Vivado Design Suite User Guide

Synthesis

UG901 (v2016.1) April 6, 2016

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document:

Date	Version	Revision History
04/06/2016	2016.1	Clarified Creating Run Strategies in Chapter 1. Added Creating a New Strategy in Chapter 1. Changed control_set_opt_threshold in Using Synthesis in Chapter 1 Added -retiming to Using Synthesis in Chapter 1 Added -max_uram, -max_bram_cascade_height, max_uram_cascade_height, -assert and -no_srlextract to Using Synthesis, Running Synthesis with Tcl, and Vivado Preconfigured Strategies in Chapter 1 Modified Multi-Threading in RTL Synthesis in Chapter 1 Changed Project Settings Dialog Box in Chapter 1 diagram to match the release. Changed Removed "s" from Register option. Added an additional option for Creating Run Strategies in Chapter 1. Replaced the following figures in Chapter 1 to match the release: Figure 1-5,Figure 1-6, Figure 1-11, Figure 1-12. Updated KEEP_HIERARCHY Example (VHDL) in Chapter 2 Modified ASYNC_REG in Chapter 2. Modified DONT_TOUCH in Chapter 2 to show only yes. Modified the recommendation for MARK_DEBUG in Chapter 2. Added EXTRACT_ENABLE in Chapter 2 EXTRACT_RESET in Chapter 2. Changed "enable" to "reset" in EXTRACT_RESET in Chapter 2 Added to Custom Attribute Support in Vivado in Chapter 2 In RAM_STYLE in Chapter 2: removed extra word, "accepted" from verbiage. Also added the ULTRA option, that it is set on a RAM or a hierarchy, and that sub-levels of hierarchy are not affected. In SRL_STYLE in Chapter 2, added additional -directives, and modified the Caution. Modified FSM_SAFE_STATE in Chapter 2 Modified USE_DSP48 in Chapter 2 Added an XDC example of fsm_safe_state added Using Synthesis Attributes in XDC files in Chapter 2 Added True Dual-Port Asymmetric RAM Write First (Verilog) in Chapter 3 Added UltraRAM Coding Templates in Chapter 3. Modified Black Boxes in Chapter 3 Modified Tristates in Chapter 3 Changed description in Advantages of SystemVerilog in Chapter 3 Added Latch With Positive Gate and Asynchronous Reset Coding Example (Verilog) in Chapter 3 Added a Caution to VHDL Assert Statements in Chapter 4 Modified Setting up Vivado to use VHDL-2008 in Chapter 5 Modified VHDL-2008 Language Support in Chapter 5 Described the unconstrained behavior For VHDL 200 in Types in Chapter 5 Modified Multiplexer Case Statement Example (Verilog) in Chapter 6. Marked $clog2 as supported only in SystemVerilog in Table 6-5. Added: "all others" to Table 6-5.

Table of Contents

Vivado Synthesis

Introduction

Synthesis is the process of transforming an RTL-specified design into a gate-level representation. Vivado® synthesis is timing-driven and optimized for memory usage and performance. Vivado synthesis supports a synthesizeable subset of:

- SystemVerilog

  IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800-2012)
- Verilog

  IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364-2005)
- VHDL

  IEEE Standard for VHDL Language (IEEE Std 1076-2002)

  VHDL 2008
- Mixed languages

  Vivado can also support a mix of VHDL, Verilog, and SystemVerilog.

The Vivado tools also support Xilinx® design constraints (XDC), which is based on the industry-standard Synopsys design constraints (SDC).

IMPORTANT: *Vivado synthesis does not support UCF constraints. Migrate UCF constraints to XDC constraints. For more information, see the "UCF to XDC Constraints Conversion" in the ISE to Vivado Design Suite Migration Guide* (UG911) [Ref 7].

There are two ways to setup and run synthesis:

- Use *Project mode*.
- Use *Non-Project mode*, applying Tool Command Language (Tcl) commands or scripts, and controlling your own design files.

See the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 8] for more information about operation modes. This chapter covers both modes in separate subsections.

Synthesis Methodology

The Vivado IDE includes a synthesis and implementation environment that facilitates a push button flow with synthesis and implementation runs. The tool manages the run data automatically, allowing repeated run attempts with varying Register Transfer Level (RTL) source versions, target devices, synthesis or implementation options, and physical or timing constraints.

Within the Vivado IDE, you can do the following:

- Create and save *strategies*. Strategies are configurations of command options, that you can apply to design runs for synthesis or implementation. See Creating Run Strategies.
- Queue the synthesis and implementation runs to launch sequentially or simultaneously with multi-processor machines. See Running Synthesis.
- Monitor synthesis or implementation progress, view log reports, and cancel runs. See Monitoring the Synthesis Run.

Using Synthesis

This section describes using the Vivado Integrated Design Environment (IDE) to set up and run Vivado synthesis. The corresponding Tcl Console commands follow each Vivado IDE procedure. See the following guides for more information regarding Tcl commands, and using Tcl:

- *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 3]
- *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894) [Ref 4]

VIDEO: *See these QuickTake Videos for more information:*
Synthesis Options *and* Synthesizing the Design.

Using Synthesis Settings

From **Flow Navigator > Synthesis**, set the synthesis options for the design, as follows:

1. Click the **Synthesis Settings** button, as shown in Figure 1-1.

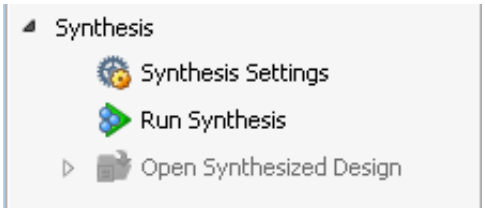

Figure 1-1: **Flow Navigator: Synthesis**

The Project Settings dialog box opens, as shown in the following figure.

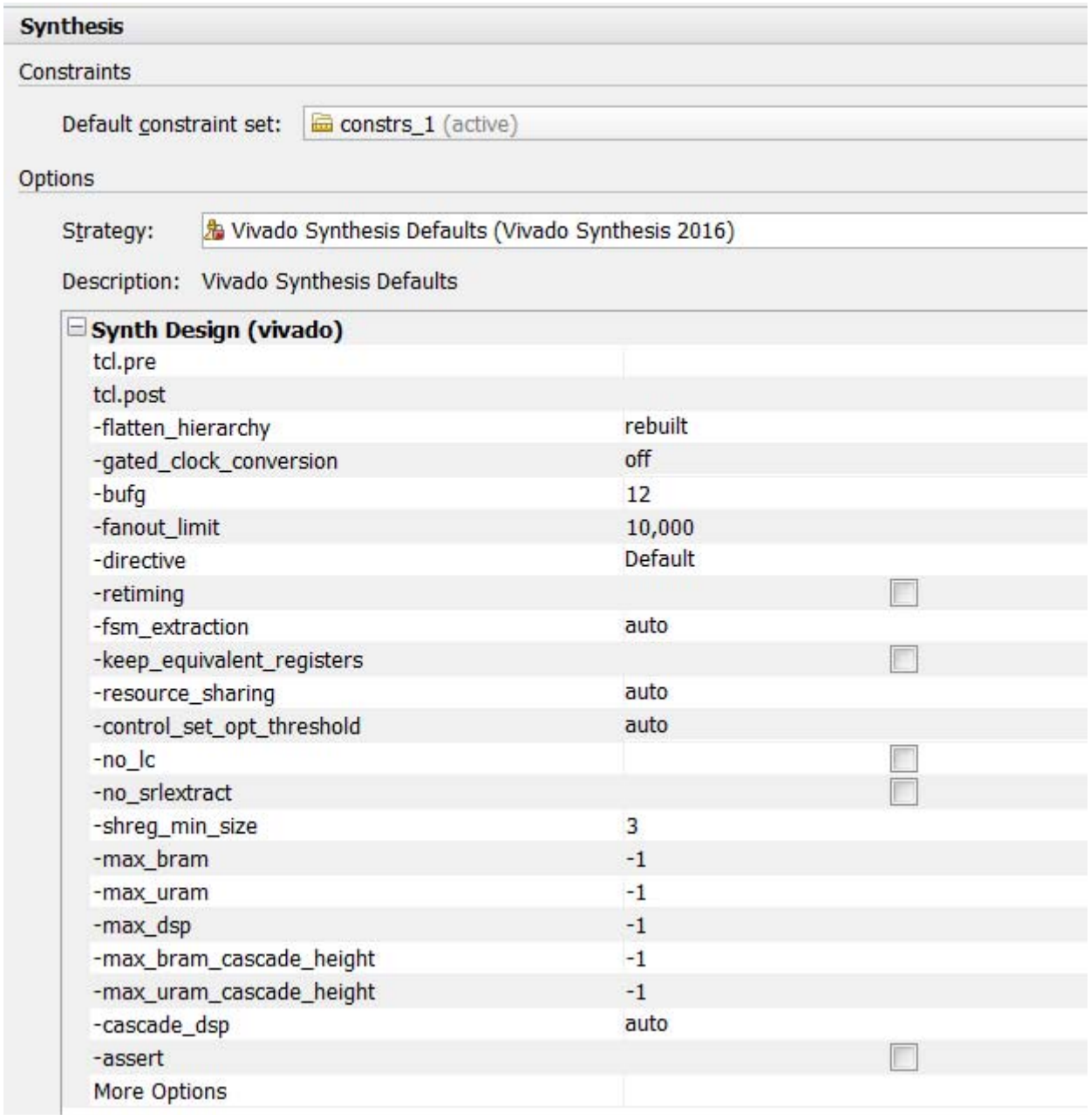

Figure 1-2: **Project Settings Dialog Box**

2. From the Project Setting dialog box, select:

   a. From Synthesis **Constraints**: Select the **Default Constraint Set** as the active *constraint set*. A constraint set is a set of files containing design constraints captured in Xilinx Design Constraints (XDC) files that you can apply to your design. The two types of design constraints are:

      - *Physical constraints*: These constraints define pin placement, and absolute, or relative, placement of cells such as block RAMs, LUTs, Flip-Flops, and device configuration settings.
      - *Timing constraints*: These constraints define the frequency requirements for the design. Without timing constraints, the Vivado Design Suite optimizes the design solely for wire length and placement congestion.

      See this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 9] for more information about organizing constraints.

      New runs use the selected constraint set, and the Vivado synthesis targets this constraint set for design changes.

      Tcl Command to Target Constraints Set

      ```
      -constrset <arg>
      ```

   b. From the **Options** area: Select a **Strategy** from the drop-down menu where you can view and select a predefined synthesis strategy to use for the synthesis run. There are different preconfigured strategies, as shown in the following figure.

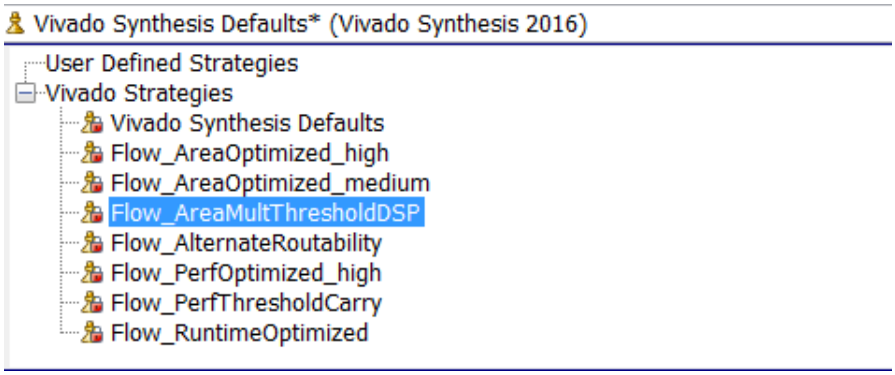

Figure 1-3: **Options - Strategies**

You can also define your own strategy. When you select a synthesis strategy, available Vivado strategy displays in the dialog box. You can override synthesis strategy settings by changing the option values as described in Creating Run Strategies.

For a list of all the strategies and their respective settings, see the **-directive** option in the following list, and refer to Table 1-2 to see a matrix of strategy default settings.

c. Select from the displayed options.

- **-flatten_hierarchy**: Determines how Vivado synthesis controls hierarchy.
  - **none**: Instructs the synthesis tool to never flatten the hierarchy. The output of synthesis has the same hierarchy as the original RTL.
  - **full**: Instructs the tool to fully flatten the hierarchy leaving only the top level.
  - **rebuilt**: When set, **rebuilt** allows the synthesis tool to flatten the hierarchy, perform synthesis, and then rebuild the hierarchy based on the original RTL. This value allows the QoR benefit of cross-boundary optimizations, with a final hierarchy that is similar to the RTL for ease of analysis.
- **-gated_clock_conversion:** Turns on and off the ability of the synthesis tool to convert the clocked logic with enables.

  The use of gated clock conversion also requires the use of an RTL attribute to work. See GATED_CLOCK, for more information.
- **-bufg**: Controls how many `BUFG`s the tool infers in the design. The Vivado design tools use this option when other `BUFG`s in the design netlists are not visible to the synthesis process.

  The tool infers up to the amount specified, and tracks how many `BUFG`s are instantiated in the RTL. For example, if the `-bufg` option is set to 12 and there are three `BUFG`s instantiated in the RTL, the tool infers up to nine more `BUFG`s.
- **-fanout_limit**: Specifies the number of loads a signal must drive before it starts replicating logic. This global limit is a general guide, and when the tool determines it is necessary, it can ignore the option. If a hard limit is required, see the `MAX_FANOUT` option described in Chapter 2, Synthesis Attributes.

  Note: The `-fanout_limit` switch does not impact control signals (such as `set`, `reset`, `clock` enable): use `MAX_FANOUT` to replicate these signals if needed.
- **-directive**: Replaces the **-effort_level** option. When specified, this option runs Vivado synthesis with different optimizations. See Table 1-2 for a list of all strategies and settings. Values are:
  - `Default`: Default settings. See Vivado Preconfigured Strategies in Table 1-2.
  - `RuntimeOptimized`: Performs fewer timing optimizations and eliminates some RTL optimizations to reduce synthesis run time.
  - `AreaOptimized_high`: Perform general area optimizations including forcing ternary adder implementation, applying new thresholds for use of carry chain in comparators, and implementing area-optimized multiplexers.

  - `AreaOptimized_medium:`
  - `AlternateRoutability`: Set of algorithms to improve route-ability (less use of MUXFs and CARRYs)
  - `AreaMapLargeShiftRegToBRAM`: Detects large shift registers and implements them using dedicated blocks of RAM.
  - `AreaMultThresholdDSP`: Lower threshold for dedicated DSP block inference.
  - `FewerCarryChains`: Higher operand size threshold to use LUTs instead of the carry chain.

- **-retiming:** This boolean option `<on|off>` provides an option improve circuit performance for intra-clock sequential paths by automatically moving registers (register balancing) across combinatorial gates or LUTs. It maintains the original behavior and latency of the circuit and does not require changes to the RTL sources. The default is `off`.
- **-fsm_extraction**: Controls how synthesis extracts and maps finite state machines. FSM_ENCODING describes the options in more detail.
- **-keep_equivalent_registers**: Prevents merging of registers with the same input logic.
- **-resource_sharing**: Sets the sharing of arithmetic operators between different signals. The values are **auto**, **on** and **off**. The **auto** value sets performing resource sharing to depend on the timing of the design, **on** means that it is always on, and **off** means that it is always off.
- **-control_set_opt_threshold**: Sets the threshold for clock enable optimization to the lower number of control sets. The default is **auto** which means the tool will choose a value based on the device being targeted. Any positive integer value is supported.

  The given value is the number of fanouts necessary for the tool to move the control sets into the `D` logic of a register. If the fanout is higher than the value, the tool attempts to have that signal drive the `control_set_pin` on that register.
- **-no_lc**: When checked, this option turns off LUT combining.
- **-no_srlextract:** When checked, this option turns off SRL extracation for the full design so that they are implemented as simple registers.
- **-shreg_min_size:** Is the threshold for inference of SRLs. The default setting is 3. This sets the number of sequential elements that would result in the inference of an SRL for fixed delay chains (static SRL). Strategies define this setting as 5 and 10 also. See Table 1-2 for a list of all strategies and settings.
- **-max_bram:** Describes the maximum number of block RAM allowed in the design. Often this is used when there are black boxes or third-party netlists in the design and allow the designer to save room for these netlists.

Note: The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.

- **-max_uram**: Sets the maximum number of Ultra RAM blocks allowed in design. The default is -1. The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.
- **-max_dsp:** Describes the maximum number of block DSP allowed in the design. Often this is used when there are black boxes or third-party netlists in the design, and allows room for these netlists. The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.
- **-max_bram_cascade_height:** Controls the maximum number of BRAM that can be cascaded by the tool. The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.
- **-max_uram_cascade_height:** Controls the maximum number of URAM that can be cascaded by the tool. The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.
- **-cascade_dsp**: Controls how adders in sum DSP block outputs are implemented. By default, the sum of the DSP outputs is computed using the block built-in adder chain. The value **tree** forces the sum to be implemented in the fabric. The values are: **auto**, **tree**, and **force**. The default is **auto**.
- **-assert:** Enable VHDL assert statements to be evaluated. A severity level of failure or error stops the synthesis flow and produces an error. A severity level of warning generates a warning.
- The `tcl.pre` and `tcl.post` options are hooks for Tcl files that run immediately before and after synthesis.

  Note: Paths in the `tcl.pre` and `tcl.post` scripts are relative to the associated run directory of the current project: `<project>/<project.runs>/<run_name>`.

  See this link in *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894) [Ref 4] for more information about Tcl scripting.

  You can use the `DIRECTORY` property of the current project or current run to define the relative paths in your scripts:

Tcl Commands to Get Property

```
- get_property DIRECTORY [current_project]
- get_property DIRECTORY [current_run]
```

Creating Run Strategies

A strategy is a set of switches to the tools, which are defined in a pre-configured set of options for the synthesis application or the various utilities and programs that run during implementation. Each major release has version-specific strategy options.

VIDEO: *See the following QuickTake Video for more information:* Creating and Managing Runs.

1. Select **Synthesis > Synthesis Settings**.
2. Select a strategy from the **Strategy** drop-down list, and click **OK**.

Saving a Modified Strategy

1. Select an existing run strategy and modify options as needed, and click the **Save Strategy As** button in the **Project Settings > Synthesis** page.

   The Save Strategy As dialog box opens, as shown in the following figure.

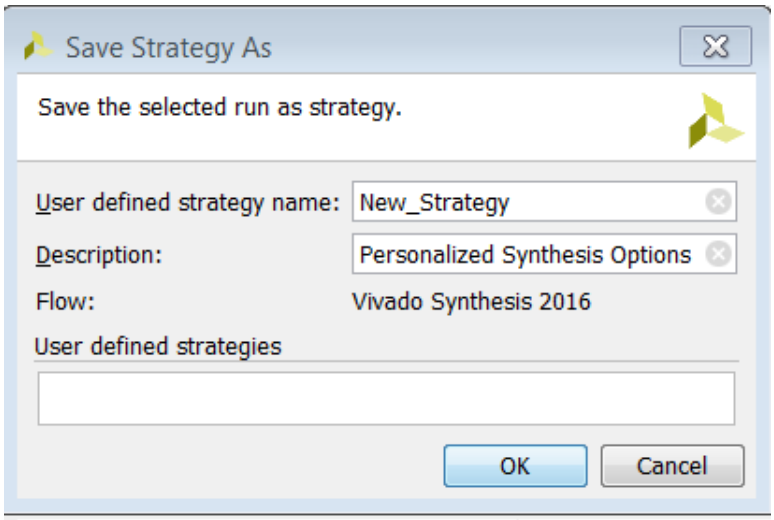

Figure 1-4: **Save Strategy As Dialog Box**

2. Specify a **User defined strategy name** and **Description**.
3. Click **OK**.

   The Strategy drop-down displays any user defined strategies.

Creating a New Strategy

You can create a custom strategy on the **Strategies** page of the Vivado Options dialog box.

1. Click **Tools > Options > Strategies**.

2. Select Vivado Synthesis from the Flow drop-down list, and select one of the existing strategies.
3. Do one of the followings:
   - Click the **Create New Strategy** toolbar button to create a copy of the selected existing strategy. Make changes as needed and click **Apply**.
   - Click the Create New Strategy toolbar button. The New Strategy dialog box opens.

Figure 1-5: **New Strategy Dialog Box**

   - Enter a **Name**, **Type** of the strategy, **Tool version**, and **Description**, and click **OK**.

   The created strategy is displayed under **User Defined Strategies**.

Setting Synthesis Inputs

Vivado synthesis allows two input types: RTL source code and timing constraints. To add RTL or constraint files to the run:

1. In the **Flow Navigator,** select the **Add Sources** command to open the Add Sources wizard, shown in the following figure.

Figure 1-6: **Add Sources Dialog Box**

2. Select an option corresponding to the files to add.

   Figure 1-7 shows the next page that displays when selecting **Add or Create Design Sources**.

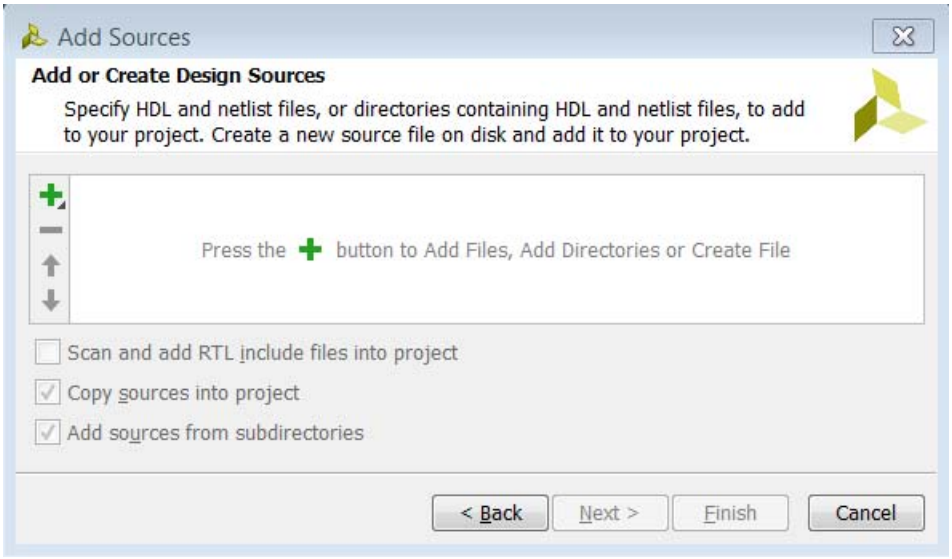

Figure 1-7: **Add or Create Sources Dialog Box**

3. Add constraint, RTL, or other project files.

   See this link the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) [Ref 15] for more information about creating RTL source projects.

   The Vivado synthesis tool reads the subset of files that can be synthesized in VHDL, Verilog, SystemVerilog, or mixed language options supported in the Xilinx tools.

Details on supported HDL constructs are provided in the following chapters:

- Chapter 3, HDL Coding Techniques
- Chapter 4, VHDL Support
- Chapter 5, VHDL-2008 Language Support
- Chapter 6, Verilog Language Support
- Chapter 7, SystemVerilog Support
- Chapter 8, Mixed Language Support

Vivado synthesis also supports several RTL attributes that control synthesis behavior. Chapter 2, Synthesis Attributes, describes these attributes.

Vivado synthesis uses the XDC file for timing constraints.

IMPORTANT: *Vivado Design Suite does not support the UCF format. See this link in the ISE to Vivado Design Suite Migration Guide (UG911) [Ref 7] for the UCF to XDC conversion procedure.*

Controlling File Compilation Order

A specific compile order is necessary when one file has a declaration and another file depends upon that declaration. The Vivado IDE controls RTL source files compilation from the top of the graphical hierarchy shown in the Sources window Compile Order window to the bottom.

The Vivado tools automatically identify and set the best top-module candidate, and automatically manage the compile order. The top-module file and all sources that are under the active hierarchy are passed to synthesis and simulation in the correct order.

In the Sources window, a popup menu provides the **Hierarchy Update** command. The provided options specify to the Vivado IDE how to handle changes to the top-module and to the source files in the design.

The default setting, **Automatic Update and Compile Order**, specifies that the tool does the following:

- Manages the compilation order as shown in the Compilation Order window

- Shows which modules are used and where they are in the hierarchy tree in the Hierarchy window

The compilation order updates automatically as you change source files.

To modify the compile order before synthesis:

1. Select a file and right-click **Hierarchy Update > Automatic Update, Manual Compile Order** so the Vivado IDE automatically determines the best top module for the design and allows manual specification of the compilation order.

   Manual Compile is `OFF` by default. If you select a file and move it in the Compile Order window, a popup menu asks if you want Manual Compile turned on, as shown in the following figure.

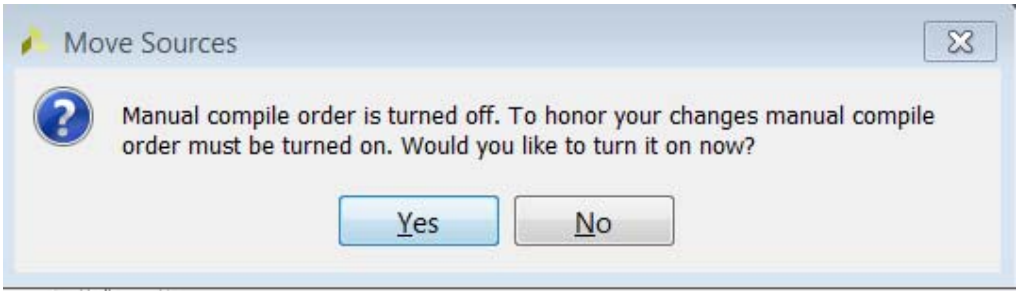

Figure 1-8: **Move Sources Option**

2. From the Sources window Compile order tab, drag and drop files to arrange the compilation order, or use the menu **Move Up** or **Move Down** commands.

   Other options are available from the Hierarchy Update context menu, as shown in the following figure.

