ULTRA LOGIC

Warp™
User's Guide

CYPRESS

Warp™

VHDL Development System

User's Guide

Cypress Semiconductor
3901 North First Street
San Jose, CA 95134
(408) 943-2600
April 1996

The following are trademarks or registered trademarks of Cypress Semiconductor Corporation: Warp, Warp2, Warp3, Nova, Galaxy, Flash370, UltraLogic, Impulse3, UltraGen, pASIC380, ISR, MAX340.

The following are trademarks or registered trademarks of Viewlogic Systems:
Powerview, Workview, Workview PLUS, ViewDraw, ViewSim, ViewTrace, ViewText, Cockpit, VCS.

The following are trademarks or registered trademarks of Microsoft Corporation: Microsoft, Windows, Windows 3.1, Windows 3.11, Windows NT, Windows 95.

The following are trademarks or registered trademarks of QuickLogic Corporation: SpDE, pASIC.

The following is a registered trademark of Intel Corporation: Pentium.

The following is a trademark of Hewlett Packard Corporation: HP-UX.

The following are trademarks of Mentor Graphics Corporation: QuickHDL, V-System.

The following is a trademark of Veribest, Inc.: Veribest.

The following is a registered trademark of AT&T: UNIX.

The following are trademarks or registered trademarks of Synopsys, Inc.: Synopsys, Design Compiler, VSS.

The following are trademarks or registered trademarks of Cadence Design Systems Inc.: Verilog, Leapfrog, Verilog-XL.

The following are trademarks or registered trademarks of Sun Microsystems, Inc.: Sun SparcStation, SunOS, Solaris.

The following is a registered trademark of Open Software Foundation: Motif.

Cypress Semiconductor Corporation may revise this publication from time to time without notice. Some states or jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

Cypress Software License Agreement

LICENSE. Cypress Semiconductor Corporation ("Cypress") hereby grants you, as a Customer and Licensee, a single-user, non-exclusive license to use the enclosed Cypress software program ("Program") on a single CPU at any given point in time. Cypress authorizes you to make archival copies of the software for the sole purpose of backing up your software and protecting your investment from loss.

TERM AND TERMINATION. This Agreement is effective from the date the diskettes are received until this Agreement is terminated. The unauthorized reproduction or use of the Program and/or documentation will immediately terminate this Agreement without notice. Upon termination you are to destroy both the Program and the documentation.

COPYRIGHT AND PROPRIETARY RIGHTS. The Program and documentation are protected by both United States Copyright Law and International Treaty provisions. This means that you must treat the documentation and Program just like a book, with the exception of making archival copies for the sole purpose of protecting your investment from loss. The Program may be used by any number of people, and may be moved from one computer to another, so long as there is **No Possibility** of its being used by two people at the same time.

DISCLAIMER. THIS PROGRAM AND DOCUMENTATION ARE LICENSED "AS-IS," WITHOUT WARRANTY AS TO PERFORMANCE. CYPRESS EXPRESSLY DISCLAIMS ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS OF THIS PROGRAM FOR A PARTICULAR PURPOSE.

LIMITED WARRANTY. The diskette on which this Program is recorded is guaranteed for 90 days from date of purchase. If a defect occurs within 90 days, contact the representative at the place of purchase to arrange for a replacement.

RESELLING. The reselling or distribution of this product can be done by Cypress authorized distributors only.

BENCHMARKING. This license Agreement does not convey to you the right to publish performance benchmarking results involving any Cypress *Warp* products. Permission to publish performance benchmarking results involving any Cypress *Warp* products must be received in writing from Cypress Semiconductor prior to publishing.

LIMITATION OF REMEDIES AND LIABILITY. IN NO EVENT SHALL CYPRESS BE LIABLE FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM PROGRAM USE, EVEN IF CYPRESS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. CYPRESS'S EXCLUSIVE LIABILITY AND YOUR EXCLUSIVE REMEDY WILL BE IN THE REPLACEMENT OF ANY DEFECTIVE DISKETTE AS PROVIDED ABOVE. IN NO EVENT SHALL CYPRESS'S LIABILITY HEREUNDER EXCEED THE PURCHASE PRICE OF THE SOFTWARE.

ENTIRE AGREEMENT. This Agreement constitutes the sole and complete Agreement between Cypress and the Customer for use of the Program and documentation. Changes to this Agreement may be made only by written mutual consent.

GOVERNING LAW. This Agreement shall be governed by the laws of the State of California. Should you have any question concerning this Agreement, please contact:

Cypress Semiconductor Corporation
Attn: Legal Counsel
3901 N. First Street
San Jose, CA 95134-1599

408-943-2600

Contents

Installation

PC Installation

To install *Warp*™, insert the CD-ROM and using the File Manager in Windows™, locate *pc\setup.exe* on the CD-ROM. Double click your left mouse button on this file. Cypress recommends that you close all your applications (except for the File Manager and the Program Manager) before running the *Warp* installation program. *Warp* Release 4.0 is a major release and should not be installed on a previous version of *Warp*. Cypress recommends installing *Warp* Release 4.0 in a new directory. If you have deleted a version of *Warp* (for *Warp3*™ customers), entering Windows after doing so might give you some messages about missing device files: this is harmless. You will have to edit the *c:\windows\system.ini* file manually to remove these devices from the file.

Warp is a 32-bit program and requires that Microsoft's® Win32s subsystem be loaded for Windows 3.1™ or Windows 3.11™. This is not required for Windows 95™or Windows NT™. The first part of the Windows installation routine will check for the existence of Win32s and offer to install it for you.

For previous *Warp3* customers, Viewlogic tools required BIGWIN in order to run. The current Viewlogic tools shipped with *Warp* no longer require this. The *Warp* installation program does not automatically remove this device from your *system.ini* file because you may have other applications that might require BIGWIN. To remove BIGWIN, you must edit the *system.ini* file in your Windows directory manually.

The following are the approximate disk space requirements for the various *Warp* products:

1. *Warp2* 60Mbytes
2. *Warp3* 110Mbytes (150 with on-line Documentation)

You may also need extra disk space for your designs.

Cypress also recommends that you run *Warp* on at least a 486-66 or a Pentium®, with at least 16MB of RAM and an additional 16MB of virtual memory.

The *Warp2*™ product is also capable of integrating itself with the Viewlogic WorkView® PLUS environment if you already have this. *Warp2* has been tested with WorkView PLUS® version 5.2.

Warp On-line Documents

Warp documentation is available on-line and it can be viewed using the **Adobe Acrobat Reader**, shipped with the CD-ROM. To install the Acrobat Reader, insert the CD-ROM and using the File Manager in Windows, locate *\pc\acroread\disk1\setup.exe* and double-click your left mouse button on this file. These files can also be installed into the *c:\warp\docs* directory during installation.

PLD Data Book

The Cypress *Programmable Logic Data Book* is also available on-line. It can be viewed using the **Acrobat Reader,** shipped with the CD-ROM. To access the databook, insert the CD-ROM and use the File Manager in Windows to locate the *\doc\databook* directory.

PC License Issues for Viewlogic Tools (*Warp3* Customers)

PC *Warp3* customers require a hardware key as well as a license file in order to use the software. This key and license file are used solely for the Viewlogic tools within the *Warp3* environment. To obtain a license file for your *Warp3* software, please fill out the registration form that was included with your *Warp3* software kit, and fax it to the number shown on the form. The HOSTID is specified on the key itself and the license file is generated based on that number. You will receive your license file within 24 hours of Cypress receiving the registration form.

PC - Known Installation Problems

Problem: *Warp* installation sometimes aborts, giving a "Divide by 0" error.

Solution: Please try installing *Warp* again. This problem will be fixed in the next update and involves the calculation of free disk space. Installing a second time changes the free disk space situation and thus avoids the error.

Problem: *Warp* installation does not work, or Viewlogic tools do not work properly.

Solution: Some newer PCs with SCSI drives must run SMARTDRIVE. Without this, some 32-bit applications have trouble accessing the disk. To avoid this problem, please make sure that you are running SMARTDRIVE. The following command in your *autoexec.bat* or *config.sys* is recommended:

```
c:\windows\smartdrive.exe /double_buffer
```

Running SMARTDRIVE even on non-SCSI machines is also a good idea, since it improves the performance of your computer.

Problem: Windows fails to come up after *Warp* installation.

Solution: Rare as this is, some configurations of Windows do this. To solve this problem, please check your *c:\windows\system.ini* file for any duplicate device drivers. If any are found, please remove the duplication, leaving only one instance of each device driver.

Problem: *Warp* fails to install.

Solution: Certain power management tools and screen savers might conflict with the installation program. Temporarily disable the screen saver during installation. If *power.exe* is being run from your *autoexec.bat* or *config.sys*, please disable this.

Commonly Asked Questions About PC Installation

What is BIGWIN, and why do I no longer need it?

Windows is essentially a 16-bit environment. The previous version of Viewlogic (5.1) used a Windows extender called BIGWIN to allow 32-bit access. All Viewlogic tools which formerly used BIGWIN have now been ported to use Microsoft's recommended extender, Win32s.

Do I need to change any settings in my *config.sys*?

Please check the settings for FILES and BUFFERS. Cypress recommends a setting of 60 for FILES and 15 for BUFFERS.

Which mice are supported?

For *Warp2* customers, a generic 2-button mouse will suffice. However, if you have configured *Warp2* to run with Viewlogic or if you are a *Warp3* customer, Cypress recommends a 3-button mouse. Although a 2-button mouse will work, many of the WorkView PLUS functions require a middle mouse button. A 2-button mouse can be made to emulate a 3-button mouse within the Viewlogic tools by using the F2, F3, F4 function keys for the left, middle and the right mouse buttons, but the user must first click the left mouse button to use these function keys. For example, to emulate the middle button, you have to single-click the left mouse button of your 2-button mouse and then press F3.

Which files are modified by the *Warp* installation?

The files *autoexec.bat*, *c:\windows\win.ini* and *c:\windows\system.ini* are always modified. A detailed description of the exact changes that are made to these files is listed in the file *warppc.txt* (available at the end of the installation or with the Release Notes icon within the *Warp R4* program group). The changes vary depending upon the product that you have installed.

SunOS/Solaris/HP Installation

The following steps show how to install *Warp* Release 4 on a SUN Sparc station running SunOS™ 4.1.x/Solaris™ 2.5 or on an HP 9000 (700 series) workstation running HP-UX™ 9.05. The following are the disk space requirements depending on the product you have purchased:

1.	*Warp2*	60Mbytes
2.	*Warp3*	160Mbytes (260 with on-line Documentation)

Warp Release 4.0 is a major release and should not be installed over a previous version of *Warp*. Cypress recommends installing *Warp* Release 4.0 in a new directory.

The *Warp2* product is also capable of integrating itself with the Viewlogic Powerview® environment if you already have this. *Warp2* has been tested with Powerview version 5.3.2.

Warp documentation is available on-line and can be viewed using the **Adobe Acrobat Reader**, shipped with the CD-ROM. The Acrobat reader can be installed during *Warp* installation or separately after completing *Warp* installation.

PLD Data Book

The Cypress *Programmable Logic Data Book* is available on-line. It can be viewed using the **Acrobat Reader,** shipped with the CD-ROM. To access the data book, mount the CD-ROM on */cdrom* and view the databook located in the directory */cdrom/doc/databook*. Refer to the following steps to mount the CD-ROM.

Step 1: Mount the CD-ROM

On SunOS 4.1.x

Login as super user on a machine that has a CD-ROM drive. Run the following commands to create and mount the */cdrom* directory.

```
mkdir /cdrom
mount -rt hsfs /dev/sr0 /cdrom
```

On Solaris 2.5

If the Volume Manager is running, it will automatically mount the CD-ROM. Otherwise, login as super user on a machine that has a CD-ROM drive. Run the following commands to create and mount the */cdrom* directory:

```
mkdir /cdrom
mount -F ufs -r /dev/dsk/c0t6d0s2 /cdrom
```

On HP-UX 9.05

Login as super user on a machine that has a CD-ROM drive. Run the following commands to create and mount the */cdrom* directory:

```
mkdir /cdrom
mount -o ro /dev/dsk/c201d3s0 /cdrom
```

In the above procedure, it is assumed that the CD-ROM drive has a SCSI ID of 6 (for Solaris) and 3 (for HP-UX). Make sure that you specify the correct ID in the above mount commands.

Step 2: Local Installation

If you are installing from a CD-ROM that is on a remote machine, skip to Step 3.

On SunOS 4.1.x

```
    cd /cdrom/sunos
    ./install.wr4
```

To install the Acrobat Reader separately, run the following commands instead of the above commands:

```
    cd /cdrom/sunos/acroread
    ./install
```

On Solaris 2.5

```
    cd /cdrom/w3r40fcs/solaris
    ./install.wr4
```

To install the Acrobat Reader separately, run the following commands instead of the above commands:

```
    cd /cdrom/w3r40fcs/solaris/acroread
    ./install
```

On HP-UX 9.05

```
    cd /cdrom/HP
    ./INSTALL.WR4
```

The above program will guide you through the rest of the installation process.

To install the Acrobat Reader separately, run the following commands instead of the above commands:

```
    cd /cdrom/HP/ACROREAD
    ./INSTALL
```

After this step is completed, please skip to Step 4.

Step 3: Remote Installation

On SunOS 4.1.x

If you are installing from a CD-ROM that is on a remote machine, export the */cdrom* directory. Add the following line:

```
/cdrom -ro
```

to the file */etc/exports* on the remote machine, if the line does not already exist. If you are modifying an existing */etc/exports*, run the following command:

```
/usr/etc/exportfs -a
```

If */etc/exports* does not exist and you have had to create it to enter the above line, run the following commands:

```
sync
reboot
```

If your `sync` command also reboots the machine, you do not need to run the above `reboot` command.

Remote login to the install machine (where you want to install the software) using the following commands:

```
rlogin <install machine> -l root
```

If */cdrom* does not exist, run the following command:

```
mkdir /cdrom.
```

Mount the remote CD-ROM on the install machine, change the directory to the CD-ROM, and run the installation script by using the following commands:

```
mount -r <remote-machine-name>:/cdrom /cdrom
cd /cdrom/sunos
./install.wr4
```

The above program will walk you through the rest of the installation process.

To install the Acrobat Reader separately, run the following commands instead of the above commands:

```
mount -r <remote-machine-name>:/cdrom /cdrom
cd /cdrom/sunos/acroread
./install
```

On Solaris 2.5

If you are installing from a CD-ROM that is on a remote machine, export the */cdrom* directory. Add the following line:

```
share -F nfs -o ro /cdrom/warp
```

to the file */etc/dfs/dfstab* on the remote machine, if the line does not already exist. If `nfsd` is not running, reboot the machine.

Run the following command to export */cdrom/warp*:

```
shareall
```

Remote login to the install machine (where you want to install the software) using the following commands:

```
rlogin <install machine> -l root
```

If */cdrom* does not exist, run the following command:

```
mkdir /cdrom
```

Mount the remote CD-ROM on the install machine, change the directory to the CD-ROM, and run the installation script by using the following commands:

```
mount -r <remote-machine-name>:/cdrom/warp3r40 /cdrom
cd /cdrom/solaris
./install.wr4
```

The above program will walk you through the rest of the installation process.

To install the Acrobat Reader separately, run the following commands instead of the above commands:

```
mount -r <remote-machine-name>:/cdrom/warp3r40 /cdrom
cd /cdrom/solaris/acroread
./install
```

On HP-UX 9.05

If you are installing from a CD-ROM that is on a remote machine, export the */cdrom* directory. Add the following line:

```
/cdrom -ro
```

to the file */etc/exports* on the remote machine, if the line does not already exist. If you are modifying an existing */etc/exports*, run the following command:

```
/usr/etc/exportfs -a
```

If */etc/exports* does not exist and you have had to create it to enter the above line, reboot the machine.

Remote login to the install machine (where you want to install the software) by using the following commands:

```
rlogin <install machine> -l root
```

If */cdrom* does not exist, run the following command:

```
mkdir /cdrom
```

Mount the remote CD-ROM on the install machine, change the directory to the CD-ROM, and run the installation script by using the following commands:

```
mount -o ro <remote-machine-name>:/cdrom /cdrom
cd /cdrom/HP
./INSTALL.WR4
```

The above program will walk you through the rest of the installation process.

To install the Acrobat Reader separately, run the following commands instead of the above commands:

```
mount -o ro <remote-machine-name>:/cdrom /cdrom
cd /cdrom/HP/ACROREAD
./INSTALL
```

On all platforms:

Unmount the CD-ROM and come back to <CD-ROM_drive host> machine.

```
cd /
umount /cdrom
exit
```

Step 4: Unmount the CD-ROM

Type in these commands:

```
cd /
umount /cdrom
```

Update User's Login Start Up File

To run *Warp*, you need to set some environment variables in the user's login start up file.

For C Shell users:

If you are using *C shell*, add the following lines to your *~/.cshrc* file:

```
setenv CYPRESS_DIR <Warp installation directory>
set path = ($CYPRESS_DIR/bin $path)
setenv SPDE_ROOT $CYPRESS_DIR/spde
setenv XVTPATH   $SPDE_ROOT/print
setenv UIDPATH   $SPDE_ROOT/%N.uid
set path = ($SPDE_ROOT $path)
```

On Sun/Solaris, also include the following line:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${SPDE_ROOT}
```

On HP, also include the following line:

```
setenv SHLIB_PATH ${SPDE_ROOT}
```

On all platforms, copy the pASIC® control file *($CYPRESS_DIR/spde/.spderc)* into your home directory.

If you have purchased the *Warp2* product and would like to configure *Warp* to run with Viewlogic tools, in addition to the above lines, add/replace the WDIR environment variable in your *~/.cshrc* file:

```
setenv WDIR ~/<pvlocal>:$CYPRESS_DIR/warpstd:<PowerView-
Install-Dir>/standard
```

It is important that the *$CYPRESS_DIR/warpstd* directory appear before the Powerview directory.

If you have purchased the *Warp3* product, add the following lines to your *~/.cshrc* file:

```
setenv WDIR ~/<pvlocal>:$CYPRESS_DIR/warpstd:$CYPRESS_DIR
/pv/standard
set path = ($CYPRESS_DIR/pv $path)
```

As noted above, *<pvlocal>* is a local directory where user specific Powerview files and user design files can be stored.

If you have installed the **Acrobat Reader**, add the following lines to your *~/.cshrc* file:

```
setenv ACROBAT_DIR <Acrobat Reader installation
directory>
set path = ($ACROBAT_DIR/bin $path)
```

Source the *~/.cshrc* file.

For Bourne shell users:

If you are using *Bourne shell,* add the following lines to *$HOME/.profile* file in the same order.

```
CYPRESS_DIR=<Warp installation directory>
PATH=$CYPRESS_DIR/bin:$PATH
export CYPRESS_DIR PATH
SPDE_ROOT=$CYPRESS_DIR/spde
XVTPATH=$SPDE_ROOT/print
UIDPATH=$SPDE_ROOT/%N.uid
PATH=$SPDE_ROOT:$PATH
```

On Sun/Solaris, also include the following lines:

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$SPDE_ROOT
export SPDE_ROOT XVTPATH UIDPATH PATH LD_LIBRARY_PATH
```

On HP, also include the following lines:

```
SHLIB_PATH=$SPDE_ROOT
export SPDE_ROOT XVTPATH UIDPATH PATH SHLIB_PATH
```

On all platforms, copy the pASIC control file *($CYPRESS_DIR/spde/.spderc)* into your home directory.

If you have purchased the *Warp2* product and would like to configure *Warp* to run with Viewlogic tools, add/replace the WDIR environment variable in your *$HOME/.profile* file:

```
WDIR=$HOME/<pvlocal>:$CYPRESS_DIR/warpstd:<PowerView-
Install-Dir>/standard
export WDIR
```

It is important that the *$CYPRESS_DIR/warpstd* directory appear before the Powerview directory.

If you have purchased the *Warp3* product, add the following lines to your *$HOME/.profile* file:

```
WDIR=$HOME/<pvlocal>:$CYPRESS_DIR/warpstd:$CYPRESS_DIR/
pv/standard
PATH=$CYPRESS_DIR/pv:$PATH
export WDIR PATH
```

As noted above, *<pvlocal>* is a local directory where user specific Powerview files and user design files can be stored.

If you have installed the **Acrobat Reader**, add the following lines to your *$HOME/.profile* file:

```
ACROBAT_DIR=<Acrobat Reader installation directory>
PATH=$ACROBAT_DIR/bin:$PATH
export ACROBAT_DIR PATH
```

Source the *$HOME/.profile* file.

For both C-shell and Bourne shell users:

If you are running **SunOS 4.1.x**, make sure that */usr/openwin/lib* and */usr/motif/lib* are included in the LD_LIBRARY_PATH before running *Warp*.

If you are running **Solaris 2.5**, make sure that */usr/openwin/lib* is included in the LD_LIBRARY_PATH before running *Warp*.

If you have configured *Warp* to run with Viewlogic tools, also make sure that the WDIR path includes a writable directory (*<pvlocal>* above).

License Issues for Viewlogic Tools (*Warp3* Customers)

Warp3 customers on the workstation platforms also require a license file. Again, this license file enables the Viewlogic Powerview tools. However, for the workstation environment there are two types of license files available: node-locked and floating. For more information on these types of license files, please refer to the Viewlogic on-line documentation. Your *Warp3* registration form should already specify the type of license that was purchased. If this is not correct, please contact the sales office through which your software was purchased. Workstation license files are specified via the HOSTID which is unique to each workstation as opposed to a hardware key in the PC environment. Please make sure you specify your HOSTID on the registration form correctly.

Run *Warp*

If you have purchased the *Warp3* product or configured the *Warp2* product to run with Viewlogic tools, run the following command:

```
powerview
```

If you have purchased the *Warp2* product, run the following command:

```
galaxy
```

To Read the On-line Documentation

To read the on-line *User's Guide*, run the following command:

```
acroread <Warp Installation directory>/doc/usguide.pdf
```

To read the on-line *Reference Manual*, run the following command:

```
acroread <Warp Installation directory>/doc/refman.pdf
```

On-line documentation files can also be opened from the Acrobat Reader's file selection dialog box. The files are in *<Warp Installation directory>/doc>*.

On-line documentation can also be found on your CD-ROM in the */docs* directory.

For more information on the Acrobat Reader, read the *ReadMe-Reader.txt* and *Help-Reader.pdf* files in the *<Acrobat Reader Installation directory>/Help* directory.

Chapter 2

Overview

Welcome to the *Warp* programmable logic design tool. *Warp* offers unparalleled performance and flexibility with a user-friendly GUI. *Warp* leads the programmable logic industry in device resource utilization and speed optimization for programmable logic. Thus, designs can be tuned to have the fastest performance with smallest area utilization. *Warp's* flexibility means flexibility of design entry methods: schematic capture, industry standard VHDL entry, or a combination of both; moreover, *Warp* is independent of device architecture, which means one design format can target all Cypress programmable logic devices. *Warp* is a fully integrated EDA tool, fully IEEE VHDL compliant, and available to run on many computer platforms.

Warp is available for all of the popular computer platforms available:

Table 2-1

Computer	Operating System
PC	Windows 3.1 Windows 95 (*Warp2* only) Windows NT (*Warp2* only)
Sun	Solaris SunOS
HP	HP-UX

Table 2-2 lists the parts for which *Warp* can target designs. These parts can be divided into three families:

1. PLDs: these are industry standard devices such as 22V10s, 16V8s, and 20v8s. They are both high performance and low cost.
2. CPLDs: the Cypress FLASH370™ high-density CPLD family and the Cypress MAX340™ multiple-array matrix, high-density CPLD family.
3. FPGAs: the Cypress pASIC380™ high-speed CMOS FPGA family.

Table 2-2

Family	Part #
PLDs	PALC16V8, 20V8, 22V10, CY7C335
CPLDs	FLASH370, MAX340
FPGAs	pASIC380 family

Streamlined Design Process

Warp's design process is intuitive and straightforward. The fully integrated *Warp* comes with all the necessary tools for design entry, design synthesis, fitting, place and route, and design verification. *Warp3* integrates the Cypress VHDL synthesis, fitting, and place and route technology with the Viewlogic Powerview or Workview PLUS environment. The Viewlogic Cockpit™ facilitates project management and intertool communication.

Project Management

The Viewlogic Cockpit provides an easy point-and-click interface for starting the design tools and design project management within the *Warp3* design flow. Tools run from within the Cockpit pass the current design name and project directory from one tool to the next.

Design Entry

The ViewDraw® tool within the *Warp3* environment is used to capture designs graphically using schematics. This tool allows the user to choose elements from a Library of Parameterized Modules (LPM) as well as create graphical symbols from lower-level VHDL blocks. The user can thus capture designs in the manner that he or she prefers.

Generating a VHDL Netlist

The *Warp3* tool converts schematic designs into an IEEE compliant VHDL netlist for synthesis. This VHDL netlist combines the schematic portion of the design with the lower-level VHDL blocks while retaining the hierarchy created by the designer.

Synthesis, Fitting, and Place and Route

Galaxy™ is the name of the Graphical User Interface (GUI) for the *Warp* VHDL compiler. *Warp* accepts VHDL as input, checks the design for proper VHDL syntax, and synthesizes the design into logic equations for a target device.

Warp features the UltraGen™ module generation technology. This technology allows the user to write VHDL code using high-level datapath operators in a truly behavioral fashion. The UltraGen technology then maps each of these high-level operators to an architecture specific module pre-optimized for area or speed, depending on the user's directive. Each portion of VHDL code, via a control file, can accept a different synthesis goal which allows for fine tuning of critical paths within a design.

For PLDs and CPLDs, the logic equations produced by the synthesis are automatically passed to the fitter, which fits the logic into a particular device. The output of the fitter is a JEDEC map which can be used to program the device using the Cypress *Impulse3*™ programmer.

For FPGAs, *Warp* translates the design into a netlist specific to the FPGA architecture. This netlist describes the interconnection of logic elements that can map directly to logic cells of the FPGA device. The place and route tool takes the FPGA netlist generated by the *Warp* compiler, places the logic equations within the logic cells, and routes the necessary signals between logic cells and I/O pins. A static timing analyzer (path analyzer) is available to evaluate worst-case delays and allows for timing-constrained placement and routing as well.

Design Verification

Nova™ is a very convenient tool for PLD or CPLD design simulation. During the initial design stage, Nova can be valuable in obtaining simulation results quickly. For full timing information and verification, Cypress offers the Viewlogic ViewSim® simulator.

ViewSim simulates the design with full timing information. With ViewSim, one can verify the functionality of the design and determine if the design meets the timing requirements of the specific system. ViewSim provides an interface to ViewTrace®, which displays waveforms and allows direct interaction with the simulator.

What's in This Manual?

Besides the Tutorials chapter, this *User's Guide* is meant more as a guide to using *Warp* and less as a reference book, and would be best digested in complete chapters.

Chapter 1 covered all of the necessary information required to get the software installed and up and running.

Chapter 3 is the Tutorials chapter. This section demonstrates the design process via a well-tailored design exercise, which illustrates the *Warp2* and *Warp3* tools respectively.

Chapter 4 contains a description of the Galaxy user interface.

Chapter 5 focuses on the SpDE place and route toolkit.

Chapter 6 discusses the Nova functional simulator used for simulating simple PLD and CPLD devices.

Chapter 7 contains valuable information regarding the use of ViewDraw for schematic and mixed-mode designs.

Chapter 8 includes information regarding simulation interfaces within the *Warp* environment as well as how to interface with third party simulators.

Chapter 9 culminates much of the information presented within this book into a guide to synthesis using *Warp*. This chapter focuses on how to get the most out of your VHDL designs.

Chapter 10 is a brief description of the device programming portion of the flow as it contains information regarding file types and device programmers.

Chapter 3

Tutorials

3.1 Introduction

The *Warp* tutorials demonstrate a common sequence of operations in *Warp2* and *Warp3*. The tutorials show the user how to create, compile, synthesize, and simulate designs.

This section presents:

- product descriptions of *Warp2* and *Warp3*
- some conventions about typography, wording, and illustrations used in this manual
- the objectives of the tutorial
- the contents of the tutorial

3.1.1 Product Descriptions

Warp2

Warp2 allows users to describe electronic designs using VHDL and then to compile and synthesize those descriptions to program Cypress devices, such as small PLDs, MAX340 EPLDs, FLASH370 CPLDs, and pASIC380 FPGAs.

Warp2 consists of three major components:

- The *Warp* VHDL compiler which is IEEE 1076/1164 compliant and translates VHDL text descriptions into JEDEC and QDIF (for pASIC380 FPGAs only) files that can be mapped onto programmable devices.
- The Nova JEDEC functional simulator which allows you to verify the correctness of a design by simulating its behavior.
- The SpDE™ toolkit which contains a set of tools for fitting designs into pASIC380 FPGAs. This tool set includes a placer, router, logic optimizer, and a path analyzer, among others.

Warp3

Warp3 enables users to define, compile and synthesize VHDL descriptions, schematic descriptions, or a combination of VHDL (text-based) descriptions with schematic drawings of electronic designs into Cypress programmable logic devices.

Finished files can be synthesized into (i.e., used to program) Cypress devices, such as small PLDs, MAX340 EPLDs, FLASH370 CPLDs, and pASIC380 FPGAs.

Warp3 integrates all of the components from the *Warp2* software with the Viewlogic Workview PLUS and Powerview CAE environments. Included with *Warp3* are the following:

- The *Warp* VHDL compiler which is IEEE 1076/1164 compliant and translates VHDL text descriptions into JEDEC and QDIF (for pASIC380 FPGAs only) files that can be mapped onto programmable devices.
- The Nova JEDEC functional simulator which allows you to verify the correctness of a design by simulating its behavior.
- The SpDE toolkit which contains a set of tools for fitting designs into pASIC380 FPGAs. This tool set includes a placer, router, logic optimizer, and a path analyzer, among others.
- Viewlogic's Workview PLUS design environment (for IBM PCs and compatibles) and the Powerview design environment (for UNIX workstations). Viewlogic provides the Cockpit, which is the central access point for all tools in the *Warp3* system. The Cockpit contains icons that bring up any tool the user needs. Viewlogic's design environments provide a large set of design tools in and of themselves. Among them are ViewDraw, a hierarchical schematic and symbol editor; ViewSim, a full-featured simulator that accurately models timing delays; ViewTrace, a waveform viewing tool that helps you analyze simulator results; and numerous other tools for special needs.

3.1.2 Conventions

The following conventions are used throughout this manual:

Table 3-1 Notational Conventions

Menu items	Whenever an item from a menu is referenced, the reference takes the form *menu-name->item-name->item-name...* The first entry in the reference is the name of the menu; the second is the name of the menu item; the third and succeeding entries indicate choices from sub-menus. Example: Select *File->Open...* tells you to pull down the *File* menu and select the *Open...* item from it.
Path names	Italics are also used to designate path names and file names.
Bold	Words are **bolded** for emphasis or to draw attention to new terms.
`Courier`	Denotes signal names and other VHDL constructs as well as system output.
`Courier`	Denotes the contents of a text file and also indicates the text of typed commands.

File Naming Conventions

Warp runs on many different platforms: IBM PCs and compatible computers, Sun workstations, and HP workstations.

IBM PCs and compatible computers specify file locations by designating a disk drive and using a backslash (\) character to distinguish directory levels, e.g., *c:\level1\level2\myfile.*

Sun and HP workstations specify file locations using a forward slash (/) and no disk drive designator, e.g., */level1/level2/myfile.*

For consistency and brevity, this manual uses the same notation as IBM PCs and compatibles when referring to file locations. UNIX platform users are asked to make appropriate translations.

Mouse Conventions

The following terms describe common actions you might perform in using this manual:

Table 3-2

Click	Place the mouse cursor over an object, then press and release the appropriate mouse button.
Double-click	Position the mouse cursor over an object, then press and release the appropriate mouse button twice in rapid succession.
Drag	Position the mouse cursor over an object or at a specified location. Press and hold the appropriate mouse button. While holding down the mouse button, move the cursor to the new location. Finally, release the mouse button.
Select	When you are instructed to "select" an option, move the cursor over the option, then click the appropriate mouse button.

Other Conventions

The following visual indicators denote special situations you may encounter while using this manual.

Note – This icon indicates a *note*: a point in the tutorial where you must exercise special caution, or where the procedure might vary depending upon the platform you're using, or where there's something else you should know about that doesn't fit into the main flow of the text.

Hint – This icon indicates a *hint*: a point in the tutorial where you could save a little time, a few keystrokes, or much frustration if you follow the hint that's being explained.

3.1.3 Differences between Operating Systems

Warp operates identically on both UNIX and Windows systems except for start-up procedure and the appearance of some objects in the display.

Naming Restrictions

In the Windows environment, file names **must** be eight or fewer alpha-numeric characters plus a file extension of up to three characters (e.g., *.exe*, *.vhd*, etc.). This is important to remember for transferring data between Windows and UNIX/ Windows 95 implementations of *Warp.*

Note – Keep in mind that some UNIX systems have trouble with spaces embedded in file names. That can be true for IBM PCs and compatible systems, too. Cypress recommends not using embedded spaces in file names.

Note – If you are working in a system with PCs (running under Windows) and UNIX workstations connected to the same network, Cypress advises you to name all VHDL files with lower case characters.

Differences in Display

Although dialog boxes and prompts may differ in appearance between the two platforms, their functionality is identical. Screen captures in this manual are taken from the Windows version of *Warp*. Differences in the UNIX version are identified when necessary. In most cases, any adjustments needed to go from Windows to UNIX versions of *Warp* displays are obvious.

For Warp3 Users Only

There are minor differences between the Windows and UNIX versions of *Warp3* in the appearance of the Cockpit, the dialog boxes, and the prompt boxes. Besides these minor differences, the operation of the *Warp3* software is virtually identical among all platforms.

3.2 Designing a Soda Machine

This section takes you step-by-step through a tutorial example using a low-level VHDL behavioral description and a high-level VHDL file that instantiates the component contained in the lower level VHDL file. When you complete this section, you will know how to:

- write an entity declaration, architecture, component and package declaration for a VHDL behavioral description of a simple circuit
- write an entity declaration and architecture that instantiates the simple circuit in a VHDL description of a higher-level circuit
- run *Warp* to compile and synthesize the design's VHDL description
- run Nova to simulate the behavior of the design
- **for *Warp3* users only**: simulate the design with ViewSim

3.2.1 Design Description

In this example, you will design a controller for a soft-drink dispensing machine. The machine has two bins to dispense regular cola and diet cola. Each bin holds three cans of soft drink. (This could be any value, but three is an easy number to simulate.)

The circuit should dispense a beverage if the user presses a button for that beverage and one or more cans of the beverage are available. The circuit should not dispense a beverage if no cans of that beverage are available. A `refill` signal should appear when both bins are empty. Finally, pressing a `reset` signal should tell the circuit that the machine has been replenished and that the bins are full again.

3.2.2 Design Solution

The solution presented in this section will proceed as follows:

First, this tutorial will describe a circuit in IEEE standard VHDL that controls the operation of one bin. It will respond appropriately to a `get_drink` signal (i.e., by giving a drink when one is available), keep count of the number of cans left in the bin, and set an `empty` signal when its bin becomes empty and is in need of resetting. This circuit will be named *binctr*. You will also add appropriate VHDL to declare this circuit a component and define a package that contains that component, which will allow you to use the circuit in higher-level designs.

Then, you will write a top-level description of a circuit that instantiates two *binctrs* and other logic as appropriate to describe the larger design described above. The larger design will be called *refill*.

After that, you'll compile and synthesize the *binctr* and *refill* VHDL descriptions into a CY7C371 JEDEC file and simulate the behavior of the resulting design using the Nova functional simulator.

3.3 Starting Warp

For *Warp2* Users

=> On Windows systems, start *Warp2* by double-clicking on the *Galaxy* icon from the *Warp R4* program group.

=> On UNIX systems, start *Warp2* by typing **`galaxy<CR>`** from within a shell window.

In both systems, the Galaxy window should appear as shown in Figure 3-2.

For *Warp3* Users

=> On Windows systems, you start *Warp3* by double-clicking on the *Cockpit* icon from the *Warp R4* program group.

=> On UNIX systems, you start *Warp3* by typing **`powerview<CR>`** from within a shell window.

On both platforms, the Cockpit should be displayed immediately after you start *Warp3*. The Cockpit is labeled "Workview PLUS Cockpit" on Windows systems, "Powerview Cockpit" on UNIX systems, and should appear as shown in Figure 3-3.

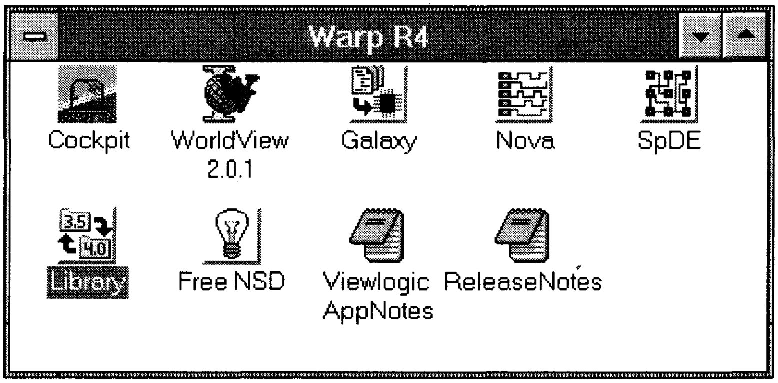


Figure 3-1 The Warp R4 Program Group

3.4 Creating A New Project

For *Warp2* Users

=> Under the *Project* menu, choose *New.* A new Galaxy window will appear, prompting you for a new name. On the PC, enter **`c:\w2tutor\w2tutor`** and on UNIX systems, enter **`<user home directory path>/w2tutor/w2tutor`**.

You should now see a Galaxy window identical to the one that appeared when you started *Warp*, except that it is named *c:\w2tutor\w2tutor*.

You have now created a project directory for your designs and called it *w2tutor*. All files pertinent to this tutorial need to be placed in this project directory. You have also created a project file called *w2tutor.wpr* which is now located in the *c:\w2tutor* directory.

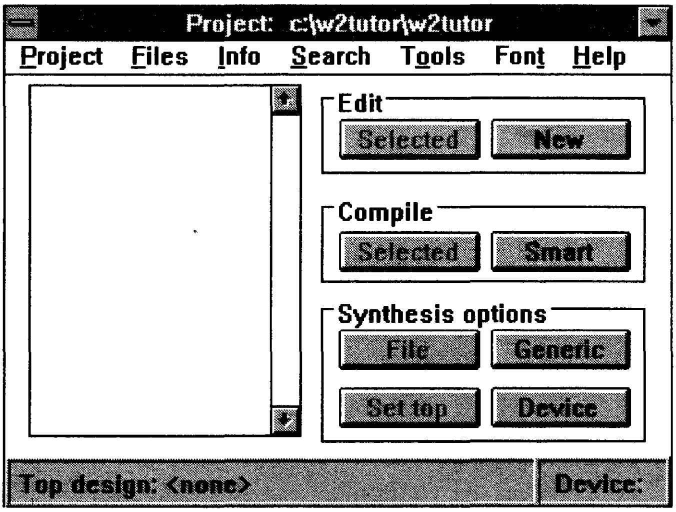
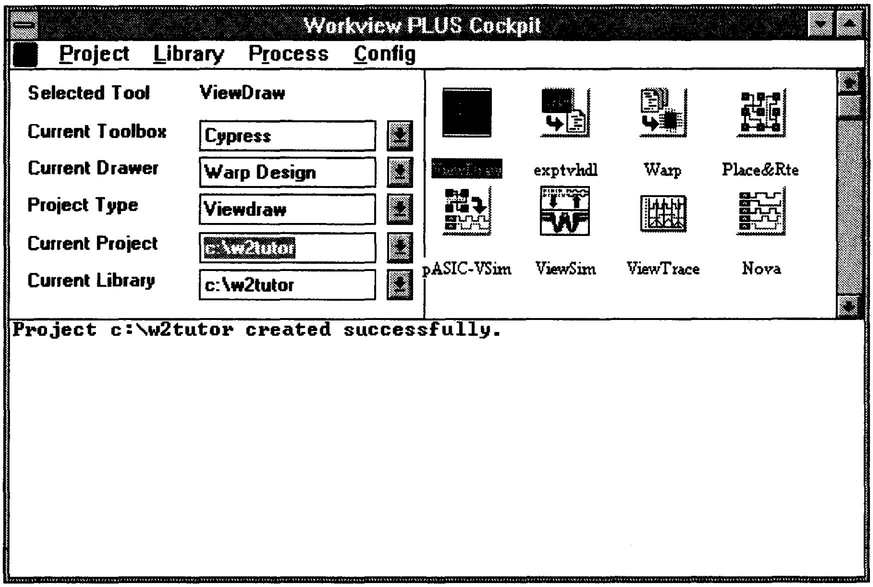


Figure 3-2 Galaxy window named c:\w2tutor\w2tutor

For *Warp3* Users

=> Select *Project->Create* from the Cockpit menu bar. When a dialog box appears, on the PC type the pathname **`c:\w2tutor`** and on UNIX systems, type **`<user home directory path>/w2tutor`** and then click *OK*



Selected Tool: ViewDraw
Current Toolbox: Cypress
Current Drawer: Warp Design
Project Type: Viewdraw
Current Project: c:\w2tutor
Current Library: c:\w2tutor

```
Project c:\w2tutor created successfully.
```

Figure 3-3 Warp3 cockpit

When doing the tutorial exercises (or any other time, for that matter), **don't write anything into the *Warp* directory**. Instead, create a separate directory to practice in, such as the *c:\w2tutor* directory you have just created above.

By default, the *Warp3* installation procedure for IBM PCs and compatible computers installs *Warp3* software into a directory named *c:\warp*. On UNIX workstations, the user needs to point the CYPRESS_DIR environment variable to point to the location of the *Warp3* software.

3.5 Creating the VHDL File

For *Warp2* Users

=> In your Galaxy window, click on *New* in the *Edit* button section on the right.

This will bring up a VHDL text editor (Figure 3-4), in which you can enter your VHDL code.

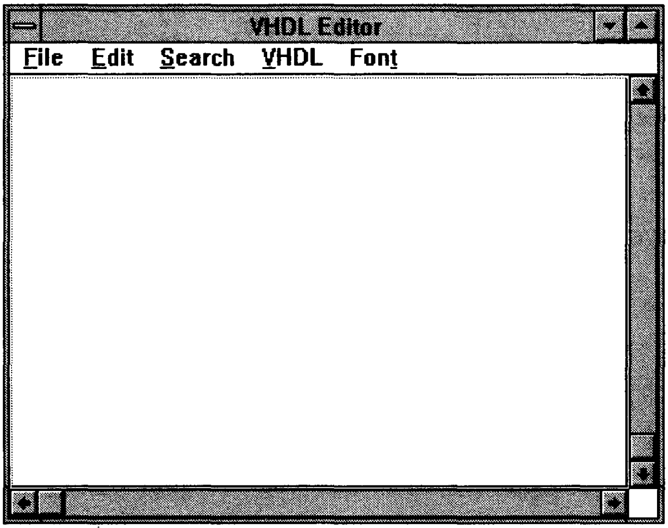


Figure 3-4 VHDL Editor

For *Warp3* Users

=> From the Cockpit window, click on the *Warp* icon. After the Galaxy window appears, click on *New* in the *Edit* button section.

This will bring up the VHDL editor in which you can enter your VHDL code.

The *binctr* VHDL description will be written in three parts:

- the **entity** declaration declares the name, direction, and data type of each port of the component
- the **architecture** describes the behavior of the component
- the **package** declaration provides the information to *Warp* to allow *binctr* to be used as a component in a higher-level design

The following pages briefly discuss the contents of each of these sections of the VHDL description. For a detailed explanation on these VHDL constructs, please refer to the text book which was included in your software kit entitled *VHDL for Programmable Logic.*

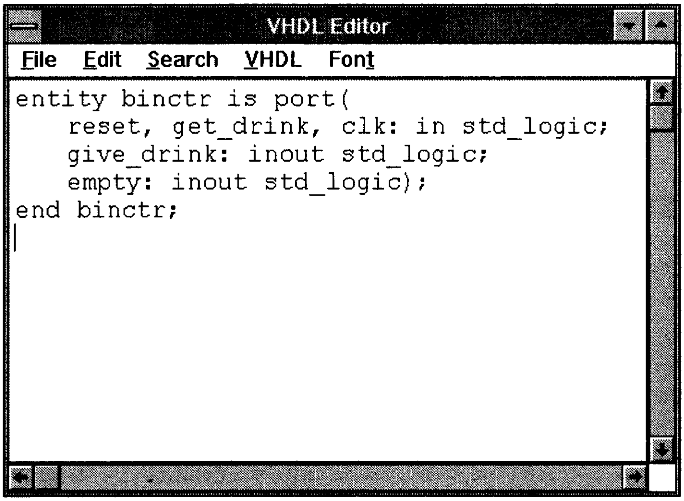
3.5.1 Writing the Entity Declaration

=> Copy the following lines into your VHDL text editor:

```
entity binctr is port(
   reset, get_drink, clk: in std_logic;
   give_drink: inout std_logic;
   empty: inout std_logic);
end binctr;
```

The entity declaration declares the name, direction, and data type of each port of the component. The entity declaration declares that entity `binctr` has five external interfaces, or ports. It has three input ports of type `std_logic`, named `reset`, `get_drink`, and `clk`, respectively. It has two input/output (inout) ports, also of type `std_logic`, named `give_drink` and `empty`, respectively.

The signals give_drink and empty are of mode inout and represent the various states that the state machine can assume. The architecture definition for the entity binctr requires that signals give_drink and empty retain their values when all other state transition conditions are untrue. This forces give_drink and empty to feed back on themselves apart from being outputs of the state machine. Hence the signals give_drink and empty are declared with mode inout.



```
entity binctr is port(
   reset, get_drink, clk: in std_logic;
   give_drink: inout std_logic;
   empty: inout std_logic);
end binctr;
```

Figure 3-5 The VHDL editor with the entity description

3.5.2 Writing the Architecture

=> Copy the following lines into your VHDL text editor ***below*** your entity declaration:

```
architecture archbinctr of binctr is
  constant full: std_logic_vector(1 downto 0):= “11”;
   -- max of 3 drinks/bin
  signal remaining: std_logic_vector(1 downto 0);
begin
  proc_label: process (clk,reset)
   begin
   if (reset = ‘1’) then
      remaining <= full;
      empty <= ‘0’;
      give_drink <= ‘0’;
   elsif (clk’event and clk = ‘1’) then
      if (remaining = “00”) then
         empty <= ‘1’;
         give_drink <= ‘0’;
      elsif (get_drink = ‘1’) then
         remaining <= remaining - 1;
         give_drink <= ‘1’;
      elsif (get_drink = ‘0’) then
         give_drink <= ‘0’;
      else
         give_drink <= give_drink;
         remaining <= remaining;
         empty <= empty;
      end if;
   end if;
  end process;
end archbinctr;
```

The architecture portion of a VHDL description describes the behavior of the component and always appears ***after*** the entity description.

The first line declares an architecture named `archbinctr` of entity `binctr`.

The next two lines declare a constant and a signal, respectively.

- The constant, named `full`, determines how many drinks are in a full bin.
- Signal `remaining`, of type `std_logic_vector` keeps track of how many drinks are left in the bin.

The begin that follows the signal declaration marks the start of the architecture body. All constant and signal declarations in the architecture of the VHDL code should precede the begin statement.

A process declaration follows, marked by the keyword process and an ensuing begin. The process declaration ends with the end process; statement.

The last line end archbinctr; denotes the end of the architecture declaration.

You can add comment lines to your VHDL code by placing "--" before the text on a line that you want commented. All or part of a line can be commented out as shown on the line that defines the constant full.

The architecture defines a process which is activated by any changes to the clk or the reset signal. The process definition shown here is a VHDL template for describing a synchronous circuit with an asynchronous reset.

Signal activity is handled in the following sequential order:

- The process within the architecture is triggered only if there is a change in the logic levels of the signals clk and reset. These signals are therefore included in the process sensitivity list. The signal order in the sensitivity list is arbitrary.
- If signal reset is '1', then signal remaining is set to full, signal empty is set to '0', and signal give_drink is set to '0'. (Notice that this means reset has priority over all other signal declarations in the architecture definition. This forces the compiler to define the reset as an asynchronous reset for the design.
- The reset signal needs to be false to initiate the execution of the rest of the statements in the process declaration.
- The clock definition statement indicates to the compiler that all registered equations in the architecture definition are to be synthesized into rising edge triggered flip-flops. The process can be changed to be falling edge sensitive by altering the clock definition statement to look like this: (if clk'event and clk = '0').
- If signal remaining has a value of "00", then signal empty is set to '1' and signal give_drink is set to '0'.

- If signal `remaining` is not a '0' and signal `get_drink` is '1', then signal `remaining` is decremented by one and signal `give_drink` is set to '1'.
- If signal `remaining` is not "00" and signal `get_drink` is '0', then signal `give_drink` is set to '0'.
- If the `if-then-elsif` conditions are false, `give_drink`, `remaining` and `empty` retain the current value of `give_drink`, `remaining` and `empty` respectively.

Several lines ending the if statement, process, and architecture follow. Note that the `end` statement of the architecture must be accompanied by the name of the architecture, which **must** match the name shown on the first line of the architecture.

3.5.3 Writing the Package Declaration

=> Copy the following lines into your VHDL text editor ***before*** the entity declaration:

```
package binctr_pkg is
  component binctr
   port(reset, get_drink, clk: in std_logic;
       give_drink: inout std_logic;
       empty: inout std_logic);
   end component;
end binctr_pkg;
```

The package declaration provides the information to the *Warp* compiler to allow *binctr* to be used as a component in a higher-level design.

Note – The package declaration **must** appear before the entity declaration and architecture in the *.vhd* file.

The first line in the above package declaration names the package. The name of the package must be distinct from the name of any component declared within that package. Using the convention <*entity*>`_pkg` works nicely.

The second line declares a component named `binctr`. The component name that appears on this line must match the name of an accompanying entity.

The `port` statement declares the name, direction, and type of each port in the component. You can copy the port statement from the entity declaration for this purpose.

An `end component` and `end binctr_pkg` statement conclude the package declaration. Note that the package named in the `end package` statement must match that shown in the first line of the package declaration.

At this point, save the file as *c:\w2tutor\binctr.vhd.*

3.5.4 Including the Libraries

=> Insert the following three lines of VHDL code ***before*** the package declaration ***and*** also ***before*** the entity declaration:

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
```

These two lines advise the compiler to link to the IEEE VHDL library, and the Cypress UltraGen module generation library. The compiler finds definitions for the various types, modes, and VHDL constructs defined in the user's VHDL code. In general, the **`library`** and **`use`** VHDL reserved words instruct the compiler to include pre-defined libraries and user created VHDL files, in compiling the selected VHDL code. For more information on the `std_logic_1164` and `std_arith` packages, see Chapter 4, "VHDL," in the *Reference Manual.*

3.6 Verifying the VHDL Syntax

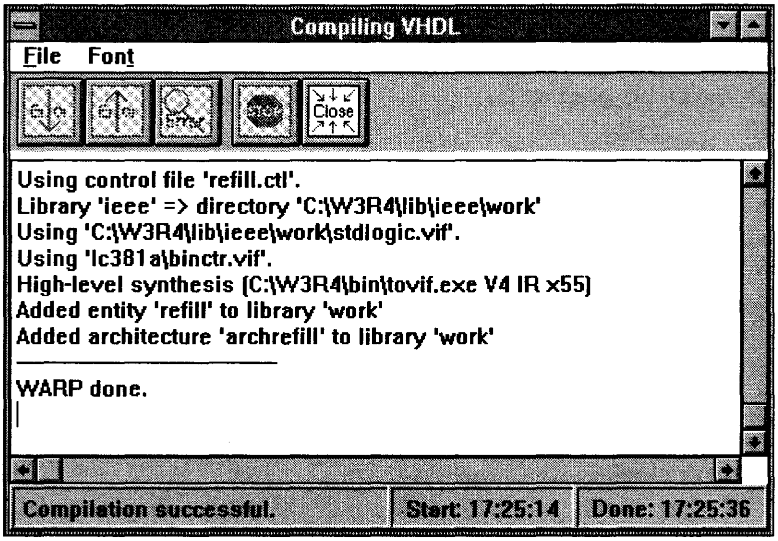
=> First, save your VHDL file by clicking on *Save* under the *File* menu.

=> Then under the *VHDL* menu, select *Compile.*

Warp runs, printing messages to keep you appraised of its progress. The verification process should run to completion without any error messages. A snapshot of this window is shown in Figure 3-6.

=> Close your VHDL file by clicking on the close box, or selecting *Close* under the *File* menu.

You have just used *Warp* to verify that the *binctr.vhd* file is syntactically correct. This step isn't strictly necessary, but it's always a good idea to compile any VHDL description once it's completed. That way, you can spot problems in your VHDL description when they are easiest to identify and correct. Later, should you encounter problems with the larger circuit, you can at least be assured that you have taken care of any errors in the lower levels of the hierarchy.



```
Using control file 'refill.ctl'.
Library 'ieee' => directory 'C:\W3R4\lib\ieee\work'
Using 'C:\W3R4\lib\ieee\work\stdlogic.vif'.
Using 'lc381a\binctr.vif'.
High-level synthesis (C:\W3R4\bin\tovif.exe V4 IR x55)
Added entity 'refill' to library 'work'
Added architecture 'archrefill' to library 'work'
_______________________
WARP done.
```

Compilation successful. | Start: 17:25:14 | Done: 17:25:36

Figure 3-6 A successful *Warp* compilation

Note – If you do get error messages, check to make sure that the various parts of the *binctr.vhd* file read **exactly** as they are listed on the preceding pages. Better yet, copy the *binctr.vhd* file from the *c:\warp\examples\wtutor* directory of the *Warp* installation, then try compiling the design again.

3.7 Creating a Top-Level Description

Now that you have defined the behavior of the lower level of the circuitry, you can describe the upper level by instantiating two components and connecting them with appropriate internal signals.

=> Open your VHDL editor by clicking on *New* in the *Edit* section of the Galaxy window. Copy the following lines into the ***new*** VHDL file.

```
library ieee;
use ieee.std_logic_1164.all;
use work.binctr_pkg.all;

entity REFILL is port (
      GIVE_cola: INOUT std_logic;
      GIVE_diet: INOUT std_logic;
      REFILL_BINS: OUT std_logic;
      RESET: IN std_logic;
      CLK: IN std_logic;
      GET_diet: IN std_logic;
      GET_cola: IN std_logic);

attribute pin_numbers of refill:entity is
" GIVE_cola:2 GIVE_diet:3 REFILL_BINS:4 RESET:10 CLK:13" &
" GET_diet:11 GET_cola:35" ;

end REFILL;
```

The first three lines advise the compiler to link to the IEEE VHDL library and the user created `binctr_pkg` VHDL package. The compiler finds definitions for the various types, modes, and VHDL constructs defined in the user's VHDL code. In general, the **library** and **use** VHDL reserved words instruct the compiler to include pre-defined libraries and user created VHDL files, in compiling the selected VHDL code. For more information on the `std_logic_1164` and `std_arith` packages, see Chapter 4, "VHDL," in the *Reference Manual.*

The entity declaration declares an entity named `refill` and defines the names, types, and mode (direction) of its seven input and output ports.

There is also a `pin_numbers` assignment attribute which assigns all the signals declared within the entity port map declaration to user specified pin numbers. Refer to Chapter 3, "Synthesis Directives," in the *Reference Manual* for more information on the `pin_numbers` attribute.

Note – The pin numbers assigned to the signals in this design are specific to the CY7C371 in the PLCC package. If a different package or device is targeted, the numbers would have to be changed appropriately. Alternatively, the `pin_numbers` attribute and the pin_number assignments can be removed.

=> Copy the following lines into your VHDL editor ***below*** the entity declaration.

```
architecture archREFILL of REFILL is
      signal empty_1: std_logic;
      signal empty_2: std_logic;
begin

bin_1: BINCTR
        port map(RESET => RESET,
                 GET_DRINK => GET_cola,
                 CLK => CLK,
                 GIVE_DRINK => GIVE_cola,
                 EMPTY => empty_1);
bin_2: BINCTR
        port map(RESET => RESET,
                 GET_DRINK => GET_diet ,
                 CLK => CLK,
                 GIVE_DRINK => GIVE_diet ,
                 EMPTY => empty_2);

refill_bins <= '1' when ((empty_1 = '1') and (empty_2 = '1'))
               else '0';

end archREFILL;
```

=> Now, save your top-level VHDL file by choosing *Save As* in the *File* menu. Enter **`refill.vhd`** as the name, then close the file.

The architecture starts by declaring two internal signals, `empty_1` and `empty_2`. These are used to connect the outputs of the two `binctr` components to create the functionality for `refill_bins`.

The two instantiations of `binctr` and the creation of the `refill_bins` equation complete the VHDL description of this circuit. With this implementation, you have translated the original vending machine problem into a VHDL design that can be synthesized into any Cypress programmable logic device. The block diagram for *refill.vhd* is shown here:

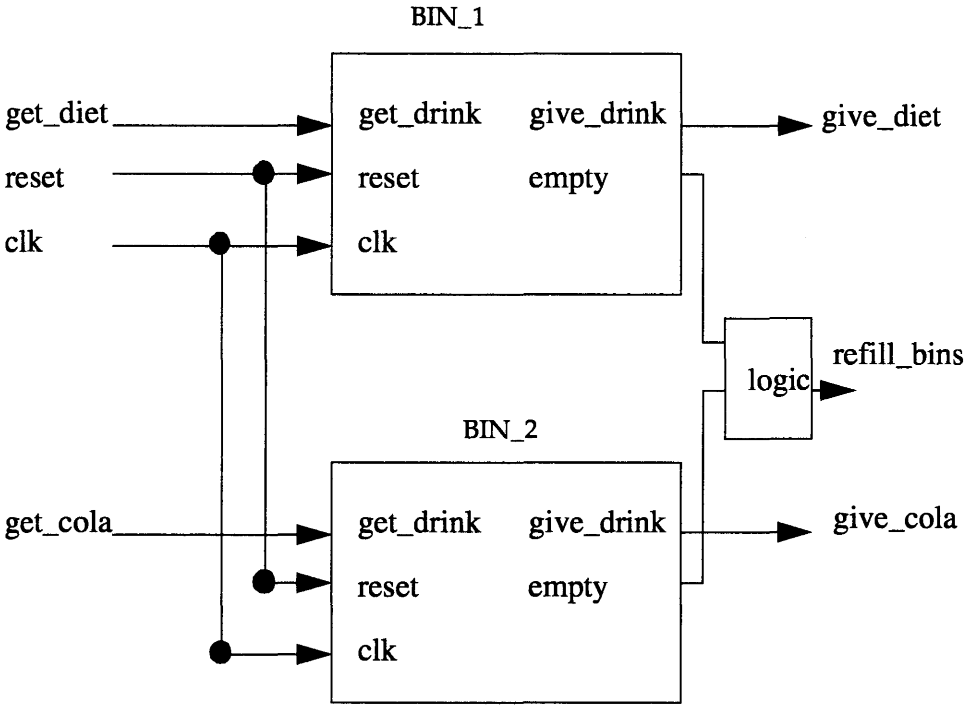
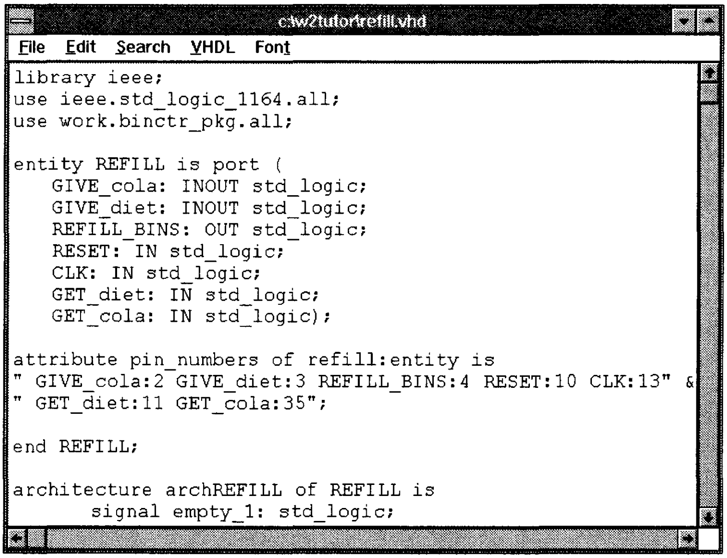


Figure 3-7 Block Diagram for refill.vhd

The final VHDL window for *refill.vhd* should look as follows:



```
library ieee;
use ieee.std_logic_1164.all;
use work.binctr_pkg.all;

entity REFILL is port (
   GIVE_cola: INOUT std_logic;
   GIVE_diet: INOUT std_logic;
   REFILL_BINS: OUT std_logic;
   RESET: IN std_logic;
   CLK: IN std_logic;
   GET_diet: IN std_logic;
   GET_cola: IN std_logic);

attribute pin_numbers of refill:entity is
" GIVE_cola:2 GIVE_diet:3 REFILL_BINS:4 RESET:10 CLK:13" &
" GET_diet:11 GET_cola:35";

end REFILL;

architecture archREFILL of REFILL is
      signal empty_1: std_logic;
```

Figure 3-8 Final VHDL window for refill.vhd

3.8 Compiling and Synthesizing a Top-Level File

The first time you ran *Warp*, earlier in the tutorial, it was simply to verify that the *binctr.vhd* file was syntactically correct.

In the following pages, you'll run *Warp* to produce a JEDEC file for a specific target device (in this case, a CY7C371 - 32 macrocell CPLD).

3.8.1 Selecting Files for Compilation

First, you need to add the files you will compile to your project.

=> In your Galaxy window labeled *w2tutor*, click on the *Files* menu and choose the *Add all* option.

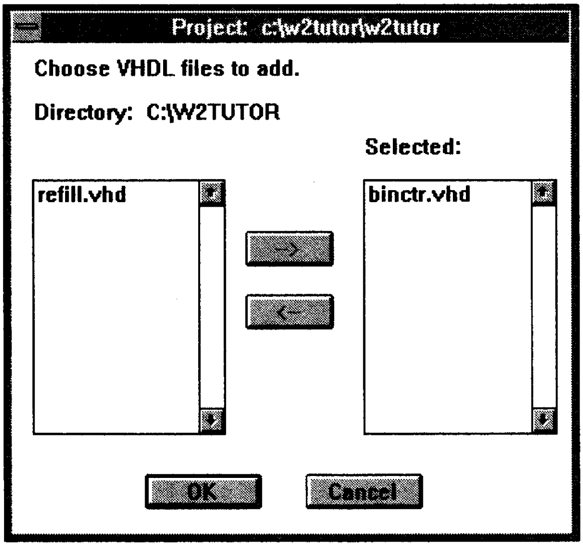
Since the only files currently in this project directory are *binctr.vhd* and *refill.vhd*, the two files needed for this tutorial, the *Add all* option is a quick way to add them to the compilation list.

If there are other VHDL files in the same directory, you need to do something slightly different:

=> Click on the *Files* menu and choose the *Add* option.

The ensuing dialog box lists the VHDL files available in the current directory on the left side, and the VHDL files selected for compilation or synthesis on the right side (Figure 3-8).

=> Highlight the VHDL file in the left-hand list, and double-click on it or choose and click on the "-->" button. To deselect a file for compilation or synthesis, highlight the VHDL file in the right-hand list and double-click on it or choose and click on the "<--" button. Click on *binctr.vhd* and *refill.vhd* until they are the only files in the right-hand list, then click *OK*.



Project: c:\w2tutor\w2tutor

Choose VHDL files to add.

Directory: C:\W2TUTOR

Selected:

refill.vhd

binctr.vhd

Figure 3-9 Dialog box to add files to your Galaxy window

Back in the main Galaxy window, both *binctr.vhd* and *refill.vhd* should be listed.

3.8.2 Selecting a Device

=> Click on the *Device* button from within the *Synthesis options* section.

The Device window should appear (Figure 3-9).

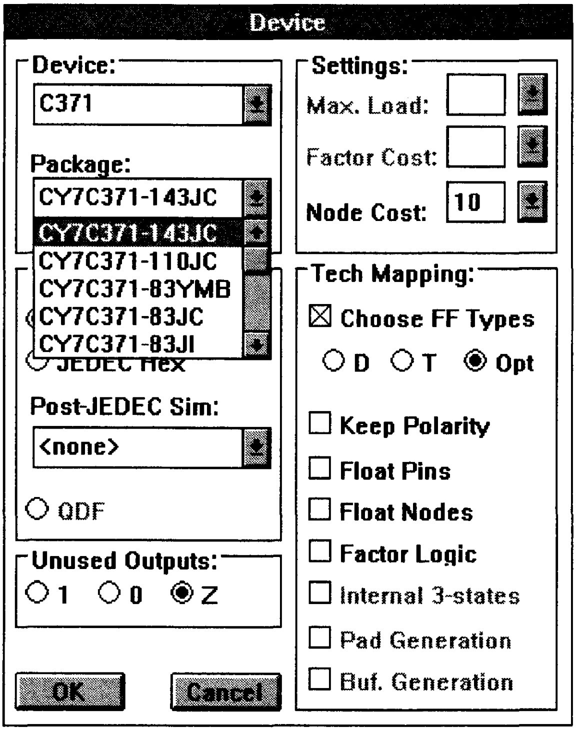
=> Click on the down-pointing arrow next to the *Device* label to activate the pull-down menu. Scroll down the menu and select the C371 by clicking on it.

C371 should appear as the targeted device in the lower right-hand corner of the main Galaxy window.

3.8.3 Selecting a Package and Speed Bin

In addition to selecting the device you wish to target, you can also choose a package type and a speed bin from the list of available packages (Figure 3-10).

Click on the arrow next to the *Package* label to activate the scroll down menu. Scroll down the menu until the your desired package is visible. Select the desired package by clicking on it. For this exercise you should choose the default package and speed bin CY7C371-143JC.



Device

Device: C371

Package: CY7C371-143JC

CY7C371-143JC
CY7C371-110JC
CY7C371-83YMB
CY7C371-83JC
CY7C371-83JI

○ JEDEC Hex

Post-JEDEC Sim: <none>

○ QDF

Unused Outputs: ○ 1 ○ 0 ● Z

Settings:
Max. Load:
Factor Cost:
Node Cost: 10

Tech Mapping:
☒ Choose FF Types
○ D ○ T ● Opt
□ Keep Polarity
□ Float Pins
□ Float Nodes
□ Factor Logic
□ Internal 3-states
□ Pad Generation
□ Buf. Generation

Figure 3-10 Available Package and Speed Bin list

3.8.4 Resolving Unused Outputs

Warp gives you the option of turning all of your unused output pins that have unused macrocells into '1's, '0's, or 'Z's. This is a global option and cannot be applied on a signal-by-signal basis. This option is useful for driving all unused pins to a certain logic level.

=> Under the *Unused Outputs* options of the Device window, choose Z, the default option of leaving all unused I/O pins to be three-stated.

Note – When using MAX340 EPLDs and FLASH370 CPLDs, it is a recommended practice to use external pull-ups for unused I/O pins.

3.8.5 Choosing Tech Mapping Options

While compiling registered equations, the fitter will use the directive from *Choose FF types* to synthesize equations. Leaving the option *Opt* selected is the best choice. This enables the fitter to make the choice between a D-FF and a T-FF implementation and then choose the implementation that uses the least number of product terms.

=> Under *Tech Mapping* options in the Device window, choose the *Opt* option.

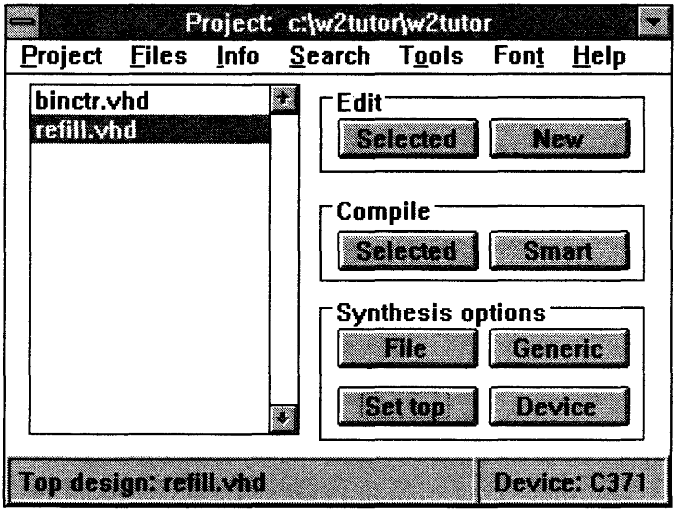
There are a few other useful options in the *Tech Mapping* options and the *Settings* options. Refer to Chapter 4, "Galaxy," of this *User's Guide* for a detailed description of each option.

=> Click on the *OK* button to dismiss the Device window.

3.8.6 Setting the Top-Level File

=> Highlight *refill.vhd* by clicking on it.

=> Click on the *Set Top* button in the *Synthesis options* section to assign your top-level VHDL file.



Top design: refill.vhd

Device: C371

Figure 3-11 Galaxy window showing refill.vhd as the top-level design

Refill.vhd should be listed as the top file in the lower left-hand corner of your Galaxy window.

3.8.7 Compiling and Synthesizing the File

=> In your Galaxy window, click on the *Smart* button in the *Compile* button group to begin compilation.

Warp starts the compilation and synthesis of the design into a CY7C371 and prints messages to keep you appraised of its progress in a pop-up window. The *Smart* compile option automatically recompiles only those files which have been modified since the last compilation.

This operation generates two files of particular interest:

- The first is named *refill.jed.* The *.jed* file can be used to program a CY7C371 device. It is also used as the input to the Nova functional simulator.
- The second file is named *refill.rpt*. It contains pinout and timing information, along with other information about the final synthesized design. You can see the *.rpt* file by pulling down the *Info* menu from the Galaxy window and clicking on *Report*. For more information on what is contained in a report file, refer to Chapter 6, "Report File," in the *Reference Manual.*

=> Close the Galaxy compilation window by clicking on the *Close* button located at the top of the Galaxy compilation window.

Note – If compilation errors occur, do the following: make sure the text of your *binctr.vhd* file is entered **exactly** as shown earlier in this chapter -- or better yet, copy it from the *c:\warp\examples\wtutor* directory -- and then run *Warp* again.

If error messages appear in the pop-up window:

=> Highlight the error message by clicking on it.

=> Click on the *Error* button (the icon that looks like a magnifying glass) located at the top of the Galaxy compilation window.

This opens the VHDL editor with the cursor on the line number that contains the error. Check the code you have entered against the text shown earlier in this chapter.

3.9 Simulating the Behavior of the Design with Nova

Once the design is synthesized, you must simulate it to verify that it functions as intended. You'll use the Nova simulator to test the behavior of the design. Nova is a simple JEDEC functional simulator. In this tutorial, you'll perform the following steps:

- start Nova
- open the *refill.jed* file
- create a new view and populate it with the signals you're interested in (and only those signals). Typically you would retain and modify the default view
- designate and edit a clock signal
- set the values of the stimulus signals in the simulation
- simulate the design
- examine results to figure out what happened

3.9.1 Starting Nova

For *Warp2* Users

=> On Windows systems, start Nova by double-clicking on the *Nova* icon within the *Warp R4* program group.

=> On UNIX systems, start Nova by typing **`nova<CR>`** at a shell prompt.

In both systems, the Nova window should appear.

For *Warp3* Users

=> On Windows systems, you start *Warp3* by double-clicking on the *Cockpit* icon from the *Warp R4* program group.

=> On UNIX systems, launch the Cockpit by typing **`powerview<CR>`** from within a shell window.

On both platforms, the Cockpit should be displayed immediately after you start *Warp3*. The Cockpit is labeled "Workview PLUS Cockpit" on Windows systems, and "Powerview Cockpit" on UNIX systems.

=> Double-click on the *Nova* icon in the Cockpit.

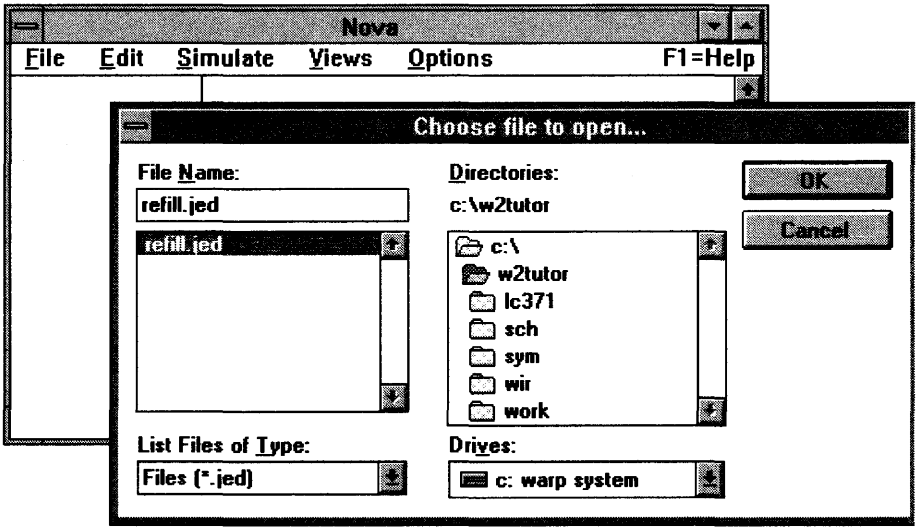
The Nova screen appears, followed by the Nova About box. The About box goes away by itself in a few seconds. If you want to make it go away faster, click anywhere in the About box.

Note – In this section, *Warp3* users will simulate their design with Nova for this exercise. A section simulating the same design with ViewSim is presented later. *Warp3* users are recommended to simulate their designs with the Viewlogic ViewSim simulator. ViewSim is a powerful simulator and has more features available than Nova.

For Both *Warp2* and *Warp3* Users

=> Open the *refill.jed* file by choosing *File->Open.*

The available JEDEC files (*.jed* extension) appear on the left side of the dialog box (Figure 3-12).



File Name: refill.jed

refill.jed

Directories: c:\w2tutor

c:\
w2tutor
lc371
sch
sym
wir
work

List Files of Type: Files (*.jed)

Drives: c: warp system

Figure 3-12 Choosing a file to open in Nova

=> Highlight *refill.jed* by clicking on it.

=> Click on the *OK* button to open *refill.jed.*

The Nova window should resemble the following figure:

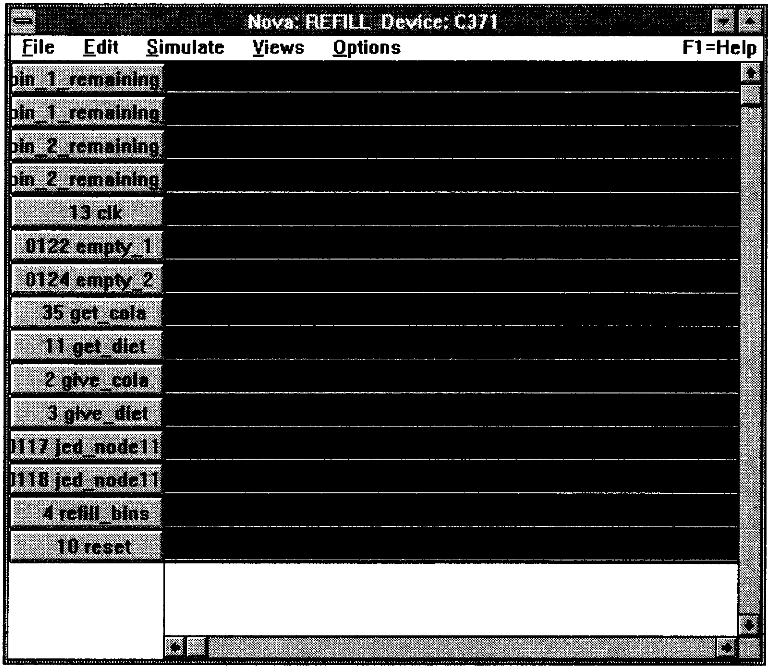


Figure 3-13 Initial Nova window for refill.jed

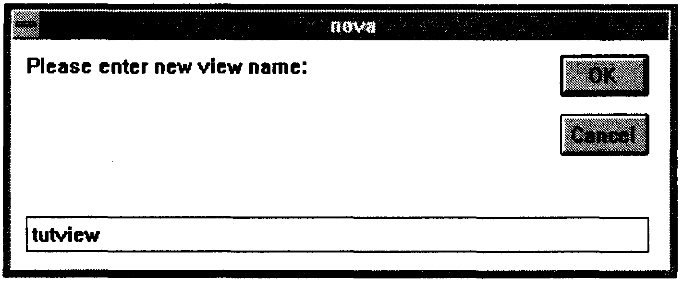
3.9.2 Creating a View

A view is the collection of signals available for viewing on the Nova screen. To make it easier to see what's going on in your simulation, you'll create your own view, selecting only the signals you desire.

=> Choose the *Edit Views* item from the *Views* menu.

=> Click on the *New View* button.

=> In the ensuing dialog window, give the new view any unique name: **tutview** will do nicely. Click *OK* to close the name dialog window.



Please enter new view name:

tutview

Figure 3-14 Nova view naming dialog box

=> Click on the signal name in the *Full View* portion of the window, then on the *Add*>> button, for each of the following signals:

```
clk
reset
get_cola
get_diet
give_cola
give_diet
refill_bin
```

=> Click on *OK*.

The new view should appear on the Nova display screen, and should look like Figure 3-15:

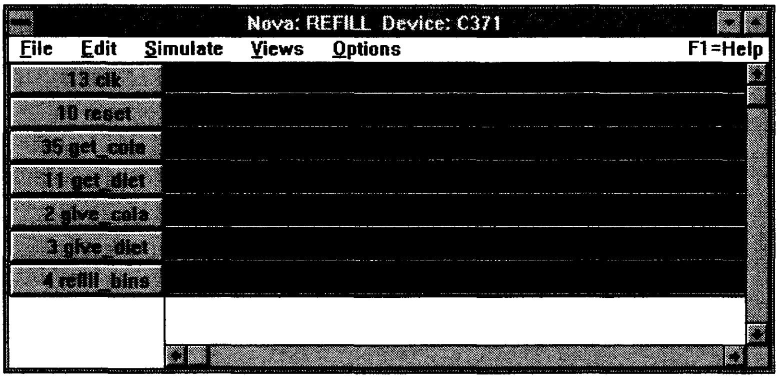


Figure 3-15 Nova screen for tutview

3.9.3 Setting Simulation Length

=> From the main Nova window, pull down the *Options* menu and select *Simulation Length.*

=> In the *Simulation Length* dialog box, click twice on the up arrow to increase simulation length to 384. Click *OK* to close the box.

These steps increase the length of the simulation so that you can see the results of the entire simulation.

3.9.4 Designating a Clock Signal

=> Position the cursor over the `clk` signal button on the left column of all the signal names.

=> Click the left mouse button.

The signal trace should turn blue when the entire signal is selected in this manner. (The trace will become a dashed line on monochrome monitors.)

=> Select the *Clock* item from the *Edit* menu.

=> When the *Edit/Clock* dialog box appears, click on *OK* to accept the default clock value of 10.

Signal `clk` appears as a series of equally-spaced, alternating highs and lows as seen in the figure below.

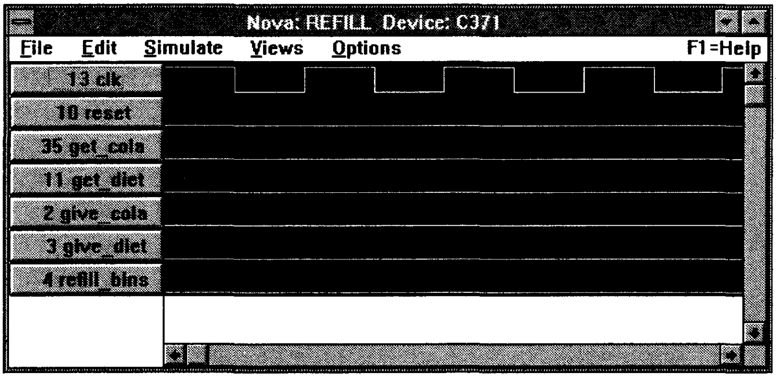


Figure 3-16 Signal clk designated

3.9.5 Setting Stimulus Signal Values

In addition to the clock signal, you need to set the values of the following input signals for your simulation: `reset`, `get_cola`, and `get_diet`.

Set `reset` to high for one rising clock edge. To do so:

=> Position the cursor on the `reset` trace, just to the left of a rising clock edge (a rising clock edge is the vertical line as the clock changes from low to high).

=> Click and hold the left mouse button.

=> Drag the cursor along the trace to the right of the rising clock edge, then release the mouse button.

This portion of the `reset` signal should now be blue.

=> Press the "**1**" key on your keyboard to cause the signal to go high for the selected period.

The portion of the `reset` that was blue should now be a pulse, indicating that the signal is high during this period.

You now need to set `get_cola` high for four non-consecutive rising clock edges.

=> Position the cursor on the `get_cola` trace, just to the left of a rising clock edge and *after* the `reset` signal has gone back to zero.

=> Click and hold the left mouse button.

=> Drag the cursor along the trace to the right of the rising clock edge, then release the mouse button.

=> Press the "**1**" key on your keyboard to cause the signal to go high for the selected period.

=> Click on the signal label buttons which appear on the left side of Nova.

This deselects the signal (the signal line turns white), allowing you to select a new piece to edit.

=> Skip the first rising clock edge *after* the `get_cola` signal has gone back to zero. Then, position the cursor on the `get_cola` trace, just to the left of the next rising clock edge (so that you have non-consecutive rising clock edges).

=> Click and hold the left mouse button.

=> Drag the cursor along the trace to the right of the rising clock edge, then release the mouse button.

=> Press the "1" key on your keyboard to cause the signal to go high for the selected period.

=> Click on the signal label button which appears on the left side of Nova.

You have created two of the four pulses needed for simulation. You need to repeat the above five steps two more times to have a total of four pulses for the get_cola signal.

You now need to set get_diet high for four non-consecutive rising clock edges. These four pulses should come after the four pulses created for the get_cola signal.

=> Position the cursor on the get_diet trace, just to the left of a non-consecutive rising clock edge.

Remember, the rising clock edge selected above should come after the get_cola signals' four pulses. Therefore, the first pulse of the get_diet signal would come no sooner than the tenth rising clock edge -- one clock edge for reset and 8 clock edges for get_cola (you need four non-consecutive rising clock edges).

=> Click and hold the left mouse button.

=> Drag the cursor along the trace to the right of the rising clock edge, then release the mouse button.

=> Press the "1" key on your keyboard to cause the signal to go high for the selected period.

=> Click on the signal label button which appears on the left side of Nova.

You need to create three more pulses for get_diet. Repeat the five steps above three more times to complete the get_diet stimulus.

You now want to set `reset` high for one rising clock edge after the last `get_diet` request.

=> Position the cursor on the `reset` trace, just to the left of a rising clock edge.

Remember, this `reset` pulse should occur after the four `get_cola` pulses and the four `get_diet` pulses.

=> Click and hold the left mouse button.

=> Drag the cursor along the trace to the right of the rising clock edge, then release the mouse button.

=> Press the "**1**" key on your keyboard to cause the signal to go high for the selected period.

Finally, you need to set `get_cola` and `get_diet` high for one rising clock edge, respectively, after the second `reset`.

=> Position the cursor on the `get_cola` trace, just to the left of a rising clock edge.

Remember, this `get_cola` pulse should occur after the second reset signal's pulse.

=> Click and hold the left mouse button.

=> Drag the cursor along the trace to the right of the rising clock edge, then release the mouse button.

=> Press the "**1**" key on your keyboard to cause the signal to go high for the selected period.

=> Position the cursor on the `get_cola` trace, just to the left of a rising clock edge.

Remember, this `get_diet` pulse should occur after the `get_cola` pulse you just created.

=> Click and hold the left mouse button.

=> Drag the cursor along the trace to the right of the rising clock edge, then release the mouse button.

=> Press the "**1**" key on your keyboard to cause the signal to go high for the selected period.

After you have set the values of the input signals, you may wish to change the screen resolution in order to fit all activity in the waveforms on one screen.

=> To do so, select the *Resolution* item from the *Options* menu, then set the resolution to two pixels per simulation tic.

The result, when complete, should look like Figure 3-17:

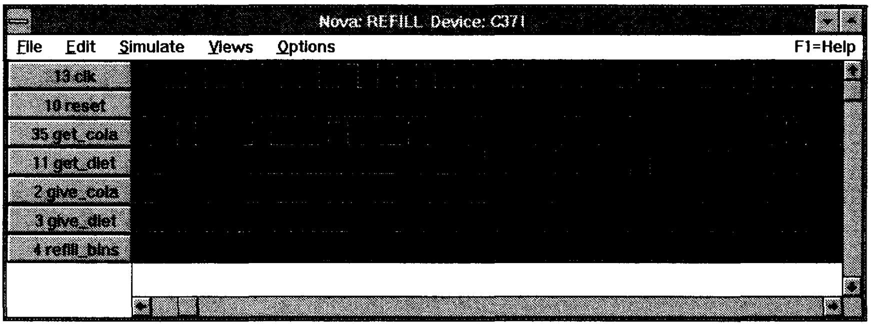


Figure 3-17 Nova window for refill.jed with all the signals set

3.9.6 Running the Simulation

=> To simulate the design, select *Execute* from the *Simulate* menu.

The results should look similar to the figure below:

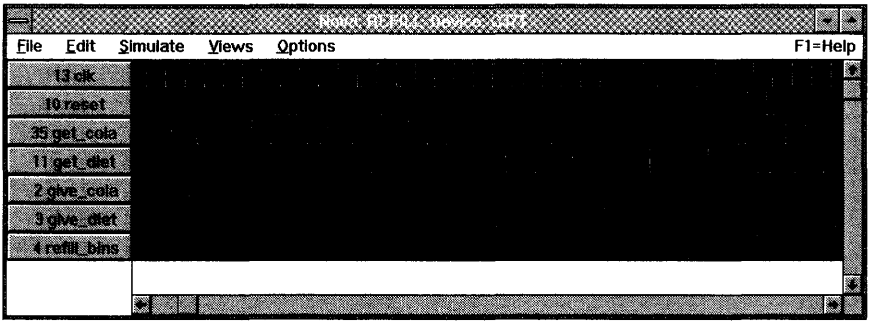


Figure 3-18 Results of refill.jed Nova simulation

The simulation starts with the drink machine empty. (Notice the state of the `refill_bin` signal at the start of the simulation.)

When the `reset` signal goes high near the start of the simulation, the `refill_bin` signal is set low. The drink machine is now ready to dispense drinks.

The drink machine dispenses three cola's in response to the first three requests for a cola. (Note the relationship between the pulses in the `get_cola` and `give_cola` signals.) After the fourth request for a cola, however, the machine does not dispense a drink; the cola bin is empty.

Similarly, the drink machine dispenses three diet's in response to the first three requests for a diet. After the fourth request for a diet, the machine does not dispense one; the diet bin is empty.

With both bins empty, the `refill_bin` signal goes high. It stays high until the `reset` signal goes high again, telling the machine that the bins have been replenished. The next two requests, for a cola and a diet respectively, are honored.

For more information on Nova, please refer to Chapter 6, "Nova," in this *User's Guide*.

Note – If the output of your simulation does not register correctly, make sure that your signals begin and end halfway between rising and falling clock edges.

3.10 Retargeting the Design to an FPGA

Warp also includes support for the pASIC380 Family of UltraLogic™ FPGAs. This section of the soda machine example will retarget the compilation to a 1000 usable gate device, the CY7C381A FPGA. In addition to re-synthesizing the design using the Galaxy interface, you will also need to place and route the design using the SpDE toolkit. This process will take the synthesized description of the VHDL file and place the logic into the FPGA device. The SpDE toolkit also generates all of the files necessary for simulation in ViewSim as well as many other third party simulation environments.

When you ran *Warp* earlier in the tutorial, you compiled *refill.vhd* to produce a JEDEC file. In the following pages, you'll run *Warp* to produce a QDIF file for a specific target device (in this case, a CY7C381A-0JC 1k gate CMOS FPGA).

3.10.1 Starting Galaxy

For *Warp2* Users

=> On Windows systems, double-click on the *Galaxy* icon from the *Warp R4* program group.

=> On UNIX systems, start *Warp2* by typing **`galaxy<CR>`** from within a shell window.

In both systems, the Galaxy window should appear.

For *Warp3* Users

=> On Windows systems, you start *Warp3* by double-clicking on the *Cockpit* icon from the *Warp R4* program group.

=> On UNIX systems, you start *Warp3* by typing **`powerview<CR>`** from within a shell window.

On both platforms, the Cockpit should be displayed immediately after you start *Warp3*. The Cockpit is labeled "Workview PLUS Cockpit" on Windows systems, "Powerview Cockpit" on UNIX systems.

=> Double-click on the *Warp* icon within the Cockpit to launch Galaxy.

Make sure that *refill.vhd* and *binctr.vhd* are the two files listed in the Galaxy window. If they are not:

=> Remove other files from the list by clicking on each file name to highlight it and then choosing *Remove* from the *File* pull-down menu.

=> To add either *binctr.vhd* or *refill.vhd* to the Galaxy window, click on the *File* menu and choose the *Add* option.

The ensuing dialog box lists the VHDL files available in the current directory on the left side, and the VHDL files selected for compilation or synthesis on the right side.

=> Highlight the VHDL file in the left-hand list, and double-click on it or choose and click on the "-->" button.

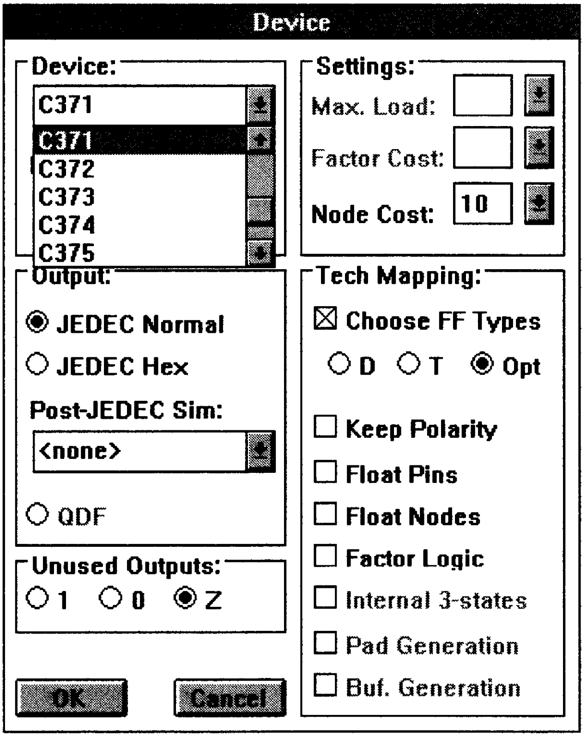
If neither *refill.vhd* or *binctr.vhd* is available, check to make sure you are in the correct project directory. You may need to close this Galaxy window and locate the *wtutor.wpr* project file you created earlier.

3.10.2 Selecting a Device

You must now select a target device and package, set synthesis options, and choose desired output options.

=> Click on the *Device* button from within the *Synthesis options* group.

The *Warp* options dialog box appears.



Device

Device: C371

Settings:
Max. Load:
Factor Cost:
Node Cost: 10

Output:
● JEDEC Normal
○ JEDEC Hex
Post-JEDEC Sim: <none>
○ QDF

Unused Outputs:
○ 1 ○ 0 ● Z

Tech Mapping:
☒ Choose FF Types
○ D ○ T ● Opt
□ Keep Polarity
□ Float Pins
□ Float Nodes
□ Factor Logic
□ Internal 3-states
□ Pad Generation
□ Buf. Generation

Figure 3-19 The Device dialog box showing the Device pull-down menu

=> Click on the down-pointing arrow next to the *Device* label to activate the pull-down menu (Figure 3-19). Scroll down the menu and select the C381A by clicking on it.

3.10.3 Selecting a Package and Speed Bin

In addition to selecting the device you wish to target, you can also choose a package type from the list of available packages.

=> Click on the arrow next to the *Package* label to activate the scroll down menu. Scroll down the menu until the desired package and speed bin is visible. Select the CY7C381A-0JC package by clicking on it.

The pin_number attributes used earlier in *refill.vhd* were specific to the CY7C371-143JC CPLD. The same choice for the pin number assignments do not hold true for the CY7C381A-0JC FPGA. We will float the pins in this compilation by selecting the "Float Pins" option from within the Tech Mapping paremeters. An "X" gets placed along side the "Float Pins" option indicating that this option is selected.

3.10.4 Selecting Other Options in the Device Window

You will use the default Tech Mapping parameters for this compilation.

Notice that the choice of flip-flop type is now grayed-out. This selection is not available for architectural reasons. The flip-flops within a macrocell of a FLASH370 part can be configured to D or T type. The multiplexor-based architecture of Cypress' FPGAs incorporates the inverting functionality of the T flip-flop into the multiplexor logic should your design call for it. Thus, a flip-flop selection is not applicable to the pASIC380 FPGA family of devices.

3.10.5 Launching the Retargeted Compilation

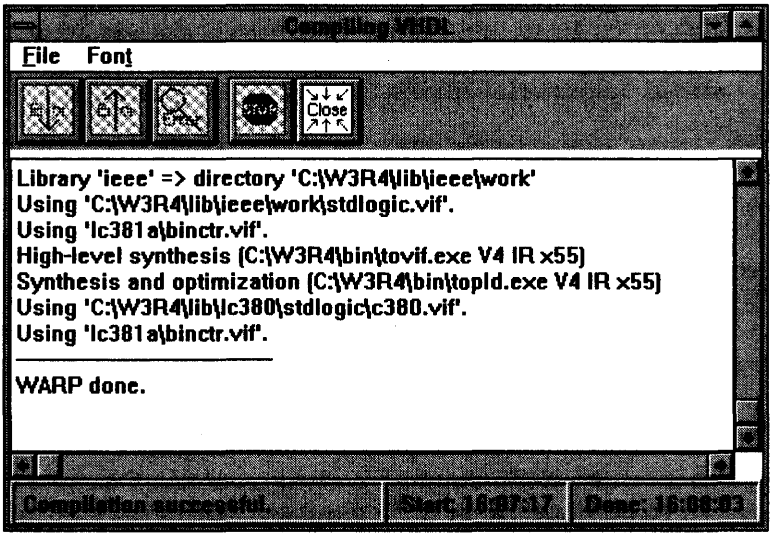
=> Click on the *OK* button to dismiss the Device window.

=> Highlight *refill.vhd* by clicking on it in the main Galaxy window.

=> Click on the *Set Top* button to set your top-level VHDL file.

=> Click on the *Smart* button in the *Compile* button group to begin compilation.

Warp starts the compilation and synthesis of the design into a CY7C381A and prints messages to keep you appraised of its progress in a pop-up window.



```
Library 'ieee' => directory 'C:\W3R4\lib\ieee\work'
Using 'C:\W3R4\lib\ieee\work\stdlogic.vif'.
Using 'lc381a\binctr.vif'.
High-level synthesis (C:\W3R4\bin\tovif.exe V4 IR x55)
Synthesis and optimization (C:\W3R4\bin\topld.exe V4 IR x55)
Using 'C:\W3R4\lib\lc380\stdlogic\c380.vif'.
Using 'lc381a\binctr.vif'.
_______________________
WARP done.
```

Compilation successful. Start: 16:07:17 Done: 16:08:03

Figure 3-20 Successful compilation of refill.vhd to produce a QDIF file

This operation generates two files of particular interest:

- The first is named *refill.qdf.* The *.qdf* file is used as an input into the SpDE toolkit. The *.qdf* file is the input file for the SpDE toolkit. SpDE will then place and route the design and can produce a *.lof* file, which can be used to program a device.
- The second file is named *refill.rpt*. It contains information about the final synthesized design. You can see the *.rpt* file by pulling down the *Info* menu from the Galaxy window and clicking on *Report.* For more information on what is contained in a report file, refer to Chapter 6, "Report File," in the *Reference Manual.*

=> Close the Galaxy compilation window by clicking on the *Close* button located at the top of the Galaxy compilation window.

Note – If compilation errors occur, do the following: make sure the texts of both files are entered **exactly** as shown earlier in this chapter -- or better yet, copy them from the *c:\warp\examples\wtutor* directory-- and then run *Warp* again.

If error messages appear in the pop-up window:

=> Highlight the error message by clicking on it.

=> Click on the *Error* button (the icon that looks like a magnifying glass) located at the top of the Galaxy compilation window.

This opens the VHDL editor with the cursor on the line number that contains the error. Check the code you have entered against the text shown earlier in this chapter.

3.10.6 Running the Place and Route Tool, SpDE

Now that you have generated the QDIF file for *refill.vhd,* you should take the synthesized description of the VHDL file and place the logic into the FPGA device. Once the design has been placed into the FPGA, SpDE, or the place and route tool, users can then generate the simulation files needed for ViewSim.

For *Warp2* Users

=> Select *SpDE* from the *Tools* pull-down menu in Galaxy.

For *Warp3* Users

=> Select *SpDE* from the *Tools* pull-down menu in Galaxy.

=> Alternatively, double-click on the *Place&Rte* icon from within the *Cockpit* to launch SpDE.

For *Warp2* and *Warp3* Users

=> Click on the *folder* button located in the upper left portion of the SpDE window. This is equivalent to selecting *File->Import->QDIF*.

An open-file window appears listing your directory structure on the right and the *.qdf* files on the left.

=> Go to your project directory if you are not already there.

=> Highlight *refill.qdf* by clicking on it.

=> Click on the *OK* button to import the file.

After importing the file, the *hammer* button will now be selectable (it was not available before loading the QDIF file).

=> Click on the *hammer* button in the SpDE window.

A window will appear containing all the tools you can run. The window appears with all of them selected.

=> Click on the *Run* button to begin the place and route process.

The place and route process will take a couple of minutes to complete. Upon completion, a window will appear stating "All chosen SpDE tools ran successfully."

=> Click on the *OK* button to close the above message window.

=> Select *Full Fit* from the *View* pull-down menu.

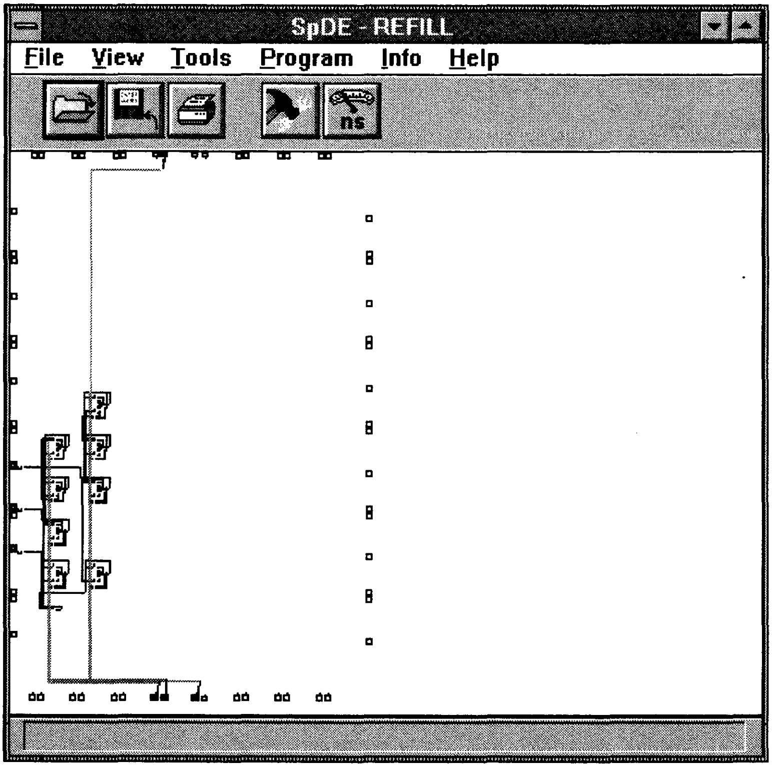


Figure 3-21 SpDE output for refill.qdf

This will allow you to see a view of the whole device with your design placed and routed (Figure 3-21). For more information on SpDE, please see Chapter 5, "SpDE," of this *User's Guide.*

=> Select *Save* under the *File* menu to save your design.

=> Select *Exit* from the *File* pull-down menu.

This tutorial ends here for all *Warp2* users. *Warp2* users can now start on the next exercise which begins at Section 3.13.

3.10.7 Running pASIC->VSim to Generate a ViewSim Model

You now need to generate a ViewSim model from the output of the SpDE tools.

=> Double-click on the *pASIC->VSim* icon in the Cockpit.

A dialog box appears, containing a command line to be executed.

=> Make the command line read **`refill`**, then click on *OK*.

A window appears, informing you of the progress of the application. When the banner of this window reads "Inactive pASIC->VSim," the application is complete.

Note – Verify that your current project directory is set to *c:\w2tutor*. Ignore the messages at the bottom of the window. If the application reports 0 errors and 0 warnings, the application ran successfully. Close the pASIC->VSim window.

You now have the files ViewSim requires to simulate the soda machine design.

3.11 Back-Annotating Pin Assignment Information

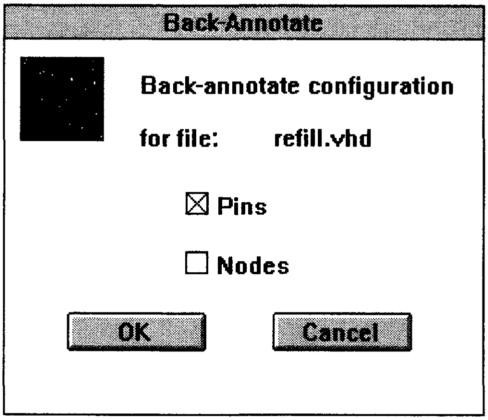
You can easily lock-in the pin assignment made by the place and route tool.

=> Double-click on the *Galaxy* icon in the Cockpit.

=> Highlight *refill.vhd* by clicking on it. If you don't have *refill.vhd* selected, do a *Files->Add* and add *refill.vhd*.

=> Choose *Annotate...* from the *File* pull-down menu.

A small window appears giving the name of the file which will be back-annotated (in this case, *refill.vhd*), and giving you the option of back-annotating the pins, the nodes or both. The *Pins* option should already be selected.



Back-Annotate

Back-annotate configuration

for file: refill.vhd

☒ Pins

□ Nodes

Figure 3-22 Annotate dialog box

=> If *Pins* is not already selected, click on the button to the left of *Pins.*

=> If *Nodes* is selected, deselect it by clicking on the button to the left of *Nodes.*

=> Click on the *OK* button to back-annotate the pin information.

Note – The back-annotation information is stored in a control file. Control files have a *.ctl* extension.

3.12 Simulating the Behavior of the Design with ViewSim

Once the design is synthesized, it's a good idea to simulate its behavior and evaluate its timing performance to ensure that it functions as intended.

Note – Before performing this step of the tutorial, copy the *refill.cmd* file from the *Warp* directory (its default location is *c:\warp\examples\wtutor\refill.cmd*) to your project directory.

You begin by launching ViewSim.

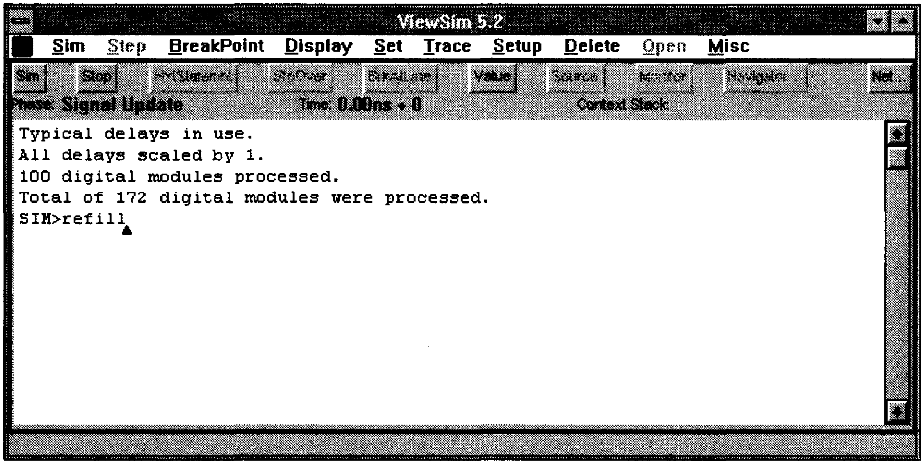
=> Double-click on the *ViewSim* icon in the Cockpit.

=> A dialog box appears. Make sure the design name reads **`refill`**.

=> Click on *OK*.

ViewSim starts up. When the ViewSim window appears ('SIM>'):

=> Type **`refill`** at the command line.



```
Typical delays in use.
All delays scaled by 1.
100 digital modules processed.
Total of 172 digital modules were processed.
SIM>refill
```

Figure 3-23 ViewSim window

The *refill.cmd* file runs, executing the following sequence of ViewSim commands (not necessarily echoed to the screen):

```
wave REFILL.wfm clk reset get_cola give_cola get_diet
   give_diet refill_bins
clock clk 0 1
h reset
l get_cola
l get_diet
cycle
l reset
cycle
h get_cola
cycle 4
l get_cola
h get_diet
cycle 4
l get_diet
h reset
cycle
l reset
```

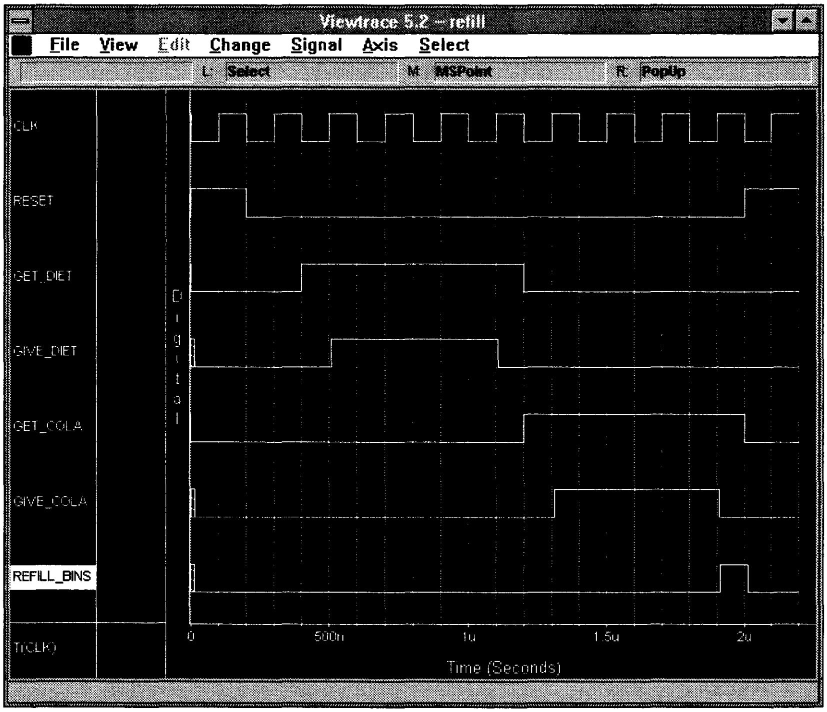


Figure 3-24 ViewTrace output window for refill.vhd

This sequence of commands does the following:

- sets up the waveforms to be traced (`clk`, `reset`, `get_cola`, `give_cola`, `get_diet` , `give_diet` , and `refill_bins`)
- sets up the clock signal
- initializes the inputs to the simulation, sets `reset` high for one clock cycle, then sets `reset` to low
- sets `get_cola` to high for four clock cycles
- sets `get_cola` to low, then sets `get_diet` to high for four clock cycles
- sets `get_diet` to low, then sets reset to high for one clock cycle

Note – Note how the fourth request to get a cola does not result in `give_cola` going high. (The cola bin is empty.) The `refill_bins` signal does not go high, however, because both bins are not empty. Later, the `refill` signal goes high after the third diet is delivered. A fourth request for a diet is ignored. Finally, the `refill` signal goes low when `reset` is asserted to indicate that the bins have been replenished.

3.13 Designing a Parking Garage Monitor

This section takes you step-by-step through another tutorial example, using hierarchy in creating a low-level and top-level behavioral description. In this section, you will:

- write an entity declaration, architecture, and package declaration for a VHDL behavioral description of a simple counter circuit using the *Warp* VHDL Browser
- write a top-level entity and architecture design using behavioral VHDL to implement logic functions and to instantiate the counter circuit, using the *Warp* VHDL Browser
- run *Warp* to synthesize the design's VHDL description
- run Nova to simulate the design's functionality
- **for Warp3 Users only:** run ViewSim to simulate the behavior of the design

When you complete this section, you will know how to:

- take advantage of Cypress' UltraGen module generation technology on various operators
- choose between Area and Speed optimization to obtain results best suited for your application

3.13.1 Design Description

This design keeps track of the total number of cars entering a parking lot garage, the total number of cars that have left the garage, and the number of cars remaining in the parking garage.

For this exercise, you will set the maximum number of parking lot spaces available to be 32. You also need to alert the parking lot attendant if the parking lot is empty or full. Additionally, the attendant should also be able to reset the count of the number of cars in the garage to a zero, at his own discretion.

3.13.2 Design Solution

The solution presented in this chapter will proceed as follows:

In this tutorial section, you're going to create two VHDL files, *counter.vhd* and *total.vhd.* The *counter.vhd* file shows the implementation of a counter that increments the count by 1 on the rising edge of a clock (trigger). It also contains an asynchronous reset signal that will reset the counter to zero. This counter is a variable size counter with default size of a 4-bit output data.

The top-level VHDL file, *total.vhd,* instantiates the counter twice. The first counter keeps track of the incoming cars, and the second counter keeps track of the cars exiting the parking garage. The top-level file takes the output of the second counter and subtracts it from the output of the first counter to give the attendant the number of cars remaining in the parking lot. There are two signals, `lot_empty` and `lot_full`, which let the attendant know if the garage is empty or full respectively. You'll set both counters to have data width of 5 bits. After that, you'll synthesize the *counter* and *total* VHDL descriptions into a CY7C371 CPLD and simulate the behavior of the resulting design using the Nova simulator. Users will also synthesize the design into a CY7C381A FPGA. *Warp3* users will simulate this design with the ViewSim simulator.

This tutorial reinforces some basic VHDL constructs and the concept of hierarchical designs. This exercise is also intended to make the user more familiar with the *Warp* tool flow.

3.14 Starting Galaxy

For *Warp2* Users

=> On Windows systems, start *Warp2* by double-clicking on the *Galaxy* icon from the *Warp R4* program group.

=> On UNIX systems, start *Warp2* by typing **`galaxy<CR>`** from within a shell window.

In both systems, the Galaxy window should appear.

For *Warp3* Users

=> On Windows systems, you start *Warp3* by double-clicking on the *Cockpit* icon from the *Warp R4* program group.

=> On UNIX systems, you start *Warp3* by typing **`powerview<CR>`** from within a shell window.

On both platforms, the Cockpit should be displayed immediately after you start *Warp3*. The Cockpit is labeled "Workview PLUS Cockpit" on Windows systems, and"Powerview Cockpit" on UNIX systems.

=> Double-click on the *Warp* icon within the Cockpit to launch Galaxy.

Note – If you have not run Galaxy earlier - before launching Galaxy, **you must first create a project.**

3.15 Creating a Project

Users who created a project for the earlier exercise can skip this section.

For *Warp 2* Users

=> Under the *Project* menu, choose *New*. A new Galaxy window will appear, prompting you for a new name. On the PC enter **`c:\w2tutor\w2tutor`** and on UNIX systems, enter **`<user home directory path>/ w2tutor/w2tutor`**.

You should now see a Galaxy window identical to the one that appeared when you started *Warp,* except that it is named c:\w2tutor\wtutor.

You have now created a project directory for your designs and called it *w2tutor.* All files pertinent to this tutorial need to be placed in this project directory. You have also created a project file called *w2tutor.wpr* which is now located in the *c:*\w2tutor directory.

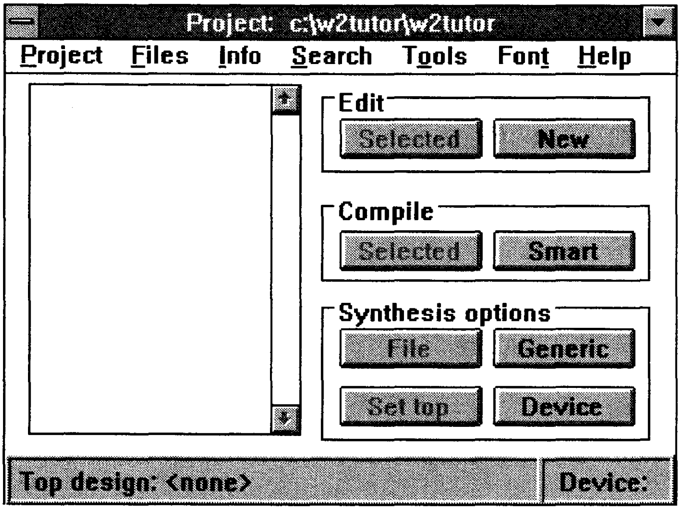


Figure 3-25 Galaxy window named c:\w2tutor\w2tutor

For *Warp3* Users

=> Select *Project->Create* from the Cockpit menu bar. When a dialog box appears, on the PC type the pathname **`c:\w2tutor`** and on UNIX systems, type **`<user home directory path>/w2tutor`** and then click *OK.*

When doing the tutorial exercises (or any other time, for that matter), **don't write anything into the *Warp* directory**. Instead, create a separate directory to practice in, like the *w2tutor* directory you have just created above.

By default, the *Warp3* installation procedure for IBM PCs and compatible computers installs *Warp3* software into a directory named *c:\warp*. On UNIX workstations, the user needs to point the CYPRESS_DIR environment variable to the location of the *Warp3* software.

3.16 Writing the VHDL File

The VHDL Browser is a good tool for beginners to learn how to design with VHDL. The Browser also acts as a way to find out the correct syntax for using VHDL constructs.

There are three parts to the *counters.vhd* file:

- the **entity** declaration declares the name, direction, and data type of each port of the component
- the **architecture** describes the behavior of the component
- the **package** declaration provides the information to *Warp* to allow counter to be used as a component in a higher-level design

The following pages briefly discuss the contents of each of these sections of the VHDL description. For a more detailed explanation of these VHDL constructs, please refer to the textbook which was included in your software kit entitled *VHDL for Programmable Logic*. You will use the VHDL Browser to create *counter.vhd*.

Note – If you would rather not type the *counter.vhd* file yourself, you can copy it from the *Warp* directory. The default location for the *counter.vhd* file is *c:\warp\examples\wtutor\counter.vhd*. From this location, copy *counter.vhd* to your project directory. Then, read along for the next few pages to help you understand the purpose of each section of a VHDL source file.

For Both *Warp2* and *Warp3* Users

=> To bring up an editor, click on the *New* button from the *Edit* button group from within Galaxy.

This brings up the Galaxy VHDL editor.

=> Choose the *Browse* option from the *VHDL* pull-down menu.

This should bring up the VHDL Browser shown in Figure 3-26.

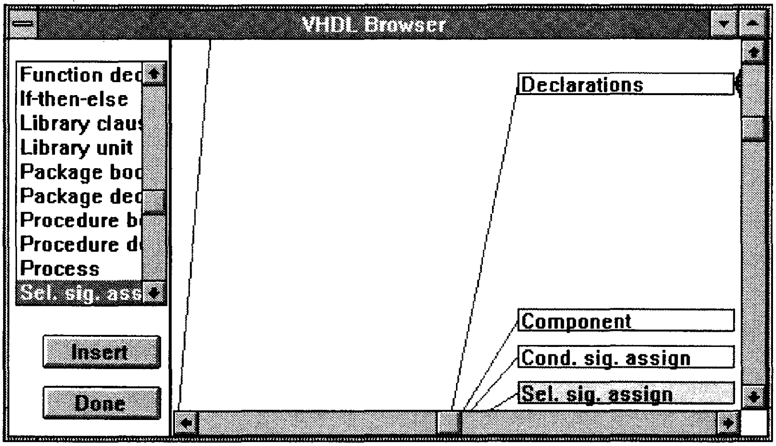


Figure 3-26 VHDL Browser

3.16.1 Writing the Entity Declaration

The entity declaration declares the name, direction, and data type of each port of the component.

You will use the VHDL Browser for creating the various parts of *counter.vhd.*

=> Choose *Entity decl.* among the various constructs listed on the left-side of the Browser screen.

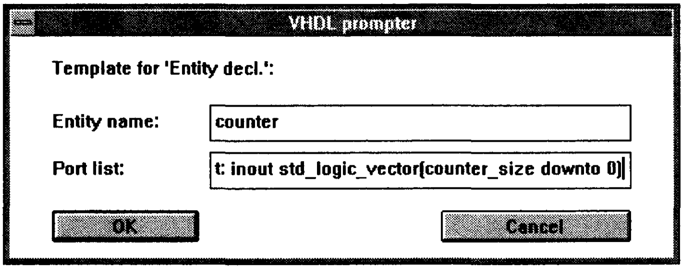
=> Double-click on it or click on *Insert.*

This brings up the VHDL prompter.

=> Type in **counter** in the field for the *Entity name:* and type the following information in the *Port list:* field without a carriage return.

```
trigger, reset : in std_logic; count: inout
std_logic_vector(counter_size downto 0)
```

This adds signals with their modes and types to the entity declaration.



Template for 'Entity decl.':

Entity name: counter

Port list: t: inout std_logic_vector(counter_size downto 0)

Figure 3-27 VHDL prompter for entity declarations

When you are done entering this information into the VHDL prompter, the VHDL prompter should look like Figure 3-27.

=> Select *OK* and the information presented to the prompter will get translated into the correct VHDL syntax in the VHDL editor.

=> Add **(counter_size: integer:= 4);** to the generic declaration so that the resulting entity declaration looks like the lines below, except for the comment line:

```
entity counter is
-- setting the default size of the counter
        GENERIC (counter_size : integer:=4 );
        PORT (trigger, reset : in std_logic;
                        count : inout
std_logic_vector(counter_size downto 0) );
END counter;
```

=> Place the cursor at the start of *counter.vhd.*

=> Choose *Library clause* in the VHDL Browser.

=> Double-click on it to bring up the VHDL prompter with the *Name:* field.

=> Type in **IEEE** and click *OK.*

=> Double-click on *Use clause* from the VHDL Browser.

=> Fill the following information in the VHDL prompter: **`ieee.std_logic_1164.all`** and click *OK*.

=> Double-click on *Use clause* from the VHDL Browser and type in **`work.std_arith.all`** and click *OK*.

=> Select *Save As* from the *File* pull-down menu to save the contents of the VHDL Editor. Save the file as *c:\w2tutor\counter.vhd.*

3.16.2 Writing the Architecture

The architecture portion of a VHDL description describes the behavior of the component. The architecture appears ***after*** the entity declaration in the *.vhd* file.

The architecture portion of this design defines a counter that increments by 1, on the rising edge of the `trigger` signal. You also need to define an asynchronous reset signal, `reset`, to reset the counter bits to 0.

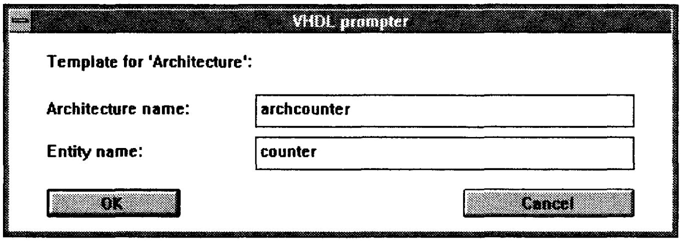
Architecture body definition:

=> Choose *Architecture* in the scrolling menu on the left of the Browser screen.

=> Double-click on *Architecture* or click on the *Insert* button.

=> Type in **`archcounter`** in the field for the *Architecture name* field and **`counter`** in the *Entity name:* field.

The VHDL prompter for the architecture definition is shown below.



Template for 'Architecture':

Architecture name: archcounter

Entity name: counter

Figure 3-28 VHDL Prompter for architecture definition

=> When you are done entering this information into the VHDL prompter, select *OK*.

The VHDL Browser will translate your entries into the correct VHDL syntax in the VHDL editor. The architecture portion of the *counter.vhd* file should look like this:

```
ARCHITECTURE archcounter OF counter IS
BEGIN
END archcounter;
```

=> Place the cursor after the `begin` statement in the VHDL Editor.

=> Highlight *Process* from the VHDL Browser.

=> Double-click on *Process* or click on the *Insert* button.

This will automatically place the information shown below in the architecture declaration portion of *counter.vhd.*

```
PROCESS
        -- Declarations
BEGIN
        -- Statements
END PROCESS;
```

You need to associate a sensitivity list with the process. The process is to be made sensitive to the signals `reset` and `trigger`.

=> Type **`(reset, trigger)`** after the word `process` in *counter.vhd.*

=> Place the cursor after the `begin` statement in the `process` statement in the VHDL Editor.

=> Highlight *if-then-else* from the VHDL Browser and double-click on it.

This will bring up a VHDL prompter.

=> Enter **`(reset = '1')`** in the *condition* field and choose *OK.*

This will add the following lines to the *counter.vhd* file:

```
BEGIN
   If reset= '1' THEN
  --ELSIF <CONDITION>
      THEN
  --ELSE
END IF:
```

=> Place the cursor after the `if-then` statement in the VHDL Browser, choose *Signal assign.* from the VHDL Browser, and double-click on it.

=> Within the VHDL browser, enter **`count`** in the *Signal name:* field and **`(others => '0')`** for the *Value:* field.

This information tells the compiler that an asynchronous reset signal `reset` is associated with the counter and that the counter bits should get reset to a '0' whenever the `reset` signal goes high.

Instead of count <="0000" you assigned `(others => '0')` in order to preserve the generic nature of the counter. For more information about the usage of this operator, please refer to Chapter 4, "VHDL," of the *Reference Manual.*

=> Uncomment the `elsif-then` statement in the VHDL Browser by deleting the `--` before `elsif` and replacing `<condition>` with the following statement: **`(trigger'event and trigger = '1').`**

This instructs *Warp* to make the counter sensitive to the rising edge of the `trigger` signal.

=> Place the cursor on the next line after the `elsif-then` clause in *counter.vhd* and choose and double-click on *Signal assign.* from the VHDL Browser.

=> Type **`count`** in the *Signal name:* field and **`count + 1`** in the *Value:* field and click *OK.*

When you have completed the architecture and entity declaration for *counter.vhd,* you should have the following architecture declaration.

```
ARCHITECTURE archcounter OF counter IS
BEGIN
  PROCESS (reset, trigger)
        BEGIN
                IF reset='1' THEN
                        count <= (others => '0');
                ELSIF (trigger'event and trigger='1') then
                        count <= count+1;
                END IF;
  END PROCESS;
END archcounter;
```

=> Save the file by clicking on *Save* from the *File* pull-down menu.

3.16.3 Writing the Package Declaration

The package declaration provides the information to the *Warp* compiler to allow *counter* to be used as a component in a higher-level design.

=> Place the cursor at the start of *counter.vhd.*

=> Highlight *Library clause* in the VHDL Browser.

=> Double-click on *Library clause* to bring up the VHDL prompter with the field *name.*

=> Type in **`IEEE`** and click *OK.*

=> Double-click on *Use clause* from the VHDL Browser.

=> Fill the following information in the VHDL prompter:
`ieee.std_logic_1164.all`

=> Double-click on *Package decl.* from the VHDL Browser.

=> Type **`count_lib`** in the VHDL prompter.

=> Place the cursor after the `package` statement.

=> Double-click on *Comp. decl.* from the VHDL Browser.

=> Type **`counter`** in the *Name:* field and click *OK* in the VHDL prompter.

The port statement declares the name, direction, and type of each port in the component. You can copy the port statement from the entity declaration for this purpose or copy the following lines:

```
GENERIC (counter_size: integer:=4 );
        PORT ( trigger, reset : in std_logic;
               count : inout std_logic_vector(counter_size
               downto 0) );
```

The package declaration should now look like the lines below:

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
PACKAGE count_lib IS
COMPONENT counter
-- setting the default size of the counter
        GENERIC (counter_size: integer:=4 );
        PORT ( trigger, reset : in std_logic;
               count : inout std_logic_vector(counter_size
               downto 0) );
END COMPONENT;
END count_lib;
```

=> Save the file as *c:\w2tutor\counter.vhd.*

Below is the complete VHDL listing of *counter.vhd*.

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;

PACKAGE count_lib IS
COMPONENT counter
-- setting the default size of the counter
        GENERIC (counter_size : integer:=4 );
        PORT ( trigger, reset : in std_logic;
               count : inout std_logic_vector(counter_size
               downto 0) );
END COMPONENT;
END count_lib;

LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;
```

```
entity counter is
-- setting the default size of the counter
        GENERIC (counter_size : integer:=4 );
        PORT ( trigger, reset : in std_logic;
               count : inout std_logic_vector(counter_size
               downto 0) );
END counter;

ARCHITECTURE archcounter OF counter IS
BEGIN
  PROCESS (reset, trigger)
        BEGIN
                IF reset='1' THEN
                        count <= (others => '0');
                ELSIF (trigger'event and trigger='1') then
                        count <= count+1;
                END IF;
  END PROCESS;
END archcounter;
```

3.16.4 Creating the Top-Level Description

You will now create a top-level VHDL file to achieve the desired functionality for a parking garage counter. You will use the component *counter* discussed earlier. You will continue using the VHDL Browser to create *total.vhd.*

=> To start a new VHDL file using the VHDL Browser, click on the *New* button in the *Edit* button group.

You will start by providing the compiler with information on the library and package needed for the `std_logic` and `std_logic_vector` types.

=> Place the cursor at the starting line of the VHDL editor.

=> Highlight *Library clause* in the VHDL Browser.

=> Double-click on *Library clause* to bring up the VHDL prompter with the field *name.*

=> Type in **IEEE** and click *OK.*

=> Double-click on *Use clause* from the VHDL Browser.

=> Fill the following information in the VHDL prompter: **ieee.std_logic_1164.all.**

=> Double-click on *Use clause* from the VHDL Browser, typing in **work.std_arith.all** when prompted.

=> Double-click on *Use clause* from the VHDL Browser, typing in **work.count_lib.all** when prompted.

By using the count_lib package, you are allowing *total.vhd* access to the counter model described in *counter.vhd.*

=> Place the cursor after the USE clauses.

=> Double-click on *Entity decl.* from the VHDL browser.

=> Type **total** in the *Entity name:* field and **car_enter, car_exit, reset, lot_empty, lot_full, count1, count2, total** in the *Port list:* field and click *OK.*

=> Place the cursor on the line before the end statement.

=> From the VHDL browser, highlight *Attr. spec.* by clicking on it.

=> Click on the *Insert* button.

=> In the *Name*: field, type **synthesis_off.**

=> In the *Symbols*: field, type **total**.

=> In the *Class:* field, type **signal**.

=> In the *Value*: field, type **true.**

=> Click *OK.*

=> Modify the entity port list to include signal type and mode to match the complete entity listing below.

This is the complete listing for the entity of *total.vhd.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;
USE work.count_lib.all;

ENTITY total IS
      GENERIC (size:integer:=5);
      PORT ( car_enter : in std_logic;
        car_exit        : in std_logic;
        reset           : in std_logic;
        lot_empty       : out std_logic;
        lot_full        : out std_logic;
        count1: inout std_logic_vector((size-1) downto 0);
        count2: inout std_logic_vector((size-1) downto 0);
       total : buffer std_logic_vector ((size-1) downto 0));
attribute synthesis_off of total: signal is true;
END total;
```

Now you are ready to create the architecture declaration for *total.vhd.*

=> Place the cursor after the entity declaration.

=> Double-click on *Architecture* in the VHDL Browser.

=> Type in **archtotal** in the *Architecture name:* field and **total** in the *Entity name:* field.

=> Place the cursor before the `begin` statement in the architecture declaration.

=> Double-click on *Signal decl.* in the VHDL Browser.

=> Type in **full** in the *Name:* field and **std_logic_vector((size-1)** in the *Type:* field.

=> Place the cursor after the `begin` statement.

=> Double-click on *Component* from the VHDL Browser.

=> Type in **counter1** in the *Label:* field and **counter** in the *Component:* field in the VHDL Browser.

=> Double-click on *Component* from the VHDL Browser.

=> Type in **counter2** in the *Label:* field and **counter** in the *Component:* field in the VHDL Browser.

=> Close the VHDL Browser by clicking on the *Done* button.

=> Modify the `<association list>` in the `port map` declaration for the label `counter1` and `counter2` so that it matches the following complete counter instantiation:

```
counter1: counter
            GENERIC MAP(size-1)
            PORT MAP(car_enter, reset, count1);

counter2: counter
            GENERIC MAP(size-1)
            PORT MAP(car_exit, reset, count2);
```

`counter1` keeps track of the number of cars coming and `counter2` keeps track of cars leaving the garage.

You now want to take the outputs from the two counters, `count1` and `count2`, and determine the difference between the two. Instead of building a subtractor from gates, you'll take advantage of the UltraGen module generation feature. *Warp* recognizes the operator "-" and knows that a subtractor is required. *Warp* then looks at your target device along with your optimization goal of speed or area, and replaces the "-" operator with a hand-crafted module of a subtractor from a library pre-optimized for either area or speed and your target device.

=> Type **total <= count1 - count2;** after the counter instantiations.

=> Type the following lines above the counter instantiations and under the word `begin` to generate the `lot_empty` and `lot_full` signal:

```
full <= (others => '1');
lot_empty <= '1' when (total = 0) else '0';
lot_full <= '1' when (total = full) else '0';
```

You could have generated the `lot_full` signal by comparing `total` with "11111". However, defining `full` as `full <= (others => '1');` allows you to make changes to the size of the design without changing the architecture of the VHDL code.

You have now finished entering the code for *total.vhd*. The final version of *total.vhd* is listed below. The block diagram for *total.vhd* is shown here:

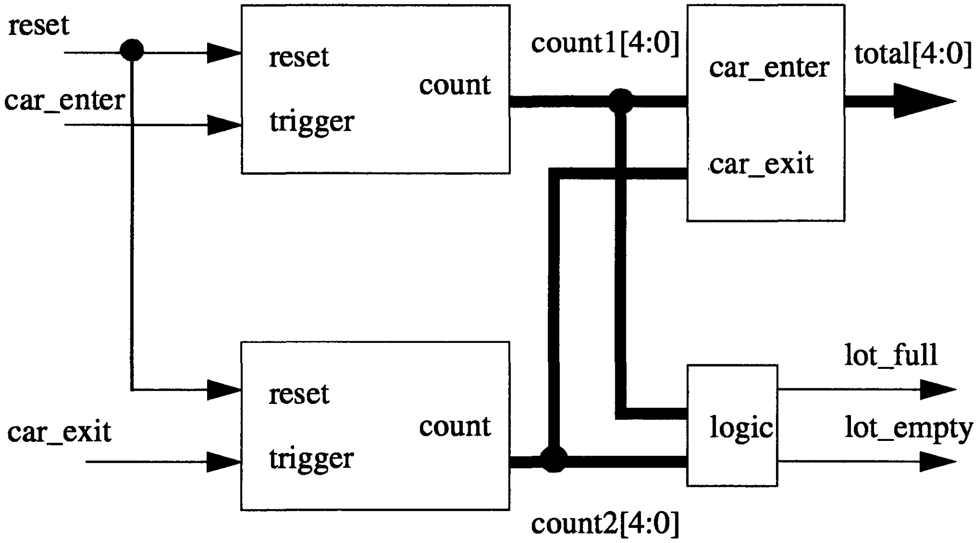


Figure 3-29 Block Diagrm for total.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;
USE work.count_lib.all;
```

```
ENTITY total IS
        GENERIC (size:integer:=5);
        PORT ( car_enter : in std_logic;
               car_exit : in std_logic;
               reset : in std_logic;
               lot_empty : out std_logic;
               lot_full : out std_logic;
               count1 : inout std_logic_vector ((size-1)
                              downto 0);
               count2 : inout std_logic_vector ((size-1)
                              downto 0);
               total : buffer  std_logic_vector((size-1)
                              downto 0));
attribute synthesis_off of total: signal is true;
END total;

ARCHITECTURE archtotal OF total IS
SIGNAL full : std_logic_vector((size-1) downto 0);

BEGIN

full <= (others => '1');
        lot_empty <= '1' when (total = 0) else '0';
        lot_full <= '1' when (total = full) else '0';

counter1: counter
         GENERIC MAP(size-1)
         PORT MAP(car_enter, reset, count1);
counter2: counter
         GENERIC MAP(size-1)
         PORT MAP(car_exit, reset, count2);

total <= count1 - count2;

END archtotal;
```

3.17 Compiling and Synthesizing the Design

3.17.1 Starting Galaxy

For *Warp2* Users

=> On Windows systems, double-click on the *Galaxy* icon from the *Warp R4* program group.

=> On UNIX systems, start *Warp2* by typing **`galaxy<CR>`** from within a shell window.

In both systems, the Galaxy window should appear.

For *Warp3* Users

=> On Windows systems, you start *Warp3* by double-clicking on the *Cockpit* icon from the *Warp R4* program group.

=> On UNIX systems, you start *Warp3* by typing **`powerview<CR>`** from within a shell window.

On both platforms, the Cockpit should be displayed immediately after you start *Warp3*. The Cockpit is labeled "Workview PLUS Cockpit" on Windows systems, and "Powerview Cockpit" on UNIX systems.

=> Double-click on the *Warp* icon within the Cockpit to launch Galaxy.

Now that the Galaxy window has appeared, you need to add the files to compile.

=> Choose *Files->Add.*

The ensuing dialog box lists the VHDL files available in the current directory on the left side, and the VHDL files selected for compilation or synthesis on the right side.

=> Highlight the VHDL file in the left-hand list, and double-click on it or choose and click on the "-->" button.

=> To deselect a file for compilation or synthesis, highlight the VHDL file in the right-hand list and double-click on the file, or click on the "<--" button.

=> Make sure that *counter.vhd* and *total.vhd* are the two files listed in the Galaxy window.

3.17.2 Selecting a Device

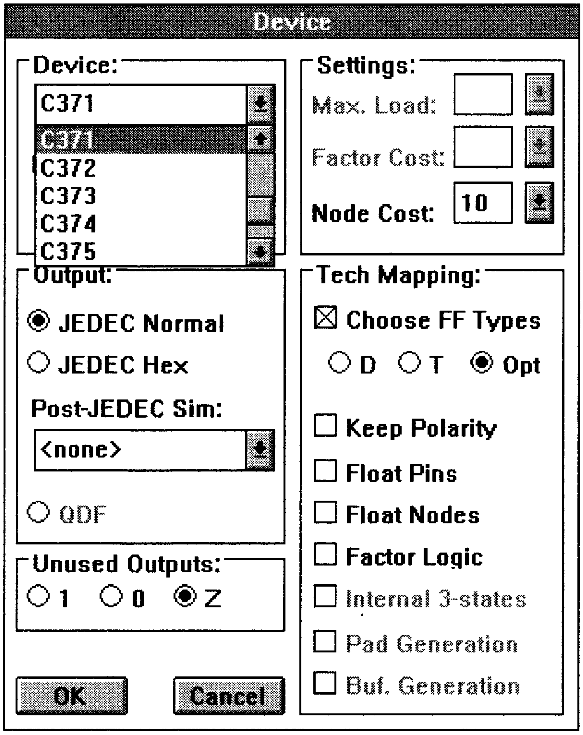
You must now select a target device and package, set synthesis options, and choose desired output options.

=> Click on the *Device* button from within the *Synthesis options* group.

The Device window should appear.

=> Click on the down-pointing arrow next to the *Device* label to activate the pull-down menu. Scroll down the menu and select the C371 by clicking on it.

C371 should appear as the targeted device in the lower right-hand corner of the main Galaxy window.



Device: C371

Settings:

Max. Load:

Factor Cost:

Node Cost: 10

Output:

● JEDEC Normal

○ JEDEC Hex

Post-JEDEC Sim: <none>

○ QDF

Unused Outputs:

○ 1 ○ 0 ● Z

Tech Mapping:

☒ Choose FF Types

○ D ○ T ● Opt

□ Keep Polarity

□ Float Pins

□ Float Nodes

□ Factor Logic

□ Internal 3-states

□ Pad Generation

□ Buf. Generation

Figure 3-30 The Device window with a C371 selected

3.17.3 Selecting a Package and Speed Bin

In addition to selecting the device you wish to target, you can also choose a package type and a speed bin from the list of available packages.

=> Click on the arrow next to the *Package* label to activate the scroll down menu. Scroll down the menu until the desired package is visible. Select the desired package by clicking on it.

3.17.4 Resolving Unused Outputs

Warp gives you the option of turning all of your unused output pins that have unused macrocells into '1's, '0's, or 'Z's. This is a global option and cannot be applied on a signal-by-signal basis. This option is useful for driving all unused pins to a certain logic level.

=> Under the *Unused Outputs* options of the Device window, choose *Z*, the default option of leaving all unused I/O pins to be three-stated.

Note – When using MAX340 EPLDs and FLASH370 CPLDs, it is a recommended practice to use external pull-ups for unused I/O pins.

3.17.5 Choosing Tech Mapping Options

While compiling registered equations, the fitter will use the directive from *Choose FF types* to synthesize equations. Leaving the option *Opt* selected is the best choice. This enables the fitter to make the choice between a D-FF and a T-FF implementation and then choose the implementation that uses the least number of product terms.

=> Under *Tech Mapping* options in the Device window, choose the *Opt* option.

There are a few other useful options in the *Tech Mapping* options and the *Settings* options. Refer to Chapter 4, "Galaxy," of this *User's Guide* for a detailed description of each option.

=> Click on the *OK* button to dismiss the Device window.

3.17.6 Setting the Top-Level File

=> Highlight *total.vhd* by clicking on it.

=> Click on the *Set Top* button in the *Synthesis options* section to assign your top-level VHDL file.

total.vhd should be listed as the top file in the lower left-hand corner of your Galaxy window.

3.17.7 Compiling and Synthesizing the File

=> In your Galaxy window, click on the *Smart* button in the *Compile* button group to begin compilation.

Warp starts the compilation and synthesis of the design into a CY7C371 and prints messages to keep you appraised of its progress in a pop-up window.

This operation generates two files of particular interest:

- The first is named *total.jed.* The *.jed* file can be used to program a CY7C371 device. It is also used as the input to the Nova functional simulator.
- The second file is named *total.rpt.* It contains pinout and timing information, along with other information about the final synthesized design. You can see the *.rpt* file by pulling down the *Info* menu from the Galaxy window and clicking on *Report.* For more information on what is contained in a report file, refer to Chapter 6, "Report File," in the *Reference Manual.*

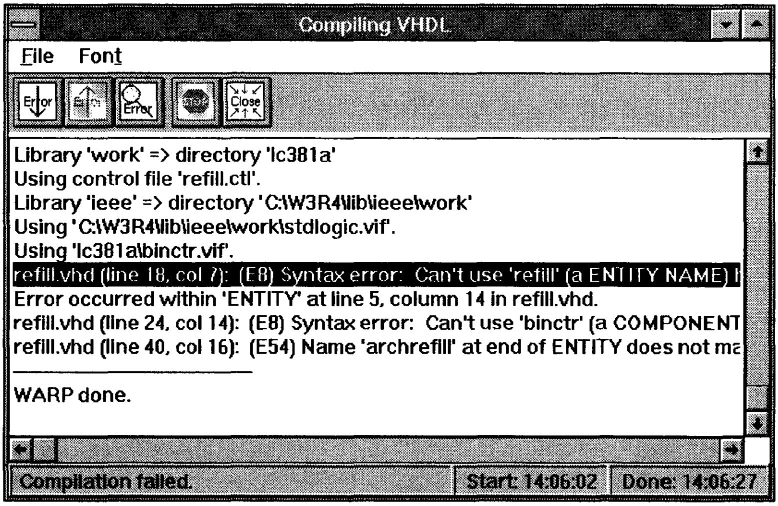
=> Close the Galaxy compilation window by clicking on the *Close* button located at the top of the Galaxy compilation window.

Note – If compilation errors occur, do the following: make sure the text of your *binctr.vhd* file is entered **exactly** as shown earlier in this chapter -- or better yet, copy it from the *c:\warp\examples\wtutor* directory -- and then run *Warp* again.

If error messages appear in the pop-up window:

=> Highlight the error message by clicking on it (Figure 3-31).

=> Click on the *Error* button (the icon that looks like a magnifying glass) located at the top of the Galaxy compilation window.



