IKOS

VirtuaLogic 3.5 User Guide

IKOS Technical Publications

Part Number 6000290-0001

Last Revision 25-Apr-2001

Important Notice

IKOS

Table of Contents

IKOS

List of Figures

List of Tables

IKOS

1 Introduction

Overview

Emulation is a technology that creates a prototype of an Application Specific Integrated Circuit (ASIC) design in hardware. The prototype is generally built through partitioning the design into smaller pieces and then mapping or compiling the design onto a large array of Field Programmable Gate Arrays (FPGAs) or custom chips. The prototype is a complete functional implementation of the design including all digital functions and memories.

The ASICs can be tested under real world operating conditions rather than using an approximation of their operating environment. In-circuit emulators get their stimuli directly from the target system, unlike simulations, which require test programs, testbenches, and stimulus files. The advantages of emulation include full system integration and debugging before the ASIC design is finished and the device fabricated. This is especially true of low level software, such as diagnostics and device drivers, which often require actual target systems for complete testing. The user can design and verify the ASICs, system hardware, and system software at the same time.

VirtuaLogic's technology advantages

The VirtuaLogic Emulation System uses a unique patented technology called Virtual Wires. Virtual Wires provides significant improvements over previous technologies because it makes emulation less expensive and easier to use.

Virtual Wires does not just map a design to the hardware, it actually compiles it for the specific hardware resources. ASIC designs do not map directly to FPGA and emulation custom chip architectures. Traditional emulation products execute a trivial translation process and map the design to the hardware, then attempt to tune the timing implementation by inserting delays into the data path to compensate for hold time violations. Virtual Wires uses advanced synthesis technology to create a functionally identical design that is targeted for the specific hardware of the VirtuaLogic emulator.

With compilation of the design into a single high speed clock and pipelining signals through the machine using this clock, there is only one important delay, which is the worst case path through a Xilinx chip. If the VCLK period is longer than this path, then no setup and hold issues occur. As a result, the operating frequency of the design is immediately known at the completion of the configuration process.

The use of Virtual Wires provides time_domain_multiplexing of multiple signals onto a single FPGA pin or backplane pin. This eliminates the constraint of interconnect, greatly simplifying the hardware and software and decreasing the cost of the hardware.

Without the need for interconnect chips, the resulting hardware is more compact and fewer ICs translate to higher reliability. In addition to eliminating the interconnect chips, the time domain multiplexing is used on the backplane which eliminates the costly connectors and backplanes. The implementation does not have the added propagation delay of the interconnect chip and board delays, therefore potentially resulting in faster emulation speeds. By eliminating interconnect challenges, Virtual Wires provides increased visibility for debugging.

RTL for VStation

The RTL compiler for VStation, hereafter referred as RTLC on VStation, is the primary RTL front-end to the VirtuaLogic emulation system or the VStation. It synthesizes RTL designs to VMW primitives to be used by VirtuaLogic's core compilation system. During this process of compilation, it also generates an RTL debug database to be used by VirtuaLogic for creating an RTL-level interface.

As a result, RTL-compiler is both a front-end to the emulator and a solution with an integrated compile and debug environment; together with IKOS' new co-modeling technology they provide unique verification capabilities for many applications and users.

RTL on VStation expands the VLE use model into earlier in the design and verification cycle. Together with the Transaction Interface Portal, it also allows partially complete designs to be mapped onto the emulator, taking advantage of its superior performance.

Some of the benefits are to:

- directly compile RTL for VStation
- preserve RTL names and debug within the RTL domain

- model memories in software through TIP early in the design cycle and model them into the emulator later for performance
- speed up turn-around times through RTL-supported Multi Module Compile and Incremental Compile

For RTLC-VLE flows (for both verilog and VHDL designs), refer to page 65.

For RTLC-VLE compilation, refer to page 118.

For RTL Compiler options, refer to page 251.

For RTLC Troubleshooting, refer to page 144.

For RTLC Debug Capabilities, refer to page 339.

Transaction Interface Portal

Transaction Level C Interface called Transaction Interface Portal (TIP) enables chip verification with abstract system models by providing a high-speed interface between the workstation running the software model and the design under test running on the VStation hardware. With this advanced capability, designers can perform concurrent hardware and software verification and ensure that the chip works in the system environment before committing to silicon.

Co-modeling, a new verification method made possible by TIP, enables system-level testing by providing high-speed communication between abstract system models and the device under test. Co-modeling is based on Transaction Interfacing. Transaction Interfacing is a technique that separates the communication functionality from timing details to create system level interfaces. Maximum verification performance can be achieved when each transaction executes multiple clock cycles in the design under test. TIP, allows users to verify system functionality with system level stimulus.

A transaction interface is ideal for integration with design flows based on a higher level of abstraction such as C. It easily integrates with abstract C_based design flows, such as CoWare, System C, and Synapps; intelligent testbenches such as Vera and Telecom Workbench from Synopsys; and software processor models such as instruction set simulator models.

The key to co-modeling performance is the speed of the TIP Interface. Since transactions are system level events such as network packets, each transaction contains numerous bits of information. The TIP quickly transfers the entire transaction in a single operation. On the

VStation, each transaction is then decomposed in the pin accurate interface of the device under test. To simplify the handling of data interfacing on the VStation, TIP provides a set of co-modeling macros which bridges the TIP Interface into the design. The co-modeling macros correspond directly to software read and write functions on the workstation, enabling easy integration of co-modeling solutions into the verification environment. *Figure 1 on page 30* gives the user view and the implementation of TIP. For further details on TIP and Co-modeling, refer to TIP user's guide.

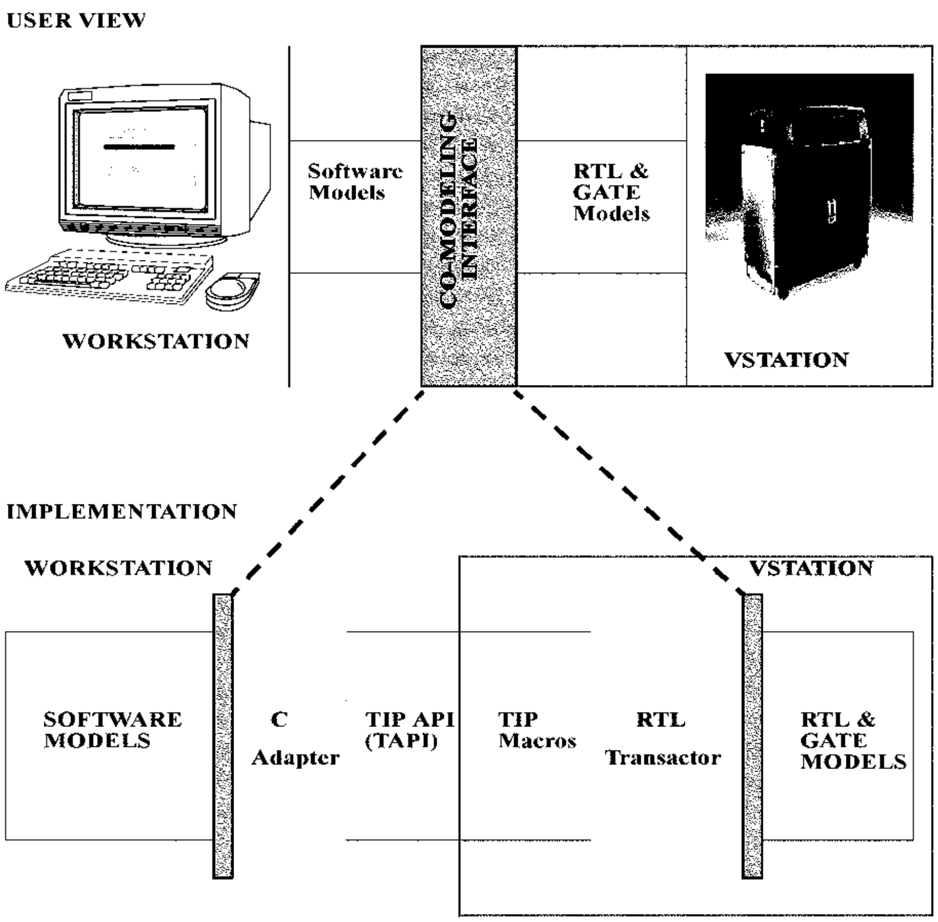


Figure 1 Transaction Interface Portal

VStation Components

The VStation consists of a reconfigurable hardware system, debug hardware, and interfacing hardware. The hardware is driven by software tools used to implement and debug the ASIC design. The entire system is called the VStation or the VirtuaLogic Emulation System.

Hardware

The VirtuaLogic Emulator is the primary hardware component. It is a configurable system that consists of a System Board, up to six Array Boards, IDS(Internal Data Sampling) capability and/or an interface to a HP logic analyzer, an interface for the target system. When programmed with the user's design, the VirtuaLogic Emulator becomes the chip prototype. Please note that the HPLA can be used wherever IDS is used in the document.

Figure 2 on page 32 shows the major system components and demonstrates the connections between them.

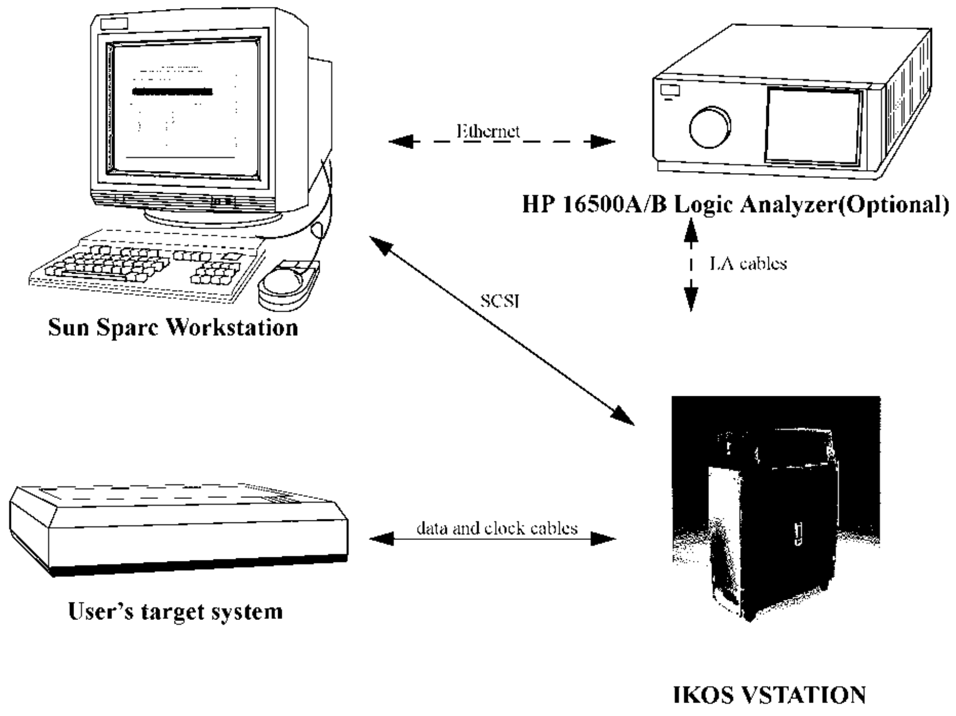


Figure 2 In-circuit system components

The emulator runs at a reduced speed relative to the final silicon due to the partitioning of the design across many chips. The resulting emulation frequency generally ranges between 500 KHz and 2 MHz. The emulation frequency is dependent on the number of logic levels in the design, the partitioning and implementation, and the size of the design. As a rule, most emulated designs run about 50 times slower than the final silicon.

IDS and HP Logic Analyzer

In order to debug the design while it is running in-circuit, an IDS(Internal Data Sampling) capability or a Hewlett Packard Logic Analyzer is used to trigger and capture data. The full event and triggering capability of the HP logic analyzer is supported. If HPLA is used, it must be connected on the Ethernet network with the Sparc^{TM} workstation and be assigned a unique IP address.

HPLA is optional and used if the user wants to store more events. Data on the IDS ranges from 4K to 1 M and for a HPLA, it is limited between 4K and 2M samples. User has to note that the IDS uploads million samples in less than 5 minutes to a contrast of 1 hour by the HPLA.

The HP logic analyzer is connected to each Array Board through the System Board. Each Array Board has the capability to multiplex up to 5000 signals to the System Board, allowing a six Array Board configuration to have up to 30,000 signals available for triggering. For more information on HP Logic Analyzer, refer to the appendix.

Target interfaces

The 512 I/O from each Array Module are connected through data cables to the target. The data cables are connected directly to the target system, or package specific target interface adaptors are available for many common package types. The interface used are TTL, LVTTL or 3.3V CMOS, 5V CMOS. TTL and LVTTL is used to both drive and receive user signals from the emulator. 3.3V or 5V CMOS logic cannot drive but can be used to receive. For more information on target interfaces, refer to the VStation hardware reference manual.

Target system

The target system is a custom system designed and built specifically for the application that is under emulation. An example is a standard Pentium PC which has been slowed down to operate between 500 KHz and 10 MHz. The use of this PC as a part of a target system can aid in the emulation of PC graphics cards, networking cards, or other PC interfaces. Additional components such as a video frame grabber may also be used as a component of the target system to assist with system slowdown.

Software

The VirtuaLogic Software consists of the following primary components:

- RTL Compiler
- VirtuaLogic Compiler
- Backend Place and Route Manager
- Virtual Probe Analysis Tools
- Diagnostics

RTL Compiler

The RTL Compilation reads in the RTL source, then generates the RTL compiled description. This description is referenced to structural netlist that is taken by the VirtuaLogic compiler as input. RTL Compiler also creates a debug database and supports enhanced debug features to allow the user to debug in RTL.

VirtuaLogic Compiler

The VirtuaLogic Compiler includes the graphical user interface, the design importer, and the resynthesis tools. In order to implement the design using the proprietary Virtual Wires technology, the design is resynthesized to an implementation that is better suited to the reprogrammable hardware. This resynthesis provides timing and interconnect resynthesis.

Timing resynthesis

Timing resynthesis is required to assure that the model provides the functionality specified in the design without creating any problems as a result of physical implementation. Since the technology is being translated from an ASIC design to an FPGA design, timing related factors like clock and data skew could introduce problems which would make the design nonfunctional. By resynthesizing, technology dependent timing issues can be avoided. This requires a transformation into a single clock design. As a result, the user does not face the challenges of traditional emulators in resolving hold time problems. This is similar to a cycle based simulator in that it creates a cycle accurate implementation of the design.

Interconnect resynthesis

When a design is partitioned into the smaller logic blocks for emulation, the partitioned groups must not exceed the pin or gate limits of the FPGA that it is being targeted for. Some blocks are limited by the number of gates available on a single FPGA, but most chips are limited by the number of pins available on the FPGA. Interconnect resynthesis resolves this issue through time domain multiplexing techniques. Time domain multiplexing effectively increases the number of pins on the chips. The result is a pipelined and multiplexed implementation of the inter-FPGA signal paths. The average usable gates on each FPGA dramatically increases, reducing the hardware cost.

The Vsyn compiler provides a unique feature in allowing the user to generate a Verilog HDL model of the implementation that is created in the emulator. This allows the designer to run the standard simulation test bench on the design EXACTLY as it is implemented in the emulator and isolate problems that may have been introduced. This eliminates issues related to memory modeling, library modeling, clock specification, lack of specification and validate any assumptions that were made in the creation of the emulation model.

Backend Place and Route manager

VirtuaLogic Backend Place and Route Manager includes the tools needed to place and route the FPGAs that are used to implement the prototype. Since there are 64 FPGAs on each Array Board, or 384 FPGAs in a 6 array board VirtuaLogic emulation system, it is desirable to place and route as many FPGAs in parallel as possible to reduce the compile time. The Backend Place and Route Manager allows the usage of multiple workstations to simultaneously place and route the FPGAs. This dramatically reduces the time required to compile a design.

Virtual probe analysis tools

Virtual Probe Analysis tools have an interface to IDS and HPLA, that allows internal nodes in the chip to be analyzed. It also includes the Waveform Browser, Netlist, and Schematic Browsers for use in debug. It provides an interactive debug environment which allows tracing of the design, selection of Virtual Probes, creation of complex trigger environments, and viewing of probe groups.

Diagnostics

The hardware includes JTAG boundary scan that allows the diagnostic software to provide 100% coverage of FPGAs, boards, and cables in the isolation of hardware problems including shorts, opens, or failed drivers and receivers. Diagnostics provide full coverage testing of the system while only requiring about one hour to run.

2 Using the Graphical User Interface (GUI)

Overview

Using the Graphical User Interface (GUI) chapter covers the following:

- Setting up the environment
- Disk space and CPU requirements
- Invoking VirtuaLogic software
- Navigating the GVL interface

Environment setup

The environment must be setup before running VirtuaLogic.

VLE setup

Enter the following in C shell:

```
setenv VMW_HOME <installation_dir>
set path = ($VMW_HOME/bin $path)
```

The VirtuaLogic software is installed on the network in the <*installation_dir*>. For example,

```
setenv VMW_HOME /hq/support/release/VirtuaLogic_vx.x
set path = ($VMW_HOME/bin $path)
```

If the software is used regularly, the above two commands should be placed in the *~/.cshrc* file. The VLE setup takes care of setting up the RTLC environment automatically.

If the software is run on a remote machine, the software must be told where to display the graphical user interface (GUI) with an environment variable as follows:

```
setenv DISPLAY <hostname>:0.0
```

This command displays the user interface on the host *hostname* to which the user is logged into and using the CPU of another machine. There must be permission on the host where the GUI appears. In this case, while logged into *hostname* , enter the following command:

```
xhost +
```

When logging in, the *.cshrc* file sets the environment variables to the default values.

RTLC setup

Enter the following in C shell:

```
setenv RTLC_HOME <rtlc-vle installation area>

setenv LM_LICENSE_FILE <rtlc-vle license key file>
```

For example,

```
setenv RTLC_HOME  $VMW_HOME/rtlc

setenv LM_LICENSE_FILE $VMW_HOME/license

set path=($RTLC_HOME/bin/solaris_sparc $path)
```

The *rtlc-vle* installation area contains two main executables in $RTLC_HOME/bin/ solaris_sparc. These executables are only for Solaris_Sparc(versions 2.5.x, 2.6, 2.7)

***rtlc-vle*: Core RTL compiler for VLE**

***rtlc-driver*: wrapper over the core rtlc-vle compiler**

Invoking the VirtuaLogic software

The VirtuaLogic compiler maintains a directory called a ***configuration*** which contains all information associated with compiling a design. The configuration directory is *<config_name>.vmw* .

A configuration can be anywhere you choose, and you can have several distinct configurations for the same design.

Configurations

A configuration is a UNIX directory used for building an emulation model. A configuration must have a *.vmw* extension (for example, *ikos top.vmw).*

The configuration directory contains the following:

- Input files that tell the VirtuaLogic compiler where to find its inputs and how to build the model
- Output files that store the compilation results

The VirtuaLogic Graphical Interface (*gvl*) works with one configuration at a time. To control and manage the configuration you are working with, use the buttons on the button bar.

Config_name

To invoke the *gvl*, enter the *gvl* command followed by the name of a configuration. For example,

```
gvl <config_name> &
```

The *gvl* will store all software-generated files relating to the configuration *top* in the directory called *top.vmw*. In this example, *top* is the name of the top-level cell in the example design.

This name is the name that the software uses to store all software-created files. The <config_name> should be in lower case characters. In this example, ***top*** is the name of the top-level cell in the example design.

```
gvl top &
```

Starting gvl on a pre-existing configuration

To start gvl on a pre-existing configuration, use one of the following procedures:

From the GUI:

- Choose *Open*

From the command line:

- *cd* to the parent directory of the configuration to be loaded
- Enter the command *gvl config name &*

or

- *gvl absolute path of conf directory>*

Building a single-ASIC configuration

To build a single-ASIC configuration, use one of the following procedures:

From the GUI:

- Choose *File - Save As*
- Enter the pathname of the configuration to be built

From the command line:

- *cd* to the parent directory of the configuration to be built
- Enter the command *gvl config name &*
- Click *Save*

gvl creates the directory *config name .vmw* and all the required files

Software executables

In addition to a GUI, all programs can be run in a batch mode which allows the model to build and debug processes to be scripted and automated. *Table 1 on page 40* describes the executables and the functions of each executable. As the GUI executes commands, it generates the necessary user input files and creates a log of the commands executed.

Table 1 VirtuaLogic commands

Command	Features/Use
gvl	Brings up a User Interface window to assist in the entry of design data and preparation for emulation
vlc	Generates a virtualized model or compiles a design and runs the place and route tools
browse	Invoke hierarchy browser on design

Table 1 VirtuaLogic commands

Command	Features/Use
tar	Generate a compressed archive of configuration, netlists, and miscellaneous files. Required option is TAR_FILE=archivename.tar
compile	Perform VLE compile
compile_clean	Clean both VLE & FPGA compile data
compile_vtask	Perform VLE & FPGA compile
inc_probe	Perform incremental probe VLE compile
inc_probe_vtask	Perform incremental probe VLE & FPGA compiles
rtl	Perform RTL compile
rtl_compile	Perform RTL & VLE compiles.
rtl_compile_vtask	Perform RTL, VLE & FPGA compiles
verify	Generate netlist for simulation verification
vprobe	Invoke the logic analyzer control program
vtask	FPGA place & route a compiled design
vtask_clean	Remove files created by vtask
mmc_compile_all	Compile all mmc modules

Starting the graphical compiler interface

To start the graphical compiler interface, type the following:

```
gvl &
```

Run the compiler graphical interface to do the following:

- Identify and produce inputs
- Invoke the compiler
- Produce a script for subsequent script-based compiler invocation

The graphical compiler driver performs the following:

- Assembles compiler inputs
- Invokes the compiler
- Verifies inputs
- Produces a script to repeat this process without using the graphical interface

Problems

To avoid problems, or if problems occur:

1. Make sure that the environment variable *VMW_HOME* is defined as the directory in which the VirtuaLogic compiler is installed.

2. Make sure that *$VMW_HOME/bin* is in the executable search path.

Disk space requirements

Disk space is a critical resource for emulation projects. As the design changes, it will be necessary to have multiple builds of a design. A common approach is to maintain the last build that was functional, to debug a current or latest build and simultaneously to be in the process of building the next version of the netlist. This approach requires sufficient disk space for 3 entire builds of a design. However, if multiple chips are integrated or blocks are independently changing, then it is necessary to have up to 3 databases for each of the chips or blocks that independently changes. Normally, disk space for up to 6 simultaneous databases provides some working margin that allows a continuous flow of removing unusable databases prior to a recompile.

In addition to the database and files generated by the VirtuaLogic tools, a build might also include the testbench used for verifying the design and Logic Analysis waveform data. Generally, waveform and vector data is very disk intensive.

The disk space required for a build is linearly related to the design size. For a design of 100,000 gates, a single configured database will require 100 Mbytes of disk space.

CPU requirements

The emulation build requires some complex software algorithms, such as partitioning, that are very CPU intensive. It also contains some tasks, the results of which are that can be distributed and collected by network bandwidth. The creation of the database requires I/O or disk bandwidth. As a result, the task times will be dependent on the CPU, I/O, and network resources. Since the primary workstation should be used for partitioning, it needs to be as powerful as possible. Remote workstations can be used to assist with the FPGA Place and Route tasks which are smaller individual jobs. The remote machines do not need to be as powerful, but the task time will be directly proportional to their available resources.

The recommended primary workstation for a design of 1M gates or larger is an Ultra-Sparc with at least 1-2 Gbytes of RAM and a large local disk for design builds. The remote workstations used for PARs should be at least a Sparc10 or Sparc 5 class with at least 64M of memory. PCs used for place and route should have a microprocessor of 300MHz or better with 256MBytes of RAM for VLE5M and 200MBytes of harddisk for software.

At the completion of a build for a specific design, the compile time and the components that make up the compile time should be analyzed. This information provides guidelines for times required for future compiles. It will also indicate the key areas that require the longest time which provide the best opportunity for compile time improvements. The single largest impact on compile time is normally found through adding additional workstations for the FPGA Place and Route tasks.

Navigating the GVL interface

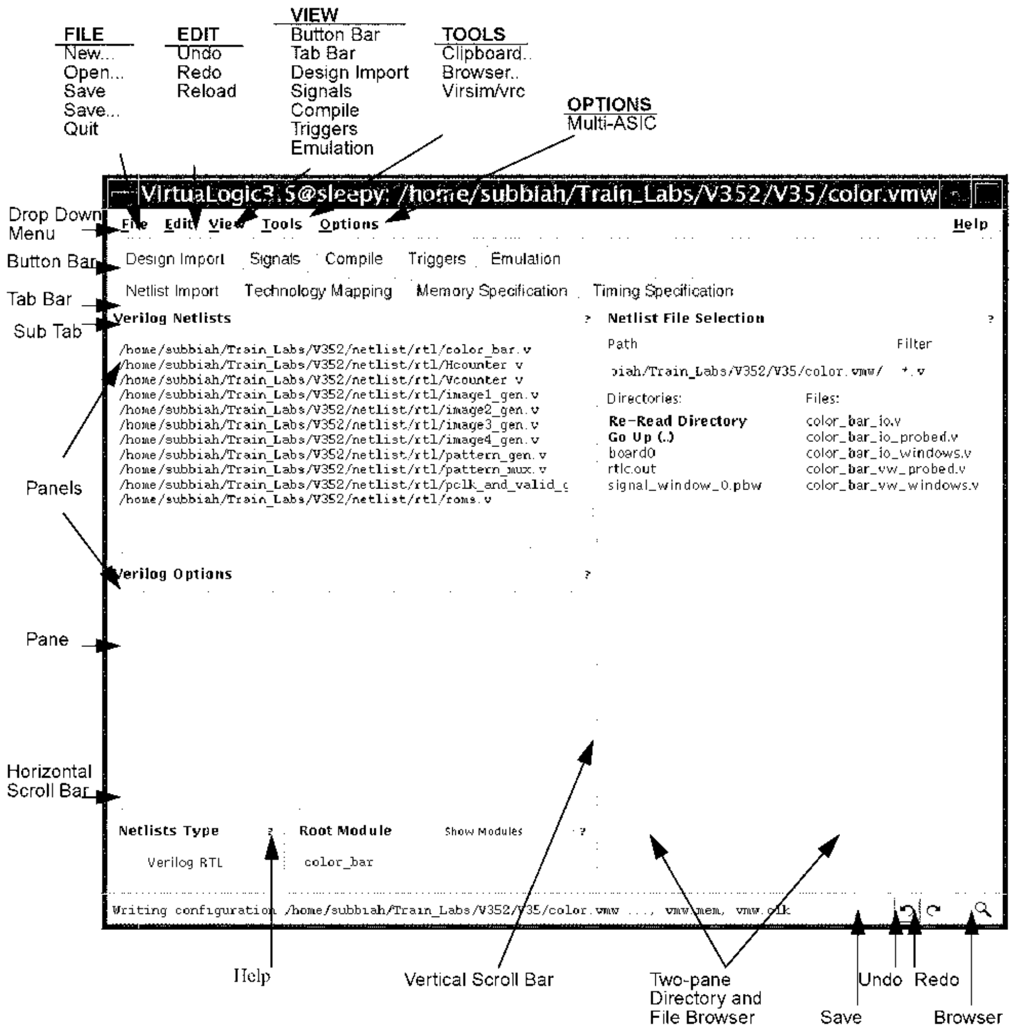
GVL window layout

The tabs across the top of the *gvl* window are attached to pages that control the primary functions of the *gvl* system. These pages contain the inputs and controls to run the needed executables.

When running *gvl*, it records the information needed to build the chip model. Once these specification files have been created, the user can edit and execute them in batch mode to iteratively repeat the compile process without using the GUI.

Screen elements

Each screen in the *gvl* window comprises a combination of elements. *Figure 3 on page 44* illustrates the file card appearance of the interface with these common elements.



```
/home/subbiah/Train_Labs/V352/netlist/rtl/color_bar.v
/home/subbiah/Train_Labs/V352/netlist/rtl/Hcounter v
/home/subbiah/Train_Labs/V352/netlist/rtl/Vcounter v
/home/subbiah/Train_Labs/V352/netlist/rtl/image1_gen.v
/home/subbiah/Train_Labs/V352/netlist/rtl/image2_gen.v
/home/subbiah/Train_Labs/V352/netlist/rtl/image3_gen.v
/home/subbiah/Train_Labs/V352/netlist/rtl/image4_gen.v
/home/subbiah/Train_Labs/V352/netlist/rtl/pattern_gen.v
/home/subbiah/Train_Labs/V352/netlist/rtl/pattern_mux.v
/home/subbiah/Train_Labs/V352/netlist/rtl/pclk_and_valid_g
/home/subbiah/Train_Labs/V352/netlist/rtl/roms.v
```

Path: biah/Train_Labs/V352/V35/color.vmw/ Filter: *.v

Directories:	Files:
Re-Read Directory	color_bar_io.v
Go Up (..)	color_bar_io_probed.v
board0	color_bar_io_windows.v
rtlc.out	color_bar_vw_probed.v
signal_window_0.pbw	color_bar_vw_windows.v

Netlists Type: Verilog RTL
Root Module: color_bar

```
Writing configuration /home/subbiah/Train_Labs/V352/V35/color.vmw ..., vmw.mem, vmw.clk
```

Figure 3 Screen elements

Drag and Drop

Following are the steps to copy or move text from one place to another using drag and drop:

- Position the cursor over the selected text
- Press the left mouse button, sweep over desired text, highlighting it
- Depress the middle mouse button and drag the text to the desired location
- Hold down *control* when pressing the middle mouse button if you want to copy the data instead of moving it
- Check if the file names are brought to the new pane with the full derectory path

Multiple selection

To select multiple items at one time, use one of the following methods:

Method 1:

- Depress the left mouse button
- Move the mouse across successive lines of text
- Drag multiple selections to desired location

Method 2:

- Press *control*
- Click the left mouse button on lines desired
- Drag the multiple selections to the desired location

Scrolling text windows

If the text does not fit in a window, use the highlighted vertical and horizontal scroll bars to position the text where it can be viewed.

To	Do This
Move one line or character	Click on the arrows
Scroll continuously	Hold the cursor on the arrows
See particular text	Drag the position icon to the text

Directory and file browsers

Selecting a file with a two-pane directory and a file browser under ***Netlist File Selection*** (illustrated in *Figure 3 on page 44*) results in the following:

- Directory text pane indicates the current directory
  - Subdirectories are listed in the subdirectory browser pane; double-click to select one as the current directory. To move to the parent directory, double-click the text line *Go Up (.)*
- Files are listed in the file browser pane; to select one, drag to the file drop site

"?" buttons

Clicking a "?" button accesses help information specific to the component or cluster of components.

Optional text fields

Optional fields for the user to fill in appear grayed out. They have a small button next to their name which the user can click to activate the field. After activating the field, type or drag text to fill it in.

Common parts of the interface

Common to all the screens are the button bar, the tab bar, the drop down menu.

Button bar

Figure 4 on page 47 shows a set of buttons which are used to perform different functions and to bring up new forms. By default, the button bar does not show up on the gvl. The user has to click view button bar to bring it up.

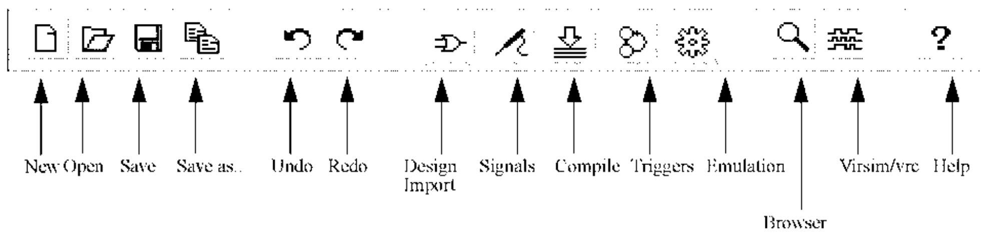


Figure 4 Button bars

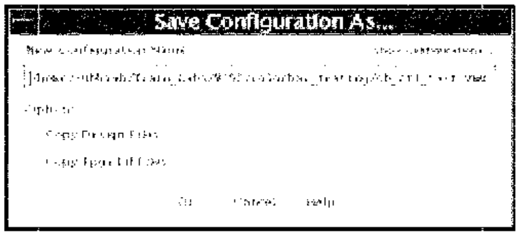
Tab bar

On the tab bar, the user can pick any file cards. Each tab brings up a different page with its own tasks. The *Design Import* page has a sub tab bar with its own file card tabs.

Drop down menu

Near the bottom of every page are buttons for the key commands which represent the most basic functions performed. These button bars give access at all times to commonly used configuration management. Its options include the following:

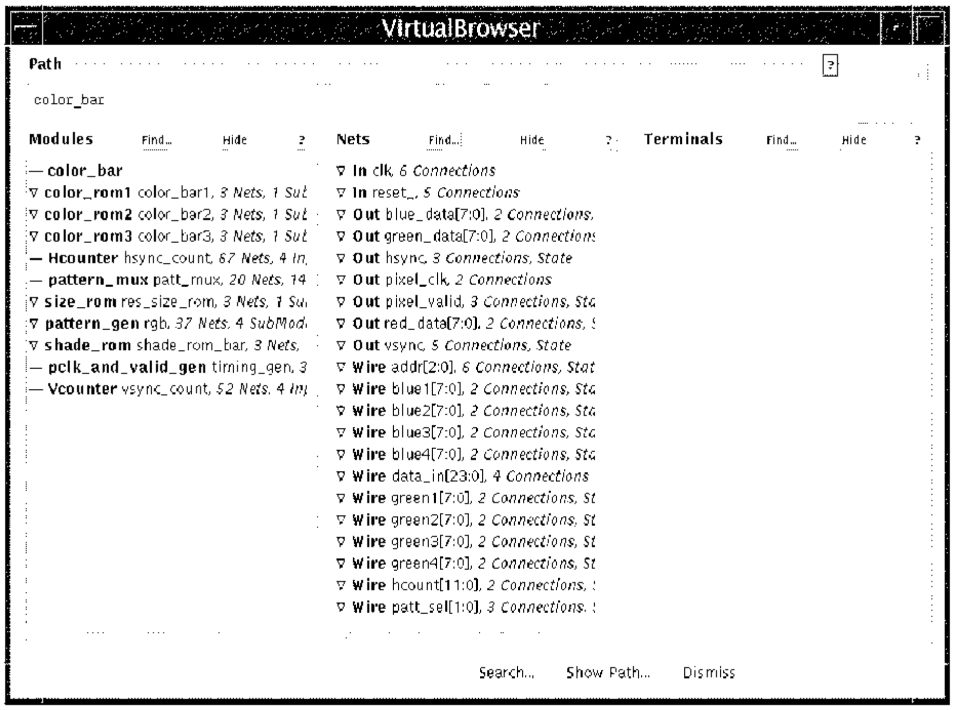
Button	Function
FILE	
Save	Saves the current configuration to disk and lights up the Save As button if a new configuration name is needed.

Button	Function
Save As	Brings up a window to create a new configuration directory and saves the current configuration there. A prompt asks for a configuration directory name.  Copy Design Files: click on box to copy any partition, place, pod-constraint,and trigger files into the new configuration. Copy FPGA Bit Files: click on box to copy any emulation bits into the new configuration
Open	Brings up a directory and file browser window to open a new or existing configuration. Specify a configuration directory name. If it exists, it will be read in. If not, it will be created with the next save operation. If the users does not specify a .vmw extension, one will be added automatically.
Quit	Exits from gvl. If the current configuration has been modified, a prompt asks if it should be saved.
EDIT	
Reload	Discards the netlist in memory (usually needed only if the netlist changed on the disk).
Undo	Undoes the last action.
Redo	Redoes the last action.
VIEW	
Button Bar	Brings up the button bars, which is discussed on page 46.
Tab Bar	Brings up the Tab bar, which is discussed on page 47.
TOOLS	
Browse	Brings up the VirtualBrowser window to browse the design hierarchy and connectivity. Refer to *VirtualBrowser on page 49* for details.
Clipboard	Creates a clipboard window to hold items selected or dragged. This is necessary to move signals from one probe group to another. Refer to *Clipboard on page 58* for details.
VIRSIM/ vrc	Brings up the virsim verification window.

VirtualBrowser

Clicking on the *Browse* button produces the *VirtualBrowser* window which lists all the modules at a design hierarchy level, the network, and the terminal connections.

Figure 5 on page 49 shows the *VirtualBrowser* window.



```
color_bar
```

```
— color_bar
▽ color_rom1 color_bar1, 3 Nets, 1 Sut
▽ color_rom2 color_bar2, 3 Nets, 1 Sut
▽ color_rom3 color_bar3, 3 Nets, 1 Sut
— Hcounter hsync_count, 67 Nets, 4 In,
— pattern_mux patt_mux, 20 Nets, 14
▽ size_rom res_size_rom, 3 Nets, 1 Su
▽ pattern_gen rgb, 37 Nets, 4 SubMod
▽ shade_rom shade_rom_bar, 3 Nets,
— pclk_and_valid_gen timing_gen, 3
— Vcounter vsync_count, 52 Nets, 4 In
```

```
▽ In clk, 6 Connections
▽ In reset_, 5 Connections
▽ Out blue_data[7:0], 2 Connections,
▽ Out green_data[7:0], 2 Connection
▽ Out hsync, 3 Connections, State
▽ Out pixel_clk, 2 Connections
▽ Out pixel_valid, 3 Connections, Sta
▽ Out red_data[7:0], 2 Connections,
▽ Out vsync, 5 Connections, State
▽ Wire addr[2:0], 6 Connections, Stat
▽ Wire blue1[7:0], 2 Connections, Sta
▽ Wire blue2[7:0], 2 Connections, Sta
▽ Wire blue3[7:0], 2 Connections, Sta
▽ Wire blue4[7:0], 2 Connections, Sta
▽ Wire data_in[23:0], 4 Connections
▽ Wire green1[7:0], 2 Connections, St
▽ Wire green2[7:0], 2 Connections, St
▽ Wire green3[7:0], 2 Connections, St
▽ Wire green4[7:0], 2 Connections, St
▽ Wire hcount[11:0], 2 Connections,
▽ Wire patt_sel[1:0], 3 Connections,
```

Figure 5 VirtualBrowser window

Following is a description of the *VirtualBrowser* window components.

Path

This text field displays the path currently selected in the browser. This path can be edited by typing and pressing return or by drag and drop from any appropriate *Virsim* or *gvl* window.

Modules

The *Modules* list shows all the instances at a design hierarchy level with module names in bold. Selecting an instance with the left-mouse button shows all the internal nets in the *Nets* list and all the instance I/O terminals and their externally connected nets in the *Terminals* list.

Double-clicking on an instance expands the instance to show the submodules underneath which are indented to show hierarchy. The submodule instances can then be selected and expanded as well.

In many designs, there may be so many modules that it becomes difficult to deal with the whole hierarchy in one scroll list. The *Set Scope* button operates on the selected instance and makes it the new *root* of the *Modules* list.

The middle-mouse button can be used to drag and drop a hierarchical instance name to the VirtuaLogic windows, *Virsim* waveform display window, or any other Motif-compliant application.

Nets

The *Nets* list shows all the nets inside the currently selected module.

Double-clicking on a scalar net displays the connected instance terminals indented below the net. Selecting one of these instance terminals will show all of that instance's terminals in the *Terminals* list. Selecting a terminal from the *Terminals* list will highlight the connected net, if any, and expand its connections. This feature can be used to trace signal flow across hierarchy and logic.

Double-clicking on a vector net (e.g., a[15:0]) will expand the vector which allows the user to follow individual connections as for scalar.

In order to present a manageable set of data to the user, the VirtuaLogic GUI always tries to vectorize nets together using pattern matching even if the input netlist has only scalars. The user can double-click on a vectorized net to work with the scalars.

Double-clicking on a net that is already expanded will collapse it again.

The middle-mouse button can be used to drag and drop hierarchical net names to other VirtuaLogic windows, Simulation Technologies' Virsim waveform display window, or any other Motif-compliant application.

Select a group of nets by holding down the left-mouse button and drag the mouse over the range of elements wanted. It can also be done by holding down the *control* button while clicking on the left-mouse button to add additional elements. The drag and drop operation will include all the nets in the group.

Terminals

The *Terminals* list shows all the terminals and connected nets of the instance most recently selected in the *Modules* and *Nets* lists.

Selecting a terminal will highlight the connected net if there is one.

Terminals are displayed in vector form just as nets are. Similarly, double-clicking a vector terminal will display the scalars underneath.

In some cases, a vector terminal will be shown as connected to a vector net with the same *signature*. In this case, the user can navigate vector connections without expanding them.

Double-clicking on an expanded vector will collapse it again.

Find

The *Find* is used to find a string in the list. Clicking on the *Find* button will produce the window illustrated in *Figure 6 on page 51*.

Type into the box the name of the string in the list to find.

- Case sensitivity, if selected, uses case sensitive string comparisons
- Wildcard match, if selected, allows wildcards
  - * means match any number of characters (0 or more); place at the beginning and end of the string to match an embedded string
  - ? means match exactly one character

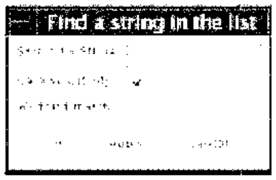


Figure 6 Find window

Search

The *Search* button searches hierarchically through a design for objects matching a given pattern.

Show path

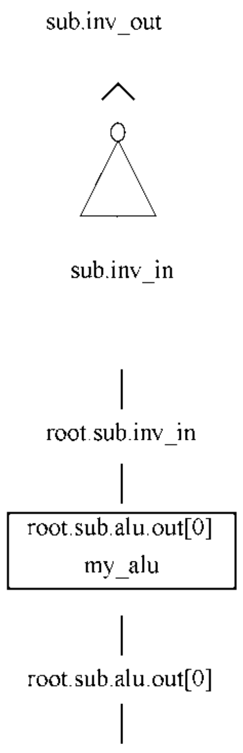
Graphical Path Display command draws electrical connectivity graphs of subsets of the design. It is used to trace cones of logic to some terminal point. A typical example would be to trace a net's fanout until it hits a state or memory element showing all the modules and connections in between.

Minimally, the path display engine must be provided the following:

- A starting point indicated by the nets selected in the *VirtualBrowser*. These nets can be selected while the path display invocation dialog box is still up
- A direction: drivers(fanin) or receivers(fanout)
- A set of termination conditions. Whenever the path tracer matches one of the conditions, it will stop that branch of the design traversal and draw it; other branches well continue to be traversed

The resulting graphical display has to have the following features:

- Nets are drawn with red line segments and indicate the direction of the signal (i.e., fanin or fanout) with arrows and are labeled with the full path of the net
- Primary I/Os are drawn as green boxes and labeled with the I/O name
- Combinational primitives are drawn with graphics
- Other modules are drawn as boxes with the module name and path displayed inside, as well as terminal names
- The resulting drawing will show the original selected nets at the top; the gate level view shows all nets as scalars, the rtl-level view shows a vector view corresponding to the original RTL source
- state elements are drawn in purple with triangle in the lower left corner.



- Power is drawn in red. Ground is drawn in black. Combinational primitives are drawn in green, Flop and latch primitives are drawn in purple with a triangle at the lower left corner. Primary inputs are drawn in dark green.
- Feedback nets are drawn as brown arcs. These typically occur when drawing a path that does not stop at state or memory elements. Typically the user will press the *Stop at State* button to cut the feedback.
- All terminals and net path names displayed in the graphics can be selected and used for drag and drop to the *Clipboard, VirtualBrowser,* or *Virsim*. When selected, the path names are placed in the X cut buffer so they can be pasted to any standard X window.
- The user can display as many graphical paths as needed, each in their own window.

- There are seven zoom levels. The three highest zoom levels display text in three different sized fonts. The four lowest zoom levels draw the lines and boxes only. They are mainly useful for giving a context for scrolling around a large zoomed-in display. By pressing the *Apply* button in the dialog box twice, the user can see two magnification levels side by side.
- As the mouse is moved over the graphics, the path of the object underneath the mouse is displayed at the top of the window. This can help the user find a way around at low zoom levels.

Figure 7 on page 54 shows the graphical path browser in flattened design mode.

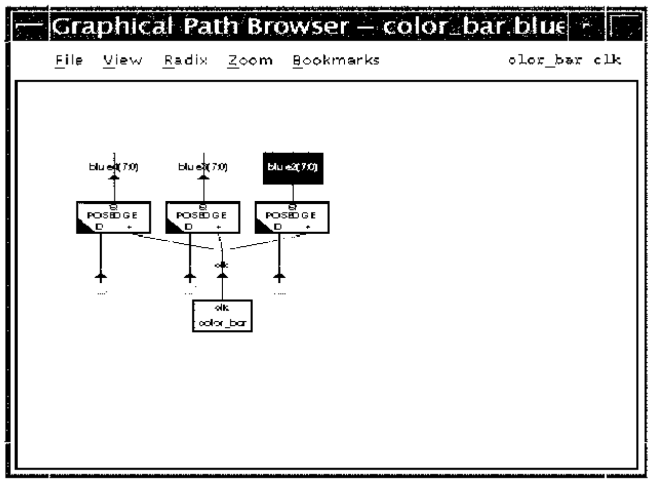

Graphical Path Browser – color_bar.blue
File View Radix Zoom Bookmarks
olor_bar clk
blue0(7:0)
blue1(7:0)
blue2(7:0)
POSEDGE
D
clk
color_bar

Figure 7 Graphical path browser(Flattened design mode)

Figure 8 on page 55 shows the graphical path browser in hierarchical design mode.

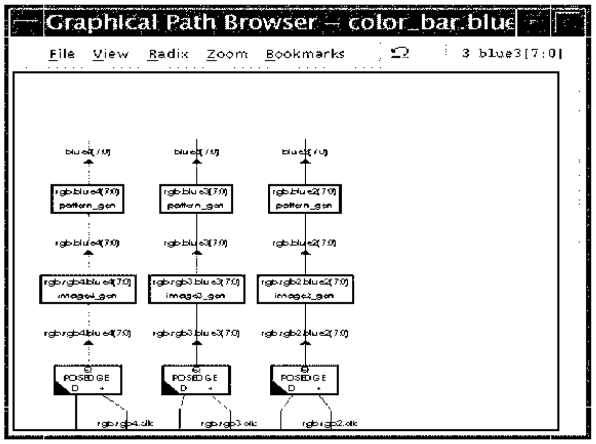

Graphical Path Browser – color_bar.blue
File View Radix Zoom Bookmarks
3 blue3[7:0]
blue4[7:0]
blue3[7:0]
blue2[7:0]
rgb.blue4[7:0] pattern_gen
rgb.blue3[7:0] pattern_gen
rgb.blue2[7:0] pattern_gen
rgb.blue4[7:0]
rgb.blue3[7:0]
rgb.blue2[7:0]
rgb.rgb4.blue4[7:0] image4_gen
rgb.rgb3.blue3[7:0] image3_gen
rgb.rgb2.blue2[7:0] image2_gen
rgb.rgb4.blue4[7:0]
rgb.rgb3.blue3[7:0]
rgb.rgb2.blue2[7:0]
POSEDGE
rgb.rgb4.clk
rgb.rgb3.clk
rgb.rgb2.clk

Figure 8 Graphical path browser(Design hierarchy mode)

Graphical path browser also has the capability of switching feature from RTL design to Gate level netlist and vice versa.

Additional features

Double-clicking on a terminal or net causes the path below that element to be collapsed. Collapsed terminals and nets are displayed with a button below them. Collapsed elements can be expanded by clicking their button or by double-clicking. Moreover, elements that are stopped by specification in the *Show Path* dialogue window are displayed as collapsed. Expanding these elements expands the path continuing from them until the next set of stopped nodes.

Two simple examples to illustrate the function of the path display mechanism are as follows:

1. Draw a schematic of a module
   - In the *VirtualBrowser*, select a module and then select all its output nets
   - Click on the *Show Path* button
   - In the dialog box, select the following:
     - fanin
     - stop at I/O
   - Click on OK

This will trace the *fanin* of all the outputs all the way to that module's inputs. Try this on a simple module at first to see what it generates.

If you try this example on a complex module, the result will probably be an error message as follows:

```
Maximum number of nodes (1000) exceeded -- increase?
```

At this point, the user can increase the *Maximum number of nodes* field in the dialogue box. Note that if a very complex module is selected, the graphical display will not be very helpful; therefore, the default value is set fairly low. The node count limit is somewhat arbitrary as it counts all the traversals done by the path display engine, including those that do not lead in a successful termination. It also counts traversals into the library and primitive modules which are not displayed to the user.

2. Trace a clock tree
   - In the *VirtualBrowser*, select a module somewhere in the hierarchy and then select a clock net
   - Click on the *Show Path* button
   - In the dialogue box, select the following:
     - Fanout
     - Show Hierarchy
     - Stop at State
   - Click on OK

This will show all fanout of the clock net that eventually winds up in state elements. Note that if you have selected a starting point in the hierarchy that has a lot underneath it, the result can be overflowing the default maximum node count. The user should increase it or limit the scope by starting at a lower position in the hierarchy.

Regular expression syntax

A regular expression is zero or more *branches*, separated by '|'. It matches anything that matches one of the *branches*.

A *branch* is zero or more *pieces* concatenated. It matches a match for the first, followed by a match for the second, etc.

A *piece* is an *atom* possibly followed by '*', '+', or '?'. An *atom* followed by * matches a sequence of zero or matches of the *atom*. An *atom* followed by '+' matches a sequence of one or more matches of the *atom*. An *atom* followed by '?' matches a match of the *atom* or null string.

An *atom* is a regular expression in parenthesis (matching a match for the regular expression), a *range* '.' (matching any single character, see below), '^' (matching the null string at the beginning of the input string), '$' (matching the null string at the end of the input string), a '\' followed by a single character (matching that character), or a single character with no other significance (matching that character).

A *range* is a sequence of characters enclosed in '[ ]'. It normally matches any single character from the sequence. If the sequence begins with '^', it matches any single character not from the rest of the sequence. If two characters in the sequence are separated by '-', this is shorthand for the full list of ASCII characters between them (e.g., [0-9] matches any decimal digit). To include a literal ']' in the sequence, make it the first character (following a possible '^'. To include a literal '-', make it the first or last character.

Ambiguity

If a regular expression could match two different parts of the input string, it will match the one which begins earliest. If both begin in the same place but match different lengths, or match the same length in different ways, the following applies:

In general, the possibilities in a list of branches are considered in the left-to-right order, the possibilities for '*', '+', and '?' are considered longest-first, nested constructs are considered from the outermost in, and concatenated constructs are considered left most-first. The match that will be chosen is the one that uses the earliest possibility in the first choice that has to be made. If there is more than one choice, the next will be made in the same manner (earliest possibility) subject to the decision on the first choice, and so forth.

For example, '(ab|a)b*c''could match 'abc' in one of two ways. The first choice is between 'ab' and 'a'; since 'ab' is earlier and does not lead to a successful overall match, it is chosen. Since the 'b' is already spoken for, the 'b*' must match its last possibility (the empty string) since it must respect the earlier choice.

In the particular case where no '|' are present and there is only one '*', '+', or '?', the net effect is that the longest possible match will be chosen. Therefore, 'ab*' presented with 'xabbbby' will match 'abbbb'. Note that if 'ab*' is tried against 'xabyabbbz', it will match 'ab' just after 'x' due to the begins-earliest rule. (In effect, the decision on where to start the match is the first choice to be made, hence subsequent choices must respect it even if this leads them to less preferred alternatives).

Regular expression example

The regular expression 'a[0-9]*(b)|(c)' matches any string beginning with 'a', followed by any number of digits, followed by 'b' or 'c', including 'ab', 'a08398b', 'a08398d'

Clipboard

Clicking on *Clipboard* creates a *Clipboard* window to hold items selected or dragged. This is necessary to move signals from one probe group to another.

Figure 9 on page 58 shows the *Clipboard* window.

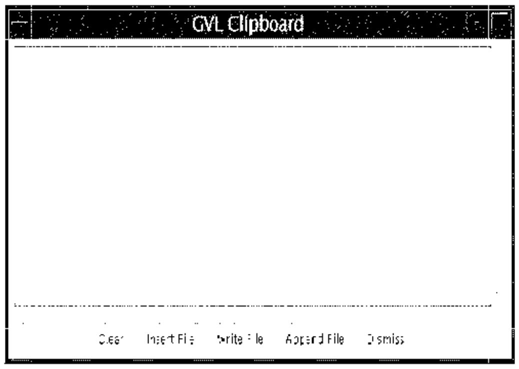


Figure 9 Clipboard window

Clear

Deletes all the text in the clipboard.

Insert file

Inserts the contents of a file into the clipboard.

Write file

Writes the contents of the clipboard into a file, overwriting what was there before.

Append file

Appends the contents of the clipboard onto an exiting file, or creates it if it does not yet exist.

Dismiss

Closes the window.

Reload

Selecting the *Reload* button discards the netlist in memory. This is usually needed only if the netlist changed on the disk.

Undo

Selecting the *Undo* button undoes the last action.

Quit

Selecting the *Quit* button exits the *gvl*. If the current configuration has been modified, a prompt asks if it should be saved.

Errors window

The *Errors* list is used for display, navigation, and cut and paste of error information.

Most errors displayed here can be navigated by clicking on the *Visit(next)* button. Depending on the nature of the error, a user interface component may be highlighted or a text editor may be brought up, pointing to the specified line in a file.

Some errors are not navigable. In this case, the *Visit(next)* button is grayed out.

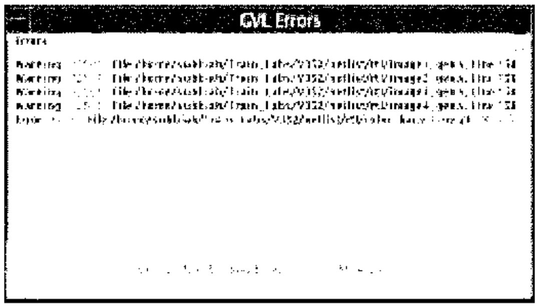
The text editor used depends on the environment variable EDITOR, as per UNIX convention. This can be overridden using the environment variable VMW_EDITOR.

Due to the way *gvl* uses $EDITOR to show lines in text files, IKOS recommends using *emacsclient* for an $EDITOR. Typically, several errors will be tracked consecutively in the same file. *Emacsclient* will read in the file only the first time. Move the cursor to the appropriate line number each time you double-click. The *emacs* and *vi* editors will read in the file from scratch each time in a new window.

For this reason, *gvl* will ignore $EDITOR values other than *emacsclient*. If the user wants another editor, specify it as $VMW_EDITOR, *gvl* will use it; however, the way it deals with multiple errors in the same file is not ideal.

If $EDITOR and $VMW_EDITOR are not set, or $EDITOR is set to something other than *emacsclient*, then *gvl Clipboard* is used to display errors. This editor is suitable for viewing errors, making small corrections, and saving them; however, it is not a full-featured text editor.

Figure 10 on page 60 shows an *Errors* window.



```
Warning [illegible] file /home/[illegible]/Train_Labs/V352/[illegible]/image1_gen.s, line [illegible]
Warning [illegible] file /home/[illegible]/Train_Labs/V352/[illegible]/image2_gen.s, line [illegible]
Warning [illegible] file /home/[illegible]/Train_Labs/V352/[illegible]/image3_gen.s, line [illegible]
Warning [illegible] file /home/[illegible]/Train_Labs/V352/[illegible]/image4_gen.s, line [illegible]
Error [illegible]
```

Figure 10 Errors window

OK

Click to dismiss this error dialogue.

Visit(next)

Click to list the source of the next error.

Help

Brings up error specific help when available.

Show Log

Click to activate.

Save Errors

Clicking on *Save Errors* produces the *File Browser* window as shown in *Figure 11 on page 61*.

File Browser

The *Save Errors* button on the *GVL Errors* window invokes the *File Browser* window as illustrated in *Figure 11 on page 61*.

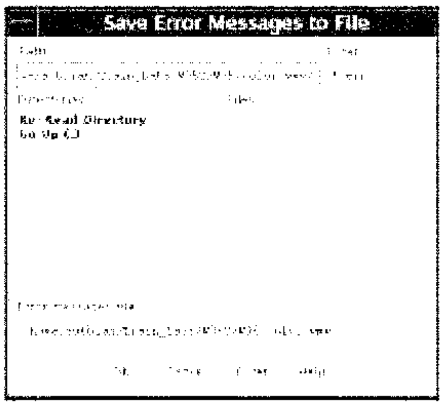


Figure 11 File browser

Path

The *Path* field indicates the current directory.

Directories

Subdirectories are listed in the *Directories* pane. Double-click to select one as the current directory. To move to the parent directory, double-click the subdirectory indicated by *Go Up (..)*.

Files

Files are listed in the *File* pane. To select one, drag to the file drop site.

OK

Click to accept the selected pathname and proceed.

Filter

Click to apply the wildcard filter to the set of files.

Cancel

Click to dismiss the dialogue box without proceeding.

Optional text fields

Optional fields for the user to fill in appear grayed out. They have a small button next to their name which is clicked to activate the field. After activating the field, type or drag the text to fill it in.

3 Design Import

Overview

Design Import creates the configuration files needed by the compiler to create an emulation model for the design. The four functions of *Design Import* are as follows:

- *Netlist import on page 70*
- *Technology mapping on page 74*
- *Memory specification on page 77*
- *Timing specification on page 88*

Together these pieces describe how the emulation system behaves. Compiling produces a design model which is simulated to verify the input specification.

For both VHDL and Verilog RTL designs, the environment variable VMW_HOME should point to the installation area.

RTL Verilog flow

VLE GUI is completely supported for Verilog flow. The user takes the *Verilog RTL* design and imports the netlists using the design import form. The netlist type is mentioned as Verilog RTL in the design import form. This RTL Verilog design has to be compiled by ***rtlc*** first and then taken to ***vsyn*** compile.

Figure 12 on page 66 shows the design flow for Verilog RTL designs.

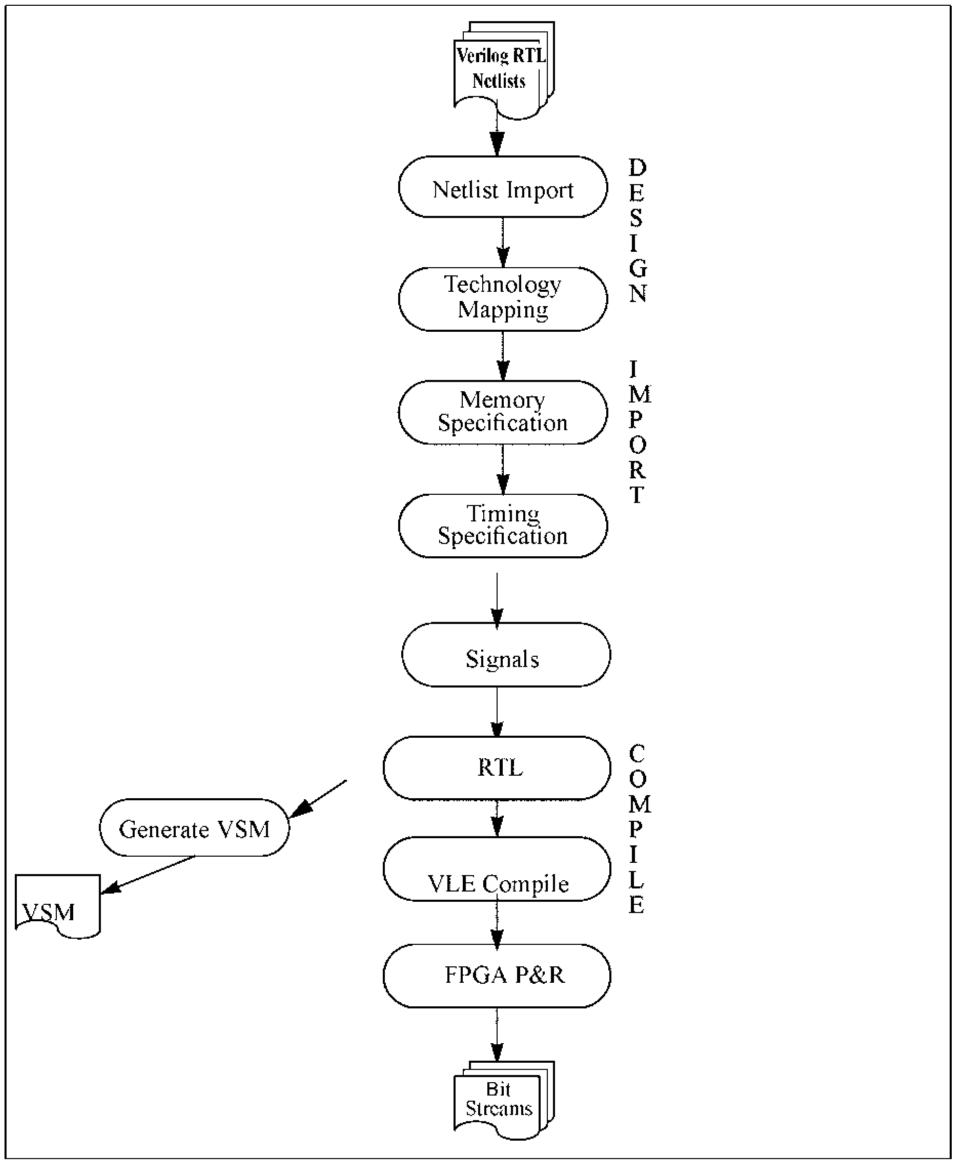


Figure 12 Design flow for RTL Verilog designs

Verilog RTL user flow

Steps Using GUI:

- Import Verilog RTL files through the GUI
- Specification of the netlist type through the GUI
- Specification of the Root(top-level) Module
- Specification of memory information
- Specification of timing information
- Specification of the RTLC compiler options
- Specification of the VSYN compiler options
- Specification of machines available for XILINX compilation
- Start Compilation(single button compilation)

RTL VHDL flow

VLE GUI is not supported for the VHDL flow. VHDL flow is used as a stand alone tool. The user takes the *VHDL RTL* design and compiles the design through *rtlc-vle* on the command line prompt. For VHDL, the RTLC_HOME environment variable must be set in order to use rtlc-vle.