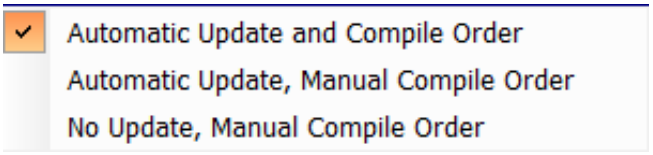

Figure 1-9: **Hierarchy Update Options**

   See this link in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 8] for information about design flows.

Defining Global Include Files

The Vivado IDE supports designating one of more Verilog or Verilog Header source files as global `` `include `` files and processes those files before any other sources.

Verilog typically requires that you place an `` `include `` statement at the top of any Verilog source file that references content from a another Verilog or header file.

Designs that use common header files might require multiple `include statements to be repeated across multiple Verilog sources used in the design.

To designate a Verilog or Verilog header file as a global `include file:

1. In the Sources window, select the file.
2. Check the **IS_GLOBAL_INCLUDE** checkbox in the Source File Properties window, as shown in the following figure.

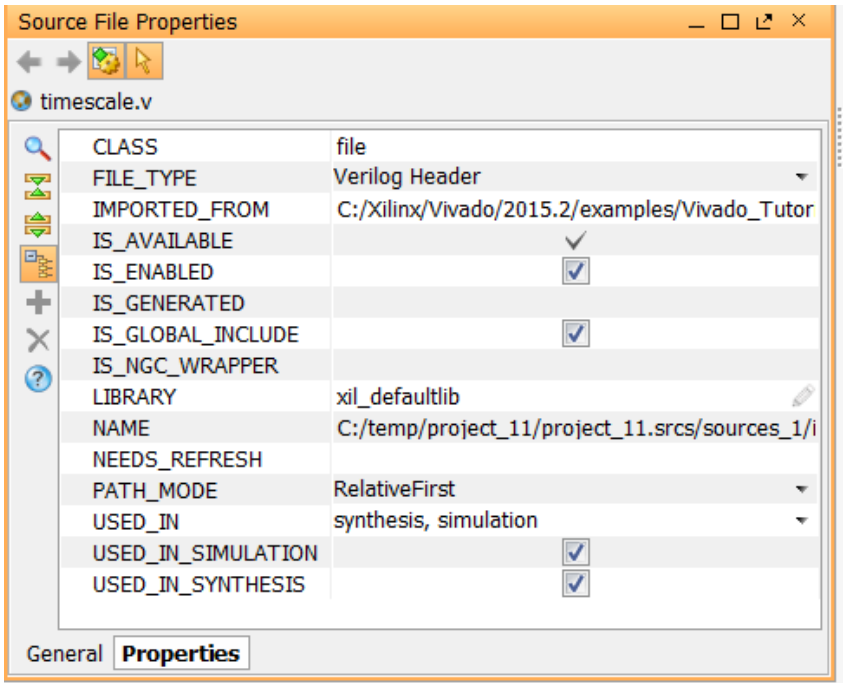

Figure 1-10: **Source File Properties Window**

TIP: *In Verilog, reference header files that are specifically applied to a single Verilog source (for example; a particular `define macro), with an `include statement instead of marking it as a global `include file.*

See this link in the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 2], for information about the Sources window.

Running Synthesis

A *run* defines and configures aspects of the design that are used during synthesis. A synthesis run defines the:

- Xilinx device to target during synthesis
- Constraint set to apply
- Options to launch single or multiple synthesis runs

- Options to control the results of the synthesis engine

To define a run of the RTL source files and the constraints:

1. From the main menu, select **Flow > Create Runs** or click the **Create Runs** button to open the Create New Runs wizard.

   The Create New Runs dialog box opens, as shown in the following figure.

Figure 1-11: **Create New Runs Dialog Box**

2. Select **Synthesis**, and click **Next**.

   The Configure Synthesis Runs dialog box opens, as shown in the following figure.

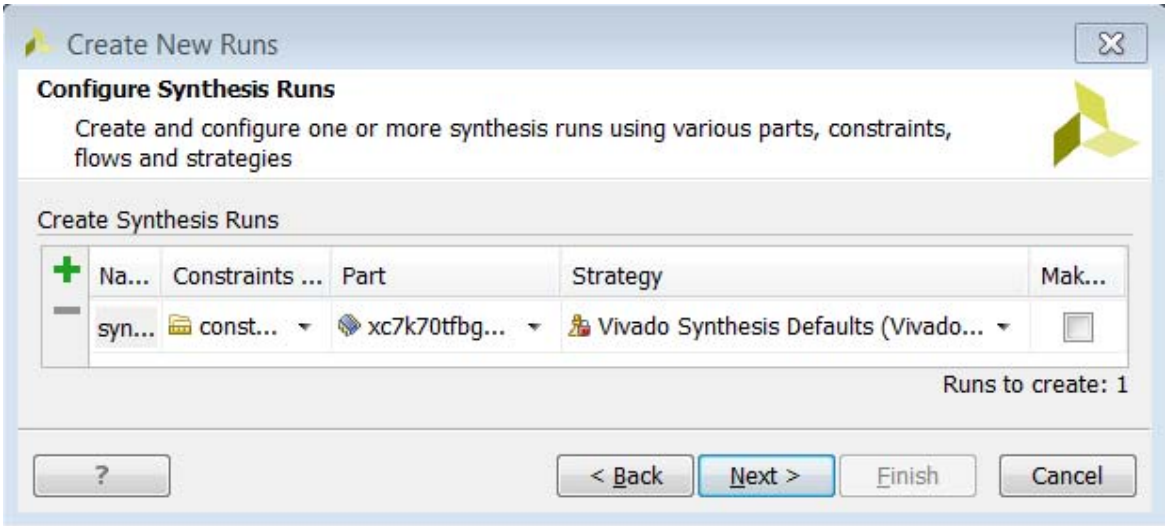

Figure 1-12: **Configure Synthesis Runs Dialog Box**

Configure the synthesis run with the **Name**, **Constraints Set**, **Part, Strategy,** and check **Make Active**, if you want this run to be the *active* run.

The Vivado IDE contains a default strategy. You can set a specific name for the strategy run or accept the default name(s), which are numbered as `synth_1`, `synth_2`, and so forth. To create your own run strategy, see Creating Run Strategies.

For more detailed information on constraints, see this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 9].

For more detailed information about constraint processing order, see this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 9].

After some constraints are processed for a project, those constraint attributes can become design *Properties*. For more information regarding properties, see the *Vivado Design Suite Properties Reference Guide* (UG912) [Ref 12].

The Launch Options dialog box opens, as shown in the following figure.

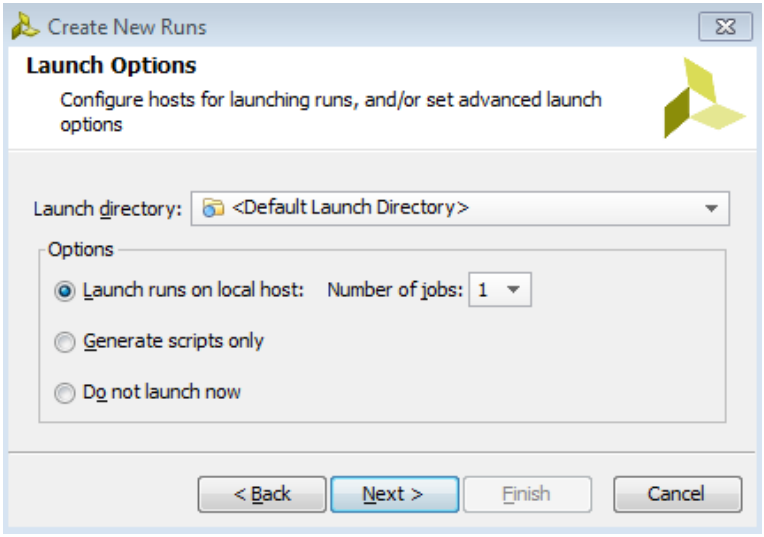

Figure 1-13: **Launch Options Dialog Box**

3. In the Launch Options dialog box, set the options, as follows, then click **Next**.
   - In the **Launch Directory** drop-down option, browse to, and select the directory from which to launch the run.
   - In the **Options** area, choose one of the following:
     - **Launch Runs on Local Host**: Runs the options from the machine on which you are working. The **Number of jobs** drop-down lets you specify how many runs to launch.
     - **Launch Runs on Remote Hosts**: Launches the runs on a remote host (Linux only) and configure that host.

IMPORTANT: *The Remote Host option is available on Linux only.*

See this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 5], for more information about launching runs on remote hosts in Linux. The **Configure Hosts** button lets you configure the hosts from this dialog box.

- **Generate scripts only**: Generates scripts to run later. Use `runme.bat` (Windows) or `runme.sh` (Linux) to start the run.
- **Do not launch now**: Lets you save the settings that you defined in the previous dialog boxes and launch the runs at a later time.

After setting the Create New Runs wizard option, viewing the Create New Runs Launch Options summary, and starting a run, you can see the results in the Design Runs window, as shown in Figure 1-14.

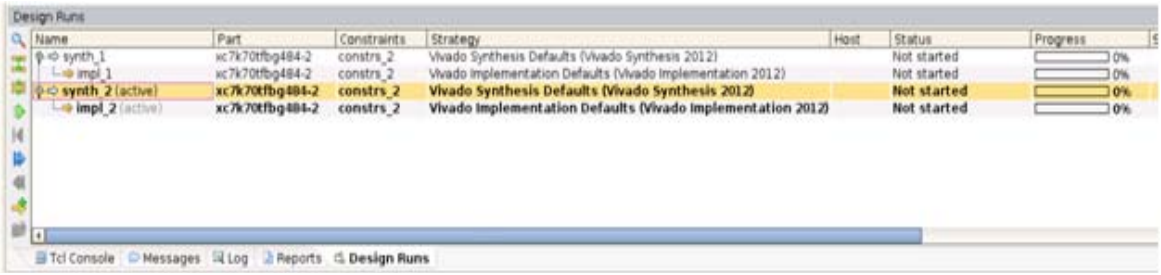

Figure 1-14: **Design Runs Window**

Using the Design Runs Window

The Design Runs window displays the synthesis and implementation runs created in a project and provides commands to configure, manage, and launch the runs.

If the Design Runs window is not already displayed, select **Window > Design Runs** to open the Design Runs window. A synthesis run can have multiple implementation runs. To expand and collapse synthesis runs, use the tree widgets in the window. The Design Runs window reports the run status, (when the run is not started, is in progress, is complete, or is out-of-date).

Runs become out-of-date when you modify source files, constraints, or project settings.

To reset or delete specific runs, right-click the run and select the appropriate command.

Setting the Active Run

Only one synthesis run and one implementation run can be *active* in the Vivado IDE at any time. All the reports and tab views display the information for the active run.

The Project Summary window only displays compilations, resource, and summary information for the active run.

To make a run active, select the run in the Design Runs window and use the **Make Active** command from the popup menu to set it as the active run.

Launching a Synthesis Run

To launch a synthesis run, do one of the following:

- From the Flow Navigator section, click the **Run Synthesis** command.
- From the main menu, select the **Flow > Run Synthesis** command.
- In the Design Runs window, right-click the run, and select **Launch Runs**.

The first two options start the active synthesis run. The third option opens the Launch Selected Runs window.

Here, you can select to run on local host, run on a remote host, or generate the scripts to be run. See this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 5], for more information about using remote hosts.

TIP: *Each time a run is launched, Vivado synthesis spawns a separate process. Be aware of this when examining messages, which are process-specific.*

Setting a Bottom-Up Out-of-Context Flow

You can set a bottom-up flow by selecting any HDL object to run as a separate *out-of-context* (OOC) flow. For an overview of the OOC flow, see this link in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 8].

The OOC flow behaves as follows:

- Lower OOC modules are run separately from the top-level, and have their own constraint sets.
- OOC modules can be run as needed.
- After you have run synthesis on an OOC module, it does not need to be run again, unless you change the RTL or constraints for that run.

  This can result in a large runtime improvement for the top-level because synthesis no longer needs to be run on the full design.

To set up a module for an OOC run:

1. Find that module in the hierarchy view, and right-click the **Set As Out-Of-Context for Synthesis** option, shown in the following figure.

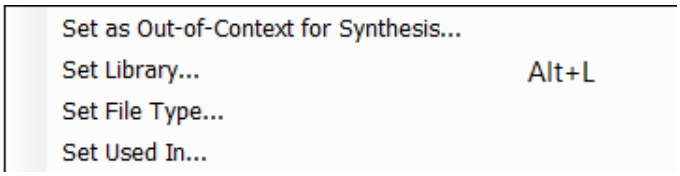

Figure 1-15: **Set As Out-of-Context for Synthesis Option**

This brings up the Set as Out-of-Context for Synthesis dialog box as shown in the following figure.

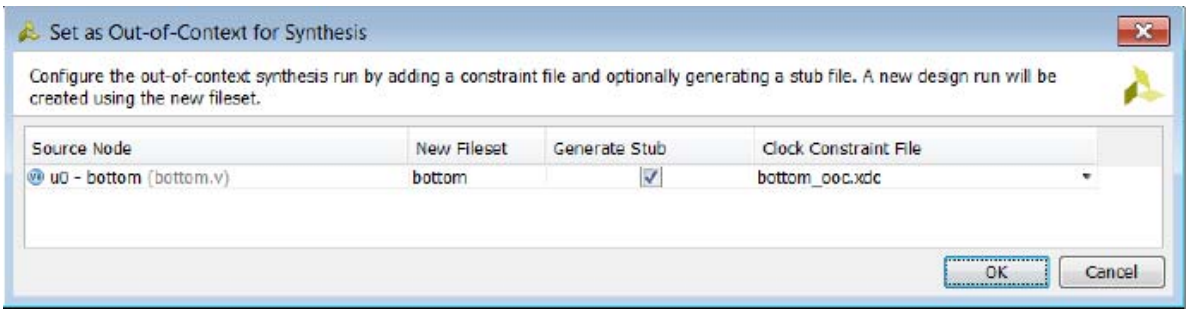

Figure 1-16: **Set as Out-of-Context for Synthesis Dialog Box**

The Set as Out-of-Context for Synthesis dialog box displays the following information and options:

- **Source Node**: Module to which you are applying the OOC.
- **New Fileset**: Lists the New Fileset name, which you can edit.
- **Generate Stub**: A checkbox that you can check to have the tool create a stub file.
- **Clock Constraint File**: Choose to have the tool create a new XDC template for you, or you can use the drop-down menu to copy an existing XDC file over to this Fileset. This XDC file should have clock definitions for all your clock pins on the OOC module.

RECOMMENDED: *Leave the stub file option on. If you turn it off, you must create your own stub files and set them in the project.*

2. Click **OK**.

   The tool sets up the OOC run automatically. You can see it as a new run in the Design Runs tab, and also see it as a block source in the Compile Order tab, as shown in the following figure.

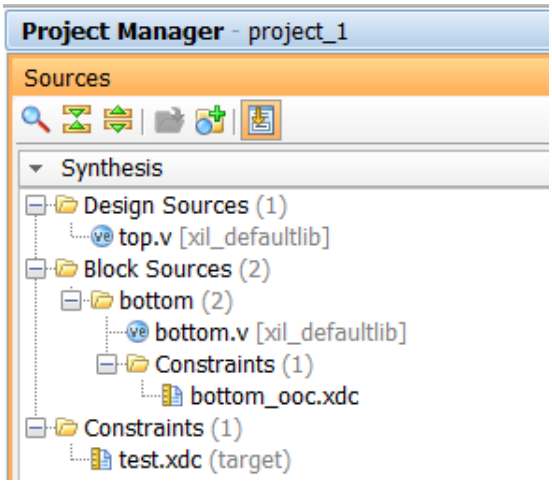

Figure 1-17: **Compile Order Tab**

When you set a flow to **Out-of-Context**, a new run is set up in the tool.

To run the option, right-click and select **Launch Runs**, as described in Launching a Synthesis Run. This action sets the lower-level as a top module and runs synthesis on that module without creating I/O buffers.

Note: You can also right-click and run implementation as well for analysis purposes only. This will not have accurate timing because the clock tree is not routed. To have the tool route the clock, use the following command in the XDC file that is being used for the OOC level for every clock that was defined in that XDC file.

To retrieve the ports using a Tcl command, type the following in the Tcl Console:

- **Tcl Command:** `set_property HD.CLK_SRC_BUFGCTRL_X0Y0 [get_ports\ <clk_port_name>]`

Note: For more information, see the *Vivado Design Suite User Guide: Hierarchical Design* (UG905) [Ref 6].

IMPORTANT: *The implemented IP is for analysis only and is not used during synthesis or implementation of the top-level design. For information on using an implemented version of the IP, see the Vivado Design Suite User Guide: Hierarchical Design (UG905)* [Ref 6].

The run saves the netlist from synthesis and creates a stub file (if you selected that option) for later use. The stub file is the lower-level with inputs and outputs and the black-box attribute set.

When you run the top-level module again, the bottom-up synthesis inserts the stub file into the flow and compiles the lower-level as a black box. The implementation run inserts the lower-level netlist, thus completing the design.

CAUTION! *Do not use the Bottom-Up OOC flow when there are Xilinx IP in OOC mode in the lower-levels of the OOC module. To have Xilinx IP in an OOC module, turn off the IP OOC mode. Do not use this flow when there are parameters on the OOC module, or the ports of the OOC module are user-defined types. Those circumstances cause errors later in the flow.*

Manually Setting a Bottom-Up Flow and Importing Netlists

To manually run a bottom-up flow, instantiate a lower-level netlist or third-party netlist as a black box, and the Vivado tools will fit that black box into the full design after synthesis completes. The following sections describe the process.

IMPORTANT: *Vivado synthesis does not synthesize or optimize encrypted or non-encrypted synthesized netlists; consequently, XDC constraints or synthesis attributes do not have an effect on synthesis with an imported core netlist. Also, Vivado synthesis does not read the core netlist and modify the instantiated components by default; however, Vivado synthesis does synthesize secure IP and RTL. Constraints do affect synthesis results.*

Creating a Lower-Level Netlist

To create a lower-level netlist, set up a project with that netlist as the top-level module. Before you run synthesis, set the out-of-context (OOC) mode as shown in the following figure.

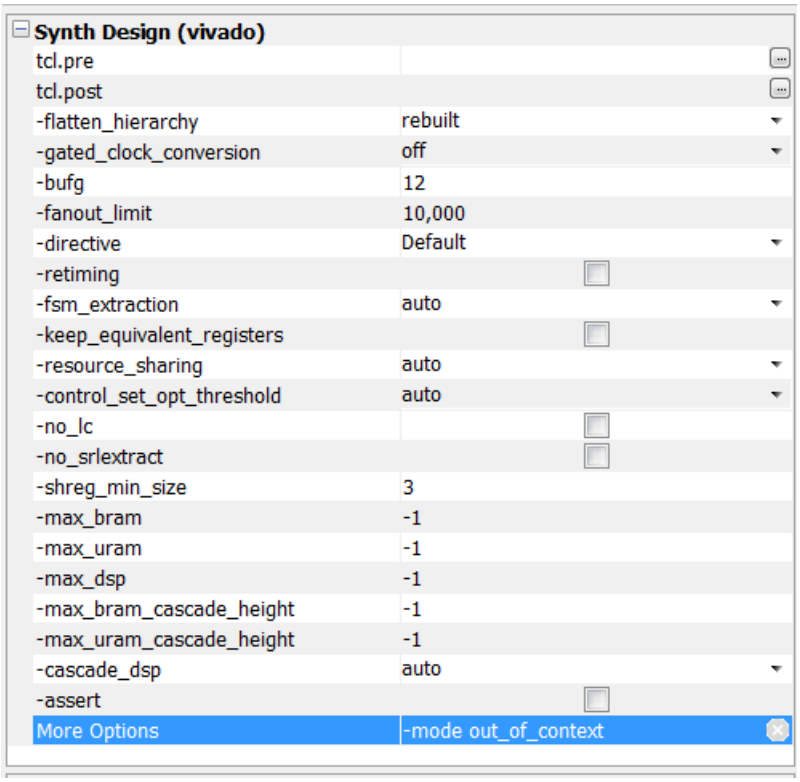

Figure 1-18: **Synthesis Constraint Options**

This instructs the tool to not insert any I/O buffers in this level.

After you run synthesis, open the synthesized design, and in the Tcl Console, type the following in the Tcl Console:

- **Tcl Command:** `write_edif <design_name>.edf`

Instantiating the Lower-Level Netlist in a Design

To run the top-level design with the lower-level netlist or third-party netlist, instantiate the lower-level as a black box. To do so, you must provide a description of the port in lower-level to the Vivado tool. In the Setting a Bottom-Up Out-of-Context Flow, this is referred to as a *stub* file.

IMPORTANT: *The port names provided to the Vivado tool and the port names in the netlist must match.*

In VHDL, describe the ports with a component statement, as shown in the following code snippet:

```
component <name>
   port (in1, in2 : in std_logic;
     out1 : out std_logic);
end component;
```

Because Verilog does not have an equivalent of a component, use a wrapper file to communicate the ports to the Vivado tool. The wrapper file looks like a typical Verilog file, but contains only the ports list, as shown in the following code snippet:

```
module <name> (in1, in2, out1);
  input in1, in2;
  output out1;
endmodule
```

Putting Together the Manual Bottom-Up Components

After you create the lower-level netlist and instantiate the top-level netlists correctly, you can either add the lower-level netlists to the Vivado project in Project mode, or you can use the `read_edif` or `read_verilog` command in Non-Project mode.

In both modes, the Vivado tool merges the netlist after synthesis.

Note: If a design is from third-party netlists only, and no other RTL files are meant to be part of the project, you can either create a project with just those netlists, or you can use the `read_edif` and `read_verilog` Tcl commands along with the `link_design` Tcl command in Non-Project mode.

Using Third-Party Synthesis Tools with Vivado IP

Xilinx IP that is available in the Vivado IP Catalog is designed, constrained, and validated with the Vivado Design Suite synthesis.

Most Xilinx-delivered IP has HDL that is encrypted with IEEE P1735, and no support is available for third-party synthesis tools for Xilinx IP.

To instantiate Xilinx IP that is delivered with the Vivado IDE inside of a third-party synthesis tool, the following flow is recommended:

1. Create the IP customization in a managed IP project.
2. Generate the output products for the IP including the synthesis design checkpoint (DCP).

   The Vivado IDE creates a stub HDL file, which is used in third-party synthesis tools to infer a black box for the IP (`_stub.v|_stub.vhd`).

The stub file contains directives to prevent I/O buffers from being inferred; you might need to modify these files to support other synthesis tool directives.

3. Synthesize the design with the stub files for the Xilinx IP.
4. Use the netlist produced by the third-party synthesis tool, and the DCP files for the Xilinx IP, then run Vivado implementation. For more information, see this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 19].

Moving Processes to the Background

As the Vivado IDE initiates the process to run synthesis or implementation, an option in the dialog box lets you put the process into the background. When you put the run in the background, it releases the Vivado IDE to perform other functions, such as viewing reports.

Monitoring the Synthesis Run

Monitor the status of a synthesis run from the Log window, shown in the following figure. The messages that show in this window during synthesis are also the messages included in the synthesis log file.

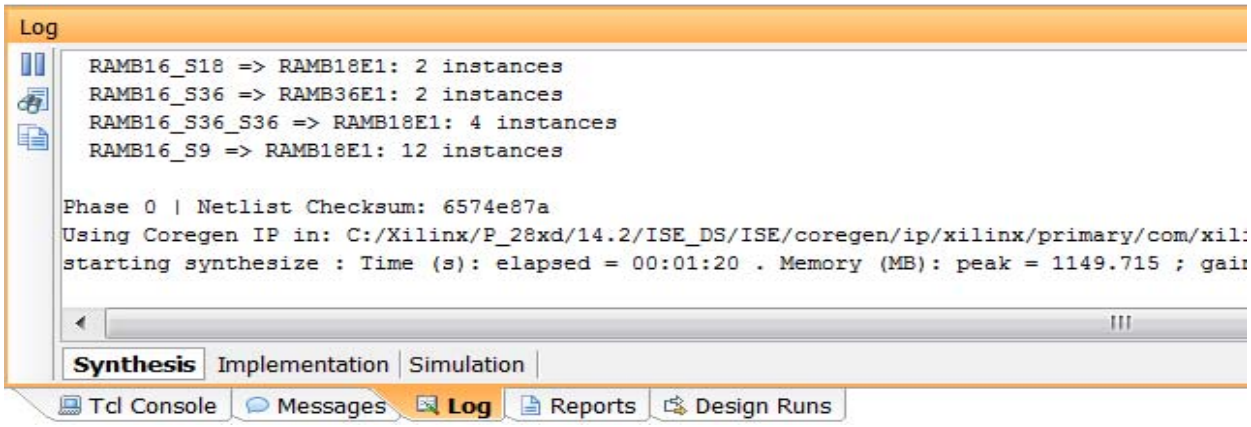

Figure 1-19: **Log Window**

Following Synthesis

After the run is complete, the Synthesis Completed dialog box opens, as shown in the following figure.

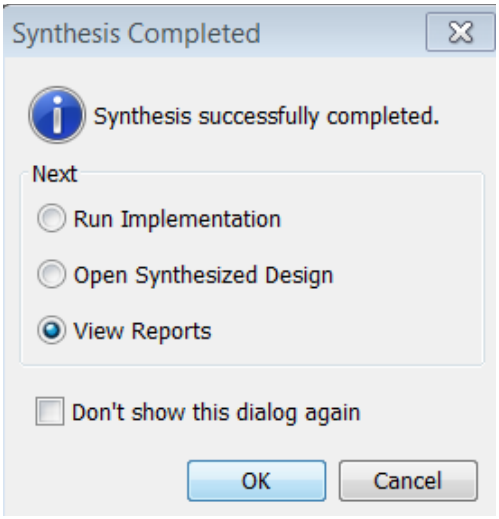

Figure 1-20: **Synthesis Completed Dialog Box**

Select an option:

- **Run Implementation**: Launches implementation with the current Implementation Project Settings.
- **Open Synthesized Design**: Opens the synthesized netlist, the active constraint set, and the target device into Synthesized Design environment, so you can perform I/O pin planning, design analysis, and floorplanning.
- **View Reports:** Opens the Reports window so you can view reports.
- Use the **Don't show this dialog again** checkbox to stop this dialog box display.

