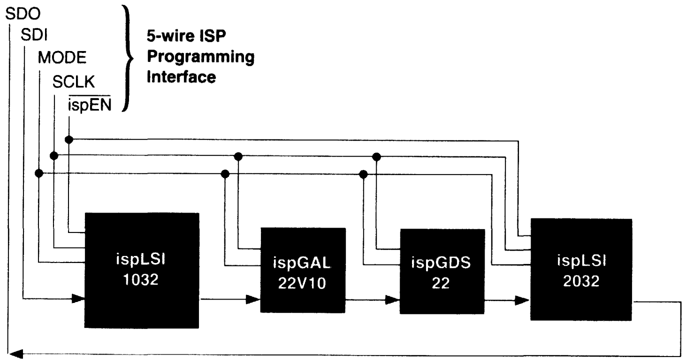
Introduction

This section describes programming Lattice ISP devices from a hardware point of view. It is divided into two subsections. The first subsection "Getting Started Fast" is intended to give the reader enough ISP hardware information to easily implement Lattice's ISP solutions using the Lattice ISP tools. The second subsection "ISP Expert" gives more details on low-level, device-specific programming algorithms. Since these algorithms are transparently handled by Lattice's programming tools, the second subsection is intended for those readers who want a thorough understanding of the programming procedures, which would be required for any custom implementation of ISP.

Subsection I "Getting Started Fast" **Page**

Subsection II "ISP Expert" **Page**

Figure 1. Multiple ISP Device Programming Interface



In-System Programming (ISP) Interface

Programming Lattice's ispLSI, ispGAL, and ispGDS devices is based on a simple serial ISP programming interface (Figure 1). The basic elements of the ISP programming interface are the mode control (MODE), serial data in (SDI), serial data out (SDO), and serial clock (SCLK) inputs, as well as a three-state programming control state machine integrated into each ISP device. Lattice's ISP devices utilize nonvolatile E^2CMOS technology and require only five-volt, TTL-level programming signals from the ISP interface for in-system programming. The internal three-state state machine, which determines whether the device is in the normal operation state or in the programming states, is controlled by the four ISP programming pins. MODE and SDI furnish control inputs to the state machine, SDI and SDO make up the programming data inputs and outputs to and from an internal shift register, and SCLK provides the clock. ispLSI devices use a fifth programming pin, $\overline{\text{ispEN}}$, to multiplex the functions of the SDI, SDO, SCLK, and MODE pins between ISP functions during programming and user-defined logic functions during normal PLD operation.

The internal state machine controls the sequence of programming operations such as identifying the ISP device, shifting in the appropriate data and commands, controlling the internal programming pulse widths to ensure proper programming voltage margins, and erasing the device. Programming consists of shifting the logic implementation stored in a JEDEC file into the device serially through the SDI pin along with the appropriate address and commands, programming the data into the E^2CMOS logic elements, and shifting the data from the logic array out through the SDO pin for verification.

ISP Programming Pins

The programming pins used to program Lattice devices are each described in detail in this section. Figure 2 shows the ispLSI 1032 84-Pin PLCC device pinout.

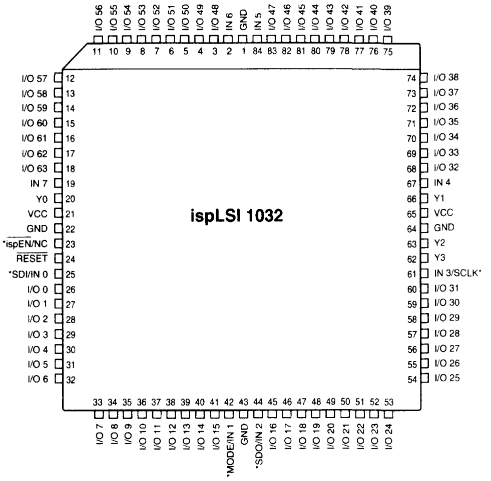
The Serial Data In (SDI) pin performs two different functions. First, it acts as the data input to the serial shift register built inside each ISP device. Second, it functions as one of the two control pins for the programming state machine. Because of this dual role, the function of SDI is controlled by the MODE pin. When MODE is low, SDI becomes the serial input to the shift register, and when MODE is high, SDI becomes a control signal for the programming state machine. Internally, the SDI signal is multiplexed to various shift registers in the device. The different shift instructions of the state machine determine which of these shift registers receives input from SDI.

The MODE signal, combined with the SDI signal, controls the programming state machine, as described in the "ISP State Machine Operation" section which follows.

The Serial Clock (SCLK) pin provides the serial shift register with a clock. SCLK is used to clock the internal serial shift registers and clock the ISP state machine between states. State changes and shifting data in are performed on low-to-high transitions. When MODE is high, SCLK controls the programming state machine, and when MODE is low, SCLK acts as a shift register clock to shift data in or out or to start an operation. When shifting data out, the data is available and valid on SDO only after a subsequent high-to-low transition of SCLK.

The Serial Data Out (SDO) pin is connected to the output of the internal serial shift registers. As previously stated, the selection of which shift register to output is determined by the ISP state machine's shift instruction. When MODE is driven high, SDO connects directly to SDI, bypassing the device's shift registers.

Figure 2. ispLSI 1032 84-Pin PLCC Pinout Diagram

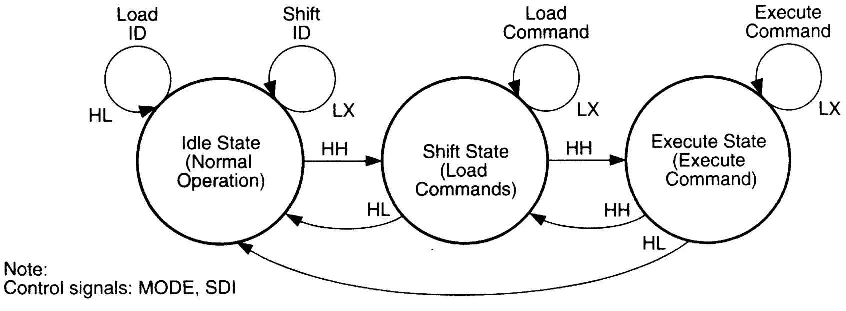


The ispEN pin, only utilized on the ispLSI devices, determines which mode the device is in, namely *Edit Mode* (ISP programming mode) or *Normal Mode* (normal device operation mode). When ispEN is driven low on an ispLSI device, the device I/O pins are put into a high impedance state (by internal active pull-up resistors equivalent to 100KΩ) and the device enters Edit Mode.

ISP State Machine Operation

The programming state machine controls which mode the device is in, and provides the means to read and write data to the device (Figure 3). Four ISP programming pins are used to load and unload data, and to control the state machine. The three states defined in the state machine diagram are the IDLE State, SHIFT State, and EXECUTE State. Instruction codes, which are shifted into the device in the SHIFT state, control which instruction is to be

Figure 3. Programming State Machine



Load ID
Shift ID
Load Command
Execute Command
HL
LX
LX
LX
Idle State (Normal Operation)
HH
Shift State (Load Commands)
HH
Execute State (Execute Command)
HL
HH
HL

Note:
Control signals: MODE, SDI

executed in the EXECUTE state. In the SHIFT and EXECUTE states, all the I/O pins are 3-stated. To transition between states, MODE is held high, SDI is set to the appropriate level, and SCLK is clocked. The ispGAL22V10 and ispGDS devices, unlike ispLSI devices which employ an $\overline{\text{ispEN}}$ input pin, rely on the state machine to put the device I/O pins in a high impedance state. The IDLE state puts the ispGAL and ispGDS devices into Normal Mode, and the remaining two states put the devices into ISP programming mode, which places the device I/O pins in the high impedance state.

Idle/ID State

The Idle/ID state is the first state activated when the device enters the Edit Mode (ISP programming mode). The state machine is in the Idle/ID state when the device is idle, in the Edit Mode, or when the user needs to read the device identification (each ISP device type is assigned a unique identification code. See the "ISP Expert" section). The eight-bit device identification is loaded into the shift register by driving MODE high, SDI low, and clocking the ISP state machine with SCLK. Once the ID is loaded, it is read out serially by driving MODE low. Notice that when the device ID is read serially, SDI can either be high or low (called "don't care") and the state machine needs only seven clocks to read out eight bits of device ID. The default state for the control signals is MODE high and SDI low. State transition to the Command Shift State occurs when both MODE and SDI are high while the ISP state machine gets a clock transition. As with most shift registers, the Least Significant Bit (LSB) of the ID gets shifted out from SDO first.

Command Shift State

This state is strictly used for shifting instructions into the state machine. The entire instruction sets for the ispLSI, ispGDS, and ispGAL devices are listed in the "ISP Expert" section. When MODE is low and SDI is "don't care" in the Command Shift State, SCLK shifts the instruction into the state machine. Once the instruction is shifted into the state machine, the state machine must transition to the Execute State to execute the instruction. Driving both MODE and SDI high and applying the clock transfers the state machine from the Command Shift State to the Execute State. If needed, the state machine can move from the Command Shift State to the Idle/ID State by driving MODE high and SDI low.

Execute State

In the Execute State, the state machine executes instructions that are loaded into the device in the Command Shift State. For some instructions, the state machine requires more than one clock to execute the command. An example of this multiple clock requirement is the address or data shift instruction. The number of clock pulses required for these instructions depends on the device shift register sizes. When executing instructions such as Program, Verify, or Bulk Erase, the necessary timing requirements must be followed to make sure that the commands are executed properly. For specific timing information refer to the appropriate data sheets.

To execute a command, MODE is driven low and SDI is “don't care.” For multiple clock instructions, the control signals must remain in the same state throughout the duration of the execution. MODE high and SDI high will take the state machine back to the Command Shift State and MODE high and SDI low will take the state machine to the Idle/ID State.

ISP Device Programming Configurations

Serial Daisy Chain

Advantages

One of the main advantages of daisy chained ISP programming is the simplified hardware interface. The number of ISP devices that can be connected to the same serial interface is limited only by the signal drive capability of the ISP programming control logic. One serial daisy chain is capable of providing the necessary programming interface, minimizing the hardware overhead for in-system programming. Software controls generated from PCs, microcontrollers, and test equipment can program and reconfigure ISP devices during various board-level design, test, and manufacturing stages.

Programming Configuration

As shown previously in Figure 1, all the MODE, SCLK, and $\overline{\text{ispEN}}$ (if using ispLSI devices) pins of the ISP devices are connected to the ISP interface, and the first device's SDO is connected to the second device's SDI, and each following SDO to the SDI of the next ISP device. This configuration allows a large string of ISP devices to be programmed, in-system, in a serial daisy chain.

Parallel

For low-density ISP devices daisy chain programming is the most common configuration, but for high-density devices, with multiplexed programming and logic pins and the ispEN feature, other programming configurations are also common. ISP devices can be programmed in one of two parallel configurations. The first parallel configuration, called Dedicated ISP Pins, dedicates all ISP programming pins to programming. The second parallel configuration, called Parallel Multiplex below, is mainly used for ispLSI devices. In this configuration, the functions of the ISP programming pins can be multiplexed between acting as programming pins and acting as inputs for normal logic functions.

Dedicated ISP Pins

Figure 4 illustrates one configuration for programming multiple ISP devices, where the ISP programming pins (MODE, SDI, SDO, and SCLK) are dedicated to programming functions. Although this scheme precludes the use of the ISP programming control signal pins as separate dedicated inputs for system logic functions on ispLSI devices, it is the easiest to implement. Each of the four programming control signal pins in each ISP device is connected (i.e. SDI of the ispLSI 1032 is connected to SDI of the ispLSI 1048 and SDI of the ispLSI 1016; MODE of the ispLSI 1032 is connected to MODE of the ispLSI 1048 and MODE of the ispLSI 1016; etc.). With this scheme, the ispEN signal for each ispLSI device is enabled (ispEN low) separately, and one device is placed in Edit (ISP programming) Mode at a time. With one device in Edit Mode, the other devices will be in Normal Mode and can continue to perform normal system logic functions. ispEN is the only programming interface signal that cannot be used for general logic functions.

Parallel Multiplex

Figure 5 illustrates a multiplexing scheme which allows the user to control the ISP programming through multiple independent ispEN signals for the ispLSI devices. The multiple ispEN signals not only control the ispEN inputs of the ispLSI devices, but also act as the control signals for multiplexing the functional and ISP programming signals. This scheme differs from the previous one in that the ISP programming signals are not dedicated to programming. Instead, the ISP programming signals MODE, SDI and SCLK function as inputs for both normal functional mode and the ISP programming mode. SDO, however, functions as an input in normal functional mode and as an output in ISP programming mode. Figure 5 also shows the difference in controlling these different programming signals. Please note that when multiplexing the programming interface signals, the input driving the SDO pin must be 3-stated during programming to avoid signal contention. As previously stated, the ISP programming pins on the ispGAL and ispGDS devices are dedicated to ISP programming, so this configuration is not utilized often for the ispGAL and ispGDS devices. The concept can be modified to multiplex the MODE pin instead of the ispEN pin and becomes useful in some ispGAL and ispGDS applications.

Figure 4. Dedicated ISP Pins Configuration

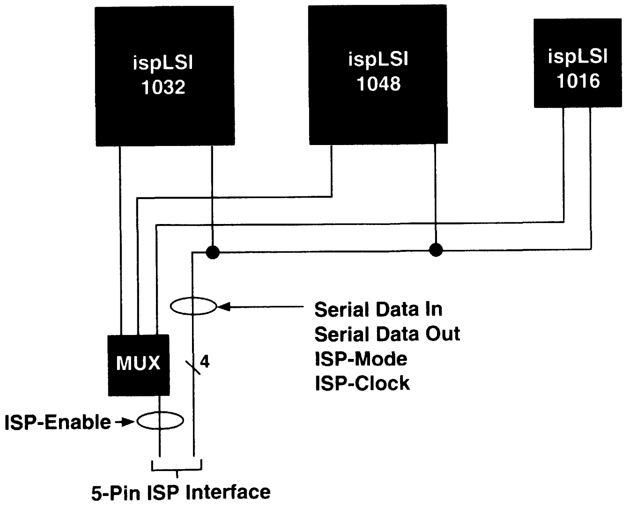
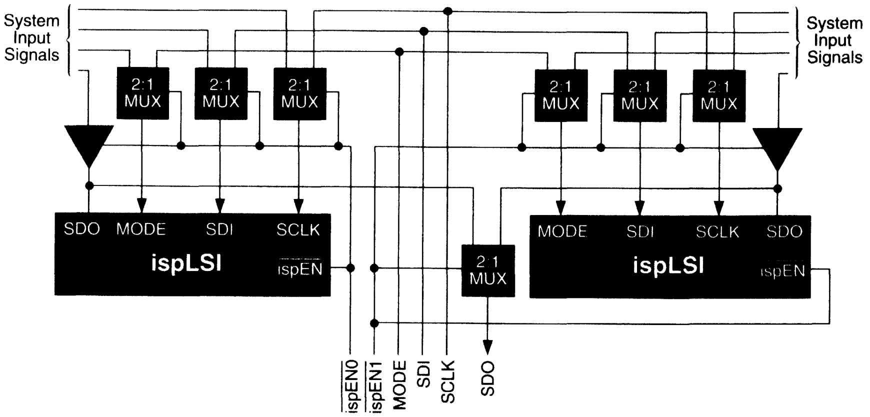


Figure 5. Parallel Multiplex Configuration



Hardware Considerations

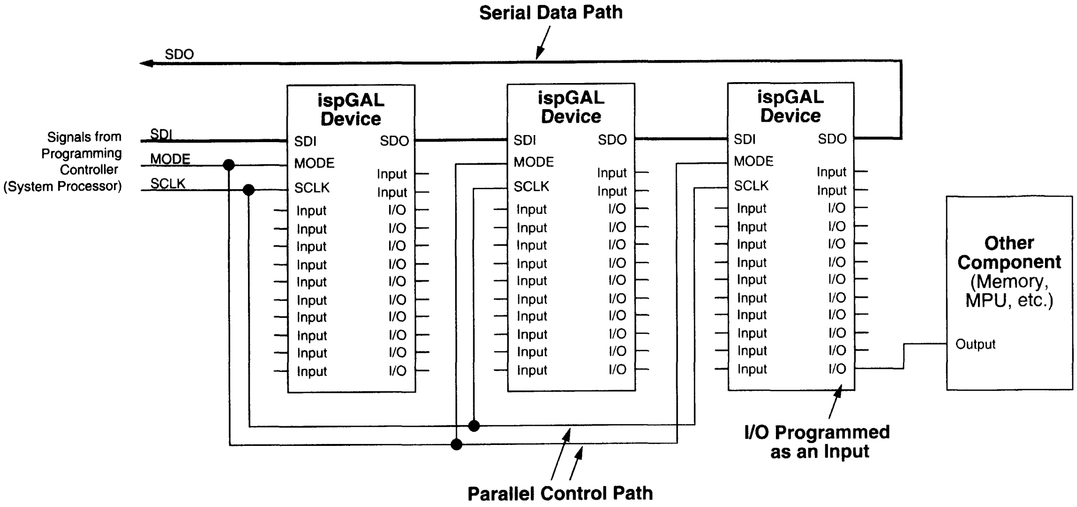
Lattice's In-System Programming (ISP) technology makes the use of Programmable Logic incredibly simple. Using ISP, multiple devices can be programmed using a single serial daisy chain programming loop. However, as with any high performance semiconductor component, systems must be designed to insure good signal integrity without signal conflicts between components. By doing so, reliable operation can be obtained over a wide range of operating conditions. This section discusses some basic programming hardware issues which should be considered when implementing a system using Lattice ISP.

All ISP programming specifications such as the programming cycle and data retention are guaranteed when programming ISP devices over the commercial temperature range (0 to 70° C). It is critical that the programming and bulk erase pulse width specifications are met by the programming platform to insure proper in-system programming. Lattice's ISP Daisy Chain Download and ispCODE software insures that these specifications are met when using a PC programming platform.

When using the ispDOWNLOAD cable in a daisy chained configuration, Lattice recommends using a maximum of eight ISP devices in a single chain. This is to insure proper programming signal integrity (pulse width, shape, etc.) at the ISP devices. The eight devices can be any combination of ispLSI, ispGAL, and ispGDS devices arranged in any order. The recommended number of devices is based on a typical system board environment with proper signal terminations and typical trace lengths. The actual number of devices that can be programmed in a serial chain may vary according to the system board environment. When using more than eight devices, additional buffering of the ISP programming signals is recommended. Alternatively, multiple programming loops can be employed which are electrically isolated from one another.

I/O pins on ISP devices may be defined as inputs once the devices are programmed. As a result, they typically will be driven by the outputs of other components once mounted on the board. Care must be taken to ensure that I/O pins are not enabled prematurely during programming. To do so when the device is partially programmed can cause contention with other signal drivers since I/O pins destined to be configured as inputs may not be 3-stated yet. This conflict can cause improper device programming and potential damage (Figure 6).

Figure 6. ISP Serial Daisy Chain


Serial Data Path
SDO
Signals from Programming Controller (System Processor)
SDI
MODE
SCLK
ispGAL Device
SDI
SDO
MODE
SCLK
Input
I/O
Other Component (Memory, MPU, etc.)
Output
I/O Programmed as an Input
Parallel Control Path

All ISP devices are shipped from Lattice with a fuse pattern that will put all I/O pins in the high impedance state prior to programming. The output 3-state is controlled by the ispEN signal on the ispLSI devices. For the ispGAL and ispGDS devices, the output 3-state is controlled by the programming state machine (Shift and Execute states 3-state the devices). When implementing custom ISP programming code, it is important for the ispGAL and ispGDS that the ISP state machine be kept within the Shift and Execute states until the completion of programming. This procedure keeps the partially programmed device or devices from conflicting with other components on the board.

ISP programming signal default states must be maintained during normal device operation. The ispEN pin on the ispLSI devices has an internal pull-up to place the devices in normal functional mode when the pin is not driven externally. The ispGAL and ispGDS devices' MODE or SDI signals must be tied low through a 1.2KΩ pull-down resistor during normal functional mode (on-chip pull-downs are not provided). It is not acceptable to let these pins float during normal operation. In addition, it is recommended that the ispDOWNLOAD cable have its ispEN signal tied to a decoupling capacitor (.01µF) to ground on the system board.

Hardware Programming Tools

isp Engineering Kits

Lattice provides both a PC-based (Model 100) and a Sun Workstation-based (Model 200) isp Engineering Kit. The isp Engineering Kits function as stand-alone device programmers for prototyping.

isp Engineering Kit Model 100

The isp Engineering Kit Model 100 provides designers with a quick and inexpensive means of evaluating and prototyping new designs using Lattice ispLSI devices. This kit is designed for engineering purposes only and is not intended for production use. The kit programs devices from the parallel printer port of a host PC using the Lattice pDS or pDS+ PC-based designs tools. By connecting a system cable (included) from the host PC to the isp Engineering Kit, or connecting from the host PC to the target device on the system board, a JEDEC file can be easily downloaded into the ispLSI device(s) (Figure 7).

isp Engineering Kit Model 200

The isp Engineering Kit Model 200 provides a prototyping solution for UNIX systems. This easy-to-use, inexpensive kit is designed for evaluating and prototyping new designs using Lattice ispLSI and pLSI devices. It is intended for engineering purposes only and is not intended for production use. The kit programs devices from the RS-232 serial port of a host workstation using the pDS+ workstation-based design tools by connecting a system download cable (included) from the host workstation to the isp Engineering Kit.

ispDOWNLOAD Cable

The ispDOWNLOAD Cable product is designed to facilitate in-system programming of all Lattice ISP devices on a printed circuit board directly from the parallel port of a PC. After completion of the logic design and creation of a JEDEC file by a logic compiler such as the pDS, pDS+ Fitter or ispGDS Compiler software, Lattice's ISP Daisy Chain Download Software programs devices on the end-product p.c. board by generating programming signals directly from the parallel port of a PC which then pass through the ispDOWNLOAD Cable to the device. With this cable and a connector on the p.c. board, no additional components are required to program a device (Figure 8).

Figure 7. ispEngineering Kit Model 100

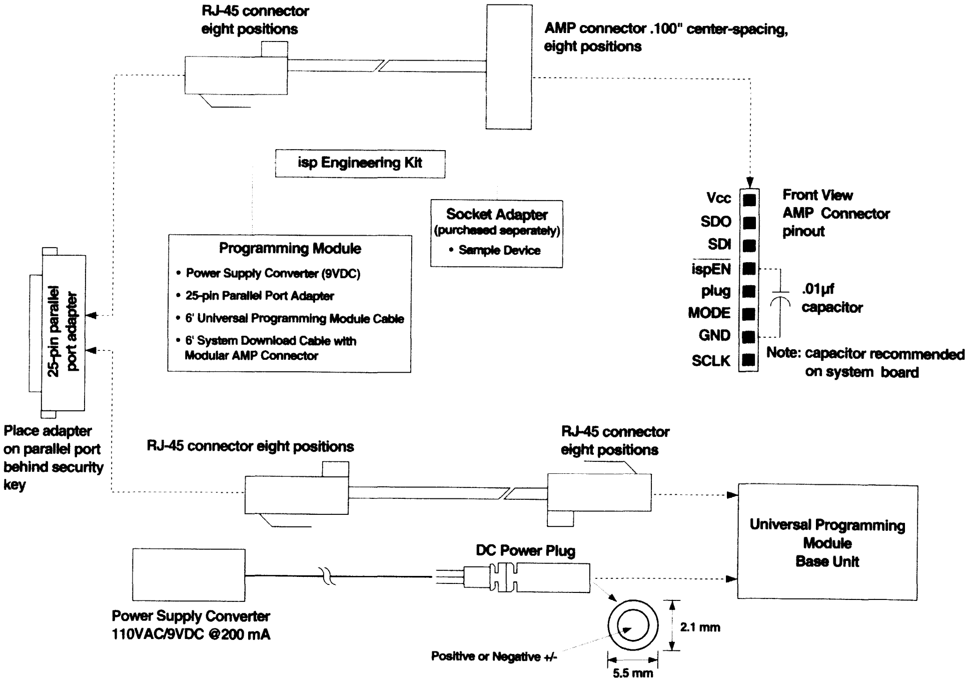
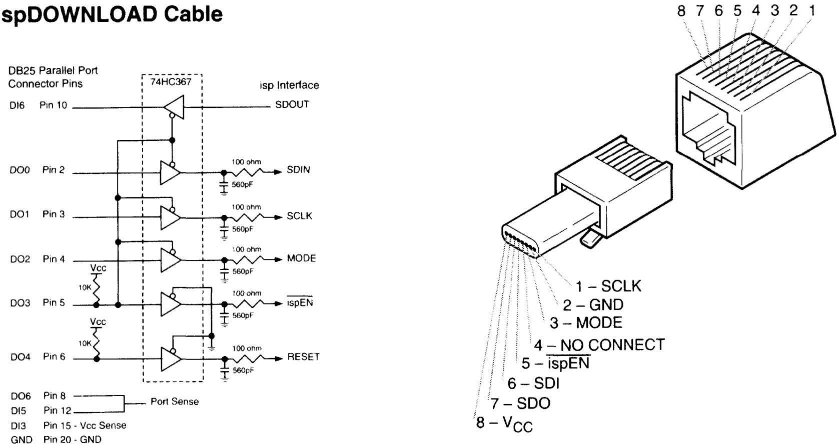

RJ-45 connector
eight positions
AMP connector .100" center-spacing, eight positions
isp Engineering Kit
Socket Adapter
(purchased seperately)
• Sample Device
Programming Module
• Power Supply Converter (9VDC)
• 25-pin Parallel Port Adapter
• 6' Universal Programming Module Cable
• 6' System Download Cable with Modular AMP Connector
25-pin parallel port adapter
Place adapter on parallel port behind security key
Vcc
SDO
SDI
ispEN
plug
MODE
GND
SCLK
Front View
AMP Connector
pinout
.01µf
capacitor
Note: capacitor recommended on system board
RJ-45 connector eight positions
RJ-45 connector
eight positions
Universal Programming Module
Base Unit
DC Power Plug
Power Supply Converter
110VAC/9VDC @200 mA
Positive or Negative +/-
2.1 mm
5.5 mm
0813A

Figure 8. ispDOWNLOAD Cable



Note: The pin numbers in Figure 8 are for reference only. Do not use pin numbers as the socket pinout for board layout.

ispStarter Kits

The ispStarter Kits are designed to make Lattice's innovative in-system programmable device technology available in a single, complete package. The isp Starter Kits contain all the software, hardware, device samples, and information you need to begin designing with Lattice's ISP products.

The ispStarter Kits include pDS Starter or pDS+ ABEL logic development software for ispLSI 1016 and ispLSI 2032 devices, ispGDS compiler software, ispCODE, isp Daisy Chain Download Software, an ispLSI 2032-80LJ, an ispGAL22V10B-15LJ, an ispGDS14-7J, and an ispDOWNLOAD Cable.

Programming Times

The ISP programming times can be approximated by the number of rows that are required to program on a given device and the programming pulse width. Assuming that the overhead of shifting data and other miscellaneous functions are an order of magnitude smaller in time duration and therefore negligible, the total programming time ranges can be calculated as shown in Table 1.

Table 1. Programming Times of ISP Devices

Device	Total Programming Time in Seconds (40ms Programming Pulse)
ispGDS	< 1
ispGAL22V10	1.84
ispLSI 1016	7.68
ispLSI 1024	8.16
ispLSI 1032	8.64
ispLSI 1048	9.60
ispLSI 1048C	12.4
ispLSI 2032	8.16
ispLSI 3256	14.4

User Electronic Signature (UES)

The Lattice ispGAL, ispGDS, and ispLSI families can ease problems associated with document control and device traceability, thanks to a feature called the User Electronic Signature (UES).

The UES is basically a user's "notepad" provided in electrically erasable (E^2) cells on each ispGAL, ispGDS, and ispLSI device. The UES consists of an extra row that is appended to the programmable array and allocated for data storage. The physical size of the UES varies by device type. Table 2 indicates the various sizes of the UES.

In the course of system development and production, the proliferation of PLD architectures and patterns can be significant. To further complicate the record-keeping process, design changes often occur, especially in the early stages of product development. The task of maintaining which pattern goes into what device for which socket becomes exceedingly difficult. What's more, once a manufacturing flow has been set, it becomes important to "label" each PLD with pertinent manufacturing information, which is beneficial in the event of a customer problem or return.

Table 2. UES Sizes

ISP Device	UES Size (bits)
ispGAL 22V10	64
ispGDS	32
ispLSI 1016	80
ispLSI 1024	120
ispLSI 1032	160
ispLSI 1048	240
ispLSI 1048C	240
ispLSI 2032	40
ispLSI 3256	338

Lattice incorporated the UES to store such design and manufacturing data as the manufacturer's ID, programming date, programmer make, pattern code, checksum, PCB location, revision number, and/or product flow. This assists users with the complex chore of record maintenance and product flow control. In practice, the UES can be used for any of a number of ID functions.

Within the various bits available for UES data storage, users may find it helpful to define specific fields to make better use of the available storage. A field may use only one bit (or all bits), and can store a wide variety of information. The possibilities for these fields are endless, and their definition is completely up to the user.

Even with the device's security feature enabled, the UES can still be read. With a pattern code stored in the UES, the user can always identify which pattern has been used in a given device. As a second safety feature, when a device is erased and re-patterned, the UES row is automatically erased. This prevents any situation in which an old UES might be associated with a new pattern.

It is the user's responsibility to update the UES when reprogramming. It should be noted that UES information will be included in the checksum reading. Therefore, when the UES is modified the checksum will also change.

The UES may be accessed (read or write) through one of three methods. First, most third-party programmers support the UES option through the programmer's user interface, so programming or verifying the UES is as simple as programming or verifying any other array. Second, the UES may be embedded within the JEDEC file by selecting the proper fuse locations in the fuse map. Third, the UES can be written or read using Lattice's ispCODE software with routines provided in the ispCODE library. Further information on using ispCODE software to program the UES can be found in the Lattice Data Book.

ispLSI Programming Details

The following sections describe the programmable state machine instruction set, timing parameters, device layout, and programming algorithms as they apply to ispLSI devices in general. Table 3 lists the eight-bit device ID's for all the ispLSI devices.

Table 4 lists the instructions that can be loaded into the state machine in the Command Shift State and then executed in the Execute State. Notice that the device identification is read during the Idle/ID State, and this operation does not require an instruction.

While it is possible to erase the individual arrays of the device, it is recommended that the entire device be erased (UBE) and programmed in one operation. This Bulk Erase operation should precede every programming cycle as an initialization.

When a device is secured by programming the security cell (PRGMSC), the on-chip verify and load circuitry is disabled. The device should be secured as the last procedure, after all the device verifications have been completed. The only way to erase the security cell is to perform a bulk erase (UBE) on the device.

Timing

When programming ispLSI devices, there are several critical timing parameters that must be met to ensure proper programming. The two most critical parameters are the programming pulse width (t_{pwp}) and the bulk erase pulse width (t_{bew}). These pulse widths determine the programming and erasing times of the E^2 cells. Figure 9 shows these critical program and erase timing specifications.

Table 3. ispLSI Device ID Codes

Device	MSB LSB
ispLSI 1016	00000001
ispLSI 1024	00000010
ispLSI 1032	00000011
ispLSI 1048	00000100
ispLSI 1048C	00000101
ispLSI 2032	00010101
ispLSI 3256	00100010

In addition to the two programming and erasing specifications, the following timing specifications must be met.

t_{isp} - Specifies the time it takes to get into the ISP mode after $\overline{\text{ispEN}}$ is activated. Or, the time it takes to come out from the ISP mode after $\overline{\text{ispEN}}$ becomes inactive.

t_{su} - Set up time of the control signals before SCLK. Or, the set up time of input signals against other control signals (if applicable).

t_h - Hold time of the control signal after SCLK. It also applies to the same input signals from the set up time.

t_{clkl} - Minimum clock pulse width, low.

t_{clkh} - Minimum clock pulse width, high.

t_{pwv} - Verify or read pulse width. The minimum time requirement from the rising clock edge of a verify/load instruction execution to the next rising clock edge (Figure 9).

t_{rst} - Power on reset timing requirement. t_{rst} must elapse after power up before any operations are performed on the device.

All the programming timing parameters are summarized in the timing diagram (Figures 9 and 10).

Figure 9. ispLSI Program, Verify & Bulk Erase Timing

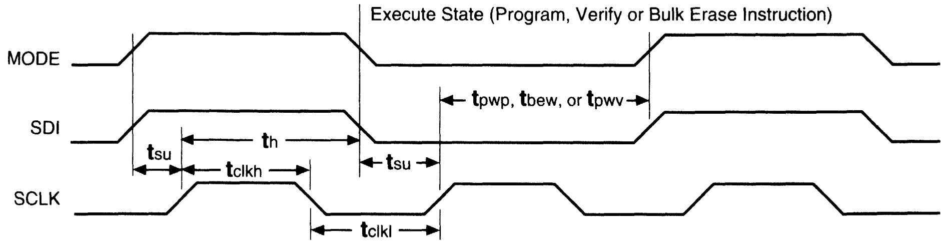


Figure 10. ispLSI Programming Timing Requirements

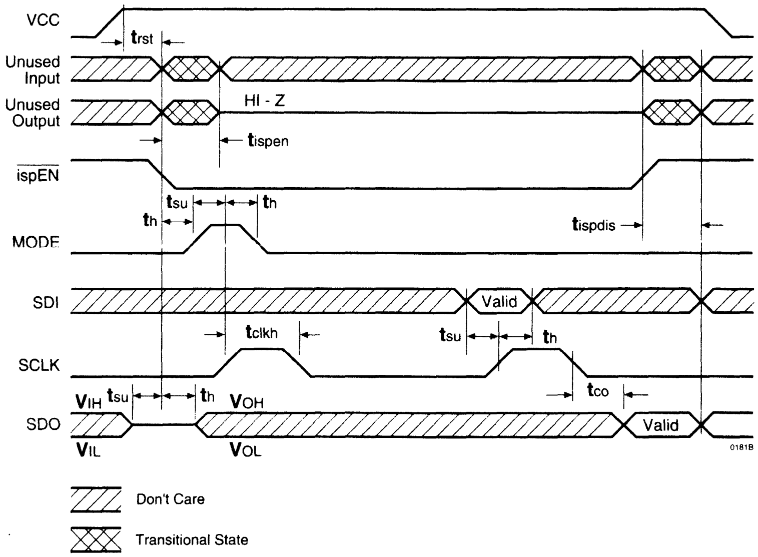


Table 4. ispLSI Programming State Machine Instruction Set

Instruction	Operation	Description
00000	NOP	No operation performed.
00001	ADDSHFT	Address Register Shift: Shifts address into the address shift register from SDIN.
00010	DATASHFT	Data Register Shift: Shifts data into or out of the data serial shift register.
00011	UBE	User Bulk Erase: Erase the entire device.
00100	GRPBE	Global Routing Pool Bulk Erase: Bulk erases the GRP array only.
00101	GLBBE	Generic Logic Block Bulk Erase: Bulk erases the GLB array only.
00110	ARCHBE	Architecture Bulk Erase: Bulk erases the architecture array and I/O configuration only.
00111	PRGMH	Program High Order Bits: The data in the Data shift register is programmed into the addressed row's high order bits.
01000	PRGML	Program Low Order Bits: The data in the Data shift register is programmed into the addressed row's low order bits.
01001	PRGMSC	Program Security Cell: Programs the security cell of the device.
01010	VER/LDH	Verify/Load High Order Bits: Load the data from the selected row's high order bits into the Data shift register for verification.
01011	VER/LDL	Verify/Load Low Order Bits: Load the data from the selected row's low order bits into the Data shift register for verification.
01100	GLBPRLD	Generic Logic Block Preload: Preloads the registers in the GLB with the data from SDIN. All registers in the GLB form a serial shift register. Refer to device layout section for details.
01101	IOPRLD	I/O Preload: Preloads the I/O registers with the data from SDIN. All registers in the I/O cell form a serial shift register (the same order as GLB registers).
01110	FLOWTHRU	Flow Through: Bypasses all the internal shift registers and SDOUT becomes the same as SDIN.
10010	VE/LDH	Verify Erase/Load High Order Bits: Load the data from the selected row's high order bits into the Data shift register for erased verification.
10011	VE/LDL	Verify Erase/Load Low Order Bits: Load the data from the selected row's low order bits into the Data shift register for erased verification.

Device Layout

To translate the JEDEC format programming file into the serial data stream format for programming ispLSI devices, it is necessary to know the physical device layout and programming architecture. Two main factors determine how the translation must be implemented: the length of the address shift register and the length of the data shift register. The length of the address shift register indicates how many rows of data are to be programmed into the device. The length of the Data shift register indicates how many bits are to be programmed in each row. Both registers operate on a First In First Out (FIFO) basis, where the Least Significant Bit (LSB) of the data or address is shifted in first and the Most Significant Bit (MSB) of the data or address is shifted in last. For the Data shift register, the low order bits and the high order bits are separately shifted in.

Each ispLSI device has a predefined number of address rows and data bits needed to access its E^2CMOS cells during programming. The data bits span the columns of the E^2 array. From this information, the number of programming cells (or fuses) are determined. Table 5 highlights the address and data shift register (SR) sizes for currently available ispLSI devices. The JEDEC file for these ispLSI devices will reflect the number of cells (fuses) seen in Table 5. The total number of cells becomes critical if the programming patterns are to be stored in an on-board memory storage of limited capacity such as EPROM or PROM.

The L-fields in the JEDEC programming file indicate the link or fuse numbers of the device. The first cell of the device is indicated by cell number L00000. L-fields of subsequent lines are optional. From this reference cell location, all other cell locations are determined by relative position. A zero (0) in the cell location indicates that the E^2 cell in that particular location is programmed (or has a logic connection intact). A one (1) in the cell location indicates that the cell is erased (equivalent to an open connection). The logic compiler software automatically generates this JEDEC standard programming file after the design has been fit into the device.

Fuse Map to Device Conversion

While the ispCODE software takes care of this detail, it is important to understand how the JEDEC fuse map is mapped onto the physical ispLSI device during programming. The physical layout of the fuse pattern begins with Address Row 0 and ends with the maximum Address Row N and is determined by the length of the Address SR as described in Table 5. Spanning the Address Rows are the outputs of the High-Order Data SR and Low-Order Data SR, as described in Table 6. Programming fuses on a given row are enabled by a "1" within the Address Shift Register for the appropriate row and the use of state machine instructions that selectively operate on the High-Order Data SR or the Low-Order Data SR. For example, the PRGMH instruction programs the High-Order data bits within the device for the selected Address Row and the PRGML instruction programs the Low-Order data bits (Table 4 lists the ISP state machine instructions). Referring to Figure 11, the starting cell (L00000) of the JEDEC fuse map shifts into the device at the physical location corresponding to Address Row 0, High-Order Data SR bit 0. The "n" and "m" in the figure refer to the Address SR length and the Data SR length, respectively, of the device (Table 5). A series of sequential shifts eventually results in the last cell location (Total # of Cells - 1) of the JEDEC fuse map shifting into Address Row (n-1), Low-Order Data SR bit (m-1) on the actual device.

The ispCODE Software routines make use of a bit packed data format, called ispSTREAM™, to transfer data between the JEDEC fuse map and the physical device locations. The binary ispSTREAM format uses one bit to represent the state of each of the programmable cells, instead of the byte value used in an ASCII JEDEC file. Considering the additional characters present in a JEDEC file, this adds up to a space savings of more than a factor of eight. In addition, the ispSTREAM does not require any parsing; the bits are simply read from the file and shifted into the device. As only 804 bytes are required to store the pattern for an ispGAL device, multiple patterns can be stored in a small amount of memory. The JEDEC fuse map can be translated into ispSTREAM format using the isp_jedtoisp function and the ispSTREAM format can be translated into a JEDEC fuse map using the isp_isptojed function.

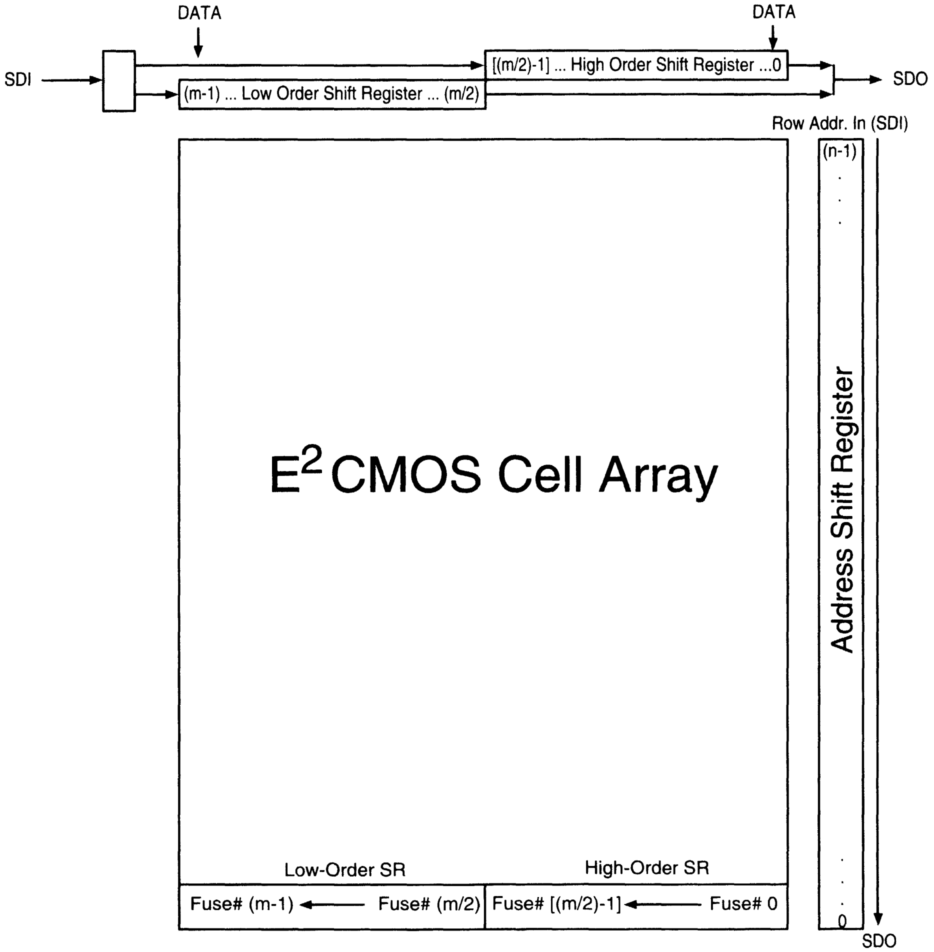
Table 5. ispLSI Address and Data Shift Register and Total Cell Summary

	ispLSI 1016	ispLSI 1024	ispLSI 1032	ispLSI 1048/C	ispLSI 2032	ispLSI 3256
Address SR Length	96	102	108	120/155	102	180
Data SR Length/Address	160	240	320	480/480	80	676
Total Number of Cells	15,360	24,480	34,560	57,600/74,400	8,160	121,680

Table 6. Summary of ispLSI Data Shift Register Bits

Data SR Bits	ispLSI 1016	ispLSI 1024	ispLSI 1032	ispLSI 1048/C	ispLSI 2032	ispLSI 3256
High Order Data SR LSB	0	0	0	0	0	0
High Order Data SR MSB	79	119	159	239	39	337
Low Order Data SR LSB	80	120	160	240	40	338
Low Order Data SR MSB	159	239	319	479	79	675
Data SR Size (Bits)	160	240	320	480	80	676

Figure 11. ispLSI Device to Fuse Map Translation


DATA
DATA
SDI
[(m/2)-1] ... High Order Shift Register ...0
(m-1) ... Low Order Shift Register ... (m/2)
SDO
Row Addr. In (SDI)
(n-1)
E2 CMOS Cell Array
Address Shift Register
Low-Order SR
High-Order SR
Fuse# (m-1) ← Fuse# (m/2)
Fuse# [(m/2)-1] ← Fuse# 0
0
SDO

Algorithms

Command Stream

The first step in programming an ispLSI device is to determine the device type to be programmed. This is ascertained by reading the eight-bit ID of every device. By keeping SDI to a known level (either high or low), the ID shift can be terminated when a sequence of eight ones or eight zeros is read. From the device ID, the serial bit stream for programming can be arranged. A typical programming sequence is listed below:

1) ADDSHFT command shift

2) Execute ADDSHFT command

3) Shift address

4) DATASHFT command shift

5) Execute DATASHFT command

6) Shift high order data

7) PRGMH command shift

8) Execute PRGMH

9) DATASHFT command shift

10) Execute DATASHFT command

11) Shift low order data

12) PRGML command shift

13) Execute PRGML

14) Repeat from 1) until all rows are programmed

Diagnostic Register Preload

This section explains how to preload all of the buried registers and I/O registers to a known state to test the logic function of a device. The process of loading the register reduces the time necessary to test a function that is deeply embedded in the logic of an ispLSI device.

To preload a device, the ISP state machine uses the same five pins as are used for programming (ispEN, SDI, MODE, SDO and SCLK). Two state machine commands preload all of the registers: GLBPRLD and IOPRLD. These commands enable two shift registers and allow data to be loaded into the device. The steps for loading data into the device are listed below:

1. Enter the ISP programming mode by driving ispEN to Vil.

2. Load command GLBPRLD and execute command (wait one tclk).

3. Clock in the GLB preload data.

4. Load the command IOPRLD and execute the command (wait one tclk).

5. Clock in the I/O preload data.

6. Return to the Normal Mode by driving ispEN pin to Vih.

7. Execute the vectors.

When preloading a device it is important to keep the dedicated input pins (RESET, Y0, Y1, Y2 and Y3) in the same state as in the previous vector. If the state of these pins is switched during the preload sequence, the register may not load correctly and the results are not guaranteed.

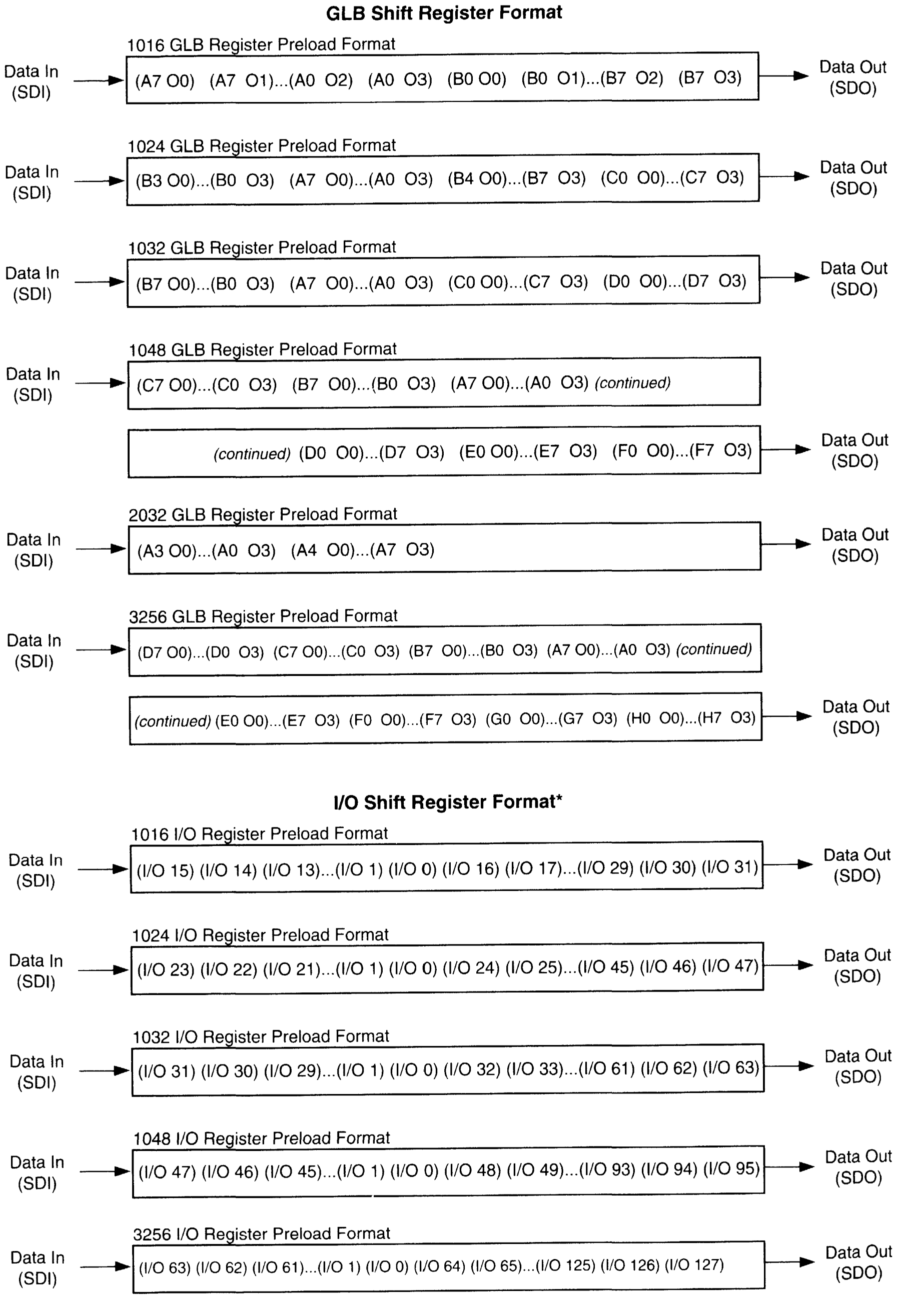
The preload feature is not recommended for designs using product term resets. The asynchronous nature of the resets can cause registers to be reset unexpectedly; therefore the results are not guaranteed.

There are two shift registers used to preload an ispLSI device: the GLB shift register and the I/O shift register (Table 7). The data format for both devices is shown in Figure 12. The GLB registers are listed with their outputs (i.e. (A7 O0) indicating output 0, of GLB A7).

