ABEL Design Manual

Version 8.0

Technical Support Line: 1- 800-LATTICE or (408) 428-6414
DSNEXP-ABL-DM Rev 8.0.1

Copyright

This document may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Lattice Semiconductor Corporation.

The software described in this manual is copyrighted and all rights are reserved by Lattice Semiconductor Corporation. Information in this document is subject to change without notice.

The distribution and sale of this product is intended for the use of the original purchaser only and for use only on the computer system specified. Lawful users of this product are hereby licensed only to read the programs on the disks, cassettes, or tapes from their medium into the memory of a computer solely for the purpose of executing them. Unauthorized copying, duplicating, selling, or otherwise distributing this product is a violation of the law.

Trademarks

The following trademarks are recognized by Lattice Semiconductor Corporation:

Generic Array Logic, ISP, ispANALYZER, ispATE, ispCODE, ispDCD, ispDOWNLOAD, ispDS, ispDS+, ispEXPERT, ispGDS, ispGDX, ispHDL, ispJTAG, ispSmartFlow, ispStarter, ispSTREAM, ispSVF, ispTA, ispTEST, ispTURBO, ispVECTOR, ispVerilog, ispVHDL, ispVM, Latch-Lock, LHDL, pDS+, RFT, and Twin GLB are trademarks of Lattice Semiconductor Corporation.

E^2CMOS, GAL, ispGAL, ispLSI, pDS, pLSI, Silicon Forest, and UltraMOS are registered trademarks of Lattice Semiconductor Corporation.

Project Navigator is a trademark of Data I/O Corporation. ABEL-HDL is a registered trademark of Data I/O Corporation.

Microsoft, Windows, and MS-DOS are registered trademarks of Microsoft Corporation.

IBM is a registered trademark of International Business Machines Corporation.

Lattice Semiconductor Corporation
5555 NE Moore Ct.
Hillsboro, OR 97124
(503) 268-8000

December 1999

Limited Warranty

Lattice Semiconductor Corporation warrants the original purchaser that the Lattice Semiconductor software shall be free from defects in material and workmanship for a period of ninety days from the date of purchase. If a defect covered by this limited warranty occurs during this 90-day warranty period, Lattice Semiconductor will repair or replace the component part at its option free of charge.

This limited warranty does not apply if the defects have been caused by negligence, accident, unreasonable or unintended use, modification, or any causes not related to defective materials or workmanship.

To receive service during the 90-day warranty period, contact Lattice Semiconductor Corporation at:

Phone: 1-800-LATTICE
Fax: (408) 944-8450
E-mail: applications@latticesemi.com

If the Lattice Semiconductor support personnel are unable to solve your problem over the phone, we will provide you with instructions on returning your defective software to us. The cost of returning the software to the Lattice Semiconductor Service Center shall be paid by the purchaser.

Limitations on Warranty

Any applicable implied warranties, including warranties of merchantability and fitness for a particular purpose, are hereby limited to ninety days from the date of purchase and are subject to the conditions set forth herein. In no event shall Lattice Semiconductor Corporation be liable for consequential or incidental damages resulting from the breach of any expressed or implied warranties.

Purchaser’s sole remedy for any cause whatsoever, regardless of the form of action, shall be limited to the price paid to Lattice Semiconductor for the Lattice Semiconductor software.

The provisions of this limited warranty are valid in the United States only. Some states do not allow limitations on how long an implied warranty lasts, or exclusion of consequential or incidental damages, so the above limitation or exclusion may not apply to you.

This warranty provides you with specific legal rights. You may have other rights which vary from state to state.

Table of Contents

Preface 7
- What is in this Manual 8
  - Where to Look for Information 8
- Documentation Conventions 9
- Related Documentation 10

Chapter 1 ABEL-HDL Overview 11
- Programmable Design in ispDesignExpert 12
  - What is Programmable Designing? 12
  - What is ABEL-HDL? 14
- Overview of Design 15
  - Projects. 15
  - Project Sources 16
  - Design Hierarchy 17
  - Design Compilation 17
  - Design Simulation. 17
  - Device Programming 17

Chapter 2 ABEL-HDL Hierarchical Designs 18
- Why Use Hierarchical Design? 19
- Approaches to Hierarchical Design 19
  - Creating a new Hierarchical Design 19
    - Top-down Design 20
    - Bottom-up Design 20
    - Inside-out (Mixed) Design 20
- Specifying a Lower-level Module 20

Chapter 3 Compiling ABEL-HDL Designs 23
- Overview of ABEL-HDL Compiling 24
  - Design Entry. 24
    - Creating a Design Using ABEL-HDL Sources 24
  - Design Compliation 29
    - Keeping Track of Process: Auto-update 29
    - Compiling an ABEL-HDL Source File 30
    - Using Properties and Strategies 31
  - Design Simulation. 34

Chapter 4 ABEL-HDL Design Considerations . . . 37
Overview of ABEL-HDL Design Considerations . . . 38
Hierarchy in ABEL-HDL . . . 38
Instantiating a Lower-level Module in an ABEL-HDL Source . . . 39
Identifying I/O Ports in the Lower-level Module. . . . 39
Declaring Lower-level Modules in the Top-level Source . . . 40
Instantiating Lower-level Modules in Top-level Source . . . 40
Hierarchy and Retargeting and Fitting . . . 41
Redundant Nodes . . . 41
Merging Feedbacks. . . . 41
Post-linked Optimization . . . 41
Hierarchical Design Considerations. . . . 42
Prevent Node Collapsing . . . 42
Node Collapsing. . . . 42
Selective Collapsing . . . 42
Pin-to-pin Language Features . . . 43
Device-independence vs. Architecture-independence . . . 43
Signal Attributes . . . 43
Signal Dot Extensions . . . 43
Pin-to-pin vs. Detailed Descriptions for Registered Designs . . . 44
Using := for Pin-to-pin Descriptions . . . 44
Resolving Ambiguities. . . . 44
Detailed Circuit Descriptions . . . 45
Detailed Descriptions: Designing for Macrocells. . . . 45
Examples of Pin-to-pin and Detailed Descriptions . . . 47
Pin-to-pin Module Description . . . 47
Detailed Module Description . . . 47
Detailed Module with Inverted Outputs. . . . 48
When to Use Detailed Descriptions . . . 50
Using := for Alternative Flip-flop Types . . . 50
Using Active-low Declarations . . . 51
Polarity Control . . . 53
Polarity Control with Istype . . . 53
Using Istype ‘neg’, ‘pos’, and ‘dc’ to Control Equation and Device Polarity . . . 53
Using ‘invert’ and ‘buffer’ to Control Programmable Inversion . . . 54
Flip-flop Equations . . . 54
Feedback Considerations — Dot Extensions . . . 55
Dot Extensions and Architecture-Independence . . . 56
Dot Extensions and Detail Design Descriptions . . . 58
Using Don’t Care Optimization. . . . 60
Exclusive OR Equations . . . 62
Optimizing XOR Devices . . . 62
Using XOR Operators in Equations . . . 62
Using Implied XORs in Equations . . . 62
Using XORs for Flip-flop Emulation . . . 63
JK Flip-Flop Emulation . . . 63

State Machines 65
Use Identifiers Rather Than Numbers for States 65
Powerup Register States 67
Unsatisfied Transition Conditions 67
D-Type Flip-Flops 67
Other Flip-flops 68
Precautions for Using Don’t Care Optimization 68
Number Adjacent States for One-bit Change 72
Use State Register Outputs to Identify States 72
State Register Bit Values 73
Using Symbolic State Descriptions 74
Symbolic Reset Statements 74
Symbolic Test Vectors 75
Using Complement Arrays 75
ABEL-HDL and Truth Tables 77
Basic Syntax - Simple Examples 78
Influence of Signal polarity 79
Using .X. in Truth tables conditions 80
Using .X. on the right side 81
Special case: Empty ON-set 82
Registered Logic in Truth tables 82

Preface

This manual provides information on ABEL-HDL design sources, hierarchical structure, compiling, and design considerations. It is assumed that you have a basic understanding of ABEL-HDL design.

What is in this Manual

This manual contains the following information:

- Introduction to ABEL-HDL design
- Hierarchical design in ABEL-HDL
- ABEL-HDL compiling
- ABEL-HDL design considerations

Where to Look for Information

Chapter 1, ABEL-HDL Overview – Provides an overview of ABEL-HDL designs.

Chapter 2, ABEL-HDL Hierarchical Designs – Discusses the hierarchical structure in ABEL-HDL designs.

Chapter 3, Compiling ABEL-HDL Designs – Provides information on the compiling of ABEL-HDL designs.

Chapter 4, ABEL-HDL Design Considerations – Discusses the design considerations in ABEL-HDL designs.

Documentation Conventions

This user manual follows the typographic conventions listed here:

Convention	Definition and Usage
Italics	Italicized text represents variable input. For example: `design.1` This means you must replace *design* with the file name you used for all the files relevant to your design. Valuable information may be italicized for emphasis. Book titles also appear in italics. The beginning of a procedure appears in italics. For example: *To run the functional simulation:*
Bold	Valuable information may be boldfaced for emphasis. Commands are shown in boldface. For example: Select **File** ⇒ **Open** from the Waveform Viewer.
`Courier Font`	Monospaced (Courier) font indicates file and directory names and text that the system displays. For example: The `C:\isptools\ispsys\config` subdirectory contains...
`Bold Courier`	Bold Courier font indicates text you type in response to system prompts. For example: **`SET YBUS [Y0..Y6];`**
|...|	Vertical bars indicate options that are mutually exclusive; you can select only one. For example: `INPUT|OUTPUT|BIDI`
"Quotes"	Titles of chapters or sections in chapters in this manual are shown in quotation marks. For example: See Chapter 1, "Introduction."
✍ ***NOTE***	Indicates a special note.
▲ ***CAUTION***	**Indicates a situation that could cause loss of data or other problems.**
❖ ***TIP***	Indicates a special hint that makes using the software easier.
⇒	Indicates a menu option leading to a submenu option. For example: **File** ⇒ **New**

Related Documentation

In addition to this manual, you might find the following reference material helpful:

- *ispDesignExpert User Manual*
- *ispDesignExpert Tutorial*
- *ABEL-HDL Reference Manual*
- *Schematic Entry User Manual*
- *Design Verification Tools User Manual*
- *ispLSI Macro Library Reference Manual*
- *ispLSI 5K/8K Macro Library Supplement*
- *ISP Daisy Chain Download User Manual*
- *ispEXPERT Compiler User Manual*
- *VHDL and Verilog Simulation User Manual*

These books provide technical specifications for the LSC device families and give helpful information on device use and design development.

Chapter 1 *ABEL-HDL Overview*

This chapter covers the following topics:

- Programmable Design in ispDesignExpert
- Overview of Design

Programmable Design in ispDesignExpert

What is Programmable Designing?

Programmable designing is creating a design that can be implemented into a programmable device. PLDs (Programmable Logic Devices) and CPLDs (Complex PLDs) are a few examples of programmable devices.

Figure 1-1 shows an example Design. This design has lower-level ABEL-HDL files (not shown).

```
MODULE twocnt
TITLE 'two counters having a race'
"Demonstrates ability to use multiple levels of ABEL-HDL Hierarchy,
"and to collapse lower-level module nodes into upper level modules.
"For example, each counter has four REGISTER nodes, and this module
"has four COMBINATORIAL pins. The lower-level registers are
“correctly flattened into the top-level combinatorial outputs. No
“dot extensions are used, allowing the system to determine the best
“feedback path to use. This design uses the advanced fit properties
“REMOVE REDUNDANT NODES and MERGE EQUIVALENT FEEDBACK NODES.
"Constants
c,x = .c.,.x.;

"Inputs
             clk, en1, en2, rst                    pin ;
"Outputs
             a3, a2, a1, a0, b3, b2, b1, b0        pin ;
             ov1, ov2                              pin istype
'reg,buffer';
"Submodule declarations
             hiercnt interface (clk,rst,en -> q3, q2, q1, q0);
"Submodule instances
             cnt1 functional_block hiercnt;
             cnt2 functional_block hiercnt;
```

Figure 1-1. Example of a Top-level ABEL-HDL source for a Design

```
Equations
              cnt1.clk = clk;
              cnt2.clk = clk;
              cnt1.rst = rst;
              cnt2.rst = rst;
              cnt1.en = en1;
              cnt2.en = en2;
"Each counter may be enabled independent of the other. This module
may be used as a Sub-module for a higher-level design, as these
counters may be cascaded by feeding the ovoutputs to the en inputs
of the next stage.
              ov1.clk = clk;
              ov2.clk = clk;

              ov1 := a3 & a2 & a1 & !a0 & en1;  "look-ahead carry -
overflow
              ov2 := b3 & b2 & b1 & !b0 & en2;  "indicator
              a3 = cnt1.q3;  a2 = cnt1.q2;  a1 = cnt1.q1;   a0 =
cnt1.q0;
              b3 = cnt2.q3;  b2 = cnt2.q2;  b1 = cnt2.q1;   b0 =
cnt2.q0;
test_vectors
([clk,rst,en1,en2] -> [a3,a2,a1,a0,b3,b2,b1,b0,ov1,ov2])
 [ 0  , 0, 0  , 0   ] -> [ x, x, x, x, x, x, x, x, x,  x ];
 [ c  , 1, 0  , 0   ] -> [ 0, 0, 0, 0, 0, 0, 0, 0, 0,  0 ];
 [ c  , 0, 1  , 0   ] -> [ 0, 0, 0, 1, 0, 0, 0, 0, 0,  0 ];
 [ c  , 0, 1  , 0   ] -> [ 0, 0, 1, 0, 0, 0, 0, 0, 0,  0 ];
 [ c  , 0, 1  , 0   ] -> [ 0, 0, 1, 1, 0, 0, 0, 0, 0,  0 ];
 [ c  , 0, 0  , 1   ] -> [ 0, 0, 1, 1, 0, 0, 0, 1, 0,  0 ];
 [ c  , 0, 0  , 1   ] -> [ 0, 0, 1, 1, 0, 0, 1, 0, 0,  0 ];
END
```

Figure 1-1. Example of a Top-level ABEL-HDL source for an Design (Continued)

What is ABEL-HDL?

ABEL-HDL is a hierarchical logic description language. ABEL-HDL design descriptions are contained in an ASCII text file in the ABEL Hardware Description Language (ABEL-HDL). For example, the following ABEL-HDL code describes a one-bit counter block:

```
MODULE obcb
TITLE 'One Bit Counter Block'
"Inputs
  clk, rst, ci      pin ;
"Outputs
  co                pin istype 'com';
  q                 pin istype 'reg';
Equations
  q.clk = clk;
  q := !q.fb & ci & !rst        "toggle if carry in and not reset
     # q.fb & !ci & !rst          "hold if not carry in and not reset
     # 0 & rst;                 "go to 0 if reset
  co = q.fb & ci;               "carry out is carry in and q = 1
END
```

For detailed information about the ABEL-HDL language, refer to the ***ABEL-HDL Reference Manual*** and the online help of ispDesignExpert. An online version of the *ABEL-HDL Reference Manual* is provided in the ispDesignExpert CD (accessible by selecting **Help** ⇒ **Manuals** from the ispDesignExpert Project Navigator).

Overview of Design

With ispDesignExpert, you can create and test designs that will be physically implemented into Programmable devices. ispDesignExpert uses the Project Navigator interface (Figure 1-2) as the front-end to all the design tools which creates an integrated design environment that links together design, simulation, and place-and-route tools.

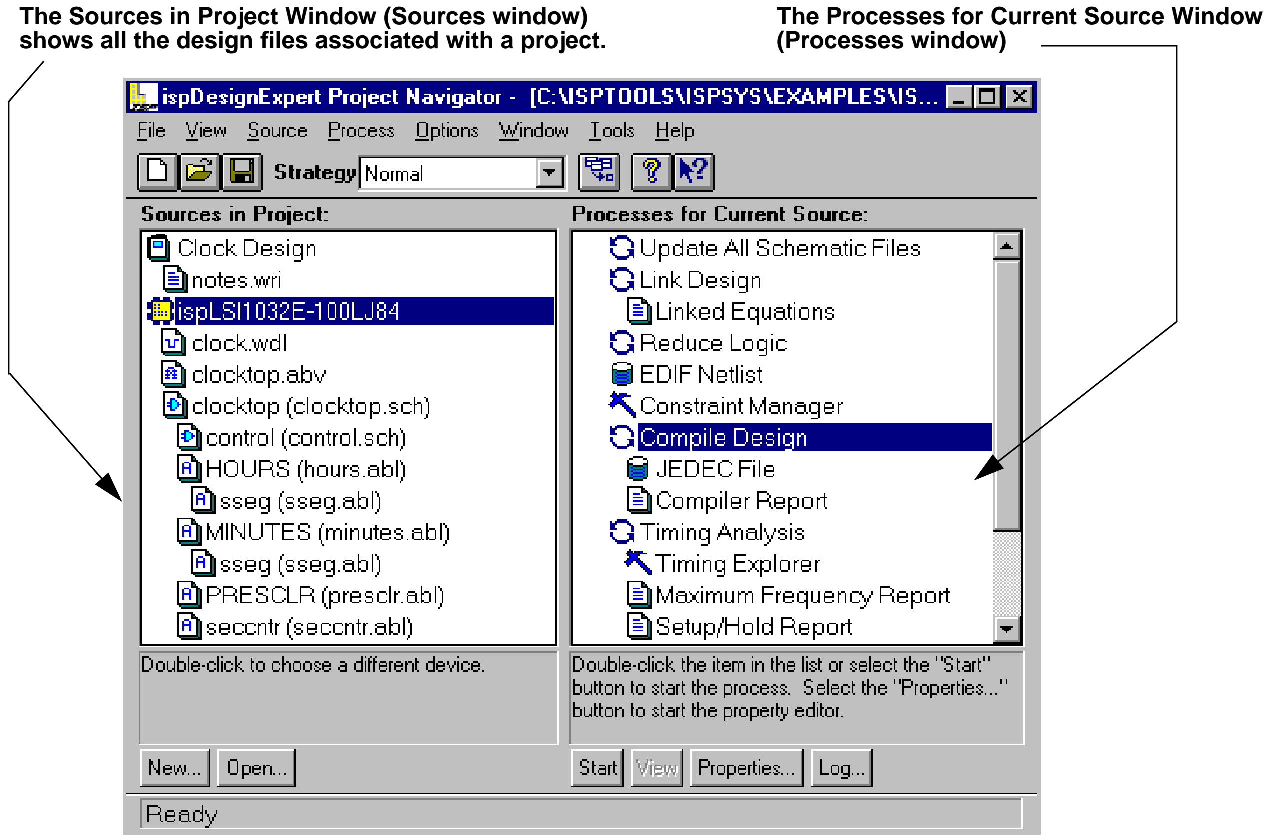

Figure 1-2. ispDesignExpert Project Navigator

Projects

In ispDesignExpert, a single design is represented by a single project that is created and modified using the Project Navigator. The project contains all the logical descriptions for the design. In addition, the project can contain documentation files, and test files.