TIP: *You can revert to having the dialog box present by selecting* **Tools > Options > Windows Behavior**.

Analyzing Synthesis Results

After synthesis completes, you can view the reports, and open, analyze, and use the synthesized design. The Reports window contains a list of reports provided by various synthesis and implementation tools in the Vivado IDE.

VIDEO: *See the following QuickTake Video for more information:* Advanced Synthesis using Vivado

Open the **Reports** view, as shown in the following figure, and select a report for a specific run to see details of the run.

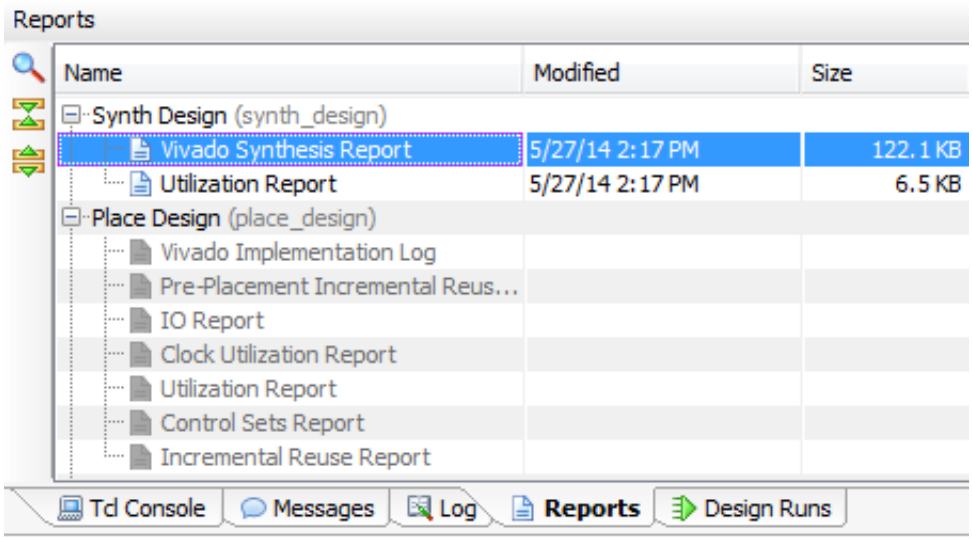

Figure 1-21: **Synthesis Reports View**

Using the Synthesized Design Environment

The Vivado IDE provides an environment to analyze the design from several different perspectives. When you open a synthesized design, the software loads the synthesized netlist, the active constraint set, and the target device.

To open a synthesized design, from the **Flow Navigator > Synthesis**, select **Open Synthesized Design**.

You can also open a design from the main menu, by selecting **Flow > Open Synthesized Design**.

With a synthesized design open, the Vivado IDE opens a Device window, as shown in the following figure.

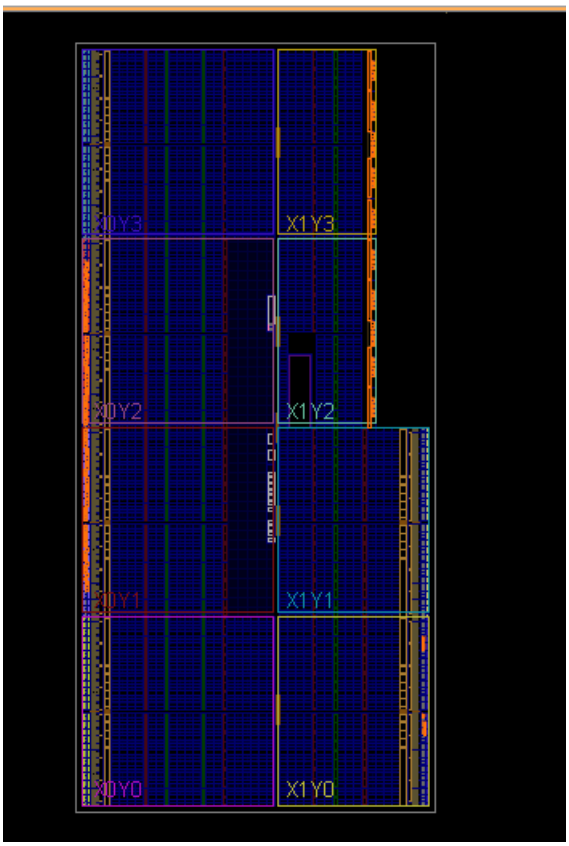

Figure 1-22: **Device Window**

From this perspective, you can examine the design logic and hierarchy, view the resource utilization and timing estimates, or run design rule checks (DRCs).

For more information, see the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 10].

Viewing Reports

After you run Vivado synthesis, a Vivado Synthesis Report and a Utilization report are available from the Reports tab, as shown in the following figure.

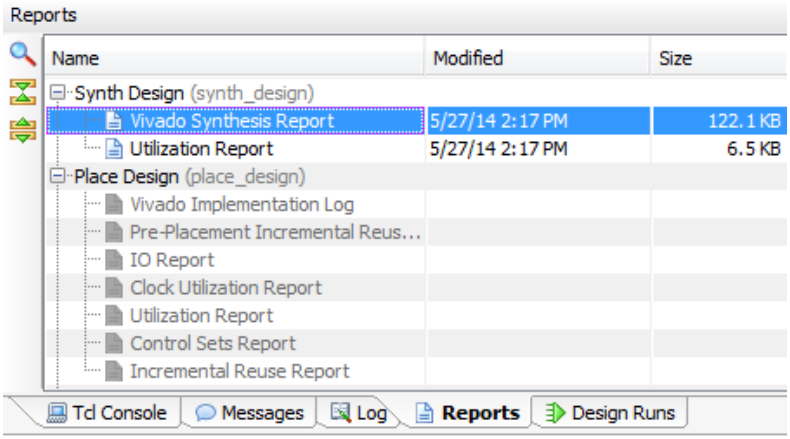

Figure 1-23: **Reports Tab Listings after Synthesis**

Exploring the Logic

The Vivado IDE provides several logic exploration perspectives: All windows cross-probe to present the most useful information:

- The Netlist and Hierarchy windows contain a navigable hierarchical tree-style view.
- The Schematic window allows selective logic expansion and hierarchical display.
- The Device window provides a graphical view of the device, placed logic objects, and connectivity.

Exploring the Logic Hierarchy

The Netlist window displays the logic hierarchy of the synthesized design. You can expand and select any logic instance or net within the netlist.

As you select logic objects in other windows, the Netlist window expands automatically to display the selected logic objects, and the information about instances or nets displays in the Instance or Net Properties windows.

The Synthesized Design window displays a graphical representation of the RTL logic hierarchy. Each module is sized in relative proportion to the others, so you can determine the size and location of any selected module.

To open the Hierarchy window:

1. In the Netlist window, right-click to bring up the context menu.
2. Select **Show Hierarchy**, as shown in the following figure. Also, you can press **F6** to open the Hierarchy window.

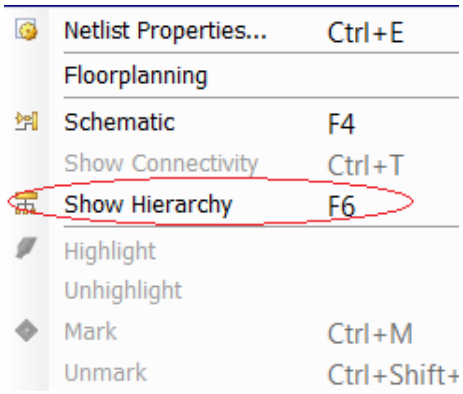

Figure 1-24: **Show Hierarchy Option**

Exploring the Logical Schematic

The Schematic window allows selective expansion and exploration of the logical design. You must select at least one logic object to open and display the Schematic window.

In the Schematic window, view and select any logic. You can display groups of timing paths to show all of the instances on the paths. This aids floorplanning because it helps you visualize where the timing critical modules are in the design.

To open the Schematic window:

1. Select one or more instances, nets, or timing paths.
2. Select **Schematic** from the window toolbar or the right-click menu, or press the **F4** key.

The window opens with the selected logic displayed, as shown in the following figure.

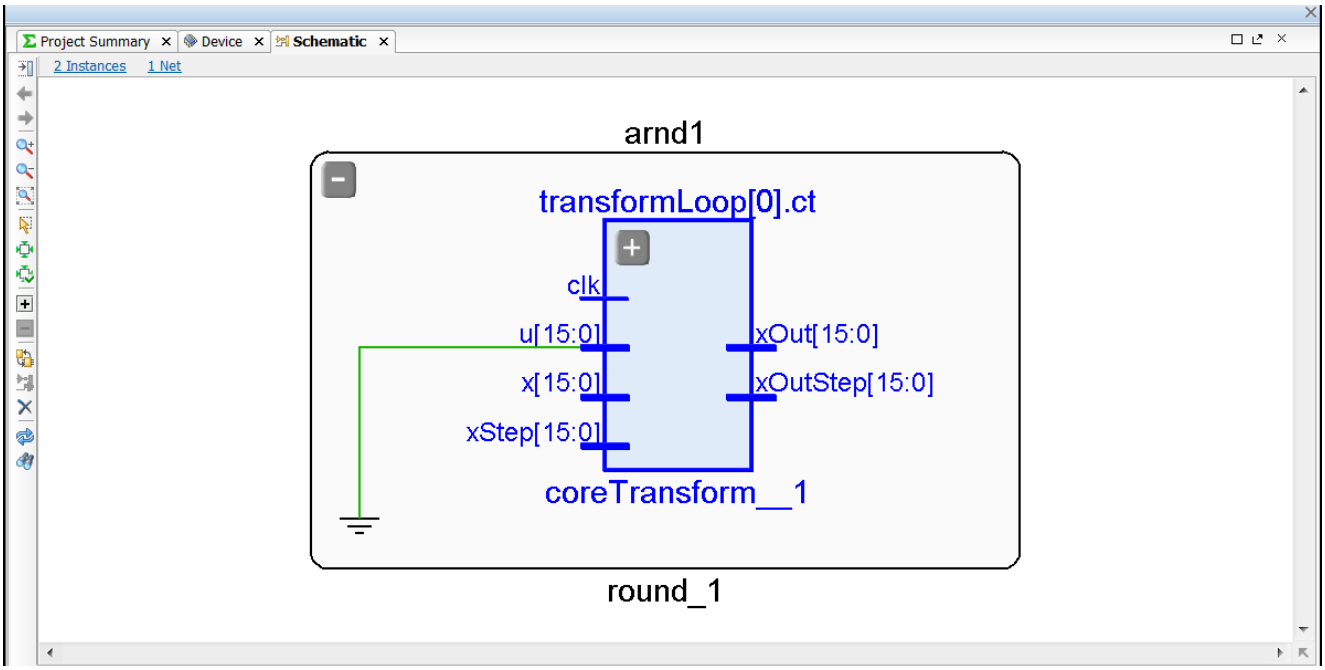

Figure 1-25: **Schematic Window**

You can then select and expand the logic for any pin, instance, or hierarchical module.

Running Timing Analysis

Timing analysis of the synthesized design is useful to ensure that paths have the necessary constraints for effective implementation. The Vivado synthesis is timing-driven and adjusts the outputs based on provided constraints.

As more physical constraints, such as `Pblocks` and `LOC` constraints, are assigned in the design, the results of the timing analysis become more accurate, although these results still contain some estimation of path delay. The synthesized design uses an estimate of routing delay to perform analysis.

You can run timing analysis at this level to ensure that the correct paths are covered and for a more general idea of timing paths.

IMPORTANT: *Only timing analysis after implementation (place and route) includes the actual delays for routing. Running timing analysis on the synthesized design is not as accurate as running timing analysis on an implemented design.*

Running Synthesis with Tcl

The Tcl command to run synthesis is `synth_design`. Typically, this command is run with multiple options, for example:

```
synth_design -part xc7k30tfbg484-2 -top my_top
```

In this example, `synth_design` is run with the `-part` option and the `-top` option.

In the Tcl Console, you can set synthesis options and run synthesis using Tcl command options. To retrieve a list of options, type `synth_design -help` in the Tcl Console. The following snippet is an example of the `-help` output: `synth_design -help`.

```
Description:
Synthesize a design using Vivado Synthesis and open that design
Syntax:
synth_design  [-name <arg>] [-part <arg>] [-constrset <arg>] [-top <arg>]\
              [-include_dirs <args>] [-generic <args>] \
             [-verilog_define <args>] [-flatten_hierarchy <arg>]\
              [-gated_clock_conversion <arg>] [-directive <arg>\]
              [-rtl][-retiming <arg>] [-bufg <arg>] [-no_lc]
              [-fanout_limit <arg>][-shreg_min_size <arg>] [-mode <arg>]\
              [-fsm_extraction <arg>][-assert] [-no_srlextract]\
              [-keep_equivalent_registers] [-resource_sharing <arg>]\
              [-control_set_opt_threshold <arg>] [-max_bram <arg>]\
              [-max_dsp <arg>][-max_uram] [-cascade_dsp <arg>]\
              [-max_cascade_uram_height <arg>]
              [-max_cascade_uram_height <arg>]\
              [-quiet] [-verbose]\
Returns:
design object
Usage:
Name                           Description
  ----------------------------------------
[-name]                        Design name
[-part]                        Target part
[-constrset]                   Constraint fileset to use
[-top]                         Specify the top module name
[-include_dirs]                Specify verilog search directories
[-generic]                     Specify generic parameters.
                               Syntax: -generic <name>=<value> -generic
                               <name>=<value>...
[-verilog_define]              Specify verilog defines. Syntax:
                                -verilog_define <macro_name>[=<macro_text>]
                                -verilog_define <macro_name>[=<macro_text>]
[-flatten_hierarchy]            Flatten hierarchy during LUT mapping.
                                Values:
                                 full, none, rebuilt
                                 Default: rebuilt
[-gated_clock_conversion]        Convert clock gating logic to flop enable.
                                 Values: off, on, auto
```

```
                                  Default: off
[-directive]                      Synthesis directive. Values:
                                  default,
                                  AreaOptimized_high,
                                  AreaMutlThresholdDSP
                                  AlternateRoutability
                                  FewerCarryChains
                                  RunTimeOptimized
                                  AreaOptimized_medium
                                  AreaMapLargeShiftRegToBRAM
                                  Default: default
[-retiming]                       This boolean option <on|off> provides an
                                  option improve circuit performance for
                                  intra-clock sequential paths by
                                  automatically moving registers
                                  (register balancing) across combinatorial
                                  gates or LUTs.Default: off
  [-rtl]                          Elaborate and open an rtl design.
  [-bufg]                        Max number of global clock buffers used by
                                  synthesis
                                  Default: 12
[-no_lc]                          Disable LUT combining.
                                  Do not allow combining.Default:off
                                  LUT pairs into single dual output LUTs.
[-fanout_limit]                   Fanout limit. This switch does not impact
                                  control signals (such as set,reset, clock
                                  enable) use MAX_FANOUT to replicate these
                                  signals if needed.
                                  Default: 10000
[-shreg_min_size]                 Minimum length for chain of registers to be
                                  mapped onto SRL
                                  Default: 3
[-mode]                           The design mode. Values: default,
                                  out_of_context
                                  Default: default
[-fsm_extraction]                 FSM Extraction Encoding. Values: off,
                                  one_hot, sequential, johnson, gray, auto
                                  Default: auto
[-keep_equivalent_registers]      Prevents registers sourced by the same
                                  logic from being merged.
                                  (Note that the merging can
                                  otherwise be prevented using the synthesis
                                  KEEP attribute)
[-resource_sharing]               Sharing arithmetic operators. Value: auto,
                                  on, off
                                  Default: auto
[-control_set_opt_threshold]      Threshold for synchronous control set
                                  optimization to lower number
                                  of control sets.
                                  Valid values are 'auto', integer 0 to 16.
                                  The higher the number, the more control set
                                  optimization will be performed and fewer
                                  control sets will result.
```

	To disable control set optimization completely, set to 0. Default: auto
[-max_bram]	Maximum number of block RAM allowed in design. (Note -1 means that the tool will choose the max number allowed for the part in question Default: -1
[-max_uram]	Maximum number of URAM allowed in design. (Note -1 means that the tool will choose the max number allowed for the part in question. Default: -1
[-max_dsp]	Maximum number of block DSP allowed in design. (Note -1 means that the tool will choose the max number allowed for the part Default: -1
[-cascade_dsp]	Controls how adders in sum DSP block outputs are implemented.
[-max_bram_cascade_height]	Controls the maximum number of BRAM that can be cascaded by the tool. The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.
[-max_uram_cascade_height]	Controls the maximum number of URAM that can be cascaded by the tool. The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.
[-assert]	Enable VHDL assert statements to be evaluated. A severity level of failure stops the synthesis flow and produces an error.
[-no_srlextract]	Prevents the extraction of shift registers so that they get implemented as simple registers.
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during execution

For the `-generic` option, special handling needs to happen with VHDL boolean and `std_logic` vector type because those type do not exist in other formats. Instead of `TRUE`, `FAlse`, or 0010, for example, Verilog standards should be given.

For boolean, the value for `FALSE` is as follows:

```
-generic my_gen=1'b0
```

For `std_logic` vector the value for 0010 is:

```
-generic my_gen=4'b0010
```

IMPORTANT: *Overriding string generics or parameters is not supported.*

IMPORTANT: *If you are using the* `-mode out_of_context` *option on the top-level, do not use the* `PACKAGE_PIN` *property unless there is an I/O buffer instantiated in the RTL. The out_of_context option tells the tool to not infer any I/O buffers including tristate buffers. Without the buffer, you will get errors in placer.*

A verbose version of the help is available in the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 3]. To determine any Tcl equivalent to a Vivado IDE action, run the command in the Vivado IDE and review the content in the Tcl Console or the log file.

Multi-Threading in RTL Synthesis

On multiprocessor systems, RTL synthesis leverages multiple CPU cores by default (up to 4) to speed up compile times.

The maximum number of simultaneous threads varies, depending on the number of processors available on the system, the OS, and the stage of the flow (see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 5]). The Tcl parameter `general.maxThreads`, which is common to all threads in Vivado, gives control to the user to specify the number of threads to use when running RTL synthesis. For example:

```
Vivado% set_param general.maxThreads <new limit>
```

Where the `<new limit>` must be an integer from 1 to 8 inclusive. For RTL synthesis, 4 is the maximum number of threads that can be set effectively.

Tcl Script Example

The following is an example `synth_design` Tcl script:

```
# Setup design sources and constraints
read_vhdl -library bftLib [ glob ./Sources/hdl/bftLib/*.vhdl ]
read_vhdl ./Sources/hdl/bft.vhdl
read_verilog [ glob ./Sources/hdl/*.v ]
read_xdc ./Sources/bft_full.xdc
# Run synthesis
synth_design -top bft -part xc7k70tfbg484-2 -flatten_hierarchy rebuilt
# Write design checkpoint
write_checkpoint -force $outputDir/post_synth
# Write report utilization and timing estimates
report_utilization -file utilization.txt
report_timing > timing.txt
```

Setting Constraints

The following table shows the supported Tcl commands for Vivado timing constraints. The commands are linked to more information about the command in the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 3].

Table 1-1: **Supported Synthesis Tcl Commands**

Command Type	Commands			
Timing Constraints	create_clock	create_generated_clock	set_false_path	set_input_delay
	set_output_delay	set_max_delay	set_multicycle_path	get_cells
	set_clock_latency	set_clock_groups	set_disable_timing	get_ports
Object Access	all_clocks	all_inputs	all_outputs	
	get_clocks	get_nets	get_pins	

For details on these commands, see the following documents:

- *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 3]
- *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 9]
- *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 10]
- *Vivado Design Suite Tutorial: Using Constraints* (UG945) [Ref 11]

Vivado Preconfigured Strategies

The following table shows the preconfigured strategies and their respective settings.

Table 1-2: **Vivado Preconfigured Strategies**

Options\Strategies	Default	Flow_Area_ Optimized_High	Flow_ AreaOptimized_ medium	Multi ThresholdDSP	Flow_ AltRoutability	Perf Optimized_ high	Perf ThreshholdCarry	Runtime Optimized
-flatten_hierarchy	rebuilt	rebuilt	rebuilt	rebuilt	rebuilt	rebuilt	rebuilt	none
-gated_clock_conversion	off	off	off	off	off	off	off	off
-bufg	12	12	12	12	12	12	12	12
-fanout_limit	10,000	10,000	400	10,000	10,000	400	10,000	10,000
-directive	Default	AreaOptimized _high	Default	AreaMutl Threshold DSP	Alternate Routability	Default	FewerCarry Chains	RunTime Optimized
-retiming	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked
-fsm_extraction	auto	auto	auto	auto	auto	one_hot	auto	off
-keep_equivalent_registers	unchecked	unchecked	checked	unchecked	checked	checked	checked	unchecked
-resource_sharing	auto	auto	off	auto	auto	off	off	auto
-control_set_opt_ threshold	auto	1	auto	auto	auto	auto	auto	auto
-no_lc	unchecked	unchecked	checked	unchecked	checked	checked	checked	unchecked
-no_srlextract	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked
-shreg_min_size	3	3	5	3	10	5	3	3
-max_bram	-1	-1	-1	-1	-1	-1	-1	-1
-max_uram	-1	-1						
-max_dsp	-1	-1	-1	-1	-1	-1	-1	-1
-max_bram_cascade_height	-1	-1	-1	-1	-1	-1	-1	-1
-max_uram_cascade_height	-1	-1	-1	-1	-1	-1	-1	-1
-cascade_dsp	auto	auto	auto	auto	auto	auto	auto	auto
-assert	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked

Synthesis Attributes

Introduction

In the Vivado® Design Suite, Vivado synthesis is able to synthesize attributes of several types. In most cases, these attributes have the same syntax and the same behavior.

- If Vivado synthesis supports the attribute, it uses the attribute, and creates logic that reflects the used attribute.
- If the specified attribute is not recognized by the tool, the Vivado synthesis passes the attribute and its value to the generated netlist.

It is assumed that a tool later in the flow can use the attribute. For example, the `LOC` constraint is not used by synthesis, but the constraint is used by the Vivado placer, and is forwarded by Vivado synthesis.

Supported Attributes

ASYNC_REG

The `ASYNC_REG` is an attribute that affects many processes in the Vivado tools flow. The purpose of this attribute is to inform the tool that a register is capable of receiving asynchronous data in the D input pin relative to the source clock, or that the register is a synchronizing register within a synchronization chain.

The Vivado synthesis, when encountering this attribute treats it as a `DONT_TOUCH` attribute and pushes the `ASYNC_REG` property forward in the netlist. This process ensures that the object with the `ASYNC_REG` property is not optimized out, and that tools later in the flow receive the property to handle it correctly.

For information on how other Vivado tools handle this attribute, see ASYNC_REG in the *Vivado Design Suite Properties Reference Guide* (UG912) [Ref 12].

You can place this attribute on any register; values are `FALSE` (default) and `TRUE`. This attribute can be set in the RTL or the XDC.

IMPORTANT: *Care should be taken when putting this attribute on loadless signals. The attribute and signal might not be preserved.*

ASYNC_REG Verilog Example

```
(* ASYNC_REG = "TRUE" *) reg [2:0] sync_regs;
```

ASYNC_REG VHDL Example

```
attribute ASYNC_REG : string;
attribute ASYNC_REG of sync_regs : signal is "TRUE";
```

BLACK_BOX

The BLACK_BOX attribute is a useful debugging attribute that can turn a whole level of hierarchy off and enable synthesis to create a black box for that module or entity. When the attribute is found, even if there is valid logic for a module or entity, Vivado synthesis creates a black box for that level. This attribute can be placed on a module, entity, or component. Because this attribute affects the synthesis compiler, it can only be set in the RTL.

BLACK_BOX Verilog Example

```
(* black_box *) module test(in1, in2, clk, out1);
```

IMPORTANT: *In the Verilog example, no value is needed. The presence of the attribute creates the black box.*

BLACK_BOX VHDL Example

```
attribute black_box  : string;
attribute black_box  of beh : architecture is "yes";
```

For more information regarding coding style for Black Boxes, see Black Boxes, page 152.

CASCADE_HEIGHT

The CASCADE_HEIGHT attribute is an integer used to describe the length of the cascade chains of large RAMS that are put into block RAMs. When a RAM that is larger than a single block RAM is described, the Vivado synthesis tool determines how it must be configured.

Often, the tool chooses to cascade the block RAMs that it creates. This attribute can be used to shorten the length of the chain.

This attribute is placed on the RAM in question and can be placed in the RTL files.

A value of 0 or 1 for this attribute effectively turns off any cascading of block RAMs.

CASCADE_HEIGHT Verilog example

```
(* cascade_height = 4 *) reg [31:0] ram [(2**15) - 1:0];
```

CASCADE_HEIGHT VHDL example

```
attribute cascade_height : integer;
attribute cascade_height of ram : signal is 4;
```

CLOCK_BUFFER_TYPE

Apply CLOCK_BUFFER_TYPE on an input clock to describe what type of clock buffer to use.

By default, Vivado synthesis uses BUFGs for clocks buffers.

Supported values are "BUFG", "BUFH", "BUFIO", "BUFMR", "BUFR" or "none".

The CLOCK_BUFFER_TYPE attribute can be placed on any top-level clock port. It can be set only in the RTL. It is not supported in XDC.

CLOCK_BUFFER_TYPE Verilog Example

```
(* clock_buffer_type = "none" *) input clk1;
```

CLOCK_BUFFER_TYPE VHDL Example

```
entity test is port(
  in1 : std_logic_vector (8 downto 0);
  clk : std_logic;
  out1 : std_logic_vector(8 downto 0));
  attribute clock_buffer_type : string;
  attribute clock_buffer_type of clk: signal is "BUFR";
end test;
```

DIRECT_ENABLE

Apply DIRECT_ENABLE on an input port or other signal to have it go directly to the enable line of a flop when there is more than one possible enable or when you want to force the synthesis tool to use the enable lines of the flop.