Table 7. Preload Shift Registers

Device	GLB Shift Reg. Length	I/O Shift Reg. Length
ispLSI 1016	64 bits	32 bits
ispLSI 1024	96 bits	48 bits
ispLSI 1032	128 bits	64 bits
ispLSI 1048/C	192 bits	96 bits
ispLSI 2032	32 bits	N/A
ispLSI 3256	256 bits	128 bits

Figure 12. GLB Shift Register and I/O Shift Register Format


GLB Shift Register Format
1016 GLB Register Preload Format
Data In (SDI)
(A7 O0) (A7 O1)...(A0 O2) (A0 O3) (B0 O0) (B0 O1)...(B7 O2) (B7 O3)
Data Out (SDO)
1024 GLB Register Preload Format
Data In (SDI)
(B3 O0)...(B0 O3) (A7 O0)...(A0 O3) (B4 O0)...(B7 O3) (C0 O0)...(C7 O3)
Data Out (SDO)
1032 GLB Register Preload Format
Data In (SDI)
(B7 O0)...(B0 O3) (A7 O0)...(A0 O3) (C0 O0)...(C7 O3) (D0 O0)...(D7 O3)
Data Out (SDO)
1048 GLB Register Preload Format
Data In (SDI)
(C7 O0)...(C0 O3) (B7 O0)...(B0 O3) (A7 O0)...(A0 O3) (continued)
(continued) (D0 O0)...(D7 O3) (E0 O0)...(E7 O3) (F0 O0)...(F7 O3)
Data Out (SDO)
2032 GLB Register Preload Format
Data In (SDI)
(A3 O0)...(A0 O3) (A4 O0)...(A7 O3)
Data Out (SDO)
3256 GLB Register Preload Format
Data In (SDI)
(D7 O0)...(D0 O3) (C7 O0)...(C0 O3) (B7 O0)...(B0 O3) (A7 O0)...(A0 O3) (continued)
(continued) (E0 O0)...(E7 O3) (F0 O0)...(F7 O3) (G0 O0)...(G7 O3) (H0 O0)...(H7 O3)
Data Out (SDO)
I/O Shift Register Format*
1016 I/O Register Preload Format
Data In (SDI)
(I/O 15) (I/O 14) (I/O 13)...(I/O 1) (I/O 0) (I/O 16) (I/O 17)...(I/O 29) (I/O 30) (I/O 31)
Data Out (SDO)
1024 I/O Register Preload Format
Data In (SDI)
(I/O 23) (I/O 22) (I/O 21)...(I/O 1) (I/O 0) (I/O 24) (I/O 25)...(I/O 45) (I/O 46) (I/O 47)
Data Out (SDO)
1032 I/O Register Preload Format
Data In (SDI)
(I/O 31) (I/O 30) (I/O 29)...(I/O 1) (I/O 0) (I/O 32) (I/O 33)...(I/O 61) (I/O 62) (I/O 63)
Data Out (SDO)
1048 I/O Register Preload Format
Data In (SDI)
(I/O 47) (I/O 46) (I/O 45)...(I/O 1) (I/O 0) (I/O 48) (I/O 49)...(I/O 93) (I/O 94) (I/O 95)
Data Out (SDO)
3256 I/O Register Preload Format
Data In (SDI)
(I/O 63) (I/O 62) (I/O 61)...(I/O 1) (I/O 0) (I/O 64) (I/O 65)...(I/O 125) (I/O 126) (I/O 127)
Data Out (SDO)

* ispLSI and pLSI 2000 family members do not have I/O registers.

Boundary Scan

The Lattice ispLSI 3000 family of devices supports the IEEE 1149.1 Boundary Scan specifications. The following sections explain in detail how to interface to the devices through the Test Access Port (TAP), how the boundary scan registers are implemented within the devices, and the boundary scan instructions that are supported by the ispLSI and pLSI 3000 family

Test Access Port (TAP)

The test access port of the boundary scan is accessed through six interface signals: TDI, TDO, SCLK, BSCAN, TMS, TRST. These interface signals have two functions in the case of the ispLSI 3000 family; they serve as both the Boundary Scan interface and in-system programming interface signals. For the pLSI 3000 family, the six interface signals are only used for the boundary scan TAP interface. Table 8 describes the interface signals.

The above mentioned six signals are dedicated for Boundary Scan use for the pLSI family of devices. As ISP programming is accomplished through the same pins, five of the six signals have both Boundary Scan interface and ISP functions on the ispLSI devices. TRST is the only signal that does not have a dual function. It is used only to reset the TAP controller state machine. The sequencing of test routines are governed by the TAP controller state machine. The state machine uses the TMS and TCK signals as its inputs to sequence the states. Figure 13A is the IEEE1149.1 specified state machine. The condition for the state transition is the state of the TMS input condition before TCK within a given state. The timing diagram is also shown in Figure 13B.

The main features of the TAP controller state machine include Test-Logic-Reset state to reset the controller and the Run-Test states. Two main components of the Run-Test states are Data Register (DR) control states and Instruction Register (IR) control states. Both of these register control states are organized in a similar manner. The user can capture the registers, shift the register string, or update the registers. Capturing the DRs simply loads the DR with the data from the corresponding functional input, output, or I/O pins. The IR capture, on the other hand, loads the IRs with the previously executed instruction bits. Shift register states serially shift

Table 8. Boundary Scan Interface Signals

pLSI 3000 Family	ispLSI 3000 Family	Pin Function Description
BSCAN	BSCAN/ispEN	Active high signal on this pin selects the Boundary Scan function while active low signal selects the ISP function on the ispLSI devices. Internal pullup on this pin drives the signal high when the external pin is not driven.
TCK	TCK/SCLK	Test Clock function for Boundary Scan and serial clock for the the ISP function.
TMS	TMS/MODE	Test Mode Select for Boundary Scan and MODE control for the ISP function.
TDI	TDI/SDI	Test Data Input for Boundary Scan and Serial Data Input for the ISP function. Functions as a serial data input pin for both interfaces.
TRST	TRST	Test Reset Input is an asynchronous signal to initialize the TAP controller to the Test-Logic-Reset state.
TDO	TDO/SDO	Test Data Output for Boundary Scan and Serial Data Output for the ISP function. Functions as a serial data output pin for both interfaces.

Figure 13A. TAP Controller State Machine

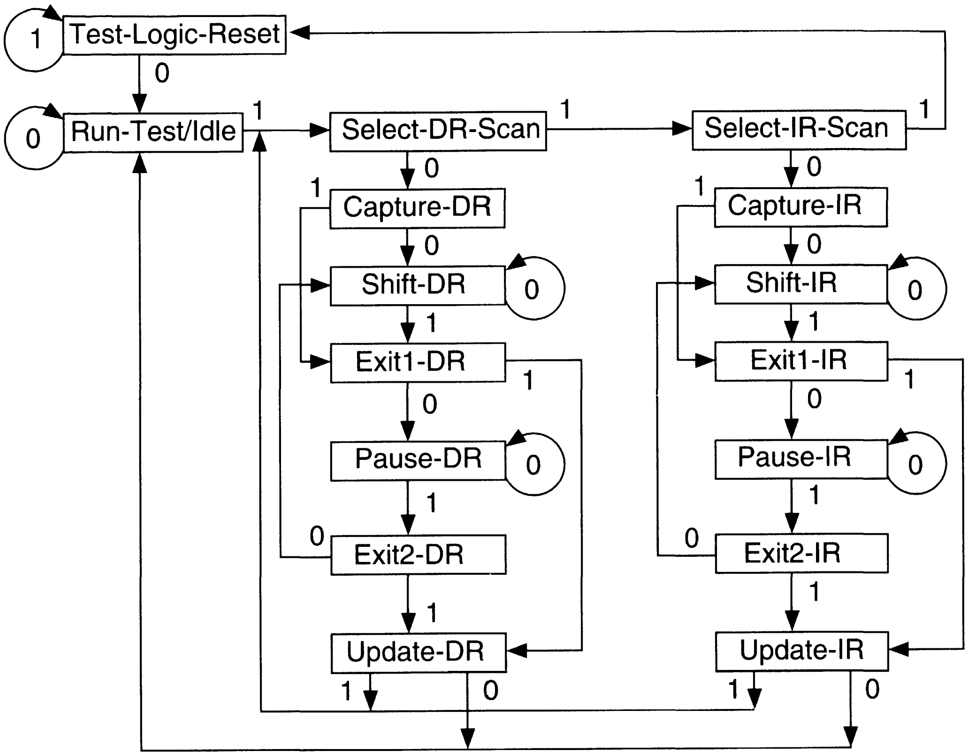
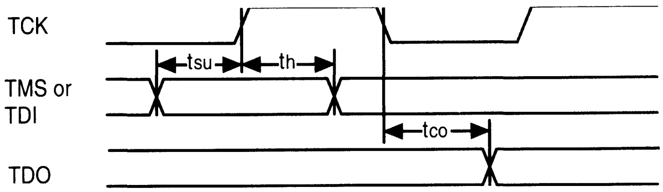


Figure 13B. TAP Controller Timing Diagram



the DR and IR. In the case of DR shift, the data is shifted according to the order of the inputs, outputs, and I/Os defined in the Boundary Scan section of each device data sheet. The IRs are shifted out from the least significant bit first. During update register states, the DRs update the latches to drive the external pins and the IRs update the instruction bits with the instruction that is to be executed.

Boundary Scan Registers

In order to support Boundary Scan, two types of data registers are defined for the ispLSI and pLSI devices — I/O cell registers and input cell registers. The main purpose of these registers is to capture test data from the appropriate signals and shift data to either drive the test pins or examine captured test data.

Figure 14 describes the register for the I/O cell. The I/O cell, by definition, must have three components: one register component captures the output enable (OE) signal, the second component captures the output data, and the third captures the input data. These components make up the three registers that are part of the shift register string for each of the I/O pins. Only parts of the I/O cell registers will have valid data when I/O pins are configured as input-only or output-only, thus the test routines must be able to monitor the appropriate register bits. The update registers are used mainly to store data that is to be driven onto the I/O pins. The multiplexer controls are driven by the signal from the TAP controller at appropriate states.

The function of an input cell register is much simpler than that of an I/O cell. Figure 15 illustrates the single input register cell. The purpose of the Input cell is to capture the input test data and shift the data out of the shift register string.

Boundary Scan Instructions

Lattice ispLSI and pLSI devices support the three mandatory instructions defined by the Boundary Scan definition. The following paragraphs describe each of the instructions and its instruction code. A two-bit shift register is defined within the devices to implement the Instruction shift register.

The SAMPLE/PRELOAD (Instruction Code - 01) instruction is used to sample the pins that are to be tested. During the Capture-DR state, while executing this instruction, the DRs are loaded with the state of the pins which can then be examined after shifting the data through TDO. The PRELOAD part of this instruction is simply loading the DRs during Shift-DR state with the desired condition for each of the pins.

The EXTEST (Instruction Code - 00) instruction drives the external pins with the previously updated values from the DR during the Update-DR state.

The BYPASS (Instruction Code - 11) instruction is used to bypass any device that is not accessed during any part of the test. The definition of the BYPASS instruction allows TDI not to be driven during the Shift-IR state. In order to shift in the correct instruction code, the TDI pin has an internal pull-up to drive logic high. A bypassed boundary scan device has a single bypass register as shown in Figure 16.

Figure 14. Boundary Scan I/O Cell

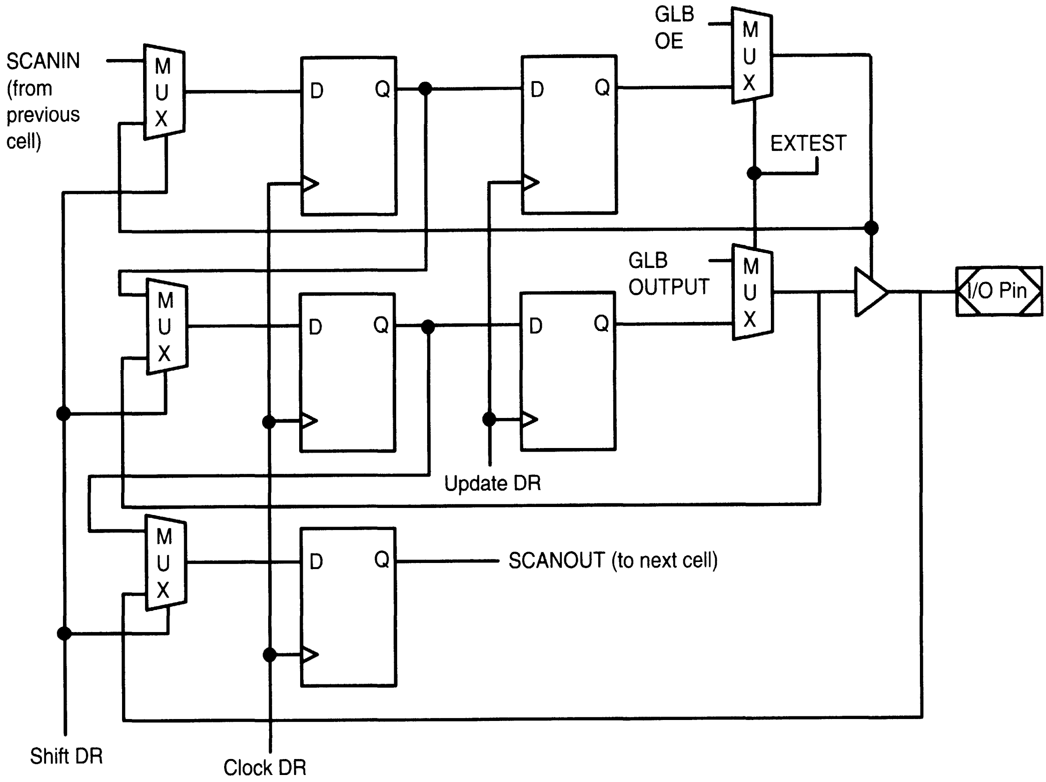


Figure 15. Boundary Scan Input Cell

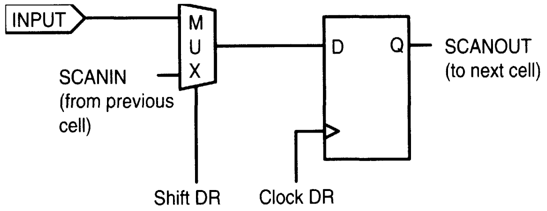
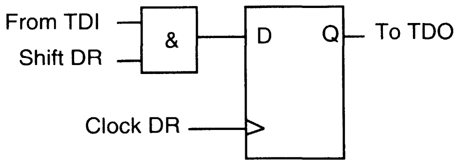


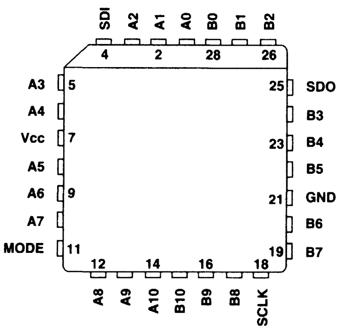
Figure 16. Bypass Register



ispGDS Programming Details

The following sections describe the state machine instruction set, timing parameters, device layout, and programming algorithms as they apply to ispGDS devices in general. Figure 17 shows the ispGDS 28-pin device pinout.

Figure 17. ispGDS 28-Pin PLCC Pinout Diagram



Shift Registers

The ispGDS devices have three shift registers, the Device ID shift register, the Instruction shift register and the Data shift register. All shift registers operate on a First In First Out (FIFO) basis, and are chosen by which state the programming state machine is in.

The Device ID shift register is only accessible in the IDLE state. It is eight bits long, and is only used to shift out the device ID. The ispGDS device IDs are 70-72 (hex) (Table 10). The Instruction shift register is only accessible in the SHIFT state. It is five bits long, and is only used to shift the Instruction Codes into the device. The Device ID and Instruction shift registers expect the LSB to be shifted in first. The Data shift register is 24 bits long, and is used to shift all addresses and data into or out of the device. The Data shift register is only accessible in the EXECUTE state when executing a SHIFT_DATA instruction (Table 9).

To program an ispGDS device, data is read from a serial bit stream and shifted into the shift registers. Twenty -four bits are read at a time, shifted into the device, and then a programming operation is performed. The exact sequence, and the methods for converting a JEDEC map into a serial bit stream are explained in the "ispGDS Internal Architecture" section.

Timing

Programming the ispGDS devices properly requires that a number of timing specifications be met. The specifications relating to programming and erasing the E^2CMOS cells are the most critical. In addition to a minimum pulse width, there is also a maximum timing specification. Refer to the ispGDS programming mode timing specifications in the Data Book for the timing requirements. Timing diagrams for the programming mode specifications are shown in Figures 18, 19, and 20.

Table 9. ispGDS Programming State Machine Instruction Set

Instruction	Operation	Description
00000	NOP	No operation performed.
00010	SHIFT_DATA	Clocks data into, or out of, the Data Shift Register.
00011	BULK_ERASE	Erases the entire device.
00101	ERASE_ARRAY	Erases everything except the Architecture rows.
00110	ERASE_ARCH	Erases the Architecture rows only.
00111	PROGRAM	Programs the Shift Register data into the addressed row.
01010	VERIFY	Load data from the selected row into the Serial Shift Register.
01110	FLOWTHRU	Disables the Shift Register (SDI=SDO).

Figure 18. ispGDS Programming Mode Timing

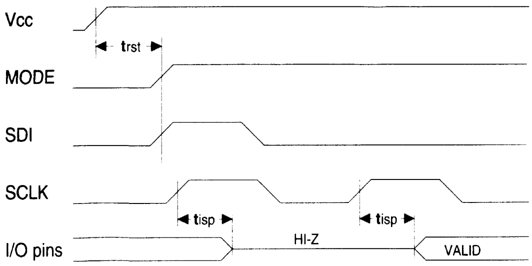


Figure 19. ispGDS Shift Register Timing

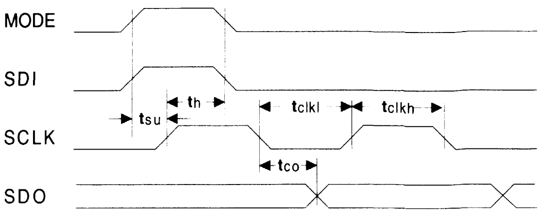
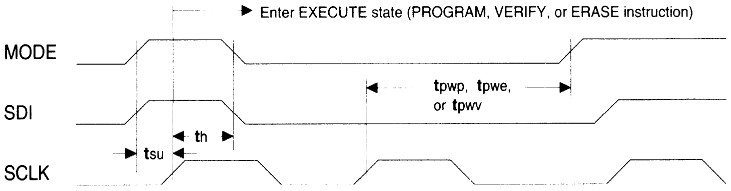


Figure 20. ispGDS Program, Verify, and Erase Timing

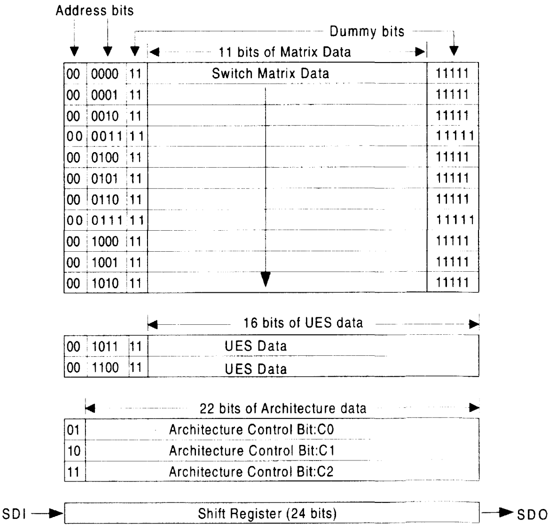


ispGDS Internal Architecture

This section covers the details of constructing the ispSTREAM format. Only 49 bytes are required to store the pattern for an ispGDS device. If you are using the supplied software tools, a conversion utility (complete with source code) is included to convert an industry-standard JEDEC file to ispSTREAM format. All of the Lattice software routines read and write this ispSTREAM.

The ispGDS devices are composed of two basic architectural components (Figure 21). The first component consists of three rows of architectural information, which contain the three bits that control the function of each I/O cell. The rows are 24 bits long, providing one bit for each I/O cell (the ispGDS18 and ispGDS14 do not use all of the bits). The second component contains the cell data for the switch matrix area of the device and the User Electronic Signature (UES) data area. There are two UES rows of 24 bits each, and 11 switch matrix rows of 24 bits each.

Figure 21. ispGDS Architecture



00	0000	11	Switch Matrix Data	11111
00	0001	11		11111
00	0010	11		11111
00	0011	11		11111
00	0100	11		11111
00	0101	11		11111
00	0110	11		11111
00	0111	11		11111
00	1000	11		11111
00	1001	11		11111
00	1010	11		11111

00	1011	11	UES Data
00	1100	11	UES Data

01	Architecture Control Bit:C0
10	Architecture Control Bit:C1
11	Architecture Control Bit:C2

Shift Register (24 bits)

Although the shift register lengths are 24 bits long, it is not composed entirely of data area. In the architectural section, two bits are used for addressing. In the matrix/ UES area, six bits are used for addressing. In the switch matrix area, there are only 11 bits of actual data, and seven dummy bits which exist only to make the shift registers the same length. These seven bits are read as a one, or a logic High on SDO. For the UES, there are 16 bits of actual data in each row and two dummy bits.

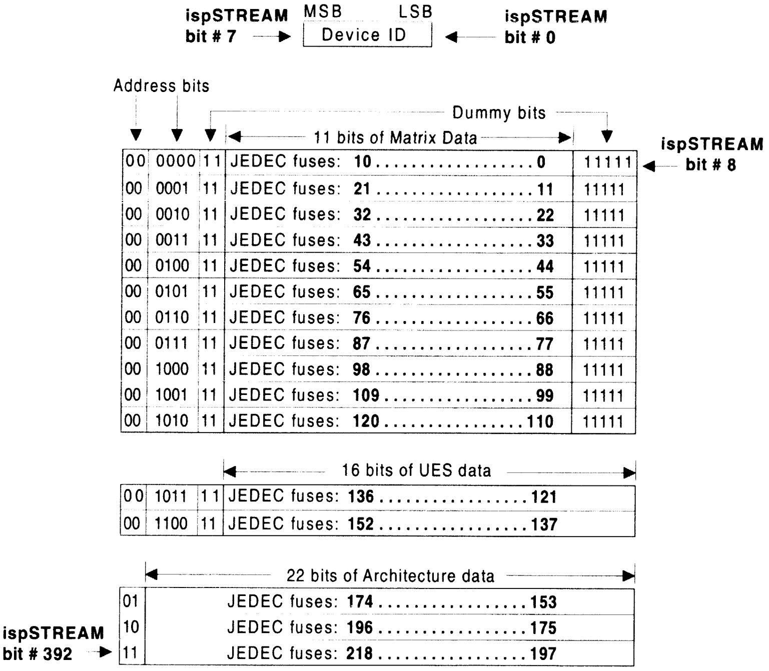
ispGDS ispSTREAM Format

To convert the information in a standard JEDEC file into the ispSTREAM format, add all of the addressing information and the placeholding bits (dummy bits). The objective is to include every bit needed for programming. For the three architecture rows, simply add the two address bits.

For the UES and Switch Matrix rows, there are eight bits to add. The first two bits are always 00, which distinguishes this area from the Architectural row. In addition, there are four bits needed to address the specific row, and two bits needed as placeholders. In the Switch Matrix rows, there are also 5 bits needed for placeholding at the end of the rows. The various placeholding bits are built into the device so that all rows appear to be the same length, thus simplifying programming operations.

The ispSTREAM uses one bit for each programmable cell. This means that each row includes 24 bits, or three *bytes* of storage. With three bytes of storage per row, and 16 rows per device, the ispSTREAM uses only 48 bytes of storage area. However, there is one extra byte used at the front of the file to store the device ID code, and a 32-bit checksum. The ID code is identical to the one that is hardwired into the device. This ID code ensures that the ispSTREAM type is the same as the device to be programmed. For example, if an ispSTREAM is stored in EPROM, it is stacked end to end. The ID code determines not only which device type the ispSTREAM belongs with, but its length, and thus, where the next pattern starts. All ispSTREAM formats, regardless of which Lattice In-System Programmable device they are intended for, contain this ID code in the first byte. See Figure 22 for details of the ispSTREAM format, and Figure 23 for the JEDEC map.

Figure 22. ispGDS ispSTREAM Format

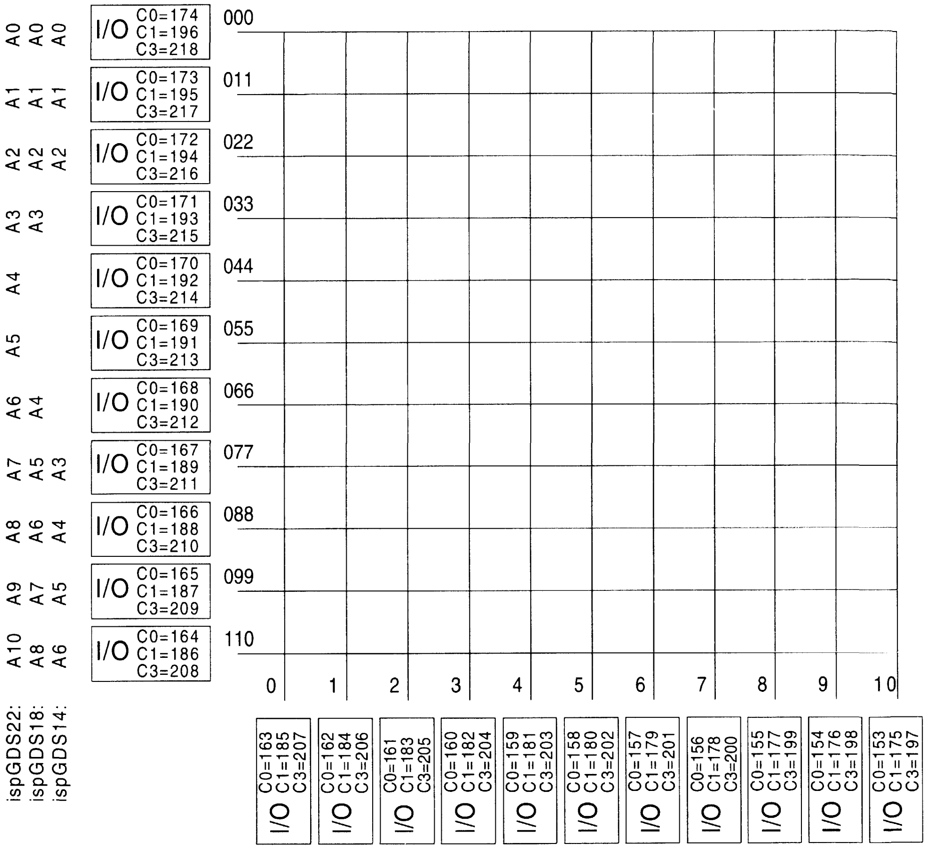


00	0000	11	JEDEC fuses: **10** **0**	11111
00	0001	11	JEDEC fuses: **21** **11**	11111
00	0010	11	JEDEC fuses: **32** **22**	11111
00	0011	11	JEDEC fuses: **43** **33**	11111
00	0100	11	JEDEC fuses: **54** **44**	11111
00	0101	11	JEDEC fuses: **65** **55**	11111
00	0110	11	JEDEC fuses: **76** **66**	11111
00	0111	11	JEDEC fuses: **87** **77**	11111
00	1000	11	JEDEC fuses: **98** **88**	11111
00	1001	11	JEDEC fuses: **109** **99**	11111
00	1010	11	JEDEC fuses: **120** **110**	11111

00	1011	11	JEDEC fuses: **136** **121**
00	1100	11	JEDEC fuses: **152** **137**

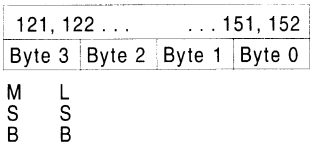
01	JEDEC fuses: **174** **153**
10	JEDEC fuses: **196** **175**
11	JEDEC fuses: **218** **197**

Figure 23. ispGDS JEDEC Fuse Map



ispGDS22:	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
ispGDS18:	B8	B7	B6	B5			B4	B3	B2	B1	B0
ispGDS14:	B6	B5	B4					B3	B2	B1	B0

User Electronic Signature



Algorithms

ispGDS device programming is described as a hierarchical set of algorithms and functions. This section contains high-level algorithms for erasing, programming, verifying, and loading ispGDS devices. A universal set of functions is used to make up the algorithms and enable them to be written in a modular format. The individual functions are explained in the next section. Note that most procedures leave the device in the SHIFT state. These algorithms and functions closely follow the ispCODE source code library that Lattice provides.

To simplify the algorithms, all operations use an ispSTREAM format as the data structure from which to read and write. The ispSTREAM contains the address information and simplifies the operations considerably. Working from the ispSTREAM, the device appears as an array of 16 rows, each 24 bits long.

Program Algorithm

Before programming a device, it must be erased. Cells can be programmed (set to a JEDEC zero) using the programming command, but only an Erase procedure erases a cell (set a cell back to a JEDEC "1" (one)). In the algorithm in Listing 1, the entire device is erased (Bulk Erased), and then the entire device is programmed.

Listing 1. ispGDS Programming Algorithm

To program a device:

```
Call procedure: Get_ID ( to check device type)
Call procedure: Change_State ( from IDLE to SHIFT state)
(Erase entire device)
      Call procedure: Shift_Command, with command: ERASE
      Call procedure: Change_State (to EXECUTE State)
      Call procedure: Execute_Command (starts operation)
      Call procedure: Wait (Erase_Time)
      Call procedure: Change_State ( to SHIFT state)
Set row_count =0
Loop until row_count = 15
(Program one row on each loop)
      Call procedure: Shift_Command, with command: SHIFT_DATA
      Call procedure: Change_State (to EXECUTE State)
      Call procedure: Shift_Data_In, with data location in ispSTREAM at (row_count
      x 24)
      Call procedure: Change_State ( to SHIFT state)
      Call procedure: Shift_Command, with command: PROGRAM
      Call procedure: Change_State (to EXECUTE State)
      Call procedure: Execute_Command (starts operation)
      Call procedure: Wait (Program_Time)
      Call procedure: Change_State ( to SHIFT state)
End Loop
```

Load Algorithm

The load algorithm in Listing 2 is the same for all ispGDS devices. First, the 13 rows of array data (11 rows for the array matrix, and two for the UES) are read, and then the three rows of architectural information are read. After each row is read, it is stored in an ispSTREAM format.

In order to load each row's data into the shift register, it is necessary to load the address of the row into the appropriate area of the shift register. Because of the unique way the different areas of the device are addressed, the simplest way to get the addresses into the device in the proper order is to use an existing ispSTREAM to supply those addresses. In other words, the full data for each row is loaded from the ispSTREAM into the device. When a VERIFY command is executed, the device's data for the same row is then loaded into the shift register to be shifted out. This method will be used in this algorithm.

When using an existing ispSTREAM to supply the addresses, the data should not be the same as the expected data, or a failure to verify may not be detected. To avoid this possibility, a pattern that contains all "1s" (ones) for data can be used (and is supplied with the software tools provided by Lattice). This ispSTREAM still has the addresses intact, but all programmable cell data is set to a "1" (one) (erased state).

Listing 2. Load Algorithm

To load a device:

```
Call procedure: Get_ID ( to check device)
Call procedure: Change_State ( from IDLE to SHIFT state)
Set row_count =0
Loop until row_count = 15
      Call procedure: Shift_Command, with command: SHIFT_DATA
      Call procedure: Change_State (to EXECUTE State)
      Call procedure: Shift_Data_In, with data location in Source ispSTREAM at
      (row_count x 24)
      Call procedure: Change_State ( to SHIFT state)
      Call procedure: Shift_Command, with command: PROGRAM
      Call procedure: Change_State (to EXECUTE State)
      Call procedure: Execute_Command (starts operation)
      Call procedure: Change_State ( to SHIFT state)
      Call procedure: Shift_Command, with command: SHIFT_DATA
      Call procedure: Change_State (to EXECUTE State)
      Call procedure: Shift_Data_Out, with data location in Target ispSTREAM at
      (row_count x 24)
      Call procedure: Change_State ( to SHIFT state)
End Loop
```

Verify Algorithm

A row by row verification procedure is used to verify the ispGDS device. This procedure is basically the same as the Load algorithm, except that each row is compared with (instead of stored in) an ispSTREAM as the data is shifted out of the device. Note that the special pattern used for verifying is used to load the addresses, as in the Load algorithm.

Listing 3. Verify Algorithm

To verify a device:

```
Call procedure: Get_ID (to check device type)
Call procedure: Change_State ( from IDLE to SHIFT state)
Set row_count =0
Loop until row_count = 15
      Call procedure: Shift_Command, with command: SHIFT_DATA
      Call procedure: Change_State (to EXECUTE State)
      Call procedure: Shift_Data_In, with data location in Source ispSTREAM at
      (row_count x 24)
      Call procedure: Change_State ( to SHIFT state)
      Call procedure: Shift_Command, with command: VERIFY
      Call procedure: Change_State (to EXECUTE State)
      Call procedure: Execute_Command (starts operation)
      Call procedure: Wait (Verify_Time)
      Call procedure: Change_State ( to SHIFT state)
      Call procedure: Shift_Command, with command: SHIFT_DATA
      Call procedure: Change_State (to EXECUTE State)
      Call procedure: Shift_Data_Out, with data location a 24 bit temporary buffer
      Compare temp row buffer with data location in ispSTREAM to be verified
      against, at (row_count x 24) Verify Error if the 24 bits don't match
      Call procedure: Change_State ( to SHIFT state)
End Loop
```

ispGDS Procedures

This section describes the procedures that make up the program, verify, and load algorithms for the ispGDS family of devices. The procedures are written so that each algorithm may be written in a high-level modular format, calling one of the following procedures to actually change pin levels and handle timing.

Important: Notice that most of the procedures are written so that the state machine is left in the Shift State, ready to perform the next operation. This point is important in keeping all the routines compatible.

Goto_IDLE Procedure

The Goto_IDLE procedure resets the programming state machine to the IDLE state, regardless of which state it is in.

Procedure Steps:

```
set MODE pin High, and SDI pin Low
wait Tsu
bring SCLK pin High
wait Tclkh
bring SCLK pin Low
(END Procedure)
```

Get_ID Procedure

The 8-bit device ID codes identify the three different ispGDS devices (Table 10). The ID is read in the IDLE state by first loading the ID into the shift register and then clocking the data out. The ID is loaded by holding MODE high and SDI low and clocking the device. The ID is clocked out of the device by holding MODE low and clocking SCLK. Only seven clock cycles are required, since the first bit is available at SDO after the ID is loaded.

Table 10. ispGDS Device Codes

Device	Pins	Device ID
ispGDS22	28	0111 0010 (72 hex)
ispGDS18	24	0111 0001 (71 hex)
ispGDS14	20	0111 0000 (70 hex)

Procedure Steps:

```
set MODE pin High, and SDI pin Low
wait Tsu
Set SCLK pin High
wait Tclkh
Set SCLK pin Low
set count =0
get value from SDO and store in temp_buffer[0]
set count = 1
loop until count == 7
        bring SCLK pin High
        wait Twh
        bring SCLK pin Low
        wait Twl
        get value from SDO and store in
        temp_buffer[count]
End loop
( Device ID code is now stored in the temp_buffer array)
(END procedure)
```

Change_State Procedure

The Change_State procedure changes the programming state machine to the next state, according to the state diagram.

Procedure Steps:

```
set MODE pin High, and SDI pin High
wait Tsu
bring SCLK pin High
wait  Th
set MODE pin Low, and SDI pin Low
wait Tclkh
bring SCLK pin Low
(END Procedure)
```

Shift_ Command Procedure

The Shift_Command procedure shifts a five-bit command into the device's shift register. The various commands should be coded so the procedure can use a mnemonic (such as PROGRAM), and the controlling software can use the appropriate five-bit sequence for that command.

Procedure Steps:

```
set MODE pin Low
set count =0
loop until count == 4
        get next bit of command code (count = bit number)
        set SDI pin to bit value
        wait Tsu
        bring SCLK pin High
        wait Tclkh
        bring SCLK pin Low
        count = count +1
End loop
(END Procedure)
```

Shift_ Data_In Procedure

The Shift_Data_In procedure explains the steps to clock a row of data into the device, reading the data from an ispSTREAM. This procedure shifts in 22 bits of data, and is used for all 16 rows.

Procedure Steps:

```
set MODE pin Low
set count =0
loop until count == 23
        get next bit from ispSTREAM ( bit number = count x
        row_number)
        set SDI pin to bit value
        wait Tsu
        bring SCLK pin High
        wait Tclkh
        bring SCLK pin Low
End loop
(END Procedure)
```

Shift_ Data_Out Procedure

The Shift_Data_In procedure explains the steps to clock a row of data out of the device and store it in an ispSTREAM. This procedure shifts out 22 bits of data, and is used for all 16 rows.

Procedure Steps:

```
set MODE pin Low
wait Tsu
set count =0
loop until count == 23
        bring SCLK pin High
        wait Tclkh
        bring SCLK pin Low
        get value of SDO pin and store as next bit in
        ispSTREAM ( bit number = count x row_number)
End loop
(END Procedure)
```

Execute_Command Procedure

The Execute_Command procedure causes many of the commands to begin executing after the state machine is in the EXECUTE state.

Procedure Steps:

```
set MODE pin Low, and SDI pin Low
wait Tsu
bring SCLK pin High
wait Twh
bring SCLK pin Low
(END Procedure)
```

Wait Procedure

The Wait procedure waits the indicated time to ensure that various timing parameters are met. This procedure is likely to be used when executing the PROGRAM and ERASE procedures, which need a long delay (tens of milliseconds). The other timing parameters may be guaranteed by the system timing. Various timing parameters should be coded so that a mnemonic may be passed to the procedure.

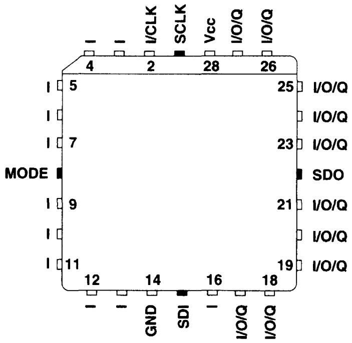
Procedure Steps:

```
wait the indicated time
(END Procedure)
```

ispGAL Programming Details

The following sections describe the state machine instruction set, timing parameters, and device layout as they apply to ispGAL devices in general. Figure 24 shows the ispGAL22V10 28-pin device pinout.

Figure 24. ispGAL22V10 28-Pin PLCC Pinout Diagram



Shift Registers

The ispGAL device has four shift registers: Device ID, Instruction, Data, and Architecture. All shift registers operate on a First In-First Out (FIFO) basis, and are enabled by the programming state machine.

The Device ID shift register is only accessible in the IDLE state. It is eight bits long, and is only used to shift out the device ID. For the ispGAL22V10, the ID is defined to be 08 (hex). The Instruction shift register is only accessible in the SHIFT state. It is five bits long, and is only used to shift the Instruction Codes into the device. The Data and Instruction shift registers expect the LSB to be shifted in first. The Data shift register is 138 bits long, and is used to shift all addresses and data into or out of the device. The Data shift register is only accessible in the EXECUTE state when executing a SHIFT_DATA instruction. The Architecture shift register is 20 bits long and the Output Logic Macro Cell (OLMC) 1's S1 architecture bit is shifted in first and OLMC 10's S0 architecture bit is shifted in last. The Architecture shift register is accessed during the EXECUTE state, when the ARCH_SHIFT instruction is executed.

To program an ispGAL device, data is read from a serial bit stream and shifted into the shift registers. The data is read 138 bits at a time, shifted into the device, and then programmed into the device through a programming operation. Table 11 describes the instructions for the ispGAL state machine. The exact sequence and methods for converting a JEDEC map into a serial bit stream are explained in the Internal Architecture section.

Timing

Programming the ispGAL devices properly requires that a number of timing specifications be met. Most critical are the specifications relating to programming and erasing the E^2CMOS cells. In addition to a minimum pulse width, there is also a maximum specification for these parameters. Refer to the ispGAL programming mode timing specifications for the timing requirements, which are identical to the ispGDS specifications. Diagrams for the programming mode specifications are shown in Figures 18, 19, and 20 of the ispGDS timing section in this manual.

Table 11. ispGAL Programming State Machine Instruction Set

Instruction	Operation	Description
00000	NOP	No operation performed.
00010	SHIFT_DATA	Clocks data into, or out of, the Data Shift Register.
00011	BULK_ERASE	Erases the entire device.
00101	ERASE_ARRAY	Erases everything except the Architecture rows.
00110	ERASE_ARCH	Erases the Architecture rows only
00111	PROGRAM	Programs the Serial Shift Register data into the addressed row
01010	VERIFY	Load data from the selected row into the Serial Shift Register.
01101	IOPRLD	Preload the I/O register with given data.
01110	FLOWTHRU	Disables the Shift Register (SDI=SDO).
10100	ARCH SHIFT	Enables the Architecture shift register for shifting data into or out of the register.

Securing an ispGAL Device

The ispGAL devices are not secured by an instruction. To secure ispGAL devices, row 61 must be programmed in the same manner that other data rows are programmed. When programming this security row, the data bits are "don't care."

Internal Architecture

This section describes the internal architecture of the device as it relates to programming.

This section covers details of constructing the ispSTREAM format. If you are using the supplied software tools, a conversion utility (complete with source code) is included to convert an industry-standard JEDEC file to ispSTREAM format. All of the Lattice software routines read and write the ispSTREAM format.

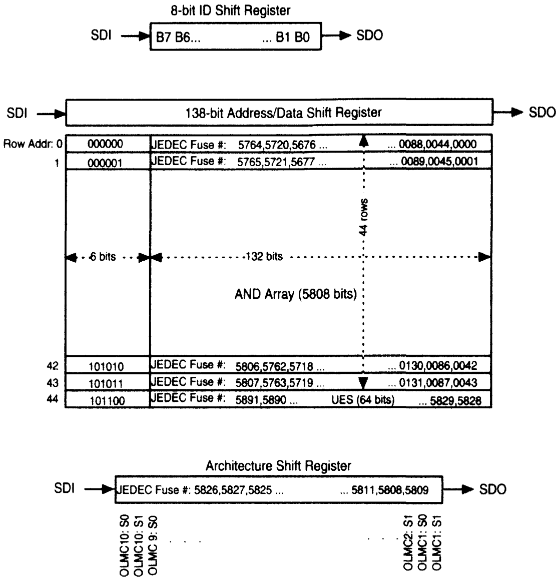
Three components comprise the ispGAL device programming architecture (Figure 25): 44, 132-bit rows of AND array, one 64-bit row of User Electronic Signature (UES), and one 20-bit row of architecture information.

The AND array section of the physical layout is organized so that each column of JEDEC fuse numbers shown in the logic diagram of the ispGAL22V10 corresponds to one row of shift register for the device layout. Each physical row is 132 bits long. With each row of AND array data, there is a 6-bit row address associated with it, which including the row address bits, makes the shift registers 138 bits long. The row address bits must be shifted into the shift register along with the AND array data. Executing a PROGRAM command following the combination of data and row address shift programs the row that is specified by the shift instruction.

The I/O preload (IOPRLD) is performed in the same order as the Architecture shift register shown in Figure 25. Once in I/O Preload, the length of the shift register is determined by the number of I/Os that are configured as registered output (see the "Architecture" section of the ispGAL22V10 data sheet). The length of the shift register and the order must be determined before IOPRLD can be executed.

The UES row is unique in that it is only 64 bits long. When the row address bits are added to the row itself, the total shift register length required to fully specify the UES row is 70 bits long. In other words, only 70 bits out of the 132-bit shift register are used for the UES. The 20-bit Architecture shift register is selected when the ARCH_SHIFT instruction is executed. The OLMC 0, S1: OLMC 0, S0; OLMC 1, S1: OLMC 1, S0: etc. are shifted in order with the last bit of the shift register being OLMC 10, S0.

Figure 25. ispGAL Device Shift Register Layout



Algorithms and Procedures

The ispGAL's programming algorithm and programming procedure are very similar to the ispGDS. For the sake of brevity, please refer to the algorithm and procedures section in the ispGDS section if you are interested in this information. If you have further questions, please call the Lattice Hotline at 1-800-FASTGAL.

ISP Daisy Chain Details

This section provides a detailed look at the issues associated with daisy chain programming. Before examining the details, the reader should understand the differences between ISP devices. This section describes those differences and the unique programming features of each ISP device.

ISP Overview for Daisy Chain

Similarities and Differences Between Devices

For the purpose of cascading, ISP devices can be categorized into two device groups: ispLSI and ispGDS/ ispGAL. Table 12 highlights the similarities between these device groups.

Using the same state machine controls makes it possible to program multiple ISP devices by operating all the cascaded devices' state machines in parallel. This synchronizes all the devices during programming within the daisy chain to a known state. However, having all ISP devices in the same state does not mean that all devices are executing the same instruction. The ability of each device in the daisy chain to execute a different instruction makes it possible to selectively program one or multiple ISP devices at a time.

For the ispLSI devices, the active $\overline{\text{ispEN}}$ signal enables the programming mode. By driving $\overline{\text{ispEN}}$ low, all of the device I/Os are put into a high-impedance state for programming and the programming functions for SDI, SDO, Mode and SCLK are enabled.

For the ispGDS and ispGAL devices, on the other hand, the I/Os are put into a high-impedance state when the programming state machine goes into the Command Shift State. The ispGDS and ispGAL devices do not use a dedicated $\overline{\text{ispEN}}$ pin for this function.

Most shift operations, such as ID shift and command shift, are the same for the ispLSI and the ispGDS/ispGAL devices. However, one difference exists in the way that the address and data are shifted into the devices. The ispLSI devices have separate address and data shift commands. The row(s) are selected by the address that is shifted-in prior to each programming command. The data can then be shifted with the data shift instruction. With ispGDS and ispGAL devices, both address and data are shifted-in with a single shift command (the address is part of the Data shift register). When executing commands that only require a row address, a dummy data stream or no data can be shifted in place of the data stream.

ISP Daisy Chain Programming

A specific illustration of multiple device programming in a daisy chained environment is shown in Figure 1. The example shows ISP programming aspects such as identifying the devices in the daisy chain, shifting commands, bypassing devices, and executing commands.

All of the programming state machines run in parallel which keeps the devices synchronized. The programming information for the ISP devices is summarized in Table 13. Similar details for any ISP device can be found in the ispLSI Architecture Description in the data book and in the appropriate device data sheet.

Table 12. Common Features of the ISP Device Families

Common Features	**ispLSI**	**ispGDS/ispGAL**
ID shift register length	8-Bits	8-Bits
Command shift register length	5-Bits	5-Bits
Programming signals	MODE, SDI, SDO, & SCLK	MODE, SDI, SDO, & SCLK
State Machine	3-state with same MODE & SDI controls for state transitions	3-state with same MODE & SDI controls for state transitions
FLOWTHRU instruction	Yes	Yes
Different Features		
$\overline{\text{ispEN}}$ signal	Yes	No
Address & Data shift register	Different shift instructions for address & data	Both address and data is shifted with one shift command
Fuse map sizes	Varies for different high density devices	Varies for different low density devices

The first procedure of the programming sequence identifies the devices in the ISP chain. The following procedure describes one way of reading the device IDs.

```
Load_ID Procedure
set ispEN = L
set MODE, SDI = H, L
clock SCLK  (Load ID)
Continue to Shift_ID Procedure ...
```

At this point, the 8-bit ID registers are loaded with the hardwired device IDs. Figure 26A shows the configuration of the ID shift registers.

After the device ID has been loaded, the following shift ID procedure sequentially shifts the IDs through to the last device's SDO. While the ID is being shifted out, keep SDI at a known logic level so that the end of the ID stream can be identified. This is especially important when there are an unknown number of devices in the ISP daisy chain. By detecting a sequence of eight zeros or eight ones, the ISP controller can detect the end of the ID string.

```
Shift_ID Procedure
... Continued from Load_ID Procedure
set MODE, SDI = L, H
clock SCLK (Shift ID)
if last 8 SDO = H then go to End
else go to Shift_ID
End
```

Now, all of the devices within the ISP daisy chain and their order can be properly identified. The next step is to match the proper JEDEC fuse map file to the appropriate device. There are several programming options at this point. To simplify the programming routines however, this example programs the devices one at a time. In programming time critical applications, the daisy chained devices can be programmed in parallel. The parallel programming routines must keep track of the differences in the fuse map lengths between different ISP devices.

The following procedures illustrate how to shift commands, shift data, and execute commands to program the ispGAL22V10. Since the ispGAL22V10 is the second device in the ISP daisy chain, these procedures also illustrate how to put the other devices into flow-through mode. The following procedure shifts the SHIFT_DATA command into the ispGAL22V10 and the FLOWTHRU command into the rest of the ISP devices.

```
Load_Command Procedure
... Continued from end of Shift_ID Procedure
set MODE, SDI = H, H
clock SCLK (Shift State)
set MODE = L
Loop
set SDI = command stream (Figure 26B)
clock SCLK (Shift Command)
End Loop
End Procedure
```

Table 13. ISP Programming Information

Description	ispLSI 1032	ispGAL22V10	ispGDS22	ispLSI 2032
Device ID (8-bits)	0000 0011	0000 1000	0111 0010	0001 0101
Command Register	5 bits	5 bits	5 bits	5 bits
Address Shift Register	108 bits	n/a	n/a	102 bits
Data/Addr. & Data Shift Register	160 bits	(6+132) bits	(6+18) bits	40 bits

```
Execute_Command Procedure
set MODE, SDI = H, H
clock SCLK (Execute State)
set MODE = L
Loop 138 times
    set SDI = data stream (Figure 26C)
    clock SCLK (Execute SHIFT_DATA Command)
End Loop
set MODE, SDI = H, H
clock SCLK (Shift State)
End Procedure
```

At the end of the Execute_Command procedure, the state machine is returned to the Shift State. This readies the devices for another command shift procedure. For the ispGAL22V10, the DATA_SHIFT instruction of 138 bits includes the row address and the data associated with the row. Similar procedures can be used to complete the programming of the ispGAL22V10.

Figure 26A. ID Shift Register Configuration

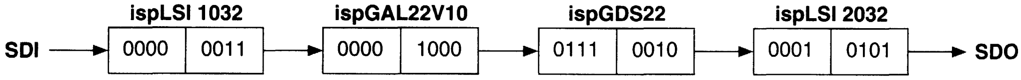


Figure 26B. ISP Command Stream

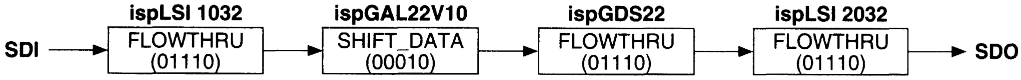
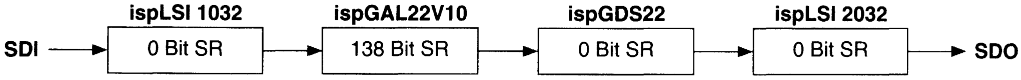


Figure 26C. ISP Data Stream



Software Basics

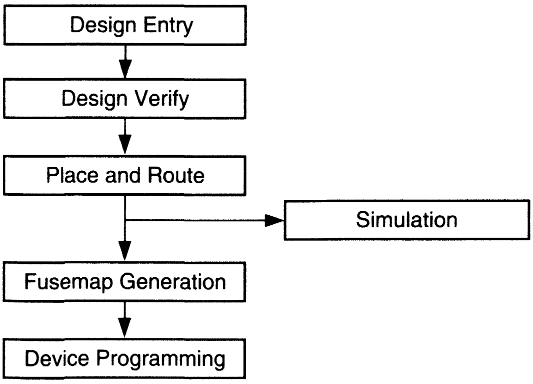
Introduction

This section explains the wide variety of software tools available for the Lattice ISP devices. First, the general ISP design flow is revisited to help the reader understand how each software tool fits in the flow. Second, the software tools that generate a JEDEC fuse map from a logic design are highlighted, finishing the first half of the ISP general design flow. Third, the software tools that program the ISP devices from JEDEC fuse maps are described, completing the second half of the ISP general design flow. Finally, the section closes with an additional software consideration: choosing the proper ATE programming approach.

ISP Design Flow

Once the system design has been organized into functional components, and the logic functions which need to be incorporated in the selected components are defined, the logic design phase begins with design entry. The general design flow is shown in Figure 1. Generating a fuse map from the design and programming the ISP device completes the general ISP design flow.