A project represents one design, but you have the option of targeting your design to a specific device. When you switch the target device, the processes and design flow in the Project Navigator changes to one that is appropriate for the new target device.

Project Sources

In ispDesignExpert, a project (design) consists of one or more source files.

Each type of source is identified by an icon and name in the Sources in Project window. The Sources in Project window is the large scrollable window on the left side of the Project Navigator display. The Sources in Project window lists all of the sources that are part of the project design.

In addition to the sources that describe the function of the design, every project contains at least two special types of sources: the project notebook and the device.

Project Notebook – The project notebook is where you enter the title and name of the project. You can also use the project notebook to keep track of external files (such as document files) that are related to your project.

Device – The device is a source that includes information about the currently selected device.

The supported sources are:

- ABEL-HDL module (`.abl`)
- schematic module(`.sch`)
- VHDL module (`.vhd`)
- Verilog HDL module (`.v`)
- test vector file (`.abv`)
- graphic waveform stimulus (`.wdl`)
- VHDL test bench (`.vhd`)
- Verilog test fixture (`.tf`)

Figure 1-3 shows the sources as they appear in the Project Navigator. The top-level ABEL-HDL file `RM12PS6K` contains INTERFACE statements that instantiate (links to) the lower-level ABEL-HDL files `PSSR8X16` and `RAM12`.

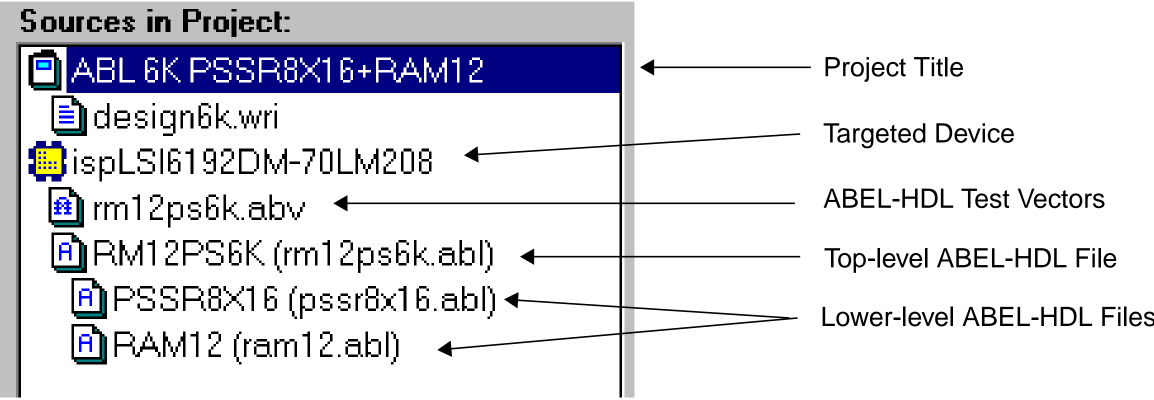

Figure 1-3. Sources in a Design Project

Design Hierarchy

When designs can be broken into multiple levels, this is called hierarchical designing. ispDesignExpert supports full hierarchical design, which permits you to create a design that is divided into multiple levels, either to clarify its function or permit the easy reuse of functional blocks. For instance, a large complex design does not have to be created as a single module. By using hierarchical designing, each component or piece of a complex design could be created as a separate module. Figure 1-3 shows a two-level hierarchy.

For more information on hierarchical designing, refer to **Chapter 2, "ABEL-HDL Hierarchical Designs"**.

Design Compilation

After design entry, you can compile your design using the ispEXPERT Compiler. The compiler first verifies designs for correct syntax, then optimizes and parittions designs, and fits logic and performs place-and-route to map the logic to specific devices, it finally generates a JEDEC fusemap file used to program the device and a netlist file for post-route simulation.

The compiler gathers all compilation results and writes this information to the ispEXPERT Compiler report that can be read using **Process** ⇒ **View** from the Project Navigator.

If an error occurs, the compiler stops and issues the auto-make log file (`automake.log`) in the Report Viewer. Using the log file information, you can change your design and recompile it.

Design Simulation

In ispDesignExpert, functional and timing simulation is available using ABEL-HDL Test Vector (`.abv`) files or Waveform Description Language (`.wdl`) files. The functional and timing simulator and Waveform Viewer enable you to verify your design before implementing it into a specific device. For more information on simulation, refer to the ***Design Verification Tools User Manual***.

Device Programming

After the compiler produces a fusemap of your finished design, the integrated ISP Download System in ispDesignExpert enables you to download the JEDEC device programming file to an ispLSI device using an ispDOWNLOAD cable. See the ***ISP Daisy Chain Download User Manual*** for more details.

Chapter 2 *ABEL-HDL Hierarchical Designs*

ispDesignExpert supports full hierarchical design. Hierarchical structuring permits a design to be broken into multiple levels, either to clarify its function or permit the easy reuse of lower-level sources. For instance, a large complex design does not have to be created as a single module. By using hierarchical design, each component or piece of a complex design could be created as a separate module.

A design is hierarchical when it is broken up into modules. For example, you could create a top-level ABEL-HDL describing a design. In the ABEL-HDL file, you could interface to lower-level modules that describe pieces of the design.

The module represented by the ABEL-HDL interface is said to be at one level below the ABEL-HDL file in which the INTERFACE statement appears. Regardless of how you refer to the levels, any design with more than one level is called a hierarchical design. In ispDesignExpert, there is no limit to the number of hierarchical levels a design can contain.

This chapter covers the following topics:

- Why Use Hierarchical Design?
- Approaches to Hierarchical Design
- Specifying a Lower-level Module in an ABEL-HDL Module

Why Use Hierarchical Design?

The primary advantage of hierarchical design is that it encourages modularity. For instance, a careful choice of the circuitry you select to be a module will give you a module that can be reused.

Another advantage of hierarchical design is the way it lets you organize your design into useful levels of abstraction and detail.

Approaches to Hierarchical Design

Hierarchical designs consists of **ONE** top-level. The lower-level modules can be of any supported source (ABEL-HDL sources) and are represented in the top-level module by a place-holder. You can create the top-level module first or create it after creating the lower-level modules. Figure 2-1 illustrates a two-level hierarchical project.

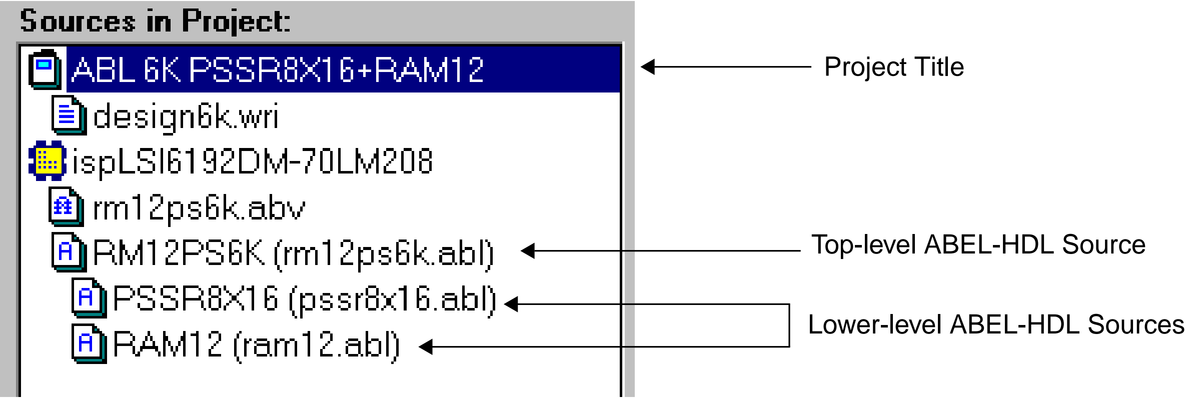

Figure 2-1. Example of a Hierarchical Project in the Project Navigator

Creating a new Hierarchical Design

Hierarchical entry is a convenient way to enter a large design one piece at a time. It is also a way of organizing and structuring your design and the design process. The choice of the appropriate methodology can speed the design process and reduce the chance of design or implementation errors.

There are three basic approaches to creating a multi-module hierarchical design:

- Top-down
- Bottom-up
- Inside-out (mixed)

Regardless of the approach you choose, you start from those parts of the design that are clearly defined and move up or down to those parts of the design that need additional definition.

The following sections explain the philosophy and techniques of each approach.

Top-down Design

In top-down design, you do not have to know all the details of your project when you start. You can begin at the top, with a general description of the circuit's functionality, then break the design into modules with the appropriate functions. This approach is called "stepwise refinement" – you move in order from a general description to modularized functions and to the specific circuits that perform those functions.

In a top-down design, the uppermost schematic usually consists of nothing but Block symbols representing modules (plus any needed power, clocking, or support circuitry). These modules are repeatedly broken down into simpler modules (or the actual circuitry) until the entire design is complete.

Bottom-up Design

In bottom-up design you start with the simplest modules, then combine them in schematics at increasingly higher levels. Bottom-up design is ideal for projects in which the top-level behavior cannot be defined until the low-level behavior is established.

Inside-out (Mixed) Design

Inside-out design is a hybrid of top-down and bottom-up design, combining the advantages of both. You start wherever you want in the project, building up and down as required.

ispDesignExpert fully supports the mixed approach to design. This means that you can work bottom-up on those parts of the project that must be defined in hardware first, and top-down on those parts with clear functional definitions.

Specifying a Lower-level Module

The following steps outline how to specify a lower-level module in a design module.

1. In a Text Editor, open your ABEL-HDL file (**File** ⇒ **Open**) or create a new ABEL-HDL file (**File** ⇒ **New**).
2. In the ABEL-HDL file, use the **INTERFACE** and **FUNCTIONAL_BLOCK** keywords to instantiate lower-level files.

You can also use the INTERFACE keyword in lower-level files to link to upper-level ABEL-HDL modules (not upper-level schematics).

You can place multiple instances of the same interface in the same design by using the FUNCTIONAL_BLOCK statement.

Refer to the ***ABEL-HDL Reference Manual*** for more information.

3. The interface must have same names as the pin names (ABEL-HDL) in the lower-level module.

   Figure 2-2, Figure 2-3 and Figure 2-4 show one upper-level ABEL-HDL module and different ways to implement the lower-level modules:

```
MODULE nand1

TITLE 'Hierarchical nand gate -
             Instantiates an and gate and a not gate.'

I1, I2, O1 pin;

" The following code defines the interfaces (components)
" and1 and not1.  And1 corresponds to the lower-
" level module AND1.vhd, AND1.ABL, or AND1.SCH.
" For component AND1, the IN1, IN2, and OUT1 interface names
" correspond to IN1, IN2, and OUT1 in the lower-level module.

and1 INTERFACE(IN1, IN2 -> OUT1);
not1 INTERFACE(IN1 -> OUT1);

" The following code defines the instances for the interfaces
" using the functional_block statement.  For the and1 interface,
" there is one instance named my_and.

my_and functional_block and1;
my_not functional_block not1;

EQUATIONS

             my_and.IN1 = I1;
             my_and.IN2 = I2;
             my_not.IN1 = andinst.OUT1;
             O1 = my_not.OUT1;

END
```

Figure 2-2. Top-level ABEL-HDL Module for NAND1

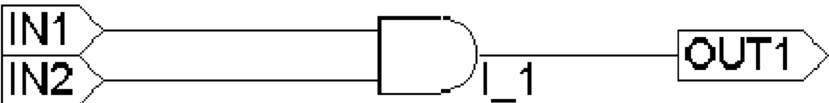

Figure 2-3. Lower-level Schematic for AND1 Interface

TIP If you are in a lower-level schematic, you can click the **Use Data From This Block** button in the New Block Symbol dialog box (**Add** ⇒ **New Block Symbol**) to automatically create a functional block symbol for the current schematic.

The name of the lower-level schematic must match the Block Name (schematic), or the interface name (ABEL-HDL) in the upper-level module. This associates the lowe-level module with the symbol representing it. The schematic (Figure 2-3) must be named `AND.sch`.

The nets in the lower-level schematic correspond to the pin names (schematics), or pine names (ABEL-HDL) in the upper-level module.

```
MODULE and1
TITLE 'and1 gate -
              Instantiated by nand1 - Simple hierarchy example'

" The pins must match the Symbol pins (schematic),
" or interface names (ABEL-HDL) in the upper-level module.

IN1, IN2, OUT1 pin;
EQUATIONS

              OUT1 = IN1 & IN2;

TEST_VECTORS

              ([ IN1, IN2] -> [OUT1])
               [   0,   0] -> [ 0];
               [   0,   1] -> [ 0];
               [   1,   0] -> [ 0];
               [   1,   1] -> [ 1];
END
```

Figure 2-4. Lower-level ABEL-HDL Module for AND1 Interface

❖ **TIP** It is best to create the lowest-level sources first and then import or create the higher-level sources.

Chapter 3 *Compiling ABEL-HDL Designs*

This chapter provides information on what the ispEXPERT Compiler functions during compiling ABEL-HDL designs. It covers the following topics:

- Design Entry
- Design Compilation
- Design Simulation

Overview of ABEL-HDL Compiling

Design Entry

In ispDesignExpert, when you create an ABEL-HDL module and import that module into a design, this is called design entry. Design entry for ABEL-HDL modules is primarily a function of the Project Navigator and a Text Editor (used to enter the ABEL-HDL code). The following sections use a sample to describe how to enter the design in a project.

Creating a Design Using ABEL-HDL Sources

Follow the steps to describe the design using ABEL-HDL.

To start a new project and set up a new directory for this tutorial:

1. Start ispDesignExpert. The Project Navigator window appears.
2. Select **File** ⇒ **New Project**. The Create New Project dialog box (Figure 3-1) appears.

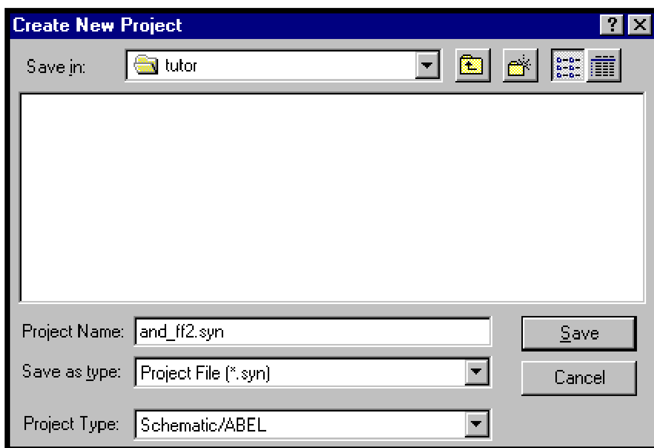

Figure 3-1. Create New Project Dialog Box

3. Select Schematic/ABEL in the Project Type field. This specifies the design source in ispDesignExpert.
4. Navigate to a directory where you want to save your project files, enter a project name `and_ff2.syn` in the Project Name field.
5. Click **Save** to exit the Create New Project dialog box. The Project Navigator displays the new project with the defalut device ispLSI5384E-125LB388.

To change the name of the project:

1. Double-click on the project notebook icon or project name Untitled that appears at the top of the Sources in Project window. The Project Properties dialog box (Figure 3-2) appears.

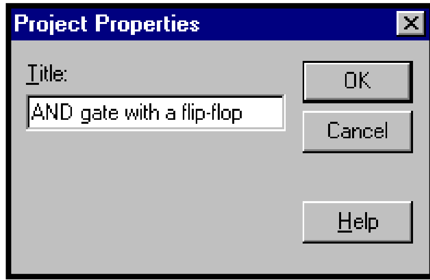

Figure 3-2. Project Properties Dialog Box

2. Enter a descriptive title `AND gate with a flip-flop` in the Title field.
3. Click **OK** to save the change.
4. Select **File** ⇒ **Save** from the Project Navigator to save the changes to your new project.

To enter the ABEL-HDL description:

1. Select **Source** ⇒ **New** to create a new design source. The New Source dialog box (Figure 3-3) appears.

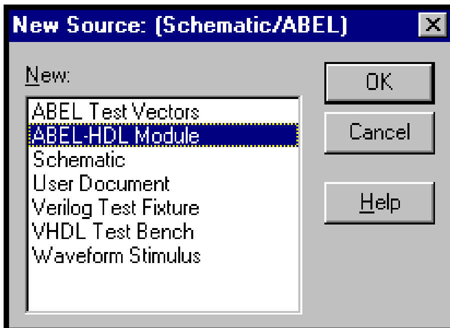

Figure 3-3. New Source Dialog Box

2. Select ABEL-HDL Module in the New field.
3. Click **OK** to close the dialog box. The Text Editor loads and the New ABEL-HDL dialog box (Figure 3-4) appears prompting you for a module name, file name, and title.