The DIRECT_ENABLE attribute can be placed on any port or signal.

DIRECT_ENABLE Verilog Example

```
(* direct_enable = "yes" *) input ena3;
```

DIRECT_ENABLE VHDL Example

```
entity test is port(
```

```
  in1 : std_logic_vector (8 downto 0);
  clk : std_logic;
  ena1, ena2, ena3 : in std_logic
  out1 : std_logic_vector(8 downto 0));
  attribute direct_enable : string;
  attribute direct_enable of ena3: signal is "yes";
end test;
```

DIRECT_RESET

Apply DIRECT_RESET on an input port or other signal to have it go directly to the reset line of a flop when there is more than one possible reset or when you want to force the synthesis tool to use the reset lines of the flop.

The DIRECT_RESET attribute can be placed on any port or signal.

DIRECT_RESET Verilog Example

```
(* direct_reset = "yes" *) input rst3;
```

DIRECT_RESET VHDL Example

```
entity test is port(
  in1 : std_logic_vector (8 downto 0);
  clk : std_logic;
  rst1, rst2, rst3 : in std_logic
  out1 : std_logic_vector(8 downto 0));
  attribute direct_reset : string;
  attribute direct_reset of rst3: signal is “yes”;
end test;
```

DONT_TOUCH

Use the DONT_TOUCH attribute in place of KEEP or KEEP_HIERARCHY. The DONT_TOUCH works in the same way as KEEP or KEEP_HIERARCHY attributes; however, unlike KEEP and KEEP_HIERARCHY, DONT_TOUCH is forward-annotated to place and route to prevent logic optimization.

CAUTION! *Like* KEEP *and* KEEP_HIERARCHY*, be careful when using* DONT_TOUCH*. In cases where other attributes are in conflict with* DONT_TOUCH*, the* DONT_TOUCH *attribute takes precedence.*

The values for DONT_TOUCH are TRUE/FALSE or yes/no. You can place this attribute on any signal, module, entity, or component.

Note: The DONT_TOUCH attribute is not supported on the port of a module or entity. If specific ports are needed to be kept, either use the `-flatten_hierarchy none` setting, or put a DONT_TOUCH on the module/entity itself.

RECOMMENDED: *Set this attribute in the RTL only. Signals that need to be kept are often optimized before the XDC file is read. Therefore, setting this attribute in the RTL ensures that the attribute is used.*

DONT_TOUCH Verilog Examples

Verilog Wire Example

```
(* dont_touch = "yes" *) wire sig1;
assign sig1 = in1 & in2;
assign out1 = sig1 & in2;
```

Verilog Module Example

```
(* DONT_TOUCH = "yes" *)
module example_dt_ver
(clk,
In1,
In2,
out1);
```

Verilog Instance Example

```
(* DONT_TOUCH = "yes" *) example_dt_ver U0
(.clk(clk),
.in1(a),
.in2(b),
out1(c));
```

DONT_TOUCH VHDL Examples

VHDL Signal Example

```
signal sig1 : std_logic
attribute dont_touch : string;
attribute dont_touch of sig1 : signal is "true";
....
....
sig1 <= in1 and in2;
out1 <= sig1 and in3;
```

VHDL Entity Example

```
entity example_dt_vhd is
port (
  clk : in std_logic;
  In1 : in std_logic;
  In2 : in std_logic;
  out1 : out std_logic
);
  attribute dont_touch : string;
  attribute dont_touch of example_dt_vhd : entity is "true|yes";
end example_dt_vhd;
```

VHDL Component Example

```
entity rtl of test is
attribute dont_touch : string;
  component my_comp
  port (
  in1 : in std_logic;
  out1 : out std_logic);
end component;
attribute dont_touch of my_comp : component is "yes";
```

VHDL Example on Architecture

```
entity rtl of test is
attribute dont_touch : string;
attribute dont_touch of rtl : architecture is "yes";
```

EXTRACT_ENABLE

EXTRACT_ENABLE controls whether registers infer enables. Typically, the Vivado tools extract or not extract enables based on heuristics that typically benefit the most amount of designs. In cases where Vivado is not behaving in a desired way, this attribute overrides the default behavior of the tool. If there is an undesired enable going to the CE pin of the Flip-Flop, this attribute can force it to the D input logic. Conversely, if the tool is not inferring an enable that is specified in the RTL, this attribute can tell the tool to move that enable to the CE pin of the flop.

EXTRACT_ENABLE is placed on the registers and is supported in the RTL code only. It can take boolean values of: "yes" and "no".

EXTRACT_ENABLE Verilog Example

```
(* extract_enable = "yes" *) reg  my_reg;
```

EXTRACT_ENABLE VHDL Example

```
signal my_reg : std_logic;
attribute extract_enable : string;
attribute extract_enable of my_reg: signal is "no";
```

EXTRACT_RESET

EXTRACT_RESET controls if registers infer resets. Typically, the Vivado tools extract or not extract resets based on heuristics that typically benefit the most amount of designs. In cases where Vivado is not behaving in a desired way, this attribute overrides the default behavior of the tool. If there is an undesired synchronous reset going to the Flip-Flop, this attribute can force it to the D input logic.

Conversely, if the tool is not inferring a reset that is specified in the RTL, this attribute can tell the tool to move that reset to the dedicated reset of the flop.

Note: This attribute can only be used with synchronous resets; asynchronous resets are not supported with this attribute.

EXTRACT_RESET is placed on the registers, and supported in the RTL code only. It can take the boolean values: "yes" or "no".

EXTRACT_RESET Verilog Example

```
(* extract_reset = "yes" *) reg my_reg;
```

EXTRACT_RESET VHDL Example

```
signal my_reg : std_logic;
attribute extract_reset : string;
attribute extract_reset of my_reg: signal is "no";
```

FSM_ENCODING

FSM_ENCODING controls encoding on the state machine. Typically, the Vivado tools choose an encoding protocol for state machines based on heuristics that do the best for the most designs. Certain design types work better with a specific encoding protocol.

FSM_ENCODING can be placed on the state machine registers. The legal values for this are "one_hot", "sequential", "johnson", "gray", "auto", and "none". The "auto" value is the default, and allows the tool to determine best encoding. This attribute can be set in the RTL or the XDC.

FSM_ENCODING Verilog Example

```
(* fsm_encoding = "one_hot" *) reg [7:0] my_state;
```

FSM_ENCODING VHDL Example

```
type count_state is (zero, one, two, three, four, five, six, seven);
signal my_state : count_state;
attribute fsm_encoding : string;
attribute fsm_encoding of my_state : signal is "sequential";
```

FSM_SAFE_STATE

FSM_SAFE_STATE instructs Vivado synthesis to insert logic into the state machine that detects there is an illegal state, then puts it into a known, good state on the next clock cycle.

For example, if there were a state machine with a "one_hot" encode, and that is in a "0101" state (which is an illegal for "one_hot"), the state machine would be able to recover. Place the FSM_SAFE_STATE attribute on the state machine registers. You can set this attribute in either the RTL or in the XDC.

The legal values for FSM_SAFE_STATE are:

- "auto": Uses Hamming-3 encoding for auto-correction for one bit/flip.
- "reset_state": Forces the state machine into the reset state using Hamming-2 encoding detection for one bit/flip.
- "power_on_state": Forces the state machine into the power-on state using Hamming-2 encoding detection for one bit/flip.
- "default_state": Forces the state machine into the default state specified in RTL: the state that is specified in "default" branch of the case statement in Verilog or the state specified in the others branch of the case statement in VHDL; even if that state is unreachable, using Hamming-2 encoding detection for one bit/flip.

FSM_SAFE_STATE Verilog Example

```
(* fsm_safe_state = "reset_state" *) reg [7:0] my_state;
```

FSM_SAFE_STATE VHDL Example

```
type count_state is (zero, one, two, three, four, five, six, seven);
  signal my_state : count_state;
  attribute fsm_safe_state : string;
  attribute fsm_safe_state of my_state : signal is "power_on_state";
```

FULL_CASE (Verilog Only)

FULL_CASE indicates that all possible case values are specified in a case, casex, or casez statement. If case values are specified, extra logic for case values is not created by Vivado synthesis. This attribute is placed on the case statement.

IMPORTANT: *Because this attribute affects the compiler and can change the logical behavior of the design, it can be set in the RTL only.*

FULL_CASE Example (Verilog)

```
(* full_case *)
  case select
  3'b100 : sig = val1;
  3'b010 : sig = val2;
  3'b001 : sig = val3;
endcase
```

GATED_CLOCK

Vivado synthesis allows the conversion of gated clocks. To perform this conversion, use:

- A switch in the Vivado GUI that instructs the tool to attempt the conversion.
- The RTL attribute that instructs the tool about which signal in the gated logic is the clock.

  Place this attribute on the signal or port that is the clock.

To control the switch:

1. Select **Flow Navigator > Synthesis Settings.**
2. In the Options area, set the `-gated_clock_conversion` option to one of the following values:
   - `off`: Disables the gated clock conversion.
   - `on`: Gated clock conversion occurs if the `gated_clock` attribute is set in the RTL code. This option gives you more control of the outcome.
   - `auto`: Gated clock conversion occurs if either of the following events are `TRUE`:
     - the `gated_clock` attribute is set to `TRUE`.
     - the Vivado synthesis can detect the gate and there is a valid clock constraint set. This option lets the tool make decisions.

GATED_CLOCK Example (Verilog)

```
(* gated_clock = "true" *) input clk;
```

GATED_CLOCK Example (VHDL)

```
entity test is port (
  in1, in2 : in std_logic_vector(9 downto 0);
  en : in std_logic;
  clk : in std_logic;
  out1 : out std_logic_vector( 9 downto 0));
  attribute gated_clock : string;
  attribute gated_clock of clk : signal is "true";
end test;
```

IOB

The IOB is not a synthesis attribute; it is used downstream by Vivado implementation. This attribute indicates if a register should go into the I/O buffer. The values are TRUE or FALSE. Place this attribute on the register that you want in the I/O buffer. This attribute can be set in the RTL or the XDC.

IOB Verilog Example

```
(* IOB = "true" *) reg sig1;
```

IOB VHDL Example

```
signal sig1:std_logic;
attribute IOB: string;
attribute IOB of sig1 : signal is "true";
```

IO_BUFFER_TYPE

Apply the IO_BUFFER_TYPE attribute on any top-level port to instruct the tool to use buffers. Add the property with a value of "NONE" to disable the automatic inference of buffers on the input or output buffers, which is the default behavior of Vivado synthesis. It can be set in the RTL and the XDC.

IO_BUFFER_TYPE Verilog Example

```
(* io_buffer_type = "none" *) input in1;
```

IO_BUFFER_TYPE VHDL Example

```
entity test is port(
  in1 : std_logic_vector (8 downto 0);
  clk : std_logic;
  out1 : std_logic_vector(8 downto 0));
  attribute io_buffer_type : string;
  attribute io_buffer_type of out1: signal is "none";
end test;
```

KEEP

Use the KEEP attribute to prevent optimizations where signals are either optimized or absorbed into logic blocks. This attribute instructs the synthesis tool to keep the signal it was placed on, and that signal is placed in the netlist.

For example, if a signal is an output of a 2 bit AND gate, and it drives another AND gate, the KEEP attribute can be used to prevent that signal from being merged into a larger LUT that encompasses both AND gates.

KEEP is also commonly used in conjunction with timing constraints. If there is a timing constraint on a signal that would normally be optimized, KEEP prevents that and allows the correct timing rules to be used.

Note: The KEEP attribute is not supported on the port of a module or entity. If you need to keep specific ports, either use the `-flatten_hierarchy none` setting, or put a DONT_TOUCH on the module or entity itself.

CAUTION! *Take care with the KEEP attribute on signals that are not used in the RTL later. Synthesis keeps those signals, but they do not drive anything. This could cause issues later in the flow.*

CAUTION! *Be careful when using KEEP with other attributes. In cases where other attributes are in conflict with KEEP, the KEEP attribute usually takes precedence.*

Examples are:

- When you have a MAX_FANOUT attribute on one signal and a KEEP attribute on a second signal that is driven by the first; the KEEP attribute on the second signal would not allow fanout replication.
- With a `RAM STYLE="block"`, when there is a KEEP on the register that would need to become part of the RAM, the KEEP attribute prevents the block RAM from being inferred.

The supported KEEP values are:

- TRUE: Keeps the signal.
- FALSE: Allows the Vivado synthesis to optimize, if the tool makes that determination. The FALSE value does not force the tool to remove the signal. The default value is FALSE.

You can place this attribute on any signal, register, or wire.

RECOMMENDED: *Set this attribute in the RTL only. Because signals that need to be kept are often optimized before the XDC file is read, setting this attribute in the RTL ensures that the attribute is used.*

Note: The KEEP attribute does not force the place and route to keep the signal. Instead, this is accomplished using the DONT_TOUCH attribute.

KEEP Example (Verilog)

```
(* keep = "true" *) wire sig1;
assign sig1 = in1 & in2;
assign out1 = sig1 & in2;
```

KEEP Example (VHDL)

```
signal sig1 : std_logic;
attribute keep : string;
attribute keep of sig1 : signal is "true";
....
....
sig1 <= in1 and in2;
out1 <= sig1 and in3;
```

KEEP_HIERARCHY

KEEP_HIERARCHY is used to prevent optimizations along the hierarchy boundaries. The Vivado synthesis tool attempts to keep the same general hierarchies specified in the RTL, but for QoR reasons it can flatten or modify them.

If KEEP_HIERARCHY is placed on the instance, the synthesis tool keeps the boundary on that level static.

This can affect QoR and also should not be used on modules that describe the control logic of 3-state outputs and I/O buffers. The KEEP_HIERARCHY can be placed in the module or architecture level or the instance. This attribute can only be set in the RTL.

KEEP_HIERARCHY Example (Verilog)

On Module:

```
(* keep_hierarchy = "yes" *) module bottom (in1, in2, in3, in4, out1, out2);
```

On Instance:

```
(* keep_hierarchy = "yes" *)bottom u0 (.in1(in1), .in2(in2), .out1(temp1));
```

KEEP_HIERARCHY Example (VHDL)

On Architecture:

```
attribute keep_hierarchy : string;
attribute keep_hierarchy of beh : entity is "yes";
```

On Instance:

```
attribute keep_hierarchy : string;
attribute keep_hierarchy of u0 : label is "yes";
```

MARK_DEBUG

MARK_DEBUG specifies that a net should be debugged using the Vivado hardware manager. This can prevent optimization that might have otherwise occurred to that signal. However, it provides an easy means to later observe the values on this signal during FPGA operation.

This attribute is applicable to net objects (get_nets): any net accessible to the internal array.

Note: Some nets can have dedicated connectivity or other aspects that prohibit visibility for debug purposes.

The MARK_DEBUG values are: "TRUE" or "FALSE".

Syntax

Verilog Syntax

To set this attribute, place the proper Verilog attribute syntax before the top-level output port declaration:

```
(* MARK_DEBUG = "{TRUE|FALSE}" *)
```

Verilog Syntax Example

```
// Marks an internal wire for debug
(* MARK_DEBUG = "TRUE" *) wire debug_wire,
```

VHDL Syntax

To set this attribute, place the proper VHDL attribute syntax before the top-level output port declaration.

Declare the VHDL attribute as follows:

```
attribute MARK_DEBUG : string;
```

Specify the VHDL attribute as follows:

```
attribute MARK_DEBUG of signal_name : signal is "{TRUE|FALSE}";
```

Where signal_name is an internal signal.

VHDL Syntax Example

```
signal debug_wire : std_logic;
attribute MARK_DEBUG : string;
-- Marks an internal wire for debug
attribute MARK_DEBUG of debug_wire : signal is "TRUE";
```

XDC Syntax

```
set_property MARK_DEBUG value [get_nets <net_name>]
```

Where <net_name> is a signal name.

XDC Syntax Example

```
# Marks an internal wire for debug
set_property MARK_DEBUG TRUE [get_nets debug_wire]
```

RECOMMENDED: *Often, the use of* MARK_DEBUG *is on pins of hierarchies, and can be used on any elaborated sequential element, such as* RTL_REG. MARK_DEBUG *attributes are intended go on nets, it is recommended that you use both the* get_nets *and the* get_pins *command as shown, such as:* set_property MARK_DEBUG true [get_nets -of [get_pins\ hier1/hier2/<flop_name>/Q]]. *This recommended use ensures that the* MARK_DEBUG *goes onto the net connected to that pin, regardless of its name.*

MAX_FANOUT

MAX_FANOUT instructs Vivado synthesis on the fanout limits for registers and signals. You can specify this either in RTL or as an input to the project. The value is an integer.

This attribute only works on registers and combinatorial signals. To achieve the fanout, it replicates the register or the driver that drives the combinatorial signal. This attribute can be set in the RTL or the XDC.

MAX_FANOUT overrides the default value of the synthesis global option -fanout_limit. You can set that overall design default limit for a design through **Project Settings > Synthesis** or using the -fanout_limit command line option in synth_design.

The MAX_FANOUT attribute is enforced whereas the -fanout_limit constitutes only a guideline for the tool, not a strict command. When strict fanout control is required, use MAX_FANOUT. Also, unlike the -fanout_limit switch, MAX_FANOUT can impact control signals. The -fanout_limit switch does not impact control signals (such as set, reset, clock enable), use MAX_FANOUT to replicate these signals if needed.

Note: Inputs, black boxes, EDIF (EDF), and Native Generic Circuit (NGC) files are not supported.

IMPORTANT: *NGC format files are not supported in the Vivado Design Suite for UltraScale devices. It is recommended that you regenerate the IP using the Vivado Design Suite IP customization tools with native output products. Alternatively, you can use the* NGC2EDIF *command to migrate the NGC file to EDIF format for importing. However, Xilinx recommends using native Vivado IP rather than XST-generated NGC format files going forward.*

MAX_FANOUT Example (Verilog)

On Signal:

```
(* max_fanout = 50 *) reg sig1;
```

MAX_FANOUT Example (VHDL)

```
signal sig1 : std_logic;
attribute max_fanout : integer;
attribute max_fanout of sig1 : signal is 50;
```

Note: In VHDL, `max_fanout` is an `integer`.

PARALLEL_CASE (Verilog Only)

`PARALLEL_CASE` specifies that the case statement must be built as a parallel structure. Logic is not created for an `if -elsif` structure. Because this attribute affects the compiler and the logical behavior of the design, it can be set in the RTL only.

```
(* parallel_case *)  case select
  3'b100 : sig = val1;
  3'b010 : sig = val2;
  3'b001 : sig = val3;
endcase
```

IMPORTANT: *This attribute can only be controlled through the Verilog RTL.*

RAM_STYLE

`RAM_STYLE` instructs the Vivado synthesis tool on how to infer memory. Accepted values are:

- `block`: Instructs the tool to infer `RAMB` type components.
- `distributed`: Instructs the tool to infer the LUT RAMs.
- `register`: Instructs the tool to infer registers instead of RAMs.
- `ultra`: Instructs the tool to use the Zynq UltraScale+™ URAM primitives.

By default, the tool selects which RAM to infer, based upon heuristics that give the best results for most designs. Place this attribute on the array that is declared for the RAM or a level of hierarchy.

- If set on a signal, the attribute will affect that specific signal.
- If set on a level of hierarchy, this is affects all the RAMS in that level of hierarchy. Sub-levels of hierarchy are not affected.

This can be set in the RTL or the XDC.

RAM_STYLE Example (Verilog)

```
(* ram_style = "distributed" *) reg [data_size-1:0] myram [2**addr_size-1:0];
```

RAM_STYLE Example (VHDL)

```
attribute ram_style : string;
attribute ram_style of myram : signal is "distributed";
```

For more information about RAM coding styles, see RAM HDL Coding Techniques.

ROM_STYLE

ROM_STYLE instructs the synthesis tool how to infer ROM memory. Accepted values are:

- block: Instructs the tool to infer RAMB type components
- distributed: Instructs the tool to infer the LUT ROMs. By default, the tool selects which ROM to infer based on heuristics that give the best results for the most designs.

This can be set in the RTL and the XDC.

ROM_STYLE Example (Verilog)

```
(* rom_style = "distributed" *) reg [data_size-1:0] myrom [2**addr_size-1:0];
```

ROM_STYLE Example (VHDL)

```
attribute rom_style : string;
attribute rom_style of myrom : signal is "distributed";
```

For information about coding for ROM, see ROM HDL Coding Techniques.

SHREG_EXTRACT

SHREG_EXTRACT instructs the synthesis tool on whether to infer SRL structures. Accepted values are:

- YES: The tool infers SRL structures.
- NO: The does not infer SRLs and instead creates registers.

Place SHREG_EXTRACT on the signal declared for SRL or the module/entity with the SRL. It can be set in the RTL or the XDC.

SHREG_EXTRACT Example (Verilog)

```
(* shreg_extract = "no" *) reg [16:0] my_srl;
```

SHREG_EXTRACT Example (VHDL)

```
attribute shreg_extract : string;
attribute shreg_extract of my_srl : signal is "no";
```

SRL_STYLE

SRL_STYLE tells the synthesis tool how to infer SRLs that are found in the design. Accepted values are:

- register: The tool does not infer an SRL, but instead only uses registers.
- srl: The tool infers an SRL without any registers before or after.
- srl_reg: The tool infers an SRL and leaves one register after the SRL.
- reg_srl: The tool infers an SRL and leaves one register before the SRL.
- reg_srl_reg: The tool infers an SRL and leaves one register before and one register after the SRL.

Place SRL_STYLE on the signal declared for SRL. This attribute can be set in RTL only. It is not supported in the XDC.

In addition, this attribute can only be used on static SRLs. The indexing logic for dynamic SRLs is located within the SRL component itself. Therefore, the logic cannot be created around the SRL component to look up addresses outside of the component.

CAUTION! *Use care when using the* SRL_STYLE *attribute with the* SHREG_EXTRACT *attribute or the* shreg_min_size *command line switch. Both take priority over the* SRL_STYLE *attribute. For example, if* SHREG_EXTRACT *is set to* NO*, and* SRL_STYLE *is set to* srl_reg*, the* SHREG_EXTRACT *takes precedence, and only registers are used.*

SRL_STYLE Verilog Examples

```
(* srl_style = "register" *) reg [16:0] my_srl;
```

SRL_STYLE VHDL Examples

```
attribute srl_style : string;
attribute srl_style of my_srl : signal is "reg_srl_reg";
```

TRANSLATE_OFF/TRANSLATE_ON

TRANSLATE_OFF and TRANSLATE_ON instruct the Synthesis tool to ignore blocks of code. These attributes are given within a comment in RTL. The comment should start with one of the following keywords:

- synthesis
- synopsys
- pragma

TRANSLATE_OFF starts the ignore, and it ends with TRANSLATE_ON. These commands cannot be nested.

This attribute can only be set in the RTL.

TRANSLATE_OFF/TRANSLATE_ON Verilog Example

```
// synthesis translate_off
Code....
// synthesis translate_on
```

TRANSLATE_OFF/TRANSLATE_ON VHDL Example

```
-- synthesis translate_off
Code...
-- synthesis translate_on
```

CAUTION! *Be careful with the types of code that are included between the translate statements. If it is code that affects the behavior of the design, a simulator could use that code, and create a simulation mismatch.*

USE_DSP48

USE_DSP48 instructs the synthesis tool how to deal with synthesis arithmetic structures. By default, unless there are timing concerns or threshold limits, synthesis attempts to infer mults, mult-add, mult-sub, and mult-accumulate type structures into DSP48 blocks.

Adders, subtractors, and accumulators can go into these blocks also, but by default are implemented with the logic instead of with DSP48 blocks. The USE_DSP48 attribute overrides the default behavior and force these structures into DSP48 blocks.

Accepted values are yes and no. This attribute can be placed in the RTL on signals, architectures and components, entities and modules. The priority is as follows:

1. Signals
2. Architectures and components
3. Modules and entities

If the attribute is not specified, the default behavior is for Vivado synthesis to determine the correct behavior. This attribute can be set in the RTL or the XDC.

USE_DSP48 Example (Verilog)

```
(* use_dsp48 = "yes" *) module test(clk, in1, in2, out1);
```

USE_DSP48 Example (VHDL)

```
attribute use_dsp48 : string;
attribute use_dsp48 of P_reg : signal is "no"
```

Custom Attribute Support in Vivado

Vivado synthesis supports the use of *custom attributes* in RTL. A custom attribute is an attribute who's behavior synthesis is not already aware. Often, custom attributes are intended for use in other tools downstream from the synthesis process.

CAUTION! *When Vivado synthesis encounters unknown attributes, it attempts forward those attributes; but the designer needs to understand the risk. A custom attribute does not stop synthesis optimizations from occurring, which means that if synthesis can optimize an item with a custom attribute, it does so, and the attribute is lost.*

If you need custom attributes go through synthesis, you must use the DONT_TOUCH or KEEP_HIERARCHY attributes to prevent synthesis from optimizing the objects that need the attributes.

There are two types of objects that can have custom attributes: hierarchies and signals.

When using custom attributes on hierarchies, the `-flatten_hierarchy` switch must be set to `none` or a `KEEP_HIERARCHY` placed on that level, because synthesis by default flattens the design, optimizes the design, and then rebuilds the design.

After a design if first flattened, the custom attribute on the hierarchy is lost.

Example with Custom Attribute on Hierarchy (Verilog)

```
(* my_att = "my_value", DONT_TOUCH = "yes" *) module test(....
```

Example with Custom Attribute on Hierarchy (VHDL)

```
attribute my_att : string;
attribute my_att of beh : architecture is "my_value"
attribute DONT_TOUCH : string;
attribute DONT_TOUCH of beh : architecture is "yes";
```

Care should be taken when using custom attributes on signals as well. When a custom attribute is seen on a signal, the synthesis tool attempts to put that attribute on the item; however this item could be translated to a register or a net depending on how the tool evaluates the RTL code. Also, as with hierarchies, just because a signal has a custom attribute, the tool can perform optimizations on that signal, and the attribute will be lost. To retain custom attribute on signals with custom attributes you must place the `DONT_TOUCH` or the `KEEP` attribute on those signals.