```
Library 'work' => directory 'lc381a'
Using control file 'refill.ctl'.
Library 'ieee' => directory 'C:\W3R4\lib\ieee\work'
Using 'C:\W3R4\lib\ieee\work\stdlogic.vif'.
Using 'lc381a\binctr.vif'.
refill.vhd (line 18, col 7):  (E8) Syntax error:  Can't use 'refill' (a ENTITY NAME) I
Error occurred within 'ENTITY' at line 5, column 14 in refill.vhd.
refill.vhd (line 24, col 14):  (E8) Syntax error:  Can't use 'binctr' (a COMPONENT
refill.vhd (line 40, col 16):  (E54) Name 'archrefill' at end of ENTITY does not ma
________________________

WARP done.
```

```
Compilation failed.                    Start: 14:06:02   Done: 14:06:27
```

Figure 3-31 Compile window with an error message

This opens the VHDL editor with the cursor on the line number that contains the error. Check the code you have entered against the text shown earlier in this chapter.

3.18 Simulating the Behavior of the Design with Nova

Once the design is synthesized, you should simulate and verify its behavior to ensure that it functions as intended.

You'll use the Nova simulator to test the behavior of the design implementation. In this tutorial, you'll perform the following steps:

- start Nova
- open the *total.jed* file
- set the values of the stimulus signals in the simulation
- simulate the design
- examine results to figure out what happened

3.18.1 Starting Nova

For *Warp2* Users

=> On Windows systems, start Nova by double-clicking on the *Nova* icon within the *Warp R4* program group.

=> On UNIX systems, start Nova by typing **`nova<CR>`** at a shell prompt.

In both systems, the Nova window should appear.

For *Warp3* Users

=> On Windows systems, you start *Warp3* by double-clicking on the *Cockpit* icon from the *Warp R4* program group.

=> On UNIX systems, launch the Cockpit by typing **`powerview<CR>`** from within a shell window.

On both platforms, the Cockpit should be displayed immediately after you start *Warp3*. The Cockpit is labeled "Workview PLUS Cockpit" on Windows systems, and "Powerview Cockpit" on UNIX systems.

=> Double-click on the *Nova* icon within the Cockpit to start Nova.

The Nova screen appears, followed by the Nova *About* box. The *About* box goes away by itself in a few seconds. If you want to make it go away faster, click anywhere in the *About* box.

For Both *Warp2* and *Warp3* Users

=> Open the *total.jed* file by choosing *File->Open.*

The available JEDEC files (*.jed* extension) appear on the left side of the dialog box.

=> Highlight *total.jed* by clicking on it.

=> Click on the *OK* button to open *total.jed*.

3.18.2 Creating a View

A view is the collection of signals available for viewing on the Nova screen. To make it easier to see what's going on in your simulation, you'll create your own view to show only the signals you want to see.

=> Choose the *Edit Views* item from the *Views* menu.

=> Click on the *New View* button.

=> In the ensuing dialog box, give the new view any unique name; **`carview`** will do nicely.

=> Click on the signal name in the *Full View* portion of the window, then on the *Add*>> button, for each of the following signals:

```
car_enter
car_exit
reset
lot_empty
lot_full
count1_0
count1_1
count1_2
count1_3
count1_4
count2_0
count2_1
count2_2
count2_3
count2_4
total_0
total_1
total_2
total_3
total_4
```

=> Click on *OK.*

=> From the *Edit* menu, select *Create Bus.* Add the bits, `total_4`, then `total_3`, `total_2`, `total_1` and `total_0` in descending order. Fill in the name box with **`Total`**.

=> Again, from the *Edit* menu, select *Create Bus.* Add the bits, `count1_4`, `count1_3`, `count1_2`, `count1_1`, and `count1_0`, in descending order. Name this bus **Entered.**

=> Follow the steps above to create a bus named **Exited** for `count2_4`, `count2_3`, `count_2`, `count2_1` and `count2_0`.

=> Choose the *Edit Views* item from the *Views* menu.

=> From within **carview**, on the right-hand side, highlight the following signal names :

```
count1_0
count1_1
count1_2
count1_3
count1_4
count2_0
count2_1
count2_2
count2_3
count2_4
total_0
total_1
total_2
total_3
total_4
```

=> Then select *cut* and click *OK.*

3.18.3 Setting the Simulation Length

=> From the *Options* menu, select *Simulation Length.*

=> Set the simulation length to be 512 by clicking on the up arrow four times.

You have set a longer simulation length so that you can see more results of the simulation.

3.18.4 Setting a Clock Signal

=> Position the cursor over the signal `car_enter` on the left-hand list of all the signal names.

=> Click the left mouse button.

The signal trace should turn blue when the entire signal is selected in this manner. (The trace will become a dashed line on monochrome monitors.)

=> Select the *Clock* item from the *Edit* menu.

=> When the *Edit->Clock* dialog box appears, set the clock period to be 10ns and the clock high time to be 5ns.

The clock waveform for the signal `car_enter` simulates cars entering the parking garage.

Set `reset` to high for one rising clock edge. To do so:

=> Position the cursor on the *reset* trace, just to the left of a rising clock edge of the `car_enter` signal.

=> Click and hold the left mouse button.

=> Drag the cursor along the trace to the right of the rising clock edge, then release the mouse button.

This portion of the signal should now be blue.

=> Press the "**1**" key on your keyboard to cause the signal to go high for the selected period.

The new view should appear on the Nova display screen, and should look like the following figure:

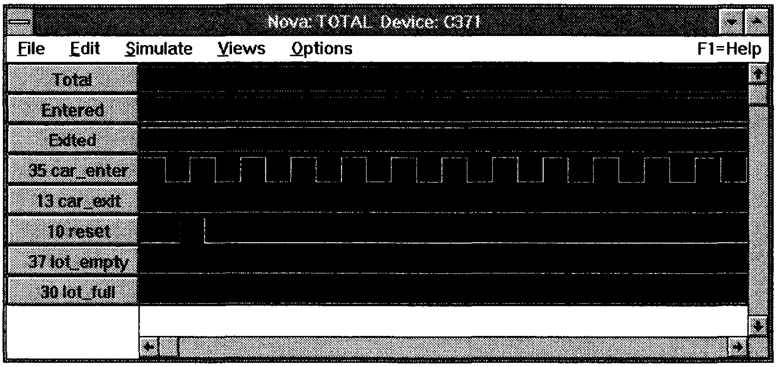


Figure 3-32 Nova screen with car enter signal set

=> Select *Execute* in the *Simulate* menu.

- Notice that when you set the reset signal to high, both counters were set to 0 and Total was set to 0.
- The bus Total was created in order to read the value of Total (display is hexadecimal by default). The other two bus signals, Entered and Exited reflect the total number of cars that have entered and exited the parking garage. The relationship between the three bus signals is illustrated by the equation: Total = Entered - Exited. To read the value, move the mouse pointer to the bottom of the window (the white region) and click on the left mouse button. This brings up the measuring cursor. Choose the measuring cursor with the left mouse button and move it along the time scale to display the hexadecimal value of the bus signals. For more information on handling buses in Nova, refer to Chapter 6, "Nova," in this *User's Guide*.
- Notice that the signal lot_empty was initially high when Total = "00" (hexadecimal).
- Notice that the signal lot_full becomes high when Total = "1F" (hexadecimal). The counter is designed to wrap around once it counts to its maximum value (1F). Therefore, the counter is triggered by car_enter and wraps around to a "00" after counting to "1F". This, in turn, makes the signal lot_empty go high.

You will now modify the stimulus to see how the design behaves when both car_enter and car_exit signals are asserted at various points in the simulation.

=> Click the left mouse button on the signal car_enter on the left-hand list of all the signal names.

The signal trace should turn blue when the entire signal is selected in this manner. (The trace will become a dashed line on monochrome monitors.)

=> Press the "**0**" key on your keyboard to cause the signal to go low for the selected period.

=> deselect the signal by clicking on the list of all the signal names.

=> Position the cursor on the car_enter trace after reset goes low.

=> Click and hold the left mouse button.

=> Drag the cursor along the trace to the right until you reach the half-way point of the Nova window, then release the mouse button.

=> Select the *Clock* item from the *Edit* menu.

=> When the *Edit Clock* dialog box appears, click on *OK* to accept the default clock period of 10ns and the clock high time of 5ns.

=> Position the cursor on the `car_exit` trace after the fourth pulse on the `car_enter` trace goes low.

=> Click and hold the left mouse button.

=> Drag the cursor along the trace to the right until you reach the end of the Nova window, then release the mouse button.

=> Select the *Clock* item from the *Edit* menu.

=> When the *Edit Clock* dialog box appears, click on *OK* to accept the default clock period of 10ns and the clock high time of 5ns.

=> Select *Execute* in the *Simulate* menu.

The `car_enter` signal should oscillate for the first half of the simulation while the `car_exit` signal remains low. During the second half of the simulation, `car_exit` should oscillate while `car_enter` remains low. This should simulate cars entering and leaving. `Total` should be incremented for the first half of the simulation, and decremented for the second half.

The final simulation screen should look similar to Figure 3-33.

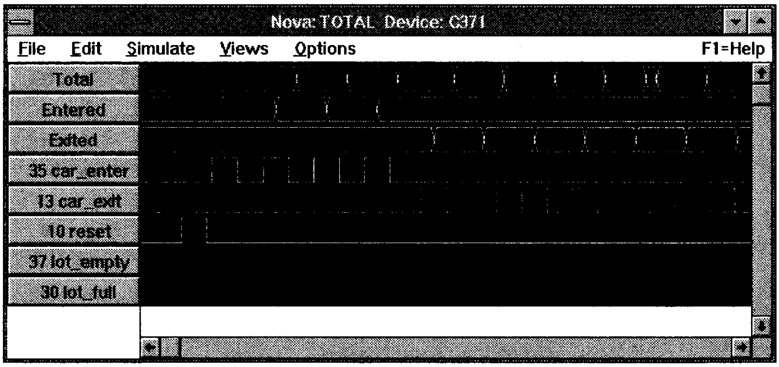


Figure 3-33 Nova simulation for total.jed

3.18.5 Retargeting the Design to an FPGA

Warp also includes support for the pASIC380 family of UltraLogic FPGAs. This section of the parking garage example will retarget the compilation to a 1000 usable gate device, the CY7C381A FPGA. In addition to re-synthesizing the design using the Galaxy interface, you will also need to place and route the design using the SpDE toolkit. This process will take the synthesized description of the VHDL file and place the logic into the FPGA device. The SpDE toolkit also generates all of the files necessary for simulation in the ViewSim tool as well as many other third party simulation environments.

In the following pages, you'll run *Warp* to produce a QDIF file for a specific target device (in this case, a CY7C381A -0JC 1k gate CMOS FPGA).

3.18.6 Starting Galaxy

For *Warp2* Users

=> On Windows systems, double-click on the *Galaxy* icon from the *Warp R4* program group.

=> On UNIX systems, start *Warp2* by typing **`galaxy<CR>`** from within a shell window.

In both systems, the Galaxy window should appear.

For *Warp3* Users

=> On Windows systems, you start *Warp3* by double-clicking on the *Cockpit* icon from the *Warp R4* program group.

=> On UNIX systems, you start *Warp3* by typing **`powerview<CR>`** from within a shell window.

On both platforms, the Cockpit should be displayed immediately after you start *Warp3*. The Cockpit is labeled "Workview PLUS Cockpit" on Windows systems, and "Powerview Cockpit" on UNIX systems.

=> Double-click on the *Warp* icon within the Cockpit to launch Galaxy.

Make sure that *counter.vhd* and *total.vhd* are the two files listed in the Galaxy window. If they are not:

=> Remove other files from the list by clicking on each file name to highlight it and then choosing *Remove* from the *Files* pull-down menu.

=> To add either *counter.vhd* or *total.vhd* to the Galaxy window, click on the *Files* menu and choose the *Add* option.

The ensuing dialog box lists the VHDL files available in the current directory on the left side, and the VHDL files selected for compilation or synthesis on the right side.

=> Highlight the VHDL file in the left-hand list, and double-click on it or choose and click on the "-->" button.

If neither *counter.vhd* or *total.vhd* is available, check to make sure you are in the correct project directory. You may need to close this Galaxy window and locate the **w2tutor** project you created earlier.

3.18.7 Selecting a Device

You must now select a target device and package, set synthesis options, and choose desired output options.

=> Click on the *Device* button from within the *Synthesis options* group.

The *Warp* options dialog box appears.

=> Click on the down-pointing arrow next to the *Device* label to activate the pull-down menu. Scroll down the menu and select the C381A by clicking on it.

3.18.8 Selecting a Package and Speed Bin

In addition to selecting the device you wish to target, you can also choose a package type and speed bin from the list of available packages.

=> Click on the arrow next to the *Package* label to activate the scroll down menu. Scroll down the menu until the desired package and speed bin is visible (for this tutorial, you may choose any package). Select the desired package by clicking on it.

3.18.9 Selecting Other Options in the Device Window

You will use the default Tech Mapping parameters for this compilation.

Notice that the choice of flip-flop type is now grayed-out. This selection is not available for architectural reasons. The flip-flops within a macrocell of a FLASH370 part can be configured to D or T type. The multiplexor-based architecture of Cypress' FPGAs incorporates the inverting functionality of the T flip-flop into the multiplexor logic should your design call for it. Thus a flip-flop selection is not applicable to the pASIC380 FPGA family of devices.

3.18.10 Launching the Retargeted Compilation

=> Click on the *OK* button to dismiss the Device window.

=> Highlight *total.vhd* by clicking on it in the main Galaxy window.

=> Click on the *Set Top* button to set your top-level VHDL file.

=> Click on the *Smart* button in the *Compile* button group to begin compilation.

Warp starts the compilation and synthesis of the design into a CY7C381A and prints messages to keep you appraised of its progress in a pop-up window.

This operation generates two files of particular interest:

- The first is named *total.qdf*. The *.qdf* file is used as an input into the SpDE toolkit. The *.qdf* file is the input file for the SpDE toolkit. SpDE will then place and route the design and can produce a *.lof* file, which can be used to program a device.
- The second file is named *total.rpt*. It contains information about the final synthesized design. You can see the *.rpt* file by pulling down the *Info* menu from the Galaxy window and clicking on *Report*. For more information on what is contained in a report file, refer to Chapter 6, "Report File," in the *Reference Manual.*

=> Close the Galaxy compilation window by clicking on the *Close* button located at the top of the Galaxy compilation window.

Note – If compilation errors occur, do the following: make sure the texts of both files are entered **exactly** as shown earlier in this chapter -- or better yet, copy them from the *c:\warp\examples\wtutor* directory -- and then run *Warp* again.

If error messages appear in the pop-up window:

=> Highlight the error message by clicking on it.

=> Click on the *Error* button (the icon that looks like a magnifying glass) located at the top of the Galaxy compilation window.

This opens the VHDL editor with the cursor on the line number that contains the error. Check the code you have entered against the text shown earlier in this chapter.

3.18.11 Running the Place and Route Tool, SpDE

Now that you have generated the QDIF file for *total.vhd,* you should take the synthesized description of the VHDL file and place the logic into the FPGA device. Once the design has been placed into the FPGA, SpDE, the place and route tool, can then generate the simulation files needed for ViewSim.

For *Warp2* Users

=> Select *SpDE* from the *Tools* pull-down menu in Galaxy.

For *Warp3* Users

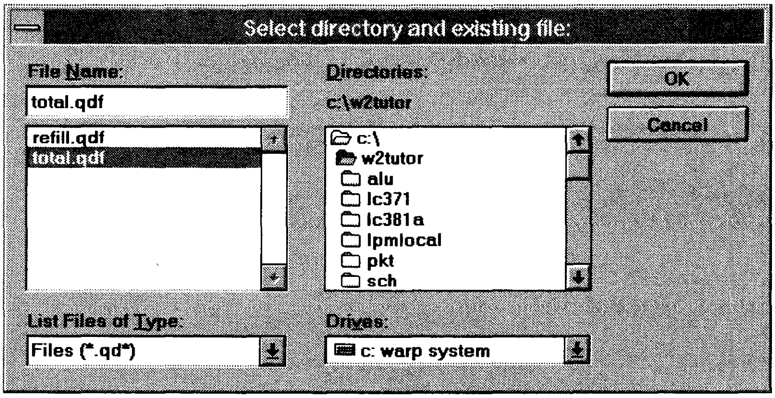
=> Select *SpDE* from the *Tools* pull-down menu in Galaxy.

=> Alternatively, double-click on the *Place&Rte* icon from within the *Cockpit* to launch SpDE.

For *Warp2* and *Warp3* Users

=> Click on the *folder* button located in the upper left portion of the SpDE window. This is equivalent to selecting *File->Import->QDIF.*

An open-file window appears listing your directory structure on the right, and the *.qdf* files on the left.


Select directory and existing file:

File Name: total.qdf

refill.qdf
total.qdf

Directories: c:\w2tutor

c:\
w2tutor
alu
lc371
lc381a
lpmlocal
pkt
sch

List Files of Type: Files (*.qd*)

Drives: c: warp system

Figure 3-34 Choosing total.qdf in SpDE

=> Go to your project directory if you are not already there.

=> Highlight *total.qdf* by clicking on it.

=> Click on the *OK* button to import the file.

Ignore all the warnings that come up. SpDE reports all unused gates when reading a file in. This should not affect the design's functionality.

After importing the file, the *hammer* button will now be selectable (it was not available before loading the QDIF file).

=> Click on the *hammer* button in the SpDE window.

A window will appear containing all the tools you can run. The window appears with all of them selected.

=> Click on the *Run* button to begin the place and route process.

The place and route process will take a couple of minutes to complete. Upon completion, a window will appear stating "All chosen SpDE tools ran successfully."

=> Click on the *OK* button to close the above message window.

=> Select *Full Fit* from the *View* pull-down menu.

This will allow you to see a view of the whole device with your design fitted into it (Figure 3-35). For more information on SpDE, please see Chapter 5, "SpDE," of this *User's Guide.*

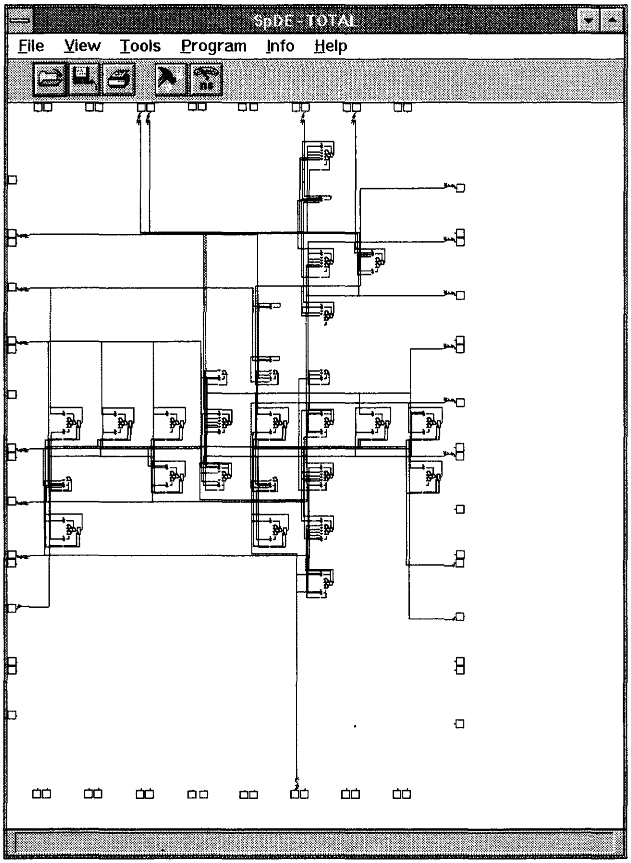


Figure 3-35 SpDE output for total.qdf

=> Select *Save* under the *File* menu to save your design.

=> Select *Exit* from the *File* pull-down menu.

This tutorial ends here for all *Warp2* users. The remainder of this chapter is dedicated to users of the *Warp3* tool set. For *Warp2* users, please continue reading with Chapter 4, "Galaxy," of this *User's Guide*.

3.18.12 Running pASIC->VSim to Generate a ViewSim Model

You now need to generate a ViewSim model from the output of the SpDE toolkit.

=> Double-click on the *pASIC->VSim* icon in the Cockpit.

A dialog box appears, containing a command line to be executed.

=> Make the command line read **`total`**, then click on *OK*.

A window appears, informing you of the progress of the application. When the banner of this window reads "inactive pASIC->VSim," the application is complete.

Note – Verify that your current project directory is set to *c:\w2tutor*. Ignore the messages at the bottom of the window. If the application reports 0 errors and 0 warnings in the text, the application ran successfully. Close the pASIC->VSim window.

You now have the files ViewSim requires to simulate the parking garage design.

3.19 Back-Annotating Pin Assignment Information

You can easily lock-in the pin assignment made by the place and route tool.

=> Double-click on the *Galaxy* icon in the Cockpit.

=> Highlight *total.vhd* by clicking on it. If you don't have *total.vhd* selected, do a *Files->Add* and add *total.vhd*.

=> Choose *Annotate...* from the *Files* pull-down menu.

A small window appears giving the name of the file which will be back-annotated (in our case, *total.vhd*), and giving us the option of back-annotating the pins, the nodes or both. The *Pins* option should already be selected.

=> If *Pins* is not already selected, click on the button to the left of *Pins*.

=> If *Nodes* is selected, deselect it by clicking on the button to the left of *Nodes*.

=> Click on the *OK* button to back-annotate the pin information.

Note – The back-annotation information is stored in a control file. Control files have a *.ctl* extension.

3.20 Simulating the Behavior of the Design with ViewSim

Once the design is synthesized, you should simulate its behavior and evaluate its timing performance to ensure that it functions as intended.

Note – Before performing this step of the tutorial, copy the *total.cmd* file from the *Warp* directory (its default location is *c:\warp\examples\wtutor\total.cmd*) to your project directory.

You begin by launching ViewSim.

=> Double-click on the *ViewSim* icon in the Cockpit.

=> A dialog box appears. Make sure the design name reads **`total`**.

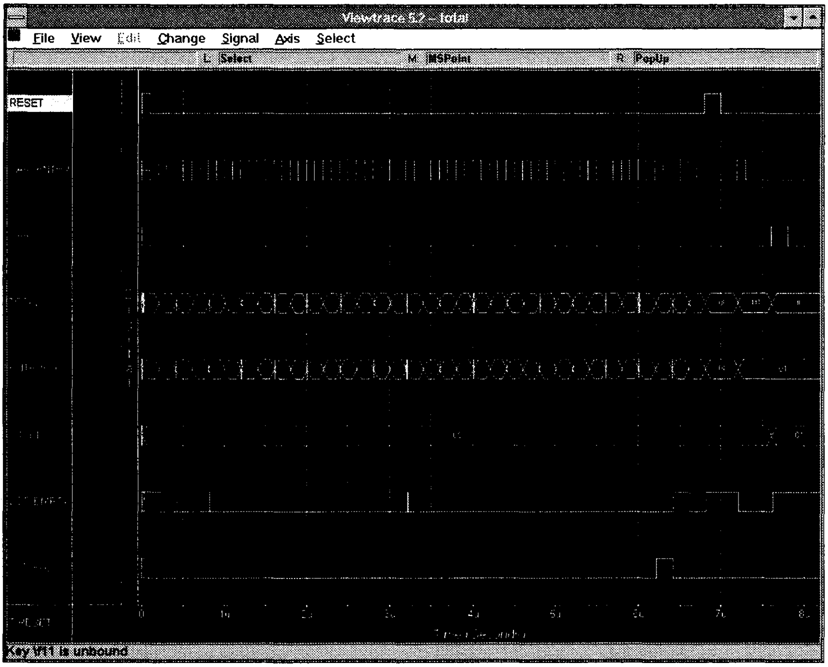
=> Click on *OK*.

ViewSim starts up. When the ViewSim window appears ('SIM>'):

=> Type **`total`** at the command line.

The *total.cmd* file runs, executing the following sequence of ViewSim commands (not necessarily echoed to the screen):

```
restart
vector total total_[4:0]
vector entered count1_[4:0]
vector exited count2_[4:0]
stepsize 100ns

wave total.wfm reset car_enter car_exit total entered exited
   lot_empty lot_full

h reset
cycle
l reset car_enter car_exit
cycle

clock car_enter 1 0
cycle 33

h reset
l car_enter
cycle

l reset car_enter
cycle

h car_enter
cycle

l car_enter
cycle

l car_enter
h car_exit
cycle

l car_enter car_exit
cycle

l car_enter
cycle
```



Key Vf11 is unbound

Figure 3-36 ViewTrace output window for total.vhd

The following observations can be made from the ViewTrace window:

- The vector command in *total.cmd* advises ViewSim to create a bus output for the signals `total`, `count1` and `count2`. The signals `count1` and `count2` are renamed to `entered` and `exited` respectively.
- Notice, the signal `lot_empty` was initially high and `lot_full` was initally low when `total` = "00" (hexadecimal). You need to zoom in to look at this portion of the ViewTrace window. Use "**F9**" to zoom in and drag the mouse over the area you want to zoom over. A red color is spread over the portion chosen. Click on "**F3**" to select the area and zoom in. Refer to the ViewTrace User's Guide in the Design Entry and Digital Simulation Solutions collection within the Viewlogic on-line documentation set to get more details on using ViewTrace.
- The signal `car_enter` is clocked thrity three times by using the command **`cycle 33`**. This increments the value of the bus signal `entered` 33 times.

- Notice, the signal lot_full becomes high when total = "1F". Once again, you need to zoom in to look at this portion of the ViewTrace window. The counter is designed to wrap around, once it counts to its maximum value (1F). Therefore, the counter which is triggered by car_enter, wraps around to a "00" after counting to "1F". This in turn makes the signal lot_empty go high.
- The reset signal is pulled high to reset the values of total, entered and exited to 0.
- Notice, when the signal car_exit goes high, the value of total is decremented by 1. In this simulation, the signal total changes from "01" to "00" and the signal lot_empty becomes high again for the rest of the simulation.
- Remember: total = entered - exited.
- Notice that signal lot_empty has three glitches in this simulation. These glitches occur when total changes values from "01" to "02", "03" to "04", and "0F" to "10". Signal lot_empty has these glitches because lot_empty is purely combinatorial and the inputs to this signal arrive at slightly different times. Take for example the first case, "01" to "02." In binary, this translates to "01" to "10". If the rightmost bit changes to a '0' before the leftmost bit changes to a '1', the value of the counter will have "00" for a brief moment in time. This causes the lot_empty signal to become active and thus glitch. These glitches are intentionally left as they are to give the user a flavor of the real time simulation output that ViewSim provides. Also, the duration and occurence of these glitches will vary with different devices. Users can eliminate these glitches by making the signal lot_empty to be a registered signal. This is left as an exercise to the user.

This concludes the parking garage tutorial. The next exercise is completely for *Warp3* users only since it covers schematic entry and mixed-mode entry within the Viewlogic, Workview PLUS, and Powerview environments.

3.21 Designing an ALU Circuit

For *Warp3* Users Only

This tutorial exercise can only be done using *Warp3* because it requires interaction with the Viewlogic software provided with *Warp3*.

Overview

This exercise takes you step-by-step through the design of a 4-bit ALU. The design is done in two stages. The first stage will be the construction of the ALU's core logic which you will do in a schematic using LPM components. The second stage is constructing the control logic. You will create the control logic using VHDL and then combine this VHDL with the previously captured schematic design. When you complete this tutorial, you will know how to:

- create a schematic for the ALU circuit, instantiate and position LPM components, label input and output ports, wire components together, and save the schematic.
- create a symbol for the ALU circuit schematic to allow it to be instantiated as a component in a higher level schematic design.
- write an entity declaration, architecture, and package declaration for a VHDL description of a simple ALU controller circuit.
- create a symbol for the VHDL description to allow it to be instantiated as a component in a schematic design.

3.22 Starting Warp3

=> On Windows systems, you start *Warp3* by double-clicking on the *Cockpit* icon from the *Warp R4* program group.

=> On UNIX workstations, type **powerview<CR>** on the command line of a shell tool.

The Cockpit is the access point for all *Warp3* tools. To start a tool, double-click on the tool's icon from within the Cockpit window.

The Cockpit is organized into "toolboxes," which are themselves organized into "drawers." At any time, the Cockpit will display the tools available in the current drawer of the current toolbox. By default, the current toolbox is named *Cypress,* and the current drawer is named *Warp Design.*

To change the current toolbox or drawer, click on the down-arrow next to the *Current Toolbox* or *Current Drawer* label, then select a toolbox or drawer from the available ones listed.

3.23 Creating a Project

You must set the project directory before running any *Warp3* tools. *Warp3* stores all generated files and creates sub-directories in the project directory.

To create a new directory and set the project directory to it:

=> Select *Create* from the *Project* pull-down menu in the Cockpit menu bar.

=> When a dialog box appears, type a complete pathname, then click *OK.*

The named directory is created if it doesn't already exist and becomes the current project directory.

To set the project directory to an existing project:

=> Select *Set Current* from the *Project* pull-down menu in the Cockpit menu bar.

A dialog box appears, listing current projects.

=> Select one, then click *OK.*

You will create a new project directory which you will use for all the files created in this tutorial.

For the sake of this tutorial, set the project directory to **`c:\w2tutor\alu`**.

On a Windows system:

=> Select *Create* from the *Project* pull-down menu.

=> Type **`c:\w2tutor\alu`**, then click *OK.*

On UNIX:

=> From the Cockpit, select *Create* from the *Project* pull-down menu.

=> Type **`<user home directory path>/w2tutor/alu`**, then click *OK.*

3.24 Creating the Schematic

In this tutorial, you'll use ViewDraw to create a schematic called PLD. You'll also use ViewText® to write the VHDL file for the ALU controller circuit that you'll call *cntrl.vhd.*

3.24.1 Starting ViewDraw

=> Double-click on the *ViewDraw* icon from the Cockpit.

=> Then click *OK* in the ensuing dialog box.

A message appears, informing you of ViewDraw's progress in loading the various modules (it takes a few seconds to load the modules).

An Alert window will appear stating:

```
Warning-I:Library 'lpmlocal' not found in
viewdraw.ini

Cypress LPM library has been disabled.

Open ViewDraw and Click Menu Cypress=>Initialize
LPM.

Exit ViewDraw and then Re-Open it to enable the
LPM library.
```

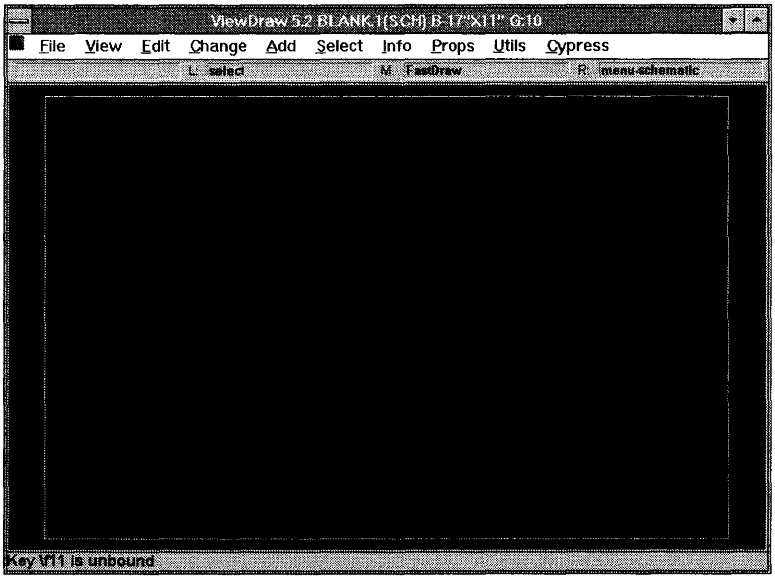
This is a **normal** informational message when creating a new project.

The ViewDraw *File->Open* dialog box appears.

=> Type **`pld`** on the *Enter name:* line in the dialog box.

=> Click *OK*.

=> A new, blank schematic sheet appears (Figure 3-37).



Key \F11 is unbound

Figure 3-37 Blank ViewDraw schematic sheet

=> From the *Cypress* pull-down menu in the ViewDraw window, select *Initialize LPM.*

A dialog box appears, prompting you to enter a directory path.

=> On a Windows system, enter **`c:\w2tutor\alu\lpmlocal`** if this is not already entered.

=> On a UNIX system, enter **`<user home directory path>/w2tutor/alu/lpmlocal`** if this is not already entered.

=> Click on the *OK* button to initialize the LPM library.

A message should appear in the bottom, left corner of the ViewDraw window stating:

```
Done with LPM Initialization!
```

=> Select *Quit viewdraw* from the *Red Square* pull-down menu, and click on the *OK* button.

Now that you have initialized the LPM library, you need to restart ViewDraw and open your PLD schematic window.

=> Double-click on the *ViewDraw* icon from the Cockpit.

=> On a Windows system, click *OK* in the Command Line Options dialog box.

=> Type **`pld`** in the *Enter name:* field.

=> Click on the *OK* button to open your PLD schematic window.

3.24.2 Instantiating the LPM Components

In this section, you will choose various components from the LPM Library to form the functionality of a 4-bit ALU circuit.

In the following pages, you will:

- call up the LPM components, select the right size and options, and place them onto the schematic sheet
- position the components relative to each other, in preparation for wiring
- wire the components together
- label input and output ports
- save the finished schematic

Note – It is a good practice to save your schematics whenever possible by typing "**w**" in your schematic sheet.

The process of calling up a component and placing it on a schematic sheet is called "instantiating" a component. In this section of the tutorial, you'll instantiate the LPM components required for this circuit.

Components to be instantiated include: **Madd_sub, Mcompare, Mand, Mor, Mxor, Minv, Mmux, Gnd, IN** and **OUT**.

To instantiate a component:

=> Select *Add->LPM Symbol* from the ViewDraw menu bar.

The *Warp* LPM *Add Cell* dialog box appears (Figure 3-38).

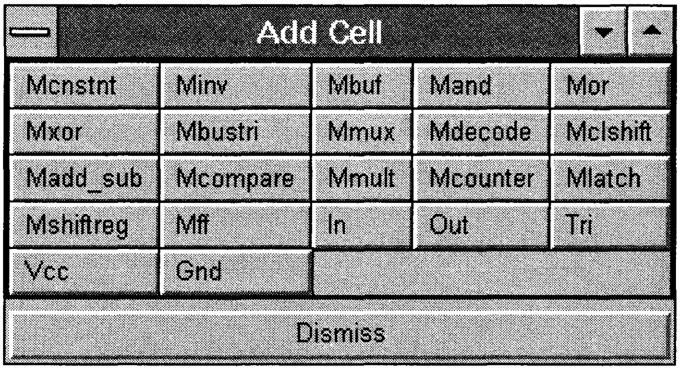


Figure 3-38 The Warp Add Cell dialog box

The *Add Cell* dialog box shows the 22 different LPM components supported. Do not dismiss this dialog box until you are completely finished instantiating all of the necessary components. Please refer to Chapter 5, "LPM Reference," of the *Reference Manual* for details on each of the LPM components.

=> Find the component you want to instantiate by clicking on a component name with the left mouse button. This brings up the dialog box which has all the options that can be set or changed for the LPM component chosen.

=> After you have first chosen the individual component and then chosen its desired options, click on the *Accept* button.

=> Move the cursor to somewhere (anywhere) in the schematic. If you have a 3-button mouse, click the middle mouse button to place an instance of the component in that spot. If you have a 2-button mouse, move the cursor to anywhere in the schematic, click your left mouse button and press "**F3**" to place an instance of the component. Move the cursor and click the middle mouse button to place another instance of the component. Repeat until you have instantiated the component as often as necessary.

Repeat the above process to select all of the required components and place them on the schematic sheet.

=> After selecting all of the components, click on *Cancel* to close the *Add Cell* dialog box.

Below is a step-by-step introduction to creating the PLD schematic with LPM components using the general procedure described above.

3.24.3 Detailed Description of Creating the PLD Schematic

A detailed description of the process of selecting the components for the PLD schematic with the desired options is presented here. The discussion assumes that the user has a 3-button mouse. If you have a 2-button mouse, while placing a selected component:

=> Click with the left-mouse button.

=> Press "**F3**."

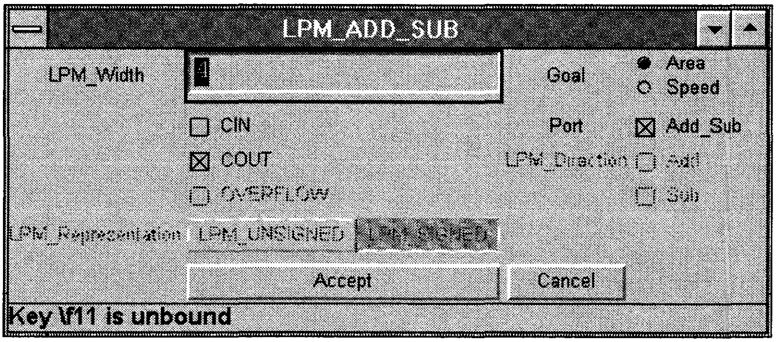
Don't worry about precise placement; you will re-position each instance later. For now, just get instances onto the schematic sheet.

Madd_sub

This is the very first LPM component that you will add to the PLD schematic. This component can be configured to act as an adder/subtractor.

=> Click on the *Madd_sub* component button.

A dialog box (Figure 3-39) will appear which presents you with several options for the adder/subtractor component. You need to specify that you will be operating on 4-bit-wide buses, and you need to select the carry out (*cout*) option.


LPM_ADD_SUB

LPM_Width 4

Goal ● Area ○ Speed

□ CIN

☒ COUT

□ OVERFLOW

Port ☒ Add_Sub

LPM_Direction □ Add □ Sub

LPM_Representation LPM_UNSIGNED LPM_SIGNED

Key \f11 is unbound

Figure 3-39 Dialog box for the adder/subtractor component

=> To specify 4-bit-wide buses, you enter a **4** in the width field, *LPM_Width.*

=> To select the carry out, click on the *cout* check box so that an "x"appears in the check box.

=> Select **area** optimization for this model.

All the other components selected in this exercise, except *Mcompare* do not have different implementations for speed versus area, so that the selection between speed/area is grayed out.

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Place the adder-subtractor near the top as seen in the picture of the final PLD schematic (Figure 3-50).

=> To place the component in the schematic, use either the middle mouse button (3-button mouse) or a combination of left mouse button and "**F3**" (2-button mouse).

A picture of this element is shown in Figure 3-40.

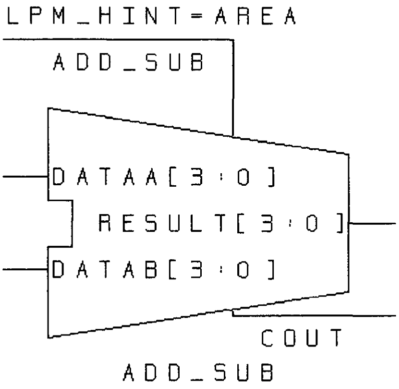


Figure 3-40 Madd_sub component

Mcompare

This component provides six outputs. This component performs equality compares and magnitude compares.

=> Select *Add->LPM Symbol* from the ViewDraw menu bar if the LPM component dialog box is no longer on your screen.

=> Click on the *Mcompare* component button.

The dialog box which appears will be different from the dialog box that was presented to you for the Madd_sub component. You still must specify the bus width to be four bits wide.

=> Enter a **4** in the width field, *LPM_Width.*

Since this component is a comparator, you can choose to include up to six output signals: "agb" (A greater than B), "ageb" (A greater than or equal to B), "aeb" (A equal to B), "aneb" (A not equal to B), "alb" (A less than B), and "aleb" (A less than or equal to B), where A and B are your two inputs to the Mcompare component. You will be using all of these outputs from Mcompare.

=> If they are not already selected, click on the check boxes to the left of the labels `alb`, `agb`, `aeb`, `aneb`, `ageb`, and `aleb` to select them.

=> Select **speed** optimization for this model (this is the default).

=> Place the comparator near the top as seen in the picture of the final PLD schematic (Figure 3-50).

=> To place the component in the schematic, use the middle mouse button or press the left mouse button and"**F3**."

A picture of this element is shown in Figure 3-41.

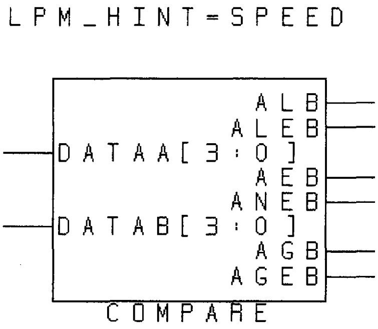


Figure 3-41 Mcompare component

Mand

The next component you will add is an AND gate.

=> Select *Add->LPM Symbol* from the ViewDraw menu bar if the LPM component dialog box is no longer on your screen.

=> Click on the *Mand* component button.

The dialog box that appears contains two fields: one to specify the width of the signals (*LPM_Width*) and the other to specify the number of signals to be ANDed together(*LPM_Size*).

=> Enter **4** in the *LPM_Width* field and **2** in the *LPM_Size* field.

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Place the AND gate as shown in the picture of the final PLD schematic (Figure 3-50) by clicking the middle mouse button or the left mouse button and "**F3**."

A picture of this element is shown in Figure 3-42.

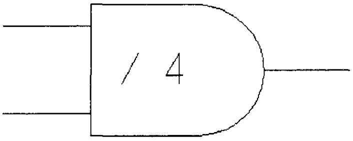


Figure 3-42 Mand component

Mor

The next component you will add is an OR gate.

=> Select *Add->LPM Symbol* from the ViewDraw menu bar if the LPM component dialog box is no longer on your screen.

=> Click on the *Mor* component button.

The dialog box that appears contains two fields: one to specify the width of the signals (*LPM_Width*) and the other to specify the number of signals to be ORed together(*LPM_Size*).

=> Enter **4** in the *LPM_Width* field and **2** in the *LPM_Size* field.

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Place the OR gate as shown in the picture of the final PLD schematic (Figure 3-50) by clicking the middle mouse button or the left mouse button and "**F3**."

A picture of this element is shown in Figure 3-43.

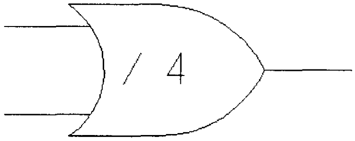


Figure 3-43 Mor component

Mxor

The next component you will add is an XOR gate.

=> Select *Add->LPM Symbol* from the ViewDraw menu bar if the LPM component dialog box is no longer on your screen.

=> Click on the *Mxor* component button.

The dialog box that appears contains two fields: one to specify the width of the signals (*LPM_Width*) and the other to specify the number of signals to be XORed together (*LPM_Size*).

=> Enter **4** in the *LPM_Width* field, and **2** in the *LPM_Size* field.

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Place the XOR gate as shown in the picture of the final PLD schematic (Figure 3-50) by clicking on the middle mouse button or on the left mouse button and "**F3**."

A picture of this element is shown in Figure 3-44.

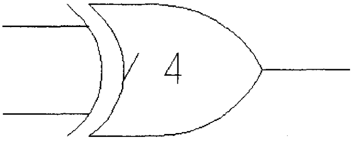


Figure 3-44 Mxor component

Minv

The next component you will add is an inverter. You need two instantiations of this component to invert the values on the two input buses, A and B.

=> Select *Add->LPM Symbol* from the ViewDraw menu bar if the LPM component dialog box is no longer on your screen.

=> Click on the *Minv* component button.

The dialog box that appears contains only one input field: the bus-width of the signal to be inverted (*LPM_Width*).

=> Enter **4** in *LPM_Width.*

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Press the middle mouse button or the left mouse button and "**F3**" to place the component in the schematic.

=> Repeat the above 4 steps to instantiate one more 4-bit-wide inverter -- or select the 4-bit-wide inverter using the left mouse button and press "**c**" on your keyboard to copy the component.

=> Again, press the middle mouse button or the left mouse button and "**F3**" to place the component in the schematic.

=> Place the INV gates as shown in the picture of the final PLD schematic (Figure 3-50).

A picture of this element is shown in Figure 3-45.

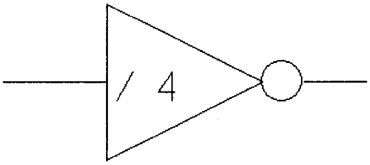


Figure 3-45 Minv component

Mmux

The next component you will add is a Multiplexor.

=> Select *Add->LPM Symbol* from the ViewDraw menu bar if the LPM component dialog box is no longer on your screen.

=> Click on the *Mmux* component.

The dialog box which appears contains two parameters: one specifies the bus width of the muxed signals and the mux's output signal (*LPM_Width*) and the other specifies the number of selection lines needed (*Number of Sel*).

=> Enter **4** for the bus width parameter and **3** for the number of selection lines.

Notice that after you enter **3** in the parameter for the number of select lines, an "8" will appear in the field to the right (*LPM_Size*). This field tells you how many signals can be multiplexed. Having three select lines means that you can choose between one of the eight inputs. In general, a mux will be a 1-of-2^n mux where n is the number of select lines. With this component, you have created an 8-to-1 mux with 3 select lines, and all the inputs to the mux are 4-bits wide.

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Place the MUX as shown in the picture of the final PLD schematic (Figure 3-50).

A picture of this element is shown in Figure 3-46.

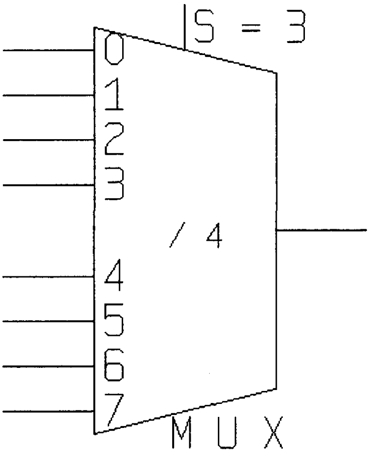


Figure 3-46 Mmux component

Gnd

You need to instantiate an LPM ground component called Gnd. You will use this ground component later, when you wire the components together.

=> Select *Add->LPM Symbol* from the ViewDraw menu bar if the LPM component dialog box is no longer on your screen.

=> Click on the *Gnd* component button.

The dialog box that appears contains a parameter to specify the bus width of the ground you wish to instantiate.

=> Enter **2** in the width field, *LPM_Width.*

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Place the GND gate as shown in the picture of the final PLD schematic by clicking on the middle mouse button or on the left mouse button and "**F3**."

A picture of this element is shown in Figure 3-47.

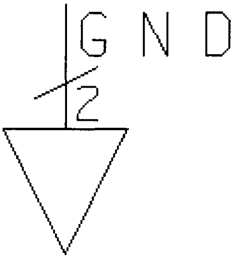


Figure 3-47 Gnd component

In

You now need to instantiate two 4-bit-wide input pins for the operands, A and B.

=> Select *Add->LPM Symbol* from the ViewDraw menu bar if the LPM component dialog box is no longer on your screen.

=> Click on the *IN* component button.

The dialog box that appears contains only one input parameter: the bus-width of the signal.

=> Enter **4** for this parameter.

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Press the middle mouse button or the left mouse button and "**F3**" to place the component in the schematic.

=> Repeat the above three steps to instantiate one more 4-bit-wide input pin--or, select the 4-bit-wide input pin using the left mouse button and press "**c**" on your keyboard to copy the component.

=> Again, press the middle mouse button or the left mouse button and "**F3**" to place the component in the schematic.

A picture of this element is shown in Figure 3-48.

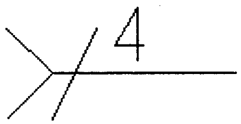


Figure 3-48
In component

You also need a single-bit input to provide an input to the Madd_sub which determines if the performed operation is an addition or a subtraction.

=> Click on the *IN* component button.

The dialog box that appears contains only one input parameter:
the bus-width of the signal.

=> Enter **1** for this parameter.

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Press the middle mouse button or the left mouse button and "**F3**" to place the component in the schematic.

You also need one 3-bit input to feed the 3 select lines of the 8-to-1 mux.

=> Click on the *IN* component button.

The dialog box that appears contains only one input parameter:
the bus-width of the signal.

=> Enter **3** for this parameter.

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Place these IN ports as shown in the picture of the final PLD schematic (Figure 3-50) by clicking on the middle mouse button or on the left mouse button and "**F3**."

Out

You now need to instantiate a 4-bit-wide output pin for the schematic.

=> Select *Add->LPM Symbol* from the ViewDraw menu bar if the LPM component dialog box is no longer on your screen.

=> Click on the *OUT* component button.

The dialog box that appears contains only one input parameter: the bus-width of the signal.

=> Enter **4** for this parameter.

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Press the middle mouse button or the left mouse button and "**F3**" to place the component in the schematic.

A picture of this element is shown in Figure 3-49.

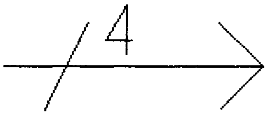


Figure 3-49
Out component

You also need a single-bit output to hold the carry-out value from the Madd_sub.

=> Click on the *OUT* component button.

The dialog box that appears contains only one input parameter: the bus-width of the signal.

=> Enter **1** for this parameter.

=> Click on the *Accept* button and move the mouse pointer into the schematic window.

=> Place these OUT ports as shown in the picture of the final PLD schematic (Figure 4-50) by clicking on the middle mouse button or on the left mouse button and "**F3**."

With these components in your schematic sheet, you have all the components necessary to build the desired 4-bit ALU.

When you are finished, the sheet should look something like Figure 3-50.

=> At this point, you can close the *Add cell* dialog box by clicking on *Dismiss.*

3.24.4 Positioning Components

Once the components are instantiated on the schematic sheet, you will want to position them to enhance the readability and ease-of-wiring of the schematic.

You'll want to position the input ports on the left side of the schematic, the various components just to the right of the input ports, and the output ports on the right of the schematic.

=> Select the component using the left mouse button.

=> Press "**m**" on the keyboard, or select *Move* from the *Edit* pull-down menu.

=> Move the cursor to where you want the component to be.

=> Click on the middle mouse button (for a 3-button mouse) or click on the left mouse button and press "**F3**" (for a 2-button mouse).

=> Repeat until all components are positioned as shown in Figure 3-50.

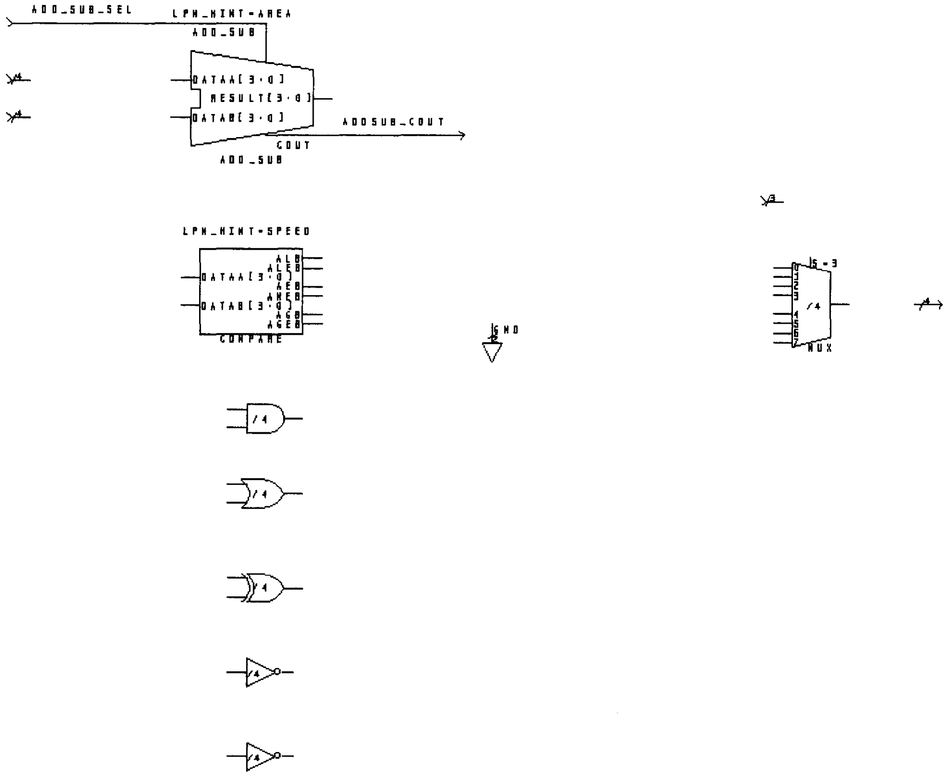


Figure 3-50 Positioned components for PLD schematic design

3.24.5 Wiring Components Together

Once components are positioned, you must wire them together to make the circuit.

The final circuit should look like Figure 3-51.

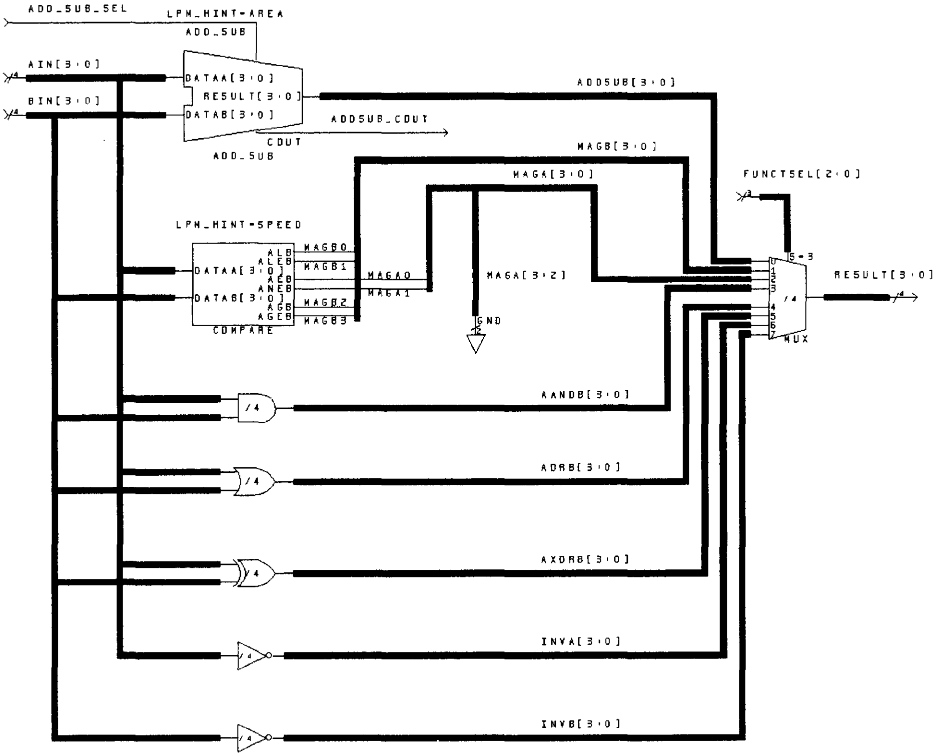


Figure 3-51 Final circuit of PLD design

Connecting Two Components

=> Move the cursor to the origin of the net.

=> To create a net: Select *Net* from the *Add* pull-down menu in the ViewDraw menu bar, or type "**n**" at the keyboard.

=> To create a bus: Select *Bus* from the *Add* pull-down menu in the ViewDraw menu bar, or type "`b`" at the keyboard.

=> Specify points along the net/bus by clicking the middle mouse button ("`F3`" if you have a two button mouse).

=> Click the left mouse button once to back up one segment on the net or bus, twice to back up two segments, etc.

=> To connect the net/bus to a component pin, move the cursor to a point on the pin and click the middle mouse button ("`F3`" on a 2-button mouse).

=> To connect the net/bus to another net or bus, move the cursor to a point on the net or bus and click the middle mouse button ("`F3`" on a 2-button mouse).

=> To leave the net/bus dangling, form the net/bus and click the middle and then the right mouse buttons ("`F3`" and then the right mouse button on a 2-button mouse).

Note – Make sure to connect all input pins on components to something. ViewDraw does not allow unconnected inputs.

- If you want to delete a net or a bus, select the net or the bus and type "`d`" to delete it.
- If you want to undo anything that you did to your schematic, type in "`u`."

Note – It is a good practice to save your schematics whenever possible by typing "`w`" in your schematic sheet.

For the benefit of users who have not used ViewDraw before, Section 3.24.7 below, entitled "Detailed Description of Instantiating, Labeling, and Wiring," will walk you through individual steps involved in creating the final PLD schematic.

3.24.6 Labeling Nets and Buses

Once components are positioned on the schematic, you need to label everything.

Labeling the Component Ports

=> Select the net/bus, using the left mouse button.

=> Select *Label* from the *Add* pull down menu the ViewDraw menu bar.

=> Click on the *label* line of the dialog box.

=> Delete the contents of the *label* line, if any.

=> Type the new label, e.g., `clk`. Case doesn't matter; all characters will be upper-case on the schematic.

=> For buses, you also need to specify the bus width along with its label, e.g., `data[7:0]`.

=> Click *OK*.

=> Move the cursor to where you want the label to appear (i.e., next to the port it's associated with).

=> Click the middle mouse button to place the label.

=> Repeat until all buses and nets are labeled as in Figure 3-51.

Hint – Here's a shortcut method for adding several labels to input and output ports of a component at once:

- Select a port.
- Select *Label* from the *Add* menu.
- Type the names of the ports to be labeled as one comma-separated list. DO NOT insert spaces after the commas. The string should look like: `label1,label2,label3,...,...,...`
- Click on the *OK* button. The first label in the list appears.
- Position the label, then click the middle mouse button.
- Click the left mouse button to select the next item to be labeled.
- Press the "`l`" key on your keyboard to activate the next label.
- Repeat the last three steps for each port to be labeled.

For the benefit of people who have not used ViewDraw before, the section below, "Detailed Description of Instantiating, Labeling, and Wiring," will walk you through individual steps involved in creating the final schematic.

Note – It is a good practice to save your schematics whenever possible by typing "**w**" in your schematic sheet.

3.24.7 Detailed Description of Instantiating, Labeling, and Wiring

=> Using a bus connection, connect the output of the adder-subtractor to input 0 of the 8-to-1 mux. Click on a blank area of the schematic with your left mouse button to deselect everything, and then click on the piece of the bus you just drew. Since all nets and buses must have labels, you must now give a label to the bus you just drew. Press "`l`" on the keyboard, or choose *Label* from the *Add* pull-down menu, and type in **`addsub[3:0]`**. This names the bus "addsub[3:0]."

=> Since you're using 4-bit buses, you'll need to use two bus connections in order to have all six of the comparator outputs available for selection by the mux. Beginning at input 1 of the mux, draw a bus in an L shape (see the zoomed-in view of the schematic in Figure 3-52) and terminate it at an end point by pressing the middle mouse button and then the right mouse button, or by pressing "**`F3`**" and then the right mouse button. Repeat this free-floating connection beginning at input 2 of the mux and ending, in an L shape, near the comparators' outputs. Click on a blank area of the schematic to deselect everything already selected. You again need to label the two buses you just drew. Clicking on a piece of the bus connected to mux input 1, press "`l`" on the keyboard, or choose *Label* from the *Add* pull-down menu, and type in **`magb[3:0]`**. Again, click on a blank area of the schematic to deselect everything already selected. Clicking on a piece of the bus connected to mux input 2, press "`l`" on the keyboard, or choose *Label* from the *Add* pull-down menu, and type in **`maga[3:0]`**.

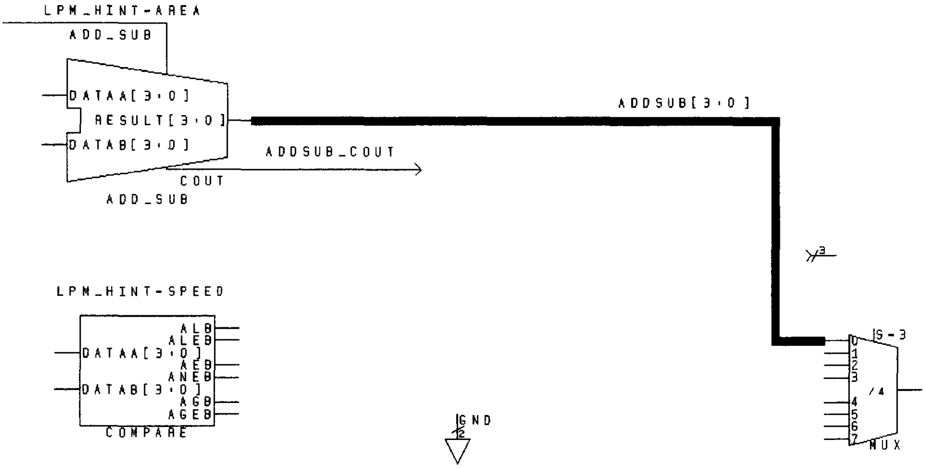


Figure 3-52 Connecting the mux to the comparator

=> You now need to connect the single-bit nets which comprise the comparators' outputs to the two buses you just drew. Connect the ALB output of the comparator to the **magb[3:0]** bus. This is done in the same way as connecting two pins together. Position the mouse pointer so that it is over the ALB output pin of the comparator, and press "**n**" on the keyboard. Move the mouse pointer over the **magb[3:0]** bus and press the middle mouse button, or "**F3**" on the keyboard, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the net you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **magb0**. By naming the net **magb0**, you have assigned this net to be bit 0 of the 4-bit-wide **magb** bus. You must repeat these steps for the ALEB, AGB, and AGEB outputs of the comparator.

=> Position the mouse pointer over the ALEB output pin of the comparator, and press "**n**" on the keyboard. Move the mouse pointer so that it is over the **magb[3:0]** bus and press the middle mouse button, or "**F3**" on the keyboard, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the net you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **magb1**.

=> Position the mouse pointer over the AGB output pin of the comparator, and press "**n**" on the keyboard. Move your mouse pointer so that it is over the **magb[3:0]** bus and press the middle mouse button, or "**F3**" on the keyboard, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the net you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **magb2**.

=> Position the mouse pointer over the AGEB output pin of the comparator, and press "**n**" on the keyboard. Move your mouse pointer so that it is over the **magb[3:0]** bus and press the middle mouse button, or "**F3**" on the keyboard, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the net you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **magb3**.

=> Having made the connections to the bus **magb[3:0]**, you now need to turn our attention to **maga[3:0]**. Connect the AEB output of the comparator to the **maga[3:0]** bus. Position the mouse pointer so that it is over the AEB output pin of the comparator, and press "**n**" on the keyboard. Move the mouse pointer over the **maga[3:0]** bus and press the middle mouse button, or "**F3**" on the keyboard, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the net you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **maga0**. By naming the net **maga0**, you have assigned this net to be bit 0 of the 4-bit-wide **maga** bus. You must repeat these steps for the ANEB output of the comparator.

=> Position the mouse pointer over the ANEB output pin of the comparator, and press "**n**" on the keyboard. Move your mouse pointer so that it is over the **maga[3:0]** bus and press the middle mouse button, or "**F3**" on the keyboard, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the net you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **maga1**.

=> You have connected the two remaining outputs of the comparator, but the **maga[3:0]** bus still has two unused bits. You will connect these last two bits to ground. The ground component should be placed close to the **maga[3:0]** bus, since that is where you will use it. You make the connection between the ground component and the **maga[3:0]** bus by positioning the mouse pointer over the pin of the ground component.

Then, press "`b`" on the keyboard. Move the mouse pointer so that it is over the **`maga[3:0]`** bus and press the middle mouse button, or "**`F3`**" on the keyboard, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the bus you just drew and press "`l`" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **`maga[3:2]`**. This assigns the two most significant bits of the **`maga`** bus to ground.

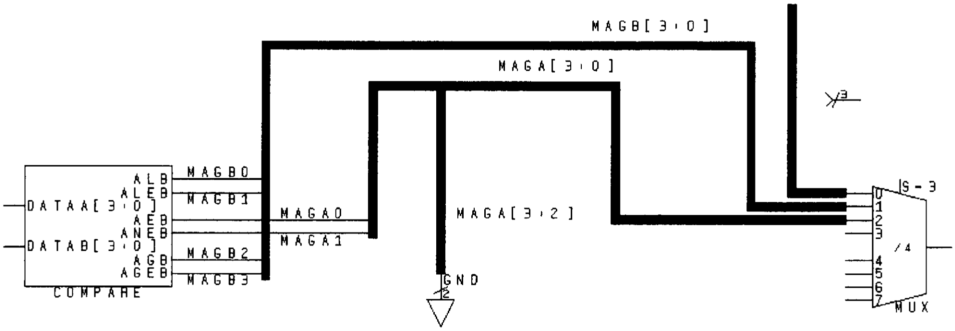


Figure 3-53 Connection the comparators to the mux

=> You will now make the five remaining connections to the 8-to-1 mux (Figure 3-54). Using a bus connection, connect the output of the AND gate to input 3 of the 8-to-1 mux. Click on a blank area of the schematic to deselect everything, and then click on a piece of the bus you just drew. Press "`l`" on the keyboard, or choose *Label* from the *Add* pull-down menu, and type in **`AandB[3:0]`**. Connect the output of the OR gate, XOR gate, and the two inverters in the same way. The output of the OR gate should be connected to input 4 of the 8-to-1 mux; the output of the XOR gate should be connected to input 5 of the 8-to-1 mux; the output of the first inverter gate, the inverse of A, should be connected to input 6 of the 8-to-1 mux; finally, the output of the second inverter gate, the inverse of B, should be connected to input 7 of the 8-to-1 mux.

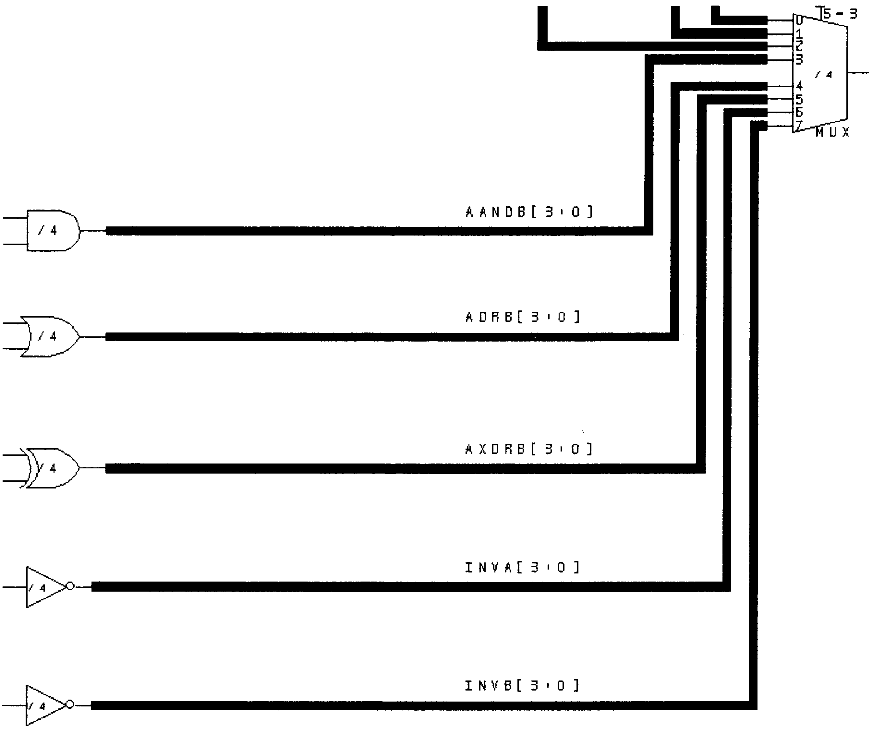


Figure 3-54 Connecting the mux to AND, OR, XOR, and inverters

=> The next components you will connect are the 4-bit-wide input pins for the operands, A and B. Position the mouse pointer over the input pin which will be used for operand A, and then press "**b**" on the keyboard. Make the bus long enough so that you will be able to connect the input of the inverter gate to this bus. Click the middle mouse button and then the right mouse button, or, if you're using a 2-button mouse, press "**F3**" and then the right mouse button, to make the free-floating connection. Now position the mouse pointer over the input pin which will be used for operand B, and press "**b**" on the keyboard. Make the bus long enough so that you will be able to connect the input of the inverter gate to this bus also. Click the middle mouse button and then the right mouse button, or, if you're using a 2-button mouse, press "**F3**" and then the right mouse button, to make the free-floating connection.

=> The convention used in this tutorial is to assign the A input to the upper input, and the B input to the lower of the two inputs to the components instantiated. Position the mouse pointer over the A, upper, input of the adder-subtractor. Press "**b**" on the keyboard and move the mouse pointer such that it is on top of the bus you just drew for the A operand input. Click the middle mouse button, or press "**F3**," to make the connection between the upper input of the adder-subtractor and the A-input bus. Now you must connect the lower input of the adder-subtractor to the B-input bus. Position the mouse pointer over B, the lower input of the adder-subtractor. Press "**b**" on the keyboard and move the mouse pointer so that it is on top of the bus you just drew for the B operand input. Click the middle mouse button, or press "**F3**," to make the connection between the lower input of the adder-subtractor and the B-input bus.

=> Repeat these steps for each of the remaining components: comparator, AND gate, OR gate, and XOR gate. The inverters have only one input each. Therefore, one of the inverters should have its input connected to the operand A bus, and the other should have its input connected to the operand B bus. Look at Figure 3-55 for a zoomed-in view of what the A and B input buses should look like after making the connections. If you made the A and B input buses too long, so that now they have pieces beyond the inputs to the inverters, click on the overhanging piece. Press "**d**" on the keyboard, or choose *Delete* from the *Edit* pull-down menu, to delete these unused pieces. Repeat this step for the other bus if necessary.

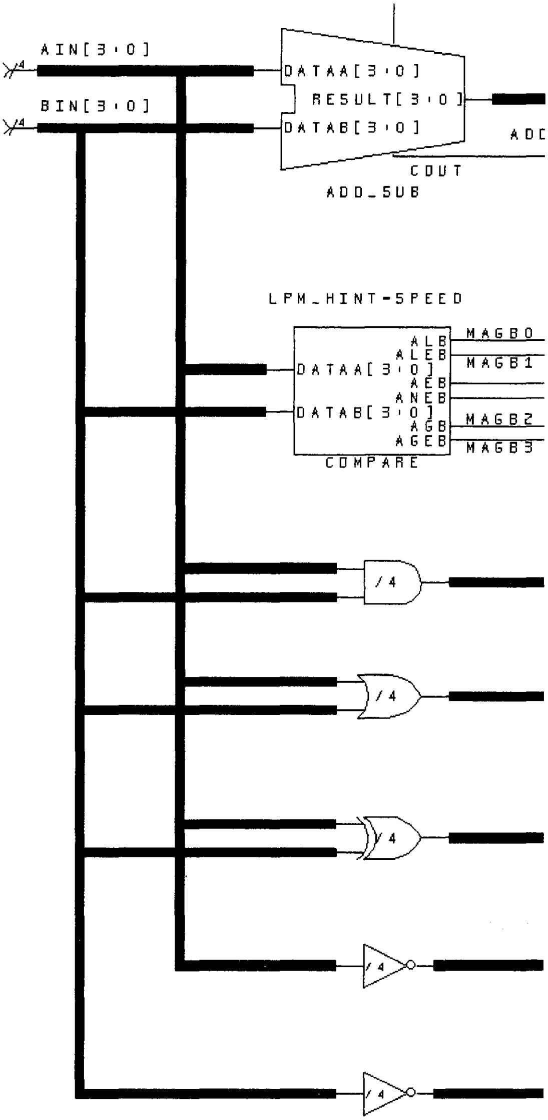


Figure 3-55 Connecting the A and B input buses

=> You'll connect the 3-bit input port to the select lines of the mux with a bus and label the bus now. Position the mouse pointer over the input pin component, and press "**b**" on the keyboard. Move your mouse pointer so that it is over the select lines of the 8-to-1 mux, and press the middle mouse button, or "**F3**" on the keyboard, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the bus you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **functsel[2:0]**.

=> You now have to connect the 4-bit output pin to the output of the 8-to-1 mux. Position the mouse pointer over the output pin, and press "**b**" on the keyboard. Move the mouse pointer so that it is over the output bus of the 8-to-1 mux. Press the middle mouse button, or "**F3**" on the keyboard, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the bus you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **result[3:0]**.

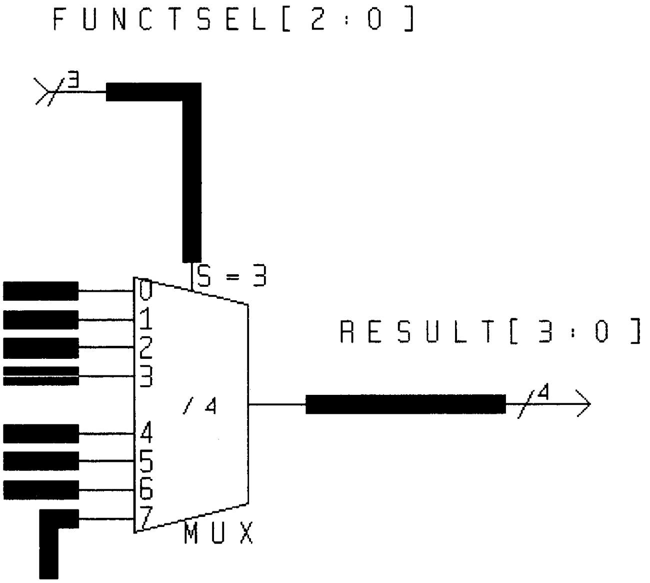


Figure 3-56 Connecting the mux to input and output ports

=> You need to connect the 1-bit input port to the ADD_SUB select pin in the Madd_sub component. Position the mouse pointer over the input pin, and press "**n**" on the keyboard. Move the mouse pointer so that it is over the add/subtract input signal of the adder-subtractor. Press the middle mouse button, or "**F3**" on the keyboard, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the net you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **add_sub_sel**.

=> You need to connect the `cout` output from the Madd_sub component to a 1-bit output port. Position the mouse pointer over the output pin component and press "**n**" on the keyboard. Move your mouse pointer so that it is over the `cout` signal of the Madd_sub component, and press the middle mouse button, or "**F3**" on the keyboard, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the net you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **addsub_cout**.

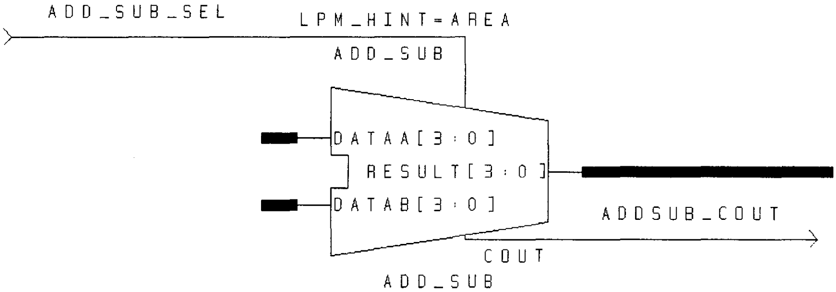


Figure 3-57 Connecting the Add-sub component

Now verify that your schematic matches up with Figure 3-51.

3.24.8 Saving the Schematic

Once all the components are positioned, labeled, and connected, save the schematic.

=> Select *Write* from the *File* pull-down menu in the ViewDraw menu bar, or type "**w**" at the keyboard.

ViewDraw saves the schematic and warns you about unconnected pins, unlabeled ports, and other potential problems.

Actually, it's a good idea to save or write the file frequently while drawing a schematic. You can do this by typing a "**w**" in the schematic window. Just ignore ViewDraw's warning messages until you think you're finished. Then, an error message could be telling you that you aren't finished yet.

Hint – If ViewDraw gives you an error message when you write a file, you probably forgot to connect or label something. It will also flag a warning if your input/output port sizes do not match the widths of the net/bus connected to them. All internal buses must also be labeled.

If ViewDraw produces any error messages at this stage, go back and make sure that:

- All component pins are connected to something.
- All input and output ports are labeled.
- All nets are wired so that the schematic looks **exactly** like Figure 3-51.

If no error messages appear, you're finished with the schematic.

3.25 Generating a Symbol from the Schematic

Once the final changes to the PLD schematic are complete, the next step is to create a symbol that can be instantiated onto a higher-level schematic.

=> To create the symbol, select *Schematic to Symbol* under the *Cypress* menu.

A 'Schematic to Symbol' pop-up window appears with the inputs listed on the left and the outputs on the right.

=> Click on *Accept.*

=> When the message in the left-hand corner of the ViewDraw window reads "symbol:PLD.1 is created successfully," the application is complete.

If errors were reported, go back and carefully edit the schematic. Make sure that the schematic matches **exactly** as shown in Figure 3-51.

3.26 Writing the VHDL File

ViewText is Workview PLUS's (and Powerview's) ASCII text editor.

You'll use ViewText to create the VHDL description for *cntrl.vhd.* You'll convert this description into a symbol, which you'll then instantiate onto a top-level circuit.

=> To start ViewText, select *Edit text file...* from the *Red Square* menu in the ViewDraw window.

Note – If you are using ViewDraw on an IBM PC or compatible computer: the ViewText text editor is not available from the Red Square in the Cockpit. It's only available from the *Red Square* menu in ViewDraw or ViewSim or other tools *contained* in the Cockpit. Make sure you pull down the *Red Square* menu from the ViewDraw window.

The ViewText dialog box appears.

=> Type **`cntrl.vhd`** on the line labeled *File Name,* then click *OK.*

The ViewText window appears. Now you can start entering the text of the *cntrl.vhd* file.

You will now build a controller for the PLD circuit that you created earlier in this tutorial. You will write a simple ALU controller in VHDL.

The *cntrl* VHDL description will be written in three parts:

- the **entity** declaration declares the name, direction, and data type of each port of the component
- the **architecture** describes the behavior of the component
- the **package** declaration provides the information to the *Warp* compiler to allow *cntrl* to be used as a component in a schematic

The following pages discusses the contents of each of these sections of the VHDL description.

Note – If you would rather not type in the *cntrl.vhd* file, you can copy it from the *Warp* directory. The default location for the *cntrl.vhd* file is *c:\warp\examples\wtutor\alu\cntrl.vhd.* From here, copy *cntrl.vhd* to your project directory. Copy this file to directory *c:\w2tutor\alu*. Then, read along for the next few pages to help you understand the purpose of each section of a VHDL source file.

3.26.1 Writing the Entity Declaration

The entity declaration declares the name, direction, and data type of each port of the component.

=> Copy the following lines into your ViewText text editor:

```
entity cntrl is port (
   instr:        in     std_logic_vector(3 downto 0);
   functionsel:  out    std_logic_vector(2 downto 0);
   addsubsel:    out    std_logic);
end cntrl;
```

All signals declared in the entity are of type `std_logic`.

The signal `instr` is 4-bits wide and is used to determine the logic function to be performed by the ALU.

The signal `functionsel` is 3-bits wide and is used to drive the select lines of the 8-to-1 mux described in the schematic earlier.

The signal `addsubsel` is 1-bit wide and determines if the Madd_sub component instantiated in the schematic earlier, performs an addition or a subtraction.

3.26.2 Writing the Architecture

The architecture portion of a VHDL description describes the behavior of the component. The architecture appears *after* the entity declaration in the *.vhd* file.

=> Copy the following lines into your VHDL text editor below your entity declaration:

```
architecture archcntrl of cntrl is

begin

control:  process (instr)
   begin
         case instr is
            when x"0" =>
               functionsel     <= b"000";
-- select the output of the adder/subtractor
               addsubsel       <= '1'; -- A + B
            when x"1" =>
               functionsel     <= b"000";
-- select the output of the adder/subtractor
               addsubsel       <= '0'; -- A - B
            when x"2" =>
               functionsel     <= b"001";
-- select magb[3:0] = [A>=B        A>B     A<=B    A<B]
               addsubsel       <= '-';
            when x"3" =>
               functionsel     <= b"010";
-- select maga[3:0] = [0    0      A/=B    A=B]
               addsubsel       <= '-';
            when x"4" =>
               functionsel     <= b"011";
-- select the output of A and B
               addsubsel       <= '-';
            when x"5" =>
               functionsel     <= b"100";
-- select the output of A or B
               addsubsel       <= '-';
            when x"6" =>
               functionsel     <= b"101";
-- select the output of A xor B
               addsubsel       <= '-';
            when x"7" =>
               functionsel     <= b"110";
```

```
-- select the output of the inverse of A
               addsubsel      <= '-';
            when x"8" =>
               functionsel    <= b"111";
-- select the output of the inverse of B
               addsubsel      <= '-';
            when others =>
-- to cover all the other possibilities
               functionsel    <= "---";
               addsubsel      <= '-';
         end case;
   end process control;
end archcntrl;
```

In the architecture definition above, the first line declares an architecture named `archcntrl` of entity `cntrl`. You use a CASE-WHEN statement to define the logic for the signals `functionsel` and `addsel`, based on the values assumed by `instr`.

The IEEE 1164 VHDL standard allows you to assign '-' (don't care) values as genuine output assignments. *Warp* will use these don't care conditions in doing logic optimization.

The result of every value assumed by `instr` results in a particular assignment to the select lines of the mux and the add_sub select line for the Madd_sub component.

3.26.3 Writing the Package Declaration

The package declaration provides the information to the *Warp* compiler to allow `cntrl` to be used as a component in a schematic. The package declaration must appear ***before*** the entity declaration or architecture in the *.vhd* file.

=> Copy the lines below into your VHDL text editor ***above*** the entity and architecture declarations.

```
package cntrlpkg is
      component cntrl port (
         instr:       in      std_logic_vector(3 downto 0);
         functionsel: out     std_logic_vector(2 downto 0);
         addsubsel:   out     std_logic);
      end component;
end cntrlpkg;
```

The first line declares the name of the package. The name of the package must be distinct from the name of any component declared within that package. Using the convention `<entity>pkg` works nicely.

The second line declares a component named `cntrl`. The component name that appears on this line must match the name of an accompanying entity.

The port statement declares the name, direction, and type of each port in the component. You can copy the port statement from the entity declaration for this purpose.

An `end component` and `end cntrlpkg` statement conclude the package declaration. Note that the package named in the `end package` statement must match that shown in the first line of the package declaration.

3.26.4 Including Libraries

=> Insert the following two lines of VHDL code ***before*** the package declaration ***and before*** the entity declaration:

```
library ieee;
use ieee.std_logic_1164.all;
```

These two lines advise the compiler to link to the IEEE VHDL library. The compiler finds definitions for the various types, modes, and VHDL constructs defined in the user's VHDL code. In general, the **`library`** and **`use`** VHDL reserved words instruct the compiler to include pre-defined libraries and user created VHDL files, when compiling the selected VHDL code. For more information on the `std_logic_1164` and `std_arith` packages, see Chapter 4, "VHDL," in the *Reference Manual*.

=> Save the file as *c:\w2tutor\alu\cntrl.vhd* and close the ViewText window.

=> Minimize the ViewDraw window also.

3.27 Verifying the VHDL File

At this point in the tutorial, you'll use *Warp* to verify that the *cntrl.vhd* file is syntactically correct. This step **is strictly necessary** for this tutorial since we are going to create a schematic symbol from this VHDL file. The *Warp3* system requires the user to compile any VHDL file before creating a schematic symbol from it. It's always a good idea, anyway, to compile any VHDL description once you've completed it. That way, you can spot problems in your VHDL description when they are easiest to identify and correct. Later, should you encounter problems with the larger circuit, you can at least be assured that you have taken care of any bugs at the lower levels of the hierarchy.

3.27.1 Starting Galaxy

Galaxy is the user interface for the *Warp* VHDL synthesis compiler.

=> To start Galaxy, double-click on the *Warp* icon in the Cockpit.

The Galaxy window appears.

3.27.2 Compiling the VHDL File

=> Select *Add* from the *Files* pull-down menu in the Galaxy window.

This dialog box lists the VHDL files available in the current directory on the left side, and the VHDL files selected for compilation/synthesis on the right side.

=> To select the *cntrl.vhd* file for compilation, select *cntrl.vhd* from the list of files on the left-hand side, then click on the "-->" button. Click on *OK* to close this dialog box.

=> In the main Galaxy window, click on the *Selected* button in the *Compile* button group. Ensure that *cntrl.vhd* is not selected as the *Top Design* in the left-hand bottom corner of the Galaxy window..

Warp runs, printing messages to keep you appraised of its progress. The compilation process should run to completion, without any error messages.

=> If the compilation is successful, close the Galaxy window by clicking on the close box and then clicking *OK*.

Note – If you do get error messages, check to make sure that the various parts of the *cntrl.vhd* file read **exactly** as they are listed on the preceding pages. Better yet, copy the *cntrl.vhd* file from the *c:\warp\examples\wtutor\alu* directory, then run *Warp* again.

3.28 Generating a Symbol for the VHDL File

Once the control VHDL description is written and compiled without errors, the next step is to create a control symbol that can be instantiated onto a schematic.

=> To create the symbol, first restore your ViewDraw window or open a new one if it is not already running by double-clicking on the *ViewDraw* icon in the Cockpit.

=> In ViewDraw, choose *VHDL To Symbol* from the *Cypress* pull-down menu

=> Type **`cntrl`** in the *VHDL Source Filename* field

=> Then click *OK* in the ensuing dialog box.

This runs the "vhdl2sym" application. When the left-hand corner in the ViewDraw window reads "symbol created" the application is complete.

If no errors were reported, go on to the next step of the tutorial.

If errors were reported, go back and carefully edit the *cntrl.vhd* file. Make sure that its three parts--the package declaration, entity declaration, and architecture--read **exactly** as shown in the earlier sections. Also make sure that the VHDL file compiled successfully.

3.29 Creating a Top-Level Schematic

In this section of the tutorial, you'll use the symbols created for `cntrl` and `pld` in the preceding sections in a top-level schematic.

If ViewDraw is already running, skip the first step.

=> Double-click on the *ViewDraw* icon from within the Cockpit.

=> Select *File->Open* from your ViewDraw menu bar.

The ViewDraw *File->Open* dialog box appears.

=> Type **`top_alu`** on the *Enter name*: line in the dialog box, then click *OK*.

A new blank schematic sheet appears.

=> Now bring your PLD schematic window, if it is still open, to the foreground once again, and select *Dismiss Window* from the *Red Square* menu to close the PLD schematic window.

You should now have a single ViewDraw schematic window open. This window should be blank and it should be called **`top_alu`**.

3.29.1 Instantiating a Component

The process of calling up a component from a library and placing it on a schematic sheet is called "instantiating" a component. Components to be instantiated include: `cntrl` (underlying VHDL code for ALU controller), `pld` (ALU schematic built using LPM elements), three IN ports, and two OUT ports.

=> Press "**`i`**," or choose *Comp* from the *Add* pull-down menu in ViewDraw, and click on the button displaying the project directory, `c:\w2tutor\alu`.

The upper half of this dialog box should then include `pld` and `cntrl` components.

=> Click on the `pld` component, and move the mouse pointer into the schematic window. Click on the middle mouse button to place the component in the schematic. Repeat the same steps to place the control unit's symbol in the schematic.

=> Click on *Cancel* when both symbols have been instantiated in your schematic window.

=> Now Select *LPM Symbol* from the *Add* pull-down menu in the ViewDraw menu bar and create three 4-bit IN ports, a 1-bit OUT port and a 4-bit OUT port.

You now have three input ports: `instr`, `A`, and `B`, and two output ports: `answer` and `cout`, to be defined in the top-level schematic.

When you are finished, the sheet should resemble Figure 3-58.

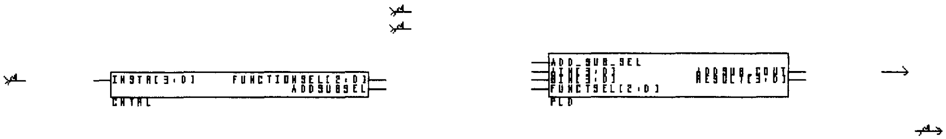


Figure 3-58 top_alu schematic with components positioned

=> At this point, you can close this dialog box by clicking *Cancel.*

For those users who are new to ViewDraw, a step-by-step description of component instantiation, labeling, and wiring will be given in Section 3.29.5 below entitled, "Detailed Description of Instantiating, Labeling, and Wiring."

3.29.2 Positioning Components

Once the components are instantiated on the schematic sheet, you should position them to enhance the readability and ease-of-wiring of the schematic.

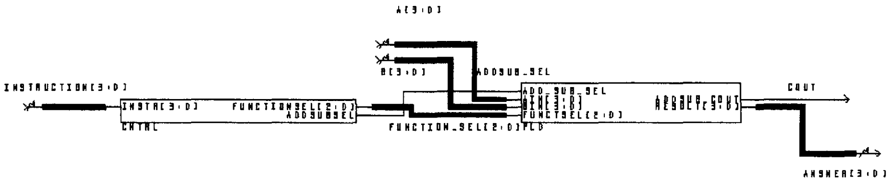


Figure 3-59 top_alu schematic design

You'll want to position the `cntlr` and `pld` symbols in the center of the schematic, the input ports on the left side of the schematic and the output ports on the right. This format will allow you to see the logic flow within the circuit.

=> Select a component using the left mouse button.

=> Press "**m**" on the keyboard, or select *Move* from the *Edit* pull-down menu.

=> Move the cursor to where you want the component to be.

=> Click on the middle mouse button (for a 3-button mouse) or click on the left mouse button and press "**F3**" (for a 2-button mouse).

=> Repeat until all components are positioned properly.

3.29.3 Wiring Components Together

Once components are positioned and labeled, you must wire them together to make the circuit.

The final circuit should look like Figure 3-59.

Connecting Two Components

=> Move the cursor to the origin of the net.

=> To create a net: Select *Net* from the *Add* pull-down menu in the ViewDraw menu bar, or type "**n**" at the keyboard.

=> To create a bus: Select *Bus* from the *Add* pull-down menu in the ViewDraw menu bar, or type "**b**" at the keyboard.

=> Specify points along the net/bus by clicking the middle mouse button ("**F3**" if you have a two button mouse).

=> Click the left mouse button once to back up one segment on the net, twice to back up two segments, etc.

=> To connect the net/bus to a component pin, move the cursor to a point on the pin and click the middle mouse button ("**F3**" on a 2-button mouse).

=> To connect the net/bus to another net or bus, move the cursor to a point on the net or bus and click the middle mouse button ("**F3**" on a 2-button mouse).

=> To leave the net/bus dangling, form the net/bus and click the middle and then the right mouse buttons ("**F3**" and then the right mouse button on a 2-button mouse).

Note – Make sure to connect all input pins on components to something. ViewDraw does not allow unconnected inputs.

- If you want to delete a net or a bus, select the net or the bus and type "**d**" to delete it.
- If you want to undo anything that you did to your schematic, press "**u**."

Note – It is a good practice to save your schematics whenever possible by typing "**w**" in your schematic sheet.

For the benefit of users who have not used ViewDraw before, the section below entitled, "Detailed Description of Instantiating, Labeling, and Wiring," will walk you through individual steps involved in creating the final `top_alu` schematic.

3.29.4 Labeling Nets and Buses

Once components are positioned on the schematic, you need to label the nets. The `top_alu` schematic should look like Figure 3-59.

Labeling the Component Ports

=> Select the net/bus, using the left mouse button.

=> Select *Label* from the *Add* pull down menu the ViewDraw menu bar, or type "**l**" on the keyboard.

=> Click on the *label* line of the dialog box.

=> Delete the contents of the *label* line, if any.

=> Type the new label, e.g., **clk**. Case doesn't matter; all characters will be upper-case on the schematic.

=> For buses, you also need to specify the bus width along with its label, e.g., **`data[7:0]`**.

=> Click *OK*.

=> Move the cursor to where you want the label to appear (i.e., next to the port its associated with).

=> Click the middle mouse button to place the label.

=> Repeat until all buses and nets are labeled as in Figure 3-59.

Hint – Here's a shortcut method for adding several labels to input and output ports of a component at once:

- Select a port.
- Select *Label* from the *Add* menu.
- Type the names of the ports to be labeled as one comma-separated list. DO NOT insert spaces after the commas. The string should look like: **`label1,label2,label3,...,...,...`**
- Click on the *OK* button. The first label in the list appears.
- Position the label, then click the middle mouse button.
- Click the left mouse button to select the next item to be labeled.
- Press the "`1`" key on your keyboard to activate the next label.
- Repeat the last three steps for each port to be labeled.

For the benefit of people who have not used ViewDraw before, the section below will walk you through individual steps involved in creating the final schematic.

Note – It is a good practice to save your schematics whenever possible by typing "**`w`**" in your schematic sheet.

3.29.5 Detailed Description of Instantiating, Labeling, and Wiring

=> Press "**i**" or choose *Comp* from the *Add* pull-down menu in ViewDraw, and click on the button displaying the project directory, *c:\w2tutor\alu*. The upper half of this dialog box should then include `pld` and `cntrl` components. Click on the `pld` component, and move the mouse pointer into the schematic window. Click on the middle mouse button to place the component in the schematic. Repeat the same steps to place the control unit's symbol in the schematic.

=> You'll begin by instantiating the 4-bit output. Select *LPM Symbol* from the *Add* pull-down menu in the ViewDraw menu bar. Click on the *OUT* component button. The dialog box that appears contains one parameter, the bus width of the output. Enter **4** in the width field. Click on the *OK* button and move the mouse pointer into the schematic window. Press the middle mouse button to place the component in the schematic. Place this output pin component near the **result[3:0]** output of the `pld` symbol. Position the mouse pointer over the output pin component you just created, and press "**b**" on the keyboard. Move the mouse pointer so that it is over the **result[3:0]** output pin of the `pld` symbol, and press the middle mouse button to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the bus you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **answer[3:0]**.

=> Now, you must create a single-bit output pin for `cout`. Select *LPM Symbol* from the *Add* pull-down menu in the ViewDraw menu bar. Click on the *OUT* component button. The dialog box that appears contains one parameter, the bus width of the output. Enter **1** in the width field. Click on the *OK* button and move the mouse pointer into the schematic window. Press the middle mouse button to place the component in the schematic. Place this output pin component near the `COUT` output of the `pld` component. Position the mouse pointer over the output pin component you just created, and press "**n**" on the keyboard. Move the mouse pointer such that it is over the `ADDSUB_COUT` output pin of `pld`, and press the middle mouse button to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then click on the bus you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **cout**.

3

=> You now have to add three 4-bit-wide input pins into the top-level schematic. Select *LPM Symbol* from the *Add* pull-down menu in the ViewDraw menu bar. Click on the *IN* component button. The dialog box that appears contains one parameter, the bus width of the input. Enter **4** in the width field. Click on the *OK* button and move the mouse pointer into the schematic window. Press the middle mouse button to place the component in the schematic. Place the input pin component near the **instr[3:0]** input of the `cntrl` unit. Position the mouse pointer over the input pin component you just created, and press "**b**" on the keyboard. Move the mouse pointer such that it is over the **instr[3:0]** input of the `cntrl` unit, and press the middle mouse button to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the bus you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **instruction[3:0]**.

=> You can copy the existing 4-bit input pin to create the two 4-bit input pins you need for the operands, A and B. Click on the output pin component you just instantiated. A white box should appear surrounding the component telling you that you have selected the component. Press "**c**" on the keyboard, or choose *Copy* from the *Edit* pull-down menu, to make another instance of this 4-bit-wide input pin. Move the new instance over to the `pld` symbol so that you can easily connect the input pin to the **AIN[3:0]** input. Press the middle mouse button to place the output pin component. Click on a blank area of the schematic to deselect anything that had been selected. Position the mouse pointer over the input pin, and press "**b**" on the keyboard. Move mouse pointer such that it is over the **AIN[3:0]** input of the `pld`. Press the middle mouse button to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then click on the bus you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **A[3:0]**.

=> You repeat these steps to create the B input pin. Move your mouse pointer such that it is over the **BIN[3:0]** input of `pld`. Press the middle mouse button to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the bus you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **B[3:0]**.

=> You have now added all of the input and output pins needed in the top-level. All that remains is to make the internal connections between the `cntrl` units' outputs, and the `pld`'s inputs. You'll begin with the 3-bit bus used to select a signal in the 8-to-1 mux contained within the `pld` schematic. Position the mouse pointer over the **functionsel[2:0]** output pin of the control unit, and press "**b**" on the keyboard. Move the mouse pointer such that it is over the **function[2:0]** input of `pld`. Press the middle mouse button to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then click on the bus you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **function_sel[2:0]**.

=> The final connection you need to make in the top-level schematic is to the `pld` component, the signal which tells the adder-subtractor whether to perform A + B or A - B. Position the mouse pointer over the `addsubsel` output pin of the control unit, and press "**n**" on the keyboard. Move mouse pointer such that it is over the **add_sub_sel** input of the `pld` component. Press the middle mouse button, to make the connection. Click on a blank area of the schematic to deselect anything already selected. Then, click on the net you just drew and press "**l**" on the keyboard, or choose *Label* from the *Add* pull-down menu. Type **addsub_sel**.

=> When this step is completed, the `top_alu` schematic should look like Figure 3-59.

3.29.6 Saving the Schematic

Once all the components are positioned, labeled, and connected, save the `top_alu` schematic.

=> Select *Write* from the *File* pull-down menu in the ViewDraw menu bar.

ViewDraw saves the schematic and warns you about unconnected pins, unlabeled ports, and other potential problems.

Actually, it's a good idea to write the file frequently while drawing a schematic. Just ignore ViewDraw's warning messages until you think you're finished. Then, an error message could be telling you that you aren't finished yet.

Hint – If ViewDraw gives you an error message when you write a file, you probably forgot to connect or label something.

If ViewDraw produces any error messages at this stage, go back and make sure that:

- all component pins are connected to something
- all input and output ports are labeled
- all nets are wired so that the schematic looks **exactly** like Figure 3-59

If no error messages appear, you're finished with the `top_alu` schematic.

3.30 Exporting the Top-Level Schematic

You need to convert this top-level, mixed mode hierarchical design into an IEEE compliant VHDL file. This file is the input to the *Warp* compiler.

=> After saving the file, within ViewDraw choose *Export to VHDL* from the *Cypress* pull-down menu of the `top_alu` schematic window.

=> Choose *type* to be *std_logic.*

This menu option takes the `top_alu` schematic and generates VHDL code which you can synthesize using *Warp*.

A window appears, informing you of the progress of the application. When the banner on this window reads "0 errors and 0 warnings," the application is complete.

=> Close the application window. After the export process is complete, close all open *Warp3* windows, except for the Cockpit, NSD and STDIO windows.

3.31 Compiling and Synthesizing the Top-Level Schematic

=> Following the steps illustrated in Sections 3.18.5 to 3.18.9, run *Warp* to produce a QDIF file for a specific target device (in this case, a CY7C382A).

=> Choose the default options in Galaxy to compile *top_alu.vhd.*

This operation generates two files of particular interest (among others):

- The first is named *top_alu.qdf.* The *.qdf* file is used as an input into the SpDE toolkit. The *.qdf* file is the input file for the SpDE toolkit. SpDE will then place and route the design and can produce a *.lof* file, which can be used to program a device.
- The second file is named *top_alu.rpt.* It contains information about the final synthesized design. You can see the *.rpt* file by pulling down the *Info* menu from the Galaxy window and clicking on *Report.* For more information on what is contained in a report file, refer to Chapter 6, "Report File," in the *Reference Manual.*

=> Once you have compiled and synthesized this design into a pASIC380 FPGA device, you can quit the Galaxy window. (Choose *Quit* from the *Project* menu.)

Note – If compilation errors occur, do the following: make sure the text of your file is entered **exactly** as shown earlier in this section -- or better yet, copy it from the *c:\warp\examples\wtutor\alu* directory -- and then run *Warp* again.

3.32 Placing and Routing the Design

If you targeted a pASIC380 device when synthesizing your design via *Warp,* you must process the resulting *.qdf* file with the SpDE toolkit in order to place and route the design into the chosen device.

To launch SpDE, double-click on the *Place&Rte* icon in the Cockpit.

=> Click on *OK* in the ensuing dialog box.

The SpDE window appears.

=> Click on the *folder* button located in the upper left portion of the SpDE window. This is equivalent to selecting *File->Import->QDIF*.

An open-file window appears listing your directory structure on the right and the *.qdf* files on the left.

=> Go to your project directory if you are not already there.

=> Highlight *top_alu.qdf* by clicking on it.

=> Click on the *OK* button to import the file.

After importing the file, the *hammer* button will now be selectable (it was not available before loading the QDIF file).

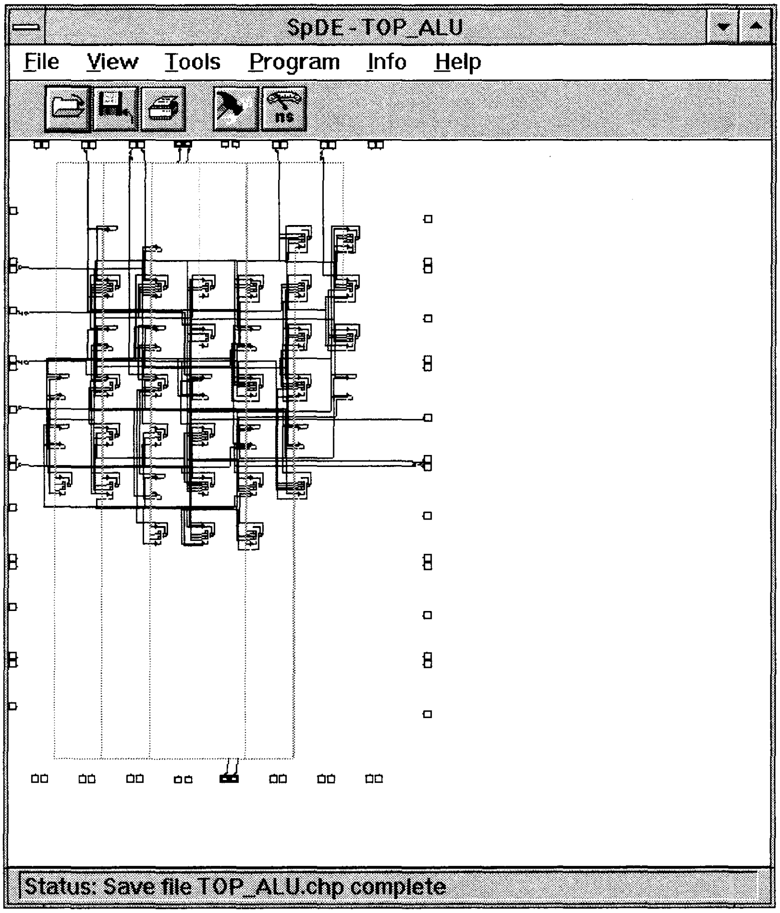
=> Click on the *hammer* button in the SpDE window.

A window will appear containing all the tools you can run. The window appears with all of them selected.

=> Click on the *Run* button to begin the place and route process.

The place and route process will take a couple of minutes to complete. Upon completion, a window will appear stating "All chosen SpDE tools ran successfully."

=> Click on the *OK* button to close the above message window.



Status: Save file TOP_ALU.chp complete

Figure 3-60 SpDE output for top_alu.qdf

=> Select *File->Save* from the SpDE menu bar, save your changes, and then *File->Exit* to exit.

Refer to Chapter 5, "SpDE," in this *User's Guide* for more detailed information about the SpDE toolkit.

3.33 Simulating the Behavior of the Design with ViewSim

Once the design is synthesized, you should simulate its behavior and evaluate its timing performance to ensure that it functions as intended.

You'll use the ViewSim simulator to test the behavior and timing requirements of the design implementation. In this tutorial, you'll perform the following steps:

- create a ViewSim model
- start ViewSim
- set the values of the stimulus signals in the simulation
- simulate the design
- examine the results to figure out what happened

3.33.1 Running pASIC->VSim

=> To generate a ViewSim model from the output of the SpDE tools, double-click on the *pASIC->VSim* icon in the Cockpit.

=> A dialog box appears, containing a command line to be executed. Make the command line read **`top_alu`**, then click on *OK*.

A window appears, informing you of the progress of the application. When the banner of this window reads inactive "pASIC->VSim," the application is complete.

Note – Verify that your current project directory is set to *c:\w2tutor\alu*. Ignore the messages at the bottom of the window. If the application reports 0 errors and 0 warnings in the sixth text line from the bottom, the application ran successfully.

=> Close the pASIC->VSim window.

3.33.2 Running ViewSim

Once the design is synthesized, it's a good idea to simulate its behavior and evaluate its timing performance to ensure that it functions as intended.

Note – Before performing this step of the tutorial, copy the *total.cmd* file from the *Warp* directory (its default location is *c:\warp\examples\wtutor\alu\top_alu.cmd*) to your project directory.

=> To simulate the behavior of the design, double-click on the *ViewSim* icon in the Cockpit.

A dialog box appears.

=> Make sure the design name reads **top_alu**, then click on *OK*.

ViewSim starts up.

=> When the ViewSim window appears ('SIM>'), type **top_alu** at the command line.

The *top_alu.cmd* file runs, executing the following sequence of ViewSim commands (not necessarily echoed to the screen):

```
restart
stepsize 25ns
vector instr instruction[3:0]
vector answer answer[3:0]
vector a a[3:0]
vector b b[3:0]
wave top_alu.wfm instr a b answer cout

assign a 9\h
assign b 5\h
assign instr 0\h

cycle 5

assign instr 1\h

cycle 5

assign instr 2\h
```

```
cycle 5

assign instr 3\h

cycle 5

assign instr 4\h

cycle 5

assign instr 5\h

cycle 5

assign instr 6\h

cycle 5

assign instr 7\h

cycle 5

assign instr 8\h

cycle 5
```

The commands beginning with "`a`" are vector assignments: I'm assigning a value to a predefined vector. The backslash h, "`\h`," tells ViewSim that the value is in hexadecimal format. The `cycle 5` statements advance the simulation time by 5 simulation cycles. Please see the ViewSim Reference Manual in the Design Entry and Digital Simulation Solutions collection within the Viewlogic on-line documentation set for more details on these commands.

- The sequence of instructions is add, subtract, magnitude compare, equality compare, and, or, xor, inverse of `A`, and inverse of `B`.
- The simulation begins by assigning `9\h` to signal `A` , `5\h` to signal `B`, and `0\h` to `instr`. Since instruction 0 is addition, you should see `E\h` on the answer bus with a `Cout` of 0. After the `cycle 5` statement, you see that the answer bus does contain the value `E\h`.

- Before the next simulation cycle, the instruction is changed to subtraction by assigning **1\h** to the **instr** signal. The design should perform **9\h - 5\h = 3\h**. You then cycle the simulator and see that the answer bus now contains the value **3\h**. The result is **3\h** because of the way the LPM subtractor is defined. Our implementation uses a default borrow-in of '1' for subtraction operations, and a carry-in of '0' for addition operations.
- The instruction bus, signal **instr**, is now changed to **2\h**. This should allow you to view the magnitude comparison outputs of the comparator: greater than or equal, greater than, less than or equal, and less than. Since **A > B**, the most significant bit and the second most significant bit of this bus should be a logic high. Thus, you should see **C\h** on the answer bus. After the **cycle 5** statement, you see the bus change to **C\h**.
- The next command changes the instruction to **3\h**, allowing us to view the equality comparison outputs of the comparator: equal and not equal. **A** is still greater than **B**, which means that only the not equal to bit will be a logic high. This is the second least significant bit on the bus. Therefore, you should see **2\h**. After the **cycle 5** command is executed, you see the answer bus change to **2\h**.
- You now change the instruction to **4\h** which should produce **A** AND **B** on the answer bus. **9\h** is "1001" in binary and **5\h** is "0101" in binary. ANDing these together should yield "0001", or **1\h**. This matches the value seen on the answer bus after the **cycle 5** statement is executed.
- The instruction moves to **A** OR **B** by assigning **5\h** to **instr**. After selecting **A** OR **B**, the answer bus should contain the value "1101" in binary which is **D\h**. The answer bus in the ViewTrace window does contain the value **D\h**.
- The next instruction executed is **A** XOR **B**. This should result in a value of "1100" in binary which is **C\h**. After the **cycle 5** statement is processed, you see the answer bus change to **C\h**.
- The final two instructions, **7\h** and **8\h**, are the inverse of **A** and the inverse of **B**, respectively. Inverting A should result in "0110" in binary which is **6\h**. As predicted, the answer bus contains **6\h** after the **cycle 5** statement is executed. Finally, inverted **B** should yield "1010" in binary which is **A\h**. The last value you see on the answer bus is **A\h**. Thus, your 4-bit ALU is functioning correctly.

- You can also run commands directly from the ViewSim command line. Test the comparator by assigning **B** to be equal to **A**. In the bottom part of the ViewSim window type, **assign b 9\h** to assign **9\h** to signal **B**. Now type **assign instr 2\h** to assign **2\h** to **instr**. This is the instruction which shows the magnitude comparison outputs of our ALU. Now type **cycle 5** to advance the simulation and show the results. Since **A** is now equal to **B**, the most significant bit and the second least significant bit should be a logic high. This corresponds to a value of **A\h**. As predicted, the answer displays the value **A\h**.
- For your final simulation statement, change the instruction to show the equality comparison outputs of the comparator. Type **assign instr 3\h** in the bottom part of the ViewSim window. Then, type **cycle 5**. The answer bus should change to **1\h** because **A** is equal to **B**.

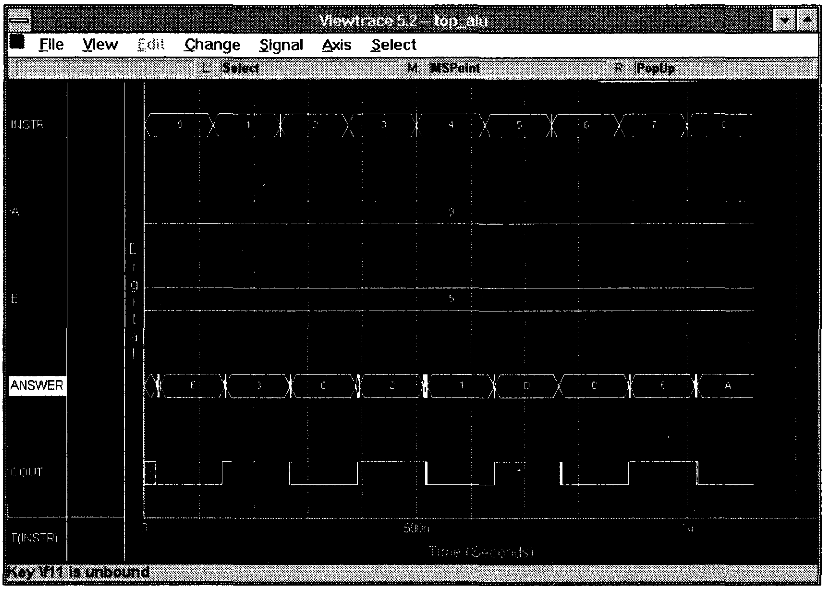


Figure 3-61 ViewTrace output for top_alu design

3.34 Back-Annotating Pin Assignments

You can easily lock in the pin assignment made by the place and route tool.

=> Double-click on the *Galaxy* icon in the Cockpit.

=> Highlight *top_alu.vhd* by clicking on it. If you don't have *top_alu.vhd* selected, do a *Files->Add* and add *top_alu.vhd*.

=> Highlight *top_alu.vhd* by clicking on it.

=> Choose *Annotate...* from the *Files* pull-down menu.

A small window appears giving the name of the file which will be back-annotated (in this case, *top_alu.vhd*), and giving you the option of back-annotating the pins, the nodes or both. The *Pins* option should already be selected.

=> If *Pins* is not already selected, click on the button to the left of *Pins*.

=> If *Nodes* is selected, deselect it by clicking on the button to the left of *Nodes*.

=> Click on the *OK* button to back-annotate the pin information.

Note – The back-annotation information is stored in a control file. Control files have a *.ctl* extension.

3.35 Conclusion

Warp is a very powerful VHDL synthesis tool which supports a broad range of PLD, CPLD, and FPGA devices. The *Warp* development system is based entirely on IEEE 1076/1164 compliant VHDL and supports a wide range of user options. The tutorials presented here simply scratched the surface of the power contained within the *Warp* software. Please continue reading chapters 4, 5, and 6, for more detailed information on the Galaxy, SpDE, and Nova tools respectively. Chapters 7 and 8 contain information on designing with ViewDraw and using many popular simulation tools including ViewSim. Chapter 9 is a must read for users interested in optimizing their designs for either area or speed. This chapter covers the design tuning cycle and how to take advantage of the many user controls contained in the Galaxy menus. Finally, Chapter 10 discusses programming files and their uses.

The *Reference Manual* is also a resource containing information on such things as command line options, synthesis directives, LPM schematic modules, and report files. Chapter 4 of the Reference Manual, combined with the *VHDL for Programmable Logic* textbook, is a complete guide to designing programmable logic using VHDL. Finally the appendices in the *Reference Manual* contain a list of error messages, a glossary, and the BNF of the VHDL language.

Be sure to visit the Cypress Semiconductor home page on the World Wide Web at http://www.cypress.com. This Web site contains the latest information on all of Cypress' products as well as software updates, frequently asked questions, and a solutions database for your support needs.

Chapter 4

Galaxy

4.1 Overview

Galaxy is the Graphical User Interface (GUI) for the synthesis and fitting engine within *Warp*.

Warp is Cypress Semiconductor Corporation's name for its VHDL compilation and synthesis software. *Warp* accepts VHDL source files as input. The primary output of a *Warp* run is a *.jed* or a *.qdf* file. The *.jed* file can be used as input to a PLD programmer. The *.qdf* file can be used as input to SpDE for automatic placement and routing of Cypress' FPGA devices.

Galaxy provides a GUI to perform the following tasks:

- Select VHDL source files for compilation or synthesis with built-in error tracking.
- Produce post-fitting simulation models with timing for PLD and CPLD devices.
- Back-annotate post place and route (or fitting) pin and node placements into the control file.
- Manage VHDL libraries that can be shared by multiple users or designs.
- Choose various synthesis options, such as type of flip-flops to use, degree of optimization, etc.
- Provide a VHDL editor with syntax templates.
- Provide a report file browser.
- Access to third party tools (such as Exemplar) that can produce Cypress PLA format files.

These are only some of the highlights of Galaxy. This chapter describes how to use Galaxy to run *Warp* and perform the above tasks. It assumes familiarity with common user interface operations, such as the use of scroll bars, menus, buttons, and opening and closing windows. Galaxy supports the Microsoft Windows windowing environment on the PC and the Motif® interface for UNIX platforms running the X-windowing system.

4.2 Starting Galaxy

For *Warp3* Users

To start Galaxy on a UNIX workstation, the user must first invoke the Viewlogic Powerview environment. This is done by typing **`powerview`** on the command line of a shell window. Once within the Powerview environment, select the Cypress tool box (if not already selected) and then double-click on the icon marked *Warp*.

To start Galaxy on a PC running Windows, first start the Cockpit using the Cockpit icon in the *Warp R4* program group. Once the Cockpit is up, select the Cypress tool box (if not already selected) and then double-click on the icon marked *Warp*.

For *Warp2* Users

To start Galaxy on a UNIX workstation, simply type **`galaxy`** on the command line of a shell window in the directory where the project resides.

To start Galaxy while running Windows on a PC, double-click on the Galaxy icon in the *Warp R4* program group.

4.3 Project Management

4.3.1 What Is a Project?

Before Galaxy can process a design, a *Warp* project must be created. A *Warp* project consists of information regarding the environment, options used to compile the design, the order of compilation of individual design files, and so on. The project helps Galaxy maintain a set of project files and options that are used to synthesize or compile a design targeting a PLD or FPGA. The project can also be simply a reusable library. The concept of a library is explained in later sections of this manual.

The *Warp* project information is stored in a file with an extension *.wpr*. This file contains all conceivable information regarding the project including fonts and last locations for individual windows, compiler options, device/synthesis options, the target simulator, etc.

A typical project in Galaxy consists of a set of VHDL files with one file serving as the top level design file and an optional control file. The VHDL files could have been produced by other tools (such as ViewDraw) or manually typed.

Note – Galaxy cannot be used unless the user first creates a project.

Galaxy is also capable of maintaining and editing multiple projects simultaneously. In Galaxy, each project is assigned its own window.

4.3.2 *Warp* Project vs. Viewlogic Project

A *Warp* project is significantly different from a Viewlogic project. A Viewlogic project helps organize a set of schematics and symbols from multiple directories and combines them to make a design. Viewlogic projects are also used to record information about designs but are generally oblivious of VHDL compilation or synthesis concerns. Invoking Galaxy from the Viewlogic Cockpit allows Galaxy to track designs being maintained in the Viewlogic environment. The Export VHDL utility within ViewDraw links a Galaxy project with a Viewlogic project.

The data link between Galaxy and Viewlogic is comprised of:

- a top level design file (created using the Export VHDL utility within ViewDraw); and
- any lower level VHDL design modules for a mixed mode design.

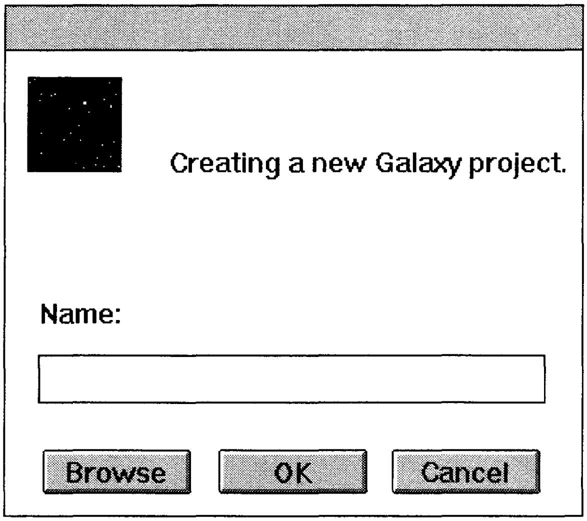
Once the above information is entered into Galaxy, it needs to be updated only if the design hierarchy changes.

4.3.3 Creating a Project

Galaxy automatically creates a project when first invoked. On a PC running Windows, Galaxy also remembers the last project in use and automatically invokes this project as the current project.

The following are three ways to create a project in Galaxy:

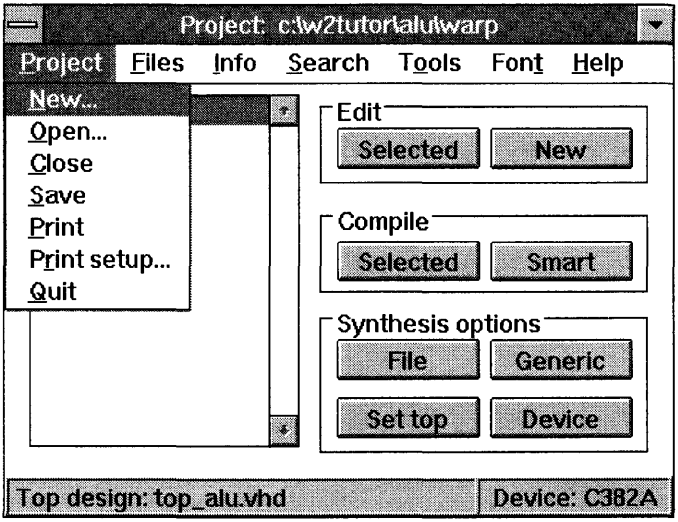
- When Galaxy is invoked, if there is no current project (on the PC) or if there are no project files in the current working directory (on UNIX platforms), Galaxy displays a dialog box that helps create a project. This dialog box prompts the user for the name of the project and the directory in which to create the project.



Creating a new Galaxy project.

Figure 4-1 The dialog box which appears when there is no project in the current directory or when using Galaxy for the first time

- If a project already exists, another method of creating a project is using the menu item *Project-> New* to create a new project.



Top design: top_alu.vhd

Device: C382A

Figure 4-2 The Project->New menu item

- Copying a project file (file with the *.wpr* extension) to another name with a *.wpr* extension also creates a new project.

Warp3 users must create a *Warp* project in the same directory as the current Viewlogic project.

A new *Warp* project will have no files or options associated with it. The following is an example of an empty project:

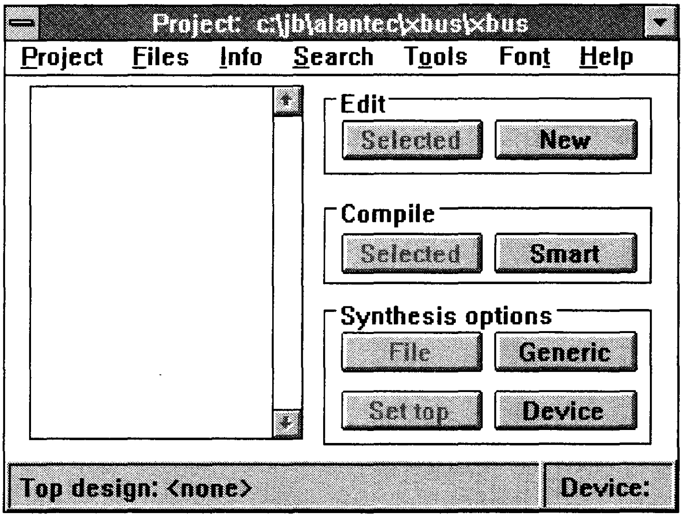


Figure 4-3 Galaxy window of a project that has just been created

4.3.4 Including Design Files in a Project

There are two types of files that can be added to a *Warp* project: VHDL files or a PLA file.

To add a VHDL file, use the menu item *Files->Add...*, which will create a dialog box allowing the selection of one or more VHDL files. Selecting *OK* on this dialog box adds the selected VHDL files into the project in the order in which they are selected. These new files are added below the current file in the project window if a file is currently selected.

It is important that the files listed in the project window be in the correct order. The higher the level of hierarchy for a given design file, the lower it must be listed in the files for a given project. This implies that the top level design file must be the last file.

Once the files are listed in the project, the user can also use other menu items under the *Edit* menu to delete, cut, copy, move up, or move down files.

Galaxy also supports a PLA project if the *.pla* file is imported from a third party tool, such as Exemplar, which is capable of producing Cypress-format PLA files. PLA projects are restricted to a single file project. In addition, once a PLA file is added to a project, VHDL files cannot be added to the project, and vice-versa.

4.3.5 Setting a Design File as Top Level

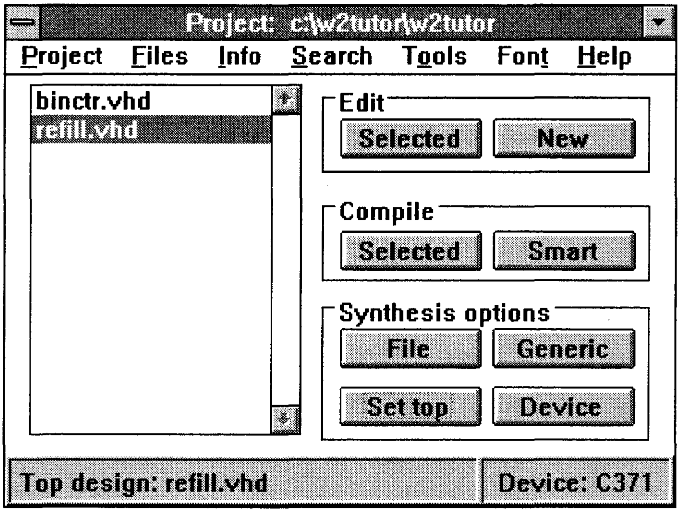
Once all the necessary VHDL files are added into the project in the correct order, the user must set the top level VHDL file. This is not necessary if the project is intended simply to compile files into a library instead of targeting a device.

If there is no top level file, all VHDL files in the current project are assumed for compilation only. Only the top level VHDL file is actually synthesized, and only one top level file is allowed in a project.

To identify a given file as the top level file, do the following:

- Make sure that the file is listed in the project window. If not, use the *Files->Add...* menu item to add it to the project.
- Select the top-level file by single-clicking on the file name in the project window with the left mouse button. This should highlight the top-level file.
- Single-click with the left mouse button on the *Set top* button found in the lower portion of the project window.

Once the top level file is set, the bottom of the project window will reflect the name of the top level file.



binctr.vhd
refill.vhd

Top design: refill.vhd

Device: C371

Figure 4-4 Galaxy window with refill.vhd set as the top level design

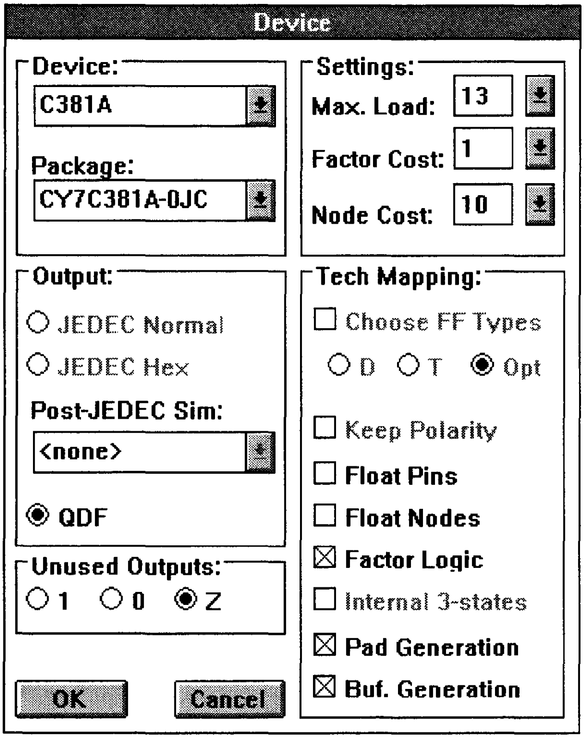
4.4 Targeting a Device

A target device and its related options can be selected using the *Device* button in the project window.

4.4.1 Selecting a Device and Device Package

For designs intended to target a device, a target device must be selected before the design can be synthesized.

Clicking on the *Device* button displays the *Device* dialog box. This box allows the user to select the device. As the user selects a given device, the dialog box automatically enables or disables appropriate options.



Device

Device: C381A

Package: CY7C381A-0JC

Settings:

Max. Load: 13

Factor Cost: 1

Node Cost: 10

Output:

○ JEDEC Normal

○ JEDEC Hex

Post-JEDEC Sim: <none>

● QDF

Unused Outputs: ○ 1 ○ 0 ● Z

Tech Mapping:

□ Choose FF Types

○ D ○ T ● Opt

□ Keep Polarity

□ Float Pins

□ Float Nodes

☒ Factor Logic

□ Internal 3-states

☒ Pad Generation

☒ Buf. Generation

Figure 4-5 Device dialog box

In the upper left corner of the *Device* dialog box are the *Device* and *Package* prompts, whose menus can be invoked using the left mouse button. First, select the device from the list of devices available in the *Device* menu. This changes the list of available Packages for that device and causes *Warp* to select a package by default. The user may also specify a package by selecting from the package selection menu.

4.4.2 Selecting Technology Mapping/Synthesis Options

The *Device* dialog box contains technology mapping options that are available for the device selected. At any given time, a set of options will be enabled or disabled depending on the device selected.

The following sections describes the various technology mapping or synthesis options and their related synthesis directive or command-line options, both of which are described in Chapters 2 and 3, "Command Line language" and "Synthesis Directives" of the *Warp Reference Manual*.

4.4.2.1 Choose FF Types

This option sets the global default for flip-flop selection and optimization. When enabled, this option has three possible sub-options: `D`, `T` or `Opt`.

- The `D` option will force all flip-flops in the design to D-Type.
- The `T` option will force all flip-flops in the design to T-Type.
- The `Opt` options will allow the fitter to choose the best implementation based on area savings. This option is highly recommended.

Turning off the *Choose FF Types* completely will force the fitter to use the flip-flops that were specified in the design via VHDL attributes or the control file.

The synthesis directive `ff_type`, if found in the control file or the source VHDL design, will override this option for the signal or signals that have that attribute specified.

Related Directive: `ff_type`
Command-Line-Option: -`fo` or `-fd` or `-ft`
Priority: Attributes have higher priority

4.4.2.2 Keep Polarity

When enabled, this option forces the fitter to use the polarity selected by the user in the VHDL or control file. Without this option selected, the fitter automatically selects the best polarity for all outputs.

The synthesis directive **polarity**, if found in the control file or the source VHDL design, will override this option for the signal or signals that have that attribute specified.

Related Directive: **polarity**
Command-Line-Option: **-fp** or **-fk**
Priority: Attributes have higher priority

4.4.2.3 Float Pins

When enabled, this option causes any fixed pinout specified in the source VHDL file or the control file to be ignored so that the fitter or place and route tool can find new solutions.

Related Directive: **pin_numbers**
Command-Line-Option: **-ff**
Priority: Galaxy option overrides any attributes

4.4.2.4 Float Nodes

This option is similar to Float Pins except this applies to internal nodes of the device. For CPLDs, this option affects any pre-placement information for buried/ internal nodes for the design, and for FPGA devices, this option affects any **fixed_ff** attributes applied to the outputs of flip-flops.

Related Directive: **node_num** or **fixed_ff**
Command-Line-Option: **-fn**
Priority: Galaxy option overrides any attributes

4.4.2.5 Factor Logic

This is a very important option for pASIC FPGA devices. This option enables the multi-level logic factoring algorithms within *Warp* and can cause the design size to be reduced quite dramatically. This option is recommended for all FPGA designs and if needed can be disabled using the **no_factor** directive on an individual signal after having evaluated the consequences.

For FPGAs, this option also has a related setting found in the *Settings* section of the *Device* dialog box. This setting indicates how aggressive *Warp* should be when factoring equations. The lower the number, the smaller the potential factor. In low settings, *Warp* looks for any potential reduction of the design size as a criteria for creating a factor, whereas with higher settings, *Warp* waits for more significant savings in area before a factor is created. The lowest setting of "1" is recommended for most designs.

Related Directive: **no_factor**
Command-Line-Option: **-fl**
Priority: Attributes have higher priority

4.4.2.6 Pad Generation

By default, *Warp* allocates the input/clock pads of FPGA devices to signals which it determines need the pads the most. All other signals will be assigned to I/O pins which have a normal drive strength. Disabling this option causes the *Warp* synthesis engine to skip automatic pad generation even if the user has specified the **pad_gen** attributes in his VHDL or control files. This option applies to FPGA devices only.

Related Directive: **pad_gen** and **max_load**
Command-Line-Option: **-yp**
Priority: Galaxy option overrides any attributes

4.4.2.7 Buffer Generation

This option is similar to Pad Generation, except that Buffer Generation applies to *Warp's* automatic buffer generation algorithm. Disabling this option forces *Warp* to skip the buffer generation phase. This option applies to FPGAs only.

A related setting that affects the way buffers are generated is the **max_load** setting. This setting specifies the default maximum allowable loading before a buffer is generated by *Warp*. Individual signals can have a different **max_load** setting via a VHDL attribute or control file. This setting is found in the upper right corner of the *Device* dialog box.

Related Directive: **buffer_gen** and **max_load**
Command-Line-Option: **-yb**
Priority: Attributes have higher priority

4.4.3 Controlling Synthesis Parameters

Max. Load

This setting controls the buffer and pad generation algorithm for FPGAs. This value sets the default maximum fanout allowed on any signal in the design.

Related Directive: **buffer_gen, pad_gen** and **max_load**
Command-Line-Option: **-ym#**
Priority: Attributes have higher priority

Factor Cost

This parameter only affects the FPGA synthesis and it controls the aggressiveness of the logic factoring algorithms. Typically the default value of "1" produces the smallest and the fastest designs. In rare cases, though, increasing this parameter results in fewer factors and may result in faster designs.

Related Directive: **no_factor**
Command-Line-Option: **-f1#**
Priority: Attributes have higher priority

Node Cost

This parameter controls the aggressiveness of the Virtual Substitution algorithm. In general, the default of "10" is recommended. Setting this parameter to "0" will allow *Warp* to synthesize the design in a way that is compatible with the previous releases. Setting the parameter to its highest value of "11" will cause *Warp* to expend maximum effort in trying to collapse combinatorial equations. Setting this parameter to lower numbers will allow *Warp* to create nodes while being less aggressive in collapsing equations. For FPGAs, lowering the settings might produce significantly better results. Please refer to the Command Line Language chapter of the Reference Manual (-v option) and also the Synthesis Directives chapter of the Reference Manual (synthesis_off directive).

Related Directive: **synthesis_off**
Command-Line-Option: **-v#**
Priority: Attributes have higher priority

4.4.4 Setting Output Options

On the left side of the *Device* dialog box is the output section.

Output Format

For PLDs and CPLDs, *Warp* allows two types of JEDEC files. JEDEC Normal is the standard format which uses one byte to set each fuse. JEDEC Hex packs 8 fuses into one byte using hexadecimal notation. The selection of the JEDEC format depends on whether or not the programmer supports the JEDEC Hex format.

For FPGA devices, the only output available from *Warp* is the QDIF netlist which can be used as input to the place and route program called SpDE. SpDE in turn can produce a LOF file which can be used to program FPGA devices. See the Device Programming chapter for more information.

Related Directive: None
Command-Line-Option: **`-fh`**
Priority: N/A

Simulation Output

For PLDs and CPLDs, *Warp* can also produce a simulation model that allows the user to simulate the design with timing information. A list of simulators is available in the *Post-JEDEC Sim:* menu.

For pASIC Devices, the target simulator must be selected from within SpDE. See Chapter 5, "SpDE," for more information on how to use SpDE.

Related Directive: None
Command-Line-Option: None
Priority: N/A

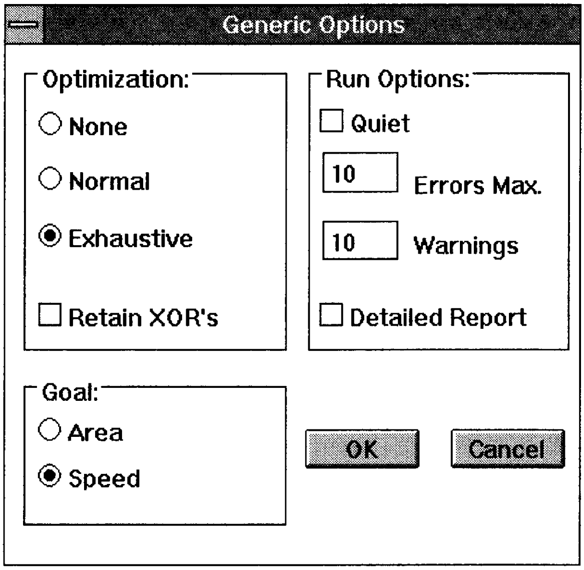
Unused Outputs

This option instructs the fitter to program unused I/O pins so that they either drive a logic level "0" or "1" or are simply three-stated.

Related Directive: None
Command-Line-Option: **`-ful`** or **`-fuh`** or **`-fuz`**
Priority: N/A

4.5 Generic Options

Various device independent compiler options can be selected using the *Generic* button in the project window.



Generic Options

Optimization:
○ None
○ Normal
● Exhaustive
□ Retain XOR's

Run Options:
□ Quiet
10 Errors Max.
10 Warnings
□ Detailed Report

Goal:
○ Area
● Speed

Figure 4-6 Generic Options dialog box

4.5.1 Optimization Level

4.5.1.1 Effort

This option sets the effort that *Warp* spends optimizing the design. For most designs, Exhaustive optimization (**-o2**) is recommended. Exhaustive optimization is a super set of Normal optimization and performs additional tasks such as state machine optimization and don't-care optimization. Exhaustive optimization is required for other technology mapping functionality such as logic factoring and multiplexor optimization.

Related Directive: **opt_level**
Command-Line-Option: **-o0** or **-o1** or **-o2**
Priority: Attributes have higher priority

4.5.1.2 Retain XORs

This option is provided for compatibility with previous releases and might be useful in special conditions. This option causes *Warp* to retain XOR operators found in the design instead of collapsing them into their fanout. The fitter or place and route programs would therefore handle these XORs and map them appropriately to the target device.

Related Directive: None
Command-Line-Option: **`-xor2`**
Priority: N/A

4.5.2 Synthesis Goals

With this option, the user can set the default goal for synthesis of datapath operators found within the design.

Related Directive: **`goal`**
Command-Line-Option: **`-ygs`** or **`-yga`**
Priority: Attributes have higher priority

4.5.3 Error/Warning Message Options

4.5.3.1 Quiet

In verbose mode, the *Warp* synthesis engine displays the equation, signal, or component being processed as it is being processed. If this option is enabled, *Warp* suppresses many of the messages that might otherwise be displayed during compilation and synthesis. On the PC (Windows 3.1 or Windows 95) platforms, enabling quiet synthesis will help conserve system resources and also allows the synthesis task to be completed much faster. Quiet mode is the default operating mode for *Warp*.

Related Directive: None
Command-Line-Option: **`-q`**
Priority: N/A

4.5.3.2 Max. Errors

This setting forces the *Warp* compiler to abort once the error limit has been reached. This can be especially useful for large VHDL files.

Related Directive: None
Command-Line-Option: **-e#**
Priority: N/A

4.5.3.3 Max. Warnings

This setting forces the *Warp* compiler to abort once the warning limit has been reached.

Related Directive: None
Command-Line-Option: **-w#**
Priority: N/A

4.5.3.4 Detailed Report

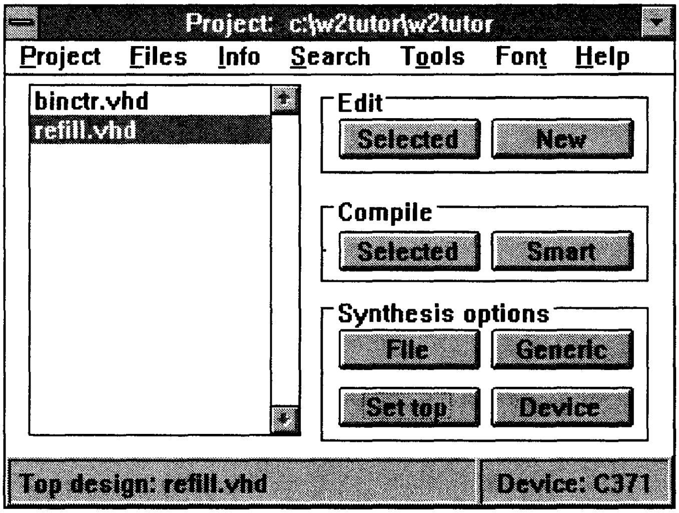
This setting, when enabled, forces the *Warp* compiler to produce a very verbose report file. For large designs, this option produces very large report files.

Related Directive: None
Command-Line-Option: **-yv#**
Priority: N/A

4.6 Compiling a Design

A design may be compiled in two ways:

- Select the desired files in the list box and then click on the *Selected* button.
- Click on the *Smart* button. Galaxy will recompile those files that have been modified since the last compilation. If any compilation options have been changed, however, Galaxy will have to recompile all the files.



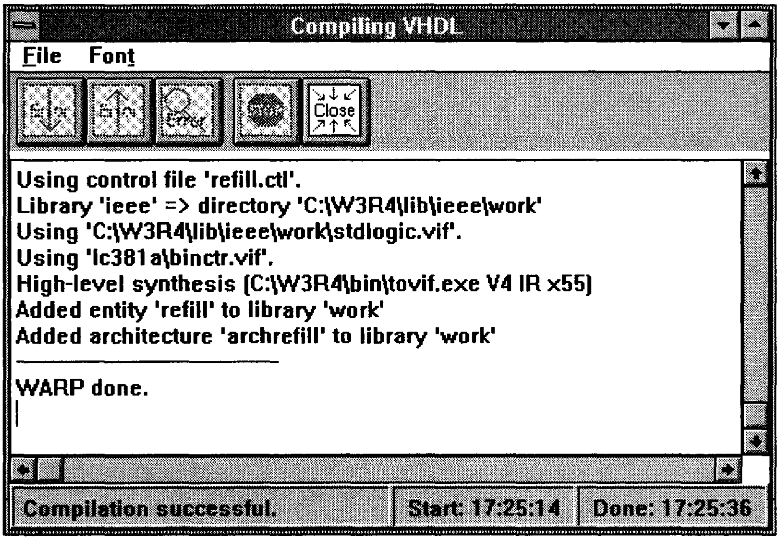
binctr.vhd
refill.vhd

Top design: refill.vhd

Device: C371

Figure 4-7 The Galaxy window showing the buttons for compilation

In either case, Galaxy opens up a *Compile* window or clears out the *Compile* window which is already open. This window displays the messages from the *Warp* VHDL compiler and also shows the time that the compilation started, along with the current time.