The output netlist file is generated in terms of vmw primitive gates and stored in <out_dir>/out.synth.v file (by default *rtlc.out/out.synth.v*). The generated netlist file is imported using the design import form. In the design import form, the netlist type is mentioned as Verilog Gates. This design is taken directly to the ***vsyn*** compile as the rtl compile stage is already completed. For RTL compiler process, refer to *RTL Compile on page 118*.

Figure 13 on page 68 shows the design flow for RTL designs using VHDL.

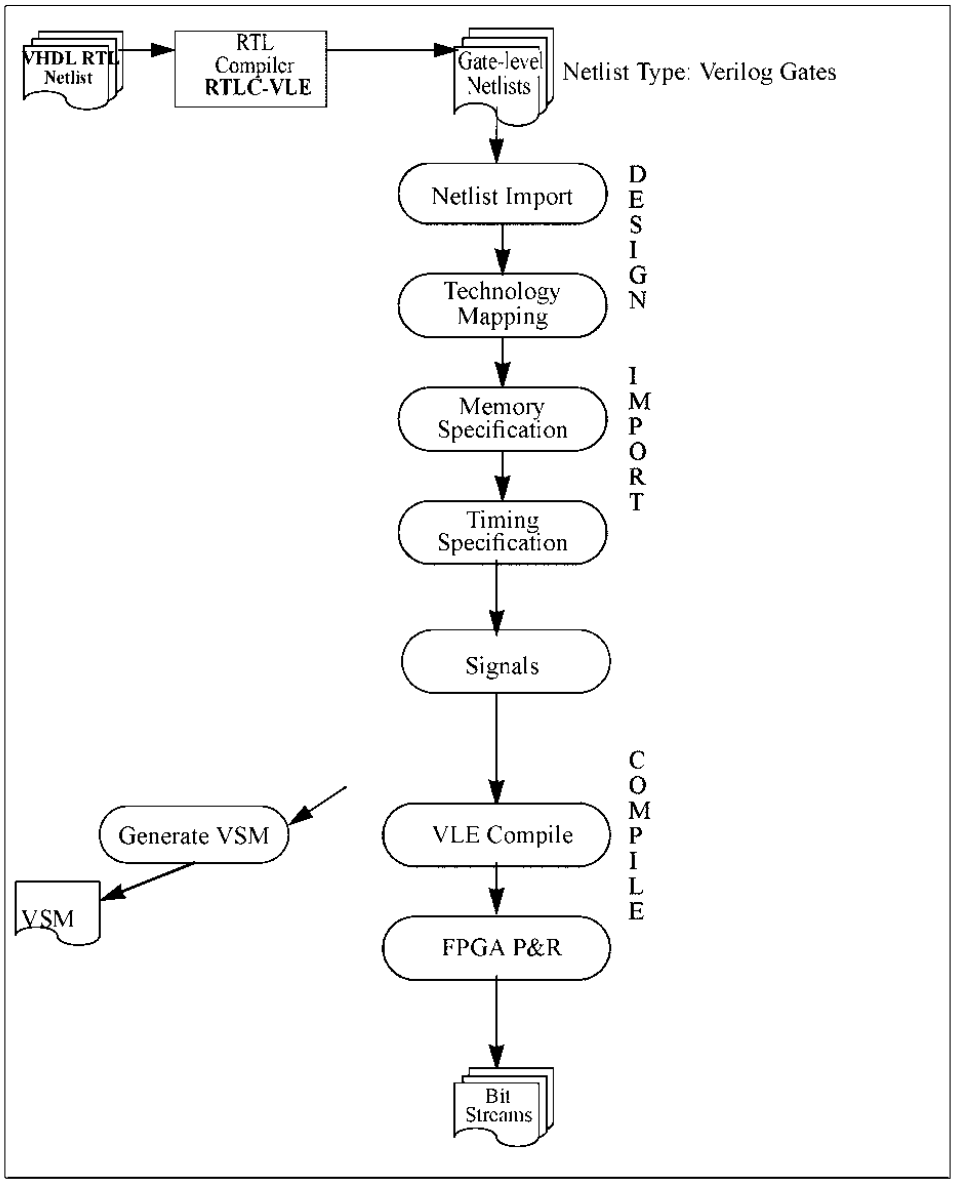


Figure 13 Design flow for RTL VHDL designs

VHDL RTL user flow

For designs with multiple logical libraries, a *.jaguarc* file is created in the current working directory or user's home directory specifying all the logical libraries.

Consider a design which consists of the libraries chiplib, chipmem and work, and the files are chiptop.vhd, chipcore.v, chipmem.v and chiplib.v which are analyzed into the logical libraries work, work, chipmem, chiplib respectively. A *.jaguarc* file is created by user for this case, which looks like

```
MY_WORK => /path/to/lib/work
CHIPMEM  => /path/to/lib/chipmem
CHIPLIB     => /path/to/lib/chiplib
```

If the files chiptop.vhd and chipcore.vhd depend on chipmem and chiplib, rtlc-vle is called in the following order for analysis:

```
% rtlc-vlc -analyze -hdl vhdl -w chiplib chiplib.vhd

% rtlc-vlc -analyze -hdl vhdl -w chipmem chipmem.vhd

% rtlc-vlc -analyze -hdl vhdl -w chipcore.vhd chiptop.vhd # default lib = work
```

If the top level entity is chiptop, compilation is invoked by:

```
% rtlc-vlc -ent chiptop -import chipmem
```

The netlist is generated in verilog and is found in rtlc.out/out.synth.v.

NOTE

VSYN compiler supports only verilog netlist as input. VHDL RTL designs are passed through *rtlc-vle* to get the verilog gate level netlist.

Design import

The *Design Import* reads in the **RTL**-level netlist for the verilog designs. It checks the design for syntax errors and unacceptable Verilog constructs.

To verify the emulation inputs and to establish a benchmark functionality for the design, it is required to have a simulatable design model and testbench (i.e., design netlist, technology library, memory models, and test fixture models), each of which is a distinct Verilog model or library as follows:

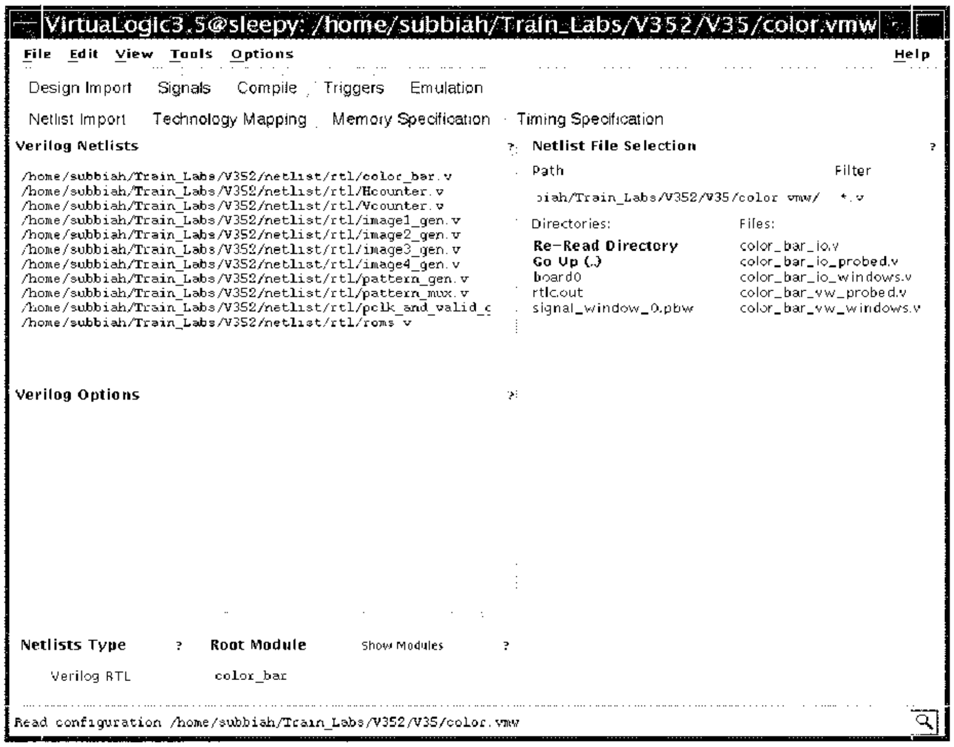
- Design netlist is a Verilog RTL design or structural representation of the design, using primitive cells from the Application Specific Integrated Circuit (ASIC) vendor's library
- Technology library is technology mapping for primitives in the ASIC vendor's library into IKOS primitives
- Memory models are behavioral models for all memory elements in the design
- Design testbench (test fixture) is a simulation model that provides stimulus and measures responses of the ASIC design to test if it is functioning correctly; the user must be familiar enough with the testbench to recognize when it indicates success or failure

To use these pieces most efficiently, structure each piece as one or more separate files.

Netlist import

The netlist is defined in terms of library primitives and it must be in Verilog RTL/structural format. The library primitives can be an ASIC vendor library or the IKOS library. In addition, the user must identify the top module in the hierarchy of the design to the software. The top module in the hierarchy is also called the design name.

Figure 14 on page 71 shows the *Netlist Import* window.


VirtuaLogic3.5@sleepy: /home/subbiah/Train_Labs/V352/V35/color.vmw
File Edit View Tools Options Help
Design Import Signals Compile Triggers Emulation
Netlist Import Technology Mapping Memory Specification Timing Specification
Verilog Netlists
Netlist File Selection
Path
Filter
Directories:
Files:
Re-Read Directory
Go Up (..)
Verilog Options
Netlists Type
Root Module
Show Modules

```
/home/subbiah/Train_Labs/V352/netlist/rtl/color_bar.v
/home/subbiah/Train_Labs/V352/netlist/rtl/Hcounter.v
/home/subbiah/Train_Labs/V352/netlist/rtl/Vcounter.v
/home/subbiah/Train_Labs/V352/netlist/rtl/image1_gen.v
/home/subbiah/Train_Labs/V352/netlist/rtl/image2_gen.v
/home/subbiah/Train_Labs/V352/netlist/rtl/image3_gen.v
/home/subbiah/Train_Labs/V352/netlist/rtl/image4_gen.v
/home/subbiah/Train_Labs/V352/netlist/rtl/pattern_gen.v
/home/subbiah/Train_Labs/V352/netlist/rtl/pattern_mux.v
/home/subbiah/Train_Labs/V352/netlist/rtl/pclk_and_valid_c
/home/subbiah/Train_Labs/V352/netlist/rtl/roms v
```

Path: `biah/Train_Labs/V352/V35/color vmw/` Filter: `*.v`

Directories:	Files:
Re-Read Directory	color_bar_io.v
Go Up (..)	color_bar_io_probed.v
board0	color_bar_io_windows.v
rtlc.out	color_bar_vw_probed.v
signal_window_0.pbw	color_bar_vw_windows.v

Netlists Type: Verilog RTL Root Module: color_bar

```
Read configuration /home/subbiah/Train_Labs/V352/V35/color.vmw
```

Figure 14 Netlist import

Netlists

The *Netlists* frame is used to list the Verilog source files for emulation.

Entering pathnames

Use one of the following procedure to select files which contain Verilog code for the design:

- Click the left mouse button in the frame and type the desired files.
- Select the directories and files wanted from the *Netlist File Selection* with the left mouse button and drag and drop them with the middle mouse button.
  - *Path* contains the current directory

  - Choose all file by typing * in the *Filter* field
  - A *bin* in the *Path* field and *.v in the *Filter* field brings up all subdirectories of *bin* in the *Directories* pane and all files matching *bin* *.v in the *Files* pane
- Use "X windows" to copy, cut and paste from another "X window" (e.g., xterm or text editor).

Select files for just the netlist and memories. If there are prototypes or behavioral descriptions in the memories, import these files also. This makes producing a memory specification much easier.

Do not include primitive libraries or testbench files.

Netlists requirements

- Verilog netlists must be entirely structural
- Primitive components must be in a library supported by VirtuaLogic (refer to *Technology mapping on page 74* for information)
- A netlist can be divided into multiple files, each on its own line
- Module definitions for memories may be as follows:
  - Omitted from the netlist
  - Included as interface prototypes
  - Included as a behavioral specification

In all cases, memories must be specified using the *Memory Specification* window. Refer to *Memory specification on page 77* for details.

Netlist defines

The *Netlist Defines* frame provides a means to set Verilog preprocessing variables for use when reading in the netlist. This corresponds to the Verilog simulation command line options as follows:

```
+define+VARIABLE
```

and

```
+define+VARIABLE=VALUE
```

For each variable to be defined, type the following:

```
VARIABLE
```

To set it to a value, type the following:

```
VARIABLE=VALUE
```

Each Verilog preprocessor variable in the *Netlist Defines* frame should appear on its own line. The user can cut and paste the preprocessor *Netlist Defines* information from the simulation *Makefiles*.

The syntax to undefine a variable is as follows:

```
-VARIABLE
```

This is equivalent to the Verilog simulation command line syntax:

```
-define-VARIABLE
```

Input netlist type

All input netlists must be of the same type: Verilog Gates or Verilog RTL. Select the desired netlist language.

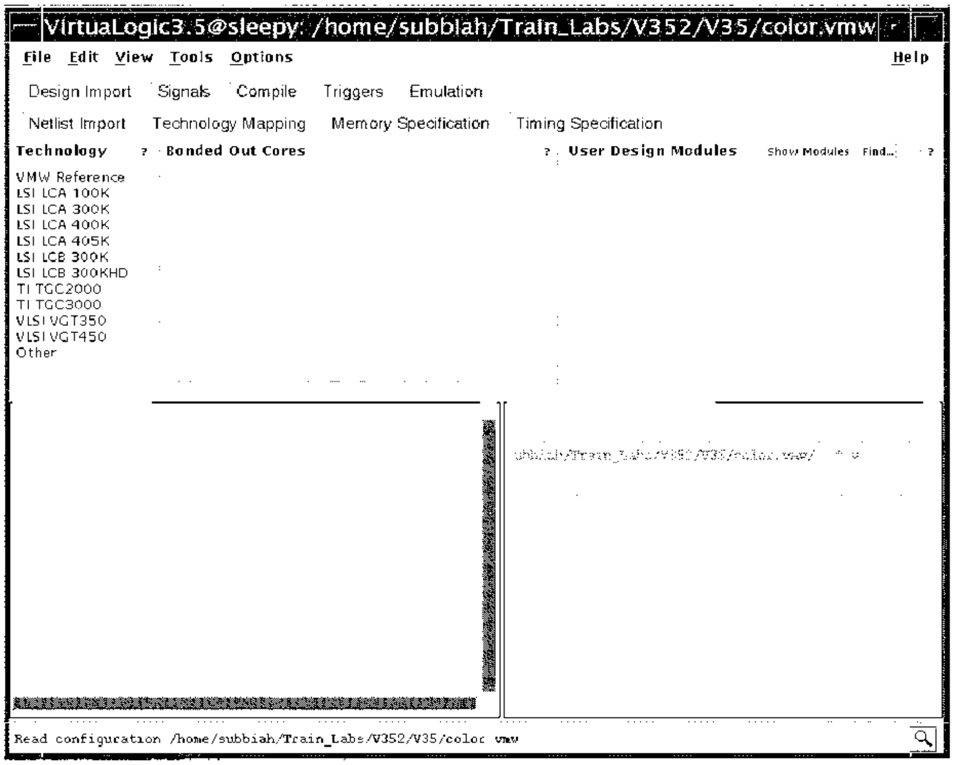
Root module

The *Root Module* frame specifies the name of the top-level module in the user's design. Type the name of the root module directly or select the module from a list of the candidates by pressing the *List Modules* button. The netlist will then be loaded into memory and scanned for candidate memory modules. Select the top-level module by clicking the left mouse button.

Technology mapping

The *Technology Mapping* frame allows the user to specify the library or libraries to use for the design and export any modules in the netlist that will be implemented outside of the emulator.

The *Technology Mapping* window is shown in *Figure 15 on page 74.*


VirtuaLogic3.5@sleepy: /home/subbiah/Train_Labs/V352/V35/color.vmw
File Edit View Tools Options Help
Design Import Signals Compile Triggers Emulation
Netlist Import Technology Mapping Memory Specification Timing Specification
Technology ? Bonded Out Cores ? User Design Modules Show Modules Find... ?
VMW Reference
LSI LCA 100K
LSI LCA 300K
LSI LCA 400K
LSI LCA 405K
LSI LCB 300K
LSI LCB 300KHD
TI TGC2000
TI TGC3000
VLSI VGT350
VLSI VGT450
Other

Read configuration /home/subbiah/Train_Labs/V352/V35/color vmw

Figure 15 Technology mapping

Technology

The *Technology* frame lists the technologies directly supported by VirtuaLogic. The technology mapping library, commonly referred to as the library, is a structural Verilog netlist that maps vendor or customer primitive cells to VirtuaLogic primitives, letting the

VirtuaLogic Compiler understand the functionality of the design. The library is the lowest level of hierarchy that can be observed in the *VirtuaLogic Browser* or probes. The contents of library cells are not visible to the user.

Select the library technology of the input netlist. The technologies directly supported are listed in the pane. Other technologies can be used by selecting *Other* and then supplying additional Verilog files that map the primitives used by the netlist into the VirtuaLogic reference library.

When the user clicks *Other*, the *Technology Files* frame is turned on; therefore, the files can be listed by typing their pathnames, double-clicking in the file browser, etc.

If VirtuaLogic does not support a desired technology (not on the menu), create a Verilog technology mapping model file. If help is required, contact an IKOS support person.

The library can contain any syntactically legitimate construct in the VirtuaLogic structural Verilog subset. It can also contain any valid Verilog behavioral code as long as the code is used exclusively for verification purposes and it does not contain semantic content. Refer to page 325 for details.

Bonded out cores

The list of *Bonded Out Cores* identifies module names that are implemented outside the emulator. Their I/Os are routed to the emulator I/O terminals. If there are structural contents for the *Bonded Out Core* in the Verilog netlist, they are ignored.

Select the set of *Bonded Out Cores* by pressing the button labelled *Show Module (for Drag and Drop to Bonded Out Cores)*. This will scan the design and put all the module names in the list below the button. Select the names with the middle mouse button and drag and drop them to the *Bonded Out Core* pane. Also, the names can be typed directly by clicking the mouse in the pane.

Module exporting allows the user to specify hierarchical blocks that are inside the design, but will be implemented outside the emulator.

Use this functionality for any of the following reasons:

- To identify a bonded core for some internal element. For example, an ARM processor implemented on the target system instead of inside the emulated design
- To remove some nonemulatable element from the design, such as a DAC from a video chip
- To facilitate splitting a design across more than one emulator

Module exporting has the following results:

- Removes the instance of a module and all its contents from the emulator.
- Adds top-level I/Os to the emulation model which corresponds to the ports of the exported module.

Instance removal example

Figure 16 on page 76 shows the removal of an instance of a module from the emulator and the addition of top-level design terminals to connect the interface to the emulator. Inputs of the exported instance become top-level outputs of the emulator, and outputs of the exported instance become inputs to the emulator.

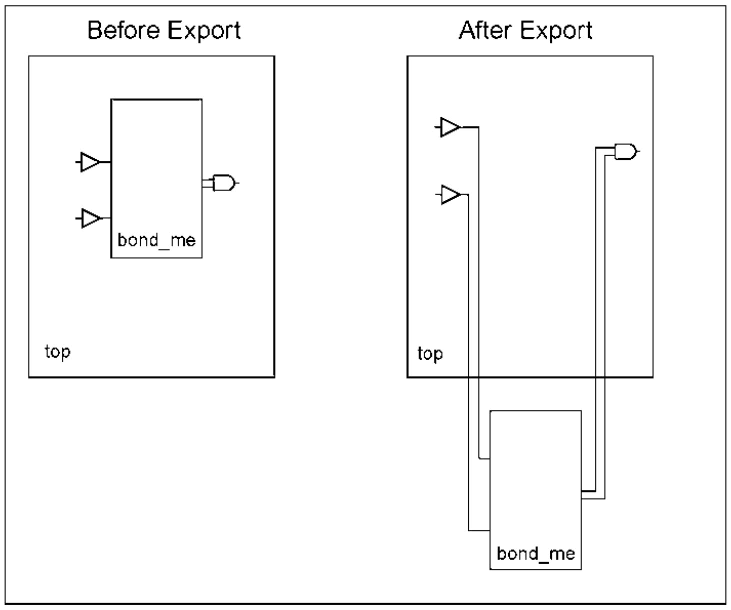


Figure 16 Instance removal

Technology files

The *Technology Files* pane is used to explicitly list library files when the desired implementation library is not yet directly supported by VirtuaLogic.

Type the full pathname of the Verilog library filename here, use the file browser beneath to double-click on the desired filename, or drag and drop with the middle mouse button.

Memory specification

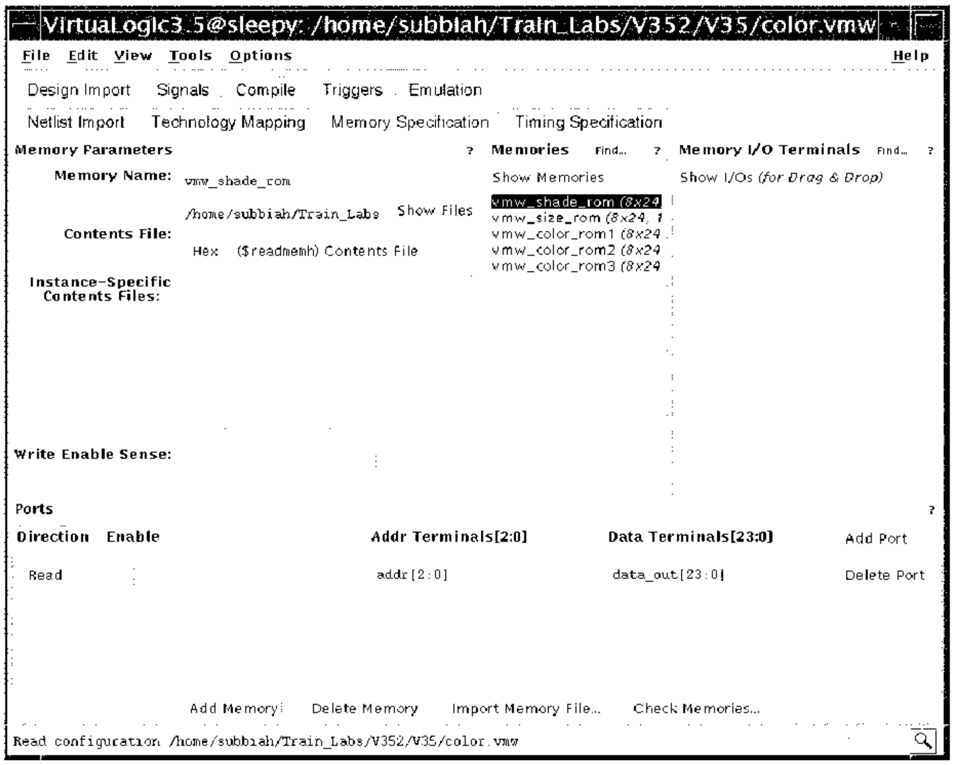
The *Memory Specification* window is used to specify the behavior of RAMs and ROMs in the design. Memories that can be described using this window have the following characteristics:

- Fully encoded addresses
- Unidirectional data ports (read or write, not read/write)
- Single bit, full word write enables
- Optional single bit, full word read enable

Memories that do not have these characteristics require the creation of a wrapper or shell netlist which adds gate-level constructs to map a memory cell into the emulator. Some examples of memories that would require a shell netlist are as follow:

- A memory with a single address that goes to a read and write port
- A memory with a write enable and a clock
- A memory with separate byte enables for each byte of the memory

Figure 17 on page 78 shows the *Memory Specification* window.


File Edit View Tools Options Help
Design Import Signals Compile Triggers Emulation
Netlist Import Technology Mapping Memory Specification Timing Specification
Memory Parameters ?
Memories Find... ?
Memory I/O Terminals Find... ?
Show Memories
Show I/Os (for Drag & Drop)
Show Files
Hex ($readmemh) Contents File
Ports ?
Add Port
Delete Port
Add Memory
Delete Memory
Import Memory File...
Check Memories...

VirtuaLogic3.5@sleepy: /home/subbiah/Train_Labs/V352/V35/color.vmw

Memory Name: vmw_shade_rom

Contents File: /home/subbiah/Train_Labs

Instance-Specific Contents Files:

Write Enable Sense:

vmw_shade_rom *(8x24*
vmw_size_rom *(8x24, 1*
vmw_color_rom1 *(8x24*
vmw_color_rom2 *(8x24*
vmw_color_rom3 *(8x24*

Direction	Enable	Addr Terminals[2:0]	Data Terminals[23:0]
Read		addr[2:0]	data_out[23:0]

Read configuration /home/subbiah/Train_Labs/V352/V35/color.vmw

Figure 17 Memory specification

Memory parameters

The *Memory Parameters* pane includes the following: *Memory Name, Contents File, Instance-Specific Contents Files,* and *Write Enable Sense*.

The most common mistake in defining the memories is to define the write or read enable sense incorrectly.

Memory name

The *Memory Name* frame is a text field. Simply type in the memory name or click on the memory name appearing in the *Show Memories* box.

Contents file

The *Contents File* is used to preload memory with data before emulation begins. It is required for ROMs (memories without write ports) and is optional for RAMs. The file name specified here is the default for all instances of the memory.

To specify initial contents for a memory, enable the *Contents File* text field by typing the path or using the *Show Files* button to the right of the text field.

The memory contents are formatted in hex or binary and support the same file formats as the Verilog-XL commands. For example,

```
$readmemh
```

and

```
$readmemb
```

Instance-Specific contents file

The *Instance-Specific Contents Files* frame is used to override the *Contents File* for a specific memory instance. The syntax to use is as follows:

```
<format> contents_file_pathname <instance_path>
```

where *format* is hex or binary

For example,

```
bin ./data/p0_0.dat box0.J111.slice_0.rom
bin ./data/p0_1.dat box0.J111.slice_1.rom
```

If a memory is instantiated more than once in a design, and each instance of that memory has unique contents, the *Instance-Specific Contents Files* field is used to specify the data format, instance name, and contents files for each instance of the memory.

The user can enter as many filename/instance pairs as needed, one pair per line. The file names can be typed or dragged and dropped from the *Show Files* button. The instance pathnames can be obtained from the *Browse* button.

Write enable sense

The *Write Enable Sense* specifies whether the write enable is *edge* sensitive, *level* sensitive, or logic sensitive. The *Write Enable Sense* pull down selects one polarity for all read and write enables of a memory. Memories with both high and low enables must be specially modeled. The available choices and their behaviors are as follow:

Rising Edge

- write port: The memory will write the data on the write port data terminals into the address on the write port address terminals at the rising edge of the write enable
- read port with output enable: The read data will be driven while the output enable is high and tristated when the output is low

Falling Edge

- write port: The memory will write on the falling edge of the write enable signal
- read port with output enable: The read data will be driven when the output enable is low and tristated when the output enable is high

Active High

- write port: The memory will write the data on the write port data terminals, into the address on the write port address terminals, while the write enable is high
- read port with output enable: The read data will be driven while the output enable is high and tristated when the output is low

  Writes occur before reads; therefore, if an address is written and read at the same time, the data being written will also be read

Active Low

- write port: The memory will write while the write enable is low
- read port with output enable: The read data will be driven when the output enable is low and tristated when the output enable is high

  Writes occur before reads; therefore, if an address is written and read at the same time, the data being written will also be read

Active High Unordered and Active Low Unordered

The unordered enables behave the same as the Active High and Active Low selections, with the exception of the guaranteed write before read behavior. Unordered enables should not be used unless recommended by a member of the IKOS technical staff.

Note that races between the write enable, the write address and data are not physically possible in the VirtuaLogic emulator. If data and address change on the clock edge at the same moment when a level sensitive write enable becomes active, the new values of data and address will be used in the write transaction.

Memories

The *Memories* list shows the module names of each type of memory, address and data widths, and the number of read and write ports. Initially, the *Memories* pane lists only ***Unnamed Memory** Empty.*

Show memories

The *Show Memories* lists the current valid choices of memories and also the bonded out cores and empty modules. Click the button to display the list of current memories. Select one by one of the non memories and delete it before proceeding to the timing specification.

Defining memories with netlist prototypes

Initially, this list will be empty. The easiest way to define the memory types is to include Verilog module definitions (prototypes) for them in the netlists. Selecting the *Show Memories* button brings in all the user netlist modules that have no structural contents (nets or submodules) defined. These are considered *memory candidates* (Verilog design modules with no structural contents not already defined as memories). If any of the modules listed are not memories, select them with the left mouse button, one at a time, and delete them with the *Delete Memory* button.

For each memory module, the user must fill it out with the following procedure:

1. Select it with the left mouse button.
2. Select the *Add Port* button enough times to have the desired number of ports.

3. Change the direction of the write ports by clicking the port's *Direction* menu and selecting *Write*.

4. If any of the read ports have an output enable, click the *small box* next to the enable field of that port.

5. Follow the directions in the section, *Memory I/O terminals on page 82*.

Defining memories without netlist prototypes

Do the following for each memory:

1. Click in the *Memory Name* field and type in the name of the memory module.

2. Select the *Add Port* button enough time to have the desired number of ports.

3. Change the direction of the write ports by clicking that port's *Direction* menu and selecting *Write*.

4. If any of the read ports have an output enable, click the *small box* next to the enable field for that port.

5. Click in the port field and type in the names of the appropriate terminals. Verilog vector syntax can be used and multiple scalars can be placed on this line, separated by spaces. In many cases, sets of scalars can be easily entered using a *synthetic* vector. For example,

```
a4 a3 a2 a1 a0 can be represented as a<4:0>
\a[4] \a[3] \a[2] \a[1] \a[0] can be represented as \a[<4:0>]
a_4_ a_3_ a_2_ a_1_ a_0_ can be represented as a_<4:0>_
```

This can greatly reduce the amount of typing required to enter in memory specifications. This syntax can also be used in the timing specification and is used in the *Virtual Browser*.

Memory I/O terminals

The *Memory I/O Terminals* frame is used as a source for dragging and dropping terminal names into the *Data Terminals[?:0]*, *Addr Terminals[?:0]*, and *Enable* fields.

Following are the instructions:

1. Make sure the *Netlists*, *Netlist Defines*, and *Root Module* have been filled out in the *Netlist Import* window.

2. Select the *Show I/Os (for drag and Drop)* button. If the graphical interface has not read in the netlist yet, it will read it now. If the netlist is very large, this may take some time. If you prefer not to wait and know the memory's I/O terminals, you can type them by clicking the left mouse button in the appropriate field.

3. Select the address terminals by pressing the left mouse button and dragging the mouse over that data terminals, and then release the button. If the address terminals are not adjacent to each other in the list, then repeat the process to pick-up the ones missed the first time or hold down the *control* key while clicking the mouse to add new entries to the selected set. Normally this will not be necessary as the related terminals will usually be adjacent in the list, in fact they will often be represented as a vector or a synthetic vector.

4. Press the middle mouse button over the selected terminals and while the middle mouse button is depressed, move the mouse to the address field. Then release the mouse button.

5. The terminal names dragged will be removed from the *Memory I/O Terminals* list.

6. Repeat the process for the data terminals and enable terminals.

Ports

VirtuaLogic supports memories with an arbitrary number of ports. When describing a memory, specify the *Direction* (read or write), *Enable*, *Address*, and *Data* terminals.

The user interface comes up with one blank port. For each additional port in the memory (a RAM will have at least two), select the *Add Port* button.

Do the following for each port:

1. Choose whether it is a read or a write port by clicking on the option menu in the *Direction* column. Read is the default.

2. Read ports have an optional *Enable* signal. For a read port, ungray the field by clicking in the small *box* to the left of it. On a color system, this will turn the box from gray to pink. For write ports, the enable signal is required.

3. Fill in the name or the *Enable*, *Address*, and *Data* terminals. If you have already specified the memory name and there is a Verilog prototype file for it among the design files, then use the *Memory I/O Terminals* list to drag and drop the terminal names. Otherwise, type the names.

Vectors

The fastest way to enter in the address and terminal names is with vector syntax. Obviously, this is easy to do if the netlist uses vectors. For example, an 8-bit address might have a terminal name of addr[7:0]. This can be typed directly into the *Addr Terminals[?:0]* field for a port or drag and drop them from the *Memory I/O Terminals* frame.

However, the user interface tries to let the user do this even if the Verilog netlist has had its vectors flattened with a synthesis tool, and in some cases, if it was hand written using scalar, say, to conform to a memory element in an ASIC library.

For example, the user interface uses textual pattern matching to group together terminals that look like they ought to be part of the same vector because they are escaped vector slices (e.g., \addr[<n:0>]), or if there are several terminal names with a common prefix followed by a number (e.g., addr0, addr1 ...). The escaped vector slices will be collapsed into standard vector notation (e.g., \addr[<n:0>]), while the similar looking scalar names will get a new notation, using angle brackets, called *synthetic vectors* (e.g., addr<7:0>).

The *Memory I/O Terminals* will vectorize as much as possible To deal with scalars individually, double click on a vector name in the list.

The vector notation can be typed directly into the terminal field or directly into the memory file (*$CONFIG/vmw.mem*) using a text editor.

Scalar

If for some reason using vectors does not work, the user must fill out the *Address* field and *Data* field by entering each scalar, separated by spaces (e.g., a7 a6 a5 a4 a3 a2 a1 a0).

Output enable sense

The *Output Enable Sense* is used for read ports that are given an enable terminal. It specifies active low or active high. Output enables are always level sensitive.

Add memory

The *Add Memory* button is used to add a new memory to the configuration.

Delete memory

The *Delete Memory* button deletes a highlighted memory from the *Show Memories* frame.

Import memory file

The *Import Memory File* button is used to specify a new memory file.

Typically, this will be found as *vmw.mem* in another configuration directory (*extension.vmw*). Note that this will add to the existing memories.

The contents of this file will be read into the memory editor and will be written to *vmw.mem* in the current configuration the next time it is saved.

Note that the semantics of the file management differs between clock file, memory files, trigger files, and probe files. Refer to the sections below for details.

Memory files

In memory files , two memory files can be sensibly combined into a larger one. It is usually not recommended to swap between multiple memory files and *gvl* always uses *vmw.mem* as the name of the active memory file. The only file management the user is expected to do is to merge in a memory file from another configuration.

Check memory

The *Check Memory* button is used to check that the memories are consistently specified (e.g., port widths match).

Memory example

The following example illustrates the production of a memory specification for a three-port memory named *RegFile* which has two read ports, one write port, and 256 32-bit-wide memory words.

Figure 18 on page 86 shows a schematic of the module.

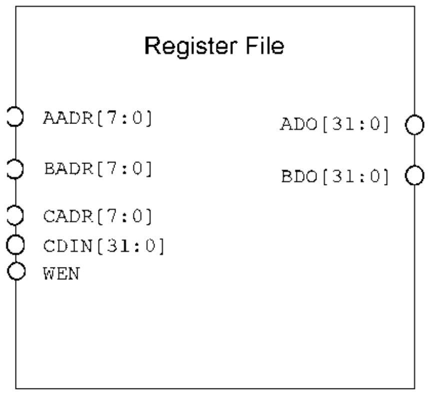


Figure 18 RegFile schematic

The schematic representation uses the following:

- AADR and ADO are the address and data terminals for the first read port
- BADR and BDO are the address and data terminals for the second read port
- CADR, CDIN and WEN are the address, data, and write enable terminals for the write port
- WEN is level-sensitive, active low

Adding the memory

To add memory, click in the *Memory Name* text field, and type in *RegFile* (memory name). Each time the *Add Memory* is selected, a blank port form appears.

Adding the port information

To add the port information:

Step	Action	Notes
1	Under Direction, select Read	First port is a read port
2	Click in the Addr Terminals text box, and type AADR[7:0]	The address terminals for this port are named ADDR[7:0]
3	Click in the Data Terminals text box, and type ADO[31:0] into the data name field	The data terminals for this port are named ADO[31:0]
4	Click on Add Port	Add a port

5	Repeat steps 2-4 for the second read port	
6	Repeat steps 2-4 but click Write	This port is a write port
7	Click in the Enable box and type **WEN**	The write enable is named WEN

Adding the memory parameters information

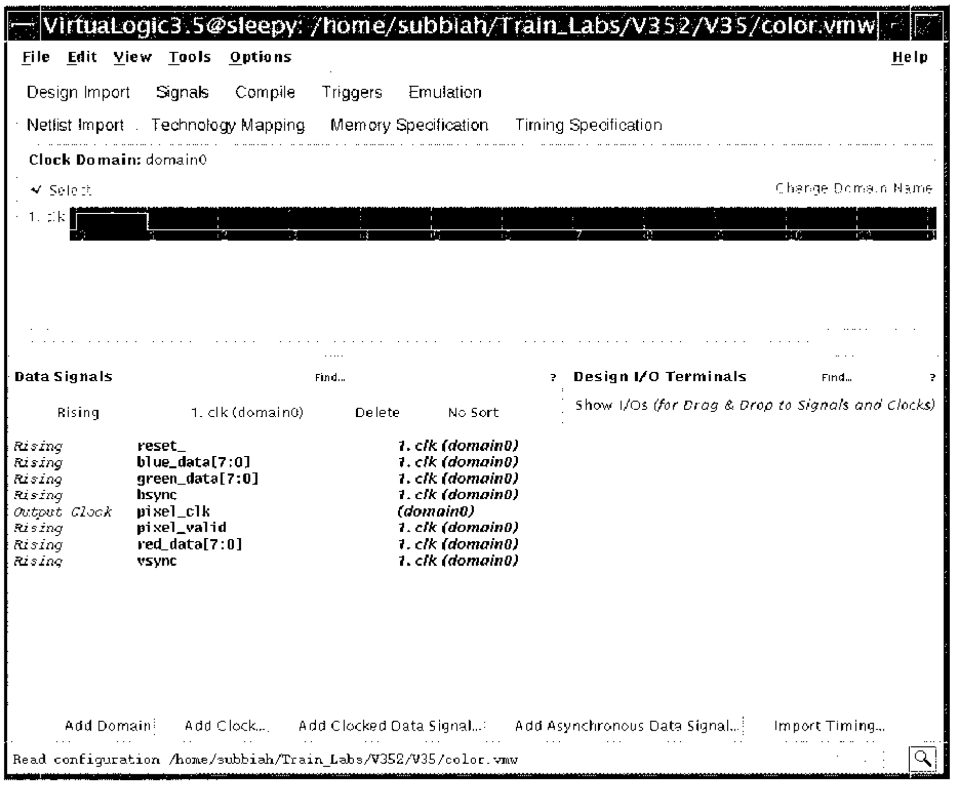
To add memory parameter information:

Step	Action	Notes
1	Ignore the Contents File field	RegFile is a memory with no required initial contents
2	Select Active Low from the Write Enable Sense menu	RegFile is write enable which is active low

Timing specification

The *Timing Specification* frame enables the user to produce a file with information about the timing behavior of clock signals and data signals from which the compiler can determine the timing behavior internal to the ASIC. When creating a timing specification, the user must classify each primary input to the chip as a clock signal or data signal and define its behavior.

Figure 19 on page 88 shows the *Timing specification* window.


VirtuaLogic3.5@sleepy: /home/subbiah/Train_Labs/V352/V35/color.vmw
File Edit View Tools Options Help
Design Import Signals Compile Triggers Emulation
Netlist Import Technology Mapping Memory Specification Timing Specification
Clock Domain: domain0
Select
Change Domain Name
1. clk
Data Signals
Find...
Design I/O Terminals
Find...
Rising
1. clk (domain0)
Delete
No Sort
Show I/Os (for Drag & Drop to Signals and Clocks)
Add Domain
Add Clock...
Add Clocked Data Signal...
Add Asynchronous Data Signal...
Import Timing...

```
Rising        reset_              1. clk (domain0)
Rising        blue_data[7:0]      1. clk (domain0)
Rising        green_data[7:0]     1. clk (domain0)
Rising        hsync               1. clk (domain0)
Output Clock  pixel_clk           (domain0)
Rising        pixel_valid         1. clk (domain0)
Rising        red_data[7:0]       1. clk (domain0)
Rising        vsync               1. clk (domain0)
```

```
Read configuration /home/subbiah/Train_Labs/V352/V35/color.vmw
```

Figure 19 Timing specification

Clock domain

The *Clock Domain* frame is used to specify the design clock waveforms. In particular, the VirtuaLogic compiler needs to understand the relative order of transitions between all the design clocks. In creating a timing specification, the user must classify each primary input to the IC as a clock signal or data signal and define its behavior.

A clock is an input signal whose transitions cause state elements to change state. Clock signals include the following:

- Signals driving flip-flop clock pins, latch enable pins, or memory write enable pins
- Asynchronous set or reset to a clocked element (latch or flip-flop)
- Combines with other signals to become a clock

Clock signals are grouped into clock domains when the signals have integrally related frequencies and fixed, specified phase-relationships (i.e., these frequencies must be integral multiples of some common base frequency, and the frequency of one of the clocks must be the base frequency).

A clock domain is a collection of the following:

- Phase-locked clock signals with a known phase-relationship
- The logic and state elements in the design which are synchronous to any of these clock signals

Individual clocks can be added to the clock list in the following two ways:

1. Drag and drop clock terminal names from the *Design I O Terminals* frame by clicking on the *Show I O* button. If the user has entered the netlists and root module name, the tool will read in the netlist and display the top-level design I/Os. Drag and drop the clocks from the terminals list by using the middle mouse button.

2. Select the *Add Clock* button. A window will pop-up for entering the clock name. If it is an internally generated clock, for example from a phase lock loop, then the user can specify a hierarchical name. The VirtuaLogic circuitry will automatically generate the clock waveform specified, relative to the external clocks. The *Add Clock* button can be used if reading the netlist would take too long or if you want to specify the clocks before the netlist is completed.

Note that it is easier to add all the clocks at once and then specify the behavior of each, than to complete each clock in succession.

Once the clocks are added, the waveform can be edited by direct manipulation (horizontal and vertical dragging) with the left mouse button or using the buttons on the right to add or remove periods and invert the whole waveform.

The important factor is the relative order of transitions of the clock, not the number of time units (vertical gray lines) consumed by a clock waveform

It is easier to understand compiler messages if the clock domains have meaningful names. To name a clock domain, click the *Change Domain Name* button in that domain. Then enter the desired name in the dialog box.

There is no limit on the number of clocks within a domain and a domain can include external clock signals and internal clock signals, as long as all the signals have a fixed phase-relationship.

External clocks include all design inputs which carry signals that trigger state changes within the state elements of the design which include the following:

- Periodic signals used for flip-flop clocks or latch gates
- Aperiodic signals, such as system resets, that are applied to asynchronous preset or clear terminals of state elements

The VirtuaLogic compiler can derive the timing of internal clock signals produced from external clock signals with acyclic combinatorial circuits without user input. The compiler cannot derive the timing of some internal clock signals, such as those produced by an internal phased-lock loop or other cyclic or analog clock generation circuits. Periodic clock outputs from such circuits require description in the timing specification just as external periodic clock signals.

All clocks must have an integral frequency relationship to some base clock. The VirtuaLogic compiler synthesizes logic which uses the base clock; therefore, the target system must supply it to the emulator, even if the design does not use it. For example, if the design has clock inputs with a 3:2 ratio, such as 75 MHz and 50 MHz, it is necessary to supply the emulator with the base clock which is the lowest common integral factor of all clocks in a clock domain. In this example, the base clock frequency would be 25 MHz.

Building the waveforms

As each clock is added, a waveform appears on the screen next to each clock name. Waveforms reflect the behavior of the clocks and must show the order of the edges.

The clock waveforms show the number of edges in each clock and the edge ordering between clocks. A pair of edges can have one of three orderings: Before, Same Time, and After. If the edges line up, they happen at the same time otherwise they happen in the indicated order. The important factor is the ordering or coincidence of edges relative to each other, not the actual spacing between the edges

Create up and down segments to represent a clock edge. Again, use one of two methods:

- Click on the boxes at the end of each line
    - **!** (inverts all pulses, reversing rises and falls).
    - + (adds a pulse)
    - **-** (deletes the last pulse)
- Position the cursor on a line until the cursor turns into an arrow. Depress the left mouse button, and while holding it down, move the mouse up. A pulse, the width of one segment, appears on the screen. To make a double length pulse, push up the next segment or pull on the vertical edge of the pulse. Move vertical segments (rise and fall) left or right in the same fashion.

NOTE

only draw enough edges to represent one complete cycle of the slowest clock in the domain.

Multiple domains

The VirtuaLogic system supports multiple clock domains within designs. While clocks within the same domain must have a known frequency and phase relationship, clocks within different domains are assumed to be completely asynchronous; that is, have no known frequency or phase relationship.

Some circuit forms that are supported within a single domain are not supported across multiple domains. Each state element in the design can make transitions in only one domain. You cannot clock a state element with a combination of signals from different domains.

Figure 20 on page 92 shows a circuit form, supported when all signals are in the same domain, but unsupported when signals are from different domain.

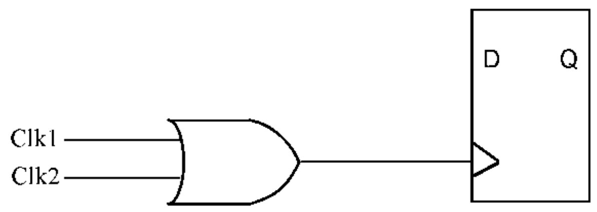


Figure 20 Unsupported circuit when Clk1 and Clk2 are in different domains

Beyond the restriction on creating clock signals by gating together signals from different domains, there is no further restriction on the circuit forms supported for multiple domains.

Data signals

The *Data Signals* frame is used to specify the relative timing of data signals to design clock edges. VirtuaLogic requires this information to analyze the timing of each signal in the design, and perform timing resynthesis to achieve robust emulation results and efficient resource utilization via time multiplexing.

Data signals are defined in terms of what causes them to change on the target system. Typically this is a clock, but it could also be another data signal. Each data signal can have a relationship with only one clock in one domain. All data signals are assigned to a domain because each data signal has a fixed relationship to a clock (or multiple clocks) in the same domain.

Data signals are signals which are not clocks or asynchronous reset signals. They include the following:

- The collection of signals connected to D and Q terminals of flip-flops and latches
- The address and data terminals of memories
- Any signals from which other data signals, in combination, are derived

Each synchronous signal transitions (for design inputs) or is sampled (for design outputs) on the rising or falling edge of one or more design clocks. Once a data signal is added (by dragging and dropping from the *Design I O Terminals* or by selecting the *Add Clocked Data Signals* button), the user can select the clock and edge on which its timing is based.

Signals that can be transitioned or sampled on more than one clock or on both edges of a clock are handled by adding the signals to the list multiple times.

Asynchronous signals are entered by selecting the *edge Asynchronous* as opposed to *Rising* or *Falling*.

Signals are not allowed to transition on clocks of more than one domain.

Rising

The *Rising* button is used to indicate the direction of the transition to which the data signal is externally timed. The default is *Rising*. Click on this button for the following selection:

- Rising (selects rising edge timing for inputs, output, and bidirects)
- Falling (selects falling edge timing for inputs, outputs, and bidirects)
- Both (selects both edge timing for inputs, outputs, and bidirects)
- Asynchronous (selects asynchronous timing for inputs only)
- NoConnect (ties inputs to ground and disconnects outputs)
- One (Tie-High) (ties inputs high)
- Zero (Tie-Low) (ties inputs low)
- Feedthrough (defines feedthrough pair)
- Feedthrough* (defines many feedthrough groups)

Rising edge synchronous inputs

Describing an input as rising edge synchronous indicates that the input will only change value as a result of the rising edge of the clock on the target system. The specification of an input as rising edge synchronous describes the target system timing, not the timing of the emulated design as shown in *Figure 21 on page 94*.

The emulator samples each synchronous signal input some time after each edge specified in the signal's timing specification. This sample occurs at least 100 ns after the edge and 50 ns before the next edge in the signal's timing specification as shown in *Figure 21 on page 94*.

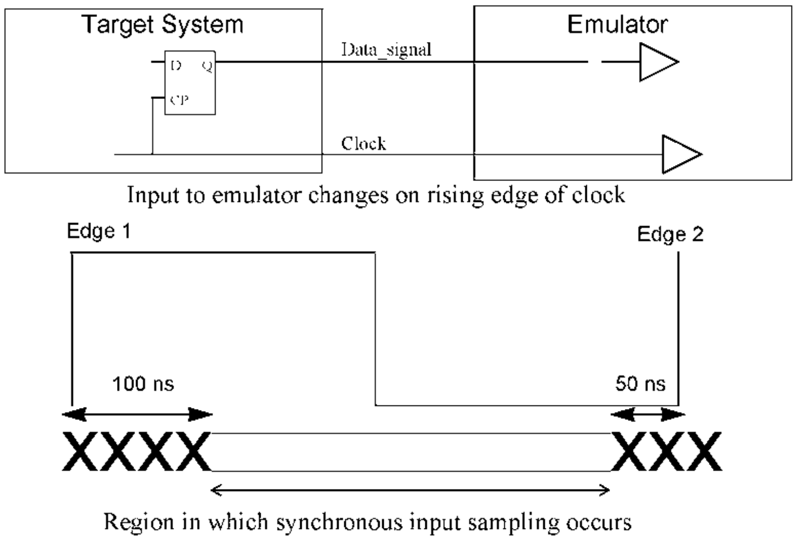
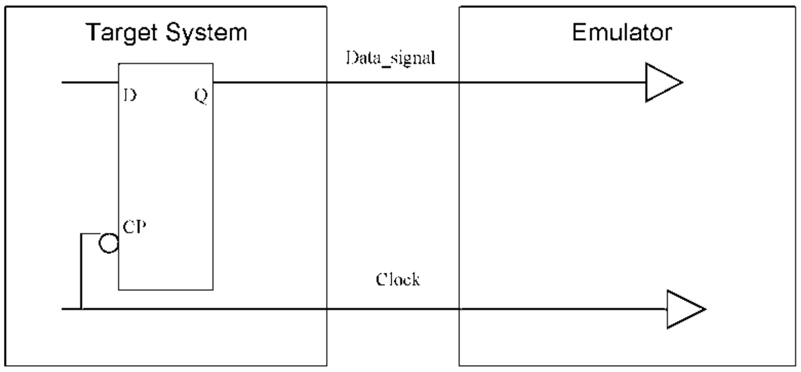

Target System
D
Q
CP
Data_signal
Clock
Emulator
Input to emulator changes on rising edge of clock
Edge 1
Edge 2
100 ns
50 ns
XXXX
XXX
Region in which synchronous input sampling occurs

Figure 21 Rising edge synchronous inputs

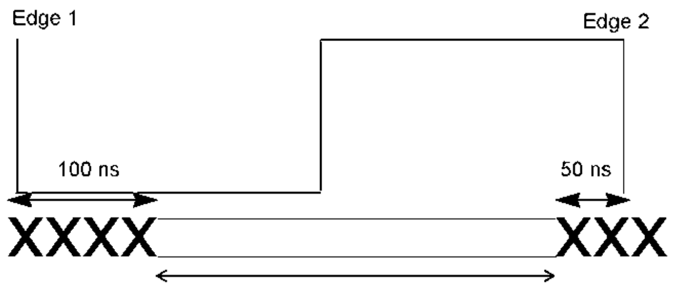
Falling edge synchronous inputs

Describing an input as falling edge synchronous indicates that the input will only change value as a result of the falling edge of the clock on the target system. The specification of an input as falling edge synchronous describes the target system timing, not the timing of the emulated design as shown in *Figure 22 on page 95.*

The emulator samples each synchronous signal input some time after the falling edge of the specified clock. This sample occurs at least 100 ns after the edge and 50 ns before the next edge in the signal's timing specification as shown in *Figure 22 on page 95.*



Input to emulator changes on falling edge of clock



Region in which synchronous input sampling occurs

Figure 22 Falling edge synchronous inputs

Both edge synchronous inputs

Describing an input as both edge synchronous indicates that the input will only change value as a result of an edge of the clock on the target system. The specification of an input as both edge synchronous describes the target system timing, not the timing of the emulated design as shown in *Figure 23 on page 96*.

The emulator samples each synchronous signal input some time after an edge of the specified clock. This sample occurs at least 100 ns after the edge and 50 ns before the next edge in the signal's timing specification as shown in *Figure 23 on page 96*.

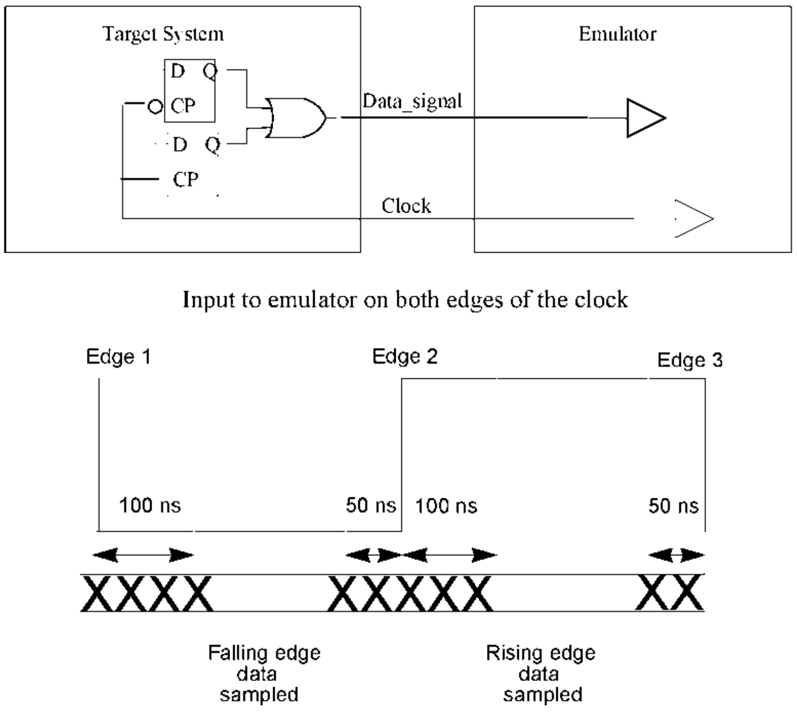
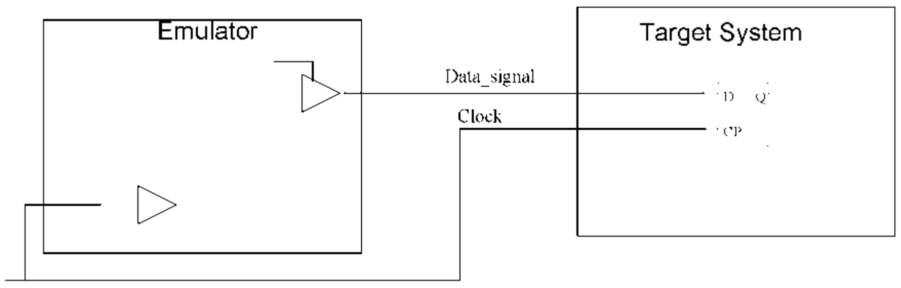


Figure 23 both edge synchronous inputs

Rising edge synchronous outputs

Describing an output as rising edge synchronous indicates that the target system will sample the data on the rising edge of the specified clock. The specification of an output as rising edge synchronous describes the target system timing, not the timing of the emulated design as shown in *Figure 24 on page 97*.

The emulator produces a correct output value on each synchronous output signal so each edge in the signal's timing specification can sample the signal. The system produces correct output values with at least 100 ns setup and 50 ns hold time to all clock edges listed in the timing specification and shown in *Figure 24 on page 97*.



Target system captures data on rising edge of clock

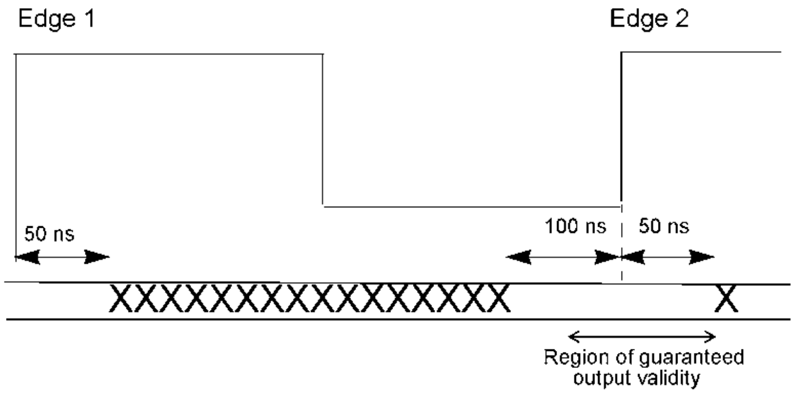


Figure 24 rising edge synchronous outputs

Falling edge synchronous outputs

Describing an output as falling edge synchronous indicates that the target system will sample the data on the falling edge of the specified clock. The specification of an output as falling edge synchronous describes the target system timing, not the timing of the emulated design as shown in *Figure 25 on page 98*.

The emulator produces a correct output value on each synchronous output signal so each edge in the signal's timing specification can sample the signal. The system produces correct output values with at least 100 ns setup and 50 ns hold time to all clock edges listed in the timing specification and shown in *Figure 25 on page 98*.