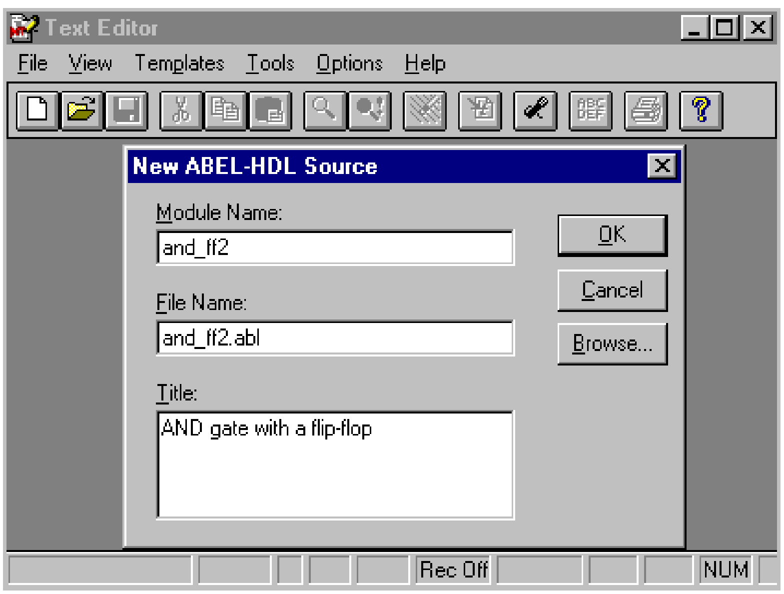

Figure 3-4. New ABEL-HDL Source Dialog Box

4. In the Module Name field, enter `and_ff2`.
5. In the File Name field, enter `and_ff2.abl` (the file extension can be omitted).

NOTE The module name and file name should have the same base name as demonstrated above. (The base name is the name without the 3 character extension.) If the module and file names are different, some automatic functions in the Project Navigator might fail to run properly.

6. If you like, enter a descriptive title `AND gate with a flip-flop` in the Title text box.
7. When you have finished entering the information, click the **OK** button. You now have a template ABEL-HDL source file as shown in Figure 3-5.

Figure 3-5. Template ABEL-HDL Source File

For detailed syntax on ABEL-HDL language, refer to the ***ABEL-HDL Reference Manual***.

To enter the logic description:

8. Add declarations for the three inputs (two AND gate inputs and the clock) and the output by entering the following statements in the ABEL-HDL source file. If a TITLE statement exists in the template file, enter these statements after the TITLE statement:

```
input_1, input_2, Clk        pin;
output_q                     pin istype 'reg';
```

These two statements declare four signals (input_1, input_2, Clk, and output_q).

NOTE ABEL-HDL does not have an explicit declaration for inputs and outputs; whether a given signal is an input or an output depends on how it is used in the design description that follows. The signal `output_q` is declared to be type 'reg', which implies that it is a registered output pin. The actual behavior of `output_q`, however, is specified using one or more equations.

9. To describe the actual behavior of this design, enter two equations in the following manner:

```
Equations
   output_q         := input_1 & input_2;
   output_q.clk      = Clk;
```

These two equations define the data to be loaded on the registered output, and define the clocking function for the output.

Specifying Test Vectors

The method for testing ABEL-HDL designs is to use test vectors. Test vectors are sets of input stimulus values and corresponding expected outputs that can be used with the functional and timing simulator. Test vectors can be specified in two ways. They can be specified in the ABEL-HDL source, or they can be specified in an external Test Vector file (`.abv`). When you specify the test vectors in the ABEL-HDL source, the system will create a dummy ABV file (*`design`*`-vectors`) that points to the ABEL-HDL source containing the vectors.

As the test vectors in this sample is very short, we just add them to the ABEL-HDL source file.

To add the test vectors to the ABEL-HDL source file:

10. Type the following test vectors before the END statement in the `and_ff2.abl` file.

```
Test_vectors
        ([Clk, input_1 , input_2] -> output_q)
         [ 0 ,    0    ,    0    ] ->    0;
         [.C.,    0    ,    0    ] ->    0;
         [.C.,    0    ,    1    ] ->    0;
         [.C.,    1    ,     1    ] ->    1;
```

Figure 3-6 shows the complete ABEL-HDL source file.

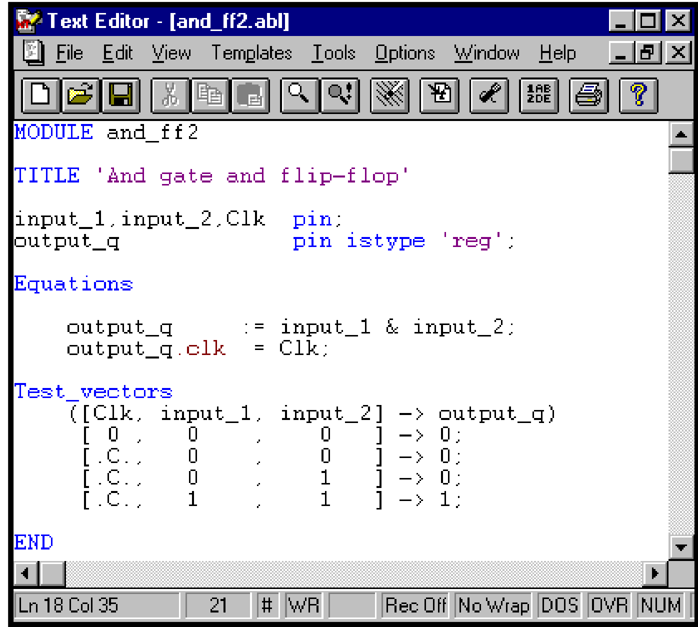

Figure 3-6. Sample ABEL-HDL Source File `and_ff2.abl`

11. Select **File** ⇒ **Save** from the Text Editor to save the ABEL-HDL source file.
12. Select **File** ⇒ **Exit** to exit the Text Editor.

After creating the ABEL-HDL source file, the Project Navigator updates the Sources window to include the new ABEL-HDL source (notice the ABEL-HDL source icon). The Project Navigator also updates the Processes window to reflect the steps necessary to process this source file.

Design Compliation

In general, compiling involves every process after Design Entry that prepares your design for simulation and implementation. These processes include compiling and optimizing steps which can be done at the level of a single module or for the entire design.

However, which processes are available for your design depends entirely on which device architecture you want to implement your design.

This chapter discusses some of the general considerations and processes used in ABEL-HDL compiling. For more information about design considerations, refer to **Chapter 4, "ABEL-HDL Design Considerations."**

Keeping Track of Process: Auto-update

Figure 3-7 shows the Project Naviagor window for the `and_ff2` ABEL-HDL module.

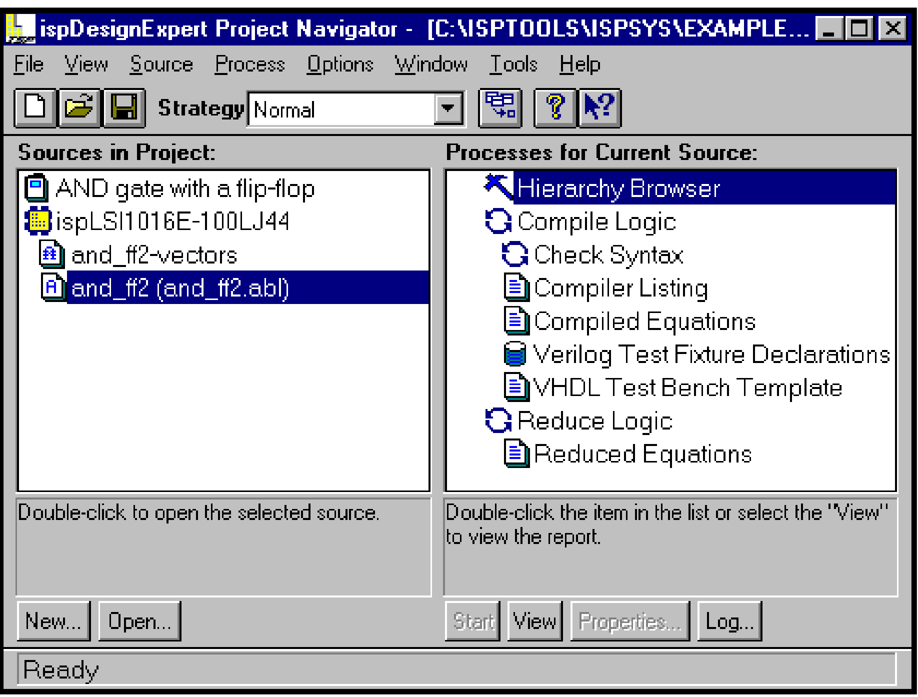

Figure 3-7. Project Naviagtor Window with `and_ff2.syn` Loaded

There are more processes required for an ABEL-HDL source file than for a schematic, because the ABEL-HDL source file requires compilation and optimization before you can run a simulation. And the Project Navigator knows what processes are required to generate a simulation file from an ABEL-HDL source, you can double-click on the end process you want. The auto-update feature automatically runs any processes required to complete the process you request.

Device-related processes, such as mapping the selected ABEL-HDL source file to a JEDEC file, will be available in the Processes for Current Source window after you select a device for the design.

Compiling an ABEL-HDL Source File

The Project Navigator's auto-updating reprocesses sources when they are needed to perform the process you request. You do not need to worry about when to recompile ABEL-HDL source files.

However, you can compile an individual source file by highlighting the file in the Sources window and double-clicking on Compile Logic in the Processes window. Alternatively, you can double-click on a report in the Processes window and compile automatically.

To compile an ABEL-HDL file and view the report:

1. Highlight a ABEL-HDL source file (`and_ff2.abl`) in the Sources window.
2. Double-click Compiled Equations in the Processes window.

   The source file is compiled and the resulting compiled equations are displayed in the Report Viewer (Figure 3-8). If the ABEL-HDL file contains syntax errors, the errors are displayed in a view window and an error indication appears in the Processes window.

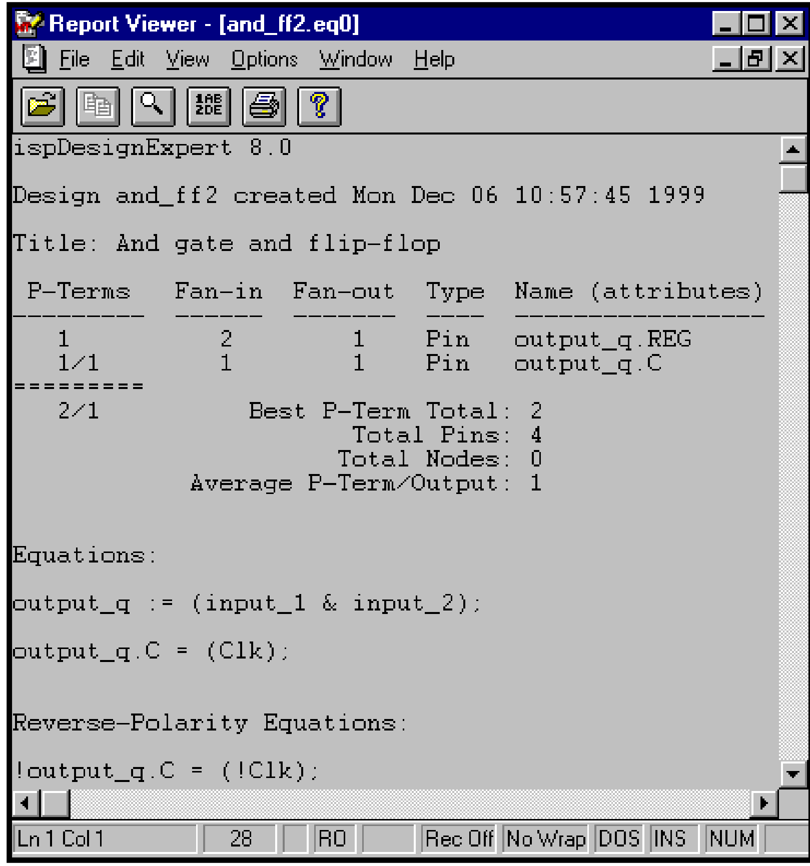

Figure 3-8. Compiled Equations for `and_ff2`

In this example, the compiled equations are identical to the equations that you entered in the ABEL-HDL source file. This is because the equations were simple Boolean equations that did not require any advanced compiling in order to be processed.

Using Properties and Strategies

For many processes (such as the compiling and optimizing steps shown above), there are processing options you can specify. These options include compiler options (such as custom arguments or processing changes) and optimization options (such as node collapsing). You can use properties to specify these options.

Properties

The properties available at any given time depend on the following conditions:

- The selected type of source file in the Sources window (for example, ABEL-HDL).
- The selected process in the Processes window

To see how properties are set:

1. Highlight the ABEL-HDL source file in the Sources window (by clicking on the `and_ff2` ABEL-HDL source).
2. Highlight (do not double-click) Compile Logic in the Processes window.
3. Click the **Properties** button below the Processes window.

   The Properties dialog box (Figure 3-9) appears with a menu of properties. This properties menu is specific to the Compile Logic process for an ABEL-HDL source.

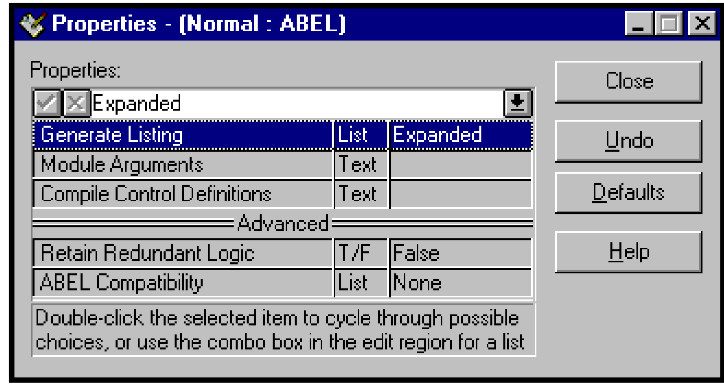

Figure 3-9. Properties Dialog Box

4. In the Properties dialog box, select the Generate Listing property.
5. Click on the arrow to the right of the text box (at the top of the properties menu), and select the Expanded option from the list.
6. Click on the **Close** button to accept the setting and exit the Properties dialog box.

To get information on a property:

1. Click on a property in the Properties Dialog box.
2. Press the **Help** button.

Strategies

Another way to set options in your project is to use strategies. A strategy is a set of properties (processing options) that you have specified for some or all of the sources in your project. Strategies can be useful as your processing requirements change, depending on factors such as size and speed tradeoffs in synthesis, or whether your design is being processed for simulation or final implementation.

With strategies, you do not have to modify the properties for every source in the design if you want to change the processing options. Strategies allow you to set up properties once, then associate a strategy with a source to which you want to apply the properties. You can create new strategies that reflect different properties for the entire project, and then associate one or more custom strategies with the sources in your project.

To see how strategies work:

1. Select **Source** ⇒ **Strategy** from the Project Navigator. The Define Strategies dialog box (Figure 3-10) appears.

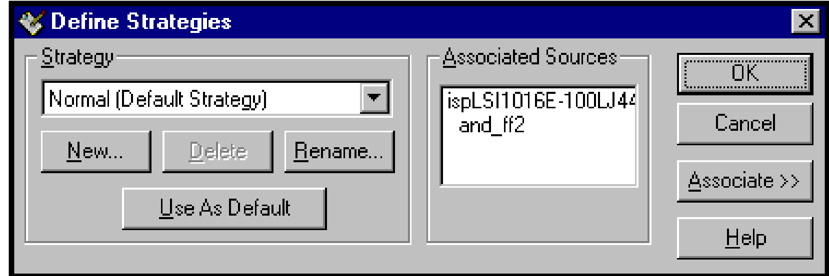

Figure 3-10. Define Strategies Dialog Box

2. Click the **New** button, the New Strategy dialog box (Figure 3-11) appears.

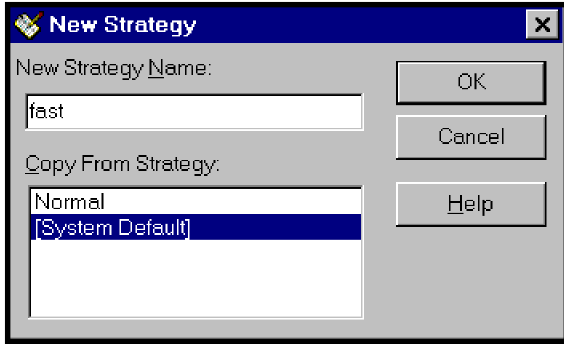

Figure 3-11. New Strategy Dialog Box

3. Enter a name for the strategy in the New strategy Name field.
4. Click the **OK** button. The new strategy appears in the Strategy drop-down list box in the Define Strategies dialog box.

To associate a source with a new strategy:

1. Select a strategy in the Strategy field of the Define Stratigies dialog box.
2. Click the **Associate** button.
3. Highlight the ABEL-HDL source `and_ff2` in the Source to Associate with Strategy field.
4. Click the **Associate with Strategy** button.

   The `and_ff2` source appears in the Associated Sources list box (Figure 3-12).

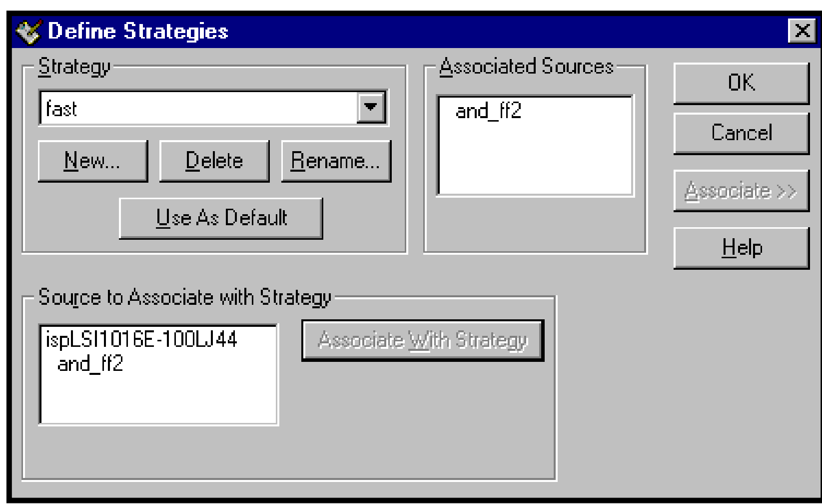

Figure 3-12. Expanded Define Strategies Dialog Box

There is a shortcut method to associate a source with a strategy from the Project Navigator. Highlight a source and use the **Strategy** drop-down list box in the toolbar to associate an existing strategy with the selected source.