Finally, because a signal in RTL could describe both a register and the net coming out of the register, the synthesis tool checks any items with both custom attributes and the `DONT_TOUCH` attribute. If the net in question is driven by a register, synthesis copies that custom attribute to the register and the net, because there are multiple ways of using custom attributes, and sometimes the attribute is wanted on the register and sometimes the net.

Example with Custom Attribute on a Signal (Verilog)

```
(* my_att = "my_value", DONT_TOUCH = "yes" *) reg my_signal;
```

Example with Custom Attribute on a Signal (VHDL)

```
attribute my_att : string;
attribute my_att of my_signal : signal is "my_value";
attribute DONT_TOUCH : string;
attribute DONT_TOUCH of my_signal : signal is "yes";
```

Using Synthesis Attributes in XDC files

Some synthesis attributes can also be set from an XDC file as well as the original RTL file. In general, attributes that are used in the end stages of synthesis and describe how synthesis-created logic is allowed in the XDC file. Attributes that are used towards the beginning of synthesis and affect the compiler are not allowed in the XDC.

For example, the `KEEP` and `DONT_TOUCH` attributes are not allowed in the XDC. This is because, at the time the attribute is read from the XDC file, components that have the `KEEP` or `DONT_TOUCH` attribute might have already been optimized and would therefore not exist at the time the attribute is read. For that reason, those attributes must always be set in the RTL code. For more information on where to set specific attributes, see the individual attribute descriptions in this chapter.

To specify synthesis attributes in XDC, type the following in the Tcl Console:

```
set_property <attribute> <value> <target>
```

For example:

```
set_property MAX_FANOUT 15 [get_cells in1_int_reg]
```

In addition, you can set these attributes in the elaborated design, as follows:

1. Open the elaborated design, shown in Figure 2-1, and select the item on which to place an attribute, using either of the following methods:
   - Click the item in the schematic.
   - Select the item in the RTL Netlist view.

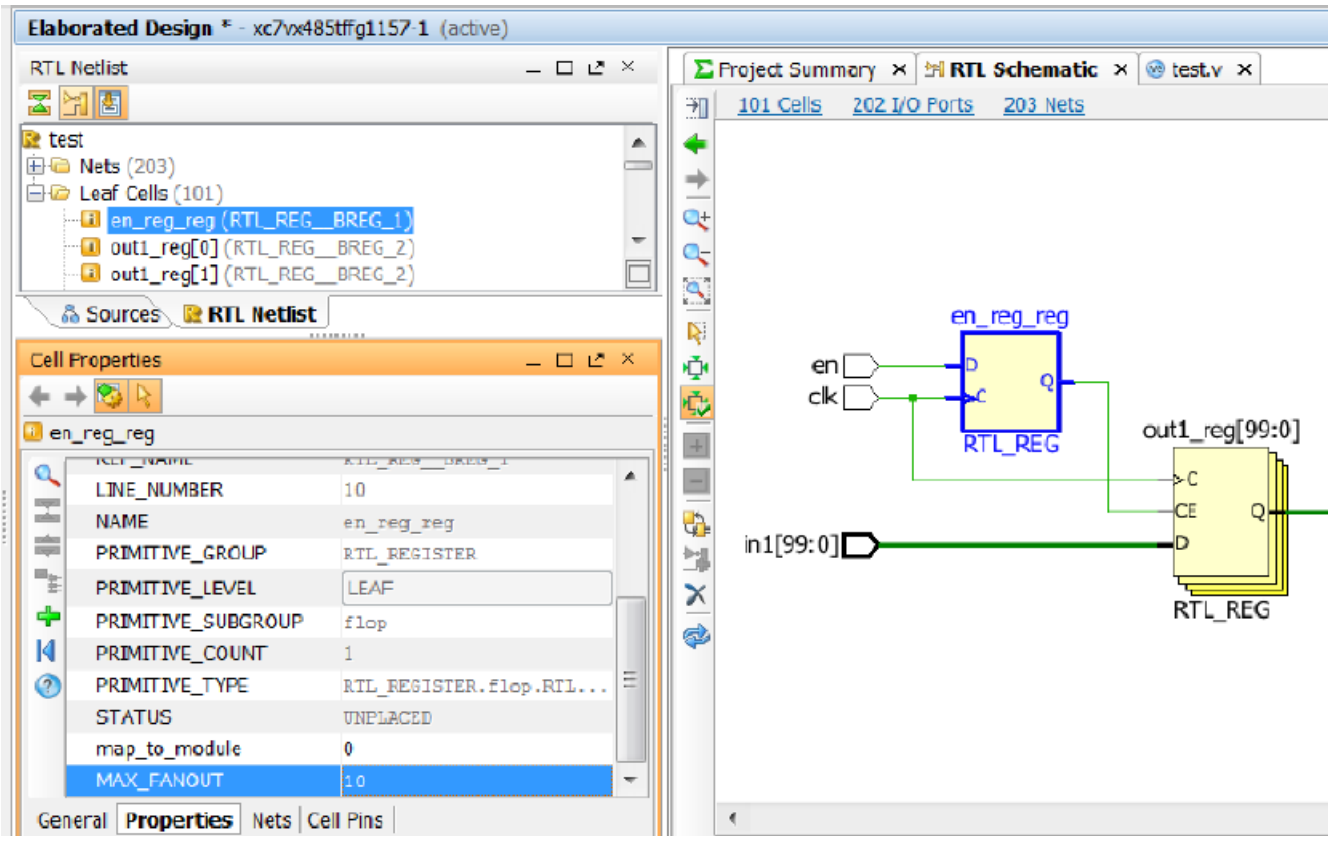

Figure 2-1: **Adding an XDC Property from the Elaborated Design View**

2. In the Cell Properties window, click the Properties tab, and do one of the following:
   - Modify the property.
   - If the property does not exist, right-click, select **Add Properties**, and select the property from the window that appears, or click the **+** sign.
3. After the properties are set, select **File > Save Constraints**.

   This saves the attributes to your current constraint file or creates a new constraint file if one does not exist.

HDL Coding Techniques

Introduction

Hardware Description Language (HDL) coding techniques let you:

- Describe the most common functionality found in digital logic circuits.
- Take advantage of the architectural features of Xilinx® devices.
- Templates are available from the Vivado® Integrated Design Environment (IDE). To access the templates, in the Window Menu, select **Language Templates**.

Coding examples are included in this chapter. Download the coding example files from:

Coding Examples

Advantages of VHDL

- Enforces stricter rules, in particular strongly typed, less permissive and error-prone
- Initialization of RAM components in the HDL source code is easier (Verilog initial blocks are less convenient)
- Package support
- Custom types
- Enumerated types
- No `reg` versus `wire` confusion

Advantages of Verilog

- C-like syntax
- More compact code
- Block commenting
- No heavy component instantiation as in VHDL

Advantages of SystemVerilog

- More compact code compared to Verilog
- Structures and enumerated types for better scalability
- Interfaces for higher level of abstraction
- Supported in Vivado synthesis

Flip-Flops, Registers, and Latches

Vivado synthesis recognizes Flip-Flops, Registers with the following control signals:

- Rising or falling-edge clocks
- Asynchronous Set/Reset
- Synchronous Set/Reset
- Clock Enable

Flip-Flops, Registers and Latches are described with:

- sequential process (VHDL)
- `always` block (Verilog)
- `always_ff` for flip-flops, `always_latch` for Latches (SystemVerilog)

The `process` or `always` block sensitivity list should list:

- The clock signal
- All asynchronous control signals

Flip-Flops and Registers Control Signals

Flip-Flops and Registers control signals include:

- Clocks
- Asynchronous and synchronous set and reset signals
- Clock enable

Coding Guidelines

- Do not asynchronously set or reset registers.
  - Control set remapping becomes impossible.
  - Sequential functionality in device resources such as block RAM components and DSP blocks can be set or reset synchronously only.
  - If you use asynchronously set or reset registers, you cannot leverage device resources, or those resources are configured sub-optimally.
- Do not describe flip-flops with both a set and a reset.
  - No Flip-flop primitives feature both a set and a reset, whether synchronous or asynchronous.
  - Flip-flop primitives featuring both a set and a reset may adversely affect area and performance.
- Avoid operational set/reset logic whenever possible. There may be other, less expensive, ways to achieve the desired effect, such as taking advantage of the circuit global reset by defining an initial content.
- Always describe the clock enable, set, and reset control inputs of flip-flop primitives as active-High. If they are described as active-Low, the resulting inverter logic will penalize circuit performance.

Flip-Flops and Registers Inference

Vivado synthesis infers four types of register primitives depending on how the HDL code is written:

- `FDCE`: D flip-flop with Clock Enable and Asynchronous Clear
- `FDPE`: D flip-flop with Clock Enable and Asynchronous Preset
- `FDSE`: D flip-flop with Clock Enable and Synchronous Set
- `FDRE`: D flip-flop with Clock Enable and Synchronous Reset

Flip-Flops and Registers Initialization

To initialize the content of a Register at circuit power-up, specify a default value for the signal during declaration.

Flip-Flops and Registers Reporting

- Registers are inferred and reported during HDL synthesis.
- The number of Registers inferred during HDL synthesis might not precisely equal the number of Flip-Flop primitives in the Design Summary section.
- The number of Flip-Flop primitives depends on the following processes:
  - Absorption of Registers into DSP blocks or block RAM components
  - Register duplication
  - Removal of constant or equivalent Flip-Flops

Flip-Flops and Registers Reporting Example

```
---------------------------------------------------------------------------------
RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Registers :
                8 Bit    Registers := 1

Report Cell Usage:
-----+----+-----
     |Cell|Count
-----+----+-----
3    |FDCE|    8
-----+----+-----
```

Flip-Flops and Registers Coding Examples

The following subsections provide VHDL and Verilog examples of coding for Flip-Flops and registers. Download the coding example files from: Coding Examples.

Register with Rising-Edge Coding Example (Verilog)

```
// 8-bit Register with
//Rising-edge Clock
//Active-high Synchronous Clear
//Active-high Clock Enable
// File: registers_1.v

module registers_1(d_in,ce,clk,clr,dout);
input [7:0] d_in;
input ce;
input clk;
```

```
input clr;
output [7:0] dout;
reg [7:0] d_reg;

always @ (posedge clk)
begin
 if(clr)
  d_reg <= 8'b0;
 else if(ce)
  d_reg <= d_in;
end

assigndout = d_reg;
endmodule
```

Flip-Flop Registers with Rising-Edge Clock Coding Example (VHDL)

```
-- Flip-Flop with
--Rising-edge Clock
--Active-high Synchronous Clear
--Active-high Clock Enable
-- File: registers_1.vhd

library IEEE;
use IEEE.std_logic_1164.all;

entity registers_1 is
 port(
  clr, ce, clk : in  std_logic;
  d_in         : in  std_logic_vector(7 downto 0);
  dout         : out std_logic_vector(7 downto 0)
 );
end entity registers_1;
architecture rtl of registers_1 is
begin
 process(clk) is
 begin
  if rising_edge(clk) then
    if clr = '1' then
    dout <= "00000000";
   elsif ce = '1' then
     dout <= d_in;
   end if;
  end if;
 end process;
end architecture rtl;
```

Latches

The Vivado log file reports the type and size of recognized Latches.

Inferred Latches are often the result of HDL coding mistakes, such as incomplete if or case statements.

Vivado synthesis issues a warning for the instance shown in the following reporting example. This warning lets you verify that the inferred Latch functionality was intended.

Latches Reporting Example

```
=========================================================================
*                     Vivado.log                                        *
=========================================================================

WARNING: [Synth 8-327] inferring latch for variable 'Q_reg'

========================================================================= Report
Cell Usage:
-----+----+-----
     |Cell|Count
-----+----+-----
2    |LD  |    1
-----+----+-----
=========================================================================
```

Latch With Positive Gate and Asynchronous Reset Coding Example (Verilog)

```
// Latch with Positive Gate and Asynchronous Reset
// File: latches.v
module latches (
                input G,
                input D,
                input CLR,
                output reg Q
               );
always @ *
begin
 if(CLR)
  Q = 0;
 else if(G)
  Q = D;
end

endmodule
```

Latch With Positive Gate and Asynchronous Reset Coding Example (VHDL)

```
-- Latch with Positive Gate and Asynchronous Reset
-- File: latches.vhd
library ieee;
use ieee.std_logic_1164.all;

entity latches is
 port(
  G, D, CLR : in  std_logic;
  Q         : out std_logic
 );
end latches;

architecture archi of latches is
begin
 process(CLR, D, G)
 begin
  if (CLR = '1') then
   Q <= '0';
  elsif (G = '1') then
   Q <= D;
  end if;
 end process;
end archi;
```

Tristates

- Tristate buffers are usually modeled by a signal or an `if-else` construct.
- This applies whether the buffer drives an internal bus, or an external bus on the board on which the device resides
- The signal is assigned a high impedance value in one branch of the `if-else`.

Download the coding example files from: Coding Examples.

Tristate Implementation

Inferred Tristate buffers are implemented with different device primitives when driving an:

- Internal bus (`BUFT`)
  - A `BUFT` inferred is converted automatically to a logic realized in LUTs by Vivado synthesis

- When an internal Bus inferring a BUFT is driving an output of the top module Vivado synthesis is inferring an OBUF

- External pin of the circuit (OBUFT)

Tristate Reporting Example

Tristate buffers are inferred and reported during synthesis.

```
================================================================================
*                          Vivado log file                                     *
================================================================================
Report Cell Usage:
-----+-----+-----
     |Cell |Count
-----+-----+-----
1    |OBUFT|    1
-----+-----+-----
================================================================================
```

Tristate Description Using Concurrent Assignment Coding Example (Verilog)

```
// Tristate Description Using Concurrent Assignment
// File: tristates_2.v
//
module tristates_2 (T, I, O);
input  T, I;
output O;

assign O = (~T) ? I: 1'bZ;

endmodule
```

Tristate Description using Combinatorial Process Implemented with OBUF (VHDL)

```
-- Tristate Description Using Combinatorial Process
-- Implemented with an OBUF (internal buffer)
--
-- File: tristates_3.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity tristates_3 is
 generic(
  WIDTH : integer := 8
```

```
  );
  port(
    T : in  std_logic;
    I : in  std_logic_vector(WIDTH - 1 downto 0);
    O : out std_logic_vector(WIDTH - 1 downto 0)
  );

end tristates_3;

architecture archi of tristates_3 is
  signal S : std_logic_vector(WIDTH - 1 downto 0);

begin
  process(I, T)
  begin
    if (T = '1') then
      S <= I;
    else
      S <= (others => 'Z');
    end if;
  end process;

  O <= not (S);

end archi;
```

Tristate Description Using Combinatorial Process Implemented with OBUFT Coding Example (VHDL)

```
-- Tristate Description Using Combinatorial Process
-- Implemented with an OBUFT (IO buffer)
-- File: tristates_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity tristates_1 is
  port(
    T : in  std_logic;
    I : in  std_logic;
    O : out std_logic
  );
end tristates_1;

architecture archi of tristates_1 is
begin
  process(I, T)
  begin
```

```
    if (T = '0') then
      O <= I;
    else
      O <= 'Z';
    end if;
  end process;

end archi;
```

Tristate Description Using Combinatorial Always Block Coding Example (Verilog)

```
// Tristate Description Using Combinatorial Always Block
// File: tristates_1.v
//
module tristates_1 (T, I, O);
input  T, I;
output O;
reg    O;

always @(T or I)
begin
  if (~T)
    O = I;
  else
   O = 1'bZ;
end

endmodule
```

Shift Registers

A Shift Register is a chain of Flip-Flops allowing propagation of data across a fixed (static) number of latency stages. In contrast, in Dynamic Shift Registers, the length of the propagation chain varies dynamically during circuit operation.

Download the coding example files from: Coding Examples.

Static Shift Register Elements

A static Shift Register usually involves:

- A clock
- An optional clock enable
- A serial data input
- A serial data output

Shift Registers SRL-Based Implementation

Vivado synthesis implements inferred Shift Registers on SRL-type resources such as:

- `SRL16E`
- `SRLC32E`

Depending on the length of the Shift Register, Vivado synthesis does one of the following:

- Implements it on a single SRL-type primitive
- Takes advantage of the cascading capability of SRLC-type primitives
- Attempts to take advantage of this cascading capability if the rest of the design uses some intermediate positions of the Shift Register

Shift Registers Coding Examples

The following subsections provide VHDL and Verilog coding examples for shift registers.

32-Bit Shift Register Coding Example One (VHDL)

This coding example uses the concatenation coding style.

```
--  32-bit Shift Register
--  Rising edge clock
--  Active high clock enable
--  Concatenation-based template
--  File: shift_registers_0.vhd

library ieee;
use ieee.std_logic_1164.all;
entity shift_registers_0 is
 generic(
  DEPTH : integer := 32
 );
 port(
  clk   : in  std_logic;
  clken : in  std_logic;
```

```
    SI     : in  std_logic;
    SO     : out std_logic
  );

end shift_registers_0;

architecture archi of shift_registers_0 is
  signal shreg : std_logic_vector(DEPTH - 1 downto 0);
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if clken = '1' then
        shreg <= shreg(DEPTH - 2 downto 0) & SI;
      end if;
    end if;
  end process;
  SO <= shreg(DEPTH - 1);
end archi;
```

32-Bit Shift Register Coding Example Two (VHDL)

The same functionality can also be described as follows:

```
--  32-bit Shift Register
--  Rising edge clock
--  Active high clock enable
--  foor loop-based template
--  File: shift_registers_1.vhd

library ieee;
use ieee.std_logic_1164.all;
entity shift_registers_1 is
  generic(
    DEPTH : integer := 32
  );
  port(
    clk    : in  std_logic;
    clken  : in  std_logic;
    SI     : in  std_logic;
    SO     : out std_logic
  );

end shift_registers_1;

architecture archi of shift_registers_1 is
  signal shreg : std_logic_vector(DEPTH - 1 downto 0);
begin
  process(clk)
  begin
```

```
    if rising_edge(clk) then
      if clken = '1' then
        for i in 0 to DEPTH - 2 loop
          shreg(i + 1) <= shreg(i);
        end loop;
        shreg(0) <= SI;
      end if;
    end if;
  end process;
  SO <= shreg(DEPTH - 1);
end archi;
```

8-Bit Shift Register Coding Example One (Verilog)

This coding example uses a concatenation to describe the Register chain.

```
//  8-bit Shift Register
//  Rising edge clock
//  Active high clock enable
//  Concatenation-based template
//  File: shift_registers_0.v

module shift_registers_0 (clk, clken, SI, SO);
parameter WIDTH = 32;
inputclk, clken, SI;
output SO;

reg[WIDTH-1:0] shreg;

always @(posedge clk)
  begin
    if (clken)
      shreg = {shreg[WIDTH-2:0], SI};
  end

assign SO = shreg[WIDTH-1];

endmodule
```

32-Bit Shift Register Coding Example Two (Verilog)

```
// 32-bit Shift Register
// Rising edge clock
// Active high clock enable
// For-loop based template
// File: shift_registers_1.v

module shift_registers_1 (clk, clken, SI, SO);
parameter WIDTH = 32;
```

```
inputclk, clken, SI;
output SO;
reg[WIDTH-1:0] shreg;

integer i;
always @(posedge clk)
begin
  if (clken)
    begin
      for (i = 0; i < WIDTH-1; i = i+1)
        shreg[i+1] <= shreg[i];
      shreg[0] <= SI;
    end
end
assign SO = shreg[WIDTH-1];
endmodule
```

SRL Based Shift Registers Reporting

```
---------------------------------------------------------------------------------
 Start RAM, DSP and Shift Register Reporting
---------------------------------------------------------------------------------

Static Shift Register:
|Module Name|RTL Name                 |Length|Width|Reset Signal|Pull out first Reg|Pull out last Reg|SRL16E|SRLC32E|
|-----------|-------------------------|------|-----|------------|------------------|-----------------|------|-------|
|top        |top_rtl_inst/shreg_reg[31]|32    |1    |NO          |NO                |YES              |0     |1      |
|-----------|-------------------------|------|-----|------------|------------------|-----------------|------|-------|

---------------------------------------------------------------------------------
 Finished RAM, DSP and Shift Register Reporting
---------------------------------------------------------------------------------
```

```
Report Cell Usage:
-----+-------+-----
     |Cell   |Count
-----+-------+-----
1    |SRLC32E|    1
```

Dynamic Shift Registers

A Dynamic Shift register is a Shift register the length of which can vary dynamically during circuit operation.

A Dynamic Shift register can be seen as:

- A chain of Flip-Flops of the maximum length that it can accept during circuit operation.
- A Multiplexer that selects, in a given clock cycle, the stage at which data is to be extracted from the propagation chain.

The Vivado synthesis tool can infer Dynamic Shift registers of any maximal length.

Vivado synthesis tool can implement Dynamic Shift registers optimally using the SRL-type primitives available in the device family.

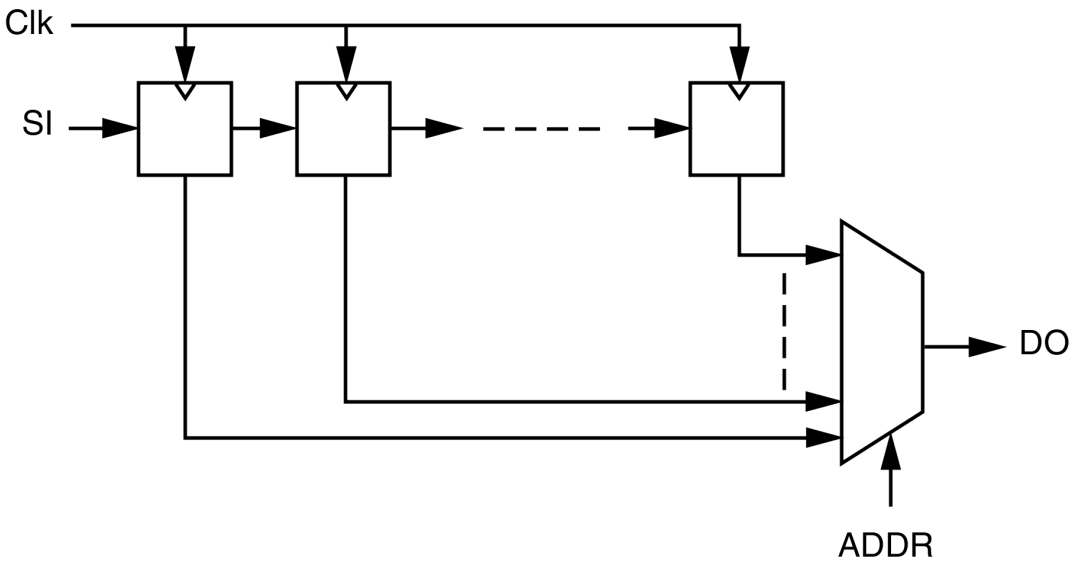

Figure 3-1: **Dynamic Shift Registers Diagram**

Dynamic Shift Registers Coding Examples

Download the coding example files from: Coding Examples

32-Bit Dynamic Shift Registers Coding Example (Verilog)

```
// 32-bit dynamic shift register.
// Download:
// File: dynamic_shift_registers_1.v

module dynamic_shift_register_1 (CLK, CE, SEL, SI, DO);
parameter SELWIDTH = 5;
inputCLK, CE, SI;
input[SELWIDTH-1:0] SEL;
outputDO;

localparam DATAWIDTH = 2**SELWIDTH;
reg [DATAWIDTH-1:0] data;

assign DO = data[SEL];

always @(posedge CLK)
 begin
  if (CE == 1'b1)
    data <= {data[DATAWIDTH-2:0], SI};
  end
endmodule
```

32-Bit Dynamic Shift Registers Coding Example (VHDL)

```
-- 32-bit dynamic shift register.
-- File:dynamic_shift_registers_1.vhd
-- 32-bit dynamic shift register.
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity dynamic_shift_register_1 is
 generic(
  DEPTH     : integer := 32;
  SEL_WIDTH : integer := 5
 );
 port(
  CLK : in  std_logic;
  SI  : in  std_logic;
  CE  : in  std_logic;
  A   : in  std_logic_vector(SEL_WIDTH - 1 downto 0);
  DO  : out std_logic
 );

end dynamic_shift_register_1;

architecture rtl of dynamic_shift_register_1 is
 type SRL_ARRAY is array (DEPTH - 1 downto 0) of std_logic;

 signal SRL_SIG : SRL_ARRAY;

begin
 process(CLK)
 begin
  if rising_edge(CLK) then
   if CE = '1' then
    SRL_SIG <= SRL_SIG(DEPTH - 2 downto 0) & SI;
   end if;
  end if;
 end process;

 DO <= SRL_SIG(conv_integer(A));

end rtl;
```

Multipliers

Vivado synthesis infers Multiplier macros from multiplication operators in the source code. The resulting signal width equals the sum of the two operand sizes. For example, multiplying a 16-bit signal by an 8-bit signal produces a result of 24 bits.

RECOMMENDED: *If you do not intend to use all most significant bits of a device, Xilinx recommends that you reduce the size of operands to the minimum needed, especially if the Multiplier macro is implemented on slice logic.*

Multipliers Implementation

Multiplier macros can be implemented on:

- Slice logic
- DSP blocks

The implementation choice is:

- Driven by the size of operands
- Aimed at maximizing performance

To force implementation of a Multiplier to slice logic or DSP block, set `USE_DSP48` attribute on the appropriate signal, entity, or module to either:

- no (slice logic)
- yes (DSP block)

DSP Block Implementation

When implementing a Multiplier in a single DSP block, Vivado synthesis tries to take advantage of the pipelining capabilities of DSP blocks. Vivado synthesis pulls up to two levels of Registers present:

- On the multiplication operands
- After the multiplication

When a Multiplier does not fit on a single DSP block, Vivado synthesis decomposes the macro to implement it. In that case, Vivado synthesis uses either of the following:

- Several DSP blocks
- A hybrid solution involving both DSP blocks and slice logic

Use the KEEP attribute to restrict absorption of Registers into DSP blocks. For example, if a Register is present on an operand of the multiplier, place KEEP on the output of the Register to prevent the Register from being absorbed into the DSP block.