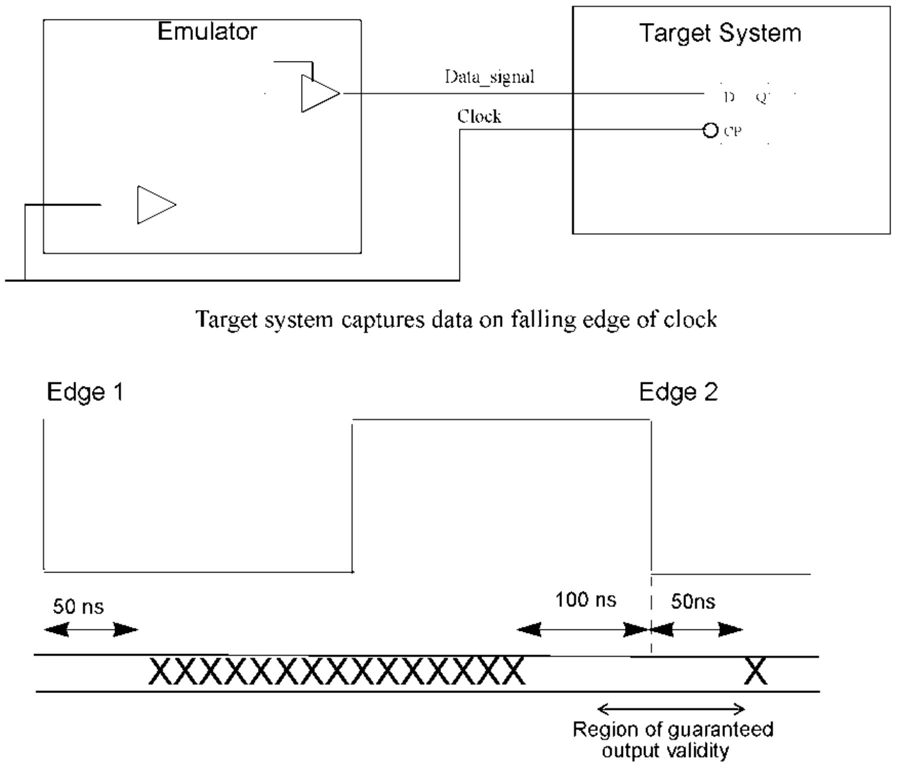
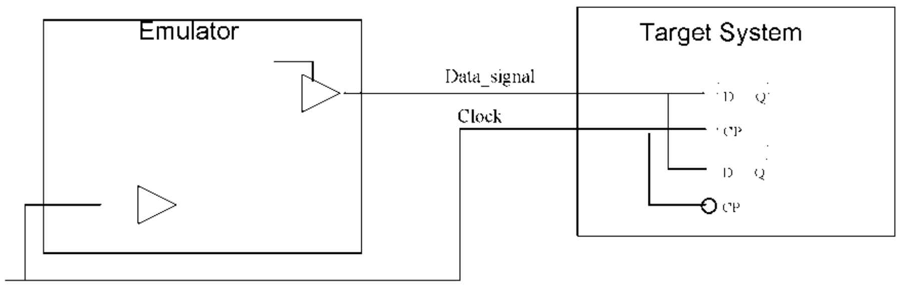

Emulator
Target System
Data_signal
Clock
D
Q
CP
Target system captures data on falling edge of clock
Edge 1
Edge 2
50 ns
100 ns
50ns
XXXXXXXXXXXXXXX
X
Region of guaranteed output validity

Figure 25 Falling edge synchronous outputs

Both edge synchronous outputs

Describing an output as both edges synchronous indicates that the target system will sample the data on the both edges of the specified clock. The specification of an output as both edges synchronous describes the target system timing, not the timing of the emulated design as shown in *Figure 26 on page 99.*

The emulator produces a correct output value on each synchronous output signal so each edge of the specified clock can sample the signal. The system produces correct output values with at least 100 ns setup and 50 ns hold time to each clock edges and shown in *Figure 26 on page 99.*



Target system captures data on both edges of clock

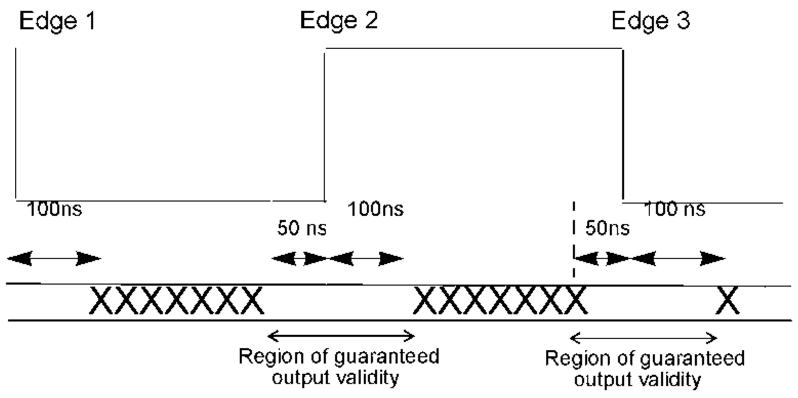

Edge 1
Edge 2
Edge 3
100ns
50 ns
100ns
50ns
100 ns
XXXXXXX
XXXXXXX
X
Region of guaranteed output validity
Region of guaranteed output validity

Figure 26 Both edges synchronous outputs

Asynchronous inputs

Asynchronous inputs from the target system into the emulated design fall into two categories: inputs driving preset and clear terminals of state elements, and pure data inputs. These categories are handled differently and are described below.

Asynchronous preset and reset signals

Like clocks, asynchronous inputs that go to the preset and clear terminals of state elements cause the storage element to change state. Like clocks, these asynchronous resets must be driven on the clock cable of the emulator. Unlike clocks, the user does not have to specify edge behavior.

All state elements reset by an asynchronous input will be reset at the same time, no races are possible.

For asynchronous reset signals there is an exception to the rule that state elements make transitions only as a result of signals from a single domain. Asynchronous external signals or signals from a domain other than that of the clock or gate of the state element can drive asynchronous reset signals which are connected to asynchronous clear or set terminals of state elements. The VirtuaLogic compiler supports these signals.

Exercise care when using external asynchronous reset signals which are not driven from the same clock domain as a clock signal. This creates a potential race condition. The VirtuaLogic emulator whenever a clock and asynchronous reset signal change within one vclock period of one another. Internal synchronizers within the emulator resolve the race and all state elements see the same ordering of the edges.

In your design, the time window within which edges on a clock and asynchronous reset signals constitute a race condition is much smaller, and state elements do not necessarily resolve the race consistently.

Unlike the physical implementation of your design, if multiple state elements in the emulator are simultaneously exposed to clock and reset, all see the same ordering of these edges. In your design, when clock and reset edges are nearly coincidental, some state elements might determine that the clock edge preceded the reset edge while others might determine the opposite.

Asynchronous data signals

Signals from the target that do not change as a result of a clock edge can be defined as asynchronous data signals. There is no guarantee that all fanouts of an asynchronous data input will receive the data at the same time, or in the same user clock cycle. The amount of time it takes the asynchronous signal to propagate to different fanouts is based on the length of the path. Path length can change each time the design is compiled.

Unconnected inputs and outputs

In many cases, signals that exist on the emulated design will not be connected to the target system because that functionality will not be emulated. Test and analog I/O are examples of signals that will not be connected between the emulator and the target. These signals can be specified as NoConnect.

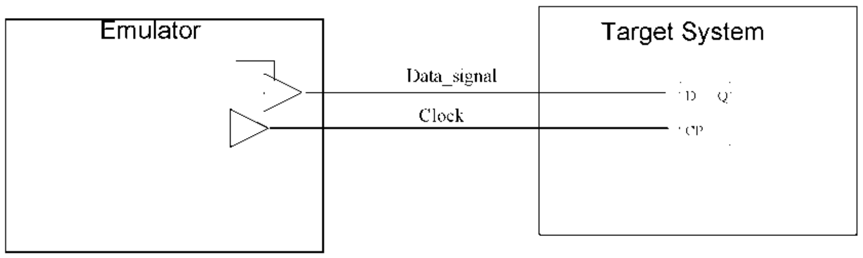
Outputs that are specified as NoConnect will not be implemented in the emulation model, and the logic driving these outputs will be deleted.

Inputs that are specified as NoConnect will be tied to ground. It is recommended that you explicitly tie off unused inputs. Inputs that should be tied to ground should be specified as zero and inputs that should be tied to VCC should be specified as one. This way the timing specification will be correct and self-documenting.

Output clocks on the target system

Outputs that trigger state changes on the target system have two special considerations. First, these signals must not have unexpected transitions which will cause false clocking. Second, these signals must be generated to provide proper setup and hold time relationships with data latched by these clocks. This type of signal must be specified as an output clock.

A signal specified as an output clock is generated at 100 ns before any other output in that clock domain. This guarantees that setup and hold requirements will be met at the target system which is shown in *Figure 27 on page 102*.



Emulator generates clock for target system

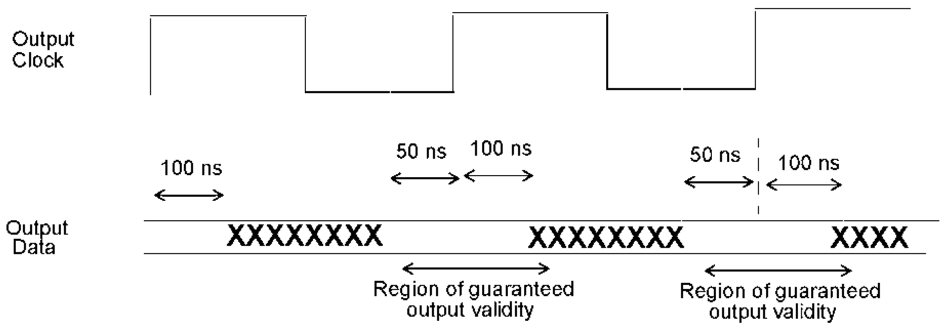
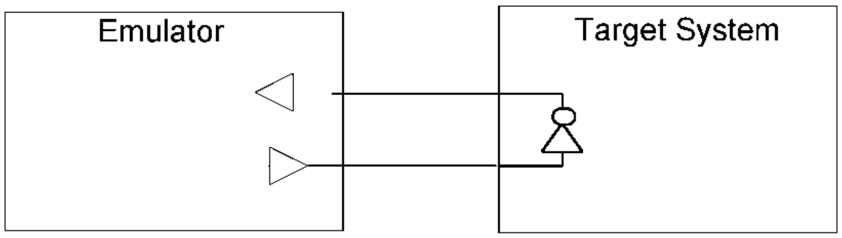

Output Clock
100 ns
50 ns
100 ns
50 ns
100 ns
Output Data
XXXXXXXX
XXXXXXXX
XXXX
Region of guaranteed output validity
Region of guaranteed output validity

Figure 27 Output clock

Inputs derived from outputs

Some inputs to the emulator change as a result of an output of the emulator changing, rather than due to a clock edge. A memory output enable is the most common example. Outputs and inputs connected via a communications path through the target system must be specified as a feedthrough pair.

When an input signal and an output signal are specified as feedthrough, the input signal will not be sampled until at least 50 ns after the output signal has changed as shown in *Figure 28 on page 103*.



Emulator input derived from emulator output

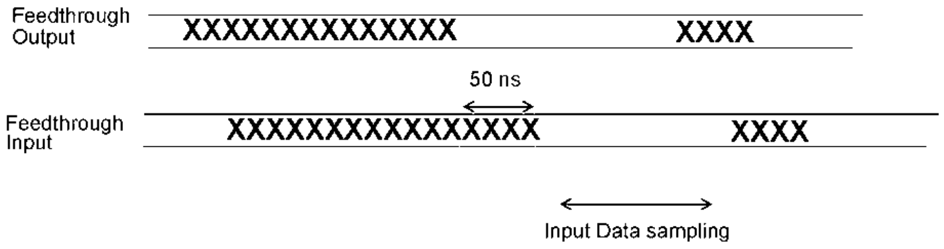


Figure 28 Feedthrough

Feedthrough signals

A feedthrough is a pair of pins, a signal from the emulator that passes through combinatorial logic on the target system and then passes back into the emulator. One pin relationship to another will be defined as a *feedthrough* and if busses or multiple signals have a feedthrough relationship, they will be specified as a *feedthrough**. The feedthrough or feedthrough* specification tells the emulator to sample the input based on changes in the output, rather than the clock. If the signal gets registered by a clock on the target system, then it must also be described as one of the other types. A signal can normally only have one specification, unless it is a feedthrough. In that case, a net can be both a feedthrough and a synchronous data signal.

A typical example of feedthroughs is a memory on the target system, where changes in the address of the memory cause changes in the data without the occurrence of a clock edge. Another example would be tri-state drivers implemented on the target system where an output from the emulator controls the OE pin of the tri- state and the output of the tri-state drives inputs to the emulator. The user can specify vectors for the source (emulator output) and destination (emulator input).

A feedthrough tells the compiler that an output signal leaves the emulated design and combinationally causes an input to the emulated design to change. A classic case is a memory output enable. If the emulated chip drives out an output enable that causes a RAM to start driving data to the chip, this is a feedthrough. If a feedthrough is not used here, the user will not get a correct circuit behavior.

To use a feedthrough in a clock file, the syntax is as follows:

```
Feedthrough <terminal> <terminal>
```

The order of the terminals is not important. Note that either or both of the terminals may be defined elsewhere in the clock file as rising, falling, etc.

Specifying the following:

```
Feedthrough outsig[2:0] insig[2:0]
```

is equivalent to specifying:

```
Feedthrough outsig[2] insig[2]
Feedthrough outsig[1] insig[1]
Feedthrough outsig[0] insig[0]
```

Feedthrough* signals are feedthroughs where every source signal is assumed to drive every destination signal. In this case, the vectors do not need to be the same length. Specifying the following:

```
Feedthrough* outsig[2] insig[2]
```

is equivalent to specifying:

```
Feedthrough outsig[2] insig[2]
Feedthrough outsig[2] insig[1]
Feedthrough outsig[2] insig[0]
Feedthrough outsig[1] insig[2]
Feedthrough outsig[1] insig[1]
Feedthrough outsig[1] insig[0]
Feedthrough outsig[0] insig[2]
Feedthrough outsig[0] insig[1]
Feedthrough outsig[0] insig[0]
```

For scalar signals, Feedthrough* is identical to Feedthrough.

For GUI use, select the *Design Import* tabcard, then select the *Timing Specification* tabcard. Below the *Data Signals* is a a button with *Rising* as the default. Click on *Rising* to invoke the selections and choose *Feedthrough*.

Feedthroughs and verify simulation

The way to determine a potential feedthrough in a verify simulation is with a setup time warning that tracks *VMW_INPUT_SETUP*.

For example, *VMW_INPUT_SETUP* is by default 100 time units. Running verify simulation results in a VSM warning as follows:

```
VirtuaLogic-Verify Warning: Design i/o "pad_TXFULL_": unexpected transition seen at
  time 3392008
```

By viewing the clock time, the user can determine that the pad_TXFULL_ has changed after the *VMW_INPUT_SETUP* time has elapsed and the user increases it to 200 time units. After running the simulation again, the following warning message is displayed:

```
VirtuaLogic-Verify Warning: Design i/o "pad_TXFULL_": unexpected transition seen at
  time 3392008
```

This repeated warning indicates that it might be a feedthrough case.

The user must determine what output caused the feedthrough input to change by using a testbench debug, help from another designer, etc.

If after a few of these are tracked down, you decide that all of these are related only to the testbench and do not have anything to do with the target, then the best thing to do is to switch to the vector based verify model to debug.

This happens most often if you are emulating submodules of a chip and have had to define all of the I/O as rising, falling, etc. It is very common for the RTL that models the rest of the chip to respond in a combinatorial manner to outputs from the emulated block.

This means there are cases when the verify simulation might not work. There are cases when the users will have the following:

- Declare nets as feedthroughs that are not in the target to get the testbench verify simulation to work
- Use the vector shell read and give up on the testbench which will probably make the verify simulation harder to debug if there is a mismatch

Bidirectional signals

Bidirectional signals generally are specified as synchronous rising, falling, or both edges. The input and output of the bidirectional signals will be timed to the same edge. If the target system requires that the input and output of a bidirectional signal be timed to different edges of the clock, they can be specified to do so by editing the file *vmw.clk*. The format for this file is documented in *Timing specification on page 332*.

Bidirectional output clocks are supported. If the signal feeds back into the emulation model as a clock, include the signal in an *XFLT* file.

Design I/O terminals

The *Design I/O Terminals* frame is used as a source for dragging and dropping terminal names into the *Add Clock* and *Add Clocked Data Signals* pop-up frame. These frames expect terminals on the top-level module and they are a way to accomplish this.

Instructions

1. The *Netlists*, *Netlist Defines*, and *Root Modules* must be filled out in the *Netlist Import* window.
2. Select the *Show I/O* button. If the graphical interface has not read in the netlist yet, it will read it now. If the netlist is very large, this may take some time. If you prefer not to wait and know the names of the top-level terminals, you can use the *Add Clock* and *Add Clocked Data Signals* buttons which will prompt the user for the names.
3. Select the clock terminals by pressing the left mouse button, dragging the mouse over the clock terminals, and then releasing the button. If the clock terminals are not adjacent to each other in the list, then repeat the process to pickup the ones missed the first time or hold down the <*control*> key while clicking the mouse to add new entries to the selected set.
4. Press the middle mouse button over the selected set and while the middle mouse button is depressed, move the mouse to the clock frame. Blank clock waveforms will be added to that frame for the user to edit.
5. The terminal names dragged will be removed from the I/O terminals list.
6. Repeat this process for the I/O signals.

Add domain

The *Add Domain* button adds a new blank domain pane when clicked.

Add clock

The *Add Clock* button is used to specify the clock name. If it is an internally generated clock (e.g., from a phase lock loop), then the user can specify a hierarchical name. The VirtuaLogic circuitry will automatically generate the clock waveform specified, relative to the external clocks.

Add clocked data signal

The *Add Clocked Data Signal* button is used to add clock signals.

By default, the signal will transition on the rising edge of the first clock in the selected domain.

A clock must be added before adding a signal, and the timing of each I/O signal must be specified relative to an edge of a clock. When a new signal is added to the list, it is given a default clock which may be changed; however, there must be a clock to begin with to add the signal.

Add asynchronous data signal

The *Add Asynchronous Data Signal* button invokes a window for the user to type an asynchronous signal name. After confirming the dialogue box, it will be added to the signals window. The user must do this for all the asynchronous primary I/Os of the circuit under emulation.

Import timing

The *Import Timing* button is used to specify an existing clock timing file. Typically, this will be found as *vmw.clk* in another configuration directory (extension.*vmw*).

Note that this will replace the existing timing information, it will not be combined.

The contents of this file will be read into the timing editor and will be written to *vmw.clk* in the current configuration the next time it is saved.

Note that the semantics of the file management differs between clock file, memory files, trigger files, and probe files. Refer to the sections below for details.

Clock files

Clock files are self-consistent and cannot be combined. It is not recommended to swap between multiple clock files and *gvl* always uses *vmw.clk* as the name of the active clock file. The only file management the user is expected to do is to copy the clock file from another configuration.

Gate counting

Gates used for memories have to be considered. There are 2 parts of memory interface gates. First, is the logic that converts our internal SRAM into whatever memory the gates need. Since different types of memories require different amount of gates as wrappers, there is no formula to calculate this. For instance, a simple register file or SRAM may have little overhead, but a DDR memory may have more overhead. However, this wrapper is not very significant to the overall gate-count. We can simply synthesize the wrapper to find the gate count.

Second, is the logic that interfaces with the internal SRAMs. As a rule of thumb, we can start with 250 gates for every port (read or write) per 8bit of data. This formula suggest that depth is not a factor. Therefore, for memories that are wide but shallow, it may be worth-while to synthesize them. To synthesize the memory, we can estimate the gate-count as 10 gates per bit. Level verse edge sensitive memories internally have essentially the same gate counts. We only have uni-directional ports, not bi-directional, in our internal memory interfacing.

We can use -Dump g in the compile option to obtain a histogram of all the VMW-primitives being used. This will give a good estimate for the gate-count. For more details on -Dump g switch, refer to Compiler options pane .

RTLC creates a hierarchical area report. This report has all the information on gates for each module. Refer to page 140 for a sample area report.

IKOS

4 Signals

Overview

There are two main categories of inputs that must be specified prior to compilation. The first category is the overall design specification, i.e., the netlists, the memory details, and the clock domains. The second category of compiler inputs is the signal probing information. The signal probing information specifies the visibility of emulated design nets, both for the purposes of viewing emulated waveforms and for use as inputs to triggering specifications.

The VirtuaLogic compiler supports two approaches to specifying signal visibility: "explicit probing" and "100% Visibility". In the explicit approach, the desired set of visible nets is explicitly specified by the user. In the 100% Visibility approach, which is a compiler option, the compiler automatically adds extra monitoring logic to the design and also determines a set of nets to monitor. With this automatically generated infrastructure, any signal in the emulated design can be reconstructed. The explicit probing approach can be used in conjunction with 100% Visibility. This allows for partial visibility of some signals prior to the reconstruction process. In particular, signals for use in trigger descriptions must always be explicitly probed.

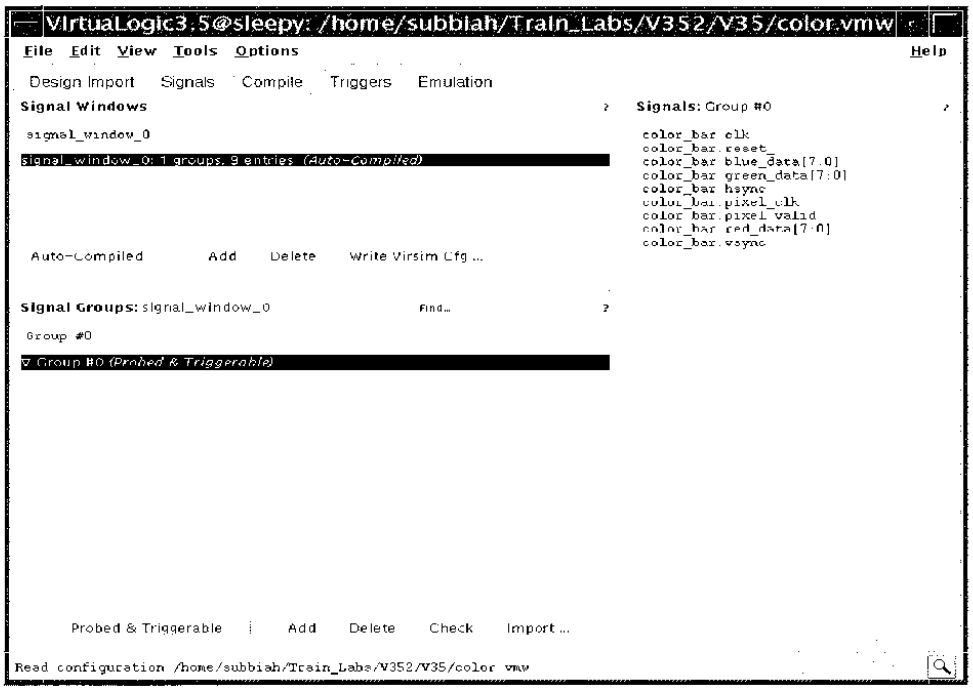
To help manage the complexity of explicit signal probing, sets of signals are organized hierarchically. At the bottom of this hierarchy are *Signals*, which are individual net names or wildcard expressions that expand into individual net names. *Signals* are collected into sets called *Signal Groups*. *Signal Groups* in turn are collected into larger sets called *Signal Windows* with window meaning "a window into the design".

A user may create as many signal windows as desired, and multiple signal windows may be compiled simultaneously into a design. Only one compiled signal window at a time, however, can be selected for emulation.

A signal window can contain approximately 8250 individual scalar nets. These individual scalar net that is captured from the array board and sent to the system board is called as “core probe”. Maximum number of core probes that an array board can capture and sent to a system board is 5000. The number of core probes for a 6 array board virtualogic emulation system cannot exceed 30000. The system board will ignore all the signals except 8250 of them at a time. 8250 signals can be chosen by the user by specifying the desired signal window for emulation. Provided the overall 30000 limit is not exceeded, there is no particular limit on the number of signal groups in a window, nor on the number of signals in a group. Note that if the 100% Visibility compiler option is enabled, the automatically generated probing will count against the *Signal Window* and core probe limits.

The type and content of *Signal Windows* and their groups are specified using the Signals page in GVL. This page is shown in *Figure 29 on page 110*. The remainder of this chapter details the use of the *Signals* page.

Figure 29 on page 110 displays the *Signals* Page.


VirtuaLogic3.5@sleepy: /home/subbiah/Train_Labs/V352/V35/color.vmw
File Edit View Tools Options
Help
Design Import Signals Compile Triggers Emulation
Signal Windows
Signals: Group #0
Auto-Compiled Add Delete Write Virsim Cfg ...
Signal Groups: signal_window_0
Find...
Probed & Triggerable Add Delete Check Import ...

```
signal_window_0
signal_window_0: 1 groups, 9 entries (Auto-Compiled)

Group #0
▽ Group #0 (Probed & Triggerable)

color_bar clk
color_bar.reset_
color_bar blue_data[7.0]
color_bar green_data[7:0]
color_bar hsync
color_bar.pixel_clk
color_bar.pixel valid
color_bar red_data[7·0]
color_bar.vsync

Read configuration /home/subbiah/Train_Labs/V352/V35/color vmw
```

Figure 29 *Signals* page

Signal windows pane

The *Signal Windows* Pane is used primarily to select a particular signal window for subsequent viewing or editing with the *Signal Groups* and *Signals* panes. It is also used to create or delete *Signal Windows*.

The list within the pane shows the *Signal Window* defined for the current configuration. To select a particular signal window, simply use the mouse to click the list. The *Signal Groups* pane and the *Signals* pane will update accordingly.

To change the name of a signal window, first select it in the list, ensure that the name field has the keyboard focus (click with the mouse), and then use the keyboard.

When the 100% Visibility is turned on, the user only needs to specify probes for the purpose of triggering. The 100% Visibility system will automatically probe all primary inputs, clocks, and memory outputs. This automatic probing will count against the *Compiled Signal Windows* and core probe limitations.

100% Visibility is the ability to provide access to values of all design nodes during a display time window. The duration of this display window is based on the storage depth of the IDS and corresponds roughly to the duration achievable via probe-based visibility. The position of this window is selected via triggering, as occurs for current probe-based visibility.

In addition to enabling 100% Visibility, it is also necessary to manually select probes and probe windows for any signals that are to be used in triggers. All inputs to build triggers should be manually selected in order to ensure that they are available.

100% Visibility imposes some hardware overhead cost when compiled into models. The total cost is design dependent, typically averaging between 10% and 15% size increase. Certain structures can be particularly expensive and may lead to more excessive costs.These include large storage macros implemented as gates, heavily latch-based design styles, large numbers and/or very high fanout cross domain or asynchronous nets, particularly if the nets go to asynchronous preset or clear terminals.

Space improvements can be obtained by the following:

- Modeling storage macros as memories
- Treating high-fanout asynchronous inputs as synchronous
- Using *Compile option-Sr* to convert asynchronous preset/clear modeling to synchronous preset/clear modeling

- Using quasi-static annotations on the highest fanout cross domain nets if they are not already set as a quasi-static net

Auto-compiled/not compiled

As mentioned in the Overview, multiple *Signal Windows* can be simultaneously compiled into a design. The *Auto-Compiled Not Compiled* drop-down list allows the user to select which signal windows are to be compiled into the design. The current compilation choice for each signal window is displayed in the window list.

Delete

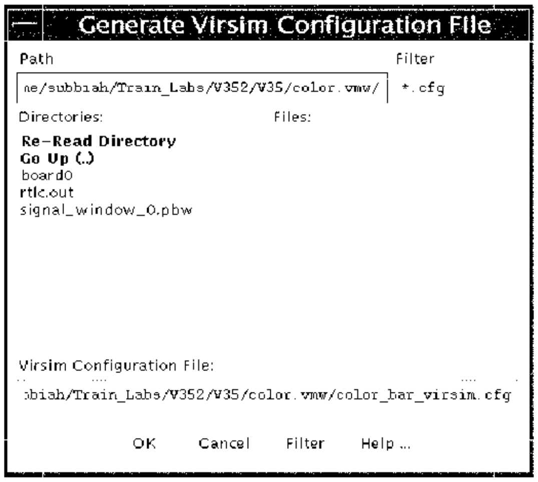
The *Delete* button is used to delete the selected signal window.

Add

The *Add* button is used to create a new empty signal window. The name of the new window will be automatically generated. This name can then be changed as described earlier.

Write virsim configuration

The *Write Virsim Configuration* button invokes a dialog, shown in *Figure 30 on page 113* to generate a *Virsim* configuration file for the currently selected signal window.



Generate Virsim Configuration File

Path: `ne/subbiah/Train_Labs/V352/V35/color.vmw/`

Filter: `*.cfg`

Directories:

Re-Read Directory
Go Up (..)
board0
rtlc.out
signal_window_0.pbw

Files:

Virsim Configuration File: `:biah/Train_Labs/V352/V35/color.vmw/color_bar_virsim.cfg`

Figure 30 Generate virsim configuration file window

The generated configuration file has waveform groups matching the signal groups, and has *Virsim* expressions for each *VirtualLogic* synthetic vector in the signals list. Synthetic vectors are vectors that *Synopsys* has flattened into "a_0_, a_1_, a_2_, ..." and which"gvl" recombines using the syntax "a_<31:0>".

The user will see their groups in *Virsim* organized in the same way as the groups appear in the *Signal Window*. Therefore, signal groups should be arranged before compiling, so that they appear in the desired order for *Virsim*.

A configuration file that has been explicitly updated using the *Virsim* menu will be overwritten if the *OK* button is used.

Signal groups pane

The *Signal Groups* Pane is used primarily to select a particular signal group for subsequent viewing or editing with the *Signals* pane. It is also used to create or delete *Signal Groups*, and as a convenient drag and drop source or target for manipulating group contents.

The tree list within the pane shows the signal groups defined for the currently selected signal window. To select a particular group, simply use the mouse to click the list. The *Signals* pane will update accordingly to show the signals within the selected group.

To change the name of a *Signal Group*, first select it in the list, ensure that the group name field has the keyboard focus (click with the mouse), and then use the keyboard.

Double clicking a *Signal Group*, or single clicking the expansion arrow for a *Signal Group* will show the full set of nets for that group. This includes the effect of any wildcard expansions contained in the *Signals* for the group. Depending on the complexity, wildcard expansions can be time consuming since the design netlists must be processed.

Group types

Signal Groups can be one of three types: *probed*, *triggerable*, or *probed & triggerable*.

- Signals that are within *probed* groups are only visible; they cannot be used as inputs to trigger descriptions.
- Signals within pure *triggerable* groups can only serve as inputs to trigger descriptions; they are otherwise not visible.
- Signals within *probed & triggerable* groups are both visible and usable as inputs to trigger descriptions.

In general, groups types should be chosen to be both probed & triggerable.The more restrictive group types are available for fine tuning compiler resource usage.

Add

The *Add* button is used to create a new empty signal group. The name of the new group will be automatically generated. This name can then be changed as described earlier.

Delete

The *Delete* button is used to delete the selected signal group.

Check

The *Check* button is used to verify that the probes are legal, existent nets and will fit in the probe hardware.

Import probes

The *Import Probes* button invokes a dialog to read in a set of probes from another configuration, augmenting existing probes.

Typically, the imported probes will come from the *vmw.prb* file in another configuration directory (extension .vmw) or in one of the *Signal Windows* subdirectories (extension .pbw) of the configuration.

The imported signals will be merged with the existing signals respective of group names.

The contents of this file will be read into the probe editor and written to *vmw.prb* in the current configuration or *Signal Window* the next time it is saved.

Signals pane

The *Signals* pane is where the user can explicitly enter the nets for the current signal group. There can be an arbitrary number of signal groups. Selecting a signal group will show that group's signals in the list.

The signals entered into the *Signals* pane should each be placed on one line in the form of Verilog paths. They can be typed in directly by clicking in the *Signals* pane or they can be dragged and dropped from the *VirtualBrowser* or from *Virsim's* hierarchy window.

Wildcards (* and ?) are accepted in the *Signals* list. For example, **a.b.*** means every net inside module **a.b**. A module path (not ending in a net) such as **a.b** can be specified. This causes all the interface nets on that module to be probed, leaving out the internal nets that would be included by specifying **a.b.***.

There are three special wildcard syntax forms which can be used to help probe classes of nets as follows:

- <path>@all

This is a special construct that is used to provide 100% Visibility into a moderately sized subsystem. It automatically probes all the primary inputs into the subsystem, plus all the state element outputs within the subsystem. This is sufficient to show all design nodes within the subsystem because the waveform viewer, *Virsim*, can reconstruct combinational output given the inputs.

For example, the probe specification:

```
a.b.c@all
```

will provide 100% Visibility into the Verilog scope *a.b.c*, including everything hierarchically beneath *a.b.c*, down to the primitive level.

```
<path>@state
```

This probes all state elements in the path. It is essentially obsoleted by *@all*.

```
<path>@memory
```

This probes all memory elements in the path. It is essentially obsoleted by *@all*.

For subsystems not conveniently isolated to a single hierarchical scope, a series of selections can be combined to capture the desired signals, such as the following:

```
Net a.b.c@all
```

Within *Virtualogic*, @all is somewhat limited in its application because of the size restrictions of the probe hardware. @all can provide subsystem level full visibility for subsystems up to between 8K and 15K gates, depending on design style.

NOTE

Note that all three of these special syntaxes are disabled when 100% Visibility is turned on.

IKOS

5 Compiler

Overview

Design compilation takes the design database produced by the Netlist Import and Verification phase and processes it into a downloadable emulator image for the emulation platform. To achieve this, it combines the previously entered logical information with new information relevant to the physical compilation process. Complete GUI support is provided for RTL Verilog flow but for VHDL RTL flow, VHDL to verilog netlist compilation is done by running *rtlc* stand-alone, after which full GUI support is provided for compiling the generated verilog netlist.

The features of the *Compile* form are as follows:

Additional major topics discussed in this section are as follows:

RTL Compile

RTLC-VLE flow for ICE on page 119 explains the RTLC-VLE flow.

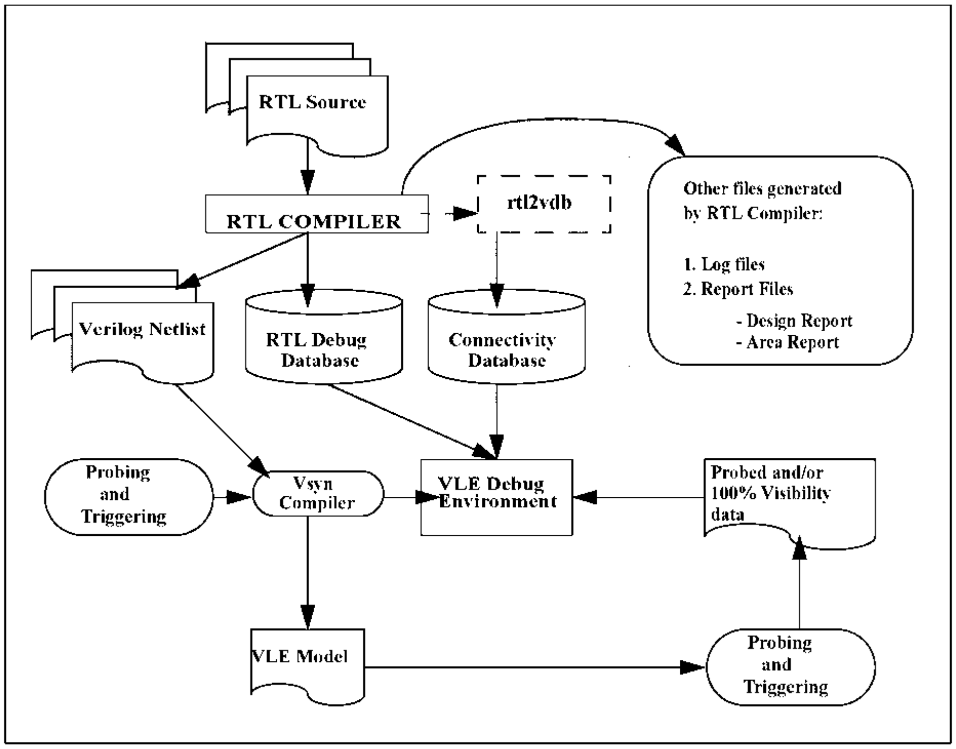


Figure 31 RTLC-VLE flow for ICE

RTL Compiler flow

Input

RTLC-VLE supports the Synopsys DC synthesizable subset of

- Verilog
- VHDL

Output

During the process of compilation of the RTL design, the RTL compiler, together with its driver, writes out the following information as a set of files which will be used by VirtuaLogic. RTLC generates

- verilog netlists optimized for VLE architecture
- empty module descriptions for behavioral blocks
- RTL debug database for RTL level debugging by the user
- log files and report files for providing information to the user about the design.

Log

During the course of compilation (including analysis), rtlc-vle creates a log file(*rtlc.log*) under *config name.vmw rtlc.out*. Description of the log file is given below.

- All message go to the log file
- Each design unit has its own section
- Messages are arranged according to the design-unit in which they occur
- It has the summary of number of errors for each design unit
- Incremental runs update and format the log.

The Figure 32 shows a sample Log file.

```
 Report: Hcounter

  Status 4506: Module Hcounter: Pre-processing...
  Status 4508: Module Hcounter: Compiling...
 SimWarning 5784: File /home/subbiah/Train_Labs/V352/netlist
 /rtl/Hcounter.v, Line
21, Module Hcounter, Net data: Although this signal is not
 part of the sensitivity list of this block, it is being read.
 This may lead to simulation mismatch.
 Notice 5205: Module Hcounter, Net tdata[11]: Latch inferred.
 Notice 5205: Module Hcounter, Net tdata[10]: Latch inferred.
 Notice 5205: Module Hcounter, Net tdata[9]: Latch inferred.
 Notice 5205: Module Hcounter, Net tdata[8]: Latch inferred.
 Notice 5205: Module Hcounter, Net tdata[7]: Latch inferred.
 Notice 5205: Module Hcounter, Net tdata[6]: Latch inferred.
 Notice 5205: Module Hcounter, Net tdata[5]: Latch inferred.
 Notice 5205: Module Hcounter, Net tdata[4]: Latch inferred.
 Notice 5205: Module Hcounter, Net tdata[3]: Latch inferred.
 Notice 5205: Module Hcounter, Net tdata[2]: Latch inferred.
 Notice 5205: Module Hcounter, Net tdata[1]: Latch inferred.
 Notice 5205: Module Hcounter, Net tdata[0]: Latch inferred.
 Errors:0  Warnings:0  RTLErrors:0  RTLWarnings:0  SimErrors:0

 Report: Vcounter
  Status 4506: Module Vcounter: Pre-processing...
  Status 4508: Module Vcounter: Compiling...
 Errors:0  Warnings:0  RTLErrors:0  RTLWarnings:0  SimErrors:0

 Report: color_bar
 Status 4508: Module color_bar: Compiling...
 Errors:0  Warnings:0  RTLErrors:0  RTLWarnings:0  SimErrors:0


 Report: image1_gen
 Status 4506: Module image1_gen: Pre-processing...
 Status 4508: Module image1_gen: Compiling...
 Warning 5437: Module image1_gen, Net datav[0]: This net is hanc
 Warning 5437: Module image1_gen, Net datav[0]: This net is hanc
 Errors:0  Warnings:2  RTLErrors:0  RTLWarnings:0  SimErrors:0


 Report: image2_gen
 Status 4506: Module image2_gen: Pre-processing...
 Status 4508: Module image2_gen: Compiling...
 Warning 5437: Module image2_gen, Net datav[0]: This net is hanc
 Warning 5437: Module image2_gen, Net datav[0]: This net is hanc
 Errors:0  Warnings:2  RTLErrors:0  RTLWarnings:0  SimErrors:0
```

Figure 32 A Sample log file

Messages

The messages can be classified into:

Table 2 RTLC messages

Messages	Description
Status	denotes the progress of compilation
Info	useful information
Notice	important information which must be noted
Warning	normal warnings
RtlWarning	warnings related to synthesizability problems
SimWarning	warnings related to potential simulation mismatches
Error	normal errors
RtlErrors	errors related to synthesizability problems
Fatal Errors	errors related to potential simulation mismatches
SimErrors	errors related to potential simulation mismatches

Status messages

Status messages are useful for denoting the progress of compilation, In general, they can be ignored.

Examples:

```
Status 4508: Module image2_gen: Compiling...
Status 4518: Reading target library...
```

Info messages

Info messages provide some useful information or statistics about the compilation, but they do not depict any kind of problems with the design.

Examples:

```
Info 4842: Compilation successfully completed.
Info 4873: Module vmw_ram16x17: This module has been specified
  as an 'import' module. It won't be compiled.
Info 4861: Maximum count for message 5205 reached. This
  message will not be displayed for this module.
Info 4835: Total CPU time taken for compilation: 37 secs..
Info 4856: Total lines of RTL compiled: 10193
```

Notice messages

Notice messages are messages that tell the user how a particular construct is being handled. For example, Notice messages will tell what latches are inferred in the design.

Examples:

```
Notice 5205: Module Hcounter, Net tdata[3]: Latch inferred.
Notice 5209: ===encountered. Treating as ==
```

Warnings

Warnings denote non-fatal problems with the design, and also alert the user to how RTLC handles certain kinds of constructs.

Verilog Examples

- Assumptions taken

```
always @ (a or b or c)
if (c ===1`b0)
   z= 1`bx;
else
   z = b ? a : ~a;
```

```
Warning 5404: Module m: Reading 'X' value either for
assignment or for comparison. 'X' value will be ignored
and treated as zero.
```

- Potential Design Issue

```
always @ (a or b)
    z = a ? b : 1`bz;
always @ (c or d)
    z = c ? d : 1`b0;
```

```
Warning 5411: Module m, Net z: This tri-state net has non
tri-state drivers.
```

- Recommendations

reg [32:0] mem_core [32:0];

```
Warning 5516: Module m: Net c: The signal/variable is a
potential memory of size 1024 bits. It is advisable to use
IKOS memory models in this case. To use IKOS memories,
black-box this module using switch "-import m" during
compilation
```

VHDL Examples

- **Assumptions taken**

```
process (a , b , c) begin
  if (c == 'X') then
    z <= '0';
  else
    z <= a xor b xor c;
  end if;
end process;
```

```
Warning 5431: File x.vhd, Line 12: All equality
comparisons with 'X', 'W', 'U', or 'Z' are treated as
FALSE. This can potentially lead to simulation mismatch.
```

- **Potential Design Issue**

```
function invert (a: std_logic)
return std_logic is
  --ikos translate_off
if (a = 'x') then return '1';
else return not a;
end if;
  --ikos translate_on
end function;
```

```
Warning 5520: File f.vhdl, Line 28: Whole function/
procedure body is inside synthesis_off/translate_off
directive. Please check that this is what is intended.
```

- **Recommendations**

signal q: std_logic := '1';

```
Warning 5501: File i.vhdl, Line 9, Module
MY_WORK.TOP(RTL): Default initial values for signals/
variables is ignored. This may cause potential simulation
mismatch. If you want to compile initial values for state
points then please recompile using option "-
compile_vhdl_inits"
```

RtlWarnings and RtlErrors

RtlWarnings and RtlErrors denote synthesizability problems with the design:

-enable_BHV_messages switch can be used to see exactly what is causing the synthesizability problem. Assert/report in VHDL and $display() etc. in Verilog are ignored

Assertions example

```
process (clk, en, d))
begin
if (clk'event and clk = '1') then
assert not (en = '0') report "disabled";
  q<=d;
  end if;
  end process;
```

RtlWarning 3502: File assert.vhd, Line 12: Assertion statement will be ignored.

Bad clocking style verilog example:

```
  always @ (posedge clk or reset) // should be posedge reset
    if (reset)
      q = 0;
    else
      q = d;
```

```
RtlError 1155: File clk.v, Line 5: Both edge control and
non-edge control expressions cannot be specified in the
sensitivity list.
```

Bad clocking style VHDL example:

```
  process (clk, reset, d) begin
    if (clk'event and clk = '1') then
      q <= d;
    elsif (reset = '1') then
      q <= '0';
    end if
  end process;
```

```
RtlError 3524: File clk.vhd, LIne 14: The IF-statement
does not confirm with any description style that can be
compiled.
```

```
  // the above code should be
  if (reset = '1') then
      q<= '0';
  elsif (clk'event and clk = '1') then
    q <= d;
  end if;
```

Initial Blocks example:

```
  initial q = 0;
  always @ (posedge clk)
  q = d;
```

```
RtlError 1102: File init.v, Line 4: INITIAL statements are
not supported.
```

Errors and Fatals

Errors and Fatals are usually conditions where RTLC cannot proceed without a fix from the designer. Depending on their severity, they may be either fatal to a module causing RTLC to proceed to the next module or they may be fatal to the entire design causing the compilation to abort.

General error example:

```
z <= a + b;    // a, b are bit_vectors
```

```
Error 2549: File nr.vhd, Line 23: Either type mismatch or
no visible function for this case.
```

The above example is an unsupported/illogical constructs. This leads to disabling compilation of the particular module.

Fatal Error example:

```
always @ (a or b) begin
  t = 0;
  ...
  z1 = a/t;    // Fatal 7186: Division by zero. Exiting...
  t = b;
  z2 = a/t;
end
```

```
Fatal 7025: Division operator in a non-static expression
where both the operands are non-static, or right operand
is static but not a power of two is currently not
supported. Exiting...
```

Inconsistencies across hierarchy:

When imported/non-RTL modules have different prototypes due to instantiation with different generics, the user gets the following error.

```
Fatal 7064: File top1.vhd, Line 22, Module gate: Ports of
non-RTL/imported modules are dependent on generics.
Exiting...
```

When black boxes are instantiated by positional association, the user gets the following error.

```
Fatal 7052: Instance of black-box module mem_core has
implicit port connections. Only explicit/named port
connections can be supported. Exiting...
```

Systemic Errors:

```
Fatal 6803: Environment variable RTLC_HOME not set.

Fatal 7133: Out of memory. Memory allocation failed. Try
to run the design on a workstation with more swap area.

Fatal 6811: Segmentation violation.
```

SimErrors

SimErrors are more severe than normal errors but less severe than Fatal errors. This means that compilation will proceed normally in case of SimErrors but a non-zero exit status will be returned.

The RTL compiler errors out on design structures which are syntactically correct and synthesizable, but which will lead to mismatches between software simulated and emulated behavior. Such structures are called Simulation Errors. Nevertheless, there may be situations where such structures cannot be avoided. Simulation Errors are the result of some basic semantic differences between software simulation and emulation.

SimErrors generated by rtlc are warning to user indicating that these errors may be one of the reasons for simulation mismatch. These warnings are emitted by *rtlc* for X's in the design, signal initialization, race conditions, undefined outputs for functions. So these are the indications to the user to change the design.

These situations are listed and illustrated below:

- Incomplete Sensitivity Lists(Verilog)
- Undefined Function/Task Outputs(Verilog)
- Multiple Drivers(Verilog)
- Four-state Reads(Verilog)
- Gate Strengths and Delays(Verilog)
- Clock variable data(VHDL)
- Function return null(VHDL)

Incomplete sensitivity lists

Reading a signal inside an *always* block which is not part of its sensitivity list will lead to a simulation mismatch. For instance, consider the following Verilog RTL code:

```
input a, b, c;
output z;
reg z;
always @(a or b)
    z <= a + b + c;
```

In this case, software simulation results in the value of *z* being updated only if either *a* or *b* or both change, but not *c*. However, in the hardware implementation (of a full adder), *z* is sensitive to *c*. This will result in a simulation mismatch.

The compiler errors out whenever such a condition is detected.

Allowing incomplete sensitivity lists prevents the compiler from erroring out. However, there is still a potential for simulation mismatch as the sensitivity list is assumed to be complete in the hardware implementation.
Consider the following always block:

```
always @(select) // line 5 of file accum.v
  begin
    if (select)
      accum = data;
    else
      accum = ~data;
  end
```

In this case, the net accum is being read, but it is not present in the sensitivity list of the block. In software simulation, the block will not be entered and accum will not be re-evaluated if data changes. However, in the concurrent hardware implementation, accum will always get re-evaluated whenever data changes. For illustration, consider the following value changes:

```
Input stimuli Sw sim results Hw results
--------------------- -------------- ----------
select <= 0; data <= 0 accum = 1 accum = 1
select <= 0; data <= 1 accum = 1 accum = 0 (Mismatch!)
```

The compiler will issue the following error for this:

```
SimError 5784:
File accum.v, Line 5, Module accumulator, Net data:
Although this signal is not part of the sensitivity list of
this block, it is being read. This may lead to simulation
mismatch.
```

Issues with non-blocking assignments:
On similar lines, consider the following block:

```
always @(accum or data) // line 25 of file accum.v
```

```
begin
  temp <= accum ^ data;
  if (temp == 0)
    zero = 1;
  else
    zero = 0;
end
```

In this case, the net zero will be evaluated using the old value of temp in software simulation. However, it will be evaluated using the new value in the hardware implementation, which can lead to a simulation mismatch. Consider the following value changes:

```
Input stimuli Sw sim results Hw results
---------------------- ------------------ ------------------
accum <= 0; data <= 0 temp = 0; zero = 1 temp = 0; zero = 1
accum <= 0; data <= 1 temp = 0; zero = 1 temp = 1; zero = 0 (Mismatch!)
```

The compiler will issue the following error in this case:

```
SimError 5784:
File accum.v, Line 25, Module accumulator, Net temp:
Although this signal is not part of the sensitivity list of
this block, it is being read. This may lead to simulation
mismatch.
```

The solution here will be to either change the assignment to temp to a blocking assign or to put temp in the sensitivity list of the block.

Similar example on ISL:

```
always @ (in1 or in2) begin
  tmp <= in1 & in2;
  out = tmp ? 0 : 1;
end
```

```
SimError 5784: File isl.v, Line 7, Module isl, Net tmp:
Although this signal is not part of the sensitivity list of
this block, it is being read. This may lead to simulation
mismatch.
```

Undefined function/task outputs

Functions or tasks which have undefined outputs or return values in one or more paths will result in simulation mismatches. Consider the following Verilog RTL code:

```
function sumif1 (cnd, a, b);
  input cnd, a, b;
  if (cnd)
    sumif1 = a + b;
endfunction
```

In this case, the return value of *sumif*1 is undefined if *cnd* is zero. This may lead to a simulation mismatch.

Allowing undefined function or task outputs prevents the compiler from erroring out. All undefined outputs are assigned to zero. Thus, in the above case, allowing undefined function outputs results in *sumif*1 getting assigned to zero if *cnd* is zero.
Consider the following function definition:

```
function reset_data;
  input reset;
    if (reset)
      reset_data <= 0;
endfunction
```

The return value of this function is not defined when reset is inactive. This may lead to a simulation mismatch with RTL. Note that the hardware implementation will assign all undriven outputs of the function to zero.

The compiler will issue the following error for this:

```
SimError 5785:
Module reseter, Function reset_data: Return value is
undefined in one or more paths in this function. This
may lead to simulation mismatch.
```

Multiple drivers

In case of multiple drivers in the design, there is a possibility of simulation mismatch if all the drivers are active at the same time.

Multiple combinatorial drivers or multiple sequential drivers on the same clock are error conditions. Multiple sequential drivers on different clocks (or different edges of the same clock) are handled automatically.

Allowing multiple drivers prevents the compiler from erroring out. Multiple combinatorial drivers are always treated as wired or circuits.
Consider the following example where there are multiple combinatorial non-tristated drivers for the net current_state:

```
always @(posedge clock)
```

```
    current_state <= next_state;

  always @(reset)
    current_state <= initial_state;
```

In this case, current_state is being driven both by next_state (through a flip-flop) and also by initial_state. This may cause next_state to go to X in a software simulation and may cause a simulation mismatch.
The compiler will issue the following error for this:

```
SimError 5783:
Module state_gen, Net current_state: This signal has
multiple drivers. This will lead to simulation mismatch.
```

Four-state reads

Reading four-state values such as X and Z make no sense in real hardware and emulation, and so compared to the result of software simulation that may produce unexpected mismatches. In general, four-state reads tend to occur in comparisons and assignments.

For example, consider the following Verilog RTL structures:

```
if (current_state == 2'bx)
  next_state = 0;
else
  next_state = current_state;
```

and

```
if (go_idle == 1)
  next_state = 2'bx;
```

Allowing four state reads prevents the compiler from erroring out under these conditions. Instead all assignments to X are taken as an assignment to zero, and all comparisons with X or Z are treated as FALSE. Thus in the first example given above, the if branch is never taken, and in the second example, the assignment to next_state is taken as an assignment to zero.

Note that assignments to Z which are valid tri-state assignments are handled by inferring appropriate tri-state devices.
Consider the following example which has an explicit X assignment:

```
input [1:0] operator;
input [7:0] operand;
```

```
reg [7:0] accumulator;

always @(posedge clock)
begin
 case (operator)
 2'b00: accumulator = accumulator + operand;
 2'b11: accumulator = accumulator - operand;
 default: accumulator = 8'bx; // line 15 of file alu.v
 endcase
end
```

In this case, the default case assigns Xs to accumulator. This is a common situation where the default statement is not intended to be reached.
The compiler will issue the following error:

```
SimError 5782:
File alu.v, Line 15, Module alu: Reading 4-state value
('X' or 'U' or 'W' or 'Z') for assignment or comparison.
This may lead to simulation mismatch.
```

Gate strengths and delays

Specifications of strengths and delays in gate instantiations are ignored by the RTL compiler. This leads to a simulation mismatch.

```
bufif1 #1(YA,A,EN); // tristate, line 5 of gsd.v
```

```
SimError 5786:
File gsd.v, Line 5, Module gsd: Delays and Strengths
associated with gate instances are ignored.
This may lead to simulation mismatch.
```

```
module busHolder (io);
inout io;
wire internal;
not (pull0, pull1)not_i(io, internal), not_o(internal,io);
endmodule
```

```
SimError 5786:

File busHolder.v, Line 5, Module busHolder: Delays an
Strengths associated with gate instances are ignored. This may
lead to simulation mismatch.
```

The following two simerrors are applicable only for VHDL source codes. Previous versions of software used to error out for clock variable data and function return null, now these are categorized under simerrors.

Clock variable data

This simerror is generated when a clock variable is used as data(either read or write). Here is a testcase which clearly explains the clock variable data simerror:

```
entity clock1 is
 port(in1,in2: bit;
      clk : bit;
      output : out bit);
end;
architecture arch of clock1 is
begin
   process(clk,in1,in2)
   begin
      if (clk'event and clk = '1') then
         output <= clk;
      end if;
   end process;
end;
```

SimError Message

```
File clock1.vhdl, Line 13: Clock variable is being used as
data.
```

In this case, output gets clk as data when there is a positive edge of clk. This causes a simerror and needs necessary corrections from the designer in order to proceed further with the compiling.

Function return null

This error is generated when the function body has no return statement. Here is a testcase which clearly explains the function return null simerror:

```
architecture if_in_func2 of if_in_func2 is
 function find_max(in1,in2,in3,in4 : integer )
     return integer is
 begin
--Synopsys synthesis_off
    if(in1> in2) then
```

```
            if(in1>in3) then
                if(in1 > in4) then
                  return in1;
                else
                  return in4;
                end if;
             else
                if(in3 > in4) then
                  return in3;
                else
                  return in4;
                end if;
            end if;
         else
           if ( in2 > in3 ) then
                if(in2 > in4) then
                    return in2;
                else
                    return in4;
                end if;
            else
               if (in3 > in4) then
                    return in3;
               else
                    return in4;
               end if;
           end if;
         end if;
     --Synopsys synthesis_on
      end;
     begin
        max <= find_max(in1,in2,in3,in4);
     end;
```

The "Function return null" simerror is generated for the above testcase, since the user has set the *SYNTHESIS OFF* for the function body including the return statement.

SimError Message

```
Processing function rtlcF_IF_IN_FUNC2_IF_IN_FUNC2_FIND_MAX.
Probably the function cannot be compiled.
```

Report Files

Apart from the log file, RTLC also generates a design report file and an area report file. The design report file contains useful information about the design (& potential issues in the design):

1. potential design issues like combinational loops, clocks driving combinational logic, gated clocks, etc.

2. latches in the design.

3. Flipflops with any set/reset.

4. Incomplete case statements (only in verilog).

5. multiple driver nets.

6. Tristate buffers inferred in the design.

7. All the primary and secondary clocks.

8. Full case/parallel case inference.

9. Details of inferred tri-states.

10. Combinational loops.

11. Details of resources shared, if any.

A sample design report file is shown in *Figure 33 on page 136*.

```
Report: Vcounter
==============================================================
clock-name| Type      | Drives cmb logic| Drives I/p to F/F
==============================================================
 clk      | Primary  |          N       |          N
 hsync    | Primary  |          N       |          N
==============================================================

==============================================================

Name       | BUS SLICE  |Type    |  AR  | AS  |Clock/Enable

==============================================================

vsync      | ---        | FF     |   N  | N   |    clk

count_out  |   [11:0]   | FF     |   N  | N   |    clk

carry      | ---        | FF     |   N  | Y   |    hsync

count      |   [11:0]   | FF     |   N  | Y   |    hsync

pcount     |   [11:0]   | FF     |   N  | Y   |    hsync

vcount     |   [11:0]   | FF     |   N  | N   |    clk

vsync_del  | ---        | FF     |   Y  | N   |    hsync
```

Figure 33 Sample design report

DRC-- Flip Flops:

- Net name, Part select, Clock name
- if the flipflop has asynchronous set/reset
- Checks initializations for F/Fs without AR/AS

Example:

```
always @ (posedge clk or negedge set)
if (!set) data_out = 4'b1;   //asynchronous active low set.
else data_out = data_in[7:4];   //synchronous data transfer
```

Name	Name	Type	AS	AR	Clock/ Enable
data_out	[3:0]	FF	Y	N	clk

Name	Name	Type	AS	AR	Clock/ Enable
q	---	FF	N	N	clk

DRC-- Latches:

- Net name, Part select and enable
- Always check latches with internal enables

Example:

```
always @ (sel or in1 or in2)
  casex (sel)
  2'b0x:  out = 0;
  2'11: data = in1 + in2;     // missed out sel == 2'b 10
endcase
```

Name	Name	Type	AS	AR	Clock/ Enable
data	[31:0]	Latch	--	--	**internal**

DRC-- Tri-states

- Net name, Part select

Example:

```
inout [31:0] data;
always @ (addr or data)
  if (wren)
    core[addr] = data;
else
  temp = core[addr];
assign data = (wren) ? 'bz: temp;
```

Name	Name	Type	AS	AR	Clock/ Enable
data	[31:0]	TRIBUF	--	--	--

DRC -- Clocks

- Primary, derived, gated
- Can show potential race conditions

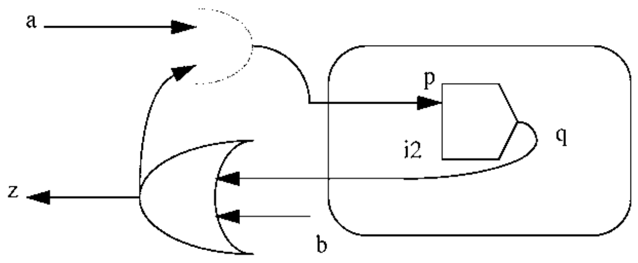
Example:

```
always @ (posedge clk)clk2 = ~clk2;
always @ (posedge clk)clk4 = ~clk4 & enable;
always @ (clk2 or clk4)wclk = clk2 & clk4;
```

Clock-name	Type	Drives combinatorial logic	Drives input to FF
clk	Primary	N	N
clk2	Derived	YES	YES
clk4	Derived	YES	N
wclk	Gated	N	N

DRC -- Combinatorial Loops

- Reports all combinatorial feedback paths
- pay attention to large loops
- run with -preserve to see actual names



Combinatorial Loop Warnings:

1) z p(i2) --> q z

2) rtlcn1 --> rtlcn2 --> rtlcn3

/* Loop consists of internal nets only. To determine the user nets leading to the loop, please run with *-preserve* or *-debug* option.*/

DRC -- CASE Inference

- Case type, whether full or parallel
- User-defined full/parallel case pragmas supported
- Use -enable case_pragmas

Example:

```
always @ (a or b)
  casex (a) // ikos full_case
    2'b0x: z1 <= {a[0],b[0]};
    2'b1x: z2 <= {a[1],b[1]};
  endcase
always @ (a or b)
  case (b)
     2'b00: z2 <= b;
     2'b11: z1 <= a;
  endcase
```

Case Type	Line No	Full	Parallel
CASEX	4	User	No
CASE	9	No	Auto

DRC -- Multiple Drivers

- Multiple drivers always have to be resolved
  - Multiple combinational drivers
  - Combinational and sequential drivers
  - Sequential drivers on the same clock edge

Example:

```
always @ (cnd1)
if (cnd1) out = in1;
assign out = (cnd2) ? in2 : 1'b0; // Signals with multiple drivers: Name(out) Bus(--)
```

- Sequential drivers on different clocks/edges are supported.

DRC Summary:

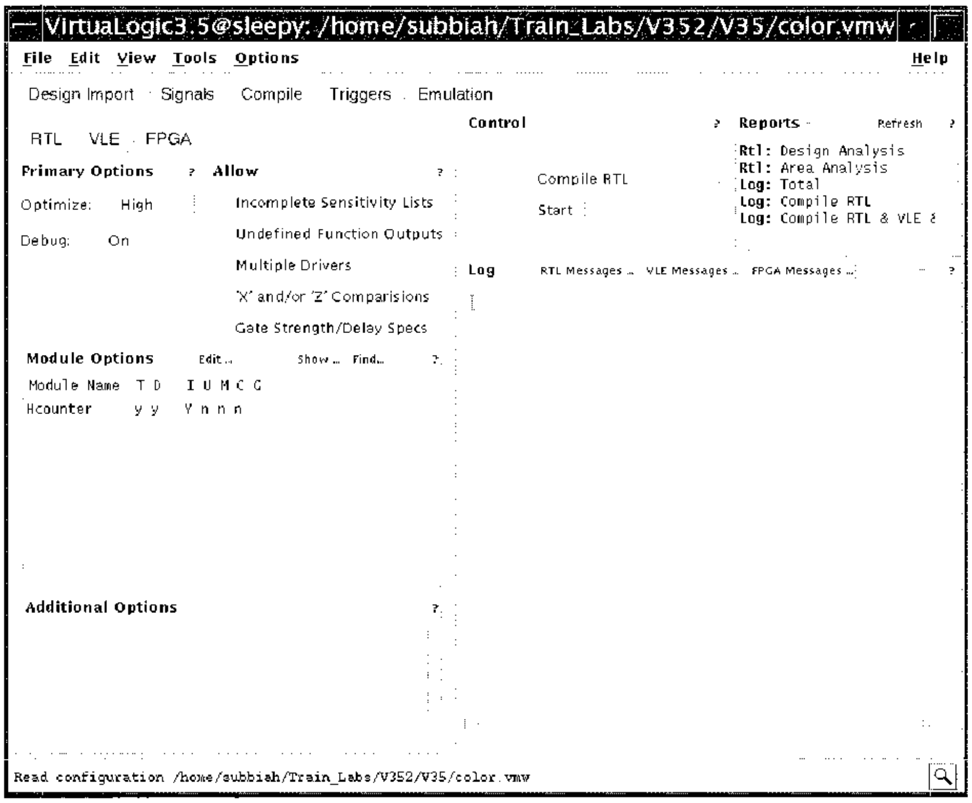
- Records RTLC interpretation of design constructs
- Can directly point to design issues
- Can find and solve design problems much faster at compile time instead of having trouble debugging in the run time.