Design Simulation

The following section briefly discusses simulation and waveform viewing. For further information on simulation, refer to the ***Design Verification Tools User Manual***.

To simulate the design:

1. Highlight the test vector file (`.abv`) in the Sources window.

   In this tutorial, as the test vectors are specified in the ABEL-HDL module, the `and_ff2-vectors` in the Sources window is actually a dummy test vector file that links to the test vectors in the `and_ff2.abl` file.

2. Double-click on the Functional Simulation process in the Processes window.

   The Project Navigator builds all of the files needed to simulate the design and then runs the functional simulator.

   The Simulator Control Panel (Figure 3-13) appears after a successful compiling of the test vectors.

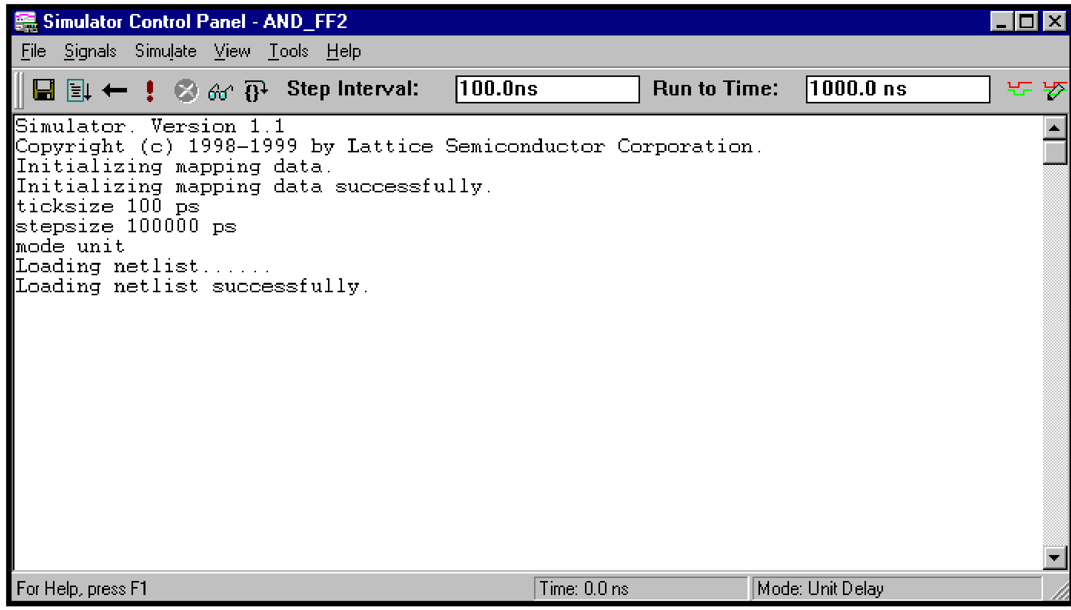

Figure 3-13. Simulator Control Panel Window

3. From the Simulator Control Panel, click the **Run** icon or select **Simulate** ⇒ **Run**.

   The simulator runs from the initial time until the time value defined in the Run to Time field.

4. Select **Tools** ⇒ **Waveform Viewer** after the simulator stops. The Waveform Viewer opens with the signals and their waveforms (Figure 3-14).

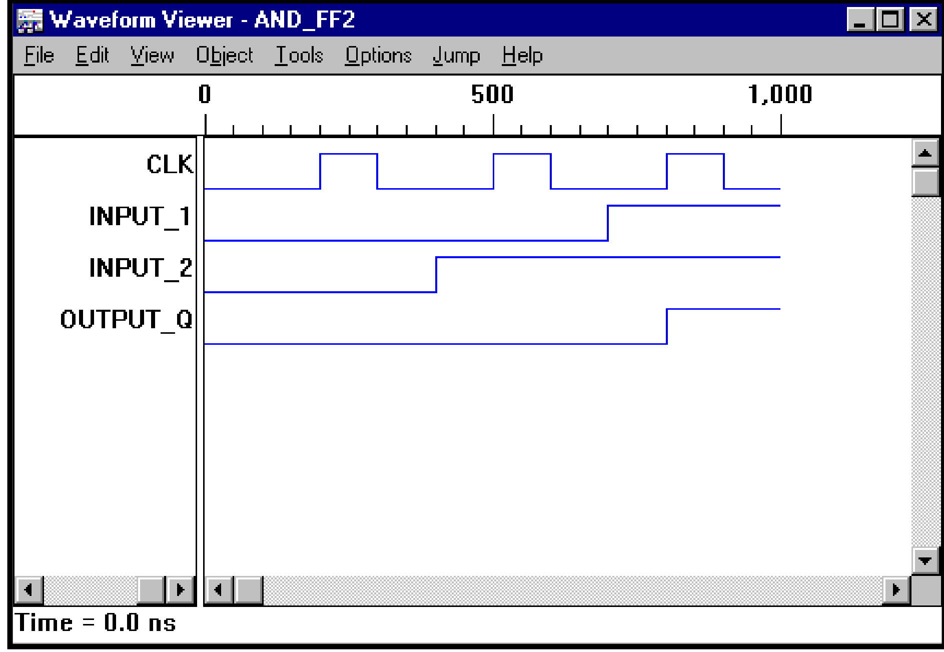

Figure 3-14. Waveform Viewer Window

TIP	If you keep the **View ⇒ Show Waveforms** menu item checked in the Simulator Control Panel, the Waveform Viewer will be invoked during the simulation process for you to monitor the simulation results.

The Waveform Viewer works like a logic analyzer. You can display any signal in the design, group signals into buses and display them in a choice of radices. You can also jump between times, place cursors and measure delta times, and do other typical logic analyzer tasks.

For more information on selecting waveforms to view in the Waveform Viewer, refer to the ***Design Verification Tools User Manual***.

Chapter 4 *ABEL-HDL Design Considerations*

This chapter covers the following topics:

- Overview of ABEL-HDL Design Considerations
- Hierarchy in ABEL-HDL
- Hierarchical Design Considerations
- Node Collapsing
- Pin-to-pin Language Features
- Pin-to-pin vs. Detailed Descriptions for Registered Designs
- Using Active-low Declarations
- Polarity Control
- Flip-flop Equations
- Feedback Considerations — Dot Extensions
- Using Don't Care Optimization
- Exclusive OR Equation
- State Machines
- Using Complement Arrays
- ABEL-HDL and Truth Tables

Overview of ABEL-HDL Design Considerations

This chapter discusses issues you need to consider when you create a design with ABEL-HDL. The topics covered are listed below:

- Hierarchy in ABEL-HDL
- Hierarchical Design Considerations
- Node Collapsing
- Pin-to-Pin Architecture-independent Language Features
- Pin-to-Pin Vs. Detailed Descriptions for Registered Designs
- Using Active-low Declarations
- Polarity Control
- Istypes and Attributes
- Flip-flop Equations
- Feedback Considerations — Using Dot Extensions
- @DCSET Considerations and Precautions
- Exclusive OR Equations
- State Machines
- Using Complement Arrays
- ABEL-HDL and Truth Tables

Hierarchy in ABEL-HDL

You use hierarchy declarations in an upper-level ABEL-HDL source to refer to (instantiate) an ABEL-HDL module.

To instantiate an ABEL-HDL module:

In the lower-level module: (optional)

1. Identify lower-level I/O Ports (signals) with an INTERFACE statement.

In the top-level source:

2. Declare the lower-level module with an INTERFACE declaration.
3. Instantiate the lower-level module with FUNCTIONAL_BLOCK declarations.

NOTE Hierarchy declarations are not required when instantiating an ABEL-HDL module in a schematic. For instructions on instantiating lower-level modules in schematics, refer to your schematic reference.

Instantiating a Lower-level Module in an ABEL-HDL Source

Identifying I/O Ports in the Lower-level Module

The way to identify an ABEL-HDL module's input and output ports is to place an INTERFACE statement immediately following the MODULE statement. The INTERFACE statement defines the ports in the lower-level module that are used by the top-level source.

You must declare all input pins in the ABEL-HDL module as ports, and you can specify default values of 0, 1, or Don't-care.

You do not have to declare all output pins as ports. Any undeclared outputs become No Connects or redundant nodes. Redundant nodes can later be removed from the designs during post-link optimization.

The following source fragment is an example of a lower-level INTERFACE statement.

```
module lower
interface (a=0, [d3..d0]=7 -> [z0..z7]) ;
title 'example of lower-level interface statement ' ...
```

This statement identifies input a, d3, d2, d1 and d0 with default values, and outputs z0 through z7. For more information, see "Interface (lower-level)" in the ***ABEL-HDL Reference Manual***.

Specifying Signal Attributes

Attributes specified for pins in a lower-level module are propagated to the higher-level source. For example, a lower-level pin with an 'invert' attribute affects the higher-level signal wired to that pin (it affects the pin's preset, reset, preload, and power-up value).

Output Enables (OE)

Connecting a lower-level tristate output to a higher-level pin results in the output enable being specified for the higher-level pin. If another OE is specified for the higher-level pin, it is flagged as an error. Since most tristate outputs are used as bidirectionals, it might be important to keep the lower-level OE.

Buried Nodes

Buried nodes in lower-level sources are handled as follows:

Dangling Nodes	Lower-level nodes that do not fanout are propagated to the higher-level module and become dangling nodes. Optimization may remove dangling nodes.
Combinational nodes	Combinational nodes in a lower-level module become collapsible nodes in the higher-level module.
Registered nodes	Registered nodes are preserved with hierarchical names assigned to them.

Declaring Lower-level Modules in the Top-level Source

To declare a lower-level module, you match the lower-level module's INTERFACE statement with an INTERFACE declaration. For example, to declare the lower-level module given above, you would add the following declaration to your upper-level source declarations:

```
lower interface (a, [d3..d0] -> [z0..z7]) ;
```

You could specify different default values if you want to override the values given in the instantiated module, otherwise the instantiated module must exactly match the lower-level interface statement. See "Interface (top-level)" in the ***ABEL-HDL Reference Manual*** for more information.

Instantiating Lower-level Modules in Top-level Source

Use a FUNCTIONAL_BLOCK declaration in an top-level ABEL-HDL source to instantiate a declared lower-level module and make the ports of the lower-level module accessible in the upper-level source. You must declare sources with an INTERFACE declaration before you instantiate them.

To instantiate the module declared above, add an interface declaration and signal declarations to your top-level declarations, and add port connection equations to your top-level equations, as shown in the source fragment below:

```
DECLARATIONS
   low1 FUNCTIONAL_BLOCK lower ;
   zed0..zed7 pin ;               "upper-level inputs
   atop pin istype 'reg,buffer';  "upper-level output
   d3..d0 pin istype 'reg,buffer'; "upper-level ouputs
EQUATIONS
   atop = low1.a;                 "wire this source's outputs
   [d3..d0] = low1.[d3..d0] ;     "to lower-level inputs
   low1.[z0..z7] = [zed0..zed7];  "wire this source's inputs
                                 "to lower-level outputs
```

See "Functional_block" in the ***ABEL-HDL Reference Manual*** for more information.

Hierarchy and Retargeting and Fitting

Redundant Nodes

When you link multiple sources, some unreferenced nodes may be generated. These nodes usually originate from lower-level outputs that are not being used in the top-level source. For example, when you use a 4-bit counter as a 3-bit counter. The most significant bit of the counter is unused and can be removed from the design to save device resources. This step also removes trivial connections. In the following example, if `out1` is a pin and `t1` is a node:

```
out1 = t1;
t1 =  a86;
```

would be mapped to

```
out1 = a86;
```

Merging Feedbacks

Linking multiple modules can produce signals with one or more feedback types, such as .FB and .Q. You can tell the optimizer to combine these feedbacks to help the fitting process.

Post-linked Optimization

If your design has a constant tied to an input, you can re-optimize the design. Re-optimizing may further reduce the product terms count.

For example, if you have the equation

```
out = i0 & i1 || !i0 & i2;
```

and i0 is tied to 1, the resulting equation would be simplified to

```
out = i1;
```

Hierarchical Design Considerations

The following considerations apply to hierarchical design.

Prevent Node Collapsing

Use the signal attribute ‘keep’ to indicate that the combinational node should not be collapsed (removed). For example, the following ABEL-HDL source uses the 'keep' signal attribute:

```
MODULE sub1
TITLE 'sub-module 1'
a,b,c pin;
d   pin ;
e   node istype 'keep';
Equations
e = a $ b;
d = c & e;
END
```

Node Collapsing

All combinational nodes are collapsible by default. Nodes that are to be collapsed (or nodes that are to be preserved) are flagged through the use of signal attributes in the language. The signal attributes are:

Istype 'keep'	Do not collapse this node.
'collapse'	Collapse this node.

Collapsing provides multi-level optimization for combinational logic. Designs with arithmetic and comparator circuits generally generate a large number of product terms that will not fit to any programmable logic device. Node collapsing allows you to describe equations in terms of multi-level combinational nodes, then collapse the nodes into the output until it reaches the product term you specify. The result is an equation that is optimized to fit the device constraints.

Selective Collapsing

In some instances you may want to prevent the collapsing of certain nodes. For example, some nodes may help in the simulation process. You can specify nodes you do not want collapsed as Istype 'keep' and the optimizer will not collapse them.

Pin-to-pin Language Features

ABEL-HDL is a device-independent language. You do not have to declare a device or assign pin numbers to your signals until you are ready to implement the design into a device. However, when you do not specify a device or pin numbers, you need to specify pin-to-pin attributes for declared signals.

Because the language is device-independent, the ABEL-HDL compiler does not have predetermined device attributes to imply signal attributes. If you do not specify signal attributes or other information (such as the dot extensions, which are described later), your design might not operate consistently if you later transfer it to a different target device.

Device-independence *vs.* Architecture-independence

The requirement for signal attributes does not mean that a complex design must always be specified with a particular device in mind. You may still have to understand the differences between GAL devices and ispLSI devices, but you do not have to specify a particular device when describing your design.

Attributes and dot extensions help you refine your design to work consistently when moving from one class of device architecture to another; for example from devices having inverted outputs to those with a particular kind of reset/preset circuitry. However, the more you refine your design, using these language features, the more restrictive your design becomes in terms of the number of device architectures for which it is appropriate.

Signal Attributes

Signal attributes remove ambiguities that occur when no specific device architecture is declared. If your design does not use device-related attributes (either implied by a DEVICE statement or expressed in an ISTYPE statement), it may not operate the same way when targeted to different device architectures. See “Pin Declaration,” “Node Declaration” and “Istype” in the ***ABEL-HDL Reference Manual*** for more information.

Signal Dot Extensions

Signal dot extensions, like attributes, enable you to more precisely describe the behavior of a circuit that may be targeted to different architectures. Dot extensions remove the ambiguities in equations.

Refer to **“Feedback Considerations — Dot Extensions” on page 55** and “Language Structure” in the ***ABEL-HDL Reference Manual*** for more information.

Pin-to-pin *vs.* Detailed Descriptions for Registered Designs

You can use ABEL-HDL assignment operators when you write high-level equations. The **=** operator specifies a combinational assignment, where the design is written with only the circuit's inputs and outputs in mind. The **:=** assignment operator specifies a registered assignment, where you must consider the internal circuit elements (such as output inverters, presets and resets) related to the memory elements (typically flip-flops). The semantics of these two assignment operators are discussed below.

Using := for Pin-to-pin Descriptions

The **:=** implies that a memory element is associated with the output defined by the equation. For example, the equation;

```
Q1 := !Q1 # Preset;
```

implies that Q1 will hold its current value until the memory element associated with that signal is clocked (or unlatched, depending on the register type). This equation is a pin-to-pin description of the output signal Q1. The equation describes the signal's behavior in terms of desired output pin values for various input conditions. Pin-to-pin descriptions are useful when describing a circuit that is completely architecture-independent.

Language elements that are useful for pin-to-pin descriptions are the ":=" assignment operator, and the .CLK, .OE, .FB, .CLR, .ACLR, .SET, .ASET and .COM dot extensions described in the ***ABEL-HDL Reference Manual***. These dot extensions help resolve circuit ambiguities when describing architecture-independent circuits.

Resolving Ambiguities

In the equation above (Q1 := !Q1 # Preset;), there is an ambiguous feedback condition. The signal Q1 appears on the right side of the equation, but there is no indication of whether that fed-back signal should originate at the register, come directly from the combinational logic that forms the input to the register, or come from the I/O pin associated with Q1. There is also no indication of what type of register should be used (although register synthesis algorithms could, theoretically, map this equation into virtually any register type). The equation could be more completely specified in the following manner:

```
Q1.CLK   =  Clock;       "Register clocked from input
Q1 := !Q1.FB # Preset; "Reg. feedback normalized to pin value
```

This set of equations describes the circuit completely and specifies enough information that the circuit will operate identically in virtually any device in which you can fit it. The feedback path is specified to be from the register itself, and the .CLK equation specifies that the memory element is clocked, rather than latched.

Detailed Circuit Descriptions

In contrast to a pin-to-pin description, the same circuit can be specified in a detailed form of design description in the following manner:

```
Q1.CLK =  Clock;            "Register clocked from input
Q1.D   =  !Q1.Q # Preset;   "D-type f/f used for register
```

In this form of the design, specifying the D input to a D-type flip-flop and specifying feedback directly from the register restricts the device architectures in which the design can be implemented. Furthermore, the equations describe only the inputs to, and feedback from, the flip-flop and do not provide any information regarding the configuration of the actual output pin. This means the design will operate quite differently when implemented in a device with inverted outputs versus a device with non-inverting outputs.

To maintain the correct pin behavior, using detailed equations, one additional language element is required: a ‘buffer’ attribute (or its complement, an ‘invert’ attribute). The ‘buffer’ attribute ensures that the final implementation in a device has no inversion between the specified D-type flip-flop and the output pin associated with Q1. For example, add the following to the declarations section:

```
Q1 pin istype ‘buffer’;
```

Detailed Descriptions: Designing for Macrocells

One way to understand the difference between pin-to-pin and detailed description methods is to think of detailed descriptions as macrocell specifications. A macrocell is a block of circuitry normally (but not always) associated with a device’s I/O pin. Figure 4-1 illustrates a typical macrocell associated with signal Q1.