```
Using control file 'refill.ctl'.
Library 'ieee' => directory 'C:\W3R4\lib\ieee\work'
Using 'C:\W3R4\lib\ieee\work\stdlogic.vif'.
Using 'lc381a\binctr.vif'.
High-level synthesis (C:\W3R4\bin\tovif.exe V4 IR x55)
Added entity 'refill' to library 'work'
Added architecture 'archrefill' to library 'work'
______________________
WARP done.
```

```
Compilation successful.    Start: 17:25:14    Done: 17:25:36
```

Figure 4-8 A *Compile* window showing a successful compilation

To abort the compilation, click on the *Stop* button on the toolbar. The *Close* button also aborts the compilation.

The user may change the size and location of the *Compile* window as well as the font used to display messages. Galaxy remembers these choices and stores them in the project file.

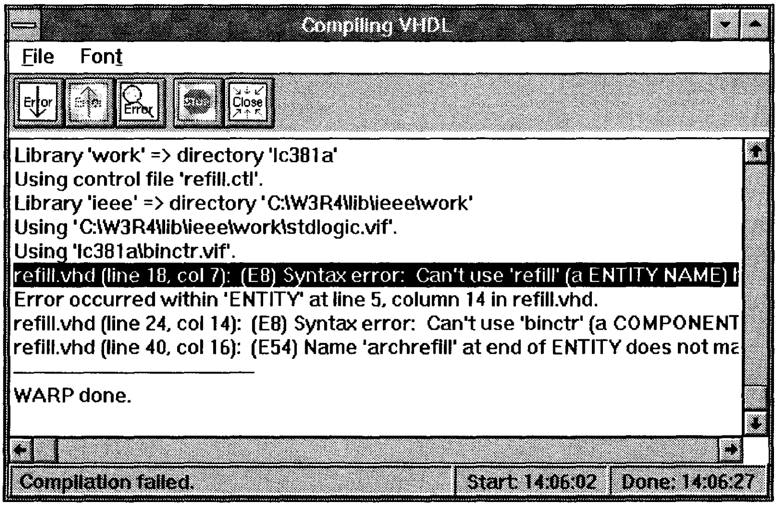
4.6.1 Report Files

When *Warp* has finished running, the user may save the messages to a file by selecting *Save as* under the *File* menu.

The user may also view the report files *Warp* generates. The *.vhd* file of the report file to be seen must be selected from the *Project* window and then *Report file* selected from the *Info* menu.

4.6.2 Error Tracking

If the design has errors, they will appear in the message area of the *Compile* window. Clicking on an error or using the two leftmost buttons of the *Compile* window's toolbar will highlight one of the errors.



```
Library 'work' => directory 'lc381a'
Using control file 'refill.ctl'.
Library 'ieee' => directory 'C:\W3R4\lib\ieee\work'
Using 'C:\W3R4\lib\ieee\work\stdlogic.vif'.
Using 'lc381a\binctr.vif'.
refill.vhd (line 18, col 7):  (E8) Syntax error:  Can't use 'refill' (a ENTITY NAME) h
Error occurred within 'ENTITY' at line 5, column 14 in refill.vhd.
refill.vhd (line 24, col 14):  (E8) Syntax error:  Can't use 'binctr' (a COMPONENT
refill.vhd (line 40, col 16):  (E54) Name 'archrefill' at end of ENTITY does not ma
________________________

WARP done.
```

Figure 4-9 A *Compile* window showing an error from compilation

A press of the *Return* key or a click on the magnifying glass button of the toolbar brings up an edit box containing the file where the error is and places the cursor near the error.

4.7 Library Management

4.7.1 What Is a Library?

A VHDL library is a repository of compiled design units which may be used within other VHDL files. These consist of packages, entities, and architectures.

Warp implements this concept by storing the compiled files, along with an *index* file, inside a separate directory. The directory name will be referred to as the "physical" name of the library. *Warp* provides a facility for assigning a "logical" name to a given library; this is the name specified in a USE clause in the designs.

4.7.2 The Library Window

To open Galaxy's library manager, select *Libraries* under the *Files* menu. A window will appear with three lists: Libraries, Designs, and Units.

The leftmost list, *Libraries,* lists the library directories within the project's directory. The user may select one library from this list.

The middle list, *Designs,* lists the design files within the library that is selected in the *Libraries* list. The user may select one of these designs.

The rightmost list, *Units,* shows the VHDL design units (packages, entities, and architectures) within the design file selected in the middle list.

To edit one of the design files, double-click on it. To locate a particular design unit in the file, double-click on the unit name.

4.7.3 Creating a Library

To create a library, select *Create library* under the *File* menu. A name prompt appears so that Galaxy can create the library as a sub-directory of the project's directory.

4.7.4 Deleting a Library

Select the library in the leftmost list and then choose *Delete library* under the *File* menu.

4.7.5 Removing a Design from a Library

To remove a particular design from a library, select the file in the middle list, and then choose *Remove design* under the file menu. The design will be removed from the library, but the original file will not be deleted.

4.7.6 Compiling Design Files to a Library

If the library does not already exist, then follow the instructions in "Creating a Library" above.

In the *Project* window, select the desired file to compile and then click on the *File* button in the Compiler Options area. An options box will open.

If the *Top Design* box is checked, click on it so that it is off. Then find the desired target library in the list and select it.

4.7.7 Using Design Units from a Library

Suppose that a design file has been compiled into the library *logic* and that the file contains a package called `gates_pkg`. To use this in another design, type the following:

```
LIBRARY logic;
USE logic.gates_pkg.all;
```

4.7.8 Assigning a Logical Name to a Library

Typically, when *Warp* compiles a design, any libraries that are referenced in the design are searched for in the current directory first, and then the system directory *($CYPRESS_DIR\lib)*. But if you have defined your own libraries as described in the above sections, and if these libraries do not exist in the current directory, it will be necessary for you to associate that directory with a logical name. To view all the current associations and to add/modify/delete these associations, use the *Files->Assign Library...* menu item in the Library Window.

Consider the following example, where main design is in directory *A* and you have a library named 'logic' defined and compiled in directory *B*. If library 'logic' has been compiled successfully, there should be a directory *A\logic* where the compiled design files will be placed by *Warp*.

While editing the project for your main design, using the 'Add' button in the Library Assignment dialog box, enter the name *(logic)* and the path *(..\B\logic)*. Also note that Library assignment allows the name to be anything you wish. In other words, even though the original library was created and compiled as the library *logic*, *Warp* allows this library to be assigned to any VHDL legal name.

4.8 Integrated Editor

Galaxy contains a simple text editor which can handle large files, locate errors, and help the user write VHDL.

4.8.1 Editing an Existing VHDL File

To edit an existing file, select it in the list of the *Project* window and then click on the *Selected* button in the Edit area. If more than one file is selected, Galaxy opens an edit box for each.

A file can also be edited by double-clicking on it.

4.8.2 Editing a New File

To create a new file, click on the *New* button in the Edit area. After it is saved, a file may be added to a project with the *Add* option under the *Edit* menu.

4.8.3 General Editing

The Galaxy edit box provides the following features:

4.8.3.1 File

New clears out current contents of edit box.

Open opens a file.

Save saves the current file.

Save as saves into a new file.

Print prints the contents of the edit box.

Print setup sets up the printer.

Exit dismisses the edit box.

4.8.3.2 Edit

Cut cuts the current selection and saves it in on the clipboard.

Copy copies the current selection to the clipboard.

Paste pastes from the clipboard to the current edit point.

4.8.3.3 Search

Find brings up a dialog box for a search.

Find next finds the same string again.

Replace brings up a dialog to do search-and-replace.

Replace current.

Replace the last-found text.

4.8.3.4 Font and Style

The font or style of the edit box may be changed globally. Galaxy remembers the choices and stores them in the project file.

4.8.4 VHDL Browser

Selecting *Browse* under the *VHDL* menu brings up the VHDL browser window.

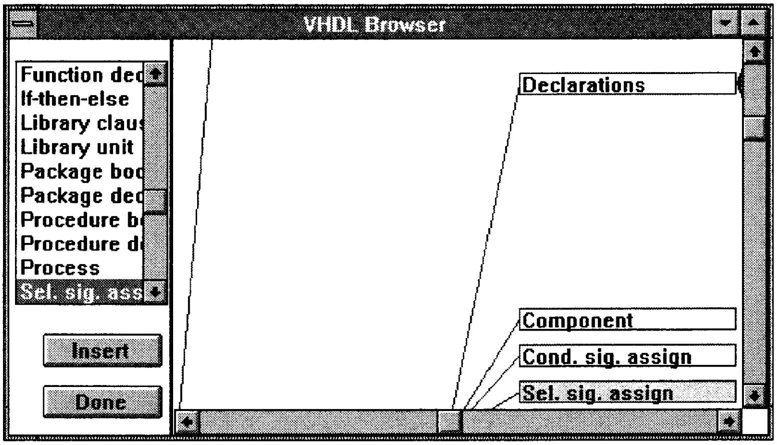


Figure 4-10 The VHDL Browser

The area to the right in the window contains a syntax tree for the VHDL language.

The list box to the left contains an alphabetical list of the VHDL constructs for which Galaxy has templates.

A template can be brought up in three ways:

- Double-click on the item in the list box.
- Select the item in the list and click on *Insert*.
- Click on an item in the syntax tree.

Some items in the syntax tree do not have templates. A box will turn yellow if it has a template associated with it.

More of the syntax tree is visible by enlarging the window. To scroll through the syntax trees, press and drag the left mouse button on an empty part of the tree's area.

4.9 Simulation

The user interface for selecting a simulator for post place and route simulation depends on the device. For FPGAs, this selection must be made in SpDE. For all other devices, use the Device dialog box to select a simulator of your choice. Please refer to the Simulation Chapter for more detail.

4.10 Back-Annotation

After a successful place and route, Galaxy allows automatic back-annotation of placement information such as pin numbers (Pins) and macrocell locations (Nodes). Back-annotation is possible for the top level design file. Use the *Files->Annotate...* menu item to invoke back-annotation. The back-annotation dialog box allows you to back-annotate pin and/or nodes. Upon clicking on *OK,* Galaxy will either create or modify the control (*ctl* extension) file and place the necessary synthesis directives.

4

Chapter 5

SpDE

5.1 Introduction

This chapter describes the following topics in *Warp's* SpDE tool kit:

- SpDE Design Flow
- SpDE Graphical Interface: the SpDE Window
- SpDE Design Tools
- SpDE Analysis Tools
- Design Considerations: Speeding Up High-Fanout Nets

Warp's SpDE tool kit provides a complete suite of FPGA place and route tools, allowing the user to complete the FPGA design process, to perform timing analysis on completed FPGA designs, to generate simulation models for timing simulation, and to generate programming files for device programming.

5.1.1 Design Placement and Routing

The final step in the FPGA design flow is to map logic fragments generated by the *Warp* compiler into logic cells of the target FPGA device. This process is called place and route. SpDE provides automatic placement and routing, requiring virtually no user intervention.

5.1.2 Design Viewing and Path Analysis

After automatic placement and routing, SpDE's Physical Viewer displays the graphed results. Along with the Path Analyzer, the user performs static timing analysis on critical paths and highlights them in the Physical Viewer.

5.1.3 Design Simulation and Programming

After design completion and analysis, the user can choose to generate simulation models for timing simulation in many environments for device programming.

5.2 SpDE Design Flow

The following is a typical SpDE design flow. For details on the description of each tool, please refer to Section 5.4, "SpDE Design Tools."

5.2.1 Starting SpDE

To start SpDE, double-click on the *Place&Rte* icon in the Workview PLUS Cockpit (on IBM PCs and compatibles) or the Powerview Cockpit (on UNIX workstations).

5.2.2 Importing Files

Warp designs targeted for pASIC designs are first compiled and synthesized by the *Warp* compiler into logic fragments and then saved as QDIF files. These files have to be imported into SpDE before placement and routing can take place.

To import a QDIF or EDIF file, use:

File ->Import QDIF / EDIF

During import, the Design Verifier is automatically invoked, and it checks for design errors, architectural violations, and potential problems.

5.2.3 Running the Tools

Once a QDIF file has been imported, the user starts the SpDE design tools by using the following:

Tools-> Run / Tools...

A summary of the available tools and their respective tasks are listed below.

Table 5-1 Summary of SpDE tools

SpDE Tool	Function
Logic Optimizer	Logic Optimization
Placer and Router	Automatic Placement & Routing
Delay Modeler	Timing Analysis
Back-Annotation	Simulation Model Generation
Automatic Test Vector Generator	Test Vector Generation
Sequencer	Programming File Generation

5.2.4 Design Viewing and Path Analysis

After automatic placement and routing, the user can use the Physical Viewer and the Path Analyzer to view the physical placement and routing of the design logic and to examine the timing for critical paths. The Path Analyzer can be invoked as follows:

Tools->Path Analyzer

The user can also examine resource utilization and the report file by using:

Info-> Cell Utilization.../ Report File...

5.2.5 Save and Exporting

After automatic placement and routing, the final design implementation can be saved as a *.chp* file. This can be done by selecting the following:

File->Save/ Save As...

A device programming file can be exported in LOF format by selecting the following:

File ->Export LOF...

File Formats

Here are the various file formats that SpDE uses as input or generates as output.

Table 5-2 SpDE file formats

File Type	Format(s)	Description
Input	QDIF (*.qdf*) EDIF (*.ed**)	*Warp* and some third-party tools (e.g., Exemplar's Galileo) generate QDIF files, while others (e.g., Synopsys'® Design Compiler™) generate an EDIF file as input to SpDE.
Output	CHIP (*.chp*)	Post-place and route designs are saved in this format.
Simulation	varies	Different types of simulation timing models are generated depending on the target simulator. Refer to Chapter 8, "Simulation."
Programming	LOF (*.lof*)	For use with the Cypress *Impulse3* or 3rd party device programmers.

5.3 SpDE Graphical Interface: The SpDE Window

5.3.1 The File Menu

The *File* menu includes commands related to creating, opening, and saving completed designs; importing synthesized designs for placement and routing; exporting programming files; printing the physical view of the layout; and exiting SpDE.

New clears any current design and initializes SpDE to operate on a new design.

Open brings up a dialog box for selecting an existing CHIP (*.chp*) file. The specified CHIP file will then be loaded, replacing the current design.

Save saves the currently completed design as a CHIP file.

Save As brings up a dialog box, allowing the user to save the current completed design as a CHIP file with a user-specified name. This menu item gives the user the capability of saving backup or reference copies of a design.

Import loads files generated by *Warp* Galaxy or other synthesis tools in QDIF or EDIF format for placement and routing.

Export LOF creates a Link Object Format (LOF) file with the specified name. This file contains all the data required to program and test pASIC devices. The file may be compressed at the user's option. This file can be used to program the pASIC on a device programmer.

Print Setup (PC version only) invokes the printer setup dialog box for setting the default printer driver.

Print (PC version only) prints the current physical view using the parameters set in Print Setup.

Exit terminates SpDE and prompts the user to save the design if it has been modified since the last save. For PC users, double-clicking the Control-Menu box (in the upper left corner of the SpDE window) is a shortcut for *Exit*.

Below the *Exit* menu item, the last five accessed files are listed (numbered 1-5). This provides a shortcut for the *Open* and *Import* commands. Clicking on a CHIP file from this list opens the file; clicking on another file type from this list imports the file.

5.3.2 The View Menu

The *View* menu (Figure 5-1) includes commands for manipulating the physical view of the design, highlighting specific nets, and specifying physical view options.

Zoom In magnifies the physical view of the current design. The user can simply click on a specific spot as the center of the view and zoom in by a scale factor of 1.25, or click and drag the mouse pointer to define a viewing rectangle. The view adjusts to fit the specified rectangle as closely as possible. The shortcut key for *Zoom In* is Ctrl-Z.

Zoom Out de-magnifies the view. The shortcut key for *Zoom Out* is Ctrl-X.

Full Fit modifies the scale factor to fit a view of the entire pASIC layout on the screen. The shortcut key for *Full Fit* is Ctrl-F.

Normal Fit sets the scale factor to its initial value and centers the view on the selected position. The shortcut key for *Normal Fit* is Ctrl-N.

Preferences sets physical view options. These are discussed in greater detail in the next section, "Preferences."

Highlight Net allows the user to select nets to be highlighted in the physical view. This menu item is discussed in greater detail in the section on "Highlight Net" on Page 203.

Redraw redraws the physical view.

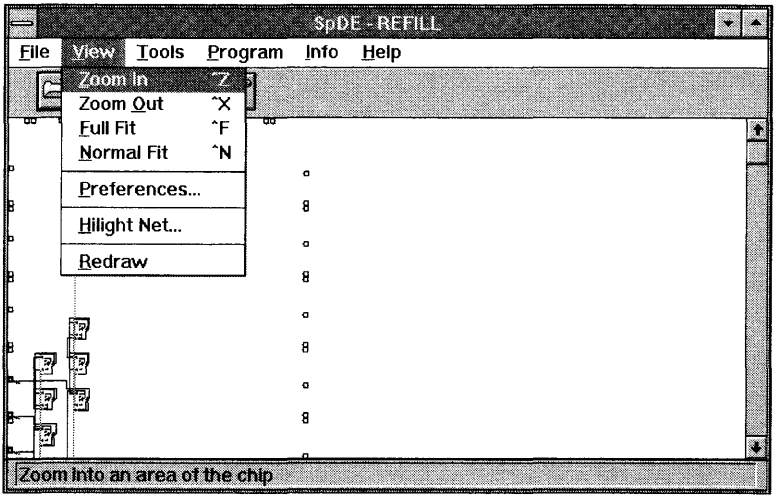


Figure 5-1 View menu

Preferences

Selecting *Preferences* from the *View* menu brings up the *Preferences* dialog box (Figure 5-2).

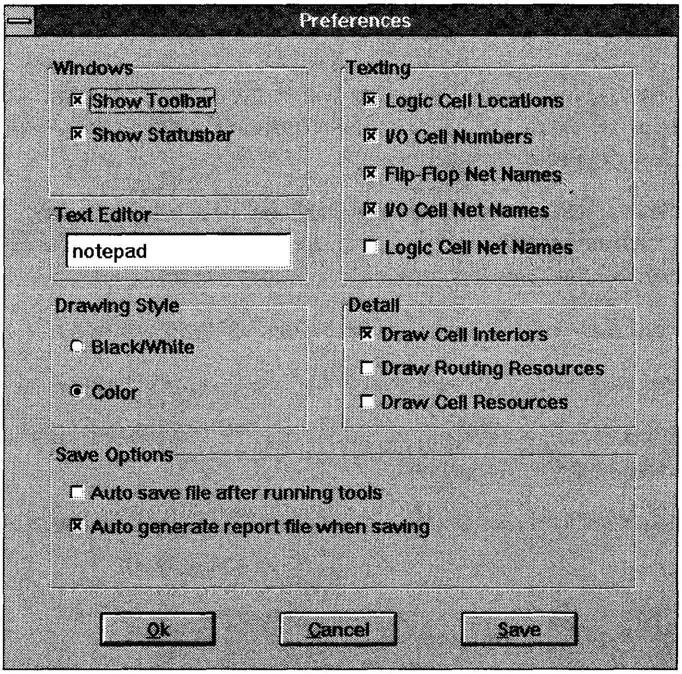
The Texting group of check boxes includes items relating to text labels in the physical view. Table 5-3 lists the check boxes and examples of the items they control. Text items can be turned off to increase redraw speed or to simplify the physical view.

Table 5-3 Examples of check box items

Item	Example
Logic Cell Locations	A1, A2, B1, C1, H12
I/O Cell Numbers	3, 12, 24, 42
Flip-Flop Net Names	specified in schematic
I/O Cell Net Names	specified in schematic
Logic Cell Net Names	specified in schematic

The *Flip-Flop Net Names* option only includes the net names of logic cell flip-flop outputs. The *Logic Cell Net Names* option includes the net names of all logic cell inputs and outputs.

The Drawing Style group of radio buttons controls the use of color in classifying interconnect. In *Black/White* mode, all wires are shown in black. In *Color* mode, short wires are shown in blue and black, express wires are shown in green, I/O wires are shown in red, and clock traces are shown in mauve.



Preferences

Windows

☒ Show Toolbar

☒ Show Statusbar

Text Editor

notepad

Drawing Style

○ Black/White

● Color

Texting

☒ Logic Cell Locations

☒ I/O Cell Numbers

☒ Flip-Flop Net Names

☒ I/O Cell Net Names

☐ Logic Cell Net Names

Detail

☒ Draw Cell Interiors

☐ Draw Routing Resources

☐ Draw Cell Resources

Save Options

☐ Auto save file after running tools

☒ Auto generate report file when saving

Figure 5-2 Preferences dialog box

The Detail group includes the *Draw Cell Interiors* check box. Deselecting this check box replaces the detailed logic cells with simple boxes, thereby increasing redraw speed.

Clicking *OK* accepts all preference settings for tools that are subsequently executed. Clicking the *Save* button does the same and also records the preference settings. These settings are then used the next time SpDE is invoked.

Highlight Net

Selecting *Highlight Net* brings up the *Highlight Net* dialog box (Figure 5-3) and redraws the Physical View in light gray, which allows the highlighted nets to stand out. The net list box on the left of the dialog box contains the names of nets not currently highlighted, while the net list box on the right contains the names of nets that are currently highlighted.

To highlight nets, select one or more nets from the box on the left, or specify a net name in the *Wildcard Selection* field on the left side of the dialog box, and click on the right arrow button.

To remove nets from the highlight list, select one or more nets from the box on the right, or specify a net name in the *Wildcard Selection* field on the right side of the dialog box, and click on the left arrow button.

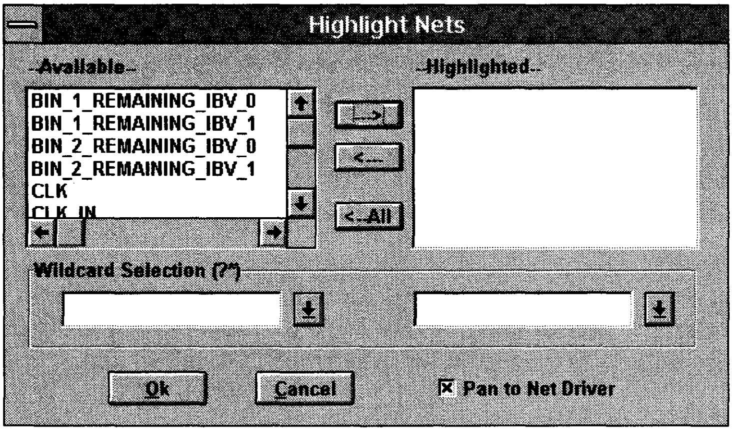
When using the *Wildcard Selection* fields, the wildcard characters "*" and "?" are accepted. The "*" character matches zero or more occurrences of any character. The "?" character matches a single occurrence of a character.

Double-clicking on a net name in either list moves it to the other list.

All nets can be removed from the highlight list by clicking on the *ALL* button.

Once in highlight net mode, the highlight status may be toggled by clicking directly on the desired nets in the physical view.

To exit highlight net mode, click *Cancel*. The Physical View will be redrawn in normal mode.



BIN_1_REMAINING_IBV_0
BIN_1_REMAINING_IBV_1
BIN_2_REMAINING_IBV_0
BIN_2_REMAINING_IBV_1
CLK
CLK_IN

☒ Pan to Net Driver

Figure 5-3 Highlight Net dialog box

5.3.3 The Tools Menu

The *Tools* menu is used to configure and run the optimizing, placing, routing, sequencing, delay modeling, back-annotation, and path analysis tools.

The *Tools* menu (Figure 5-4) contains three items: *Run Tools, Path Analyzer,* and *Options.*

.

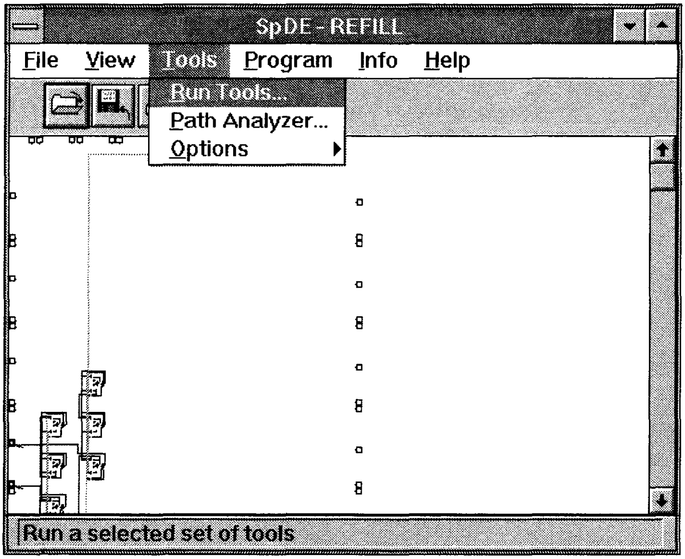

SpDE - REFILL
File View Tools Program Info Help
Run Tools...
Path Analyzer...
Options
Run a selected set of tools

Figure 5-4 Tools menu

Run Tools opens the *Select Tools to Run* window (Figure 5-5). This window is used to select which tools are to be run on the design. Two types of logic optimization can be run. This selection is made in the *SpDE Tools Options* window (Figure 5-6). Disabled tools are grayed out. Tools that have already been run are unchecked. Detailed descriptions of the Logic Optimizer, Placer, Router, Delay Modeling, Back-Annotation, and ATVG tools can be found in Section 5.4, "SpDE Design Tools." The Sequencer tool is only used to create data needed to program devices and has a short description in that section.

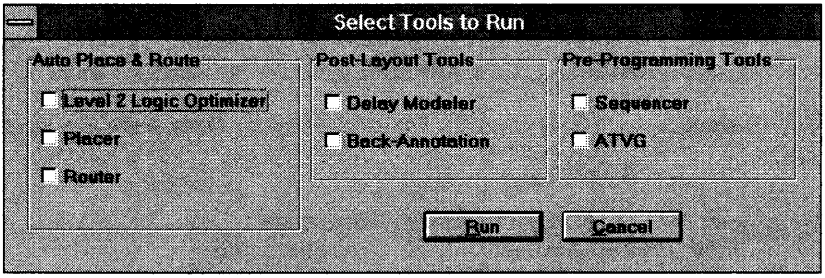
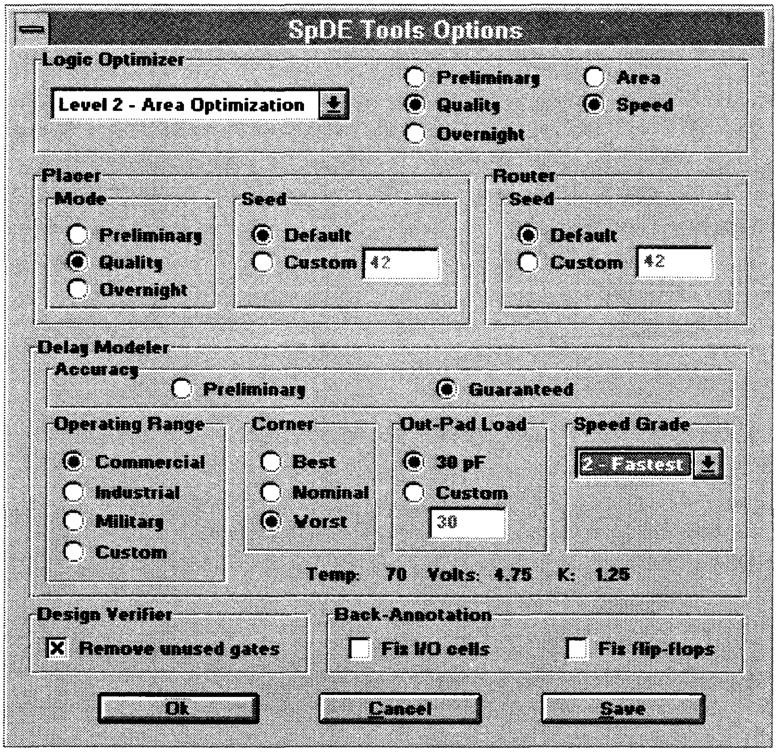


Figure 5-5 Select Tools To Run dialog box

The Path Analyzer can be used to determine operating frequency, setup and hold times, and clock skew. This menu item is discussed in greater detail in Section 5.5.2, Path Analyzer."

Options allows the user to select *General* or *Simulator* options. General options are the options for all tools. Simulator options affect only the Back-Annotation tool. The *General Tools Options* and *Simulator Options* windows are shown in Figures 5-6 and 5-7.



SpDE Tools Options

Logic Optimizer

Level 2 - Area Optimization

○ Preliminary ● Quality ○ Overnight

○ Area ● Speed

Placer

Mode: ○ Preliminary ● Quality ○ Overnight

Seed: ● Default ○ Custom 42

Router

Seed: ● Default ○ Custom 42

Delay Modeler

Accuracy: ○ Preliminary ● Guaranteed

Operating Range: ● Commercial ○ Industrial ○ Military ○ Custom

Corner: ○ Best ○ Nominal ● Worst

Out-Pad Load: ● 30 pF ○ Custom 30

Speed Grade: 2 - Fastest

Temp: 70 Volts: 4.75 K: 1.25

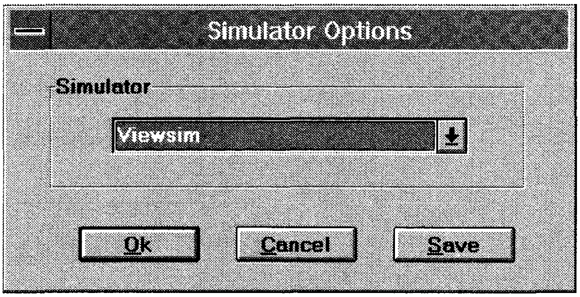
Design Verifier

☒ Remove unused gates

Back-Annotation

□ Fix I/O cells □ Fix flip-flops

Figure 5-6 General Tools Options window



Simulator Options

Simulator

Viewsim

Figure 5-7 Simulator Options window

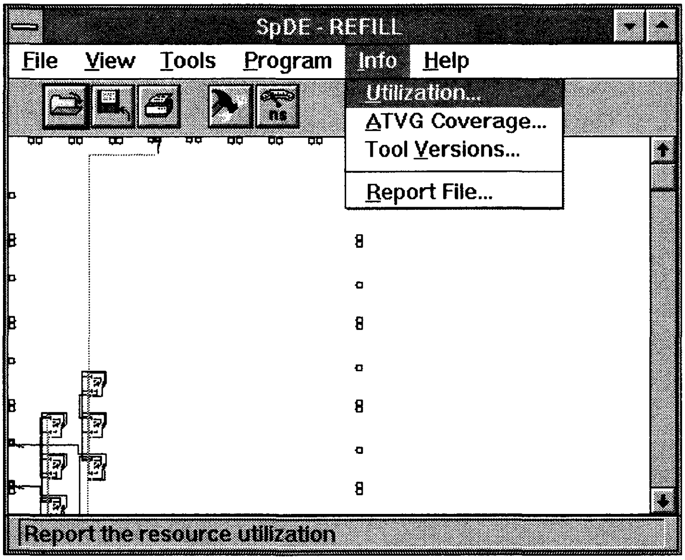
The *SpDE Tools Options* window is used to configure each of the SpDE tools. A detailed description of each tool's options can be found in the following sections of this chapter.

The *Simulator Options* window is used to select the simulator output that is desired. The *Back-Annotation* tool uses this information to produce the proper timing netlist. When the *Back-Annotation* tool is run, the file(s) appropriate to the selected simulation option are created. See Section 5.4.5, "Delay Modeler and Back-Annotation," for details.

5.3.4 The Info Menu

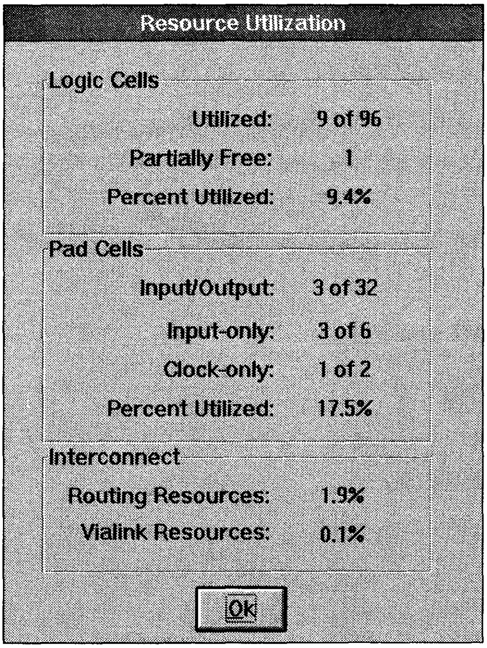
The *Info* menu (Figure 5-8) includes items that provide statistics and other information about a design.

Cell Utilization reports the number of cells of different types used in the design (Figure 5-9) as well as the number of logic, input-only, clock-only, and bidirectional cells. Also included is a count of "partially-free" cells; these are logic cells with a free AND fragment. These cells can accept any macro that can be implemented in a single AND fragment. This information is provided to allow fine-tuning of high-utilization designs.



Report the resource utilization

Figure 5-8 Info menu



Logic Cells

Utilized: 9 of 96

Partially Free: 1

Percent Utilized: 9.4%

Pad Cells

Input/Output: 3 of 32

Input-only: 3 of 6

Clock-only: 1 of 2

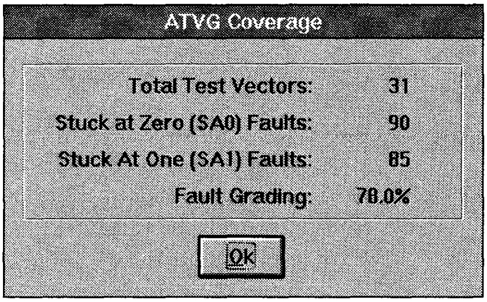
Percent Utilized: 17.5%

Interconnect

Routing Resources: 1.9%

Vialink Resources: 0.1%

Figure 5-9 Typical Cell Utilization dialog box



Total Test Vectors: 31

Stuck at Zero (SA0) Faults: 90

Stuck At One (SA1) Faults: 85

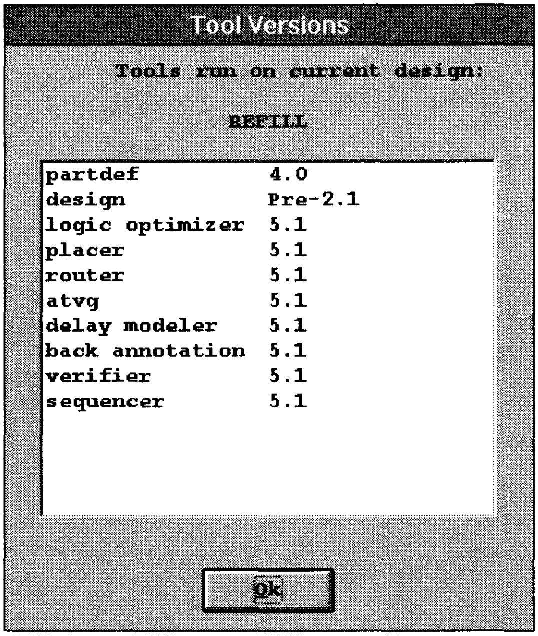
Fault Grading: 78.0%

Figure 5-10 Typical ATVG coverage dialog box

ATVG Coverage reports on the design's test coverage (Figure 5-10). These statistics may be used as guidelines to improve the design in order to increase test coverage. Of particular interest is the Fault Grading statistic, which indicates the quality of test coverage produced by ATVG.

Tool Versions is provided for diagnostic purposes. *Tool Versions* lists the tools that have been run on the current design, including the version number of each tool listed (Figure 5-11).

Report File displays the Cypress report file produced by the *Warp* compiler, including a SpDE tools appendix.



Tools run on current design:

REFILL

```
partdef            4.0
design             Pre-2.1
logic optimizer    5.1
placer             5.1
router             5.1
atvg               5.1
delay modeler      5.1
back annotation    5.1
verifier           5.1
sequencer          5.1
```

Figure 5-11 Typical Tools Versions dialog box

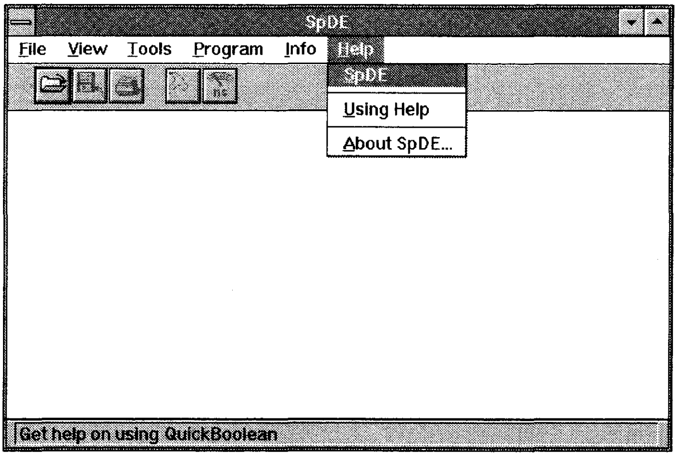
5.3.5 The Help Menu

The *Help* menu (Figure 5-12) includes commands for on-line help on SpDE.

SpDE invokes SpDE's on-line help.

Using Help provides introductory information on using the Help facility itself.

About SpDE provides information on the SpDE Toolkit, including the revision number of each tool and the configuration.



Get help on using QuickBoolean

Figure 5-12 Help menu

5.4 SpDE Design Tools

This section describes the SpDE design tools:

The Design Verifier analyzes the file for design errors, architectural violations, and potential problems.

The Logic Optimizer partitions designs into logic cells using sophisticated technology mapping algorithms.

The Placer takes the design from the Logic Optimizer and places the logic cells in optimal locations on the chip.

The Router connects I/O and logic cells, using the pASIC interconnect resources.

The Delay Modeler calculates delays for the Path Analyzer and writes the delays and delay scale to a file in the ViewSim simulator.

The Back-Annotation tool writes pin numbers and fixed flip-flop numbers to a file that CypBack uses to back-annotate schematics.

The Automatic Test Vector Generator generates test vectors that can be used to test pASIC devices after they have been programmed.

The Sequencer is the tool that generates the programming file used to program the pASIC devices.

5.4.1 Design Verifier

When a design is imported from *Warp* into SpDE as a *.qdf* file, the Design Verifier analyzes the file for design errors, architectural violations, and potential problems. If any exist, a window will pop up displaying the appropriate messages. The messages are categorized as follows, according to their severity:

Notes simply provide information which might be of interest to the user, such as the removal of unused gates. No user action is required.

Warnings provide information on potential design problems, such as excessive fanout, which could impact performance and might require user intervention.

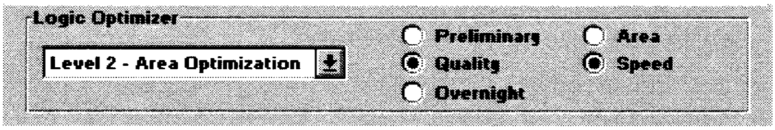
Errors provide information on design problems, such as floating inputs, that prevent a part from being programmed, even though the tools can still be run.

Fatal Errors provide information on extreme errors, such as excessive resource usage, which prevent the tools from being run.

5.4.2 Logic Optimizer

The Logic Optimizer is the first tool to be run after a design netlist has been loaded into SpDE. Logic optimization is the first step in automatic placement and routing (APR). The Logic Optimizer uses sophisticated technology mapping algorithms to partition the design into logic cells.

Three levels of optimization are available in the Logic Optimizer: Level 0 - Packer, Level 1 - Technology Mapper, and Level 2 Area/Speed Optimizer. The optimization levels can be selected from the *SpDE Tools Options* window (Figure 5-13), which appears after *Options/General* is selected from the *Tools* menu.



Logic Optimizer

Level 2 - Area Optimization

○ Preliminary ○ Area

● Quality ● Speed

○ Overnight

Figure 5-13 SpDE Tools Options window: Optimizer options

Level 0 Optimization: The Packer

The **Level 0 Packer** simply "packs" logic symbols (hard macros) from the imported QDIF file into logic cells, leaving all net connections intact. As many as four macro symbols may be packed into a single logic cell. No logic optimizations are done. The Level 2 Optimizer is the preferred logic optimizer in almost every case, but Level 0 optimization is provided for versatility and compatibility with old designs.

Level 1 Optimization: The Technology Mapper

The **Level 1 Technology Mapper** provides automatic logic cell optimization. The Technology Mapper introduces and removes inversion bubbles in order to improve capacity and performance. In more general terms, the Logic Optimizer merges gates when possible in order to achieve more efficient implementations. Because the Technology Mapper uses a more sophisticated algorithm than the Packer, the Mapper sometimes takes longer to run. For example, the Packer never takes more than a few seconds to run, but the Technology Mapper can take from a few seconds to several minutes, depending on the complexity of the design.

Level 2 Optimization: Area / Speed Optimization

The **Level 2 Area/Speed Optimizer** performs automatic logic cell optimization like the Technology Mapper. The selection of Area versus Speed is passed directly from *Warp* and is simply used here in SpDE to reflect which option was selected. Changing this option in SpDE has no effect on the results of your design.

Note – Internal nets may be deleted as a result of Level 1 and Level 2 optimization.

Logic Optimization Modes

For Level 1 and Level 2 optimization, the user may choose *Preliminary*, *Quality*, or *Overnight* mode from the *SpDE Tools Options* window. (Level 0 optimization is a simple, predictable algorithm that does not require different modes.)

Preliminary Level 1 optimization takes half the time of the Quality mode.

Quality Level 1 optimization is recommended for high quality results.

Overnight (or Exhaustive) mode produces slightly better results than Quality mode on some designs, but with a significantly longer run time.

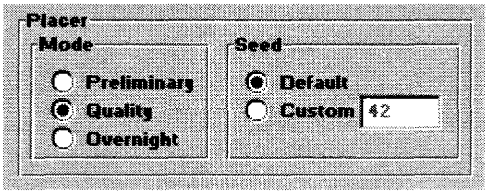
5.4.3 Placer

Placement is the second step in automatic placement and routing. The Placer places the logic fragments generated by the Logic Optimizer in optimal locations of the FPGA device to minimize routing delays. The Placer also allows users to fix the placement of I/O cells and registers. Fixing I/O cell placement can ease circuit board design, while fixing register placement offers precise control over internal routing delays.

Via the Path Analyzer, the Placer offers timing-driven placement. The user can enter timing constraints for specific nets in the Path Analyzer, and the information is then passed to the Placer to ensure that critical timing goals are considered with a higher priority during placement. When run from the Path Analyzer, the Placer determines optimal locations by looking at the nets' connecting logic cells and by looking at timing constraints added by the designer. (See "Timing-Driven Placement" on Page 216.)

Placer Options

Figure 5-14 shows the dialog box by which the user selects Placer options. Two kinds of options are available: the user can select the seed value for the Placer, the placement mode, or both.



Placer

Mode
○ Preliminary
● Quality
○ Overnight

Seed
● Default
○ Custom 42

Figure 5-14 Placer options

Placer Seed - The placement seed initializes the placement process and sets a starting point for the decisions made during automatic placement. The seed for the Placer is an integer between 1 and 32767. The user chooses either a custom seed or the default seed value (42). Changing the seed value sets a different starting point for the placer, which can produce a slightly different (and possibly improved) placement.

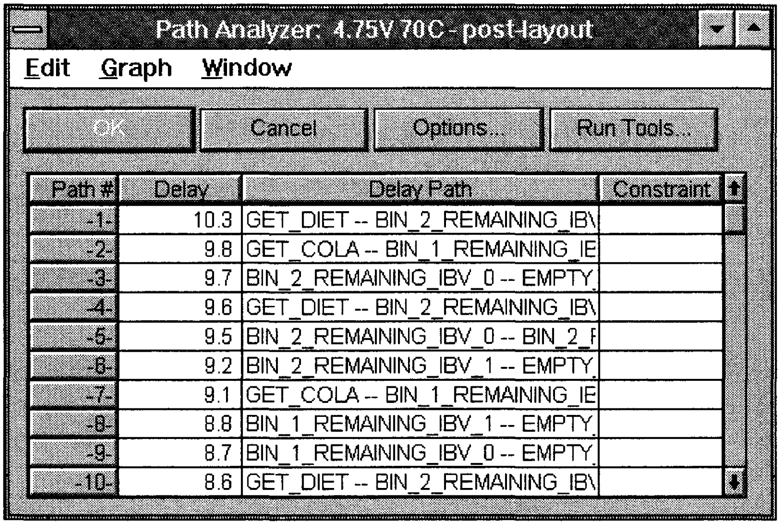
Placement Mode - Three placement modes are available: *Preliminary*, *Quality*, and *Overnight*. These three modes have the following characteristics:

- *Preliminary* placement is faster than quality placement, but usually by only a few minutes. Results are not as predictable and usually not as good as Quality placement. Cypress recommends at least Quality placement of designs before programming chips.
- *Quality* placement is the default placement mode. As its name implies, it produces high-quality placements.
- *Overnight* placement exists primarily for the curious. Results of Overnight mode placement are usually about 4% better than Quality placement, but at a significant cost in run time (about ten times). Thus, a design that places in six minutes in Quality mode will take about an hour in Overnight placement mode.

Timing-Driven Placement

The Placer works closely with the Path Analyzer (Figure 5-15) to provide timing-driven placement, an advanced technique issued to produce optimal results. User-specified constraints are fed from the Path Analyzer to the Placer. Paths not meeting the specified constraints are automatically boosted in priority until the constraint is met. Timing-driven placement allows the user to obtain peak performance without resorting to fixed placements.

Constraints can be entered for these paths directly in the Path Analyzer. Once the Path Analyzer has been run, paths not meeting the desired goal can be easily identified.



Path #	Delay	Delay Path	Constraint
-1-	10.3	GET_DIET -- BIN_2_REMAINING_IB\	
-2-	9.8	GET_COLA -- BIN_1_REMAINING_IE	
-3-	9.7	BIN_2_REMAINING_IBV_0 -- EMPTY_	
-4-	9.6	GET_DIET -- BIN_2_REMAINING_IB\	
-5-	9.5	BIN_2_REMAINING_IBV_0 -- BIN_2_F	
-6-	9.2	BIN_2_REMAINING_IBV_1 -- EMPTY_	
-7-	9.1	GET_COLA -- BIN_1_REMAINING_IE	
-8-	8.8	BIN_1_REMAINING_IBV_1 -- EMPTY_	
-9-	8.7	BIN_1_REMAINING_IBV_0 -- EMPTY_	
-10-	8.6	GET_DIET -- BIN_2_REMAINING_IB\	

Figure 5-15 Using timing constants in the Path Analyzer

Hint – It is important to set the constraints realistically—set each constraint at or just slightly below the required value. One of the keys to timing-driven placement is the concept of "good enough." Once a critical path has met its constraint, the Placer boosts the priority elsewhere in order to optimize all critical paths.

For each path with a constraint, the Placer estimates the delay throughout the placement process. If a constraint is met, the Placer continues to optimize the nets in the path normally. If a constraint is not met, the Placer boosts the priority of the nets in the paths; the boost in priority is proportional to the difference between the constraint and the estimated value. In other words, paths near their constraints are boosted less than paths far from their constraints.

Hint – Constraints should be added only where required. The dynamic delay estimation mentioned above adds work to the placement process. Each constraint specified slows the placement process. The design may have to be restructured in order to achieve the desired performance.

Constraints are stored with the design database. Once the constraints have been specified, all subsequent Placer runs operate in timing-driven mode. This can be verified during placement from the *SpDE Status* window—under normal placement the heading is Placer, while under timing-driven placement the heading is Timing-Driven Placer.

Note – Constraints are not saved to a separate file, so that each time a QDIF file is imported into SpDE, all constraints are cleared.

Fixed Placement

Although the Placer automatically determines placement for logic cells and I/O pads, it also supports fixed assignments of both I/O and flip-flops when required. The Placer does not modify fixed assignments by the designer.

Fixing the Placement of I/O Pads

Design constraints sometimes require some or all I/O cell locations to be fixed. For example, an existing printed circuit board (PCB) might dictate a precise pinout. Alternatively, a high-speed PCB might require fixing a small number of critical pins in order to limit skew. The SpDE Placer can handle these cases.

To fix I/O pad (i.e., pin) placements, use the **`pin_numbers`** attribute in VHDL. (See the *Warp Reference Manual* for syntax information about this attribute.)

Fixing the Placement of Flip-Flops

Design constraints rarely require logic cell locations to be fixed. To allow designers a greater degree of flexibility, however, the Placer allows some or all of a pASIC's flip-flop macros to be fixed. One scenario that would dictate fixed flip-flops would be if all the bits of an 8-bit register need to appear on the output pins with absolute minimum skew. The Placer, not realizing this design constraint, might sacrifice the skew on the outputs in order to produce a circuit that was faster overall. By manually fixing the flip-flops on logic cell locations adjacent to the output pins, the designer can meet the design constraint.

To fix flip-flop (i.e., internal) placements, use the **`fixed_ff`** attribute in VHDL. (See the *Warp Reference Manual* for syntax information about this attribute.)

Note – In order for placement to proceed correctly, the **`fixed_ff`** attribute must be applied to each element of a bus and not to the bus as a whole.

Two flip-flop macros cannot be assigned to the same location. The naming convention for pASIC logic cells assigns a character to each column and a decimal number to each row. SpDE verifies the uniqueness of location assignments for fixed placement with the Design Verifier.

Locking Down a Previous Pin Assignment

Sometimes an I/O placement must be "locked down." This means that for all subsequent place and route runs, the I/O pins do not change their locations.

To back-annotate I/O pin locations, the following steps need to be followed:

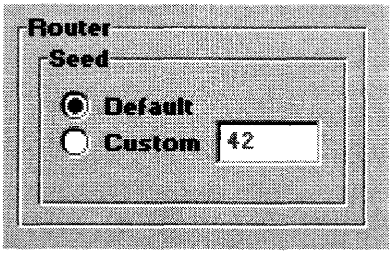
- Run the *Back-Annotation Tool* in SpDE (which creates and writes out placement information to *.atr* file).
- Select *Annotate...* from the Galaxy *File* menu to annotate to your control file, or select *Back Annotation...* from the *Cypress* menu in ViewDraw to annotate pins to your schematic.

5.4.4 Router

The Router employs highly optimized algorithms to connect I/O and logic cells using the pASIC interconnect resources. The Router's optimization capabilities minimize routing delays and routing resources required. This finely tuned arrangement produces excellent performance with high utilization.

Seed Value

Figure 5-16 shows the area of the *SpDE Tools Options* window that allows the user to set the seed value for the Router. The seed value may be an integer between 0 and 32767, inclusive. If the router seed is changed, the resulting route may be slightly different. This option is rarely needed but is provided for versatility.



Router

Seed

● Default

○ Custom 42

Figure 5-16 Router options

Interconnect Resources

The Router uses four different types of routing resources for fast and efficient connection between logic cells and I/O pads. These resources are **clock networks, express wires, quad wires,** and **segmented wires.**

Clock Networks: Two clock networks exist on each pASIC FPGA. Both of these dedicated resources are capable of connecting to the clock, set, or reset of any flip-flop in the FPGA. Each clock network must be driven by one of the clock cells (CKPADs) located at specific pins depending on the package used.

Segmented Wires: Segmented wires are the most abundant routing resource. These wires traverse the distance of one logic cell. High-drive pads cannot drive segmented wires, so the Router restricts nets on High-Drive pads to be routed on quad or express wires.

Quad Wires: Quad wires span four times the distance on the chip that segmented wires do (four logic cells). Quad wires may be used in routing any net in the design, including nets driven by High-Drive pads and parallel logic (see "Special Routing Cases," below).

Note – Quad wires are not available in all devices.

Express Wires: Express wires span the entire length or height of the FPGA device. They are used for high-fanout nets or for nets that need to travel across the device.

Special Routing Cases

High-Drive Pads: High-Drive Pads (HDPADs) must drive either quad wires or express wires. On devices that do not have quad wires, high-drive pads must drive express wires.

Parallel Logic: The pASIC architecture allows quad or express wires to be driven from higher-drive sources, such as HDPADs or parallel logic. Parallel logic is a logic configuration in which two identical gates (with the same inputs) have their output nets attached for higher drive capability. There is a restriction on the type of gates that can be tied in parallel. For more information refer to Section 5.6, "Design Considerations: Speeding Up High-Fanout Nets" and its discussion on **double-buffering**.

Hint – SpDE warns the user if he uses more than the recommended limit of high-drive nets (nets driven by high-drive pads or parallel logic). The router may have difficulty completing successfully in these cases.

5.4.5 Delay Modeler and Back-Annotation

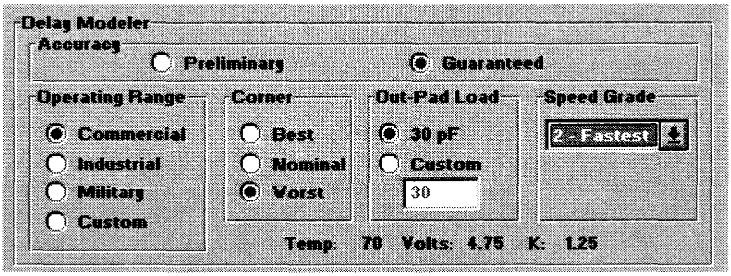
The Delay Modeler performs precise post-layout delay calculations using state-of-the-art circuit analysis techniques. Processing the complete results of place and route, the Delay Modeler analyzes the results of packing, placement, and routing to determine intrinsic delays and routing delays for the entire design. The Back-Annotation tool writes these precisely calculated delays directly into the timing simulation output for accurate simulation results.

Delay Modeler

The Delay Modeler performs a comprehensive timing analysis, accounting for load, slew rate, signal propagation, and intrinsic delay. The tool uses a precise model of the pASIC device and calculates the effects of fanout, packing, placement, and routing.

The Delay Modeler can perform best-case, nominal, or worst-case analysis. The results of the worst-case analysis account for process variation, temperature, and voltage.

The Delay Modeler may be run in *Preliminary* or *Guaranteed* mode (Figure 5-17). In *Preliminary* mode, the Delay Modeler uses statistical estimates for the impedance of ViaLinks in the device. In *Guaranteed* mode, the more accurate ViaLink impedances calculated by the Sequencer are used.



Delay Modeler

Accuracy: ○ Preliminary ● Guaranteed

Operating Range: ● Commercial ○ Industrial ○ Military ○ Custom

Corner: ○ Best ○ Nominal ● Vorst

Out-Pad Load: ● 30 pF ○ Custom 30

Speed Grade: 2 - Fastest

Temp: 70 Volts: 4.75 K: 1.25

Figure 5-17 Delay Modeler options

Hint – For a machine without a math co-processor, the run time of the Delay Modeler may be prohibitive during design debugging. In that case, the *Preliminary* setting should be used. Once the design is stable, the *Guaranteed* setting can be used to ensure proper timing performance.

The *Operating Range* radio button group controls the voltage and temperature ranges that the Delay Modeler uses. The default setting is *Commercial.* The *Custom* setting allows a user-specified temperature and voltage. See "Custom Temperature and Voltage," below.

The *Corner* radio button group selects the corner of the selected operating range. The default is *Nominal* (25 degrees Celsius and 5 Volts, regardless of the operating range selected). *Best* selects the lowest temperature and highest voltage in the selected operating range; *Worst* selects the highest temperature and lowest voltage in the operating range. Simulation should always be performed at the *Worst* corner.

The *Speed Grade* radio button group selects the pASIC speed grade to be analyzed.

The *Out-Pad Load* radio button group selects the capacitive loading on the output pins. The default is 30 pF. The *Custom* setting allows a user-specified load in the range of 0 pF to 150 pF for all output pins.

Note – The Delay Modeler has been tuned for peak accuracy within the recommended fanout ranges. High-fanout nets that produce fanout warnings are calculated to the highest accuracy possible, but these results are not guaranteed.

Custom Temperature and Voltage

To change the temperature and voltage setting for the Custom operating range, the user must edit the *spde.ini* file located in *$CYPRESS_DIR\spde\data*. For the Sun platform, this file should be named *.spderc* and should be in the home directory. The following lines should be changed:

Table 5-4 Edits to spde.ini and .spderc files

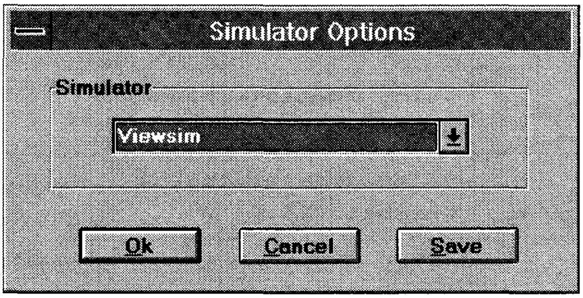
PC: *spde.ini*	SUN: *.spderc*
`[delay modeler]`	...
...	`delay modeler.customvccbest=5.0`
`CustomVCCBest=5.0`	`delay modeler.customvccnominal=5.0`
`CustomVCCNominal=5.0`	`delay modeler.customvccworst=5.0`
`CustomVCCWorst=5.0`	`delay modeler.customtempbest=25.0`
`CustomTempBest=25.0`	`delay modeler.customtempnominal=25.0`
`CustomTempNominal=25.0`	`delay modeler.customtempworst=25.0`
`CustomTempWorst=25.0`	...

The measurement units for CustomVCC variables are Volts. The measurement units for CustomTemp variables are Celsius degrees. The voltage range should not vary more than +/- 10% from nominal (5V). The *Best, Nominal,* and *Worst* extensions of these variables represent best, nominal, and worst process factors for the devices. SpDE selects which variable to use based on the *Corner* setting from the *SpDE Tools Options* window.

Simulation Back-Annotation

The Back-Annotation tool produces files to send timing and placement information back to the design entry and simulation tools. A variety of simulation tools are supported for back-annotated simulation. (The simulator with *Warp3* is ViewSim.) The simulator in SpDE can be specified by selecting the *Options Simulator* items from the *Tools* menu (see Figure 5-18).

.



Simulator: Viewsim

Figure 5-18 Simulator options for Back-Annotation

If the default simulator is changed by making a selection from the *Simulator Options* window, a click on the *Save* button will write this information into the *spde.ini* file (*.spderc* for Sun workstations). The table lists the files the Back-Annotation tool creates for each of several simulator settings.

Note – After changing the simulator, the user must still run the Back-Annotation tool to create the simulation netlist.

The Verilog® simulator also requires a primitive file, which describes the functionality of the primitive components specified in the *design.v* file. This primitive file is design-independent, and is shipped with SpDE. The filename for this primitive file is *$CYPRESS_DIR\spde\data\qlprim.v*, where *$CYPRESS_DIR* is the directory in which the *Warp* software is installed.

Table 5-5 Files created during back-annotation

Simulator Setting	**Files Created**	**Function**
Verilog	design.*v* *design.sdf*	verilog netlist delay back-annotation file
ViewSim	*design.vl* *design.dtb* *design.var*	intermediate file for spde2vl delay back-annotation file variable values for *.dtb* file
LMC EDIF	*design.edo* *design.kf*	LMC EDIF netlist device characteristics files
Intergraph EDIF	*design.edo* *design.kf*	Intergraph EDIF netlist device characteristics files

When back-annotating to ViewSim, the spde2vl program must be run after back-annotation to create the Viewlogic *.vsm* file needed for simulation.

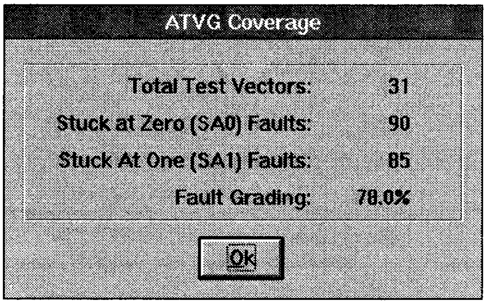
The icon that executes the spde2vl program from the Cockpit is labeled *pASIC->Vsim.*

The Back-Annotation tool also supports the back-annotation of pin placement information back to the source design. For information on this process, see Section 5.4.3, "Placer."

5.4.6 Automatic Test Vector Generator

The Automatic Test Vector Generator (ATVG) automatically generates vectors that can be used on devices after they have been programmed.

The ATVG tool makes use of an internal scan path in pASIC devices that allows values to be applied to and read from each flip-flop on the device. Once the ATVG tool has been run in SpDE, fault coverage can be ascertained by selecting *ATVG Coverage* from the *Info* menu.



Total Test Vectors: 31

Stuck at Zero (SA0) Faults: 90

Stuck At One (SA1) Faults: 85

Fault Grading: 78.0%

Figure 5-19 Sample ATVG Coverage window

Testing Overview

A major problem encountered in testing FPGAs is the inability to access directly the vast majority of circuit nodes from the chip periphery. Internal faults, therefore, must be made observable at the output pins by creating a set of input stimuli that will exercise the appropriate path and cause faults to appear as invalid output level changes.

Stuck-At Faults

Stuck-at-0 (SA0) and stuck-at-1 (SA1) fault analysis is an effective means of evaluating test sequences for their ability to detect potential faults in circuits. SpDE's ATVG tool uses an advanced testing technique capable of detecting these faults.

A SA0 fault results from a condition that holds a given signal at a logical 0 regardless of the signal being asserted on that line. Similarly, a SA1 fault is a line that is held at logical 1 regardless of the asserted signal. To illustrate this concept, consider the simple case of the 3-input AND gate shown in Figure 5-20. The truth table for this function is given in Table 5-6.

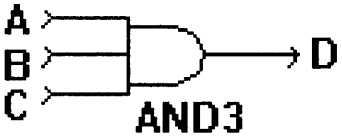


Figure 5-20 Stuck-at-Fault example

There are eight potential faults in this circuit, representing each node (A, B, C, D) stuck-at-0 and stuck-at-1. If A or B or C were stuck at logical low, D would also assume logical low, indicating an incorrect state for the last state (vector 8) of the truth table and hence a fault. Similarly, stuck-at-1 states for either A or B or C would show up as invalid outputs in vectors 4, 6, or 7 of the truth table, respectively.

Table 5-6 Truth table for 3-input AND gate

A	B	C	D	Vector	Tests
0	0	0	0	1	D for SA1
0	0	1	0	2	D for SA1
0	1	0	0	3	D for SA1
0	1	1	0	4	A, D for SA1
1	0	0	0	5	D for SA1
1	0	1	0	6	B, D for SA1
1	1	0	0	7	C, D for SA1
1	1	1	1	8	A, B, C, D for SA0

By applying appropriate inputs to the circuit, the SA0 and SA1 faults may be detected at node D. Table 5-7 lists these inputs and their expected responses. Sets of input conditions and resultant output states are commonly referred to as **test vectors**.

The table indicates the fault(s) detected for each test vector.

Table 5-7 Detected faults

A	B	C	D	Vector	Tests
0	1	1	0	1	A, D for SA1
1	0	1	0	2	B, D for SA1
1	1	0	0	3	C, D for SA1
1	1	1	1	4	A, B, C, D for SA0

As this example demonstrates, a total of four test vectors are required to find all the potential faults in this simple AND gate example. Actually, SA0 and SA1 faults for node D are indistinguishable from faults in the input signals A, B, and C. From a functionality point of view, this is not important since sufficient information is generated to confirm correct or incorrect operation of the circuit.

Fault Grading

Fault grading is a quantitative measure of the testability of a circuit and is defined by the following expression:

$$\mathbf{FG} = \frac{\text{SA0 faults detected} + \text{SA1 faults detected}}{\text{total number of detectable faults}}$$

In this example, the total number of detectable faults is six (the total number of detectable faults is simply two times the number of inputs). Note that faults at the output D are not counted because they are "covered" by faults at the inputs. The actual number of potential faults detected by these test vectors is also six, resulting in 100% fault coverage. Furthermore, these test vectors comprise an optimum set required to test the sample circuit, even though the truth table for it has the eight vectors shown in the table.

Design Considerations

The pASIC's testability features allow the designer to achieve a high degree of fault coverage. Testability can be further increased by designing with a few simple rules in mind.

- Avoid combinatorial loops. The ATVG tool cannot test combinational loops such as those shown in Figure 5-21. Logic driven by or driving these loops may be untestable.
- Using the output of a gate or flip-flop to clock, clear, or preset another flip-flop reduces testability.

  The ATVG tool tries to use all of the flip-flops in the FPGA device for testing purposes. Clocking, setting, or clearing a flip-flop by internal logic renders the flip-flop useless to ATVG, reducing testability. Examples of logic structures that reduce testability in this way are shown in Figure 5-21.

Although gated clocks reduce testability, a buffer or inverter in the path between an input pad and the clock of the flip-flop does NOT reduce testability. A flip-flop clock, clear, or set driven by a bi-directional pin (BiPAD) also reduces fault coverage.

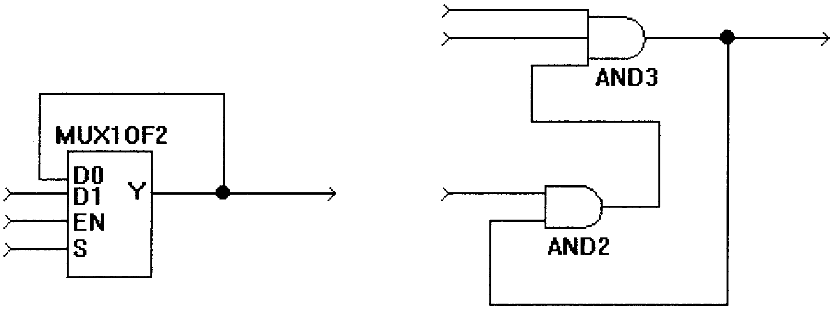


Figure 5-21 Examples of combinatorial loops (feedback)

- Certain input-only pins (HDPADs) should not be used to drive logic that controls an asynchronous set or reset of a flip-flop, as doing so either disables ATVG or reduces coverage.

  A subset of the input-only pins (HDPADs) should not be used to drive asynchronous sets or resets (of flip-flops) directly or through a logic path. If so, ATVG will be disabled or fault coverage will be reduced.

The table shows the pin numbers for different packages.

Table 5-8 Set/reset restrictions

Package	Input-pins (HDPADs) with restrictions driving sets or resets.	
	Disables ATVG	Lowers Coverage
44 pin PLCC	11	10
68 pin PLCC	17	16
68 pin CPGA	A7	B7
84 pin PLCC	22	21, 66
84 pin CPGA	C6	B7
100 pin TQFP	12	11, 65
144 pin TQFP	18	17, 93
144 pin CPGA	C8	B8, P7

Because of the test mode requirements of pASIC devices, certain input-only (also known as high-drive) pins cannot be used in a multiple HDPAD configuration without disabling ATVG, unless they are used only to drive flip-flop clock inputs. The table lists the high-drive pins that cannot be used in HD2PADs or HD3PADs without disabling ATVG.

Table 5-9 HDPAD restrictions

Package	Pins not to include in HD2PADs or HD3PADs, unless used as clock only (disables ATVG)
44 pin PLCC	10, 36
68 pin PLCC	16, 54
68 pin CPGA	B7, K7
84 pin PLCC	21, 66
84 pin CPGA	B7, K7
100 pin TQFP	11, 65
144 pin TQFP	17, 93
144 pin CPGA	B8, P7

5.4.7 Sequencer

The Sequencer is the tool that generates the programming file used to program the pASIC devices. This file is generated internally and must be written out using the *Export LOF*

selection in the *File* menu. The sequencer tool must be run before the LOF file can be written out.

5.5 SpDE Analysis Tools

After automatic placement and routing, SpDE allows the user to view logic cells and examine the timing of critical paths. Two tools are available for these purposes:

- Physical Viewer
- Path Analyzer

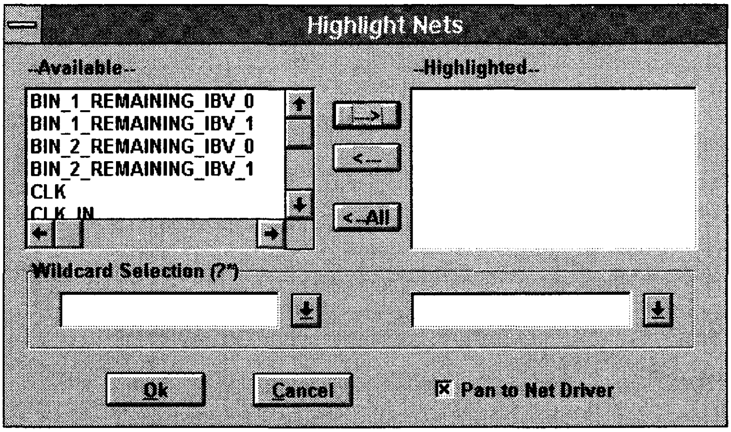
The following section discusses how to highlight critical nets for physical viewing using the Physical Viewer, as well as the calculation of critical timing parameters such as clock skew, operation frequency, setup time, and hold time using the Path Analyzer.

5.5.1 Highlight Net

Highlight Net mode helps the user analyze a design by highlighting and un-highlighting nets. Highlight Net mode cannot be used until a design has been placed and routed.

To open the *Highlight Nets* window (Figure 5-22), select the *Highlight Net...* item from the *View* menu in SpDE.

.



BIN_1_REMAINING_IBV_0
BIN_1_REMAINING_IBV_1
BIN_2_REMAINING_IBV_0
BIN_2_REMAINING_IBV_1
CLK
CLK_IN

☒ Pan to Net Driver

Figure 5-22 Highlight Nets window

To highlight nets, select one or more nets from the box on the left, or specify a net name in the *Wildcard Selection* field on the left side of the dialog box, and click on the right arrow button.

To remove nets from the highlight list, select one or more nets from the box on the right, or specify a net name in the *Wildcard Selection* field on the right side of the dialog box, and click on the left arrow button.

When using the Wildcard Selection fields, the wildcard characters "*" and "?" are accepted. The "*" character matches zero or more occurrences of any character. The "?" character matches a single occurrence of a character.

Double-clicking on a net name in either list moves it to the other list.

All nets can be removed from the highlight list by clicking on the *ALL* button.

Once in highlight net mode, the highlight status may be toggled by clicking directly on the desired nets in the physical view.

To exit highlight net mode, click *Cancel.* The Physical View will be redrawn in the normal mode.

Double-clicking on a net in the Highlighted box un-highlights the net in the physical view. Clicking on a highlighted wire in the physical view un-highlights the net.

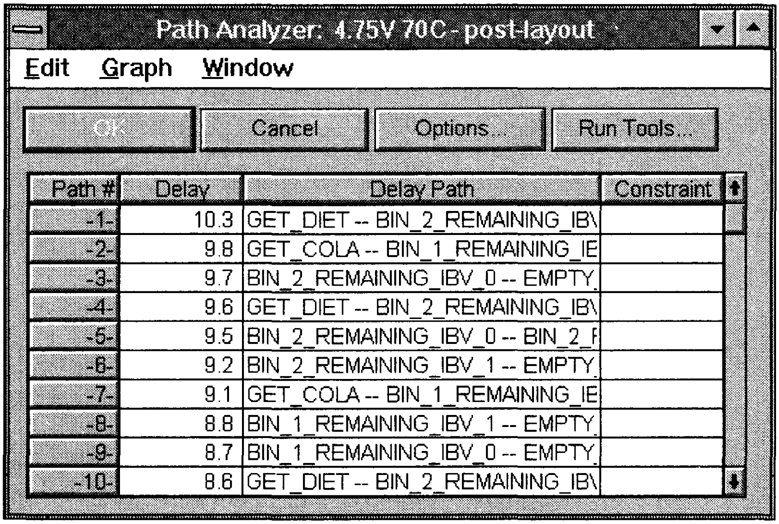
Pan to Net Driver

The *Highlight Nets* window contains a check box named *Pan to Net Driver*. When this box is checked, SpDE automatically pans to the driver of the net that is selected. This is true whether the net is selected by clicking on a wire in the physical view or by selecting a net from the available list.

5.5.2 Path Analyzer

The Path Analyzer is a powerful static timing analyzer that can be used to determine operating frequency, setup and hold times, and clock skew. The Path Analyzer performs static timing analysis of the circuit delays from the Delay Modeler. The Path Analyzer offers automatic analysis of all signals or of a user-specified subset of signals in the completed design. Working closely with the Physical Viewer, the Path Analyzer instantly identifies critical paths for optimization. Once the critical path has been identified, the user can use the Timing-Driven Placer to optimize the placement in order to achieve specified operating constraints.

To run the Path Analyzer, select *Path Analyzer* from the *Tools* menu. Results are displayed in a four-column spreadsheet format (Figure 5-23).



Path #	Delay	Delay Path	Constraint
-1-	10.3	GET_DIET -- BIN_2_REMAINING_IB\	
-2-	9.8	GET_COLA -- BIN_1_REMAINING_IE	
-3-	9.7	BIN_2_REMAINING_IBV_0 -- EMPTY	
-4-	9.6	GET_DIET -- BIN_2_REMAINING_IB\	
-5-	9.5	BIN_2_REMAINING_IBV_0 -- BIN_2_F	
-6-	9.2	BIN_2_REMAINING_IBV_1 -- EMPTY	
-7-	9.1	GET_COLA -- BIN_1_REMAINING_IE	
-8-	8.8	BIN_1_REMAINING_IBV_1 -- EMPTY	
-9-	8.7	BIN_1_REMAINING_IBV_0 -- EMPTY	
-10-	8.6	GET_DIET -- BIN_2_REMAINING_IB\	

Figure 5-23 Path Analyzer window

The *Path #* column is displayed in a push-button format, for reasons to be explained shortly. The *Delay* column indicates the delay in nanoseconds. For post-layout analysis, these delays are determined by the Delay Modeler. The *Delay Path* column displays the starting and ending nets of each path.

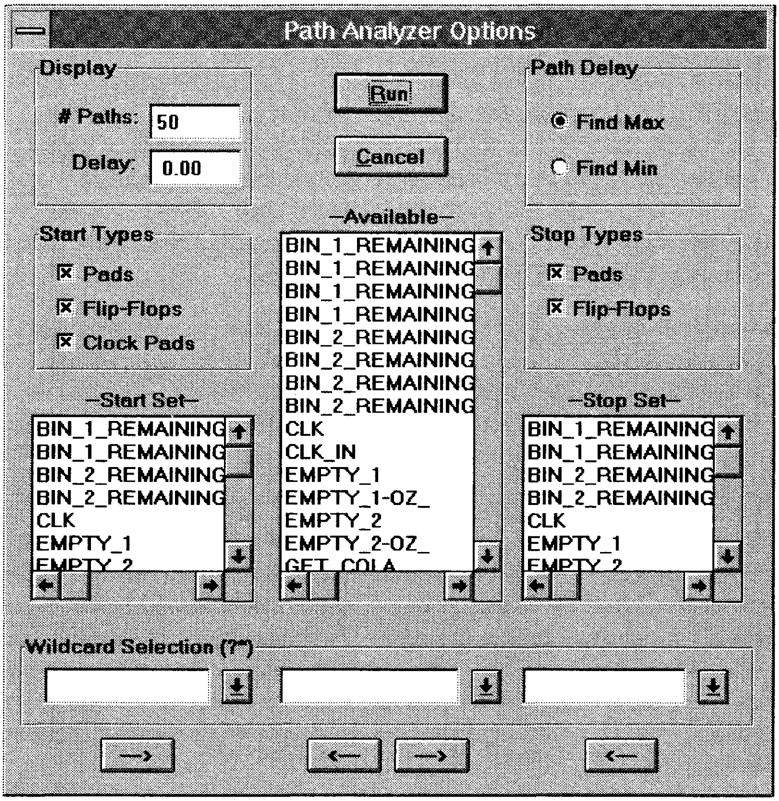
If the starting point is a pad, the net attached to the off-chip terminal on the pad will be used; if the starting point is a flip-flop, the net attached to the output of the flip-flop will be used. Likewise, if the ending point is a pad, the net attached to the off-chip terminal on the pad will be used; if the ending point is a flip-flop, the net attached to the output of the flip-flop will be used.

Expanding Paths

To expand a path into its component trails, position the cursor over the desired button in the *Path #* column and double-click. The Path # button changes from -1- to +1+ (assuming path number "1") to indicate that the path has been expanded. The component trails are indented and listed in blue to differentiate them from the other *Delay Path* rows. Each trail lists a delay value in nanoseconds, along with an R or F token to denote a rising- or falling-edge delay.

Path Analyzer Options

All Path Analyzer options are set from the *Path Analyzer Options* dialog box (Figure 5-24), which appears when the user clicks the *Options* button in the Path Analyzer window.



Path Analyzer Options

Display

# Paths: 50

Delay: 0.00

Path Delay

● Find Max

○ Find Min

Start Types

☒ Pads

☒ Flip-Flops

☒ Clock Pads

Stop Types

☒ Pads

☒ Flip-Flops

—Start Set—

BIN_1_REMAINING
BIN_1_REMAINING
BIN_2_REMAINING
BIN_2_REMAINING
CLK
EMPTY_1
EMPTY_2

—Available—

BIN_1_REMAINING
BIN_1_REMAINING
BIN_1_REMAINING
BIN_1_REMAINING
BIN_2_REMAINING
BIN_2_REMAINING
BIN_2_REMAINING
BIN_2_REMAINING
CLK
CLK_IN
EMPTY_1
EMPTY_1-OZ_
EMPTY_2
EMPTY_2-OZ_
GET_COLA

—Stop Set—

BIN_1_REMAINING
BIN_1_REMAINING
BIN_2_REMAINING
BIN_2_REMAINING
CLK
EMPTY_1
EMPTY_2

Wildcard Selection (?*)

Figure 5-24 Path Analyzer Options dialog box

The *Run* button at the top of the window re-runs the Path Analyzer with the newly specified options. The *Cancel* button returns to the Path Analyzer and discards any newly specified option selections.

The *Path Delay* group of radio buttons selects maximum or minimum path delays. Each trail along a given path includes a rising-edge delay and a falling-edge delay. If *Find Max* is selected, the Path Analyzer sums the larger of these edge delays at each trail; if *Find Min* is selected, the Path Analyzer sums the smaller of these edge delays. This selection does not change the operating conditions. (In other words, it does not change worst-case commercial to best-case commercial.) *Find Max* lists signals in order of longest delay to shortest. *Find Min* lists signals in order of smallest delay to longest.

The *Display* group determines the number of paths calculated and listed in the path analyzer spreadsheet. The # *Paths* entry limits the number of paths to the specified value. The *Delay* entry is interpreted with regard to the *Path Delay* setting—if *Find Max* is selected, paths are listed if their delay is greater than or equal to the specified value; if *Find Min* is selected, paths are listed if their delay is less than or equal to the specified value.

The remaining lower sections of the dialog box are used to select the *Start Set* and *Stop Set* that specify the desired paths. The *Start Set* list box specifies the starting nets for path analysis, while the *Stop Set* list box specifies the ending nets for path analysis. Providing specific *Start Set* and *Stop Set* information limits the amount of data in the spreadsheet report, making interpreting the results of the Path Analyzer easier.

The *Start Types* and *Stop Types* check boxes provide the easiest method for selecting the *Start Set* and *Stop Set* list box entries. By default, all of these check boxes are selected. The *Pads* check box selects all nets attached to the external terminals of all pads; this check box selects I/O pads, high-drive pads, and clock pads. The *Flip-Flops* check box selects all nets attached to the output terminals of all flip-flops. The *Clock Pads* check box selects all nets attached to the external terminals of any pad functioning as a clock (not only the internally buffered clocking networks).

Selecting one of these check boxes adds all of the appropriate nets to the desired set. Deselecting one of these check boxes removes all of the appropriate nets from the desired set. For example, if none of the *Start Types* check boxes are selected, then selecting the *Pads* check box adds all pad nets to the *Start Set* list box. Selecting the *Clock Pads* check box results in no change, as all pad nets are already selected. Deselecting the *Clock Pads* check box, however, removes the clock pad nets from the *Start Set* list box, leaving only non-clock pad nets.

Nets can be selected manually using the *Available* list box in the center of the dialog box. Select a net or nets in this list box, then click on one of the arrow buttons below the *Available* list box. Clicking on the left-arrow button adds the selected nets to the *Start Set* list box, while clicking on the right-arrow button adds the selected nets to the *Stop Set* list box.

Likewise, the *Start Set* and *Stop Set* list boxes can be "pruned" by selecting a net or nets and clicking on the arrow button below the list box involved.

Groups of nets can be selected using the combo buttons below each list box. In Figure 5-24, for example, the bus IB[0:3] can be selected by clicking in the combo button just below the *Available* list box and typing IB* or IB[?] and pressing the *Tab* key. Once the desired nets are selected, they can be acted upon using the arrow buttons, as described previously.

Note – When entering text to select nets from the Start Set, Available, or Stop Set lists, wildcards can be used. An asterisk ("*") represents one or more characters; a question mark ("?") represents a single character (e.g., **`addr*`** would select addr[0], addr[1], addr[2], etc.).

Graphing

The Path Analyzer provides essential information about the performance of the design. Occasionally, it is useful to view this information in graphical form. The Path Analyzer's Graph menu can be used to create two types of graphs: *Path vs. Delay* and *Delay Histogram* graphs, shown in Figure 5-25.

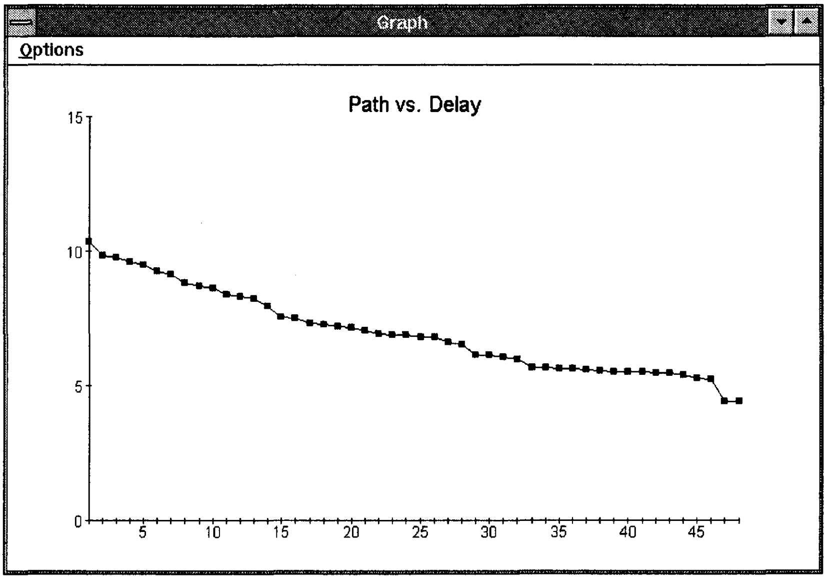


Figure 5-25 Path vs. Delay graph

The Path vs. Delay graph shows the path delays on the Y-axis and the path numbers on the X-axis. Double-clicking on points in this graph has the same highlighting effect as double-clicking on paths in the Path Analyzer.

The *Delay Histogram* graph uses a range of path delays as "buckets" on the X- axis. The number of paths falling into a delay range "bucket" is shown as a Y- value for each range.

Both graphs feature an options menu that provides the capability to copy the graph to the clipboard (as a bitmap) or to print the graph to the current printer. Each graph can also be customized with the *Graph/Options* menu command from the main *Path Analyzer* window.

Key Calculations

Using the Path Analyzer, key information can be determined with simple arithmetic.

Hint – The results of these calculations will always be conservative, for reasons provided below. The calculations do, however, provide a quick and convenient means of determining worst-case design performance.

Clock Skew

Click the Path Analyzer's *Option* button to bring up the *Path Analyzer Options* dialog box.

- Set the *Path Delay* radio button to *Find Max.*
- In the *Start Types* check box group, activate only the *Clock Pad* check box. The clock pad must be used as a clock not an input.
- In the *End Types* check box group, activate only the *Flip-Flops* check box.
- Click the *Run* button to execute the Path Analyzer.
- Make a note of the first path listed and the last path listed. (Note that the # Paths setting must be high enough to list all specified paths; in this case, it's the number of flip-flops used in the design.) The Clock Skew is given by:

```
SKEW = first_path - last_path
```

The clock skew calculation is always conservative, as the calculation ignores the fact that clock skew is meaningful only between flip-flops on a common path.

Operating Frequency

Click the Path Analyzer's *Option* button to bring up the *Path Analyzer Options* dialog box.

- Set the *Path Delay* radio button to *Find Max.*
- In the *Start Types* check box group, activate only the *Flip-Flops* check box.
- In the *End Types* check box group, activate only the *Flip-Flops* check box.
- Click the *Run* button to execute the Path Analyzer.
- Note the delay of the critical path listed. The operating frequency is given by:

$$F_{max} = 1/(\textit{critical\_path} + \text{clock SKEW})$$

The designer must determine the critical path in his design. This will come from a knowledge of the circuit function and its implementation. The designer may have to use some analysis to determine which paths are the frequency determining paths. Many designs contain false paths; therefore, the maximum delay path listed in the path analyzer may not be (and usually is not) the frequency determining path. False paths may include the following:

- Data paths with multiplexing and with various data sources and destinations where the longest path through the logic is never used.
- Long paths purposely given to signals which arrive long before they are required. Such paths are encountered in counters where high order stages reach their state long before the low order stage finally triggers a toggle in the counter.

Clock skew must be chosen for the critical path or paths identified. The largest skew identified is not necessarily the one to be used in the equation above. The number used must be the skew relevant to the critical delay path.

Setup Time

Click the Path Analyzer's *Option* button to bring up the *Path Analyzer Options* dialog box.

- Set the *Path Delay* radio button to *Find Max.*
- In the *Start Types* check box group, activate only the *Pads* check box.
- In the *End Types* check box group, activate only the *Flip-Flops* check box.
- Click the *Run* button to execute the Path Analyzer.

- Make a note of the first path listed; call it *pads_to_ffs*.
- Click the Path Analyzer's *Option* button to bring up the *Path Analyzer Options* dialog box.
- Set the *Path Delay* radio button to *Find Min.*
- In the *Start Types* check box group, activate only the *Clock Pad* check box.
- In the *End Types* check box group, activate only the *Flip-Flops* check box.
- Click the *Run* button to execute the Path Analyzer.
- Note the delay of the first path listed; call it *clock_to_ffs*. The setup time is given by:

$$t_{setup} = \texttt{pads\_to\_ffs} - \texttt{clock\_to\_ffs}$$

The setup time calculation is always conservative, because the two calculations often apply to different flip-flops.

Hold Time

Click the Path Analyzer's *Option* button to bring up the *Path Analyzer Options* dialog box.

- Set the *Path Delay* radio button to *Find Min.*
- In the *Start Types* check box group, activate only the *Pads* check box.
- In the *End Types* check box group, activate only the *Flip-Flops* check box.
- Click the *Run* button to execute the Path Analyzer.
- Make a note of the first path listed; call it *pads_to_ffs*.
- Click the Path Analyzer's *Option* button to bring up the Path Analyzer Options dialog box.
- Set the *Path Delay* radio button to *Find Max.*
- In the *Start Types* check box group, activate only the *Clock Pad* check box.
- In the *End Types* check box group, activate only the *Flip-Flops* check box.
- Click the *Run* button to execute the Path Analyzer.
- Note the delay of the first path listed;,call it *clock_to_ffs*. The hold time is given by:

$$t_{hold} = \textit{clock\_to\_ffs} - \textit{pads\_to\_ffs}$$

This calculation will typically yield a negative number. The Hold Time calculation is always pessimistic, as the calculation ignores the fact that the two measurements are likely along different paths.

5.6 Design Considerations: Speeding Up High-Fanout Nets

This section describes several techniques for speeding up the performance of designs created by the *Warp* system's SpDE tools. For more information, refer to Chapter 9, "Synthesis."

For high-fanout, timing-critical nets, designers should consider improving design performance using buffering techniques. In some cases, solutions such as paralleling or pipelining can be used.

Five techniques that can be used to improve circuit performance are described on the following pages:

- double buffering
- split buffering
- selective buffering
- paralleling
- pipelining

5.6.1 Double Buffering

The pASIC architecture allows two sources to drive a net in specific cases. This is called double buffering. Using two gates to drive a high-fanout net speeds up the performance of the net dramatically.

Figure 5-26 is an example of double buffering in a schematic.

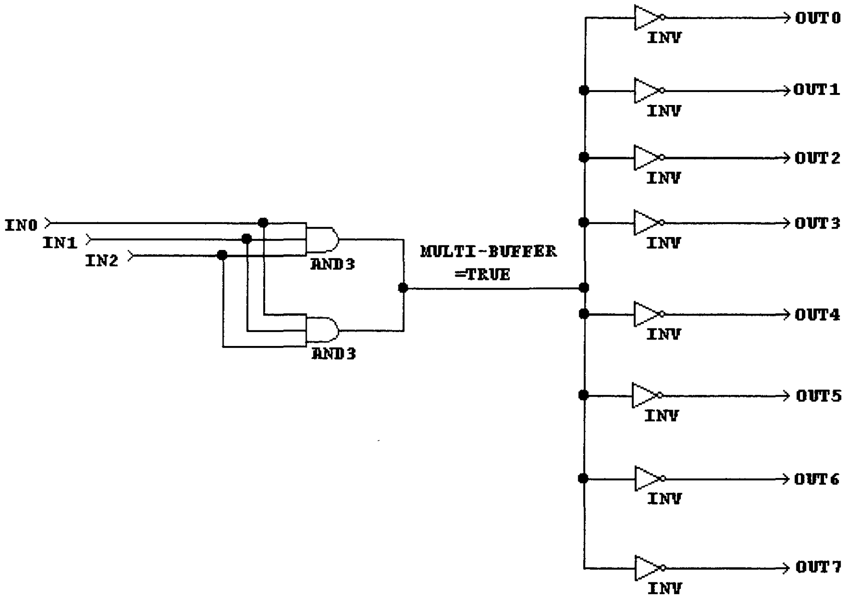


Figure 5-26 Double buffering example

Double buffering is legal as long as the two gates driving the high-fanout net are identical gates, with the same nets on the inputs and output. Each gate must fit into an AND-fragment (PAfrag_a library element). Double buffering is an excellent performance solution, and offers the best skew and delay characteristics of all buffering solutions for fanouts of 8 to 16. An example of double buffering in a VHDL source file is the following:

```
-- Resolution function for wired-or.  Used to create
-- legal VHDL for double-buffering techniques
-- employed for pasic.
-----------------------------------------------------
use work.resolutionpkg.all;
use work.GATESPKG.all;
use work.cypress.all;
use work.rtlpkg.all;
```

```
entity DOUBLEBUF is
    port(IN0: IN bit;
         IN1: IN bit;
         IN2: IN bit;
   OUT7: INOUT bit;
   OUT6: INOUT bit;
   OUT5: INOUT bit;
   OUT4: INOUT bit;
   OUT3: INOUT bit;
   OUT2: INOUT bit;
   OUT1: INOUT bit;
   OUT0: INOUT bit);
end DOUBLEBUF;

architecture archDOUBLEBUF of DOUBLEBUF is
  -- net to be resolved
  signal multiple_driver: multi_buffer bit;
begin
  multiple_driver <= IN0 AND IN1 AND IN2; -- driver #1
  multiple_driver <= IN0 AND IN1 AND IN2; -- driver #2
  OUT0 <= NOT multiple_driver;
  OUT1 <= NOT multiple_driver;
  OUT2 <= NOT multiple_driver;
  OUT3 <= NOT multiple_driver;
  OUT4 <= NOT multiple_driver;
  OUT5 <= NOT multiple_driver;
  OUT6 <= NOT multiple_driver;
  OUT7 <= NOT multiple_driver;
end archDOUBLEBUF;
```

Note – Double buffering on an 8x12 (1000 usable gates) or 12x16 (2000 usable gates) device requires the use of express wires. These devices have limited express wire resources, so only a few double buffers should be used. Refer to the Section 5.4.4, "Router," for more information.

5.6.2 Split Buffering

Split buffering breaks a wide-fanout net into two or more nets.

Figure 5-27 is an example of split buffering. Without the buffers, the DFF drives a fanout of 8. As configured in the illustration, the DFF drives a fanout of 2, and each buffer drives a fanout of 4.

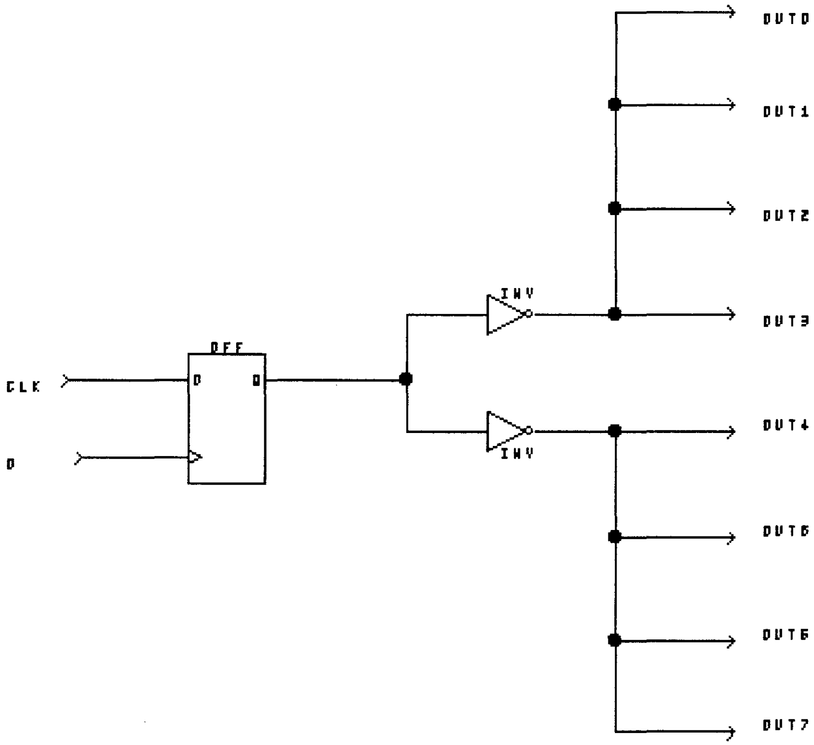


Figure 5-27 Circuit demonstrating split buffering

Note – Adding buffers introduces a logic cell delay to the net. This added delay must be balanced against the gain in reducing the fanout. Simple split buffering (as demonstrated in Figure 5-27) is generally employed only with fanouts of 16 or greater.

5.6.3 Selective Buffering

Selective buffering is the selective use of buffers in situations where a high-fanout net has a small number of critical destinations and a large number of less-critical ones.

Figure 5-28 is an example of selective buffering. The DFF drives a fanout of 8, but only one of the destinations is in the critical path of the circuit. Inserting a single buffer between the DFF output and the 7 non-critical destinations restructures the circuit, so that the DFF drives a fanout of two without adding any logic cell delay in the critical path.

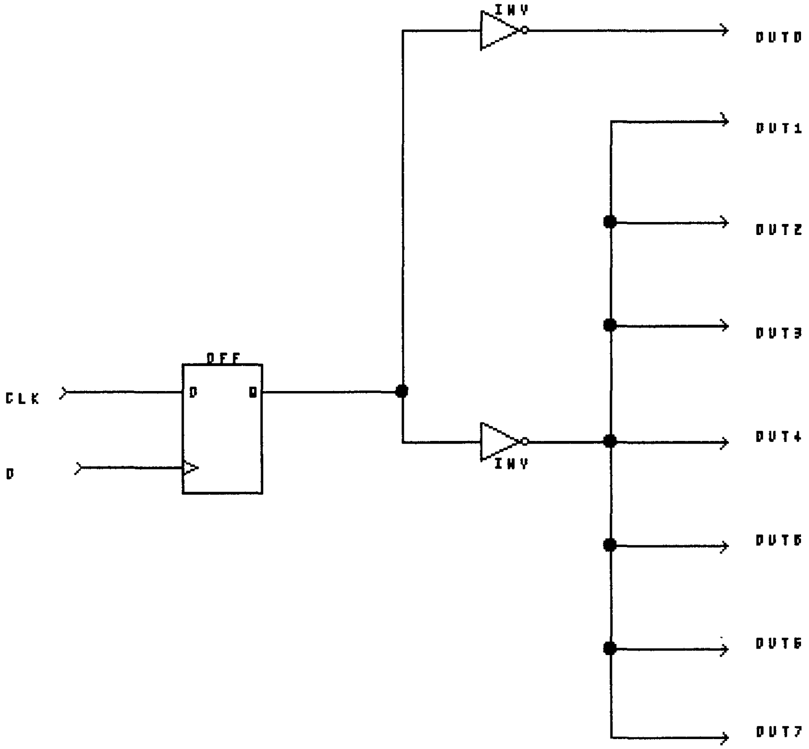


Figure 5-28 Circuit demonstrating selective buffering

Hint – Buffers should be introduced with care and skill. **Selective buffering** offers tremendous improvement in circumstances where the circuit has a few clearly identifiable critical paths.

5.6.4 Paralleling

Paralleling is a design technique that duplicates the logic driving a high-fanout load to reduce the effective fanout. Duplicating the logic avoids the delay introduced by adding buffers to the circuit.

Successful buffering must balance reduced fanout against the additional delay that use of buffers causes. Paralleling is an alternative that does not introduce this added delay.

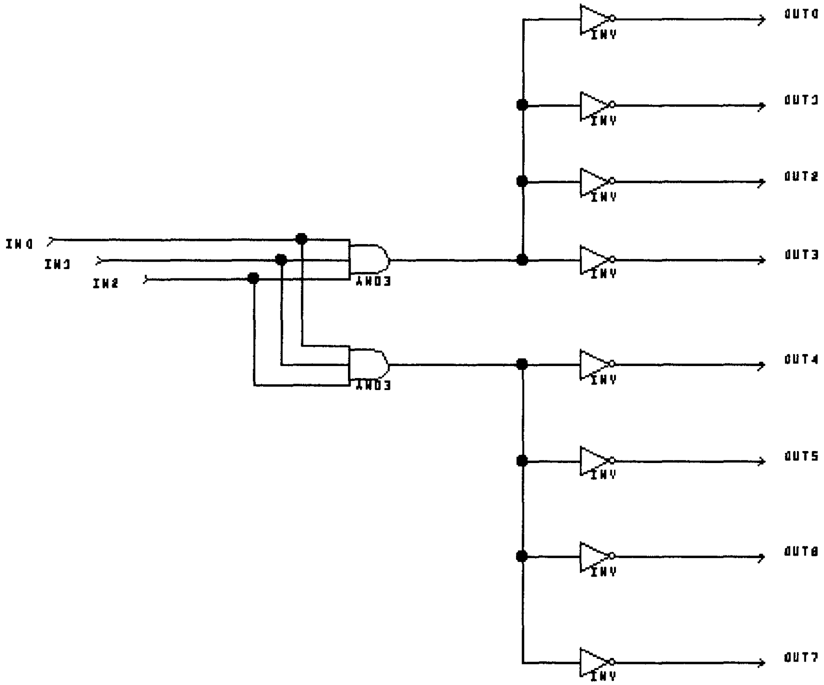


Figure 5-29 Circuit demonstrating paralleling

Figure 5-29 is an example of paralleling. The AND gate has been duplicated, with each of its inputs tied to the corresponding input on the "twin" gate. Each AND gate drives a fanout of 8, effectively halving the fanout, without introducing the added delay associated with buffering. By duplicating the AND gate, however, the fanout on each of the input nets has been increased.

Paralleling is similar to double buffering, except that the outputs are not tied together. Paralleling should be used instead of double buffering when:

- skew is not critical
- too many express wires have already been used for high-drive inputs or double buffers (see the section on the Router)
- the logic to be replicated does not fit into an AND fragment of the larger cell (no larger than a PAfrag_a library element)

5.6.5 Pipelining

Pipelining is the technique of inserting registers in long combinatorial paths, effectively increasing the system clock rate.

Inserting registers in long combinatorial paths shortens the length of the critical path and allows operations to be overlapped, increasing the system clock rate. The pASIC architecture promotes pipelining, as each logic cell contains a D flip-flop. As a result, a design can be pipelined with little or no increase in the number of logic cells used.

For more information on achieving high performance or high utilization in designs, see Chapter 9, "Synthesis."

Chapter 6

Nova

6.1 Introduction

Nova is Cypress Semiconductor Corporation's name for its JEDEC-based functional simulator.

The Nova user interface provides an easy way to:

- specify JEDEC files to simulate
- read or write stimulus files
- convert files from *.jed* to ViewSim format
- edit input waveform traces
- simulate the behavior of a design
- alternate between various views (i.e., collections of signals) and specify signals to be included in each view
- specify the length and resolution of a simulation
- specify segments, where initial conditions can be reapplied and edited in order to compare results of differing initial conditions side-by-side
- other useful capabilities

This manual describes how to use Nova to simulate designs. It assumes familiarity with common user interface operations for the computer, such as the use of scroll bars, menu buttons, and opening and closing windows.

Advanced users may refer to Section 6.9, "Nova JEDEC Simulator Quick Reference Sheet," for a brief overview of the major functionality.

6.2 Starting Nova

On Sun workstations, typing **nova** on the command line brings up the Nova window. On PCs and compatibles, double-clicking on the *Nova* icon in the Cypress group window brings up the Nova window.

By default, Nova comes up ready to run on a color screen.

To start Nova on a monochrome Sun workstation, type **nova -m** on the command line.

To set Nova to come up in monochrome mode when running Windows on an IBM PC or compatible computer, do the following:

- Select the *Nova* icon from the *Warp R4* group window.
- Select *Properties* from the File menu.
- Edit the *Command Line* entry to include the `-m` option.
- Click *OK.*

6.3 The Nova Window

The Nova window (Figure 6-1) consists of a menu bar with several items across the top; a column of buttons along the left side, listing pin and node numbers and signal names; an area for displaying traces; and scroll bars across the bottom and right sides.

Menu Bar

The menu items are *File, Edit, Simulate, Views,* and *Options*. Under each of these items are menus for selecting related actions. The menus are ordered so that the most common operation is at the top. The contents of each menu are described in greater detail later in this chapter.

Only two menu items, *Open* and *Exit,* are enabled in the File menu when Nova first opens. When the user opens a *.jed* file, the other menu items will be enabled.

Node Numbers, Signal Names

The left-hand side of the Nova window consists of a column of buttons, displaying pin and node numbers and their associated signal names. A node is an area of a circuit containing one or more points whose locations the user may wish to trace. (For information about different values within a node, refer to Section 6.5.4, "Nodes.")

To change the width of the buttons where signal names are displayed, use the *Signal Name Size* item in the *Options* menu.

Trace Area

The trace area displays the values of the nodes/signals listed in the left-hand column.

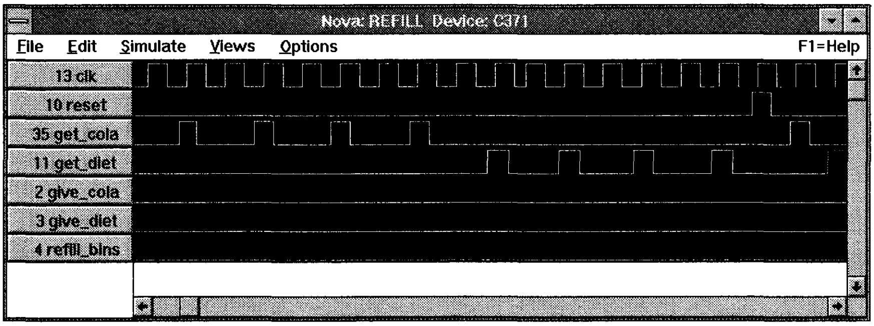


Figure 6-1 Main Nova Window

The window displays up to two measuring cursors, which allows the user to see precisely the value(s) of several signals at a single time. To display the first cursor, click at the bottom of the trace window. To display a second cursor, click at the bottom of the trace window while pressing the Shift key.

To change the position of either cursor, click and hold on the cursor at the bottom of the trace window, then drag the cursor to its new position. The cursor's horizontal position in simulation tics is displayed next to each cursor.

Note that a simulation tic does not represent any set amount of real-time delay. Instead, a simulation tic is simply a unit of simulation time.

6.4 The File Menu

The *File* Menu contains items related to opening JEDEC files for simulation, reading and writing stimulus files, and saving output files in various formats.

The *File* menu (Figure 6-2) in the Nova dialog box contains the following items:

- *Open...*
- *Write Sim (*.sim)*
- *Write Trace (*.psd)*
- *Read Stimulus File*

- *Write JEDEC Vectors*
- *Write JEDEC File (*.jed)*
- *Disassemble to ViewSim Format (*.vhd)*
- *Exit*
- *About...*

The operations of each of these menu items are discussed in greater detail on the next few pages.

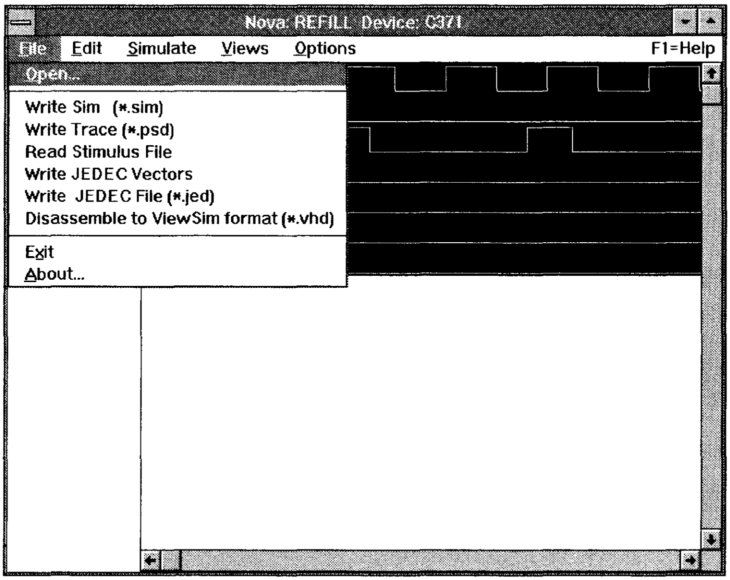


Figure 6-2 Nova File Menu

6.4.1 Opening Files

The *Open...* item in the File menu selects which *.jed* file to open and tells Nova what device is targeted in simulation.

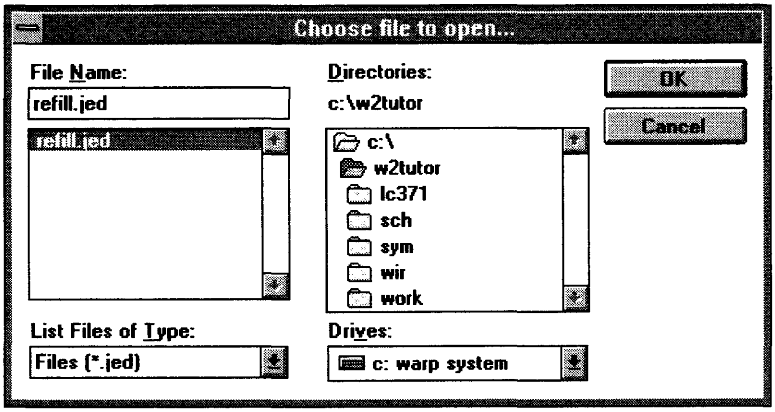
Selecting *Open...* brings up the *Open Files* dialog box (Figure 6-3). The *File Name* line specifies the names of files to view or to open in the Files window. By default, this line reads "**.jed.*"

To open a file, the user can select a file from the list shown in the *Files* window, or type the name of the file on the *File name* line. Selecting a *.jed* file and clicking on *Open* closes the dialog box and displays traces. (If a stimulus file of the form *filename.sim* or *filename.stm* exists, it is also read automatically.) Clicking on *Cancel* closes the dialog box without opening a file.

The *Select Device* dialog box (Figure 6-4) comes up when the user clicks on *Open* in the *Open Files* dialog box, and the file to be opened is a *.jed* file not created by *Warp*. The *Select Device* dialog box maps a JEDEC file to a device.

Selecting a device with the wrong number of fuses brings up a message box stating: "Wrong device type for this JEDEC - QF doesn't match." This indicates that the number of fuses in the selected device does not match the number in the JEDEC file.

Note – If Nova says that it cannot find file *devices.dat*, check to make sure that the CYPRESS_DIR environment variable is set correctly. Nova uses this file to find the proper pin names and numbers for each target device and package.



File Name: refill.jed

Directories: c:\w2tutor

List Files of Type: Files (*.jed)

Drives: c: warp system

Figure 6-3 Open Files Dialog Box

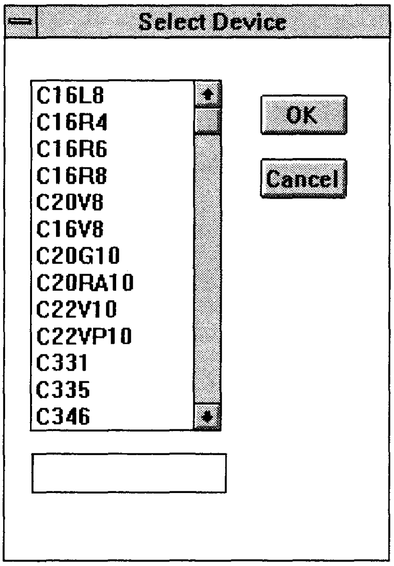


Figure 6-4 Select Device Dialog Box

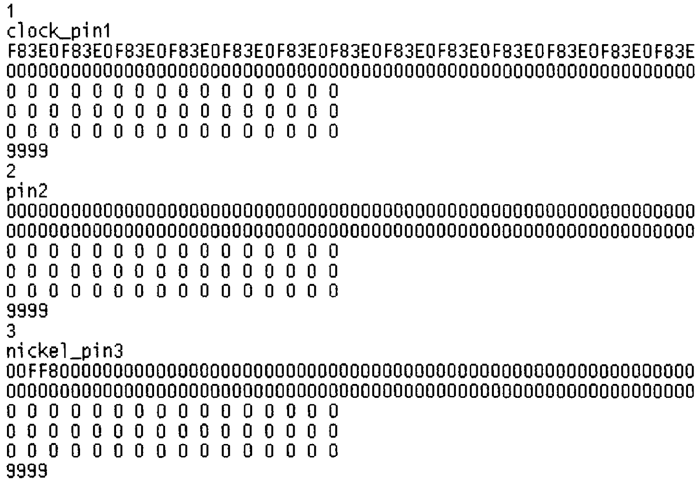
6.4.2 Reading and Writing Stimulus Files

Write Sim and *Write Trace* save simulation data. *Read Stimulus File* reads data stored by a previous Write Sim operation.

Write Sim saves the current simulation data to *filename.sim*, where *filename* is the prefix of the file the user is simulating. If a *.sim* file already exists with this filename, the new simulation data overwrites the old. The *.sim* file (see Figure 6-5) can be re-read with the *Read Stimulus File* option.

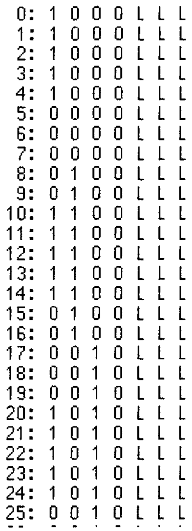
Write Trace saves the trace information to *filename.psd*, where *filename* is the prefix of the file being simulated. The *.psd* file (see Figure 6-6) provides a column-oriented, human-readable record of trace values during the simulation. Bus values are not written to the file.

Read Stimulus File reads simulation data from a *.sim* file. Because reading in the simulation file may change some of the settings the user has set for the current simulation, a message box is displayed, asking if the stimulus file should be read in. A *Yes* reply reads in the *.sim* file. A *No* reply returns the user to the main Nova window. The *filename.sim* file is automatically read when the *filename.jed* file is opened.



```
1
clock_pin1
F83E0F83E0F83E0F83E0F83E0F83E0F83E0F83E0F83E0F83E0F83E0F83E0F83E
0000000000000000000000000000000000000000000000000000000000000000
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
9999
2
pin2
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
9999
3
nickel_pin3
00FF800000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
9999
```

Figure 6-5 Portion of .sim File



```
 0: 1 0 0 0 L L L
 1: 1 0 0 0 L L L
 2: 1 0 0 0 L L L
 3: 1 0 0 0 L L L
 4: 1 0 0 0 L L L
 5: 0 0 0 0 L L L
 6: 0 0 0 0 L L L
 7: 0 0 0 0 L L L
 8: 0 1 0 0 L L L
 9: 0 1 0 0 L L L
10: 1 1 0 0 L L L
11: 1 1 0 0 L L L
12: 1 1 0 0 L L L
13: 1 1 0 0 L L L
14: 1 1 0 0 L L L
15: 0 1 0 0 L L L
16: 0 1 0 0 L L L
17: 0 0 1 0 L L L
18: 0 0 1 0 L L L
19: 0 0 1 0 L L L
20: 1 0 1 0 L L L
21: 1 0 1 0 L L L
22: 1 0 1 0 L L L
23: 1 0 1 0 L L L
24: 1 0 1 0 L L L
25: 0 0 1 0 L L L
```

Figure 6-6 Portion of *.psd* File

6.4.3 Writing JEDEC Vectors

Write JEDEC Vectors appends vector information to the JEDEC file. The vectors can be used to test parts after they are programmed. If the JEDEC file already contains vector information, the new vector information overwrites the old.

6.4.4 Converting Between File Formats

The *File* menu includes items that allow the user to convert vector information into different file formats, depending on what he wants to do with it.

Figure 6-7 shows the various file types that can be input to or output from *Warp,* Nova, or a device programmer.

Write JEDEC File (**.jed*) writes out a JEDEC file from the data available to the simulator. The dialog box options include instructions in the JEDEC file to blow the security fuse when the device is programmed, or to write the JEDEC file using a compressed "K-field" hexadecimal representation.

Disassemble to ViewSim format (**.vhd*) writes out a Viewlogic VHDL file which is used to simulate the design in ViewSim. The file is created in the *vhd.* subdirectory

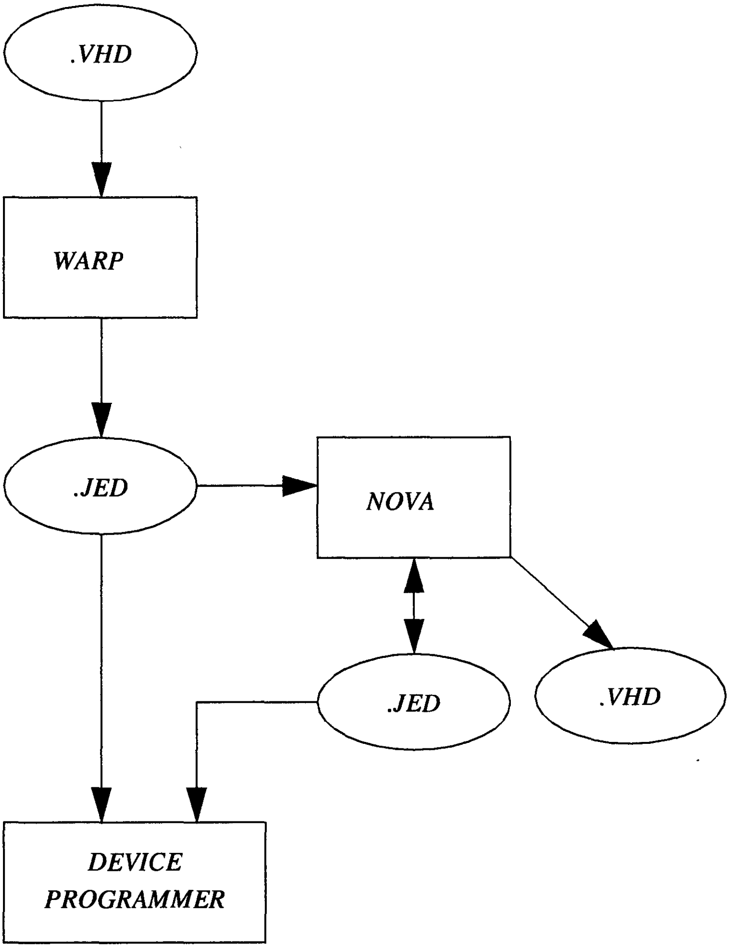


Figure 6-7 Possible Data Paths and File Formats

6.4.5 About and Exit

The *File* menu's *About* item displays some basic information about the Nova simulator. The *Exit* item exits the simulator.

Besides displaying version information about the Nova simulator, the *About* dialog box also includes a *Help* button. Clicking on *Help* brings up help about Nova.

6.5 The Edit Menu

Use the items in the *Edit* Menu to modify trace information displayed on the screen. With the *Edit* menu, the user can set the selected range of a trace; create and delete view nodes; create, delete and edit buses; and change the bus radix.

Items in the *Edit* Menu (Figure 6-8) include the following:

- *High, Low* sets the selected trace or portion of a trace to 1 or 0, respectively.
- *Clock* sets up repetitive pulses.
- *Pulse* sets up a single pulse.
- *Node Defaults* specifies the default source for the displayed value of a node.
- *Create View Node* creates a new trace and selects the point within a node at which the displayed value is measured.
- *Delete View Node* deletes traces from the simulation.
- *Create Bus* groups traces for display as a single entity called a bus, used when thinking of groups of signals as a single value is more convenient. Bus values are only displayed when a measuring cursor is present.
- *Delete Bus* un-defines a previously defined bus.
- *Edit Bus* adds or removes signals from a bus.
- *Bus Radix* specifies radix used to display a bus value.

These items are described in greater detail on the following pages.

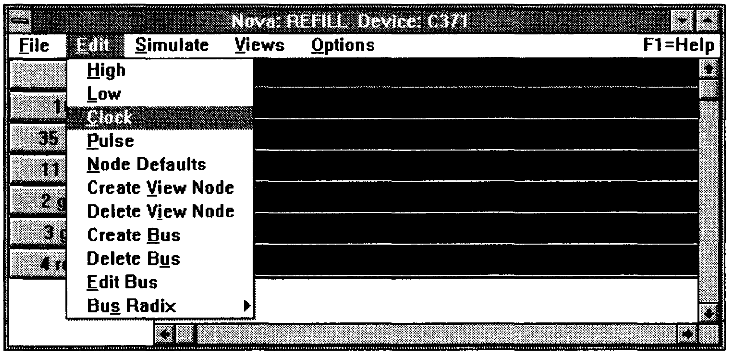


Figure 6-8 Nova Edit Menu

6.5.1 Setting Signals High or Low

With the Nova user interface, the user can easily set the value of all or a selected portion of an input signal to high or low.

To set an entire input signal to high or low:

- Click on the button containing the name of the signal in the Nova window to select it. On color monitors, the button changes color, and the trace turns blue when selected. On monochrome monitors, the button goes to inverse video, and the trace changes to a dotted line when selected.
- Select *High* or *Low* from the *Edit* menu as desired, or type "1" or "0."

To set a portion of an input signal high or low (see Figure 6-9):

- De-select the signal.
- Click and hold the mouse button on the trace at the left edge of the selected area.
- Drag the mouse to the right edge of the selected area.
- Then select *High* or *Low* from the *Edit* menu, as appropriate, or type "1" or "0."

Both the left and middle buttons of a 3-button mouse perform the same action when clicked to position an edge.

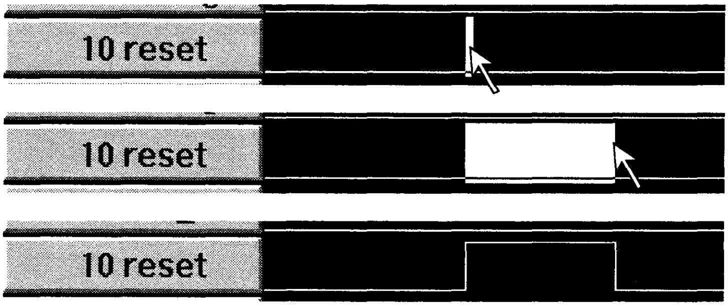


Figure 6-9 Setting a Portion of a Signal High or Low

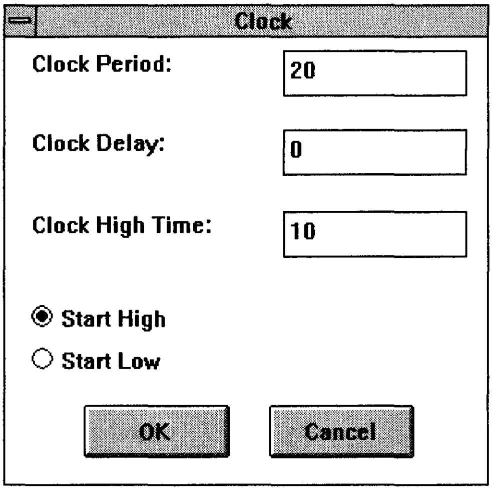
Top: press and hold the mouse button at the left edge of the selected area. Middle: drag to the right edge of the selected area, and release the mouse button. Bottom: select *High* or *Low* from the Edit menu, or type "1" or "0" from the keyboard.

6.5.2 Setting Up Clock Signals (Repetitive Pulses)

The *Clock* item under the *Edit* menu allows the user to set up repetitive pulses on a selected signal or portion thereof.

To set up a repetitive pulse or clock signal, select a signal or a portion of a signal, then select the *Clock* item under the *File* menu. This brings up the *Clock* dialog box (Figure 6-10), which allows the user to fill in various repetitive pulse parameters.

- *Clock Period* specifies the period of repetition for the pulse in simulator tics.
- *Clock Delay* specifies the number of simulator tics to wait (beginning with the left edge of the selected area) before starting the repetition. The default is 0 tics.
- *Clock High Time* specifies the amount of time that the selected signal should be set to 1 during each repetition. The default is 5.
- *Start High* and *Start Low* specify whether each repetition starts with the signal set to 0 or 1.
- *OK* sets up the repetitive pulse.
- *Cancel* closes the *Clock* dialog box without affecting the trace.



Clock Period: 20

Clock Delay: 0

Clock High Time: 10

● Start High

○ Start Low

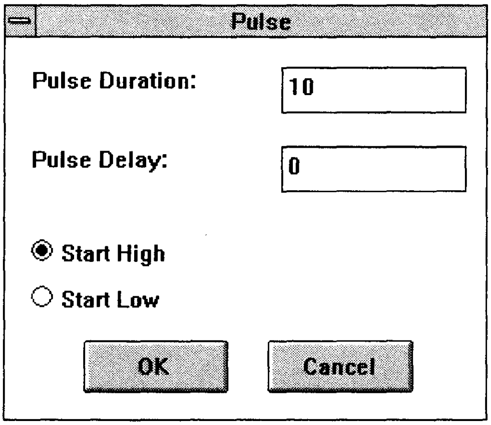
Figure 6-10 Clock Dialog Box

6.5.3 Setting Up Non-Repetitive Pulses

The *Pulse* item under the *Edit* menu allows the user to set up single pulses on a selected signal or portion thereof.

To set up a single pulse on a signal, select a signal or a portion of a signal, then select the *Pulse* item under the *File* menu. This brings up the *Pulse* dialog box (Figure 6-11), which allows the user to fill in various pulse parameters.

- *Pulse Duration* specifies the length of the pulse, in simulator tics.
- *Pulse Delay* specifies the number of simulator tics to wait (starting from the start of the simulation) before applying the pulse. The default is 0.
- *Start High* and *Start Low* specify whether the pulse sets the signal to 0 or 1.
- *OK* sets up the pulse.
- *Cancel* closes the *Pulse* dialog box without affecting the trace.



Pulse

Pulse Duration: 10

Pulse Delay: 0

● Start High

○ Start Low

Figure 6-11 Pulse Dialog Box

6.5.4 Nodes

A node is an area of a circuit containing one or more points at which the user may wish to trace a signal. Nova allows the user to specify the exact point or points within a node at which to trace signal values, to set the default value of a node, and to force one or more positions in a node to known values.

To Nova, a node is:

- any input to an array
- any output from an array
- any pin on the device
- any other electrical position that needs to be modeled but does not meet the first three criteria

For each node, the user can:

- create a view node, i.e., specify one or more positions within a node from which to trace values
- specify the means by which a node is assigned its value
- force any position in a node to a known value (this is often useful for multi-segment simulations)

Each of these capabilities is discussed in greater detail on the following pages.

6.5.5 Selecting Node Points to View

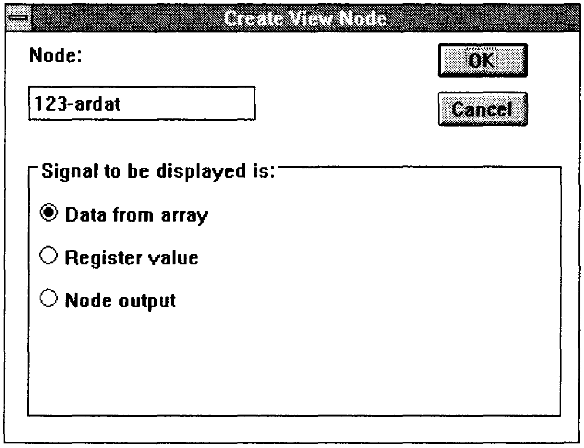
Many nodes contain several points at which the user can trace simulation values. *Create View Node* allows the user to select which of those points to view.

A view node allows the user to see what is happening at various points inside a node. Selecting *Create View Node* brings up the *Create View Node* dialog box (Figure 6-12), which allows the user to select points to view within a selected node. To bring up this dialog box, the user must select a node with the current view set to *FULL*. (See Section 6.7, "The Views Menu," for information about changing views.)

The *Create View Node* dialog box displays the node name with the view node name to be created directly below it. Nova creates the view node name by taking the node number, followed by a '-' and an extension to represent the selected signal to be displayed.

The view node points that can be displayed depend upon the selected node. Examples of view node points that can be displayed include:

- *Data from Array* - This is the data at the output of an OR-XOR combination of gates. Extension is "ardat."
- *Out value before OE* - This is the data on the output pin if the output enable is asserted. This includes the output buffer inversion, if there is one. Extension is "b_oe."
- *OE Value* - This is the state of the output enable. If high, OE is asserted so the output is driven. Extension is "oe."
- *Node Output* - This is the data on the pin. This is the default view for output nodes. Extension is "out."



Node:

123-ardat

Signal to be displayed is:

● Data from array

○ Register value

○ Node output

Figure 6-12 Create View Node Dialog Box

- *Feedback at input* - This is the data at the D input of the input register, if there is one. If there is no input register, feedback at input and feedback to array are identical. Extension is "fbkin."
- *Feedback to array* - This is the data that is being fed to the array. It differs from feedback at input because it may be the other side of a register. Extension is "fbk_ar."

Selecting *OK* closes the *Create View Node* dialog box and creates a view node, displayed at the end of the node list. Selecting *Cancel* closes the *Create View Node* dialog box without creating the view node.

To delete a view node, select the view node to delete, then select *Delete View Node* from the *Edit* menu.

6.5.6 Setting Input Node Values

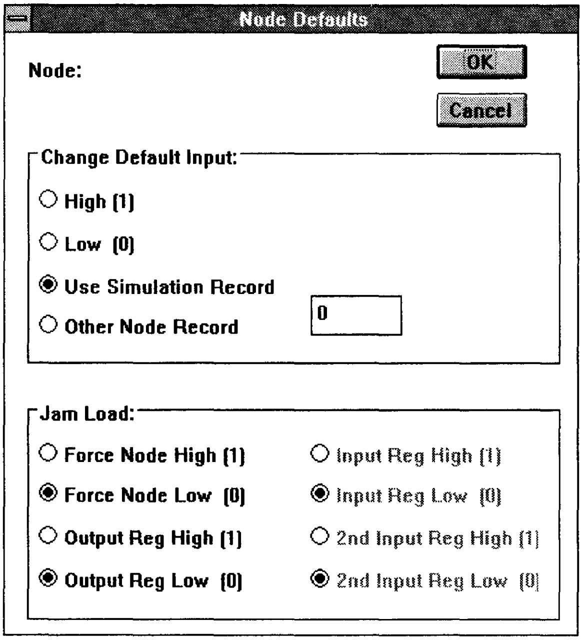
Node Defaults allows the user to specify the default source for the displayed value of a node.

Selecting *Node Defaults* brings up the *Node Defaults* dialog box (Figure 6-13).

Use the *Change Default Input* window of the *Node Defaults* dialog box to specify the source for the value of an input node. The current setting is shown highlighted within this window.

There are four possible settings for each input. They are:

- *High* (1): tie the signal to Vcc
- *Low* (0): tie the signal to Vss (ground)
- *Use Simulation Record*: use the value(s) in the simulation record (*.sim* file)
- *Other Node Record*: tie the signal to another node. Enter the node number on the line to the right of the *Other Node Record* button



Node:

Change Default Input:

○ High (1)

○ Low (0)

● Use Simulation Record

○ Other Node Record 0

Jam Load:

○ Force Node High (1) ○ Input Reg High (1)

● Force Node Low (0) ● Input Reg Low (0)

○ Output Reg High (1) ○ 2nd Input Reg High (1)

● Output Reg Low (0) ● 2nd Input Reg Low (0)

Figure 6-13 Node Defaults Dialog Box

6.5.7 Forcing Output Node Values

Node Defaults also allows the user to force the value of an output node at a specified point.

Selecting *Node Defaults* brings up the *Node Defaults* dialog box (Figure 6-13).

The Jam Load window of the *Node Defaults* dialog box can be used to force an output node to a specified value. Values of these nodes rarely need to be modified for normal simulations; however, for multi-segment simulation (for long counters and other long-period design) or if there are problems in simulating the start-up condition of a circuit, the values may need to be changed. The current setting is shown highlighted.

Depending on the type of node, it may be possible to select from *Force Node High*(1), *Force Node Low*(0), *Output Reg High*(1), *Output Reg Low* (0), *Input Reg High* (1), *Input Reg Low* (0), *2nd Input Reg High* (1), and *2nd Input Reg Low*(0).

6.5.8 Working with Buses

At times, grouping several traces in a simulation and viewing them as a single trace may be more convenient. This is possible with the *Create Bus, Delete Bus,* and *Edit Bus* options in the *Edit* menu.

Selecting *Create Bus* brings up the *Bus* dialog box. This dialog box combines nodes into a user-named bus. The View list in the dialog box contains the names of all nodes in the current view. The Bus list holds the names of each node in the bus. A bus may be made up of any number of nodes.

Selecting *OK* closes the Bus dialog box and creates a bus with the specified bus name. Buses are placed at the top of the trace area. Selecting *Cancel* closes the *Bus* dialog box without changing the trace area. It is not possible to input values to a bus.

To add a node to the bus: select the node from the *View* list and select the *Add>>* button. Double-clicking on the node name also adds the selected node to the bus. The new node is added below the selected nodes of the bus.

Note – Nodes can be added only to a bus in the current view.

Clicking on the *Add-by-Name* button brings up a dialog box that asks the user to specify the name(s) of signals to add to the bus. The use of wild card characters is permitted. A "?" matches a single character; a "*" matches any string of characters. The construct *name*[*m*:*n*] denotes a range of signals, numbered from *m* through *n*, beginning with the characters *name*. For example, "input[0:3]" matches signals `input0`, `input1`, `input2`, and `input3`.

To remove a node from the bus: Select the node to be removed and select the *Cut* button. Double-clicking on the node name in the Bus list also removes the node from the bus.

To change a node's position in the bus: Select the node, then click *Cut*. Select another node, then click *Paste*. The node that was previously cut will be inserted below the newly selected node.

To name the bus: Click on the line below the words *Bus Name* and enter the name for the bus. If no name is provided, the bus is named *generic bus*.

To delete a bus: Select a bus trace by clicking on the bus name button or the bus trace. After the bus is selected, selecting the *Delete Bus* item from the *Edit* menu brings up a dialog box which can remove the bus from the trace area.

Edit Bus brings up the same Bus dialog box used for creating the bus. The bus name line is filled in, and the nodes in the bus are displayed in the Bus list. Buses can be added, removed, or have their names changed from this dialog box.

After all changes have been completed, selecting *OK* closes the bus dialog box and applies the modifications to the selected bus. Selecting *Cancel* closes the dialog box without updating the bus.

Bus Radix brings up a submenu that allows the user to choose how bus information is displayed. The three choices are *binary, octal* and *hexadecimal. Hexadecimal* is the default.

6.6 The Simulate Menu

The *Simulate* Menu has only one menu item: *Execute.*

Selecting *Execute* from the *Simulate* Menu (Figure 6-14) simulates the design's operation. The Nova screen is redrawn, and the resulting waveforms are displayed.

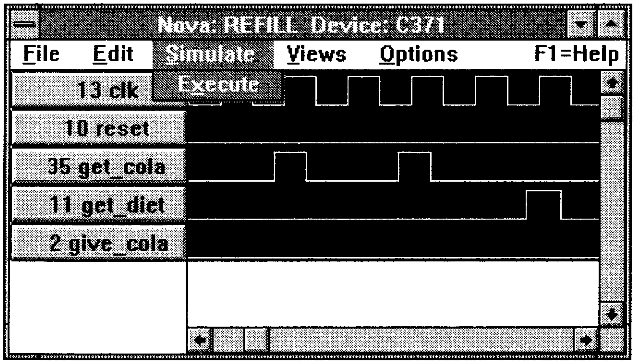


Figure 6-14 Simulate Menu

6.7 The Views Menu

Items in the *Views* menu allow the user to select the views (i.e., groupings of traces) in the trace area.

The *Views* menu (Figure 6-15) contains five items:

- *Edit Views* allows the user to create and edit views.
- *Select View* allows the user to select a view to display.
- *Delete View* allows the user to remove one or more views from the list.
- *Zoom In (2X)* multiplies the displayed timescale resolution factor by two.
- *Zoom Out (1/2X)* divides the displayed timescale resolution factor by two.

Each of these items is discussed in greater detail in the following pages.

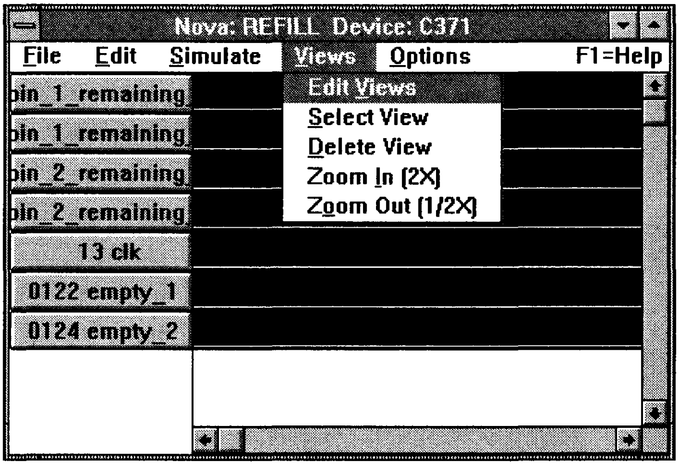


Figure 6-15 Views Menu

6.7.1 Editing Views

Edit Views allows the user to create new views and to add, remove, or exchange traces in existing views.

Three views are automatically created with each *.jed* file: full, pins-only, and pins & registers. The full view (default) lists all nodes in the design. This view cannot be edited. The pins-only view contains only nodes that are attached to pins. The pins & registers view contains all nodes attached to registers or pins.

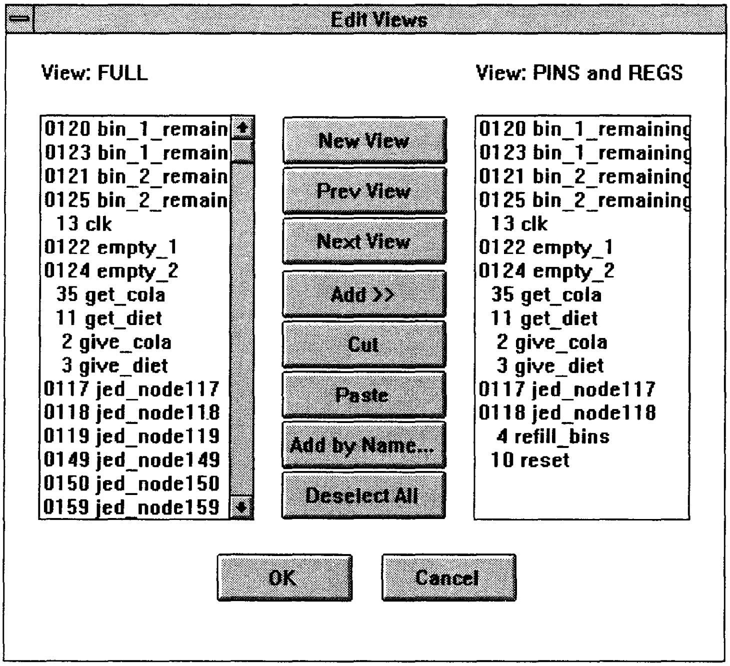
Selecting *Edit Views* displays the *Edit Views* dialog box (Figure 6-16), used to edit the current view. The view list on the left displays the FULL view, which contains the default traces for all nodes. Use this list, along with appropriate buttons, to add or remove traces from the view list on the right.

To create a new view: Click on *New View.* A name prompt will appear, which is placed at the top of the right-hand view list.

To move between views: Click on *Next View* or *Previous View.*

To add a trace to a view: Select one or more traces from the left (Full) view window, then click *Add>>*. If a trace is also selected in the right window, the new traces are inserted after the selection; otherwise, the new traces are added to the end of the view.

To remove traces from a view: Select the traces in the right window, then click on *Cut.*



Edit Views

View: FULL

```
0120 bin_1_remain
0123 bin_1_remain
0121 bin_2_remain
0125 bin_2_remain
  13 clk
0122 empty_1
0124 empty_2
  35 get_cola
  11 get_diet
   2 give_cola
   3 give_diet
0117 jed_node117
0118 jed_node118
0119 jed_node119
0149 jed_node149
0150 jed_node150
0159 jed_node159
```

View: PINS and REGS