Figure 1. General Design Flow



Design Entry to Fuse Map Generation

Introduction

The basic function of PLD logic design entry and fitter/ compiler software is to convert a logic design into a JEDEC standard programming fuse map file (also known as a JEDEC file). There are several ways to create a JEDEC file from a design. The software you choose depends on which type of device you are using and what type of design entry you want to perform. To create a JEDEC file from a design intended for a low-density device, you can use most standard logic compilers including ABEL and CUPL. Additionally, Lattice offers a stand-alone logic compiler for ispGDS devices. To create a JEDEC file from a design intended for a high-density device, such as the ispLSI 1000, 2000, or 3000, the design can be entered using several software environments including Lattice's pDS Boolean entry, pDS+ ABEL, pDS+ Viewlogic, pDS+ LOG/iC, and pDS+ Cadence.

ispGAL Third-Party Logic Compiler Support

For design engineers who are familiar with standard third-party compiler software packages, Table 1 shows the third-party logic compilers that support all ispGAL22V10 devices.

Table 1. ispGAL22V10 Logic Compiler Support

Vendor	Logic Compiler
Accel Tech.	Tango PLD
Cadence	PIC Designer Composer
	PIC Designer Concept
Data I/O	ABEL
ISDATA	LOG/iC
Logical Devices	CUPL
Mentor Graphics	PLSynthesis II
Minc	PLDesigner-XL
OrCAD	OrCAD PLD
Omation	Schema-PLD
Viewlogic	ViewPLD

ispGDS Compiler Support

To simplify the development of ispGDS devices, Lattice offers an ispGDS assembler named "GASM" which processes the input ASCII files to generate the JEDEC compatible fuse map files required for the ispGDS devices. Free ispGDS assembler software is available from the Lattice Hillsboro BBS at 503-693-0215 under GDSPKG.ZIP file. This software is also available on diskette by calling the Lattice Hotline at 1-800-327-8425 (FASTGAL). For design engineers who are familiar with standard third-party compiler software packages, ABEL from Data I/O and CUPL from Logical Devices also support all ispGDS devices.

Using the ispGDS Compiler

The compiler will accept an ASCII text file containing the ispGDS programming instructions, and will create JEDEC and .DOC files.

Compiler Syntax

The basic compiler syntax supports inserting comments, title, device type, pin assignments, and input/output assignments. The ispGDS compiler source file comment lines are denoted with quotation marks at the beginning of the comment lines. The title is defined with the key word "title = ". Any text following the "title =" key word that is within single quotes is defined to be the title of the design. Similarly, the device type is defined by the key word "device =" followed by one of the three valid device types – ispgds22, ispgds18, ispgds14. The compiler syntax also allows the user to assign pin names by typing in a 10 character pin name followed by at least a single space, the "pin" key word and the pin number. This pin assignment is optional since the compiler syntax allows the user to use the "pin" key word and the pin number directly in the input/output assignments.

The output pins are assigned on the left side of the equation and the input pins are assigned on the right side of the equation. To assign an output pin to either high or low, simply assign "H" or "L" respectively on the right side of the equation. If you need to assign an input pin to multiple output pins, use one line for each assignment, as shown in the following example. In the example below, pin 28 is an input that is routed to three outputs — pin 1, pin 2 and pin 3. Furthermore, each output's polarity can be individually defined. The example shows pin 3 as an active low polarity whereas pin 1 and pin 2 are defined to be active high polarity.

```
 pin 1 = pin 28
 pin 2 = pin 28
!pin 3 = pin 28
```

Assembling a File

To use the assembler, create an ASCII ispGDS source file, then invoke the assembler from the DOS command line. For example:

```
gasm <test.gds>
```

where test.gds is the name of the ispGDS source file. GASM will create a JEDEC file with the same base name, and a .JED extension, like "test.jed," and a doc file with a .DOC extension, like "test.doc." Listing 1 illustrates an ispGDS source format.

Listing 1. ispGDS Source Format

The following text is an example of a ispGDS source file.

```
"This is a comment (line begins with quotation mark)
title = 'DIP SWITCH REPLACEMENT CONFIGURATION'

" the ispgds device type (ispgds22, ispgds18, ispgds14)
device = ispgds22

" pin names are defined as follows

pin_name  pin 28

" pin 1 is an output connected to pin 28
pin 1 = pin_name
pin 2 = pin_27

" pin 3 is another output connected to pin 28

pin 3 = pin 28

" pin 5 is always high
pin 5 = h

"pin 6 is always low
pin 6 = l
pin 8 = pin 22

"! defines the inverted output for pin 9
!pin 9 = pin 20

pin 10 = pin 19
pin 12 = pin 17
pin 13 = pin 16
pin 14 = pin 15
```

Notes

If you get an error regarding "pin 0", you may have duplicated an output pin assignment (by assigning different input signals to the same output pin). Refer to the line number in the assembler error message to locate the source of the problem.

ispLSI Design Entry and Fitter Support

Lattice offers a wide variety of design entry and device fitter tools which handle logic entry, device compilation, and device programming. These tools support a variety of user interfaces and entry methods including: Microsoft Windows® GUI, Cadence Concept/Verilog-XL, Mentor Graphics, Synopsys, Viewlogic ViewDraw/ViewSynthesis and PROcapture/PROsynthesis, Data I/O ABEL HDL or VHDL and Synario, ISDATA LOG/iC Design System, and ORCAD. Design flows illustrating these development software systems are shown in Figures 2a, 2b, 2c, 2d, 2e, 2f, 2g, and 2h.

Design Entry/Synthesis

Lattice's pDS Software allows the user to manually partition the logic to control design fit and performance. Using the Microsoft Windows environment, logic functions are placed into Generic Logic Blocks (GLBs) and I/O Cells. This can be done by using the Edit, Cut, Copy, and Paste functions to enter Boolean equations and/or predefined functions from the Lattice Macro or user libraries.

In the ABEL environment, in addition to Boolean design entry, the ABEL HDL formats allow high-level descriptions of counters, adders, comparators, etc. These HDL languages also support state machines, truth tables and case constructs for behavioral design implementations. The Lattice interfaces allow many existing PLD designs to be easily integrated and converted into ispLSI devices.

For standard CAE schematic designs, the pDS+ Cadence, pDS+ Synario, pDS+ Mentor, pDS+ Viewlogic and pDS+ ORCAD software applications provide support for graphical and hierarchical logic implementation using the Lattice library of primitives and macros. The interfaces also allow easy integration of system or user-created functions into a hierarchical schematic using either a top-down or bottom-up design methodology.

The Lattice Synopsys Synthesis Libraries offer design entry using device-independent Verilog HDL or VHDL design languages. These designs are synthesized by Synopsys Design Compiler or FPGA Compiler using the Lattice Synopsys Libraries into an ispLSI device-compatible EDIF design netlist. The pDS+ Synopsys fitter will then partition, optimize, and fit the design into ispLSI devices. The EDIF design netlist may also be imported into either Cadence Concept, Viewlogic, or Mentor Graphics schematic capture design tools where device implementation control attributes are applied. The Lattice pDS+ Cadence, pDS+ Mentor, or pDS+ Viewlogic Fitters then fit the designs.

Design Verification

After entering the logic for the design, the next step is verification. Verification checks the logic entries for syntax and design rule violations, minimizes the logic equations as necessary, and then maps the logic to the physical gates in the GLBs and IOCs. You can verify each cell individually, as each step is completed, or you can verify your whole design after all cells are completed.

Partitioning

Partitioning using the pDS Software is done by the user as part of the design entry process. The advanced pDS+ Fitter tools incorporate Lattice's automatic partitioner which accepts converted data from designs entered in Cadence, Mentor Graphics, Synopsys, Viewlogic, ABEL, Synario, LOG/iC, and ORCAD tools. Lattice-specific attributes for design entry are available to guide the partitioner in order to optimize device features and performance.

Place and Route

All Lattice design tools offer automatic place and route. This entails placing the GLB (Generic Logic Block) and IOC (I/O Cell) logic and routing (or interconnecting) the source signals to their destinations. In the ispLSI devices, the Global Routing Pool (GRP) provides fast interconnects from external inputs and GLB feedback to GLB inputs. The Output Routing Pool (ORP) provides flexible interconnects from GLB outputs to external pins.

Post-route Simulation

After place and route, a netlist for full timing and functional simulation may be passed to the Mentor Quicksim II, Viewlogic PROsim™ or ViewSim™, or Cadence Verilog-XL simulators. Board- and system-level behavioral simulation models are available from the Logic Modeling division of Synopsys.

Documentation

Report files, containing partitioned equations and pin-out information, may be generated for routed or unrouted designs. The pDS and pDS+ Software can also generate reports with post-route maximum timing delays. These reports give specific details about the design and device resources utilized, GLB usage and fanout, pin names, and signal attributes in text and table formats.

Figure 2a. pDS Design Flow

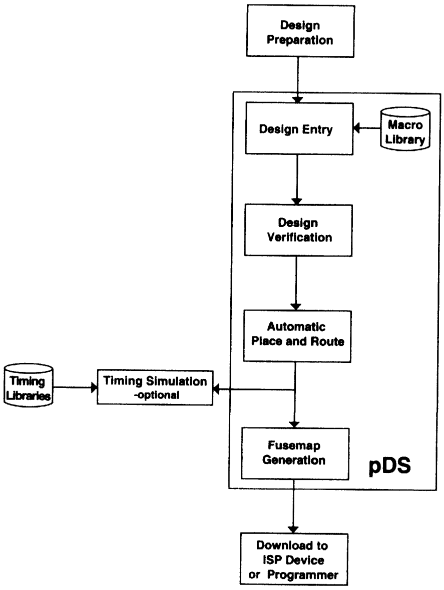


Figure 2b. pDS+ Cadence Design Flow with Synopsys Option

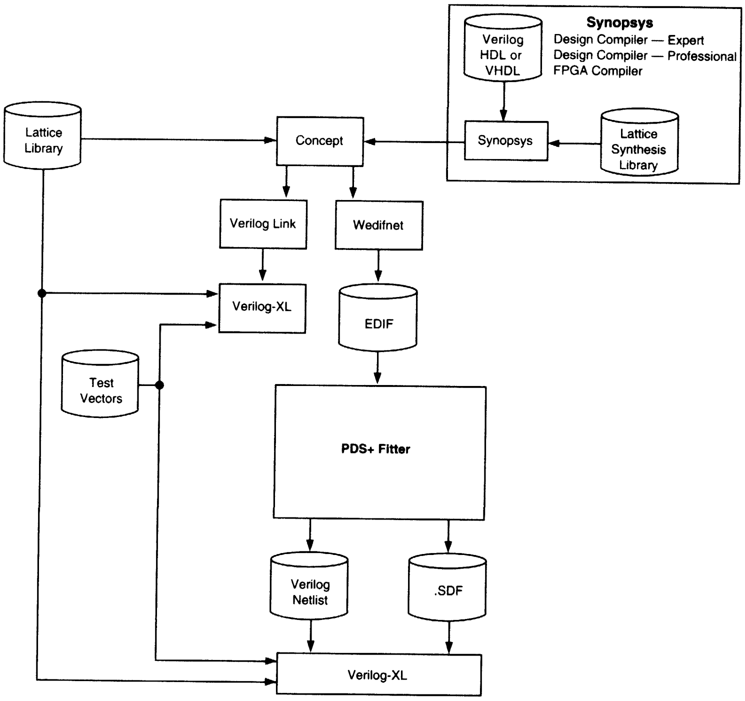


Figure 2c. pDS+ Mentor Design Flow with Synopsys or Autologic Option

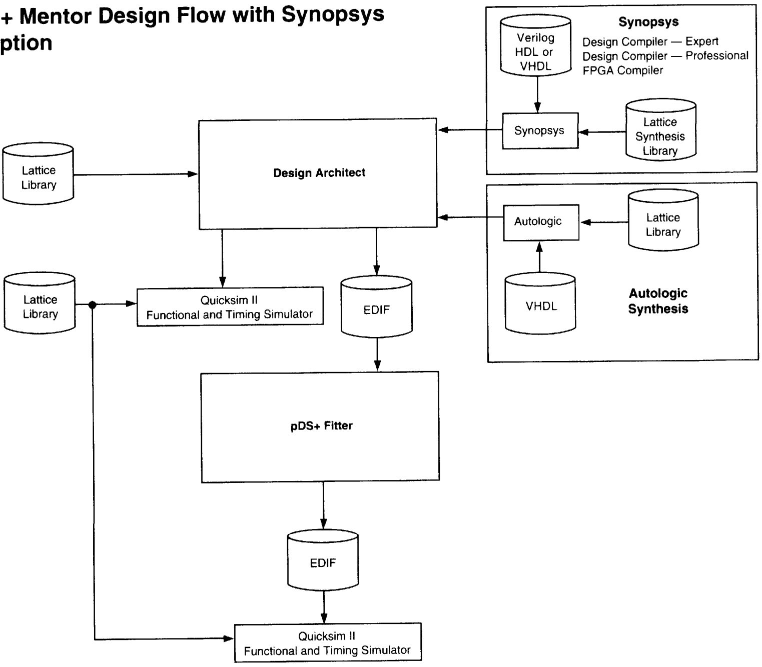


Figure 2d. pDS+ Synopsys Design Flow

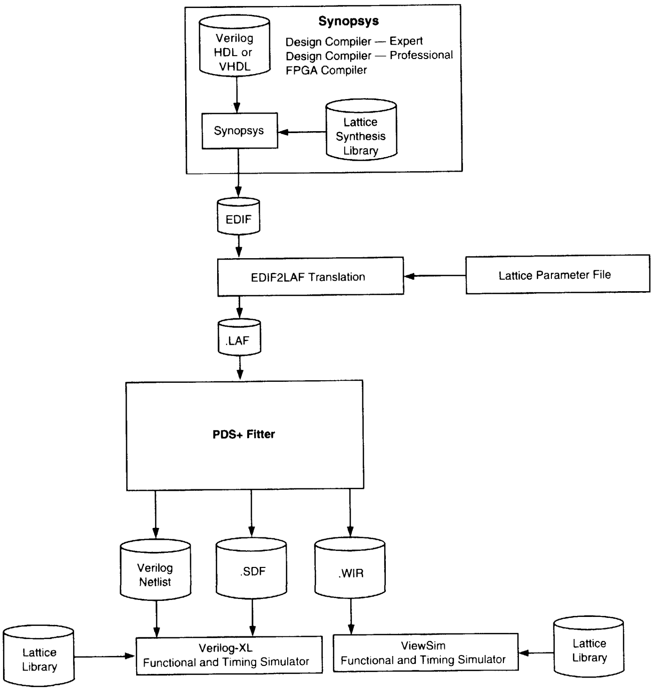


Figure 2e. pDS+ Viewlogic Design Flow with Synopsys Option

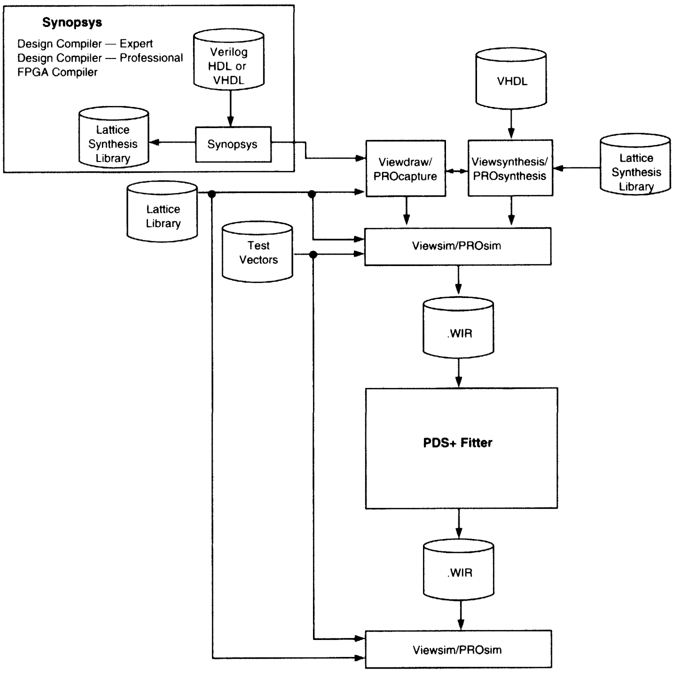


Figure 2f. pDS+ ABEL and pDS+ Synario Design Flow

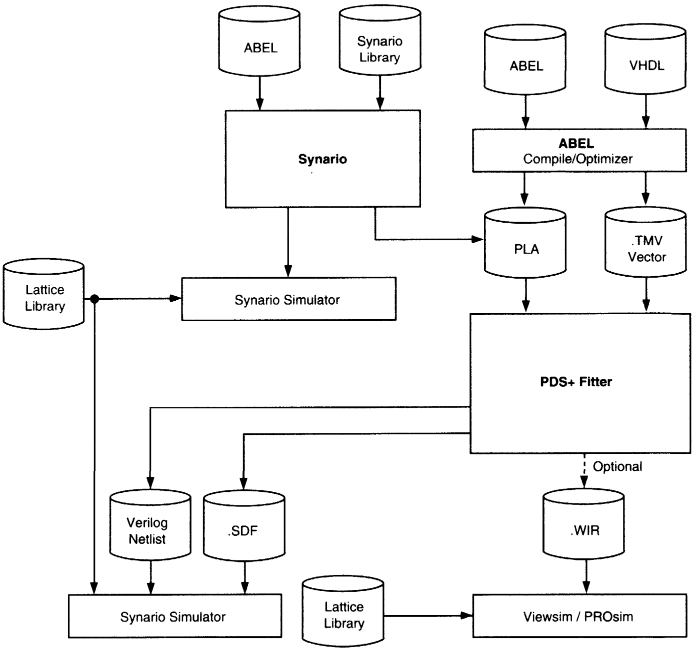


Figure 2g. pDS+ LOG/iC Design Flow

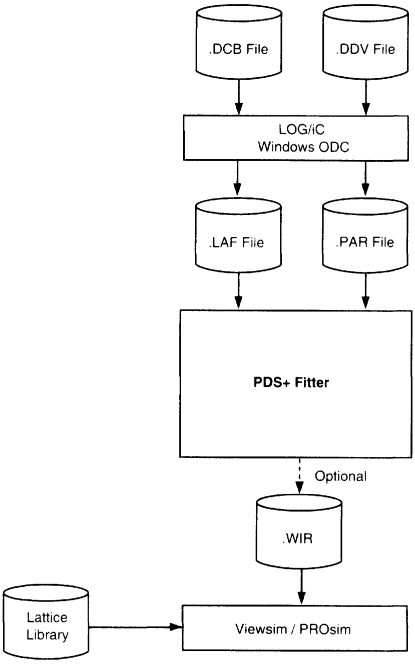
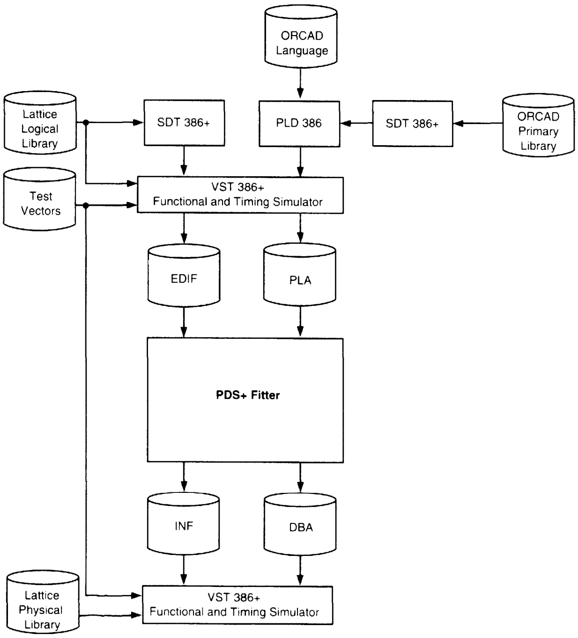


Figure 2h. pDS+ ORCAD Design Flow



Device Programming

Programming information is generated on a routed design by the FuseMap Generator for a specific ispLSI device. It is an ASCII file written in the JEDEC format. Using ABEL software, the user may optionally append test vectors to the JEDEC file. This allows post-programming functional testing on the actual device.

ISP Programming Software

Introduction

Once the JEDEC file has been generated for a given design, the design information, which is stored in the JEDEC file, must be downloaded into the proper device. The download method depends on the hardware available and what design stage you are in. For example, you might program the system with ISP devices during prototyping using a PC. Then, when the system goes to full production, you can use ATE for programming. Finally, if field upgrades are necessary, you can use the system's embedded microprocessor to reprogram the ISP devices. Table 2 summarizes the download methods supported by Lattice.

Table 2. ISP Programming Platform and Download Methods

Programming Platform	Download Methods
PC	ISP Daisy Chain Download for Windows ISP Daisy Chain Download for DOS ISP Serial Programmer ispCODE C++ Source Routines
Workstation	ISP DOWNLOAD for the Sun ispCODE C++ Source Routines
Embedded Processor	ispCODE Executed by Microprocessor
ATE	ISP Serial Programmer- Test Vector Generation Portion ispCODE C++ Source Routines
Third-Party Programmer	Standard JEDEC File Download

ispCODE

Overview

This section is a guide to using the Lattice ispCODE software in custom software applications. ispCODE is C++ source code that you can use to program one or more ISP devices. The ispCODE software contains a library of programming routines designed to be easily ported to different applications and platforms. You can use the classes in this function library to implement your specific requirements without knowing the lower level details of how the ISP devices are controlled. Table 3 lists the classes in the function library.

Also included in the ispCODE package is C++ source code for an example Windows application which programs, reads, and verifies ISP Devices using the parallel port of your PC. You can use this application as an example of how to access the ispCODE library. You can easily modify ispCODE to develop a customized Windows application for ISP programming. More information on this application, called ISP Serial Programmer, follows this section.

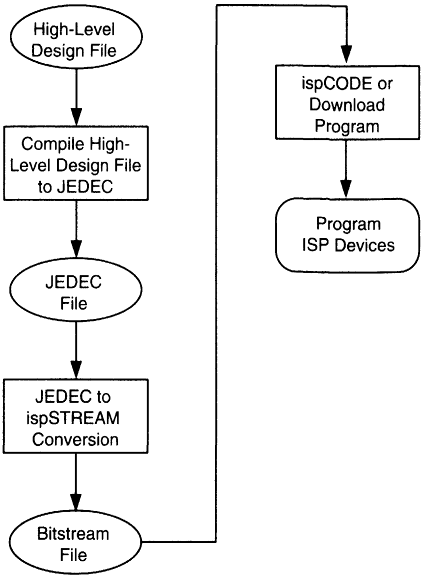
If you only need to access the ISP devices through the parallel port of a PC, and do not require any customization, you can use Lattice's ISP Daisy Chain Download software. This is a stand-alone program that performs commonly used ISP functions. It can be used without modification.

If you are developing an ISP application on a platform other than a PC or Sun Workstation, or you need custom features, you will need the ispCODE library. To make customizations easier, you will find that platform-specific issues, such as port addressing, and timing delays, are localized in a few routines.

The ISP Design Flow

Figure 3 shows the typical design flow for an ISP application. The user starts with a high-level design description in ABEL, Viewlogic, CUPL, pDS, or some other form. For the ispGAL 22V10 and ispGDS devices, this description is compiled by ABEL or another supported compiler and translated into an ASCII JEDEC file. For the ispLSI devices, the description is compiled using one of the Lattjce pDS+ Fitters, which produces a JEDEC file. In either case, the JEDEC file then undergoes another transformation into a binary ispSTREAM file. This file format is a compressed form of the JEDEC file, and is small enough to be stored in an EPROM in the user's target system if required (handy for microprocessor-driven programming). The ispCODE uses the ispSTREAM format to transfer specific design information to and from the ISP devices.

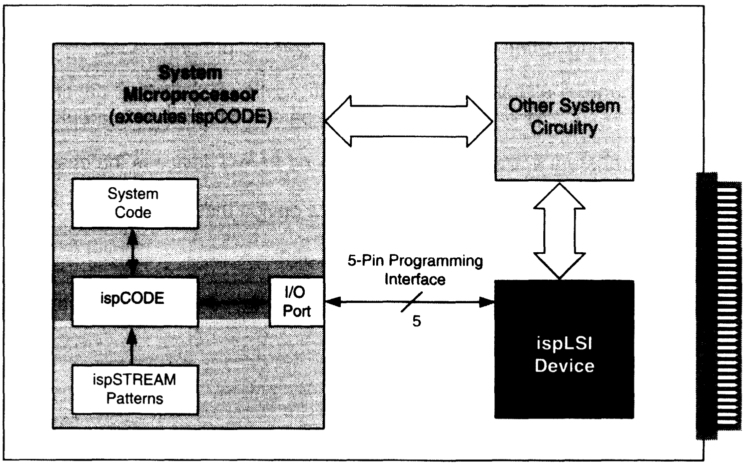
Figure 3. ISP Programming Process


High-Level Design File
Compile High-Level Design File to JEDEC
JEDEC File
JEDEC to ispSTREAM Conversion
Bitstream File
ispCODE or Download Program
Program ISP Devices

A Typical ispCODE Application

A generalized block diagram may help you understand how ispCODE is used in a typical application (Figure 4). In this application, the ispCODE is modified as needed, and runs on a target microprocessor. The microprocessor interfaces with the ISP devices through a four- or five-wire TTL level interface (five signals are required for use with the ispLSI devices, and four signals for the ispGDS™ and ispGAL22V10).

Figure 4. Configuring an ispLSI® Device from an On-Board Microprocessor



Advantages of C++

The ispCODE software was developed using Borland C++ 3.1. However, the only compiler-specific features used are in the demo Windows interface, since Borland's OWL 1.0 Library was used. If you are only using the function library, then there should be little dependence on the compiler, other than to support standard C++ constructs.

Since the ispCODE is written in C++, you will find it easy to port to other applications. The ispCODE classes correspond to the physical hardware you are interfacing to, making it easy to understand. All access to the ISP devices occurs through the parallel port of the PC. The PortClass, the class used to access the port that interfaces to the ISP connector, contains all the functions and data structures necessary to control the parallel port. These functions are described in "Function Library Interface" and are listed in Table 3 in this section. If you need to access the parallel port, or change the parallel port method, then this is the only class you need to be concerned with. Also, through C++ support of inheritance, you can easily extend the functionality of a class, and still be able to upgrade the software to a newer version without impacting your customization. For example, you may need to add some custom control signals to the parallel port. Instead of modifying the parallel port class PortClass directly, create your own port class that is derived from PortClass. Any new functions you add should then be added to your port class. If Lattice releases a new version of the ispCODE in the future, simply recompile with the new library.

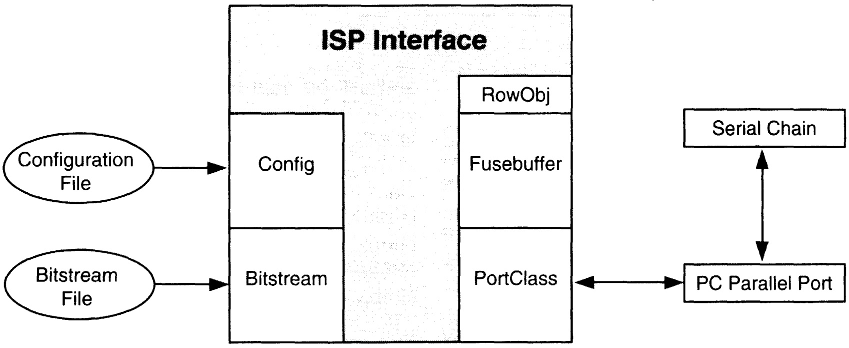
The software is structured into a user interface portion (serial.cpp) which calls a function library (funclib.cpp) to work with the serial chain. This enables you to modify the Windows interface easily without worrying about the lower level details of manipulating the serial chain, and also allows you to easily remove the windows interface portion and use the high level calls into the function library if you have no need for the Windows interface.

The different classes used in the function library are listed in Table 3. Figure 5 shows how these classes interact and how they are instantiated inside the ISPInterface class.

Table 3. Different Classes Used in the Function Library

ISPInterface	This class controls all access to the daisy chain hardware, and uses other classes to access the configuration file, bitstream files, and PC port interfaces.
Config	This class handles all interface with the configuration file, such as parsing the file, verifying that the bitstream files exist, and storing the parsed information in memory.
RowObj	This class transfers a row of information to and from a device in the chain.
Fusebuffer	This class contains the fuse information for a device that is being read or verified.
PortClass	This class controls all access to the parallel port.
Bitstream	This class reads a bitstream file, and is used to retrieve fuse info from the bitstream file.

Figure 5. Interaction of ISP Interface Class Functions



Customizing ispCODE

To develop a custom application using ispCODE, please follow the guidelines listed below:

- Use high-level calls to the function library, and avoid modifying the function library. As Lattice adds new devices or enhances programming algorithms, the contents of the function library may change. If you change the contents of the function library, you may find updating to newer revisions difficult. If you access the function library through the function interfaces described in the Function Library Interface, then you will be able to replace the function library with a newer version without any impact.
- Timing is critical for successful programming. ispCODE uses the Windows Multimedia timer for high resolution timing (accuracy of 1 ms). If you port to a different platform, then you will need to modify this routine to support your target hardware, keeping in mind that whatever approach you use should ensure that the minimum and maximum times as specified in the Lattice Data Book will be met. Refer to the section on Timing for further details.
- If you are porting to another platform other than the PC, you will need to modify how the ISP interface signals are written out to the parallel port. Refer to the section on Port Addressing for further details.

Function Library Interface

You can perform all the ISP programming functions (reading, writing, and verifying) through three calls to functions in the function library. There are a few function calls that you must make for initialization prior to performing these functions. However, you will see that for almost all applications these few function calls will provide all the versatility you need.

These function calls are summarized in Table 4. For more detailed information, refer to the section on Programming, Verifying, and Reading the Serial Chain.

Porting Considerations

In general, there are only two routines that may need to be modified for running ispCODE on different platforms: the routine to control timing (ISPInterface::Wait) and the class to access the port that interfaces to the ISP connector (PortClass).

Timing

You must ensure that minimum and maximum program times are met through the use of a delay routine. In general, it is more important to ensure that a minimum programming time is met. If you modify the timing routine, the delay errors must occur on the large side. When the software calls this routine, the routine simply consumes CPU time until the timing delay is met.

In this example, ispCODE achieves 1 ms resolution through use of calls to the Multimedia services of Windows 3.1. This is shown in the code fragment in Listing 2.

This routine is passed an integer value that represents the amount of time in milliseconds that the Wait routine should wait before returning. Since waiting is the only function that this routine performs, adapting it to different platforms is an easy task.

Table 4. Function Calls

Class::Function	Purpose
PortConfig	Class to access the parallel port, and drive the ISP interface signals. You will need to modify it for use on platforms other than the PC.
ISPInterface::IDChain	A member function of ISPInterface that builds a list of IDs in the serial chain.
Config	A class to parse and load the configuration file.
ISPInterface::ChainExecute	A member function of ISPInterface that executes command on a particular device (read, write, or verify) in the serial chain.
ISPInterface::Wait	A member function of ISPInterface that waits for a specified amount of time, then returns. Since it uses the Windows Multimedia timer services, it will need to be modified for different platforms.

Listing 2. Excerpt from <serial.cpp>

```
ISPInterface::Wait(int val){
// val is the time in milliseconds
   long unsigned int current_time=0,start_time;
....
   timeBeginPeriod(1); // set to one millisecond and reset time count
   current_time=timeGetTime();      // get the current value
   while((timeGetTime()-current_time)<(val)){  // no chance of rollover
     NULL; // hog cpu time until finished
   }
   timeEndPeriod(1);    // free up this time
   return(0);
}
```

Port Addressing

In the Windows environment, ispCODE uses the parallel port of the PC to access the ISP Serial Chain. If you decide to use a different hardware platform, then obviously this interface class will need to be changed as well. The PortClass class has a member function which corresponds to each bit of the ISP interface (SDI, SDO, MODE, $\overline{\text{ispEN}}$, SCLK). To set or clear a bit, call the appropriate member function with the desired bit value. PortClass also has a function (Toggle Clock) which can be called without any value. It toggles the clock from zero to one, then back to zero (Listing 3).

The routines to change the ISP signal values in turn call two lower level routines: GetPortBit and SetPortBit, with a bitmask which is used to set or clear the desired bit on the port. These two routines are the only ones that directly access the parallel port of the PC, through the use of import and export functions. If you port the code to a different platform, you can simply replace the import and export functions they call with routines for your target hardware. Also note that the SetPortBit routine supports some special configurations for Lattice Demo Board use, through the use of a switch statement. You can safely delete these options for your application (Listing 4).

Listing 3. Excerpt from <funclib.cpp>

```
/*
ChangePort: Modify the port address pointers InPort and OutPort
*/
     void ChangePort(int iport, int oport){
        InPort=iport;
        OutPort=oport;
     }
/*
 SetClock: set the clock bit
*/

     void SetClock(int val) {
        SetPortBit(CLOCK, val);
     }
/*
 ToggleClk: Generate a clock pulse
*/
     void ToggleClk(void){
        SetClock(0);
        SetClock(1);
        SetClock(0);
     }
/*
 SetSDI: Set the Serial Data In bit
*/
     void SetSDI (int val) {
        SetPortBit(SDI, val);
     }

/*
 SetISPEN: Set the ISP Enable bit
*/
     void SetISPEN (int val){
        SetPortBit(ISPEN, val);
     }
/*
 SetMode: Set the Mode bit
*/
     void SetMode ( int val){
        SetPortBit(MODE,val);
     }

/*
 ReadSDO: Read the SDO bit
*/
     int ReadSDO ( void ){
        return(GetPortBit(SDO));
     }
```

Listing 4. Excerpt from <funclib.cpp>

```
/*
Get the value of a bit from the port
*/
int PortClass::GetPortBit(unsigned char BitMask ){
           unsigned char tmp;
           assert ((InPort==IPORT0)||(InPort==IPORT1)||(InPort==IPORT2));
           tmp=inportb(InPort);
           if((tmp&BitMask)==0)return(0);
           else return(1);
       }

/*
   Set the bit in a port to a specified value
*/

void PortClass::SetPortBit(unsigned char BitMask, int val){
           assert ((val==1)||(val==0));
           assert ((OutPort==OPORT0)||(OutPort==OPORT1)||(OutPort==OPORT2));

// do the special decoding for demo board versus standard usage.
// the serial demo board needs to do some decoding for mode. the
// standard usage doesn't. the demo board also has a serial shift
// register clock that requires special decoding.

           if(BitMask==MODE){
              switch (ModeSel){
                 case DAISY: if(val == 1){
                                PortVal |= MODE;
                                PortVal &= ~MODE2;
                             }
                             else PortVal &=~( MODE | MODE2 );
                             break;
                 case GDS   : if(val == 1){
                                PortVal |=MODE2;
                                PortVal &=~MODE;
                             }
                             else PortVal &= ~(MODE | MODE2);
                             break;
                 case I22V  : if(val == 1)PortVal|=(MODE|MODE2);
                             else PortVal &= ~(MODE|MODE2);
                             break;
```

Note: The cases DAISY, GDS, and I22V are used for some special addressing for the Lattice Applications Demo boards. You can ignore these cases, and just modify the STANDARD case for your application.

```
                 case STANDARD : if(val== 1)PortVal|=BitMask;
                                 else PortVal &= ~BitMask;
                                 break;
              }
           }
           else if (BitMask == SRCLK) { //special stuff for serial clock

              if (val == 1){
                 PortVal &=~MODE;
                 PortVal |=MODE2;
              }
              else PortVal&=~(MODE|MODE2);
           }
           else { // not a serial clock or mode bit
              if (val == 1){
                 PortVal|=BitMask;
               }
              else {
                 PortVal&=~BitMask;
              }
           }
           outportb(OutPort, PortVal);
       }
```

ispSTREAM File Size

If you are interested in putting the ispSTREAM files in EPROM or ROM (to support in-system microprocessor - driven programming for example), the size of the ispSTREAM files will be of interest. The sizes of the ispSTREAM files are listed in Table 5. Use this table as a rough guide; the actual file sizes may be slightly different, but should be within a few bytes of these values.

Table 5. ispSTREAM File Sizes

Lattice Device	ispSTREAM File Size (bytes)
ispGDS14/18/22	51
ispGAL22V10	739
ispLSI 1016	1922
ispLSI 1024	3062
ispLSI 1032	4322
ispLSI 1048	7202
ispLSI 1048C	9362
ispLSI 2032	1022
ispLSI 3256	15212

Configuration File Syntax

The syntax for the configuration file is quite simple. It is an ASCII text file that defines the ispSTREAM files to be loaded into the ISP devices. There is one line per device, and the first line corresponds to the first device (the device whose SDO pin is connected to the programming port). The basic syntax is the device name followed by the ispSTREAM file name. If you leave the ispSTREAM file name off, that device is not programmed or verified. The quote character (“) is treated as a comment: anything following the quote is ignored. An example is shown in Listing 5.

Programming, Verifying, and Reading the Serial Chain

If you want to use the function library in your own application, use the following list as a guide to which functions to run and when to run them. The code fragments are from the Windows interface code (serial.cpp). Use the following list as a guide as to how the high-level routines should be called, and the order in which they should be called:

1. Use an ASCII text file format for the configuration file (see the preceding section "Configuration File Syntax" for more information). You can then pass the configuration to the function library file and let it handle the job of parsing and validating the file information.
2. Scan the serial chain before doing anything else, and verify that the chain is valid. Do this by calling the function ISPInterface::IDChain. The code fragment from serial.cpp shows how this is done (Listing 6).
3. Now call the function Config::Readline() repeatedly. This verifies that the file contents match what is in the chain (Listing 7).
4. If the config file has been successfully parsed by this point, all the necessary data structures have been created for programming, reading, and verifying the ISP serial chain. Each of these three functions can now be accomplished by a single function call (Listing 8).

Listing 5. Configuration File Syntax

```
ISPGDS14   gds.isp   “first device
ISP1032              “second device, no file to load
ISP22V10   22v.isp   “third device
```

Listing 6. Excerpt from <serial.cpp>

```
void  TMainWindow::IDENTIFYCHAINCONFIGURATION(RTMessage){
PC->SetPortBit(ISPEN,0);
if(isp->IDChain()){
...
MessageBox(HWindow,"The serial chain was read successfully
}
  else {  // display the error
...
    CF->Reset();
}
```

Listing 7. Excerpt from <serial.cpp>

```
void TMainWindow::LOADCONFIGURATIONFILE(RTMessage)
{
CF->Reset(); // reset all the params -- in case we're loading 2nd, 3rd... time
verified=T;  // flag for successful parsing of config file
fh=Note: insert the file handle to your config file here
CF->cfh=fh;
...
   do {
     rc=CF->ReadLine();
     if (strstr(rc,"ERROR")!=NULL){
        verified=F;
        break;                          //stop at the first error
     }
   } while (strstr(rc,"eof")==NULL);    // while its not an EOF

   if(verified==F){
Note: insert config file syntax error handling code here
        CF->Reset();
   }
   if((fh!=NULL)&&(verified==T)){
   /*
      Now there are two lists of devices -- one in CF->Devices, and one in
      isp->Devices. Make sure the two lists match up. (CF is the list from
      the config file, isp is the list from the scan of the serial chain.)
   */

   // first check to see if the number of devices are the same. If not, stop

      if(CF->devcnt != isp ->ChainLength){
Note: insert length mismatch message here
         CF->Reset();
         goto lcfreturn;
      }
      // now make sure the device lists are the same
      for(i=0;i<CF->devcnt;i++){
         if(CF->Devices[i] != isp->Devices[i]){
            // mismatch
Note: insert device mismatch message here
            CF->Reset();
            goto lcfreturn;
         }
      }
Note: insert successful parsing of config file message here
   }
   else {
Note: insert unsuccessful parsing of config file message here
      CF->Reset();
   }
lcfreturn: //return
   CF->CloseFile();
   if(MsgBuffer != NULL)free(MsgBuffer);
   return;
}
```

Listing 8. Excerpt from <serial.cpp>

```
void TMainWindow::PROGRAM(RTMessage)
{
.....
   PC->SetPortBit(ISPEN,0);
   isp->Chain_Execute("Programming Status",CHAIN_PROGRAM, isp->Devices,CF-
>DeviceISP, isp->ChainLength);
   PC->SetPortBit(ISPEN,1);
.....
}

void TMainWindow::VERIFY(RTMessage)
{
   PC->SetPortBit(ISPEN,0);
   isp->Chain_Execute("Verify Status",CHAIN_VERIFY, isp->Devices,CF->DeviceISP,
isp->ChainLength);
   PC->SetPortBit(ISPEN,1);
}

void TMainWindow::READD(RTMessage)
{
   PC->SetPortBit(ISPEN,0);
   isp->Chain_Execute("Read Status",CHAIN_READ, isp->Devices,CF->DeviceISP, isp-
>ChainLength);
   PC->SetPortBit(ISPEN,1);
}
```

ISP Serial Programmer

ISP Serial Programmer is a compiled version of the example Windows application (serial.cpp) included with ispCODE. This application not only programs, reads, and verifies ISP devices using the parallel port of a PC, but will support ATE programming in a future release. A more thorough description of the ISP Serial Programmer software is included in the "In-System Programming on a PC or Sun Workstation" section in this manual.

ISP Daisy Chain Download

ISP Daisy Chain Download software supports programming of all Lattice ISP devices in a serial daisy chain programming configuration in a PC environment. Two varieties of this software exist: one for a Windows environment (called ISP Daisy Chain Download for Windows), and the other for a DOS environment (called ISP Daisy Chain Download for DOS). This software is available from Lattice. A more thorough description of the ISP Daisy Chain Download software is included in the "In-System Programming on a PC or Sun Workstation" section in this manual.

ISP Download for the Sun

ISP Download for the Sun supports programming of Lattice's ispLSI devices in a single device programming configuration on an isp Engineering Kit Model 200 Programmer. A more thorough description of ISP Download for the Sun software is included in the "In-System Programming on a PC or Sun Workstation" section in this manual.

ATE Programming Approaches

As ISP programming is controlled by TTL logic signals, one of the biggest advantages of the ISP feature is being able to program the devices during board-level testing on Automatic Test Equipment (ATE). By performing the PLD device programming and board-level testing in one step, the manufacturing flow can be streamlined. As a result of streamlining, costs are reduced and reliability is improved. To support ATE programming, Lattice provides several software utilities to help convert the ISP programming signals to various ATE platforms.

There are two types of file formats available to support ATE programming. One is the test vector format, in which the programming sequence is converted into the test vector streams driven from the ATE. ispCODE provides the test vector conversion utilities for HP, Teradyne, and GenRad test vector formats. Refer to the "ATE Programming of ISP Devices" section of this manual for more details. An alternative to test vectors is the tester's high-level language syntax. For testers which support high-level language, ISP programming routines can be developed similarly to the ispCODE routines. An example routine for the GenRad tester is provided in the "ATE Programming of ISP Devices" section of this manual.

In order to evaluate the feasibility of ATE programming, the tester memory requirement, the programming implementation (test vector vs. high-level language), and the programming times must be considered. The following tips should be considered when determining how to implement ISP programming on ATE.

1. Approximately 200K of memory depth is required to sequence the ispLSI 1048 programming algorithm in a vector sequence. High-level language routines may be an alternative to consider.
2. Programming times for each one of the ISP devices are provided in the "Hardware Basics" section of this manual. Consider the largest device when estimating the minimum ISP programming time.
3. In order to save tester time, multiple ISP devices should be programmed in parallel. Parallel programming is easily controlled on the tester by controlling programming signals for each device with independent signal drivers. Refer to the "ATE Programming of ISP Devices" section of this manual.

Notes

User In-System Programming Options

Guide to This Section

This section covers the full range of ISP programming options using third-party programmers, IBM compatible PCs, Sun Workstations, embedded processors, and ATE (Automatic Test Equipment). Your options for in-system programming of Lattice ISP devices will vary according to which environment you are in. To provide a quick reference guide, Table 1 highlights the sections of Section 3 that are most relevant for a given area of interest. For a complete understanding of all of the programming options available for programming ISP devices, please read all of Section 3.

Table 1. Section Reference

If your interest is...	**...please refer to this section.**	**Page**
Design and Development	• In-System Programming on a PC or Sun Workstation	3-3
	• In-System Programming from an Embedded Processor	3-11
Manufacturing	• ATE Programming of ISP Devices	3-25
	• Third-Party Programmers	3-47
Field Upgrades	• In-System Programming on a PC or Sun Workstation	3-3
	• In-System Programming from an Embedded Processor	3-11

Notes

In-System Programming on a PC or Sun Workstation

ISP Programming

This section discusses the many ways that you can program Lattice ISP devices using either an IBM® compatible PC or Sun Workstation®. This section is intended to give you a basic understanding of the range of programming tools that Lattice offers to make using Lattice ISP devices easier.

Selecting a Lattice Programming Tool

Lattice provides a wide variety of programming tools to meet your ISP programming needs. To select the appropriate Lattice programming tool, use the guidelines listed in Table 1.

Table 1. Selecting the Appropriate Programming Tool

Development Environment	Appropriate Tool
IBM-PC/Windows	ISP Daisy Chain Download for Windows
IBM-PC/DOS	ISP Daisy Chain Download for DOS
Sun Workstation	ISP Download for the Sun
Custom	ispCODE

Programming and Verifying

This section presents an example of ISP programming and verification using each of the Lattice ISP programming tools. It is not intended as a complete reference for using the Lattice programming tools. Instead, it provides a basic overview of each of the Lattice ISP programming products.

Examples are provided for the Lattice products listed below. The complete reference manual for each of these products is listed in parentheses under the product name. Please refer to these documents for detailed information about using these products.

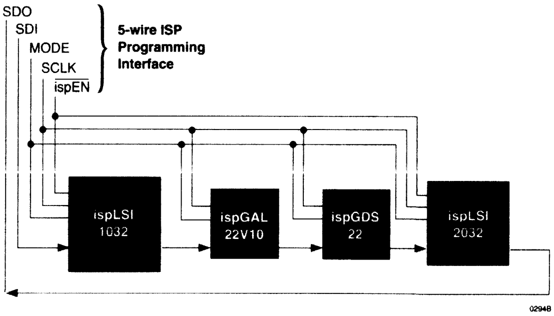
- ISP Daisy Chain Download for Windows - (ISP Daisy Chain Download Reference Manual v. 1.00)
- ISP Daisy Chain Download for DOS - (ISP Daisy Chain Download Reference Manual v. 1.00)
- ISP Download for the Sun - (pDS+ Software Documentation v. 2.00)
- ispCODE - (ispCODE Programmer's Reference Manual 1994)

ISP Daisy Chain Download for Windows

ISP Daisy Chain Download for Windows allows you to program one or more ISP devices connected in a daisy chain using an IBM PC. ISP Daisy Chain Download for Windows requires the following:

- A JEDEC file for each device you want to program
- ispDOWNLOAD cable to attach to the parallel port of an IBM PC
- Microsoft Windows® 3.1
- Target hardware (circuit board) with ISP interface (see Figure 1) or the Lattice isp Engineering Kit Model 100

Figure 1. Five-Wire ISP Interface



Download Process

Follow the steps below to download to your ISP daisy chain.

1. Invoke ISP Daisy Chain Download for Windows.
2. Generate a new configuration file.
3. Verify the configuration file.
4. Program the chain.

These steps are explained in more detail next.

Note: The configuration file, *design*.DLD, can be used in the DOS or Windows environment.

Invoking ISP Daisy Chain Download for Windows

To invoke ISP Daisy Chain Download for Windows, double click the ISP Daisy Chain Download for Windows icon as shown in Figure 2. The main window appears (Figure 3).

Figure 2. ISP Daisy Chain Download for Windows

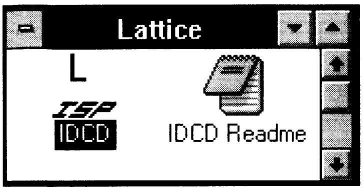
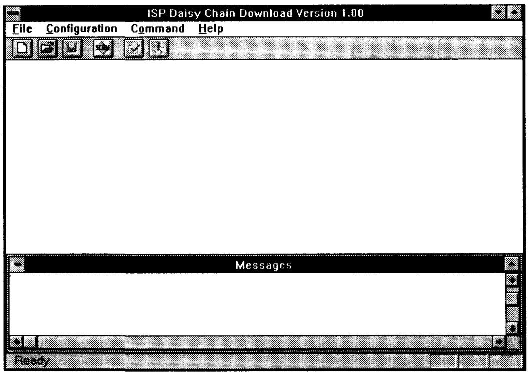


Figure 3. The ISP Daisy Chain Download for Windows Main Window


ISP Daisy Chain Download Version 1.00
File Configuration Command Help
Messages

Ready

Generating a Configuration File

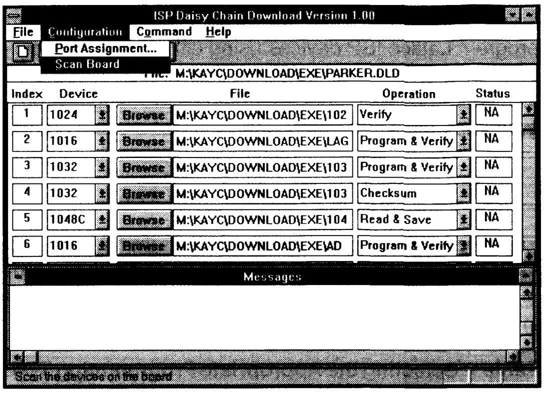
ISP Daisy Chain Download for Windows uses a configuration file to define the following information about your chain:

- The position and type of each device
- What operation to perform (read, program, verify, etc.) on each device

If the PC is connected to the target hardware or ISP Engineering Kit Model 100, the easiest approach to creating a configuration file is to use the **Configuration ➠ Scan Board** command. This creates a basic configuration file which contains all the devices in the chain, but no information about what operation to perform or what JEDEC files to use.

To create the configuration file, select the **Configuration➠ Scan Board** Option (Figure 4).

Figure 4. Selecting Configuration➠ Scan Board


ISP Daisy Chain Download Version 1.00
File Configuration Command Help
Port Assignment...
Scan Board
Messages

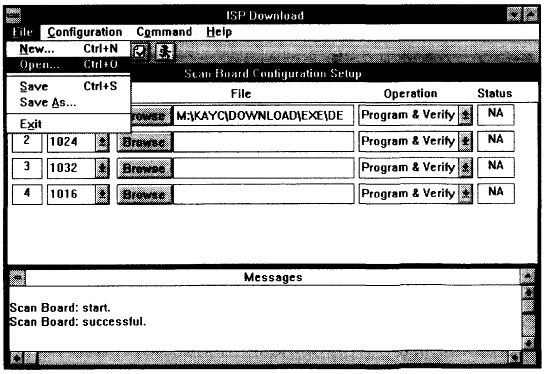
File: M:\KAYC\DOWNLOAD\EXE\PARKER.DLD

Index	Device		File	Operation	Status
1	1024	Browse	M:\KAYC\DOWNLOAD\EXE\102	Verify	NA
2	1016	Browse	M:\KAYC\DOWNLOAD\EXE\LAG	Program & Verify	NA
3	1032	Browse	M:\KAYC\DOWNLOAD\EXE\103	Program & Verify	NA
4	1032	Browse	M:\KAYC\DOWNLOAD\EXE\103	Checksum	NA
5	1048C	Browse	M:\KAYC\DOWNLOAD\EXE\104	Read & Save	NA
6	1016	Browse	M:\KAYC\DOWNLOAD\EXE\AD	Program & Verify	NA

Scan the devices on the board

The next step is to select a JEDEC file for each device in the chain that you want to program. You can do this by either entering the file name directly or using the browse button (Figure 5).