Multipliers Coding Examples

Unsigned 16x24-Bit Multiplier Coding Example (Verilog)

```
// Unsigned 16x24-bit Multiplier
//1 latency stage on operands
//3 latency stage after the multiplication
// File: multipliers2.v
//
module mult_unsigned (clk, A, B, RES);

parameter WIDTHA = 16;
parameter WIDTHB = 24;
inputclk;
input[WIDTHA-1:0]A;
input[WIDTHB-1:0]B;
output [WIDTHA+WIDTHB-1:0] RES;

reg[WIDTHA-1:0]rA;
reg[WIDTHB-1:0]rB;
reg[WIDTHA+WIDTHB-1:0] M [3:0];

integer i;
always @(posedge clk)
 begin
  rA <= A;
  rB <= B;
  M[0] <= rA * rB;
  for (i = 0; i < 3; i = i+1)
    M[i+1] <= M[i];
 end

assign RES = M[3];

endmodule
```

Unsigned 16x16-Bit Multiplier Coding Example (VHDL)

```
-- Unsigned 16x16-bit Multiplier
-- File: mult_unsigned.vhd
--
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_unsigned.all;

entity mult_unsigned is
 generic(
  WIDTHA : integer := 16;
  WIDTHB : integer := 16
 );
 port(
  A   : in  std_logic_vector(WIDTHA - 1 downto 0);
  B   : in  std_logic_vector(WIDTHB - 1 downto 0);
  RES : out std_logic_vector(WIDTHA + WIDTHB - 1 downto 0)
 );
end mult_unsigned;

architecture beh of mult_unsigned is
begin
 RES <= A * B;
end beh;
```

Multiply-Add and Multiply-Accumulate

The following macros are inferred:

- Multiply-Add
- Multiply-Sub
- Multiply-Add/Sub
- Multiply-Accumulate

The macros are inferred by aggregation of:

- A Multiplier
- An Adder/Subtractor
- Registers

Multiply-Add and Multiply-Accumulate Implementation

During Multiply-Add and Multiply-Accumulate implementation:

- Vivado synthesis can implement an inferred Multiply-Add or Multiply-Accumulate macro on DSP block resources.
- Vivado synthesis attempts to take advantage of the pipelining capabilities of DSP blocks.

- Vivado synthesis pulls up to:
  - Two register stages present on the multiplication operands.
  - One register stage present after the multiplication.
  - One register stage found after the Adder, Subtractor, or Adder/Subtractor.
  - One register stage on the add/sub selection signal.
  - One register stage on the Adder optional carry input.
- Vivado synthesis can implement a Multiply Accumulate in a DSP48 block if its implementation requires only a single DSP48 resource.
- If the macro exceeds the limits of a single DSP48:
  - Vivado synthesis processes it as two separate Multiplier and Accumulate macros.
  - Vivado synthesis makes independent decisions on each macro.

Macro Implementation on DSP Block Resources

Macro implementation on DSP block resources is inferred by default in Vivado synthesis.

- In default mode, Vivado synthesis:
  - Implements Multiply-Add and Multiply-Accumulate macros.
  - Takes into account DSP block resources availability in the targeted device.
  - uses all available DSP resources.
  - Attempts to maximize circuit performance by leveraging all the pipelining capabilities of DSP blocks.
  - Scans for opportunities to absorb Registers into a Multiply-Add or Multiply-Accumulate macro.

Use the `KEEP` attribute to restrict absorption of Registers into DSP blocks. For example, to exclude a register present on an operand of the Multiplier from absorption into the DSP block, apply `KEEP` on the output of the register. For more information about the `KEEP` attribute, see KEEP.

Download the coding example files from: Coding Examples.

Complex Multiplier Examples

The following examples show complex multiplier examples in VHDL and Verilog. Note that the coding example files also include a complex multiplier with accumulation example that uses three DSP blocks for the UltraScale architecture.

Complex Multiplier Example (Verilog)

Fully pipelined complex multiplier using three `DSP48` blocks.

```
//
// Complex Multiplier (pr+i.pi) = (ar+i.ai)*(br+i.bi)
// file: cmult.v

module cmult # (parameter AWIDTH = 16, BWIDTH = 18)
    (
     input clk,
     input signed [AWIDTH-1:0]     ar, ai,
     input signed [BWIDTH-1:0]     br, bi,
     output signed [AWIDTH+BWIDTH:0] pr, pi
    );

reg signed [AWIDTH-1:0]ai_d, ai_dd, ai_ddd, ai_dddd ;
reg signed [AWIDTH-1:0]ar_d, ar_dd, ar_ddd, ar_dddd ;
reg signed [BWIDTH-1:0]bi_d, bi_dd, bi_ddd, br_d, br_dd, br_ddd ;
reg signed [AWIDTH:0]addcommon ;
reg signed [BWIDTH:0]addr, addi ;
reg signed [AWIDTH+BWIDTH:0]mult0, multr, multi, pr_int, pi_int ;
reg signed [AWIDTH+BWIDTH:0]common, commonr1, commonr2 ;

always @(posedge clk)
 begin
   ar_d   <= ar;
   ar_dd  <= ar_d;
   ai_d   <= ai;
   ai_dd  <= ai_d;
   br_d   <= br;
   br_dd  <= br_d;
   br_ddd <= br_dd;
   bi_d   <= bi;
   bi_dd  <= bi_d;
   bi_ddd <= bi_dd;
 end

// Common factor (ar ai) x bi, shared for the calculations of the real
and imaginary final products
//
```

```
always @(posedge clk)
 begin
   addcommon <= ar_d - ai_d;
   mult0     <= addcommon * bi_dd;
   common    <= mult0;
 end

// Real product
//
always @(posedge clk)
 begin
    ar_ddd   <= ar_dd;
    ar_dddd  <= ar_ddd;
    addr     <= br_ddd - bi_ddd;
    multr    <= addr * ar_dddd;
    commonr1 <= common;
    pr_int   <= multr + commonr1;
 end

// Imaginary product
//
always @(posedge clk)
 begin
   ai_ddd   <= ai_dd;
   ai_dddd  <= ai_ddd;
   addi     <= br_ddd + bi_ddd;
   multi    <= addi * ai_dddd;
   commonr2 <= common;
   pi_int   <= multi + commonr2;
 end

assign pr = pr_int;
assign pi = pi_int;

endmodule // cmult
```

Complex Multiplier Examples (VHDL)

Fully pipelined complex multiplier using three `DSP48` blocks.

```
-- Complex Multiplier (pr+i.pi) = (ar+i.ai)*(br+i.bi)
--
--
-- cumult.vhd
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
entity cmult is
  generic(AWIDTH : natural := 16;
          BWIDTH : natural := 16);
  port(clk    : in  std_logic;
     ar, ai : in  std_logic_vector(AWIDTH - 1 downto 0);
     br, bi : in  std_logic_vector(BWIDTH - 1 downto 0);
     pr, pi : out std_logic_vector(AWIDTH + BWIDTH downto 0));
end cmult;

architecture rtl of cmult is
  signal ai_d, ai_dd, ai_ddd, ai_dddd            : signed(AWIDTH - 1 downto
0);
  signal ar_d, ar_dd, ar_ddd, ar_dddd            : signed(AWIDTH - 1 downto
0);
  signal bi_d, bi_dd, bi_ddd, br_d, br_dd, br_ddd : signed(BWIDTH - 1 downto
0);
  signal addcommon                                : signed(AWIDTH downto 0);
  signal addr, addi                               : signed(BWIDTH downto 0);
  signal mult0, multr, multi, pr_int, pi_int      : signed(AWIDTH + BWIDTH
downto 0);
  signal common, commonr1, commonr2               : signed(AWIDTH + BWIDTH
downto 0);

begin
  process(clk)
  begin
    if rising_edge(clk) then
      ar_d   <= signed(ar);
      ar_dd  <= signed(ar_d);
      ai_d   <= signed(ai);
      ai_dd  <= signed(ai_d);
      br_d   <= signed(br);
      br_dd  <= signed(br_d);
      br_ddd <= signed(br_dd);
      bi_d   <= signed(bi);
      bi_dd  <= signed(bi_d);
      bi_ddd <= signed(bi_dd);
    end if;
  end process;

  -- Common factor (ar - ai) x bi, shared for the calculations
  -- of the real and imaginary final products.
  --
  process(clk)
  begin
    if rising_edge(clk) then
      addcommon <= resize(ar_d, AWIDTH + 1) - resize(ai_d, AWIDTH + 1);
      mult0     <= addcommon * bi_dd;
      common    <= mult0;
    end if;
  end process;

  -- Real product
```

```
    --
    process(clk)
    begin
      if rising_edge(clk) then
        ar_ddd   <= ar_dd;
        ar_dddd  <= ar_ddd;
        addr     <= resize(br_ddd, BWIDTH + 1) - resize(bi_ddd, BWIDTH + 1);
        multr    <= addr * ar_dddd;
        commonr1 <= common;
        pr_int   <= multr + commonr1;
      end if;
    end process;

    -- Imaginary product
    --
    process(clk)
    begin
      if rising_edge(clk) then
        ai_ddd   <= ai_dd;
        ai_dddd  <= ai_ddd;
        addi     <= resize(br_ddd, BWIDTH + 1) + resize(bi_ddd, BWIDTH + 1);
        multi    <= addi * ai_dddd;
        commonr2 <= common;
        pi_int   <= multi + commonr2;
      end if;
    end process;

    --
    -- VHDL type conversion for output
    --
    pr <= std_logic_vector(pr_int);
    pi <= std_logic_vector(pi_int);

  end rtl;
```

Pre-Adders in the DSP Block

When coding for inference and targeting the DSP block, it is recommended to use signed arithmetic and it is a requirement to have one extra bit of width for the pre-adder result so that it can be packed into the DSP block.

Pre-Adder Dynamically Configured Followed by Multiplier and Post-Adder (Verilog)

```
// Pre-add/subtract select with Dynamic control
// dynpreaddmultadd.v
module dynpreaddmultadd  # (parameter SIZEIN = 16)
        (
          input clk, ce, rst, subadd,
          input  signed [SIZEIN-1:0] a, b, c, d,
          output signed [2*SIZEIN:0] dynpreaddmultadd_out
        );

// Declare registers for intermediate values
reg signed [SIZEIN-1:0] a_reg, b_reg, c_reg;
reg signed [SIZEIN:0]   add_reg;
reg signed [2*SIZEIN:0] d_reg, m_reg, p_reg;

always @(posedge clk)
begin
 if (rst)
  begin
   a_reg   <= 0;
   b_reg   <= 0;
   c_reg   <= 0;
   d_reg   <= 0;
   add_reg <= 0;
   m_reg   <= 0;
   p_reg   <= 0;
  end
 else if (ce)
  begin
   a_reg   <= a;
   b_reg   <= b;
   c_reg   <= c;
   d_reg   <= d;
   if (subadd)
    add_reg <= a - b;
   else
    add_reg <= a + b;
   m_reg   <= add_reg * c_reg;
   p_reg   <= m_reg + d_reg;
```

```
  end
end

// Output accumulation result
assign dynpreaddmultadd_out = p_reg;

endmodule // dynpreaddmultadd
```

Pre-Adder Dynamically Configured Followed by Multiplier and Post-Adder (VHDL)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dynpreaddmultadd is
 generic(
  AWIDTH : natural := 12;
  BWIDTH : natural := 16;
  CWIDTH : natural := 17
 );
 port(
  clk    : in  std_logic;
  subadd : in  std_logic;
  ain    : in  std_logic_vector(AWIDTH - 1 downto 0);
  bin    : in  std_logic_vector(BWIDTH - 1 downto 0);
  cin    : in  std_logic_vector(CWIDTH - 1 downto 0);
  din    : in  std_logic_vector(BWIDTH + CWIDTH downto 0);
  pout   : out std_logic_vector(BWIDTH + CWIDTH downto 0)
 );
end dynpreaddmultadd;

architecture rtl of dynpreaddmultadd is
 signal a          : signed(AWIDTH - 1 downto 0);
 signal b          : signed(BWIDTH - 1 downto 0);
 signal c          : signed(CWIDTH - 1 downto 0);
 signal add        : signed(BWIDTH downto 0);
 signal d, mult, p : signed(BWIDTH + CWIDTH downto 0);

begin
 process(clk)
 begin
  if rising_edge(clk) then
   a <= signed(ain);
   b <= signed(bin);
   c <= signed(cin);
   d <= signed(din);
   if subadd = '1' then
```

```
      add <= resize(a, BWIDTH + 1) - resize(b, BWIDTH + 1);
    else
      add <= resize(a, BWIDTH + 1) + resize(b, BWIDTH + 1);
    end if;
    mult <= add * c;
    p    <= mult + d;
  end if;
 end process;

 --
 -- Type conversion for output
 --
 pout <= std_logic_vector(p);

end rtl;
```

Using the Squarer in the UltraScale DSP Block

The UltraScale™ DSP block (DSP48E2) primitive can compute the square of an input or of the output of the pre-adder.

Download the coding example files from: Coding Examples

The following are examples of the square of a difference; this can be used to efficiently replace calculations on absolute values of differences.

It fits into a single DSP block and runs at full speed. The coding example files mentioned above also include an accumulator of square of differences which also fits into a single DSP block for the UltraScale architecture.

Square of a Difference (Verilog)

```
// Squarer support for DSP block (DSP48E2) with
// pre-adder configured
// as subtractor
// File: squarediffmult.v

module squarediffmult # (parameter SIZEIN = 16)
                        (
                          input clk, ce, rst,
                          input  signed [SIZEIN-1:0]   a, b,
                          output signed [2*SIZEIN+1:0] square_out
                        );

   // Declare registers for intermediate values
reg signed [SIZEIN-1:0]   a_reg, b_reg;
```

```
reg signed [SIZEIN:0]     diff_reg;
reg signed [2*SIZEIN+1:0] m_reg, p_reg;

always @(posedge clk)
begin
 if (rst)
  begin
    a_reg    <= 0;
    b_reg    <= 0;
    diff_reg <= 0;
    m_reg    <= 0;
    p_reg    <= 0;
  end
 else
  if (ce)
  begin
    a_reg     <= a;
    b_reg     <= b;
    diff_reg <= a_reg - b_reg;
    m_reg  <= diff_reg * diff_reg;
    p_reg  <= m_reg;
 end
end

// Output result
assign square_out = p_reg;

endmodule // squarediffmult
```

Square of a Difference (VHDL)

```
-- Squarer support for DSP block (DSP48E2) with pre-adder
-- configured
-- as subtractor
-- File: squarediffmult.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity squarediffmult is
 generic(
   SIZEIN : natural := 16
 );
 port(
   clk, ce, rst : in  std_logic;
   ain, bin     : in  std_logic_vector(SIZEIN - 1 downto 0);
```

```
    square_out   : out std_logic_vector(2 * SIZEIN + 1 downto 0)
  );
end squarediffmult;

architecture rtl of squarediffmult is

  -- Declare intermediate values
  signal a_reg, b_reg : signed(SIZEIN - 1 downto 0);
  signal diff_reg     : signed(SIZEIN downto 0);
  signal m_reg, p_reg : signed(2 * SIZEIN + 1 downto 0);

begin
  process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        a_reg    <= (others => '0');
        b_reg    <= (others => '0');
        diff_reg <= (others => '0');
        m_reg    <= (others => '0');
        p_reg    <= (others => '0');
      else
        a_reg    <= signed(ain);
        b_reg    <= signed(bin);
        diff_reg <= resize(a_reg, SIZEIN + 1) - resize(b_reg, SIZEIN +
1);
        m_reg    <= diff_reg * diff_reg;
        p_reg    <= m_reg;
      end if;
    end if;
  end process;

  --
  -- Type conversion for output
  --
  square_out <= std_logic_vector(p_reg);

end rtl;
```

FIR Filters

Vivado synthesis infers cascades of multiply-add to compose FIR filters directly from RTL.

There are several possible implementations of such filters; one example is the systolic filter described in the *7 Series DSP48E1 Slice User Guide* (UG479) [Ref 13] and shown in the "8-Tap Even Symmetric Systolic FIR" (Figure 3-6).

Download the coding example files from: Coding Examples.

8-Tap Even Symmetric Systolic FIR (Verilog)

```
// sfir_even_symmetric_systolic_top.v
// FIR Symmetric Systolic Filter, Top module is
sfir_even_symmetric_systolic_top

// sfir_shifter - sub module which is used in top level
(* dont_touch = "yes" *)
module sfir_shifter #(parameter dsize = 16, nbtap = 4)
                  (input clk, [dsize-1:0] datain, output [dsize-1:0]
dataout);

   (* srl_style = "srl_register" *) reg [dsize-1:0] tmp [0:2*nbtap-1];
   integer i;

   always @(posedge clk)
     begin
        tmp[0] <= datain;
        for (i=0; i<=2*nbtap-2; i=i+1)
          tmp[i+1] <= tmp[i];
     end

   assign dataout = tmp[2*nbtap-1];

endmodule

// sfir_even_symmetric_systolic_element - sub module which is used in
top
module sfir_even_symmetric_systolic_element #(parameter dsize = 16)
    (input clk, input signed [dsize-1:0] coeffin, datain, datazin,
input signed [2*dsize-1:0] cascin,
     output signed [dsize-1:0] cascdata, output reg signed
[2*dsize-1:0] cascout);

   reg signed [dsize-1:0]   coeff;
   reg signed [dsize-1:0]   data;
   reg signed [dsize-1:0]   dataz;
   reg signed [dsize-1:0]   datatwo;
   reg signed [dsize:0]     preadd;
   reg signed [2*dsize-1:0] product;

   assign cascdata = datatwo;

   always @(posedge clk)
     begin
        coeff   <= coeffin;
        data    <= datain;
```

```
        datatwo <= data;
        dataz   <= datazin;
        preadd  <= datatwo + dataz;
        product <= preadd  * coeff;
        cascout <= product + cascin;
      end

endmodule


module sfir_even_symmetric_systolic_top #(parameter nbtap = 4, dsize
= 16, psize = 2*dsize)
          (input clk, input signed [dsize-1:0] datain, output signed
[2*dsize-1:0] firout);

   wire signed [dsize-1:0] h [nbtap-1:0];
   wire signed [dsize-1:0] arraydata [nbtap-1:0];
   wire signed [psize-1:0] arrayprod [nbtap-1:0];

   wire signed [dsize-1:0] shifterout;
   reg  signed [dsize-1:0] dataz [nbtap-1:0];

   assign h[0] =    7;
   assign h[1] =   14;
   assign h[2] = -138;
   assign h[3] =  129;

   assign firout = arrayprod[nbtap-1]; // Connect last product to
output

   sfir_shifter #(dsize, nbtap) shifter_inst0 (clk, datain,
shifterout);

   generate
      genvar I;
      for (I=0; I<nbtap; I=I+1)
        if (I==0)
          sfir_even_symmetric_systolic_element #(dsize) fte_inst0
(clk, h[I],        datain, shifterout,    {32{1'b0}}, arraydata[I],
arrayprod[I]);
        else
          sfir_even_symmetric_systolic_element #(dsize) fte_inst
(clk, h[I], arraydata[I-1], shifterout, arrayprod[I-1],
arraydata[I], arrayprod[I]);
   endgenerate

endmodule // sfir_even_symmetric_systolic_top
```

8-Tap Even Symmetric Systolic FIR (VHDL)

```
--
-- FIR filter top
-- File: sfir_even_symmetric_systolic_top.vhd


-- FIR filter shifter
-- submodule used in top (sfir_even_symmetric_systolic_top)
library ieee;
use ieee.std_logic_1164.all;

entity sfir_shifter is
  generic(
    DSIZE : natural := 16;
    NBTAP : natural := 4
  );
  port(
    clk     : in  std_logic;
    datain  : in  std_logic_vector(DSIZE - 1 downto 0);
    dataout : out std_logic_vector(DSIZE - 1 downto 0)
  );
end sfir_shifter;

architecture rtl of sfir_shifter is

  -- Declare signals
  --
  type CHAIN is array (0 to 2 * NBTAP - 1) of std_logic_vector(DSIZE - 1
downto 0);
  signal tmp : CHAIN;

begin
  process(clk)
  begin
    if rising_edge(clk) then
      tmp(0) <= datain;
      looptmp : for i in 0 to 2 * NBTAP - 2 loop
        tmp(i + 1) <= tmp(i);
      end loop;
    end if;
  end process;

  dataout <= tmp(2 * NBTAP - 1);

end rtl;

--
-- FIR filter engine (multiply with pre-add and post-add)
-- submodule used in top (sfir_even_symmetric_systolic_top)
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
entity sfir_even_symmetric_systolic_element is
  generic(DSIZE : natural := 16);
  port(clk                      : in  std_logic;
     coeffin, datain, datazin : in  std_logic_vector(DSIZE - 1 downto 0);
     cascin                   : in  std_logic_vector(2 * DSIZE downto 0);
     cascdata                 : out std_logic_vector(DSIZE - 1 downto 0);
     cascout                  : out std_logic_vector(2 * DSIZE downto 0));
end sfir_even_symmetric_systolic_element;

architecture rtl of sfir_even_symmetric_systolic_element is

  -- Declare signals
  --
  signal coeff, data, dataz, datatwo : signed(DSIZE - 1 downto 0);
  signal preadd                      : signed(DSIZE downto 0);
  signal product, cascouttmp         : signed(2 * DSIZE downto 0);

begin
  process(clk)
  begin
    if rising_edge(clk) then
      coeff      <= signed(coeffin);
      data       <= signed(datain);
      datatwo    <= data;
      dataz      <= signed(datazin);
      preadd     <= resize(datatwo, DSIZE + 1) + resize(dataz, DSIZE + 1);
      product    <= preadd * coeff;
      cascouttmp <= product + signed(cascin);
    end if;
  end process;

  -- Type conversion for output
  --
  cascout  <= std_logic_vector(cascouttmp);
  cascdata <= std_logic_vector(datatwo);

end rtl;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sfir_even_symmetric_systolic_top is
  generic(NBTAP : natural := 4;
          DSIZE : natural := 16;
          PSIZE : natural := 33);
  port(clk    : in  std_logic;
     datain : in  std_logic_vector(DSIZE - 1 downto 0);
     firout : out std_logic_vector(PSIZE - 1 downto 0));
end sfir_even_symmetric_systolic_top;

architecture rtl of sfir_even_symmetric_systolic_top is
```

```
  -- Declare signals
  --
  type DTAB is array (0 to NBTAP - 1) of std_logic_vector(DSIZE - 1 downto 0);
  type HTAB is array (0 to NBTAP - 1) of std_logic_vector(0 to DSIZE - 1);
  type PTAB is array (0 to NBTAP - 1) of std_logic_vector(PSIZE - 1 downto 0);

  signal arraydata, dataz : DTAB;
  signal arrayprod        : PTAB;
  signal shifterout       : std_logic_vector(DSIZE - 1 downto 0);

  -- Initialize coefficients and a "zero" for the first filter element
  --
  constant h : HTAB := ((std_logic_vector(TO_SIGNED(63, DSIZE))),
    (std_logic_vector(TO_SIGNED(18, DSIZE))),
    (std_logic_vector(TO_SIGNED(-100, DSIZE))),
    (std_logic_vector(TO_SIGNED(1, DSIZE))));
  constant zero_psize : std_logic_vector(PSIZE - 1 downto 0) := (others =>
'0');

begin

  -- Connect last product to output
  --
  firout <= arrayprod(nbtap - 1);

  -- Shifter
  --
  shift_u0 : entity work.sfir_shifter
    generic map(DSIZE, NBTAP)
    port map(clk, datain, shifterout);

  -- Connect the arithmetic building blocks of the FIR
  --
  gen : for I in 0 to NBTAP - 1 generate
  begin
    g0 : if I = 0 generate
      element_u0 : entity work.sfir_even_symmetric_systolic_element
        generic map(DSIZE)
        port map(clk, h(I), datain, shifterout, zero_psize, arraydata(I),
arrayprod(I));
    end generate g0;
    gi : if I /= 0 generate
      element_ui : entity work.sfir_even_symmetric_systolic_element
        generic map(DSIZE)
        port map(clk, h(I), arraydata(I - 1), shifterout, arrayprod(I - 1),
arraydata(I), arrayprod(I));
    end generate gi;
  end generate gen;

end rtl;
```

Convergent Rounding (LSB Correction Technique)

The DSP block primitive leverages a pattern detect circuitry to compute convergent rounding (either to even, or to odd).

The following are examples of the convergent rounding inference, which infers at the block full performance, and also infers a 2-input AND gate (1 LUT) to implement the LSB correction.

Rounding to Even (Verilog)

```
// Convergent rounding(Even) Example which makes use of pattern detect
// File: convergentRoundingEven.v
module convergentRoundingEven (
                                input clk,
                                input [23:0] a,
                                input [15:0] b,
                              output reg signed [23:0] zlast
                                 );

reg signed [23:0] areg;
reg signed [15:0] breg;
reg signed [39:0] z1;

reg pattern_detect;
wire [15:0] pattern = 16'b0000000000000000;
wire [39:0] c = 40'b0000000000000000000000000111111111111111; // 15 ones

wire signed [39:0] multadd;
wire signed [15:0] zero;
reg  signed [39:0] multadd_reg;

// Convergent Rounding: LSB Correction Technique
// ---------------------------------------------
// For static convergent rounding, the pattern detector can be used
// to detect the midpoint case. For example, in an 8-bit round, if
// the decimal place is set at 4, the C input should be set to
// 0000.0111.  Round to even rounding should use CARRYIN = "1" and
// check for PATTERN "XXXX.0000" and replace the units place with 0
// if the pattern is matched. See UG193 for more details.

assign multadd = z1 + c + 1'b1;

always @(posedge clk)
begin
  areg  <= a;
  breg  <= b;
  z1    <= areg * breg;
  pattern_detect <= multadd[15:0] == pattern ? 1'b1 : 1'b0;
  multadd_reg <= multadd;
```