```
0120 bin_1_remaining
0123 bin_1_remaining
0121 bin_2_remaining
0125 bin_2_remaining
  13 clk
0122 empty_1
0124 empty_2
  35 get_cola
  11 get_diet
   2 give_cola
   3 give_diet
0117 jed_node117
0118 jed_node118
   4 refill_bins
  10 reset
```

Figure 6-16 Edit Views Dialog Box

To exchange (i.e., re-order) traces within a view: Select one or more traces from the right view window, then click *Cut.* Then, select another trace from the right view window and click *Paste.* The previously cut trace(s) are inserted after the selected trace.

Add-by-Name brings up a dialog box that asks the user to specify the name(s) of traces to add. The use of wild card characters is permitted. A "?" matches a single character; a "*" matches any string of characters. The construct *name*[*m*:*n*] denotes a range of signals, numbered from *m* through *n*, beginning with the characters *name*. For example, "input[0:3]" matches signals `input0`, `input1`, `input2`, and `input3`. The user can also use multiple expressions separated by spaces.

Deselect All unselects all selected traces in either window.

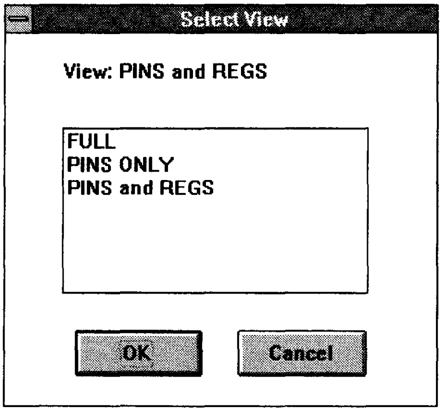
Selecting *OK* closes the *Edit Views* dialog box and updates the trace area to reflect changes made to the view. Selecting *Cancel* closes the *Edit Views* dialog box without making any changes to the view.

6.7.2 Selecting and Deleting Views

Select View allows the user to change the active view. *Delete View* allows the user to remove a view from the list of available views.

Select View brings up the *Select View* dialog box (Figure 6-17). The *View* line gives the name of the current view. To change the current view, select the desired view from the list, then click *OK* or type a carriage return. Clicking *Cancel* closes the *Select View* dialog box without affecting the active view.

Delete View also brings up the *Select View* dialog box. Select the view to delete from the scrollable list. The *FULL* view may not be removed and is not included in this list. Clicking *OK* or typing a carriage return applies the change to the list of views. If the current active view is removed, the active view changes to *FULL*. *Delete View* has no undo, so the user should be certain the view being deleted is correct before clicking on *OK* or typing a carriage return. *Cancel* closes the *Select View* dialog box without deleting the selected view.



View: PINS and REGS

Figure 6-17 Select View Dialog Box

6.7.3 Zoom In, Zoom Out

Zoom In doubles the time scale resolution of the trace window, i.e., by doubling the number of pixels in the X-axis used to display one tic of simulation time. The result is to "zoom in" on the view of displayed traces.

Zoom Out does the reverse of *Zoom In.*

The resolution setting must be 1 or greater. The default is 5. Attempting to set the time scale resolution lower than 1 has no effect.

6.8 The Options Menu

The *Options* Menu contains items that allow the user to specify the simulation length, create or delete simulation segments, and specify the viewing resolution of the trace area.

The *Options* Menu (Figure 6-18) contains five items:

- *Simulation Length* allows the user to set the length of the simulation.
- *Create Segment* allows the user to create a segment, or "new-start-point," within the simulation.
- *Delete Segment* allows the user to delete a previously created segment from the simulation.
- *Resolution* allows the user to stretch and compress displayed traces.
- *Signal Name Size* allows the user to specify the width in characters of Nova's signal name buttons.

Each of these items is described in greater detail in the following pages.

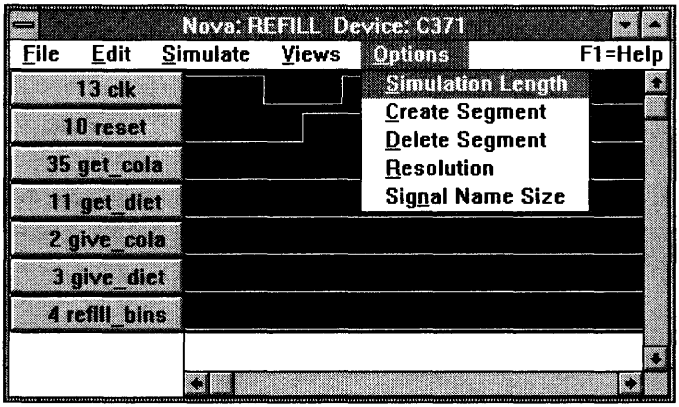


Figure 6-18 Options Menu

6.8.1 Simulation Length

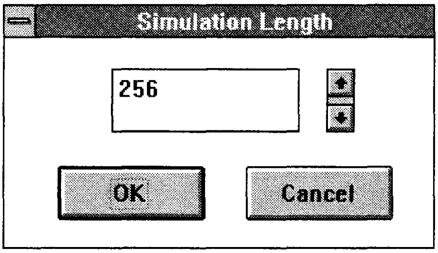
Simulation Length allows the user to set the length of the simulation.

Selecting *Simulation Length* brings up the *Simulation Length* dialog box (Figure 6-19).

The minimum and default simulation length is 256 tics. The maximum simulation length is 9984. Clicking on the up arrow adds 64 tics to the simulation length, to a maximum of 9984. Clicking on the down arrow subtracts 64 tics from the simulation length, to a minimum of 256 tics.

The user can also set the simulation length by typing a number on the line next to the up and down arrows. The number will be rounded downward to the nearest multiple of 64.

Clicking *OK* closes the *Simulation Length* dialog box and sets the simulation length to be used on the next simulator run. Clicking *Cancel* closes the *Simulation Length* dialog box without affecting the simulation length.



256

Figure 6-19 Simulation Length Dialog Box

Note – Any repetitive input signals such as clocks should be respecified whenever the simulation length is increased.

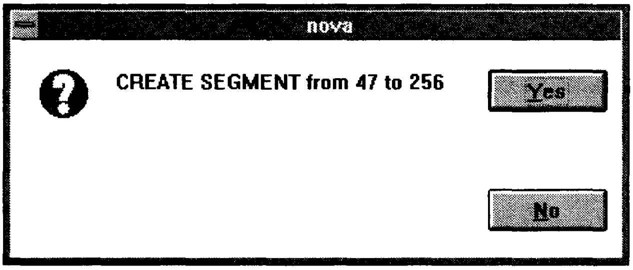
6.8.2 Creating and Deleting Segments

Create Segment allows the user to create a segment, or "new start point," within the simulation. *Delete Segment* deletes a previously created start boundary.

A segment is a point in the simulation at which various nodes are reset to their "jam load" values (set through the *Node Defaults* dialog box).

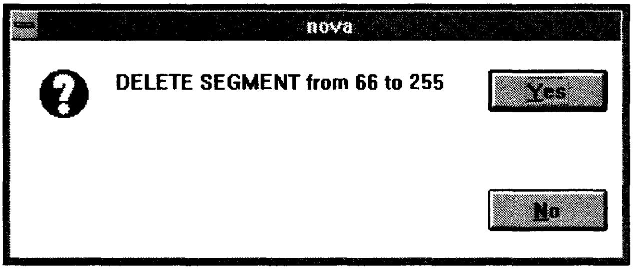
To create a simulation segment, position the leftmost measuring cursor at desired beginning of the segment, then select *Create Segment* from the *Options* menu to bring up the *Create Segment* dialog box (Figure 6-20). The dialog box indicates the starting and ending boundaries of the segment. Selecting *Yes* closes the dialog box and creates the new simulation segment. Selecting *No* closes the dialog box without creating the segment. Up to 15 segments may be created.

To delete a segment, position the leftmost measuring cursor within the segment to be deleted, then select *Delete Segment* to bring up the *Delete Segment* dialog box (Figure 6-21). The dialog box indicates the segment boundaries for the segment to be deleted. Selecting *Yes* closes the dialog box and deletes the segment. Selecting *No* closes the dialog box without removing the segment.



CREATE SEGMENT from 47 to 256

Figure 6-20 Create Segment Dialog Box



DELETE SEGMENT from 66 to 255

Figure 6-21 Delete Segment Dialog Box

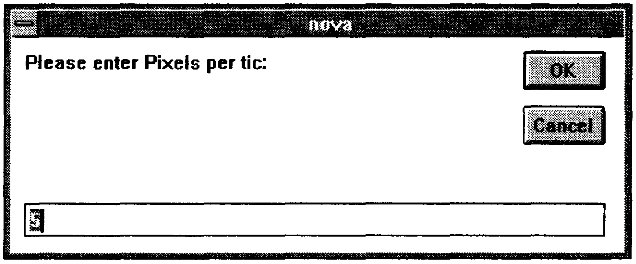
6.8.3 Resolution

Resolution allows the user to stretch and compress displayed traces.

Selecting *Resolution* from the *Options* menu brings up the *Resolution* dialog box (Figure 6-22). This dialog box allows the user to set the number of screen pixels on the X-axis to be used per simulation tic. Varying this number effectively stretches or compresses the traces displayed on the screen.

The pixels-per-tic setting may be any number between 1 and 100. The default is 5. The larger the number, the more "stretched" the traces appear; the smaller the number, the more compressed the traces appear.

Selecting *OK* closes the *Resolution* dialog box and updates the trace display. Selecting *Cancel* closes the *Resolution* dialog box without updating the trace display.



Please enter Pixels per tic:

5

Figure 6-22 Resolution Dialog Box

6.8.4 Signal Name Size

Signal Name Size allows the user to specify the width in characters of Nova's signal name buttons.

6.9 Nova JEDEC Simulator Quick Reference Sheet

6.9.1 Simulating a Circuit

- Start Nova.

  Select *Nova Functional Simulator* from the Galaxy tools menu or double-click on the *Nova* icon in the Windows program.

- Load the JEDEC file which was produced from *Warp.*

  Select *Open* from the *File* menu.

- Edit Input Stimulus.

  To edit a signal, select the signal with the left mouse button. Go to the *Edit* menu to set the signal high or low, or to configure the signal as a clock. To edit portions of the signal, select the portion with the mouse, then type "0" or "1" to set that portion of the signal high or low. There is also a "pulse" feature which allows the user to set up single pulses.

- Run Simulation.

  Select *Execute* from the *Simulate* menu.

6.9.2 Arranging Signals

Using the *Views* menu in Nova, the user can choose what signals he wants to see and in what order they are displayed. The default views are FULL, PINS ONLY, and PINS and REGS.

To create a new view, choose *Edit Views* under the *Views* menu and select *New View*.

When creating a new view, wild cards are recognized. To enter all available signals, select *Add by Name* and type an asterisk ("*"). Optionally, enter the names of the signals to see individually (or by using a combination of signal names and wildcards).

When done, click on *OK*. This view can now be edited. To add new signals to the view, double-click on the signal on the left-hand side. To delete a signal, double-click on the signal in the view (on the right-hand side). Alternately, the user can cut and paste signals in the view. Signals are always pasted under the currently selected signal.

The order of the signals can be changed by using cut and paste or by using *Add by Name*.

6.10 Creating Buses

- Select *Create Bus* from the *Edit* Menu.
- Choose an appropriate bus name under the *Bus Name* field of the pop-up menu.
- Add signal names to the bus by double clicking on the signal names on the left-hand side, selecting the signals, and clicking on *Add,* or by using *Add by Name*. Wildcards are allowed when using *Add by Name.*
- Click on *OK*.

If the signals are not in the current View, they cannot be added to a bus.

These signals must be deleted from the view if the user does not want to see them individually. Don't do this if it's an input bus because buses are only useful for output. Data cannot be input as a bus, only as individual bits of a bus.

To see the bus value, a measuring cursor is necessary. A measuring cursor is brought up by clicking the left mouse button in the white area near the bottom of the Nova window. During a single Nova session, the measuring cursor cannot be deleted once it has been activated. A second measuring cursor can be activated by holding the shift key down while clicking the left mouse button in the white area near the bottom of the Nova window.

6.11 Miscellaneous

To save input stimulus, view information, and buses for the next simulation, select *Write Sim* from the *File* menu.

Nova is purely a functional simulator. **There is no timing information in Nova.** There is only the concept of simulation tics. A given device may be modeled as several smaller blocks. For instance, a FLASH370 device can be divided into smaller parts:

- input cells
- PIM
- PTM
- macrocell
- I/O cell

As a result, the user may see a propagation between his input and his output. If the user experiences strange results, he should increase clock period or separate simultaneously changing input signals. Some rules of thumb (if there are problems) are to make pulses > 20 tics wide and provide > 10 tics of setup time.

Color of traces:

Blue means the waveform can be edited.

White means an input or an output that is three-stated.

Red means an output that is being driven.

Pin numbers or node numbers are displayed to the left of the signal name. Zoom control is available under the *Views* menu.

The simulation length may be changed by choosing *Simulation Length* under the *Options* menu.

Printing Nova Output (PC):

- Place the mouse cursor in the window to be captured.
- Hit *Alt* and *Print Screen* simultaneously.

  This will place the window in a buffer.
- Go into a text editor (MS Word/ Microsoft Write) and *Paste* from the buffer.
- Print from the text editor.

Chapter 7

Schematic Entry

7.1 Overview

The *Warp* tools use VHDL as the primary design entry mechanism. *Warp3,* however, also supports schematic entry as a design entry mechanism via ViewDraw. *Warp3* also supports mixed-mode design entry where portions of the design are entered in VHDL and portions are entered in ViewDraw, graphically.

When using ViewDraw, *Warp3* provides a very powerful and sophisticated user interface that allows users to capture designs efficiently. With *Warp3,* the user can:

- use VHDL descriptions, schematics, or both to describe any design
- compile and synthesize the resulting design description
- fit the resulting logic circuits into a particular PLD or CPLD, or place and route the design into an FPGA (the resulting files may be used for programming the device)
- verify the design with a timing simulator

There are several other tasks that can be performed, but this overview describes how to use ViewDraw for design entry. Figure 7-1 shows this process flow.

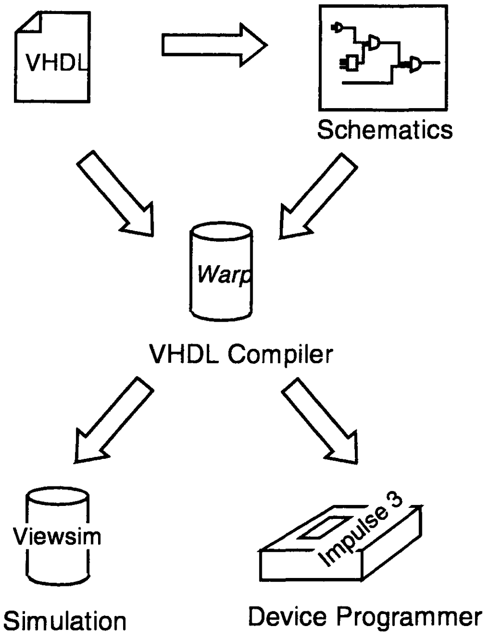


Figure 7-1 *Warp3* design flow

7.2 LPM Library

7.2.1 What Is LPM?

LPM is an acronym for Library of Parameterized Modules. This is a specification maintained by the Electronics Industries Association (EIA). The LPM specification contains a small set of highly parametrizable library elements. This specification is based on the EDIF (Electronic Design Interchange Format) version 2.0.0 standard and also specifies how data containing these parameterized modules can be interchanged between third party CAE systems.

Cypress has chosen the LPM standard for its schematic library because of its flexibility and interoperability. *Warp3* provides a graphical user interface to allow design entry with these LPM elements. With this graphical interface, the user can create, modify and manage LPM elements. To obtain a detailed description of the library and its functionality, the user should refer to Chapter 5, "LPM," of the *Warp Reference Manual.*

The rest of this chapter assumes that the user is familiar with ViewDraw and the Powerview or Workview PLUS environment.

7.2.2 How to Use LPM

Since LPM is a set of parameterized elements where the number and width of the pins can be varied, and the ViewDraw schematic capture system does not allow the pins for a given symbol block to vary, *Warp* automatically and dynamically creates and maintains custom symbols that are pre-programmed for a specific use.

For example, there is a common interface for an LPM_COUNTER. With this interface, the user can select or deselect many options such as enable, carry-in, or load. Instead of creating a symbol that has all possible pins for a given symbol, *Warp* automatically creates a custom symbol that has only those features required by the user. This is done because some of the LPM elements have a rather large number of optional features, and without a mechanism to create dynamic symbols, design entry with such symbols would be cumbersome.

When the user requests an LPM symbol configured in a certain way, *Warp* creates this element and stores it in a special library called *lpmlocal.* The *lpmlocal* library consists of a set of symbols and data files that manage all the symbols in a user's private library. The names assigned to these dynamically created symbols are meaningful only to the software and do not imply anything about the symbol itself. The *lpmlocal* library should never be edited by users manually. *Warp* automatically creates and manages this information.

ViewDraw uses the *viewdraw.ini* file to locate libraries. ViewDraw searches the current project directory as well as the directories listed in the WDIR environment variable for this initialization file. This file contains, among other things, a set of library names and the directories where these libraries can be found. A sample *viewdraw.ini* file is shipped with *Warp* and can be found in the *warpstd* subdirectory where *Warp* is installed. A portion of this file is shown here:

```
| Format: DIR [DirType(s)] DirPath (LibName)
|
|   DirType:    p or pw   - primary / writable
|               w         - writable (read/write)
|               r         - read-only
|               m or rm   - read-only megafile
|
|   DirPath:    directory specification
|
|   LibName:   library name aka library alias or VHDL library
|               name (optional) 32 characters or less.
                Must begin with a letters
DIR [p] .
DIR [r] c:\warp\lib\sheet (sheet)
DIR [r] c:\warp\lib\io (io)
DIR [r] c:\warp\lib\mcparts (mcparts)
DIR [r] c:\warp\lib\prim (primitive)
```

Lines starting with the " | " character are comments. The first directory below the comments is the current project directory, and the rest of the directories are libraries. To this list of libraries, another library must be added that represents the *lpmlocal* library. This library must be writable by the user because *Warp* creates symbols dynamically on behalf of the user. An example of such a library would be:

```
DIR [w] c:\mydir\myproj\lpmlocal (lpmlocal)
```

where *c:\mydir\myproj\lpmlocal* is a directory where *Warp* stores the symbols it creates. Without a valid location for the *lpmlocal* library, the *Warp* LPM functionality will be disabled. If this directory is being shared by other users in a network environment, this directory must be writable by everyone using this library. The *viewdraw.ini* file should be copied to the current project directory, and then this change should be made to the file.

7.2.3 Creating the lpmlocal Library

When ViewDraw is invoked for the first time in a new Viewlogic project, the LPM functionality is disabled and step-by-step instructions are printed on how to enable the LPM functionality and the creation of the lpmlocal library.

7.2.4 Creating an LPM Element

To create an LPM element once ViewDraw has been opened for editing a schematic, use the menu item *Add->LPM Symbol.*

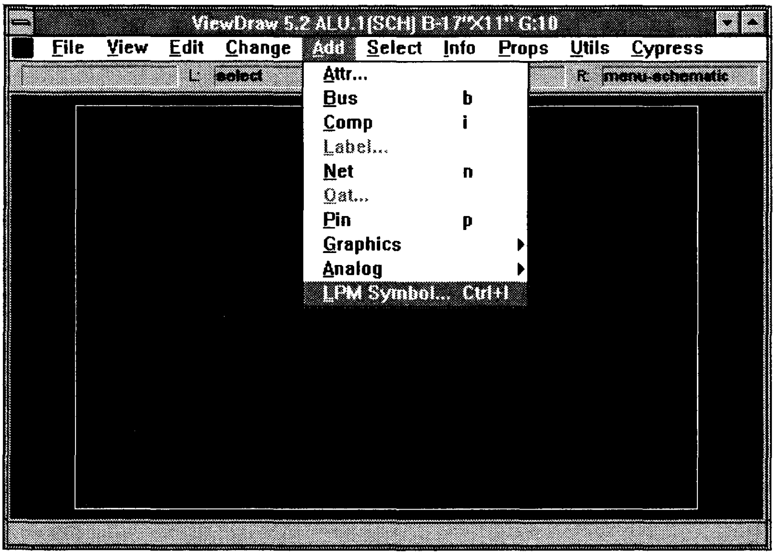


Figure 7-2 Add LPM Symbol

When this menu item is selected, ViewDraw prompts the user for the type of module to be instantiated. This dialog box is titled *Add Cell* and is shown in the following figure:

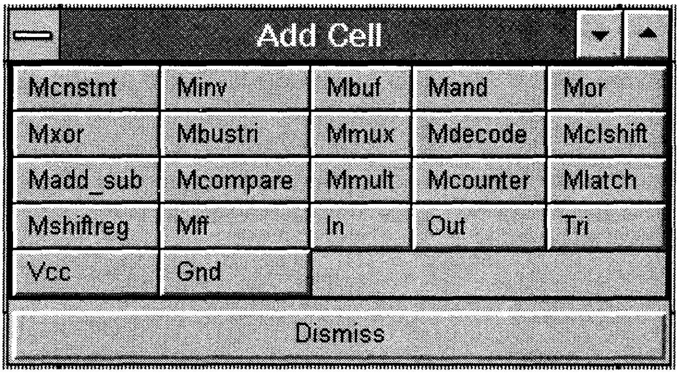
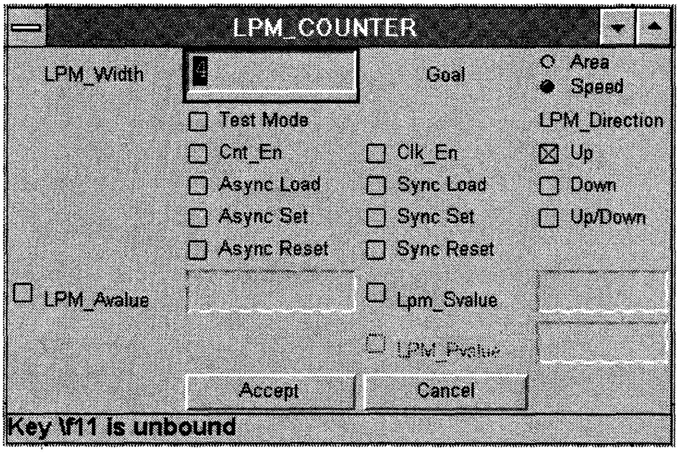


Figure 7-3 Add Cell dialog box

The user selects the desired module by single clicking the left mouse button. This action results in another dialog box that prompts the user to enter all the options that are applicable for the module selected. For example, if the Mcounter module was selected, the following dialog box would pop up:



LPM_Width 4

Goal ○ Area ● Speed

□ Test Mode

LPM_Direction

□ Cnt_En □ Clk_En ☒ Up

□ Async Load □ Sync Load □ Down

□ Async Set □ Sync Set □ Up/Down

□ Async Reset □ Sync Reset

□ LPM_Avalue □ Lpm_Svalue

□ LPM_Pvalue

Key \f11 is unbound

Figure 7-4 Mcounter dialog box

After selecting the appropriate items in this dialog box, a single mouse click on the *Accept* button removes this dialog box. At this point, the custom symbol that *Warp* has dynamically created is attached to the cursor and is ready to be placed in the schematic.

7.2.5 Modifying an LPM Element

If the user wishes to modify an LPM symbol already placed in the schematic, he should first select the LPM symbol to be modified and then choose the *Change->LPM Symbol* menu item. Only one LPM symbol may be selected at a time. When this menu item is selected, *Warp* displays the appropriate dialog box for the given LPM symbol, identical to the dialog box that was used during the initial creation of the LPM symbol.

7.2.6 Creating/Modifying a Non-LPM Element

A non-LPM element is essentially a user or library symbol which does not constitute a parameterized symbol. Instances of these elements are created using the regular ViewDraw methods. The *Add->Comp* menu item is used to create an instance of a non-LPM symbol, and the *Change->Comp* menu item should be used to change an existing instance. These menu items should not be used to edit or create instances of LPM symbols. Other than this restriction, an LPM symbol is similar to any other symbol within ViewDraw.

7.3 Exporting the Schematic

Once the schematic has been completed, the design can be converted into VHDL and compiled into a PLD, CPLD, or FPGA device. This can be accomplished by using the menu item *Cypress->Export VHDL:*

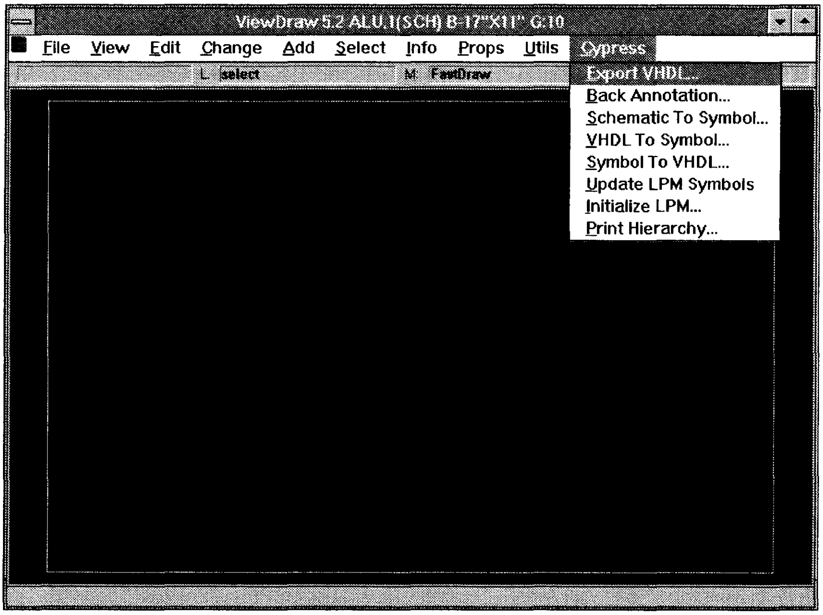
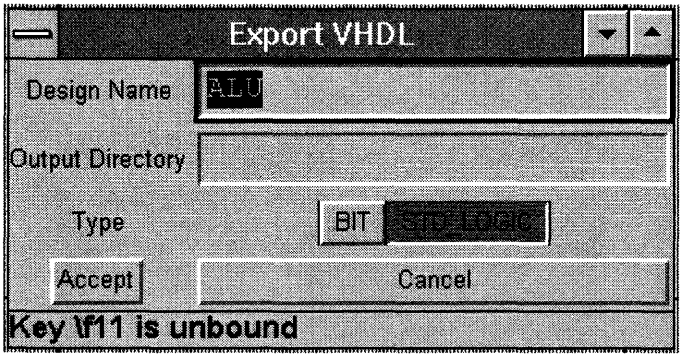


Figure 7-5 Export VHDL menu selection

When this option is selected, the following dialog box pops up:



Design Name ALU

Output Directory

Type BIT [✓ STD_LOGIC]

Key \f11 is unbound

Figure 7-6 Export VHDL dialog box

In this dialog box, *Design Name* is simply the name of the schematic being netlisted and *Output Directory* is the directory in which the netlist should be created. Leaving the *Output Directory* blank will create the netlist in the current project directory.

At this time, the user can also choose the type of netlist to be produced by the netlister. Currently, two types are supported: `bit` and `std_logic`. In VHDL, each signal has a type associated with it. This option simply allows a choice between these two different types. The `bit` type is supported only for compatibility with the previous release. The `std_logic` type is recommended for all new designs.

Clicking the left mouse button on the button marked *Accept* will cause the following actions:

- Check and Save the current schematic if it is not already saved.
- Invoke the batch program hi1076 to perform the actual netlisting.
- Netlist any synthesis directives found in the design.

The output file name has the same name as the top level design with a *.vhd* extension. This file also contains a hierarchical netlist for all the lower level blocks. Once this file is created, the design is ready to be synthesized using the *Warp* compiler.

7.4 Back-Annotation

Once a design has been successfully placed into a device, *Warp* allows the user to fix the pinout for that design.

To back-annotate pin-numbers into the design schematic, the user must select the menu item *Cypress->Back-Annotation....*

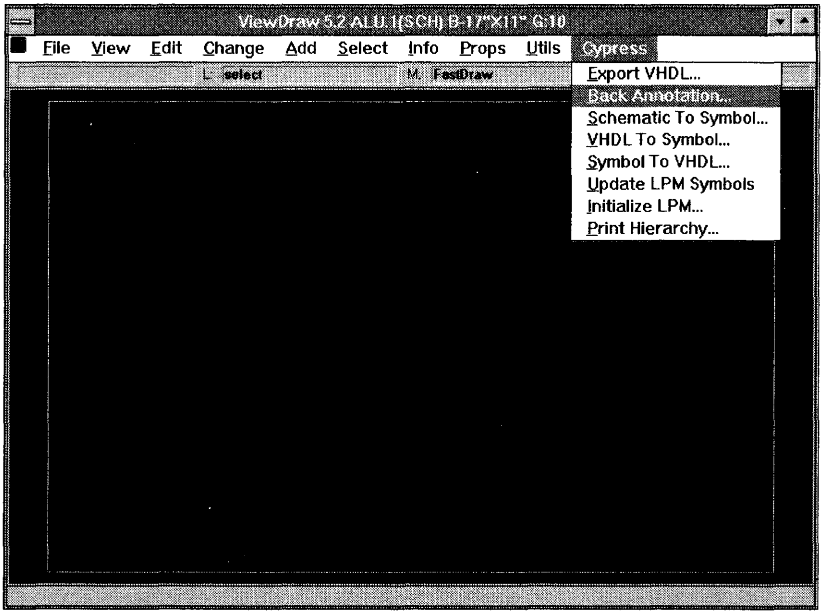


Figure 7-7 Back-Annotation menu selection

A simple dialog box appears showing the design name to be back-annotated. Clicking on *OK* does the following:

- Invokes a batch program that queries the pinout results and creates a list of pin names and their associated pin-numbers.
- Edits the current schematic (and all its associated sheets) to place the **#** attribute, so that future VHDL netlisting will force the pins to be placed in the same location.

The buses are back-annotated in a special way. Buses require that multiple pin-numbers must be back-annotated. This is accomplished by creating an attribute with a "," (comma) separated list of pin-numbers.

Note – Back-annotation will have no effect if the design has not been successfully fit or placed and routed into a device.

7.5 Using the Schematic Libraries from Release 3.5

The release 3.5 library and the release 4.0 library elements are not compatible with each other. To use the release 3.5 library mechanism, the user must do the following:

On the PC

In the *Warp R4* Program Group, invoke the program item named *Library*. This program will modify the *viewdraw.ini* file located in the *c:\warp\warpstd* directory as well as reconfigure the library directory in preparation for synthesis using the appropriate library. This will allow the user to create a new project directory via the Cockpit configured for either the release 3.5 or 4.0 library.

On UNIX Systems

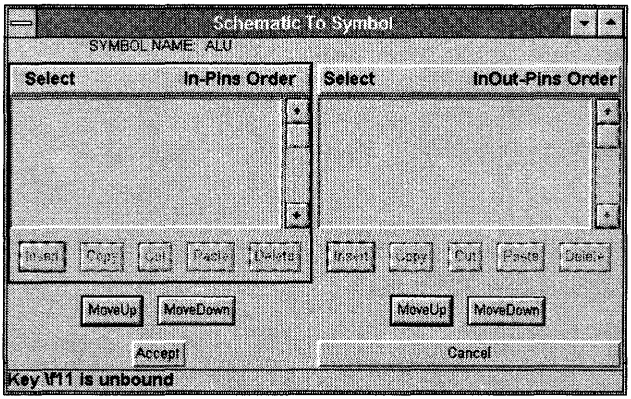
The user must first login as the user who installed *Warp* on the system, to ensure that he has the proper permissions to modify the installation directory and then execute the program **`cypver`**. This assumes that *$CYPRESS_DIR/bin* is in the user's path and that the environment variable *$CYPRESS_DIR* is pointing to the *Warp* installation directory. The **`cypver`** command modifies the *Warp* installation allowing the user to switch between the 3.5 and 4.0 libraries.

The above programs do not automatically modify all projects and any *viewdraw.ini* files that might exist in those directories. Following the template provided in *$CYPRESS_DIR/warp/warpstd/viewdraw.ini*, the user must modify his own *viewdraw.ini* files.

7.6 Schematic to Symbol

In *Warp3,* the user can use the *Schematic to Symbol* found in the Cypress menu to generate a symbol for a schematic circuit. The resulting symbol can then be instantiated in other, higher-level schematics.

When *Schematic to Symbol* is run, a dialog box allows the inputs and the outputs of the symbol to be reordered. Once the ordering of the pins is satisfied, clicking on *Accept* will create the symbol.


SYMBOL NAME: ALU

Key \f11 is unbound

Figure 7-8 Schematic To Symbol dialog box

Note – A new symbol cannot be generated if the symbol is already loaded into ViewDraw. To work around this problem, simply close all other ViewDraw windows or re-renter ViewDraw and only load the schematic for which the symbol is needed.

7.7 VHDL To Symbol

The VHDL To Symbol utility can be invoked in ViewDraw under the Cypress menu bar. This utility differs from the Viewlogic VHDL2sym tool, which can be found in the Circuit Design drawer. The Cypress version of the VHDL To Symbol translator requires that the VHDL file be first compiled using Galaxy as a non top-level file.

When this utility is invoked, a list of VHDL components for which symbols can be generated is displayed so that the user can select exactly which symbols need to be generated. If errors have been detected for symbols, the dialog box for VHDL To Symbol allows viewing these errors. The order of the pins for each of the symbols is determined by the order in which they were listed in the VHDL file. Please note that these VHDL components must be defined within a package.

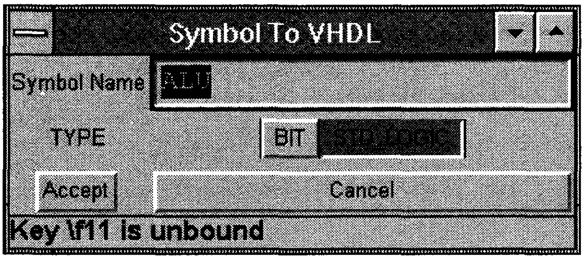
This utility is useful for designing in a bottom-up fashion, in which the user starts at the lowest level (being VHDL) and works up to a top-level graphical schematic.

> **Note** – A new symbol cannot be generated if the symbol is already loaded into ViewDraw. To work around this problem, simply close all other ViewDraw windows or re-renter View-Draw and only load the schematic for which the symbol is needed.

To run VHDL To Symbol, invoke the VHDL To Symbol and enter the name of the VHDL file (without the *.vhd*) extension.

7.8 Symbol to VHDL

Symbol to VHDL takes as input the name of a symbol, and translates a ViewDraw symbol into a VHDL file. The VHDL file has the same name as the symbol, except with a *.vhd* extension. This implies that the symbol name should be a VHDL legal name. The VHDL entity name is the same as the symbol name.



Symbol Name: ALU

TYPE: BIT [✓ STD LOGIC]

Key \f11 is unbound

Figure 7-9 Symbol to VHDL dialog box

7.9 Update Library

Since the *lpmlocal* library contains symbols that are sequentially named as the user requests new LPM symbols, it is highly likely that two different users using different *lpmlocal* libraries can have a like-named LPM symbol whose feature set may be completely different. Furthermore, a symbol with a given feature set may exist in one library and not in the other. Sharing or transporting of user schematics would therefore be impossible. To solve this problem, *Warp* provides a synchronization utility. Whenever a schematic is imported from another user, selecting the *Cypress->Update LPM Symbols* will ensure the integrity of the current schematic and its hierarchy by resolving any conflicts and regenerating all of the LPM symbols.

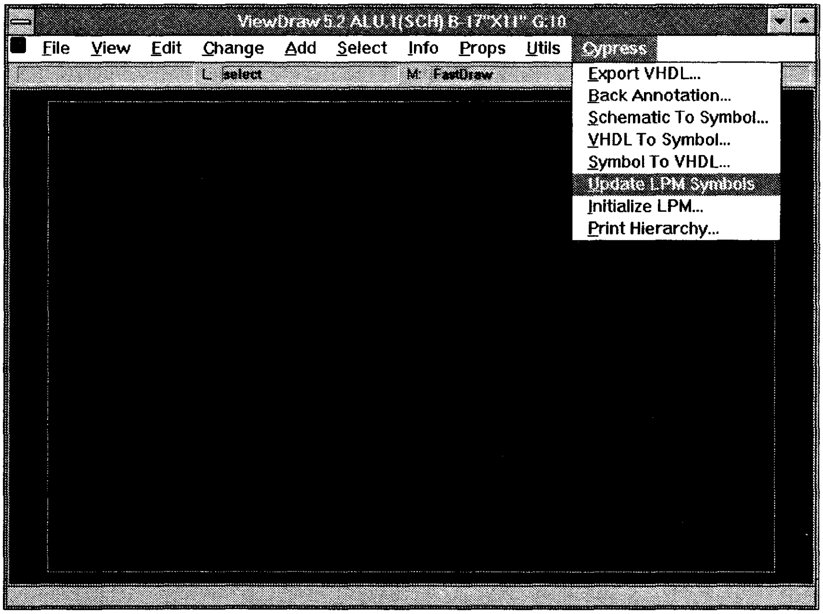


Figure 7-10 Cypress Update LPM Symbols

7.10 Print Hierarchy

This menu item prints the hierarchy for a schematic. This is helpful in being able to view a schematic's organization when the schematic contains many lower level schematics or modules. Please note that this utility cannot analyze the hierarchy of VHDL modules.

Chapter 8

Simulation

8.1 Introduction

Warp supports pre-synthesis VHDL simulation and post-synthesis VHDL and Verilog simulation. For post-synthesis simulation, *Warp* adheres to the following methodology: it generates all the VHDL and Verilog files required to simulate the design, and provides an easy way to integrate these HDL (Hardware Description Languages) files into the target simulation environment. In order to simulate the design, the user should be familiar with the desired simulation environment.

The VHDL and Verilog simulators supported are listed in Tables 8-1 and 8-2, respectively.

Table 8-1 Supported VHDL simulators

Simulator	Vendor	Pre-/Post-synthesis
ViewSim	Viewlogic	Post-synthesis
SpeedWave ™	Viewlogic	Pre-/Post-synthesis
V-System™/QuickHDL™	Model Technology/ Mentor Graphics	Pre-/Post-synthesis
VSS ™	Synopsys	Pre-/Post-synthesis
Leapfrog ™	Cadence	Pre-/Post-synthesis
IEEE1164 VHDL	N/A	Pre-/Post-synthesis

Table 8-2 Supported Verilog simulators

Simulator	Vendor	Pre-/Post-synthesis
VeriBest	Intergraph	Post-synthesis
VCS ™/Chronologic	Viewlogic	Post-synthesis
Verilog-XL ™	Cadence	Post-synthesis
IEEE1364 Verilog	N/A	Post-synthesis

Unless otherwise specified, pre- and post-synthesis simulation support is available for all devices.

8.2 Pre-synthesis Simulation

V-System

Scripts for compiling the Cypress pre-synthesis libraries into the user's work directory are available in *$CYPRESS_DIR/warp/lib/prim/presynth/scripts (c:\warp\lib\prim\presynth\scripts).* On UNIX platforms, to build the complete library for STD_LOGIC types, run the following command:

```
$CYPRESS_DIR/lib/prim/presynth/scripts/vsys_std
```

This command will compile all the necessary files in a work directory at the current location.

The `std_logic_1164`, `std_logic_arith`, `std_logic_unsigned`, `numeric_bit`, and `numeric_std` packages are already part of the compiled *ieee* library and accelerated for V-System/Workstation V4.4g (V-System/VHDL Windows V4.3g).

On PCs, for the Windows version of V-System, invoke the V-System, pull down the *File->Directory* and select the directory in which the library is to be compiled. Then in the *Transcript* window, the following is entered (note the "do" command):

```
do c:\warp\lib\prim\presynth\scripts\vsys_std
```

Similarly, to run pre-synthesis simulation using BIT types, use the following commands:

(V-System/UNIX Workstation)

```
$CYPRESS_DIR/lib/prim/presynth/scripts/vsys_bit
```

(V-System/VHDL Windows)

```
do c:\warp\lib\prim\presynth\scripts\vsys_bit
```

If the user already has command files written for ViewSim or SpeedWave (*.cmd*), they can be easily converted to V-System (*.do)* files. In order to make this conversion seamless, the user must not use the shorthand commands for ViewSim (i.e., **a** for assign, **c** for cycle, **l** for low, **h** for high, etc.). If the longhand conventions are used, they will map directly to the *.do* file syntax.

SpeedWave

Scripts for compiling the Cypress pre-synthesis libraries into the user's work directory are available in *$CYPRESS_DIR/lib/prim/presynth/scripts*. To build the complete library for STD_LOGIC types, run the following command:

```
$CYPRESS_DIR/lib/prim/presynth/scripts/spwv_std
```

This command will compile all the necessary files in a work directory at the current location.

To run pre-synthesis simulation using BIT types, run the following command:

```
$CYPRESS_DIR/lib/prim/presynth/scripts/spwv_bit
```

These commands will build the necessary directory for pre-synthesis simulation of the user's design. If the user already has command files for ViewSim, they can be used with SpeedWave with minor changes. All port signals must be prefixed with a / in the SpeedWave command file. This change is not backward compatible with ViewSim.

Before running the above scripts, make sure that the environment variable VANTAGE_VSS is set correctly, to point to SpeedWave root directory.

Other Simulators

For the rest of the simulators specified in Table 8-1, compile the packages in *$CYPRESS_DIR/lib/prim/presynth/std* or *$CYPRESS_DIR/lib/prim/presynth/bit* (on PCs, *c:\warp\lib\prim\presynth\std* or *c:\warp\lib\prim\presynth\bit)* and the VHDL design file into your *work* library and simulate using the target simulator commands. The proper order of compiling these files can be obtained by looking at one of the scripts in *$CYPRESS_DIR/lib/prim/presynth/scripts* *(c:\warp\lib\prim\presynth\scripts)*. The process for other simulators is similar to that mentioned above.

8.3 Post-synthesis Simulation Design Flow for PLDs and CPLDs

The design flow for the post-synthesis simulation support for Cypress PLD and CPLD devices is shown in Figure 8-1.

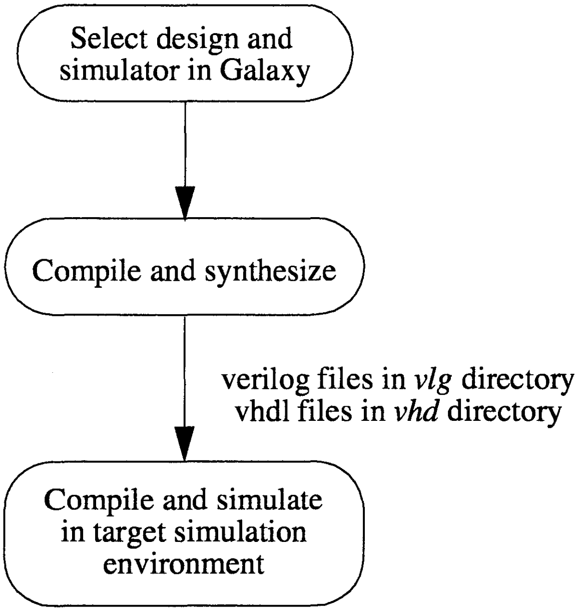

Select design and simulator in Galaxy
Compile and synthesize
verilog files in *vlg* directory
vhdl files in *vhd* directory
Compile and simulate in target simulation environment

Figure 8-1 Simulation design flow for PLDs and CPLDs

8.3.1 Select a Design

Refer to Chapter 4, "Galaxy," for details on how to select a design and a device.

8.3.2 Select a Simulator

The supported simulators are listed in the *Devices* dialog box of the *Galaxy* window, under the *Post-JEDEC Sim* section. Select the target device and package from the *Device* and *Package* menus, respectively, and the simulator from the *Post-JEDEC Sim* menu.

8.3.3 Compile a Design

After selecting the design, target device, and simulator, compile the design from the *Galaxy* window. *Warp* creates a set of VHDL or Verilog files which are required for simulation in the *vhd* or *vlg* sub-directories, respectively. The *vhd* and *vlg* sub-directories are created automatically if they do not already exist. The filenames for the post-synthesis simulation models will have the same base name as the top-level design file.

8.3.4 VHDL Simulation

V-System

A script for compiling the Cypress post-synthesis primitive libraries into the user's primitive directory is available in *$CYPRESS_DIR/lib/prim/presynth/scripts/vsysprim* (on PCs, *c:\warp\lib\prim\presynth\scripts\vsysprim).* On UNIX platforms, to build the complete primitives library run the following command:

```
$CYPRESS_DIR/lib/prim/presynth/scripts/vsysprim
```

This command will compile all the necessary files in a primitive directory at the current location.

On PCs, for the Windows version of V-System, pull down the *File->Directory* and select the directory in which the library is to be compiled. Then in the *Transcript* window the following is entered (note the "do" command), write the following command:

```
do c:\warp\lib\prim\presynth\scripts\vsysprim
```

Once the primitive library has been built, the target design can be compiled (vcom) and simulated (vsim) with commands such as the following:

- **`vcom vhd\<file name>.vhd`**
- **`vsim <entity name>`**

SpeedWave

A script for compiling the Cypress post-synthesis primitive libraries into the user's primitive directory is available in *$CYPRESS_DIR/lib/prim/presynth/scripts/spwvprim*. To build the complete primitives library run the following command:

```
$CYPRESS_DIR/lib/prim/presynth/scripts/spwvprim
```

Once the primitive library has been built, the target design can be compiled into a selected work area using the following command:

```
analyze -dbg 2 -libfile vsslib.ini -src vhd/<file name>.vhd
```

The simulation process at that point is the same as for ViewSim simulation.

Other Simulators

For the rest of the simulators specified, compile the packages in *$CYPRESS_DIR/lib/prim/vhdl* (on PCs, *c:\warp\lib\prim\vhdl*) and the VHDL design file into your *primitive* library and simulate using the target simulator commands. The proper order of compiling these files can be obtained by looking at one of the scripts in *$CYPRESS_DIR/lib/prim/presynth/scripts/*prim* (on PCs, *c:\warp\lib\prim\presynth\scripts\prim). The process for these other simulators is similar to that mentioned below for Verilog Simulation.

8.3.5 Verilog Simulation

In order to simulate the design, the user should be familiar with the target simulation environment. When a Verilog simulator is selected, *Warp* creates a template file which assists the user in submitting the correct set of Verilog files, in the proper order, to the target Verilog compiler. The template file, whose name and format vary with the target simulator, is created in the *vlg* directory. The steps needed to compile the design in different simulator environments are described below.

VeriBest

The template file that *Warp* creates is called *design_name.sup*. Its format is conformed to the support file format within VeriBest (refer to the VeriBest simulator manual for details). Load the support file into the VeriBest environment (*File->Open_Setup_File*) and select the *analyze* command to compile. The design is now ready for simulation in the VeriBest environment.

VCS

The template file that *Warp* creates is called *design_name.fls*. This file contains the list of files and their respective order to be compiled with the Verilog compiler. Specify this file name in the VCS command line, as shown below.

```
vcs -f <top_level_design>.fls
```

Verilog-XL

The template file that *Warp* creates is called *design_name.fls*. This file contains the list of files and their respective order to be compiled with the Verilog compiler. Once the files are compiled, they are ready for simulation in the Verilog-XL environment.

Note – Make sure that the *vlg* directory is in the search path of the target simulator.

8.4 Post-synthesis Simulation Design Flow for FPGAs

The design flow for the post-synthesis simulation support is shown in Figure 8-2.

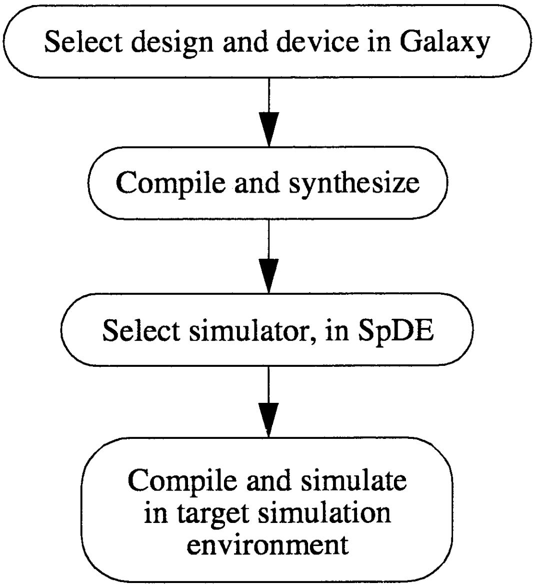

Select design and device in Galaxy
Compile and synthesize
Select simulator, in SpDE
Compile and simulate in target simulation environment

Figure 8-2 Simulation design flow for FPGAs

8.4.1 Select a Design

Refer to Chapter 4, "Galaxy," for details on how to select a design and device.

8.4.2 Compile a Design

After selecting the design and target device, compile the design from the *Galaxy* window.

8.4.3 Select a Simulator

A variety of simulators are supported for post-synthesis simulation. The supported simulators are listed in the *Tools->Options->Simulator* dialog box within the SpDE place and route tool. Select the target simulator from this menu. See Chapter 5, "SpDE" for more information on the SpDE place and route tool.

8.4.4 Run SpDE

After selecting the simulator, run SpDE with the back-annotation tool selected.

8.4.5 ViewSim Simulation

Warp3 integrates directly into the Viewlogic Powerview and Workview PLUS environments, and FPGA post-synthesis simulation is fully supported with the ViewSim simulator. After running SpDE, the spde2vl executable must be run. This program is run by double-clicking on the *pASIC->VSim* icon in the Cockpit. This utility will generate the necessary files for ViewSim simulation.

8.4.6 VHDL Simulation

For VHDL Simulation, simply select the appropriate simulator from the SpDE *Tools->Options->Simulator* menu and run the back-annotation tool from within SpDE. This will create a *.vhd* file and a *.sdf* file compliant with the VITAL specification. These files in conjunction with the VHDL primitive models provided allow the user to simulate a design with any VITAL compliant VHDL simulator.

8.4.7 Verilog Simulation

When a Verilog simulator is selected, *Warp* creates a verilog design file *(design.vq)* and a delay back-annotation file *(design.sdf)* where *design* is the top-level design name. The device specific primitives used in the design are available in *$CYPRESS_DIR/spde/data/qlprim.v* (on PCs, *c:\warp\spde\data\q\prim.v*). To simulate the design, compile *design.vq* and the above mentioned primitive file in the target simulator environment.

8.5 Post-synthesis VHDL Simulation in ModelT Environment

The following are the steps required for post-synthesis/layout simulation of pASIC targeted designs with Model T's V-System:

- A *qlprims* library needs to be created with the *mtiprim.vhd* file which is supplied by Cypress. Follow these steps:

  Create a *qlmodel* directory.

  Copy the file *mtiprim.vhd* (from *$CYPRESS_DIR/spde/data*) to the *qlmodel* directory.

  While in the *qlmodel* directory, create a new library called *qlprims* with the V-System's vlib.

  ```
  vlib qlprims
  ```

  Map the *qlprims* library to it's source:

  ```
  vmap qlprims <path>/qlmodel/qlprims
  ```

  Compile the *mtiprim.vhd* file to the *qlprims* library:

  ```
  vcom -work qlprims mtiprims.vhd
  ```

- Load design (<*design.qdf*>) into SpDE and select Model Tech V-System from the *Tools* -> *Options* -> *Simulator* menu. SpDE will create <design>.vhq and <*design*>*.sdf* files when the tools are run.

Note – As SpDE creates the *.vhq* file, it may inform you that vectors in your entity will be broken out into individual signals unless you have a *.vhh* file. Please ignore these messages as this feature is not yet supported by *Warp*.

- Compile <design>.vhq:

  ```
  vcom <design>.vhq
  ```

- Simulate:

  ```
  vsim -t ps -sdftyp/-sdfmin/-sdfmax <design>.sdf
  <design>
  ```

Note – '-t ps' must be used because timing numbers in the SDF file are in picoseconds. Because of this you must be careful about the default cycle time which is 100 time units (in this case 100ps). Either reset the default time units/cycle or explicitly indicate a time for your 'run' statements in the *.do* file to prevent surprises.

8.6 Post-synthesis Verilog Simulation In VeriBest Environment

Following are the steps involved in the post-synthesis simulation of a CPLD targeted design in the Intergraph VeriBest environment.

- Select Intergraph from *Devices* dialog box of Galaxy and compile the design.
- Create a test bench model to give test vectors to the design. Following is a test bench model:

```
module <design>_tbench () ;
   // test bench name is <design>_tbench
reg in1, .... ;
wire out1, .... ;

initial
begin
   // specify test vectors
end

   // instantiate the design
<design> inst1 (in1, ....., out1, .....) ;
   // In the above, <design> is the Verilog model name
      of the design. It is created by Warp and is in
      the file <design>.vlg in vlg sub-directory

endmodule
```

- Load and compile the Verilog files generated by *Warp* into the Veribest™ environment and simulate the design.

```
% veribld

   File->Open_Setup_File <design>.sup
   Analyze
   Simulate
   // in the above, File, Analyze, Simulate are the
      menu buttons in veribld
```

Following are the steps involved in the post-synthesis simulation of a pASIC targeted design in the Intergraph VeriBest environment:

- Load design (*<design>.qdf*) into SpDE and select Verilog from the *Tools->Options->Simulator* dialog box of SpDE
- Run SpDE tools from *Tools->Run Tools* dialog box making sure that the back-annotation option is selected.
- Create a test bench model to give test vectors to the design. The following is a test bench model:

```
module <design>_tbench () ;
   // test bench name is <design>_tbench
reg in1, .... ;
wire out1, .... ;

initial
begin
   // specify test vectors
end

   // instantiate the design
<design> inst1 (in1, ....., out1, .....) ;
   // In the above, <design> is the Verilog model name
      of the design.  It is created by SpDE and is in
      the file <design>.vq

   // Include <design>.sdf file generated by SpDE
initial
begin
$sdf_annotate(“<design>.sdf”,<design>_tbench.inst1) ;
end

endmodule
```

- Load and compile the Verilog files generated by SpDE into the Veribest environment and simulate the design.

  ```
  % veribld

     Add->$CYPRESS_DIR/spde/data/qlprim.v
     Add-><design>.vq
     Add-><design_tbench>
     Analyze
     Simulate

     // in the above, Add, Analyze, Simulate are the
        menu buttons in veribld
  ```

Note – Refer to the Verilog language reference manual and Simulator guide for details of test bench model and simulator usage.

Chapter 9

Synthesis

9.1 Synthesis Directives

This chapter introduces synthesis directives—what they are, what they are used for, how to use them, and when to use them. This chapter is organized into five sections. The first section is an introduction. It explains directives and discusses a strategy for using them effectively. It also includes two design examples to illustrate how to apply them. The second section describes those directives that can be used to optimize a design for the fewest device resources. The third section describes those directives that can be used to optimize a design for timing goals, including operating frequency, clock to output delay, setup time, and combinatorial propagation delays. The fourth section describes directives used for controlling the type and location of specific resources used in a device. The final section describes directives used for documentation, including part selection and pin number assignment.

9.1.1 Understanding Synthesis Directives

Synthesis directives may be used to influence the implementation of a design. They are used in an iterative fashion to refine, improve, or constrain the results of synthesis. For example, the `goal` directive is used by the synthesizer to select either area-efficient or speed-efficient design implementations. Synthesis directives may be applied to components that have been either instantiated in a schematic or inferred by the synthesizer from VHDL code. The `buffer_gen` directive causes buffers to be inserted for high-fanout signals. `Synthesis_off` creates a factoring point for logic equations and is used for area or speed optimization (or both). The `pin_numbers` directive specifies the pin numbers to be used for signals. These and other directives are discussed in the pages to follow, but the following section discusses a strategy for designing with synthesis directives.

9.1.2 Design Flow and Strategy for Using Directives

Directives are a powerful mechanism to influence the synthesis process, but they should be used judiciously. Careless or excessive use of directives can, in fact, subvert the very design goals that are sought. This section describes a strategy for using directives and choosing the appropriate one(s) to achieve the user's goals.

Until the user becomes familiar with the effects of using the different directives, Cypress does not recommend applying any of them in the first iteration of a design. After synthesis and fitting—or place and route, in the case of an FPGA design—the design may fit in the desired device and meet timing goals. In this case, the design is complete—no directives are necessary. If, however, after the initial iteration of synthesis and fitting, the design does not fit or meet timing goals, the design may need tuning. *Tuning,* illustrated in Figure 9-1, is the process of (1) identifying and applying an appropriate directive that may help to reduce resource utilization or realize timing targets, (2) resynthesizing and fitting the design, and (3) verifying that the design meets area and speed goals. In some cases, this tuning process may have to be repeated in order to compare multiple implementations of the design.

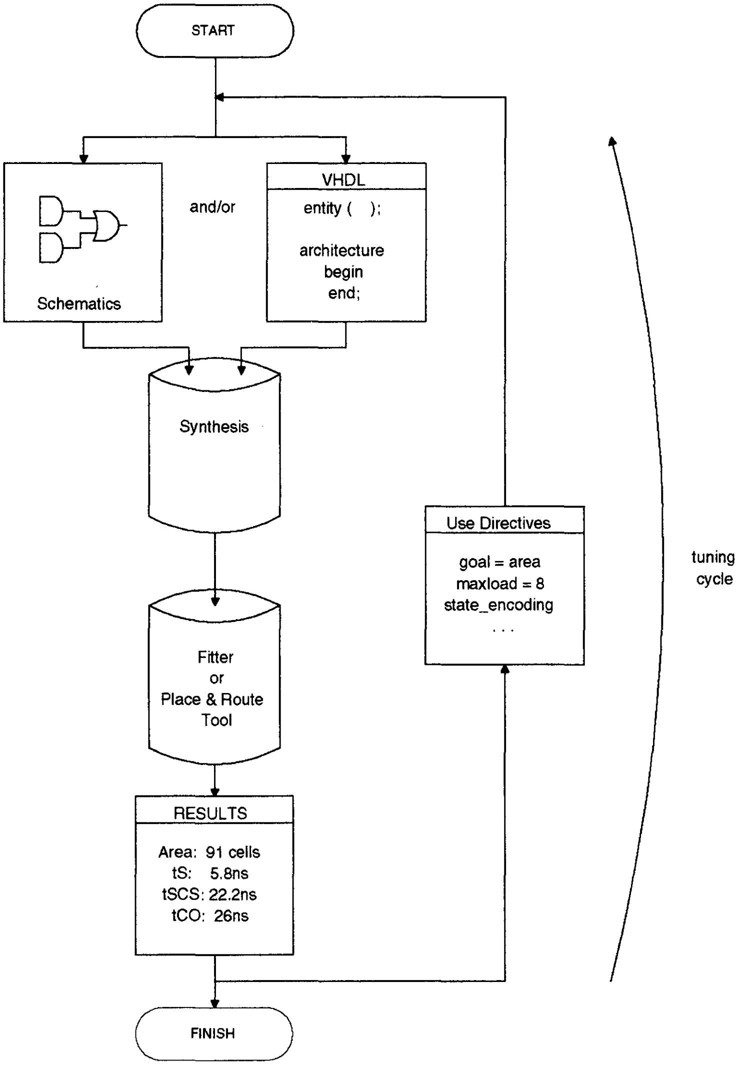


Figure 9-1 Tuning

9.1.3 Available Directives

Table 9-1 can be used to select an appropriate directive for tuning a design. Those directives listed first are most likely to have the greatest impact on a design implementation and should be selected first when tuning. The other directives are used in special cases or for documentation purposes. Device selection and pin number assignment are included in the documentation category, although they are also functional directives that can have a significant impact on area and speed. Later in this chapter, each of the directives listed in the table is explained in greater detail, with the focus on understanding scenarios when using a particular directive is appropriate. The syntax and effect of all directives is explained in "Synthesis Directives," Chapter 3, of the *Warp Reference Manual.*

For each of the directives listed in Table 9-1, the "Applicable Devices" column indicates whether the directive is useful for CPLDs, FPGAs, or both. The "Used for..." column indicates whether the directive can be used for area optimization, speed optimization, specific control, or documentation.

The next section describes how to apply directives.

Table 9-1 Available Synthesis Directives

Directive	Applicable Devices		Used for...			
	CPLDs	FPGAs	area	speed	control	doc.
goal	x	x	x	x		
state_encoding	x	x	x	x		
buffer_gen		x	x	x		
max_load		x		x		
pad_gen		x		x	x	
synthesis_off	x	x	x	x	x	
dont_touch		x	x	x	x	
no_latch	x		x	x	x	
lab_force	x				x	
pin_avoid	x				x	
polarity	x				x	
sum_split	x				x	
node_num	x				x	
fixed_ff		x			x	
ff_type	x				x	
no_factor	x	x			x	
opt_level	x	x	x	x	x	
part_name	x	x	x	x		x
order_code	x	x	x	x		x
pin_numbers	x	x	x	x		x

9.1.4 Scope and Inheritance

Each of the synthesis directives has a scope: some are intended for signals, others for components. Some of the directives also have an inheritance. A directive intended for a signal can be placed on an architecture or entity so that all signals defined in that architecture or entity inherit that directive. This is called hierarchical inheritance. Not all directives have an inheritance, however. Non-hierarchical directives are meant for the exact object that they are attached to and will be ignored if not applied to the appropriate object.

Hierarchical directives have the following order of precedence (from least to greatest):

- entity
- architecture
- component declarations
- component instantiations
- signals

Thus, a hierarchical directive placed on an architecture is overridden by a directive placed on a signal within that architecture. In other words, a hierarchical directive intended for a signal, if placed on an architecture, serves as a default for all signals within that architecture. Likewise, a hierarchical directive placed on a component instantiation overrides a directive placed on an architecture. This allows for an occurrence of a component to have a different value than the default directive for all components.

9.1.5 Applying Directives

Some directives are available via the command line or Galaxy switches. *Warp* also provides three other methods for applying synthesis directives: with VHDL attributes, with schematic attributes, or with a top-level control file. Values of directives passed through the GUI or the command line act as default values. Directives applied using VHDL attributes, schematic attributes, or the control file override default values. The only exceptions are the **`part_name`** and **`order_code`** directives. The GUI or command line, discussed below, will override all **`part_name`** and **`order_code`** attributes.

Using the GUI or command line. Certain directives may be controlled from the GUI or command line. An example of this is the **goal** attribute which can be selected to provide area or speed optimization. If speed is selected, then it becomes the default value. If a component has a VHDL or schematic goal attribute applied to it, however, and the value of the attribute is area, then the speed value is overridden with the area value for that component.

Using VHDL attributes. VHDL permits the use of user-defined attributes to adorn objects with information. *Warp* has thus created a user-defined (as opposed to pre-defined) attribute for each directive. This permits a directive to be applied to an object with the use of an attribute. The general syntax of an attribute used to place a directive on a signal is the form:

attribute *directive_name* of *object*:class is *value*;

Such attributes are placed in the appropriate declarative region of the VHDL code, typically in either the entity declarative region or the architecture body declarative region. The object is the actual name or identifier of the entity, architecture, component instantiation label, or signal. Class is used to identify the class of the object (i.e., entity, architecture, or component instantiation label, or signal).

Examples of applying directives using attributes are given below. Next is a discussion of the application of directives with schematic attributes and a top-level control file.

Using schematic attributes. Directives may be applied to objects in schematics (with *Warp3*) using attributes by selecting the appropriate object and choosing *Attribute* from the *Add* menu. After selecting *Add->Attribute,* a dialog box appears in which the user may enter the directive in the form:

directive_name=value

The **goal** directive for area or speed optimization is **not** applied as an attribute. It is chosen through the *Add->LPM Symbol* dialog box. The directive chosen here overrides the command line or GUI switch.

Using a control file. A top level control file may also be used to specify synthesis directives. In the case of conflict, directives placed in a control file override directives specified with VHDL or schematic attributes. The format of the control file is defined in Chapter 3, "Synthesis Directives," of the *Warp Reference Manual.* Each directive may be applied in the control file using a syntax similar to that of attributes:

attribute *directive_name* **[of]** *object***[:class] is** *value***[;]**

The words in square brackets [ ] are optional and are simply ignored. Specifying the class is also optional.

The next section illustrates how to apply directives in a design by using the tuning strategy shown above. The two examples shown below demonstrate the merits of both CPLDs and FPGAs. These design examples were compiled using a pre-release version of the *Warp* software. Your results may vary slightly from those presented here, but the general concepts will remain true.

9.2 Example 1—DRAM Controller

The code of the following listing is used to describe a fictitious DRAM controller. Understanding the details of the code is not necessary for comprehending the subsequent design optimization strategy. This example will first optimize this design for a pASIC380 FPGA and then retarget it to a FLASH370 CPLD.