Figure 5. Selecting a JEDEC File for Each Device



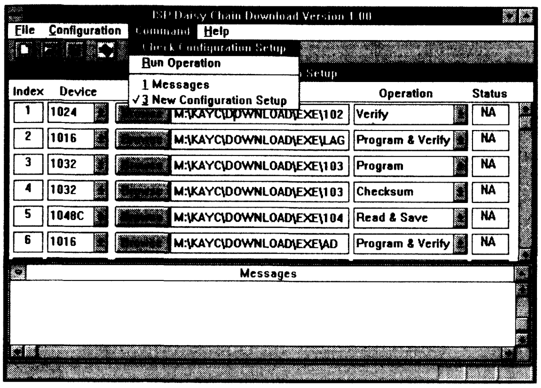
			File	Operation	Status
		rowse	M:\KAYC\DOWNLOAD\EXE\DE	Program & Verify	NA
2	1024	Browse		Program & Verify	NA
3	1032	Browse		Program & Verify	NA
4	1016	Browse		Program & Verify	NA

```
Scan Board: start.
Scan Board: successful.
```

Verifying the Configuration File

Once you have created a configuration file, verify that the configuration file is valid by using the **Command → Check Configuration Set** option (Figure 6).

Figure 6. Verifying the Configuration File


ISP Daisy Chain Download Version 1.00
File
Configuration
Command
Help
Check Configuration Setup
Run Operation
1 Messages
✓3 New Configuration Setup
Setup
Messages

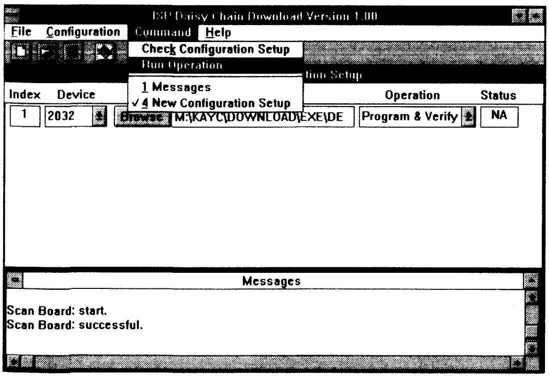
Index	Device		Operation	Status
1	1024	M:\KAYC\DOWNLOAD\EXE\102	Verify	NA
2	1016	M:\KAYC\DOWNLOAD\EXE\LAG	Program & Verify	NA
3	1032	M:\KAYC\DOWNLOAD\EXE\103	Program	NA
4	1032	M:\KAYC\DOWNLOAD\EXE\103	Checksum	NA
5	1048C	M:\KAYC\DOWNLOAD\EXE\104	Read & Save	NA
6	1016	M:\KAYC\DOWNLOAD\EXE\AD	Program & Verify	NA

If your configuration file passes this test, then you can proceed to programming the chain.

Programming or Verifying the Chain

Once you have a valid configuration file, you can perform operations on the chain. To do this, select the **Command → Run Operation** from the main menu (Figure 7).

Figure 7. Performing Operations on the Chain


ISP Daisy Chain Download Version 1.00
File
Configuration
Command
Help
Check Configuration Setup
Run Operation
1 Messages
✓4 New Configuration Setup
tion Setup
Browse
Messages

Index	Device		Operation	Status
1	2032	M:\KAYC\DOWNLOAD\EXE\DE	Program & Verify	NA

```
Scan Board: start.
Scan Board: successful.
```

ISP Daisy Chain Download for DOS

Lattice provides a DOS version of the download software for programming an ISP daisy chain. ISP Daisy Chain Download for DOS requires the following:

- A JEDEC file for each device you want to program
- ispDOWNLOAD cable to attach to the parallel port of a PC
- Target hardware (circuit board) with an ISP interface (see Figure 1) or the Lattice isp Engineering Kit Model 100

Download Process

The following is a step by step guide to downloading to an ISP daisy chain using ISP Daisy Chain Download for DOS.

1. Invoke ISP Daisy Chain Download for DOS.
2. Create a download configuration file.
3. Verify the download.
4. Program or Verify the chain.

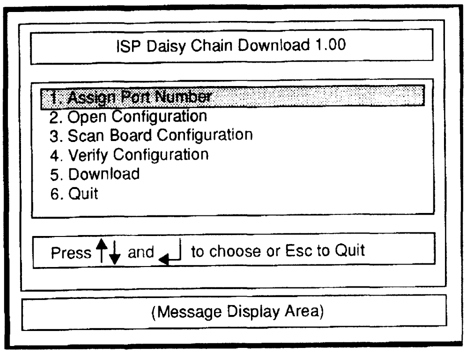
These steps are explained in more detail in the following sections.

Invoking ISP Daisy Chain Download for DOS

From the DOS command line, enter *ddownld.* The ISP Daisy Chain Download for DOS main window appears (Figure 8).

Note: It is preferable to invoke ISP Daisy Chain Download for DOS from the DOS prompt and not from Windows DOS. Invoking the program from Windows DOS can cause timing variations.

Figure 8. ISP Daisy Chain Download for DOS Main Window


ISP Daisy Chain Download 1.00

Press ↑↓ and ↵ to choose or Esc to Quit

(Message Display Area)

Creating a Configuration File

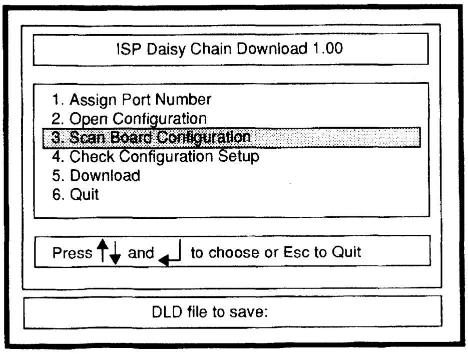
ISP Daisy Chain Download for DOS uses a configuration file to define the following information about your chain:

- The position and type of each device
- What operation to perform (read, program, verify, etc.) on each device

If you have the PC connected to your target hardware, the easiest approach to creating a configuration file is to use the ISP Daisy Chain Download for DOS **Scan Board Configuration** command. This creates a basic configuration file which contains all the devices in the chain, but no information on what operation to perform or what JEDEC files to use.

To create the configuration file, select the **Scan Board Configuration** command (Figure 9).

Figure 9. The Scan Board Configuration Command

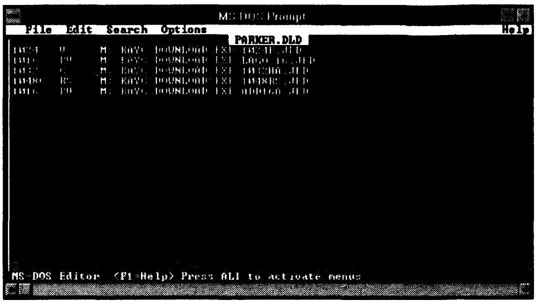

ISP Daisy Chain Download 1.00

Press ↑↓ and ↵ to choose or Esc to Quit

DLD file to save:

This creates a basic configuration file with the name you specify. Now use a text editor to enter the operations and JEDEC files needed by each device (Figure 10). Table 2 lists the operation codes that are entered in the second column of the configuration file.

Figure 10. Sample .DLD File



```
1024    V     M:\KAYC\DOWNLOAD\EXE\1024F.JED
1016    PV    M:\KAYC\DOWNLOAD\EXE\LAGO16.JED
1032    C     M:\KAYC\DOWNLOAD\EXE\1032HA.JED
1048C   RS    M:\KAYC\DOWNLOAD\EXE\1048RS.JED
1016    PV    M:\KAYC\DOWNLOAD\EXE\ADD16A.JED
```

MS-DOS Editor <F1=Help> Press ALT to activate menus

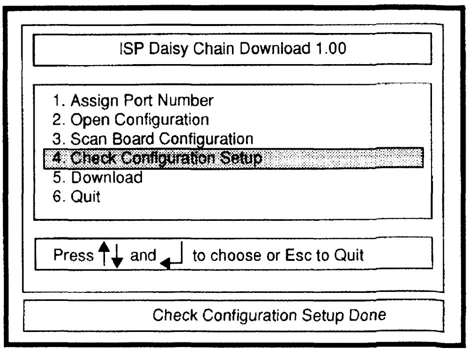
Table 2. Configuration File Operation Codes

Code	Operation
pv	program & verify
v	verify
c	checksum
rs	read & save
e	erase
nop	no operation

Verifying the Configuration File

As creating the configuration file is a manual process, it is important that you verify the file before proceeding. To do this, select the **Check Configuration Setup** command from the ISP Daisy Chain Download for DOS menu (Figure 11). This ensures that your configuration file is valid.

Figure 11. Verifying the Configuration File


ISP Daisy Chain Download 1.00

Press ↑↓ and ↵ to choose or Esc to Quit

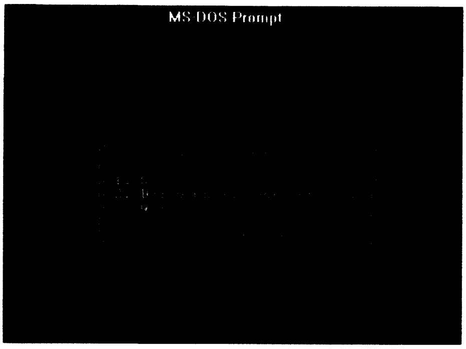
Check Configuration Setup Done

Programming or Verifying the Chain

Once the configuration file has been verified, you can download to the chain. Select the **Download** option from the ISP Daisy Chain Download for DOS main menu. When the download option menu is displayed, choose the **Daisy Chain Configuration** option which programs one or more devices in the daisy chain (Figure 12).

Note: If you are programming a single device and do not wish to create a configuration file, choose option 1 in Figure 12, **Single Device**.

Figure 12. Downloading the Chain



Once you have selected the Daisy Chain configuration option, the commands you specified in the configuration file will be performed. For example, if the configuration file calls for Program & Verify, the Program and Verify functions will be performed.

ISP Download for the Sun

You can download and program an ISP device from the Sun Workstation using the Lattice ISP Download for the Sun utility. This utility requires the following:

- A JEDEC file for each device you want to program
- isp Engineering Kit Model 200

This configuration supports in-system programming of a single ISP device either in the isp Engineering Kit socket adapter or directly on the circuit board via a download cable.

Download Process

To perform a download using ISP Download for the Sun, follow the steps below:

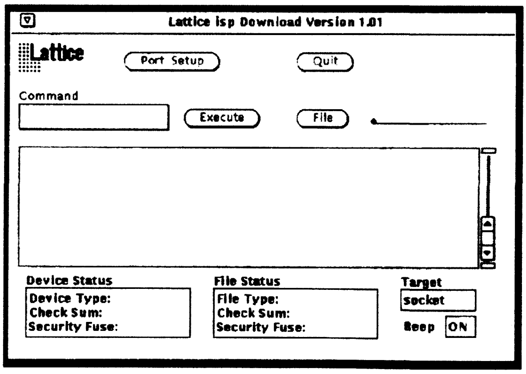
1. Connect the Model 200 to your Sun Workstation.
2. Invoke the ISP Download utility.
3. Select your target hardware.
3. Choose a JEDEC file to download.
4. Program the Device.

These steps are discussed in more detail in the following sections.

Invoking ISP Download for the Sun

To invoke ISP Download for the Sun, type *DOWNLOAD* at the UNIX prompt (or, choose **Lattice ➠ Download** from the Lattice menu in pDS+ Viewlogic). The main window appears (Figure 13).

Figure 13. The ISP Download for the Sun Main Window


Lattice isp Download Version 1.01

Device Status
Device Type:
Check Sum:
Security Fuse:

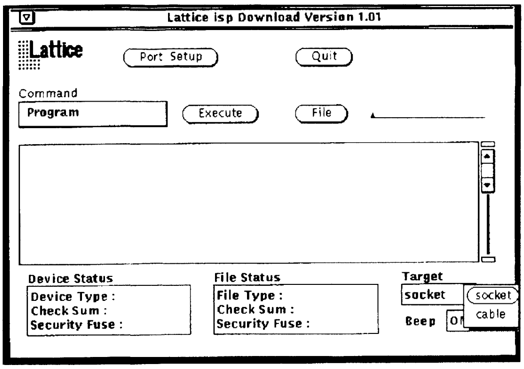
File Status
File Type:
Check Sum:
Security Fuse:

Target socket
Beep ON

Selecting the Target Hardware

Select your target hardware by clicking on the Target window in the lower right hand corner of the ISP Download for the Sun main window (Figure 14). Choose **socket** if you are using the isp Engineering Kit Model 200, or **cable** if your are downloading directly to the board using the download cable. If you are downloading via the cable, connect the cable from the Sun, through the isp Engineering Kit, to the circuit board.

Figure 14. Selecting the Target Hardware



Command: Program

Device Type :
Check Sum :
Security Fuse :

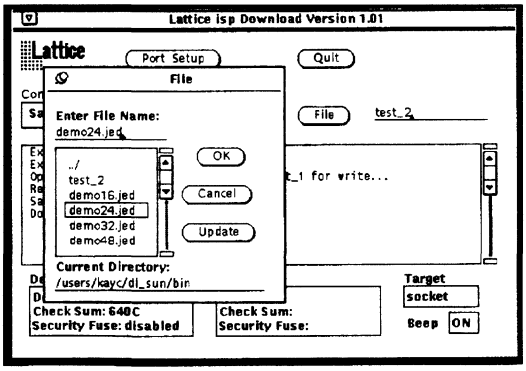
File Type :
Check Sum :
Security Fuse :

Target: socket

Choosing a JEDEC file

Once you have configured the ISP Download for the Sun program, choose a JEDEC file for programming the device. Select the **File** button from the main window. The File Selection dialog appears (Figure 15).

Figure 15. File Selection Dialog



Enter File Name:
demo24.jed

../
test_2
demo16.jed
demo24.jed
demo32.jed
demo48.jed

Current Directory:
/users/kayc/dl_sun/bin

test_2

t_1 for write...

Check Sum: 640C
Security Fuse: disabled

Check Sum:
Security Fuse:

Target: socket

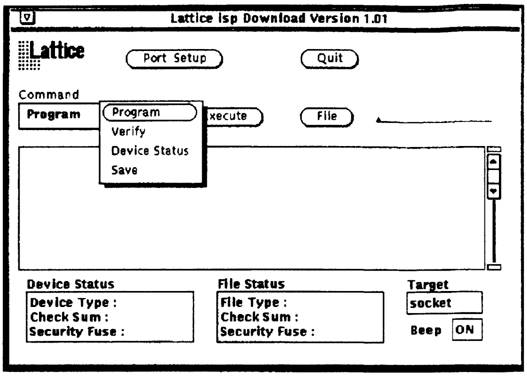
Beep: ON

Choose a JEDEC file from this dialog box and click the **OK** button.

Programming the Device

Once the JEDEC file has been chosen, you can choose to program or verify the ISP device. To select the program or verify command, place the cursor in the Command box and click the left mouse button. The Command Selection menu appears (Figure 16).

Figure 16. The Command Selection Menu



Command: Program

Device Type :
Check Sum :
Security Fuse :

File Type :
Check Sum :
Security Fuse :

Target: socket

Beep: ON

To program the device, select the **Program** option. To verify the device, select the **Verify** option. After you have chosen an option, click the **Execute** button.

ispCODE

Lattice ispCODE contains C++ routines for interfacing with Lattice ISP devices. These routines can be customized to meet your specific hardware needs. Included with this source code are a project file (SERIAL.PRJ) and all of the files required by the Borland C++ compiler to compile a version of the ispCODE which has a user interface. This user interface is intended to demonstrate how ispCODE can be used in your custom application. When you compile the ispCODE using the project file SERIAL.PRJ, the executable (SERIAL.EXE) is created which you can invoke while running windows. This ispCODE demonstration will be referred to as the ISP Serial Programmer throughout this section.

To perform a download or verification using the ispCODE ISP Serial Programmer, follow the steps below.

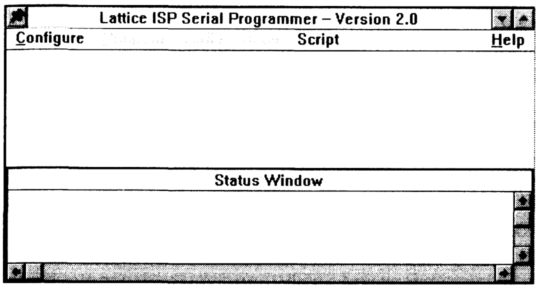
1. Connect a download cable, such as the Lattice ispDOWNLOAD cable, or isp Engineering Kit Model 100 to your PC parallel port.
2. Invoke the ISP Serial Programmer by double clicking the ISP Serial Programmer icon, or by running SERIAL.EXE.
3. Perform a scan of the attached chain.
4. Load a configuration file.
5. Program or verify the chain.

These options are discussed in more detail in the following sections.

Invoking the Serial Programmer

The ISP Serial Programmer is invoked by either double clicking the ISP Serial Programmer icon, or by executing SERIAL.EXE from the Windows Program Manager. Once invoked, the ISP Serial Programmer displays the main menu (Figure 17).

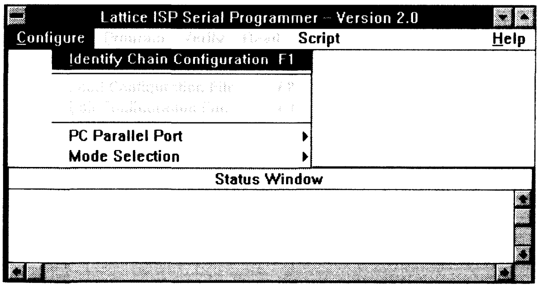
Figure 17. The ISP Serial Programmer Main Menu



Scanning the Chain

First, perform a scan of the attached chain by selecting **Configure ➡ Identify Chain Configuration** from the main window, or by pressing F1. This identifies the devices attached to the PC's parallel port. The device names appear in the status window (Figure 18).

Figure 18. Scanning the Attached Chain

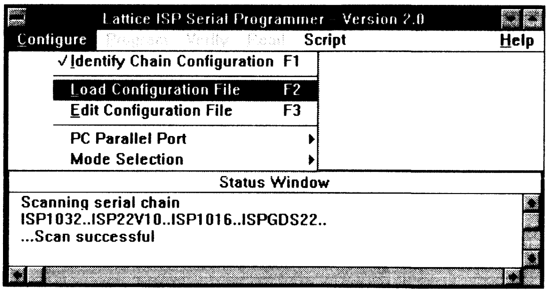

Lattice ISP Serial Programmer – Version 2.0
Configure
Script
Help
Identify Chain Configuration F1
PC Parallel Port
Mode Selection
Status Window

Creating and Loading a Configuration File

The ISP Serial Programmer uses a configuration file to define what devices are in the chain, and what files to load into each device. This is an ASCII text file that you can create with any text editor. Whenever you scan the chain, an empty configuration file called "template.cfg" is created. You can then edit this file (by selecting **Configure ➡ Edit Configuration File** or pressing F3), and add the JEDEC file names next to the devices you want to program. Save this file under a new name, so it is not overwritten the next time you scan the chain.

You must load a configuration file before any other operation is performed. Load the configuration file by selecting **Configure ➡ Load Configuration File**, or by pressing F2. The software verifies the configuration file, and enables the program, verify, and read options from the main menu (Figure 19).

Figure 19. Loading the Configuration File

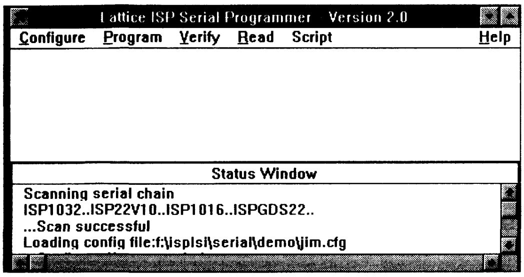

Lattice ISP Serial Programmer – Version 2.0
Configure
Script
Help
√Identify Chain Configuration F1
Load Configuration File F2
Edit Configuration File F3
PC Parallel Port
Mode Selection
Status Window

```
Scanning serial chain
ISP1032..ISP22V10..ISP1016..ISPGDS22..
...Scan successful
```

Programming or Verifying the Chain

Once the configuration file has been successfully loaded, you can program or verify the serial chain. Note in the figure below that the Program, Verify, and Read options are now available, indicating the successful loading of the configuration file. To program the devices, select the **Program** option from the main window. To verify the devices, select **Verify** from the main window (Figure 20).

Figure 20. Verifying the Devices



```
Scanning serial chain
ISP1032..ISP22V10..ISP1016..ISPGDS22..
...Scan successful
Loading config file:f:\isplsi\serial\demo\jim.cfg
```

Customization

As Lattice makes all the source code available for the ISP Serial Programmer, you can modify the application to meet your specific needs. However, modifying the source code may not be necessary. The ISP Serial Programmer provides a **script** option which allows you to record a series of commands for the programmer to execute. One of these commands is an execute command, which allows you to run another Windows or DOS program as part of the script. This may be sufficient for your customization needs. Refer to the on-line help for more information.

Another reason for using ispCODE may be to add field upgrade capability to your application. If you are using ISP with an embedded system, you may want to be able to load new versions of ISP programming information into your system via a floppy or serial link, such as a modem. By using ispCODE in your application, you can update your system far more easily than if you had to remove and reprogram logic manually.

In-System Programming from an Embedded Processor

Overview

This section describes how to program Lattice ISP devices using an embedded processor. The first section shows the use of a microprocessor to control ISP, including the construction of a simple ISP port. The second section shows an 8051 microcontroller used as an ISP controller and covers the procedures and assembly code required for processor-based ISP programming. The 8051 assembly code is written in a modular format. A higher level of routines provides the user with device-level functions such as Read ID, Bulk Erase, Program, and Verify. In an attempt to provide routines that can be used across all ISP device families, specific routines are written for the ispGDS22 devices. Only slight modifications to the basic functions used across these devices are required to adapt the routines to program any other ISP device.

Programming with a Microprocessor

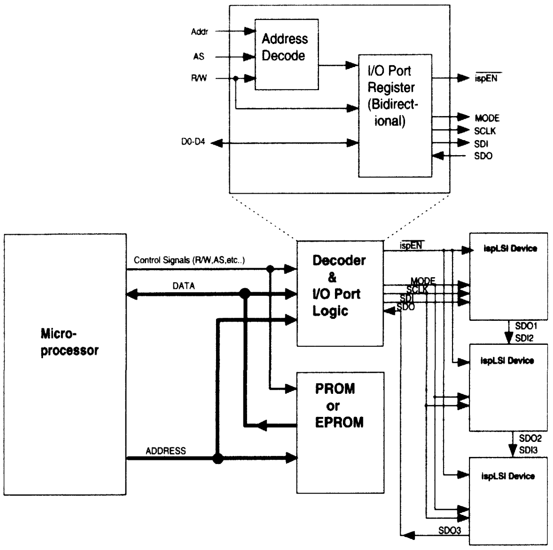
There are several ways to define the ISP programming hardware for microprocessor-based in-system programming, depending on the type of storage device used and how the ISP devices are to be programmed. Since an additional step is necessary to convert a JEDEC file into an ispSTREAM, the fuse map information can be stored in a JEDEC standard fuse map file, simplifying design changes. Or, since a JEDEC standard fuse map file takes an order of magnitude more memory space to store the fuse map information than an ispSTREAM, the ISP programming routines can be stored in an ispSTREAM. This section presents a hardware configuration in which the fuse map information is stored in a JEDEC standard fuse map file.

The hardware configuration shown in Figure 1 uses an 8-bit wide EPROM to store the JEDEC fuse map file and object code, which is created from ispCODE C++ source code. (See the "Software Basics" section in this manual for a complete description of ispCODE). The patterns are then read from the EPROM by the microprocessor and converted into serial stream format. The ISP signals are driven from the decoder and I/O port which decodes the proper ISP read/write address space (similar to the I/O port definition of the previous setup). Similarly, fuse map memory addresses must be defined to be properly read from the EPROM.

3

The I/O port can be implemented using a dedicated port chip or a PLD. In the case of a PLD implementation, the device must have five pins for the ISP port, five pins for the data port, one pin each for AS and R/W, and enough additional pins to provide address inputs to the decoder. If a partial rather than full decode can be used, the number of address pins can be reduced. For example, for

Figure 1. Microprocessor-Based ISP



a processor with a 16-bit I/O space, if a block of 256 locations can be allocated to the ISP port, only the upper eight addresses (A8-A15) need to be decoded and the function can fit into a 24-pin low-density GAL such as the GAL 20RA10. If a full decode in needed, a high-density PLD with ample pins such as Lattice's ispLSI or pLSI 2032 or 1016 should be used.

Most hardware timing requirements can be satisfied by the microprocessor software instruction execution time. Only the program, verify, and bulk erase times require the software to have wait cycles. Many microprocessor boards will not have a timer chip to time the wait states. However, the instruction execution times can typically be estimated accurately. Therefore, timing loops must be inserted into the instructions to control critical hardware timing.

Within ispCODE source code, before the object code is created, address spaces for the ISP read/write locations and the EPROM read locations must be defined. The storage space requirement for the object code must also be determined if the code is going to reside in the storage device. Based on the ispCODE functions, the object code which is capable of executing basic ISP functions typically does not exceed 8K bytes of memory. This memory requirement is directly proportional to the number of ISP and user interface functions.

Programming with a Microcontroller

The advantage of using a microcontroller-based ISP interface is that the ports are integrated with the processor. As shown in Figure 2, the interface to the ISP devices is accomplished through I/O port 1. RAM and EPROM access is also shown in the block diagram through the use of I/O port 0 and I/O port 2. These specific connections may be changed according to the user's application. Direct connections are made from the port to the ISP pins of the device. The pinout used on I/O port 1 is listed below:

SDI	P1.0
SCLK	P1.1
MODE	P1.2
SDO	P1.3

The address and data to the RAM and EPROM are multiplexed through a 74LS373 latch and the control signals are routed through the 74LS138 decoder.

The timing requirements for ISP programming are with respect to the SCLK signal. To shift between states, the SDI and MODE control signals are set to the required values for a state transition and SCLK is applied. In this manner, the set up times are easily met and interrupts can occur at any time during the application of ISP signals.

Software Overview

The main function of the embedded processor assembly software routines is to drive the ISP programming state machine (Figure 3), while ensuring that all programming timing requirements are met. The programming pulse width is controlled by a counter delay. The resulting pulse width of the counter depends on the clock frequency of the microcontroller and the bit width of the counter. The values used here are based on a clock rate of 12 MHz and the use of the default state of the counter which is 13 bits.

Figure 2. Microcontroller Block Diagram

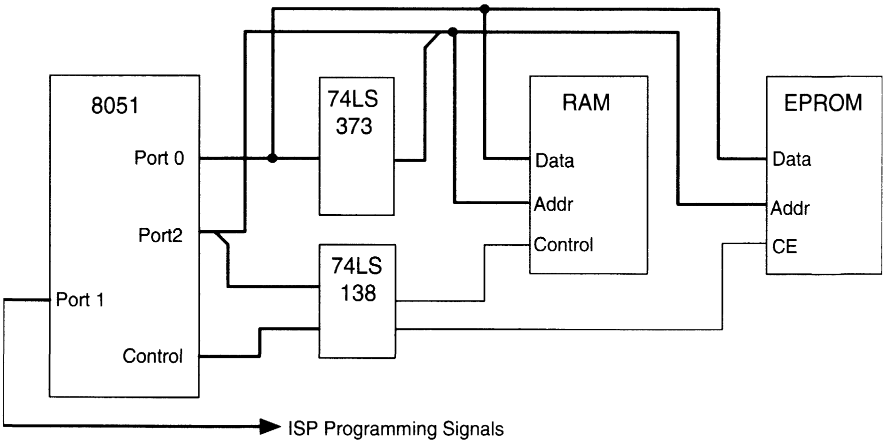
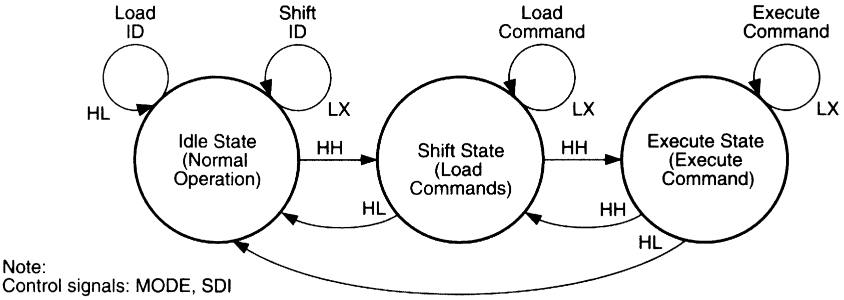


Figure 3. ISP Programming State Machine



Note:
Control signals: MODE, SDI

Software resources, such as internal registers, are self-contained within the routines. These resources are freed after the routines have been properly executed. The only resources that have to be dedicated to ISP in this example are the I/O port signals that are used to drive the ISP programming signals. The user has a choice to free the I/O port resources if the I/O port signals are multiplexed. The ORG statement indicates the beginning address of the subroutine and is used to place the assembly routines within the main code. If the routine codes are intended for a user application using the same internal register resources, the registers may be pushed onto a stack to preserve them. Upon completion of the subroutine, they are popped off the stack so that there is no interference with the calling assembly program.

JEDEC Map Creation and Storage

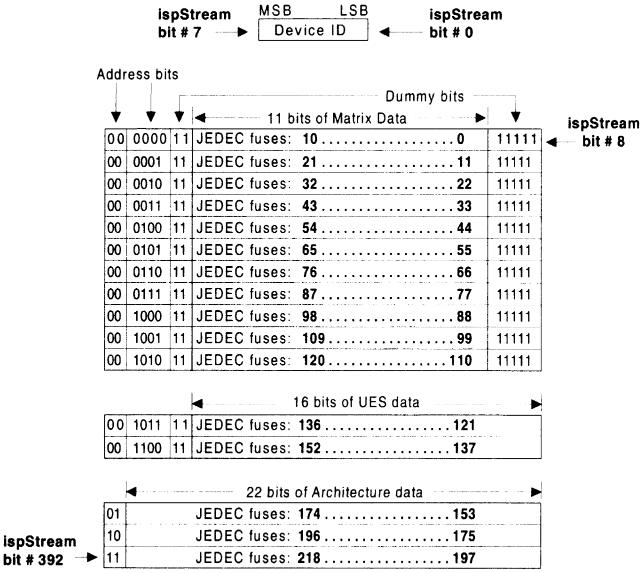
The JEDEC pattern is stored as part of the 8051 executable code. It is differentiated from the assembly listing by an assembler directive which flags the program space to be used as data bytes. The JEDEC map is placed at the bottom of the assembly file, as memory, and referenced with a label to mark its position. Assignments are automatically adjusted by the assembler. Placing the JEDEC map at the bottom also simplifies using sequential JEDEC files for multiple device programming. This is desirable for PC boards intended for multiple function programming during manufacturing or in the field.

The JEDEC Fuse Map Shift Procedures

In the source listing section of this document, the JEDEC fuse map for the ispGDS22 is listed as a sequential order of bytes. Each byte is broken into a bit sequence and written out serially to a port pin. Another port pin is then used as a clock driver for clocking in the serial data information. The JEDEC file is stored so that it is read out sequentially from top to bottom. In the case of ispGDS and ispGAL devices, each line of the JEDEC file contains both the data and address information and can be programmed as a single stand-alone line. However, in the case of ispLSI devices, the address shift routine must be executed as a separate routine prior to shifting in the data bits.

When shifting the JEDEC data from program memory, a shift left operation is used on each byte to shift out the data to the I/O port. This means that a reverse bit order is stored in the bytes. If the order of the bits is to be maintained, then a shift right instruction may be used to shift the bits out of each of the bytes. The stored ispGDS JEDEC file is in the same format as Figure 4. Bits in the JEDEC file are shifted LSB first. This means that the five "1s" will be shifted out, starting at the right hand side of the table. Note that this method of shifting out the bits requires reverse ordered bytes of information.

Figure 4. ispGDS JEDEC Fuse Map



00	0000	11	JEDEC fuses: 10..................0	11111
00	0001	11	JEDEC fuses: 21..................11	11111
00	0010	11	JEDEC fuses: 32..................22	11111
00	0011	11	JEDEC fuses: 43..................33	11111
00	0100	11	JEDEC fuses: 54..................44	11111
00	0101	11	JEDEC fuses: 65..................55	11111
00	0110	11	JEDEC fuses: 76..................66	11111
00	0111	11	JEDEC fuses: 87..................77	11111
00	1000	11	JEDEC fuses: 98..................88	11111
00	1001	11	JEDEC fuses: 109.................99	11111
00	1010	11	JEDEC fuses: 120................110	11111

00	1011	11	JEDEC fuses: 136................121
00	1100	11	JEDEC fuses: 152.................137

01	JEDEC fuses: 174.................153
10	JEDEC fuses: 196.................175
11	JEDEC fuses: 218.................197

ISP Programming Routines

The programming routines were constructed in a modular format through the use of subroutines. Each subroutine is constructed so that it controls the appropriate state transitions as shown in Figure 3. The routines can be easily traced keeping in mind the ISP states and state transitions. The names of the subroutines follow the names of the state machine closely for ease of readability and comprehension.

Programming Sequence

The code is broken into three major blocks. Each is executed in sequence. The first block is that of device identification. In this section, note that the identification of the device is hard coded. The next major block performs a bulk erase on the device. The last block loads the JEDEC map into the device.

Prior to these blocks is the configuration area of the 8051 microcontroller. This is standard configuration information that is generally present at the top of any 8051 microcontroller assembler file. Typical information contained in the configuration area includes port addresses and interrupt type of addresses.

The identification of the device is marked in the assembler code by the label "READID." The completion of this section is marked by the label "END_READID:."

The first step in reading the ID is to establish the starting state from which the action is to take place. Normally this is done by moving to the first state of the state machine, which is the idle state (Listing 1).

Listing 1. Embedded Processor Programming Algorithm

Follow the steps below to read the device identification:

- Move to the IDLESTATE.
- In the IDLESTATE execute the LOAD ID Command.
- Enable the shifting of the IDENTIFICATION of the device.
- Set up of the registers for a loop counter and for storing the device ID.
- Clock each bit of the device ID with an SCLK signal and do a shift store of the information.
- Compare the ID that was shifted in with the ID of an ispGDS22 and continue if the identification matches. Otherwise, enter into an endless loop with no other actions.

Follow the steps below to bulk erase the part prior to programming:

- From the IDLESTATE, the subroutine BULK_ERASE is called. It in turn performs the following actions:
  - It places the state machine into the IDLESTATE
  - Changes state to the SHIFTSTATE , getting the ISP state machine ready for shifting in a command
  - Shifts in the Bulk erase command
  - Changes state to the EXECUTE STATE
  - Calls EXECUTE COMMAND once and waits 200 milliseconds
  - Returns to the point from where the subroutine call was made (Note that at this point Bulk erasing of the part is recommended so the device will be programmed from a known starting point)

3

Follow the steps below to program the device:

- Place the device in the IDLESTATE
- Change state to the SHIFTSTATE
- Shift in the SHIFTDATA command
- Move to the EXECUTE_STATE
- Execute the command

The state machine is now ready for shifting in each line of the JEDEC file:

- Prepare the registers for the looping that is required for the shifting in of the bytes and bits within each byte of information
- Load each of the bytes in its turn in a shift register and shift out.

Once a complete line has been shifted out, call the subroutine PROGRAM which does the following:

- Calls the subroutine SHIFTSTATE
- Moves from the EXECUTE_STATE to the SHIFT STATE
- Calls the subroutine PROGRAM_CMD which shifts in the program command and returns to the point where it was called
- Moves to the EXECUTE_STATE
- EXECUTEs the command
- Delays the movement out of the execute above to allow for the programming of the line of the JEDEC file
- Shifts to the SHIFTSTATE state
- Calls the SHIFTSTATE subroutine which performs the functions as noted above
- Moves to the EXECUTE_STATE

```
- EXECUTES the command
- Returns to the point from where the PROGRAM subroutine was called
```

After completion of the PROGRAM subroutine call, go back to the top of the loop that loads in another row of the JEDEC file and repeat the procedure until all of the rows and bits have been loaded in. On completion of programming the device, continue with execution of the user code or go into a loop which tells the user that the programming of the device has been completed (such as the indefinite subroutine call to PROG_COMP which flashes an LED).

Listing 2 is the assembler listing that follows the algorithm in Listing 1.

Listing 2. Assembler Listing

```
;$MOD51
$TITLE(ISP PROGRAMMER FOR 8051)
$PAGEWIDTH(132)
$DEBUG
$OBJECT
$NOPAGING

;  Variable declarations
;
      P0              DATA      080H      ;PORT 0
      P1              DATA      090H      ;PORT 1
      P2              DATA      0A0H      ;PORT 2
      P3              DATA      0B0H      ;PORT 3

      SP              DATA      081H      ;STACK POINTER
      DPL             DATA      082H      ;DATA POINTER - LOW BYTE
      DPH             DATA      083H      ;DATA POINTER - HIGH BYTE
      PCON            DATA      087H      ;POWER CONTROL
      TCON            DATA      088H      ;TIMER CONTROL
      TMOD            DATA      089H      ;TIMER MODE
      TL0             DATA      08AH      ;TIMER 0 - LOW BYTE
      TL1             DATA      08BH      ;TIMER 1 - LOW BYTE
      TH0             DATA      08CH      ;TIMER 0 - HIGH BYTE
      TH1             DATA      08DH      ;TIMER 1 - HIGH BYTE

      IE              DATA      0A8H      ;INTERRUPT ENABLE
      IP              DATA      0B8H      ;INTERRUPT PRIORITY
      ACC             DATA      0E0H      ;ACCUMULATOR
      B               DATA      0F0H      ;MULTIPLICATION REGISTER

      IT0             BIT       088H      ;TCON.0 - EXT. INTERRUPT 0 TYPE
      IE0             BIT       089H      ;TCON.1 - EXT. INTERRUPT 0 EDGE FLAG
      IT1             BIT       08AH      ;TCON.2 - EXT. INTERRUPT 1 TYPE
      IE1             BIT       08BH      ;TCON.3 - EXT. INTERRUPT 1 EDGE FLAG
      TR0             BIT       08CH      ;TCON.4 - TIMER 0 ON/OFF CONTROL
      TF0             BIT       08DH      ;TCON.5 - TIMER 0 OVERFLOW FLAG
      TR1             BIT       08EH      ;TCON.6 - TIMER 1 ON/OFF CONTROL
      TF1             BIT       08FH      ;TCON.7 - TIMER 1 OVERFLOW FLAG
      RI              BIT       098H      ;SCON.0 - RECEIVE INTERRUPT FLAG
      TI              BIT       099H      ;SCON.1 - TRANSMIT INTERRUPT FLAG
      RB8             BIT       09AH      ;SCON.2 - RECEIVE BIT 8
      TB8            BIT       09BH      ;SCON.3 - TRANSMIT BIT 8
      REN            BIT       09CH      ;SCON.4 - RECEIVE ENABLE
      SM2            BIT       09DH      ;SCON.5 - SERIAL MODE CONTROL BIT 2
      SM1            BIT       09EH      ;SCON.6 - SERIAL MODE CONTROL BIT 1
      SM0            BIT       09FH      ;SCON.7 - SERIAL MODE CONTROL BIT 0

      EX0            BIT       0A8H      ;IE.0 - EXTERNAL INTERRUPT 0 ENABLE
      ET0            BIT       0A9H      ;IE.1 - TIMER 0 INTERRUPT ENABLE
      EX1            BIT       0AAH      ;IE.2 - EXTERNAL INTERRUPT 1 ENABLE
      ET1            BIT       0ABH      ;IE.3 - TIMER 1 INTERRUPT ENABLE
      ES             BIT       0ACH      ;IE.4 - SERIAL PORT INTERRUPT ENABLE

      SCON           DATA      098H      ;SERIAL PORT CONTROL
      SBUF           DATA      099H      ;SERIAL PORT BUFFER

      EA             BIT       0AFH      ;IE.7 - GLOBAL INTERRUPT ENABLE
      RXD            BIT       0B0H      ;P3.0 - SERIAL PORT RECEIVE INPUT
      TXD            BIT       0B1H      ;P3.1 - SERIAL PORT TRANSMIT OUTPUT
      INT0           BIT       0B2H      ;P3.2 - EXTERNAL INTERRUPT 0 INPUT
      INT1           BIT       0B3H      ;P3.3 - EXTERNAL INTERRUPT 1 INPUT
      T0             BIT       0B4H      ;P3.4 - TIMER 0 COUNT INPUT
      T1             BIT       0B5H      ;P3.5 - TIMER 1 COUNT INPUT
      WR             BIT       0B6H      ;P3.6 - WRITE CONTROL FOR EXT. MEMORY
      RD             BIT       0B7H      ;P3.7 - READ CONTROL FOR EXT. MEMORY
      PX0            BIT       0B8H      ;IP.0 - EXTERNAL INTERRUPT 0 PRIORITY
      PT0            BIT       0B9H      ;IP.1 - TIMER 0 PRIORITY
      PX1            BIT       0BAH      ;IP.2 - EXTERNAL INTERRUPT 1 PRIORITY
      PT1            BIT       0BBH      ;IP.3 - TIMER 1 PRIORITY
      PS             BIT       0BCH      ;IP.4 - SERIAL PORT PRIORITY

      RS1            BIT       0D4H      ;PSW.4 - REGISTER BANK SELECT 1
      F0             BIT       0D5H      ;PSW.5 - FLAG 0
      AC             BIT       0D6H      ;PSW.6 - AUXILIARY CARRY FLAG
      CY             BIT       0D7H      ;PSW.7 - CARRY FLAG
      SDI            EQU       P1.0
      SCLK           EQU       P1.1
      MODE           EQU       P1.2
      SDO            EQU       P1.3
      ESC            EQU       1BH        ;escape character


;NOTE THAT THE JEDEC_TABLE IS LOCATED AT THE BOTTOM OF THE FILE.


;-----END OF VARIABLES----------------------

            ORG 2100H       ;START OF PROGRAM
            AJMP BEGIN
            ORG 2103H       ;EXTERNAL INTERUPT
            SETB B.0        ;SET FLAG TO PROGRAM
            RETI
```

```
;============== INTERRUPT SERVICE ROUTINE FOR TIMER0 INTERRUPT ==========

            ORG 210BH
            SETB TFO          ; TIMER HAS OVERFLOW
            RETI

BEGIN:          SETB EA       ;ENABLE ALL INTERRUPTS, GLOBAL
            SETB EX0          ;ENABLE INTERRUPT 0
            MOV B,00H
            AJMP BLINK        ;OUT OF RESET JUMP TO
                              ;THE TABLE START


BLINK:          SETB P1.7
            JB B.0,START
            ACALL DELAY200
            ACALL DELAY200
            CLR P1.7
            ACALL DELAY200
            AJMP BLINK


START:

;############### READ ID OF THE DEVICE #########################

READID:     ACALL   IDLESTATE       ;THESE 2 FUNCTION CALLS
            ACALL   LOAD_ID         ;ACCOMPLISH THE SAME THING
            ACALL   SHIFT_EN        ;CLEARING THE MODE BIT SO THAT
                                    ;SHIFTING CAN TAKE PLACE

            MOV     R0,#07H         ; LOOP COUNTER SET UP FOR 7 COUNT
            MOV     R1,#00H         ; TEMP REG FOR ID BYTE
            SETB    P1.5            ; LED INDICATOR FOR ID OFF

LABEL:      MOV     A,R1
            JB      P1.3,HI_BIT     ; JUMP IF ITS A HI ON SDIN (P1.3)
            CLR     ACC.7           ; IF NOT ITS A LOW, PUT A LOW AT MSB
            AJMP    OVER1
HI_BIT:     SETB    ACC.7           ; IT IS A HIGH, PUT A HIGH AT MSB
OVER1:
            RR      A               ; SHIFT RIGHT
            MOV     R1,ACC          ; MOVE IT TO TEMP REG R1
            ACALL   SCLOCK          ; CLOCK AND GET READY FOR NEXT BIT
            DJNZ    R0,LABEL        ; GET NEXT VALUE, DO THIS 7 TIMES

            CJNE    A,#072H,NO_ID   ; 72H IS THE DEVICE ID FOR ispGDS22
            CLR     P1.5            ; SET P1.5 IF THE CORRECT ID
            AJMP    CONT1
NO_ID:      SETB    P1.5            ; LED LIGHTS ON THE BOARD
            AJMP    NO_ID           ; SO THAT SIGNAL INTEGRITY CHECKS
                                    ; CAN BE MADE.

CONT1:          NOP
```

```
; END_READ ID
; TO THIS POINT THE DEVICE ID HAS BEEN READ
; AND GENERAL BULK ERASE AND PROGRAMMING IS
; TO TAKE PLACE


          ACALL   BULK_ERASE

; AT THIS POINT HAVE READ ID AND BULK ERASED THE PART


; SET UP FOR JEDEC PROGRAMMING SEQUENCE OF THE DEVICE
          ACALL   IDLESTATE

          ACALL   SHIFTSTATE        ;MOVED FROM EXECUTE STATE TO
                                    ;LOAD COMMAND STATE


          ACALL   SHFTDATA_CMD      ; SETUP FOR PLACING DATA
                                    ; INTO THE SHIFT COMMAND

          ACALL   EXECUTE_STATE
          ACALL   EXECUTE           ; EXECUTE SHIFT CMD


;NOW DEVICE IS READY FOR THE LOADING OF THE JEDEC PATTERN
;LOADING OF THE JEDEC PATTERN IS DONE IN THE EXECUTE STATE AFTER THE
;SHIFTDATA COMMAND.

;LOOP COUNTERS ARE SET UP SO THAT LOADING OF THE JEDEC FILE CAN TAKE PLACE.

;=============== PROGRAM THE DEVICE ======================

;THE NEXT 4 LINES INITIALIZE COUNTERS AND DATA POINTER

;PROGRAMMING OF THE DEVICE IS TO TAKE PLACE IN THE
;EXECUTE STATE OF THE DEVICE.

          MOV     R3,#16D           ; ROW COUNTER FOR ispGDS
          MOV     R2,#03D           ; COLUMN COUNTER 3 BYTES/ROW
          MOV     R1,#08D           ; SHIFT COUNTER WITH THE BYTE

          MOV    DPTR,#JEDEC_TABLE ; ADDR OF JEDEC FILE

;===================================================================
ROWS:
LOADBYTE: CLR     A                 ; CLEAR ACC
          MOVC    A,@A+DPTR         ;LOAD FIRST BYTE OF JEDEC IN A

;MOVE CODE BYTE RELATIVE TO DPTR TO Acc
;METHOD OF BRINGING IN STORED DATA IN THE CODE SEGMENT OF MEMORY.
```

```
LOOP8:      JB  ACC.7,ITS_1          ;JUMP IF MSB IS 1

ITS_0:      CLR SDI                  ;NO! MSB WAS 0, SO LOAD A 0
            CLR MODE
            ACALL SCLOCK
            JMP OVER                 ;JUMP OVER TEST FOR BIT=1, IT WAS 0

ITS_1:      SETB SDI                 ;MSB IS 1, SO LOAD A 1
            CLR MODE
            ACALL SCLOCK
OVER:

            RL A                     ; GET NEXT BIT IN ACC INTO MSB
                                     ; POSITION, ROTATE LEFT

            DJNZ R1,LOOP8
            INC DPTR                 ; MOVE DATA POINTER TO THE NEXT BYTE
            MOV R1,#08D              ; RESET BIT POSITION COUNTER FOR ACC


            DJNZ R2,LOADBYTE
            MOV R2,#03D              ; RESET BYTE COUNTER 3 BYTES PER ROW



            ACALL PROGRAM            ;PROGRAM A SINGLE ROW, PROGRAM A ROW
                                     ;AND RETURN

            DJNZ R3,ROWS             ;READY TO LOAD NEXT ROW WITH NEW
                                     ;DATA



            ACALL IDLESTATE          ;PLACE THE DEVICE IN A KNOWN STATE
                                     ;READY FOR NORMAL OPERATION.
PROG_COMP:  SETB P1.5
            ACALL DELAY200           ;ON COMPLETION OF THE JEDEC PROGRAMMING
            ACALL DELAY200           ;THE DEVICE IS PLACED INTO AN ENDLESS
            CLR P1.5                 ;LOOP OF FLASHING AN LED
            ACALL DELAY200
            AJMP PROG_COMP

;##############################################################
;                 SUBROUTINES
;##############################################################

;============== BULK ERASE THE DEVICE =====================
BULK_ERASE: CALL    IDLESTATE          ; GO TO IDLESTATE
            ACALL   SHIFTSTATE         ; GO TO SHIFT STATE
            ACALL   BERASE_CMD       ; LOAD BULK ERASE CMD
            ACALL   EXECUTE_STATE    ; GOTO EXECUTE STATE
            ACALL   EXECUTE          ; EXECUTE BULK ERASE CMD
            ACALL   DELAY200         ; CALL DELAY ROUTINE
            RET
```

```
;-------ID SHIFT ENABLE-LX--------------
SHIFT_EN:   CLR MODE          ;JUST SET THE REQUIRED BITS
            RET               ;WITHOUT PULSING THE CLOCK

;-------SUBROUTINE FOR THE PROGRAMMING OF THE SHIFTED ROWS

PROGRAM:    ACALL   SHIFTSTATE      ; GOTO SHIFT STATE (MOVE FROM THE
                                    ; EXECUTE STATE TO THE SHIFT STATE)
            ACALL   EXECUTE
            ACALL   PROGRAM_CMD     ; LOAD PROGRAM COMMAND
            ACALL   EXECUTE_STATE   ; GOTO EXECUTE STATE
            ACALL   EXECUTE         ; EXECUTE PROGRAM CMD
            ACALL   DELAY50
            ACALL   DELAY50

            ACALL   SHIFTSTATE      ; PROGRAMMING OF ONE ROW IS DONE
            ACALL   EXECUTE         ; GET READY FOR NEXT ROW

;THE RETURN FROM THIS COMMAND IS IN THE SHIFT STATE (LOAD COMMANDS ;STATE)

             ACALL   SHFTDATA_CMD    ;GET READY TO SHIFT IN NEXT ROW
             ACALL   EXECUTE_STATE
             ACALL   EXECUTE
             RET

;-------IDLESTATE --HL-------------

IDLESTATE:        SETB MODE         ;ALSO LOADS THE DEVICE ID
            CLR SDI                 ;SO THAT IT CAN BE SHIFTED OUT
            ACALL SCLOCK
            RET
;-------SHIFTSTATE ---HH---------------

; GENERIC SUBROUTINE FOR THE CHANGING OF STATE
SHIFTSTATE:     SETB MODE
            SETB SDI
            ACALL SCLOCK
            RET

;-------EXECUTESTATE ---HH-------------

EXECUTE_STATE:    SETB SDI
            SETB MODE
            ACALL SCLOCK
            RET
;-------LOAD ID STATE---HL-------------

LOAD_ID:        SETB MODE
            CLR SDI
            ACALL SCLOCK
            RET
```

```
;———————TOGGLE SCLK——LHL———————————————————
SCLOCK:           CLR SCLK
            SETB SCLK
            CLR SCLK
            RET

;——————DELAY 50MS————————————————————————
;FOR 12 MHZ     13 BIT UP COUNTER OVERFLOW GENERATES AN INTERUPT
;   65535 - (50MS/12US)=61368——> EFB8  HEX

DELAY50:        MOV TH0, #0FH
            MOV TL0, #0BAH
            SETB TR0                ;START TIMER0 SET TCON.4

COUNTING:     JB TF0, TIMEOUT
            SJMP COUNTING           ; COUNTING IN LOOP

TIMEOUT:      CLR TF0
            CLR TR0                 ; CLEAR TCON.4
            RET

;——————DELAY 200MS———————————————————————
DELAY200:     MOV R0, #04H
LOOP4:        ACALL DELAY50
            DJNZ R0, LOOP4
            RET

;————————BULK ERASE COMMAND————————————————————
; BULK ERASE COMMAND: 00011
BERASE_CMD:   CLR MODE                 ; MODE=0
            SETB SDI                 ; SDI=1
            ACALL SCLOCK             ; SCLK LHL CLOCK IN 2 ONES
            ACALL SCLOCK             ; SCLK LHL
            CLR  SDI                 ; SDI=0
            ACALL SCLOCK             ; SCLK LHL
            ACALL SCLOCK             ; SCLK LHL
            ACALL SCLOCK             ; SCLK LHL
            RET

;———EXECUTE COMMAND——LL————————————————————————
;THIS COMMAND CAN BE USED TO EXECUTE LOAD COMMAND AS WELL AS EXECUTE
;INSTRUCTIONS

EXECUTE:      CLR SDI
              CLR MODE
            ACALL SCLOCK
            RET
;————————————————————————————————————

;————————PROGRAM COMMAND————————————————————

;THIS SUB ROUTINE DOES NOT HAVE THE SET UP REQUIRED PRIOR TO ENTERING
;INTO IT.
```

```
; PROGRAM COMMAND: 00111
PROGRAM_CMD:    CLR MODE                ; MODE=0
            SETB SDI                ; SDI=1
            ACALL SCLOCK            ; SCLK LHL
            ACALL SCLOCK            ; SCLK LHL
            ACALL SCLOCK            ; SCLK LHL
            CLR  SDI                ; SDI=0
            ACALL SCLOCK            ; SCLK LHL
            ACALL SCLOCK            ; SCLK LHL
            RET

;————SHIFT DATA COMMAND————————————

; SHIFT DATA COMMAND: 00010
; SET UP OF THE COMMAND TO ACCEPT PROGRAMMING INFORMATION

SHFTDATA_CMD:   CLR MODE                ; MODE=0
            CLR  SDI                ; SDI=0
            ACALL SCLOCK            ; SCLK LHL
            SETB SDI                ; SDI=1
            ACALL SCLOCK            ; SCLK LHL
            CLR  SDI                ; SDI=0
            ACALL SCLOCK            ; SCLK LHL
            ACALL SCLOCK            ; SCLK LHL
            ACALL SCLOCK            ; SCLK LHL
            RET


;—— JEDEC DATA CONTAINS DUMMY BITS AND ADDRESSES ————
;NOTE THAT IN THE DATA BLOCK BELOW THAT THE COMPILER REQUIRES THAT
;NUMBERS DO NOT HAVE A LEADING A LETTER AND AS SUCH A LEADING 0 MUST
;BE INCLUDED.
;BIT MAPPING AND TRANSLATION OF THE JEDEC PATTERN THAT IS TO BE
;PROGRAMMED INTO THE PART. NOTE THE BYTES AND BITS HAVE BEEN REVERSED
;IN THE ORDER THAT IS SHOWN IN THE MANUAL.

;BIT SHIFTING IS DONE LEFT TO RIGHT.

; F    A    A    A    C    0
;1111 1010 1010 1010 1100 0000
; F    A    A    A    E    0
;1111 1010 1010 1010 1110 0000
; F    A    A    A    D    0
;1111 1010 1010 1010 1101 0000
; F    A    A    A    F    0
;1111 1010 1010 1010 1111 0000
; F    A    A    A    C    8
;1111 1010 1010 1010 1100 1000
; F    A    A    A    E    8
;1111 1010 1010 1010 1110 1000
; F    A    A    A    D    8
;1111 1010 1010 1010 1101 1000
; F    A    A    A    F    8
;1111 1010 1010 1010 1111 1000
```

```
; F    A    A    A    C    4
;1111 1010 1010 1010 1100 0100
; F    A    A    A    E    4
;1111 1010 1010 1010 1110 0100
; F    A    A    A    D    4
;1111 1010 1010 1010 1101 0100
; A    A    A    A    F    4
;1010 1010 1010 1010 1111 0100
; A    A    A    A    C    C
;1010 1010 1010 1010 1100 1100
; A    A    A    A    A    A
;1010 1010 1010 1010 1010 1010
; A    A    A    A    A    9
;1010 1010 1010 1010 1010 1001
; A    A    A    A    A    B
;1010 1010 1010 1010 1010 1011

JEDEC_TABLE:

DB  0FAH,0AAH,0C0H

DB  0FAH,0AAH,0E0H

DB  0FAH,0AAH,0D0H

DB  0FAH,0AAH,0F0H

DB  0FAH,0AAH,0C8H

DB  0FAH,0AAH,0E8H

DB  0FAH,0AAH,0D8H

DB  0FAH,0AAH,0F8H

DB  0FAH,0AAH,0C4H

DB  0FAH,0AAH,0E4H

DB  0FAH,0AAH,0D4H

DB  0AAH,0AAH,0F4H

DB  0AAH,0AAH,0CCH

DB  0AAH,0AAH,0AAH

DB  0AAH,0AAH,0A9H

DB  0AAH,0AAH,0ABH

END
; END OF THE CODE SEGMENT FOR THE ispGDS22.
; USER CODE CAN BE APPENDED FROM THIS POINT FORWARD.
```

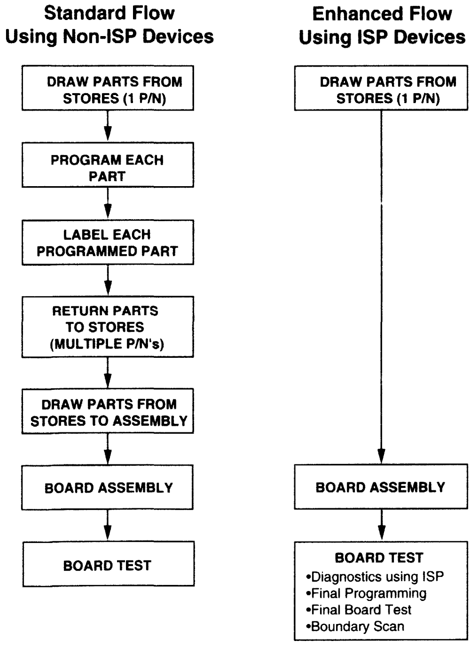
ATE Programming of ISP Devices

Overview

This section discusses how you can use Automatic Test Equipment (ATE) to program and verify ISP devices. By using ATE to perform the production programming of ISP devices, you can avoid the overhead and time penalties associated with the use of stand-alone device programmers. You can also enhance the testability of your product with ATE programming, since you can develop custom configurations of ISP devices specifically for board-level testing, then reconfigure the ISP devices to the production pattern after board test is complete.