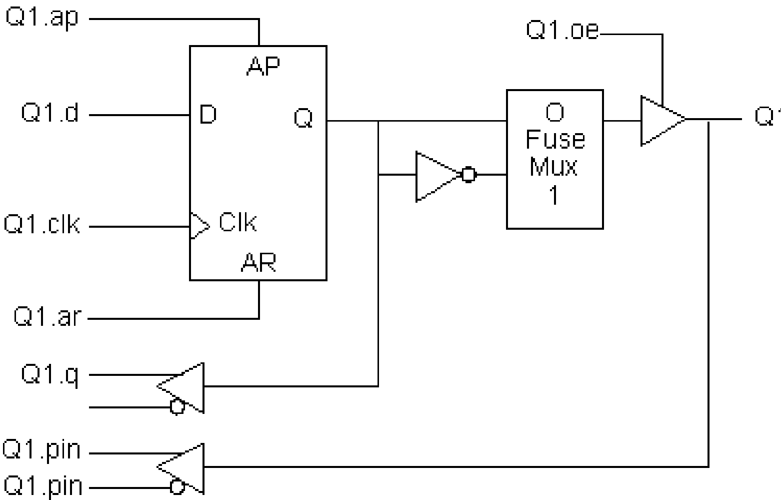

Figure 4-1. Detailed Macrocell

Detailed descriptions are written for the various input ports of the macrocell (shown in the figure above with dot extension labels). Note that the macrocell features a configurable inversion between the Q output of the flip-flop and the output pin labeled Q1. If you use this inverter (or select a device that features a fixed inversion), the behavior you observe on the Q1 output pin will be inverted from the logic applied to (or observed on) the various macrocell ports, including the feedback port Q1.q.

Pin-to-pin descriptions, on the other hand, allow you to describe your circuit in terms of the expected behavior on an actual output pin, regardless of the architecture of the underlying macrocell. Figure 4-2 illustrates the pin-to-pin concept:

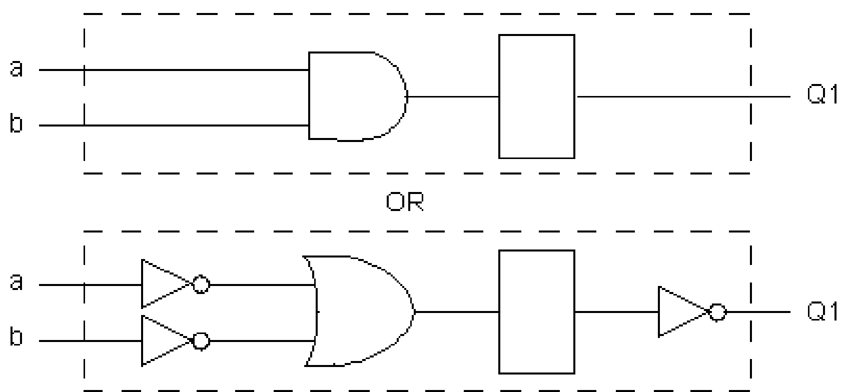

Figure 4-2. Pin-to-pin Macrocell

When pin-to-pin descriptions are written in ABEL-HDL, the “generic macrocell” shown above is synthesized from whatever type of macrocell actually exists in the target device.

Examples of Pin-to-pin and Detailed Descriptions

Two equivalent module descriptions, one pin-to-pin and one detailed, are shown below for comparison:

Pin-to-pin Module Description

```
module Q1_1
        Q1              pin     istype 'reg';
        Clock,Preset     pin;

equations
        Q1.clk  =  Clock;
        Q1     := !Q1.fb # Preset;

test_vectors ([Clock,Preset] -> Q1)
              [ .c. ,   1  ] -> 1;
              [ .c. ,   0  ] -> 0;
              [ .c. ,   0  ] -> 1;
              [ .c. ,   0  ] -> 0;
              [ .c. ,   1  ] -> 1;
              [ .c. ,   1  ] -> 1;
end
```

Detailed Module Description

```
module Q1_2
        Q1              pin     istype 'reg_D,buffer';
        Clock,Preset     pin;

equations
        Q1.CLK  =  Clock;
        Q1.D    = !Q1.Q # Preset;

test_vectors ([Clock,Preset] -> Q1)
              [ .c. ,   1  ] -> 1;
              [ .c. ,   0  ] -> 0;
              [ .c. ,   0  ] -> 1;
              [ .c. ,   0  ] -> 0;
              [ .c. ,   1  ] -> 1;
              [ .c. ,   1  ] -> 1;
end
```

The first description can be targeted into virtually any device (if register synthesis and device fitting features are available), while the second description can be targeted only to devices featuring D-type flip-flops and non-inverting outputs.

To implement the second (detailed) module in a device with inverting outputs, the source file would need to be modified as shown in the following section.

Detailed Module with Inverted Outputs

```
module Q1_3
        Q1                pin     istype 'reg_D,invert';
        Clock,Preset      pin;

equations
        Q1.CLK   =  Clock;
       !Q1.D     =  Q1.Q # Preset;

test_vectors ([Clock,Preset] -> Q1)
              [ .c. ,   1  ] -> 1;
              [ .c. ,   0  ] -> 0;
              [ .c. ,   0  ] -> 1;
              [ .c. ,   0  ] -> 0;
              [ .c. ,   1  ] -> 1;
              [ .c. ,   1  ] -> 1;
end
```

In this version of the module, the existence of an inverter between the output of the D-type flip-flop and the output pin (specified with the 'invert' attribute) has necessitated a change in the equation for Q1.D.

As this example shows, device-independence and pin-to-pin description methods are preferable, since you can describe a circuit completely for any implementation. Using pin-to-pin descriptions and generalized dot extensions (such as `.FB`, `.CLK` and `.OE`) as much as possible allows you to implement your ABEL-HDL module into any one of a particular class of devices. (For example, any device that features enough flip-flops and appropriately configured I/O resources.) However, the need for particular types of device features (such as register preset or reset) might limit your ability to describe your design in a completely architecture-independent way.

If, for example, a built-in register preset feature is used in a simple design, the target architectures are limited. Consider this version of the design:

```
module Q1_5l
        Q1              pin     istype 'reg,buffer';
        Clock,Preset     pin;

equations
        Q1.CLK  =  Clock;
        Q1.AP   =  Preset;
        Q1     :=  !Q1.fb ;

test_vectors ([Clock,Preset] -> Q1)
              [ .c. ,   1  ] -> 1;
              [ .c. ,   0  ] -> 0;
              [ .c. ,   0  ] -> 1;
              [ .c. ,   0  ] -> 0;
              [ .c. ,   1  ] -> 1;
              [ .c. ,   1  ] -> 1;
end
```

The equation for Q1 still uses the **:=** assignment operator and `.FB` for a pin-to-pin description of Q1's behavior, but the use of `.AP` to describe the reset function requires consideration of different device architectures. The `.AP` extension, like the `.D` and `.Q` extensions, is associated with a flip-flop input, not with a device output pin. If the target device has inverted outputs, the design will not reset properly, so this ambiguous reset behavior is removed by using the ‘buffer’ attribute, which reduces the range of target devices to those with non-inverted outputs.

Using `.ASET` instead of `.AP` can solve this problem if the fitter being used supports the `.ASET` dot extension.

Versions 5 and 7 of the design above and below are unambiguous, but each is restricted to certain device classes:

```
module Q1_7l
        Q1              pin     istype 'reg,invert';
        Clock,Preset    pin;

equations
        Q1.CLK  =  Clock;
        Q1.AR   =  Preset;
        Q1     :=  !Q1.fb ;

test_vectors ([Clock,Preset] -> Q1)
              [ .c. ,   1  ] -> 1;
              [ .c. ,   0  ] -> 0;
              [ .c. ,   0  ] -> 1;
              [ .c. ,   0  ] -> 0;
              [ .c. ,   1  ] -> 1;
              [ .c. ,   1  ] -> 1;
end
```

When to Use Detailed Descriptions

Although the pin-to-pin description is preferable, there will frequently be situations when you must use a more detailed description. If you are unsure about which method to use for various parts of your design, examine the design's requirements. If your design requires specific features of a device (such as register preset or unusual flip-flop configurations), detailed descriptions are probably necessary. If your design is a simple combinational function, or if it matches the "generic" macrocell in its requirements, you can probably use simple pin-to-pin descriptions.

Using := for Alternative Flip-flop Types

In ABEL-HDL you can specify a variety of flip-flop types using attributes such as istype 'reg_D' and 'reg_JK'. However, these attributes do not enforce the use of a specific type of flip-flop when a device is selected, and they do not affect the meaning of the **:=** assignment operator.

You can think of the **:=** assignment operator as a memory operator. The type of register that most closely matches the **:=** assignment operator's behavior is the D-type flip-flop.

The primary use for attributes such as `istype 'reg_D'`, `'reg_JK'` and `'reg_SR'` is to control the generation of logic. Specifying one of the `'reg_'` attributes (for example, `istype 'reg_D'`) instructs the AHDL compiler to generate equations using the .D extension regardless of whether the design was written using .D, := or some other method (for example, state diagrams).

NOTE You also need to specify istype 'invert' or 'buffer' when you use detailed syntax.

Using **:=** for flip-flop types other than D-type is only possible if register synthesis features are available to convert the generated equations into equations appropriate for the alternative flip-flop type specified. Since the use of register synthesis to convert D-type flip-flop stimulus into JK or SR-type stimulus usually results in inefficient circuitry, the use of **:=** for these flip-flop types is discouraged. Instead, you should use the .J and .K extensions (for JK-type flip-flops) or the .S and .R extensions (for SR-type flip-flops) and use a detailed description method (including 'invert' or 'buffer' attributes) to describe designs for these register types.

There is no provision in the language for directly writing pin-to-pin equations for registers other than D-type. State diagrams, however, may be used to describe pin-to-pin behavior for any register type.

Using Active-low Declarations

In ABEL-HDL you can write pin-to-pin design descriptions using implied active-low signals. Active-low signals are declared with a '!' operator, as shown below:

```
!Q1 pin  istype 'reg';
```

If a signal is declared active-low, it is automatically complemented when you use it in the subsequent design description. This complementing is performed for any use of the signal itself, including as an input, as an output, and in test vectors. Complementing is also performed if you use the `.fb` dot extension on an active-low signal.

The following three designs, for example, operate identically:

Design 1 — Implied Pin-to-Pin Active-low

```
module act_low2
        !q0,!q1  pin istype 'reg';
        clock    pin;
        reset    pin;

equations
        [q1,q0].clk = clock;
        [q1,q0] := ([q1,q0].FB + 1) & !reset;

test_vectors ([clock,reset] -> [ q1, q0])
              [ .c. ,  1  ] -> [ 0 , 0 ];
              [ .c. ,  0  ] -> [ 0 , 1 ];
              [ .c. ,  0  ] -> [ 1 , 0 ];
              [ .c. ,  0  ] -> [ 1 , 1 ];
              [ .c. ,  0  ] -> [ 0 , 0 ];
              [ .c. ,  0  ] -> [ 0 , 1 ];
              [ .c. ,  1  ] -> [ 0 , 0 ];
end
```

Design 2 — Explicit Pin-to-Pin Active-low

```
module act_low1
        q0,q1    pin istype 'reg';
        clock    pin;
        reset    pin;

equations
        [q1,q0].clk = clock;
        ![q1,q0] := (![q1,q0].FB + 1) & !reset;

test_vectors ([clock,reset] -> [!q1,!q0])
              [ .c. ,  1  ] -> [ 0 , 0 ];
              [ .c. ,  0  ] -> [ 0 , 1 ];
              [ .c. ,  0  ] -> [ 1 , 0 ];
              [ .c. ,  0  ] -> [ 1 , 1 ];
              [ .c. ,  0  ] -> [ 0 , 0 ];
              [ .c. ,  0  ] -> [ 0 , 1 ];
              [ .c. ,  1  ] -> [ 0 , 0 ];
end
```

Design 3 — Explicit Detailed Active-low

```
module act_low3
        q0,q1    pin istype 'reg_d,buffer';
        clock    pin;
        reset    pin;

equations
        [q1,q0].clk = clock;
        ![q1,q0].D := (![q1,q0].Q + 1) & !reset;

test_vectors ([clock,reset] -> [!q1,!q0])
              [ .c. ,  1  ] -> [ 0 , 0 ];
              [ .c. ,  0  ] -> [ 0 , 1 ];
              [ .c. ,  0  ] -> [ 1 , 0 ];
              [ .c. ,  0  ] -> [ 1 , 1 ];
              [ .c. ,  0  ] -> [ 0 , 0 ];
              [ .c. ,  0  ] -> [ 0 , 1 ];
              [ .c. ,  1  ] -> [ 0 , 0 ];
end
```

Both of these designs describe an up counter with active-low outputs. The first example inverts the signals explicitly (in the equations and in the test vector header), while the second example uses an active-low declaration to accomplish the same thing.

Polarity Control

Automatic polarity control is a powerful feature in ABEL-HDL where a logic function is converted for both non-inverting and inverting devices.

A single logic function may be expressed with many different equations. For example, all three equations below for F1 are equivalent.

```
(1) F1 = (A & B);
(2) !F1 = !(A & B);
(3) !F1 = !A # !B;
```

In the example above, equation (3) uses two product terms, while equation (1) requires only one. This logic function will use fewer product terms in a non-inverting device than in an inverting device. The logic function performed from input pins to output pins will be the same for both polarities.

Not all logic functions are best optimized to positive polarity. For example, the inverted form of F2, equation (3), uses fewer product terms than equation (2).

```
(1) F2 = (A # B) & (C # D);
(2) F2 = (A & C) # (A & D) # (B & C) # (B & D);
(3) !F2 = (!A & !B) # (!C & !D);
```

Programmable polarity devices are popular because they can provide a mix of non-inverting and inverting outputs to achieve the best fit.

Polarity Control with Istype

In ABEL-HDL, you control the polarity of the design equations and target device (in the case of programmable polarity devices) in two ways:

- Using Istype 'neg', 'pos' and 'dc'
- Using Istype 'invert' and 'buffer'

Using Istype ‘neg’, ‘pos’, and ‘dc’ to Control Equation and Device Polarity

The ‘neg’, ‘pos’, and ‘dc’ attributes specify types of optimization for the polarity as follows:

‘neg’ — **Istype ‘neg’** optimizes the circuit for negative polarity. Unspecified logic in truth tables and state diagrams becomes a 0.

‘pos’ — **Istype ‘pos’** optimizes the circuit for positive polarity. Unspecified logic in truth tables and state diagrams becomes a 1.

‘dc’ — **Istype ‘dc’** uses polarity for best optimization. Unspecified logic in truth tables and state diagrams becomes don't care (X).

Using ‘invert’ and ‘buffer’ to Control Programmable Inversion

An optional method for specifying the desired state of a programmable polarity output is to use the ‘invert’ or ‘buffer’ attributes. These attributes ensure that an inverter gate either does or does not exist between the output of a flip-flop and its corresponding output pin. When you use the ‘invert’ and ‘buffer’ attributes, you can still use automatic polarity selection if the target architecture features programmable inverters located before the associated flip-flop.

NOTE The ‘invert’ and ‘buffer’ attributes do not actually control device or equation polarity — they only enforce the existence or nonexistence of an inverter between a flip-flop and its output pin.

The polarity of devices that feature a fixed inverter in this location, and a programmable inverter before the register, cannot be specified using ‘invert’ and ‘buffer’.

Flip-flop Equations

Pin-to-pin equations (using the **:=** assignment operator) are only supported for D flip-flops. ABEL-HDL does not support the **:=** assignment operator for T, SR or JK flip-flops and has no provision for specifying a particular output pin value for these types.

If you write an equation of the form:

```
Q1 := 1;
```

and the output, Q1, has been declared as a T-type flip-flop, the ABEL-HDL compiler will give a warning and convert the equation to

```
Q1.T = 1;
```

Since the T input to a T-type flip-flop does not directly correspond to the value you observed on the associated output pin, this equation will not result in the pin-to-pin behavior you want.

To produce specific pin-to-pin behavior for alternate flip-flop types, you must consider the behavior of the flip-flop you used and write detailed equations that stimulate the inputs of that flip-flop. A detailed equation to set and hold a T-type flip-flop is shown below:

```
Q1.T = !Q1.Q;
```

Feedback Considerations — Dot Extensions

The source of feedback is normally set by the architecture of the target device. If you don't specify a particular feedback path, the design may operate differently in different device types. Specifying feedback paths (with the .FB, .Q or .PIN dot extensions) eliminates architectural ambiguities. Specifying feedback paths also allows you to use architecture-independent simulation.

The following rules should be kept in mind when you are using feedback:

- **No Dot Extension** — A feedback signal with no dot extension (for example, count := count+1;) results in pin feedback if it exists in the target device. If there is no pin feedback, register feedback is used, with the value of the register contents complemented (normalized) if needed to match the value observed on the pin.
- **.FB Extension** — A signal specified with the .FB extension (for example, count := count.fb+1;) results in register feedback normalized to the pin value if a register feedback path exists. If no register feedback is available, pin feedback is used, and the fuse mapper checks that the output enable does not conflict with the pin feedback path. If there is a conflict, an error is generated if the output enable is not constantly enabled.
- **.COM Extension** — A signal specified with the .COM extension (for example, count := count.com+1;) results in OR-array (pre-register) feedback, normalized to the pin value if an OR-array feedback path exists. If no OR-array feedback is available, pin feedback is used and the fuse mapper checks that the output enable does not conflict with the pin feedback path. If there is a conflict, an error is generated if the output enable is not constantly enabled.
- **.PIN Extension** — If a signal is specified with the .PIN extension (for example, count := count.pin+1;), the pin feedback path will be used. If the specified device does not feature pin feedback, an error will be generated. Output enables frequently affect the operation of fed-back signals that originate at a pin.
- **.Q Extension** — Signals specified with the .Q extension (for example, count.d = count.q+1;) will originate at the Q output of the associated flip-flop. The fed-back value may or may not correspond to the value you observe on the associated output pin; if an inverter is located between the Q output of the flip-flop and the output pin (as is the case in most registered PAL-type devices), the value of the fed-back signal will be the complement of the value you observe on the pin.