A sample area report file is shown in *Figure 34 on page 140*.

```
Report: Vcounter
MODULE: Vcounter (type: USER/RTL)

--------------------------------------------------------------
Instance                 Count     Area: Comb   Seqn( FFs,LTs)   Macro
--------------------------------------------------------------

VMW_BUF                    9         9p     0        0(    0,   0)     0
M_RTLSIM_MUXN_2_1         27        27p    81        0(    0,   0)    81
M_RTLSIM_INCR_12           3        36p   138        0(    0,   0)   138
M_RTLSIM_DECR_13           1        26p    50        0(    0,   0)    50
M_RTLSIM_EQ_12             1        16p    42        0(    0,   0)    42
VMW_AND2                   8         8p    16        0(    0,   0)     0
VMW_FD                    25        25p     0      200(   25,   0)     0
VMW_FDC                   25        25p     0      225(   25,   0)     0
VMW_FDP                    1         1p     0        8(    1,   0)     0
VMW_INV                   65        65p     0        0(    0,   0)     0
VMW_LUT2                  41        41p    82        0(    0,   0)     0
VMW_LUT3                   5         5p    15        0(    0,   0)     0
VMW_LUT4                   5         5p    20        0(    0,   0)     0
--------------------------------------------------------------
TOTALS:                  216       289p   444      433(   51,   0)   311
--------------------------------------------------------------
Total combinational area: 444   (macro: 311, non-macro: 133)
Total sequential    area: 433   (flip-flops: 51, latches: 0)
Total (comb + seqn) area: 877
```

Figure 34 Sample area report

Figure 35 on page 141 shows the RTL Compile Form.


VirtuaLogic3.5@sleepy: /home/subbiah/Train_Labs/V352/V35/color.vmw
File Edit View Tools Options Help
Design Import Signals Compile Triggers Emulation
RTL VLE FPGA
Primary Options
Allow
Control
Reports
Refresh
Log
RTL Messages ... VLE Messages ... FPGA Messages ...
Module Options
Edit...
Show ... Find...
Additional Options

Optimize: High
Debug: On

Incomplete Sensitivity Lists
Undefined Function Outputs
Multiple Drivers
'X' and/or 'Z' Comparisions
Gate Strength/Delay Specs

Compile RTL
Start

```
Rtl: Design Analysis
Rtl: Area Analysis
Log: Total
Log: Compile RTL
Log: Compile RTL & VLE &
```

Module Name	T	D	I	U	M	C	G
Hcounter	y	y	Y	n	n	n	

```
Read configuration /home/subbiah/Train_Labs/V352/V35/color.vmw
```

Figure 35 RTL Compile form

Note: For single button compile, The user must enter the required information on all the three (RTL, VLE, FPGA)forms and change the option in the Control pane to *Compile RTL & VLE & FPGA* and then click *Start*.

Primary options

The Primary Options pane specifies the optimization level and RTL source visibility of the RTL compilation process.

Optimization level

The optimization level determines how aggressively combinatorial logic is optimized. The primary goal of this optimization process is to minimize the gate area cost of the compiled netlist.

- **Off:** The Off optimization level switches the optimizer off altogether. The result is very fast compile times, but un-optimized netlists.
- **Low:** The Low optimization level performs very basic optimization. It results in reasonably fast compile time, but the compiled netlist is not optimal in terms of area.
- **Medium:** The Medium optimization level is a good trade-off between compile time and optimization. It invokes the logic optimizer in a semi-aggressive mode where a select set of optimization steps are performed, but time-consuming optimizations are sacrificed for speed. This is the default.
- **High:** The High optimization level invokes the logic optimizer in an aggressive mode where a full set of optimization steps are performed, resulting in a highly optimized netlist. Note that invoking the compiler with a high optimization level may result in large compile time. However, the additional RTL options switch -opt_timeout_limit is provided to prevent unreasonably large compilation time.

In general, the medium optimization level does a good job of optimization in a reasonable amount of time. Low optimization level is used for faster compiles when area cost of the netlist is not a constraint. High optimization level is used, when compile time is not a constraint(compiler run in batch mode) or if there is trouble partitioning or placing the netlist into the emulator.

RTL source debug

The debug option allows visibility into RTL-level signals and also allows RTL source level debugging, where breakpoints can be set in the RTL source code.

Note that it is possible to include or exclude certain modules from debugging mode. See *Module specific options on page 143*.

Note that the debug option can be specified simultaneously with the high optimization level. Debug mode typically results in about a 5% increase in gate count.

Simulation errors (Allow)

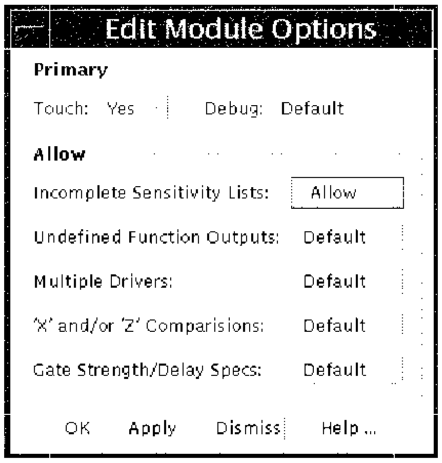
This section describes the areas where the netlist generated by RTLC-VLE may not confirm to exact RTL simulation behavior. It also describes the actions that RTLC-VLE takes on encountering such areas in the user's RTL code. For information on Sim errors, refer to *SimErrors on page 127*.

Module specific options

In addition to the Primary Options and Simulation Errors (Allow) options which apply to the whole design "globally", some RTL options can be set on a module specific level. The module-specific options are:

1. Debug/Don't debug
2. Allow/Disallow 4-state reads
3. Allow/Disallow multiple drivers
4. Allow/Disallow incomplete sensitivity lists
5. Allow/Disallow undefined function/task outputs
6. Allow/Disallow gate strength/delay specs

For example, if the entire design has to be compiled with debug information except for a few memories or IP cores, then you can turn debugging off for those modules using this pane and turn on global debug in the Primary Options pane. *Figure 36 on page 143* shows the *Edit Module Options* window.


Edit Module Options

Primary

Touch: Yes Debug: Default

Allow

Incomplete Sensitivity Lists: Allow

Undefined Function Outputs: Default

Multiple Drivers: Default

'X' and/or 'Z' Comparisions: Default

Gate Strength/Delay Specs: Default

Figure 36 Edit module options window

RTLC additional options

RTLC compiler switches can be classified into following categories:

- Design input switches
- Language recognition switches
- Output file switches
- Messaging control switches
- Selective compilation switches
- Debug and preservation switches
- Optimization switches

For syntax, usage notes and examples on RTLC options, refer to *RTLC Additional Options on page 251* in Compiler options reference guide chapter.

RTLC troubleshooting

1. **Compilation time**

If *rtlc-vle* is seen taking an abnormally large time to compile a single module, use the -opt_timeout_limit <seconds> option. The default value for this options is 10 minutes.

2. **Area cost**

If a single module is seen taking an abnormally large area, consider using the -opt_level 4, -res_share and -lut_map options. Also check if the -debug/-preserve options have been specified. Please report this problem to the RTLC team.

3. **Incremental problems**

If for any reason, *rtlc-vle* does not recompile a modified module, use the -force_module option and file a bug report with the RTLC team.

4. **Incorrect compilation/internal errors**

If there are some internal errors during compilation or if logic generation is incorrect, consider using the following options:

Option	Description
-import <module_name>	Disables compilation of the specified module

Option	Description
main <module_name>	Treats the specified module as a top-level module enabling compilation of a smaller hierarchical partition

Besides this, there are some internal options that are useful only to the RTLC team for debugging purposes. These are given here only for completeness.

Option	Description
-opt_level 0	Turns off combinatorial logic optimization
-no_sweep	Turns off a generic netlist optimization step
-no_drc	Turns off design rule checking
-dont_flatten	Turns off flattening of internally generated modules

RTL messages

A new window comes up with all RTL compilation warning and error messages as shown in the *Figure 37 on page 145*

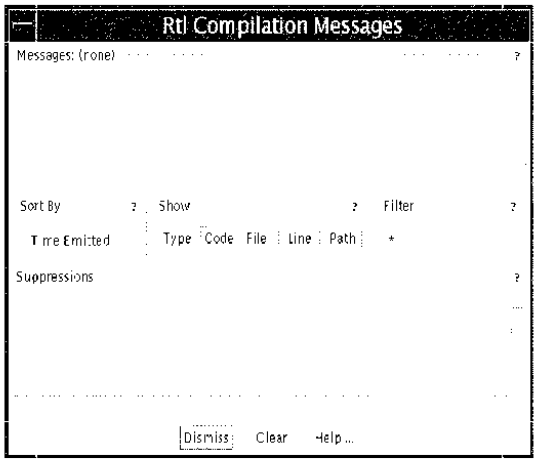


Figure 37 RTL Compilation messages

VLE/VSYN Compile

Target hardware pane

Emulation platform

The *Emulation Platform* options are *VLE-2M-IDS* and *VLE-5M*. VLE-2M-IDS stands for VLE-2Ms with internal data sampling.

Emulator boards

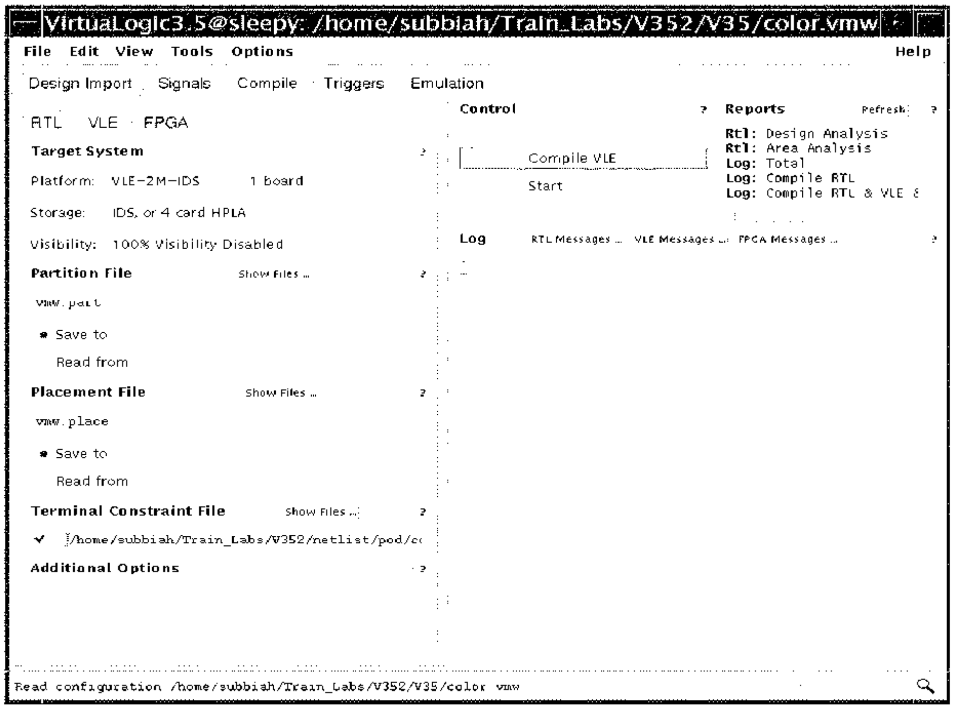
The *Emulator Boards* option allows the user to select from one to six boards. This number is indicating the number of Array Boards in the system.

Normally, the users should indicate the number of Array Boards in the emulator regardless of the number needed for the particular design, except when compiling the design for use in a *Multi-ASIC* configuration. In this case, specify the number of boards required for the design only. If this is not followed, the drawback is that the place and route is done for unnecessary FPGAs also.

Storage

The *Storage* option allows the user to specify the configuration for IDS. Select *Storage* to be *IDS, or 4 card HPLA* makes the event transfer occur faster.

VLE Compile form on page 147 shows the VLE compile form.


VirtuaLogic3.5@sleepy: /home/subbiah/Train_Labs/V352/V35/color.vmw
File Edit View Tools Options Help
Design Import Signals Compile Triggers Emulation
RTL VLE FPGA
Control
Compile VLE
Start
Reports Refresh
Log
RTL Messages ... VLE Messages ... FPGA Messages ...
Show Files ...
?

Target System

Platform: VLE-2M-IDS 1 board

Storage: IDS, or 4 card HPLA

Visibility: 100% Visibility Disabled

Partition File

vmw.part

● Save to

Read from

Placement File

vmw.place

● Save to

Read from

Terminal Constraint File

✓ /home/subbiah/Train_Labs/V352/netlist/pod/c

Additional Options

Rtl: Design Analysis
Rtl: Area Analysis
Log: Total
Log: Compile RTL
Log: Compile RTL & VLE &

Read configuration /home/subbiah/Train_Labs/V352/V35/color vmw

Figure 38 VLE Compile form

Compiler configuration pane

100% Visibility

The *100% Visibility* button is a toggle switch used to turn the feature on or off. The default behavior upon starting up a new configuration in *gvl* is to enable 100% Visibility. The 100% Visibility system automatically probes all primary inputs, clocks, and memory outputs. This automatic probing counts against the *Signals Window* and Core Probe limitations.

The 100% Visibility is a feature that allows any design signal to be viewed in the waveform trace. When 100% Visibility is enabled, the only signals that need to be explicitly probed are those needed for IDS trigger expressions. A subset of the signals are automatically probed by the compiler, and the remainder of the signals are reconstructed on demand.

The 100% Visibility provides access to values of all design nodes during a display time window. The duration of this display window is based on the capture depth of IDS and corresponds roughly to the duration achievable via current probe-based visibility. The position of this window is selected via triggering as it occurs for probe-based visibility.

In addition to enabling 100% Visibility, it is also necessary to manually select probes and signals windows for any signals which are to be used in triggers. All trigger candidates should be manually selected in order to ensure that they are available.

The 100% Visibility feature requires the use of the emulator out-of-circuit for the post-processing step. Refer to the page 233 for details.

The @all, @state, and @memory probes are ignored when 100% Visibility is on. However, it is up to the user to remove any probes that are not needed for triggering. It is recommended to minimize the number of explicit probe requests when 100% Visibility is on because significant IDS resources are required to make 100% Visibility work.

For information on @all, @state, and @memory, refer to page 115.

100% Visibility cable

To perform 100% Visibility extraction based on HP probing, an additional cable connection is required. A cable is supplied with a 26-pin male connector on one side and four BNC connectors (red, blue, green, and either white or black) on the other side. The blue BNC connects to the HP logic analyzer trigger-out port. The black or white is connected to the HP logic analyzer trigger-in port. The 26-pin connector connects to the bottom center of the emulator system board (bottom-most board). The additional cable is not needed when using IDS.

Refer to the VLE-5M Hardware Reference Manual for complete details on this cable.

100% Visibility benefits

When 100% Visibility is enabled, the user can see the value of any node in the design using *Virsim*. This is essentially equivalent to using Verilog-XL with *Virsim*, tracing all the logic values.

There is still a limitation that the user can only see the time window recorded by the IDS. Only specify the probe points that you wish to use in triggering, beyond the primary design inputs, which will be automatically be recorded by the IDS in support of 100% Visibility.

100% Visibility costs

Some of the capacity of the emulation hardware will be consumed by extra logic relative to running with 100% Visibility turned off. The exact overhead percentage should be small; however, with a given hardware configuration, there will be some designs that will not fit unless 100% Visibility is turned off.

The total cost is design dependent, typically averaging between a 10% and 15% size increase. Certain structures can be particularly expensive and may lead to more excessive costs. These include large storage macros implemented as gates, heavily latch-based design styles, large numbers and/or very high fanout cross-domain or asynchronous nets, particularly if the nets go to asynchronous preset or clear terminals.

Capacity utilization can be improved by the following:

- Modeling storage macros as memories
- Treating high fanout asynchronous inputs as synchronous
- Using the *-Sr* compiler option to convert asynchronous preset/clear modeling to synchronous preset/clear modeling
- Using quasi-static annotations on the highest fanout cross-domain nets if they are not already quasi-static

A runtime performance penalty beyond that otherwise incurred for any heavily probed design is not expected. There is an extra delay when uploading from the IDS following a trigger, beyond that incurred with 100% Visibility turned off. The estimated additional delay is up to five minutes depending on the number of samples recorded by the IDS. To reduce this time, user might want to use a faster host for the VIRSIM. During this time, the emulator must be taken out of circuit to assist with the analysis of the IDS data.

There will be an extra delay when starting *Virsim*, approximately one to two minutes, depending on design size. The user can "pipeline" this by invoking *Virsim* before triggering occurs or during upload.

100% Visibility restrictions

Following are the situations when 100% Visibility may need to be deactivated:

- When the capacity costs are too high
- When the user cannot accept the use requirements of an emulator out-of-circuit for processing of the IDS data
- When some constructs in the design prevent 100% Visibility from working properly

- When using conditional capture on the IDS (refer to the section below for details
- When the available FPGA capacity is reached

Conditional capture

100% Visibility feature does not support conditional capture. Use of conditional capture, (*Store Nostore*), in triggers is not supported. The 100% Visibility only operates on a contiguous time window and this window needs to include a small amount of time prior to the trigger in trigger at window beginning mode or a large amount of time prior to the trigger in trigger at window end mode. Do not use Store/Nostore in triggers for 100% Visibility mode display.

Data from a 100% Visibility compiled signals window can be used in non-100% Visibility mode, and under these circumstances the user can use conditional capture in the trigger.

Partition file pane

To save or read a partition file type in a filename or click on the *Show Files* button and select a file from the list to overwrite an existing file.

Partitioning results can be saved and used in a later run as long as the set of design modules, top-level I/O, and libraries do not change. To read in a previously written partition file, use *Read from* button.

On subsequent runs, select the *Read from* button with the existing filename. The *Partition File* information is saved from run to run, along with the rest of the configuration.

Placement file pane

To save or read a placement file, type in the filename or click on the *Show Files* button and select a file from the list.

The *Save to* button is used to save placement results to a file, specified in the field. Placement is a somewhat time consuming part of compilations. Placement results of previous runs can be saved and used in later runs as long as the set of design modules and libraries do not change.

On subsequent runs, select the *Read from* button with an existing filename. The *Placement File* information is saved from run to run, along with the rest of the configuration.

Terminal constraint file pane

The *Terminal Constraint File* pane is used to indicate which design I/O signals are mapped to which physical emulator pod-pins so that a hardware testbench can be wired to the emulator to maintain consistency across design changes.

This file is created with a text editor. Each line in the file has the design I/O name and the emulator terminal name in the form *Jnnn.nn*, where *n* is a digit.

Create a file, *vmw.pod*, that specifies the emulator terminals to which to connect the design I/Os. The *Terminal Constraint File* lets the user control the binding of design I/Os to emulator terminals so the design I/O pinout can be matched on the emulator to the pinout of the target system.

Following is a fragment from a pod constraint file (# is a comment character).

```
# design terminal name
#emulator terminal name
clk     \J100.41
a[6]    \J111.63
a[5]    \J111.13
a[4]    \J111.62
a[3]    \J111.12
```

If you do not specify a pod constraint file, one will be generated during compile, called *vmw.pod* using arbitrary I/O to pin assignments. The file can be edited to setup the desired pin constraints and used in a subsequent compilation by specifying the filename in this field.

Compiler options pane

The *Compiler Options* pane allows the user to specify options to the compiler. For compiler options, refer to *VLE Compiler Options on page 271* in Compiler options reference guide chapter.

Improved emulation performance

Lazy reset is a modeling style for asynchronous preset/clear behavior of state elements that trades off precise timing behavior for improved emulation performance. When enabled, lazy reset defers asynchronous preset/clear induced output changes on state elements until the next active clock edge.

The user can selectively exempt state elements from lazy reset which allows imprecise modeling for most state elements while retaining the more precise modeling for the selected set.

Save the name(s) of the module(s) to a file and invoke the *-NoSrfi* switch with this file's hierarchical prompt on the command line.

Refer to *-NoSrfi on page 293* for additional information.

No-Flows for modeling

No-flow annotation is a command-driven mechanism for indicating to the VirtuaLogic compiler that some apparent combinational path within a design is really a false path. To make this indication, the user must identify a net or nets within the design that split a false path into one or more disjointed real paths. Accomplishing this split enables compilability or improves performance.

No-Flows to compile a design

Large combinational cycles or combinational cycles with multiple top-level I/Os make it impossible to compile. The VirtuaLogic compiler automatically handles most combinational cycles with few exceptions. When an unhandled cycle exists in the design and the cycle is due to a false path, the user can break into the cycle using a no-flow.

Unsupported cycles include the following:

- Cycles whose output reaches state clocks or gates
- Cycles involving top-level bidirectional I/O
- Cycles involving memory elements
- Very large cycles (in excess of 100 elements)

False paths almost always result from overly conservative treatment of flow through latches. To remove false combinational cycles, add no-flows on the output of key latches or add no-flows on top-level I/Os. Use no-flows after the compile process has indicated something wrong (but has not identified the source).

No-flows to improve emulation speed

The user can break multicycle paths with no-flows to give the compiler multiple cycles in which to evaluate multicycle logic.

Acyclic paths in the design that receive multiple cycles to propagate may limit performance in the VirtuaLogic, since by default, the emulator propagates data along the entire path in a single clock cycle. In such cases, the user can annotate an arbitrary net along the path (ideally the midpoint), using the no-flow mechanism. This lets the VirtuaLogic compiler use two cycles for dataflow on the path. (Annotation of two nets provides for three cycles of propagation time, etc.)

This technique is very similar to the introduction of an explicit pipelining register; however, it occurs without netlist modifications.

Figure 39 on page 153 illustrates using no-flow to break multicycle paths.

Assume the adder has two clock cycles to compute outputs S32-S64. Marking Co31 as a no-flow conveys this information to the VirtuaLogic compiler. The effect is similar to what would occur if the design contained an explicit pipeline register on net Co31. (However, this implicit registering is not guaranteed to occur. It occurs optionally if it improves performance.)

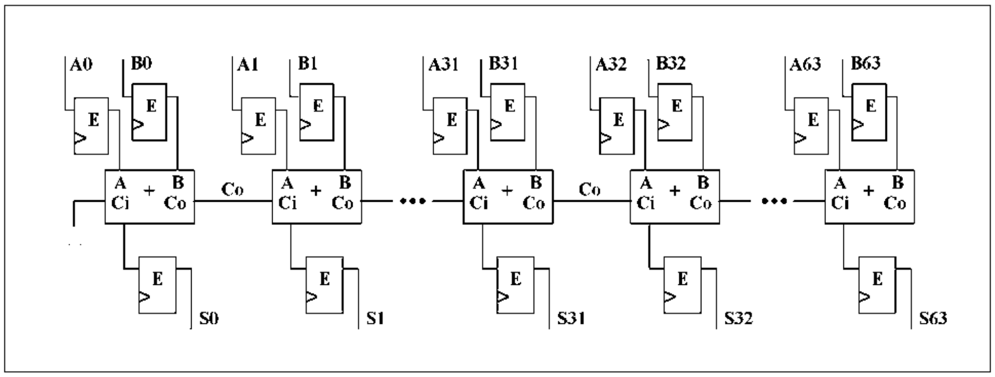


Figure 39 No-flows to break multicycle paths

Special no-flow semantics for bidirectional top-level I/Os

If a top-level bidirectional I/O net is marked no-flow, the no-flow nature only applies to internal flow paths.

No-flow annotations on top-level bidirectional I/O nets break the flow path from outwardly directed data to inwardly directed data, but maintain paths between the output driver and I/O as well as I/O to inward data. *Figure 40 on page 154* illustrates this.

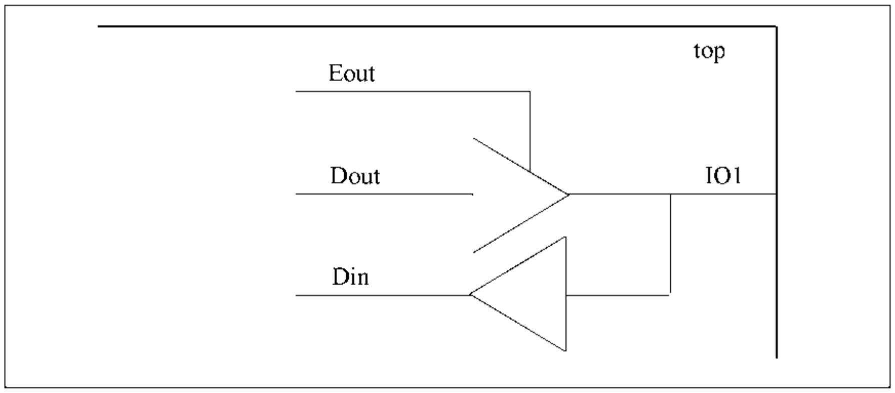


Figure 40 No-flow bidirectional I/O net

Only paths from Dout to Din and Eout to Din are considered false paths. Dout and Eout to IO1 out and IO1 in to Din are considered real paths. Enter the following:

```
Net top.IO1
```

Visibility for bidirectional I/Os with no-flows

By default, the IKOS emulation compiler models paths which go out primary bidirectional IOs and then come back into the circuit. In some cases, these are false paths. These paths can be annotated as being false paths by using a no-flow on the bidirectional IO which has special no-flow semantics.

The result of using bidirectional IOs is that the single IO wire really has two potentially distinct values on each cycle, an outbound value and an inbound value. The outbound value is the final value that the external wire will take on at the end of a clock cycle. The inbound value is the pin value used for internal computation. The inbound value has a delay of one cycle before reflecting any change due to a contribution from the design internals as a result of the no-flow. Contributions from outside the device show up with no delay, as do model internal values as viewed from outside the emulation model.

Historically, the value displayed when viewing a bidirectional IO with a no-flow was the outbound value, which reflects the final resolved value for the wire, accounting for current internal and external contributions.

VirtuaLogic 2.1 has changed the display for this wire to correspond to the inbound value, where again, the inbound value corresponds to the resolved value of any current external stimulus with the internal driven value from the prior clock cycle.

Stated differently, when viewing or triggering on a bidirectional IO which has had a no-flow annotation, the data will appear with a one cycle delay relative to the external value, corresponding to the value which will be used for the cycle on feedback paths inside the emulator. This is of particular note for triggers, which may need to be adjusted to reflect this delay if they use expressions which combine bidirectional IO values which have received no-flows with other IOs which are not no-flowed.

Using No-Flows

To convey no-flow information, create a *vmw.nfl* file consisting of lines with the following syntax:

```
net <net_expression>
terminal <terminal_expression>
net-state <net_expression>
```

In the above entries, *net_expression* is a hierarchical netname or regular expression matching a collection of nets using vector, synthetic vector, or wildcard notation. The *terminal_expression* is a hierarchical terminal name or regular expression where:

- The first components(s) of the hierarchical name match the name of one or more instances
- The final component identifies a terminal name on the instance, for example:

  a.b.c<2:0>.QN

- Identifies the collection of nets connected to the QN terminal of instances a.b.c0, a.b.c1, and a.b.c2

The *net* keyword indicates that any path between the source(s) of the net and all its destinations are false paths.

The *terminal* keyword, if applied to a terminal that drives a net, is equivalent to the *net* keyword. If applied to a terminal that is a net receiver or fanout, *terminal* indicates that paths between the net's driver and the specified fanout are false paths, but paths involving other net fanouts are not affected. (Terminal expression must match a terminal on a user-primitive instance only.)

Almost all no-flow situations use *net*; there is only one specific legal use for *net-state*. The *net-state* can be used to break flow paths between sets of latches whose gate active regions are mutually exclusive.

No-flow nets may insert full design clock cycle delay. To eliminate these full clock cycle delays, use the *-PUi* option. Refer to *-PUi on page 280* for information.

No-Flows on Buses

If you do not expect to transmit and receive in the same cycle, then no-flow should work even if you get a cycle delay.

No-Flows in Combinational Loops

Placing no-flows in combinational logic loops does not guarantee whether there will be a full cycle delay or not. It depends on how the partitioner partitions that part of the logic. However, if you do not want a cycle delay in the combinational logic loop, use the *-PUi* option and place the devices in the combinational logic loop in one partition. This will guarantee no cycle delay with the no-flows in that loop.

Refer to *-PUi on page 280* for additional information.

No-Flow at output of latches

Transparent latches make up combinational loops. The effect of placing a no-flow at the output of the transparent latch is the latch is replaced by a flip-flop. Usually, it does not matter; however, if it does, then use the process described in *No-Flows in Combinational Loops on page 156*.

Net tie-offs

Net tie-offs allows the user to fix the value of a particular net or terminal at a constant value which overrides the natural value that the design produces. Overriding such a value can often be useful for the following:

- Work around a design problem
- Disable test modes or logic

With net tie-offs, the user can achieve these ends without editing the netlist source.

To use net tie-offs, create a file with the following syntax:

```
Net <net_expression> <value>

Terminal <terminal_expression> <value>
```

The *value* is a constant Boolean value, either 0 or 1. For the definition of *net expression* and *terminal expression*, refer to *Using No-Flows on page 155.*

If a net or terminal expression yields multiple nets, they are all tied to the single specified value. If a net is identified using the *net* keyword, or the net-sourcing terminal is specified with the *terminal* keyword, all fanouts of the net receive the specified value.

If the *terminal* keyword identifies a net destination or fanout, only the identified fanout receives the specified value while all other fanouts receive the value that the design naturally produces.

Designs with multiple asynchronous clocks

The VirtuaLogic system supports up to 14 asynchronous clocks. Designs with two or more asynchronous clocks may require special care in compilation, depending on the interactions between the clock domains in the design being compiled.

Script driven activities

This section describes the functionality of options for running the compiler with scripts.

Script driven generation of virtualized model

A virtualized model can be generated without using the VirtuaLogic graphical interface (*gvl*).

In addition to the graphical mechanism for invoking the VirtuaLogic compiler, there is a script driven mechanism. The scripted mechanism uses a script that the *gvl* automatically creates. The user can repeat the process through the VirtuaLogic compiler without the need for any user intervention.

To generate a virtualized model for your design, type the following command:

```
vlc <config_name> verify
```

In the command line, *config_name* is the name of your configuration.

If the directory wanted is the one you are currently working in, enter the following:

```
vlc . verify
```

Script driven design compilation

A design can be compiled using the VirtuaLogic compiler (*vlc*) script.

In addition to the *gvl*, a script to invoke the *vlc* can be used. This mechanism uses a script that the *gvl* automatically creates. Follow these steps:

1. From the current directory, change to the working directory from which you originally ran the *gvl*.

2. Type the following command to generate a configuration image for your design:
   ```
   vlc <config_name> compile
   ```

   where <*config_name*> is the name of the configuration to compile.

Script driven Place and Route

A downloadable emulation model can be produced using a script.

The textual interface is useful when a graphical terminal is not available; for example, over a modem from home. Following are the steps to do this:

1. From your current directory, **cd** to the configuration directory.

2. Type this command to generate a configuration image for your design:

   vlc <config_name> vtask *[VirtuaLogic 3.5]*

   where *config_name* is the name of the configuration to compile.

vlc commands

The rtl, verify, compile, and vtask are the three basic steps to a configuration process. The other functions are provided for archiving a design database.

For syntax, usage notes and examples on vlc commands, refer to *vlc commands on page 318* in Compiler options reference guide chapter.

VRC

vlc . browse_constants

The command *vlc . browse constants* can be used to bring up *vrc* in order to see the value of netlist constants. This includes the values of any inherent netlist constant nets, any nets tied to constants using the *-TNfi file* tie-off capability, and the zero values applied by default to undriven nets in the netlist. Refer to *-TNfi on page 290* for additional information.

Incorrect net value in the circuit

SYMPTOM:

The net is always 0 when looked at and the user expects it to toggle or the net is stuck at 1 when the user expects a 0.

LIKELY REASONS:

- Incorrect tie-off in the tie file
- Floating net

DIAGNOSIS:

Use *vlc . browse constants* to determine if the net is actually determined to be a constant during compilation. If so, the user can trace back the source of this constant in the value-annotated path display window to find why it is a constant.

Design removal during second dead logic elimination

SYMPTOM:

The design gets mostly removed during the second pass of dead logic elimination.

LIKELY REASONS:

- Floating or disconnected active low asynchronous preset or clear in library for some state element model
- Floating or disconnected clock in the library for some state element model
- Mistake in the internal clock specification leading to a dangling net at the root of the clock tree
- Bad tie-off in the clock tree

DIAGNOSIS:

Select a flip-flop that you think should not go away. Observe its clock and asynchronous preset/clear using *vlc . browse constants*.

If the first flip-flop selected is correct, it is probably deleted because it feeds into flip-flops having one of the problems above. Select another flip-flop and repeat

Running repeat configurations

As a configuration of an entire design is completed using the GUI, all user inputs are saved to files in text format. A configuration can be acted upon again using a batch program called *vlc*. If any user inputs need to be changed, the input files can be modified to reflect the changes and the entire configuration can be run through a batch file.

Vprobe batch-mode

Vprobe is a process to facilitate the probing process downloading the triggers, uploading the data from the HP logic analyzer etc.. Basically anything that can be done using the GUI can be done using batch mode. 100% visibility is not recommended using vprobe at this time.

To start the process, type vlc . vprobe. This will start vprobe and mount the HP logic analyzer(if applicable), there will be a list of commands as shown:

```
Starting vprobe

Commands: depth download exit lock quit unlock xwinon
```

Those commands are rather self-explanatory. Depth is to set how much data to capture while downloading the trigger. After downloading the trigger the command set becomes:

```
Commands: exit lock quit stop unlock
```

Typical command to be used here is stop (to stop the trigger), or simply wait for the trigger condition to occur. After the trigger condition is done, the command set is:

```
Commands: depth download exit lock quit run unlock upload xwinon
```

Here the command *upload* is used to upload the waveforms. To view the waveforms, it is strongly recommended to use the *GUI* because there is no good reason to do that with batch mode. In order to get the correct syntax in vprobe, one should run it once with the gui and view the emulate.log. Determine which line is the line for those commands stated above and use that as an example for the correct syntax.

VRUN batch-mode

Vrun is a program to run the emulation process such as connecting to the emulator, downloading the design etc. There is one serious draw-back in this mode in the sense that there is no straight-forward way to provide feedback from the emulator. This way it is difficult to determine when to load in a new sets of vectors. Two possible work-arounds are described later in this section. Here are the procedures for running scripts in vrun. Again, it is recommended that one first performs vrun in gui and examine the emulate.log for proper syntax.

First, rsh to the host of the emulator. Then type:

```
vrun -s script_file
```

to run vrun in batch mode. A sample script-file is as follow where comments are indicated by {}:

```
connect 1 {to connect to emulator, usually 1 unless there are more than 1 emulator connect
  to the host}
```

configure top_mod {top-level module} -probe probe_window_0.pbw/system0 {probe-window that you want to use, follow by /system0}

```
enable {enable the pods}
```

set_output4 {toggle the user-bit4, in most cases, user-bits are used as resets. Similarly, user-bits 1, 2 and 3 can be set by set_output1, set_output2 and set_output3. The first toggle changes it from 1 to 0}

```
set_output4 {toggle again change it back to 1}
```

system /bin/sleep 60 {system command allows you to execute regular shell-script. However, you will need to specify the entire path such as /bin/command. This command is extremely useful in reloading the memory. Since the memory-contend file is fixed, we will need to change the file by either mv or cp. This can be done by the system command. This is one of the workaround for the memory loading problem described earlier. We can run the design once to get an idea as of how long it will take to finish one set of vectors. Then we use the sleep command to wait for the vectors to finish before going to reload the new set of vectors}system /bin/echo "It is working" {more example about system command for executing shell-script, you can use as many as desired}

system -timeout 60 /bin/csh filename {The first part is to time-out after 60 seconds. The timeout command is purely optional, the second part is very useful. One can have a separate shell-script file to specify different commands and run it within vrun}
disable {disable the pods}

wait_for_file filename {wait for a file to be generated. One need to specify the full path of the file. This is another useful way to work-around memory-loading problem. Vprobe can set up trigger condition, and upon triggering, it generates a .vrc file and a .vdb file. If we set the trigger condition to be at the end of the current vector set, we can use wait_for_file to wait for the generation of those files as an indication of a new set of vectors is needed}

memory all {upload all memory, or specify explicit instance in full hierarchical format} -file test {filename where the memory content needs to go} -format X {X for hex, u for unsigned decimal, d for signed decimal and b for binary}
reload {reload memory content}
quit {to quit out of vrun}

The vrun batch mode is also useful when a user needs to run emulator from a remote site with telnet. However, always make sure that the environment is set-up properly, for one mistake that hangs up the system will make resetting the system impossible.

Creating a new configuration database

If a design bug is discovered and a new netlist is generated, then a new configuration will be required. The user will want to retain the current database until the new one can be tested and verified. The user can copy the current database files that contain the setup information by completing the following steps:

1. Open the old design with *gvl*.
2. Select *Save as* from the bottom panel of the GUI.

   In the dialog box, enter a new *working-dir config-name.vmw*

   Do not copy emulation bits (Button in the "out" position)

   Copy auxiliary files (Button in the "in" position)
3. *cd* to the new working directory.

4. Make the needed changes (copy new netlists, edit *params.mak*, or add new probes)
5. Compile the design using the *vlc* command or GUI. Do a touch *vmw.depend* to activate the batch mode compile.

Suggestions for repeat configurations

- Keep all the netlists in a separate netlist directory along with memory files or other data. This makes it easy to copy all the netlists between compiles.
- Always make the netlist names the same and keep track of them with date/time stamps
- Whenever a netlist is copied, use the *-p* option with the copy to retain the date/time stamp
- Always use the same config-name. Make it the same as the TOP name in the design. This makes it easier to write generic scripts.
- Keep a log in a global *README* file of what changes with each configuration
- If possible, use a revision control system to control netlist versions

Configuration input files

The configuration input files are described in *Table 3 on page 163*.

Table 3 Configuration input files

File	Contents
params.mak	Design input data from all forms, including netlist file names, compiler options, hardware resources and all settings
vmw.clk	Timing specification data
vmw.mem	Memory mapping data from Memory Specification form
vmw.defines	Verilog macro define statements from Netlist Import form
vmw.pod	I/O pin out, also called terminal constraints
vmw.prb	Probe Groups defined in the Probe form
vmw.pwl	Probe windows defined in the Analysis form

Refer to *vlc commands on page 318* in the Compiler Chapter for complete information on the commands.

Runtime state read/init/force

A user state element can be read/initialized and forced in VStation. It is currently only available from **vrun** and is not yet available from TAPI. To use this feature with co-model designs, the user need to bring up the design under TAPI_DEBUG mode and control **vrun** explicitly from *gvl*.

Terminology

Read:	Sample the state elements in the emulator and write them to a disk file. This can be done in-circuit.
Set:	Force 1 or more state elements to specified values, run one complete user clock cycle to propagate fanout through VirtualWires, and then release the lock on the state elements.
Force:	Force one or more state elements to specified values and leave them forced until explicitly released. (This is also called **stick** in some circles).

The **vrun** force capability requires a compile using the vsyn compilation option: *-force filename* , where the filename looks like an *xftl* file, specifying either flop module paths or q-net paths (wildcards are acceptable). If every design flop is made forcible (e.g. with MODULE root) then an overhead for force around 10 to 20% can be anticipated beyond that already incurred for 100% visibility. This has not been fully calibrated yet. Enabling of forcing for flops in a subsystem will result in significantly smaller overhead.

New vrun Commands

The following table gives the new vrun commands and descriptions.

Table 4 New vrun Commands

Command	Description
state_ctrl -ones	Sets all state elements to 1
state_ctrl -zeros	Sets all state elements to 0
state_ctrl -read_all filename [-format hdob]	Writes values of all states to file
state_ctrl -random seed	Sets state elements to randomized values; where seed is an integer between 0 and 31

Table 4 New vrun Commands

Command	Description
state_ctrl -set_states filename	Sets state values listed in the file, without changing the values ofother states
state_ctrl -set_all filename	Sets all state values. Zeros: any state elements that are omitted from the file, after warning the user
state_ctrl -force_states filename	Forces state elements to retain the values specified in the filename until explicitly released
state_ctrl -force_off	Release all forced state elements

- All operations that set state require that the I/Os are disabled first.
- All state input files use Verilog notation for vector values. (e.g. a[3:0] = 4'd7).

vrun has abilitiy to reconfigure memories with previous memory upload

vrun can save user memory contents and restore them to the previously saved user memory contents.

Table 5 Saving and restoration of memories

Switch	Description
memory all -file dir_name	saves the user memories into the *dir_name* directory with one file per user memory saved in the dir_name directory
reconfigure -use_dir dir_name	restores the user memory contents from the *dir_name* directory

The saving and restoration of memories work on all user memories when the emulator is out of circuit.

Ways to improve compile time

Front End Compile(vlc .compile)

Workstation

There are two workstation upgrades that will significantly impact performance:

- Faster machine - faster compile with a near linear relationship between speed of machine and compile time
- Increase memory to the point where swapping to disk is eliminated - can yield orders of magnitude faster compile times

Software

There are two software options that can be exercised to expedite compile time:

- re-read partition and place files
- use multi ASIC compile

To re-read partition and place files, use -Pfi/-Pi compiler options only when minor design changes have been made(i.e., a buffer changed to an inverter, or AND gate changed to OR gate).

To use multi ASIC compile, compile blocks of your design in seperate vmw configuration and bring in these blocks as ASICs in multi ASIC compile when your design can be partitioned easily and the "blocks" portions of your design are stable.

Place and Route (vlc .vtask)

Workstation or PCs

There are three ways to improve place and route times:

- use more, faster machines
- use PC farm
- increase memory on machines in #1, #2 such that swapping is eliminated or at least significantly reduced

VLE messages

A new window comes up with all VLE compilation warning and error messages as shown in the *Figure 41 on page 167.*

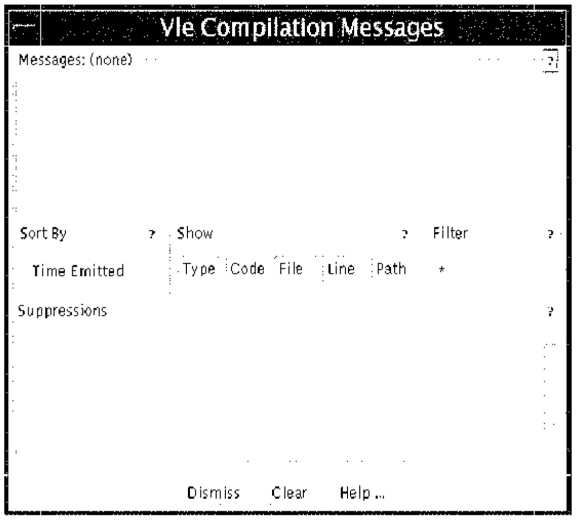


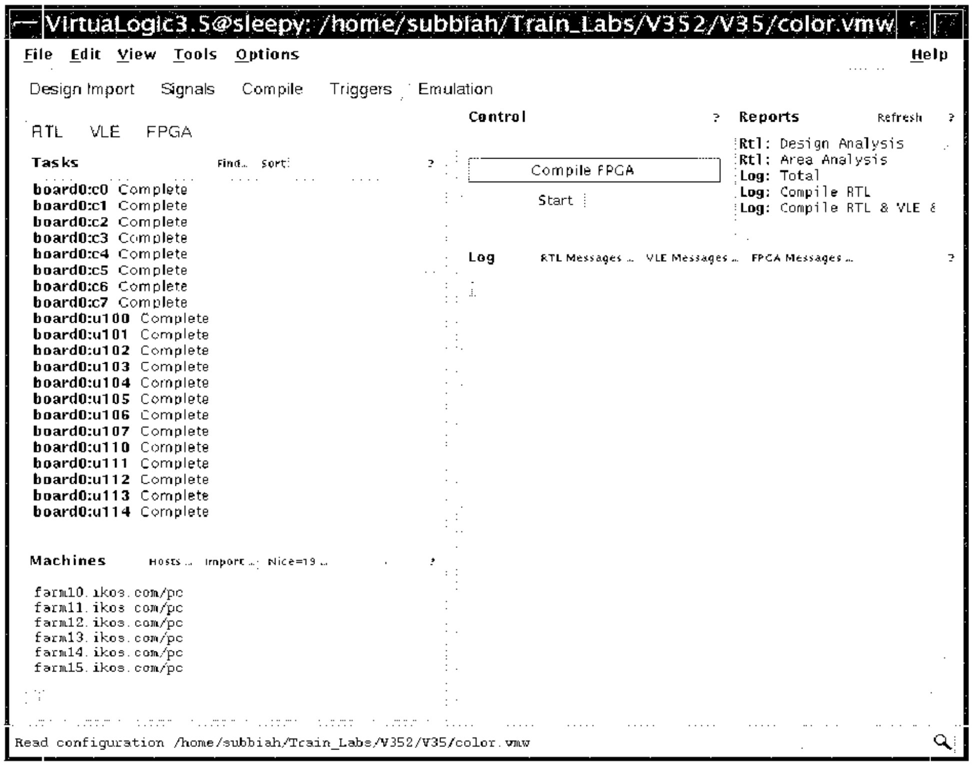
Figure 41 VLE Compilation messages

FPGA Compile

The compile process generates netlists and timing constraints for all of the FPGAs in the system. These FPGAs must be compiled into a downloadable bit stream before emulation begins.

FPGA Compile completes the place and route of all the FPGAs on the Array boards and the System board. FPGAs containing a lot of logic may be quite time consuming to complete FPGA Compile. To reduce this task, the backend FPGA software distributes the FPGA Compile jobs to machines on the network. The user can have as many machines as there are chips. By placing the machines in order from most powerful to least powerful, the more complex chips will be assigned to the more powerful machines. If a machine is known to be very busy, or has a high load, do not use it. The backend FPGA task will take the time of the longest chip if there is one machine per device. If there is less than one machine per device, then at the completion of a device, the machine will start another job.

Figure 42 on page 168 displays the *FPGA Compile* Form.


VirtuaLogic3.5@sleepy: /home/subbiah/Train_Labs/V352/V35/color.vmw
File Edit View Tools Options Help
Design Import Signals Compile Triggers Emulation
RTL VLE FPGA
Control ?
Reports Refresh ?
Tasks Find... Sort ?
Compile FPGA
Start
Log RTL Messages ... VLE Messages ... FPGA Messages ... ?
Machines Hosts ... Import ... Nice=19 ... ?

```
Rtl: Design Analysis
Rtl: Area Analysis
Log: Total
Log: Compile RTL
Log: Compile RTL & VLE &

board0:c0 Complete
board0:c1 Complete
board0:c2 Complete
board0:c3 Complete
board0:c4 Complete
board0:c5 Complete
board0:c6 Complete
board0:c7 Complete
board0:u100 Complete
board0:u101 Complete
board0:u102 Complete
board0:u103 Complete
board0:u104 Complete
board0:u105 Complete
board0:u106 Complete
board0:u107 Complete
board0:u110 Complete
board0:u111 Complete
board0:u112 Complete
board0:u113 Complete
board0:u114 Complete

farm10.ikos.com/pc
farm11.ikos.com/pc
farm12.ikos.com/pc
farm13.ikos.com/pc
farm14.ikos.com/pc
farm15.ikos.com/pc

Read configuration /home/subbiah/Train_Labs/V352/V35/color.vmw
```

Figure 42 FPGA Compile form

Machines

The *Machines* pane is used to specify a list of Solaris workstations and PC hosts on which FPGA Compile jobs can be queued.

A single Place and Route(PAR) job typically takes 20 minutes and one of these is required for each FPGA. There are 64 FPGAs per VirtuaLogic emulation board, in addition to the FPGAs dedicated to probe logic. This can result in about 400 PAR jobs in a full compilation for a six board system.

It is advantageous to use as many machines as are available. The FPGA compilation software distributes the FPGA Compile jobs to machines on the network. The machine names are specified in the *Machine List* pane, one machine name per line.

Refer to *All known hosts on page 170* for information on how to obtain a list of machine names on the network. Note that routers, file servers, and non-Solaris hosts will all be listed. It is the users responsibility to choose the correct machine names from the list.

The *Machines* pane is read when a PAR run is started or reread with the *-newlist* command. The user can change this machine list while the PAR jobs are running. When the editing is completed, select the *Reset Host List* button. The Solaris machines must be able to see the *SVMW_HOME* area where the VirtuaLogic software is installed, and it must support the *rsh* facility.

The PCs must be configured with the RSH daemon. After this has been completed, the Xilinx and VirtuaLogic software can be downloaded from Solaris as follows:

```
$VMW_HOME/pc_setup.csh pchost1 pchost2 pchost3 ...
```

Refer to page 371 for information on how to obtain and install the RSH software.

The list of known hosts is updated every few minutes, along with the load averages for UNIX machines, to help the user in selecting unloaded machines.

Remote machine resources

To determine the resources available on remote UNIX machines, rlogin into the machine and check it for the following resources:

- memory

```
dmesg | grep mem
```

- CPU type/class

```
dmesg | grep cpu
```

- Each machine must have a minimum of 50 MB tmp space to be able to run an FPGA job. To check the tmp space,

```
df /tmp
```

Note that the Xilinx tools used for FPGA Compile do not support SunOS. To successfully complete an FPGA Compile, the remote machine must be able to *cd* to the software location and the *config-name.vmw* location. Make sure these disks are mounted and accessible from every remote machine.

```
ls -l `which ppr`

ls -l full-hierarchical-path-name/config-name.vmw
```

When logging into a remote machine, if it asks for a password, then the FPGA Compile software will not be able to run on this machine. To resolve this, a file can be added in the users home directory called *.rhosts*. Place the machine host names that are going to be used for the FPGA Compile in the *.rhosts* file. This will allow these machines to be logged into without requesting the user's password.

All known hosts

The *All Known Hosts* pane shows the available hosts on the network. It can be used to drag and drop machine names into the *Machine List* which are the machines used to run PAR jobs.

gvl determines the available hosts through one of the following three methods:

```
rup
```

This method is preferred. It gives only the UNIX hosts that accept the *rup* protocol and updates the host list and the load averages once every two and a half minutes.

It requires that the system administrator enables the *rstatd* daemon which is very common.

This method typically works; however, it is slow and it takes resources since it does a network broadcast to get hosts to send their load average information.

```
ypcat hosts
```

This method works in the yellow pages environment. It offers no dynamic update and no load average information. It does not distinguish between UNIX and non-UNIX hosts.

```
cat /etc/hosts
```

This method is the last resort option if the yellow pages method does not work.

The PC hosts listed from *$VMW_HOME/env/pc_hosts.mach* are included. This file is updated when a new PC host is configured using the following script:

```
$VMW_HOME/bin/pc_setup.csh
```

Note that when dragged and dropped into the *Machine List*, the PC machines will have the *pc* appended.

Niceness

The *Niceness* slider controls the priority of the parallel PAR jobs that are spawned on UNIX machines across the network as specified in the *Machine List*.

The default priority is 19 which is the lowest (nicest) priority. The slider adjusts the priority of all new PAR tasks that are spawned from the configuration even if the FPGA Compile run has already been started. It does not change the priority of individual PAR jobs that are already in process.

The highest priority available is 0 which is what normal user jobs run at. If possible, do not run jobs in this highest priority mode.

The user cannot set the priority higher than 0 without being a super-user.

A single PAR job run at 19 should not be noticeable to a workstation user unless the physical memory on the workstation is very tight. The PAR jobs generally take less than 32 MB on a UNIX and 75 MB on a PC.

FPGA compile tasks

The *FPGA Compile Tasks* pane lists all the Xilinx PAR tasks which must be completed before emulation can begin. They correspond to the FPGAs in the emulator box.

Each task goes through the following four phases:

- Prepping
- Ready
- Running

- Completed

The *Prepping* occurs on the machine from which the user started the PAR run and is typically an I/O intensive task. To avoid overly stressing the file server, only one *Prepping* is run at a time.

The *Running* occurs on the machines given in the *Machine List*. They are run in parallel.

gvl back-annotates onto the list a task's phase and the machine it is running on.

More information can be obtained about a task by double-clicking it or selecting it and pressing <*return*>. This will show a list of all the files related to the task and the file size in kilobytes. Double-clicking on a file will bring it up in a text editor.

The files are summarized as follows:

***.log**	log files for the Xilinx tools; look at these if there are errors
***.bit**	raw bits for the emulator download (binary)
***.map**	Xilinx data file (ascii)
***.xff**	Xilinx data file (ascii)
***.dld**	Compressed bit files for the emulator (binary)

Reset host list

Use this button after editing the *Machine List* pane. The *Machine List* can be edited while a PAR job is running. Press the *Reset Host List* button to reload the *Machine List* after editing.

Stopping FPGA compile during compilation

Either of the following will stop FPGA Compile during a compilation:

- Select the *Interrupt* button
- Type <control-c> (or equivalent break) in the FPGA Compile log pane

Either of these methods will allow the FPGA Compile run to shutdown gracefully, including the proper termination of any active processes on remote machines. In contrast, performing a **kill -9** targeting the *vtask* process can leave remote processes running and therefore, should be avoided.

Run FPGA compilation from the command line

Do the following to run a FPGA compilation from the command line:

1. *cd* to the configuration (*.vmw*) directory
2. Enter vlc . vtask

To run FPGA compilation from a script, you must redirect the input and output of the program. The syntax is as follows:

```
vlc . vtask < /dev/null >& pprs.log
```

Task management

The FPGA compile task manager, *vtask*, is an interactive program and can be controlled from the *FPGA Compile Log* pane or from the command line if you are running from a UNIX shell.

vtask commands

For syntax and usage notes on vtask commands, refer to *vtask commands on page 321* in Compiler options reference guide chapter.

Hung jobs

The FPGA compile task manager, *vtask*, is an interactive program. Entering the *status* command on the command line will report what jobs are running on each machine. Do the following to determine if jobs are hung:

1. Use the *status* command from *vtask* to determine which machines are still running
2. Login to the machines still running. Use *top* or *ps* to determine if the job is getting CPU cycles. Following are reasons why a job is not getting CPU cycles:

- Other jobs on the machine have priority. By default, the jobs are run at *nice 19*. If the last job(s) are on the busiest machines in the company, interactively remove the machines at the *vtask* command line or kill *vtask* (control-C, not kill -9). Edit the *machlist.mach* file to place the least loaded machines first on the list and then restart PARs.
- A network or disk problem. The job may be slow because the network is slow. The user should *cd* to the directory containing the FPGA data (e.g., board0) and do a *ls -ltr* on the data for that FPGA. If the log file has been written recently, the best solution may be to just wait.

If the job is getting no cycles and the log file has not been written within the last hour, get the offending machines out of the *Machine List* and try to reboot the machines.

vtask command

SYNTAX:

```
vtask.sh <number of Array boards> [options]
```

OPTIONS:

-fC	This continues compiling FPGAs even if there is an error in one of them. The default is to abort.
-newlist <n>	This rereads the machine list every <n> seconds so that a global resource manager (e.g., LSF) can control the host resources used by *vtask*.

USAGE NOTES:

The *vtask.sh* command has many other options which are not listed because they are set automatically from the *vlc* script. The above two options are the only options that the user should change.

FPGA messages

A new window comes up with all RTL compilation warning and error messages as shown in the *Figure 43 on page 175*.

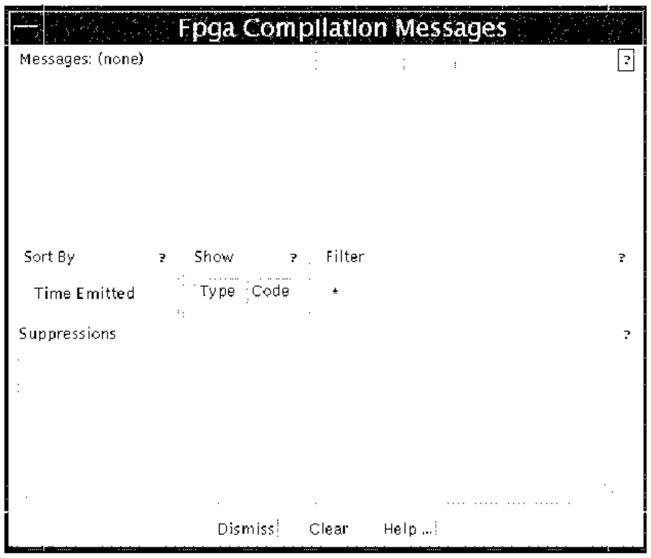


Figure 43 FPGA Compilation messages

Control

The Compile Control pane of the Compilation tabcard is used to start or interrupt a compilation run.

There are three main stages of compilation: RTL, VLE, and FPGA. In addition to the compiler options two more optiona are there with the standard compile: Generate VSM, and Incremental compilation. The first of these modes generates a 'Virtual Simulation Model', which can be used to verify the VirtualWires emulated behavior against a simulation test bench. The Incremental compile mode can be used to quickly recompile a design if, and only if, the only change to the inputs is to the list of Signals probed.

A number of the compile stages can be selected to run compilation automatically in sequence. The supported sequences are:

- Compile RTL
- Compile RTL & Generate VSM
- Compile RTL & VLE

- Compile RTL & VLE & FPGA
- Compile VLE
- Compile VLE & FPGA
- Compile FPGA
- Generate VSM
- Incremental VLE
- Incremental VLE & FPGA

Note that sequences that involve the FPGA stage are actually overlapped with the preceding VLE stage; the FPGA stage is started as soon as the VirtualWires model content for the first FPGA is known.

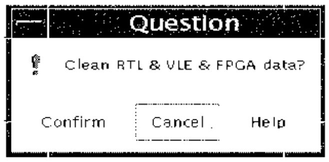
The RTL compilation options do not show up in this list if the Netlist Type is not set to 'Verilog RTL'.

These compile options also include options for data deletion such as

- Clean RTL & VLE & FPGA
- Clean VLE & FPGA
- Clean FPGA

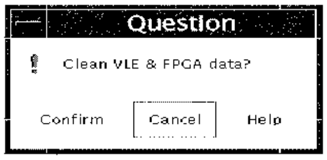
Clean

When selected, it pops up a new window asking for data to delete as shown in the *Figure 44 on page 176*. Use this when a prior run terminated abnormally and you want to start a fresh run.



Clean RTL & VLE & FPGA data?

Figure 44 RTL & VLE & FPGA data deletion



Clean VLE & FPGA data?

Figure 45 VLE & FPGA data deletion

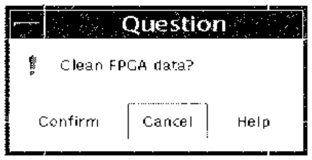


Figure 46 FPGA data deletion

Reports

This pane has useful information that includes the names of report files and log files that were creted during the compilation process. Report files such as design report file, area report file and log files such as vmw.log, rtl.log, compile.log and vtask.log are listed in this pane.

Generate VSM

The *Generate VSM* button is used to initiate model generation. It generates a Verilog netlist of the retimed (virtualized) design to verify I/O timing, memory specification, and to generate vectors for the hardware functional test.

Refer to *Generating a VSM on page 178* for additional information.

Virtualized Simulation Model

The Virtualized Simulation Model (VSM) is a Verilog model of the design that incorporates the results of the timing resynthesis process based on the netlist, memory specification, and timing specification provided as inputs.

Using this model creates an easy to debug environment for quickly resolving any emulation compilation problems so that the user can focus on the debug of real problems in-circuit.

VSM limitations

The VSM is not an exact representation of what is implemented in the emulator, but it reflects the partial timing resynthesis process that occurs with reduced *vclock*. Because a VSM is a cycle-based model, the testbench that goes with this model cannot be dependent on any clock period delays or special timing.

As with any testbench, if the testbench does not exercise a particular function, it is not verified. Use testbenches with the elements being tested. For example, use a testbench that exercises all of the internal memory modules to verify designs with lots of internal memories modeled for the emulator. If custom libraries have been created manually or developed without the use of direct mapping tools, run a more extensive verification.

VSM can also be generated during compilation with the *-vsm* compile switch option.

Preserving design hierarchy

The VSM has its own hierarchy, independent of the input design hierarchy. If the testbench exercising the design contains out of scope references for stimulus or monitoring the design, the user can choose to preserve the hierarchy of the original design. On the command line of the *Compiler* form, insert the following command:

```
-vhn
```

If this is done before generating the VSM, the resulting model retains user names and hierarchy information helpful in debug. The resulting netlist is significantly larger and will take longer to simulate, but may be easier to debug.

Generating a VSM

To generate a VSM, click *Gen Virtualized Model* on the *Verification* form. This invokes the VirtuaLogic compiler with the arguments necessary to generate the VSM. The compiler will parse the design inputs, and generate two Verilog files in the configuration directory. These are the VSMs named *<design_name>_verify.v* and the vector shell, named *vector_shell.v*.