Advantages of ISP ATE Programming

Figure 1. ISP Manufacturing Flow



ATE programming allows a dramatic simplification of the standard PLD flow which translates into significant cost savings, specifically for the reasons listed below:

- Reduced manufacturing steps (Figure 1)
- Reduced handling/no bent leads
- Elimination of mixed patterns/wrong socketing
- Simplified inventory requirements

Overview of ATE Programming Process

All Lattice ISP devices are programmed through the use of four or five TTL level signals, referred to as the ISP interface. Data is serially shifted into the device and, through the use of ISP programming instructions, used to program the device (see section "ISP Design Flow" in The Basics of ISP). Since this interface uses TTL levels, it is easily driven by an ATE tester.

Since the device programming information is defined as a JEDEC file, any ATE programming solution requires that you first create a JEDEC file, then have some way of translating the JEDEC file into signals on the ISP interface driven by the ATE. There are two methods of performing this translation that are currently available:

- Create test vectors to program the devices using a translation tool from Lattice
- Write a program in the ATE's high-level language

Both of these methods are discussed in the next sections. A brief overview of the advantages and disadvantages of each approach is shown in Table 1.

Table 1. Translation Methods

ATE Method	Advantages	Disadvantages
Test Vector Generation	• Can use existing Lattice utilities • Format can be changed for different tester requirements	• Large number of vectors • Tester must be able to accept test vector input
Custom Tester Program	• Compact (few vectors) • May be faster than test vector approach	• User must understand ISP programming • May require more development time

Generating Test Vectors

The test vector approach to programming ISP devices requires that you create a set of vectors from a JEDEC file which will program the device using the ISP interface. The number of vectors required will be quite large (for example, the ispLSI 1032 requires 150K vectors for ATE program and verify), thus you will be required to use some sort of software tool to create the test vectors from a JEDEC file.

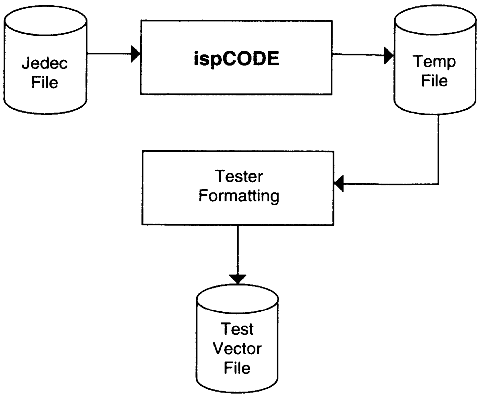
Lattice provides test vector creation tools free of charge via the Lattice BBS, as described in the following section "Lattice Test Vector Creation Software". This software creates the test vectors necessary to program a device from a JEDEC file.

The tools are based on Lattice's ispCODE™ software. The ispCODE software supports the programming of one or more ISP devices through the parallel port of a PC, which is attached through a cable, called the ispDOWNLOAD™ cable, to the ISP interface. The software drives the pins of the parallel port to program the ISP device. The test vector creation software simply redirects these pin values to a file, instead of to the parallel port of the PC. The file contents can then be processed and converted to the vector format required by a particular tester (Table 2). This process is illustrated in Figure 2. What is called a temporary file in Figure 2 would have been the parallel port of the PC. But as you can see from the figure, the internal mechanism of how this process works is transparent. You simply input a JEDEC file, and get test vectors out in your tester format.

Table 2. Lattice Supported Testers

Company	Model
Hewlett Packard	All testers including: Models 3060, 3065, 3070, 3073
GenRad	GR228X/e Series
Teradyne	Z1800 Series & Z8000 Series –Vector Processor Option must be installed

Figure 2. Test Vector Creation Software



As an example, Hewlett Packard's board testers require vectors in their "HP-PCF" format. An example of HP-PCF compliant test vectors is shown in Listing 1.

Listing 1. HP-PCF Test Vector Format

```
! *******************************************************************
!                         TEST PROGRAM HEADER
! *******************************************************************
!
! Created: Thu Mar 31 15:56:13 1994
! JEDEC input: cnt32a.jed
!
! *******************************************************************
!                         DECLARATION SECTION
! *******************************************************************

! SINGLE-PIN ASSIGNMENTS:

assign SCLK  to pins 1
assign SDI   to pins 2
assign SDO   to pins 3
assign MODE  to pins 4
assign ISPEN to pins 5

family TTL
! TYPE CLASSIFICATIONS:
nondigital UNUSED
```

```
inputs SCLK, SDI, MODE, ISPEN
outputs SDO

pcf order is SDO, ISPEN, MODE, SCLK, SDI

! ******************************************************************
!                      VECTOR EXECUTION SECTION
! ******************************************************************
unit "ul"
pcf
"X0101"
"X0111"
! Vector 100460
"X0000"
"X0010"
"X0001"
"X0011"
"X0000"
"X0010"
"X0000"
"X0010"
"X0000"
"X0010"
! Vector 100470
"X0101"
"X0111"
"X0001"
"X0011"
"X0001"
"X0011"
(deleted section of vectors)
"X0101"
"X0111"
"X0000"
"X0010"
end pcf
wait 45m
pcf
"X0101"
"X0111"
! Vector 100810
"X0100"
"X0110"
end pcf
end unit
```

Lattice Test Vector Generation Software

Lattice has created utilities to aid in the generation of test vectors. The most current (and recommended) approach is to use the Lattice ISP Serial Programmer software, available on the Lattice Hillsboro BBS as SERIAL.ZIP and also as uncompiled C++ source code in the example Windows application included with ispCODE. In addition to providing serial programming of the ISP device family through the parallel port of a PC, the utility also has an option to generate test vectors. These test vectors also support serial programming.

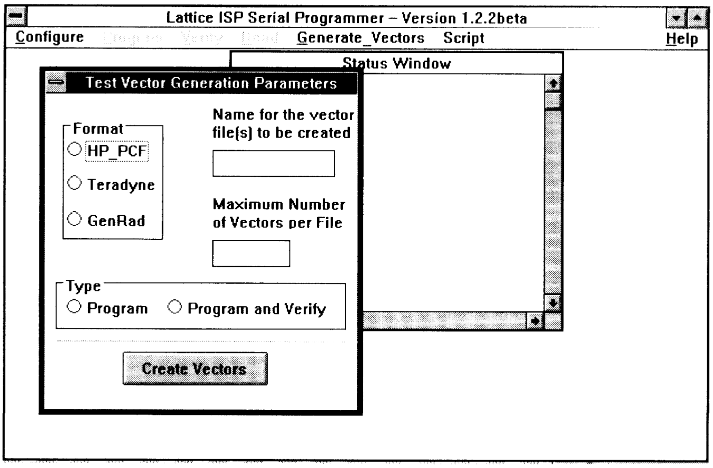
An earlier utility, called JED2PCF, is also available on the BBS as JED2PCF.ZIP. This is a DOS, command-line version that generates HP-PCF format vectors only for the ispLSI 1016, 1024, 1032, and 1048 devices. It does not support serial programming. The vectors it generates support programming one device at a time only (no daisy chain support).

Moving back to the more integrated Lattice ISP Serial Programmer Software, test vector generation is started by selecting the Generate_Vectors option. The software then displays a dialog box which gives the user control over various vector generation options (Figure 3).

Vector Generation Options

The various test vector options are summarized in Table 3. Since this is a preliminary version of the software, these options are subject to change.

Figure 3. Test Vector Generation Parameter Dialog Box



Lattice ISP Serial Programmer – Version 1.2.2beta
Configure
Generate_Vectors
Script
Help
Status Window
Test Vector Generation Parameters
Format
○ HP_PCF
○ Teradyne
○ GenRad
Name for the vector file(s) to be created
Maximum Number of Vectors per File
Type
○ Program
○ Program and Verify
Create Vectors

Table 3. Test Vector Generation Options

Catagory	Options
Format	HP-PCF: Test Vectors Teradyne: Test Vectors GenRad: Test Vectors
Type	Program: Generates Vectors for Programming Only Program and Verify: Generates Vectors for both Programming and Verification
Name for Vector Files	One or more files are created containing the vectors. If there are too many vectors per file, additional files are created (see "Max Vectors Per File" below). The first six characters of the name you enter are used as the base for the file name and an additional two characters indicate which file it is in the set of vector files (example: vector01, vector02...etc.).
Max Vectors per Files	Your tester may only be able to handle a certain number of vectors per file. The "Max Vectors Per File" field sets the limit. When the limit is reached, a new file is created to contain the additional vectors.

Writing Software in High-Level Language

Another approach to programming Lattice ISP devices on a tester is to develop a high-level language program in the tester's language. For example, some models of the GenRad tester family support a programming language similar to Pascal. Developing a high-level language test program for programming ISP devices requires a thorough understanding of the ISP interface. Also, users need to know how the fuses in the JEDEC file will correspond to bits in the rows of data that are shifted into the device. For the ispLSI family, this correspondence is straightforward. For the ispGAL22V10 and ispGDS devices, however, the fuse map to data bit mapping is more complex, requiring extra bits for row addressing, and for the ispGAL22V10, some difficult cross referencing.

Fortunately, some Teradyne testers are IBM PC controlled, so they can use C or C++, the same language as ispCODE. Thus, when high-level routines need to be implemented in Teradyne testers, you only have to modify the existing ispCODE routines to suit your application. Please refer to the ispCODE section in this manual for a thorough discussion of the ispCODE routines.

For users who do not have testers which support the C language, the following example is a step-by-step guide to programming the ispLSI 1032 on a GenRad tester. The GenRad language is based on Pascal with simple additions to control properties of the tester. The test program flow chart is described in the GenRad ISP Program Procedure which begins on the next page. To speed up the programming time, the address shift register is not reloaded for each location. Instead, a "1" is clocked through the shift register. This saves time but requires the JEDEC file to be altered so that the first address is last. A simple AWK program, detailed in Listing 2, completes this task before the file is moved over to the tester.

Listing 2. AWK program for modifying the JEDEC file

```
£!  /bin/sh
if test $£ -lt 1
then
echo Usage : jedconv [filename.jed]
echo Please reenter file name with extension !
echo
exit 1
fi
£
echo This program takes the standard Lattice JEDEC file and
echo converts it for accelerated programming. The new version
echo is saved as isp.tsr.
echo
echo converting ispLSI jedec file ......
echo
£
awk 'lenth($1)>79 && length($1)<81 {print $1 > "tempjed"}' $*
awk '{x[NR]=$0}
      END {for(i=NR; i>0; i=i-4)
      printf("%s\n%s\n%s\n%s\n",x[i-3],x[i-2],x[i-1],x[i] )> "isp.tsr"}' tempjed
\rm tempjed

echo Conversion complete.
echo
```

The GenRad ISP Program Procedure

The following is an overview of the GenRad program. The complete listing follows the overview.

Header Section

Step	Actions	Examples
1.	Define Procedure	test U1 dproc = d_fail_proc
2.	Define Signal Types	signal IN8, IO48, IO49 .... SDO_IN2 : hcmos_logic hcmos_currentset verify ; signal SDI_IN0, SCLK_IN3, MODE_IN1, ISPEN, RESETX : hcmos_logic hcmos_currentset;
3.	Define Variables	VAR lapse : integer; .... fuse_map : array [1 .. 34560] of char; data_reg : array [1 .. 320] of logic; .... testjed : text ;

4.	Define delay times	cycle default interval:= 500n;
		@(400n) sense ()
		end;
		cycle prog_delay interval := 2m;
		end cycle;

5.	Begin Program	begin
6.	Define device	d_component := 'U1';
7.	Set up burst mode	burst initialize active nomaxtime begin

Bulk Erase Section

Step	Actions	Examples
8.	Move to shift state	sck MODE_IN1 := 1 SDI_IN0 := 1;
9.	Take MODE low	last_event MODE_IN1 :=0;

10.	Clock in Bulk erase command	sck SDI_IN0 := 1;
		sck SDI_IN0 := 1;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;

11.	Move to execute state	sck MODE_IN1 := 1 SDI_IN0 := 1;

12.	Execute erase command	last_event MODE_IN1 :=0;
		sck;

13.	Wait for bulk erase to complete	for lapse :=1 to 120 do
		begin
		prog_delay;
		end;

Programming Section

Step	Actions	Examples

14.	Move to shift state	last_event MODE_IN1 :=1;
		sck;

15.	Take MODE low	last_event MODE_IN1 :=0;

16.	Load Address Shift command	sck SDI_IN0 := 1;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;

17.	Move to execute state	last_event MODE_IN1 :=1;
		sck SDI_IN0 := 1;

18.	Execute command	last_event MODE_IN1 :=0 SDI_IN0 :=0;

19.	Fill Address buffer with '0's	for addr_reg :=1 to 107 do
		begin
		sck;
		end;

20.	Clock in '1' to address row 108	sck SDI_IN0 :=1;

21.	Move to idle state	last_event MODE_IN1 := 1 SDI_IN0 :=0;
		sck;

22.	Move out of burst mode	end burst initialize
23.	Open program file	reset (testjed, '/work/fk/isp/isp.data');

24.	Set up Loops to copy ISP data to an array	for line_num := 0 to 431 do
		begin
		for char_num := 1 to 80 do
		begin

25.	Copy the JEDEC file into an array	read (testjed,fuse_map[(char_num+(80 * line_num))]);
		end;

26.	Ignore the carriage return	readln(testjed);
		end;

27.	Begin programming loop	for addr_num := 0 to 107 do
		begin

28.	Convert array from type char to type logic	for fuse_num := 1 to 320 do
		begin

		if fuse_map[(fuse_num+(320*addr_num))]='1' then
		data_reg[fuse_num] := b'1
		else
		data_reg[fuse_num] := b'0
		end;

29.	Move back to burst mode	burst blow_test_function active nomaxtime inherit initialize; begin

30.	Move to shift state	last_event MODE_IN1 := 1;
		sck SDI_IN0 := 1;

31.	Take MODE low	last_event MODE_IN1 :=0;

32.	Load data shift command	sck SDI_IN0 := 0;
		sck SDI_IN0 := 1;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;

33.	Move to execute state	last_event MODE_IN1 :=1 SDI_IN0 := 1;
		sck;

34.	Execute command	last_event MODE_IN1 :=0 SDI_IN0 :=0;

35.	Shift in high order bits	for fuse_num := 1 to 160 do
		begin
		sck SDI_IN0 := data_reg[fuse_num];
		end;

36.	Move to shift state	last_event MODE_IN1 := 1 SDI_IN0 := 1;
		sck;

37.	Take MODE low	last_event MODE_IN1 :=0;

38.	Load program high order	sck SDI_IN0 := 1;
		bits commandsck SDI_IN0 := 1;
		sck SDI_IN0 := 1;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;

39.	Move to execute state	last_event MODE_IN1 :=1 SDI_IN0 := 1;
		sck;

40.	Execute command	last_event MODE_IN1 :=0 SDI_IN0 :=0;
		sck;

41.	Wait for device to finish programming	for lapse := 1 to 25 do
		begin
		prog_delay;
		end;

42.	Move to shift state	last_event MODE_IN1 := 1 SDI_IN0 := 1;
		sck;

43.	Take MODE low	last_event MODE_IN1 :=0;

44.	Load data shift command	sck SDI_IN0 := 0;
		sck SDI_IN0 := 1;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;

45.	Move to execute state	last_event MODE_IN1 :=1 SDI_IN0 := 1;
		sck;

46.	Execute command	last_event MODE_IN1 :=0 SDI_IN0 :=0;

47.	Shift in low order bits	for fuse_num := 161 to 320 do
		begin
		sck SDI_IN0 := data_reg[fuse_num];
		end;

48.	Move to shift state	last_event MODE_IN1 := 1 SDI_IN0 := 1;
		sck;

49.	Take MODE low	last_event MODE_IN1 :=0;

50.	Load program low order bits command	sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 1;
		sck SDI_IN0 := 0;

51.	Move to execute state	last_event MODE_IN1 :=1 SDI_IN0 := 1;
		sck;

52.	Execute command	last_event MODE_IN1 :=0 SDI_IN0 :=0;
		sck;

53.	Wait for device to finish programming	for lapse := 1 to 25 do
		begin
		prog_delay;
		end;

54.	Move to shift state	last_event MODE_IN1 := 1 SDI_IN0 := 1;
		sck;

55.	Take MODE low	last_event MODE_IN1 :=0;

56.	Load address shift command	sck SDI_IN0 := 1;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;
		sck SDI_IN0 := 0;

		sck SDI_IN0 := 0;

57.	Move to execute state	last_event MODE_IN1 :=1 SDI_IN0 := 1;
		sck;

58.	Execute command	last_event MODE_IN1 :=0 ;
59.	Clock the address shift register one place	sck SDI_IN0 := 0;

60.	Move to idle state	last_event MODE_IN1 := 1;
		sck;

61.	Move out of burst mode	end burst blow_test_function;
62.	End of Array proc loop	end;

The following is a complete listing of a GenRad program to program a Lattice ispLSI 1032 device.

Listing 3. GenRad Program for Programming ispLSI 1032 Device.

```
(* Test and programming routine for ispLSI1032.

   Test & program sequence is :

   1. ID-CHECK
   2. FLOWTHROUGH TEST
   3. BULK-ERASE
   4. PROGRAM TEST S/W
   5. TEST DEVICE
   6. BULK-ERASE
   7. PROGRAM MAIN S/W
   8. VERIFY DEVICE
   9. SET UES
  10. SECURE
*)

test U1 dproc=d_fail_proc ;

  signal IN6, IO48, IO49, IO50, IO51, IO52, IO53, IO54, IO55,
      IO56, IO57, IO58, IO59, IO60, IO61, IO62, IO63, IN7, Y0, Y2,
      Y1, IN4, IO0, IO1, IO2, IO3, IO4, IO5, IO6, IO7, IO8, IO9,
      IO10, IO11, IO12, IO13, IO14, IO15, IN5, IO16, IO17, IO18,
      IO19, IO20, IO21, IO22, IO23, IO24, IO25, IO26, IO27, IO28,
      IO29, IO30, IO31, IO46, Y3, IO38, IO39, IO40, IO32, IO33,
      IO34, IO35, IO36, IO37, IO47, IO43, IO44, IO41, IO42, IO45,
      SDO_IN2 .
      : hcmos_logic hcmos_currentset verify;
      SDI_IN0, SCLK_IN3, MODE_IN1, ISPEN, RESETX
      : hcmos_logic hcmos_currentset;

VAR
  yesno    : char;
  testjed  : text;
```

```
mainjed  : text;
verfout  : text;
lapse    : integer;
err_cnt  : integer;
addr_reg : integer;
addr_num : integer;
line_num : integer;
char_num : integer;
fuse_num : integer;
veri_cnt : integer;
data_reg : array[1..320]  of logic;
verflgic : array[1..320]  of logic;
verfchar : array[1..320]   of char;
fuse_map : array[1..34560] of char;

cycle default interval:=500n;
     @(400n) sense()
end;

cycle sck interval:=2.7u;
    sclk_in3  :@(700n, 1.7u) drive (1) q0;
    sdi_in0       :@0n drive();
    mode_in1  :@0n drive();
    sdo_in2   :@2.5u sense();
    ispen   :@0n drive();
    resetx        :@0n drive();
end cycle;

cycle prog_delay interval:= 2m;
end cycle;

cycle verify_pause interval:= 30u;
end cycle;

cycle isp_sig interval:= 10u;
    ispen   :@0n drive();
end cycle;

begin
d_component:='U1';

writeln('Initial sequence running');

burst initialize active nomaxtime;
    begin

(* test m1 *)

(***************************)
(*  SEQUENCE 1 : ID CHECK  *)
(***************************)

sck ISPEN:=1 SDI_IN0:=1 MODE_IN1:=1 RESETX:=1
    SDO_IN2=b'U; (*initialize clk*)
```

```
isp_sig ISPEN:=0; (*ispen low for 10us  to enter prog state*)
$ ISPEN:=0 SDI_IN0:=0 MODE_IN1:=0 ; (*put device in idle state*)

$ RESETX:=0; (*hold low throughout to prevent internal data contention*)

sck MODE_IN1:=1; (*load device id to shift reg*)

$ SDI_IN0:=1 MODE_IN1:=0; (*prepare to read id*)

(*the id for an ispLSI1032 is 00000011. The 1st bit is active as soon as
  mode goes low and is the lsb. 7 more clocks will shift out the id on
  the SDO pin, then on clk#8 the level at SDI (as it was at clk#1)
  will appear at SDO*)

$ SDO_IN2=1;             (*read 1st id bit*)
sck SDO_IN2=1;           (*read 2nd bit*)
sck SDO_IN2=0;           (*read 3rd bit*)
sck SDO_IN2=0;           (*read 4th bit*)
sck SDO_IN2=0;           (*read 5th bit*)
sck SDO_IN2=0;           (*read 6th bit*)
sck SDO_IN2=0;           (*read 7th bit*)
sck SDO_IN2=0;           (*read 8th bit*)
sck SDO_IN2=1;           (*SDI i/p shifted from clk#1*)
$ SDO_IN2=b'U;

(***********************************)
(*  SEQUENCE 2 : FLOWTHROUGH TEST  *)
(***********************************)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load flowthru command, instruction is 01110 loading lsb 1st*)
sck MODE_IN1:=0 SDI_IN0:=0;
sck SDI_IN0:=1;
sck SDI_IN0:=1;
sck SDI_IN0:=1;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

sck MODE_IN1:=0; (*execute flowthru command*)

(* check sdi = sdo *)
$ SDI_IN0:=1 SDO_IN2=1;
$ SDI_IN0:=0 SDO_IN2=0;
$ SDI_IN0:=1 SDO_IN2=1;
$ SDI_IN0:=0 SDO_IN2=0;
$ SDI_IN0:=1 SDO_IN2=1;
$ SDI_IN0:=0 SDO_IN2=0;
$ SDI_IN0:=1 SDO_IN2=1;
$ SDI_IN0:=0 SDO_IN2=0;
$ SDI_IN0:=1 SDO_IN2=1;
$ SDI_IN0:=0 SDO_IN2=0;
$ SDO_IN2=b'u;
```

```
(****************************)
(*  SEQUENCE 3 : BULK ERASE  *)
(****************************)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load bulk erase command, instruction is 00011 loading lsb 1st*)
sck MODE_IN1:=0 SDI_IN0:=1;
sck SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

sck MODE_IN1:=0; (*execute erase command*)

for lapse := 1 to 120 do (*wait 240ms for erase to finish*)
begin
prog_delay;
end;

(**********************************)
(*  SEQUENCE 4 : PROGRAM TEST S/W  *)
(**********************************)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load address shift command, instruction is 00001 loading lsb 1st*)
sck MODE_IN1:=0 SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

$ MODE_IN1:=0 SDI_IN0:=0; (*execute address shift command*)

for addr_reg := 1 to 107 do (*initialize address register*)
begin
sck;
end; (*addr reg now full of zero's*)

sck SDI_IN0:=1; (*address row 107 set to '1' and ready to program*)

sck MODE_IN1:=1 SDI_IN0:=0; (*enter idle state*)

end burst initialize; (* END OF BURST *)

writeln('Reading isp data from file');

reset(testjed,'/work2/fk/isp/isp.data'); (*open ispdata file*)
```

```
(*load the jedec data from file to burst array*)
for line_num := 0 to 431 do (*432 lines in the isp file*)
begin
for char_num := 1 to 80 do  (*each line is 80 chars long*)
begin
read(testjed,fuse_map[(char_num + (80 * line_num))]);(*load the array*)
end;
readln(testjed); (*ignore carriage return at end of line*)
end;
(* the array 'fusemap' now contains the isp file*)

(* START OF DEVICE ARRAY PROGRAMMING LOOP *)

writeln('Programming loop running');

for addr_num := 0 to 107 do (*address loop counter*)
begin

(*load fuse map array one address at a time to data reg array and
  simultaneously convert type 'char' to type 'logic'*)
for fuse_num := 1 to 320 do
begin
if fuse_map[(fuse_num + (320 * addr_num))] = '1' then
data_reg[fuse_num] := b'1
else
data_reg[fuse_num] := b'0;
end;

burst blow_test_function active nomaxtime inherit initialize;
     begin

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load data shift command, instruction is 00010 loading lsb 1st*)
sck MODE_IN1:=0 SDI_IN0:=0;
sck SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

$ MODE_IN1:=0 SDI_IN0:=0; (*execute data shift command*)

(* shift in 160 high order bits for row to be programmed *)

for fuse_num := 1 to 160 do
begin
sck SDI_IN0:= data_reg[fuse_num];
end;

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load program high data command, instruction is 00111 loading lsb 1st*)
```

```
sck MODE_IN1:=0 SDI_IN0:=1;
sck SDI_IN0:=1;
sck SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

sck MODE_IN1:=0 SDI_IN0:=0; (*execute program high data command*)

for lapse := 1 to 25 do (*wait 50ms for high bits of row to be programmed*)
begin
prog_delay;
end;

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load data shift command, instruction is 00010 loading lsb 1st*)
sck MODE_IN1:=0 SDI_IN0:=0;
sck SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

$ MODE_IN1:=0 SDI_IN0:=0; (*execute data shift command*)

(* shift in 160 low order bits for row to be programed *)

for fuse_num := 161 to 320 do
begin
sck SDI_IN0:= data_reg[fuse_num];
end;

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load program low data command, instruction is 01000 loading lsb 1st*)
sck MODE_IN1:=0 SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=1;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*);

sck MODE_IN1:=0 SDI_IN0:=0; (*execute program low data command*)

for lapse := 1 to 25 do (*wait 50ms for low bits of row to be programmed*)
begin
prog_delay;
end;

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)
```

```
(*load address shift command, instruction is 00001 loading lsb 1st*)
sck MODE_IN1:=0 SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

sck MODE_IN1:=0 SDI_IN0:=0; (*move addr reg to next row, the address
                             reg will be clear after the last loop*)

sck MODE_IN1:=1; (*move to idle state*)

end burst blow_test_function; (* END OF BURST *)

end; (* END OF ARRAY PROGRAMMING LOOP *)

writeln('Device programmed.');


(******************************)
(* SEQUENCE 8 : VERIFY DEVICE *)
(******************************)

writeln('Verifying...');

burst verification active nomaxtime inherit blow_test_function;
     begin

(* device is in idle state *)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load address shift command, instruction is 00001 loading lsb 1st*)
sck MODE_IN1:=0 SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

$ MODE_IN1:=0; (*execute address shift command*)

sck SDI_IN0:=1; (* address last row *)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load ver/ldh command, instruction is 01010 loading lsb 1st*)
sck MODE_IN1:=0 SDI_IN0:=0;
sck SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=1;
```

```
 sck SDI_IN0:=0; (*load complete*)

 sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)
 sck MODE_IN1:=0 SDI_IN0:=0; (*execute ver/ldh command*)

 verify_pause; (* wait 30u for data reg to load *)

 sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

 (*load data shift command, instruction is 00010 loading lsb 1st*)
 sck MODE_IN1:=0 SDI_IN0:=0;
 sck SDI_IN0:=1;
 sck SDI_IN0:=0;
 sck SDI_IN0:=0;
 sck SDI_IN0:=0; (*load complete*)

 sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

 $ MODE_IN1:=0 SDI_IN0:=0; (*execute data shift command*)

(*clock out the high order bits from the data reg *)

 for veri_cnt := 1 to 160 do
 begin
 $ verflgic[veri_cnt]:=sdo_in2;
 sck ;
 end;

 sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

 (*load ver/ldl command, instruction is 01011 loading lsb 1st*)
 sck MODE_IN1:=0 SDI_IN0:=1;
 sck SDI_IN0:=1;
 sck SDI_IN0:=0;
 sck SDI_IN0:=1;
 sck SDI_IN0:=0; (*load complete*)

 sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

 sck MODE_IN1:=0 SDI_IN0:=0; (*execute ver/ldl command*)

 verify_pause; (* wait 30u for data reg to load *)

 sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

 (*load data shift command, instruction is 00010 loading lsb 1st*)
 sck MODE_IN1:=0 SDI_IN0:=0;
 sck SDI_IN0:=1;
 sck SDI_IN0:=0;
 sck SDI_IN0:=0;
 sck SDI_IN0:=0; (*load complete*)

 sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)
```

```
$ MODE_IN1:=0 SDI_IN0:=0; (*execute data shift command*)

(*clock out the low order bits from the data reg *)
for veri_cnt := 161 to 320 do
begin
$ verflgic[veri_cnt]:=sdo_in2;
sck ;
end;

sck MODE_IN1:=1 SDI_IN0:=0; (*move to idle state*)

end burst verification; (* END OF BURST *)

(* convert data type and compare data *)

err_cnt := 0;
yesno := 'n';

for veri_cnt := 1 to 320 do
begin
if verflgic[veri_cnt] = b'1 then (*convert type logic to type char*)
verfchar[veri_cnt]:='1'
else
verfchar[veri_cnt]:='0';

if verfchar[veri_cnt] <> fuse_map[veri_cnt] then (*compare with jedfile*)
err_cnt := err_cnt + 1;

end;

(* failure routine *)
if err_cnt > 0 then
begin
setfail;
writeln('Verification failure!!');
writeln('failed ',err_cnt,' bit(s) out of 320.');
write('Write error file? [y/n]');
readln(yesno);
end
else
writeln('Verify has passed.');

(* write out error file if req'd for programmers attention*)
if yesno = 'y' then
begin
write('Writing to file...');
rewrite(verfout,'/work2/fk/isp/verify.err');
for line_num := 0 to 3 do
begin
for veri_cnt := 1 to 80 do
begin
write(verfout,verfchar[(veri_cnt + (80 * line_num))]);
end;
writeln(verfout);
```

```
end;
writeln('Done.');
writeln('Last line written to "verify.err"');
end;

(******************************)
(*  SEQUENCE 5 : TEST DEVICE  *)
(******************************)

writeln('Testing function');

burst test_device active nomaxtime inherit verification;
begin

(*test vectors here to test counter example in lattice book*)

isp_sig ISPEN:=1; (*wait 10us to leave prog state*)

$ SDI_IN0:=1 MODE_IN1:=1 RESETX:=1; (*hold prog pins*)

$ Y0:=0 IO0:=1 IO1:=1 IO2:=0;
$ IO2:=1;
$ Y0:=1;
$ Y0:=0; (*CNTR IS RESET O/P'S ARE LOW*)
$ IO2:=0; (*READY TO CNT*)

$ IO36=0 IO37=0 IO38=0 IO39=0 IO32=0;(*0000*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=0 IO37=0 IO38=0 IO39=1 IO32=0;(*0001*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=0 IO37=0 IO38=1 IO39=0 IO32=0;(*0010*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=0 IO37=0 IO38=1 IO39=1 IO32=0;(*0011*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=0 IO37=1 IO38=0 IO39=0 IO32=0;(*0100*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=0 IO37=1 IO38=0 IO39=1 IO32=0;(*0101*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=0 IO37=1 IO38=1 IO39=0 IO32=0;(*0110*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;
```

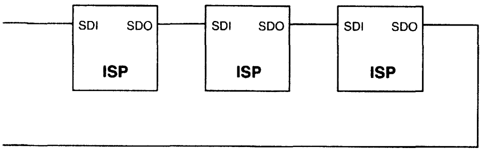
```
  $ IO36=0 IO37=1 IO38=1 IO39=1 IO32=0;(*0111*)
  $ Y0:=1 nofails;
  $ Y0:=0 nofails;

  $ IO36=1 IO37=0 IO38=0 IO39=0 IO32=0;(*1000*)
  $ Y0:=1 nofails;
  $ Y0:=0 nofails;

  $ IO36=1 IO37=0 IO38=0 IO39=1 IO32=0;(*1001*)
  $ Y0:=1 nofails;
  $ Y0:=0 nofails;

  $ IO36=1 IO37=0 IO38=1 IO39=0 IO32=0;(*1010*)
  $ Y0:=1 nofails;
  $ Y0:=0 nofails;

  $ IO36=1 IO37=0 IO38=1 IO39=1 IO32=0;(*1011*)
  $ Y0:=1 nofails;
  $ Y0:=0 nofails;

  $ IO36=1 IO37=1 IO38=0 IO39=0 IO32=0;(*1100*)
  $ Y0:=1 nofails;
  $ Y0:=0 nofails;

  $ IO36=1 IO37=1 IO38=0 IO39=1 IO32=0;(*1101*)
  $ Y0:=1 nofails;
  $ Y0:=0 nofails;

  $ IO36=1 IO37=1 IO38=1 IO39=0 IO32=0;(*1110*)
  $ Y0:=1 nofails;
  $ Y0:=0 nofails;

  $ IO36=1 IO37=1 IO38=1 IO39=1 IO32=1;(*1111 + carry*)
  $ Y0:=1 nofails;
  $ Y0:=0 nofails;

  $ IO36=0 IO37=0 IO38=0 IO39=0 IO32=0;(*0000*)
  $ Y0:=1 nofails;
  $ Y0:=0 nofails;

  $ IO36=b'u IO37=b'u IO38=b'u IO39=b'u IO32=b'u
    IO0:=b'z IO1:=b'z IO2:=b'z Y0:=b'z
    SDI_IN0:=b'z MODE_IN1:=b'z RESETX:=b'z ISPEN:=b'z;

end burst test_device; (* END OF BURST *)

writeln('Finished');

end test U1;
```

Interfacing the ATE to the ISP Hardware

The ATE to ISP hardware interface will be different depending on whether or not you are programming the ISP devices in a daisy chain or parallel configuration.

Case 1 (Serial Programming): For programming multiple ISP devices *sequentially* in a daisy chain configuration.

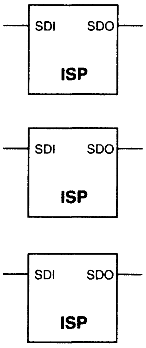
Figure 4. Serial (Daisy Chain) Programming



If case 1 and Figure 4 describe your ISP hardware configuration, there is a single daisy chain interface to which your tester must interface. All the ISP devices will be programmed by the tester through this single 4- or 5-signal interface referred to as the ISP interface. The Lattice software utilities will be able to generate the vectors required to program your hardware in this configuration without modification.

Case 2 (Parallel Programming): For programming one or more ISP devices *in parallel.*

Figure 5. Parallel Programming



If case 2 and Figure 5 describe your ISP configuration, there are multiple ISP interfaces that you will want to program at the same time. The Lattice software currently does not support the automatic generation of parallel programming vectors for multiple ISP devices. This means that to support parallel programming you will need to provide a means to concatenate vectors from separate files together into a single wide vector file. To do this, write a text modification program that takes the test vector file(s) for each ISP device and uniquely numbers all occurrences of SCLK, SDI, SDO, MODE, and ISPEN. Then, the single-pin assignments of all the files must be concatenated and the pins must be incremented producing a list similar to Listing 4.

Listing 4. Parallel Programming Algorithm for Test Vectors on ATE (For two ISP devices)

```
! SINGLE-PIN ASSIGNMENTS:

assign SCLK0  to pins 1
assign SDI0   to pins 2
assign SDO0   to pins 3
assign MODE0  to pins 4
assign ISPEN0 to pins 5
assign SCLK1  to pins 6
assign SDI1   to pins 7
assign SDO1   to pins 8
assign MODE1  to pins 9
assign ISPEN1 to pins 10
...etc.
```

Then the type classifications must be modified, appending the numbered pins:

```
!TYPE CLASSIFICATIONS
nondigital UNUSED

inputs      SCLK0, SDI0, SDO0, MODE0, ISPEN0, SCLK1,
            SD1, MODE1, ISPEN1, SCLK2,...etc.
```

Listing 4. Parallel Programming Algorithm for Test Vectors on ATE (continued)

```
outputs      SDO0, SDO1, SDO2,...etc.

pcf order    SDO0, ISPEN0, MODE0, SCLK0, SDI0, SDO1,
             ISPEN1, MODE1, SCLK1, SDI, SDO2,...etc.
```

Finally, the test vectors must be concatenated to the appropriate width:

```
pcf
"X0101X0101X0101X..."
"X0111X0111X0111X..."
...etc.
```

The tester can then direct the parts of the wide vector to the appropriate ISP interface.

Programming and Verify Wait Times

Meeting the program and verify wait times is essential to correctly program ISP devices. Since the means to implement these delays may vary from tester to tester, the vector generation utilities will simply insert a WAIT statement in the test vectors wherever a wait is required, followed by the amount of time in milliseconds that the wait should take. It is the your responsibility to ensure that your tester meets the time specified by the wait statement.

Vector Cycle Times

You must ensure that the maximum clock rate, and data setup and hold times are met during tester programming. Some customers have found a 650 ns cycle time to meet their needs, however this time should be used as a guideline only.

Pulldowns on the MODE/SDI

Both the MODE and SDI pins have internal pull-ups. If you allow these pins to float, you may find that the device transitions into a programming mode, since the active state of these signals is active high. It is recommended that you use pulldowns to ensure that these pins are pulled down to the inactive state.

Calculating the Required Pulldown Value

Lattice's ISP devices have internal pull-ups with a value of 50-70KΩ. If you connect ISP devices in parallel, then you must also take into account the fact that you are putting these pull-ups in parallel, and therefore the effective pull-up value may be much smaller. For example, five devices in parallel will give you an equivalent resistance of 10KΩ.

Maintaining ISP Interface Values

If you are using multiple files to program the ISP devices due to a large number of vectors, you need to ensure that the tester does not let the ISP interface float at any time during the programming cycle. If you are processing multiple modules, ensure that you maintain the last vector value of a module until the next vector of the new module is started.

Third-Party Programmers

Third-Party Programming Support

Lattice Semiconductor works with several industry-leading programming manufacturers to ensure that high quality programming support is available for Lattice ISP devices. Table 1 lists of the programming vendors approved to program the ispLSI, ispGAL, and ispGDS devices. For the ispLSI devices, Lattice has worked with third-party socket adapter vendors to provide programming support for low-cost programmers, such as 28-pin programmers. These adapters route the necessary programming signals from the programmer to the devices and use a standard 28-pin pinout.

Table 1. Qualified Programmers

Programmer Vendor	Model
Advin Systems	Pilot U-40 Pilot U-84 Pilot GL Pilot GCE
BP Microsystems	PLD 1128 CP-1128 BP-1200
Data I/O	2900 3900 Unisite
Logical Devices	Allpro 40 Allpro 88
SMS Micro Systems	Expert
Stag	Eclipse ZL30A ZL30B System 3000 Quasar 1040 Quasar 1084
System General	Turpro-1 Turpro-1/FX

Device Selection

When programming Lattice's ispLSI devices, there are two types of adapters you can use. First, you can use an adapter supplied by the programming vendor. These adapters make electrical connections to all pins and may be capable of applying test vectors. Or, you can use a 28-pin adapter from one of the manufacturers listed below.

- PROCON Technologies
- EDI Corporation
- Emulation Technology

When using 28-pin adapters, the correct algorithm must be selected and specified with an asterisk (*). These algorithms are listed in Table 2.

Table 2. 28-Pin Adapter Algorithms

pLSI 1016*	ispLSI 1016*
pLSI 1024*	ispLSI 1024*
pLSI 1032*	ispLSI 1032*
pLSI 1048*	ispLSI 1048*
pLSI 2032*	ispLSI 2032*
pLSI 3256*	ispLSI 3256*

Third-Party Adapters

Table 3 lists universal socket adapters by Emulation Technology, EDI Corporation, and PROCON Technologies that may be used to program Lattice's ispLSI devices with 28-pin programmers. These adapters route the necessary programming signals from the programmer to the appropriate pins of the device using a 28-pin, .600 mil socket.

Third-Party Programmers

Table 3. Third-Party Adapters

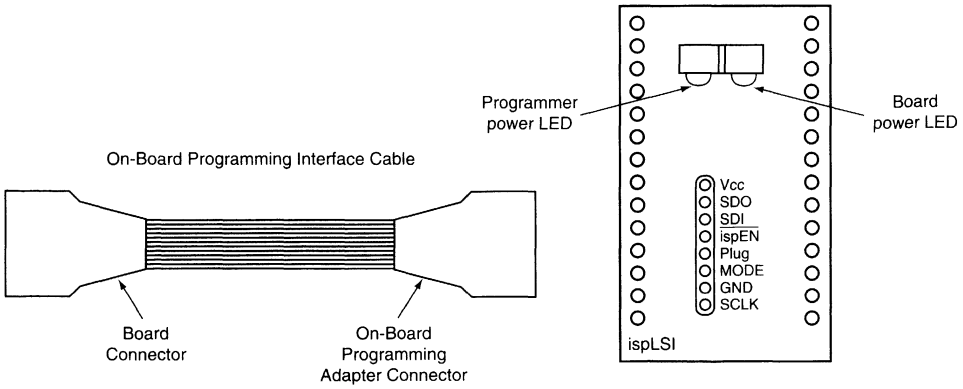
Device	Package Type	Emulation Part Number	EDI Part Number	PROCON Part Number
ispLSI or pLSI 1016	44-PLCC	AS-44-28-03P-6YAM (Hinged lid)	44PL/28D6-ZL-L1016 (Hinged lid) 44PL/28D6-ZAL-L1016 (Auto eject)	325-044-1221-028L (Auto eject)
	44-TQFP	AS-44-28-01Q600	Not available	Not available
ispLSI or pLSI 1024	68-PLCC	AS-68-28-03P-6YAM (Hinged lid)	68PL/28D6-ZL-L1024 (Hinged lid) 68PL/28D6-ZAL-L1024 (Auto eject)	325-068-1221-028L (Auto eject)
ispLSI or pLSI 1032	84-PLCC	AS-84-28-02P-6YAM (Hinged lid)	84PL/28D6-ZL-L1032 (Hinged lid) 84PL/28D6-ZAL-L1032 (Auto eject)	325-084-1221-028L (Auto eject)
	84-CPGA 100-TQFP	AS-84-28-01PG-6 AS-100-28-01Q-3 (300 MIL) AS-100-28-01Q-6 (600 MIL)	Not available Not available	Not available Not available
ispLSI or pLSI 1048	120-PQFP	AS-120-28-01Q-6YAM (Hinged lid)	120QF/28D6-ZL-L1048 (Hinged lid)	325-120-1221-028L (Hinged lid)
ispLSI or pLSI 1048C	128-PQFP 133-CPGA	AS-128-28-02Q-6YAM AS-133-28-01PG-6	Not available Not available	Not available Not available
ispLSI or pLSI 2032	44-PLCC	AS-44-28-03P-6YAM	44PL/28D6-ZL-L1016 (Hinged lid) 44PL/28D6-ZAL-L1016 (Auto eject)	325-044-1221-028L (Auto eject)
ispLSI or pLSI 3256	167-CPGA	AS-167P10-28-6-pLSI3256	Not available	Not available

On-board Programming Adapters

An alternative to programmer-based programming is "on-board programming", which is performed with an on-board programming adapter. Using an interface board and ribbon cable, the on-board programming adapter routes the programming signals from a programmer to the PC board (Figure 1). An interface header must be included on the board to route the programming signals to the device. The following adapters are currently available from PROCON Technology:

- ispLSI (Supports: ispLSI 1048C/1048/1032/1024/1016)
- ispGDS (Supports: ispGDS14/18/22)
- ispGAL22V10

Figure 1. The PROCON ispLSI On-board Programming Adapter routes the programming signals from a 28-pin programmer to a header on the PC board.



Design Requirements

In order to program a device on-board, the board must be designed with the following considerations:

- An 8-pin header must be included on the board to connect the 8-pin ribbon cable.
- The normal programming signals, along with VCC and GND, must be routed to an 8-pin header interface. The signal definitions are provided below:

VCC	(VCC Supply)
SDO	(Serial Data Out)
SDI	(Serial Data In)
$\overline{\text{ispEN}}$	(ISP pin for ispLSI devices)
Plug	(Plug-Alignment)
Mode	(Mode control pin)
GND	(GROUND Supply)
SCLK	(Clock Driver)

Programming

To program a device on-board, insert the interface board into the programmer and select the device. Connect the ribbon cable from the programmer to the interface board. The LED labeled "Board Power" will illuminate to show that power is applied to the board and the cable is properly connected. Download the JEDEC file to the programmer and program the device. The LED labeled "Programmer Power" will illuminate when the programmer applies power to the socket. A design example is shown in Figure 2.