- **.D Extension** — Some devices allow feedback of the input to the register. To select this feedback, use the .D extension. Some device kits also support .COM for this feedback; refer to your device kit manual for detailed information.

Dot Extensions and Architecture-Independence

To be architecture-independent, you must write your design in terms of its pin-to-pin behavior rather than in terms of specific device features (such as flip-flop configurations or output inversions).

For example, consider the simple circuit shown in the following (Figure 4-3). This circuit toggles high when the Toggle input is forced high, and low when the Toggle is low. The circuit also contains a three-state output enable that is controlled by the active-low Enable input.

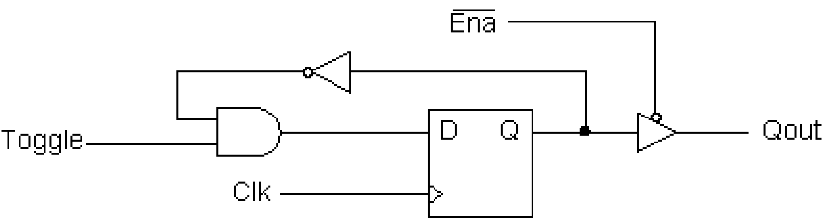

Figure 4-3. Dot Extensions and Architecture-independence: Circuit 1

The following simple ABEL-HDL design describes this simple one-bit synchronous circuit. The design description uses architecture-independent dot extensions to describe the circuit in terms of its behavior, as observed on the output pin of the target device. Since this design is architecture-independent, it will operate the same (disregarding initial powerup state), irrespective of the device type.

```
module pin2pin;
        Clk      pin 1;
        Toggle   pin 2;
        Ena      pin 11;
        Qout     pin 19 istype 'reg';

equations
        Qout      := !Qout.FB & Toggle;
        Qout.CLK   = Clk;
        Qout.OE    = !Ena;

test_vectors([Clk,Ena,Toggle] -> [Qout])
             [.c., 0 ,  0   ] ->   0;
             [.c., 0 ,  1   ] ->   1;
             [.c., 0 ,  1   ] ->   0;
             [.c., 0 ,  1   ] ->   1;
             [.c., 0 ,  1   ] ->   0;
             [.c., 1 ,  1   ] ->  .Z.;
             [ 0 , 0 ,  1   ] ->   1;
             [.c., 1 ,  1   ] ->  .Z.;
             [ 0 , 0 ,  1   ] ->   0;
end
```

Figure 4-4. Pin-to-pin One-bit Synchronous Circuit module pin2pin

If you implement this circuit in a simple GAL16LV8 device (either by adding a device declaration statement or by specifying the P16R8 in the Fuseasm process), the result will be a circuit like the one illustrated in the following figure (Figure 4-5). Since the GAL16LV8 features inverted outputs, the design equation is automatically modified to take the feedback from Q-bar instead of Q.

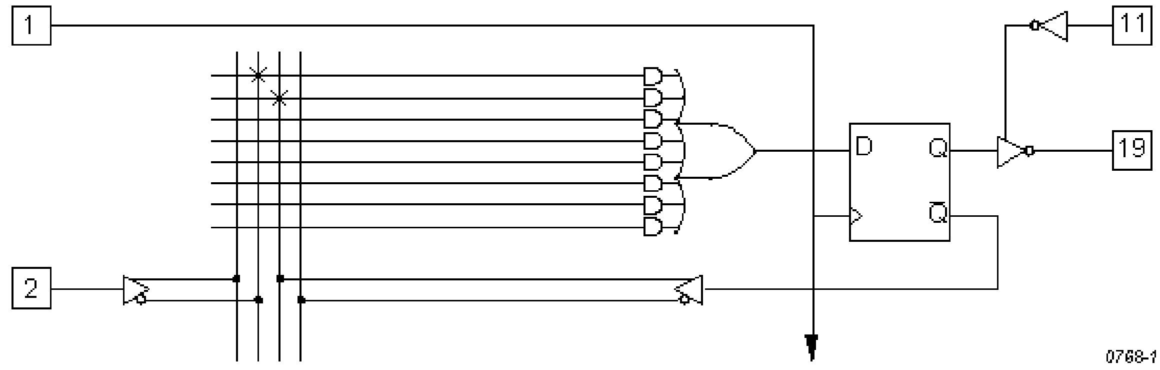

Figure 4-5. Dot Extensions and Architecture-independence: Circuit 2

Dot Extensions and Detail Design Descriptions

You may need to be more specific about how you implement a circuit in a target device. More-complex device architectures have many configurable features, and you may want to use these features in a particular way. You may want a precise powerup and preset operation or, in some cases, you may need to control internal elements.

The circuit previously described (using architecture-independent dot extensions) could be described, for example, using detailed dot extensions in the following ABEL-HDL source file.

```
module detail1
        d1       device  'P16R8';
        Clk      pin 1;
        Toggle   pin 2;
        Ena      pin 11;
        Qout     pin 19 istype 'reg_D';

equations
       !Qout.D    = Qout.Q & Toggle;
        Qout.CLK  = Clk;
        Qout.OE   = !Ena;

test_vectors([Clk,Ena,Toggle] -> [Qout])
             [.c., 0 ,  0    ] ->   0;
             [.c., 0 ,  1    ] ->   1;
             [.c., 0 ,  1    ] ->   0;
             [.c., 0 ,  1    ] ->   1;
             [.c., 0 ,  1    ] ->   0;
             [.c., 1 ,  1    ] ->  .Z.;
             [ 0 , 0 ,  1    ] ->   1;
             [.c., 1 ,  1    ] ->  .Z.;
             [ 0 , 0 ,  1    ] ->   0;
end
```

Figure 4-6. Detailed One-bit Synchronous Circuit with Inverted Qout

This version of the design will result in exactly the same fuse pattern as indicated in Figure 4-5. As written, this design assumes the existence of an inverted output for the signal Qout. This is why the Qout.D and Qout.Q signals are reversed from the architecture-independent version of the design presented earlier.

NOTE The inversion operator applied to Qout.D does not correspond directly to the inversion found on each output of a P16R8. The equation for Qout.D actually refers to the D input of one of the GAL16LV8's flip-flops; the output inversion found in a P16R8 is located after the register and is assumed rather than specified.

To implement this design in a device that does not feature inverted outputs, the design description must be modified. The following example shows how to write this detailed design:

```
module detail2
        Clk      pin 1;
        Toggle   pin 2;
        Ena      pin 11;
        Qout     pin 19 istype 'reg_D';

equations
        Qout.D    = !Qout.Q & Toggle;
        Qout.CLK  = Clk;
        Qout.OE   = !Ena;

test_vectors([Clk,Ena,Toggle] -> [Qout])
             [.c., 0 ,  0   ] ->   0;
             [.c., 0 ,  1   ] ->   1;
             [.c., 0 ,  1   ] ->   0;
             [.c., 0 ,  1   ] ->   1;
             [.c., 0 ,  1   ] ->   0;
             [.c., 1 ,  1   ] ->  .Z.;
             [ 0 , 0 ,  1   ] ->   1;
             [.c., 1 ,  1   ] ->  .Z.;
             [ 0 , 0 ,  1   ] ->   0;
end
```

Figure 4-7. Detail One-bit Synchronous Circuit with non-inverted Qout

Using Don’t Care Optimization

Use Don’t Care optimization to reduce the amount of logic required for an incompletely specified function. The @DCSET directive (used for logic description sections) and ISTYPE attribute ‘dc’ (used for signals) specify don’t care values for unspecified logic.

Consider the following ABEL-HDL truth table:

```
truth_table    ([i3,i2,i1,i0]->[f3,f2,f1,f0])
                [ 0, 0, 0, 0]->[ 0, 0, 0, 1];
                [ 0, 0, 0, 1]->[ 0, 0, 1, 1];
                [ 0, 0, 1, 1]->[ 0, 1, 1, 1];
                [ 0, 1, 1, 1]->[ 1, 1, 1, 1];
                [ 1, 1, 1, 1]->[ 1, 1, 1, 0];
                [ 1, 1, 1, 0]->[ 1, 1, 0, 0];
                [ 1, 1, 0, 0]->[ 1, 0, 0, 0];
                [ 1, 0, 0, 0]->[ 0, 0, 0, 0];
```

This truth table has four inputs, and therefore sixteen (24) possible input combinations. The function specified, however, only indicates eight significant input combinations. For each of the design outputs (f3 through f0) the truth table specifies whether the resulting value should be 1 or 0. For each output, then, each of the eight individual truth table entries can be either a member of a set of true functions called the on-set, or a set of false functions called the off-set.

Using output f3, for example, the eight input conditions can be listed as on-sets and off-sets as follows (maintaining the ordering of inputs as specified in the truth table above):

```
on-set of f3    off-set of f3
 0 1 1 1         0 0 0 0
 1 1 1 1         0 0 0 1
 1 1 1 0         0 0 1 1
 1 1 0 0         1 0 0 0
```

The remaining eight input conditions that do not appear in either the on-set or off-set are said to be members of the dc-set, as follows for f3:

```
dc-set of f3
  0 0 1 0
  0 1 0 0
  0 1 0 1
  0 1 1 0
  1 0 0 1
  1 0 1 0
  1 0 1 1
  1 1 0 1
```

Expressed as a Karnaugh map, the on-set, off-set and dc-set would appear as follows (with ones indicating the on-set, zeroes indicating the off-set, and dashes indicating the dc-set):

If the don't-care entries in the Karnaugh map are used for optimization, the function for f3 can be reduced to a single product term (f3 = i2) instead of the two (f3 = i3 & i2 & !i0 # i2 & i1 & i0) otherwise required.

The ABEL-HDL compiler uses this level of optimization if the @DCSET directive or ISTYPE 'dc' is included in the ABEL-HDL source file, as shown below.

```
module dc
  i3,i2,i1,i0    pin;
  f3,f2,f1,f0    pin istype 'dc,com';

truth_table  ([i3,i2,i1,i0]->[f3,f2,f1,f0])
              [ 0, 0, 0, 0]->[ 0, 0, 0, 1];
              [ 0, 0, 0, 1]->[ 0, 0, 1, 1];
              [ 0, 0, 1, 1]->[ 0, 1, 1, 1];
              [ 0, 1, 1, 1]->[ 1, 1, 1, 1];
              [ 1, 1, 1, 1]->[ 1, 1, 1, 0];
              [ 1, 1, 1, 0]->[ 1, 1, 0, 0];
              [ 1, 1, 0, 0]->[ 1, 0, 0, 0];
              [ 1, 0, 0, 0]->[ 0, 0, 0, 0];
end
```

Figure 4-8. Source File Showing Don't Care Optimization

This example results in a total of four single-literal product terms, one for each output. The same example (with no istype 'dc') results in a total of twelve product terms.

For truth tables, Don't Care optimization is almost always the best method. For state machines, however, you may not want undefined transition conditions to result in unknown states, or you may want to use a default state (determined by the type of flip-flops used for the state register) for state diagram simplification.

When using don't care optimization, be careful not to specify overlapping conditions (specifying both the on-set and dc-set for the same conditions) in your truth tables and state diagrams. Overlapping conditions result in an error message.

For state diagrams, you can perform additional optimization for design outputs if you specify the @dcstate attribute. If you enter @dcstate in the source file, all state diagram transition conditions are collected during state diagram processing. These transitions are then complemented and applied to the design outputs as don't-cares. You must use @dcstate in combination with @dcset or the 'dc' attribute.

Exclusive OR Equations

Designs written for exclusive-OR (XOR) devices should contain the 'xor' attribute for architecture-independence.

Optimizing XOR Devices

You can use XOR gates directly by writing equations that include XOR operators, or you can use implied XOR gates. XOR gates can minimize the total number of product terms required for an output or they can emulate alternate flip-flop types.

Using XOR Operators in Equations

If you want to write design equations that include XOR operators, you must either specify a device that features XOR gates in your ABEL-HDL source file, or specify the 'xor' attribute for all output signals that will be implemented with XOR gates. This preserves one top-level XOR operator for each design output. For example,

```
module X1
         Q1      pin     istype 'com,xor';
        a,b,c    pin;
equations
        Q1 = a $ b & c;
end
```

Also, when writing equations for XOR PALs, you should use parentheses to group those parts of the equation that go on either side of the XOR. This is because the XOR operator ($) and the OR operator (#) have the same priority in ABEL-HDL. See example **octalf.abl**.

Using Implied XORs in Equations

High-level operators in equations often result in the generation of XOR operators. If you specify the 'XOR' attribute, these implied XORs are preserved, decreasing the number of product terms required. For example,

```
module X2
        q3,q2,q1,q0      pin istype 'reg,xor';
        clock            pin;
        count = [q3..q0];
equations
        count.clk = clock;
        count := count.FB + 1;
end
```

This design describes a simple four-bit counter. Since the addition operator results in XOR operators for the four outputs, the 'xor' attribute can reduce the amount of circuitry generated.

NOTE The high-level operator that generates the XOR operators must be the top-level (lowest priority) operation in the equation. An equation such as
count := (count.FB + 1) & !reset; does not result in the preservation of top-level XOR operators, since the & operator is the top-level operator.

Using XORs for Flip-flop Emulation

Another way to use XOR gates is for flip-flop emulation. If you are using an XOR device that has outputs featuring an XOR gate and D-type flip-flops, you can write your design as if you were going to be implementing it in a device with T-type flip-flops. The XOR gates and D-type flip-flops emulate the specified T-type flip-flops. When using XORs in this way, you should not use the 'xor' attribute for output signals unless the target device has XOR gates.

JK Flip-Flop Emulation

You can emulate JK flip-flops using a variety of circuitry found in programmable devices. When a T-type flip-flop is available, you can emulate JK flip-flops by ANDing the Q output of the flip-flop with the K input. The !Q output is then ANDed with the J input.

Figure 4-9 illustrates the circuitry and the Boolean expression.

JK Flip-flop Emulation Using T Flip-flop

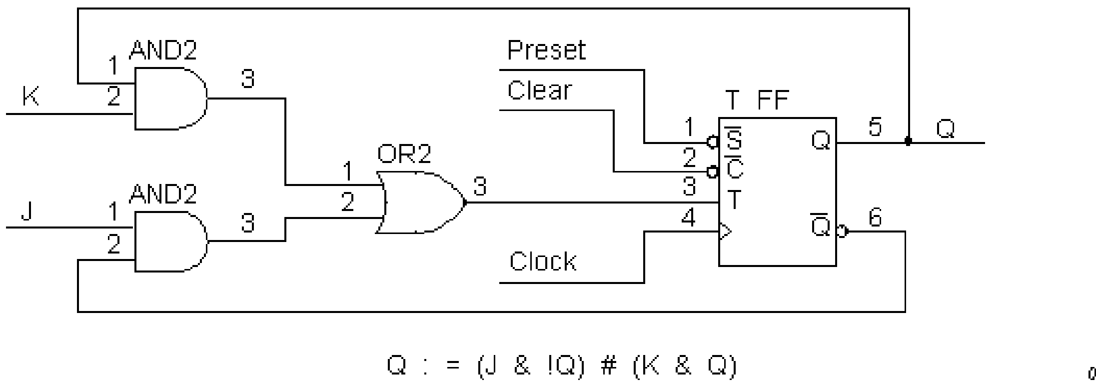

Figure 4-9. JK Flip-flop Emulation Using T Flip-flop

You can emulate a JK flip-flop with a D flip-flop and an XOR gate. This technique is useful in devices such as the GAL20VP8. The circuitry and Boolean expression is shown in Figure 4-10.

T Flip-flop Emulation Using D Flip-flop

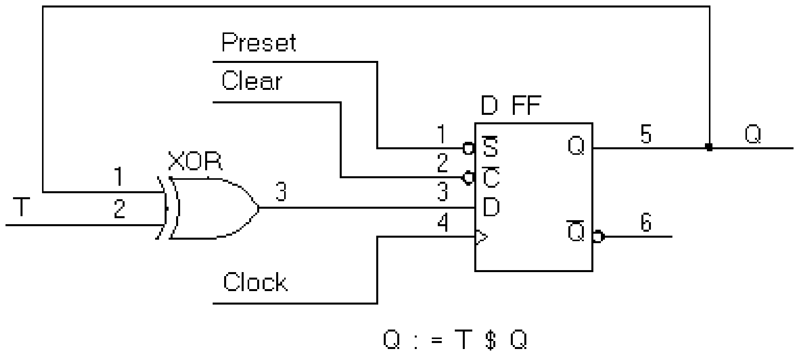

Figure 4-10. T Flip-flop Emulation Using D Flip-flop

Finally, you can also emulate a JK flip-flop by combining the D flip-flop emulation of a T flip-flop, Figure 4-10, with the circuitry of Figure 4-1. The following figure illustrates this concept.

JK Flip-flop Emulation, D Flip-flop with XOR

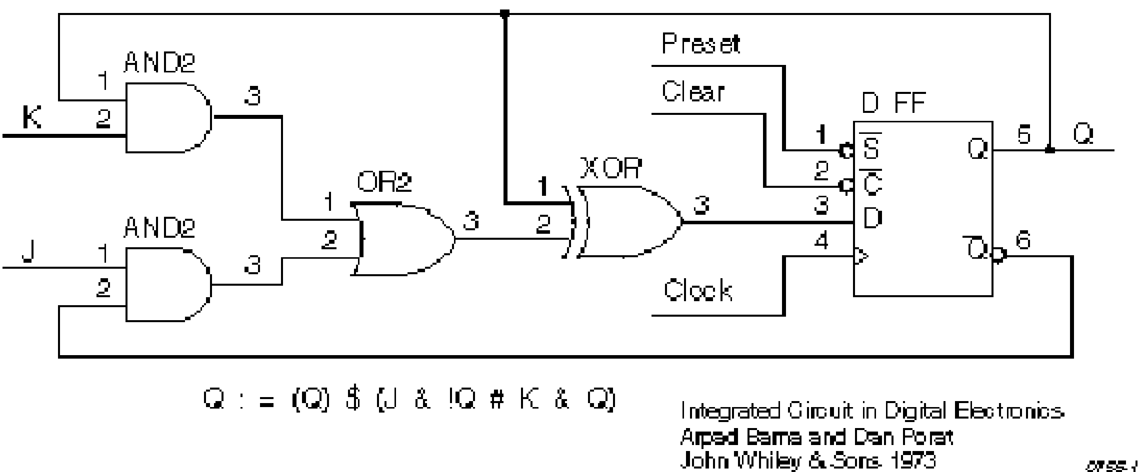

Figure 4-11. JK Flip-flop Emulation, D Flip-flop with XOR

State Machines

A state machine is a digital device that traverses a predetermined sequence of states. State-machines are typically used for sequential control logic. In each state, the circuit stores its past history and uses that history to determine what to do next.