```
end

// Unit bit replaced with 0 if pattern is detected
always @(posedge clk)
 zlast <= pattern_detect ? {multadd_reg[39:17],1'b0} : multadd_reg[39:16];

endmodule // convergentRoundingEven
```

Rounding to Even (VHDL)

```
-- Convergent rounding(Even) Example which makes use of pattern
detect
-- File: convergentRoundingEven.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity convergentRoundingEven is
  port (clk   : in  std_logic;
        a     : in  std_logic_vector (23 downto 0);
        b     : in  std_logic_vector (15 downto 0);
        zlast : out std_logic_vector (23 downto 0));
end convergentRoundingEven;

architecture beh of convergentRoundingEven is

  signal ar : signed(a'range);
  signal br : signed(b'range);
  signal z1 : signed(a'length + b'length - 1 downto 0);

  signal multaddr       : signed(a'length + b'length - 1 downto 0);
  signal multadd        : signed(a'length + b'length - 1 downto 0);
  signal pattern_detect : boolean;

  constant pattern : signed(15 downto 0) := (others => '0');
  constant c : signed := "00000000000000000000000011111111111111111";

  -- Convergent Rounding: LSB Correction Technique
  -- ---------------------------------------------
  -- For static convergent rounding, the pattern detector can be used
  -- to detect the midpoint case. For example, in an 8-bit round, if
  -- the decimal place is set at 4, the C input should be set to
  -- 0000.0111.  Round to even rounding should use CARRYIN = "1" and
  -- check for PATTERN "XXXX.0000" and replace the units place with 0
  -- if the pattern is matched. See UG193 for more details.

begin

  multadd <= z1 + c;
```

```
  process(clk)
  begin
    if rising_edge(clk) then
      ar       <= signed(a);
      br       <= signed(b);
      z1       <= ar * br;
      multaddr <= multadd;
      if multadd(15 downto 0) = pattern then
        pattern_detect <= true;
      else
        pattern_detect <= false;
      end if;
    end if;
  end process;

  -- Unit bit replaced with 0 if pattern is detected
  process(clk)
  begin
    if rising_edge(clk) then
      if pattern_detect = true then
        zlast <= std_logic_vector(multaddr(39 downto 17)) & "0";
      else
        zlast <= std_logic_vector(multaddr(39 downto 16));
      end if;
    end if;
  end process;

end beh;
```

Rounding to Odd (Verilog)

```
// Convergent rounding(Odd) Example which makes use of pattern detect
// File: convergentRoundingOdd.v
module convergentRoundingOdd (
                               input clk,
                               input [23:0] a,
                               input [15:0] b,
                         output reg signed [23:0] zlast
                              );

reg signed [23:0] areg;
reg signed [15:0] breg;
reg signed [39:0] z1;

reg pattern_detect;
wire [15:0] pattern = 16'b1111111111111111;
```

```
wire [39:0] c = 40'b0000000000000000000000000111111111111111; // 15
ones

wire signed [39:0] multadd;
wire signed [15:0] zero;
reg  signed [39:0] multadd_reg;

// Convergent Rounding: LSB Correction Technique
// ---------------------------------------------
// For static convergent rounding,  the pattern detector can be
// used to detect  the midpoint case. For example,  in an 8-bit
// round, if the decimal place is  set at 4, the C input should
// be  set to  0000.0111.   Round to  odd  rounding should  use
// CARRYIN =  "0" and  check for  PATTERN "XXXX.1111"  and then
// replace  the  units place  bit  with  1  if the  pattern  is
// matched. See UG193 for details

assign multadd = z1 + c;

always @(posedge clk)
begin
  areg  <= a;
 breg  <= b;
 z1    <= areg * breg;
 pattern_detect <= multadd[15:0] == pattern ? 1'b1 : 1'b0;
 multadd_reg <= multadd;
end

always @(posedge clk)
  zlast <= pattern_detect ? {multadd_reg[39:17],1'b1} :
multadd_reg[39:16];

endmodule // convergentRoundingOdd
```

Rounding to Odd (VHDL)

```
-- Convergent rounding(Odd) Example which makes use of pattern detect
-- File: convergentRoundingOdd.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity convergentRoundingOdd is
  port (clk   : in  std_logic;
        a     : in  std_logic_vector (23 downto 0);
        b     : in  std_logic_vector (15 downto 0);
        zlast : out std_logic_vector (23 downto 0));
end convergentRoundingOdd;
```

```
architecture beh of convergentRoundingOdd is

  signal ar : signed(a'range);
  signal br : signed(b'range);
  signal z1 : signed(a'length + b'length - 1 downto 0);

  signal multadd, multaddr : signed(a'length + b'length - 1 downto 0);
  signal pattern_detect    : boolean;

  constant pattern : signed(15 downto 0) := (others => '1');
  constant c       : signed :=
"00000000000000000000000001111111111111111";

  -- Convergent Rounding: LSB Correction Technique
  -- ---------------------------------------------
  -- For static convergent rounding,  the pattern detector can be
  -- used to detect  the midpoint case. For example,  in an 8-bit
  -- round, if the decimal place is  set at 4, the C input should
  -- be  set to  0000.0111.   Round to  odd  rounding should  use
  -- CARRYIN =  "0" and  check for  PATTERN "XXXX.1111"  and then
  -- replace  the  units place  bit  with  1  if the  pattern  is
  -- matched. See UG193 for details

begin

  multadd <= z1 + c;

  process(clk)
  begin
    if rising_edge(clk) then
      ar       <= signed(a);
      br       <= signed(b);
      z1       <= ar * br;
      multaddr <= multadd;
      if multadd(15 downto 0) = pattern then
        pattern_detect <= true;
      else
        pattern_detect <= false;
      end if;
    end if;
  end process;

  process(clk)
  begin
    if rising_edge(clk) then
      if pattern_detect = true then
        zlast <= std_logic_vector(multaddr(39 downto 17)) & "1";
      else
        zlast <= std_logic_vector(multaddr(39 downto 16));
```

```
        end if;
      end if;
    end process;

end beh;
```

RAM HDL Coding Techniques

Vivado synthesis can interpret various ram coding styles, and maps them into Distributed RAMs or Block RAMs. This action:

- Makes it unnecessary to manually instantiate RAM primitives
- Saves time
- Keeps HDL source code portable and scalable

Download the coding example files from: Coding Examples.

Choosing Between Distributed RAM and Dedicated Block RAM

Data is written synchronously into the RAM for both types. The primary difference between distributed RAM and dedicated block RAM lies in the way data is read from the RAM. See the following table.

Table 3-1: **Distributed RAM versus Dedicated Block RAM**

Action	Distributed RAM	Dedicated Block RAM
Write	Synchronous	Synchronous
Read	Asynchronous	Synchronous

Whether to use distributed RAM or dedicated block RAM can depend upon the characteristics of the RAM described in the HDL source code, the availability of block RAM resources, and whether you have forced a specific implementation style using `RAM_STYLE` attribute. See RAM_STYLE.

Memory Inference Capabilities

Memory inference capabilities include the following:

- Support for any size and data width. Vivado synthesis maps the memory description to one or several RAM primitives
- Single-port, simple-dual port, true dual port
- Up to two write ports

- Multiple read ports

Provided that only one write port is described, Vivado synthesis can identify RAM descriptions with two or more read ports that access the RAM contents at addresses different from the write address.

- Write enable
- RAM enable (block RAM)
- Data output reset (block RAM)
- Optional output register (block RAM)
- Byte write enable (block RAM)
- Each RAM port can be controlled by its distinct clock, port enable, write enable, and data output reset
- Initial contents specification
- Vivado synthesis can use parity bits as regular data bits to accommodate the described data widths

  Note: For more information on parity bits see the user guide for the device you are targeting.

UltraRAM Coding Templates

UltraRAM is described in "Chapter 2, UltraRam Resources" of the *UltraScale Architecture Memory Resources User Guide* (UG573) as follows:

> "UltraRAM is a single-clocked, two port, synchronous memory available in UltraScale+™ devices. Because UltraRAM is compatible with the columnar architecture, multiple UltraRAMs can be instantiated and directly cascaded in an UltraRAM column for the entire height of the device. A column in a single clock region contains 16 UltraRAM blocks.
>
> Devices with UltraRAM include multiple UltraRAM columns distributed in the device. Most of the devices in the UltraScale+ family include UltraRAM blocks. For the available quantity of UltraRAM in specific device families, see the *UltraScale Architecture and Product Overview* (DS890) [Ref 1].

The following files are included in the Coding Examples:

- `xilinx_ultraram_single_port_no_change.v`
- `xilinx_ultraram_single_port_no_change.vhd`
- `xilinx_ultraram_single_port_read_first.v`
- `xilinx_ultraram_single_port_read_first.vhd`
- `xilinx_ultraram_single_port_write_first.v`
- `xilinx_ultraram_single_port_write_first.vhd`

The Vivado tool includes templates of ULTRA_RAM VHDL and Verilog code. The following figure shows the template files.

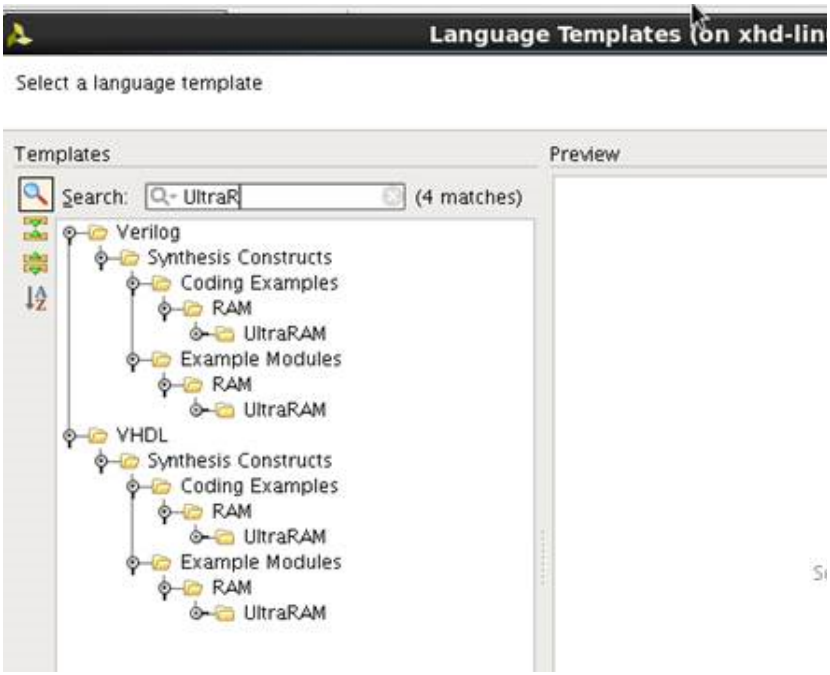

Figure 3-2: **ULTRA_RAM Coding Templates**

See the *UltraScale Architecture Memory Resources* (UG573) [Ref 21] for more information.

RAM HDL Coding Guidelines

Download the coding example files from: Coding Examples.

Block RAM Read/Write Synchronization Modes

You can configure Block RAM resources to provide the following synchronization modes for a given read/write port:

- Read-first: Old content is read before new content is loaded.
- Write-first: New content is immediately made available for reading Write-first is also known as read-through.
- No-change: Data output does not change as new content is loaded into RAM.

Vivado synthesis provides inference support for all of these synchronization modes. You can describe a different synchronization mode for each port of the RAM.

Distributed RAM Examples

The following sections provide VHDL and Verilog coding examples for distributed RAM.

Dual-Port RAM with Asynchronous Read Coding Example (Verilog)

```
// Dual-Port RAM with Asynchronous Read (Distributed RAM)
// File: rams_dist.v

module rams_dist (clk, we, a, dpra, di, spo, dpo);

input clk;
input we;
input [5:0] a;
input [5:0] dpra;
input [15:0] di;
output [15:0] spo;
output [15:0] dpo;
reg[15:0] ram [63:0];

always @(posedge clk)
begin
 if (we)
   ram[a] <= di;
end

assign spo = ram[a];
assign dpo = ram[dpra];

endmodule
```

Single-Port RAM with Asynchronous Read Coding Example (VHDL)

```
-- Single-Port RAM with Asynchronous Read (Distributed RAM)
-- File: rams_dist.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_dist is
 port(
   clk : in  std_logic;
   we  : in  std_logic;
   a   : in  std_logic_vector(5 downto 0);
   di  : in  std_logic_vector(15 downto 0);
   do  : out std_logic_vector(15 downto 0)
 );
end rams_dist;
```

```
architecture syn of rams_dist is
 type ram_type is array (63 downto 0) of std_logic_vector(15 downto
0);
 signal RAM : ram_type;
begin
 process(clk)
 begin
  if (clk'event and clk = '1') then
   if (we = '1') then
    RAM(conv_integer(a)) <= di;
   end if;
  end if;
 end process;

 do <= RAM(conv_integer(a));

end syn;
```

Single-Port Block RAMs

Single-Port Block RAM with Resettable Data Output (Verilog)

```
// Block RAM with Resettable Data Output
// File: rams_sp_rf_rst.v

module rams_sp_rf_rst (clk, en, we, rst, addr, di, dout);
input clk;
input en;
input we;
input rst;
input [9:0] addr;
input [15:0] di;
output [15:0] dout;

reg[15:0] ram [1023:0];
reg[15:0] dout;

always @(posedge clk)
begin
  if (en) //optional enable
    begin
      if (we) //write enable
        ram[addr] <= di;
      if (rst) //optional reset
        dout <= 0;
      else
        dout <= ram[addr];
    end
end

endmodule
```

Single Port Block RAM with Resettable Data Output (VHDL)

```
-- Block RAM with Resettable Data Output
-- File: rams_sp_rf_rst.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_sp_rf_rst is
 port(
  clk  : in  std_logic;
  en   : in  std_logic;
  we   : in  std_logic;
  rst  : in  std_logic;
  addr : in  std_logic_vector(9 downto 0);
  di   : in  std_logic_vector(15 downto 0);
  do   : out std_logic_vector(15 downto 0)
 );
end rams_sp_rf_rst;

architecture syn of rams_sp_rf_rst is
 type ram_type is array (1023 downto 0) of std_logic_vector(15 downto
0);
 signal ram : ram_type;
begin
 process(clk)
 begin
  if clk'event and clk = '1' then
   if en = '1' then            -- optional enable
    if we = '1' then         -- write enable
     ram(conv_integer(addr)) <= di;
    end if;
    if rst = '1' then        -- optional reset
     do <= (others => '0');
    else
     do <= ram(conv_integer(addr));
    end if;
   end if;
  end if;
 end process;

end syn;
```

Single-Port Block RAM Write-First Mode (Verilog)

```
// Single-Port Block RAM Write-First Mode (recommended template)
// File: rams_sp_wf.v
module rams_sp_wf (clk, we, en, addr, di, dout);
input clk;
input we;
input en;
input [9:0] addr;
input [15:0] di;
output [15:0] dout;
reg[15:0] RAM [1023:0];
reg[15:0] dout;

always @(posedge clk)
begin
  if (en)
  begin
    if (we)
      begin
        RAM[addr] <= di;
        dout <= di;
      end
   else
    dout <= RAM[addr];
  end
end
endmodule
```

Single-Port RAM with Read First (VHDL)

```
-- Single-Port Block RAM Read-First Mode
-- rams_sp_rf.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_sp_rf is
 port(
   clk  : in  std_logic;
   we   : in  std_logic;
   en   : in  std_logic;
   addr : in  std_logic_vector(9 downto 0);
   di   : in  std_logic_vector(15 downto 0);
   do   : out std_logic_vector(15 downto 0)
 );
end rams_sp_rf;
```

```
architecture syn of rams_sp_rf is
 type ram_type is array (1023 downto 0) of std_logic_vector(15 downto
0);
 signal RAM : ram_type;
begin
 process(clk)
 begin
  if clk'event and clk = '1' then
   if en = '1' then
    if we = '1' then
     RAM(conv_integer(addr)) <= di;
    end if;
    do <= RAM(conv_integer(addr));
   end if;
  end if;
 end process;

end syn;
```

Single-Port Block RAM Write-First Mode (VHDL)

```
-- Single-Port Block RAM Write-First Mode (recommended template)
--
-- File: rams_02.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_sp_wf is
 port(
  clk  : in  std_logic;
  we   : in  std_logic;
  en   : in  std_logic;
  addr : in  std_logic_vector(9 downto 0);
  di   : in  std_logic_vector(15 downto 0);
  do   : out std_logic_vector(15 downto 0)
 );
end rams_sp_wf;

architecture syn of rams_sp_wf is
 type ram_type is array (1023 downto 0) of std_logic_vector(15 downto
0);
 signal RAM : ram_type;
begin
 process(clk)
 begin
```

```
    if clk'event and clk = '1' then
      if en = '1' then
        if we = '1' then
          RAM(conv_integer(addr)) <= di;
          do                      <= di;
        else
          do <= RAM(conv_integer(addr));
        end if;
      end if;
    end if;
  end process;

end syn;
```

Single-Port Block RAM No-Change Mode (Verilog)

```
// Single-Port Block RAM No-Change Mode
// File: rams_sp_nc.v

module rams_sp_nc (clk, we, en, addr, di, dout);

input clk;
input we;
input en;
input [9:0] addr;
input [15:0] di;
output [15:0] dout;

reg[15:0] RAM [1023:0];
reg[15:0] dout;

always @(posedge clk)
begin
  if (en)
  begin
    if (we)
      RAM[addr] <= di;
    else
      dout <= RAM[addr];
  end
end
endmodule
```

Simple Dual-Port Block RAM Examples

Simple Dual-Port Block RAM with Single Clock (Verilog)

```
// Simple Dual-Port Block RAM with One Clock
// File: simple_dual_one_clock.v

module simple_dual_one_clock (clk,ena,enb,wea,addra,addrb,dia,dob);

input clk,ena,enb,wea;
input [9:0] addra,addrb;
input [15:0] dia;
output [15:0] dob;
reg[15:0] ram [1023:0];
reg[15:0] doa,dob;

always @(posedge clk) begin
 if (ena) begin
    if (wea)
        ram[addra] <= dia;
 end
end

always @(posedge clk) begin
  if (enb)
    dob <= ram[addrb];
end

endmodule
```

Simple Dual-Port Block RAM with Single Clock (VHDL)

```
-- Simple Dual-Port Block RAM with One Clock
-- Correct Modelization with a Shared Variable
-- File:simple_dual_one_clock.vhd

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity simple_dual_one_clock is
 port(
   clk   : in  std_logic;
   ena   : in  std_logic;
   enb   : in  std_logic;
   wea   : in  std_logic;
   addra : in  std_logic_vector(9 downto 0);
   addrb : in  std_logic_vector(9 downto 0);
   dia   : in  std_logic_vector(15 downto 0);
```

```
    dob   : out std_logic_vector(15 downto 0)
  );
end simple_dual_one_clock;

architecture syn of simple_dual_one_clock is
  type ram_type is array (1023 downto 0) of std_logic_vector(15 downto
0);
  shared variable RAM : ram_type;
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      if ena = '1' then
        if wea = '1' then
          RAM(conv_integer(addra)) := dia;
        end if;
      end if;
    end if;
  end process;

  process(clk)
  begin
    if clk'event and clk = '1' then
      if enb = '1' then
        dob <= RAM(conv_integer(addrb));
      end if;
    end if;
  end process;

end syn;
```

Simple Dual-Port Block RAM with Dual Clocks (Verilog)

```
// Simple Dual-Port Block RAM with Two Clocks
// File: simple_dual_two_clocks.v

module simple_dual_two_clocks
(clka,clkb,ena,enb,wea,addra,addrb,dia,dob);

input clka,clkb,ena,enb,wea;
input [9:0] addra,addrb;
input [15:0] dia;
output [15:0] dob;
reg[15:0] ram [1023:0];
reg[15:0] dob;

always @(posedge clka)
begin
```

```
  if (ena)
    begin
      if (wea)
        ram[addra] <= dia;
    end
end

always @(posedge clkb)
begin
  if (enb)
    begin
      dob <= ram[addrb];
    end
end

endmodule
```

Simple Dual-Port Block RAM with Dual Clocks (VHDL)

```
-- Simple Dual-Port Block RAM with Two Clocks
-- Correct Modelization with a Shared Variable
-- File: simple_dual_two_clocks.vhd
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity simple_dual_two_clocks is
 port(
  clka  : in  std_logic;
  clkb  : in  std_logic;
  ena   : in  std_logic;
  enb   : in  std_logic;
  wea   : in  std_logic;
  addra : in  std_logic_vector(9 downto 0);
  addrb : in  std_logic_vector(9 downto 0);
  dia   : in  std_logic_vector(15 downto 0);
  dob   : out std_logic_vector(15 downto 0)
 );
end simple_dual_two_clocks;

architecture syn of simple_dual_two_clocks is
 type ram_type is array (1023 downto 0) of std_logic_vector(15 downto
0);
 shared variable RAM : ram_type;
begin
 process(clka)
 begin
  if clka'event and clka = '1' then
```

```
    if ena = '1' then
     if wea = '1' then
      RAM(conv_integer(addra)) := dia;
     end if;
    end if;
   end if;
  end process;

  process(clkb)
  begin
   if clkb'event and clkb = '1' then
    if enb = '1' then
     dob <= RAM(conv_integer(addrb));
    end if;
   end if;
  end process;

end syn;
```

Dual-Port Block RAM with Two Write Ports in Read First Mode Example (Verilog)

```
// Dual-Port Block RAM with Two Write Ports
// File: rams_tdp_rf_rf.v

module rams_tdp_rf_rf
(clka,clkb,ena,enb,wea,web,addra,addrb,dia,dib,doa,dob);

input clka,clkb,ena,enb,wea,web;
input [9:0] addra,addrb;
input [15:0] dia,dib;
output [15:0] doa,dob;
reg[15:0] ram [1023:0];
reg[15:0] doa,dob;

always @(posedge clka)
begin
  if (ena)
    begin
      if (wea)
        ram[addra] <= dia;
      doa <= ram[addra];
    end
end

always @(posedge clkb)
begin
  if (enb)
```

```
    begin
      if (web)
        ram[addrb] <= dib;
      dob <= ram[addrb];
    end
end

endmodule
```

True Dual-Port Block RAM Examples

Dual-Port Block RAM with Two Write Ports in Read-First Mode (VHDL)

```
-- Dual-Port Block RAM with Two Write Ports
-- Correct Modelization with a Shared Variable
-- File: rams_tdp_rf_rf.vhd

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity rams_tdp_rf_rf is
 port(
  clka  : in  std_logic;
  clkb  : in  std_logic;
  ena   : in  std_logic;
  enb   : in  std_logic;
  wea   : in  std_logic;
  web   : in  std_logic;
  addra : in  std_logic_vector(9 downto 0);
  addrb : in  std_logic_vector(9 downto 0);
  dia   : in  std_logic_vector(15 downto 0);
  dib   : in  std_logic_vector(15 downto 0);
  doa   : out std_logic_vector(15 downto 0);
  dob   : out std_logic_vector(15 downto 0)
 );
end rams_tdp_rf_rf;

architecture syn of rams_tdp_rf_rf is
 type ram_type is array (1023 downto 0) of std_logic_vector(15 downto
0);
 shared variable RAM : ram_type;
begin
 process(CLKA)
 begin
  if CLKA'event and CLKA = '1' then
   if ENA = '1' then
    DOA <= RAM(conv_integer(ADDRA));
```

```
      if WEA = '1' then
       RAM(conv_integer(ADDRA)) := DIA;
      end if;
    end if;
  end if;
 end process;

 process(CLKB)
 begin
  if CLKB'event and CLKB = '1' then
   if ENB = '1' then
    DOB <= RAM(conv_integer(ADDRB));
    if WEB = '1' then
     RAM(conv_integer(ADDRB)) := DIB;
    end if;
   end if;
  end if;
 end process;

end syn;
```

Block RAM with Optional Output Registers (Verilog)

```
// Block RAM with Optional Output Registers
// File: rams_pipeline

module rams_pipeline (clk1, clk2, we, en1, en2, addr1, addr2, di,
res1, res2);
input clk1;
input clk2;
input we, en1, en2;
input [9:0] addr1;
input [9:0] addr2;
input [15:0] di;
output [15:0] res1;
output [15:0] res2;
reg[15:0] res1;
reg[15:0] res2;
reg[15:0] RAM [1023:0];
reg[15:0] do1;
reg[15:0] do2;

always @(posedge clk1)
begin
    if (we == 1'b1)
        RAM[addr1] <= di;
    do1 <= RAM[addr1];
end
```

```
always @(posedge clk2)
begin
    do2 <= RAM[addr2];
end

always @(posedge clk1)
begin
    if (en1 == 1'b1)
        res1 <= do1;
end

always @(posedge clk2)
begin
    if (en2 == 1'b1)
        res2 <= do2;
end
endmodule
```

Block RAM with Optional Output Registers (VHDL)

```
-- Block RAM with Optional Output Registers
-- File: rams_pipeline.vhd
library IEEE;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rams_pipeline is
 port(
  clk1, clk2   : in  std_logic;
  we, en1, en2 : in  std_logic;
  addr1        : in  std_logic_vector(9 downto 0);
  addr2        : in  std_logic_vector(9 downto 0);
  di           : in  std_logic_vector(15 downto 0);
  res1         : out std_logic_vector(15 downto 0);
  res2         : out std_logic_vector(15 downto 0)
 );
end rams_pipeline;

architecture beh of rams_pipeline is
 type ram_type is array (1023 downto 0) of std_logic_vector(15 downto
0);
 signal ram : ram_type;
 signal do1 : std_logic_vector(15 downto 0);
 signal do2 : std_logic_vector(15 downto 0);
begin
 process(clk1)
```