Device	PROCON Part Numbers
ispLSI	325-isp-1221-028L
ispGDS	325-GDS-1221-028L
ispGAL22V10	325-GAL-1221-028L

On-board Programming Adapters

Multiple Devices

To program multiple devices on-board using the On-board Programming scheme, two design approaches may be used:

1. Include Multiple interface headers to program each individual device.

2. Include a rotary switch or DIP switch on the board to select the device to program. If all of the devices are ispLSI devices, then the $\overline{\text{ispEN}}$ control signal will serve to select which device is programmed. The number of devices that are programmed determines the number of rotary/DIP switches required. A design example is shown in Figure 3.

Figure 2. Programming a single ispLSI device on-board using a PROCON adapter. The programming signals are routed to a header connector on the PC board which is then connected to the Interface Cable of the adapter.

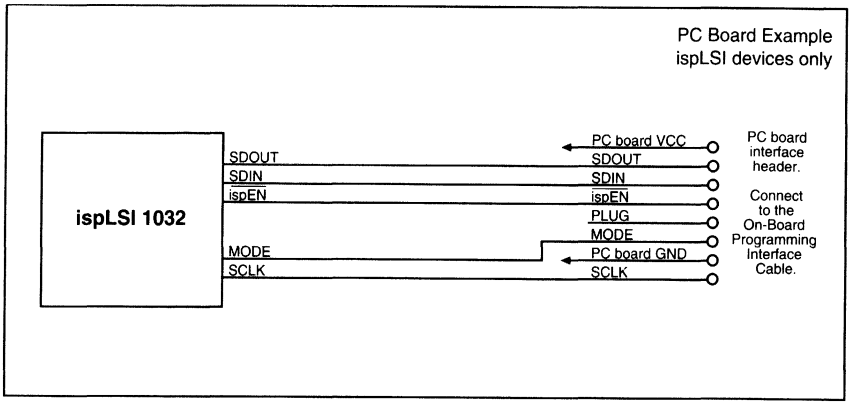
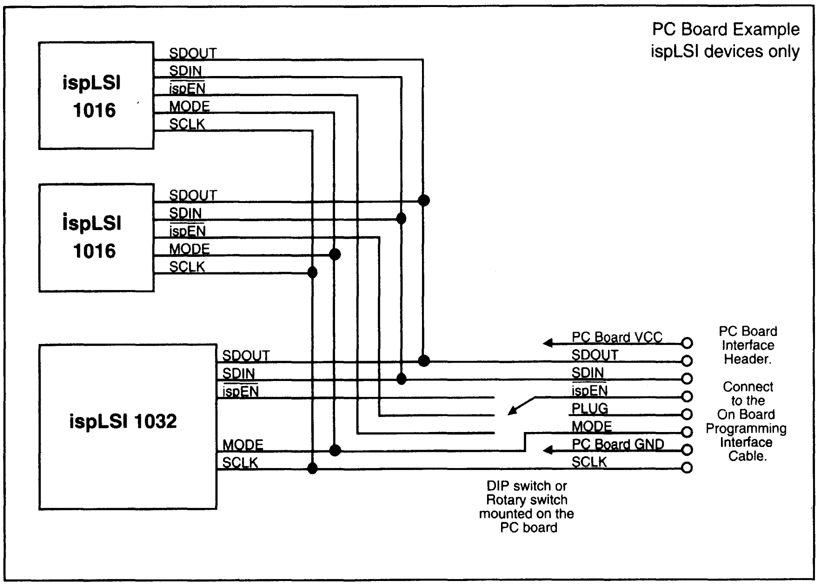


Figure 3. Programming ispLSI devices on-board using a PROCON adapter. The PROCON adapter routes the programming signals from a universal programmer to the PC board.



On-board Programming Adapters

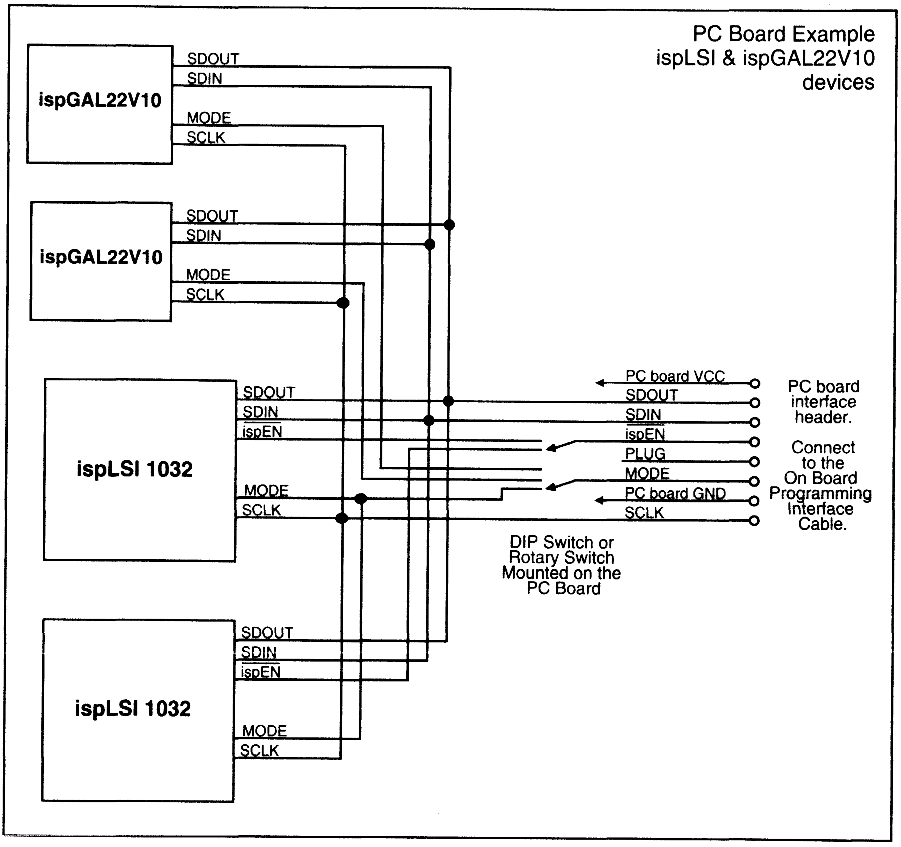
Mixed ISP Devices

When programming multiple devices on-board using the on-board programming adapter, two switches are required to select the device to program. In Figure 4, all of the SDOUT pins are connected together, which stipulates that only one device may be enabled at a time. If ispLSI devices are mixed with either the ispGAL22V10 or ispGDS devices then both ispEN and MODE must be used as control signals to select each device to program. ispEN will enable and select the appropriate ispLSI device, and MODE will enable and select the desired ispGAL22V10 or ispGDS device. Table 4 reviews the switches required for common on-board programming.

Table 4. Switches Required for Common On-Board Programming

Configuration	Switches
ispLSI 1016 & ispGDS14	1. Switch ispEN to enable the ispLSI devices. 2. Switch MODE to enable or disable the ispGDS14.
ispLSI 1016, ispLSI 1032, & ispGAL22V10	1. Switch ispEN to select one of the ispLSI devices or to open the ispGAL22V10. 2. Switch MODE to enable or disable the ispGAL22V10.
ispLSI 1016, ispLSI 1032, ispGDS22, & ispGAL22V10	1. Switch ispEN to select one of the ispGAL22V10 and the ispLSI devices. 2. Switch MODE to enable and select either the ispGAL22V10 or the ispGDS22.

Figure 4. In-System Programming with either an ispGAL22V10 or ispGDS device on-board using a PROCON adapter to route the programming signals from a universal programmer to the PC board.



Third-Party Programmers

Programming and Adapter Vendors

Advin Systems
1050-L East Duane Ave
Sunnyvale, CA 94086
Tel: (408) 243-7000
FAX: (408) 736-2503

BP Microsystems
1000 N Post Oak Road
Houston, TX 77055-7237
Tel: (713) 688-4600
1-800-225-2102
FAX: (713) 688-0902
BBS: (713) 688-9283

Data I/O Corp.
10525 Willows Road
P.O. Box 97046
Redmond, WA 98073-9746
Tel: 1-800-426-1045
1-800-247-5700
FAX: (206) 882-1043
Data I/O Corp.
Tel: 31 (0) 20-6622866
In Japan contact:
Data I/O Corp.
Tel: (03) 432-6991

EDI Corporation
P.O. Box 366
Patterson, CA 95363
Tel: (209) 892-3270
Fax: (209) 892-3610

Emulation Technology
2344 Walsh Ave, Bldg F
Santa Clara, CA 95051
Tel: (408) 982-0660
Fax: (408) 982-0664

PROCON Technology, Inc
1333 Lawrence Expwy, Suite 207
Santa Clara, CA 95051
Tel: (408) 246-4456
Fax: (408) 246-4435

Logical Devices, Inc.
692 South Military Trail
Deerfield Beach, FL 33442
Tel: (305) 428-6868
FAX: (305) 428-1811

SMS Microcomputer
Im Grund 15
D-88239 Wangen
Germany
Tel: (49) 7522-9728-21
FAX: (49) 7522-9728-50
In the U.S. contact:
SMS North America, Inc.
17411 NE Union Hill Road, Suite 100
N.E. Redmond, WA 98052
Tel: (206) 883-8447
FAX: (206) 883-8601

Stag Programmers, Ltd.
In Europe contact:
Silver Court, Watchmead
Welwyn Garden City
Herts, England AL7 1LT
United Kingdom
Tel: 011-44-707-332148
FAX: 011-44-707-371503
In the U.S. contact:
Stag Microsystems
1600 Wyatt Dr., Suite 3
Santa Clara, CA 95054
Tel: 1-800-227-8836
Tel: (408) 988-1118
FAX: (408) 988-1232

System General Corp.
3F, No. 1, Alley 8, Lane 45
Bao Shing Road
Shin Dian
Taipei, Taiwan R.O.C.
Tel: 886-2- 9173005
FAX: 886-2- 9111283
In the U.S. contact:
System General Corp.
1603A South Main Street
Milpitas, CA 95035
Tel: (408) 263-6667
FAX: (408) 262-9220

Section 1: ISP Overview

Section 2: The Basics of ISP

Section 3: ISP Programming Options

Section 4: Application Notes and Article Reprints

Section 5: General Information

Index

Selecting the Best Device for In-System Programmability

This article is reprinted from *Computer Design's ASIC Design* – December 1993.

Selecting the best device for in-system programmability

Familiarity with the technologies, as well as the benefits and drawbacks involved with each, will help you choose the in-system-programmable complex PLD or FPGA that best suits your design needs.

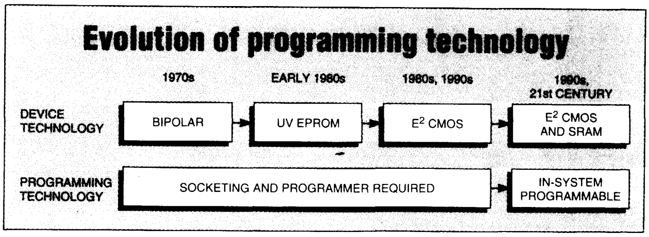
In developing an HDTV interface, David Harper, a senior design engineer at Convex Computer, faced a problem very common in leading-edge electronics companies: The industry standards were not yet complete, but product development had to go on. His interface board had to connect a Convex I/O channel to the emerging industry-standard frame-buffer format in the infant HDTV electronics market. But two different frame-buffer formats were contending for industry leadership, and it wasn't clear which format would emerge as the dominant standard.

Harper needed to design a system that could accommodate either one, or possibly both, frame-buffer standards. By populating the interface circuit board with several Lattice ispLSI high-density PLDs, Harper designed a product that could be configured to conform to either frame-buffer specification after the logic devices were soldered onto the circuit board. This in-system programmability led to a product with a wider addressable market, as well as lowering design and production costs for Convex, because a single piece of hardware could be programmed for two different products.

The ability to design one product for multiple uses is perhaps the most exotic benefit of in-system-programmable technology, but it isn't the only one. Designing in-system-programmable devices into a product can improve productivity and reduce costs across the life cycle of a product: engineering, production, maintenance, and in-field upgrades.

Why in-system programmability?

The first PLD programming technology, dating from the early days of the bipolar PAL, programmed tiny fuses on a PLD using a standalone device programmer. As PLDs migrated through subsequent technologies—notably UV EPROM and E^2CMOS—engineers continued to rely upon a standalone programming step to load the logic into each device. As a result, engineers would add sockets for the PLDs to their circuit boards in case design changes required new PLDs to be programmed and inserted.



As PLDs have migrated from bipolar one-time-programmable technology to UV EPROM and E^2CMOS, engineers have had to maintain device sockets and a separate programming step in manufacturing. But in-system-programmable technology does away with all that.

Furthermore, manufacturing personnel had to execute a programming step in product manufacturing prior to assembly.

As programmable devices now hurdle the 10,000-gate barrier, however, they incorporate many more I/O pins and are manufactured in very fine pin-pitch plastic quad flat packs and thin quad flat packs. As a result, they're increasingly delicate and intolerant to the manual programming techniques previously used with older packages. Moreover, sockets can reduce the signal integrity of the programmable device and introduce handling and inventory steps that often compromise end-product quality as well as add extra cost.

With an in-system programmable device, on the other hand, the programming pattern can be changed at any time by applying signals to the programming pins of the in-system programmable device. This programming can even be performed after the device is soldered onto the board if the engineer has made an accommodation for the programming interface. This very simple change in the programming step means that designers don't need to add sockets to have reprogrammability. Nor is a programmer needed. Moreover, in-system programmability makes possible a new way of organizing and executing product development for higher productivity and lower costs.

Selection criteria to consider

To select an in-system-programmable device, you must consider several factors before committing large financial and engineering resources to a particular technology or architecture. The most fundamental criterion is process technology.

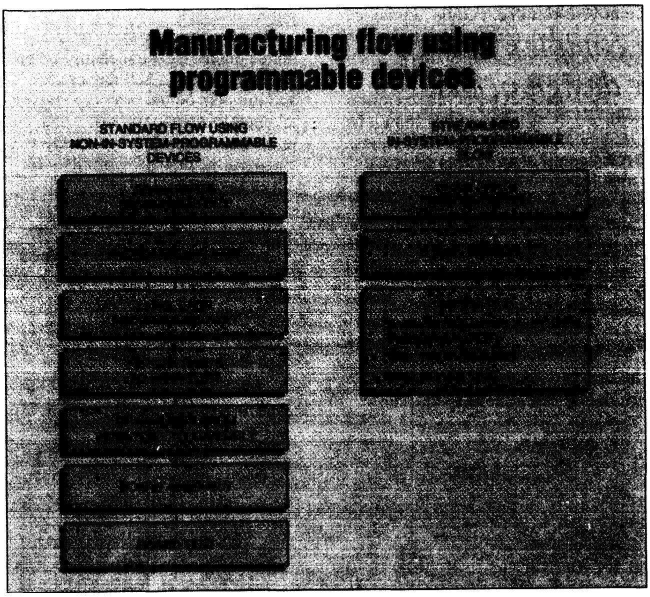
Today's high-density, programmable-logic market offers four basic storage technologies: antifuse, UV EPROM, SRAM and E^{2}CMOS. Although most devices use one of these four technologies exclusively, a few offer a hybrid approach, using SRAM for the logic and EPROM for power-off storage. Of the four basic technologies, only SRAM and E^{2}CMOS offer in-system programmability. Of the devices manufactured using these two technologies, not all incorporate in-system-programmable interfaces.

The significant advantage of SRAM devices is that they may be reprogrammed on the fly an almost unlimited number of times. This feature makes SRAM a desirable technology for applications that require continual updates or dynamic-reprogramming capability.

The main drawback of SRAM-based devices, on the other hand, is their volatility. SRAM in-system-programmable devices lose their logic configuration when system power is turned off. Every time the system is powered up, the programming pattern must be transferred from an external memory device into the SRAM in-system-programmable devices. This creates two key difficulties. First, the SRAM-based devices require a power-up delay to allow time to load the data from the EPROM device into the SRAM logic cells. Second, additional chips and valuable board real estate are required. Some manufacturers have attempted to eliminate the effect of the second limitation by including the EPROM as part of the device. Although these drawbacks do not always present a serious problem, they eliminate SRAM in-system-programmable devices from consideration in applications requiring fully functional logic at power-up. Examples include a memory decoder for a CPU that must be operational at CPU power-up and applications where board real estate carries a particularly high premium.

The second key in-system-programmable technology, E^{2}CMOS, is non-volatile and electrically erasable within milliseconds. Because there is no need for a memory chip to load the logic at each power up, the E^{2}CMOS solution requires no support circuitry, saving critical real estate. E^{2}CMOS devices may be reprogrammed up to 1,000 times, which is sufficient for virtually all in-system-programmable applications except those requiring dynamic or continuous reprogramming.

After device technology, you need to consider programming requirements. Some in-system-programmable devices require a special 12 to 14-V supply to program the device. A board with such devices must incorporate extra control circuitry and must provide this special voltage with an extra power supply such as a dc-to-dc converter. These requirements add up to lost board space with lower reliability, higher power consumption and, ultimately, higher design costs. Other in-system-programmable devices in both SRAM and E^{2}CMOS technologies use a standard 5-V logic supply voltage for programming and reprogramming. This key feature will ultimately lower system design costs and power consumption.


Manufacturing flow using programmable devices
STANDARD FLOW USING NON-IN-SYSTEM-PROGRAMMABLE DEVICES
STREAMLINED IN-SYSTEM-PROGRAMMABLE FLOW

Using in-system-programmable devices, companies can bypass separate programming, marking, and inventory steps, thereby simplifying the manufacturing flow.

Programming interface options

The next consideration is the simplicity of the programming interface. A simple programming interface conserves board real estate and minimizes layout complexity and design costs compared to complex connectors or programming pad configurations. The key distinction you should consider is whether the interface is serial or parallel. Parallel programming options require extra layout resources for running interface signals across the board. These extra resources inevitably lead to higher design costs. In-system-programmable devices with serial interfaces usually require far fewer interface signals, making them more reliable and cost-effective, as well as easier to design into the system. Serial-interface in-system-programmable devices are available in both SRAM and E^{2}CMOS technologies.

The final consideration concerns the use of in-system-programmable devices as test resources. If test capabilities are a critical design consideration, you should look for devices with interfaces compatible with the IEEE 1149.1 boundary-scan standard.

In the optimal boundary-scan, in-system-programmable solution, the in-system-programmable and boundary-scan signals share the same dedicated pins. This enables a single interface and the use of identical pins to implement both board test and device reconfiguration.

The need for boundary scan is becoming ever more evident as more components per square inch are packed onto every board. Boundary scan significantly increases test coverage for design errors at the board-test level before they become expensive system-level test problems. Using an in-system-programmable device that isn't compliant with boundary scan offsets this advantage, increasing the risk of higher product-failure rates due to poor testability.

Candidate applications

As systems become denser and more highly integrated, in-system programmability becomes a more critical technology. This is because programmable-logic die is less accessible, making conventional programmer technology inefficient. This situation is especially true for multi-chip modules (MCMs).

In-system programmability offers advantages for many design and system challenges, but not all. This technology is usually not appropriate for extremely high-volume or very cost-sensitive designs because even the minimal additional costs associated with in-system-programmable board overhead may become prohibitive. Products in the early stages of their life cycles, including high-volume applications, are excellent candidates for in-system-programmable devices because they often need numerous changes before they are committed to inflexible ASIC chip sets, for example. A large percentage of new designs can benefit from in-system programmability in production, field upgrades, and generic functionality. And engineers are continually finding more innovative uses for in-system-programmable devices.

More mature products that do not need to be changed during manufacturing or in the field, or only serve one function, however, may not need the benefits that in-system-programmable devices offer because by their very nature they do not require many engineering change orders.

Multi-chip module packaging has perhaps the most critical need for in-system programmability. As demonstrated earlier, in-system programmability streamlines design processes using delicate fine-lead packaging and increasingly dense circuit boards. It also contributes to a more complete test strategy. With the higher integration and intricacies of MCMs, each of these advantages becomes more significant. Conventional non-in-system-programmable technologies require the die to be removed from the module, a practice that often damages the MCM. With in-system programmability this problem is nonexistent.

Test is another point of contention with MCMs as the modules cannot be tested with conventional techniques. Using an in-system-programmable design strategy that incorporates boundary scan will enable almost full testability of an MCM. The designer may fully develop the MCM package and attach the die before the design is fully tested. When errors are found, the design can be reprogrammed inside the MCM with a very short turn-around time.

Optimized development cycle

The use of in-system-programmable devices in the engineering lab can greatly reduce the design/debug cycle as well as prototype-development time. In a normal design cycle, an engineer generally designs the circuit, has a circuit board built, and begins debugging the board. In most circuits, the engineer corrects the inevitable errors and implements specification and design changes by physically removing socketed PLDs and inserting updated versions. He may have to modify the circuit board's traces and layout to accommodate resulting design and pinout changes.

When using in-system-programmable devices, design changes that previously took a half day of cuts and jumpers on the prototype board take just minutes. You make changes to the logic equations within the PLD and send the new programming to the PLD on the circuit board through the programming interface. This quick design-turnaround time can save you days to weeks within a development schedule.

Since in-system-programmable devices eliminate the need for prototyping sockets, there's no need to redesign the prototype board for production. Because the prototype and production boards may be identical, their capacitance and inductance, and, therefore, their ac performance,

Programmable logic parts with in-system programming capabilities continue to grow in popularity as more board designs adopt PQFP, TQFP and other fine-lead packages. Frank Morris, Valerie Young and Brian Reilly implemented the logic on NEC America's digital loop carrier board with four in-system programmable devices from Lattice to ensure lead conformance and overall product quality.

may also be identical. This eliminates unpleasant surprises in product performance when the first production units are manufactured.

The opening Convex example points out another benefit: A single circuit board may be designed for multiple uses. This technique is also known as reconfigurable hardware. With reconfigurable hardware, an engineer can design and build a product and then reconfigure it to accommodate any number of industry standards or product features. A generic PC card could be customized, for instance, to work with different network protocols.

Programmable Logic Technology Comparison

	E^2CMOS	SRAM	Antifuse	UV EPROM
Non-volatile	Yes	No	Yes	Yes
Single-chip solution	Yes	No	Yes	Yes
Reprogrammability	Yes	Yes	No	Yes (slow)
In-system programmability	Yes	Yes	No	No
Program time	Fast	Fast	Slow	Fast
Erasure time	Fast	Fast	N/A (one-time programmable)	Slow
Testability	Full	Full	Limited	Limited
Yield	100%	100%	93-97%	98-99%

Optimized production flow

Using typical PLDs, the production flow consists of taking blank parts from inventory, programming and marking them, sending each part back to inventory with a specific part number, then pulling the appropriate part number as needed to assemble the production cards. Programming a high-pin-count PLD is problematic because its fine pin pitch makes it incompatible with automatic programmer handlers. Consequently, production personnel must program all conventional high-density PLD devices manually. An operator has to place and remove each device from the socket on the programmer, a task that's very difficult to accomplish without severely bending the fine leads or destroying lead coplanarity.

A product using in-system-programmable devices enjoys a much simplified manufacturing flow, leading to higher quality and more accurate prototypes. First, as thin quad flat packs, plastic quad flat packs and other fragile chip packaging gain in popularity, manufacturing and assembly of circuit boards with sockets can become a serious quality-control problem. The additional handling steps for programming often leads to bent pins and lower part-utilization rates. With an in-system-programmable design strategy, inventory hassles are significantly reduced because the parts go directly from the receiving dock to placement on the printed circuit board, eliminating the standalone programming and mark operation entirely. In addition, multiple, blank in-system-programmable devices can be loaded into auto-insertion equipment and placed directly onto the board without sockets and without regard for which pattern goes into a particular board location. During final circuit-board test, the individual logic patterns are programmed into each device using the board-test station via the simple in-system-programmable programming interface.

Many products also require field upgrades to maintain their accuracy or to update their functionality. Instrumentation equipment is a good example because it often requires recalibration over a period of months or years to maintain accuracy and precision. With in-system-programmable parts embedded in the system and an appropriate interface to the programming pins, a field-service technician or even a customer can upgrade a product's hardware as easily as software upgrades are distributed today.

System reliability issues

Socketing of programmable devices can be a consistent source of system-reliability problems. By removing sockets from the circuit board and soldering an in-system-programmable device directly onto the board, the signal integrity of the leads isn't compromised by the socket connections. In addition, properly soldered joints are much more reliable than socket connections. These benefits result in greater system performance and overall reliability in terms of MTBF.

The use of in-system-programmable devices also lets the test engineer develop more flexible circuit-board-test procedures. For example, a test engineer can program in-system-programmable devices to interconnect the circuit-board traces and so achieve nearly full fault coverage of those traces. He or she can then quickly reconfigure the in-system-programmable devices to generate test signals for exercising dedicated logic devices on the board. By doing so, test engineers can significantly enhance the fault coverage of the board and the speed of tests. After board test, the test engineer can program the final logic patterns into the in-system-programmable devices.

Design engineers can use in-system-programmable devices to design boards with programmable configuration options, instead of dip switches or component swapping. This multiple-configuration approach can greatly improve system-level performance and reliability by reducing device counts, eliminating the need for sockets and improving testability.

The Lattice isp solution

After considering in-system programmable benefits, you can evaluate various device families for features that match their system requirements. Lattice Semiconductor, for example, fields three in-system-programmable LSI (ispLSI) device families based on the company's proprietary E^2CMOS technology: the flagship ispLSI 1000 family and the recently announced ispLSI 2000 and ispLSI 3000 families.

The ispLSI 1000 family, introduced in 1991, was the first available E^2CMOS in-system-programmable solution on the market to offer 2,000 to 8,000 gates and up to 110-MHz speed. The ispLSI 2000 family expanded upon the ispLSI 1000 family architecture to deliver speeds of up to 135 MHz. The ispLSI 3000 family offers device densities of up to 14,000 gates with 110-MHz speed and IEEE 1149.1 boundary-scan test capabilities.

The Lattice ispLSI devices follow a simple reprogramming scheme. Five pins are dedicated to in-system programming: serial data in (SDI), serial data out (SDO), mode control (Mode), serial clock (SCLK) and isp enable (ispEN). During the reprogramming operation, ispEN is asserted low, the four remaining ispLSI pins become active, and all other output pins become three-stated to prevent any bus contention during the reprogramming cycle. The programming of the device is then controlled by an internal state machine that's operated by using the SDI and Mode pins. You would use the design software provided by Lattice on a workstation or PC to serially load a 5-bit command into the device, followed by the design file in JEDEC format, all using a 5-V reprogramming voltage. Lattice also offers a software routine called ispCODE which gives you pre-written working C routines that can be incorporated in a system processor as part of the working system code.

The ability to design one product for multiple uses is perhaps the most exotic benefit of in-system-programmable technology, but it isn't the only one. Designing in-system-programmable devices into a product can improve productivity and reduce costs across the life cycle of a product.

Lattice's ispLSI 3000 family of devices share the isp programming signals with the standard boundary-scan signals, enabling the same interface to do both board test and logic reconfiguration.

An isp design example

Brian Reilly of NEC America (Hillsboro, OR) found an application that may not have been completed without ispLSI devices. Reilly's task was to design new circuit boards for an NEC digital loop carrier that accepts 96 phone pairs and digitally compresses them down to eight pairs. The boards were to be part of the system's common control unit and consisted of a 68020-based control CPU board and a custom high-capacity, serial-interface board. NEC encountered a set of engineering problems: trying to successfully implement the functional requirements which included the need for non-volatility, maximizing the amount of logic in the smallest amount of board real estate, minimizing board- and system-test costs, maintaining high product reliability and minimizing board rework caused by engineering change orders.

While all these constraints pointed to in-system programmability, the criterion that made in-system programmability unavoidable was the need for NEC to start building the hardware before the logic was fully designed. By implementing the design with Lattice ispLSI devices, NEC was able to meet a very tight product-development cycle, one which would have been impossible without an in-system-programmable strategy. Board layout was finished, and assembly took place weeks before the final logic was completed and implemented into the devices. Looking back on the experience, Reilly says, "The ability to change the logic on the board really saved our bacon." ❐

In-System Programmable Logic in High Volume Manufacturing

This paper was presented at the 1993 PLD Design Conference and Exhibit in Santa Clara, California and appeared in the Conference Proceedings, Track 2.

In System Programmable Logic in High Volume Manufacturing

Introduction

With systems and PC boards continuing to decrease in size with increased logic functionality, combined with the high integration levels of today's logic devices, there has never been greater pressure on board level testability. Traditional board test methodologies are no longer adequate for today's highly integrated systems. Several IC manufactures are attempting to address this problem by supplying devices which actually aid the test engineer in their testing of the board.

The challenge for today's design and test engineers is to design in a comprehensive board test methodology while at the same time reduce the cost of test fixturing. As the PC board becomes more and more complex it becomes harder and more expensive to have a bed of nails test to test each portion of the logic on the board.

Help comes from an unlikely source, In-system programmable logic or ISP HDPLDs. In system programmable logic can aid in virtually every stage of the product design and manufacturing cycle, up to and including installation at the customer. However this paper will focus specifically on the high volume manufacturing and testability areas.

In System Programmability (ISP), the ability to program and reprogram logic devices while "in-system". This concept is being pioneered primarily with High-Density PLDs (hereafter referred to collectively as HDPLDs).

ISP is revolutionizing the system designs of the 90's. ISP is an enabling technology that allows designers to define and develop systems with capabilities previously unachievable. With ISP technology, Virtual Hardware, the concept of hardware as flexible and easy to modify as software, becomes a reality. Hardware functions can be programmed and modified real time to expand product features, shorten system design and debug, simplify field upgrades, and perhaps most importantly, enhance product testability.

Technology Overview

The HDPLDs available on the market today can be categorized into four different and distinct CMOS technologies; Anti-fuse, SRAM, EPROM (UVCMOS) and E^2PROM (E^2CMOS). Of these four technologies. only three offer reprogrammability.

Of the three reprogrammable CMOS technologies, only SRAM and E^2CMOS provide in-system reprogrammability. UVCMOS can only be reprogrammed after the device has been erased by exposure to UV light (up to 20 minutes erasure time). The following manufactures offer in-system reprogrammability: Lattice Xilinx, AT&T and Concurrent.

In-system programmable and reprogrammable devices can be programmed, erased and reprogrammed while soldered directly to the printed circuit board (PCB). The actual implementation of ISP defers slightly between manufactures but the major concepts are the same. In circuit reprogrammable logic devices program and reprogram using a single 5 Vdc supply and either a serial or parallel programming interface for the loading and programming of binary bit patterns (JEDEC files). Conversely standard programmable logic devices require a super voltage (typically over 12 volts) to be applied to program and erase.

Reconfigurability for Test

Testability

Device board level testability is becoming the limiting factor in the high-tech manufacturing arena, the success or failure of a state-of-the-art product often depends upon the time required to build that product. In the case of products incorporating dedicated microprocessors, data transmission circuitry, or other complex electronic hardware, most of the time-to-build is consumed by testing and integration.

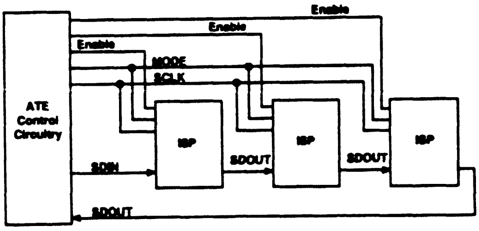
Advances in packaging technology have allowed the development of smaller and more dependable carriers, and have facilitated the onset of extremely high density, "lights-out", automated manufacturing. Advances in a number of interrelated areas (such as IR soldering, pick-and-place, adhesives, sensor technology,etc.) have opened the way to high density assembly techniques that would have been considered impossible only a few years ago. Although not commonplace, some size and/or weight critical products are currently built using a number of unique bulk-reducing construction techniques. Assembly processes which effect double-sided surface mount, chip-and-wire with epoxy cover, dense pack SIPs, or sandwich-mounted flat-packs are all valid means of producing a smaller, lighter, and more reliable final product. Unfortunately, highly advanced high density products are, at best, painfully difficult to test, and can be utterly impractical to repair.

In the manufacturing environment, where replacement parts, known good (golden) prototypes, and sophisticated test equipment are available, the major bottleneck to testing is related to circuit access. Quite simply, many device packages and the boards or sub-assemblies in which these packages are incorporated have few to no internal access paths. This is due, in some cases, to the density and placement of device-to-board interconnects (i.e., adjacent, stacked, double-sided, sandwiched, or epoxied surface mount...). In other cases it can be a result of package pin-count limitations (de-rated LCC,PLCC, pin-grid, or ceramic hybrid packages...). In rare instances, it can even be due to manufacturing or marketing concerns (sync, async, or mixed state machines with secured control patterns and/or "trap" states...). It becomes literally impractical to attempt to drive or receive test signals to or from a unit under test (UUT) from any physical location other than the designed-in system-level contact points.

This situation often renders externally applied test solutions inefficient or unreliable.The consequences, to automated test, are clear. Feedback loops cannot be eliminated. Test (stimulus) vectors cannot be inserted. Response vectors cannot be captured. Digital logic "lumps" cannot be simplified. State machines cannot be reduced. Asynchronous control paths cannot be opened or manipulated. By default, the product or sub-assembly will be verified via some form of BIST (built-in self test) or functional (power-up) diagnostics or may not be tested at all. BIST, even when implemented during the early stages of a design, it is expensive in terms of design time, real estate, and parts count. Functional diagnostics (F-diags), at either the system or sub-assembly level, are expensive in terms of development and support. Non-test is expensive in terms of raw yield. None of these solutions are either comprehensive or comfortable. As circuit complexity continues to increase, driven by the synergetic advances in electronic technology mentioned above, the bottleneck(s) associated with high volume manufacture cannot but worsen. Existing approaches to BIST, DFT (Design For Test), and F-diags, will become more expensive and less effective. Current UUT access techniques will become more costly and less reliable. Only a radical departure from traditional automated test practices and procedures will be able to provide a cost effective solution to tomorrow's test, diagnostic, and verification challenges.

Help comes to the test and manufacturing community from an unlikely source, in system programmable high density logic devices. The ISP approach to BIST was to add serial networking capability to the PCB. With the addition of ISP HDPLDs, independent synchronous serial port capability can help uncover hidden logic. The result is an entirely new approach to testability, RFT (Reconfiguration For Test). Like the JTAG-MSDS 4-wire serial interface (a.k.a. IEEE P1149.1 proposal), the ISP protocol defines a clock input, a serial data input, a mode/control input, and a serial data output.

The serial output of one device can be connected to the serial data input of another, thus allowing an unlimited number of devices to be cascaded.



(NOTE: The ISP serial communication protocol is NOT JTAG-MSDS compatible! ISP devices do not incorporate compliant JTAG-MSDS instruction, data, or identification registers.) Also like the JTAG-MSDS 4-wire serial interface, the ISP protocol can be visualized as an on-board, synchronous, serial FIFO (First-In, First-Out), ring style, local-area-network for devices. An off-board host (or communication master) can communicate with any device on the ring by noting its' relative position and by appropriately transmitting data through the FIFO loop. An ISP device can be completely erased,completely examined, and completely repatterned via the ISP protocol. No unusual voltages or control signals (beyond the signals necessary to exercise the FIFO) are required.

All the voltage/current sources, timing control circuitry, pulse generators, and algorithmic state generators are built into the device itself (EECMOS). This capability allows ISP based hardware to be modified, or reconfigured, at any point in the life cycle of any given unit or sub-assembly. Thus an extremely complex circuit, composed of interlocked state machines of different types, can be reconfigured into a simpler, more easily tested, configuration. In the factory, feedback loops can be re-routed or eliminated. Latches and/or memory elements can be reduced or isolated. Signal paths into the core of an inaccessible "lump" of hardware can be opened. Test vectors can be introduced without back-drive. In brief, all of the testability problems associated with HDPLD based designs which include feedback or asynchronous paths,can be addressed without the need for physical access to an implementation.

How ISP can solve potential test problems, let us look at the manufacturing needs of a large-volume,medium scale, high speed digital logic design. For the sake of argument,we will assume that all DFT (design for testability) SSI and MSI functions have been implemented using ISP HDPLD devices.This means that all the ATE/ATP (automatic test equipment/automatic test procedure) defeating nuances of a typical small to medium scale logic design can be circumvented. We will further assume that this is a consumer application, and that in-system logic reconfiguration, field service access, and design security are not an issue. This means that the ISP protocol interface(s) can be brought to a spring pin (pogo-pin) compatible pad array on the board or substrate, and that electrical access can be achieved via a typical vacuum or clamp type fixture. To be brief, we will describe the design application as a "board". In reality this application could as well be any type of module or assembly incorporating logic devices. With any PCB or circuit assembly there is a quantifiable level of testability that is achievable, which is dependent on the complexity of the board design.

In an attempt to quantify this uncertainty, a board is typically assigned a "fault-coverage" rating. Good (highly testable) boards are rated at 70% to 99%. Bad (difficult to test) boards are rated at 70% and under. The rating is usually described as "stuck-at fault coverage".This is due to the theory that any function-inhibiting failure can be traced to a "stuck" node or equivalent. Given enough time and effort, anything can be tested well. The reason many boards receive a low fault-coverage rating is because:

(1) The resources necessary to improve the rating by creating a better test exceed the anticipated savings which will be realized throughout the product's life cycle by repairing defective boards identified by the better test.

(2) The resources necessary to perform an improved test on a given lot of boards exceeds the anticipated savings which will be realized by repairing defective boards identified by the improved test.

NOTE: This analysis often does not include the cost to salvage defective boards identified by the customer or by the end-user.

Boards which fail in service must often be replaced or repaired at any cost. Thus, in the case of an unusually long product life cycle, there may be ongoing and increasing pressure to improve the fault-coverage of a given board or sub-assembly due to a high field failure rate. Statistically, the logic circuit configurations which most often cause a loss of confidence in the functionality of a new-board are typified in the implementation of a multiple-device state-machine.

The design function of an ISP device is to permit the device to be repeatedly reprogrammed, via the programming interface, after permanent installation. This provides two important secondary capabilities which, as a side-effect, eliminate the loss of confidence which HDPLDs, programmed and arranged as multiple-device state-machines, normally introduce into a logic design. Essentially, any multiple-device state-machine, or any logic circuit which incorporates DFT violations normally associated with a multiple-device state-machine, can be effectively tested if the components have an ISP compatible interface.

ISP HDPLDs have the capability of being externally interconnected (SDOUT to SDIN). This allows the ISP device to be configured in a serial cascadeable arrangement. The ISP "daisy chain" allows all ISP compatible devices to be verified independent of the function or placement of any given device. The ISP protocol interface is a very simple, low speed, synchronousring, which can be quickly and easily verified by ATE or by entry-level test personnel. Given a verified "daisy chain", confidence in the functionality of the individual devices in the chain approaches 90%. Thus the devices which were previously the most difficult and expensive to verify have become the easiest and most cost effective to verify.

Once the chain is verified. any or all of the devices in the chain can be reprogrammed with special self-test patterns. or may be reprogrammed to accept ATE originated stimuli or to drive ATE receivers.

Given a "daisy chain" which has accepted a stimulus and generated a response which does NOT depend upon the design function(s) of any adjacent non-ISP devices, confidence in the functionality of the individual devices in the chain exceeds 90%. Thus the portions of a design which previously exhibited the worst fault-coverage rating now exhibit the best rating.

A verified ISP "daisy chain" greatly facilitates access to and verification of adjacent non-ISP devices and modules. An entire ISP chain or any portion or device may be programmed such that all inputs and outputs remain in a high impedance state whenever the device is powered-up or put into the programming mode. This capability allows a board to be divided into small "lumps" of logic which tremendously reduces the resources required to generate test software. A major cause of new-board mortality, excessive back-drive current, can be completely eliminated via the thoughtful placement of ISP compatible devices by an RFT (reconfiguration for test) conscious design engineer. In the event of an unusually difficult to test "lump", an ISP chain can be programmed to serve as a source of elementary test stimuli. Simple counters, decoders. ATE controlled enable/disable signals, ATE controlled read/write signals, and the like, can be reprogrammed into an ISP chain via the ISP interface.

This new functionality can be entirely dedicated to an intermediate step in the test process for a new-board, and the chain later returned to its primary function after the non-ISP portions of the new-board are satisfactorily verified. This capability, is referred to as: Reconfiguration For Test (RFT)

Unfortunately, such practices extract a high price in terms of production costs (extra gates, solder holes, board area, etceteras) and in terms of performance (5ns to 30ns per ATE controllable gate. infant mortality due to backdrive related stress). In the absence of traditional DFT, the price extracted is in terms of fault coverage and diagnostic engineering resources.These problems can all be solved by using ISP protocol devices where ever feedback signals need to be generated or interpreted. RFT allows feedback loops to be opened. eliminated,or tied to a test node. Given an unused input and an unused output on an ISP protocol device used to generate a feedback signal, that signal can be routed to the unused output where it may be sensed by ATE without influencing the UUT. Additionally, the unused input can be routed to the portion of logic which is normally driven by the feedback. Thus the ATE itself is made into a series component in the asynchronous feedback loop. Once the feedback signal has been examined by the ATE, a replica can be created and driven back into the logic normally driven by the feedback signal.

This serves the intent of the traditional DFT requirement for physical interruption of asynchronous feedback loops, but without the need for switches or jumpers. It allows an electrical interruption in the feedback circuit, but without propagation delays due to extra gates, and without potential stress failures due to excessive backdrive. Overall, the use of ISP devices in critical asynchronous feedback circuits allows:

(1) A reduction in time required to achieve acceptable fault coverage;
(2) A reduction in resources required to achieve acceptable fault coverage;
(3) No performance penalties;
(4) No backdrive overstress;
(5) Minimal additional hardware;
(6) Full DFT compliant loop control and interruption.

Reduction of tight hardware kernels In any system design there are invariably certain sections or modules which are uncompromisingly speed critical. An example of such a speed critical logic module or sub-module would be the address decode and access arbitration circuitry for a multiple-port cache RAM bank. Fundamentally, the response time of such a circuit is so critical that no allowance can be made for added functionality or for control which does not enhance the primary design goal or which, at a minimum, incurs no performance overhead. This includes any circuitry which might facilitate testability. As a rule, such circuits constitute less than 20% of a typical system's real estate and consume more than 80% of the diagnostic resources applied to that particular system. This is not a comfortable situation,but until recently the economics which evaluate the return on diagnostic-related expenditures (versus the life-cycle of a system or viability of a manufacturing process) have mandated this style of diagnostic resource allocation.The usual approach to testing such circuitry (since it is known in advance that some compromise in both fault coverage and fault isolation will have to be made), is to attempt to model the group of devices that make up the performance critical portions of the system as though they were a single MSI or LSI full custom component.

This requires considerable ingenuity on the part of the diagnostic engineer since he/she must generate a complete set of test software/patterns for this pseudo-device as though it were an outside vendor's unsupported new product.This approach can be tremendously improved via the use of ISP HDPLDs to implement the performance critical portions (and therefore the least ATE accessible portions) of any given system. Since ISP HDPLDs are state-of-the-art CMOS programmable logic devices, there is no performance penalty. A "threaded signal" performance critical logic circuit can be completely repatterned, many times,during the performance of an ATE program, to allow the ISP protocol devices to be fully tested. Also, given a high enough percentage of ISP protocol devices in any group of devices which make up a performance critical design, even non-ISP device testability can be improved by using the ISP HDPDSs to implement ATE access paths during the execution of an ATE program.

Although more diagnostic resources (testability guru time) are required to implement an ATE program which includes ISP device RFT control array patterns, the overall savings in time required to test a tight, performance critical hardware module, is actually reduced. This allows increased fault coverage and greatly increased fault isolation, but with an overall reduced demand for resources.Trade-offs can be made which allow greatly increased testability without the expenditure of extra resources, or which provide a net savings in diagnostic resource utilization without any testability loss.

Using ISP to increase visibility into the board in the previous RFT discussions, little mention has been made regarding the tremendous opportunities offered by ISP protocol devices for increased visibility into a new-board during the ATE/ATP (automatic test equipment/automatic test procedure) portions of the manufacturing process. Points mentioned earlier deal mainly with single function, lumped logic modules, constructed partially or even entirely of ISP protocol devices. While this perspective encourages increased fault coverage and better fault isolation of performance critical circuits and/or multiple-device state-machines via the introduction of RFT principles, it does not adequately describe the benefits available by using ISP protocol devices to partition an entire new-board or system for the purpose of enhancing system-level testability.

Most system-level designs incorporate a variety of special-purpose devices (example: RAMs, ROMs, UARTs, controllers, processors, etc.) which are nestled among, and interconnected by, a large quantity of general purpose ("glue" chips) devices . In the last few years, PLDs and HDPLDs have come to replace many of the older SSI and MSI "glue" devices. But for most practical purposes, a HDPLD can be conceptually classified as a multiple SSI and MSI devices breadboard in a package. From a testability viewpoint, HDPLDs offer a reduction in "glue"device package count, but not in logic complexity or in density. In an average large-volume, medium scale, high speed digital logic design, HDPLDs will often account for 20% or more of the logic device package count. HDPLDs, in this type of application, are not famous for improving either the fault coverage or the fault isolation of a logic design. ISP HDPLDs, however, offer exactly this benefit. Since ISP HDPLDs can be serially reprogrammed, it is possible to verify the correct operation of a ISP device's internal logic CA (control array, or fuse map) via programming interface "daisy chain". Verification of any given device's CA implies a very high probability that the entire device is functional. Once an entire chain has been verified, this string of reprogrammable logic devices, extending into a logic design, can be used to improve the visibility into a new-board.

The simplest means of improving visibility is to use the ISP HDPLDs as test signal routing switches. Given a single ATE accessible input to an ISP device, any or all of the device's outputs may be programmed to track that input. Likewise, given a single ATE accessible output, any or all of the device's inputs may be programmed to be tracked. If the ISP HDPLDs have interconnected inputs and outputs, test signals can be routed in and out through any number of discrete ISP devices. (Note: This type of testability enhancement generally requires that a design be implemented with RFT in mind). More complex, is the use of a verified ISP chain to generate simple algorithmic test patterns for the stimulation of non-ISP devices down-stream. Since most HDPLDs perform very well as sequential synchronous state-machines, a device or series of devices with one or more ATE accessible inputs can be programmed to generate a deterministic output pattern in response to any combination of ATE generated clock or data signals.
In other words, an ISP HDPLD can be used as part of an ATE/ATP test solution for difficult-to-access non-ISP portions of a logic design. (Note: This type of testability enhancement always requires that a design

be implemented with RFT in mind.) Still more complex, is the use of a verified ISP chain to capture simple synchronous response signals from non-ISP devices up-stream. Because registered outputs work very well as serial shift registers, a device or series of devices with one or more ATE accessible outputs can be programmed to capture several sequential logic values sensed by any or all of the device's inputs. In other words, an ISP HDPLD can be used to capture the results of several successive test patterns and can even perform elementary processing to facilitate signature analysis. Ultimately, a multi-function (generic) system can be designed such that one of the target operations for the design is to allow some form of ATE to thoroughly exercise a thorough subset of all possible logical functions. Such a system would be flexible (due to RFT capabilities) enough to allow every device-external signal conductor to be individually exercised.

The ATE should be able to apply a combination of random and tailored test vectors to any device or multiple-device hardware kernel and to sense the subsequent responses via any signal conductor. This type of testability enhancement would utilize ISP HDPLDs as though the ISP devices themselves made up a serially accessible distributed test processor, and as though the logic CA patterns for a chain of ISP devices acted like individual test processor instructions.

We have seen how ISP HDPLDs can help in the testability of complex logic boards, however there are still several others areas in the product life cycle that also benefit from ISP.

In System Reprogrammability at the Prototype Stage

During any system design cycle, major board building blocks such as microprocessor and RAM are selected first. Decisions regarding system logic tend to be deferred to the later stages of the design process. When using ISP devices, the designer can fully populate his prototype board with its major building blocks, interconnecting all functions with programmable logic. Design changes, whether they require added or modified logic, can be made in minutes using ISP HDPLDs.

Manufacturing Advantages

At present, there are no auto-handlers capable of handling the programming of the higher pin counts associated with today's HDPLDs. As a result, all non-ISP high pin count devices must be programmed by hand, using a standard logic programmer.

It is a non-trivial task to insert a high pin count, small lead pitch device into a programming socket adapter, program, label (or mark) and re-inventory the device without bending the delicate package leads or pins. Auto-inserting devices also increases the risk of exposing the devices to potential ESD environments.

Field Upgrades

ISP HDPLDs provide an ideal way to reconfigure boards and/or upgrade product features in the field. Using conventional logic technology, once a system is installed at a customer location, it becomes very expensive and difficult for the supplier to upgrade the customer to the latest hardware revision, fix hardware bugs or enable hardware options.

HDPLD Device Security

Most ISP HDPLD devices, even though programmed on board, can still assert the security feature eliminating the risk of the JEDEC pattern being read out of the device. If the ISP requires a new or updated JEDEC pattern, the device is erased (which is done automatically before the programming), a new pattern is programmed into the device and the security cell is reasserted.

Conclusion

The challenge for today's design and test engineers is to design in a comprehensive board test methodology while at the same time reduce the cost of test fixturing. As the PC board becomes more and more complex, it becomes harder and more expensive to have a bed of mails test to test each portion of the logic on the board. Therefore the test engineer must now work hand-in-hand to find solutions to these problems. Fortunately, there are options available like ISP HDPLDs to assist in the debug and manufacturing test of complex PCBs.

Higher product quality and reliability can result from the superior test coverage ISP offers. Special test logic can be programmed temporarily into the hardware to facilitate exhaustive product testing. The elimination of defects at an early stage of board check-out reduces more expensive system-level failures later in the final manufacturing process.

ISP HDPLDs are opening doors of opportunity in almost every facet of systems design and test.

Notes

ispLSI Configurable Memory Controller

Introduction

There are many advantages of using the in-system programmable ispLSI devices. In board level designs, as well as during manufacturing, the flexibility of hardware reconfiguration can lead to many innovative system designs. Once configured, the ispLSI devices' non-volatile E^{2}CMOS cells will retain their configuration even when the power is turned off. The guaranteed 1,000 programming cycles and 20 year data retention of the ispLSI device will allow the user to reliably reconfigure the device as often as required.

This application note highlights the advantages of designing with ispLSI devices and how they can lead to innovative design ideas which translate to ease of use and instant updates without board layout changes. The flexibility of design is illustrated with the use of the Dynamic Random Access Memory (DRAM) controller. This example shows a typical microprocessor and memory interface with the memory controller controlling the DRAM access and refresh timing requirements. The use of Lattice pLSI and ispLSI Development System (pDS) Software is also illustrated in this application note. The Lattice Design Fiie (.ldf) listing file generated by the software is also attached at the end of this section.