Simulating a VSM

Figure 47 on page 179 shows the process flow for verifying a VSM. Each testbench is unique and can require different solutions, but the process flow provides a general guideline for most environments.

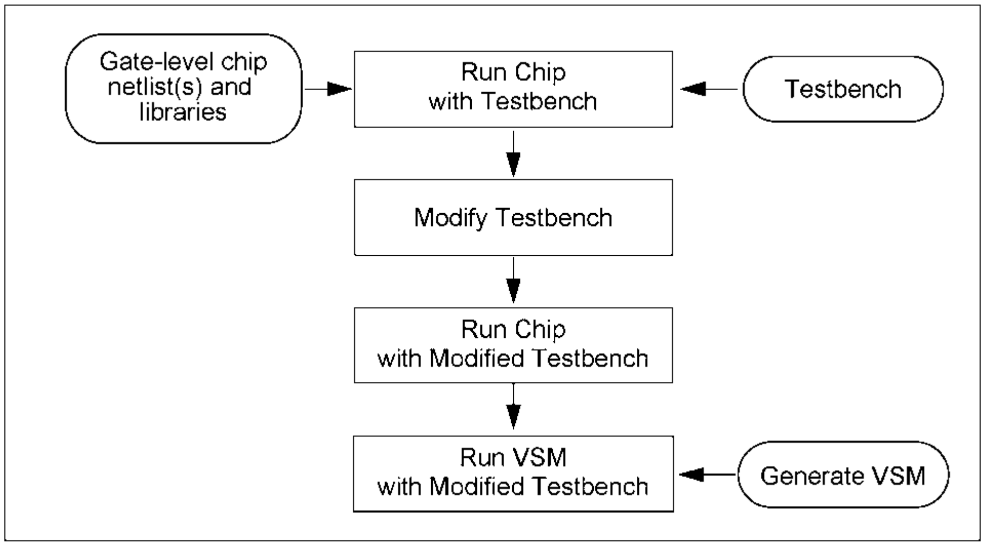

Gate-level chip netlist(s) and libraries
Run Chip with Testbench
Testbench
Modify Testbench
Run Chip with Modified Testbench
Run VSM with Modified Testbench
Generate VSM

Figure 47 VSM verification process steps

Step 1: Simulate the input netlist

Use any simulator that supports Verilog HDL to verify the VSM.

Before running the testbench on the VSM, run the testbench on the original netlist. Make sure that the design is the same version as that of the netlist synthesized for emulation. Matching the versions does the following:

- Provides the baseline for verification
- Eliminates tracing problems related to different sources
- Verifies that the simulation environment is setup properly

Step 2: Prepare the testbench

The VSM is designed to be simulated using your original testbench. There are two categories of testbench behavior which may require testbench modification or abandoning the testbench for a vector based methodology as follows:

- Timing Checks: The VSM is accurate only to user clock edges. If the testbench checks for outputs, or generates inputs at specific times, instead of at clock edges, these times will probably need to be modified.

- Cross Scope References: The VSM has a different hierarchy than the original design. While most of the hierarchy can be preserved, some cross scope references such as those to memories will still fail. These references will need to be modified or commented out.

Step 3: Resimulate the modified Testbench

If any changes are made to the testbench, rerun the original gate-level simulation to be certain that the changes do not cause unexpected behavior.

Step 4: Simulate the VSM

To run the simulation with the VSM, make the following changes to the way you invoke Verilog:

- Remove the original netlist and libraries from the list of input files
- Add the path to the VSM, *design_verify.v*, and the technology mapping library, if one is used
- Add the following IKOS primitive libraries:

  -v $VMW_HOME/lib/vmw_reference.v

  -v $VMW_HOME/lib/vmw_synthprim.v
- Add any IKOS specific definitions, if needed

Resolving simulation scenarios

Following are several common situations that might require diagnosis and resolution when a VSM is simulated:

Initializing the design

If the memory contents are defined in the *Memory Specification* form, the VSM contains *readmem* statements. The *readmem* statements load the memory with the contents of the file defined in the *Memory Specification* form. Hierarchical path names have changed and the memories have been remodeled; therefore, any *readmem* statements in the testbench that load memories in the design will fail. These should be commented out or disabled with an *ifdef* statement.

The *VMW_INIT_STATE* variable initializes the contents of the memory and flip-flops to zero at the beginning of the testbench to prevent unknown output data if the memory read occurs before the memory write. This variable can resolve initialization problems; however, do not use it unnecessarily. Using *VMW_INIT_STATE* for the VSM creates a false initialization implemented in the VSM that cannot be duplicated in the emulator. If you need *VMW_INIT_STATE* to make the testbench pass, the same testbench is not useful for hardware functional testing which applies vectors to the emulator I/O pins to verify that the actual implementation in the emulator is functional.

Timescale issues

By default, the VSM does not specify a timescale. If the simulation testbench uses timescale directives, apply the same timescale to the VSM by doing the following:

- Place the timescale directive in a file named *vmw.timescale* in the simulation directory
- Define the Verilog variable *VMW_USER_TIMESCALE* which includes this file at the start of the virtualized model definition

Input timing issues

The timing specification defines when changes on emulator inputs can occur. The VSM watches for inputs that change outside of this region and produces warning messages when this occurs. These warning messages take the following form:

```
Design i/o <ioname>: unexpected transition seen at time <time>
```

The duration of the window in which input changes can occur is dependent on your design and the timescale setting specified in the testbench.

The window in which changes can occur on a particular signal starts whenever a clock edge mentioned for the signal in the timing specification occurs and ends after the expiration of a period controlled by the Verilog variable *VMW_INPUT_SETUP* whose default setting is 100 time units. Therefore, by default, input signals can change during a window that is 100 Verilog time units in length (that is, equivalent in length to #100) and starts at each clock edge to which the input is timed.

If this default value is inconsistent with your testbench and timescale, you can override it. Provide a new value to the variable *VMW_INPUT_SETUP* using the following command:

```
+define+VMW_INPUT_SETUP=<val>
```

The <*val*> is a numeric value.

Otherwise, the edge that causes input changes from the timing specification can be omitted. To fix this, modify the timing specification.

Clock ordering and period issues

The timing specification defines an ordering for all clock edges within a clock domain. The VSM watches the order of clock edges in a domain and produces warning messages if this is inconsistent with the timing specification.

The VSM also requires some amount of elapsed Verilog simulation time to process internal changes that occur as a result of each clock edge. This time is equal to the value of *VMW_INPUT_SETUP* plus a small design-dependent component.

On the first occurrence of either condition, the VSM produces a warning as follows:

```
Warning: Unexpected clock edge <clockname1-direction1> seen at time <time1>. The
VirtuaLogic Verification simulation model will not be synchronized with your simulation
environment until the first clock edge <clockname> listed in the gvl timing specification
for this domain <domainname> has occurred. to maintain synchronization, a minimum
separation between non-coincident clock edges is required. You may need to modify your
simulation environment's behavior to avoid simulation mismatches due to delayed
synchronization or clock overrun.
```

Additional occurrences produce the following warnings:

```
Unexpected clock edge <clockname1-direction1> seen at time <time1>.

Preceding clock edge was <clockname2-direction2> at time <time2>.
```

Warnings due to clock ordering probably reflect an error in the timing specification. To fix this type of problem, adjust the timing specification to reflect the actual clock edge ordering.

If clock edges are occurring too rapidly in succession, change the testbench and/or timescale granularity to provide more simulation time between consecutive clock edges.

Vector capture

In some cases, the testbench is too complicated to run or takes excessive simulation time to use. In these cases, the user can capture the input and output vectors that the testbench applies and evaluates from the design. The user can then apply these vectors to the VSM for a simplified testbench environment.

The advantage of using vectors is that this technique eliminates the complexity of the testbench. The technique is often used where the testbench is proprietary. The disadvantage is that the testbench often provides useful debug information and automated verification of the model that is no longer available.

To implement this option, add the *<working dir>/<config name>.vmw/<vector shell>.v* file to the list of files in the testbench. The *vector shell.v* file is instantiated between the chip in the testbench and simply collects the I/O of the chip as it passes through this block.

The model in *vector shell.v* has the name *design-name -sample* so the testbench must be changed where it instantiates the design to instantiate *design-name -sample* instead.

This model, in turn, instantiates *design-name* .

The resulting vectors are in a file called *<working dir>/<config name>.vmw/ <design name>.sample.vec* which becomes part of the new testbench.

Following are the steps to run them with the VSM:

1. Run Verilog with the VSM files, *<design name> verify.v,* and the libraries mentioned in *Step 4: Simulate the VSM on page 180* **.**

2. Use *define* statements to put the testbench in vector read mode instead of vector write mode:

```
+define + VMW_READ_VECTORS
+define + VMW_VECTOR_LENGTH=<vector length>
```

This simply applies the vectors and collects the outputs.

Example

In this example, first Verilog is run with the chip (comprising *top.v*, *chip.v*, *mem rtl.v*), the testbench, and the *<vector shell>.v* file. The testbench has been modified to call the *<vector shell>.v* instance. So the testbench now calls *TOP sample* instead of *TOP*.

```
verilog -v /hq/support/release/VirtuaLogic_v2.0/lib/vmw_reference.v \
-v /hq/support/release/VirtuaLogic_v2.0/lib/vmw_synthprim.v \
testbench.v.trp ../TOP.vmw/vector_shell.v top.v chip.v mem_rtl.v
```

The result of this simulation run is a file called *TOP sample.vec* which has 141 lines. The Verilog is run again, without the chip model, the *<vector shell>.v* file, or the testbench. The *VMW READ VECTORS* mode must be enabled, and the number of vectors to run must be defined. These define statements apply the vectors to the VSM.

```
verilog -v /hq/support/release/VirtuaLogic_v2.0/lib/vmw_reference.v \
-v /hq/support/release/VirtuaLogic_v2.0/lib/vmw_synthprim.v \
+define+VMW_READ_VECTORS +define+VMW_VECTOR_LENGTH=141 \
../scc.vmw/TOP_verify.v
```

The vectors in *TOP_sample.vec* are cycle-by-cycle and contain the clocks. No timing information is imbedded in the vectors. The expected output results are captured from running the testbench on the chip model, so even if the testbench fails, the vectors reflect how the chip responded to the input stimulus. The emulation model built from the same netlist should have the same response to the input stimulus.

VSM summary

To simulate the VSM, run a simulation in which you make the following changes to your simulation arguments:

1. Remove all Verilog source files defining the root module and its contents.
2. Add in the new Verilog source file *<config_dir>/<root>_verify.v*.
3. Add in the following arguments to search the required VMW libraries:

   ```
   -v $VMW_HOME/lib/vmw_reference.v
   ```

   ```
   -v $VMW_HOME/lib/vmw_synthprim.v
   ```

4. After modifying your simulation arguments, perform a simulation.

Incremental probe compile

The *Incremental Probe Compile* button is used to run the compiler using the existing database with only the probes modified.

FPGA compile

The *FPGA Compile* allows the user to instruct the compiler to start FPGA placement and routing as soon as the data for a single FPGA is ready *(Auto-state Enabled)* or wait for the manual input from the user *(Auto-start disabled)*. As a result, the FPGA placement and routing can begin well before the compiler has completed which significantly reduces the time required to cycle a design.

Note that the FPGA placement and routing requires that a machine list has been prepared. If the compiler is run with the *Auto-start Enabled* feature, and the FPGA machine list is empty, then the FPGA scheduling program (*vtask*) will wait indefinitely until the host list is reset.

When using the *vlc* script, specify the target *compile vtask*. This will cause compilation and FPGA compilation to occur in overlapped fashion. The compile and then *vtask vlc* targets have their former behavior, resulting in nonoverlapped compilation of front-end and FPGAs, respectively.

Start compile

The *Start Compile* button is used for the design compilation process. While compiling, this button becomes the *Interrupt* button.

Interrupt

The *Interrupt* button becomes activated when the *Compile* is running. Click on the button to interrupt the running of the Compile.

LOG pane

The *Log* pane shows the log of the RTL, VLE and FPGA compiler run. It is intended for informational purposes only. If desired, the user can cut and paste from it to other X windows.

If the window is scrolling too fast to read, use the *Pause* button. This does not stop compilation; it buffers the compiler output until you are ready to read it again. Use the *Resume* button to continue. The *Pause* button is relabeled *Resume* when it is selected.

A log file is automatically updated. Selecting the *Show Log* button will invoke a text window with a log of the last completed run. The run in progress is logged to a temporary file and renamed when completed.

IKOS

6 Multi Module Compile (MMC1)

Overview

This phase of Multi-Module Compile adds an automatic multiple box compilation capability. This enables taking a single large module and allowing pre-specified sub-pieces of the module to be separately and independently compiled and recompiled, onto two to four VLE-5M or VLE-2M-IDS Emulators.

MMC1 release supports MMC across multiple boxes as follows:

- Supports multi-box
- Provides automatic compilation of designs requiring 2 to 4 emulators
- Allows you to determine the communication requirements
- Requires users to manually partition the design into the emulators
- Allocates and coordinates the dataflow between boxes such that you can use cross-wires more than once per cycle as needed
- Allocates, coordinates, and multiplexes the dataflow between boxes
- Allows compiling modules that should work but it does not debug
- Incremental compiling of individual emulator sub-modules
- Visibility is still available at the box level only

This chapter describes the high level concepts of MMC1 and its specific uses. The fundamental concept of MMC1 is to capture the conditions on the interface between sub-modules. You must compile these conditions if they change for a sub-module and that change propagates across the interface to other sub-modules. You can recompile a sub-module without impacting other sub-modules if interface conditions are not changed. The sub-module boundary conditions are saved in a sub-module without impacting other sub-modules' databases, that are read and updated every time you compile a portion of the design.

IKOS' Virtual Wires technology has been extended to work across cables between boxes to accommodate all of the I/Os in the interfaces between sub-modules. IKOS' Virtual Wires technology allows signal multiplexing onto physical wires with automatic scheduling and routing of the wires. This increases the signal connectivity capacity of a VLE box by a factor of 50 and allows for simple file-driven compilation.

MMC1 requires VirtuaLogic software version 3.3.5 or greater.

NOTE

MMC1 is not supported with TIP.

When MMC is used with the RTL compiler, an ASCII file is generated containing the list of all the RTL modules that were successfully compiled by the RTLC-driver. This file is used by the VSYN compiler for subsequent steps in the compilation. The mapping between the RTL modules and the multi-module units used by VSYN is kept and updated by VSYN.

User input

MMC requires two major additional input files, *vmw.resources* and *vmw.changes* plus one clock file modification, *vmw.clk*.

The *vmw.resources* file allows the user to specify the multi-box hardware setup including the cables that connect the boxes. It also allows the user to specify which sub-modules map to which pieces of hardware. For details of this file, refer to the *vmw.resources file on page 198*.

You must specify the targeted modules in the *vmw.changes* file to drive the incremental compile mode. Refer to the *vmw.changes file on page 200* for details of this file.

You must expand the clock file, *vmw.clk* to include any clocks that exist at the sub-module boundaries.

The clock boundaries need to be added as internal clocks. This can be done in the Timing Specification tab in *gvl*. You must distribute all top-level external clocks to all sub-modules. The root of the clock tree in each sub-module MUST be such an internal clock, except for the top level sub-module which has a primary I/O source for its clock.

Example:

top.clock- top level primary I/O clock signal

top.submod.clock- connected to top.clock

For two boxes, the clock file will look like the following:

```
Clock clock External 0

Clock top.submod.clock Internal 1
```

Refer to Design Import chapter for more information on the clock file (vmw.clk).

File structure

Since MMC involves multiple boxes, the compile process creates new configuration subdirectories underneath the original design configuration. These configuration directories are called *mod0.vmw*, *mod1.vmw*, etc. Each one corresponds to a box, where *mod0* is the first sub-module specified in the *vmw.resources* file, *mod1* is the second, etc. This is where each of the output files and output directories will be created during their respective single box compiles.

Because the configuration directory is being duplicated for each sub-module, you must use the following filenames: *vmw.clk, vmw.ftl, vmw.mem, vmw.pod, vmw.pwl, vmw.qsf, vmw.tie, *.pwb (*all probe windows must end in *.pbw), *.pbw vmw.prb.*

CAUTION

Do not rename the **.bcdb* files or the *mod*.vmw* directories.

Overview of MMC phases and flows

Manual box partitioning

Before using MMC for the first time, you must split the design up into 2 to 4 pieces that will be in the different emulators. Although, this partitioning is completely manual, it is validated by the compiler. There are some restrictions on the types of signals that can go between boxes and also on how the design can be partitioned. Please refer to the *MMC restrictions on page 201*. After the design is partitioned, the *vmw.resources* file created to match the partition, the *vmw.clk* file modified to include the sub-module interface clock signals, the automated compile process can then be initiated.

MMC phases

There are four phases in the MMC process: Top Level Analysis (**TLA**), Local Analysis (**LA**), Global Resource Allocation (**GRA**) and Local Compile (**LC**). These phases are briefly described below to aid the user in understanding any messages, errors or warnings that they might encounter.

Top level analysis

TLA has the responsibility for discovering the connectivity between all of the specified sub-modules. sub-modules are specified in an input file called the *vmw.resources* file. The top level of the design is required to be a sub-module.

NOTE

Sub-modules can be any of the modules that exist in the design hierarchy. sub-modules, in relation to each other, currently can only be in a one deep hierarchy with the top level sub-module.

TLA produces boundary condition database files (*.bcdb*).

At the end of this phase, an interbox I/O report (*vmw.log*) is produced.

Local analysis

LA analyzes an individual sub-module to determine the boundary I/O behavior. First it does the topological transformation to isolate the specified sub-module from the rest of the design. Then it does a clock domain analysis and a transition and sample analysis for the whole sub-module. Finally the boundary condition database is updated with the I/O behavior.

Global resource allocation

GRA maps the design onto the hardware. The hardware description comes from the *vmw.resources* file, which includes a description of each of the emulator boxes and what cables are attached to them. GRA allocates and schedules the design I/Os across the cables, and updates the boundary condition database.

Local compile

LC is a slightly modified version of the normal compiler. LC has been modified to support a new type of primary I/Os that are specified by the boundary condition database and also supports a new type of I/O device that can communicate across a cable to another emulator.

MMC flows

There are two MMC flows: Initial Compile and Incremental Compile. The initial compile flow is used when the design is first compiled, or when the whole design needs to be recompiled. The incremental compile flow is used when only part of the design has been changed and the design has already been compiled at least once.

Each of the flows controls the order of execution of the MMC phases. The initial compile phase execution order is as follows:

1. TLA
2. LA on all sub-modules
3. GRA
4. LC on all sub-modules
5. Run Field-Programmable Gate Array (FPGA) compiles

The incremental compile phase execution order is as follows:

1. TLA
2. LA on all sub-modules
3. GRA
4. Decision point:
   - If manual mode, report a warning and exit if any other modules besides those specified in the *vmw.changes* file need recompiling
   - If auto mode, or no changes have propagated, continue to LC
5. LC on all user-specified sub-modules and any unspecified sub-module onto which changes are propagated
6. Run FPGA compiles as necessary

If FPGAs have not been compiled, the user must type the following command from each sub-module's directory (*modx.vmw*):

```
vlc.vtask
```

This will compile all FPGAs in the sub-module configuration directories.

Refer to page 158 *for Script Driven Place and Route* and page 173 *for vtask Commands* for the complete **vtask** command description.

How to invoke the MMC compiler

The MMC compiler is currently not supported by the Graphical User Interface (GUI) and therefore needs to be invoked directly from the command line. To invoke the compiler from the configuration directory, execute the following:

```
$VMW_HOME/bin/vlc . <mmc target>
```

MMC target commands

Initial compile

The following table lists the MMC Target commands used for the first compile:

Table 6 MMC target commands (initial compile)

command	Description
mmc_clean_all	This command deletes all generated files and subdirectories.
mmc_clean	This command cleans all generated files while preserving the configurations created by the **mmc_subdirs** target. This is very useful if you want to preserve sub-module specific compiler settings or probe input files.
mmc_subdirs	This command causes the generation of the configuration subdirectories.
mmc_compile_all	This command is used when compiling the design for the first time or recompiling the full design. This does not execute any of the FPGA compiles.
mmc_cv_all	This command deletes all generated files and subdirectories. The FPGA compiles are executed after the MMC VSYN compile is done.

Incremental compile

The user can run a manual or automatic mode of incremental compiling as listed in the following table:

Table 7 MMC target commands (incremental compile)

Command	Description
mmc_compile_manual	This command constrains or restricts the recompiling to only the specified modules in the *vmw.changes* file. If a change propagates, then the compile will exit. This command does not execute any of the FPGA compiles.
mmc_compile_auto	This command just automatically recompiles everything that is necessary. This command does not execute any of the FPGA compiles.
mmc_cv_manual	This command constrains or restricts the recompiling to only the specified modules in the *vmw.changes* file. If a change propagates, then the compile will exit. The FPGA compiles are executed after the MMC VSYN compile is done.
mmc_cv_auto	This command just automatically recompiles everything that is necessary. The FPGA compiles are executed after the MMC VSYN compile is done.

NOTE

The **mmc_compile_manual** command should only be used when trying to restrict what gets recompiled (i.e. to avoid the long Place And Route (PAR) process).

Procedure for compiling mixed architecture multi-box configuration

If VLE-5M and VLE-2M-IDS boxes are used together in an MMC configuration, you must perform the following procedure:

1. Use **mmc_subdirs** mmc target command to generate subdirectories
2. In each of the module subdirectories (*modx.vmw*), modify the PLATFORM line of the *params.mak* file to reflect the mixed configuration
3. For VLE-5M, PLATFORM = PP and for VLE-2M-IDS, PLATFORM = PD
4. Use **mmc_cv_all** mmc target command to compile all the way to bits

CAUTION

Do Not Use GVL to compile VLE-2M-IDS configurations.

New *vsyn* parameters used by the mmc driver (for the advanced user)

The MMC target commands automatically generate these new parameters, therefore most users will not need to perform them directly. They are documented here to facilitate using any scripts or other advanced user applications.

The following arguments are required for all of the different MMC phases:

```
-MMCResources vmw.resources
```

specifies the MMC resource file. It is suggested to use *vmw.resources* for the default resource filename.

```
-MMCBoundary <design name>.bcdb
```

specifies the master/design level boundary condition database file. We suggest that you use <design name>.bcdb filename.

Required *vsyn* arguments (for the advanced user)

For MMC top level analysis phase, execute the following:

```
tla
```

For MMC local analysis phase, execute the following:

```
la <sub-module name>
```

<sub-module name> specifies the name of the sub-module to be processed. For specifying the module type of the sub-module being compiled, execute the following:

```
Root <sub-module type>
```

The root argument usually specifies the design module type, but in this case, it specifies the module type of the sub-module being compiled.

For MMC global allocation phase, execute the following:

```
gra
```

For MMC local compile phase, execute the following:

```
lc <sub-module name>
```

To specify an incremental compile (valid only in TLA, LA, or GRA), execute the following:

```
MMCIncrFlow
```

File formats

vmw.resources file

The *vmw.resources* file format has three parts:

1. Emulator hardware configuration
2. Cable connections
3. Map of the design sub-modules

Comments can be used in this file by typing # followed by your comment.

The syntax for specifying the emulator hardware configuration is:

```
board box[1-N].board[1-6]
```

Valid boxes are box1, box2, etc. Valid boards are board1, board2, etc. and board0 is the System Board.

Each line does the following two things:

1. It specifies that a box exists
2. It verifies that a board in that box exists

NOTE

Multiple lines are required to specify that one box has more than one board.

Example:

board box1.board1
board box2.board1
board box3.board1
board box3.board2

The above example shows that boxes 1 and 2 are one Array Board systems and box 3 is a two board system.

The syntax for the cables going in between the emulators are:

cable <cable connector> <cable connector>

cable connectors are: J<box><board><port>

There are six ports (headers) on each Array Board (1-6).

Example:

cable J111 J211 # box1.board1.port1 <-> box2.board1.port1
cable J112 J312
cable J213 J313
The syntax for mapping the sub-modules to the hardware specified is:

module <type> [full hierarchical name] on <resource>

where **<type>** specifies the type of verilog sub-module. (i.e. there exists a verilog statement: module <type name> (I/Os.....);) and **[full hierarchical name|** specifies the full hierarchical name of the sub-module. This is optional for the top level sub-module since its type is the same as the name. **<resource>** indicates one of the emulator boxes specified earlier in the file.

NOTE

Currently '\' characters are NOT allowed in the hierarchical name even though it is valid verilog syntax.

Example:

module topon box1
module child0 top.child0on box2
module child1 top.child1on box3

vmw.changes file

The *vmw.changes* file specifies the sub-modules that are to be recompiled in the incremental flow. The syntax of each line is:

<hierarchical name of the module>

Comments can be used in this file by typing # before your comment. Type the name of the module(s) as per *vmw.resources* file entry.

Example:

```
top
top.child0
```

Visibility

Normal visibility will only be available at the box level granularity. This means that probe results from different emulator boxes will not be combined. Otherwise, probe data and trigger values will be handled like they are now, both from the user point of view and the way the *vsyn* compiler deals with them.

NOTE

Internal Data Sampling (IDS) is supported. 100% visibility is NOT supported as yet.

Since centralized triggering across the multiple boxes is not supported, it is sometimes difficult to find triggers across boxes that will allow the trigger point to be synchronized near the point of interest. One workaround is to embed a resetable internal counter with its outputs going to each box that could be used in trigger expressions to help synchronization. The outputs of the counter need to be probed and the -SDPN flag needs to be used to avoid having the counter be dead logic eliminated.

MMC restrictions

MMC circuit restrictions:

- No asynchronous cross module signals
- No cross domain cross module signals
- No sub-module wire throughs
- Only one clock domain can be shared between 2 modules
- No cross box tristate bus where both boxes have tristate drivers
- No cross scope references between boxes

MMC sub-module topology restrictions:

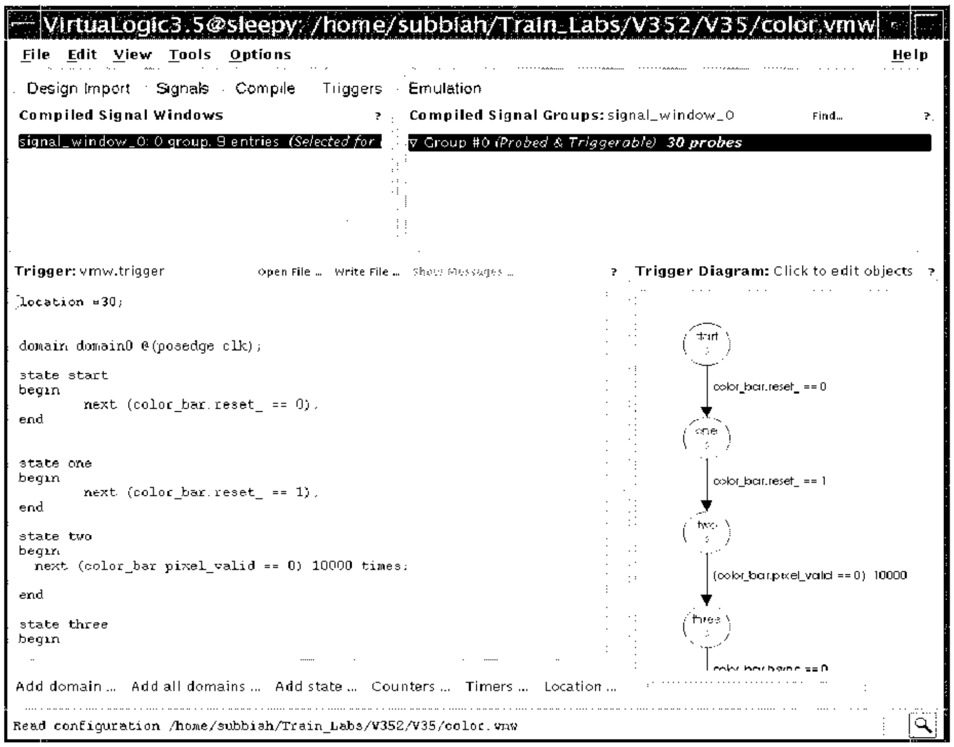
- Top level must be a sub-module (i.e. the top level must go into one of the boxes)
- Top level I/O must be in root module
- Sub-modules cannot overlap or contain each other (except top level)
- Sub-modules must be specified as only one point in the hierarchy,
  i.e. A.B.C, not A.B.C and A.B.D
- No partial box sub-modules

7 Triggers

Overview

The *Triggers* tab allows the user to determine the criteria for triggering. The *Triggers* Form is shown in *Figure 48 on page 204*, and it consists of four panes as titled below:

- *Compiled signal windows pane on page 204*
- *Compiled signal groups pane on page 205*
- *Trigger pane on page 205*
- *Trigger diagram pane on page 211*


VirtualLogic3.5@sleepy: /home/subbiah/Train_Labs/V352/V35/color.vmw
File Edit View Tools Options Help
Design Import Signals Compile Triggers Emulation
Compiled Signal Windows ?
Compiled Signal Groups: signal_window_0 Find... ?
Trigger: vmw.trigger Open File ... Write File ... Show Messages ... ?
Trigger Diagram: Click to edit objects ?
start
one
two
three
color_bar.reset_ == 0
color_bar.reset_ == 1
(color_bar.pixel_valid == 0) 10000
color_bar.hsync == 0
Add domain ... Add all domains ... Add state ... Counters ... Timers ... Location ...

signal_window_0: 0 group, 9 entries *(Selected for*

▽ Group #0 *(Probed & Triggerable)* ***30 probes***

```
location =30;


domain domain0 @(posedge clk);

state start
begin
        next (color_bar.reset_ == 0);
end


state one
begin
        next (color_bar.reset_ == 1);
end

state two
begin
  next (color_bar pixel_valid == 0) 10000 times;

end

state three
begin
```

```
Read configuration /home/subbiah/Train_Labs/V352/V35/color.vmw
```

Figure 48 Triggers form

Compiled signal windows pane

The *Compiled Signal Windows* pane lists the signal windows that were setup in the *Signals* tab. The user can choose any one of the *Compiled Signal Windows* to be downloaded into the VLE-5M along with the design.

The total set of nets for all *Compiled Signal Windows* is called the core probes. The number of core probes cannot exceed 30000 for the VLE-5M.

Compiled Signal Windows are limited to the number of signals that can be time domain multiplexed between the Array Boards and the IDS (Internal Data Sample). Each *Compiled Signal Window* typically contains multiple *Compiled Signal Groups*, but can have any combination of signals from any groups and is not required to include all signals from within one group.

All signals that will need to be analyzed simultaneously should be in the *Compiled Signal Window*. It is normally recommended to put all top-level signals in every *Compiled Signal Window* since trigger conditions frequently are built with the top-level signals. This allows the same trigger file to be used with any *Compiled Signal Window*.

Compiled signal groups pane

The *Compiled Signal Groups* pane is used to define sets of signals that are multiplexed together. *Compiled Signal Groups* define the probes that are physically routed to the FPGAs on the array boards.

The *Signal Groups* are defined in the *Signals* tab and they can be dragged and dropped into the *Compiled Signal Groups* pane.

Trigger pane

The *Add domain..*, *Add all domains...*, *Add state...*, *Add counter...* and *Add timer...* buttons can be used to help construct the FSM. Each generates a dialog box that prompts for information about a piece of the FSM and then inserts the appropriate syntax into the text editor.

Both the Trigger Diagram and the text can be edited on the GUI.

Open file...

If a **.trigger* file exists from another configuration directory, it can be read using the *Open File* button.

Write file...

To save a triggering statement into a file, click on *Write File* button.

Show errors...

The *Show Errors* button lists all the syntax errors in the **.trigger* file showing on the GUI. If there are no errors this button will be grayed out. If there are any errors in the defined triggers, this button will turn red. For warning, this button is available, and it is not red. The user should review any errors or warning messages that the software lists.

Add state...

The *Add State* button adds a new state to the trigger at the current cursor position. Clicking on the *Add State* button invokes the window shown in *Figure 49 on page 206*. The *Add State* window has three tabs namely *Attributes*, *Next Transition*, and *Jump Transition*.

Attributes:

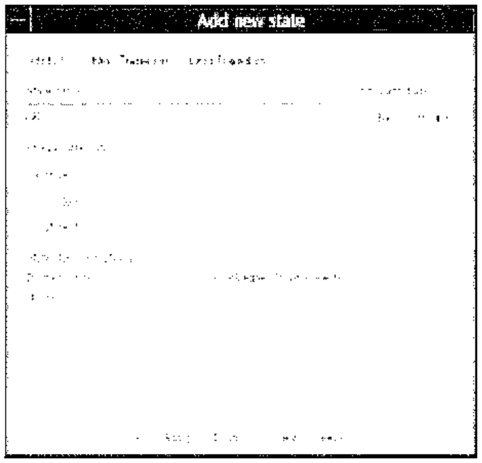


Figure 49 Add state window - attributes

State Name:	Specify a name for the *State*. It must have no spaces, only alphanumeric characters and '_'.

Control attributes:	Use the radio buttons to specify the state mentioned above as an initial state or a trigger state.
Storage attributes:	store - Sample data while the *State* is active.
	nostore - Do not sample data while *State* is active.
	store if - Sample data when the specified condition is true.
State domain (optional):	Select the domain name for the specified *State*.
Clock edge (multi-select):	Select the clock edge for the data to be sampled on.

Next and Jump Transition

Figure 50 on page 207 shows the windows for setting up the *Next Transition* statements and the *Jump Transition* statements in the trigger file.

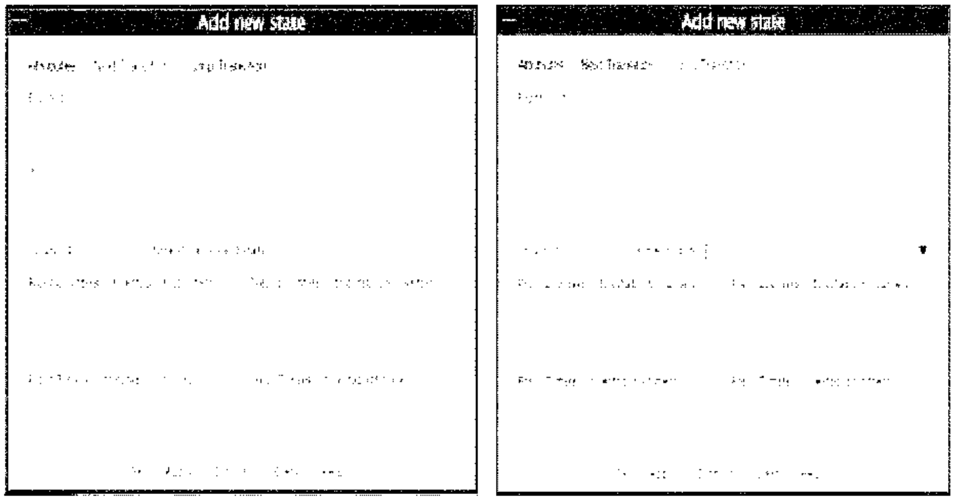


Figure 50 Next and Jump transition

For a simple state definition, the user needs to specify the state name in the *Attributes* tab and complete the expression pane in the *Next Transition* tab.

Add domain

This button is used to add a new domain to the trigger file, and when clicked, the window in *Figure 51 on page 208* pops up.

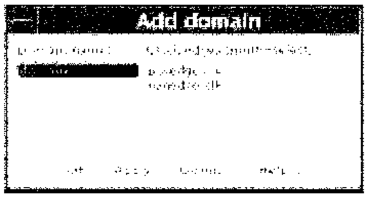
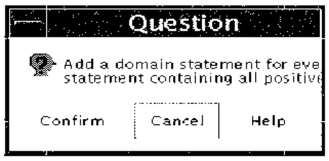


Figure 51 Add domain

In the above form the domain and the clock edge are selected and specified.

Add all domain...

The following window pops up when the *Add all domains* button is clicked.


Question

Add a domain statement for eve
statement containing all positiv

Confirm Cancel Help

Figure 52 Add all domain window

As indicated in the window, this will add all the clock domains with the clock edges to the trigger file.

Counter...

The counter dialog facilitates the creation and editing of counters. *Figure 53 on page 209* shows the dialog box.

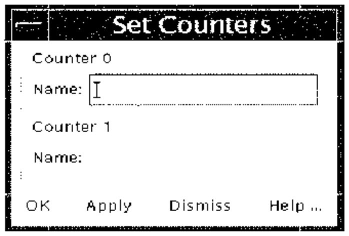


Figure 53 Add counter dialog box

There is not much to the counter dialog since only the name is required.

The name of the counter is used when manipulating the counter with reset, increment, or decrement verbs associated with transitions.

The name is also used in condition statements which test the counter value, and matches an additional condition to the transition's expression. This is specified in the state dialog's *Next* or *Jump* field named Counter condition. The emitted code will look something like

```
counter Loopy;
//...
state foo
  next (address[31:0] == 32'h0000FFFF)
    condition (Loopy == 17);
```

This example would make taking the next transition of foo conditional on the AND of the "address" and "Loopy" comparisons.

Timers

The timer dialog facilitates the creation and editing of timers. *Figure 54 on page 210* shows the pop up box.

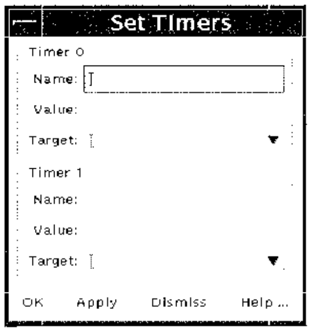


Figure 54 Add timer dialog box

There can be at most two timers in a given trigger description.

Three fields in the dialog, all required, determine the timer's identity:

Timer name -- the name of the timer, used when manipulating the timer with reset, enable, and/or disable verbs associates with transitions.

Value -- the constant value to which the timer counts before it "fires".

Target state -- the name of the state to which to jump when the timer "fires".

Location

Clicking on the Location button brings up the dialog box shown in *Figure 55 on page 210.*

Figure 55 Set trigger location dialog box

The temporal location of the trigger with respect to the probe data window is specified via a statement of the form

```
location = <value> [immediate];
```

The <value> represents a percentage and is specified as an integer between 0 and 100, in steps of 10.

The optional [immediate] keyword can be used to alter trigger system behavior for situations when a trigger occurs very early in an emulation run.

Without the immediate keyword, the triggering system treats the location specifier as an adjustment to be made after a trigger has occurred and the probe data ring buffer is full. In other words, if the ring buffer is not yet full when the trigger occurs, the triggering system will continue to emulate until the buffer is full. The immediate qualifier overrides the full buffer condition.

Trigger diagram pane

As the FSM is entered, by typing directly into the text window or into the dialog boxes, a graphical representation of the FSM is constructed in the *Trigger Diagram* pane. This pane is helpful for verifying that the state machine is what you expect. The user can edit the states, transitions, counters, timers, and location by clicking on them in this pane. when clicked, a text editor dialog pops up.

Triggering capabilities

The improved triggering capabilities described here promise to enable trouble-free, easy-to-use triggering for all common trigger scenarios. This section describes how the features of the trigger system are used.

The system described here has been designed to be easy-to-use. The system involves compiling a text description of the trigger into the firmware which evaluates the trigger at run time. There is GUI support for easily, incrementally specifying triggers, as described in the above sections; however, matters discussed in this section will be helpful even to those using the GUI exclusively.

Steps to assembling a useful trigger

The steps given here are a suggestion to get started with a workable process:

- Identify and specify possible trigger inputs
- Identify and specify desired clock edges for evaluation in each domain
- Identify and specify the patterns which describe the trigger
- Control storage as desired
- Add timers if desired
- Add counters if desired

Anatomy of the trigger system

Understanding some architectural features enables the user to more precisely and easily control trigger behavior. The trigger system consists of several major components:

- "triggerable signals" trigger inputs
- logic reduction for expression evaluation
- programmable state machine
- counters and timers

"Triggerable Signals"

A signal's being "triggerable" indicates that it might be the basis of a trigger expression, i.e., a potential input to the trigger evaluation process. At design compile time, the set of such triggerable signals will have been indicated either via GUI or directly in an ASCII file. These signals' names or aliases may then usefully appear in trigger expressions as described below, as *vsyn* will have done the necessary routing for those signals to appear as trigger inputs.

For each probe window, 1512 bits of potential trigger input signals may be specified.

Logic reduction

Logic reduction is the machinery which accomplishes the evaluation of expressions described by the user. While these could conceptually be any logic expression, there are some limits imposed by the logic reduction hardware which realizes the evaluation of expressions.

For the purposes of describing triggers, it is sufficient to understand that the logic reduction is capable of evaluating at least eight arbitrary-width comparisons of input values to constants per state. These are the first level or stage of reduction. Under certain common conditions, more than eight terms may be evaluated. Logical combinations of these first level comparisons can be three levels deep. There is no support for arithmetic operations.

Programmable state machine

States

The state machine has thirty-two possible states. Any state may be a trigger state, meaning that entering that state causes the emulation to begin the process of stopping emulation and freezing probed data. Typically only one or very few states will be trigger states.

Storage or non-storage of data is specified per-state, or may be specified conditionally based on an expression. If non-storage is indicated and the user has specified 100% visibility, a warning will be issued and any related data captures will not be suitable for 100% visibility.

Transitions

Next transitions between states form a linear sequence, while *Jump transitions* proceed from any state to any arbitrary state. It can be useful for the *Jump Transition* to proceed to *self* due to the side effects of doing this on *counters*.

The conditions for taking transitions may include expressions and counters matching given values. The conditions for taking the *next* transition are always evaluated first; only if the next condition is not taken are the conditions for taking the *jump* transition evaluated.

Evaluation of expressions

All the expressions in a given state are evaluated with respect to one clock domain and one set of clock edges. To facilitate this, the compiler determines a domain for each state and a set of clock edges with which to evaluate expressions in that state. In other words, the domain and edges control when evaluations occur. Users may explicitly control both domain and clock edges for each state if required; however, normally this is not necessary.

Counters and timers

Various counters and timers increase the flexibility of the trigger evaluation considerably with only a modest increase in complexity of hardware and software.

Expression Repeat Counts

Expressions can be given repeat counts, which amounts to the user instructing the trigger to match this expression N times before following the given transition. No repeat count is equivalent to a repeat count of one (match once).

These repeat counts are unique to each expression and are limited 32 bits in size. This is the most common sort of counter, and it is the simplest to use.

Repeat counts are always reset upon entering a state, including while taking a *jump*-to-*self* transition.

General-Purpose Counters

There are two general-purpose, 32-bit counters, independent of the expression repeat counts. These are typically used to form more complex repetitive structures. General purpose counters are explicitly manipulated, if specified, upon taking any given transition. Either counter's matching a constant can be made to be an additional condition for taking any transition.

Timers

There are also two 40 bit timers, useful for careful monitoring purposes, such as ensuring that expected progress toward triggering occurs within reasonable periods of time. Like the general purpose counters, these can be manipulated upon taking any transition.

When a timer fires, its value matches its constant; as a result, the state machine immediately and unconditionally changes to the target state associated with the given timer. Most often the target of a timer will be a trigger state. Regardless, the trigger can continue to take transitions if defined appropriately. In this context, it may be useful to note that the timer's value is reset and the timer is disabled when it fires.

Trigger description

Anatomy of a trigger description

The trigger description is a simple declarative syntax. Timers, counters, states, and domains can be declared, as required, to specify the desired trigger behavior. In addition, a trigger location assignment can be used to specify the temporal location of the trigger with respect to the data probe window.

The lexicon of the trigger descriptions is virtually identical to Verilog, including allowing comments anywhere, in the usual C or C++ form. This implies that the input is freely formatted with respect to white space, and that elements such as signals and constants appear exactly as in Verilog.

Trigger expressions are strictly Verilog expressions. Although not all forms are supported by the reduction hardware, the IKOS synthetic vector extension (angle brackets) is supported. In other words, trigger expressions are generally a subset of Verilog expressions with the synthetic vector extension for the user's convenience and consistency with other VirtuaLogic applications.

The overall form of the trigger description is a series of declarations of *timers*, *counters*, *states*, and *domains*. Detailed descriptions of the various declarations are in the following sections.

Timer declarations

A *timer* declaration consists of the keyword timer followed by the name of the timer. The declaration continues with a begin-end block which contains assignments of the value and target of the timer.

There are only two possible timers; declaring more is an error.

The general form of timer declarations is:

```
timer <timer-name>

begin

value = <timer-constant>;

target = <target-state-name>;
```

```
end
```

where items in angle brackets should be replaced as appropriate. Timer declarations generally precede state declarations.

(General-purpose) Counter declarations

A *counter* declaration consists of the keyword counter followed by the name of the counter. There are no other parameters to specify.

Since there are two general-purpose counters, declaring more than two is an error. Declaring the counter indicates that it may be used in transitions by manipulating it using counter verbs and testing against given values using condition statement. For details on counter verbs and condition statements, refer to *State declarations on page 217.*

The general form of a counter declaration is

```
counter <counter-name>;
```

Domain declarations

The simplest way to define the times of expression evaluation is to specify the usual clock edges for evaluation in a given domain only once at the top level of the trigger description. This determines the default setting of clock edges for any state associated with this domain. For more complex evaluation requirements, the domain of a state can be explicitly declared. Refer to *State Contents on page 218* for details on domain declaration within state.

The *domain* declaration specifies the details of when a trigger's inputs are evaluated in a given domain in terms of the edges of clocks. The syntax and behavior is similar to Verilog's always @ construct, except that only edges of clocks in the given domain are acceptable inputs in the edge expression.

The general form of the domain declaration is:

```
domain <domain-name> @(<edge-expression>);
```

Where < **edge-expression** > is:

```
<edge> [or <edge>]*

AND < EDGE > IS ONE OF THE FOLLOWING:

<clock-identifier>

posedge <clock-identifier>
```

```
negedge <clock-identifier>
```

The first edge form indicates both rising and falling edges of the given clock, as in Verilog.

State declarations

State declarations form the primary part of any practical trigger description. A *state* declaration consists of the keyword state followed by the name of the state, an optional state attribute list, and a begin-end block which describes the contents of the state. In particular, the contents of the *state* declaration specify the transitions out of the state, if any.

The general form of a state declaration is:

```
state <state-name>

[<state-attribute-list>]

[<state-contents>]
```

For explanations for < **state-attribute-list** > and < **state-contents** > refer to the following sections.

State Attribute List

Any of the following state attributes may be indicated by inclusion in the optional state attribute list, following the *state* name; *initial*, *trigger*; and one of *store*, *nostore*, or *store if expr*. This list is preceded by a colon; entries are separated by commas.

Only one state can be *initial*; by default, the first state in the description is the initial state. Any number of states can be trigger states, although usually there is only one trigger *state* and some error *states* which are the targets of timers or exceptional *jumps*. *store* turns storage on, and *nostore* turns it off; no storage indication retains current storage state. *store if expr* enables storage based on the value of expr. When storage is disabled and 100% visibility has been enabled, a warning will be issued, and storage and 100% visibility will be disabled on related probe windows.

The general form of a state attribute list is:

```
: <state-attribute> [, <state-attribute> ...]
```

State Contents

The state's contents consist of zero or one of domain, *next*, and *jump* declarations. It is usual but not required to follow this ordering. It is acceptable to declare only a domain; a warning is issued in this case.

Domain declaration - Domains are by default determined implicitly based on the *domain* of the first signal in that *state*, i.e., the first signal in the *next* expression, if any. Otherwise, they are determined by the first signal in the *jump* expression. Usually this default behavior will be just what is desired and offers the greatest ease of use and brevity.

For more complex evaluation requirements, the domain of a *state* can be explicitly declared using the domain declaration. Doing this is recommended whenever the domain of the *state* might not be obvious, such as when the expression involves signals from multiple clock domains. Furthermore, if there is a need to evaluate a given state differently from the default for its *domain*, the clock edges for evaluation may be explicitly declared per state.

The simple form of a domain declaration is:

```
domain <domain-name> ;
```

For syntactic details of the full form including specific clock edges, refer to < section 3.3 (outer domain declaration)>.

Next declaration - The *next* declaration consists of the keyword *next* followed by the *next expression* in parentheses, an optional *repeat count*, and *next* statements, if any.

The general form of a next declaration is:

```
next ( <next-expression> ) [ <repeat-count> ]

[<next-statements>]
```

<next-expression> - The next expression is a Verilog expression which typically involves logical combinations of the results of comparisons between various trigger input signals with constants of interest. The input form is a fully general Verilog expression but only features useful if triggers are supported in the trigger machinery.

<repeat-count> - Repeat counts immediately follow expressions, and are given by an integer followed by the keyword times. This means, match the preceding expression N times rather than repeat N times.

<next-statements> - Next statements consist of manipulations of *counters* and *timers* using verbs and indications of additional conditions to taking the transition. It is not necessary to have any *next* statements, or at the opposite extreme, all these statements may occur. Next statements may be any of the following:

- < counter-verb > counter < counter-name > ;
- < timer-verb > timer < timer-name > ;
- condition < counter-name > == < constant > ;

where < **counter-verb** > is one of:

- reset
- increment

and < **timer-verb** > is one of:

- reset
- disable
- enable

The optional condition statement allows the user to specify that the given general purpose *counter* must match the given constant in addition to the previous conditions for the transition to be taken; i.e., the expression itself is true, and the expression *repeat counter* matches or is not enabled. In other words, for the transition to be taken, the AND of the following must be true:

- the expression is true
- the expression's repeat counter matches or is not enabled
- the specified general-purpose counter matches or is not enabled

The most common cases are expected to be expression only, expression and repeat count, and expression and general purpose *counter*.

Jump Declaration - *Jump* declarations are virtually the same in form as *next* declarations, including the syntax of expression, repeat-count, and allowing all of the same statements. In addition, the *jump* transition's target state must be specified by including a target assignment of the form:

```
target = <target-state-name>;
```

Location assignment

The temporal location of the trigger with respect to the *signal window* is indicated at the top level via a *location* assignment. The value is an integer in percent; the default value is fifty, which corresponds to the trigger occurring midway in the *signal window* (middle). Zero would indicate that the trigger is to occur at the start of the *signal window* (start), and onehundred would indicate that the trigger is to occur at the end (end). At these extreme settings, the actual *signal window* will include a few samples before or after the trigger point.

The trigger position is internally rounded to multiples of ten; if rounding occurs, a warning to that effect is issued.

An immediate qualifier indicates that if the probe data buffer is not full, the trigger is to stop the run in strict adherence to the location assignment. Otherwise, the default behavior is that the run will proceed until the probe data buffer is filled.

The general form of the trigger location assignment is

```
location = <location-percentage> [immediate];
```

where < **location-percentage** > is an integer between 0 and 100, inclusive.

Trigger compilation

The following several high level features which can impact user behavior in extreme cases may be useful.

Term collapsing

Common expression forms are factored to facilitate much more effective use of the first stage of reduction hardware. These forms are:

- `&& of ==, e.g. (addr[0:15] == 16'hFFFF && reset == 1'b0)`
- `|| of !=, e.g. (data[0:7] != 8'ha0 || enable != 1'b0)`

Arbitrary expression complexities following these forms will be reduced to a single term internally. Other forms will consume multiple terms and use multiple reduction stages in a straightforward manner.

Term sharing

The *next*, *jump*, and *store if expr* within a *state* will share terms if possible. An important consequence is that the common idiom of having the *jump* expression be exactly the negation of the *next* expression, and it is supported with zero additional term consumption in the first stage. Other forms which share subexpressions will also benefit.

Trigger expressions limitations

Because the evaluation of trigger expressions occurs in a fixed size and architecture, there are inevitably some limitations. One such limitation has already been mentioned: the maximum number of distinct input terms per *state* is eight. There is also a limit on the total number of distinct terms across the entire *state* space, which is 32*4.

In a direct mapping, this would imply that each *state* could only have 4 distinct input terms, which would be unacceptable. Instead, the first stage of reductions are encoded so that only 16 sets of 8 terms are necessary, where each of these 16 combinations will be shared by two states. Only the average usage across all 32 possible states must remain below 4. A trigger would have to use almost every possible state and simultaneously use very complex expressions in almost all of these states to run into this limitation, a highly unlikely scenario.

Useful recommendations

Identify trigger inputs in the *Signals* tab of gvl.

It is wise to add as many potential trigger inputs as possible to the trigger page in order to facilitate various possible triggers without requiring incremental compiles for changed signal windows, for these compiles are comparatively lengthy compared to the nearly-instantaneous trigger compiles. Do make sure the trigger inputs are identified before design compiles. It is also often expedient to probe the trigger inputs while developing triggers whether or not these signals are interesting as probed signals.

Describe default domain clock edges.

```
domain ADomain @(posedge CLK);

domain Another @(BUSCLK);
```

While it is possible to use the trigger system without default domain statements, it is typically much more cumbersome to do so. Virtually always there are prevailing clock edges for evaluation which make sense as the default for a given domain, so this information need not be repeated and can easily be changed in a single location. Normally, the default domain statements are the only mention made of domains and clocks in trigger descriptions. The ability to describe domains and clock edges for a particular state is intended to be used infrequently.

In multiple domain designs, always consider explicitly indicating the domain for evaluation in each state.

When signals from multiple domains are used in a given state, the default domain may or may not be what is intended, or at the very least, it may not be clear what is happening. Explicitly indicating the domain using the domain statement clears up the ambiguity. It is not necessary to describe the clock edges if the default edges for the indicated domain are appropriate. Refer to *Understanding expression evaluation with respect to domains and clock edges on page 222* for better understanding.

Be very wary of cross-domain evaluations. In general, naive cross-domain expressions will result in non-deterministic results.

Summary of trigger concepts and overall syntax

Understanding expression evaluation with respect to domains and clock edges

Basics

Each state's expressions can be evaluated only with respect to one particular domain and set of clock edges. On the other hand, there may be good reason to refer to signals from various domains within a particular state, and so this is allowed.

Default domains and clocks

The trigger compiler determines the default domain for a given state by searching for the first signal among the expressions, and it uses that signal's domain as the default for the state. This may not always be what is desired, but is correct for the great majority of common cases, including all states with signals from only one domain. The default clock edges for a domain are determined using the information from the top-level domain statements, thus establishing the default edges for a given domain.

Explicit domains

Whenever the default domain behavior described above is not desired, an explicit domain statement can be used within a state.

```
state foobar

begin
```

```
domain YourDesiredDomainHere;

next (whatever == 0);

jump (andsoon == 32'hABCD0123)

target = OneMoreTimeWithFeeling;

end
```

Explicit clock edges

If this is not sufficient and explicit, unique clock edges for evaluation in a given state are required. The edges may also be specified.

```
state foobar2

begin

domain YourDesiredDomainHere @(posedge CLK or OTHERCLK);

next (whatever == 0);

jump (andsoon == 32'hABCD0123)

target = OneMoreTimeWithFeeling2;

end
```

Cross-domain evaluation

It is possible to evaluate signals from various domains in an expression; however, as indicated previously, they will be evaluated with respect to just one domain and set of clock edges.

A signal's value, as evaluated with respect to another domain's clock edges, is not always well defined. So, while defining such triggers, do make sure that you are exploiting some known relationships between the domains, or describe your trigger in such a way that such a relationship is established via some pattern.

Often cross-domain evaluation of slowly varying signals with respect to a faster domain will work properly without special attention. On the other hand, usually cross-domain evaluation where the domains in question are mutually asynchronous will not work well without explicit qualification.

Understanding overall syntax

There are relatively few special cases in the trigger description syntax, and most of the form follows from functional considerations. Here are some relationships which are useful to remember when describing triggers:

- The top level of the description consists of zero or more *state*, *counter*, *timer*, and *domain* declarations and zero or one *location* declaration
- *states* have any of *domain*, *next*, and jump
- state attributes - zero or more of *initial*, *trigger*, *store*, and *nostore* - follow the state name in a comma separated list, preceded by a colon (:)
- *next* transitions always have the condition expression in parentheses, optionally followed by N times, and can have zero or more *counter* or *timer verbs*, and possibly a condition pertaining to a general *counter*
- *jump* transitions always have the condition expression in parentheses, optionally followed by N times, can have zero or more *counter* or *timer verbs*, possibly a condition pertaining to a general counter, but also should always have a target assignment
- *counters* have nothing but a name
- *timers* should always have a value and a target

When an item contains nothing, indicate that with an empty statement (a semicolon):

```
counter foobar;
```

When an item contains just one element, it may be simply indicated:

```
state ya next (something == 0);
```

When an item contains multiple elements, a begin-end block must enclose the list of elements:

```
state ohya

begin

next (something == 0);

jump (another != 7'h3)

target = Begin;

end
```

Examples

Example 1 - Matching a given value

To match a given value, the user can use a state and a next transition, whose expressions tests the desired matters.

```
state StateName
  next (ScalarName == 1'b0) ;
```

or

```
state StateName
  next (BusName[15:0] == 16'hABCD) ;
```

Example 2 - Matching a condition N times (*Not necessarily N contiguous times)*

To match a condition N times, simply use the "N times" syntax on the transition in question.

```
state SomeState
 next (foobar == 0) 37 times ;
```

Example 3 - Matching a condition N contiguous times

To match a condition N contiguous times, user can use an N times *next* transition and a *jump-to-self* expression which inverts the *next* expression. This behaves as desired because whenever the expression is false before the count is reached, the repeat counter is reset.

```
state AState
begin
  next (one.two.address[31:0] == 32'h0A0A0A0A) 49 times ;
  jump (one.two.address[31:0] != 32'h0A0A0A0A)
    target = AState;
end
```

Example 4 - Waiting for N clock ticks

For simple domains, where user is evaluating on a single clock edge, the following will work:

```
state WaitAround
  next (1) 100 times ;
```

If the design has multiple clock domains or clock edges to be counted, the user may need to use a slightly more verbose and specific form:

```
state WaitAround
begin
  domain Simple @(posedge CLK);
  next (1) 100 times ;
end
```

Example 5 - Triggering statement occurring normally

After specifying a few states with transitions describing the pattern to be matched, specify a state with the trigger property. When the trigger machine enters that state, it will have "triggered".

```
state One
  next (foo == 1'b0) ;
state Two
  ...
state N
  next (bar == 1'b1) ;
state Done : trigger ;
```

Example 6 - Controlling storage in a trigger

To control the storage in each state, use the *store* and *nostore* attributes to unconditionally specify storage:

```
state One : store
  next (foo == 1'b0) ;
state Two : nostore
  ...
```

Example 7 - Controlling storage within a state according to the value of an expression

To conditionally control storage within a state, use the conditional storage expression:

```
state SomeState : store if (a[3:0] == 4'hA)
  next (ohmy == 1'b1) ;
```

Example 8 - Repeating a pattern N times

To repeat a pattern N times, a general purpose counter can be used to create a looping structure around the pattern to be repeated:

```
counter Loopy;
state Start
begin
  next(1)
```

```
    reset counter Loopy;
  end

  ...some pattern states...

  state StartOfLoop
    next(SomethingOrOther == 32'h1234)
      increment counter Loopy;

  ...some more pattern states...

  state EndOfLoop
  begin
    next(1)
      condition (Loopy == 17);
    jump(1)
      target = StartOfLoop;
  end
```

Example 9 - Specifying an initial state other than the first state

To specify an initial state, the *initial* property can be used. This is useful for triggers which jump to the earlier states but don't start there for some reason.

```
  state One
    next (WhatHaveYou == 0) ;
  state Two
    next (SomethingElse == 1) ;
  ...
  state ActuallyStartHere : initial
  ...
```

Example 10 - Triggering after a given period of no progress

Use a *timer* and a unique timer trigger state. While not strictly necessary, a unique timer trigger state makes it quite clear that your normal trigger did not occur.

You can specify that a given timer is to be reset, enabled, or disabled upon taking any given transition. Timers count at VCLK rate when enabled and "fire" when the timer's counter matches its specified value. When a timer "fires," the trigger machine immediately enters the timer's target state.

```
  timer TooLong
  begin
    value = 10000;
    target = TookTooLong;
  end
```

```
state Begin
  next (SomethingOrOther == 16'hABCD)
    enable timer TooLong;
...
state NormalTrigger : trigger ;
state TookTooLong : trigger ;
```

Example 11 - Matching an expression exactly N contiguous times

There are various ways to match an expression exactly N contiguous times. One way is to use a general counter, taking two states, but the recommended way which does not use a general counter takes three states.

The following is an example with three-states. Consider expression `f` to be matched exactly `N` times. Of course `f` and `N` will be replaced by an actual expression and integer respectively.