This section provides some guidelines to help you make state diagrams easy to read and maintain and to help you avoid problems. State machines often have many different states and complex state transitions that contribute to the most common problem, which is too many product terms being created for the chosen device. The topics discussed in the following subsections help you avoid this problem by reducing the number of required product terms.

The following subsections provide state machine considerations:

- Use Identifiers Rather Than Numbers for States
- Powerup Register States
- Unsatisfied Transition Conditions, D-Type Flip-Flops
- Unsatisfied Transition Conditions, Other Flip-Flops
- Number Adjacent States for a One-bit Change
- Use State Register Outputs to Identify States
- Use Symbolic State Descriptions

Use Identifiers Rather Than Numbers for States

A state machine has different “states” that describe the outputs and transitions of the machine at any given point. Typically, each state is given a name, and the state machine is described in terms of transitions from one state to another. In a real device, such a state machine is implemented with registers that contain enough bits to assign a unique number to each state. The states are actually bit values in the register, and these bit values are used along with other signals to determine state transitions.

As you develop a state diagram, you need to label the various states and state transitions. If you label the states with identifiers that have been assigned constant values, rather than labeling the states directly with numbers, you can easily change the state transitions or register values associated with each state.

When you write a state diagram, you should first describe the state machine with names for the states, and then assign state register bit values to the state names.

For an example, see Figure 4-12 for a state machine named “sequence.” (This state machine is also discussed in the design examples.) Identifiers (A, B, and C) specify the states. These identifiers are assigned a constant decimal value in the declaration section that identifies the bit values in the state register for each state. A, B, and C are only identifiers: they do not indicate the bit pattern of the state machine. Their declared values define the value of the state register (sreg) for each state. The declared values are 0, 1, and 2.

```
module Sequence
title 'State machine example';


   q1,q0                         pin   14,15 istype 'reg';
   clock,enab,start,hold,reset  pin   1,11,4,2,3;
   halt                          pin   17 istype 'reg';
   in_B,in_C                     pin   12,13 istype 'com';
   sreg                          =     [q1,q0];

"State Values...
   A = 0;          B = 1;          C = 2;

equations
   [q1,q0,halt].clk = clock;
   [q1,q0,halt].oe  = !enab;
state_diagram sreg;
   State A:             " Hold in state A until start is active.
     in_B = 0;
     in_C = 0;
     IF (start & !reset) THEN B WITH halt := 0;
     ELSE A WITH halt := halt.fb;

   State B:             " Advance to state C unless reset is active
     in_B = 1;          " or hold is active. Turn on halt indicator
     in_C = 0;          " if reset.
     IF (reset) THEN A WITH halt := 1;
     ELSE IF (hold) THEN B WITH halt := 0;
     ELSE C WITH halt := 0;

   State C:             " Go back to A unless hold is active
     in_B = 0;          " Reset overrides hold.
     in_C = 1;
     IF (hold & !reset) THEN C WITH halt := 0;
     ELSE A WITH halt := 0;

test_vectors([clock,enab,start,reset,hold]->[sreg,halt,in_B,in_C])
             [ .p. ,  0 ,  0  ,  0  ,  0 ]->[  A ,  0  ,  0 ,  0 ];
             [ .c. ,  0 ,  0  ,  0  ,  0 ]->[  A ,  0  ,  0 ,  0 ];
             [ .c. ,  0 ,  1  ,  0  ,  0 ]->[  B ,  0  ,  1 ,  0 ];
             [ .c. ,  0 ,  0  ,  0  ,  0 ]->[  C ,  0  ,  0 ,  1 ];

             [ .c. ,  0 ,  1  ,  0  ,  0 ]->[  A ,  0  ,  0 ,  0 ];
             [ .c. ,  0 ,  1  ,  0  ,  0 ]->[  B ,  0  ,  1 ,  0 ];
             [ .c. ,  0 ,  0  ,  1  ,  0 ]->[  A ,  1  ,  0 ,  0 ];
             [ .c. ,  0 ,  0  ,  0  ,  0 ]->[  A ,  1  ,  0 ,  0 ];

             [ .c. ,  0 ,  1  ,  0  ,  0 ]->[  B ,  0  ,  1 ,  0 ];
             [ .c. ,  0 ,  0  ,  0  ,  1 ]->[  B ,  0  ,  1 ,  0 ];
             [ .c. ,  0 ,  0  ,  0  ,  1 ]->[  B ,  0  ,  1 ,  0 ];
             [ .c. ,  0 ,  0  ,  0  ,  0 ]->[  C ,  0  ,  0 ,  1 ];
end
```

Figure 4-12. Using Identifiers for States

Powerup Register States

If a state machine has to have a specific starting state, you must define the register powerup state in the state diagram description or make sure your design goes to a known state at powerup. Otherwise, the next state is undefined.

Unsatisfied Transition Conditions

D-Type Flip-Flops

For each state described in a state diagram, you specify the transitions to the next state and the conditions that determine those transitions. For devices with D-type flip-flops, if none of the stated conditions are met, the state register, shown in the following figure, is cleared to all 0s on the next clock pulse. This action causes the state machine to go to the state that corresponds to the cleared state register. This can either cause problems or you can use it to your advantage, depending on your design.

D-type Register with False Inputs

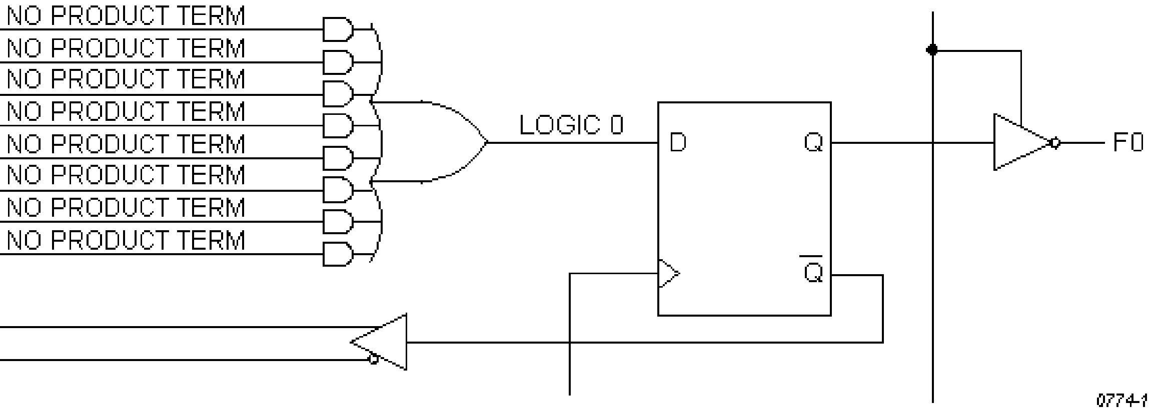

Figure 4-13. D-type Register with False Inputs

You can use the clearing behavior of D-type flip-flops to eliminate some conditions in your state diagram, and some product terms in the converted design, by leaving the cleared-register state transition implicit. If no specified transition condition is met, the machine goes to the cleared-register state. This behavior can also cause problems if the cleared-register state is undefined in the state diagram, because if the transition conditions are not met for any state, the machine goes to an undefined state and stays there.

To avoid problems caused by this clearing behavior, always have a state assigned to the cleared-register state. Or, if you do not assign a state to the cleared-register state, define every possible condition so some condition is always met for each state. You can also use the automatic transition to the cleared-register state by eliminating product terms and explicit definitions of transitions. You can also use the cleared-register state to satisfy illegal conditions.

Other Flip-flops

If none of the state conditions is met in a state machine that employs JK, RS, and T-type flip-flops, the state machine does not advance to the next state, but holds its present state due to the low input to the register from the OR array output. In such a case, the state machine can get stuck in a state. You can use this holding behavior to your advantage in some designs.

Precautions for Using Don't Care Optimization

When you use don't care optimization, you need to avoid certain design practices. The most common design technique that conflicts with this optimization is mixing equations and state diagrams to describe default transitions. For example, consider the design shown in the following figure.

```
module TRAFFIC
title 'Traffic Signal Controller'

        Clk,SenA,SenB   pin  1, 8, 7;
        PR              pin  16;        "Preset control
        GA,YA,RA        pin  15..13;
        GB,YB,RB        pin  11..9;

        "Node numbers are not required if fitter is used
        S3..S0     node 31..34 istype 'reg_sr,buffer';
        COMP       node 43;

        H,L,Ck,X        = 1, 0, .C., .X.;
        Count           = [S3..S0];

"Define Set and Reset inputs to traffic light flip-flops
        GreenA  = [GA.S,GA.R];
        YellowA = [YA.S,YA.R];
        RedA    = [RA.S,RA.R];
        GreenB  = [GB.S,GB.R];
        YellowB = [YB.S,YB.R];
        RedB    = [RB.S,RB.R];
        On      = [ 1  ,  0 ];
        Off     = [ 0  ,  1 ];

" test_vectors edited
equations
        [GB,YB,RB].AP = PR;
        [GA,YA,RA].AP = PR;
        [GB,YB,RB].CLK = Clk;
        [GA,YA,RA].CLK = Clk;
        [S3..S0].AP = PR;
        [S3..S0].CLK = Clk;
```

Figure 4-14. State Machine Description with Conflicting Logic

```
"Use Complement Array to initialize or restart
        [S3..S0].R        = (!COMP & [1,1,1,1]);
        [GreenA,YellowA,RedA]  = (!COMP & [On ,Off,Off]);
        [GreenB,YellowB,RedB]  = (!COMP & [Off,Off,On ]);

state_diagram Count
        State 0:        if ( SenA & !SenB ) then 0 with COMP = 1;
                        if (!SenA &  SenB ) then 4 with COMP = 1;
                        if ( SenA == SenB ) then 1 with COMP = 1;

        State 1:        goto  2 with COMP = 1;
        State 2:        goto  3 with COMP = 1;
        State 3:       goto  4 with COMP = 1;
        State 4:        GreenA  = Off;
                        YellowA = On ;
                        goto  5 with COMP = 1;

        State 5:        YellowA = Off;
                        RedA    = On ;
                        RedB    = Off;
                        GreenB  = On ;
                       goto  8 with COMP = 1;
         State 8:        if (!SenA &  SenB ) then  8 with COMP = 1;
                        if ( SenA & !SenB ) then 12 with COMP = 1;
                        if ( SenA == SenB ) then  9 with COMP = 1;

        State 9:        goto  10 with COMP = 1;
        State 10:       goto  11 with COMP = 1;
        State 11:      goto  12 with COMP = 1;
        State 12:       GreenB  = Off;
                        YellowB = On ;
                       goto  13 with COMP = 1;
        State 13:       YellowB = Off;
                        RedB    = On ;
                        RedA    = Off;
                        GreenA  = On ;
                        goto  0 with COMP = 1;
end
```

Figure 4-14 State Machine Description with Conflicting Logic (Continued)

This design uses the complement array feature of the Signetics FPLA devices to perform an unconditional jump to state [0,0,0,0]. If you use the **@DCSET** directive, the equation that specifies this transition

```
[S3,S2,S1,S0].R = (!COMP & [1,1,1,1]);
```

will conflict with the dc-set generated by the state diagram for S3.R, S2.R, S1.R, and S0.R. If equations are defined for state bits, the **@DCSET** directive is incompatible. This conflict would result in an error and failure when the logic for this design is optimized.

To correct the problem, you must remove the **@DCSET** directive so the implied dc-set equations are folded into the off-set for the resulting logic function. Another option is to rewrite the module as shown below.

```
module TRAFFIC1
title 'Traffic Signal Controller'


        Clk,SenA,SenB   pin   1, 8, 7;
        PR              pin  16;        "Preset control
        GA,YA,RA        pin  15..13;
        GB,YB,RB        pin  11..9;

        S3..S0          node 31..34 istype 'reg_sr,buffer';
        H,L,Ck,X        = 1, 0, .C., .X.;
        Count           = [S3..S0];

"Define Set and Reset inputs to traffic light flip flops
        GreenA  = [GA.S,GA.R];
        YellowA = [YA.S,YA.R];
        RedA    = [RA.S,RA.R];
        GreenB  = [GB.S,GB.R];
        YellowB = [YB.S,YB.R];
        RedB    = [RB.S,RB.R];
        On      = [ 1  ,  0 ];
        Off     = [ 0  ,  1 ];
" test_vectors edited
equations
        [GB,YB,RB].AP = PR;
        [GA,YA,RA].AP = PR;
        [GB,YB,RB].CLK = Clk;
        [GA,YA,RA].CLK = Clk;
        [S3..S0].AP = PR;
        [S3..S0].CLK = Clk;
```

Figure 4-15. @DCSET-compatible State Machine Description

```
@DCSET
state_diagram Count
        State 0:        if ( SenA & !SenB ) then 0;
                        if (!SenA &  SenB ) then 4;
                        if ( SenA == SenB ) then 1;
        State 1:        goto  2;
        State 2:        goto  3;
        State 3:        goto  4;

        State 4:        GreenA  = Off;
                        YellowA = On ;
                       goto  5;
        State 5:        YellowA = Off;
                        RedA    = On ;
                        RedB    = Off;
                        GreenB  = On ;
                       goto  8;
        State 6:        goto  0;
        State 7:        goto  0;

        State 8:        if (!SenA &  SenB ) then  8;
                        if ( SenA & !SenB ) then 12;
                        if ( SenA == SenB ) then  9;
        State 9:        goto  10;
        State 10:       goto  11;
        State 11:      goto  12;
        State 12:       GreenB  = Off;
                        YellowB = On ;
                       goto  13;
        State 13:       YellowB = Off;
                        RedB    = On ;
                        RedA    = Off;
                        GreenA  = On ;
                       goto  0;
        State 14:      goto  0;
        State 15:       "Power up and preset state
                        RedA    = Off;
                        YellowA = Off;
                        GreenA  = On ;
                        RedB    = On ;
                        YellowB = Off;
                        GreenB  = Off;
                        goto  0;
end
```

Figure 4-15 @DCSET-compatible State Machine Description (Continued)

Number Adjacent States for One-bit Change

You can reduce the number of product terms produced by a state diagram by carefully choosing state register bit values. Your state machine should be described with symbolic names for the states, as described above. Then, if you assign the numeric constants to these names so the state register bits change by only one bit at a time as the state machine goes from state to state, you will reduce the number of product terms required to describe the state transitions.

As an example, take the states A, B, C, and D, which go from one state to the other in alphabetical order. The simplest choice of bit values for the state register is a numeric sequence, but this is not the most efficient method. To see why, examine the following bit value assignments. The preferred bit values cause a one-bit change as the machine moves from state B to C, whereas the simple bit values cause a change in both bit values for the same transition. The preferred bit values produce fewer product terms.

State	Simple Bit Values	Preferred Bit Values
A	00	00
B	01	01
C	10	11
D	11	10

If one of your state register bits uses too many product terms, try reorganizing the bit values so that state register bit changes in value as few times as possible as the state machine moves from state to state.

Obviously, the choice of optimum bit values for specific states can require some tradeoffs; you may have to optimize for one bit and, in the process, increase the value changes for another. The object should be to eliminate as many product terms as necessary to fit the design into the device.

Use State Register Outputs to Identify States

Sometimes it is necessary to identify specific states of a state machine and signal an output that the machine is in one of these states. Fewer equations and outputs are needed if you organize the state register bit values so one bit in the state register determines if the machine is in a state of interest. Take, for example, the following sequence of states in which identification of the C*n* states is required:

State Register Bit Values

State Name	Q3	Q2	Q1
A	0	0	0
B	0	0	1
C1	1	0	1
C2	1	1	1
C3	1	1	0
D	0	1	0

This choice of state register bit values allows you to use Q3 as a flag to indicate when the machine is in any of the C*n* states. When Q3 is high, the machine is in one of the C*n* states. Q3 can be assigned directly to an output pin on the device. Notice also that these bit values change by only one bit as the machine cycles through the states, as is recommended in the section above.

Using Symbolic State Descriptions

Symbolic state descriptions describe a state machine without having to specify actual state values. A symbolic state description is shown below.

```
module SM
  a,b,clock         pin;                      " inputs
  a_reset,s_reset   pin;                      " reset inputs
  x,y               pin istype 'com';         " simple outputs

  sreg1        state_register;
  S0..S3       state;

equations
  sreg1.clk = clock;

state_diagram sreg1
  state S0:
      goto S1 with {x = a & b;
                    y = 0;    }
  state S1: if (a & b)
              then S2 with {x = 0;
                            y = 1; }
  state S2: x = a & b;
            y = 1;
            if (a) then S1 else S2;
  state S3:
      goto S0 with {x = 1;
                    y = 0; }

  async_reset S0: a_reset;
  sync_reset S0: s_reset;
end
```

Figure 4-16. Symbolic State Description

Symbolic state descriptions use the same syntax as non-symbolic state descriptions; the only difference is the addition of the **STATE_REGISTER** and **STATE** declarations, and the addition of symbolic synchronous and asynchronous reset statements.