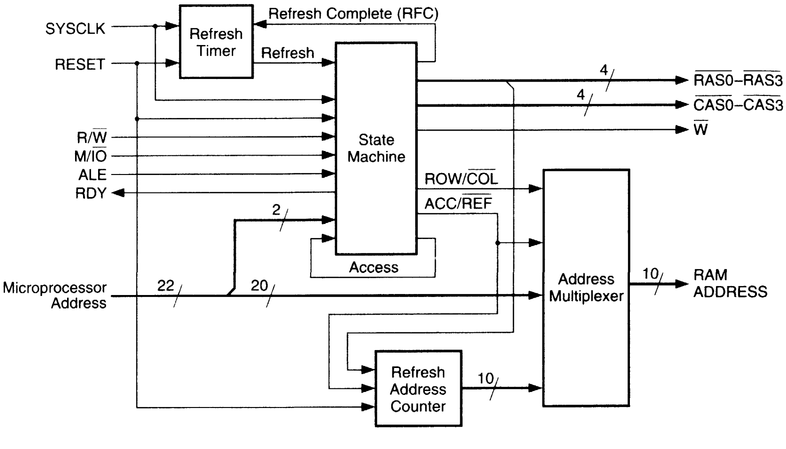
Memory Controller Logic Overview

When interfacing the microprocessor to the DRAM, the control signal and timing requirements of both the processor and the DRAM must be satisfied. In order to satisfy these requirements, the external timing controller must take the processor address, data and control signals and translate them into the control signals for the DRAM. At the same time, the DRAM timing controller must take into account the refresh requirements of the DRAM.

Figure 1 shows the block diagram of the DRAM timing controller that is implemented in the ispLSI 1032. The state machine and address multiplexer blocks are used to control the memory access request of the processor and supply the DRAM with the necessary address and control signals. DRAM refresh requirements are controlled by the refresh timer block, refresh address counter block and the address multiplexer block.

Any access request from the processor is processed by the state machine based on the processor control signals such as Read/Write (R/W), Memory/IO access (M/IO), Address Latch Enable (ALE) and the microprocessor address signals. The Ready (RDY) signal is used to inform the processor the status of the requested data. In other words, it is used to acknowledge the processor that the memory is ready to respond to the processor. The address multiplexer generates the row and column addresses necessary for the memory access cycle. The appropriate Row Address Strobe (RAS), Column Ad dress Strobe (CAS), and Write (W) signals are also generated by the state machine based on the processor

Figure 1. DRAM Timing Controller Block Diagram



inputs. To arbitrate between the memory access request and the refresh request, the state machine also generates the status signal called Access. The purpose of this signal is to keep track of an access cycle when the refresh sequence is in progress. This status signal is then used to determine whether or not to begin an access sequence after the refresh sequence. As part of the access/refresh arbitration, the state machine also issues an Access/ Refresh (ACC/REF) signal to the address multiplexer logic block. Based on this signal the address multiplexer block routes the appropriate access or refresh address on to the external DRAM address bus.

As for all DRAMs, memory refresh must be completed within a specified time. This process is completely controlled by the DRAM timing controller. The refresh timer block generates the internal refresh request signal according to the system clock speed and the DRAM refresh rate requirements. When the state machine detects this refresh request signal, the refresh sequence for the DRAM is generated as soon as time permits. This means that the refresh sequence is generated right after the refresh request or if the timing controller is in the middle of a memory access cycle the refresh sequence is generated right after the memory access cycle is complete. During the refresh sequence the row address and all the RAS signals must be activated to perform the basic RAS-only refresh. The row addresses are supplied by the refresh address counter logic block. This logic block keeps track of the rows that are being refreshed and it gets incremented every time a refresh sequence is performed. All the RAS signals are activated for refresh by the state machine.

With the basic understanding of the DRAM timing control logic complete, the next section will discuss the implementation of the logic in an ispLSI device and how to take advantage of the ISP features to make the system design, manufacturing and field updates easy and flexible.

Taking Advantage of ISP Features

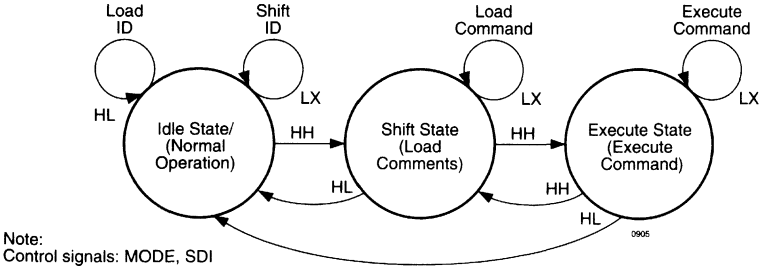
Implementing a basic DRAM timing control logic in the ispLSI 1032 takes up approximately 65% of the total logic available in the device. (It is with this in mind that the features needed for a specific design can be added to these basic logic blocks). With the ISP capability, many features can be added to accommodate the ever changing requirements of the system, microprocessor speeds, availability of DRAMs, and the memory configurations. Moreover, the changes are made only under the software control. Instead of having different production runs for various different options, the options are added at the in-system programming stage.

The programming of the ispLSI devices are handled via five TTL level interface signals. Of these five signals, four signals can be dual function, a programming function as well as an input during normal device operation. The ISP Enable (ispEN) signal is the one dedicated programming pin used to enable and disable the programming function. Once in programming mode, the mode control (MODE), serial data input (SDI), serial data clock (SCLK), and serial data output (SDO) signals control the entire programming process. The address and data required to program the device are serially shifted into the internal shift registers and the three state programming state machine steps through the programming sequence. The five-bit instructions within the state machine define all the necessary steps for programming. Figure 2 shows the ISP programming state machine with the control signal requirements for the state transitions. Refer to the "Hardware Basics" section in this manual for a more detailed programming description.

Different System Speed

Designing with a different speed microprocessor requires a different DRAM timing controller. The

Figure 2. ISP State Machine


Load ID
Shift ID
Load Command
Execute Command
HL
LX
LX
LX
Idle State/ (Normal Operation)
HH
Shift State (Load Comments)
HH
Execute State (Execute Command)
HL
HH
HL
0905

Note:
Control signals: MODE, SDI

adjustments must be made in the state machine and refresh timer logic of the controller to account for the difference in speed. Without the capabilities of the ISP features, different boards with different PLD codes must be built to work with different processor speeds. By providing a simple programming circuitry on board to support the ISP programming, the logic adjustments for different speed processor can be accomplished by in-system programming the different patterns via software control. Manufacture of these options are made simple and cost effective by not having to keep an inventory of prepatterned devices.

DRAM Feature Flexibility

DRAMs have many features from which the system designer can select. For the same DRAM configuration, the system designer can select from DRAMs that have different access schemes such as nibble mode, static column mode and page mode. Similarly, different memory refresh schemes can be chosen. The two choices of refresh schemes include the simple RAS only refresh and the option to perform hidden refresh with the CAS before RAS refresh scheme. Most of these various DRAM options can be supported by in-system programming the ispLSI devices. Again, the flexibility lies in the fact that the decision of what function the ispLSI will perform on board can be made after the decision has been made on which type of DRAMs are used on board.

Different DRAM Configuration

The ispLSI implementation of the DRAM timing controller makes the change of memory configuration very simple. Reprogramming of the address decoding and turning on the appropriate address strobe signals for different memory configuration can be done by in-system reconfiguration of the state machine and the address decoding of the ispLSI device. All of these changes can be accomplished under software control.

Memory Timing Controller Details

As shown in Figure 1 the memory timing controller consists of four different logic blocks. The refresh timer, state machine, refresh address counter and memory address multiplexer. All boolean equations for the logic blocks are developed within the Lattice pDS Software. The entire memory timing controller design assumes that all the processor signals are typical of a commercially available processor with a clock speed of 25 MHz. DRAMs are arranged in four banks of 1M X 32-bit arrangement. All timing for the access and refresh sequences are shown in the timing diagram.

Refresh Timer

The function of the refresh timer is to generate a refresh request signal every 15.5 µs. This refresh period is derived from the DRAM refresh requirement of 512 rows of refresh every 8 ms for the 1M X 1 DRAM. Based on the 25 MHz system clock frequency, the count value to divide the clock period to the refresh period is 200. Changing processor speed will only require a change of count value. Once the count value expires, the refresh timer generates an internal refresh signal to inform the state machine to perform a refresh cycle. When the state machine completes the refresh cycle, a refresh complete (RFC) signal is generated for the refresh timer. The refresh timer then resets the internal counter for the next refresh period.

ispLSI implementation of the refresh timer takes up three GLBs (A0-A2) within the device. The system clock is used to run the nine bit counter, RFC is the input signal to this block and REFRESH is the output signal of this logic block.

State Machine

The state machine can be further divided into four different sub-logic blocks. These sub-logic blocks consists of a RAS generator, CAS generator, 4-bit state machine which is divided into two state variable bits and two counter bits, and control signal generator. In the ispLSI 1032 implementation, the state machine logic block takes up 9 GLBs.

The 4-bit state machine is divided into a 2-bit state variable, named ST0 and ST1, and 2-bit state counter, named SCNT0 and SCNT1. The state diagram with its state transitions are shown in Figure 3. In each of the access and refresh states, the state counter sequences through the operation until the sequence is complete. The purpose of the state variable bits are only to keep track of the state transitions. Once the state transition has occurred, the state counter bits take the responsibility of sequencing through the state.

The three states are divided as idle state, access state and refresh state. Based on the processor control signal and the internal refresh request signal, the state transition occurs from idle state to either access state or refresh state. If the refresh and access request happen at the same time, refresh request takes precedence over access request. When the refresh request is asserted during an access cycle, the refresh cycle follows right after the access cycle. The only other condition between the access and refresh request that the state machine needs to arbitrate is when the access request occurs during the refresh sequence. The access feedback signal of the state machine is activated when the access request occurs during the refresh cycle. When the refresh cycle is complete, the access feedback signal is used to determine whether or not the access sequence needs to begin. The timing diagrams in Figure 4 and 5 illustrate the control signal sequence for the access and refresh cycles, respectively.

In addition to the external DRAM control signals, the state machine also generates the control signal for the address multiplexer and the refresh address counter. The ROW/COL signal directs the address multiplexer to output the appropriate row and column address during the access cycle. Furthermore, the address multiplexer accepts the access/refresh (ACC/REF) control signal to either direct the memory access address from the processor, or direct the refresh row address from the refresh address counter to the DRAM.

Figure 3. DRAM Timing Controller State Machine

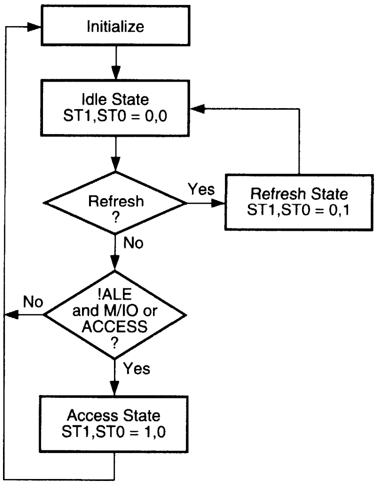
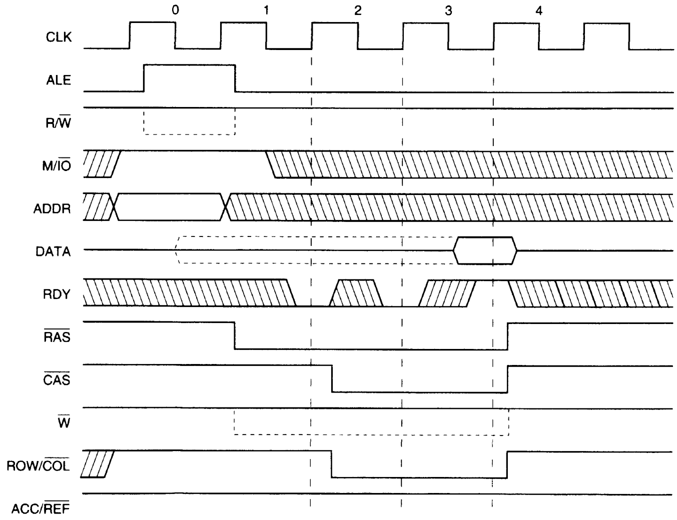


Figure 4. Access Cycle Timing



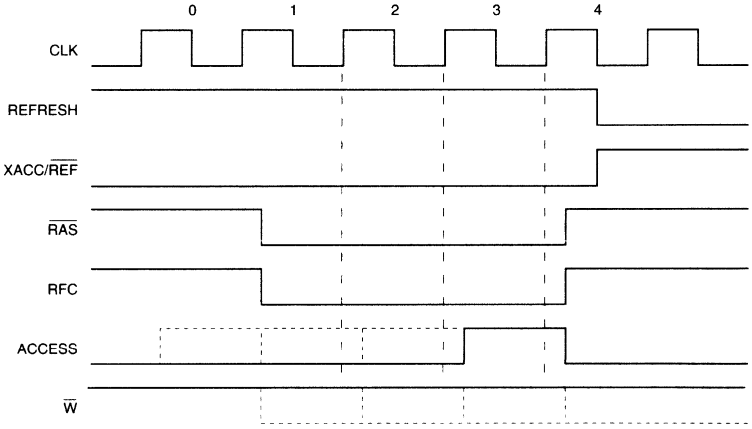
Refresh Address Counter

The refresh address counter keeps track of the rows of DRAM to be refreshed. This counter is only incremented on the falling edge of the RAS signal during refresh sequence. The ispLSI device implementation of this counter takes 3 GLBs.

Memory Address Multiplexer

In access mode, determined by the ACC/REF internal signal, the memory address multiplexer multiplexes between the row and column address. Once in the refresh cycle, the refresh address comes from the refresh address counter. It takes 3 GLBs to implement the multiplexer in the ispLSI 1032.

Figure 5. Refresh Cycle Timing



Conclusion

The intention of this application section is to give an overview of how the ISP features can be used to improve the design features and the manufacturing process by using an example of a generalized DRAM timing controller. In addition, the software example given in the document should provide a good starting point for designers who need to implement a state machine based design. With the flexibility of the ispLSI devices the possibilities are limited only by one's imagination to implement innovative designs. The following sections list the Lattice Design file with the Boolean Equations.

Design LDF Listing

```
//isp_app.ldf generated using Lattice pDS Version 2.50
LDF 1.00.00 DESIGNLDF;
DESIGN DRAM CONTROLLER 1.00;
PROJECTNAMEispAPPLICATIONS;
DESCRIPTION
DRAM CONTROLLER DESIGN FORispAPPLICATION.
IT INCLUDES FOUR MAJOR BLOCKS.
   - REFRESH TIMER
   - REFRESH ROW ADDRESS COUNTER
   - ADDRESS MUX
   - STATE MACHINE;

PART pLSI 1032-90LJ;

DECLARE
END; //DECLARE

SYM GLB  C2  1  ;
   /////    ROW ADDRESS STROBE (RAS1,RAS0) GLB      /////
   SIGTYPE IRAS1 REG OUT;
   SIGTYPE IRAS0 REG OUT;
   EQUATIONS
      IRAS1.CLK = ICLK;
      IRAS1 = !ST0 & !IA20 & IRAS1 & !IRESET    /////   REDUCED RAS1    /////
           # !ST1 & IA21 & IRAS1 & !IRESET
           # !ST0 &  ST1 & SCNT0 & SCNT1 & IA20 & !IA21 & !IRESET
           #  ST0 & !ST1 & SCNT0 & SCNT1 & !IRESET
           # !ST0 & !ST1 & IRAS1 & !IRESET
           #  ST0 & ST1 & IRAS1 & !IRESET
           #  SCNT1 & IRAS1 & !IRESET
           #  SCNT0 & IRAS1 & !IRESET;

      IRAS0 = !ST0 & IA20 & IRAS0 & !IRESET    /////   REDUCED RAS0    /////
           # !ST1 & IA21 & IRAS0 & !IRESET
           # !ST0 &  ST1 & SCNT0 & SCNT1 & !IA20 & !IA21 & !IRESET
           #  ST0 & !ST1 & SCNT0 & SCNT1 & !IRESET
           # !ST0 & !ST1 & IRAS0 & !IRESET
           #  ST0 & ST1 & IRAS2 & !IRESET
           #  SCNT1 & IRAS0 & !IRESET
           #  SCNT0 & IRAS0 & !IRESET;
   END
END;

SYM GLB  A2  1  ;
   /////    REFRESH TIMER GLB2     /////
   SIGTYPE RQ8 REG OUT;
   SIGTYPE RQ9 REG OUT;
   SIGTYPE REFRESH REG OUT;
   FJK11 (REFRESH,R_RATE,RFC,ICLK);  /////   REFRESH REQUEST SIGNAL   /////
   EQUATIONS
      RQ8.CLK = ICLK;
      RQ8 = (RQ8 & !RFC)
          $$ (RQ7 & RQ6 & RQ5 & RQ4 & RQ3 & RQ2 & RQ1 & RQ0 & !RFC);
      RQ9 = (RQ9 & !RFC)
          $$ (RQ8 & RQ7 & RQ6 & RQ5 & RQ4 & RQ3 & RQ2 & RQ1 & RQ0 & !RFC);
      R_RATE = RQ7 & RQ6 & !RQ5 & !RQ4 & RQ3 & !RQ2 & !RQ1 & !RQ0;
   END
END;
```

```
SYM GLB  A1  1  ;
   /////     REFRESH TIMER GLB1     /////
   SIGTYPE RQ4 REG OUT;
   SIGTYPE RQ5 REG OUT;
   SIGTYPE RQ6 REG OUT;
   SIGTYPE RQ7 REG OUT;
   EQUATIONS
      RQ4.CLK = ICLK;
      RQ4 = (RQ4 & !RFC)
          $$ (RQ3 & RQ2 & RQ1 & RQ0 & !RFC);
      RQ5 = (RQ5 & !RFC)
          $$ (RQ4 & RQ3 & RQ2 & RQ1 & RQ0 & !RFC);
      RQ6 = (RQ6 & !RFC)
          $$ (RQ5 & RQ4 & RQ3 & RQ2 & RQ1 & RQ0 & !RFC
      RQ7 = (RQ7 & !RFC)
          $$ (RQ6 & RQ5 & RQ4 & RQ3 & RQ2 & RQ1 & RQ0 & !RFC);
   END
END;

SYM GLB  A0  1  ;
   //////  REFRESH TIMER GLB0   /////
   SIGTYPE RQ0 REG OUT;
   SIGTYPE RQ1 REG OUT;
   SIGTYPE RQ2 REG OUT;
   SIGTYPE RQ3 REG OUT;
   EQUATIONS
      RQ0.CLK = ICLK;
      RQ0 = !RQ0 & !RFC;
      RQ1 = (RQ1 & !RFC)
          $$ (RQ0 & !RFC);
      RQ2 = (RQ2 & !RFC)
          $$ (RQ1 & RQ0 & !RFC);
      RQ3 = (RQ3 & !RFC)
          $$ (RQ2 & RQ1 & RQ0 & !RFC);
   END
END;

SYM GLB  D0  1  ;
   /////     ADDRESS MUX GLB0      /////
   SIGTYPE IRAM0 ASYNC OUT;
   SIGTYPE IRAM1 ASYNC OUT;
   SIGTYPE IRAM2 ASYNC OUT;
   SIGTYPE IRAM3 ASYNC OUT;
   EQUATIONS
      IRAM0 = ROW_COL & ACC_REF & IA0    /////  ROW SELECT   /////
         #  !ROW_COL & ACC_REF & IA10 /////  COLUMN SELECT   /////
         #  !ACC_REF & RCNTR0;             /////   REFRESH ADDR SELECT  /////
      IRAM1 = ROW_COL & ACC_REF & IA1
         #  !ROW_COL & ACC_REF & IA11
         #  !ACC_REF & RCNTR1;
      IRAM2 = ROW_COL & ACC_REF & IA2
         #  !ROW_COL & ACC_REF & IA12
         #  !ACC_REF & RCNTR2;
      IRAM3 = ROW_COL & ACC_REF & IA3
         #  !ROW_COL & ACC_REF & IA13
         #  !ACC_REF & RCNTR3;
   END
END;
```

```
SYM GLB  D1  1  ;
   /////    ADDRESS MUX GLB1      /////
   SIGTYPE IRAM4 ASYNC OUT;
   SIGTYPE IRAM5 ASYNC OUT;
   SIGTYPE IRAM6 ASYNC OUT;
   SIGTYPE IRAM7 ASYNC OUT;
   EQUATIONS
      IRAM4 = ROW_COL & ACC_REF & IA4    /////  ROW SELECT    /////
         #  !ROW_COL & ACC_REF & IA14 /////  COLUMN SELECT   /////
         #  !ACC_REF & RCNTR4;              /////   REFRESH ADDR SELECT  /////
      IRAM5 = ROW_COL & ACC_REF & IA5
         #  !ROW_COL & ACC_REF & IA15
         #  !ACC_REF & RCNTR5;
      IRAM6 = ROW_COL & ACC_REF & IA6
         #  !ROW_COL & ACC_REF & IA16
         #  !ACC_REF & RCNTR6;
      IRAM7 = ROW_COL & ACC_REF & IA7
         #  !ROW_COL & ACC_REF & IA17
         #  !ACC_REF & RCNTR7;
   END
END;

SYM GLB  D2  1  ;
   /////    ADDRESS MUX GLB2      /////
   SIGTYPE IRAM8 ASYNC OUT;
   SIGTYPE IRAM9 ASYNC OUT;
   EQUATIONS
      IRAM8 = ROW_COL & ACC_REF & IA8    /////  ROW SELECT    /////
         #  !ROW_COL & ACC_REF & IA18 /////  COLUMN SELECT   /////
         #  !ACC_REF & RCNTR8;              /////   REFRESH ADDR SELECT  /////
      IRAM9 = ROW_COL & ACC_REF & IA9
         #  !ROW_COL & ACC_REF & IA19
         #  !ACC_REF & RCNTR9;
   END
END;

SYM GLB  D5  1  ;
////// REFRESH ROW COUNTER GLB0     /////
   SIGTYPE RCNTR0 REG OUT;
   SIGTYPE RCNTR1 REG OUT;
   SIGTYPE RCNTR2 REG OUT;
   SIGTYPE RCNTR3 REG OUT;
   EQUATIONS
      RCNTR0.PTCLK = !IRAS0;     /////    USE RAS AS THE COUNTER CLOCK   ////
      RCNTR0 = !RCNTR0 & !ACC_REF     /////   COUNT DURING REFRESH      /////
           #   RCNTR0 & ACC_REF;       /////    HOLD DURING ACCESS          /////
      RCNTR1 = (RCNTR1 & !ACC_REF)
            $$ ((RCNTR0 & !ACC_REF)
           #   (RCNTR1 & ACC_REF));
      RCNTR2 = (RCNTR2 & !ACC_REF)
            $$ ((RCNTR1 & RCNTR0 & !ACC_REF)
           #   (RCNTR2 & ACC_REF));
      RCNTR3 = (RCNTR3 & !ACC_REF)
            $$ ((RCNTR2 & RCNTR1 & RCNTR0 & !ACC_REF)
           #   (RCNTR3 & ACC_REF));
   END
END;
```

```
SYM GLB  D6  1  ;
   //////  REFRESH ROW COUNTER GLB1   /////
   SIGTYPE RCNTR4 REG OUT;
   SIGTYPE RCNTR5 REG OUT;
   SIGTYPE RCNTR6 REG OUT;
   SIGTYPE RCNTR7 REG OUT;
   EQUATIONS
      /////    USE RAS AS THE COUNTER CLOCK   ////
      RCNTR4.PTCLK = !IRAS0;
      RCNTR4 =  (RCNTR4 & !ACC_REF)
      /////  COUNT DURING REFRESH   /////
            $$ ((RCNTR3 & RCNTR2 & RCNTR1 & RCNTR0 & !ACC_REF)
            #   (RCNTR4 & ACC_REF));
      /////    HOLD DURING ACCESS      /////
      RCNTR5 = (RCNTR5 & !ACC_REF)
            $$ ((RCNTR4 & RCNTR3 & RCNTR2 & RCNTR1 & RCNTR0 & !ACC_REF)
            #   (RCNTR5 & ACC_REF));
      RCNTR6 = (RCNTR6 & !ACC_REF)
            $$ ((RCNTR5 & RCNTR4 & RCNTR3 & RCNTR2 & RCNTR1 & RCNTR0 & !ACC_REF)
            #   (RCNTR6 & ACC_REF));
      RCNTR7 = (RCNTR7 & !ACC_REF)
            $$ ((RCNTR6 & RCNTR5 & RCNTR4 & RCNTR3 & RCNTR2 & RCNTR1 & RCNTR0 &
                !ACC_REF)
            #    (RCNTR7 & ACC_REF));
   END
END;

 SYM GLB  D7  1  ;
 //////  REFRESH ROW COUNTER GLB2   /////
   SIGTYPE RCNTR8 REG OUT;
   SIGTYPE RCNTR9 REG OUT;
   EQUATIONS
      RCNTR8.PTCLK = !IRAS0;     /////    USE RAS AS THE COUNTER CLOCK   ////
      RCNTR8 = (RCNTR8 & !ACC_REF)
            $$ ((RCNTR7 & RCNTR6 & RCNTR5 & RCNTR4
      ///// COUNT DURING REFRESH   /////
     & RCNTR3 & RCNTR2 & RCNTR1 & RCNTR0 & !ACC_REF) # (RCNTR8 & ACC_REF));
      /////   HOLD DURING ACCESS       /////
      RCNTR9 = (RCNTR9 & !ACC_REF)
            $$ ((RCNTR8 & RCNTR7 & RCNTR6 & RCNTR5 & RCNTR4 & RCNTR3 & RCNTR2 &
                  RCNTR1 & RCNTR0 & !ACC_REF)
             #   (RCNTR9 & ACC_REF));
   END
END;

SYM GLB  C7  1  ;
   /////     STATE BITS GLB      /////
   SIGTYPE ST0 REG OUT;
   SIGTYPE ST1 REG OUT;
   FJK11 (ST0,JST0,KST0,ICLK);
   FJK11 (ST1,JST1,KST1,ICLK);
   EQUATIONS
      JST0 = !ST1 & !ST0 & REFRESH;          /////   STATE BIT0 SET INPUT   /////
      KST0 = !ST1 & ST0 & SCNT1 & SCNT0;    /////   STATE BIT0 RESET INPUT   ///
//
      JST1 = !ST1 & !ST0 & !REFRESH & !IALE & IMIO_
```

```
           # !ST1 & !ST0 & !REFRESH & ACCESS;     /////   STATE BIT1 SET INPUT   /
////
      KST1 = ST1 & !ST0 & SCNT1 & SCNT0
           # !ST1 & ST0 & SCNT1 & SCNT0;        /////   STATE BIT0 RESET INPUT   /
////
   END
END;

SYM GLB  C6  1  ;
/////    STATE COUNTER BITS GLB      /////
   SIGTYPE SCNT0 REG OUT;
   SIGTYPE SCNT1 REG OUT;
   FJK11 (SCNT0,JSCNT0,KSCNT0,ICLK);
   FJK11 (SCNT1,JSCNT1,KSCNT1,ICLK);
   EQUATIONS
      JSCNT0 = !SCNT0 & ST1 & !ST0
           #  !SCNT0 & !ST1 & ST0;  /////   STATE COUNTER BIT0 SET INPUT /////
      KSCNT0 = SCNT0 & ST1 & !ST0
           #  SCNT0 & !ST1 & ST0
           #  ST1 & !ST0 & SCNT1 & SCNT0
           #  !ST1 & ST0 & SCNT1 & SCNT0;  /////STATE COUNTER BIT0 RESET INPUT  /
////
      JSCNT1 = !SCNT1 & SCNT0 & ST1 & !ST0
           #  !SCNT1 & SCNT0 & !ST1 & ST0; ///// STATE COUNTER BIT1 SET INPUT   /
///
      KSCNT1 = SCNT1 & SCNT0 & ST1 & !ST0
           #  SCNT1 & SCNT0 & !ST1 & ST0
           #  ST1 & !ST0 & SCNT1 & SCNT0
           #  !ST1 & ST0 & SCNT1 & SCNT0;  ///// STATE COUNTER BIT0 RESET INPUT /
////
   END
END;

SYM GLB  C5  1  ;
   /////    CONTROL SIGNALS GLB0      /////
   SIGTYPE RFC REG OUT;
   SIGTYPE ACC_REF REG OUT;
   FJK11 (RFC,JRFC,KRFC,ICLK);
   FJK11 (ACC_REF,JACC_REF,KACC_REF,ICLK);
   EQUATIONS
      JRFC = !ST1 & ST0 & SCNT1 & !SCNT0;      /////  REFRESH COMPLETE SET INPUT
/////
      KRFC = !ST1 & ST0 & SCNT1 & SCNT0; /////  REFRESH COMPLETE RESET INPUT  ///
//
      JACC_REF = !ST1 & ST0 & SCNT1 & SCNT0
            #   IRESET;                /////   ACCESS/REFRESH SET INPUT   /////
      KACC_REF = !ST1 & !ST0 & REFRESH & !IRESET;/////ACCESS/REFRESH RESET INPUT
/////
   END
END;

SYM GLB  C1  1  ;
/////    ROW ADDRESS STROBE (RAS3,RAS2) GLB      /////
   SIGTYPE IRAS3 REG OUT;
   SIGTYPE IRAS2 REG OUT;
   EQUATIONS
      IRAS3 = !ST0 & !IA20 & IRAS3 & !IRESET    /////   REDUCED RAS3    /////
           # !ST1 & !IA21 & IRAS3 & !IRESET
           # !ST0 &  ST1 & SCNT0 & SCNT1 & IA20 & IA21 & !IRESET
           #  ST0 & !ST1 & SCNT0 & SCNT1 & !IRESET
```

```
            # !ST0 & !ST1 & IRAS3 & !IRESET
            #  ST0 & ST1 & IRAS3 & !IRESET
            #  SCNT1 & IRAS3 & !IRESET
            #  SCNT0 & IRAS3 & !IRESET;
      IRAS3.CLK = ICLK;

      IRAS2 = !ST0 & IA20 & IRAS2 & !IRESET    /////   REDUCED RAS2    /////
            # !ST1 & !IA21 & IRAS2 & !IRESET
            # !ST0 &  ST1 & SCNT0 & SCNT1 & !IA20 & IA21 & !IRESET
            #  ST0 & !ST1 & SCNT0 & SCNT1 & !IRESET
            # !ST0 & !ST1 & IRAS2 & !IRESET
            #  ST0 & ST1 & IRAS2 & !IRESET
            #  SCNT1 & IRAS2 & !IRESET
            #  SCNT0 & IRAS2 & !IRESET;
      IRAS2.CLK = ICLK;

   END
END;
SYM GLB  B7  1  ;
   /////    COLUMN ADDRESS STROBE (CAS0,CAS1) GLB0      /////
   SIGTYPE ICAS0 REG OUT;
   SIGTYPE ICAS1 REG OUT;
   FJK11 (ICAS0,JCAS0,KCAS0,ICLK);
   FJK11 (ICAS1,JCAS1,KCAS1,ICLK);
   EQUATIONS
      /////   CAS0 SET INPUT   /////
      JCAS0 = ST1 & !ST0 & !IA1 & !IA0 & SCNT1 & SCNT0
          #   IRESET;
      /////CAS0 RESET              INPUT /////
      KCAS0 = ST1 & !ST0 & !IA1 & !IA0 & !SCNT1 & SCNT0 & !IRESET;
      /////   CAS1 SET INPUT   /////
      JCAS1 = ST1 & !ST0 & !IA1 & IA0 & SCNT1 & SCNT0
         #   IRESET;
      /////CAS1 RESET INPUT /////
      KCAS1 = ST1 & !ST0 & !IA1 & IA0 & !SCNT1 & SCNT0 & !IRESET;
   END
END;

SYM GLB  B6  1  ;
   /////    COLUMN ADDRESS STROBE (CAS2,CAS3) GLB1      /////
   SIGTYPE ICAS2 REG OUT;
   SIGTYPE ICAS3 REG OUT;
   FJK11 (ICAS2,JCAS2,KCAS2,ICLK);
   FJK11 (ICAS3,JCAS3,KCAS3,ICLK);
   EQUATIONS
     JCAS2 = ST1 & !ST0 & IA1 & !IA0 & !SCNT1 & SCNT0     /////   CAS2 SET INPUT
/////
        #   IRESET;
     ///// CAS2 RESET INPUT             /////
     KCAS2 = ST1 & !ST0 & IA1 & !IA0 & SCNT1 & SCNT0 & !IRESET;
     JCAS3 = ST1 & !ST0 & IA1 & IA0 & !SCNT1 & SCNT0///// CAS3 SET INPUT  /////
        #   IRESET;
     ///// CAS3 RESET INPUT             /////
     KCAS3 = ST1 & !ST0 & IA1 & IA0 & SCNT1 & SCNT0 & !IRESET;
   END
END;
```

```
SYM GLB  B5  1  ;
/////     CONTROL SIGNALS (ACCESS,WRITE) GLB1      /////
   SIGTYPE ACCESS REG OUT;
   SIGTYPE IWREG REG OUT;
   FJK11 (ACCESS,JACCESS,KACCESS,ICLK);
   FJK11 (IWREG,JWREG,KWREG,ICLK);
   EQUATIONS
      JACCESS = !IALE & IMIO_;    /////    MEMORY ACCESS REQUEST SET INPUT    ////
/
      KACCESS = ST1 & !ST0 & SCNT1 & SCNT0;/////MEMORY ACCESS REQUEST RESET
INPUT/////
        JWREG = !ACCESS & IRW_          /////   WRITE REGISTER SET INPUT    /////
             #  ST1 & !ST0 & SCNT1 & SCNT0
             #  IRESET;
       KWREG = !ACCESS & !IRW_ & !IRESET;     /////    WRITE REGISTER RESET INPUT
/////
   END
END;

SYM GLB  B4  1  ;
   /////     CONTROL SIGNALS (ROW/COL,RDY)GLB2      /////
   SIGTYPE ROW_COL REG OUT;
   SIGTYPE IRDY REG OUT;
   FJK11 (ROW_COL,JROW_COL,KROW_COL,ICLK);
   FJK11 (IRDY,JRDY,KRDY,ICLK);
   EQUATIONS
   JROW_COL = ST1 & !ST0 & SCNT1 & SCNT0/////   ROW/COL SELECT SET INPUT    /////
          #  IRESET;
   KROW_COL = ST1 & !ST0 & !SCNT1 & SCNT0 & !IRESET/////ROW/COL SELECT RESET SET
              INPUT/////
       JRDY = ST1 & !ST0 & SCNT1 & !SCNT0;    /////   READY SET INPUT    /////
       KRDY = ST1 & !ST0 & SCNT1 & SCNT0;     /////   READY RESET INPUT   /////
   END
END;

SYM IOC  IO16  1  ;
   //   ADDR 12 I/O CELL W/REG. INPUT //
   XPIN  IO  XA12;
   ID11 (IA12,XA12,IICLK);
END;

SYM IOC  IO15  1  ;
   //  ADDR 11 I/O CELL W/REG. INPUT //
   XPIN  IO  XA11;
   ID11 (IA11,XA11,IICLK);
END;

SYM IOC  IO14  1  ;
   //  ADDR 10 I/O CELL W/REG. INPUT //
   XPIN  IO  XA10;
   ID11 (IA10,XA10,IICLK);
END;

SYM IOC  IO13  1  ;
   //  ADDR 9 I/O CELL W/REG. INPUT //
   XPIN  IO  XA9;
   ID11 (IA9,XA9,IICLK);
END;
```

```
SYM IOC  IO12  1  ;
   //  ADDR 8 I/O CELL W/REG. INPUT //
   XPIN  IO  XA8;
   ID11 (IA8,XA8,IICLK);
END;

SYM IOC  IO11  1  ;
   //  ADDR 7 I/O CELL W/REG. INPUT //
   XPIN  IO  XA7;
   ID11 (IA7,XA7,IICLK);
END;
SYM IOC  IO10  1  ;
   //  ADDR 6 I/O CELL W/REG. INPUT //
   XPIN  IO  XA6;
   ID11 (IA6,XA6,IICLK);
END;

SYM IOC  IO9  1  ;
   //  ADDR 5 I/O CELL W/REG. INPUT //
   XPIN  IO  XA5;
   ID11 (IA5,XA5,IICLK);
END;

SYM IOC  IO8  1  ;

   //  ADDR 4 I/O CELL W/REG. INPUT //
   XPIN  IO  XA4;
   ID11 (IA4,XA4,IICLK);
END;

SYM IOC  IO7  1  ;
   //  ADDR 3 I/O CELL W/REG. INPUT //
   XPIN  IO  XA3;
   ID11 (IA3,XA3,IICLK);

END;
SYM IOC  Y2  1  ;
   //  INPUT REGISTER CLOCK (ALE) //
   XPIN  CLK  XICLK;
   IB11 (IICLK,XICLK);
END;

SYM IOC  IO6  1  ;
   //  ADDR 2 I/O CELL W/REG. INPUT //
   XPIN  IO  XA2;
   ID11 (IA2,XA2,IICLK);
END;

SYM IOC  IO5  1  ;
  //  ADDR 1 I/O CELL W/REG. INPUT //
  XPIN  IO  XA1;
  ID11 (IA1,XA1,IICLK);
END;

SYM IOC  IO4  1  ;
   //  ADDR 0 I/O CELL W/REG. INPUT //
   XPIN  IO  XA0;
   ID11 (IA0,XA0,IICLK);
END;
```

```
SYM IOC  IO3  1  ;
   //  READY I/O CELL, OUTPUT //
   XPIN  IO  XRDY;
   OB11 (XRDY,IRDY);
END;

SYM IOC  IO2  1  ;
   //  ADDRESS LATCH ENABLE I/O CELL /
/
   XPIN  IO  XALE;
   IB11 (IALE,XALE);
END;

SYM IOC  IO1  1  ;
   //  MEMORY OR I/O ACCESS //
   XPIN  IO  XMIO_;
   IB11 (IMIO_,XMIO_);
END;

SYM IOC  IO0  1  ;
   //  READ WRITE SELECTION //
   XPIN  IO  XRW_;
   IB11 (IRW_,XRW_);
END;

SYM IOC  Y0  1  ;
   //  SYSTEM CLOCK INPUT //
   XPIN  CLK  XSYS_CLK LOCK 20;
   IB11 (ICLK,XSYS_CLK);
END;

SYM IOC  IO17  1  ;
   //  ADDR 13 I/O CELL W/REG. INPUT /
/
   XPIN  IO  XA13;
   ID11 (IA13,XA13,IICLK);
END;

SYM IOC  IO18  1  ;
   //  ADDR 14 I/O CELL W/REG. INPUT /
/
   XPIN  IO  XA14;
   ID11 (IA14,XA14,IICLK);
END;

SYM IOC  IO19  1  ;
   //  ADDR 15 I/O CELL W/REG. INPUT /
/
   XPIN  IO  XA15;
   ID11 (IA15,XA15,IICLK);
END;

SYM IOC  IO20  1  ;
   //  ADDR 20 I/O CELLW/REG. INPUT //
   XPIN IO XA20;
   ID11 (IA20,XA20,IICLK);
END;
```

```
SYM IOC IO21 1;
   //  ADDR 21 I/O CELL W/REG.INPUT //
   XPIN IO XA21
   ID11 (IA21,XA21,IICLK);
END;

SYM IOC IO22 1;
   XPIN   IO  XRESET;
   IB11  (IRESET, XRESET);
END;

SYM  IOC  IO23   1 ;
    XPIN   IO  XREFRESH;
    IB11  (REFRESH, XREFRESH);
END;

SYM  IOC  IO24   1 ;
    XPIN   IO  XRAM0;
    OB11  (XRAM0, IRAM0);
END;

SYM  IOC  IO25   1 ;
    XPIN   IO  XRAM1;
    OB11  (XRAM1, IRAM1);
END;

SYM  IOC  IO26   1 ;
    XPIN   IO  XRAM2;
    OB11  (XRAM2, IRAM2);
END;

SYM  IOC  IO27   1 ;
    XPIN   IO  XRAM3;
    OB11  (XRAM3, IRAM3);
END;

SYM  IOC  IO28   1 ;
    XPIN   IO  XRAM4;
    OB11  (XRAM4, IRAM4);
END;
SYM  IOC  IO29   1 ;
    XPIN   IO  XRAM5;
    OB11  (XRAM5, IRAM5);
END;

SYM  IOC  IO30   1 ;
    XPIN   IO  XRAM6;
    OB11  (XRAM6, IRAM6);
END;
```

```
SYM  IOC  IO31   1 ;
    XPIN   IO  XRAM7;
    OB11  (XRAM7, IRAM7);
END;

SYM  IOC  IO32   1 ;
    XPIN   IO  XRAM8;
    OB11  (XRAM8, IRAM8);
END;

SYM  IOC  IO33   1 ;
    XPIN   IO  XRAM9;
    OB11  (XRAM9, IRAM9);
END;

SYM  IOC  IO34   1 ;
    XPIN   IO  XST0;
    OB11  (XST0, ST0);
END;

SYM  IOC  IO36   1 ;
    XPIN   IO  XST1;
    OB11  (XST1, ST1);
END;

SYM  IOC  IO38   1 ;
    XPIN   IO  XSCNT0;
    OB11  (XSCNT0, SCNT0);
END;

SYM  IOC  IO40   1 ;
    XPIN   IO  XSCNT1;
    OB11  (XSCNT1, SCNT1);
END;

SYM  IOC  IO41   1 ;
    XPIN   IO  XACCESS;
    OB11  (XACCESS, ACCESS);
END;
SYM  IOC  IO42   1 ;
    XPIN   IO  XIWREG;
    OB11  (XIWREG, IWREG);
END;

SYM  IOC  IO43   1 ;
    XPIN   IO  XROW_COL;
    OB11  (XROW_COL, ROW_COL);
END;

SYM  IOC  IO44   1 ;
    XPIN   IO  XIRDY;
    OB11  (XIRDY, IRDY);
END;
```

```
SYM  IOC  IO45   1 ;
    XPIN   IO  XRFC;
    OB11  (XRFC, RFC);
END;

SYM  IOC  IO46   1 ;
    XPIN   IO  XACC_REF;
    OB11  (XACC_REF, ACC_REF);
END;
```

Notes

Lattice Bulletin Board Systems

Introduction

Lattice maintains two Bulletin Board Systems (BBSs) to communicate with customers. One BBS is located in Milpitas, California at Lattice's Silicon Valley Design Center. This BBS provides primary ispLSI and pLSI support. The second BBS is located in Hillsboro, Oregon at Lattice's headquarters. This BBS provides primary GAL support and secondary ispLSI and pLSI support. The following two sections explain in detail how to connect to each of these BBSs and how to transfer information.

Using the Lattice Silicon Valley BBS

The Silicon Valley BBS is for ispLSI and pLSI customers, distributors and FAEs who are requesting technical support on our ispLSI and pLSI families of HDPLDs. You can use the Silicon Valley BBS to:

- Transfer designs to and from Lattice Application Engineers
- Join conferences to share and exchange information with Lattice Application Engineers and other users

Telephone number and Communication Software Setup

The telephone number for the Silicon Valley BBS is (408) 428 - 6417. The BBS supports modem speeds from 300-9600 Baud, and supports the typical default communication parameters of eight data bits, one stop bit, and no parity (8-N-1).

New User

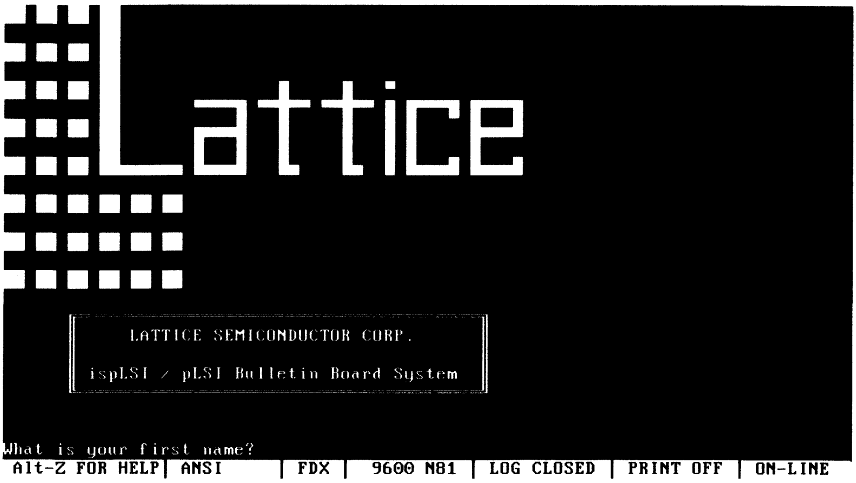
If you have not used the Silicon Valley BBS before, the system will first ask if you want the graphics mode. This mode will help a first time user by displaying different options by either blinking or displaying a different color text, if you have a color display.

You will then be asked for your first name, your last name and password (see Figure 1). The user name must be your name; do not use your company name as a user name. The password can be up to 12 alphanumeric characters long. You will also be prompted to fill out a short script. You should be prepared with information about the Lattice software you are using and the 10 digit serial number from the Lattice security block(s).

After you complete the questionnaire, the system will display the main menu. As a first time user of the BBS, you have no rights to upload or download files. Your security level must be upgraded. This is done three times a day: Monday through Friday at 8:00 a.m., 12:00 noon and 5:00 p.m. except on holidays. All times are Pacific Standard Time (PST) or Pacific Daylight Savings Time (PDT).

For subsequent access, after you have logged on to the system, you will be asked if you want to scan for messages. Answer (Y/N) and press the enter key.

Figure 1. Silicon Valley BBS Initial Screen



```
LATTICE SEMICONDUCTOR CORP.

ispLSI / pLSI Bulletin Board System

What is your first name?
Alt-Z FOR HELP| ANSI       | FDX |  9600 N81 | LOG CLOSED | PRINT OFF | ON-LINE
```

Main Menu

From this point, you'll be at the Main menu, with access to all other menus. Note that the Join option is not available to a new user until your security level has been upgraded.

Listed below is a brief description of the options available to a new user

- G - Hang up
- H - Help menu - Command options and description
- C - Leave a message to the System Administrator (SYSOP)

For users who have been upgraded and have previously joined a conference, the Figure 3 menu will be seen. This menu has seven additional commands that are accessible. When you login to the BBS and have previously joined a conference, the menu in Figure 3 will be the main menu. The conference you are in is the location you will be placed during your next successful login. Your options will be:

- D - Download a file from the BBS (Instruct the BBS computer to go into send mode
- E - Leave a message for another user
- J - Join conference - Change to conference 1 or 2
- R - Read a message left by another user
- S - Required questionnaire about design, before Up load is started
- T - Transfer protocol - user specified type of file transfer method
- U - Upload a file - Instruct the BBS computer to go into receive mode

Upload files to the BBS

If you need to upload (send) a file to the Lattice BBS, the file should be zipped up, and a readme file should be added to the zip file. The zip utilities compress the file size and help to eliminate file transmission errors. The readme file should have a description of the questions, comments and/or problem you have.

To upload a file to the BBS, do the following:

1. Type J)oin 1 or 2 <enter> - This puts you into conference 1 or 2.
2. Type T)ransfer protocol <enter>.
3. Select which file transfer method you want - (X , Y, or Z protocol).
4. Type S)cript <enter> and fill out the questionnaire about your design and the design tools used.
5. Type U)pload *filename* <enter>. *Filename* is the name it will be called on the BBS.
6. Select the send file utility on your software package. If you are using a Procomm like software package, you press the "Page Up" key.

Download files from the BBS

To download a file to the BBS, do the following:

Figure 2. Silicon Valley BBS Main Menu

System Operations	Message Operations	File Operations
G)oodbye (Hang Up)	C)omments to SYSOP	Join a conference before
H)elp Functions		Uploading or Downloading
J)oin a Conference		a File
		New Users cannot Upload or
		Download Files

Figure 3. Silicon Valley BBS Menu Selections available in conference 1 and 2

System Operations	Message Operations	File Operations
G)oodbye (Hang Up)	C)omments to SYSOP	D)ownload a File
H)elp Functions		E)nter a Message
J)oin a Conference	R)ead Message	To Upload a File:
T)rans. Protocol	S)cript Question	S)cript #1 then U)pload

1. Type J)oin 1 or 2 <enter> - This puts you into conference 1 or 2.
2. Type T)ransfer protocol <enter>.
3. Select which file transfer method you want - (X , Y, or Z protocol).
4. Type D)ownload *filename*<enter>. *Filename* is name it will be called on the BBS.
5. Select the receive file utility on your software package. If you are using a Procomm like software package, you press the "Page Down" key.

Electronic Mail

Lattice Semiconductor does support the use of "E Mail". Communications regarding ispLSI or pLSI products can be sent to "applications@lattice.com". Please include details as well as a voice telephone number.

Using the Hillsboro BBS

The Hillsboro BBS is accessible by any user with a modem and communication package. You can use the Hillsboro BBS to:

1. Transfer designs to and from Lattice Applications Engineers
2. Access the latest utilities
3. Join conferences to share and exchange information with Lattice Applications Engineers and other users
4. Send mail to and from Lattice Applications Engineers

Telephone Number and Communication Software Setup

The telephone number for the Hillsboro BBS is (503) 693-0215. The BBS supports modem speeds from 300-9600 Baud, and supports the typical default communication parameters of eight data bits, one stop bit, and no parity (8-N-1).

If You Are A New User

If you have not used the BBS before, the system will ask you a short set of questions. These questions are used to maintain statistics about our callers, and will not take long to answer.

You will be asked for a user name and password. For the user name, simply enter your name. You will also be prompted to enter a password. It is important to remember the password you enter. You will need it whenever you log on to the system, and if you forget it, you may have to have your account information deleted, and you will have to log on again as a new user.

After you complete the questionnaire, the system will display the main menu. From this point on, you will see the main menu after you log on and give the system your name and password.

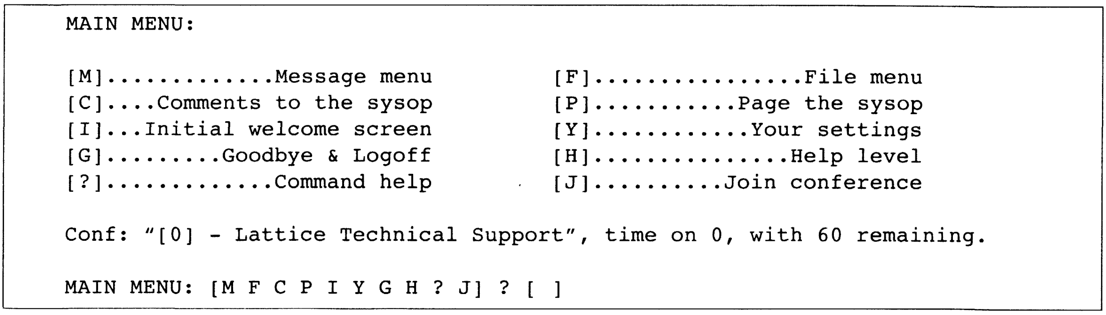
The Main Menu

The Main Menu is the top level menu, meaning that you can access all other menus from this point. The options you are most likely to use are:

f-	file menu
j-	join conference
m-	message menu

Use the file menu when you want to upload or download files. Use the join conference menu when you want to join a conference on a particular topic. Use the message

Figure 4. Hillsboro BBS Main Menu



```
MAIN MENU:

[M].............Message menu        [F]................File menu
[C]....Comments to the sysop        [P]...........Page the sysop
[I]...Initial welcome screen        [Y]............Your settings
[G].........Goodbye & Logoff        [H]...............Help level
[?].............Command help        [J]..........Join conference

Conf: "[0] - Lattice Technical Support", time on 0, with 60 remaining.

MAIN MENU: [M F C P I Y G H ? J] ? [ ]
```

menu when you want to leave a message, either as part of a conference, or to a specific individual.