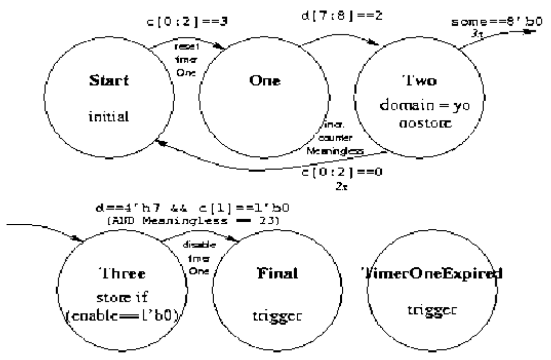
```
...
// Proceed to next state iff f N continuous times
state BeginExactly
begin
  next (f) N times;
 jump (!f)
    target = BeginExactly;
end

// Proceed to next state iff N matches again immediately;
// otherwise, "exactly N" is satisfied so we jump
//   to continue progress toward the normal trigger.
state CheckNoMore
begin
  next (f);
 jump (!f)
    target = ContinueOnward;
end

// When f is false we jump back to try again.
state KeepLooking
 jump (!f)
    target = BeginExactly;

state ContinueOnward
  ...
```

Example 12 (Advanced) - Illustrating various triggering features

```
// Annotated trigger example -- tour of features.

   location = 30;  // trigger will be at 30% point of probe (time) window

   domain single @(posedge clk);
```

```
timer One
begin
  value = 100;
  target = TimerOneExpired; // note no forward declaration of state names
end                       // are required (or allowed)

counter Meaningless;       // no properties needed, just the name

state Start : initial      // by default the first state is "initial"
  next (c[0:2] == 3)
    reset timer One;

state One                  // note that namespaces of timers, counters
  next (d[7:8] == 2)       // and states are distinct
    ;

state Two : nostore
begin
  domain = yo;
  next (some == 8'b0) 3 times;
  jump (c[0:2] == 0) 2 times
  begin
    target = One;
    increment counter Meaningless;
  end
end

state Three
: store if (enable == 1'b0)
begin
  next (   d == 4'h7      // C++ comments...
        && (c[1] == 1'b0) /* ...or C comments
                           * may occur anywhere
                           */
        )
  {
    condition Meaningless == 23;
    disable timer One;
  }
end

state Final : trigger;

state TimerOneExpired : trigger;
// end of trigger
```

Diagram of the above text:



Timers	
Name	*Constant*
One	100

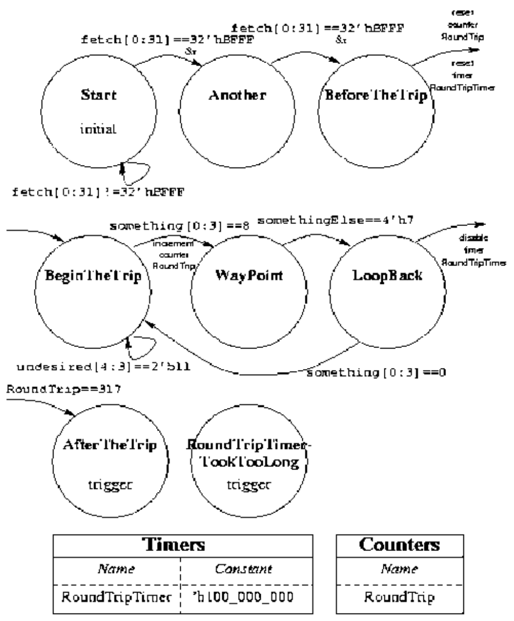
Counters
Name
Meaningless

Example 13 (Advanced) - Illustrating use of various counters

```
// Annotated trigger example -- repetitive structures.
// Trigger temporal location WRT probe window in percent.
// The default is 50.
location = 60;
domain one @(posedge clk);
domain two @(bclk or negedge cclk);
// This state illustrates how to match a condition N contiguous times,
// i.e. if the condition ever becomes false, start over.
// Note the mechanism: the jump expression is the inverse of the next
// expression, without a repeat count. This behaves as desired because
// whenever the next expression does not match, the jump transition will
// be taken back into this state, resetting the next counter.
    state Start : initial
    begin
      next (fetch[0:31] == 32'hBFFF) 8 times
        ;
      jump (fetch[0:31] != 32'hBFFF)
        target = Start;
    end

    // this state illustrates how to "match N times, not necessarily contiguous"
    state Another
    begin
```

```
  next (fetch[0:31] == 32'hBFFF) 8 times
    :
end

// Use of a general purpose counter facilitates triggering on more complex
//  repetitive behavior, e.g. a generic cycle repeating N times.
counter RoundTrip;

// Use of a timer in a repetitive structure provides a "bailout" if the
//  trigger doesn't occur within an expected period of time. This is not
//  necessary, but is a good idea in many cases.
timer RoundTripTimer
begin
  value = 'h100000000;
  target = RoundTripTimerTookTooLong;
end

// This "unconditional" state, which merely resets the counter and timer,
// could be included in the previous state if desired.
state BeforeTheTrip
begin
  next (1)
  begin
    reset counter RoundTrip;
    reset timer RoundTripTimer;
  end
end

state BeginTheTrip
begin
  next (something[0:3] == 8) 3 times
  begin
    increment counter RoundTrip;
  end
  jump (undesired[4:3] == 2'b11)
    target = BeginTheTrip;
end

state WayPoint
begin
  next (somethingElse == 4'h7)
    :
end

state LoopBack
begin
  next (1)
  begin
    condition (RoundTrip == 317);
    disable timer RoundTripTimer; // good form; required if there is more
                              // to the trigger and timer should be
                              // inactive after this piont
  end
  jump (something[0:3] == 0)
    target = BeginTheTrip;
end

state AfterTheTrip : trigger;

// If we get here the reason is clear.
state RoundTripTimerTookTooLong : trigger;

// end of trigger
```

Diagram for the above example:



Timers	
Name	*Constant*
RoundTripTimer	'h100_000_000

Counters
Name
RoundTrip

IKOS

8 Emulation

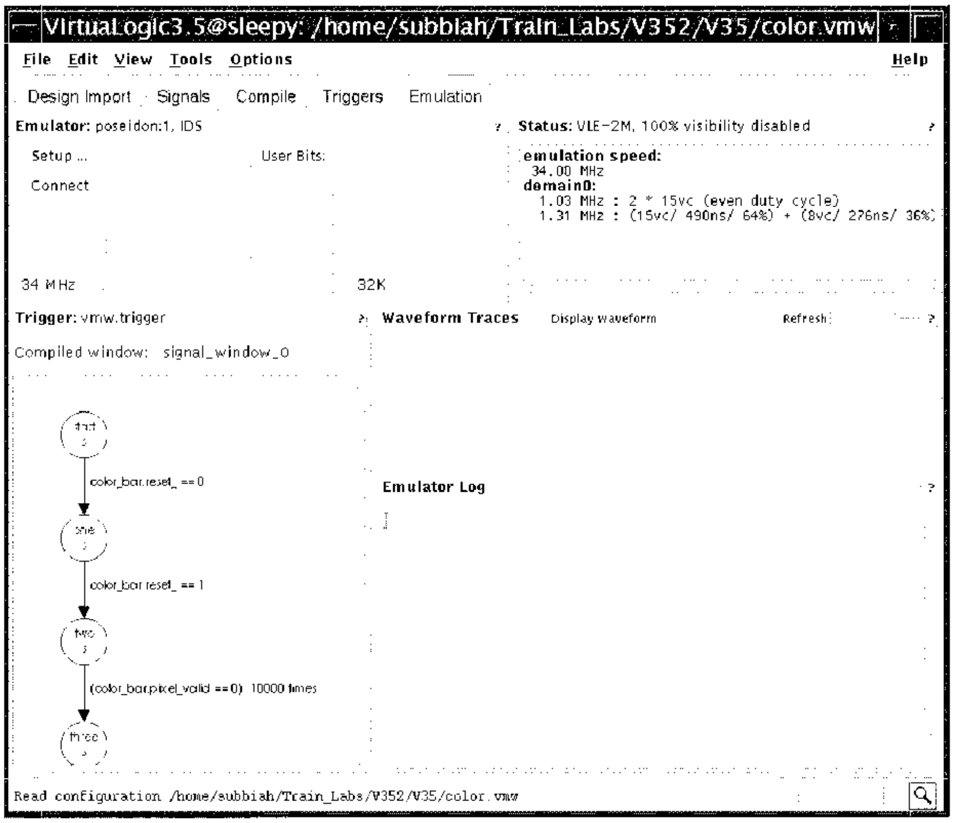
Overview

The *Emulation* form contains the controls for many of the functions that are used while debugging in-circuit.

The *Emulation* form contains 5 panes:

- *Emulator control pane on page 235*
- *Emulation status on page 244*
- *Emulator log on page 249*
- *Trigger on page 248*
- *Waveform traces on page 248*

Figure 56 on page 234 displays the *Emulation* form.


VirtuaLogic3.5@sleepy: /home/subbiah/Train_Labs/V352/V35/color.vmw
File Edit View Tools Options Help
Design Import Signals Compile Triggers Emulation
Setup ...
Connect
Waveform Traces
Display waveform
Refresh
Emulator Log
start
one
two
three
color_bar.reset_ == 0
color_bar.reset_ == 1
(color_bar.pixel_valid == 0) 10000 times

Emulator: poseidon:1, IDS

User Bits:

34 MHz

32K

Status: VLE-2M, 100% visibility disabled

```
emulation speed:
 34.00 MHz
domain0:
  1.03 MHz : 2 * 15vc (even duty cycle)
  1.31 MHz : (15vc/ 490ns/ 64%) + (8vc/ 276ns/ 36%)
```

Trigger: vmw.trigger

Compiled window: signal_window_0

```
Read configuration /home/subbiah/Train_Labs/V352/V35/color.vmw
```

Figure 56 Emulation form

Emulator control pane

Setup

The *Setup* button will pop up the *Setup Hardware* window. In this window the user specifies the information for the emulator, the Logic Analyzer, and the Virsim/Vrc Host. *Figure 57 on page 235* shows the *Setup Hardware* window.

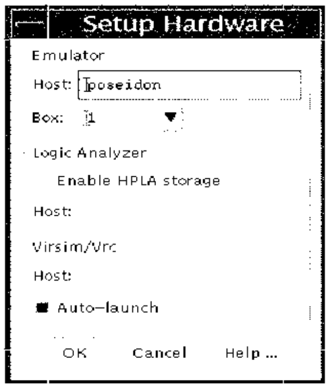

Setup Hardware
Emulator
Host: poseidon
Box: 1
Logic Analyzer
Enable HPLA storage
Host:
Virsim/Vrc
Host:
Auto–launch

Figure 57 Setup hardware dialog

Emulator

Host

The *Host* field is where to enter the name of the UNIX host that is directly connected to the VirtuaLogic emulator. The VirtuaLogic emulator is physically connected to a workstation on the network. The emulator can be controlled from any workstation on the network, but the user must specify the host to which it is attached.

Box

The *Box* button has a selection range from one to nine. This value represents the number of the emulator in which the design has to be downloaded. The typical selection, however, is one.

Logic analyzer

Host

The Host field is used to enter the hostname of the logic analyzer. Unlike the emulator, the HP logic analyzer is itself a network entity and can be controlled from any workstation.

The HP logic analyzer has a hostname on the network. VirtuaLogic uses the analyzer's network port to control it.

Virsim/Vrc

The *Host* field is an optional field that can be used to have the waveform viewer run on a different host from the user interface with the X display set to the same one that is running *gvl*.

It may be desirable to run Virsim on a host with a larger amount of memory and a faster CPU.

Host

The Host field is where to enter the name of the UNIX host that is directly connected to the Virsim.

Connect

The *Connect* button connects to the emulator host. If the host is remote, *usr ucb rsh* will be used to seamlessly operate the emulator from the local host. The user can use the *SVMW RSH* environment variable to override *usr ucb rsh* with an alternative.

This creates a remote shell on the emulator host and starts a process called *vrun*. In order to use the emulator, the user must be able to *rsh* to the emulator host without providing a password. Pressing the *Connect* button, while already connected, disconnects from the emulator.

Connecting to the emulator hardware will lock the emulator for your exclusive use. This can fail if the following conditions exist:

- Someone else is using the emulator
- The emulator is powered down

- The emulator host has not had the VirtuaLogic driver installed

Load design

The *Load Design* button finds the design name and memory files by using the design's root module name as entered on the *Netlist Import* page in the *Root Module* pane.

This will take the results of *FPGA Compile* and download them through the SCSI interface onto the emulator. It will then compile any memory contents files specified on the *Memory Specification* page for the emulator's SRAMs and download them into the emulator.

It downloads the FPGA bit streams to the devices on the Array Boards. If any compiled memories have contents files, then the contents of the memories also get downloaded. If the contents file does not specify all the addresses, the memories get filled with data that is all zero. As a result of this step, the design is implemented in the VirtuaLogic emulator, but the I/O of the chip is tri-stated so that it cannot interact with the system

At this point, the user can run a hardware functional test by pressing the *Functional Test* button. Refer to *Functional test on page 238* for additional information.

Enable I/Os

The *Enable I/Os* button is used to control the connections between the emulator and the hardware testbench. When this button is off, the connections are left in the tri-state; therefore, the emulator and testbench are isolated from each other. When this button is selected, it takes the connection out of tri-state and enables the in-circuit emulation.

If running in-circuit, before interacting with the design, the I/O Pods must be enabled. This takes the I/Os out of tri-state and allows it to connect with the target system. It may be desirable to boot the target system or at least power it up prior to enabling the I/O. The ability to tri-state all the I/O provides protection for the emulator and your system. If you suspect drive conflict between the emulator and your system, you can cause damage to the hardware by leaving the I/Os enabled.

Note if you select the *Enable I/Os* button and the TPOWER target is not asserted, it will not work. Refer to TPOWER in the VirtuaLogic Hardware Manual for information.

Emulation speed

The selection ranges from 12.0 MHz to 40.0 MHz. This controls the internal clock rate of the VirtuaLogic emulator. The range of speed depends on which emulator is connected to your system.

The emulator design speed is obtained by dividing the emulator speed by the number of virtual cycles (obtained from the compiler). This speed is displayed in the *Design Emulation Status* pane. Refer to the *Emulation status on page 244* for additional information.

Functional test

The *Functional Test* button is invoked to run a hardware functional test. Before a *Functional Test* can be run, the *Gen Virtualized Model* button on the *Compiler* page must be selected to obtain a Verilog netlist of the virtualized form of the design.

Running this netlist with your simulation infrastructure produces a file called *ROOT sample.vec*, where *ROOT* is the name you specified in the *ROOT Module* pane on the *Netlist Import* page. This vector file contains the input stimulus and expected outputs from your test infrastructure. They are saved by the behavioral Verilog included in the generated model.

The *Functional Test* applies the input vectors, samples the outputs, compares the samples to expected outputs, and generates a report based on the results. *Functional Test* uses a serial port to apply the stimulus to the design and sample the outputs. As a result, it generally runs slowly relative to in-circuit performance although it may be significantly faster than gate-level stimulation. It generally runs at approximately 500 vectors per second, where each vector is an epoch or clock transition. The speed is primarily dependent on I/O bandwidth.

As *Functional Test* runs, it reports how many vectors have been run, how many are in the test suite, and if any mismatches have occurred. The number of mismatches that are reported is the number of epochs in which one or more outputs did not match the expected output. It generates a file called *design-name sample.log* which contains the results. The file shows the results for each clock transition and determines if the expected outputs and actual outputs match.

If *Functional Test* reports that there are no mismatches, then check the *design-name sample.log* which is generated from *Functional Test* to verify that outputs were at least toggling. Remember that the test is only as good as the set of vectors that have been run.

If *Functional Test* has mismatches, then the IDS can be used to evaluate the probes that have been added to the design. This will allow further debug of the problem. It is important to create probe groups and probe windows prior to the first compile of the design, so that many internal signals are available to assist in debugging the design, or to use 100% Visibility.

Normally during in-circuit, the IDS memory always fills up because the clock is free-running. A special consideration exists when using the IDS with *Functional Test*. Since the design clock only runs while there are vectors, the *Functional Test* run may not fill up the IDS memory. It is necessary to stop the IDS from recording in order to upload the memory.

Functional test purpose

The *Functional Test* is used as a means to verify that the design, as implemented in the emulator, has correct functionality. If the design did not pass the Virtualized Simulation Model (VSM), then the *Functional Test* step cannot be expected to pass either. The *Functional Test* depends on a successful VSM verification and uses the vectors that are captured as a part of that process. Refer to page 177 for more information.

Running *Functional Test* is optional. The purpose of the VSM is to verify that all the user inputs are correct and the transformations that occur do not change the functionality of the design. However, the VSM model is only a partial model of the design, it does not have complete interconnect resynthesis. The *Functional Test* verifies that the interconnect resynthesis did not change the functionality and that the hardware is good. If the VSM has not passed, the effort required to complete *Functional Test* may be significant.

Vectors

In order to run *Functional Test*, a known good set of vectors is required. These are created by running the Virtualized Simulation Model (VSM). Refer to page 177 for detailed information. Once the VSM passes the complete vector set, then the same vector set can be used for the emulator.

For *Functional Test*, it is necessary to provide a vector file that normally gets generated when a simulation testbench is run on the verify model. The file that gets generated is called *design-name sample.vec* and it contains ascii data for vectors, one line for each epoch (clock edge). This is commonly referred to as a cycle by cycle vector; it contains no timing data, only the inputs and expected outputs for each clock transition. The clock data is also included in the vectors.

The *design-name sample.vec* file should contain a column for each I/O pin in the design. In the file called *config-name.vmw vector shell.v*, there will be a definition called VMW_WRITE_VECTORS. The *vmw sampled ios* on this module indicates the order of the signals in the *design-name sample.vec* file. In the *design-name sample.vec* file, each line of data should represent a clock transition and the inputs and outputs associated with that transition.

The *Functional Test* can only be run if the *Enable I Os* button is not selected (refer to *Enable I Os on page 237* for information). This button connects the design to the target system. In order to run *Functional Test*, the inputs to the design will be stimulated by vectors, and the outputs should not drive the target system; therefore, the I/Os must not be enabled to the target system in order to run *Functional Test*.

The *Functional Test* can be stopped with the *Interrupt* button. The speed of *Functional Test* is much slower than the emulation frequency of the design due to the I/O bandwidth of the signals that need to be sent between the emulator and the workstation. It is best to initially run a small vector set (about 1000 vectors) in *Functional Test* to verify that the design gets reset and the clock relationships look correct. If the partial test passes, then run the larger test. If it fails, it is much faster to debug while running the partial test.

During *Functional Test*, the clocks and reset to the design are provided by the vector set. As a result, the emulator has control over the clocks toggling. Due to the slow speed of the serial interface for *Functional Test*, the maximum clock speed of the design is not important.

Evaluate the *design-name sample.vec* file to verify that the clock is toggling and that reset actually occurs. This should ensure some initial functionality of the design.

Reload memory

The *Reload Memory* button recompiles the memory contents files and downloads them into the emulator.

Poke memory

The *Poke Memory* button allows setting of the value of a particular memory location within the emulator.

Upload memory

The *Upload Memory* button extracts the contents of the emulator memories' single address or whole memory into files.

Interrupt

The *Interrupt* button terminates the current operation and allows the user to keep emulating. Selecting this button will immediately terminate a functional test or memory download. It terminates these operations cleanly. For example, memory download will be interrupted between addresses.

Pressing *Interrupt* during other operations will initially be ignored. Repeatedly pressing the *Interrupt* button will terminate the emulator control program (*vrun*). The user can then reconnect to the emulator.

The user can type commands directly to *vrun* by clicking in the *Emulator Log* pane, typing the command, and pressing return.

User bits

The *User Bit n* buttons allow the user to set the value of the four software-controlled emulator outputs. When the button is depressed, the output has a value of 0. When the button is not depressed, bits 1 and 2 are left in the Z-state (*User Bit 1(1)* button and *User Bit 2 (2)* button). Moreover, when the button is not depressed, bits 3 and 4 are driven high (*User Bit 3 (3)* button and *User Bit 4(4)* button).

These four user-controlled outputs, TRST <1-4>, can be useful for resetting the target system during emulation.

TRST<1-2> are connected to pins 31 and 32 of the clock input connector. These are open-collector signals. If these signals are used, pull them up to VCC with a 470 ohm, 1/8 watt resistor on the target system.

TRST<3-4> are connected to pins 33 and 34 of the clock input connector. They are 5 V TTL totem-pole signals.

Connect analyzer

The *Connect Analyzer* button locks the HP logic analyzer for use. It can fail if the following conditions exist:

- The logic analyzer is turned off
- The logic analyzer is being used by someone else
- The logic analyzer is hung

Window

The *Window* button activates a graphical logic analyzer control window.

The user can monitor the progress of the logic analyzer by selecting the *Logic Analyzer Window* button. This shows the user, in an X window on the workstation, exactly what is displayed on the logic analyzer console. The user can manipulate this display with the mouse. To show the HP trigger display, select the following buttons:

- System
- 1 M Sample LA D
- Configuration
- Trigger 1

This will show the trigger display of the logic analyzer. Turnoff the display of the analyzer by toggling the *Window* button again.

Load trigger

The *Load Trigger* button sends a trigger to the IDS. the trigger must first be prepared on the *Triggers* form.

Once the trigger is accepted by the analyzer, it sits waiting to satisfy the trigger conditions in the *Record* mode. The trigger diagram highlights the currently active trigger state, and is updated periodically when in circuit. The diagram is not updated during functional test.

At this point, the user can stimulate the analyzer probes by running a functional test or bringing the emulator in-circuit by enabling the I/O Pods. Refer to *Functional test on page 238* and *Enable I Os on page 237* for information.

Upload waveform

The *Upload Waveform* button transfers the file (*data.raw*) from the IDS and transforms it into a Virsim waveform file. If the 100% visibility is disabled, the waveform information is written in TRACE_NAME.wave/waveform.vrc file. If 100% visibility is enabled, the waveform information is written to TRACE_NAME.wave/waveform100.vrc file. This transformation demultiplexes the Virtual Probe and associates the net name from the user's structure design with the data.

Once this is complete, select *Display Waveform* to display the results using Virsim. This causes Virsim to read in the original Verilog netlists to enable its logic browsing feature at a cost of additional start-up time.

The user can loop through downloading triggers, recording, stopping, uploading data, and running the waveform viewer in conjunction with operating the emulator through functional test and in-circuit.

100% Visibility

Once a configuration has been compiled for 100% Visibility, the usage model is then similar to the page 7-205. One difference is that when you press the Upload button, you are given the option to: (1) "complete later" (2) "complete now". This is because 100% Visibility requires use of the emulator out-of-circuit for a post-processing step.

1. By "completing later," you have the option to save the uploaded logic-analyzer data and let the in-circuit emulation proceed. You can capture and store arbitrary numbers of IDS traces while the emulator remains in-circuit, giving them each unique waveform names.

   When you want to view a particular trace, select the uploaded trace (labelled as Uploaded in the Waveform Traces panel) and press *Display Waveform*. You are then prompted to: (A) take the emulator out of circuit and extract 100% Visibility data or (B) view explicitly probed data.

   A. The emulator is taken out of circuit and used for post processing, following which all the data required for 100% Visibility of design nodes is saved in the appropriate*.wave* subdirectory. This step takes from 5 seconds to 30 minutes, depending on the depth of data stored in the IDS, and the speed of the Emulator Host workstation (the one that is SCSI-connected to the emulator). When this is complete, Virsim will be invoked (if it is not already up) and will read in the waveform file.

B. If you wish to view explicitly probed signals quickly, you do not have to take the emulator out of circuit or wait for post processing. This may be useful in some circumstances. You will have available all primary inputs to the design, plus those signals that were explicitly probed for triggering. Selecting this option invokes Virsim immediately on the explicitly probed data which is in *trace1.wave probed.vrc*.

2. If when you upload from the IDS, you elect to take your emulator out of circuit, then the post-processing will automatically occur following the IDS upload. Virsim will be invoked (if needed) and will read in the 100% Visibility waveform file, *trace1.wave waveform.vrc*, when it is complete.

Emulation status

The *Emulation Status* pane helps monitor the target system clock quality. If the target system cock is noisy or runs too fast, the title of the pane will show in red:

Target Clock Error Detected!!!!

When this message is displayed, a "Reset" button in the panel frame will become sensitive. When the target system clock has been cleaned up, the user can press the "Reset" button to clear the error display. If the clock is still too noisy or fast, the display will turn red again. When this occurs, the emulator will still attempt to operate, but some failure condition is likely.

The error display is always turned off when the I/Os are disabled. In addition, the emulator monitors the presence of target system power. If the I/Os are enabled and the emulator detects the absence of target-system power, the title of the pane will show in blue:

Target Power Missing!!!

The *Emulation Status* pane is also used to help set the clock speed for the design target system for each independent clock in each clock domain. It also indicates the status of 100% visibility functionality.

For each clock domain, the following two speeds are reported:

- A slower speed for when an even duty cycle is required. An even duty cycle in a multiclock domain is where the period between every consecutive pair of edges in the domain is equal.
- A potentially faster speed that is usable when the duty cycle can be adjusted so each edge to edge period is as long as required by the emulator.

The speed reported is that of the fastest clock in the clock domain. For example, if a domain has a 1x clock and a 2x clock, the displayed speed is that of the 2x clock. (Use of the *-clkopt* option changes this. The key concept is that the frequency reported is that of the overall repetition of all the clocks in the domain.)

In addition to the clock speed which is given in kHz or MHz, a *vcycle-count* (vc) and a minimum edge to edge period in ns is given.

The phases reported represent a breakdown of the domain's overall cycle which measures the time required between the edges of every clock.

Clock relationships

It is necessary to generate clocks for the emulator and the target system that are at the frequency required by the emulator. The simplest method is to scale all clocks by the same ratio.

If a nonsquare clock can be provided, maximum emulation frequency can be achieved. The *Emulation Status* will provide maximum frequency for a 50% duty cycle clock and another, faster frequency if a specific clock pulse can be generated. The difference between these two clocks will be greatest if the design only uses one edge of the clock. In designs that use both edges of the clock, this difference is not significant. In order to provide clocks with different duty cycles, it is recommended that a pulse generator be used for all clocks that drive the emulator.

It is not a requirement to maintain clock ratios between different domains in the system, but when first emulating, it is best to maintain relationships by slowing all clocks down by the same ratio. Once some functionality has been established, then test different clock ratios to determine the system's tolerance for different clocks. The emulator has no limits on the frequency relationships, but only establishes a pulse width for each clock epoch. However, the design might have limitations that require certain relationships, for instance, a graphics design might require that the memory clock is at least twice the frequency of the PCI clock. This might be a requirement because the synchronization circuit between the blocks expects that two memory clocks happen within each PCI clock.

Once the maximum limits for frequency and frequency ratio are determined, then run the design about 20% slower than the maximum frequency and slightly less than the frequency ratio requirement. This conservative approach will prevent loss of time debugging clock issues and simplify the setup. Always remember to check the reported frequency for each compile while it is downloading to make sure the target is running within the range of the compiled database.

Multiple domain designs as a single domain

For designs that have multiple clock domains, emulation can initially be completed by synchronizing all the clock domains to a single domain. By making a design function as a single clock domain, many setup issues such as quasi-static nets are not an issues. This will also eliminate synchronization problems. All synchronization design bugs will be masked, but it will allow initial setup and debug to be completed faster by reducing the variables. Once the synchronized environment has stable functionality, then the multiple clock domains can be implemented and debugged. This strategy allows systematic debug and elimination of factors.

To emulate a multiple clock domain design as a single domain, no netlist edits are required. In the *Timing Specification* form,

- select one clock as the master that will drive all domains
- define the top-level clock pins for all other domains as *Data Signals* and specify them as *No Connect* (with the button)
- create one clock domain and draw the waveform for the master clock
- select an internal net of each external clock in the other domains and
- draw its relative waveform to the single domain master clock

The emulator will then generate the second clock from the master clock.

Design emulation speed example

This example is a display of the design emulation speed for a four domain design with the internal speed set at speed 7, 20 MHz. The numbers are all rescaled when the user changes the internal emulator speeds.

domain0: 392kHz: 2 * 24vc
487kHz: 24vc / 1.28us + 14vc / 769ns

domain1: 444kHz: 4 * 10vc
769kHz: 10vc / 572ns + 1vc / 104ns + 3vc / 208ns + 7vc / 416ns

domain2: 215kHz: 2 * 45vc
357kHz: 8vc / 459ns + 45vc / 2.34us

domain3: 259kHz: 4 * 18vc
350kHz: 16vc / 866ns + 3vc / 204ns + 15vc / 815ns + 18vc / 967ns

In domain0, there is one clock. With an even duty cycle, it can be set at 392 kHz. However, if the duty cycle of the clock can be turned, the overall speed can usually be increased to 487 kHz. The first phase of the clock must be given 1.28 us and the second 769 ns as shown in *Figure 58 on page 247*.

Observe that if the target system can tolerate a nonuniform duty cycle, the emulation speed can usually be increased significantly.

The relationship between clock signal values and the phases reported here is controlled by the *Timing Specification* located under the *Design Import* tabcard.

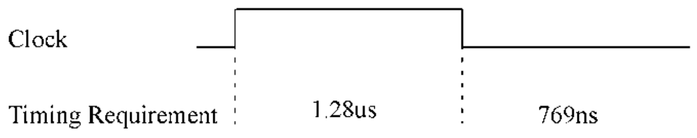


Figure 58 One Clock with Even Duty Cycle

In domain1, the maximum clock speed with an even duty cycle time is 444 kHz. A tuned duty cycle using 572 ns, 104 ns, 208 ns, and 416 ns lets it run at 769 kHz as shown in *Figure 59 on page 247*. The *Timing Specification*, located under the *Design Import* tabcard, is used to determine how long each clock should be held high or low.

Note that the minimum edge to edge period requirements are scaled when the user changes the emulator's internal clock speed, They can also change dramatically when the design is recompiled.

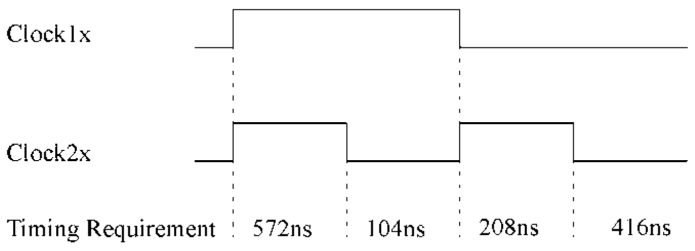


Figure 59 Two clocks with even duty cycle

Resetting the emulator target system

Reset the emulator and target system and bring the emulator out of high-impedance.

Correct in-circuit operation requires the following:

- A valid sequence of resetting the emulator and target systems
- Bringing the emulator out of its high-impedance I/O state to get the emulator and target system correctly synchronized and working together

Before resetting the in-circuit emulation, download an emulator configuration image as described previously. After configuration download, perform the following steps:

1. Apply and maintain a reset signal for the emulator and target system (emulator reset is only required when one of the design I/O terminals is a reset signal).
2. Using the VirtuaLogic interface and the *Emulation* window, click the *Enable I Os* box to bring emulator terminals out of their high-impedance state.
3. Release the reset signal(s) for the emulator and target system in the order of release in your nonemulated system; the combined system should then begin to function.

You can perform this process using a script if you connect your target system reset and/or emulator reset signals to one of the high-current emulator outputs which are under direct program control.

Trigger

Trigger pane has the trigger description given in the trigger form. It also has the signal window that was used while compilation.

Waveform traces

The *Waveform Traces* pane is used to create a new waveform trace. Click on *Display Waveform* button to see the waveforms on the GUI. For information on the waveform display tool, Virsim, refer to *Virsim control window on page 249*.

Emulator log

The *Emulation Log* window shows the log of the emulating run. It is intended for informational purposes only; however the user may be prompted from this window for information during the operation of the emulator.

Virsim control window

Virsim hierarchy

Selecting the *Hierarchy* button will invoke VirtuaLogic's *VirtualBrowser*. The user can drag and drop from the *VirtualBrowser* to the *Virsim* waveform display window.

Refer to page 49 for detailed information.

Virsim waveform

Selecting the *Waveform* button will invoke the *Waveform Window*. The *Waveform Window* graphically displays signal waveforms over simulation time. Cursors and markers are available for measuring edges and marking events that can be returned to later in the debug session. The origin of an event may be selected in the *Waveform Window* for display in the *Logic Browser* or the *Source Window*. Expression based searching can be used to locate specific events. A vector may be expanded (to show each element) or collapsed.

For help using the *Virsim Waveform Window*, select the *Help* button and then *Window*.

Virsim register

Selecting the *Register* button will invoke the *Register Window*. The *Register Window* allows the user to design views for displaying signal values and text using simple graphics tools.

Register Windows may be combined into time-synchronized link groups with other debug windows. Time-linking allows the designer to choose among a variety of change search mechanisms provided in the various debug windows; using a combination of such controls can often be the most effective method of locating and analyzing design problems, especially in mixed-level designs.

Register Windows may be used in a variety of drag and drop operations, to and from the *Waveform Window*, *Source Window*, *Logic Browser*, and other *Register Windows*.

For help using the *Virsim Register Window*, select the *Help* button and then *Window*.

Virsim source

The *Virsim Source* window is not supported in the VirtuaLogic 2.1 release.

Virsim logic

Selecting the *Logic* button will invoke the *Logic Browser*. The purpose of the *Logic Browser* is to ease design debug through improved access to, and visualization of simulation results. This is done by presenting a graphical interface specifically optimized for the tasks of design navigation, signal tracing, and value viewing. Multiple *Logic Browser* Windows may be open concurrently to view different parts of a design.

For help using the *Virsim Logic Browser*, select the *Help* button and then *Window*.

9 Compiler Options Reference Guide

Overview

This chapter covers

RTLC Additional Options on page 251

VLE Compiler Options on page 271

vlc commands on page 318 and

vtask commands on page 321.

RTLC Additional Options

RTLC compiler switches can be classified into following categories:

- Design input switches
- Language recognition switches
- Output file switches
- Messaging control switches
- Selective compilation switches
- Debug and preservation switches
- Optimization switches

Table 8 RTLC additional options

Catagories	Type	Switch	Page No
Design input switches	Verilog/ VHDL		page 253
Language recognition switches	Verilog	-synth_prefix	page 254
		-enable_case_pragmas	page 254
		-compile_celldefines	page 255
	VHDL	-max_recur_limit	page 256
		-preserve_name_case	page 256
		-compile-vhdl-inits	page 256
	-gnd_hanging_terminals		page 257
Output file switches		-out_dir	page 258
		-out_file	page 258
		-log_file	page 258
		-report_file	page 258
		-area_rep_file	page 259
Message control switches	Disable/ limit messages	-suppress	page 259
		-max_error_count	page 259
		-max_loop_cnt	page 260
		-max_mesg_count	page 260
	Allow/ Disallow SimErrors	-allow_4ST	page 260
		-allow_4ST_for_mod	page 261
		-allow_GSD	page 261
		-allow_GSD_for_mod	page 261
		-allow_ISL	page 261
		-allow_ISL_for_mod	page 262
		-allow_MDR	page 262
		-allow_MDR_for_mod	page 262
		-allow_UFO	page 262
		-allow_UFO_for_mod	page 263
		-disallow_4ST_for_mod	page 263
		-disallow_GSD_for_mod	page 263
		-disallow_ISL_for_mod	page 263
		-disallow_MDR_for_mod	page 264
		-disallow_UFO_for_mod	page 264
	-enable_BHV_messages		page 264

Table 8 RTLC additional options

Catagories	Type	Switch	Page No
Selective compilation switches		-import	page 264
		-noblack_box	page 265
		-force_module	page 265
		-force_all	page 265
		-ignore_non_rtl_gen	page 266
Debug and preservation switches		-debug	page 266
		-debug_module	page 266
		-dont_debug_module	page 267
		-preserve	page 267
		-preserve_module	page 267
		-dont_preserve_module	page 267
Optimization switches		-lut_map	page 268
		-opt_level	page 269
		-opt_timeout_limit	page 269
		-res_share	page 269

Design input switches

Verilog

- Automatic top module compilation
- -main <top_module_name> -- optional
- Standard Verilog options

Examples:

```
rtlc-vlc alu.v cntrl.v mem.v -v cells.v
rtlc-vlc -main top1 -f tops.v -y srcdir +libext+.v
```

VHDL

- -main, -ent, -arch, -conf_name <top_name>

- -w <logical-library-name>
- -analyze, -87 (default 1993 mode)

Examples

```
rtlc-vlc -analyze -w memlib mem.vhdl
rtlc-vlc -analyze -w padlib pads.vhdl
rtlc-vlc -analyze design.vhdl
rtlc-vlc -main work.top(rtl)
rtlc-vlc -ent top
```

Language recognition switches

Verilog

-synth_prefix

SYNTAX:

```
-synth_prefix <prefix>
```

USAGE NOTES:

Enable recognition of the specified string as a synthesis pragma prefix; note that "ikos" and "Synopsys" are recognized by default. Disables compilation of regions.

EXAMPLE:

```
module flop (k,d,q);
  input k,d;
  output q;
  reg q;
always @ (posedge k)
  q <=d;
  // $S translate_off
  initial  q = 0;
  // $S translate_on
endmodule
```

-enable_case_pragmas

SYNTAX:

```
-enable_case_pragmas
```

USAGE NOTES:

Enable recognition of the full_case/parallel_case pragmas in Verilog.

EXAMPLE:

```
case (sel)
2'b00 : out = in1;
2'b01 : out = ~in1;
2'b10 : out = in2;
2'b11 : out = ~in2;
endcase

case (sel)
//ikos full_case parallel_case
2'b00 : out = in1;
2'b11 : out = in2;
// no more items
endcase
```

Case Type	Line NO	Full	Parallel
CASE	12	Auto	Auto
CASE	24	USER	User

-compile_celldefines

SYNTAX:

```
-compile_celldefines
```

USAGE NOTES:

Enables compilation of modules under Verilog celldefine directives.(Default:off for library cells).

EXAMPLE:

```
`celldefine
module my_and_3(z,a,b,c);
output z; input a,b,c;
assign z = a & b & c;
endmodule;
`endcelldefine
```

VHDL

-max_recur_limit

SYNTAX:

```
-max_recur_limit <num>
```

USAGE NOTES:

Specify the maximum recursion limit for recursive functions in VHDL.

EXAMPLE:

```
function_recur_and(vector: std_logic_vector) return std_logic is
   variable result: std_logic; variable length: integer;
begin
   length = vector'length;
if (length = 1) then
   result := vector(0);
else
   result := vector(0)and recur_and(vector(length-2) downto 0);
endif;
end function;
```

Here, recursion limit = length of vector + 1, max_recur_limit 32 will fail for a 32 bit vector. Default limit is 1024.

-preserve_name_case

SYNTAX:

```
-preserve_name_case
```

USAGE NOTES:

Preserve the case of names in VHDL. The options preserves case of *nets* and *design units*.

-compile-vhdl-inits

SYNTAX:

```
-compile-vhdl-inits
```

USAGE NOTES:

Enable compilation of VHDL signal (state-point) initialization into *VMW FDPC INIT* primitives. Ignored for non-state-points.

EXAMPLE:

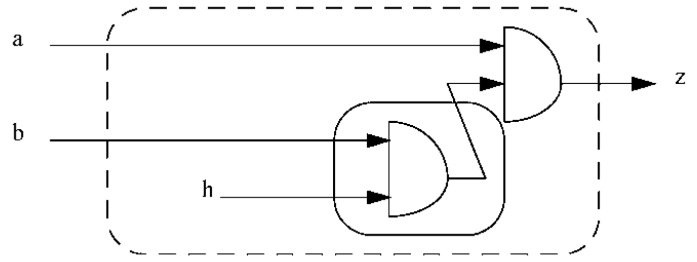
```
signal p: std_logic := 0;    // not a state-point, ignored
signal q: std_logic := '1';  // state-point
p <= a or b;
process (clk, a, b) begin
q <= a and b;               // VMW_FDPC_INIT1 inferred
end process;
z <= p xor q;
```

-gnd_hanging_terminals

SYNTAX:

-gnd_hanging_terminals (like DC)

USAGE NOTES:



- Terminal ***h*** is hanging
- With this option, it will be driven to zero

Warning 5438: Module top, Instance inst, Net b: This input has no drivers, driving zero.

Output file switches

-out_dir

SYNTAX:

-out_dir

USAGE NOTES:

Specify the output directory (default: rtlc.out). All outputs go in a single directory.

-out_file

SYNTAX:

-out_file

USAGE NOTES:

Specify the output netlist file (default: out.synth.v).

-log_file

SYNTAX:

-log_file

USAGE NOTES:

Specify the log file (default: rtlc.log)

-report_file

SYNTAX:

-report_file

USAGE NOTES:

Specify the design report file (default: rtlc.report).

-area_rep_file

SYNTAX:

```
-area_rep_file
```

USAGE NOTES:

Specify the area report file (default: area.report).

Directories NM/, NET/, INCR/

NM directory has the RTL debug database. NET directory has the gate level netlist database. INCR directory has the incremental compile database.

CAUTION: The user should not delete the files under these directories.

Message control switches

Disable/limit messages

-suppress

SYNTAX:

```
-suppress <message_number>
```

USAGE NOTES:

Suppress the specified message; note that Fatals and SimErrors cannot be suppressed.

-max_error_count

SYNTAX:

```
-max_error_count <count>
```

USAGE NOTES:

Limit instances of all error messages to the specified number (default: 256).

-max_loop_cnt

SYNTAX:

```
-max_loop_cnt <count>
```

USAGE NOTES:

Limit reporting of combinatorial loops to the specified number (default: 256).

-max_mesg_count

SYNTAX:

```
-max_mesg_count <count>
```

USAGE NOTES:

Limit instances of each message to the specified number (default: 256).

Allow/Disallow SimErrors

-allow_4ST

SYNTAX:

```
-allow_4ST
```

USAGE NOTES:

Allow 4-state reads.

-allow_4ST_for_mod

SYNTAX:

```
-allow_4ST_for_mod <module_name>
```

USAGE NOTES:

Allow 4-state reads for specified module.

-allow_GSD

SYNTAX:

```
-allow_GSD
```

USAGE NOTES:

Allow gate strengths and delays for all the modules.

-allow_GSD_for_mod

SYNTAX:

```
-allow_GSD_for_mod
```

USAGE NOTES

Allow gate strengths and delays for the specified module.

-allow_ISL

SYNTAX:

```
-allow_ISL
```

USAGE NOTES:

Allow incomplete sensitivity lists.

-allow_ISL_for_mod

SYNTAX:

-allow_ISL_for_mod <module_name>

USAGE NOTES:

Allow incomplete sensitivity lists for specified module.

-allow_MDR

SYNTAX:

-allow_MDR

USAGE NOTES:

Allow multiple drivers.

-allow_MDR_for_mod

SYNTAX:

-allow_MDR_for_mod <module_name>

USAGE NOTES:

Allow multiple drivers for specified module.

-allow_UFO

SYNTAX:

-allow_UFO

USAGE NOTES:

Allow undefined function/task outputs.

-allow_UFO_for_mod

SYNTAX:

```
-allow_UFO_for_mod <module_name>
```

USAGE NOTES:

Allow undefined function/task outputs for specified module.

-disallow_4ST_for_mod

SYNTAX:

```
-disallow_4ST_for_mod <module_name>
```

USAGE NOTES:

Disallow 4-state reads for specified module.

-disallow_GSD_for_mod

SYNTAX:

```
-disallow_GSD_for_mod <module_name>
```

USAGE NOTES:

Disallow gate strengths and delays for the specified module.

-disallow_ISL_for_mod

SYNTAX:

```
-disallow_ISL_for_mod <module_name>
```

USAGE NOTES:

Disallow incomplete sensitivity lists for specified module.

-disallow_MDR_for_mod

SYNTAX:

```
disallow_MDR_for_mod <module_name>
```

USAGE NOTES:

Disallow multiple drivers for specified module.

-disallow_UFO_for_mod

SYNTAX:

```
-disallow_UFO_for_mod <module_name>
```

USAGE NOTES:

Disallow undefined function/task outputs for specified module.

-enable_BHV_messages (RTL errors)

SYNTAX:

```
-enable_BHV_messages
```

USAGE NOTES:

This switch enables verbose reporting of RtlErrors. For example, refer to *RtlWarnings and RtlErrors on page 124.*

Selective compilation switches

-import

SYNTAX:

```
-import <module_name>
```

USAGE NOTES:

Specify the given module as imported; rtlc-vle does not synthesize this module but it writes out an empty module (prototype) for it. Typically used for memories and bonded out cores.

-noblack_box

SYNTAX:-

noblack_box

USAGE NOTES:

Disable treatment of undefined modules as black boxes.

For Verilog, *rtlc* assumes inout ports, so avoid using blackboxes for verilog. Only named port connections can be supported for Verilog. For VHDL, rtlc takes prototype from component declaration.

-force_module

SYNTAX:

-force_module <module_name>

USAGE NOTES:

Force the specified module to be compiled.

-force_all

SYNTAX:

-force_all

USAGE NOTES:

Force the entire design to be compiled; this turns off all incremental checks.

-ignore_non_rtl_gen

SYNTAX:

```
-ignore_non_rtl_gen
```

USAGE NOTES:

Non_integer generics are not supported and hence non rtl.

EXAMPLE:

```
entity mem is
  generic (data_size: integer := 32; init_file: string := "mem.dat");
  port (addr: in addr_t; data: inout data_t; wren: in bit);
end entity;
```

RtlError 3506: File nrg.vhd, Line 6: Non-integer type STRING used in declaration of generic INIT is not permitted

With this option, they are ignored.

Notice 5254: File nrg.vhd, Line 6: Non-integer type generic INIT is not supported. Ignoring the generic declaration.

Debug and preserve switches

-debug

SYNTAX:

```
-debug
```

USAGE NOTES:

Enable source-level debugging for the entire design.

-debug_module

SYNTAX:

-debug_module

USAGE NOTES:

Enable source-level debugging for the specified module.

-dont_debug_module

SYNTAX:

-dont_debug_module

USAGE NOTES:

Disable source-level debugging for the specified module.

-preserve

SYNTAX:

-preserve

USAGE NOTES:

preserve all RTL nets in the design.

-preserve_module

SYNTAX:

-preserve_module

USAGE NOTES:

Preserve all nets in the specified module

-dont_preserve_module

SYNTAX:

-dont_preserve_module

USAGE NOTES:

Does not preserve nets in the specified module; instances to zero (ground).

Optimization switches

-lut_map

SYNTAX:

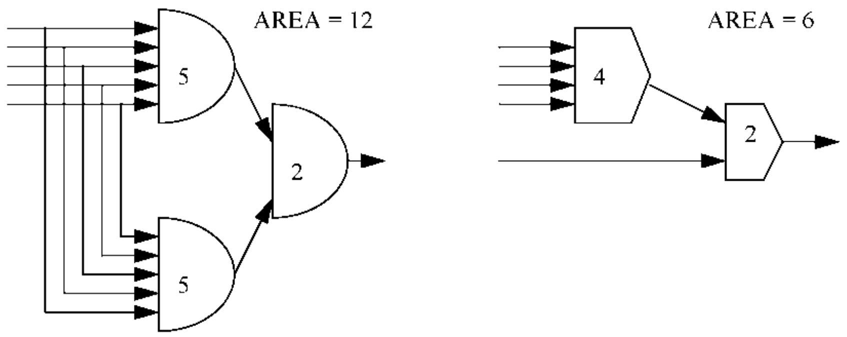
-lut_map

USAGE NOTES:

Enable LUT mapping; LUTs are written out in the file rtlc.out/out.lut.v which needs to be passed to vsyn with the -Lutlib option. (5 - 10% improvement).

EXAMPLE:

Maps combinatinal logic into FPGA look up tables. It operates on a decomposed 2-input netlist and is also more versatile than Xilinx mapper. Preserved nets are decompiled as hierarchical refernces into LUTs.



-opt_level

SYNTAX:

```
-opt_level <0/1/2/3/4>
```

USAGE NOTES:

Specify the optimization level (default: 3); 0 is the lowest and 4 the highest. Level 3 is good trade-off between compile time and gate count.

-opt_timeout_limit

SYNTAX:

```
-opt_timeout_limit
```

USAGE NOTES:

Specify the timeout limit for optimization in seconds; use this if a single module is taking an abnormally long time to compile

-res_share

SYNTAX:

```
-res_share
```

USAGE NOTES:

Shares resources in mutually exclusive paths(0-25% improvement).

EXAMPLE:

```
always @ (sel or in1 or in2 or in3)
casex (sel)
2'b0x: out = in1 + in2;
2'b10: out = in2 + on3;
2'b11: out = in1 << in2;
endcase
```

This is converted to : out = in2 + (cnd) ? in1: in3; // cummulative

where cnd = func(sel)

Compiler Directives

Disable compilation of regions

synthesis_off, synthesis_on

translate_off, translate_on

-synth_prefix synopsys, ikos

Example:

```
module alu(...);
  // ikos translate_off
initial accumulator = 0;
// ikos translate_on
  always @ (posedge clk)
     accumulator = calculate(opcode, operands);
```

VHDL Built-in Pragmas

syn_feed_through
syn_sign_extend
syn_zero_extend
syn_plus
syn_minus
syn_signed_mult
syn_unsigned_mult
syn_abs

EXAMPLE:

If there is a ripple carry adder in the design, the rtlc does not know if the code is for an adder circuit, but syn_plus, a built-in pragma, tells the rtlc to substitute that piece of code with an available adder circuit.

VLE Compiler Options

Table 9 VLE compiler options

Option	Capacity Control Arguments	Partition Control Parameter	Database File Suppression	Analysis/ Transformation Control	Output Simulation Models	Controlled by SW	Page Number
"-Ao" <string>						X	page 314
"-arrpart" <fpga_name>						X	page 312
"-bond"						X	page 313
"-Clk" <filename>						X	page 309
"-clkopt" <factor><domain_name>				X			page 298
"-CUc" <number>		X					page 277
"-CUi" <filename>		X					page 278
"-DB" <filename>						X	page 308
"-define-"						X	page 313
"-defines_file" <filename>						X	page 313
"-Dump" <options> <filename>							page 305
"-fifo_refold_port_limit" <integer>				X			page 297
"-FPi" <filename>		X					page 279
"-hvpd" <filename>						X	page 314
"-IncProbe"						X	page 311
"-LBfi"						X	page 282

Table 9 VLE compiler options

Option	Capacity Control Arguments	Partition Control Parameter	Database File Suppression	Analysis/ Transformation Control	Output Simulation Models	Controlled by SW	Page Number
"-Lib" <filename>						X	page 308
"-Mc"						X	page 312
"-Mem" <filename>						X	page 309
"-memmap"						X	page 313
"-Mm" <#>	X						page 276
"-Mmfanout"<#>	X						page 276
"-Mo" <filename>						X	page 314
"-MultiAsic" <filename>						X	page 311
"-NAfi"<filename>				X			page 287
"-Nb" <#>						X	page 308
"Net"							page 287
"-NCfi" <filename>				X			page 284
"-NFfi" <filename>				X			page 286
"-noclkopt" <filename>				X			page 298
"-noclockblocks"							page 300
"-nodb"			X				page 281
"-NoSrfi"<filename>				X			page 293
"-NoSyncQS"							page 284

Table 9 VLE compiler options

Option	Capacity Control Arguments	Partition Control Parameter	Database File Suppression	Analysis/ Transformation Control	Output Simulation Models	Controlled by SW	Page Number
"-novrc"			X				page 281
"-noXCT"				X			page 295
"-noXFT"				X			page 295
"-noXIAT"				X			page 296
"-noxnf"			X				page 281
"-noXOT"				X			page 295
"-noXSAT"				X			page 296
"-noXTAT"				X			page 296
"-NPb" <#>						X	page 309
"-Pfi" <root_module>.place						X	page 316
"-Pfo"<root_module>.place						X	page 316
"-Pi" <root_module>.part						X	page 315
"-Po" <root_module>.part						X	page 315
"-Pod" <root_module>.pod						X	page 311
"-ProbeCard" <#>						X	page 310
"-ProbeCore" <filename>						X	page 310
"-ProbeDB" <filename>						X	page 311

Table 9 VLE compiler options

Option	Capacity Control Arguments	Partition Control Parameter	Database File Suppression	Analysis/ Transformation Control	Output Simulation Models	Controlled by SW	Page Number
"-ProbeIn" <filename>						X	page 309
"-ProbeMap" <filename>						X	page 310
"-ProbeWindows" <filename>						X	page 310
"-PUi" <filename>		X					page 280
"-q"							page 303
"-QSfi"<root_module>.qsf				X			page 283
"-Root"						X	page 309
"-SDPN"				X			page 297
"-Se"				X			page 292
"-Sr"				X			page 292
"-syspart" <fpga_name>						X	page 312
"-target" <filename>						X	page 312
"-targetfile" <filename>						X	page 312
"-Terse100"							page 304
"-TerseProbe"							page 305
"-Ti" <filename>						X	page 314
"-TNfi" <filename>				X			page 290
"-v"							page 303

Table 9 VLE compiler options

Option	Capacity Control Arguments	Partition Control Parameter	Database File Suppression	Analysis/ Transformation Control	Output Simulation Models	Controlled by SW	Page Number
"-Vbc" <#>					X		page 301
"-ve"							page 304
"-vhdlout"					X		page 303
"-vhn"					X		page 301
"-Vo" <filename>					X		page 302
"-vs"							page 304
"-vsn"					X		page 301
"-vw"							page 304
"-writevpd"			X				page 281
"XCrossDomainIO"							page 299
"-XFTL"							page 299
"-xln"					X		page 302
"-Xo" <filename>						X	page 314
"-XTAT"				X			page 296

Capacity control arguments

-Mm

SYNTAX:
-Mm <#>

ARGUMENT:

<#>	This switch controls the amount of logic in each chip.

DEFAULT:

Default: 10
Range: 8 - 12
Scale: 10 = 100%; 10.5 = 105%

USAGE NOTES:

FPGA size-modeling control for partitioning.

The default for this switch is 10 which targets 100% logic allowed on each chip (cost of 5000 / 10,000). Changing [illegible] to 9 reduces the amount of logic allowed to 90%. This improves the FPGA Place and Route time by making each chip easier to place and route. However, the design might run slightly slower because of fewer gates in each chip and more chip hop required in a path. It also uses the hardware capacity less efficiently and is not recommended if a design is a tight fit in the emulator.

Increasing this number provides more utilization but at the same time increases the FPGA Place and Route time. For example,

-Mm 10.5 // expands modeled capacity by 5%

Refer to *-CUi on page 278* for information on a design that exhibits a structure leading to systematic place and route failures that are insensitive to the *-Mm* cost adjustment parameter.

-Mmfanout

SYNTAX:
-Mmfanout <#>

ARGUMENT:

<#>	Post-partition size modeling control

DEFAULT:

Default: Current value of *-Mm*
Range: Same range as *-Mn*
Scale: Choose a value greater than *-Mn*

USAGE NOTES:

FPGA size modeling control: Setting this larger than *-Mm* will control congestion. This is most appropriate when *-Dump t* indicates average transition count well in excess of one.

Refer to *-Dump on page 305* for information.

Partition control parameters

-CUc

SYNTAX:
-CUc <number>

ARGUMENT:

<number>	The *<number>* is an integer.

DEFAULT:

The default value of the parameter controlled is one.

USAGE NOTES:

The baseline adjustment value can be set to something other than one by using this switch.

Refer to *-CUi on page 278* for additional information about a variable cost adjustment factor.

-CUi

SYNTAX:

```
-CUi <filename>
```

ARGUMENT:

<filename>	User hierarchical cost control file

USAGE NOTES:

Solves the fitting problem in FPGA compilation.

On rare occasions, a design exhibits a structure leading to systematic place and route failures that are insensitive to the *-Mm* cost adjustment parameter (refer to *-Mm on page 276* for details). If an FPGA fails to compile, and the components on the FPGA are mostly from the same part of the design, the user can raise the weight of that part using the compiler option *-CUi <filename>* as follows:

- At the command line, create a new file. The file contains one or more lines of the following form:

  ```
  Module <module_expression>
  ```

  where *<module_expression>* is a regular expression matching the hierarchical name of one or more module instantiations within the design. Any module listed in the file has its modeled cost increased.

  The typical use of the file has the following form:

  ```
  Module a.b.c.********
  ```

  which matches all instantiations within the scope *a.b.c.* (The number of *s should exceed the internal hierarchical depth of the scope *a.b.c.*)

  All the modules under the hierarchy *a.b.c.* get a higher cost than they normally would during design compilation.

- Return to the *Compile* form and in the *Compiler Options* window enter the following:

  ```
  -CUi <filename>
  ```

- Recompile the design.

The *-CUi* switch allows a variable cost adjustment factor. The syntax is as follows:

```
Module <module_expression> <number>
```

For this syntax, the cost adjustment value for all modules matched by the regular expression *module expression* is *number* independent of the *-CUi* baseline established by *-CUc* or its default.

Refer to *-CUc on page 277* for additional information.

Use of the *-CUi* switch with a nondefault value is effective in situations where some FPGAs compile excessively slow due to very slow Xilinx timing analysis in the presence of highly divergent and reconvergent combinational structures.

-FPi

SYNTAX:

-FPi <filename> -Pi <partition_input_filename> -Pfi <place_input_filename> [-Po <partition_output_filename>]
[-Pfo <place_output_filename>]

ARGUMENT:

<filename>	Further partitions the contents of the designated file. Lists on separate lines the names of all noncompiling FPGAs.
-Pi <partition_input_filename>	Reads what is in the file as the partition result of the previous run. Contains the partition result of the previous run. This is required.
-Pfi <place_input_filename>	Reads in the result of placement from the previous run as output of the placer. Contains the placement from the previous run. This is required.
-Po <partition_output_filename>	Outputs the result of the partition into the specified file. Contains the result of the partition. This is optional.
-Pfo <place_output_filename>	Outputs the result of placement to the designated file. Contains the result of place. This is optional.

USAGE NOTES:

Enables further partitioning.

If one or more of the FPGAs does not compile, a message appears in the *Place and Route Log* window listing the failures. To correct the situation, take the following steps:

- At the command line, create a new file comprising the name of each noncompiled FPGA on a separate line
- Return to the *Compiler* form and in the *Compiler Options* window enter the following:

  -FPi <filename> -Pi <partition_input_filename> -Pfi <place_input_filename> [-Po <partition_output_filename>] [-Pfo <place_output_filename>]

- Recompile

-PUi

SYNTAX:

-PUi <filename>

ARGUMENT:

<filename>	Allows the user to group logic to be on the same FPGA

USEAGE NOTES:

To use that option, in the *Compiler Option* pane, enter *-PUi <filename>*.

The file is in the following form:

```
Group <group_number>
M <modulename0>
M <modulename1>
.
.
```

<modulenamen> is a hierarchical module reference or regular expression matching one or more modules. There can be several groups in the file and each group can have any number of primitive logic names. The compiler clusters all the modules in each group together so they end up in the same FPGA.

Refer to *No-Flows in Combinational Loops on page 156* for additional information on using *-PUi*.

Database file suppression switches

-writevpd

SYNTAX:

```
-writevpd
```

USAGE NOTES:

Enables *vpd* file generation. It allows compatibility with *VirSim*.

-novrc

SYNTAX:

```
-novrc
```

USAGE NOTES:

Disables *vrc* file generation.

-noxnf

SYNTAX:

```
-noxnf
```

USAGE NOTES:

Disables *xnf* file generation.

-nodb

SYNTAX:

```
-nodb
```

USAGE NOTES:

Prevents writing the database on the disk. It does not generate the *db* file and it disables the *vdb* file generation.

This file is used when doing an incremental compile to add or delete probes. This switch saves some disk space and also speeds up the compile process.

Analysis/Transformation control

-LBfi

SYNTAX:

```
-LBfi <filename>
```

ARGUMENT:

<filename>	Reads in the file with the latch bias net(s)

USAGE NOTES:

This switch improves the model execution speed for some latch-based datapaths. It is a means to control latch processing in order to insure that a particular stage of logic in a latch-based datapath is allowed the maximum time for propagation, generally a full clock cycle.

If latches clocked by opposite clock phases are used in a datapath, under some circumstances the VirtuaLogic compiler will unnecessarily constrain the flow between adjacent stages of latches to within a half clock cycle. A deep datapath example might be a 64-bit full data adder in an ALU or multiplier/divider which will be performance limiting in the emulation model.

The <filename> format is:

Net <net-expression>

where <net-expression> is a hierarchical net name or wildcard expression matching a collection of nets. Processing of any latch which either combinatorially reaches or is combinatorially reached from a net identified in the -LBfi file will be biased to allow maximum time in the direction of the latch bias; thus, latches whose outputs reach a latch bias net will give more time to the output at the expense of the input, while inputs will give more time to the input at the expense of the output.

It is not necessary to specify directly the latch input or output nets in a -LBfi file. Specifying a net for latch bias will affect all latches whose outputs combinatorially reach the specified net or whose inputs are combinatorially reached from the specified net. For example, in the 64-bit adder example cited above, using a wildcard expression to identify all nets in the adder carry chain is sufficient to impact all latches fanning into or receiving any output from the adder datapath.

If both the input and output of the same latch have a bias, controlled biasing of the latch will not occur and the task objective may not be achieved.