Symbolic Reset Statements

In symbolic state descriptions, the **SYNC_RESET** and **ASYNC_RESET** statements specify synchronous or asynchronous state machine reset logic. For example, to specify that a state machine must asynchronously reset to state Start when the Reset input is true, you write

```
ASYNC_RESET Start : (Reset) ;
```

Symbolic Test Vectors

You can also write test vectors to refer to symbolic state values by entering the symbolic state register name in the test vector header (in the output sections), and the symbolic state names in the test vectors as output values.

Using Complement Arrays

The complement array is a unique feature found in some logic sequencers. This section shows a typical use ending counter sequence.

You can use transition equations to express the design of counters and state machines in some devices with JK or SR flip-flops. A transition equation expresses a state of the circuit as a variation of, or adjustment to, the previous state. This type of equation eliminates the need to specify every node of the circuit; you can specify only those that require a transition to the opposite state.

An example of transition equations is shown in Figure 4-17, a source file for a decade counter having a single (clock) input and a single latched output. This counter divides the clock input by a factor of ten and generates a 50% duty-cycle squarewave output. In addition to its registered outputs, this device contains a set of “buried” (or feedback) registers whose outputs are fed back to the product term inputs. These nodes must be declared, and can be given any names.

Node 49, the complement array feedback, is declared (as COMP) so that it can be entered into each of the equations. In this design, the complement array feedback is used to wrap the counter back around to zero from state nine, and also to reset it to zero if an illegal counter state is encountered. Any illegal state (and also state 9) will result in the absence of an active product term to hold node 49 at a logic low. When node 49 is low, product term 9 resets each of the feedback registers so the counter is set to state zero. (To simplify the following description of the equations in Figure 4-17, node 49 and the complement array feedback are temporarily ignored.)

The first equation states that the F0 (output) register is set (to provide the counter output) and the P0 register is set when registers P0, P1, P2, and P3 are all reset (counter at state zero) and the clear input is low. The complemented outputs of the registers (with the clear input low) form product term 0. Product term 0 sets register P0 to increment the decade counter to state 1, and sets register F0 to provide an output at pin 18.

```
module DECADE
title 'Decade Counter   Uses Complement Array
Michael Holley   Data I/O Corp'

        decade          device 'F105';

        Clk,Clr,F0,PR   pin  1,8,18,19;
        P3..P0     node 40..37;
        COMP            node 49;

        F0,P3..P0  istype 'reg_sr,buffer';
        _State          = [P3,P2,P1,P0];
        H,L,Ck,X        = 1, 0, .C., .X.;

equations
    [P3,P2,P1,P0,F0].ap  = PR;
    [F0,P3,P2,P1,P0].clk = Clk;

"Output          Next State            Present State        Input
 [F0.S, COMP,            P0.S] = !P3.Q & !P2.Q & !P1.Q & !P0.Q & !Clr; "0 to 1
 [     COMP,        P1.S,P0.R] = !P3.Q & !P2.Q & !P1.Q &  P0.Q & !Clr; "1 to 2
 [     COMP,             P0.S] = !P3.Q & !P2.Q &  P1.Q & !P0.Q & !Clr; "2 to 3
 [     COMP,   P2.S,P1.R,P0.R] = !P3.Q & !P2.Q &  P1.Q &  P0.Q & !Clr; "3 to 4
 [     COMP,             P0.S] = !P3.Q &  P2.Q & !P1.Q & !P0.Q & !Clr; "4 to 5
 [F0.R, COMP,       P1.S,P0.R] = !P3.Q &  P2.Q & !P1.Q &  P0.Q & !Clr; "5 to 6
 [     COMP,             P0.S] = !P3.Q &  P2.Q &  P1.Q & !P0.Q & !Clr; "6 to 7
 [   COMP,P3.S,P2.R,P1.R,P0.R] = !P3.Q &  P2.Q &  P1.Q &  P0.Q & !Clr; "7 to 8
 [   COMP                P0.S] =  P3.Q & !P2.Q & !P1.Q & !P0.Q & !Clr; "8 to 9
 [        P3.R,P2.R,P1.R,P0.R] =                                !COMP; "Clear
"After Preset, clocking is inhibited until High-to-Low clock transition.
test_vectors   ([Clk,PR,Clr] -> [_State,F0  ])
                [ 0 , 0, 0 ] -> [   X   , X];
                [ 1 , 1, 0 ] -> [^b1111, H]; " Preset high
                [ 1 , 0, 0 ] -> [^b1111, H]; " Preset low
                [ Ck, 0, 0 ] -> [   0   , H]; " COMP forces to State 0
                [ Ck, 0, 0 ] -> [   1   , H];
"               ..vectors edited...
                [ Ck, 0, 1 ] -> [   0   , H]; " Clear
end
```

Figure 4-17. Transition Equations for a Decade Counter

The second equation performs a transition from state 1 to state 2 by setting the P1 register and resetting the P0 register. (The .R dot extension is used to define the reset input of the registers.) In state 2, the F0 register remains set, maintaining the high output. The third equation again sets the P0 register to achieve state 3 (P0 and P1 both set), while the fourth equation resets P0 and P1, and sets P2 for state 4, and so on.

Wraparound of the counter from state 9 to state 0 is achieved by means of the complement array node (node 49). The last equation defines state 0 (P3, P2, P1, and P0 all reset) as equal to !COMP, that is, node 49 at a logic low. When this equation is processed, the fuses are blown. As a result, the !COMP signal is true to generate product term 9 and reset all the “buried” registers to zero.

ABEL-HDL and Truth Tables

Truth Tables in ABEL-HDL represent a very easy and straightforward description method, well suited in a number of situations involving combinational logic.

The principle of the Truth Table is to build an exhaustive list of the input combinations (referred to as the ON-set) for which the output(s) become(s) active.

The following list summarizes design considerations for Truth Tables. Following the list are more detailed examples.

- The OFF-set lines in a Truth Table are necessary when more than one output is assigned in the Truth Table. In this case, not all Outputs are fired under the same conditions, and therefore OFF-set conditions do exist.
- OFF-set lines are ignored because they represent the default situation, unless the output variable is declared dc. In this case, a third set is built, the DC-set and the Output inside it is assigned proper values to achieve the best logic reduction possible.
- If output type dc (or @dcset) is not used and multiple outputs are specified in a Truth table, consider the outputs one by one and ignore the lines where the selected output is not set.
- Don't Cares (.X.) used on the right side of a Truth Table have no optimization effect.
- When dealing with multiple outputs of different kind, avoid general settings like @DCSET which will affect all your outputs. Use istype ‘.....DC’ on outputs for which this reduction may apply.
- Beware of Outputs for which the ON-set might be empty.
- As a general guideline, it is important not to rely on first impression or simple intuition to understand Truth tables. The way they are understood by the compiler is the only possible interpretation. This means that Truth Tables should be presented in a clear and understandable format, should avoid side effects, and should be properly documented (commented).

Basic Syntax - Simple Examples

In this example, the lines commented as L1 and L2 are the ON-set.

Lines L3 and L4 are ignored because Out is type *default* (meaning ‘0’ for unspecified combinations). The resulting equation does confirm this.

```
MODULE DEMO1
TITLE 'Example 1'
" Inputs
   A, B, C  pin;
"Output
   Out  pin istype 'com';
Truth_Table
  ([A,B,C] -> Out )
   [0,1,0] -> 1;  // L1
   [1,1,1] -> 1;  // L2
   [0,0,1] -> 0;  // L3
   [1,0,0] -> 0;  // L4
END
// Resulting Reduced Equation :
// Out = (!A & B & !C)  #  (A & B & C);
```

Example 2 differs from example 1 because Out is now type ‘COM, **DC**’. (optimizable don’t care).

In this case, the lines commented as L1 and L2 are the ON-set, L3 and L4 are the *OFF-set* and other combinations become *don’t care* (DC-set) meaning 0 or 1 to produce the best logic reduction. As a result in this example, the equation is VERY simple.

@DCSET instruction would have produced the same result as to declare Out of type dc. But @DCSET must be used with care when multiple outputs are defined: they all become dc.

```
MODULE DEMO1
TITLE 'Example 2'
" Inputs
   A, B, C  pin;
"Output
   Out  pin istype 'com, dc';
Truth_Table
  ([A,B,C] -> Out )
   [0,1,0] -> 1;  // L1
   [1,1,1] -> 1;  // L2
   [0,0,1] -> 0;  // L3
   [1,0,0] -> 0;  // L4
END
// Resulting Reduced Equation :
// Out = (B);
```

Influence of Signal polarity

We will see now with example 3 how the polarity of the signal may influence the truth table:

In this example, Out1 and Out2 are strictly equivalent. For !Out1, note that the ON-set is the 0 values. *The third line L3 is ignored*.

```
MODULE DEMO2
TITLE 'Example 3'
" Inputs
   A, B, C  pin;
"Output
   Out1pin istype 'com, neg';
   Out2pin istype 'com, neg';
   Out3pin istype 'com, neg'; // BEWARE

Truth_Table
  ([A,B,C] -> [!Out1, Out2, Out3] )
   [0,0,1] -> [  0,     1,    0 ];//L1
   [0,1,1] -> [  0,     1,    0 ];//L2
   [1,1,0] -> [  1,     0,    1 ];//L3
END
// Resulting Equations :
//     !Out1 = !Out2 = (A # !C);
// or:  Out1 =  Out2 = (!A & C);
// BUT: Out3 = (A & B & !C); <<what you wanted ?
```

For active-low outputs, one must be careful to specify **1** for the active state if the Output appears without the exclamation point (!).
0 must be used when **!**output is defined in the table header.

We recommend the style used for Out1.

For Out3, line used is L3, L1 and L2 are ignored.

Using .X. in Truth tables conditions

Don't Care used on the left side in Truth tables have no optimization purpose. they only serve as a shortcut to write several conditions in one single line.

Be careful when using .X. in conditions. This can lead to overlapping conditions which look not consistent (see example below). Due to the way the compiler work, this type of inconsistency is not checked nor reported. In fact, only the ON-set condition is taken into account, the OFF-set condition is ignored.

The following example illustrates this:

```
MODULE DEMO3
TITLE 'Example 4'
" Inputs
   A, B, C  pin;
"Output
   Outpin istype 'com';
" Equivalence
   X = .X.;
Truth_Table
  ([A,B,C] -> Out )
   [0,0,1] ->  0; //L1 ignored in fact
   [0,1,0] ->  1; //L2
   [1,X,X] ->  1; //L3
   [0,0,1] ->  1; //L4 incompatible
   [1,1,0] ->  0; //L5 incompatible
END
// Result : Out = A # (B & !C) # (!B & C)
```

L1 is in fact ignored. Out is active high, therefore only line L4 is taken into account.

Likewise, L5 intersects L3, but is ignored since it is not in the ON-set for Out.

Globally, only L2, L3 and L4 are taken into account, as we can check in the resulting equation, without any error reported.

Using .X. on the right side

The syntax allows to use .X. as a target value for an output. In this case, the condition is simply ignored.

NOTE This is **not** the method to specify optimizable don't care states. See example 2 for such an example.

Example 6 shows that-> .X. states are not optimized if DC type or @DCSET are not used.

These lines are ALWAYS ignored.

```
MODULE DEMO6
TITLE 'Example 6'
" Inputs
   A, B, C  pin;
"Output
   Outpin istype 'com';
" Equivalence
   X = .X.;
Truth_Table
  ([A,B,C] -> Out )
   [0,0,0] ->  0;
   [0,0,1] ->  X;
   [0,1,0] ->  1;
   [0,1,1] ->  X;
   [1,X,X] ->  X;
END
// As is : Out = (!A & B & !C);
// With istype 'com,DC' : Out = (B);
```

They are in fact of no use, except maybe as a way to document that output does not matter.

Special case: Empty ON-set

There is a special case which is unlikely to happen, but may sometimes occurs. Consider this example:

```
MODULE DEMO5
TITLE 'Example 5'
" Inputs
   A, B, C  pin;
"Output
   Outpin istype 'com, pos';
Truth_Table
   ([A,B,C] ->  Out )
    [0,0,1] ->  0;
    [0,1,0] ->  0;
    [1,0,0] ->  0;
//  [0,0,0] ->  1;//changes everything!
END
// Without the last line L4 :
// !Out=(A & !B & !C)# (!A & B & !C)# (!A & !B & C);
// WITH L4 : Out = (!A & !B & !C);
```

What we obtain is slightly unexpected. This table should produce Out=0; as the result. (We enumerated only OFF conditions, and the polarity is POS (or default), so unlisted cases should also turn into zeroes.)

One reason to build such a table could be when multiple outputs are defined, and when Out needs to be shut off for whatever reason.

In the absence of the line L4, the result is not intuitive. The output is 0 only for the listed cases (L1, L2, L3), and is 1 for all other cases, even if dc or pos is used.

When line L4 is restored, then the output equation becomes Out = (!A & !B & !C); because we fall in the general situation where the ON-set is not empty.

Registered Logic in Truth tables

Truth Tables can specify registered outputs. In this case, the assignment become **:>** (instead of **->**).

For more information, refer to the ***ABEL-HDL Reference Manual***.

Index

Symbols
'attribute'
  and polarity control 54
'collapse'
  selective collapsing 42
'neg'
  and polarity control 53
.D 56
.FB 55
.PIN 55
.Q 55
:=
  alternate flip-flop types 50
@DCSET
  example 61
  with state machines 68
'xor' 62
“collapse”
  collapsing nodes 42
“Keep”
  collapsing nodes 42

A
ABEL-HDL
  enter an ABEL-HDL description 25
  enter logic description 27
  enter test vectors 28
  overview 14
  properties 31
  strategies 32
ABEL-HDL Compiling 24
Active-low declarations 51
actlow1.abl 52
actlow2.abl 51
Attributes
  and architecture independence 43
Architecture independence
  attributes 43
  dot extensions 43, 56
  dot extensions, example 57
  resolving ambiguities 44
Arrays, complement 75
Attributes
  collapsing nodes 42
  in lower-level sources 39
Auto-update 29

B
Bottom-up design 20

C
Collapsing nodes 42
  selective 42
Combinational nodes 40
Compilation 17
Complement arrays 75
  example 76

D
D flip-flop
  unsatisfied transition conditions 67
Dangling nodes 40
dc
  and polarity control 53
dc.abl 61
Dc-set 60
  and optimization 61
decade.abl 76
Declarations
  active-low 51
Design hierarchy 17
Design Overview
  compilation 17
  device programming 17
  hierarchy 17
  projects 15
  simulation 17
  sources 16
Dot extensions
  and detail descriptions 58
Detail descriptions 45
  and macrocells 45
  example, dot extensions 58, 59
  example, inverting 48
  example, non-inverting 47
  when to use 50

detail1.abl 58
detail2.abl 59
Device programming 17
Devices
  programmable polarity 53
Don't Care .X.
  on left side of Truth Table 80
  on right side of Truth Table 81
Detail descriptions
  and dot extensions 58
Dot extensions
  .D 56
  .FB 55
  .PIN 55
  .Q 55
  and architecture independence 43, 56
  and architecture independence,
    example 57
  and feedback 55
  example, detail 58, 59
  no 55

E
Emulation of flip-flops 63
Equation polarity 53
Equations
  for flip-flops 54
  XOR 62

F
Feedback
  and dot extensions 55
  merging 41
Flip-flops
  and dot extensions 54
  detail descriptions 50
  D-type 67
  emulation with XORs 63
  state diagrams 50
  using := with 50

H
Hierarchical design
  abstract 19
  advantages of 19
  approaches to 19
  bottom-up 20
  defined for ABEL-HDL 20
  mixed 20
  philosophy 19
  symbols in 20
  techniques 19
  top-down 20
Hierarchical levels
  defined 18
Hierarchy 17, 38
  modular design 18, 19

I
Identifiers
  in state machines 65
Inside-out design 20
Instantiation 38
Interface
  submodule 39
Istype, and polarity control 54

J
JK flip-flop
  and := 50
  emulation of 63

L
Linking modules
  merging feedbacks 41
  post-linked optimization 41
Lower-level sources 39
  instantiating 38

M
Mixed design 20

N
Node
  collapsing 42
  combinational 40
  complement arrays 75
  dangling 40
  registered 40
  removing redundant 41
  selective collapsing 42

O
Off-set 60
One-bit changes 72
On-set 60
  in Truth Tables 78
Optimization
  and @DCSET 61
  of XORs 62
  post-linked 41
  reducing product terms 72
Output enables 39

P
pin2pin.abl 57
Pin-to-pin descriptions 44
  and flip-flops 54
  example 47
  resolving ambiguities 44
Polarity control 53
  active levels 53
Ports
  declaring lower-level 39
Post-linked Optimization 41
Powerup state 67
Preset
  built-in, example 48
Product terms
  reducing 72
Programmable designing 12
Programmable polarity, active levels for devices 53
Project sources 16
Properties 31

Q
Q11.abl 47
Q12.abl 47
Q13.abl 48
Q15.abl 49
Q17.abl 49

R
Redundant nodes 41
Registered design descriptions 44
Registered nodes 40
Registers
  bit values in state machines 72
  cleared state in state machines 67
  powerup states 67
Reset
  example, inverted architecture 49
  example, non-inverted architecture 49
  resolving ambiguities 49

S
Selective collapsing 42
sequence.abl 66
Simulation 17
Sources
  ABEL-HDL 16
  device 16
  graphic waveform stimulus 16
  project notebook 16
  schematic 16
  test vector 16
  Verilog HDL 16
  Verilog test fixture 16
  VHDL 16
  VHDL test bench 16
SR flip-flop
  and := 50
State machine example 66
  @DCSET 70
  no @DCSET 68
State machines
  and @DCSET 61, 68
  cleared register state 67
  design considerations 65
  identifiers in 65
  identifying states 72
  illegal states 67
  powerup register states 67
  reducing product terms 72
  using state register outputs 72
State registers 72
Strategies 32
Symbolic state descriptions 74

T
T flip-flop
  and equations 54
Top-down design 20
traffic.abl 68
traffic1.abl 70
Transferring designs 43
Transition conditions 67
Tristate outputs 39
Truth Tables
  ABEL-HDL 77

X
x1.abl 62
x2.abl 62
XORs
  and operator priority 63
  example 62
  flip-flop emulation 63
  implied 62
  optimization of 62