```
library ieee;
use ieee.std_logic_1164.all;
entity example is port(
   clk, rst, ads, burst:in std_logic;
   address: in std_logic_vector(31 downto 0);
   cas, ras, ack,  ref: buffer std_logic;
   row_col_address:out std_logic_vector(11 downto 0));
end example;

use work.std_arith.all;
architecture controller of example is
   type states is (idle, asdet, rasa, casa, w1, w2, w3,
      nocas, refad, wr1, wr2);
   signal state, next_state: states;
   signal match, ref_req:std_logic;
   signal count: std_logic_vector(23 downto 0);
   signal captured_address: std_logic_vector(31 downto 0);
   signal captured_burst:std_logic;
   signal col_ad:std_logic_vector(11 downto 0);
   signal burst_cnt:std_logic_vector(1 downto 0);

   constant re_ad:std_logic_vector(11 downto 0) := (others
      => ‘0’);

   alias row_ad: std_logic_vector(11 downto 0) is
captured_address(23 downto 12);
begin


-- latch in address, and value of burst
adreg: process (clk, rst)
   begin
      if rst = ‘1’ then
         captured_address <= (others => ‘0’);
         captured_burst <= ‘0’;
      elsif clk’event and clk= ‘1’ then
         if ads = ‘1’ then
            captured_address <= address;
            captured_burst <= burst;
         end if;
      end if;
   end process;

-- check address contents to see if memory access
   match <= ‘1’ when captured_address(31 downto 24) =
      “00000000” else ‘0’;
```

```
-- DRAM address multiplexer
mux: process (state, col_ad, row_ad)
   begin
      case state is
         when refad | wr1 | wr2 =>
            row_col_address <= re_ad;
         when rasa | casa | w1 | w2 | w3 =>
            row_col_address <= col_ad;
         when asdet  =>
            row_col_address <= row_ad ;
         when others =>
            row_col_address <= (others => '-');
      end case;
   end process;

-- column address, Intel order
   col_ad(11 downto 2) <= captured_address(11 downto 2);
   col_ad(1) <= captured_address(1) xor burst_cnt(1);
   col_ad(0) <= captured_address(0) xor burst_cnt(0);

-- Burst counter:
bcount: process (clk, rst)
   begin
      if rst = '1' then
         burst_cnt <= "00";
      elsif clk'event and clk = '1' then
         if state = idle then
            burst_cnt <= "00";
         elsif state = w3 then
            burst_cnt <= burst_cnt + 1;
         end if;
      end if;
   end process;
```

```
-- DRAM refress request counter
counter: process (clk, rst)
   begin
      if rst = '1' then
         count <= (others => '0');
      elsif clk'event and clk = '1' then
         if ref = '1' then
            count <= (others => '0');
         else
            count <= count + 1;
         end if;
      end if;
   end process;
   ref_req <= '1' when count = "101010101010101010101000"
else '0';

-- DRAM state machine
control: process (state, ref_req, match)
   begin
      case state is
         when idle =>
            cas <= '1'; ras <= '1';
            ack <= '1'; ref <= '0';

            if ref_req = '1' then
               next_state <= refad;
            elsif ads = '1' then
               next_state <= asdet;
            end if;

         when asdet =>
            cas <= '1'; ras <= '1';
            ack <= '1'; ref <= '0';

            if match = '1' then
               next_state <= rasa;
            else
               next_state <= idle;
            end if;

         when rasa =>
            cas <= '1'; ras <= '0';
            ack <= '1'; ref <= '0';

            next_state <= casa;
```

```
when casa =>
   cas <= ‘0’; ras <= ‘0’;
   ack <= ‘1’; ref <= ‘0’;

   next_state <= w1;

when w1 =>
   cas <= ‘0’; ras <= ‘0’;
   ack <= ‘1’; ref <= ‘0’;

   next_state <= w2;

when w2 =>
   cas <= ‘0’; ras <= ‘0’;
   ack <= ‘1’; ref <= ‘0’;

   next_state <= w3;

when w3 =>
   cas <= ‘0’; ras <= ‘0’;
   ack <= ‘0’; ref <= ‘0’;

   if (captured_burst = ‘1’ and burst_cnt /= “11”)
   then
      next_state <= nocas;
   else
      next_state <= idle;
   end if;

when nocas =>
   cas <= ‘1’; ras <= ‘0’;
   ack <= ‘1’; ref <= ‘0’;

   next_state <= casa;

when refad =>
   cas <= ‘1’; ras <= ‘0’;
   ack <= ‘1’; ref <= ‘1’;

   next_state <= wr1;

when wr1 =>
   cas <= ‘1’; ras <= ‘0’;
   ack <= ‘1’; ref <= ‘0’;

   next_state <= wr2;
```

```
            when wr2 =>
               cas <= ‘1’; ras <= ‘0’;
               ack <= ‘1’; ref <= ‘0’;

               next_state <= idle;
         end case;
      end process;

   -- clock state machine
   clocked: process (clk, rst)
      begin
         if rst = ‘1’ then
            state <= idle;
         elsif clk’event and clk = ‘1’ then
            state <= next_state;
         end if;
      end process;

   end controller;
```

9.2.1 FPGA Optimization

9.2.1.1 First Pass -- Default Options

On the first pass through synthesis and the place and route tools, this example uses the default Galaxy options—buffer generation on, pad generation on, and speed optimization for inferred arithmetic components. In the synthesis report file, several operators are inferred:

```
ex3.vhd (line 49, col 68):  Note: Substituting module
   'warp_cmp_1s1c_ss' for '='.
ex3.vhd (line 80, col 32):  Note: Substituting module
   'warp_add_1s1c_ss' for '+'.
ex3.vhd (line 94, col 24):  Note: Substituting module
   'warp_add_1s1c_ss' for '+'.
ex3.vhd (line 98, col 61):  Note: Substituting module
   'warp_cmp_1s1c_ss' for '='.
ex3.vhd (line 152, col 52):  Note: Substituting module
   'warp_cmp_1s1c_ss' for '/='.
```

The two "+" operators are used in counters. The "=" and "/=" operators are used for arithmetic comparisons. The following buffers and pads are inserted:

```
-----------------------------------------------------------
Begin PAD Generation.
-----------------------------------------------------------
Created CLKPAD for signal 'clk'
    Above signal drives 63 Clocks,  0 Set/Resets. Total = 63
Created CLKPAD for signal 'rst'
    Above signal drives  0 Clocks, 63 Set/Resets. Total = 63
Created HD1PAD  for signal 'ads'
    Above signal drives  0 Clocks,  0 Set/Resets, 34 other
   inputs. Total = 34

-----------------------------------------------------------
Begin Buffer Generation.
-----------------------------------------------------------
[max_load =  7, fanout = 18] Created 2 buffers [Duplicate]
   for 'MODULE_5_s0_g1_u0_c_1'
[max_load =  7, fanout = 11] Created 1 buffers [Duplicate]
   for 'MODULE_5_s0_g1_u0_c_2'
[max_load = 13, fanout = 25] Created 1 buffers [Duplicate]
   for 'ref_OUT'
[max_load = 13, fanout = 18] Created 2 buffers [Normal   ]
   for 'stateSBV_2'
```

Clock pads were automatically selected for the clock and reset signals because they fanout to all 63 of the flip-flops used. A high-drive pad, HD1PAD, was selected for signal `ads` because it has a large internal load. Buffers were created as well, per the defaults. The modules that were inferred have their own buffering requirements, and the remainder of the signals in the design are buffered if their loads are greater than 13, which is the default value of the **max_load** directive.

The design is imported into SpDE for place and route. In the *Tools->Options->General* dialog box of SpDE, select the level 2 area optimization (L2 area) for technology mapping. Then run all the tools, record the logic cell utilization information from the *Info->Utilization* information box, and gather the requisite timing information to calculate setup time (t_S), clock to output time (t_{CO}) and maximum clock period (t_{SCS}) for internal operation (see Chapter 5, "SpDE," to calculate setup times, clock to output delays, and operating frequency). Next, choose level 2 speed (L2 speed) optimization and gather the same information. The results are summarized in Table 9-2. For each category (area, t_S, t_{SCS}, and t_{CO}), the best result is indicated by shading in the appropriate cell of the tableThe limiting factor for operating frequency is also listed. In this case, even though the design can be internally clocked for a clock period of t_{SCS}, the t_{CO} value is such that clocking at this interval would result in the outputs never being valid.

Table 9-2 First pass FPGA results

	L2 Area	L2 Speed
Area (logic cells)	108	116
t_S (ns)	14.4	14.9
t_{SCS} (ns)	22.0	20.6
t_{CO} (ns)	39.4	32.1
limiting factor	t_{CO}	t_{CO}

If the area and speed of this implementation are acceptable, then the job is done—the design does not need tuning. If, however, the user wishes to tune this design to improve either the area, speed, or both, then he must begin the tuning cycle. For the sake of continuing this example, assume that the user has not met his goals and first optimize this design for speed, then area. For speed improvement, the user wishes to decrease the setup time and clock to output delay. Assume that the user wants to run the design with a 40 ns clock period (25 MHz), with a 10 ns setup time and 30 ns clock to output delay. If the results were only a couple of nanoseconds from the desired goals, then the path analyzer in the place and route tool should be used to enter timing constraints. The place and route tool could then replace and reroute in an attempt to meet those constraints. For moderate improvement, returning to synthesis with directives is appropriate. If the user wanted to be significantly more aggressive with the clock to output delay, he would want to consider registering the outputs. The outputs are currently designed to be combinational outputs decoded from the current state of the state machine. Adding a pipeline for registering outputs is discussed in the chapter covering state machines in the VHDL text accompanying this documentation set. For this tutorial, the discussion will focus on using directives along with place and route constraints to meet the timing goals.

9.2.1.2 Second Pass—Speed Optimization (First Tuning Cycle)

This begins the first tuning cycle. Look at Table 9-1 to determine which directives are applicable. The t_{SCS} goal has been achieved but not the t_S or t_{CO} goal. The second pass will use two directives to influence the synthesis process and improve timing: (1) The pad generation directive is used to force ads to two high drive pads (HD2PAD). This is done to improve setup times. In FLASH370 CPLDs, delays are not dependent upon internal loading of signals. With the pASIC380 FPGAs, delays are dependent upon fanout. A high drive pad increases the drive for the `ads` signal and reduce propagation delay. (2) The **state_encoding** directive is used to select one-hot encoding for the state machine. This is done to potentially improve the operating frequency and perhaps save logic cells. With sequential encoding, the 11 states of the state machine will require 4 state bits. With one-hot, 11 state bits will be required; however, with one-hot encoding, the next-state logic is simpler, so fewer overall logic cells (and fewer levels of logic cells) may be required. None of the other directives, including **buffer_gen**, will be used in this pass, to avoid introducing too many directives at the same time. Using too many directives at once limits the user's ability to determine which of them is helping or potentially hurting. Both attributes are placed directly in the code. The **pad_gen** attribute is placed in the entity declaration region, directly before the `end` statement. The **state_encoding** attribute is placed immediately after the type declaration. The attributes are:

```
attribute pad_gen of ads:signal is pad_hd2;
attribute state_encoding of states:type is one_hot_one;
```

After synthesis, the report file indicates that **one_hot_one** encoding is used:

```
State variable 'state' is represented by a Bit_vector
   (0 to 10).
State encoding (one-hot one-state) for 'state' is:
        idle :=  "10000000000";
        asdet := "01000000000";
        rasa :=  "00100000000";
        casa :=  "00010000000";
        w1  :=   "00001000000";
        w2  :=   "00000100000";
        w3  :=   "00000010000";
        nocas := "00000001000";
        refad := "00000000100";
        wr1  :=  "00000000010";
        wr2  :=  "00000000001";
```

The report file also shows that ads is indeed using HD2PAD resources:

```
-------------------------------------------------------------
Begin PAD Generation.
-------------------------------------------------------------
Created CLKPAD for signal 'clk'
    Above signal drives 70 Clocks,  0 Set/Resets. Total = 70
Created CLKPAD for signal 'rst'
    Above signal drives  0 Clocks, 70 Set/Resets. Total = 70
Created HD2PAD  for signal 'ads'
    Above signal drives  0 Clocks,  0 Set/Resets, 35 other
   inputs. Total = 35
topld:  ex4.vhd:  Note: (N1347) When using multiple high-
   drive pads, manual pin assignment is suggested

-------------------------------------------------------------
Begin Buffer Generation.
-------------------------------------------------------------
[max_load =  7, fanout = 18] Created 2 buffers [Duplicate]
   for 'MODULE_4_s0_g1_u0_c_1'
[max_load =  7, fanout = 11] Created 1 buffers [Duplicate]
   for 'MODULE_4_s0_g1_u0_c_2'
[max_load = 13, fanout = 15] Created 2 buffers [Normal   ]
   for 'stateSBV_1'
[max_load = 13, fanout = 28] Created 2 buffers [Normal   ]
   for 'ref_OUT'
```

The results of the second pass as compared to the first pass are shown in Table 9-3, with the best results for each category highlighted.

Table 9-3 Second pass FPGA results

	First Pass		Second Pass	
	L2 Area	L2 Speed	L2 Area	L2 Speed
Area (logic cells)	108	116	103	104
t_S (ns)	14.4	14.9	7.0	7.5
t_{SCS} (ns)	22.0	20.6	19.8	18.5
t_{CO} (ns)	39.4	32.1	29.8	26.2
limiting factor	t_{CO}	t_{CO}	t_{CO}	t_{CO}

These results illustrate that using directives judiciously can significantly improve the design. The setup time improvement came from using two high drive pads (of course, using two pads will increase the load external to the device and should be considered for the overall system design). The area savings and t_{SCS} and t_{CO} improvements came from using **one_hot_one** encoding.

9.2.1.3 Third Pass—Speed Optimization (Second Tuning Cycle)

The design has now exceeded the stated goals; however, continue to optimize the design to see if additional directives bring any further advantages. Using the path analyzer in SpDE to examine the delays shows that the worst case clock to output path comes from decoding the state bits through the multiplexer to the output of row_address. Highlighting these paths shows that these signals must route long distances to several loads. In an attempt to minimize this delay, use the **max_load** to buffer aggressively these signals. Since these signals are created by the synthesis process, use the control file to add directives. VHDL attributes cannot be used to apply these directives because these signals are not currently in the VHDL source code. The attributes may be added if the source code is modified. VHDL will not allow the user to apply an attribute to an object that does not exist. In the control file, apply the directives to the names of the synthesis created signals from the report file and in the path analyzer. The signals are clearly generated from the state vector. To ensure that all state bits will be buffered as appropriate, use the "*" wildcard to find all matches:

```
attribute max_load of statesbv* is 5;
attribute max_load of ref is 10;
```

The report file indicates proper buffering according to the control file:

```
----------------------------------------------------------
Begin Buffer Generation.
----------------------------------------------------------
[max_load =  7, fanout = 18] Created 2 buffers [Duplicate]
   for 'MODULE_5_s0_g1_u0_c_1'
[max_load =  7, fanout = 11] Created 1 buffers [Duplicate]
   for 'MODULE_5_s0_g1_u0_c_2'
Note:  Using config. rule 'statesbv*' to set attribute
   'max_load' on 'stateSBV_0_B0'.
Note:  Using config. rule 'statesbv*' to set attribute
   'max_load' on 'stateSBV_0_B1'.
[max_load =  5, fanout =  6] Created 2 buffers [Normal    ]
   for 'stateSBV_0'
Note:  Using config. rule 'statesbv*' to set attribute
   'max_load' on 'stateSBV_1_B0'.
Note:  Using config. rule 'statesbv*' to set attribute
   'max_load' on 'stateSBV_1_B1'.
Note:  Using config. rule 'statesbv*' to set attribute
   'max_load' on 'stateSBV_1_B2'.
[max_load =  5, fanout = 15] Created 3 buffers [Normal    ]
   for 'stateSBV_1'
[max_load = 10, fanout = 28] Created 3 buffers [Normal    ]
   for 'ref_OUT'
```

The results after place and route are only marginally better. The user is approaching the best implementation possible. To improve upon this implementation, the user could try adjusting the **max_load** to be slightly less aggressive. Being too aggressive may cause too many buffers to be inserted. The user may also iterate with the timing driven place and route tools by entering constraints via the path analyzer. See Chapter 5, "SpDE," to learn how to do this.

The summarized results of the speed optimization passes are shown in Table 9-4, highlighting the implementation that gave the best result in a given category. Because the limiting factor is t_{CO}, the results of L2 Area in the third pass work best. Next, optimize the design for area where the results may be surprising

Table 9-4 Third pass FPGA results

	First Pass		Second Pass		Third Pass	
	L2 Area	L2 Speed	L2 Area	L2 Speed	L2 Area	L2 Speed
Area (logic cells)	108	116	103	104	103	104
t_S (ns)	14.4	14.9	7.0	7.5	7.2	7.1
t_{SCS} (ns)	22.0	20.6	19.8	18.5	18.0	18.4
t_{CO} (ns)	39.4	32.1	29.8	26.2	26.0	26.6
limiting factor	t_{CO}	t_{CO}	t_{CO}	t_{CO}	t_{CO}	t_{CO}

9.2.1.4 Fourth Pass—Area Optimization

Up to this point, this example has assumed that the target device is a CY7C384A, a 2K gate device. Because only slightly more than half the resources of this device (from 104 to 116 of the 192 available logic cells) are being used, it may be worthwhile to optimize the design for area to see if it will fit in the CY7C382A, a 1K gate device with 96 logic cells. For this optimization, leave the **pad_gen** and **one_hot** attribute from the first tuning cycle, but remove the buffer generation of the subsequent cycle. In addition, change the synthesis goal in the Galaxy menu from speed to area, and resynthesize the design. The results are excellent (see Table 9-5)

Table 9-5 Fourth pass FPGA results

	First Pass		Second Pass		Third Pass		Fourth Pass	
	L2 Area	L2 Speed	L2 Area	L2 Speed	L2 Area	L2 Speed	L2 Area	L2 Speed
Area (logic cells)	108	116	103	104	103	104	91	92
t_S (ns)	14.4	14.9	7.0	7.5	7.2	7.1	5.8	5.5
t_{SCS} (ns)	22.0	20.6	19.8	18.5	18.0	18.4	22.2	22.2
t_{CO} (ns)	39.4	32.1	29.8	26.2	26.0	26.6	26.0	26.8
limiting factor	t_{CO}	t_{CO}	t_{CO}	t_{CO}	t_{CO}	t_{CO}	t_{CO}	t_{CO}

The design fits in the 1K gate device and achieves the original timing goals. In fact, they are superior in some respects to the ones that were achieved with the 2K gate device. This is not surprising because the 1K gate device is smaller. Thus, signals route smaller distances. At this point the user is finished—he has fit the design into the smallest FPGA while meeting the timing goals.

Next, the example will fit this design into a FLASH370 CPLD.

9.2.2 CPLD Optimization

9.2.2.1 First Pass

On this first pass, use the default synthesis and fitting options which yield the results summarized in Table 9-6. The "L2 Area" and "L2 Speed" columns have been removed because a fitter for CPLDs is used instead of the SpDE place and route tool.

Table 9-6 First pass CPLD results

First Pass (default)	
Macrocells	79
Product terms	225
t_S (ns)	6.0
t_{SCS} (ns)	10.0
t_{CO} (ns)	16.0
limiting factor	t_{CO}

This design requires a 128 macrocell member of the FLASH370 family of CPLDs. Not surprisingly, it has excellent speed. This is because the design is essentially a state machine, counters, and a little bit of combinational logic. This implementation has far superior performance over the FPGA, but it also requires a larger device. In the FPGA, however, if an additional pipeline were added to improve clock to output delays (the limiting factor in this design), then system speeds could approach 50 MHz. The additional pipeline would require a 2K device, resulting in different performance numbers.

A tuning cycle will not likely improve upon the speed and area of this CPLD implementation for two reasons: (1) The area versions of the counters will require just as many macrocells and product terms as the speed versions. This is because this counter is implemented very efficiently using T-type flip-flops. (2) Using the **state_encoding** attribute with the **one_hot_one** value will neither increase performance (it is already at its maximum—one pass through the logic array) nor reduce the number of required macrocells. In fact, a **one-hot** implementation will require more macrocells. It may reduce the number of product terms, but the current implementation uses only 35% of the available product terms. Gray encoding will require the same number of macrocells, but could possibly require fewer product terms. So, even though the current implementation is satisfactory, resynthesize and fit the design using the "gray" value for the **state_encoding** directive.

9.2.2.2 Second Pass -- State Machine Gray Encoding

This pass implements the **state_encoding** directive with a VHDL attribute placed in the architecture body declarative region where the **state** type is declared:

attribute state_encoding of states:type is gray;

Warp reports the following fitter error:

```
Error: Signal stateSBV_3 uses too many input
signals,(logic+OE+AR+AP).
```

This error indicates that one of the state bits requires more than the 36 inputs. The FLASH370 allows only 36 inputs into a given logic block (no other CPLD has more). Examine the report file to find the equation that verifies the veracity of this error message and to see what can be done to correct it. The equation is as follows:

```
/stateSBV_3.D =
          /stateSBV_0.Q * /stateSBV_3.Q * /stateSBV_2.Q *
          /count_2.Q * /count_1.Q * /count_0.Q * count_5.Q *
          /count_4.Q * count_3.Q * /count_8.Q * count_7.Q *
          /count_6.Q * count_11.Q * /count_10.Q * count_9.Q *
          /count_14.Q * count_13.Q * /count_12.Q *
          count_17.Q * /count_16.Q * count_15.Q *
          /count_20.Q * count_19.Q * /count_18.Q *
          count_23.Q * /count_22.Q * count_21.Q
        + /stateSBV_0.Q * stateSBV_3.Q * /stateSBV_2.Q *
          captured_address_31.Q
        + /stateSBV_0.Q * stateSBV_3.Q * /stateSBV_2.Q *
          captured_address_30.Q
        + /stateSBV_0.Q * stateSBV_3.Q * /stateSBV_2.Q *
          captured_address_29.Q
        + /stateSBV_0.Q * stateSBV_3.Q * /stateSBV_2.Q *
          captured_address_28.Q
        + /stateSBV_0.Q * stateSBV_3.Q * /stateSBV_2.Q *
          captured_address_27.Q
        + /stateSBV_0.Q * stateSBV_3.Q * /stateSBV_2.Q *
          captured_address_26.Q
        + /stateSBV_0.Q * /stateSBV_3.Q * /stateSBV_2.Q *
          /ads
        + /stateSBV_0.Q * stateSBV_3.Q * /stateSBV_2.Q *
          captured_address_25.Q
        + /stateSBV_0.Q * stateSBV_3.Q * /stateSBV_2.Q *
          captured_address_24.Q
        + /stateSBV_0.Q * stateSBV_1.Q * /stateSBV_2.Q
        + /stateSBV_1.Q * stateSBV_2.Q
        + stateSBV_0.Q * stateSBV_2.Q
```

Here, notice that the equation for this state-bit requires all inputs of the counter. This is due to the state transition out of the idle state when ref_req is asserted. In addition, notice that `captured_address` is required. This is due to the state transitions out of the `asdet` state when match is asserted. In the sequential encoding, the third bit of the state vector does not require all of these inputs: the capture_address inputs are used with a different state bit. To avoid this problem, this equation must be factored. A natural point to break this equation is with the captured_address signals or counter signals. The user can create a factoring point by applying the **synthesis_off** directive to either the match signal or the ref_req signal (or both). The next pass will show how to create one with the match signal. Creating this break-point will require a second pass through the logic array. This will result in additional delay and require additional resources. Obviously, the implementation will be inferior to the one achieved in the first pass. Nonetheless, this example will show how to work around this problem for instructional purposes. After all, it would be nice to know how to get around a problem like this one, if the user encountered it in the first pass.

It is interesting to note that the state encoding affected the number of terms in an equation.

9.2.2.3 Third Pass -- Synthesis_off

The **synthesis_off** directive is applied with a VHDL attribute, placed in the architecture declarative region where the match signal is declared:

```
attribute synthesis_off of match:signal is true;
```

The design fits. The report file indicates that gray encoding is used:

```
State variable 'state' is represented by a Bit_vector
   (0 to 3).
State encoding (gray) for 'state' is:
      casa :=  "0010";
      idle :=  "0000";
      asdet := "0001";
      rasa :=  "0011";
      casa :=  "0010";
      w1 :=    "0110";
      w2 :=    "0111";
      w3 :=    "0101";
      nocas := "0100";
      refad := "1100";
      wr1 :=   "1101";
      wr2 :=   "1111";
```

The equations also show that match was used as a factoring point:

```
/stateSBV_3.D =
         /stateSBV_0.Q * /stateSBV_3.Q * /stateSBV_2.Q *
         /count_2.Q * /count_1.Q * /count_0.Q * count_5.Q *
         /count_4.Q * count_3.Q * /count_8.Q * count_7.Q *
         /count_6.Q * count_11.Q * /count_10.Q * count_9.Q *
         /count_14.Q * count_13.Q * /count_12.Q *
         count_17.Q * /count_16.Q * count_15.Q *
         /count_20.Q * count_19.Q * /count_18.Q *
         count_23.Q * /count_22.Q * count_21.Q
        + /stateSBV_0.Q * stateSBV_3.Q * /stateSBV_2.Q *
          /match.CMB
        + /stateSBV_0.Q * /stateSBV_3.Q * /stateSBV_2.Q *
          /ads
        + /stateSBV_0.Q * stateSBV_1.Q * /stateSBV_2.Q
        + /stateSBV_1.Q * stateSBV_2.Q
        + stateSBV_0.Q * stateSBV_2.Q

    match =
          /captured_address_31.Q * /captured_address_30.Q *
          /captured_address_29.Q * /captured_address_27.Q *
          /captured_address_26.Q * /captured_address_25.Q *
          /captured_address_24.Q * /captured_address_28.Q
```

The area and speed results are summarized in Table 9-7. This implementation requires fewer product terms, but has slower performance than the first implementation.

Table 9-7 Third pass CPLD results

	First Pass (Defaults)	Second Pass (Gray Encode)	Third Pass (Synthesis_off)
Macrocells	79	Fit Error	80
Product terms	225	Fit Error	202
t_S (ns)	6.0	Fit Error	6.0
t_{SCS} (ns)	10.0	Fit Error	19.0
t_{CO} (ns)	16.0	Fit Error	16.0
limiting factor	t_{CO}	Fit Error	t_{SCS}

Either of the successful CPLD implementations provide superior speed over the FPGA implementation, but they also require a larger density device. The next example selects a design that favors FPGAs for speed and area.

9.3 Example 2—Multiply and Accumulate Function

The code of the following listing is a multiply and accumulate design. Once again, this design will be optimized first for a pASIC380 FPGA, then for a FLASH370 CPLD.

```
library ieee;
use ieee.std_logic_1164.all;

entity math is port (
   clk, rst, mac:std_logic;
   a, b:in std_logic_vector(7 downto 0);
   q: buffer std_logic_vector(15 downto 0));
end math;

use work.std_arith.all;
architecture math of math is
begin
p1: process (rst, clk)
   begin
      if rst = ‘1’ then
         q <= (others => ‘0’);
      elsif clk’event and clk=’1’ then
         q <= (a * b) + q;
      end if;
   end process;
end math;
```

9.3.1 FPGA Optimization

9.3.1.1 First Pass -- Default Options

In the first pass through the design, use the default Galaxy options—buffer generation on, pad generation on, and speed optimization for inferred arithmetic components. In the report file, the two arithmetic operators are inferred:

```
g.vhd (line 18, col 16):  Note: Substituting module
   'warp_mul_2s_ss' for '*'.
g.vhd (line 18, col 21):  Note: Substituting module
   'warp_add_2s_ss' for '+'.
```

The signals `clk` and `rst` were placed on clock pads. Some of the inputs were also selected for high drive pads because of high internal fanout. The default value for the `max_load` directive is 13, so the following buffers were inserted:

```
----------------------------------------------------------
Begin PAD Generation.
----------------------------------------------------------
Created CLKPAD for signal 'clk'
   Above signal drives 16 Clocks,  0 Set/Resets. Total = 16
Created CLKPAD for signal 'rst'
   Above signal drives  0 Clocks, 16 Set/Resets. Total = 16
Created HD1PAD  for signal 'b_1'
   Above signal drives  0 Clocks,  0 Set/Resets, 16 other
   inputs. Total = 16
Created HD1PAD  for signal 'a_1'
   Above signal drives  0 Clocks,  0 Set/Resets, 16 other
   inputs. Total = 16
Created HD1PAD  for signal 'a_2'
   Above signal drives  0 Clocks,  0 Set/Resets, 16 other
   inputs. Total = 16
Created HD1PAD  for signal 'a_3'
   Above signal drives  0 Clocks,  0 Set/Resets, 16 other
   inputs. Total = 16
```

```
----------------------------------------------------------------
Begin Buffer Generation.
----------------------------------------------------------------
[max_load = 13, fanout = 15] Created 2 buffers [Normal    ]
   for 'a_0_IN'
[max_load = 13, fanout = 15] Created 2 buffers [Normal    ]
   for 'b_0_IN'
[max_load = 13, fanout = 16] Created 2 buffers [Normal    ]
   for 'a_4_IN'
[max_load = 13, fanout = 16] Created 2 buffers [Normal    ]
   for 'a_5_IN'
[max_load = 13, fanout = 16] Created 2 buffers [Normal    ]
   for 'a_6_IN'
[max_load = 13, fanout = 16] Created 2 buffers [Normal    ]
   for 'a_7_IN'
[max_load = 13, fanout = 16] Created 2 buffers [Normal    ]
   for 'b_2_IN'
[max_load = 13, fanout = 16] Created 2 buffers [Normal    ]
   for 'b_3_IN'
[max_load = 13, fanout = 16] Created 2 buffers [Normal    ]
   for 'b_4_IN'
[max_load = 13, fanout = 16] Created 2 buffers [Normal    ]
   for 'b_5_IN'
[max_load = 13, fanout = 16] Created 2 buffers [Normal    ]
   for 'b_6_IN'
[max_load = 13, fanout = 16] Created 2 buffers [Normal    ]
   for 'b_7_IN'
```

The area and speed results are listed in Table 9-8. This time, setup times are the limiting factor. That is, the design cannot be clocked at t_{scs} because that would violate setup times.

Table 9-8 First pass FPGA results

	L2 Area	L2 Speed
Area	163	186
t_S (ns)	68.8	73.9
t_{SCS} (ns)	22.8	24.9
t_{CO} (ns)	11.3	10.4
limiting factor	t_S	t_S

If the user wanted to pursue a faster design, then he could experiment with the **max_load** directive as well as the SpDE timing driven place and route to minimize the delays of particular paths. Going through this process may yield a small improvement over the current speed numbers.

9.3.1.2 Second Pass—Area Optimization

Because the default options use speed optimization, rerun the design with area optimization on. After synthesis and place and route, the results are exactly the same as in the first pass. This is because the adder and multiplier modules for the widths used in this design have the same implementation for both speed and area optimized versions.

Next, this design will be implemented in a FLASH370 CPLD.

9.3.2 CPLD Optimization

9.3.2.1 First Pass -- Default Options

Once again, the first pass will use the default Galaxy options. This means speed optimization. With these options, the design will not fit (Table 9-9) because it requires too many macrocells. It also requires nearly all of the available product terms. So, pursue area optimization.

Table 9-9 First pass CPLD results

	First Pass (Defaults)
Macrocells	132
Product terms	620
t_S (ns)	N/A
t_{SCS} (ns)	N/A
t_{CO} (ns)	N/A
limiting factor	did not fit

9.3.2.2 Second Pass -- Area Optimization

The results of area optimization are summarized in Table 9-10:

Table 9-10 Second pass CPLD results

	First Pass (Defaults)	Second Pass (Area)
Macrocells	132	120
Product terms	620	605
t_S (ns)	N/A	87.0
t_{SCS} (ns)	N/A	66.0
t_{CO} (ns)	N/A	7.0
limiting factor	did not fit	t_S

The setup time for this combination of operations—multiply and accumulate—is the limiting factor for the maximum frequency of this design.

For this application, the FPGA performed better and required fewer resources. This is not surprising. FPGAs often do well in datapath and register-intensive applications.

Now each of the directives listed in Table 9-10 will be covered topically by the categories listed in the columns—area optimization, speed optimization, specific control, documentation/selection.

9.4 Area Optimization

This section describes the directives and techniques required to successfully implement a logic design with the minimum device resources (minimum area) being utilized. The techniques are often different for FPGA and CPLD architectures. The focus of this section is to provide recommended techniques for area optimization based on device architecture.

9.4.1 CPLD and FPGA Considerations

This section discusses area optimization methods relating to all Cypress programmable devices. The **`goal`**, **`synthesis_off`**, and **`no_factor`** directives are discussed.

9.4.1.1 The GOAL Directive

```
attribute goal of architecture_name : architecture is area;
```

or command line option: **-yga**

The goal value of **area** indicates that all modules inferred from VHDL operators will be optimized for area. The *Warp* synthesizer will select an implementation that is optimized to use the minimum device resources. A 16-bit adder example with the goal directive placed on an architecture is shown below. This code will generate a ripple carry adder with a 2-bit group as the basic unit. This adder would be implemented as carry-look-ahead if the goal was set to **speed.** A comparison of the results after compilation for each goal and target device type is shown in Table 9-11.

Table 9-11 Results of GOAL directives

FPGA		CPLD	
Area Opt.	Speed Opt.	Area Opt.	Speed Opt.
24 logic cells	45 logic cells	23 macrocells	35 macrocells
8 passes	4 passes	8 passes	3 passes

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity add16_a is port(
   a, b:in std_logic_vector (15 downto 0);
   sum:out std_logic_vector (15 downto 0));
end add16_a;

architecture archadd16_a of add16_a is
ATTRIBUTE goal OF archadd16_a : ARCHITECTURE IS area;
begin
   sum <= a + b;
end;
```

9.4.1.2 The SYNTHESIS_OFF Directive

```
ATTRIBUTE synthesis_off OF signal_name : signal IS true;
```

When the **synthesis_off** directive is set to **true**, a signal is made into a factoring point for logic equations. This directive keeps the signal from being substituted out during the optimization process. The node number is used to reference a macrocell within a CPLD.

Synthesis_off is useful for the following reasons:

- It gives the user control over which equations or sub-expressions need to be factored into a node.
- It provides better results for designs where a signal with a large functionality is being used by many other signals. If left alone, the fitter would collapse all the internal signals (which is desirable in many cases) and may drive the design's resource requirements beyond the available limits.
- It helps in cutting down on compile time for designs which have a lot of "signal redirection" (signals getting inverted or reassigned to other signals). This directive provides the logic optimizer a better control over the optimization process, by reducing the number of signals it needs to deal with.

By using the **synthesis_off** directive, the user can assign the commonly used signal to a node and bring down the resource utilization.

A side effect of using the **synthesis_off** directive is that the design will now take an extra pass through the array to achieve the same functionality. The extra pass may be required anyway, if more than 16 PTs are required.

This directive is recommended only on combinatorial signals. Registered signals are assigned to a node by natural factoring, and the **synthesis_off** directive on these signals is redundant.

This directive can be associated with signals declared both in VHDL and schematics. The BUF component can also be used in schematics and VHDL to achieve the same results as the **synthesis_off** directive. Please refer to the *Warp* Synthesis manual for more details.

This directive allows the designer to force multiple passes through logic cells for optimal density. The following example uses the **synthesis_off** directive and uses 30 Macrocells in a CY7C371. This same design requires 43 Macrocells in a CY7C371 without using the **synthesis_off** directive:

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity cpldadd is port(
   a: in std_logic_vector(7 downto 0);
   b: in std_logic_vector(7 downto 0);
   c: in std_logic_vector(7 downto 0);
   sum: out std_logic_vector(7 downto 0));
end cpldadd;

architecture areacpldadd of cpldadd is
   signal intsum: std_logic_vector(7 downto 0);
   attribute synthesis_off of intsum:signal is true;
begin

   intsum <= a + b;
   sum <= intsum + c;

end areacpldadd;
```

9.4.1.3 The NO_FACTOR Directive

```
ATTRIBUTE no_factor OF signal_name : signal IS false;
```

The **no_factor** directive when set to **true** prevents logic factoring within the *Warp* synthesis engine. This means that factors which can be shared among multiple outputs are not generated. For area optimization, the **no_factor** directive should always be set to **false**. This allows the synthesizer to create common logic that can be shared, thus reducing the resources required.

9.4.2 FPGA Considerations

This section discusses area optimization methods using the Cypress pASIC380 FPGA architecture. The **state_encoding** and **buffer_gen** directives are discussed.

9.4.2.1 The STATE_ENCODING Directive

The **state_encoding** directive specifies the internal encoding scheme for values of an enumerated type. For most state machine designs larger than 4 states, the values recommended for area optimization in FPGAs are either **one_hot_one** or **one_hot_zero**. The choice of which type to use depends on the design being implemented. A detailed description of each encoding value is provided below. Further information on state encoding schemes may be found in the VHDL textbook accompanying this document set.

```
ATTRIBUTE state_encoding OF type_name : type IS
one_hot_zero;
```

When the **state_encoding** value is set to **one_hot_zero**, the encoding of the first value in the type definition is set to 0. Each succeeding value in the type definition has its own bit position (flip-flop) in the encoding sequence. Thus, a **one_hot_zero** encoding of an enumerated type with N possible values requires N-1 bits (flip-flops). The following VHDL code will generate a **one_hot_zero** state machine design:

```
-- This state machine implements a simple traffic light.
-- The N - S light is usually green, and remains green
-- for a minimum of five clocks after being red. If a
-- car is travelling E-W, the E-W light turns green for
-- only one clock.
PACKAGE DesgnPkg IS
TYPE state IS (green_red, yellow_red, red_green,
            red_yellow);
ATTRIBUTE state_encoding OF state: type IS one_hot_zero;
END DesgnPkg;

library ieee;
use ieee.std_logic_1164.all;
use work.desgnpkg.all;

ENTITY traffic_light IS
   PORT (clk, car: IN STD_LOGIC;--E-W travelling car
      lights: BUFFER state);
END traffic_light;

ARCHITECTURE moore1 OF traffic_light IS
-- The lights (outputs) are encoded in the following
-- states. For example, the state green_red indicates
-- the N-S light is green and the E-W light is red.
-- nscount is used to verify five consecutive N-S greens
```

```
   SIGNAL nscount: INTEGER RANGE 0 TO 5;
BEGIN
   PROCESS
   BEGIN
      WAIT UNTIL clk = '1';
      CASE lights IS
         WHEN green_red =>
            IF nscount < 5 THEN
               lights <= green_red;
               nscount <= nscount + 1;
            ELSIF car = '1' THEN
               lights <= yellow_red;
               nscount <= 0;
            ELSE
               lights <= green_red;
            END IF;
         WHEN yellow_red =>      lights <= red_green;
         WHEN red_green =>
            lights <= red_yellow;
         WHEN red_yellow =>      lights <= green_red;
         WHEN others =>          lights <= green_red;
      END CASE;
   END PROCESS;
END moore1;
```

The resulting state bit assignments indicated in the report file *file-name.rpt* for the **one_hot_zero** design is shown below:

```
State variable 'lights' is represented by a Bit_vector
   (0 to 3).
State encoding (one-hot zero-state) for 'lights' is:
   green_red   :="000";
   yellow_red  :="100";
   red_green   :="010";
   red_yellow  :="001";
```

ATTRIBUTE state_encoding OF *type_name* : type IS one_hot_one;

One_hot_one state encoding is similar to **one_hot_zero**, except that zero encoding is not used. Every state value has a bit position that is set to "1" when the state variable is active. Thus, a **one_hot_one** encoding of a state machine with N possible values requires N bits (flip-flops).

If the package statement used for the **one_hot_zero** street light example above is changed to use **one_hot_one**, then the package declaration would appear as shown:

```
PACKAGE DesgnPkg IS
   TYPE state IS (green_red, yellow_red, red_green,
                  red_yellow);
   ATTRIBUTE state_encoding OF state: type IS
                 one_hot_zero;
END DesgnPkg;
```

The resulting state bit assignments indicated in the report file *vhdl_file.rpt* for the **one_hot_one** is shown below:

```
State variable 'lights' is represented by a Bit_vector
   (0 to 3).
State encoding (one-hot one-state) for 'lights' is:
   green_red  :="1000";
   yellow_red :="0100";
   red_green  :="0010";
   red_yellow :="0001";
```

9.4.2.2 Comparing one_hot_zero to one_hot_one

Notice that the **one_hot_one** design uses an extra flip-flop for the first state assignment compared to the **one_hot_zero** implementation.

This traffic light example takes six logic cells when implemented in a CY7C381A using **one_hot_zero** and seven logic cells when using **one_hot_one**. This ratio may not always be the same, depending on other specifics on how the design is implemented. A **one_hot_zero** implementation is useful in situations where only a reset is available to the registers in a PLD. If an asynchronous signal is required for initialization of the registers in a 22v10, then a reset is the only option. In this case, use a **one_hot_zero** state machine for initialization to state zero. The **one_hot_zero** may be less optimum if the Idle state (all 0s) is required to decode an output signal or multiple transitions to different states. In this case, all the state bits would have to be decoded to verify that the machine is in Idle.

In general, **one-hot** designs are faster than binary encoded because state transition decodings are simple. For FPGA architectures, a flip-flop is a less vital resource than product term inputs because of the finer grain logic implementation with a flip-flop in every cell. For these reasons it makes sense to use the one-hot technique for FPGA state machines. This is not necessarily true for CPLD state machine designs.

9.4.2.3 The BUFFER_GEN Directive

ATTRIBUTE buffer_gen OF *signal_name* : signal IS buf_none;

or command line option: **-yb**

The **buffer_gen** directive controls the buffering strategy for signals that have a high fanout (exceeding **max_load**). If a signal has a high fanout, then signal propagation delays increase significantly. The **buffer_gen** value is by default **buf_auto**. The **buf_none** value is preferred for least resources being used. Buffer generation should only be used where speed is of concern. Refer to Section 9.6.2, "Speed Optimization for FPGAs" for further details on **buffer_gen**.

9.4.3 CPLD Considerations

This section discusses area optimization methods using all Cypress PLD and CPLD architectures. The **ff_type** directive applies to area optimization in CPLDs.

9.4.3.1 The FF_TYPE Directive

ATTRIBUTE ff_type OF *signal_name* : signal IS ff_opt;

or command line option: **-fo**

The **ff_type** value of **ff_opt** tells *Warp* to synthesize the *signal_name* to the optimum flip-flop type for the logic implemented. A flip-flop is chosen based on the fewest resources required to implement the logic function. For instance, a D-type flip-flop may be chosen for register data storage functions, while a T-type (toggle) flip-flop may be chosen for counters. This option is recommended for all designs unless the designer has specific requirements to force the use of a different flip-flop.

9.5 Specific Control

This section describes specific control features of the *Warp* synthesis tool.

9.5.1 The FF_TYPE Directive (CPLD Only)

ATTRIBUTE ff_type OF *signal_name* : signal IS ff_d;

or command line option: **-fd**

The **ff_type** value of **ff_d** tells *Warp* to synthesize the *signal_name* using a D-type flip-flop. This will force the synthesizer to use a D-type flip-flop to generate *signal_name.* This directive will typically only be used if the *Warp* synthesis tool is not using the D-type flip-flop where the designer intends.

ATTRIBUTE ff_type OF *signal_name* : signal IS ff_t;

or command line option: **-ft**

The **ff_type** value of **ff_t** tells *Warp* to synthesize the **signal_name** using a T-type flip-flop. This will force the synthesizer to use a toggle flip-flop to generate **signal_name**. This directive will typically only be used if the *Warp* synthesis tool is not using a toggle flip-flop, which the designer intends for functional reasons.

9.5.2 The FIXED_FF Directive (FPGA Only)

```
ATTRIBUTE fixed_ff of signal_name : signal is
   register_location
```

or command line option: **-f*n*** [n=register location]

The **fixed_ff** directive locks a flip-flop to a specific cell location in the device. This directive overrides the default placement that the SpDE placer assigns automatically. The **fixed_ff** directive is applied to the Q output signal of a flip-flop. If the **fixed_ff** directive is assigned to any other signal besides the Q output of a flip-flop, the directive is ignored. An example follows:

```
library ieee;
use ieee.std_logic_1164.all;

ENTITY ff_type_test IS
PORT (clk, ff_D: IN STD_LOGIC;    -- Flip-flop clock, D-input
        ff_Q: OUT STD_LOGIC);  -- Flip-flop Q output
   ATTRIBUTE fixed_ff OF ff_Q:signal IS "A1";
END ff_type_test;

ARCHITECTURE arch_ff_type_test OF ff_type_test IS
```

```
BEGIN
   PROCESS
   BEGIN
      WAIT UNTIL clk = '1';
         ff_Q <= ff_D; -- Generate output
   END PROCESS;
END arch_ff_type_test;
```

The above code segment will ensure the signal `ff_Q` is generated from the flip-flop located in cell "A1" of a pASIC device. This allows the designer to manually place flip-flops to override the SpDE floor planner. This directive is used to place flip-flops in close proximity in order to reduce routing lengths for critical path signals. Flip-flops may be grouped together to provide maximum operating speed. Refer to Section 9.6.2, "Speed Optimization for FPGAs," for further details on optimizing a design for speed using the **fixed_ff** directive.

9.5.3 The NODE_NUM Directive (PLD & CPLD Only)

ATTRIBUTE node_num OF *signal_name* : signal IS integer ;

or command line option: **-fn** [n=node location]

The **node_num** directive locks a signal to a specific location in the target device. This directive overrides the default placement that the *Warp* tool would assign automatically. This directive applies to any combinatorial or sequential node within the design.

Example:

```
library ieee;
use ieee.std_logic_1164.all;

ENTITY node_num_test IS
PORT (clk, ff_D: IN STD_LOGIC;    -- Flip-flop clock, D-input
        ff_Q: OUT STD_LOGIC);  -- Flip-flop Q output
   ATTRIBUTE node_num OF ff_Q:SIGNAL IS 398;
END node_num_test;

ARCHITECTURE arch_node_num_test OF node_num_test IS
BEGIN
   PROCESS
   BEGIN
      WAIT UNTIL clk = '1';
         ff_Q <= ff_D; -- Generate output
   END PROCESS;
END arch_node_num_test;
```

The above code segment will ensure the signal `ff_Q` is generated from the macrocell driving node 398 in a CY7C374 device. Node 398 refers to buried macrocell A in logic block #1 in a CY7C374. The specific node numbers available for each FLASH370 series device may be found in the *Reference Manual* accompanying this document set. This directive allows the designer to manually place logic to override the *Warp* floor planner.

9.5.4 The LAB_FORCE Directive (CPLD Only)

```
ATTRIBUTE lab_force OF signal_name : signal IS "string";
```

The **`lab_force`** directive aids in grouping signals together as a requirement to the fitter. The **`string`** contains the name of the logic block. This directive will force `signal_name` to the **`string`** internal logic block without regard for I/O pin assignments. In most designs, the automatic assignment by the fitter is acceptable. In some cases, the user may want to constrain the fitter to obtain better partitioning than can be performed automatically. This directive should only be used if the user is intimately familiar with the target CPLD architecture. This directive can cause routing difficulties if logic is placed in an area that can block routing paths.

Example:

```
ATTRIBUTE lab_force OF ff_Q:SIGNAL IS "B2";
```

This will force the signal `ff_Q` to the lower half of logic block B in a FLASH370 device. In the following example:

```
ATTRIBUTE lab_force OF ff_Q:signal IS "B1";
```

The signal `ff_Q` is forced to the upper half of logic block B.

9.5.5 The SUM_SPLIT Directive (CPLD Only)

ATTRIBUTE sum_split OF *signal_name* : signal IS *value;*

The **sum_split** value can be **balanced** or **cascaded.** The default value is **balanced**. Use the **balanced** value if reliable balanced timing is desired at the expense of area. The following figure describes the balanced sum split concept:

ATTRIBUTE sum_split OF sum_signal:signal IS balanced;

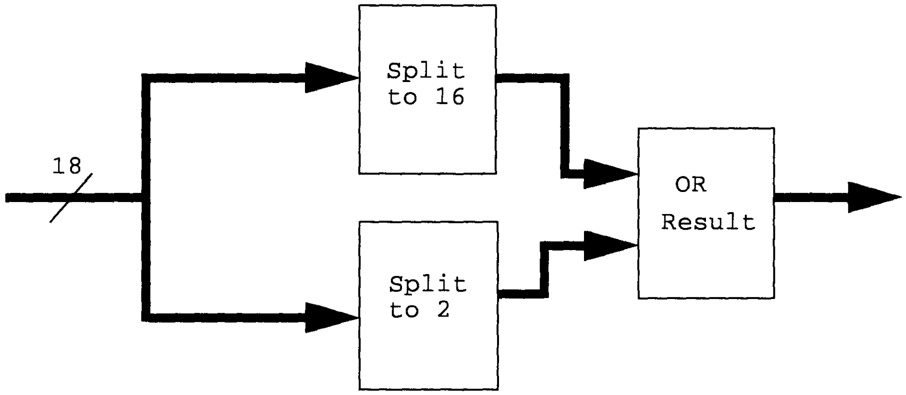


Figure 9-2 The balanced sum split concept

The **cascaded** method uses only two macrocells to implement an equation. There is no control over which product term is assigned to which macrocell. The signals that are not split into macrocell #1 will arrive at macrocell #2 sooner, thereby making the timing for the outputs different based on different arrival times. If these output signals are registered, then of course the timing generated at the outputs are the same.

ATTRIBUTE sum_split OF sum_signal:signal IS cascaded;

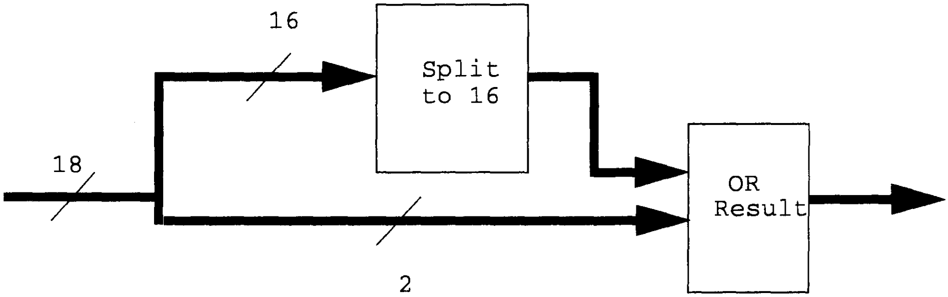


Figure 9-3 The cascaded sum split

Which **sum_split** method to use depends on the area constraints and how the design is implemented. Use the balanced method first and then the cascaded, if the design did not fit using balanced.

9.5.6 The POLARITY Directive (CPLD Only)

```
ATTRIBUTE polarity OF signal_name : signal IS value;
```

The **polarity** directive is used to select polarity for signals in a design. There are two options for polarity, **pl_keep** and **pl_opt.** The **pl_keep** option will instruct the *Warp* compiler to keep the polarity of a signal as currently specified in the design. The **pl_keep** option is useful to instruct the compiler about the desirable output sense of a signal at power up. When a circuit is initialized, it may be desirable to provide an output as a "1" or "0" and maintain this condition without the compiler changing the sense for optimization reasons. In another case, it may be desirable to keep signal senses in order to debug designs in the simulator without being concerned about compiler-induced internal inversions. In most cases, however, the **pl_opt** is the best choice. This option allows the compiler to change the sense of internal signals to provide the best optimization for a design.

9.6 Speed Optimization

This section describes the synthesis directives and techniques that may be used in optimizing a design for performance. In most cases, the techniques for speed optimization are device dependent. The discussion will cover first those directives applicable to both FPGAs and CPLDs, then those for FPGAs only.

9.6.1 Speed Optimization for both FPGAs and CPLDs

9.6.1.1 The GOAL Directive

```
ATTRIBUTE goal OF architecture_name: architecture IS speed;
```

The **goal** value of **speed** indicates that all arithmetic modules inferred from VHDL operators will be optimized for speed. The *Warp* synthesizer will select an implementation that is optimized to achieve the best performance. This is a good first step to take when optimizing a design for performance. To demonstrate the goal directive, observe the performance delta in the following 8-bit adder example implemented in a FLASH370 CPLD:

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity add8_a is port(
   a, b: in std_logic_vector (7 downto 0);
   sum: out std_logic_vector (7 downto 0));
end add8_a;

architecture archadd8_a of add8_a is
attribute goal of archadd8_a: architecture is speed;
begin
   sum <= a + b;
end;
```

Results with **goal** set to **area** was 57.0 ns (17.5 Mhz) worst case delay.

Results with **goal** set to **speed** was 27.0 ns (37 Mhz) worst case delay.

9.6.1.2 The DONT_TOUCH Directive

```
ATTRIBUTE dont_touch OF label_name: label IS true;
ATTRIBUTE dont_touch OF entity_name: entity IS true;
```

In some rare cases, a block of a design may need to be hand-optimized. The user may instruct *Warp* to leave the individually optimized block alone by applying the **dont_touch** directive to the entity or the component to prevent any optimization on the block. Under most circumstances, this directive is not needed since *Warp*'s optimization usually improves performance and resource efficiency.

```
architecture arch_accumulator of accumulator is
   ...
attribute dont_touch of block1: label is true;

begin
block1: add4 (a, b, sum);
```

OR

```
entity my_adder8 is port (
   a,b: in std_logic_vector(0 to 7);
   ...
   );
   attribute dont_touch of my_adder8: entity is true;
end entity my_adder8;
```

9.6.2 Speed Optimization for FPGAs

9.6.2.1 The BUFFER_GEN and the MAX_LOAD Directives

```
ATTRIBUTE buffer_gen OF signal_name: signal IS value;
ATTRIBUTE max_load OF signal_name: signal IS integer;
```

Buffering a signal with high fanout effectively reduces the load seen by a signal, and is used to reduce the propagation delay of that signal. *Warp* is capable of implementing several methods of buffering. By default, *Warp* attempts automatic buffering (**buf_auto**, explained below). If a different buffering technique is desired for a particular signal, or register duplication is required, then the **buffer_gen** directive may be applied to that signal. To specify a limit on the number of loads a signal should have, the **max_load** directive may be used in conjunction with the **buffer_gen** directive. When *Warp* encounters a signal with a fanout count larger than the specified **max_load** value, it buffers the signal. *Warp* has a default maximum load setting of 13.

Buffer generation options are:

buf_none : When the **buf_gen** directive is set to this value, *Warp* will not buffer this signal. It prevents resources from being used unnecessarily as buffers. This value should be used for signals which are not timing critical.

buf_auto : This is the default setting *Warp* uses for buffer generation. With this setting, *Warp* first attempts **buf_duplicate**, then attempts **buf_normal**. The **buf_register** will not be attempted.

buf_normal : A buffer tree is created between the signal source and its loads until every node has a fanout of less than the maximum load count as specified by the max_load directive. This technique is best used for signals that have very high fanout (greater than 24) and need to meet a maximum propagation delay.

buf_duplicate : The logic gate that produced the source signal is duplicated multiple times. This "paralleling" of signal sources does not create additional levels of logic but does increase the load at the source inputs. For fanout loads of less than 24, duplicate buffering will usually yield better performance than normal buffering. The source logic must fit into a pASIC primitive PAfragA, PAfragF, logico or the like. This method increases the load at the source inputs.

buf_register : Similar to duplicate buffering, registers are duplicated in parallel. This method does not create additional logic levels and works best for synchronous designs. For registered signals, this method usually yields better performance than normal buffering. This method increases the load at the register input.

Example:

```
library ieee;
use ieee.std_logic_1164.all;

entity ld_reg is port(
   d: in std_logic_vector (31 downto 0);
   address: in std_logic_vector (3 downto 0);
   q: inout std_logic_vector (31 downto 0);
   clk: in std_logic);
end ld_reg;

architecture arch_ld_reg of ld_reg is
signal reg_en: std_logic;
attribute buffer_gen of reg_en: signal is buf_normal;
attribute max_load of reg_en: signal is 8;
```

```
begin
main: process(clk)
   begin
   if (clk'event and clk = '1') then
      if (reg_en = '1') then
         q <= d;
      else
         q <= q;
      end if;
   end if;
end process;

reg_en <= '1' when (address = "1001") else '0';

end arch_ld_reg;
```

Without buffering of any kind (automatic buffer generation disabled), the reg_en signal has a fanout of 32. When importing into SpDE, the tool will warn that the reg_en signal has a high fanout. SpDE's path analyzer reveals that the worst case delay for a CY7C384A-2JC is 33.5 ns (~30 Mhz). To improve the performance, the VHDL file may be recompiled with buffer generation enabled in the device window of Galaxy (default), and **max_load** directive placed on the reg_en signal. With the **max_load** set to 8, the worst case delay is brought down to 18.2 ns (~55 Mhz). As a guideline, **max_load** should generally be set in the range of 5 to 13. Above 13 loads, the delay of a signal is mostly due to the number of loads and their associative routing. Between 5 and 13 loads, the tPD of the added buffer with its associating routing may begin to balance out the fanout delay. Below 5 loads, the buffering delay begins to outweigh the savings from load reduction. For example, when **max_load** is set to 4, the worst case delay is 19.1 ns (52 Mhz), worse than when **max_load** is 8. It should be remembered that delays in an FPGA are design-dependent and place-and-route dependent. This means that for the same **max_load** setting, different designs and place-and-route iterations will have different performances, hence the recommended range of 5 to 13 loads.

If the reg_en signal is a registered signal, as in the code below, then the **buf_register** setting should be used with the **buffer_gen** directive. In register buffering (**max_load** = 8), the register source is repeated. In this case, the register is automatically repeated four times to bring the worst case delay down to 15.6 ns (~64 Mhz).

```
architecture arch_ld_reg of ld_reg is
signal reg_en: std_logic;
attribute buffer_gen of reg_en: signal is buf_register;
attribute max_load of reg_en: signal is 8;

begin
main: process(clk)
   begin
   if (clk'event and clk = '1') then
      if (address = "1001") then
         reg_en <= '1';
      else
         reg_en <= '0';
      end if;

      if (reg_en = '1') then
         q <= d;
      else
         q <= q;
      end if;
   end if;
end process;

end arch_ld_reg;
```

9.6.2.2 The PAD_GEN Directive

attribute pad_gen OF *signal_name*: signal IS *value*;

The pASIC380 family has three different types of pins. Bidirectional pins may be configured as bidirectionals, inputs only, and three statable outputs. There are also dedicated inputs and clock input pins. The dedicated inputs are high drive inputs for use with signals with high internal fanout. Clock inputs utilize an internal clock distribution tree to achieve low skew. (Clock inputs can also double as high drive inputs.) The type of input can be specified by using the **pad_gen** directive.

```
entity counter is port (
   clock: in std_logic;
   ...
   );
   attribute pad_gen of clock: signal is pad_clock;
end entity counter;
```

Automatic : *Warp* defaults to this setting. This setting attempts to find the type of pad that best suits the implementation of the signal (bidirectional I/O, clock, or highdrive). This setting is activated when the automatic pad generation is enabled in the device window of Galaxy, and the **pad_gen** directive is set to **pad_auto** or no directive for **pad_gen** exists.

Bidirectional I/O : The majority of input signals and all output signals use a bidirectional I/O pin. These pins can be configured as always active outputs, three-state outputs, inputs, and bidirectionals. To indicate to *Warp* that a signal should utilize an I/O pin, the above directive may be used with the value set to **pad_none** or **pad_io**.

Dedicated High-Drive Input : When an input signal drives many internal logic gates (on the order of 8 or more loads), a dedicated high drive input can be used to reduce propagation delays. The high drive inputs have double the drive capability of a regular I/O input driver. Because they are intended for multiple loads in mind, high drive inputs require the use of express wires for routing. Express wires are routing resources that traverse the entire length of the device. For very large fanout counts, multiple high drive input drivers may have their outputs tied together. This requires that the input signal at the pins is the same. To have *Warp* utilize the dedicated inputs, use the above directive with the value set to: **pad_hd1**, **pad_hd2**, **pad_hd3**, or **pad_hd4**.

Clock Input : To maintain a chip-wide skew of less than one nanosecond, the clock distribution tree limits clock input signals to being wired to the reset, preset and clock inputs of each logic cell's register. To utilize the clock inputs, use the above directive with the value set to **pad_clock**.

9.6.2.3 The FIXED_FF directive

```
ATTRIBUTE fixed_ff OF signal_name: signal IS
          register_location;
```

Hand placement of logic cells within the device is generally not recommended, since an unrestricted place and route tool will be able to move logic cells near each other when necessary during placement to reduce delays and routing utilization. For cases where the user needs strict control over logic cell placement, however, hand placement of logic cells is possible using the directive **fixed_ff** on the registered signal. The signal being fixed must be a signal on the Q output of a flip-flop or logic cell. The two most common situations which potentially benefit from assigning logic cell placement are discussed below.

Logic cell placement in a column arrangement is useful when used in conjunction with high drive inputs (dedicated inputs which can drive larger loads than the standard I/O). High drive inputs require the use of vertical routing lines that span the entire height of the device for small devices (express wires) or four logic cells for larger devices (quad wires). Because of this, arranging logic cells in a single column will require the use of only one express wire or a minimum number of quad wires, thus saving resources as well as decreasing the propagation delay. Fifo and shift register applications often will have this type of situation; however, it is recommended that this be the last step in optimizing a design for performance.

Logic cell placement can also aid in minimizing register to pin delays. The *Warp* development system usually attempts to place the source logic cell near the output pin. To insure that critical output pin signals have minimal clock to valid times, however, the **fixed_ff** directive may be used to lock the logic cell near the output pin.

An example, *counter4.vhd,* is shown below (a 4-bit counter with enable). It is desired that the registers for the vector data be placed in a column. Since VHDL does not allow directives to be placed on individual signals of a vector, *Warp*'s control file is used.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity counter4 is port(
   data:  inout std_logic_vector (3 downto 0);
   clk,rst: in std_logic );
end counter4;

architecture arch_counter4 of counter4 is
begin
process (clk,rst)
   begin
   if (rst='1') then
      data <= (others => '0');
   elsif (clk'event and clk='1') then
      data <= data + 1;
   end if;
end process;

end arch_counter4;
```

After compilation, it is noted that the data signal vector has been broken down into individual signals with labels data_0, data_1, etc. A control file is made by creating a new file called *counter4.ctl*. This file contains the code:

```
attribute fixed_ff of data_0: signal is "H1";
attribute fixed_ff of data_1: signal is "H2";
attribute fixed_ff of data_2: signal is "H3";
attribute fixed_ff of data_3: signal is "H4";
```

This results in the layout shown below:

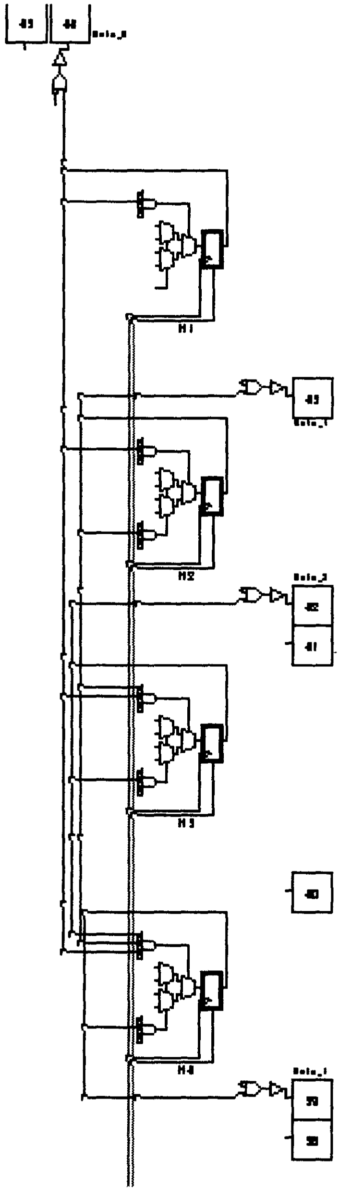


Figure 9-4 Layout with fixed_ff directive

9.6.2.4 The STATE_ENCODING Directive

```
ATTRIBUTE state_encoding OF type_name: type IS value;
```

Next-state equations for state machines with sequential encoding can be complex and product-term intensive. This is particularly undesirable in FPGAs because several cascaded logic cells may be required to complete the equations. A different state encoding scheme can reduce the complexity of the state encoding equations, thus reducing logic cell utilization and ultimately reducing state decode propagation delays. Two state encoding schemes which accomplish this are **one-hot-one** and **one-hot-zero** state encoding.

See Section 9.4.2.1, "The STATE_ENCODING Directive," for more information.

9.7 Documentation Directives

9.7.1 The PART_NAME Directive

```
ATTRIBUTE part_name OF entity_name: entity IS "part_name";
```

A user may want to specify a particular device so that the original design documents specify which device it was designed for. This directive will override any target device command line switch or a Galaxy dialog box setting.

```
entity counter is port (
   a,b: in std_logic;
   ...
   );
   attribute part_name of counter: entity is "c371";
end entity counter;
```

9.7.2 The ORDER_CODE Directive

```
ATTRIBUTE order_code OF entity_name: entity IS "order_code";
```

A particular package and speed bin of a device can be specified to the *Warp* synthesis tool by using the directive **order_code** within the design to ensure timing information reflects the speed grade of the desired part. The order codes can be found in the Ordering Code column of the ordering information table for each device in the *Cypress Semiconductor Programmable Logic Data Book.* Timing delays for CPLDs are calculated according to the speed bin specified by this directive, or if no directive is specified in the VHDL code, the compiler will use the directive specified in the device window of Galaxy.

```
entity counter is port (
   a,b: in std_logic;
   ...
   );
   attribute order_code of counter: entity is "CY7C371-
   66JC";
   end entity counter;
```

9.7.3 The PIN_NUMBERS Directive

ATTRIBUTE pin_numbers OF *entity_name*: entity IS *"string"*;

Once a design has been completed and the board is defined, it may be desirable to maintain the pin out configuration when modifications to the programmable logic design are made. Locking signals to a particular pin can be accomplished by using the **pin_numbers** directive in the design.

```
entity counter is port (
   a,b: in std_logic;
   ...
   );
   attribute pin_numbers of counter: entity is "a:6 b:7 ";
end entity counter;
```

It is recommended that whenever possible, particularly the first time a design is fitted to a device, the pins of a device should not be locked. When the pins are not locked, the fitting tools can choose the optimal fitting arrangement within the device for performance as well as minimal resource utilization. In some rare occasions, certain pin arrangements can render a fitting impossible.

Once a design has been fitted to a device (and the tool has already chosen a working pin configuration), the pin assignments can be back-annotated to the design schematic. The **pin_numbers** directive can also be used to set the pins of the design.

9.8 Directive Format Summary

A summary of the VHDL attribute formats, possible values, and command line switches are provided in Table 9-12.

Table 9-12 Directive formats

Directive	VHDL Format	Values (D=Default)	Cmd line
goal	attribute goal of *arch_name* : architecture is *value;*	speed (D), area, or combinatorial	**ygs** **yga** **ygc**
state_encoding	attribute state_encoding of *type_name* : type is *value;*	sequential (D), one_hot_zero, one_hot_one, or gray	--
buffer_gen	attribute buffer_gen of *signal_name* : signal is *value;*	buf_auto (D), buf_none, buf_normal, buf_duplicate, or buf_register	**yb**
max_load	attribute max_load of *signal_name* : signal is *integer;*	13 (D) or positive integer	**ym**
pad_gen	attribute pad_gen of *signal_name* : signal is *value;*	pad_auto (D), pad_none, pad_clock, pad_hd1, pad_hd2, pad_hd3, pad_hd4, or pad_io	**yp**
synthesis_off	attribute synthesis_off of *signal_name* : signal is *value;*	false (D) or true	--

dont_touch	attribute dont_touch of *label_name* : label is *value;*
	attribute dont_touch of *entity_name* : entity is *value;*
		false (D) or true	--

no_latch	attribute no_latch of *signal_name* : signal is *value;*	false (D) or true	**yl**
lab_force	attribute lab_force of *signal_name* : signal is *location;*	Example: "A1"	--
pin_avoid	attribute pin_avoid of *entity_name* : entity is *location;*	Example: "1 2 3"	--

Directive	VHDL Format	Values (D=Default)	Cmd line
polarity	attribute polarity of *signal_name* : signal is *value;*	pl_default (D), pl_keep, or pl_opt	**fk** **fp**
sum_split	attribute sum_split of *signal_name* : signal is *value;*	balanced (D) or cascaded	--
node_num	attribute node_num of *signal_name* : signal is *value;*	nd_auto (D) or positive integer	**fn**
fixed_ff	attribute fixed_ff of *signal_name* : signal is *location;*	Example: "A2"	**fn**
ff_type	attribute ff_type of *signal_name* : signal is *value;*	ff_default (D), ff_d, ff_t, or ff_opt	**fd** **ft** **fo**
no_factor	attribute no_factor of *signal_name* : signal is *value;*	false (D) or true	**fl**
opt_level	attribute opt_level of *signal_name* : signal is *integer;*	2 (D), 1, or 0	**o**
part_name	attribute part_name of *entity_name* : entity is *string;*	Example: "c371"	**d**
order_code	attribute order_code of *entity_name* : entity is *string;*	Example: "PALC22V10-25HC"	**p**
pin_numbers	attribute pin_numbers of *entity_name* : entity is *string;*	Example: "sig1:1 " & "sig2:2"	**ff**

Chapter 10

Device Programming

Once a design has been compiled, synthesized, and simulated, it is ready to be implemented in silicon. This implementation consists of two steps: the generation of a programming file and the programming of the device. In this section, both steps will be discussed for all devices in the Cypress programmable logic family.

The programming file type that the designer generates depends upon the device type to be programmed. Three programming file types exist for Cypress devices, JEDEC (*.jed*), POF (*.pof*), and LOF (*.lof*). The table below summarizes the file type needed for each of the Cypress device types as well as the steps required to generate these files. These steps are described in detail in this section.

Table 10-1 Programming file types

Device Type	Programming File Type	How to Generate File
Small PLDs, FLASH370 CPLDs	JEDEC	Run Galaxy Go to *Device* menu Output: JEDEC Normal Compile Design
MAX340 CPLDs	POF	Run Galaxy Go to *Device* menu Output: JEDEC Normal Compile Design Run *jed2pof.exe* from DOS
pASIC380 FPGAs	LOF	Run Galaxy Go to *Device* menu Output: QDIF Compile Design Run SpDE Import QDIF Run place and route Export LDF

10.1 Generating a JEDEC File

For programming a small PLD or a FLASH370 CPLD, a JEDEC file is required. In the *Warp* design environment, this file is created as the last step of compiling a design. JEDEC file generation is enabled in Galaxy by clicking on the *Device* button in the main project window, and then selecting *JEDEC Normal* as the

Output option. This programming file will have the same base name as the top-level design file with a *.jed* extension.

Two output file formats are possible when a small PLD or CPLD is selected, JEDEC Normal and JEDEC Hex. Both files contain the same information but slightly differ in format. Whereas the JEDEC Normal represents fuse addresses and data in binary (0 and 1), the JEDEC Hex represents them in hexadecimal (0 through F). Most device programmers require the JEDEC Normal format, and the programmer software will generate errors if the JEDEC Hex file format is used.

Some portions of a JEDEC file are included below to provide an example of the information that it contains:

Cypress c371 Jedec Fuse File: *test.jed*

```
This file was created on 12/11/95 at 10:20:55
by C37XFIT.EXE    06/MAR/95    [v3.17B] 3.5 IR x96

^Bc371*
QP44*                   Number of Pins*
QF13274*                Number of Fuses*
F0*                     Note: Default fuse setting 0*
G0*                     Note: Security bit Unprogrammed*
NOTE DEVICE c371*
NOTE PACKAGE CY7C371-143JC*
NOTE PINS aeqb:2 b_3:10 b_2:11 b_1:13 b_0:32 a_3:33 a_2:35
NOTE PINS a_0:42 a_1:43 *
NOTE NODES *
NOTE NODES *
L000000
000000000010000000000011100000000000000000100000000000000011
100000000000
* Note: LAB 1 BANK OE 0*

L000072
000000010000000011111000000000000000100000000000001000100000
010000000000
* Note: LAB 1 BANK OE 1*
```
(etc.)

```
CC3B5*                   Note: Fuse Checksum*

QV4151*               Note: Number of Test Vectors*
V0001 NL11HFZZ010NC10Z10ZZZNNZZZL11Z111N0ZLLLHHZLN*
V0002 NL11HLZZ010NC10Z10ZZZNNZZZL11Z111N0ZLLLHHZLN*
V0003 NL11HLZZ000NC10Z10ZZZNNZZZL11Z111N0ZLLLHHZLN*
(etc.)

V4151 NL11FF10Z00NCZZZZZZZZNNZZZLZZZ001N1ZLLHLHZHN*
^CFFA0                   Note: File Checksum*
```

At the top of the file is information about the design compilation, including the software revision number, date of compilation, and filename. Further down in the file is the design and device information. The QP field (QP44) tells the user that this file is for a 44-pin device. The QF field denotes the total number of fuses that can be programmed: 13,274 for a CY7C371. A few lines below this are several NOTE fields detailing the device, package, and signal names for the design signals. The device programmer does not use these fields, but simulators use them for package-specific pin numbers and signal names during simulation. Because the programming algorithm does not use this pin information, but rather only uses the fuse numbers for addressing internal locations within the device, the user can program any package of a given device type with the same JEDEC for that device. For example, the designer could use a TQFP package to compile and simulate a design, and then use the resulting JEDEC file to program a PLCC package of the same device. In short, the package information in the file is relevant not for programming but for simulation.

After the NOTE fields, the fuse address and data begins. Each L field in the JEDEC file corresponds to a region of the device. The data following the L field corresponds to the values to be programmed in those locations (1 = programmed, 0 = unprogrammed).

Near the end of the file are two checksums, a fuse checksum and a file checksum. First, the fuse checksum represents the sum of all of the fuse values in the JEDEC file. Device programmers often use this sum to verify that the pattern programmed into the device (number of fuses programmed) matches the number in the JEDEC file. By reading the fuse values from a programmed device, the programmer determines the number of fuses that were programmed. In the sample JEDEC file above, the fuse checksum is C3B5. The checksum value is always preceded by a C.

The file checksum, which is the last line in the file, represents a total value for all characters in the JEDEC file, including both fuse values and notes, comments, and signal names. Using this checksum value, the designer can tell if the programming file has been corrupted or modified. If the file has been changed, the file checksum computed by the device programmer will not match the checksum in the file, and an error will be reported. In the sample JEDEC file, the file checksum is FFA0, preceded by ^C.

Between the fuse checksum and the file checksum are test vectors for the design. Device programmers use these vectors to test the functionality of the programmed device. Using these vectors in sequence, the programmer applies inputs to the device and checks the outputs for the expected values. The QV field, found immediately after the fuse checksum, represents the number of test vectors. This sample design has 4,151 test vectors. Many third-party software companies offer products that automatically generate test vectors for a design using a JEDEC file as the input.

10.2 Generating POF Files for MAX340 CPLDs

The steps required to program the MAX340 CPLDs are identical to those discussed above for FLASH370 CPLDs with one additional step required to produce the programming file. After the design is compiled and produces a JEDEC Normal file, that file must be converted to a POF (*.pof*) file. POF files are binary programming files which are not based on the JEDEC standard. Programming algorithms developed for the MAX340 CPLDs use this format instead of the JEDEC format.

To perform the conversion from JEDEC to POF, the executable *jed2pof.exe* must be run from DOS. This program takes the device type and JEDEC filename as input and produces a file with the same base name and a *.pof* extension. The part can then be programmed on a device programmer. If you are using the *Warp* software on a PC, this utility can be found in the *c:\warp\bin\jed2pof* directory. If you are using a workstation, you can obtain this program from the Cypress Bulletin Board System (BBS) at (408) 943-2954.

10.3 Generating LOF Files for pASIC380 FPGAs

Cypress pASIC380 FPGAs require a different programming file format, the LOF (*.lof*) format. After performing place and route on a design using SpDE, the designer can generate a LOF file by going to the *File* menu in SpDE and selecting *Export LOF*. The fuse information is then stored in the LOF file (*design.lof*). If running SpDE on a PC platform, the user then has the option of zipping the LOF file after it has been generated. Doing so significantly reduces the size of the file (which can be several megabytes) and produces the format required by the Data I/O Unisite programmer. The Cypress *Impulse3* programmer uses the unzipped version of the LOF file instead of the zipped version. If the file has been zipped, the user would unzip it using pkunzip.exe, a popular shareware utility available on the Cypress BBS.

At the top of any LOF file are several fields containing information about the device type, programming file, and the software revision of the SpDE software used to generate the file. Some portions of the LOF file header are included below:

```
Design name:  test
Part name:  p16x24b*
QP144*
(etc.)

QR5.06*
(etc.)
```

In the example above, the QP field represents the number of pins on the device – 144 in this case. The QR field gives the revision number of the SpDE place and route software used to generate the LOF file.

10.4 Device Programmers

Cypress sells a programmer called the *Impulse3* that supports PROMs, small PLDs, CPLDs, and FPGAs. Different part and package combinations require various programming adapters that fit onto the base unit of the programmer. By using the correct programming adapter and generating the programming file as discussed earlier in this section, all of the Cypress devices can be programmed using *Impulse3*. Software updates for *Impulse3* are free and are available on the Cypress World Wide Web home page.

Other third-party vendors such as Data I/O, BP Microsystems, SMS, System General, and Logical Devices offer varying degrees of programming support for Cypress devices. The Data I/O Unisite has the most complete support for Cypress devices of these third-party programmers. The designer should directly contact the manufacturer of these third-party programmers for device support questions. The design flow for programming each type of device is summarized in the following graph.

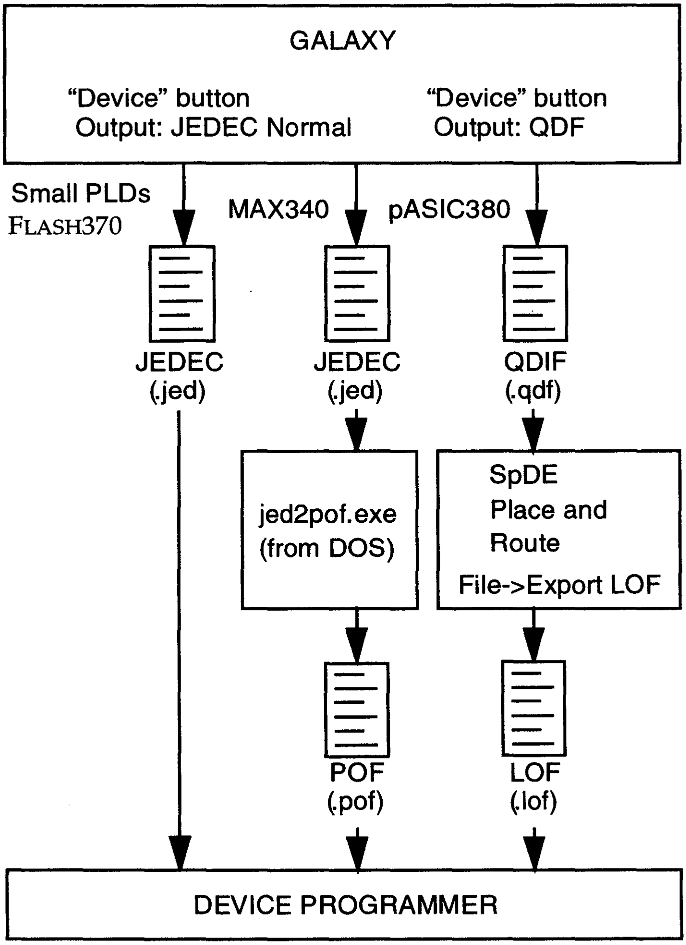


Figure 10-1 Design flow for device programming

Index

Cypress Semiconductor
3901 North First Street
San José, CA 95134
Tel.: (408) 943-2600
FAX: 408-943-2741
FAX-Back: (800) 213-5120
Internet: http://www.cypress.com

ULTRA
LOGIC