Also, use of -LBfi suppresses some forms of automatic dataflow path partitioning and can result in identification of combinatorial cycles in a design which does not exhibit them in the absence of -LBfi. If this occurs, and the cycles are too large for the VLE system to model, restrict -LBfi to areas of the design not exhibiting cycles.

-QSfi

SYNTAX:

```
-QSfi <root_module>.qsf
```

ARGUMENT:

<root_modules>.qsf	Quasi-static net file

USAGE NOTES:

This switch works in conjunction with the *-Dump q* option. It reads in a user-selected set of quasi-static nets for resolving clock domain merging ambiguities.

Refer to *-Dump on page 305* for additional information.

Refer to *-TNfi on page 290* for information on hierarchical net references in annotation files.

Refer to *-NoSyncQS on page 284* for information on quasi-static net processing.

-NoSyncQS

SYNTAX:

-NoSyncQS

USAGE NOTES:

Quasi-static net processing has changed such that all cross-domain fanouts of a given net which has been declared quasi-static will always see the same value of the net. This behavior only applies to quasi-static cross-domain nets.

Note that this behavior results in the potential masking of race conditions specifically associated with multiply sampling any quasi-static marked net. It does not impact race condition modeling between pairs of quasi-static nets, between quasi-static and nonquasi-static nets, nor multisampling of nonquasi-static nets.

The change which leads to this behavior can be suppressed by adding the compile switch *-NoSyncQS*. However, this may result in excessively large models when 100% visibility is enabled.

-NCfi

SYNTAX

-NCfi <Filename>

The *-NCfi* switch enables a feature which will be referred to as a NoClock annotation. *-NCfi* takes a <filename> argument which identifies a file containing net or terminal NoClock specifications.

ARGUMENT:

A Net NoClock specification has the following syntax :

Net <netname>

where <netname> is a hierarchical net name or regular expression and indicates that all nets matching netname have a NoClock attribute.

A Terminal NoClock specification has the following syntax:

Terminal <terminal-name>

where <terminal-name> is a hierarchical terminal name or regular expression and indicates that all nets connected to matching terminals have a NoClock attribute. If the terminal is an input terminal, the attribute only applies to the specific net fanout connected to the terminal. If the terminal is an output terminal, then the attribute applies to all net fanouts, as would be the case if a Net NoClock specification were to be used.

Example:

-NCfi vmw.nclk, where the content of vmw.nclk is:

Net top.inst_foe_core_nobuf_top.corecomp.inst_foe_core.inst_cu_shell.reset

Terminal top.hv.\h__int/grif/U157 .A

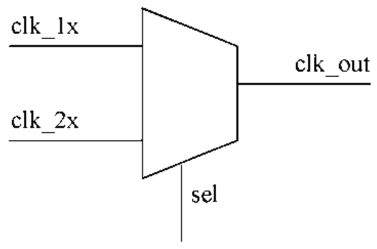
USAGE NOTES:

Designs with complex clock trees intend to consume an abnormal high number of gates and result in long PAR compile times. This is due to the pessimistic nature of the compiler and the fact that clock trees need to duplicate into each FPGA that contains a sequential elements clocked by this particular clock. The NoClock annotation is a feature that reduces clock tree complexity, then facilitate clock generation.

Usage of -Dump m0 dump option will report the clock source cost of all clocks and gated clocks. *-NCfi* is to be used to annotate part of a clock gating circuitry, net or terminal, that contributes to clock generation, whenever the compiler reports high clock source costs above 1000.

EXAMPLE:

One common example of a circuit with nets which cannot be automatically determined to be NoClock candidates are clock multiplexing circuits which select from one of several clocks (between clocks of the same clock domain). The control signals which choose between the clocks are often NoClock candidates and also often are produced by substantial logic cones which one might wish to remove from the logic tree. Multiplexor select signals are legitimate NoClock signals under a variety of circumstances. If the select value changes occur when all clocks have the same value then a delay on select doesn't impact active clock edges.



This would be the case when switching from a 1X clock to a 2X clock if the switch occurs at a point where both clocks are producing a rising edge for an instance. Similarly, this would be the case if all clocks are gated to some constant value during the select change.

Alternatively, clock switches may be very rare and glitches during the switching point may be expected and tolerated by the design, by following the clock switch by some form of reset activity. Here a NoClock may have a circuit semantic impact but no impact on the higher level operation of the device.

In addition to selects on clock switching circuits, other forms of clock gating may cause the VStation compiler to incorrectly decide that some net and logic tree are delivering active clock edges. If this is determined to be causing either excessive logic resource use or contributing to FPGA timing problems, then NoClock can be applied as a way to eliminate the net and logic cone from the clock tree.

-NFfi

SYNTAX:

```
-NFfi <filename>
```

ARGUMENT:

<filename>	Reads in the no-flow nets from the no-flow file

USAGE NOTES:

This switch allows the information for the no-flow to be read in with each compile.

A user-inserted no-flow is precisely equivalent to the injection of a pipelining flip-flop on the specified circuit net or terminal. The pipeline flip-flop is clocked by the first edge in the clock pattern provided for the clock domain associated with the no-flow. (For clock domains with higher frequency clocks, i.e., 2X clocks, the first and subsequent symmetrically placed edges are used.) For NET nodepends or TERMINAL nodepends on primitive output terminal, the clock domain used will be the domain in which the net's source driver is located. For TERMINAL nodepends on primitive input terminals, the relevant clock domain will be the clock domain of the associated receiving primitive.

This ensures that no-flows have the equivalent behavior in Virtualized Models as they do in-circuit under all conditions.

-NAfi

SYNTAX:

```
-NAfi <filename>
```

ARGUMENT:

<filename>	Reads in the net timing information from the net annotation file

USAGE NOTES:

Net timing annotation allows the user describe timing characteristics of internal nets in a design.

In some situations, the emulation compiler adopts an overly conservative model of the behavior of a design net. If the net is a clock, particularly a clock with high fanout, this modeling pessimism can lead to added capacity requirements and/or diminished performance. Net timing annotation allows the user to specify a more accurate timing behavior to improve capacity or performance issues.

Net timing annotations relate edge and/or value behavior of a net to some primary clock signal. A net timing annotation takes the *Net* form described below:

Net

SYNTAX:

```
Net <netname> <clockname>[invert] <keyword>
```

ARGUMENTS:

<netname>	This is a name or regular expression matching the hierarchical name of one or more design nets.
<clockname> [invert]	This indicates that the timing behavior specified for the signal *<clockname>* is the basis for identifying behavior of *<netname>*. The optional keyword *invert* indicates that the behavioral basis for *<netname>* is the inversion of the behavior of *<clockname>*.
<keyword>	Takes the value zeros, ones, edges, rise, or fall.
Zeros/Ones	This indicates that the signal *<netname>* is zero or one, respectively, whenever the clock prototype behavior exhibits this behavior.
Edges	This indicates that when *<clockname>* has a rising edge, *<netname>* has a rising edge or is stable, that is, it does not have a falling edge. Similarly, when *<clockname>* falls, *<netname>* falls or remains stable.
Rise/Fall	This indicates that *<netname>* only has value changes when rising/falling edges occur on *<clockname>*.

The user can apply multiple net timing annotations to a net. Behavioral characteristics associated with known values are the intersection of the behaviors of the annotations. Behavioral characteristics associated with net changes are the union of the behaviors.

EXAMPLE:

A data-dependent edge detector circuits example follows.

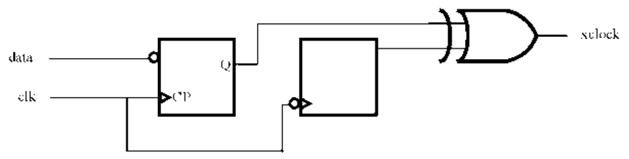
Various forms of data-dependent edge detector circuits produce conditional pulses that the user can describe using net timing annotations.

Assume in all cases that clk is a simple periodic clock.

1. You can characterize this output xclock as

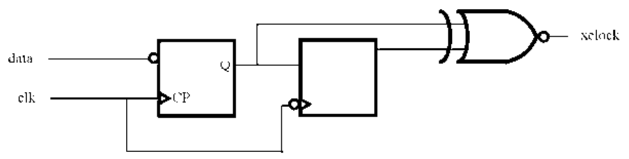
   ```
   Net xclock clk zeros
   ```

   since the signal xclock can be 0 or 1 when clk is 1 but is always 0 when clk is 0.



Without this characterization, the emulation compiler models the signal as being entirely unknown and potentially making either rising or falling transitions on both edges of clk. This action causes the state elements that are clocked by this signal to be viewed as potentially changing on either edge of clk. With the timing annotation, rising-edge state elements can be known to be active only on some rising edges of clk and similarly for falling-edge state elements.

2. In this circuit,



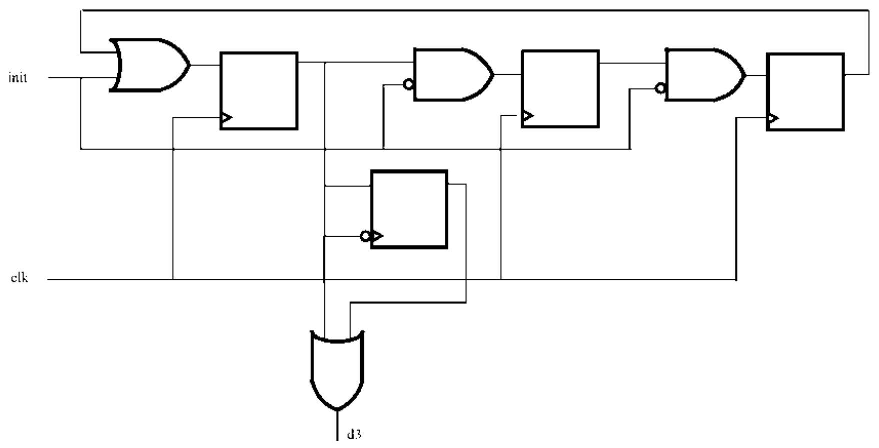
you can characterize xclock as

```
Net xclock clk invert ones
```

since the signal xclock (relative to the inversion of clk) is 1 when inverted clk is 1 and can be either 1 or 0 when inverted clk is 1.

3. A symmetrical divide-by-3 circuit is a good example of a circuit that you can describe as edges.

```
Net d3 clk edges
```



Only rising edges of d3 occur when clk rises, and only falling edges of d3 occur when clk falls.

-TNfi

SYNTAX:

-TNfi <filename>

ARGUMENT:

<filename>	This is the tie-off net filename.

USAGE NOTES:

It reads the tie-off net file. This turns on a feature that allows the user to use a file to specify nets that are tied to a constant value (0 or 1). The file syntax is as follows:

```
Net <netname> <value>

Terminal <terminal-name> <value>
```

where the *<value>* can be *one, ONE, zero*, or *ZERO*, the *<netname>* is a net's hierarchical name, and *Terminal* is a hierarchical name in the following form:

```
<modulename>.<terminal_name>
```

The specified net or the net connected to the specified terminal is tied to the constant that <*value*> specifies.

Specifically, if there is a reference to a net in one of the following files:

```
Net a.b.c
```

this reference will effect all nets which are aliases of the specified net. Aliases can result from *assign* statements,

```
assign c = d;
```

c and *d* will be aliases and any references to either will impact both. Aliases can result from hierarchical connections which are not at the user primitive level.

```
hier hier1 (.A(B), ...
```

B and *hier1.A* will be aliases and any reference to either will impact both.

In other words, tying a net on either side of a hierarchical boundary results in all fanouts of the nets on both sides of the boundary being tied.

The easiest way to understand the full implications of a tie or annotation on a net is to view the annotation as always occurring at the output of whatever primitive actually drives the net(s).

If there is a need to tie or apply annotations, like quasi-static or no-flow, to some fanouts of a net and its aliases without applying it to all fanouts, the way to do this is through *terminal* annotations.

```
Terminal a.b.d.E
```

If a.b.d is a user defined primitive and E is an input terminal, this annotation will just impact the net fanout reaching the relevant terminal. (*Terminal* annotation on output terminals are equivalent to *Net* annotations on the output net or any of its aliases.)

Be careful when using *Net* based annotation; *Terminal* annotations are safer.

Refer to *Net tie-offs on page 156* for additional information.

Refer to *vlc . browse_constants on page 159* for information on bringing up the *vrc* in order to see the value of netlist constants.

-Se

SYNTAX:
-Se

DEFAULT:

By default, enables are evaluated for primary IO in order to turnoff the IO as soon as possible.

USAGE NOTES:

Allows lazy (late evaluation of) tri-state enables on primary IO. The *-Se* allows the evaluation of IOs to be different until the next point at which the target system or the emulator sample the IO value.

-Sr

SYNTAX:
-Sr

USAGE NOTES:

Enables lazy reset. This switch allows a late evaluation of the asynchronous signals. The nature of the asynchronous set and reset requires analysis of the output of the register at all times. As a result, it increases the complexity of the design. If precise timing of asynchronous set or reset is not required for functionality, this switch holds the signal evaluation until the next clock edge.

Refer to *-NoSrfi on page 293* for additional information.

The modeling semantics of the *-Sr* switch in the VirtuaLogic 3.5 release, corresponds precisely to the conversion of affected state elements from using asynchronous preset and clear to using synchronous preset and clear. Preset and clear are only evaluated at active clock edges, (for flip-flops) or during open gate regions (for latches.)

Former semantics involved an immediate update of the state during any active preset or clear but a deferred propagation of this change to other parts of the design at the option of the emulation compiler. The behavior under this semantics could vary from compile to compile whereas the VirtuaLogic 3.5 behavior is precisely defined independent of a particular compilation.

One area of caution regarding this semantics is asynchronous preset or clear signals coming from primary asynchronous inputs or being generated in one clock domain and consumed in another. Under the former semantics, state would always be updated independent of the width of the active region of preset or clear. Under this semantics, the active region must include a clock edge of the relevant state element. Therefore, preset or clear pulses narrower than the clock period of the receiving state elements may be missed.

Synchronized preset or clear signal behavior takes precedence over clock enable terminals, for state elements containing synchronous clock enables and synchronized preset or clear in the reference library. Therefore, at clock edges, clear or preset takes place independent of the value of the clock enable if clear or preset are asserted.

As with the asynchronous versions of these signals, clear takes precedence over preset in all reference library primitives.

The 100% Visibility feature imposes some hardware overhead cost when compiled into models. Certain structures can be particularly expensive and may lead to more excessive costs. These include large storage macros implemented as gates, heavily latch-based design styles, large numbers and/or very high fanout cross-domain or asynchronous nets, particularly if the nets go to asynchronous preset or clear terminals.

Space improvement is possible by modeling storage macros as memories, treating high fanout asynchronous inputs as synchronous, using *-Sr* to convert asynchronous preset/clear modeling to synchronous preset/clear modeling using quasi-static annotations on the highest fanout cross-domain nets if they are not already quasi-static.

-NoSrfi

SYNTAX:

-NoSrfi <filename>

ARGUMENT:

<filename>	File for slow-reset instance overrides

USAGE NOTES:

Allows *-Sr* to be used for most state-elements while maintaining more precise modeling semantics for some set of user-specified state elements.

EXAMPLE:

Lazy reset is a modeling style for asynchronous preset/clear behavior of state elements that trades off precise timing behavior for improved emulation performance. When enabled, lazy reset defers asynchronous preset/clear induced output changes on state elements until the next active clock edge.

The user can selectively exempt state elements from lazy reset which allows imprecise modeling for most state elements while retaining the more precise modeling for the selected set.

Save the name(s) of the module(s) to a file and invoke the *-NoSrfi* switch with this file's hierarchical prompt on the command line.

To exempt certain modules from lazy resets, create a file with a series of lines having the following form:

```
Module <module_expr>
```

The *<module expr>* is a regular expression that matches the hierarchical path name of one or more modules in your design. For example,

Module A contains modules B and C, and module B contains modules D and E. To turn off lazy reset for module A.B.E, include this line in your file:

```
Module A.B.E
```

To turn off lazy reset on modules D and E with a single line, include the following line:

```
Module A.B.*
```

The following conditions constitute default behavior:

- If you do not choose the *-Sr* switch, lazy reset modeling does not apply to any modules
- If you choose the *-Sr* switch, but not the *-NoSrfi filename* switch, lazy reset modeling applies to all relevant modules
- If you do not choose the *-Sr* switch, but choose the *-NoSrfi filename* switch, no modules have lazy reset turned on
- If you choose the *-Sr* switch and the *-NoSrfi filename* switches, you enable lazy reset modeling for all relevant modules except for those modules listed in the file *<filename>*

Under the following conditions, errors occur:

- If the file specifies modules whose literal or wildcard-substituted names cannot be matched, the program prints out a warning message but does not exit
- If you choose the *-NoSrfi* switch, but do not indicate a filename, the compiler errors out and prints an error message
- If the word *Module* does not precede each module name, the compiler errors out and prints an error message

-noXOT

SYNTAX:

-noXOT

USAGE NOTE:

Prevents latch open transformations.

-noXCT

SYNTAX:
-noXCT

USAGE NOTES:

Prevents latch close transformations.

-noXFT

SYNTAX:

-noXFT

USAGE NOTES:

Prevents flip-flop identification.

-noXIAT

SYNTAX:
-noXIAT

USAGE NOTES:

Disables latch flow analysis/optimization. Refer to *-noXSAT on page 296* for information on the complementary operation.

-noXSAT

SYNTAX:
-noXSAT

USAGE NOTE:

Prevents latch flow analysis/optimization.

-noXTAT

SYNTAX:
-noXTAT

DEFAULT:

off

USAGE NOTES:

Disables latch tri-state flow analysis/optimization. Refer to *-XTAT on page 296* for information on the complementary operation.

-XTAT

SYNTAX:

-XTAT

DEFAULT:

off

USAGE NOTES:

Enables/disables latch tri-state flow analysis/optimization. Refer to *-noXTAT on page 296* for information on the complementary operation.

-SDPN

SYNTAX:

-SDPN

USAGE NOTES:

Saves dropped probe nets and saves probed nets from optimizing out. Suppresses elimination of dead logic on probed nets.

When using this switch, the compiler does not optimize out dead logic if the logic has been probed.

-fifo_refold_port_limit

SYNTAX:
-fifo_refold _port_limit <integer>

ARGUMENT:

<integer>	Specifies maximum number of ports above which no fifo refolding is done.

USAGE NOTES:

It arises as part of the memory resynthesis phase in VLE. Firstly, multi-domain memories get split by their data-bits into multiple slices that are the same width as the physical SRAMs in the emulator. The problem with this data slicing is that a large number of effective FIFOs are obtained, which don't share physical memories very easily. So it is possible to run out of physical SRAMs in the emulator memories on which the FIFOs are put. FIFO refolding will take these sliced FIFOs and rejoin them. This uses the physical SRAMs more frugally, but at a cost of performance because of the increase in the vcycle count.

The fifo_refold_port_limit allows the user or AE to increase performance at the cost of memory capacity, or vice versa.

If multiport FIFOs are determined to be the performance timing issue, choose a value equal to the number of ports of the memory limiting performance.

-noclkopt

SYNTAX:
-noclkopt

USAGE NOTES:

Suppresses clock folding.

-clkopt

SYNTAX:
-clkopt <factor> <domain_name>

ARGUMENTS:

<factor>

<domain_name>

DEFAULT:

The default folding factor is the ratio of the fastest clock in a domain to the slowest clock in the domain.

USAGE NOTES:

Uses the nondefault folding factor for named domain.

This implements clock folding which can reduce the resources required at an expense of emulation speed.

Use this only if the design has clocks that are exactly 1/*n* the frequency of the other clocks in the domain.

EXAMPLE:

For example, this option is used when two clocks in the one domain exist and one is twice the frequency of the other. Where *n* is the number of ways of folding, *n* is two.

XCrossDomainIO

SYNTAX:

XCrossDomainIO

DEFAULT:

By default, the emulation compiler treats primary IO with mixed clock domains, an internal clock domain which is different from the specified external clock domain as an error.

USAGE NOTES:

Prior to the VirtuaLogic 2.1 release, this construct produced a warning message but continued to compile. In most cases, it appeared that this behavior was a result of a misspecification on the part of the user; therefore, it has been made into an error.

If this construct is really desired, add the compiler option *XCrossDomainIO*. This option causes the construct to produce a warning instead of an error.

-XFTL

SYNTAX:

```
-XFTL <filename>
```

USAGE NOTES:

-XFTL is used to enable the selective conversion of flip-flops into functionally equivalent master-slave latch pair structures.

The option takes a filename argument which contains lines with the following syntax:

Module <module-expression>

or

Net <net-expression>

For the first syntax, *<module-expression>* is a regular expression matching the name of one or more module instances within the design. Any instances identified which are flip-flops are converted into functionally equivalent master-slave latch structures.

The second syntax, *<net-expression>* is a regular expression matching the name of one or more nets within the design. For any such net, all flip-flop instances whose clock terminal is combinatorially reachable either directly or indirectly from the specified net are converted into equivalent master-slave latch structures.

There are two typical scenarios in which -XFTL is used.

A. Synchronous data-dependent clocks delivered on primary data inputs. VLE does not properly model circuits in which a primary data input, (as opposed to a primary clock input) delivers the edges that clock internal flip-flops, either directly or indirectly. This constraint does not hold for latches, which can be gated by arbitrary signals, data or otherwise. As a result, *XFTL* can and must be used to convert flip-flops to latches whenever a primary data input supplies the clock edges for any design flip-flops.

   One example of such a structure occurs in AGP interfaces, in which AGP strobes, which are typically handled as data signals in VLE, deliver the clock by which the AGP data bus is sampled within some AGP interface implementations.

   In such a case, using a

```
Net <net-expression>
```

   XFTL directive which lists the internal AGP strobe nets serves to trigger the transformation, needed for a correctly functioning emulation model.

B. Reducing clock cost of high-cost clock structures

   Clock logic processing of flip-flops and latches differs for VLE. As a result, it may be advantageous from a resource usage standpoint to convert some set of flip-flops into equivalent latch structures in order to use less clock logic space. This can be conveniently achieved using *XFTL* with a net argument which is a common ancestor in the clock tree for the desired flip-flops.

-noclockblocks

Syntax:

```
-noclockblocks
```

Usage notes:

Use of the *-noclockblocks* switch enables a new algorithm for handling user clock logic. Prior to this new algorithm, in some cases, complex user clock logic could result in either of two problems

1. Unexpectedly high FPGA counts due to excessive clock logic

2. Unsatisfactory FPGA timing constraints resulting in PAR failures or very long FPGA compilation times.

These problems were typically avoided through the use of *-XFTL* and/or *-NCfi* switches.

Use of the new clock handling algorithm enabled by *-noclockblocks* avoids the need for using either *-XFTL* or *-NCfi* for the purpose of minimizing clock logic.

Control of output simulation models

-Vbc

SYNTAX:

-Vbc [<#>]

ARGUMENT:

<#>	Limits maximum size of scope in virtualized model

USAGE NOTES:

Use this option if the simulation model is too large.

-vhn

SYNTAX:
-vhn

USAGE NOTES:

Writes hierarchical version of virtualized model.

-vsn

SYNTAX:
-vsn

USAGE NOTES:

Uses short (nonuser) names in virtualized model.

-xln

SYNTAX:
-xln

USAGE NOTES:

Uses long (user) names in FPGA netlists.

If there is a need to debug XFF files, use the *-xln* switch.

This will print out long names into the XFF files. These names should have enough information so that it can be determined where they originated from in the original hierarchical design.

The *-xln* option may have some naming bugs and should be used with caution because it may cause FPGA compiles to fail.

If there is a case where there is a need to know what is in an FPGA, do the following:

- edit *params.mak*
- change *-Po* to -Pi
- change *-Pfo* to *-Pfi*
- add *-xln* to the *Compiler Options* pane
- recompile and then debug the problem
- remove *-xln* before the next "real" model build

-Vo

SYNTAX:

-Vo <filename>

ARGUMENT:

<filename>	Filename for detailed virtual wires simulation model. It is recommended that *mod* *vw.v* is used for the *<filename>*; however, it is not required to name the file this.

-vhdlout

SYNTAX:

-vhdlout

USAGE NOTES:

Produces output simulation models in VHDL.

Miscellaneous

-v

SYNTAX:

-v

USAGE NOTES:

Print version information and exit.

-q

SYNTAX:

-q

USAGE NOTES:

Quiet mode which suppress warning printouts.

-vs

SYNTAX:

-vs

USAGE NOTES:

Verilog parser warning control.

-vc

SYNTAX:

-vc

USAGE NOTES:

Verilog parser warning control.

-vw

SYNTAX:

-vw

USAGE NOTES:

Verilog parser warning control.

-Terse100

SYNTAX:

-Terse100

USAGE NOTES:

Designs that implement memories with bit-wide write enables by using an extra read port suffer from excessive automatic probing for 100% visibility. The read port is only used to feed back the "old" values to be muxed with "new" values by the bit write enables, driving a write port. This extra read-port does not need to be probed to get 100% visibility to work. So in cases where the extra probing is a constraint, a new switch -Terse100 can be employed to avoid automatically probing such read ports. This switch suppresses automatic probing of all memory read bits that fan out only to other memories.

-TerseProbe

SYNTAX:

```
-TerseProbe
```

USAGE NOTES:

Turnoff probing of distinct signal values that the netlist does not use.

Every multitransitional net that is probed uses probe channels. This switch prevents the multitransitional nets from using extra channels if the design uses only one value for the net.

-Dump

Most of the options used with this switch are helpful in the debugging stages.

SYNTAX:

```
-Dump    |option | c | f| g | m0 | m1 |m2 | p | q | t | 1 | h ||    <filename>
```

ARGUMENTS:

c	Lists epoch-critical path limits for optimization performance.
f	Prints paths that combinatorially pass through the emulator without being registered. It is used for splitting a design across multiple emulators. The information created by *-Dump f* indicates the computed internal timing behavior for all terminals, in addition to listing feedthroughs. This additional information can be used to validate the legitimacy of a candidate multibox partitioning of a design and guide the timing interface specification for this partitioning.

g	Generates a histogram of the primitive cells used in the design of all the VirtuaLogic primitives that the vendor libraries are mapped to and the number of times they are repeated throughout the design.
m0	This switch dumps information about each clock network net. It dumps the following information for a clock network net: clock name, clock time signature, clock source scalar and vector gate cost, and the number of destinations for the clock net. Refer to the EXAMPLE section below for an example of the dump format.
m1	Lists latches that have been converted during the compile.
m2	List the paths for combinatorial loops in the design. For any combinational loop questions, adding *-Dump m2* · *dumpfile* to the options line of *param.mak* and rerunning the compiler will display a list of the nets involved in the loop.
p	Lists design elements removed by dead logic elimination passes. It adds the names of all dead logic cells to the *vmw.dump* output file.
q	Lists multitransition, multisample nets. For multifanout nets, it specifically identifies individual fanout terminals as the root of MTSD (multitransition/sample domain) regions, easing the creation of terminal specific annotations in the quasi-static file. The quasi-static dump contains text which can be directly cut and pasted to create suggested terminal quasi-statics. Also, the *dump* always references its information to some specific, identifiable user design net or terminal. Under some circumstances where MTSD convergencies occurred inside of user primitives, the *dump* file referenced nets which did not correspond obviously to any net in the user design.
t	Compile will terminate prematurely and prints a histogram of the number of distinct values associated with nets in the design.
1	This is the default. It is very useful when debugging because it retains the long original signal names where

	appropriate. Without the *l* option, the user signal names are not retained and these compiler generated net names are seen instead.
h	This retains the Verilog hierarchy of the design and is only required when the Verilog testbench is accessing the "internal" of the gate-level portion via hierarchy references. The *h* option should not be used otherwise. The reason not to use *-Dump h* is it leads to longer verify compile times (more to write out), longer verify simulation run times (more to process), and larger *vpd* dumps (harder to deal with in *VirSim*). Refer to *-vhn on page 301* for information on hierarchy preservation.

DEFAULT:

l

USAGE NOTES:

Long names are always added to the simulation models and the use of maintaining hierarchy via the *-Dump h*, *-Dump l*, and *-vhn* switches are only beneficial for simulations under certain conditions. The key word here is simulation as opposed to emulation.

If there is a need to debug XFF files, use the *-xln* switch. Refer to *-xln on page 302* for details.

The hierarchy referencing for probing and triggering is taken care of for the user automatically behind the scenes with no switches required.

Refer to *-vhn on page 301* for information on hierarchy preservation.

EXAMPLE:

Following is an example of the dump format:

```
Clock-network net VMWnet137_VLA_68

TDomain domain0 SDomain domain0    Source Clock    Sampled
Transitions: |.|.
Samples    : |.|.
Known      : 1100
Up         : |...
Down       : ..|.
```

```
State-altering transitions:
        |...
Clock source cost: 10, vcost: 4.6.0.0
Destination Count: 32
Clock-network net VMWnet100_VLA_68
```

Arguments not set manually

The following arguments cannot be set manually because they are controlled by the scripting processes:

-Lib

SYNTAX:
```
-Lib <filename>
```

ARGUMENT:

<filename>	Specifies the user library mapping file

-DB

SYNTAX:
```
-DB <filename>
```

ARGUMENT:

<filename>	Identifies the name of the database file

-Nb

SYNTAX:
```
-Nb [<#>]
```

ARGUMENT:

<#>	Specifies the number of boards to target

-NPb

SYNTAX:

-NPb [<#>]

ARGUMENT:

<#>	Specifies the number of FPGAs to use

-Root

SYNTAX:

-Root

USAGE NOTE:

Root design name

-Clk

SYNTAX:

-Clk <filename>

ARGUMENT:

<filename>	Specifies the clock file name

-Mem

SYNTAX:

-Mem <filename>

ARGUMENT:

<filename>	Specifies the memory file name

-ProbeIn

SYNTAX:

-ProbeIn <filename>

ARGUMENT:

<filename>	Probe list file

-ProbeWindows

SYNTAX:

-ProbeWindows <filename>

ARGUMENT:

<filename>	Probe window list file

-ProbeCard

SYNTAX:

-ProbeCard [<#>]

ARGUMENT:

<#>	Specifies the number of HP logic analyzer cards

-ProbeCore

SYNTAX:

-ProbeCore <filename>

ARGUMENT:

<filename>	Output core probe list file

-ProbeMap

SYNTAX:

-ProbeMap <filename>

ARGUMENT:

<filename>	Probe map file

-ProbeDB

SYNTAX:

-ProbeDB <filename>

ARGUMENT:

<filename>	Database file for incremental probe

-IncProbe

SYNTAX:

-IncProbe

USAGE NOTE:

Enables incremental probe

-MultiAsic

SYNTAX:

-MultiAsic <filename>

ARGUMENT:

<filename>	Multi-ASIC control file

-Pod

SYNTAX:

-Pod <root_module>.pod

ARGUMENT:

<root_module>.pod	Terminal constraint filename

USAGE NOTES:

This switch reads in the file with the I/O terminal specifications.

-target

SYNTAX:
    -target <filename>

ARGUMENT:

<filename>	Target topology file

-targetfile

SYNTAX:
    -targetfile <filename>

ARGUMENT:

<filename>	Reads target topology filename from the file

-syspart

SYNTAX:
    -syspart <fpga_name>

ARGUMENT:

<fpga_name>	Specifies system board FPGA type

-arrpart

SYNTAX:
    -arrpart <fpga_name>

ARGUMENT:

<fpga_name>	Specifies array board FPGA type

-Mc

SYNTAX:
    -Mc

USAGE NOTE:

Enables the partitioner

-memmap

SYNTAX:

-memmap

USAGE NOTE:

Memory map file name

-define-

SYNTAX:
-define-

USAGE NOTE:

Undefines symbol

-defines_file

SYNTAX:
-defines_file <filename>

ARGUMENT:

<filename>	File containing the list of defines and undefines

-bond

SYNTAX:
-bond

USAGE NOTE:

Lists hierarchical modules for "bonding" out

-Ti

SYNTAX:

-Ti <filename>

ARGUMENT:

<filename>	Topology input file

-Mo

SYNTAX:

-Mo <filename>

ARGUMENT:

<filename>	I/O map file name

-Ao

SYNTAX:

-Ao <string>

ARGUMENT:

<string>	Base name for auxiliary files

-Xo

SYNTAX:

-Xo <filename>

ARGUMENT:

<filename>

-hvpd

SYNTAX:

-hvpd <filename>

ARGUMENT:

<filename>	Filename for hierarchy specifying *.vpd* file

-Po

SYNTAX:
-Po <root_module>.part

ARGUMENT:

<root_module>.part	Writes the *<root_module>.part* file containing the partition information

USAGE NOTES:

Always use this with the first compile. The *<root_module>* file provides information regarding the logic partitioning.

Always write out the partition file *-Po . design .part*. This is the only way to reproduce partitioning problems.

-Pi

SYNTAX:
-Pi <root_module>.part

ARGUMENT:

<root_module>.part	This switch uses the previous partition. It reads partition results to the filename.

USAGE NOTES:

Reads a previously generated partition file in a new compilation where the HP logic analyzer or number of board has changed, but not the design. Using this switch reduces the compile time significantly.

-Pfo

SYNTAX:
-Pfo <root_module>.place

ARGUMENT:

<root_module>.place	Writes the *<root_module>.place* file containing the placement information

USAGE NOTES:

This is similar to the partition file. It can be used in the future compilations of the same design.

-Pfi

SYNTAX:
-Pfi <root_module>.place

ARGUMENT:

<root_module>.place	Reads in a previously generated placement file

USAGE NOTE:

Use this with a new compile when you need to reproduce a problem.

Compiler options listed by category

The compiler options listed below can be entered manually by the user in the *Compiler Options* field. Most of these options should not be used without careful testing, especially if used in combinations. Refer to *Table 10 on page317* for a list of options that can be used in combination.

Table 10 Compiler options that can be used together

Process	**Setting**	**Explanation**
Precluster	-Pdo 20	Increases the size of the precluster blocks to 200% of normal. This reduces compile time by making fewer clusters for the timing partitioner to handle. **Use this parameter for all designs to reduce partition times.**
Timing	-Tlo 20	Increases the size of the timing partitioned blocks to 200% of normal. This reduces compile time by making fewer blocks for the mincut algorithm to handle. **Use this parameter for all designs to reduce partition times.**
Mincut	-Mm 9	Decreases the amount of logic allowed in each chip to 90% of normal. This improves FPGA Place and Route time by making each chip easier to place and route. However, it may make the design run slightly slower due to fewer gates in each chip and more potential chip hops required in a path. It also will less efficiently use the hardware capacity and is not recommended if a design is a tight fit in the emulator. If a design is quite small, a setting of eight can be used. Increasing this number can provide more utilization of each chip, but will increase FPGA Place and Route time. **Use this if a design is small relative to the hardware resources available or if a design is only slightly too large for the hardware resources.**
Partition	-Pi <file name>	**Use this if the design has not changed, but HP logic analyzer or number of boards has.**

Table 10 Compiler options that can be used together (Continued)

Process	Setting	Explanation
Clocks	-clkopt domain_name *n*	Implements clock folding which can reduce the resources required at the expense of emulation speed. For example, this option is used when two clocks in the one domain exist, and one is twice the frequency of the other. Where *n* is the number of ways of folding. In the example, *n* would be two. **Use this only if the design has clocks which are exactly 1/*n* the frequency of other clocks in the domain.**
Verilog	-Vo <file name>	This generates a Verilog file which represents the final Virtual Wires implementation. It is useful during the debug of the design. **Always use this.**

vlc commands

vlc

SYNTAX:

```
vlc <config_name> <command>
```

USAGE NOTES:

When using *vlc*, all the arguments are filled in for the user based on the information entered in *gvl*.

If the *vlc* command is run from within the *config_name.vmw* directory, then it can be referenced as the current directory. The command would look like this example, where the "." references the current directory.

```
cd *.vmw

vlc . vtask < /dev/null &
```

The *vlc* command reads the user input files that are listed in *Table 3 on page163* and executes the needed tasks to prepare a design for in-circuit emulation.

rtl

SYNTAX:

```
vlc <config_name> rtl
```

USAGE NOTES:

Performs RTL compile.

rtl_compile

SYNTAX:

```
vlc <config_name> rtl_compile
```

USAGE NOTES:

Performs RTL & VLE compiles.

rtl_compile_vtask

SYNTAX:

```
vlc <config_name> rtl
```

USAGE NOTES:

Performs RTL, VLE & FPGA compiles.

verify

SYNTAX:

```
vlc <config_name> verify
```

USAGE NOTES:

This generates the netlist for the simulation verification.

compile

SYNTAX:

```
vlc <config_name> compile
```

USAGE NOTES:

This runs the *vsyn* compile from the command line prompt.

all

SYNTAX:

```
vlc <config_name> all
```

USAGE NOTES:

This runs the *FPGA compile* with one command.

The *pprrun* and *all* commands are interactive. In order to run them in the background, it is recommended that you ignore inputs as shown in the following example.

```
vlc <config_name> all < /dev/null &
```

pprclean

SYNTAX:

```
vlc <config_name> pprclean
```

USAGE NOTE:

This removes the Xilinx files..

Do not use *pprclean* if you want to emulate the database. The *pprclean* will remove all the bit files for the Xilinx devices that are used for the download.

browse

SYNTAX:

```
vlc <config_name> browse
```

USAGE NOTE:

This invokes the Motif hierarchy browser on the design.

vprobe

SYNTAX:

```
vlc <config_name> vprobe
```

USAGE NOTE:

This invokes the HP logic analyzer control program.

rvirsim

SYNTAX:

```
vlc <config_name> rvirsim
```

USAGE NOTE:

View waveforms.

tar

SYNTAX:

```
tar TAR_FILE=archivename.tar
```

USAGE NOTE:

Generates a compressed archive of configuration, netlists, and miscellaneous files.

The *tar* creates a file that can be sent to IKOS for debug if there is any problem with the configuration. It saves the needed files to duplicate the configuration setup. This command can also be used to archive a design without requiring storage space for the large database.

vtask commands

add

SYNTAX:

```
add <machine_name>
```

or

```
add <machine_name>/pc
```

USAGE NOTES:

Adds a machine to the FPGA compile machine pool. This does not change the *machlist.mach* file.

remove

SYNTAX:

```
remove <machine>
```

USAGE NOTES:

Removes a machine from the FPGA compile machine pool. This does not change the *machlist.mach* file.

newlist

SYNTAX:

```
newlist [-l] [-f <newlist>]
```

ARGUMENTS:

<newlist>	Replace virtual machine list with *<newlist>* file. Defaults to *machlist.mach*.
-l	This allows processes on removed machines to live. Otherwise, active processes on removed machines are terminated.

nice

SYNTAX:

```
nice <niceness>
```

USAGE NOTES:

Uses *<niceness>* on subsequently spawned tasks.

status

SYNTAX:

```
status
```

USAGE NOTES:

Shows the status of tasks and machines.

exit

SYNTAX:

```
exit
```

USAGE NOTES:

Stops all tasks and exit with nonzero status.

quit

SYNTAX:

```
quit
```

USAGE NOTES:

Stops all tasks and exit with nonzero status.

help

SYNTAX:

```
help
```

USAGE NOTES:

Help page for interactive mode.

IKOS

10 Syntax, Semantics, and Reference Library

Overview

The following conventions apply to all non-Verilog files, including memory, timing, pin constraints, probe, etc.

- Blank lines are allowed
- A pound sign (#) at the beginning of a line indicates a comment

VirtuaLogic structural verilog subset

The VirtuaLogic compiler accepts a structural subset of Verilog as the input format for designs. This subset includes Verilog syntax used for structural module definition and instantiation. All Verilog code in the input netlist files that are a meaningful component of the design must use this structural subset. The supported Verilog subset does not include gates and devices (e.g., `and`, `or`, `bufif`, `pullup`, `nmos`, and `tran`).

Design files can contain behavioral code that is not part of the design, but rather aids in design verification. Behavioral code can occur in the following situations:

- Monitoring signals
- Checking for setup, hold time, or other timing violations
- Providing stimulus

The user can direct the compiler to ignore silently, warn, or produce errors when it encounters such constructs, but it can never recognize them as a meaningful part of the design.

Verilog identifiers

Following are the Verilog identifiers:

- A sequence of letters, digits, underscore (_), or dollar sign ($) that starts with a letter or underscore
- Any sequence of printable ASCII characters starting with a backslash (\) and terminated by a space

Module definition syntax

The structural Verilog subset recognized by the Verilog compiler includes the following Verilog syntactic constructs:

```
module <module_type_name> (parameter_list);

{input|output|inout} <io_list>;

{wire|tri} <wire_name_list>;

<module_type> [<module_name>] (parameter_bindings);

endmodule;
```

Parameters

```
<module_type_name>:= <verilog_identifier>

<parameter_list>:= <parameter>,...

<parameter>:= <parameter_name>
OR

<parameter_name> (parameter_rename)

<io_list>:= <parameter_name>....

<parameter_name>:= <verilog_identifier>

<wire_name_list>:= <wire_name>,...

<wire_name>:= <verilog_identifier>

<module_type>:= <user_defined or library_module_type_name>
```

<module_name>:= <verilog_identifier>

<parameter_bindings>:= <signal_name>,<signal_name>....
OR

<parameter_name> (signal_name)....

<signal_name>:=<verilog_identifier>

All legal scalar and vector signal name syntax is supported, including scalar references, vector references, vector bit and range selects, concatenations, and multiple CONCATENATIONS .

Example

```
module full-adder (a, b, cin, sum, cout);
input a, b, cin;
output sum, cout;
xor3 sumxor (.A(a), .B(b), .C(cin), .Z(sum));
A0222 carry (.A(a), .B(b), .C(a), .D(cin), .E(b), .F(cin), .Z(cout));
endmodule;
```

Simple assignments

Assignments can be used for any direct assignment from a signal or collection of signals to a signal or collection of signals.

Example

```
assign x = y;
{a, b, c}= vec[2:0];
```

Compiler directives

The following Verilog compiler directives are supported:

```
'include        'define        'undef
```

```
`ifdef          `else          `endif
`<macro_name>
```

Unsupported verilog constructs

The VirtuaLogic compiler does not recognize behavioral Verilog constructs. Unsupported constructs are listed in *Table 11 on page328*.

Table 11 Unsupported verilog constructs

Execution Control	`always`
	`initial`
	`event`
	`@`
	`#`
	`task`
	`function`
	`fork`
Conditionals	`if then else`
	`? :`
	`case`
Arithmetic	+, -, *, / etc.
Logic	|, &, ^, etc.
Other	specify

Since a netlist can have nonsemantic behavioral consistency checking code built in, the VirtuaLogic compiler provides three user options for dealing with behavioral code. From the *Compiler* form, in the *Compiler Options* pane, enter the appropriate option for how the behavioral code is to be treated and then click on *Run Compiler*. The options are described in *Table 12 on page328*.

Table 12 Compiler options for behavioral code

Compiler Option	Command Syntax	Way To Deal With Behavioral Code
Ignore silently	**-vs**	The VirtuaLogic compiler skips over any unsupported behavioral code without a warning.
Ignore but warn	**-vw**	The VirtuaLogic compiler skips over any unsupported behavioral code which produces a warning message identifying the file and line number of the occurrence of the unrecognized construct. This is the default.

Table 12 Compiler options for behavioral code

Compiler Option	Command Syntax	Way To Deal With Behavioral Code
Error	**-ve**	The VirtuaLogic compiler produces error messages as above and terminates after input parsing if any unsupported code has been encountered.

Memory specification

Textual syntax

The syntax consists of keywords and variables for specifying memory attributes and terminal binding information.

Memory attributes

The following list comprises keywords and variables the user enters to describe various attributes of the memory.

```
memory <memory_name>
sensitivity {{edge|level|level-unordered}|{0|1}}
ports {read|write}*
address-size <number>
data-size <number>
contents-file <filename>
```

Terminal bindings

The following list comprises syntax for specifying terminal binding information:

```
terminal <terminal_name> <functionality_spec>[<terminal_index>]
          OR
terminal <vector_terminal_name> <functionality_spec>
          OR
terminal <vector_terminal_name>[<left_index>:<right_index>]
 <functionality_spec>[<left_index>:<right_index>]
```

```
<functionality_spec> = terminal_type:port_number
<terminal_index> := number
<terminal_type> := address|data|we|oe
<port_number> := number
<left_index> := number
<right_index> := number
```

A *<functionality_spec>* consists of a *<terminal_type>*, indicating the following:

- The nature of the terminal is address versus data enable
- A port number indicating the port to which the terminals belong. Ports receive numbers, starting at 0, and correspond to the ports specified using the *ports* keyword

When terminal names for address or data ports use a vector syntax, you can use this within the memory specification to describe the functionality of the entire vector terminal with a single line.

When terminal names are scalars, you must describe each terminal within the address and data buses of every port with a separate line.

For example,

```
terminal AADR[7:0] address0[7:0]
terminal BADR[7:0] address1[7:0]
terminal ADO[7:0] data0[7:0]
terminal BDO[7:0] data1[7:0]
terminal CDIN[7:0] data2[7:0]
terminal CADR[7:0] address2[7:0]
```

Note that the write enable and output enable names can be abbreviated to *we* and *oe*, respectively.

```
terminal WEN write-enable2
```

Semantics

The semantics of keywords and terminal binding formats are summarized in *Table 13 on page332.*

User Input	Memory Attribute	Description
memory `<memory_name>`	Memory Module Name	Name of a module to model
sensitivity `{{`**edge**`|`**level**`|`**level-unordered**`}|{`**0**`|`**1**`}}`	Sensitivity	Indicates if memory is modeling edge- or level-sensitive storage or the polarity of write enable signals
ports `{`**read**`|`**write**`}`*****	Port Directions	Direction (read or write) of each port of the memory
address-size `<number>`	Address Width	Number of address terminals of each memory port
data-size `<number>`	Data Width	Number of data terminals of each memory port
contents-file `<filename>`	Initial Content	Name of a Verilog file in the format used by the Verilog function *$readmemb()* which specifies the initial contents of the memory; this is optional

`<terminal> <terminal_name> <functionality_spec>[terminal_index]` OR `<terminal> <vector_terminal_name> <functionality_spec>` OR `<terminal> <vector_terminal_name> [left_index:right_index] <functionality_spec> [left_index:right_index]`
 	
 `<terminal_index>:=<number>` `<functionality_spec>:= <terminal_type>:<port_number>`
 	
 `<terminal_type>:={address|data|we|oe}`
 	
 `<port_number>:=<number>`
	Terminal Name Mapping	Description of the semantics of each terminal on the memory module
		
		 The semantics include:
		• The index of the pin within terminals of the same type; that is, you can describe the second address pin of the first memory port
		• The type of the terminal (address, data, write enable, output enable)
		• The port of which the terminal is a member

Timing specification

Syntax

Following is the timing specification syntax. The syntax comprises keywords and variables for specifying the timing characteristics of clock and data signals of the design:

```
domain <domain_name>
clock {<terminal_name> |external [<clockname>]] |
        <netname> internal [<clockname>]}
edges <clockname> <direction> *
  <direction> := {{rise|r|1}|{fall|f|0}}
data <terminal_name> {Input|Output} <clockname> <direction>
  <direction> := {{rise|r|1}|{fall|f|0}}
No-connect <terminal_name>
Zero <terminal_name>
One <terminal_name>
Outclock <terminal_name>
Feedthrough <terminal_name> <terminal_name>
```

Semantics

This specification contains several pieces of information for each timing type as outlined in *Table 13 on page332.*

Table 13 Timing specification elements

User Input	Description
domain <domain_name>	A new clock domain with name for identification in the file

Table 13 Timing specification elements (Continued)

User Input	Description
clock <terminal_name> [**external** [<clockname>]] OR **clock** <netname> **internal** [<clockname>] **Restriction:** If <clockname> is omitted, it defaults to the name of the terminal or net If the keyword is omitted, it defaults to external	A new clock within a domain can be: • External – requires: – Name of a top-level design I/O corresponding to the clock – Keyword **external** – A name for subsequent reference to the signal later in the timing specification (optional) • Internal – requires: – The hierarchical name of the net whose behavior you are describing – The keyword **internal** – A name for subsequent reference within the file
edges {<clockname> <direction>}*	Specifies the periodic sequence of clock edges within the domain
<direction> := [{rise|r|1}|{fall|f|0|	Rise and 1 indicate a rising clock edge Fall and 0 indicate a falling clock edge
data <terminal_name> {**Input**|**Output**} <clockname> <direction>	Specifies I/O timing for a terminal: • **Input**|**Output** specifies one direction of a bidirectional signal • <clockname> and <direction> identify a clock edge
No-connect <terminal_name>	Identifies an I/O that is unused in emulation
Zero <terminal_name>	Indicates an input that should be internally tied to zero
One <terminal_name>	Indicates an output that is used as a check by the target system
Outclock <terminal_name> <terminal_name>	Indicates an output that is used as a clock by the target system
Feedthrough	Indicates combinatorial feedthrough between the two terminals of the target system

Probe list format

The probe list is a collection of one or more files that specify the set of signals within the design netlist that are to be probed using *VirtualProbe*.

Textual syntax

The probe list is simply a list of signal names, one per line.

```
<Signal_Name>:= <Scalar_Signal_Name>
             := <Vector_Signal_Name>
             :=
<Vector_Signal_Name>[<left_index>:<right_index>]
```

Signal Name is the complete hierarchical name of a wire or module terminal within the design. You can omit the scope portion of a name which corresponds to the top-level module in the design. The name can refer to the following:

- Scalar wire or terminal
- Bit-select from a vector wire or terminal
- Range select of a vector wire or terminal

Scalar or bit-select names cause the probing of a single bit.

Range selects cause multiple bits, either the entire range or the entire vector, to be probed.

For example,

```
D1
Addr [31:0]
Sparcle.Regfile.Mux1_Sel
```

VirtuaLogic reference library

Table 14 on page334 describes the functionality of all the reference cells in the IKOS Reference Library.

Table 14 VirtuaLogic reference library

Types	Cell Name	Inputs	Output	Functional Description
Buffer	VMW_BUF	A	Z	Z=A

Table 14 VirtuaLogic reference library (Continued)

Types	Cell Name	Inputs	Output	Functional Description
Inverter	VMW_INV	A	Z	$Z=\overline{A}$
AND	VMW_AND2	A,B	Z	Z=AB
	VMW_AND3	A,B,C	Z	Z=ABC
	VMW_AND4	A,B,C,D	Z	Z=ABCD
	VMW_AND5	A,B,C,D,E	Z	Z=ABCDE
NAND	VMW_NAND2	A,B	Z	$Z=\overline{(AB)}$
	VMW_NAND3	A,B,C	Z	$Z=\overline{(ABC)}$
	VMW_NAND4	A,B,C,D	Z	$Z=\overline{(ABCD)}$
	VMW_NAND5	A,B,C,D,E	Z	$Z=\overline{(ABCDE)}$
OR	VMW_OR2	A,B	Z	Z=A+B
	VMW_OR3	A,B,C	Z	Z=A+B+C
	VMW_OR4	A,B,C,D	Z	Z=A+B+C+D
	VMW_OR5	A,B,C,D,E	Z	Z=A+B+C+D+E

Table 14 VirtuaLogic reference library (Continued)

Types	Cell Name	Inputs	Output	Functional Description
NOR	VMW_NOR2	A,B	Z	$Z=\overline{(A+B)}$
	VMW_NOR3	A,B,C	Z	$Z=\overline{(A+B+C)}$
	VMW_NOR4	A,B,C,D	Z	$Z=\overline{(A+B+C+D)}$
	VMW_NOR5	A,B,C,D,E	Z	$Z=\overline{(A+B+C+D+E)}$
XOR	VMW_XOR2	A,B	Z	$Z=A\overline{B}+\overline{A}B$
	VMW_XOR3	A,B,C	Z	$Z=A\overline{BC}+\overline{AB}C+\overline{A}B\overline{C}+ABC$
XNOR	VMW_XNOR2	A,B	Z	$Z=(AB+\overline{AB})$
	VMW_XNOR3	A,B,C	Z	$Z=(\overline{A}BC+A\overline{B}C+AB\overline{C}+\overline{ABC})$
AND-OR	VMW_AO21	A,B,C	Z	$Z=((AB)+C)$
	VMW_AO211	A,B,C,D	Z	$Z=((AB)+C+D)$
	VMW_AO22	A,B,C,D	Z	$Z=(AB+CD)$
	VMW_AO222	A,B,C,D,E,F	Z	$Z=(AB+CD+EF)$
AND-OR Invert	VMW_AOI21	A,B,C	Z	$Z=\overline{((AB)+C)}$
	VMW_AOI211	A,B,C,D	Z	$Z=\overline{((AB)+C+D)}$
	VMW_AOI22	A,B,C,D	Z	$Z=\overline{(AB+CD)}$
	VMW_AOI222	A,B,C,D,E,F	Z	$Z=\overline{(AB+CD+EF)}$
OR-AND Invert				
	VMW_OAI21	A,B,C	Z	$Z=\overline{((A+B)C)}$
	VMW_OAI211	A,B,C,D	Z	$Z=\overline{((A+B)CD)}$
	VMW_OAI22	A,B,C,D	Z	$Z=\overline{((A+B)(C+D))}$
	VMW_OAI222	A,B,C,D,E,F	Z	$Z=\overline{((A+B)(C+D)(E+F))}$
	VMW_OAI2222	A,B,C,D,E,F,G,H	Z	$Z=(\overline{(A+B)(C+D)(E+F)(G+H)})$
Mux	VMW_MUX2	A,B,S	Z	$Z=(\overline{S}A+SB)$
	VMW_MUX2I	A,B,S	Z	$Z=(\overline{SA}+S\overline{B})$
	VMW_MUX21L	A,B,S,SN	Z	$Z=((A\overline{S}\ SN)+(BS\ \overline{SN}))$
	VMW_MUX4	A,B,D0,D1,D2,D3	Z	$Z=((D0\overline{A\ B})+(D1A\overline{B})+(D2\overline{A}\ B)+(D3AB))$
Decoder	VMW_DEC24L	A,B	Z0,Z1,Z2,Z3	$ZO=A\overline{B}+\overline{A}B+AB$
				$Z1=\overline{AB}+\overline{A}B+AB$
				$Z2=\overline{AB}+A\overline{B}+AB$
				$Z3=\overline{AB}+A\overline{B}+\overline{A}B$

Table 14 VirtuaLogic reference library (Continued)

Types	Cell Name	Inputs	Output	Functional Description

Adder	VMW_HADD	A,B	CO,S	CO=AB
				S=$\overline{A}B+A\overline{B}$

	VMW_FADD	A,B,CI	CO,S	CO=$CIA\overline{B}+CI\overline{A}B+\overline{CI}AB+CIAB$
				S=$CI\overline{A}\,\overline{B}+\overline{CI}A\overline{B}+\overline{CI}\,\overline{A}B+CIAB$

IO-Buffer	VMW_IBUF	A	Z	Z=A
	VMW_OBUF	A	Z	Z=A
	VMW_OBUFZ	A,E	Z	Z=E?A:1bz;
	VMW_BUFIZ	A,E	Z	Z=E?A:1bz;
Flip-flop	VMW_FD	D,CP	Q	D-flop: Positive-edge clock
	VMW_FD2	D,CP	Q,QN	D-flop: Positive-edge clock with $\overline{Q}$ output
	VMW_FDE	D,CP,CE	Q	D-flop: Positive-edge clock with enable
	VMW_FDN	D,CPN	Q	D-flop: Negative-edge clock
	VMW_FDP	D,CP,PRE	Q	D-flop: Positive-edge clock with async preset
	VMW_FDP2	D,CP,S	Q,QN	D-flop: Positive-edge clock with async preset w/$\overline{Q}$
	VMW_FDPE	D,CP,CE,PRE	Q	D-flop: Positive-edge clock with async preset & enable
	VMW_FDC	D,CP,CLR	Q	D-flop: Positive-edge clock with asyn clear
	VMW_FDCE	D,CP,CE,CLR	Q	D-flop: Positive-edge clock with enable and async clear
NOTE: Preset, Clear and Enable for the Fliop Flops are active high signals				
Latch	VMW_LD	D,G	Q	Latch: Positive gate
	VMW_LD2	D,G	Q,QN	Latch: Positive gate with $\overline{Q}$
	VMW_LDP	D,G,PRE	Q	Latch: Positive gate & async preset
	VMW_LDC	D,G, CLRI	Q	Latch: Positive gate & async clear
	VMW_LDN	D,GN	Q	Latch: Negative gate
	VMW_LDN2	D,GN	Q,QN	Latch: Negative gate with $\overline{Q}$
NOTE: Preset, Clear for the Latches are active high signals				

IKOS

11 RTL Debug using the GUI

Overview

This chapter covers the RTLC debug capabilities for the user to debug their designs in an efficient way. The following important features of RTLC debug features are also covered.

- Interactive state control
- Source level debug
- Pruning

RTLC debug capabilities

RTLC-VLE supports enhanced debug features to allow the user to debug in RTL. During the process of compilation, an RTL debug database is written. The RTL debug database that RTLC-VLE generates, enables full RTL visibility into the user's design. This is used by various components of the VirtuaLogic system to present an RTL interface.

This database is also used to extract debug information for RTL-to-netlist and netlist-to-RTL name transformation and RTL source level debug. *Figure 60 on page 340* shows the RTL Emulation/Debug use model.

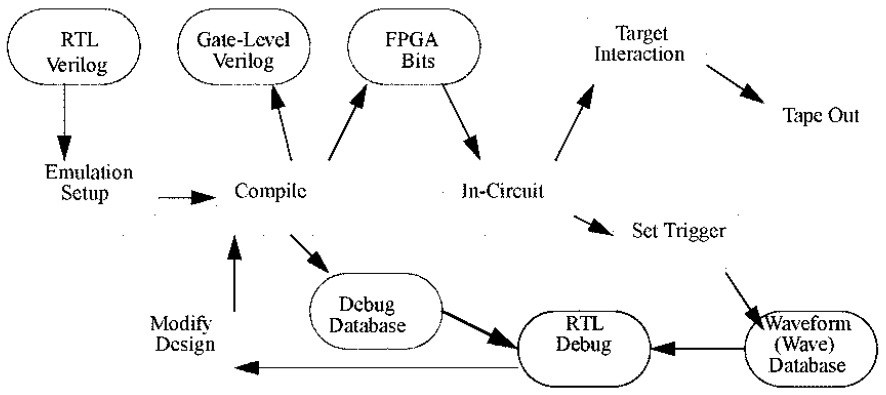


Figure 60 RTL Emulation/Debug use model

Interactive state control

In order to work on interactive state control, 100% visibility must be active for compilation.

- In-circuit visibility to flop outputs

  –vrun: state_ctrl -read_all statefile.txt -format h

A *-verilog* switch is added to the vrun *"state ctrl -read-all"* command in order to read the resulting state dump into verilog XL via an 'include statement in the module that instantiates the root model. Here the root module instance has to match with the root module name.

- State Setting (aka "Init") when out of circuit

  -vrun: state_ctrl -set statefile.txt -format h

- State Forcing (aka "Stick") with compile option

  –vsyn option: -force force_file

  -vrun: state_ctrl -force statefile.txt -format h

- If all state are made forcible, 20% capacity overhead beyond 100% visibility

Waveform file management

Filename	Purpose
rtlsrc.zip	RTL source archive for debugger display
recon.vdb	Gate Level Database
rtl.vdb	RTL connectivity Database
rtlc.out	Breakpoint Database
waveform100.vrc	100% visibility signal data
waveform100.map	100% visibility signal names

A waveform can be copied using tar and gzip of a *.wave* directory. *recon.vdb, rtl.vdb and rtlc.out* are shared via hardlinks. To view a wave file outside of *gvl*, type

$VMW_HOME/bin/vrc foo.wave/waveform.vrc

Other files in *.wave* directory can be extracted for 100% visibilty using these commands

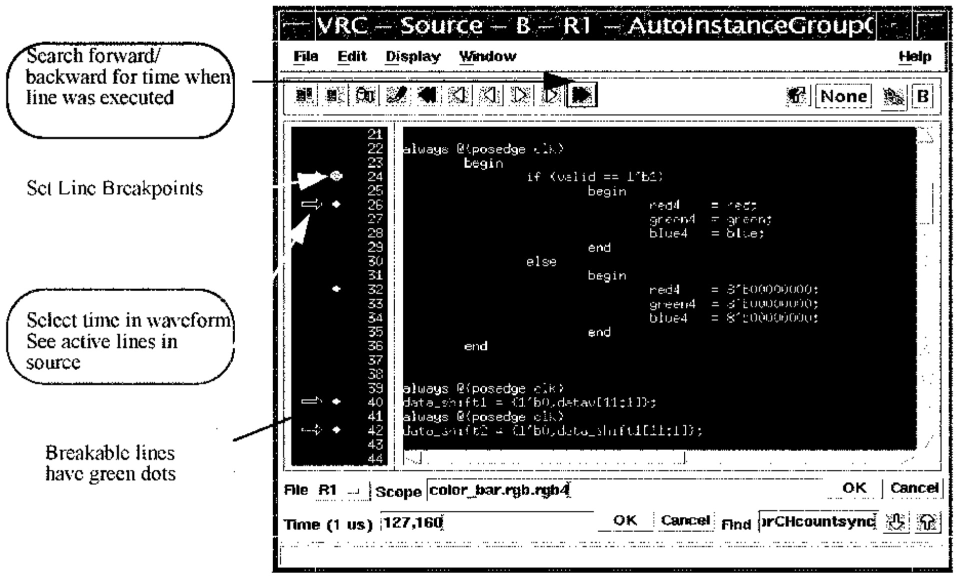
fifo_contents.vrc& .map

probed.vrc& .map

Source debug window

The source debug window displays the RTL source code. *Figure 61 on page 342* shows the source debug window. The arrows on the top move the user down the source code. The double arrows move the user down the break points or back to the previous breakpoint. The source debug window is time-linked with the waveform window. The times are correlated between the tools. *Virsim* tool has multiple time-linked windows, labelled A, B, etc. To link two windows together, give them the same letter by clicking on the little "chain" icon in the upper right.