The File Menu

When you choose the File menu from the Main menu, you will be presented with a list of options for uploading, listing, or downloading files on the BBS.

If you need to download a file, and you know the name of the file you want to download, choose option D and download the file. You don't have to be in a particular conference to download the file.

If you don't know the exact name of the file, then you can choose one of the options in Figure 6 to locate the file.

Figure 7 is a an example session where files in a particular area are listed. In this example, when "L" was entered for the file area, a list of all the file areas was displayed. From this list of areas, you can choose the specific area you are interested in listing.

Transferring Files

Once you identify the file that you want to download, choose the D option from the Download menu. You will be prompted for the file name. Alternatively, you can initiate a download after listing the files in an area. Note that one of the options in the menu listed above is D for download.

File Protocols

The BBS supports a number of different file protocols for downloading and uploading files. Which one to choose depends on your communication software. Xmodem is one of the most popular protocols, and your communication software is likely to support it.

You can display the file transfer protocol you've chosen by selecting the Y option from the main menu (see Figure 8).

When you choose the Y option, all configuration parameters are displayed. Number 14 is the file transfer protocol. By entering 14 as the setting to change, you can change to a different file transfer protocol. Note that you can choose to select the file transfer protocol each time you start a download by selecting S as the default protocol option.

Conferences

The Hillsboro BBS also provides a conference facility for you to share information with Lattice Application Engineers, and other users of Lattice Products. Conferences are simply a way to organize messages left by users so that they are grouped by a common subject. When you join a conference, messages that you read or leave will then be left in that conference area.

Figure 5. Hillsboro BBS File Menu

```
MAIN MENU: [M F C P I Y G H ? J] ? [F]
FILE MENU:

[D].......Download a file(s)         [U].........Upload a file(s)
[L].....List available files         [Q]........Quit to main menu
[N]......New files since [N]         [I]....Information on a file
[T]..............Text search         [F].......File transfer info
[G].........Goodbye & logoff         [H]...............Help level
[?].............Command help         [M].............Message menu
[V]...View a compressed file         [R].........Read a text file
[J]..........Join conference         [E].........Edit marked list

Conf: "[0] - Lattice Technical Support", time on 2, with 58 remaining.

FILE MENU: [D U L Q N I T F G H ? M V R J E] ? [ ]
```

Figure 6. Finding a File Without Knowing the Specific Name

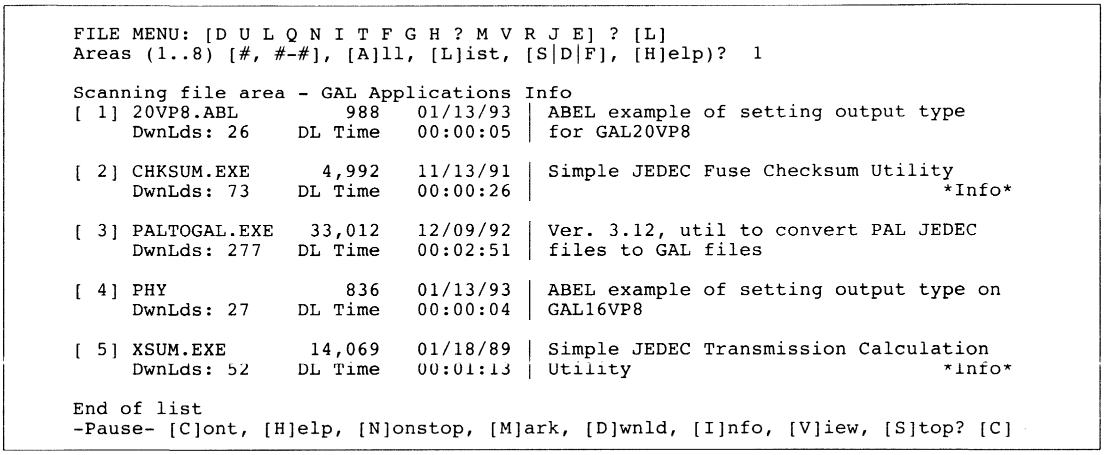
If You Want to List Files	Then Choose Option	And Enter
Uploaded after a certain date	N	The starting date for the search
Containing a specific word in their name or description	T	The word to search for
Within an area category	L	The area to list

You can join a conference from the Main menu by entering the J command (see Figure 9). After entering this command, you can either enter the number of a conference you want to join, or enter an L to list the available conferences.

Once the conference is joined, you can enter the Message menu from the Main menu (see Figure 10), and read new messages in the conference by entering the R command, or you can enter the S command to scan for new messages.

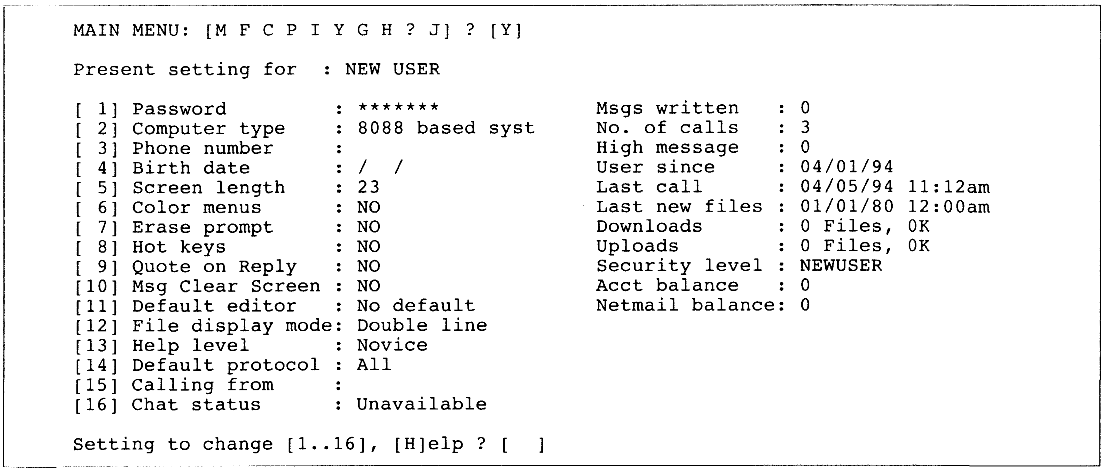
The Scan command can be an easy way to locate topics of interest. The Scan menu will list a variety of options to search through messages. For example, you can specify a word to search for anywhere in the body of a message by selecting the B option. Any messages that contain this text will be displayed (see Figure 11).

Figure 7. Example Session Showing Files Listed in a Particular Area



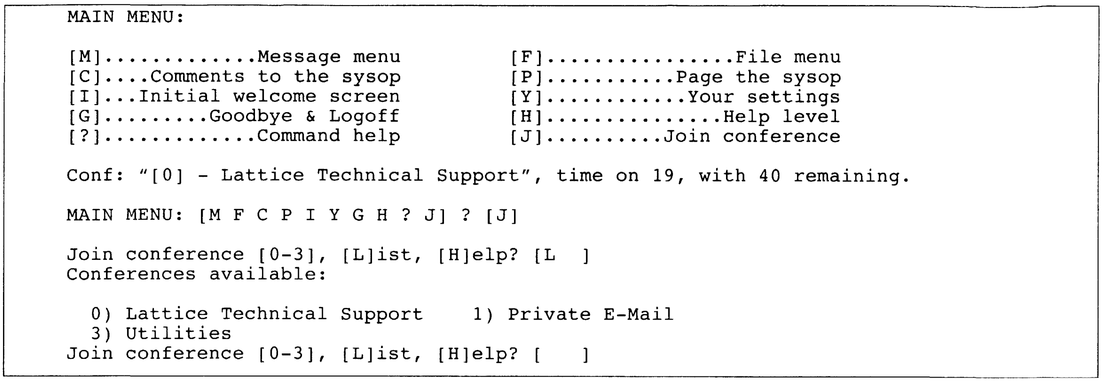
```
FILE MENU: [D U L Q N I T F G H ? M V R J E] ? [L]
Areas (1..8) [#, #-#], [A]ll, [L]ist, [S|D|F], [H]elp)?  1

Scanning file area - GAL Applications Info
[ 1] 20VP8.ABL          988   01/13/93 | ABEL example of setting output type
     DwnLds: 26     DL Time   00:00:05 | for GAL20VP8

[ 2] CHKSUM.EXE       4,992   11/13/91 | Simple JEDEC Fuse Checksum Utility
     DwnLds: 73     DL Time   00:00:26 |                                  *Info*

[ 3] PALTOGAL.EXE    33,012   12/09/92 | Ver. 3.12, util to convert PAL JEDEC
     DwnLds: 277    DL Time   00:02:51 | files to GAL files

[ 4] PHY                836   01/13/93 | ABEL example of setting output type on
     DwnLds: 27     DL Time   00:00:04 | GAL16VP8

[ 5] XSUM.EXE        14,069   01/18/89 | Simple JEDEC Transmission Calculation
     DwnLds: 52     DL Time   00:01:13 | Utility                          *Info*

End of list
-Pause- [C]ont, [H]elp, [N]onstop, [M]ark, [D]wnld, [I]nfo, [V]iew, [S]top? [C]
```

Figure 8. Selecting the Y Option from the Main Menu



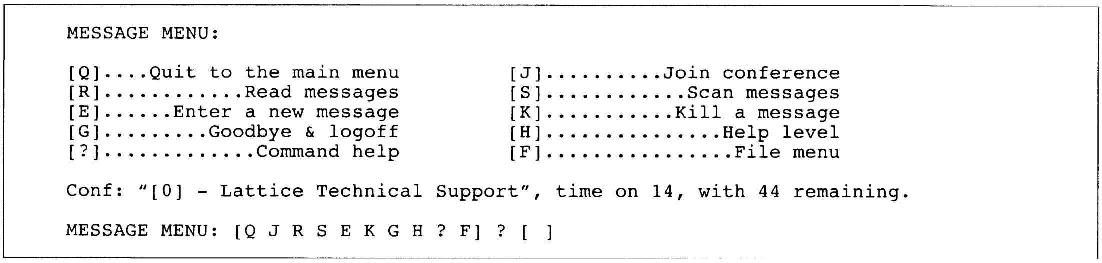
```
MAIN MENU: [M F C P I Y G H ? J] ? [Y]

Present setting for  : NEW USER

[ 1] Password         : *******               Msgs written    : 0
[ 2] Computer type    : 8088 based syst       No. of calls    : 3
[ 3] Phone number     :                       High message    : 0
[ 4] Birth date       : /  /                  User since      : 04/01/94
[ 5] Screen length    : 23                    Last call       : 04/05/94 11:12am
[ 6] Color menus      : NO                    Last new files  : 01/01/80 12:00am
[ 7] Erase prompt     : NO                    Downloads       : 0 Files, 0K
[ 8] Hot keys         : NO                    Uploads         : 0 Files, 0K
[ 9] Quote on Reply   : NO                    Security level  : NEWUSER
[10] Msg Clear Screen : NO                    Acct balance    : 0
[11] Default editor   : No default            Netmail balance: 0
[12] File display mode: Double line
[13] Help level       : Novice
[14] Default protocol : All
[15] Calling from     :
[16] Chat status      : Unavailable

Setting to change [1..16], [H]elp ? [  ]
```

Figure 9. Joining a Conference



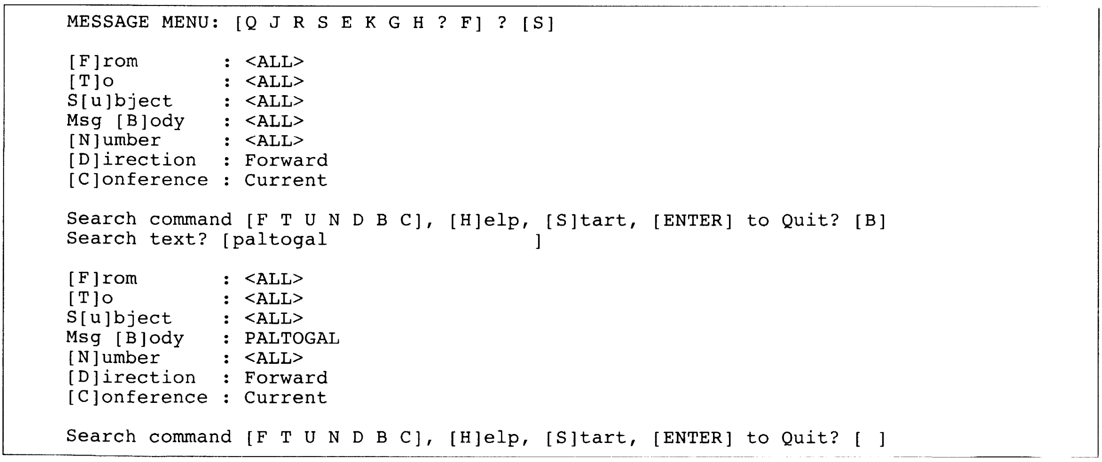
```
MAIN MENU:

[M].............Message menu        [F]................File menu
[C]....Comments to the sysop        [P]...........Page the sysop
[I]...Initial welcome screen        [Y]............Your settings
[G].........Goodbye & Logoff        [H]...............Help level
[?].............Command help        [J]..........Join conference

Conf: "[0] - Lattice Technical Support", time on 19, with 40 remaining.

MAIN MENU: [M F C P I Y G H ? J] ? [J]

Join conference [0-3], [L]ist, [H]elp? [L  ]
Conferences available:

  0) Lattice Technical Support    1) Private E-Mail
  3) Utilities
Join conference [0-3], [L]ist, [H]elp? [   ]
```

Figure 10. Message Menu



```
MESSAGE MENU:

[Q]....Quit to the main menu        [J]..........Join conference
[R]............Read messages        [S]............Scan messages
[E]......Enter a new message        [K]...........Kill a message
[G].........Goodbye & logoff        [H]...............Help level
[?].............Command help        [F]................File menu

Conf: "[0] - Lattice Technical Support", time on 14, with 44 remaining.

MESSAGE MENU: [Q J R S E K G H ? F] ? [ ]
```

Figure 11. The Scan Command



```
MESSAGE MENU: [Q J R S E K G H ? F] ? [S]

[F]rom        : <ALL>
[T]o          : <ALL>
S[u]bject     : <ALL>
Msg [B]ody    : <ALL>
[N]umber      : <ALL>
[D]irection   : Forward
[C]onference  : Current

Search command [F T U N D B C], [H]elp, [S]tart, [ENTER] to Quit? [B]
Search text? [paltogal                  ]

[F]rom        : <ALL>
[T]o          : <ALL>
S[u]bject     : <ALL>
Msg [B]ody    : PALTOGAL
[N]umber      : <ALL>
[D]irection   : Forward
[C]onference  : Current

Search command [F T U N D B C], [H]elp, [S]tart, [ENTER] to Quit? [ ]
```

Sales Offices

LATTICE SALES OFFICES

FRANCE
Lattice Semiconductor
Les Algorithmes
Bâtiment Homère
91 190 - Saint Aubin
TEL: (33) 69 33 22 77
FAX: (33) 60 19 05 21

GERMANY
Lattice Semiconductor
Hanns-Braun-Str. 50
85375 Neufahrn bei München
TEL: (49) 8165-9516-0
FAX: (49) 8165-9516-33

HONG KONG
Lattice Semiconductor
2802 Admiralty Centre, Tower 1
18 Harcourt Road
Hong Kong
TEL: (852) 529-0356
FAX: (852) 866-2315

JAPAN
Lattice Semiconductor
I•K Building 9F
1-23-3, Yanagibashi
Taitoh-ku, Tokyo 111
TEL: (81) 3-5820-3533
FAX: (81) 3-5820-3531

UNITED KINGDOM
Lattice Semiconductor
Castle Hill House
Castle Hill
Windsor
Berkshire SL4 1PD
TEL: (44) 753-830-842
FAX: (44) 753-833-457

NORTH AMERICA

CALIFORNIA
Lattice Semiconductor
1820 McCarthy Blvd.
Milpitas, CA 95035
TEL: (408) 428-6400
FAX: (408) 944-8450

Lattice Semiconductor
15707 Rockfield Plaza, Ste. 110
Irvine, CA 92718
TEL: (714) 580-3880
FAX: (714) 580-3888

FLORIDA
Lattice Semiconductor
12424 Research Pkwy.
Suite 101
Orlando, FL 32826
TEL: (407) 281-6500
FAX: (407) 658-0208

GEORGIA
Lattice Semiconductor
3091 Governors Lake Drive
Building 100, Suite 500
Norcross, GA 30071
TEL: (404) 446-2930
FAX: (404) 416-7404

ILLINOIS
Lattice Semiconductor
1 Pierce Place
Suite 500-E
Itasca, IL 60143
TEL: (708) 250-3118
FAX: (708) 250-3119

MASSACHUSETTS
Lattice Semiconductor
67 S. Bedford St.
Suite 400 West
Burlington, MA 01803
TEL: (617) 229-5819
FAX: (617) 272-3213

MINNESOTA
Lattice Semiconductor
3445 Washington Dr.
Suite 105
Eagan, MN 55122
TEL: (612) 686-8747
FAX: (612) 686-8746

NEW JERSEY
Lattice Semiconductor
115 Route 46
Suite F-1000
Mountain Lakes, NJ 07046
TEL: (201) 316-6024
FAX: (201) 316-6619

NEW YORK
Lattice Semiconductor
Linden Oaks Park
70 Linden Oaks
3rd Floor
Rochester, NY 14625
TEL: (716) 383-5320
FAX: (716) 383-5321

OREGON
Lattice Semiconductor
5555 N.E. Moore Ct.
Hillsboro, OR 97124
TEL: (503) 656-4808
FAX: (503) 656-6541

TEXAS
Lattice Semiconductor
100 Decker Ct. Ste. 280
Irving, TX 75062
TEL: (214) 650-1236
FAX: (214) 650-1237

Lattice Semiconductor
9600 Great Hills Trail
#150W
Austin, TX 78759
TEL: (512) 502-3057
FAX: (512) 343-7309

NORTH AMERICAN SALES REPRESENTATIVES

ALABAMA
CSR Electronics, Inc.
303 Williams Avenue
Ste. 931
Huntsville, AL 35801
TEL: (205) 533-2444
FAX: (205) 536-4031

ARIZONA
Summit Sales
7802 E. Gray Rd. #600
Scottsdale, AZ 85260
TEL: (602) 998-4850
FAX: (602) 998-5274

CALIFORNIA
Bager Electronics
17220 Newhope St. #209
Fountain Valley, CA 92708
TEL: (714) 957-3367
FAX: (714) 546-2654

Bager Electronics
6324 Variel Ave. #314
Woodland Hills, CA 91367
TEL: (818) 712-0011
FAX: (818) 712-0160

Criterion Sales
3350 Scott Blvd, Bldg.44
Santa Clara, CA 95054
TEL: (408) 988-6300
FAX: (408) 986-9039

Earle Associates
7585 Ronson Rd. #200
San Diego, CA 92111
TEL: (619) 278-5441
FAX: (619) 278-5443

COLORADO
Waugaman Associates
4800 Van Gordon
Wheat Ridge, CO 80033
TEL: (303) 423-1020
FAX: (303) 467-3095

CONNECTICUT
Comp Rep Associates
60 Connolly Pkwy.
Bldg. 12 Suite 210
Hamden, CT 06514
TEL: (203) 230-8369
FAX: (203) 230-8394

FLORIDA
Sales Engineering Concepts
4701 N. Federal Highway
Suite 430, Box B-11
Lighthouse Point, FL 33064
TEL: (305) 783-6900
FAX: (305) 782-7274

Sales Engineering Concepts
600 S. Northlake Blvd. #230
Altamonte Spgs, FL 32701
TEL: (407) 830-8444
FAX: (407) 830-8684

GEORGIA
CSR Electronics, Inc.
1651 Mt. Vernon Rd. # 200
Atlanta, GA 30338
TEL: (404) 396-3720
FAX: (404) 394-8387

ILLINOIS
Sumer, Inc.
1675 Hicks Rd.
Rolling Meadows, IL 60008
TEL: (708) 991-8500
FAX: (708) 991-0474

INDIANA
Electronic Sales and
Engineering
7739 E. 88th St.
PO Box 5009
Indianapolis, IN 46250
TEL: (317) 849-4260
FAX: (317) 841-0231

IOWA
Stan Clothier Company
1930 St. Andrews NE
Cedar Rapids, IA 52402
TEL: (319) 393-1576
FAX: (319) 393-7317

KANSAS
Stan Clothier Company
13000 West 87th St. #105
Lenexa, Kansas 66215
TEL: (913) 492-2124
FAX: (913) 492-1855

Sales Offices

MARYLAND
Deltatronics
24048 Sugar Cane Ln.
Gaithersburg, MD 20882
TEL: (301) 253-0615
FAX: (301) 253-9108

MASSACHUSETTS
Comp Rep Associates
100 Everett Street
Westwood, MA 02090
TEL: (617) 329-3454
FAX: (617) 329-6395

MICHIGAN
Greiner & Associates
15324 E. Jefferson Ave.
Grosse Pointe Park, MI 48230
TEL: (313) 499-0188
FAX: (313) 499-0665

MINNESOTA
Stan Clothier Company
9600 W. 76th St., Ste. #A
Eden Prairie, MN 55344
TEL: (612) 944-3456
FAX: (612) 944-6904

MISSOURI
Stan Clothier Company
3910 Old Highway 94 South
Suite 116
St. Charles, MO 63304
TEL: (314) 928-8078
FAX: (314) 447-5214

NEW JERSEY
Technical Marketing Group
175-3C Fairfield Ave.
West Caldwell, NJ 07006
TEL: (201) 226-3300
FAX: (201) 226-9518

NEW YORK
Technical Marketing Group
150 Broad Hollow Rd.
Suite 310
Melville, NY 11747
TEL: (516) 351-8833
FAX: (516) 351-8667

Tri-Tech Electronics
300 Main St.
E. Rochester, NY 14445
TEL: (716) 385-6500
FAX: (716) 385-7655

Tri-Tech Electronics
14 Westview Dr.
Fishkill, NY 12524
TEL: (914) 897-5611
FAX: (914) 897-5611

Tri-Tech Electronics
1043 Front St.
Binghampton, NY 13905
TEL: (607) 722-3580
FAX: (607) 722-3774

NORTH CAROLINA
CSR Electronics, Inc.
5848 Faringdon Place, Ste. 2
Raleigh, NC 27609
TEL: (919) 878-9200
FAX: (919) 878-9117

CSR Electronics, Inc.
6425 Creft Cr.
Indian Trail, NC 28079
TEL: (704) 882-3995
FAX: (704) 882-3999

OHIO
Makin & Associates
3165 Linwood Rd.
Cincinnati, OH 45208
TEL: (513) 871-2424
FAX: (513) 871-2524

Makin & Associates
6631 Commerce Pkwy. Ste. K
Dublin, OH 43017
TEL: (614) 793-9545
FAX: (614) 793-0256

Makin & Associates
6519 Wilson Mills Rd.
Mayfield Village, OH 44143
TEL: (216) 461-3500
FAX: (216) 461-1335

OKLAHOMA
West Associates
5555 E. 71st St. #8150
Tulsa, OK 74136
TEL: (918) 492-4300
FAX: (918) 492-4370

OREGON
Components West, Inc.
16300 SW Hart Rd.
Suite G
Beaverton, OR 97007
TEL: (503) 642-9110
FAX: (503) 642-9592

PENNSYLVANIA
Deltatronics
790 Penllyn Pike
Suite 201
Blue Bell, PA 19422
TEL: (215) 641-9930
FAX: (215) 641-9934

TENNESSEE
CSR Electronics, Inc.
Grand Union Bldg.
406 Union Ave.
Suite 550
Knoxville, TN 37902
TEL: (615) 637-0293
FAX: (615) 637-0466

TEXAS
West Associates
363 N. Sam Houston Pkwy E
Suite 615
Houston, TX 77060
TEL: (713) 999-0101
FAX: (713) 820-2001

West Associates
9171 Capital of Texas Hwy. N.
Houston Bldg. #120
Austin, TX 78759
TEL: (512) 343-1199
FAX: (512) 343-1922

West Associates
801 E. Campbell Rd. #350
Richardson, TX 75081
TEL: (214) 680-2800
FAX: (214) 699-0330

UTAH
Waugaman Associates
876 East Vine St.
Murray, UT 84107
TEL: (801) 261-0802
FAX: (801) 261-0830

VIRGINIA
Deltatronics
3562 13th St. NW
Washington, DC 20010
TEL: (202) 745-3844
FAX: (202) 483-0672

WASHINGTON
Components West, Inc.
4020 148th Ave. NE
Suite C
Redmond, WA 98052
TEL: (206) 885-5880
FAX: (206) 882-0642

WISCONSIN
Sumer, Inc.
13555 Bishops Court
Brookfield, WI 53005
TEL: (414) 784-6641
FAX: (414) 784-1436

PUERTO RICO
Sales Engineering Concepts
Condo. Buena Vista C-1
Urb. Mercedita
Ponce, P.R. 00731
TEL: (809) 841-4220
FAX: (809) 259-7223

CANADA

ALBERTA
Dynasty Components
Calgary, Alberta
TEL: (403) 560-1212
FAX: (403) 686-2364

BRITISH COLUMBIA
Dynasty Components
Vancouver, British Columbia
TEL: (604) 298-8288
FAX: (604) 298-8318

ONTARIO
Dynasty Components
1140 Morrison Dr.
Unit 110
Ottawa, Ontario
Canada, K2H 8S9
TEL: (613) 596-9800
FAX: (613) 596-9886

Dynasty Components
Toronto, Ontario
TEL: (905) 672-5977
FAX: (416) 489-3527

QUEBEC
Dynasty Components
Montreal, Quebec
TEL: (514) 843-1879
FAX: (514) 694-6826

INTERNATIONAL SALES REPRESENTATIVES AND DISTRIBUTORS

AUSTRALIA
ZATEK Australia, Ltd.
1059 Victoria Road
P.O Box 397, Suite 8
West Ryde, NSW 2114
Sydney, 3153
TEL: (61) 2 874-0122
FAX: (61) 2 874-6171

AUSTRIA
Avnet / E2000
Waidhausenstr. 19
A-1140 Wien
TEL: (43) 1-9112847
FAX: (43) 1-9113853

Steiner Electronic GmbH.
Egererstrasse 18
A-3013 Tullnerbach
TEL: (43) 2233 55 366-0
FAX: (43) 2233 55 360

BELGIUM
Alcom Electronics nv/sa
Singel 3
2550 Kontich
TEL: (32) 3 458-3033
FAX: (32) 3 458 3126

CHINA
MIE Ltd.
Wanguan Land House
5 East Zhixin Rd.
Haidian, Beijing 100083
TEL: (861) 201-7299
FAX: (861) 201-7299

CZECH REPUBLIC
HT-EUREP Electronic spoi.s.r.o.
c/o Comp. Ap spoi.s.r.o.
Rosenberggovych 10
180 00, Praha 8
TEL: (42) 2-6833858
FAX: (42) 2-6833858

DENMARK
Ditz Schweitzer
Vallensbaekvej 41
Postboks 5,
DK-2605 Brendby
TEL: (45) 42 45 30 44
FAX: (45) 42 45 92 06

FINLAND
Integrated Electronics OY
Turkhaudantie #1
00700 Helsinki
TEL: (358) 0 351 3134
FAX: (358) 0 351 3133

FRANCE
Company 3D
3-8 Rue Ambroize Croizat
91127 Palaiseau Cedex
TEL: (33) 1 64472929
FAX: (33) 1 64470084

Compress
30, Rue du Morvan
Silic 539
94633 Rungis Cedex
TEL: (33) 1 46878020
FAX: (33) 1 46866763

GERMANY
Avnet / E2000 GmbH
Stahlgruberring 12
81829 München
TEL: (49) 89 45 11 0 -01
FAX: (49) 89 45 11 0 -129

Eurodis Enatechnik GmbH
Schillerstrasse 14
25451 Quickborn
TEL: (49) 4106 612-0
FAX: (49) 4106 612-268

HONG KONG
RTI Industries Co. Ltd.
Rm. 402, Nan Fung Commercial Centre
No. 19, Lam Lok Street
Kowloon Bay, Kowloon
TEL: (852) 795 7421
FAX: (852) 795 7839

HUNGARY
HT-EUREP Electronic Kft.
X-Byte Kft.
Böszörnenyi ut 3/a
H-1126 Budapest
TEL: (36) 1-155-47-48
FAX: (36) 1-155-47-48

IRELAND
Silicon Concepts
Norebank House
Greens Hill, Kilkenny
TEL: (353) 56 64002
FAX: (353) 56 51438

ISRAEL
Telsys Ltd.
Dvora Hanevia Str.
Neve Sharet, Atidim Bldg. 3
Tel-Aviv 61 431 Israel
TEL: 03-492001-11
FAX: 03-497407

ITALY
Comprel S.P.A.
Via Po, 37
20031 – Cesano Maderno
Milano
TEL: (39) 3-62553991
FAX: (39) 3-62553967

Avnet EMG Division De Mico
Viale Vittorio Veneto, 8
20060 Cassina De Pecchi
Milano
TEL: (39) 02-95343600
FAX: (39) 02-95344371

JAPAN
Ado Electronic Indust. Co., Ltd.
18-10, Sotokanda 2-Chome
Chiyoda-ku
Tokyo 101
TEL: (81) 3-3257-2614
FAX: (81) 3-3257-1579

Macnica, Inc.
Hakusan High-Tech Park
1-22-2 Hakusan, Midori-ku
Yokohama, 226
TEL: (81) 45-939-6140
FAX: (81) 45-939-6141

Hakuto Co., Ltd.
1-13, Shinjuku 1-Chome
Shinjuku-ku, Tokyo 160
TEL: (81) 3-3355-7617
FAX: (81) 3-3355-7680

Hoei Denki Co., Ltd.
6-60, Niitaka 2-Chome
Yodogawa-Ku, Osaka 532
TEL: (81) 6-394-4596
FAX: (81) 6-396-5647

KOREA
Ellen & Company
5FL, IL Heung Sporex Bldg.
1490-25, Seocho-Dong
Seocho-ku, Seoul 137-070
TEL: (82) 2-523-2220
FAX: (82) 2-523-2345

Woo Young Tech Co., Ltd.
5th Fl. Koami Bldg.
13-31 Yoido-dong
Youngdeungpo-Ku, Seoul
TEL: (82) 2-369-7099
FAX: (82) 2-369-7091

NETHERLANDS
Alcom Electronics B.V.
P.O. Box 358
2900 AJ Capelle A/D Ijssel
TEL: (31) 10 4519533
FAX: (31) 10 4586482

NORWAY
Jakob Hatterland Electronic A/S
N-5578 Nedre Vats
TEL: (47) 53 76 30 00
FAX: (47) 53 76 53 39

POLAND
HT-EUREP Electronic sp.z.o.o.
WG Electronics
ul. Nowogrodzka 42/3
00-695 Warszawa
TEL: (48) 2-621 77 04
FAX: (48) 2-628 48 50

SINGAPORE
Technology Distribution
No. 1 Syed Alwi Rd. #05-02
Song Lin Bldg., Singapore 0620
TEL: (65) 299-7811
FAX: (65) 294-1518

SOUTH AFRICA
Pace Electronic Components
Cnr. Vanacht & Gewel St.
Isando 1600, P.O Box 701
TEL: (27) 11 974 1525
FAX: (27) 11 392 2463

SPAIN
Matrix Electronica
C/Belmonte de Tajo, 76-30 B
28019 Madrid
TEL: (34) 1 560 2737
FAX: (34) 1 565 2865

SWEDEN
Pelcon Electronics
Box 6023
Girovägen 13
175 06 Järfälla
TEL: (46) 8 795 9870
FAX: (46) 8 760 7685

SWITZERLAND
Avnet / E2000
Böhnirainstr. 11
CH-8801 Thalwil
TEL: (41) 1-7221330
FAX: (41) 1-7221340

Eurodis Electronic AG
Bahnstrasse 58/60
CH-8105 Regensdorf
TEL: (41) 01 843 31 11
FAX: (41) 01 843 34 75

Eurodis Electronic AG
Täfernstrasse 37
CH-5405 Baden-Dättwil
TEL: (41) 5684-0171
FAX: (41) 5683-3454

TAIWAN
Master Electronics
16F, No.182, Sec. 2
Tun-Hwa South Rd.
Taipei
TEL: (886) 02-732-3002
FAX: (886) 02-735-0902

Score Zap Industry
1F, No. 26, Lane 60
Wen Hu Street
Nei Hu, Taipei 114
TEL: (886) 2-627-7045
FAX: (886) 2-659-0089

UNITED KINGDOM
Micro Call
17 Thame Park Rd.
Thame, Oxon 0X9 3XD
TEL: (44) 84 426-1939
FAX: (44) 84 426-1678

Silicon Concepts, Ltd.
PEC Lynchborough Rd.
Passfield, Liphook
Hampshire GU30 7SB
TEL: (44) 428 751617
FAX: (44) 428 751603

Silicon Concepts, Ltd.
Meridale, Welsh Street
Chepstow, Gwent, NP6 5LR
TEL: (44) 291-624101
FAX: (44) 291-629878

Future Electronics
Future House
Poyle Road
Colnbrook, Berkshire SL3 0EZ
TEL: (44) 753-687000
FAX: (44) 753-689100

Sales Offices

NORTH AMERICAN DISTRIBUTORS

ALABAMA
Arrow Electronics
1015 Henderson Rd.
Huntsville, AL 35816
(205) 837-6955

Hamilton Hallmark
4890 University Square
Suite 1
Huntsville, AL 35816
(205) 837-8700

Insight Electronics
4835 University Square
Suite 19
Huntsville, AL 35818
(205) 830-1222

Marshall Industries
3313 Memorial Pkwy S.
Huntsville, AL 35801
(205) 881-9235

ARIZONA
Arrow Electronics
2415 W. Erie Drive
Tempe, AZ 85282
(602) 431-0030

Hamilton Hallmark
4637 S. 36th Place
Phoenix, AZ 85040
(602) 437-1200

Insight Electronics
1515 W. University Dr.
Suite #103
Tempe, AZ 85281
(602) 829-1800

Marshall Industries
9831S. 51st St. #C108
Phoenix, AZ 85044
(602) 496-0290

NORTHERN CALIFORNIA
Arrow Electronics
1180 Murphy Ave.
San Jose, CA 95131
(408) 441-9700

Arrow Electronics
48834 Kata Dr. #103
Fremont, CA 94538
(510) 490-9480

Hamilton Hallmark
580 Menlo Drive
Suite 2
Rocklin, CA 95765
(916) 624-9781

Hamilton Hallmark
2105 Lundy Ave.
San Jose, CA 95131
(408) 435-3500

Insight Electronics
1295 Oakmead Pkwy.
Sunnyvale, CA 94086
(408)720-9222

Marshall Industries
336 Los Coches St.
Milpitas, CA 95035
(408) 942-4600

Marshall Industries
3039 Kilgore Ave. #140
Rancho Cordova, CA 95670
(916) 635-9700

Zeus Electronics
6276 San Ignacio Ave. Ste. E
San Jose, CA 95119
(408) 629-4789

SOUTHERN CALIFORNIA
Arrow Electronics
Malibu Canyon Bus. Park
26677 W. Agoura Road
Calabasas, CA 91302
(818) 880-9686

Arrow Electronics
6 Cromwell, Suite 100
Irvine, CA 92718
(714) 587-0404

Arrow Electronics
9511 Ridgehaven Ct.
San Diego, CA 92123
(619) 565-4800

Hamilton Hallmark
4545 Viewridge Ave.
San Diego, CA 92123
(619) 571-7540

Hamilton Hallmark
3170 Pullman St.
Costa Mesa, CA 92626
(714) 641-4100

Hamilton Hallmark
10950 Washington Blvd.
Culver City, CA 90232
(310) 558-2800

Hamilton Hallmark
21150 Califa St.
Woodland Hills, CA 91367
(818) 594-0404

Insight Electronics
4333 Park Terrace Dr.
Suite 101
Westlake Village, CA 91361
(818) 707- 2101

Insight Electronics
9980 Huennekens St.
San Diego, CA 92121
(619) 587-1100

Insight Electronics
2 Venture Plaza
Suite 340
Irvine, CA 92718
(714) 727-3291

Marshall Industries
26637 Agoura Rd.
Calabasas, CA 91302
(818) 878-7000

Marshall Industries
9320 Telstar Ave.
El Monte, CA 91731-3004
(818) 307-6000

Marshall Industries
One Morgan
Irvine, CA 92718
(714) 458-5301

Marshall Industries
5961 Kearny Villa Rd.
San Diego, CA 92123
(619) 627-4140

Zeus Electronics
22700 Savi Ranch Pkwy.
Yorba Linda, CA 92687
(714) 921-9000

COLORADO
Arrow Electronics
61 Inverness Drive East
Suite 105
Englewood, CO 80112
(303) 799-0258

Hamilton Hallmark
12503 E. Euclid Drive
Suite 20
Englewood, CO 80111
(303) 790-1662

Insight Electronics
384 Inverness Drive South
Suite 105
Englewood, CO 80112
(303) 649-1800

Marshall Industries
12351 N. Grant
Thornton, CO 80241
(303) 451-8383

CONNECTICUT
Arrow Electronics
12 Beaumont Rd.
Wallingford, CT 06492
(203) 265-7741

Hamilton Hallmark
125 Commerce Ct.
Unit 6
Chesire, CT 06410
(203) 271-2844

Marshall Industries
20 Sterling Dr.
PO Box 200
Wallingford, CT 06492
(203) 265-3822

FLORIDA
Arrow Electronics
400 Fairway Dr.
Deerfield Beach, FL 33441
(305) 429-8200

Arrow Electronics
37 Skyline Dr.
Bldg. D, Suite 3101
Lake Mary, FL 32746
(407) 333-9300

Hamilton Hallmark
10491 72nd St. North
Largo, FL 34637
(813) 541-7440

Hamilton Hallmark
3350 NW 53rd St.
Suite 105-107
Ft. Lauderdale, FL 33309
(305) 484-5482

Hamilton Hallmark
7079 University Blvd.
Winter Park, FL 32792
(407) 657-3300

Insight Electronics
600 S. Northlake Blvd.
Suite 250
Altamonte Springs, FL 32071
(407) 834-6310

Insight Electronics
1 Park Place
621 NW 53rd St., Suite 240
Boca Raton, FL 33487
(407) 995-1486

Insight Electronics
13573 58th St. N., Suite 149
Clearwater, FL 34620
(813) 538-4191

Marshall Industries
380 S. Northlake Rd. #1024
Altamonte Springs, FL 32701
(407) 767-8585

Marshall Industries
2700 Cypress Ck. Rd. #D114
Ft. Lauderdale, FL 33309
(305) 977-4880

Marshall Industries
2840 Scherer Dr. #410
St. Petersburg, FL 33716
(813) 573-1399

Zeus Electronics
37 Skyline Dr. Bldg. D
Suite 1301
Lake Mary, FL 32746
(407) 333-3055

GEORGIA
Arrow Electronics
4205E River Green Pkwy.
Duluth, GA 30136
(404) 497-1300

Hamilton Hallmark
3425 Corporate Way
Suite A
Duluth, GA 30136-2552
(404) 623-4400

Insight Electronics
2400 Pleasant Hill Rd.
Suite 200
Duluth, GA 30136
(404) 717-8566

Marshall Industries
5300 Oakbrook Pkwy #140
Norcross, GA 30093
(404) 923-5750

IOWA

Arrow Electronics
375 Collins Rd. NE
Cedar Rapids, IA 52402
(319) 395-7230

Hamilton Hallmark
2335-A Blairs Ferry Rd. NE
Cedar Rapids, IA 52402
(319) 393-0033

ILLINOIS

Arrow Electronics
1140 W. Thorndale Ave.
Itasca, IL 60143
(708) 250-0500

Hamilton Hallmark
1130 Thorndale Ave.
Bensonville, IL 60106
(708) 860-7780

Insight Electronics
1365 Wiley Rd., Suite 142
Schaumberg, IL 60173
(708) 885-9700

Marshall Industries
50 E. Commerce Dr. # 1
Schaumberg, IL 60173
(708) 490-0155

Zeus Electronics
1140 West Thorndale Ave.
Itasca, IL 60143
(708) 595-9730

INDIANA

Arrow Electronics
7108 Lakeview Pkwy. W. Dr.
Indianapolis, IN 46268
(317) 299-2071

Hamilton Hallmark
655 W. Carmel Dr. #160
Carmel, IN 46032
(317) 575-3500

KANSAS

Arrow Electronics
9801 Legler Road
Lenexa, KS 66214
(913) 541-9542

Hamilton Hallmark
10809 Lakeview Drive
Lenexa, KS 66215
(913) 888-4747

Marshall Industries
10413 W. 84th Ter.
Pine Ridge Business Park
Lenexa, KS 66214
(913) 492-3121

MARYLAND

Arrow Electronics
9800J Patuxent Wood Dr.
Columbia, MD 21046
(301) 596-7800

Hamilton Hallmark
10240 Old Columbia Rd.
Columbia, MD 21046
(410) 988-9800

Marshall Industries
9130B Guilford
Columbia, MD 21046
(301) 470-2800

MASSACHUSETTS

Arrow Electronics
25 Upton Dr.
Wilmington, MA 01887
(508) 658-0900

Hamilton Hallmark
10 P Centennial Dr.
Peabody, MA 01960
(508) 532-9808

Insight Electronics
55 Cambridge St.
Suite 301
Burlington, MA 01803
(617) 270-9400

Marshall Industries
33 Upton Dr.
Wilmington, MA 01887
(508) 658-0810

Zeus Electronics
25 Upton Dr.
Wilmington, MA 01887
(508) 658-4776

MICHIGAN

Arrow Electronics
44720 Helm St.
Plymouth, MI 48170
(313) 455-0850

Hamilton Hallmark
44191 Plymouth Oaks Blvd.
#1300
Plymouth, MI 48171
(313) 416-5800

Marshall Industries
31067 Schoolcraft
Livonia, MI 48150
(313) 525-5850

MINNESOTA

Arrow Electronics
10100 Viking Drive # 100
Eden Prairie, MN 55344
(612) 941-5280

Hamilton Hallmark
9401 James Ave. South
Suite 140
Bloomington, MN 55431
(612) 881-2600

Insight Electronics
5353 Gamble Rd.
Suite 330
St. Louis Park, MN 55416
(612) 525-9999

Marshall Industries
14800 28th Ave. N.
Suite 175
Plymouth, MN 55447
(612) 559-2211

MISSOURI

Arrow Electronics
2380 Schuetz Rd.
St. Louis, MO 63146
(314) 567-6888

Hamilton Hallmark
3783 Rider Trail South
Earth City, MO 63045
(314) 291-5350

Marshall Industries
3377 Hollenberg Dr.
Bridgeton, MO 63044
(314) 291-4650

NEW JERSEY

Arrow Electronics
4 East Stow Rd. Unit 11
Marlton,NJ 08053
(609) 596-8000

Arrow Electronics
43 Route 46 East
Pinebrook, NJ 07058
(201) 227-7880

Hamilton Hallmark
1 Keystone Ave. Bldg. 36
Cherry Hill, NJ 08003
(609) 424-0110

Hamilton Hallmark
10 Lanidex Plaza West
Parsippany, NJ 07054
(201) 515-1641

Marshall Industries
101 Fairfield Rd.
Fairfield, NJ 07006
(201) 882-0320

Marshall Industries
158 Gaither Dr.
Mt. Laurel, NJ 08054
(609) 234-9100

NEW YORK

Arrow Electronics
25 Hub Drive
Melville, NY 11747
(516) 391-1300

Arrow Electronics
20 Oser Ave.
Hauppauge, NY 11788
(516) 231- 1000

Arrow Electronics
3375 Brighton-Henrietta
Townline Rd.
Rochester, NY 14623
(716) 427-0300

Hamilton Hallmark
1057 E. Henrietta Rd.
Rochester, NY 14623
(716)475-9130

Hamilton Hallmark
390 Rabro Dr.
Hauppauge, NY 11788
(516) 434-7400

Hamilton Hallmark
3075 Veterans Memorial
Ronkonkoma, NY 11779
(516) 737-0600

Marshall Industries
97 Oser Ave.
Hauppauge, NY 11788
(516) 273-2695

Marshall Industries
1250 Scottsville Rd.
Rochester, NY 14624
(716) 235-7620

Marshall Industries
100 Marshall Drive
Endicott, NY 13790
(607) 785-2345

Zeus Electronics
100 Midland Ave.
Port Chester, NY 10573
(914) 937-7400

NORTH CAROLINA

Arrow Electronics
5240 Greens Dairy Rd.
Raleigh, NC 27604
(919) 876-3132

Hamilton Hallmark
5234 Green's Dairy Road
Raleigh, NC 27604
(919) 872-0712

Marshall Industries
5224 Greens Dairy Rd.
Raleigh, NC 27604
(919) 878-9882

OHIO

Arrow Electronics
6573E Cochran Rd.
Solon, OH 44139
(216) 248-3990

Arrow Electronics
8200 Washington Village Dr. #A
Centerville, OH 45458
(513) 435-5563

Hamilton Hallmark
777 Dearborne Park Lane
Suite L
Worthington, OH 43085
(614) 888-3313

Hamilton Hallmark
5821 Harper Road
Solon, OH 44139
(216) 498-1100

Hamilton Hallmark
7760 Washington Village Dr.
Dayton, OH 45459
(513) 439-6735

Sales Offices

Marshall Industries
3520 Park Center Dr.
Dayton, OH 45414
(513) 898-4480

Marshall Industries
30700 Bainbridge Rd. Unit A
Solon, OH 44139
(216) 248-1788

OKLAHOMA
Arrow Electronics
12111 East 51st St. #101
Tulsa, OK 74146
(918) 252-7537

Hamilton Hallmark
5411 S. 125th E Ave.,Ste. 305
Tulsa, OK 74146
(918) 254-6110

OREGON
Almac-Arrow Electronics
1885 N.W. 169th Place
Beaverton, OR 97006
(503) 629-8090

Hamilton Hallmark
9750 SW Nimbus Ave.
Beaverton, OR 97005
(503) 526-6200

Insight Electronics
8705 SW Nimbus Ave. Ste. 200
Beaverton, OR 97005
(503) 644-3300

Marshall Industries
9705 SW Gemini Dr.
Beaverton, OR 97005
(503) 644-5050

TEXAS
Arrow Electronics
11500 Metric Blvd.
Suite 160
Austin, TX 78758
(512) 835-4180

Arrow Electronics
3220 Commander Dr.
Carrollton, TX 75006
(214) 380-6464

Arrow Electronics
19416 Park Row #190
Houston, TX 77084
(713) 530-4700

Hamilton Hallmark
11420 Pagemill Road
Dallas, TX 75243
(214) 553-4300

Hamilton Hallmark
12211 Technology Blvd.
Austin, TX 78727
(512) 258-8848

Hamilton Hallmark
8000 Westglen
Houston, TX 77063
(713) 781-6100

Insight Electronics
11500 Metric Blvd. #215
Austin, TX 78758
(512) 719-3090

Insight Electronics
1778 Plano Rd. #320
Richardson, TX 75081
(214) 783-0800

Insight Electronics
15437 McKaskle
Sugarland, TX 77478
(713) 448-0800

Marshall Industries
8504 Cross Park Dr.
Austin, TX 78754
(512) 837-1991

Marshall Industries
10681 Haddington
Suite 160
Houston, TX 77043
(713) 467-1666

Marshall Industries
1551 N. Glenville Dr.
Richardson, TX 75081
(214) 705-0600

Zeus Electronics
3220 Commander Dr.
Carrollton, TX 75006
(214) 380-4330

UTAH
Arrow Electronics
1946 West Parkway Blvd.
Salt Lake City, UT 84119
(801) 973-6913

Hamilton Hallmark
1100 E. 6600 South
Suite 120
Salt Lake City, UT 84121
(801) 266-2022

Insight Electronics
545 E. 4500 South
Suite E 110
Salt Lake City, UT 84117
(801) 288-9001

Marshall Industries
2355 South 1070 West
Suite D
Salt Lake City, UT 84119
(801) 973-2288

WASHINGTON
Almac-Arrow Electronics
14360 S.E. Eastgate Way
Bellevue, WA 98007
(509) 643-9992

Hamilton Hallmark
8630 154th Ave.
Redmond, WA 98052
(206) 881-6697

Insight Electronics
12002 115th Avenue, NE
Kirkland, WA 98034
(206) 820-8100

Marshall Industries
11715 N. Creek Pkwy. S.
Suite 112
Bothell, WA 98011
(509) 486-5747

WISCONSIN
Arrow Electronics
200 North Patrick Blvd.
Brookfield, WI 53045
(414) 792-0150

Hamilton Hallmark
2440 South 179th St.
New Berlin, WI 53146
(414) 797-7844

Marshall Industries
20900 Swenson Dr. #150
Waukesha, WI 53186
(414) 797-8400

CANADA

ALBERTA
Future Electronics
3833-29th St. NE
Calgary, Alberta T1Y 6B5
(403) 250-5550

Future Electronics
4606-97th Street
Edmonton, Alberta T6E 5N9
(403) 438-2858

BRITISH COLUMBIA
Arrow Electronics
8544 Baxter Place
Burnaby, British Columbia
V5A 4T8
(604) 421-2333

Future Electronics
1695 Boundary Road
Vancouver, British Columbia
V5K 4X7
(604) 294-1166

Hamilton Hallmark
8610 Commerce Ct.
Burnaby, British Columbia
V5A 4N6
(604) 420-4101

MANITOBA
Future Electronics
106 King Edward
Winnipeg, Manitoba R3H 0N8
(204) 786-7711

ONTARIO
Arrow Electronics
36 Antares Dr. Unit 100
Nepean, Ontario K2E 7W5
(613) 226-6903

Arrow Electronics
1093 Meyerside Dr.
Mississauga, Ontario L5P 1M4
(416) 670-7769

Future Electronics
1050 Baxter Road
Ottawa, Ontario K2C 3P2
(613) 820-8313

Future Electronics
5935 Airport Rd., #200
Mississauga, Ontario L4V 1W5
(905) 612-9200

Hamilton Hallmark
151 Superior Blvd.
Unit 1-6
Mississauga, Ontario L5T 2L1
(905) 564-6060

Hamilton Hallmark
190 Colonnade Rd.
Nepean, Ontario K2E 7J5
(613) 226-1700

Marshall Industries
6285 Northern Dr. #112
Mississauga, Ontario L4V 1X5
(905) 612-1771

QUEBEC
Arrow Electronics
1100 St. Regis Blvd.
Dorval, Quebec H9P 2T5
(514) 421-7411

Future Electronics
237 Hymus Blvd.
Pointe Claire, Quebec H9R 5C7
(514) 694-7710

Future Electronics
1000 St-Jean Babtiste #100
Quebec City, Quebec G2E 5G5
(418) 877-6666

Hamilton Hallmark
600 Transcanada Hwy
Suite 600
Ville St. Laurent, Quebec
H4T 1V6
(514) 335-1000

Marshall Industries
148 Brunswick Blvd.
Pointe Claire, Quebec H9R 5B9
(514) 694-8142

Index

Notes