In the source debug window, breakpoints can be set on lines of RTL code and search forward and backward through time for when those lines were hit. While the green dots are lines were breakpoints can be set, the red dots are current breakpoints. The break points are toggled on and off by clicking on the dots. The yellow arrows, on the other hand, are the breakpoints that were just hit and the blue arrows are the lines that are currently active.


Search forward/ backward for time when line was executed
Set Line Breakpoints
Select time in waveform See active lines in source
Breakable lines have green dots
VRC – Source – B – R1 – AutoInstanceGroup(
File
Edit
Display
Window
Help
None
B
File R1
Scope color_bar.rgb.rgb4
OK
Cancel
Time (1 us) 127,160
OK
Cancel
Find prCHcountsync

```
always @(posedge clk)
        begin
                if (valid == 1'b1)
                        begin
                                red4    = red;
                                green4  = green;
                                blue4   = blue;
                        end
                else
                        begin
                                red4    = 8'b00000000;
                                green4  = 8'b00000000;
                                blue4   = 8'b00000000;
                        end
        end

always @(posedge clk)
data_shift1 = {1'b0,datav[11:1]};
always @(posedge clk)
data_shift2 = {1'b0,data_shift1[11:1]};
```

Figure 61 Source level debug

Limitations

- The user cannot set breakpoints between two sequential hardware implementation of RTL statements. This is a natural consequence of a hardware implementation of RTL logic. It corresponds to the chip behavior, rather than sequential simulation model.
- The user is not able to set breakpoints in sub-programs (sequential procedures and functions)

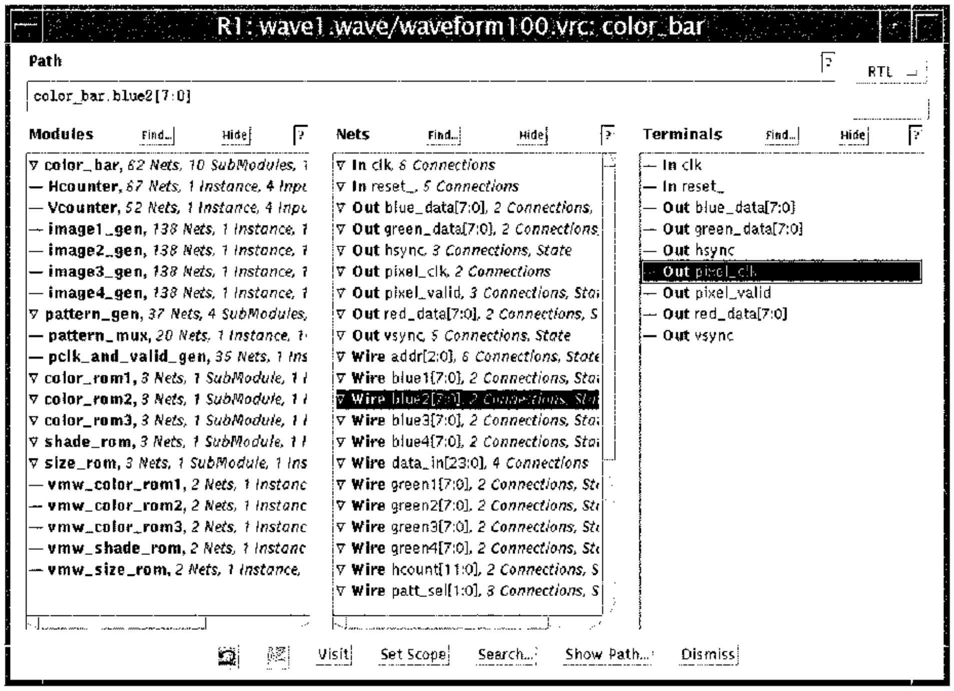
Graphical path browsing

The graphical path browsing displays logic cones, showing the user the design in RTL. It also helps the user understand the design vividly, by seeing the design in schematic view. Any signal can be dragged and dropped from the source debug window to the path browser or vice versa. All these tools are time-linked and can be used efficiently to traverse to various signals in different modules of the design.

Advantages of Path Browsing

- Draws partial RTL schematic of fan-in cone
  - Flattens hierarchy, or not
  - Stop at state elements, or any primitive
  - Backannotate signal values for current time
- Linked with source display and waveforms
  - Change time in waveform -> update signal annotation
  - Drag and Drop signals into waveform
  - Drag and Drop modules or signals into source display
  - Select module menu
  - Highlight source code associated with a schematic object
- Switch between gate-level and RTL views

Figure 62 on page 344 shows the Hierarchy Browser.


R1: wave1.wave/waveform100.vrc: color_bar

Path

```
color_bar.blue2[7:0]
```

Modules

```
▽ color_bar, 62 Nets, 10 SubModules, 1
— Hcounter, 67 Nets, 1 Instance, 4 Inpu
— Vcounter, 52 Nets, 1 Instance, 4 Inpu
— image1_gen, 138 Nets, 1 Instance, 1
— image2_gen, 138 Nets, 1 Instance, 1
— image3_gen, 138 Nets, 1 Instance, 1
— image4_gen, 138 Nets, 1 Instance, 1
▽ pattern_gen, 37 Nets, 4 SubModules,
— pattern_mux, 20 Nets, 1 Instance, 1
— pclk_and_valid_gen, 35 Nets, 1 Ins
▽ color_rom1, 3 Nets, 1 SubModule, 1 I
▽ color_rom2, 3 Nets, 1 SubModule, 1 I
▽ color_rom3, 3 Nets, 1 SubModule, 1 I
▽ shade_rom, 3 Nets, 1 SubModule, 1 I
▽ size_rom, 3 Nets, 1 SubModule, 1 Ins
— vmw_color_rom1, 2 Nets, 1 Instanc
— vmw_color_rom2, 2 Nets, 1 Instanc
— vmw_color_rom3, 2 Nets, 1 Instanc
— vmw_shade_rom, 2 Nets, 1 Instanc
— vmw_size_rom, 2 Nets, 1 Instance,
```

Nets

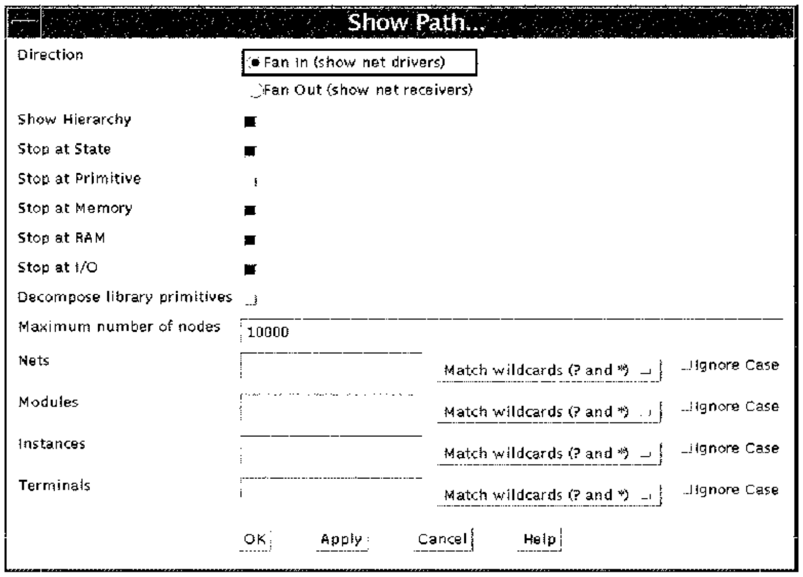
```
▽ In clk, 6 Connections
▽ In reset_, 5 Connections
▽ Out blue_data[7:0], 2 Connections,
▽ Out green_data[7:0], 2 Connections,
▽ Out hsync, 3 Connections, State
▽ Out pixel_clk, 2 Connections
▽ Out pixel_valid, 3 Connections, Sta
▽ Out red_data[7:0], 2 Connections, S
▽ Out vsync, 5 Connections, State
▽ Wire addr[2:0], 6 Connections, State
▽ Wire blue1[7:0], 2 Connections, Sta
▽ Wire blue2[7:0], 2 Connections, Sta
▽ Wire blue3[7:0], 2 Connections, Sta
▽ Wire blue4[7:0], 2 Connections, Sta
▽ Wire data_in[23:0], 4 Connections
▽ Wire green1[7:0], 2 Connections, St
▽ Wire green2[7:0], 2 Connections, St
▽ Wire green3[7:0], 2 Connections, St
▽ Wire green4[7:0], 2 Connections, St
▽ Wire hcount[11:0], 2 Connections, S
▽ Wire patt_sel[1:0], 3 Connections, S
```

Terminals

```
— In clk
— In reset_
— Out blue_data[7:0]
— Out green_data[7:0]
— Out hsync
— Out pixel_clk
— Out pixel_valid
— Out red_data[7:0]
— Out vsync
```

Figure 62 Hierarchy browser

After selecting the desired net and clicking the show path, the show path window pops up. The user now has the choice to select the fan-in or fan-out to view the driver or receiver information correspondingly. *Figure 63 on page 345* shows the Show Path dialog.



Direction	[✓ Fan In (show net drivers)] Fan Out (show net receivers)
Show Hierarchy	■
Stop at State	■
Stop at Primitive	
Stop at Memory	■
Stop at RAM	■
Stop at I/O	■
Decompose library primitives	
Maximum number of nodes	10000
Nets	Match wildcards (? and *) Ignore Case
Modules	Match wildcards (? and *) Ignore Case
Instances	Match wildcards (? and *) Ignore Case
Terminals	Match wildcards (? and *) Ignore Case

Figure 63 Show path

Figure 64 on page 346 shows the graphical path browser(RTL Topology).

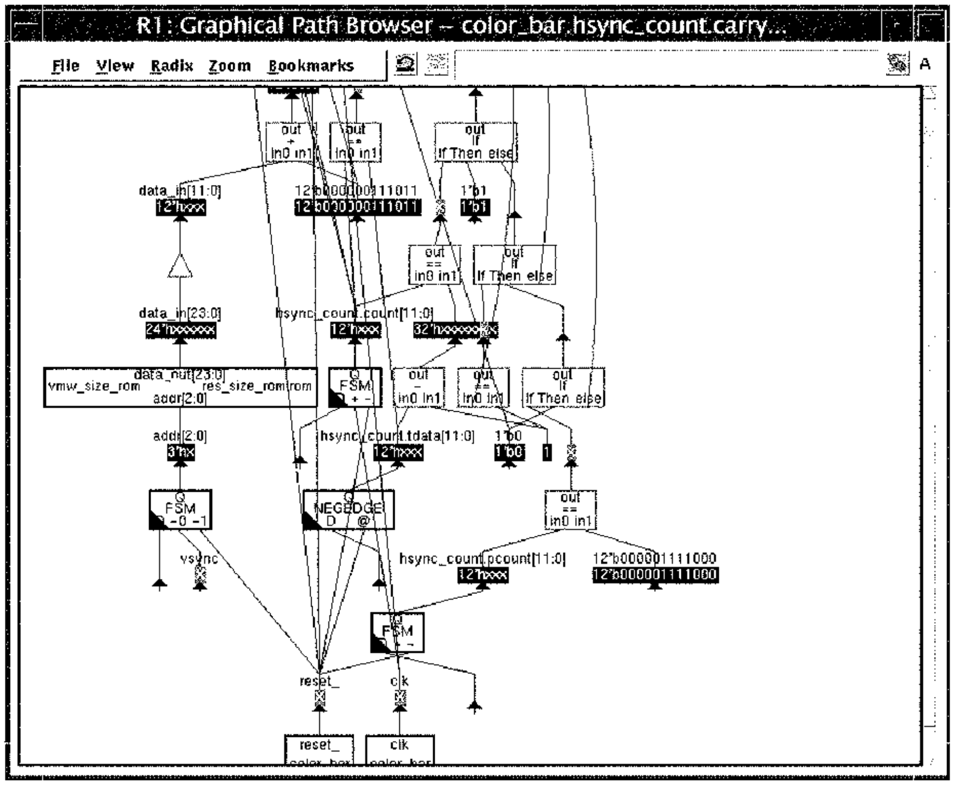
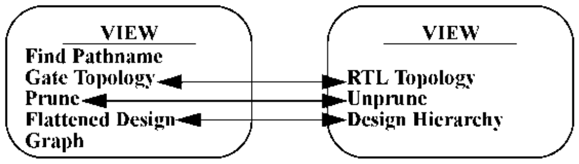


Figure 64 Graphical path browser(RTL Topology)

These are a few important graphical path browser tools under the VIEW menu



◄► represents toggle type

Figure 65 on page 347 shows the graphical path browser(Gate topology).

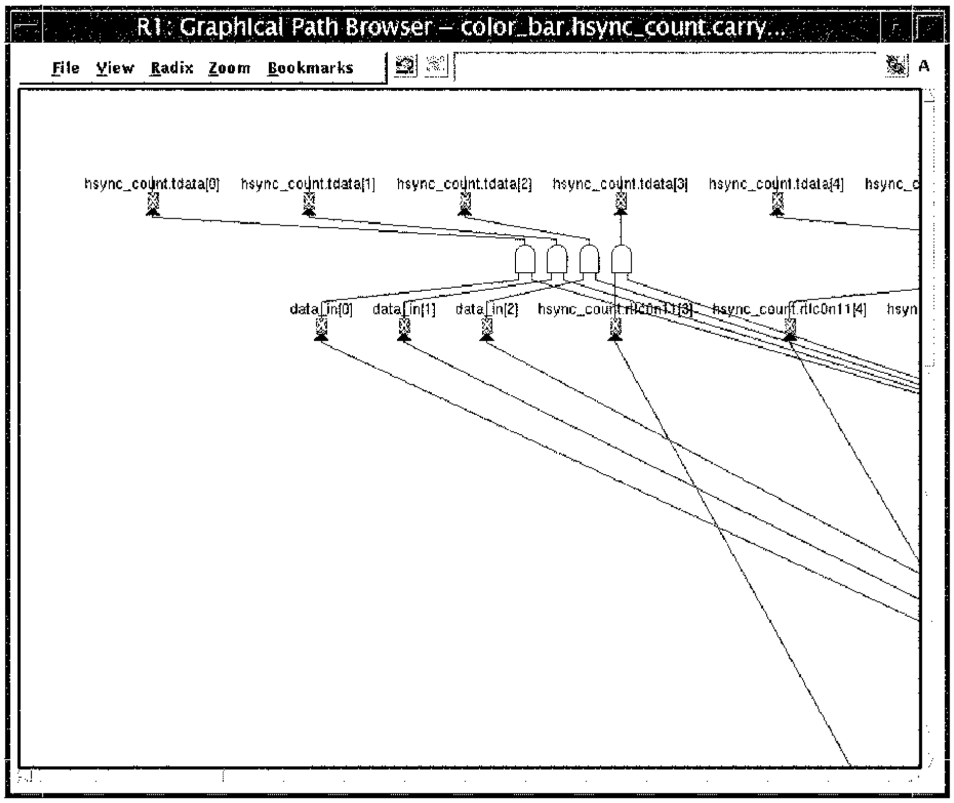


Figure 65 Graphical path browser(Gate Topology)

Find Pathname

The user enters a signal name and clicks OK to get the RTL or Gate topology of that signal. This helps the user to debug any desired signal without the use of going through the entire design. *Figure 66 on page 348* shows the Find Pathname window.

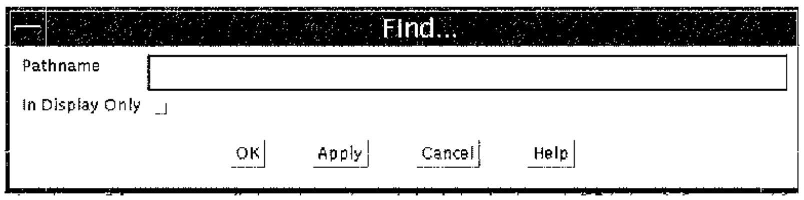


Figure 66 Find Pathname Window

Pruning

When the signal flow is driven by large CASE statements, the fan-in cone gets complex. This is because, all the circuitry that affects the signal are visible to the user. Only a small subset of the circuit affects the signal at a particular instant of time. Pruning makes the circuit a lot easier for the user to read and helps the user to focus on a single path.

Advantages of pruning

- Simplifies Diagram
- Makes the diagram more linear
- Narrows in on the cause
- Temporal Pruning focuses on subset of logic cone based on current conditions(at current time)
- What do we prune?
  - Case Statements including casex, casez
  - if/then/else
  - conditional expressions(?:)
  - Muxes
  - (N)AND, (N)OR
  - Tri-States

Figure 67 on page 349 shows the same net used in *Figure 64 on page 346* after pruning.

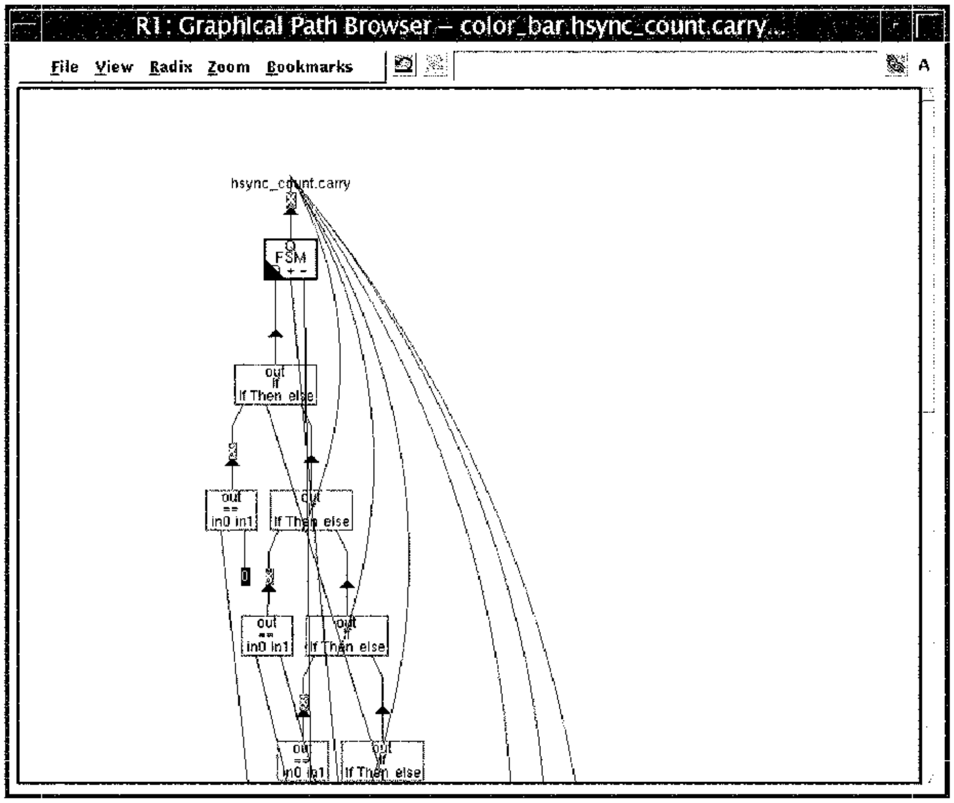
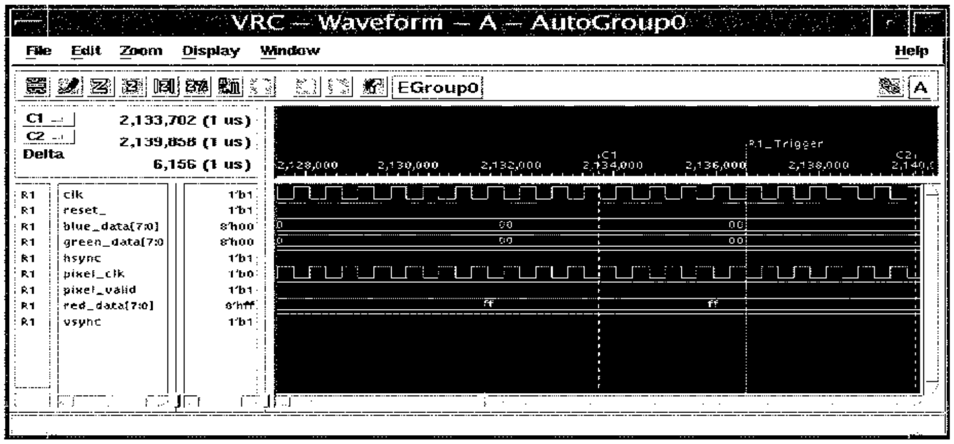


Figure 67 Pruning

Waveform viewer

The waveform window gives the change in the signals with respect to time. If the user wants to view the waveforms for more signals, the desired signals are to be dragged from the browser and dropped into the waveform viewer. The signals can be dragged from the waveform window to the source debug window to get the source code for that signal.

Figure 68 on page 350 shows the waveform viewer.



```
C1     2,133,702 (1 us)
C2     2,139,858 (1 us)
Delta      6,156 (1 us)

R1  clk                 1'b1
R1  reset_              1'b1
R1  blue_data[7:0]      8'h00
R1  green_data[7:0      8'h00
R1  hsync               1'b1
R1  pixel_clk           1'b0
R1  pixel_valid         1'b1
R1  red_data[7:0]       8'hff
R1  vsync               1'b1
```

Figure 68 Waveform viewer

Logic viewer

Logic viewer is another tool for debugging. This tool helps the user to get a logic implementation of the signal. A signal can be dragged from the waveform window or from the source debug window to the logic viewer to debug the design using logic implementation of the design. *Figure 69 on page 350* shows the Logic Viewer.

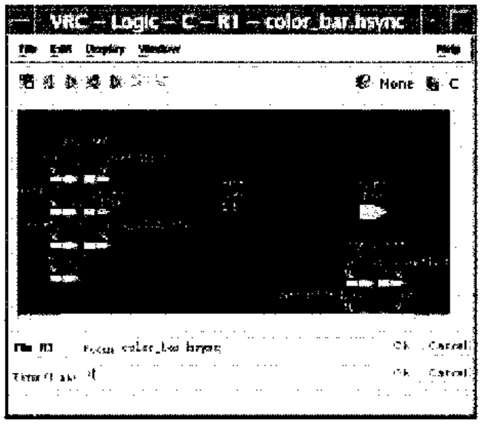


Figure 69 Logic viewer

ATI Ex. 2075
IPR2023-00922
Page 350 of 394

12 Trouble-shooting Guide

Overview

When things go wrong, quite often the situation is a standard one with standard solutions to the problem. This section presents a number of such common questions with answers to enable you to correct the situation.

The questions and answers are arranged in tables that contain the following segments:

- Statement of the problem
- Diagnosis of the problem
- Suggested solutions to the problem

The questions are associated together according to the operation during which they are likely to occur.

Software installation

These questions might arise during software installation:

Problem	Diagnosis	Solution	
		If...	Then...
The installation process cannot find *rsh*.	The VirtuaLogic Place and Route Manager uses *rsh* to start and manage tasks on multiple hosts.	Your site does not have *rsh*...	Install it. It is included in the Berkeley utilities package that should come with the operating system software.

Design import and compilation

These situations can arise when you are in the logical design compilation stage:

Problem	Diagnosis	Solution	
		If...	Then...
My design has undriven nets.	The VirtuaLogic compiler requires all nets in a design driven to some value.		• Using the qualification error window in the GUI, select a value for undriven nets singly or as a whole. OR • Use a tie-off file to give undriven nets values.

My design has multiply-driven nets.	The VirtuaLogic compiler cannot process a design which has nets with multiple drivers.	Identical modules have identical inputs for greater drive capability.	The compiler detects the situation and removes the redundant module(s).
		The modules or inputs are not identical...	• Using the qualification error window in the GUI, select one of the modules to drive the net. OR • Use a tie-off file to specify the single non-tristate driver of each net.
My design has combinatorial cycles.	The VirtuaLogic compiler processes most combinatorial cycles. This warning only alerts you that combinatorial cycles are present.		
My design has combinatorial cycles through memories.	The VirtuaLogic compiler cannot process combinatorial cycles through level-sensitive multiported memory elements.	Your design contains combinatorial cycles with level-sensitive memories...	Remodel the relevant memories, using an edge-sensitive model.
		You know read ports and write ports of the memory never use the same address...	You can convert a level-sensitive memory into an edge-sensitive memory sensitive to either edge direction.
		The write-enable signal uses clock gating so it is always disabled awhile in emulation cycles...	Convert the memory into an edge-sensitive memory.
	The VirtuaLogic compiler cannot process combinatorial cycles through level-sensitive multiported memory elements.	The write-enable signal does not use clock gating...	Introduce clock gating to create edges in write-enable in every cycle on which it is active. Convert the memory into an edge-sensitive memory.

		• The same address might appear, the memory shows read-through behavior on the read port. • Write data is stable at the beginning of the write.	Model the memory as an edge-sensitive memory sensitive to falling edges for a memory with active-low write enable or rising edges for active high.
My design has parametric test structures I do not want to emulate.	All libraries IKOS supplies are free of parametric structures.	You have built your own library	Modify the library to eliminate the parametric structures.
		Parametric or other unwanted logic is in your design and uses standard library elements	Mark any outputs to which these structures are connected as unsampled in the timing specification during *gvl*, and the VirtuaLogic compiler deletes the relevant logic.
My design contains a phase-locked loop.	The VirtuaLogic system cannot directly emulate phase-locked loops.		Use the timing specification interface for phase-locked loop modeling: 1.Identify the hierarchical name of the output net(s), from your phase-locked loop, which are the resulting clock(s). 2.For each clock net above, click the ADD Clock button to add an internal net clock specification. 3.Type the net name identified previously. 4.Use the clock waveform editing features to indicate the desired clock waveform.

	The VirtuaLogic system cannot directly emulate phase-locked loops. This process assumes your PLL or timing generator produces a periodic waveform.	Your waveform is not periodic but is a logical function of a periodic waveform and some data values derived from your circuitry.	You can model the behavior as follows: 1.Include the logic needed to compute the logical function in your PLL netlist. 2.Connect data value inputs as needed. 3.Connect a net to the logic for the periodic component. 4.To make this net an internal clock source, perform steps 1-4 to describe and create the needed periodic waveform. 5.Use the technique described previously to produce the periodic input component.
My memories do not appear when I browse on the memory form.	The compiler did not identify any of the design Verilog modules as potential memories. The compiler looks for Verilog modules that have no structural content to identify memory candidates.	You have memories in the design that Show Memories button does not list...	Write empty Verilog module definitions for the memories that specify the port names and directions. The UI needs this to assist you in preparing a memory specification.
My netlist requires Verilog compiler directive macro definitions.	The VirtuaLogic compiler fully supports Verilog compiler directive macro definition.		Specify needed definitions on the *Netlist Import* form in the box. .

My technology library is not mentioned on the *Technology Mapping* notecard in *gvl*.		The VirtuaLogic system does not support your technology library directly.	• Resynthesize your design directly to VirtuaLogic primitives. VirtuaLogic provides a Synopsys^{TM}.db library for that purpose. OR • Produce a custom library for your technology. A custom library is a set of Verilog module definitions for your technology primitives, implementing them as VirtuaLogic primitives. VirtuaLogic provides Synopsys libraries and scripts to make it easier for you.
There are warnings about unrecognized Verilog constructs when I compile my design.	The VirtuaLogic Verilog reader understands only Structural Verilog. You can safely ignore some behavioral constructs, but others confuse the compiler.	Your input file does not parse into VirtuaLogic correctly...	‘ifdef out the behavioral code with the following: **‘ifdef VIRTUALOGIC** **‘else** **behavioral code** **‘endif**
There are Verilog errors when I try to simulate the VSM.	There are several potential sources of Verilog compile errors which might arise while you are simulating the VSM.		Your testbench makes reference to internal nodes in your design in display or dump statements. Because the internal hierarchy of the VSM differs from your original hierarchy, you must eliminate these references or conditionalize with ‘ifdef.

The VSM doesn't simulate correctly in my testbench.	• Input Setup Problems	You have accepted default input maximum prop delay times in timing specification...	This has chosen a default simulation input sampling time, which might be inappropriate. Use the Verilog macro variable VMN-INPUT_SETUP to adjust the sampling delay to a value large enough to stabilize all inputs to the VSM.
Unexpected Xs appear in verify model simulations.	This may occur if the testbench has clocks which have X value for more than #1 after the start of the simulation. Look at outputs from the module with CtlFSM as its name, looking for Xs.		Modify the testbench clock generation code to produce clocks which are initialized to non-X within #1 of start of simulation.

Additional import and compilation problems

There are several common problems which might require diagnosis and resolution when you simulate a Virtualized Simulation Model (VSM).

Timescale issues

By default, the VSM specifies a timescale directive within the model:

```
`timescale 1ns/1ns
```

If this is incompatible with other timescale directives within your testbench, comment this timescale directive out of the model. It is the second line in the model.

Input timing issues

The timing specification defines when changes on emulator inputs can occur. The VSM watches for inputs which change outside of this region and produces warning messages when this occurs. These warning messages take the following form.

```
Design i/o <ioname>: unexpected transition seen at time <time>
```

The duration of the window in which input changes can occur is dependent on your design as well as the timescale setting you have chosen within your testbench.

The window in which changes can occur on a particular signal starts whenever a clock edge mentioned for the signal in the timing specification occurs and ends after the expiration of a period controlled by the Verilog symbol VMW_INPUT_SETUP, whose default setting is 100. Thus, by default, input signals can change during a window which is 100 Verilog time units in length (that is, equivalent in length to #100) and starts at each clock edge to which the input is timed.

If this default value is inconsistent with your testbench and timescale, you can override it. Provide a new value to the symbol VMW_INPUT_SETUP using the command line:

```
+define+VMW_INPUT_SETUP=<val>
```

In the above line, <val> is a numeric value.

Otherwise, you might have omitted from the timing specification an edge which causes input changes. To fix this, modify the timing specification.

Clock ordering issues

The timing specification defines an ordering for all clock edges within a clock domain. The VSM watches the order of clock edges in a domain and produces warning messages if this is inconsistent with the timing specification. These messages take the following form:

```
Out of order clock edge <clockname1-direction1> seen at time <time1>.
Preceding edge was <clockname2-direction2> at time <time2>.
```

This probably reflects an error in the timing specification.To fix it, adjust the timing specification to reflect the actual clock edge ordering.

Clock period issues

The VSM requires some amount of elapsed Verilog simulation time to process internal changes which occur as a result of each clock edge. This time is equal to the value of VMW_INPUT_SETUP plus a small design dependent component.

If clock edges occur too rapidly in succession, the VSM produces a warning of the following form:

```
Warning: design clock edges occurring too rapidly - extend.
```

Change the testbench and/or timescale granularity to provide more simulation time granularity between consecutive clock edges.

Design compilation

You could face these situations when you are compiling the physical design:

Problem	Diagnosis	Solution
I have more clocks than fit on the clock cable.	The VirtuaLogic emulation supports a maximum of 14 clock signals. This includes • All periodic clocks • Any asynchronous system inputs which trigger state changes: – Asynchronous preset or – Clear signals driving terminals but excluding asynchronous inputs which the data inputs of state elements sample	

I want to let the emulator decide the target system pinout.		Do not specify a terminal-constraint file (disable it.)
One or more FPGAs do not compile		1. At the command line, create a new file comprising the name of each non-compiled FPGA on a separate line. 2. Return to the Compile form, and in the **Options** window enter the following: **-FPi** filename **-Pi** <partition input filename **-Pfi** <place input filename> [**-Po** <partition output filename>] [**-Pfo** <place output filename> where **-FPi** further partitions the contents of the designated file. <filename> lists on separate lines the names of all the noncompiling FPGAs **-Pi** reads what is in the file as the partition result of the previous run. <partition input> contains the partition result of the previous run filename **-Pfi** reads in the result of placement from the previous run as output of the placer. place input contains the placement from the previous run filename **-Po** outputs the result of partition into the specified file. partition contains the result of partition output filename **-Pfo** outputs the result of placement to the designated file. place output contains the result of place filename 3. Recompile.

Partitioning

You could face these issues when partitioning:

Problem	Diagnosis	Solution	
		If...	Then...

My FPGA costs are too high or my FPGA compiles fail because the FPGAs are too full	-Mm 9 targets 90% of normal, or a cost of 4500 per FPGA.	A design has FPGA compile problems, this is a bug.	1.Send the problem to IKOS. 1.Reduce -Mm by 1 and recompile. 2.Reduce again to -8 or 7 if necessary.
My design does not fit into N boards.	If you are trying to put more than 170K / 340K gates per board into the system, it may not work.	1.Check the reported gate count from the computer. It looks like Number of VMW primitive modules: 236723 Number of VMW primitive gateS: 408213 Number of memory Interface gates: 105000 The correct number of boards is (Number of VMW primitive gates + Number of Memory Interface gates)/170000 or 340000 2.If you are in the 170K / 340K gate range and the design does not partition into the right number of FPGAs, call IKOS.	
My design is too slow.		Use **-dump e** to see if critical path includes false paths.	
Design that fails with a *"subprocess 11"* at customer site but completes compilation in the corporate	The problem is that the dynamic loader puts the libraries at approximately 0xf000000 - stacksize_limit. This stacksize_limit is found out by typing limit in the unix prompt. If the stacksize_limit is >768MB, this will bring the libraries down into an unexpected address range	To fix this, set the stacksize_limit to a reasonable value. example 8192**KB**.	

Configuration download

You could encounter these problems while downloading the configuration:

Problem	Diagnosis	Solution	
		If...	Then...
Cannot connect to the emulator.	• Only one process can connect to the emulator at a time. The emulator is connected as a SCSI device to a particular workstation. Processes communicating with the emulator must run on this workstation. Therefore, establish that you are using the emulator workstation.	You fail to connect due to another connected process... – You are running *vrun* (the textual runtime monitor)... – You are running *gvl*...	The system reports the failure. This must run directly on the emulator workstation. This can run anywhere, but you must provide the hostname of the emulator workstation. Verify that you have specified this hostname correctly.
Configuratio n download reports a failure			Call Tech Support.
Diagnostics fail.			Call Tech Support.
Cannot connect to the logic analyzer.			See "FIne Tuning the Ethernet" in the *Installation Guide*.
The NFS mounting for the logic analyzer partitions is impossible.			Use the FTP mode. 1.Modify params.mak. Add the following line: *VPROBE OPTIONS -ftp*. 2.*vprobe* uses ftp to access the control partition.This method is much slower than the NFS path.

Emulation

You could encounter this situation while emulating:

Problem	**Diagnosis**	**Solution**
The error light on the emulator goes on.	There is an out-of-sync user clock signal.	Adjust the user clock signal. Click the Emulator I/O PODS Enabled button to clear the light and reset the emulator.

Solaris 2.6

The user must be aware that Solaris 2.6 is not a supported operating system for VLE. Solaris 2.5.1 and 2.7 are supported for VLE. If one is specific about using Solaris 2.6, one has to give importance to this section. Most of the software doesn't get profoundly affected by the OS revision., except for Vrun. Vrun is very OS-sensitive for two reasons:

- It requires a revision-specific OS driver to talk to the emulator box
- It uses multi-threading to accelerate state replay

Vrun also runs on the machine physically connected to the emulator.

If an user suffers from Vrun going into an infinite loop while doing state-replay for 100% visibility, this problem can be solved by setting a special environment variable to turn off the usage of multi-threading using the following command.

```
setenv VMW_NO_THREADS 1
```

This command is placed inside the user's .cshrc file and sourced even before the gvl is invoked. This covers the case where gvl and vrun are running on the same machine or not.

Debug activity

You might encounter these difficulties while debugging:

Problem	**Diagnosis**	**Solution**
The trigger interface says my trigger cannot be created.	Because of the multiplexed nature of theVirtuaLogic probe mechanism, certain probe groupings might have some trigger conditions that cannot be implemented.	Consider whether a clock qualification can be added to the term. From the UI, have the VirtuaLogic compiler create a probe file with a grouping for which the trigger can be implemented. OR Try an alternative trigger, using the current probe grouping.
I need to add more probes.	The VirtuaLogic compiler system supports incremental probe additions.	Open gvl. Add new probes. On the *Compiler* page click *Incremental Probe Compile*.
The incremental probe compile reports that a probed FPGA has become too large.		Call IKOS customer support.

Virtual swapping

This feature in *vrun* (the program manager, embedded in *gvl*, running the emulator) relocates (virtually swaps) downloaded bits to determine if a problem is design related or hardware related. It is not designed to locate a bad board.

Diagnosing the problem

If you relocate bits, yet problem symptoms are completely unchanged, it is likely a design-related problem. Any test behavior change after swapping indicates bad hardware, unless the *vclock* speed is marginal, something you should rule out in advance. If virtual swapping leads you to believe you have bad hardware, you should walk a good board through every slot to isolate the bad board, unless diagnostics catch the problem.

Command syntax

virtual-swap command

After you connect the hardware (using the connect *command*) and configure it, the command *virtual-swap* is available from the *vrun* prompt and remains valid until exit.

To execute:

1. At the UNIX prompt type:

   configure <design_name> -probe <probe_window>/system0

   In the above line, *<design_name>* is the root module, and *<probe_window>* is the directory with the desired probe window. For example:

   ```
   probe_window_0.pbw/system0
   ```

2. Then type:

   virtual_swap **[<arguments>]**

   This command does not validate or invalidate any other command even though under certain swapping configurations it would be impossible to do in-circuit testing. Make sure you do a virtual swap to place the hardware in a valid state before running in-circuit.

Arguments for the *virtual_swap* command are:

[B <n>]	rotates the boards through [n mod m] slots where [m] is the number of boards in the system. Positive [n] rotates up; negative [n] rotates down. Not available on VirtuaLogic 2.1.

[R]	swaps the FPGA rows on all boards so that FPGA [BRC]is swapped with B<(R+4)%8>C. A row swap exchanges q0(quadrant 0) and q1 with q2 and q3, respectively.
[C]	swaps the FPGA columns on all boards so that FPGA [BRC] is swapped with BR<(C+4)%8>. A column swap exchanges q0 and q2 with q1 and q3, respectively.
`[N]`	means no swap. This brings the system back to its original configuration, that is, pre-swap configuration.

You cannot combine row, column, and board swapping. Each new swapping clears the previous swapping.

Types of swap

Column swap

You can always swap columns. If you use this test while running in-circuit, you must move the target data cables from Jn to J(n+3)%6 on a board.

Example

You must move a cable attached to J1 on board 3 to J4 on board 3 and a cable attached to J5 on board 2 to J2 on board 2.

Row swap

You can always swap rows. It is, however, impossible to run in-circuit with the rows swapped, and you are restricted to hardware functional testing only.

Board swap

Board swap only works on VirtuaLogic 2.0, not on VirtuLogic 2.1.

Noncompiling FPGAs

Correcting noncompiling FPGAs with -FPi switch

If one or more of the FPGAs does not compile because of an oversize FPGA (i.e., total cost greater than 13500), a message appears in the *Place and Route Log* pane which lists the failures. To correct the situation, take the following steps:

1. At the command line, create a new file comprising the name of each noncompiled FPGA on a separate line.

2. Return to the *Compile* form and in the *Options* window enter the following:

```
-FPi <filename> -Pi <partition_input_filename> -Pfi  <place_input_filename> [-Po
<partition_output_filename>]
[-Pfo <place_output_filename>]
```

ARGUMENTS:

-FPi <filename>	This further partitions the contents of the designated file. It lists on separate lines the names of all the noncompiling FPGAs.
-Pi <partition_input_filename>	This reads what is in the file as the partition results of the previous run. It contains the partition results of the previous run.
-Pfi <place_input_filename>	This reads in the result of placement from the previous run as output of the placer. It contains the placement from the previous run.
-Po <partition_output_filename>	This outputs the results of the partition into the specified file. It contains the result of partition.
-Pfo <place_output_filename>	This outputs the results of the placement to the designated file. It contains the results of place.

3. Recompile.

Refer to page 271 for additional information.

Correcting fitting problems with -CUi switch

On rare occasions, a design exhibits a structure leading to systematic place and route failures that are insensitive to the *-Mm* cost adjustment parameter (refer to page 276 for details). If an FPGA fails to compile and the components on the FPGA are mostly from the same part of the design, the user can raise the weight of that part using the compiler option -CUi *filename* as follows:

- At the command line, create a new file. The file contains one or more lines of the following form:

  ```
  Module <module_expression> [weight]
  ```

  where *<module expression>* is a regular expression matching the hierarchical name of one or more module instantiations within the design. Any module listed in the file has its modeled cost increased. The default increase amount is one. The optional [*weight*] field specifies the increase amount for all of the modules that match the *module expression*.

  The typical use of the file has the following form:

  ```
  Module a.b.c.********
  ```

  which matches all instantiations within the scope *a.b.c.* (The number of *s should exceed the internal hierarchical depth of the scope *a.b.c.*)

  All the modules under the hierarchy *a.b.c.* get a higher cost than they normally would during design compilation.

- Return to the *Compile* form and in the *Compiler Options* window enter the following:

  ```
  -CUi <filename>
  ```

- If the default increase amount needs to be changed, the *-CUc* option provides this feature. Optionally enter the following:

  ```
  -CUc <weight>
  ```

  where *<weight>* is the new default increase amount for the -*CUi* file.

- Recompile the design.

Refer to -*CUi on page 278* for additional information.

IKOS

13 PC Farm

Overview

The second stage of compile is FPGA place and route. With VirtuaLogic 2.1 or greater, the user can farm out FPGA compilation to PCs, UNIX workstations, or a combination of multiple platforms. By adding more systems to the solution, the user can dramatically speed-up design turnaround times which can significantly improve verification productivity. Utilizing 60 platforms allows the user to do the complete compile and FPGA place and route of a million gates in less than two hours.

Hardware requirements

PC:

- Intel 815 MB w/integrated e-net and video
- Intel PIII 850MHz CPU
- 512MB PC133 SDRAM
- IBM 5GB hard disk drive

Workstation:

- 4.0 with service pack 5
- Denicome Software version RSHD/NT 2.18.03 WITH PATCH installed

Software requirements

- Windows 95 or NT 4.0

- VirtuaLogic 3.3 or greater
- RSH daemon (refer to *Obtaining a RSH daemon on page 372* for details)
- Optional remote control software, for example:
  - PC-Anywhere (commercial program)
  - VNC from AT & T Laboratories Cambridge (refer to *Obtaining VNC software on page 373* for details)

PC setup

The user must setup TCP/IP networking on the PCs. The Xilinx licensing is time-bound and requires the correct date, time, and time zone. Ensure the date, time, and TCP/IP connectivity of each PC is correct before moving forward with each PC. Verify the PCs can be pinged from the UNIX workstation. IKOS recommends going through the whole process on one PC, including running a PAR job, before configuring the other PCs.

Software installation

Obtaining a RSH daemon

The user must purchase a RSH daemon from Denicomp Systems. The RSH daemon communicates with a Remote Shell Daemon (rshd). The user can download a time-bombed evaluation copy but it requires a reinstall of the registered version once it is received. Be sure to order the daemon service, not the RSH client.

The web site to order is as follows:

www.denicomp.com

The RSH daemon can also be ordered from the Public Software Library at 1-800-2424-PSL. Order RSHD/95 with product #14496 or RSHD/NT with product #14075.

A single copy, bulk discounts, or a site license is available.

RSH daemon

- Download the RSH software (refer to *Obtaining a RSH daemon on page 372* for ordering information)
- Unzip the software into an empty directory (for example, c:\install)
  - When installing the RSH daemon, accept all the installation defaults
- run the following:

```
c:\install\setup
```

- Ensure that "Windows Service" is checked in the dialog box
- Remove the installation directory (i.e., c:\install)

Test the RSH daemon

Test the PC RSH daemon from the UNIX workstation as follows:

```
rsh PCHOSTNAME "<[CON]>" dir
```

The output display should appear similar to what follows:

```
Volume in drive C has no label.

Volume Serial Number is 34EF-1473

Directory of C:\WRSHDNT

02/26/98  06:07p      <DIR>          .
02/26/98  06:07p      <DIR>          ..
    02/25/98  10:06p          188,416 WRSHDNT.EXE
    02/25/98  10:20p           52,224 WRSHDRUN.EXE
    08/27/96  09:37p           34,816 WRSHDRDR.EXE
    06/02/96  08:39p          141,824 WRSHDCTL.EXE
    08/05/96  07:39p           48,128 CTRLRSHD.EXE
    03/08/96  11:24p            3,423 LICENSE.TXT
    01/26/97  01:09p           80,218 WRSHDNT.TXT
    08/05/96  09:25p            9,517 ORDER.TXT
    02/25/98  10:18p              894 PACKING.LST
    03/21/97  11:44p            4,457 README.1ST
```

```
01/26/97  12:41p              834 SUPPORT.TXT
08/26/97  08:05p           23,552 WHOAMI.EXE
14 File(s)       588,303 bytes
1,246,101,504 bytes free
```

Obtaining a RSH daemon

The user must purchase a RSH daemon from Denicomp Systems. The RSH daemon communicates with a Remote Shell Daemon (rshd). The user can download a time-bombed evaluation copy but it requires a reinstall of the registered version once it is received. Be sure to order the daemon service, not the RSH client.

The web site to order is:

```
WWW.DENICOMP.COM
```

The RSH daemon can also be ordered from the Public Software Library at 1-800-2424-PSL. Order RSHD/95 with product #14496 or RSHD/NT with product #14075.

A single copy, bulk discounts, or a site license is available.

VMW/Xilinx software

Obtain write permission to the *VMW_HOME* directory and do the following:

```
setenv VMW_HOME <appropriate_path_here>
cd $VMW_HOME/bin
./pc_setup.csh <space_separated_pc_hostnames>
```

All the PC hostnames can be given on the command line. This will download the Xilinix software to each PC and also some IKOS software that runs on the PCs to control the compiles. This download takes a few minutes for each PC, depending on the machines and network. Following is an example:

```
$VMW_HOME/bin/pc_setup.csh pc1 pc2 pc3 ...
```

This also prepares the following file:

```
$VMW_HOME/env/pc_hosts.mach
```

This file is used by *gvl* to enumerate the PCs setup on the network. Therefore, when running *SVMW HOME bin pc setup.csh*, do it from an account that has write access to *SVMW HOME env*. Otherwise, it will setup that file in *tmp* and it will have to be copied to *SVMW HOME env* after securing write access.

IKOS recommends doing just one PC first and trying a simple PAR job using *vtask* before investing the time to do all the PCs.

VNC (optional)

Obtain VNC software from AT & T Laboratories Cambridge, as detailed below.

- Unzip the software into an empty directory (for example, c:\install)
- Run and follow the screen instructions:

  c:\install\setup
- Remove the following:

  c:\install

Obtaining VNC software

The Virtual Network Computing (VNC) is a remote display system which allows the user to view a computing desktop environment not only on the machine it is running on, but from anywhere on the Internet.

VNC is free from AT & T Laboratories Cambridge web site as follows:

http://www.uk.research.att.com/vnc

Farm usage

From the command line

- Change to the appropriate *.vmw* directory
- Create a *machlist.mach* file which includes the names of the PC Farm machines followed by *pc* extension and UNIX workstations without the */pc* extension. Following are examples:

pcfarm1/pc
pcfarm2/pc
unixfarm1
unixfarm2

- Run the following:

  vlc . vtask

From the GUI

$VMW_HOME/bin/gvl

Note that on the FPGA compile page, the host-list pane will include the PCs in the *$VMW_HOME/env/pc_hosts.mach* file. When the user drags and drops them into the machine list, they will already have the */pc* appended.

From this point on, FPGA compile software is operated the same as in prior releases.

Maintenance scripts

Contact IKOS Field Engineering for additional maintenance scripts for remote cleaning, rebooting, and status checking.

IKOS

14 Glossary

ASIC	Acronym for Application Specific Integrated Circuit.
asynchronous signals	Signals whose transitions have no known frequency or phase relationship to one another.
clock domain	A collection of phase-locked clock signals and logic whose state and signal transitions are phase-locked to the clock signals.
clock signal	A signal that triggers a state transition on a circuit state element.
configuration directory	The directory in which you keep the files which the VirtuaLogic compiler requires and produces for compilation.
data signal	A signal that does not trigger a state transition for any circuit state element.
design I/Os	Top-level I/Os of your design. Design I/Os are signals that connect your design to the target system.
edge-sensitive	Describes a state element that changes state only when a transition occurs on a state-controlling clock signal.
emulator terminal	A cabling point on the emulator for connecting your design to the target system. Each design I/O must be internally connected to an emulator terminal by way of the pin constraint file and then externally cabled to the appropriate place on the target system.
false path	A combinational path in the user's design that never propagates information or has multiple clock cycles in which to propagate itself.
feedthrough	Tells the compiler that an output signal leaves the emulated design and combinatorially causes an input to the emulated design to change.
level-sensitive	Describes a state element which can change state whenever a state controlling clock signal has a particular value.

netlist defines	The set of Verilog preprocessor macro definitions required for your design.
phase-locked signals	Signals whose transitions have a fixed, known frequency and phase relationship to one another.
probe	A signal within a design which is made visible through the VirtualProbe facility.
race condition	A pair of simultaneous signal transitions whose precise ordering affects system results.
regular expression	A character string that contains wild card characters, allowing the string to match many different strings. Several places in the VirtuaLogic GUI recognize regular expressions involving `*` and `?` characters. The `*` character matches `0` or more characters and the `?` matches exactly one character.
root module	The top-level module in your design.
scalar net	A net consisting of a single bit wire.
synthetic vector	A collection of related scalar net names bundled together for convenient handling as a group. The VirtuaLogic graphical user interface automatically produces synthetic vectors from appropriately named scalars. This occurs when a collection of nets exists with names whose only difference is some embedded numeric value. Non-numeric characters can precede the numeric characters which index the synthetic vector. Example:

	Synthetic Vector Syntax	
	Net Names	**Represented as.**
	name1, name2, ... name8	name<1:8>
	name_1_, name_2_, name_3_...	name_<1:8>_

	The left and right brackets denote a scalar vector, and the indices indicate the range of numbers present in the scalar signals. Manipulation of a synthetic vector – for example, dropping it into a selection window – has the same effect as individually manipulating each scalar component. To expand synthetic vectors into the collection of scalars from which they originated, double click on the synthetic vector.
target system	The hardware fixture into which you plug the emulator to provide stimulus. The target system is typically a modified version of the printed circuit board (PCB) which will house your finished ASIC.

terminal constraint file	A file that specifies the emulator terminal to which to connect your design I/Os. It lets you control the binding of design I/Os to emulator terminals so you can match design I/O pin out on the emulator to the pin out of your target system. Creating a terminal constraint file occurs in the emulation process between specifying the emulator configuration and creating core probes.
tie-off	A signal within a design which is forced to a constant value of 0 or 1 through the signal tie-off facility.
trigger	An expression or sequence of expressions used by a IDS to control the capture of analytic data.
vector net	A net consisting of multiple wires with a single name which can be indexed to give names for individual wires or ranges of wires from the net.
ICE	In-Circuit Emulation
DUT	Design Under Test
VLE	VirtuaLogic Emulator, the emulation hardware
VirtuaLogic	The VLE software, including the VSYN compiler, the GVL user interface, the probing and triggering subsystem VPROBE, etc.
TIP	Transition Interface Portal, a hardware and software interface to allow co-modeling on all VLE platforms
RTLC, RTLC-VLE	The RTL compiler targeted for the VLE platform
rtlc-accel	The RTL compiler targeted for the NSIM platform
GVL	VirtuaLogic GUI
VSYN	VirtuaLogic's emulation compiler
rtlc-vle	core compiler for VLE
rtlc-driver	wrapper over the core compiler for VLE
API	Application programming Interface
TAPI	TIP API
Co-Modeling	System level testing possible by providing high-speed communication between abstract system models and the DUT

Appendix A

Logic analyzer setup

The Logic Analyzer is an Hewlett Packard model 16500 B or C, using cards 16556 models A or D. The Logic Analyzer can have between one and four 16556 cards. With any number of cards, the Logic Analyzer can support a full signal window multiplexed from the System Board.

The Logic Analyzer must be cabled to the System Board. The following table is the view from the back of the HP Logic Analyzer. The J numbers reference the cable connection to the system board of the emulator. The cables are the time domain multiplexed internal nets from the emulator that will be monitored by the Logic Analyzer.

Card C is the master card in the Logic Analyzer. Note that the Logic Analyzer cards are not in order. The Logic Analyzer cards order is C,B,D,A. If the Logic Analyzer is not fully populated with sampling cards, then should be installed in this order. If the Logic Analyzer is not populated with all 4 cards, connect only the cables for the cards that are installed. The Hardware Installation guide provides directions for running Logic Analyzer diagnostics.

Logic Analyzer cables

Card Slot	Pod 1/2	Pod 3/4
A	J7	J8
B	J3	J4
C	J1	J2
D	J5	J6

Sampling data

Each probed signal will be sampled when necessary. After the clock toggles, the logic is time domain multiplexed and after the time slice occurs for the signal, then it will be sampled. The resulting waveforms will display the clocks in the design and all the probes for the design will be sampled within one of the time slices in the domain. If the internal node can change based on either edge of the design, then the compiler will actually probe the specific internal node twice so that it is sampled in each epoch.

The display will appear as if all the signals change with the clock. The actual timing implementation of the design is not indicated in the Logic Analyzer display. This is often referred to as state mode for the Logic Analyzer. If timing based data is important for signals in the design, then the Probe Group needs to be specified as un-multiplexed in the Triggers Tab.

Store

The store function can be assigned to each state of the trigger state machine. If store is selected, whenever the Logic Analyzer is in the given state, it will sample and display data based on the time domain multiplexing. When using nostore and store in a Logic Analyzer trigger, sometimes the results appear confusing because the data appears continuous in time in the waveform, but is actually sampled from only the states which store.

If the store function is not used in every state, it is possible that once the trigger has occurred, the Logic Analyzer Maximum Sample Depth will never fill up completely. If this is the case, then using the Record/Stop button on the Emulation Tab will cause the Logic Analyzer to stop sampling data.

Capture data

The user defines how data is captured relative to the trigger. Data can be captured before the trigger, after the trigger, or centered around the trigger. If data is captured after the trigger, a small percentage of cycles are captured before the trigger so the actual trigger condition can be viewed.

Maximum sample depth

The Logic Analyzer sample depth is selected by the user on the Emulation Tab. It can be set to any power of two between 4K and 2M samples. The maximum sample size is dependent on the number of Logic Analyzer capture cards. The sample size set in the GUI is the number of samples that the HP Logic Analyzer collects using the time domain multiplexing. When the vectors are updated, fewer vectors will be viewed due to the time domain multiplexing. The actual number of vectors is less than the maximum Sample Depth.

To determine the ratio of user clocks to Logic Analyzer clocks, set the Logic Analyzer to 4K samples and create a basic trigger which will occur. Let the Logic Analyzer memory fill up and upload the vectors. Determine the quantity of samples that were collected. The number of samples can change with each compile of the design, based on the time-domain multiplexing. With an understanding of the number of user clocks captured in a 4K sample, the correct Sample Depth can be estimated for future captures.

The time required to process the waveform and upload it to the screen is linearly related to the amount of data collected. In order to expedite the uploading, use smaller sample depths. If debugging a complex problem, larger sample depths can be used to capture a longer period of time. After the Sample Depth has been selected, the HP Logic Analyzer can be connected. Changing the Sample Depth after connecting to the Logic Analyzer will not take effect until the Logic Analyzer has been disconnected and reconnected.

Downloading and running the logic analyzer

Connect to logic analyzer

Information in the Logic Analyzer Setup pane on the Emulation Tab cannot be changed when the Logic Analyzer is connected. Prior to connecting to the Logic Analyzer, check that the settings are correct for the Logic Analyzer. Verify the following parameters.

- LA Probe Cards
- Sample Depth
- Signal Window
- Logic Analyzer host

The number of Logic Analyzer cards is determined before compiling. After the compile is completed, it cannot be changed. The Sample Depth, signal window and Logic Analyzer host can be changed, but only before connecting to the Logic Analyzer.

Before connecting to the Logic Analyzer, the environment must be set up properly. The Logic Analyzer is an Ethernet agent. In order to use the Logic Analyzer, it must mount the disk on the HP Logic Analyzer onto the local workstation. The mountla executable that is run must be owned by root in order to complete the mount. The mount point is config-name.vmw/hpla/control and config-name.vmw/hpla/data. If someone else has the Logic

Analyzer mounted, then it will not allow another user to mount it. If the VirtuaLogic GUI is exited without disconnecting from the Logic Analyzer manually, then the mount point remains. However, using "ftp" to connect to the logic analyzer is suggested to ensure that the system files on the logic analyzer are not deleted by mistake. The potential of this happening is high when the user does not exit from logic analyzer properly.

To use ftp option, please make sure that the following command is in params.mak file

```
VPROBE_OPTIONS = -ftp
```

In order to connect to the Logic Analyzer, the following conditions must be met:

- The Logic Analyzer must respond to the UNIX ping command. If not, check setup, hostid, and reboot the Logic Analyzer.
- The user must be able to ftp to the Logic Analyzer. If not, try ping and reboot the Logic Analyzer.

```
Using ftp mode
/hq/support/release/VirtuaLogic_v3.0.9/bin/vprobe -GUI \
        -tui vmw.trigger \
        \
        -vrcdump -hierout recon_data.map -dataout recon_data.vrc \
        -probe probe_window_0.pbw/SEE_probe_io.map \
        -nc 1 \
        -depth 0 \
        -host trainla \
        -ftp ;

Checking out license...done
Connecting to logic analyzer...
FTP: Connected to trainla.
FTP: 220 HP16500C V01.00 FUSION FTP server (Version 3.3) ready.
Starting vprobe
(vprobe)
```

Run the logic analyzer

To run the Logic Analyzer and trigger, click on the buttons

- Download Trigger

  The trigger that is defined in the Triggers Tab will be converted to HP Logic Analyzer Format and downloaded to the Logic Analyzer. Although the information is not meaningful to the user, it is a confirmation that the Logic Analyzer was programmed.

When the trigger is received by the HP Logic Analyzer, it indicates that it is waiting for a trigger with a green bar on the screen. In the gvl GUI, a message will also indicate that the Logic Analyzer is waiting on a trigger:

```
Processing trigger file: reset.trigger
Begin generating Logic Analyzer expression tree...
End generating Logic Analyzer expression tree: 100% complete
Downloading Logic Analyzer trigger file: hptrigger.txt
Logic Analyzer download successful.
Commands: exit lock quit stop unlock
Initiating Logic Analyzer status poll...
```

- Record/Stop

  This is a toggle button. Record polls for the trigger and captures data based on the trigger state machine definition. Stop causes the Logic Analyzer to stop polling or to stop waiting for more data to fill up the memory. If the trigger did not occur and the user wants to see what is occurring at the time the stop button is pressed, then the waveforms can be uploaded.

  After a trigger is downloaded, the Logic Analyzer automatically goes into Record mode. The Logic Analyzer screen will indicate this by converting the Run button to a Stop button. Once the Logic Analyzer has triggered or been stopped, it will collect the data and generate a message to the user:

```
The Logic Analyzer has stopped gathering data
Logic Analyzer status poll ended.
LA status received.
Commands: download exit lock quit run unlock upload
```

- Upload Waveform Data

  This takes the Logic Analyzer sample data and transfers it to the workstation. On the workstation, the data is reformatted for the time domain multiplexing and a file is created containing the display data in vrc format. The waveform is created with a unique name which the GUI will request from the user. When the file has been transferred from the Logic Analyzer to the workstation and reformatted, the user will see a message:

```
Uploading data.raw file from Logic Analyzer
FTP: 150 Opening data connection for data.raw
  (149.172.201.173,39169) (100352 bytes).
FTP: 226 Transfer complete.
The data file from the analyzer, trainla, was successfully
  transferred.
Loading VRC hierarchy
Processing the data file...
HP Logic Analyzer recorded 8192 samples of data
Waveform Processing complete
```

```
Logic Analyzer upload successful.
```

- Logic Analyzer Window

  The Logic Analyzer Window button brings up and XWindow display on the workstation which shows the actual HP Logic Analyzer screen. Since the Logic Analyzer has time domain multiplexed information, it is not useful to the user, other than messages that are displayed that indicate when it is waiting on a trigger.

Index

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z