```
  begin
    if rising_edge(clk1) then
      if we = '1' then
        ram(conv_integer(addr1)) <= di;
      end if;
      do1 <= ram(conv_integer(addr1));
    end if;
  end process;

  process(clk2)
  begin
    if rising_edge(clk2) then
      do2 <= ram(conv_integer(addr2));
    end if;
  end process;

  process(clk1)
  begin
    if rising_edge(clk1) then
      if en1 = '1' then
        res1 <= do1;
      end if;
    end if;
  end process;

  process(clk2)
  begin
    if rising_edge(clk2) then
      if en2 = '1' then
        res2 <= do2;
      end if;
    end if;
  end process;

end beh;
```

Byte Write Enable (Block RAM)

Xilinx supports byte write enable in block RAM.

Use byte write enable in block RAM to:

- Exercise advanced control over writing data into RAM
- Separately specify the writeable portions of 8 bits of an addressed memory

From the standpoint of HDL modeling and inference, the concept is best described as a column-based write:

- The RAM is seen as a collection of equal size columns
- During a write cycle, you separately control writing into each of these columns

Vivado synthesis inference lets you take advantage of the block RAM byte write enable feature. The described RAM is implemented on block RAM resources, using the byte write enable capability, provided that the following requirements are met:

- Write columns of equal widths
- Allowed write column widths: 8-bit, 9-bit, 16-bit, 18-bit (multiple of 8-bit or 9-bit)

For other write column widths, such as 5-bit or 12-bit (non multiple of 8-bit or 9-bit), Vivado synthesis uses separate rams for each column:

- Number of write columns: any
- Supported read-write synchronizations: read-first, write-first, no-change

True-Dual-Port Block RAM with Byte Write Enable Examples

Byte Write Enable—READ_FIRST Mode (Verilog)

```
// True-Dual-Port BRAM with Byte-wide Write Enable
//      Read-First mode
// bytewrite_tdp_ram_rf.v
//

module bytewrite_tdp_ram_rf
 #(
//-----------------------------------------------------------------
---------
parameter   NUM_COL             =   4,
parameter   COL_WIDTH           =   8,
parameter   ADDR_WIDTH          =  10,
// Addr  Width in bits : 2 *ADDR_WIDTH = RAM Depth
parameter   DATA_WIDTH      =  NUM_COL*COL_WIDTH  // Data  Width in
bits

//-----------------------------------------------------------------
-----
  ) (
     input clkA,
     input enaA,
     input [NUM_COL-1:0] weA,
     input [ADDR_WIDTH-1:0] addrA,
     input [DATA_WIDTH-1:0] dinA,
```

```
      output reg [DATA_WIDTH-1:0] doutA,

      input clkB,
      input enaB,
      input [NUM_COL-1:0] weB,
      input [ADDR_WIDTH-1:0] addrB,
      input [DATA_WIDTH-1:0] dinB,
      output reg [DATA_WIDTH-1:0] doutB
      );


   // Core Memory
   reg [DATA_WIDTH-1:0]   ram_block [(2**ADDR_WIDTH)-1:0];

   integer                i;
   // Port-A Operation
   always @ (posedge clkA) begin
      if(enaA) begin
         for(i=0;i<NUM_COL;i=i+1) begin
            if(weA[i]) begin
               ram_block[addrA][i*COL_WIDTH +: COL_WIDTH] <=
dinA[i*COL_WIDTH +: COL_WIDTH];
            end
         end
         doutA <= ram_block[addrA];
      end
   end

   // Port-B Operation:
   always @ (posedge clkB) begin
      if(enaB) begin
         for(i=0;i<NUM_COL;i=i+1) begin
            if(weB[i]) begin
               ram_block[addrB][i*COL_WIDTH +: COL_WIDTH] <=
dinB[i*COL_WIDTH +: COL_WIDTH];
            end
         end

         doutB <= ram_block[addrB];
      end
   end

endmodule // bytewrite_tdp_ram_rf
```

Byte Write Enable—READ_FIRST Mode (VHDL)

```
-- True-Dual-Port BRAM with Byte-wide Write Enable
--  Read First mode
--
-- bytewrite_tdp_ram_rf.vhd
--
-- READ_FIRST ByteWide WriteEnable Block RAM Template

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bytewrite_tdp_ram_rf is
 generic(
  SIZE       : integer := 1024;
  ADDR_WIDTH : integer := 10;
  COL_WIDTH  : integer := 9;
  NB_COL     : integer := 4
 );

 port(
  clka  : in  std_logic;
  ena   : in  std_logic;
  wea   : in  std_logic_vector(NB_COL - 1 downto 0);
  addra : in  std_logic_vector(ADDR_WIDTH - 1 downto 0);
  dia   : in  std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
  doa   : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
  clkb  : in  std_logic;
  enb   : in  std_logic;
  web   : in  std_logic_vector(NB_COL - 1 downto 0);
  addrb : in  std_logic_vector(ADDR_WIDTH - 1 downto 0);
  dib   : in  std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
  dob   : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0)
 );

end bytewrite_tdp_ram_rf;

architecture byte_wr_ram_rf of bytewrite_tdp_ram_rf is
 type ram_type is array (0 to SIZE - 1) of std_logic_vector(NB_COL *
COL_WIDTH - 1 downto 0);
 shared variable RAM : ram_type := (others => (others => '0'));

begin

 ------- Port A-------
 process(clka)
 begin
  if rising_edge(clka) then
```

```
    if ena = '1' then
      doa <= RAM(conv_integer(addra));
      for i in 0 to NB_COL - 1 loop
        if wea(i) = '1' then
          RAM(conv_integer(addra))((i + 1) * COL_WIDTH - 1 downto i *
COL_WIDTH) := dia((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
        end if;
      end loop;
    end if;
  end if;
 end process;

 ------- Port B-------
 process(clkb)
 begin
  if rising_edge(clkb) then
    if enb = '1' then
      dob <= RAM(conv_integer(addrb));
      for i in 0 to NB_COL - 1 loop
        if web(i) = '1' then
          RAM(conv_integer(addrb))((i + 1) * COL_WIDTH - 1 downto i *
COL_WIDTH) := dib((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
        end if;
      end loop;
    end if;
  end if;
 end process;
end byte_wr_ram_rf;
```

Byte Write Enable—WRITE_FIRST Mode (VHDL)

```
-- True-Dual-Port BRAM with Byte-wide Write Enable
--  Write First mode
--
-- bytewrite_tdp_ram_wf.vhd
-- WRITE_FIRST ByteWide WriteEnable Block RAM Template

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bytewrite_tdp_ram_wf is
 generic(
  SIZE       : integer := 1024;
  ADDR_WIDTH : integer := 10;
  COL_WIDTH  : integer := 9;
  NB_COL     : integer := 4
 );
```

```
 port(
  clka  : in  std_logic;
  ena   : in  std_logic;
  wea   : in  std_logic_vector(NB_COL - 1 downto 0);
  addra : in  std_logic_vector(ADDR_WIDTH - 1 downto 0);
  dia   : in  std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
  doa   : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
  clkb  : in  std_logic;
  enb   : in  std_logic;
  web   : in  std_logic_vector(NB_COL - 1 downto 0);
  addrb : in  std_logic_vector(ADDR_WIDTH - 1 downto 0);
  dib   : in  std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
  dob   : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0)
 );

end bytewrite_tdp_ram_wf;

architecture byte_wr_ram_wf of bytewrite_tdp_ram_wf is
 type ram_type is array (0 to SIZE - 1) of std_logic_vector(NB_COL *
COL_WIDTH - 1 downto 0);
 shared variable RAM : ram_type := (others => (others => '0'));

begin

 ------- Port A-------
 process(clka)
 begin
  if rising_edge(clka) then
   if ena = '1' then
    for i in 0 to NB_COL - 1 loop
     if wea(i) = '1' then
      RAM(conv_integer(addra))((i + 1) * COL_WIDTH - 1 downto i *
COL_WIDTH) := dia((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
     end if;
    end loop;
    doa <= RAM(conv_integer(addra));
   end if;
  end if;

 end process;

 ------- Port B-------
 process(clkb)
 begin
  if rising_edge(clkb) then
   if enb = '1' then
    for i in 0 to NB_COL - 1 loop
     if web(i) = '1' then
```

```
          RAM(conv_integer(addrb))((i + 1) * COL_WIDTH - 1 downto i *
COL_WIDTH) := dib((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
        end if;
       end loop;
       dob <= RAM(conv_integer(addrb));
     end if;
   end if;
  end process;
end byte_wr_ram_wf;
```

Byte-Wide Write Enable—NO_CHANGE Mode (Verilog)

```
//
// True-Dual-Port BRAM with Byte-wide Write Enable
//      No-Change mode
//
// bytewrite_tdp_ram_nc.v
//
// ByteWide Write Enable, - NO_CHANGE mode template - Vivado recomended
module bytewrite_tdp_ram_nc
  #(
    //-----------------------------------------------------------------
    parameter   NUM_COL                 =   4,
    parameter   COL_WIDTH               =   8,
    parameter   ADDR_WIDTH              =  10, // Addr  Width in bits :
2**ADDR_WIDTH = RAM Depth
    parameter   DATA_WIDTH             =  NUM_COL*COL_WIDTH  // Data  Width
in bits
    //-----------------------------------------------------------------
    ) (
       input clkA,
       input enaA,
       input [NUM_COL-1:0] weA,
       input [ADDR_WIDTH-1:0] addrA,
       input [DATA_WIDTH-1:0] dinA,
       output reg [DATA_WIDTH-1:0] doutA,

       input clkB,
       input enaB,
       input [NUM_COL-1:0] weB,
       input [ADDR_WIDTH-1:0] addrB,
       input [DATA_WIDTH-1:0] dinB,
       output reg [DATA_WIDTH-1:0] doutB
       );


   // Core Memory
   reg [DATA_WIDTH-1:0]            ram_block [(2**ADDR_WIDTH)-1:0];

   // Port-A Operation
   generate
```

```
      genvar                          i;
      for(i=0;i<NUM_COL;i=i+1) begin
         always @ (posedge clkA) begin
            if(enaA) begin
               if(weA[i]) begin
                  ram_block[addrA][i*COL_WIDTH +: COL_WIDTH] <=
dinA[i*COL_WIDTH +: COL_WIDTH];
               end
            end
         end
      end
   endgenerate

   always @ (posedge clkA) begin
      if(enaA) begin
         if (~|weA)
           doutA <= ram_block[addrA];
      end
   end


   // Port-B Operation:
   generate
      for(i=0;i<NUM_COL;i=i+1) begin
         always @ (posedge clkB) begin
            if(enaB) begin
               if(weB[i]) begin
                  ram_block[addrB][i*COL_WIDTH +: COL_WIDTH] <=
dinB[i*COL_WIDTH +: COL_WIDTH];
               end
            end
         end
      end
   endgenerate

   always @ (posedge clkB) begin
      if(enaB) begin
         if (~|weB)
           doutB <= ram_block[addrB];
      end
   end

endmodule // bytewrite_tdp_ram_nc
```

Byte-Wide Write Enable—NO_CHANGE Mode (VHDL)

```
--
-- True-Dual-Port BRAM with Byte-wide Write Enable
--  No change mode
--
-- bytewrite_tdp_ram_nc.vhd
--
-- NO_CHANGE ByteWide WriteEnable Block RAM Template

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bytewrite_tdp_ram_nc is
 generic(
  SIZE       : integer := 1024;
  ADDR_WIDTH : integer := 10;
  COL_WIDTH  : integer := 9;
  NB_COL     : integer := 4
 );

 port(
  clka  : in  std_logic;
  ena   : in  std_logic;
  wea   : in  std_logic_vector(NB_COL - 1 downto 0);
  addra : in  std_logic_vector(ADDR_WIDTH - 1 downto 0);
  dia   : in  std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
  doa   : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
  clkb  : in  std_logic;
  enb   : in  std_logic;
  web   : in  std_logic_vector(NB_COL - 1 downto 0);
  addrb : in  std_logic_vector(ADDR_WIDTH - 1 downto 0);
  dib   : in  std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
  dob   : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0)
 );

end bytewrite_tdp_ram_nc;

architecture byte_wr_ram_nc of bytewrite_tdp_ram_nc is
 type ram_type is array (0 to SIZE - 1) of std_logic_vector(NB_COL *
COL_WIDTH - 1 downto 0);
 shared variable RAM : ram_type := (others => (others => '0'));

begin

 ------- Port A-------
 process(clka)
 begin
```

```
    if rising_edge(clka) then
     if ena = '1' then
      if (wea = (wea'range => '0')) then
       doa <= RAM(conv_integer(addra));
      end if;
      for i in 0 to NB_COL - 1 loop
       if wea(i) = '1' then
        RAM(conv_integer(addra))((i + 1) * COL_WIDTH - 1 downto i *
COL_WIDTH) := dia((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
       end if;
      end loop;
     end if;
    end if;
  end process;

  ------- Port B-------
  process(clkb)
  begin
    if rising_edge(clkb) then
     if enb = '1' then
      if (web = (web'range => '0')) then
       dob <= RAM(conv_integer(addrb));
      end if;
      for i in 0 to NB_COL - 1 loop
       if web(i) = '1' then
        RAM(conv_integer(addrb))((i + 1) * COL_WIDTH - 1 downto i *
COL_WIDTH) := dib((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
       end if;
      end loop;
     end if;
    end if;
  end process;
end byte_wr_ram_nc;
```

Asymmetric RAMs

Simple Dual-Port Asymmetric RAM When Read is Wider than Write (VHDL)

```
-- Asymmetric port RAM
-- Read Wider than Write
-- asym_ram_sdp_read_wider.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asym_ram_sdp_read_wider is
 generic(
  WIDTHA     : integer := 4;
  SIZEA      : integer := 1024;
  ADDRWIDTHA : integer := 10;
  WIDTHB     : integer := 16;
  SIZEB      : integer := 256;
  ADDRWIDTHB : integer := 8
 );

 port(
  clkA  : in  std_logic;
  clkB  : in  std_logic;
  enA   : in  std_logic;
  enB   : in  std_logic;
  weA   : in  std_logic;
  addrA : in  std_logic_vector(ADDRWIDTHA - 1 downto 0);
  addrB : in  std_logic_vector(ADDRWIDTHB - 1 downto 0);
  diA   : in  std_logic_vector(WIDTHA - 1 downto 0);
  doB   : out std_logic_vector(WIDTHB - 1 downto 0)
 );

end asym_ram_sdp_read_wider;

architecture behavioral of asym_ram_sdp_read_wider is
 function max(L, R : INTEGER) return INTEGER is
 begin
  if L > R then
   return L;
  else
   return R;
  end if;
 end;

 function min(L, R : INTEGER) return INTEGER is
 begin
```

```
  if L < R then
   return L;
  else
   return R;
  end if;
 end;

 function log2(val : INTEGER) return natural is
  variable res : natural;
 begin
  for i in 0 to 31 loop
   if (val <= (2 ** i)) then
    res := i;
    exit;
   end if;
  end loop;
  return res;
 end function Log2;

 constant minWIDTH : integer := min(WIDTHA, WIDTHB);
 constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
 constant maxSIZE  : integer := max(SIZEA, SIZEB);
 constant RATIO    : integer := maxWIDTH / minWIDTH;

 -- An asymmetric RAM is modeled in a similar way as a symmetric RAM,
with an
 -- array of array object. Its aspect ratio corresponds to the port
with the
 -- lower data width (larger depth)
 type ramType is array (0 to maxSIZE - 1) of
std_logic_vector(minWIDTH - 1 downto 0);

 signal my_ram : ramType := (others => (others => '0'));

 signal readB : std_logic_vector(WIDTHB - 1 downto 0) := (others =>
'0');
 signal regA  : std_logic_vector(WIDTHA - 1 downto 0) := (others =>
'0');
 signal regB  : std_logic_vector(WIDTHB - 1 downto 0) := (others =>
'0');

begin

 -- Write process
 process(clkA)
 begin
  if rising_edge(clkA) then
   if enA = '1' then
    if weA = '1' then
     my_ram(conv_integer(addrA)) <= diA;
```

```
      end if;
    end if;
  end if;
 end process;

 -- Read process
 process(clkB)
 begin
  if rising_edge(clkB) then
   for i in 0 to RATIO - 1 loop
    if enB = '1' then
     readB((i + 1) * minWIDTH - 1 downto i * minWIDTH) <=
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO))));
    end if;
   end loop;
   regB <= readB;
  end if;
 end process;

 doB <= regB;

end behavioral;
```

Simple Dual-Port Asymmetric Ram When Read is Wider than Write (Verilog)

```
// Asymmetric port RAM
// Read Wider than Write. Read Statement in loop
//asym_ram_sdp_read_wider.v

module asym_ram_sdp_read_wider (clkA, clkB, enaA, weA, enaB, addrA,
addrB, diA, doB);
parameter WIDTHA = 4;
parameter SIZEA = 1024;
parameter ADDRWIDTHA = 10;

parameter WIDTHB = 16;
parameter SIZEB = 256;
parameter ADDRWIDTHB = 8;
input clkA;
input clkB;
input weA;
input enaA, enaB;
input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
output [WIDTHB-1:0] doB;
`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}
```

```
function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
 if (value < 2)
  log2 = value;
 else
 begin
  shifted = value-1;
  for (res=0; shifted>0; res=res+1)
   shifted = shifted>>1;
  log2 = res;
 end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
localparam minWIDTH = `min(WIDTHA, WIDTHB);

localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHB-1:0] readB;

always @(posedge clkA)
begin
 if (enaA) begin
  if (weA)
   RAM[addrA] <= diA;
 end
end


always @(posedge clkB)
begin : ramread
 integer i;
 reg [log2RATIO-1:0] lsbaddr;
 if (enaB) begin
  for (i = 0; i < RATIO; i = i+1) begin
   lsbaddr = i;
   readB[(i+1)*minWIDTH-1 -: minWIDTH] <= RAM[{addrB, lsbaddr}];
  end
 end
end
assigndoB = readB;
```

```
endmodule
```

Simple Dual-Port Asymmetric RAM When Write is Wider than Read (Verilog)

```
// Asymmetric port RAM
// Write wider than Read. Write Statement in a loop.
// asym_ram_sdp_write_wider.v

module asym_ram_sdp_write_wider (clkA, clkB, weA, enaA, enaB, addrA,
addrB, diA, doB);
parameter WIDTHB = 4;
parameter SIZEB = 1024;
parameter ADDRWIDTHB = 10;

parameter WIDTHA = 16;
parameter SIZEA = 256;
parameter ADDRWIDTHA = 8;
input clkA;
input clkB;
input weA;
input enaA, enaB;
input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
output [WIDTHB-1:0] doB;
`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
  if (value < 2)
    log2 = value;
  else
  begin
    shifted = value-1;
    for (res=0; shifted>0; res=res+1)
      shifted = shifted>>1;
    log2 = res;
  end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
```

```
localparam minWIDTH = `min(WIDTHA, WIDTHB);

localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHB-1:0] readB;

always @(posedge clkB) begin
 if (enaB) begin
  readB <= RAM[addrB];
 end
end
assign doB = readB;

always @(posedge clkA)
begin : ramwrite
 integer i;
 reg [log2RATIO-1:0] lsbaddr;
 for (i=0; i< RATIO; i= i+ 1) begin : write1
  lsbaddr = i;
  if (enaA) begin
   if (weA)
    RAM[{addrA, lsbaddr}] <= diA[(i+1)*minWIDTH-1 -: minWIDTH];
  end
 end
end

endmodule
```

Simple Dual-Port Asymmetric RAM Read First (VHDL)

```
-- asymmetric port RAM
-- True Dual port read first
-- asym_ram_tdp_read_first_first.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asym_ram_tdp_read_first is
 generic(
  WIDTHA     : integer := 4;
  SIZEA      : integer := 1024;
  ADDRWIDTHA : integer := 10;
  WIDTHB     : integer := 16;
  SIZEB      : integer := 256;
```

```
    ADDRWIDTHB : integer := 8
  );

  port(
    clkA  : in  std_logic;
    clkB  : in  std_logic;
    enA   : in  std_logic;
    enB   : in  std_logic;
    weA   : in  std_logic;
    weB   : in  std_logic;
    addrA : in  std_logic_vector(ADDRWIDTHA - 1 downto 0);
    addrB : in  std_logic_vector(ADDRWIDTHB - 1 downto 0);
    diA   : in  std_logic_vector(WIDTHA - 1 downto 0);
    diB   : in  std_logic_vector(WIDTHB - 1 downto 0);
    doA   : out std_logic_vector(WIDTHA - 1 downto 0);
    doB   : out std_logic_vector(WIDTHB - 1 downto 0)
  );

end asym_ram_tdp_read_first;

architecture behavioral of asym_ram_tdp_read_first is
  function max(L, R : INTEGER) return INTEGER is
  begin
    if L > R then
      return L;
    else
      return R;
    end if;
  end;

  function min(L, R : INTEGER) return INTEGER is
  begin
    if L < R then
      return L;
    else
      return R;
    end if;
  end;

  function log2(val : INTEGER) return natural is
    variable res : natural;
  begin
    for i in 0 to 31 loop
      if (val <= (2 ** i)) then
        res := i;
        exit;
      end if;
    end loop;
    return res;
  end function Log2;
```

```
  constant minWIDTH : integer := min(WIDTHA, WIDTHB);
  constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
  constant maxSIZE  : integer := max(SIZEA, SIZEB);
  constant RATIO    : integer := maxWIDTH / minWIDTH;

  -- An asymmetric RAM is modeled in a similar way as a symmetric RAM,
with an
  -- array of array object. Its aspect ratio corresponds to the port
with the
  -- lower data width (larger depth)
  type ramType is array (0 to maxSIZE - 1) of
std_logic_vector(minWIDTH - 1 downto 0);

  signal my_ram : ramType := (others => (others => '0'));

  signal readA : std_logic_vector(WIDTHA - 1 downto 0) := (others =>
'0');
  signal readB : std_logic_vector(WIDTHB - 1 downto 0) := (others =>
'0');
  signal regA  : std_logic_vector(WIDTHA - 1 downto 0) := (others =>
'0');
  signal regB  : std_logic_vector(WIDTHB - 1 downto 0) := (others =>
'0');

begin
  process(clkA)
  begin
    if rising_edge(clkA) then
      if enA = '1' then
        readA <= my_ram(conv_integer(addrA));
        if weA = '1' then
          my_ram(conv_integer(addrA)) <= diA;
        end if;
      end if;
      regA <= readA;
    end if;
  end process;

  process(clkB)
  begin
    if rising_edge(clkB) then
      for i in 0 to RATIO - 1 loop
        if enB = '1' then
          readB((i + 1) * minWIDTH - 1 downto i * minWIDTH) <=
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO))));
          if weB = '1' then
            my_ram(conv_integer(addrB & conv_std_logic_vector(i,
log2(RATIO)))) <= diB((i + 1) * minWIDTH - 1 downto i * minWIDTH);
          end if;
```

```
      end if;
    end loop;
    regB <= readB;
  end if;
 end process;

 doA <= regA;
 doB <= regB;

end behavioral;
```

True Dual-Port Asymmetric RAM Read First (Verilog)

```
// Asymetric RAM - TDP
// READ_FIRST MODE.
// asym_ram_tdp_read_first.v


module asym_ram_tdp_read_first (clkA, clkB, enaA, weA, enaB, weB,
addrA, addrB, diA, doA, diB, doB);
parameter WIDTHB = 4;
parameter SIZEB = 1024;
parameter ADDRWIDTHB = 10;
parameter WIDTHA = 16;
parameter SIZEA = 256;
parameter ADDRWIDTHA = 8;
input clkA;
input clkB;
input weA, weB;
input enaA, enaB;

input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
input [WIDTHB-1:0] diB;

output [WIDTHA-1:0] doA;
output [WIDTHB-1:0] doB;

`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
 if (value < 2)
```

```
    log2 = value;
  else
  begin
    shifted = value-1;
    for (res=0; shifted>0; res=res+1)
      shifted = shifted>>1;
    log2 = res;
  end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
localparam minWIDTH = `min(WIDTHA, WIDTHB);

localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHA-1:0] readA;
reg [WIDTHB-1:0] readB;

always @(posedge clkB)
begin
  if (enaB) begin
    readB <= RAM[addrB] ;
    if (weB)
      RAM[addrB] <= diB;
  end
end


always @(posedge clkA)
begin : portA
  integer i;
  reg [log2RATIO-1:0] lsbaddr ;
    for (i=0; i< RATIO; i= i+ 1) begin
      lsbaddr = i;
    if (enaA) begin
      readA[(i+1)*minWIDTH -1 -: minWIDTH] <= RAM[{addrA, lsbaddr}];

      if (weA)
        RAM[{addrA, lsbaddr}] <= diA[(i+1)*minWIDTH-1 -: minWIDTH];
    end
  end
end

assign doA = readA;
assign doB = readB;
```

```
endmodule
```

True Dual-Port Asymmetric RAM Read First (VHDL)

```
-- asymmetric port RAM
-- True Dual port read first
-- asym_ram_tdp_read_first_first.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asym_ram_tdp_read_first is
 generic(
  WIDTHA     : integer := 4;
  SIZEA      : integer := 1024;
  ADDRWIDTHA : integer := 10;
  WIDTHB     : integer := 16;
  SIZEB      : integer := 256;
  ADDRWIDTHB : integer := 8
 );

 port(
  clkA   : in  std_logic;
  clkB   : in  std_logic;
  enA    : in  std_logic;
  enB    : in  std_logic;
  weA    : in  std_logic;
  weB    : in  std_logic;
  addrA : in  std_logic_vector(ADDRWIDTHA - 1 downto 0);
  addrB : in  std_logic_vector(ADDRWIDTHB - 1 downto 0);
  diA    : in  std_logic_vector(WIDTHA - 1 downto 0);
  diB    : in  std_logic_vector(WIDTHB - 1 downto 0);
  doA    : out std_logic_vector(WIDTHA - 1 downto 0);
  doB    : out std_logic_vector(WIDTHB - 1 downto 0)
 );

end asym_ram_tdp_read_first;

architecture behavioral of asym_ram_tdp_read_first is
 function max(L, R : INTEGER) return INTEGER is
 begin
  if L > R then
   return L;
  else
   return R;
  end if;
```

```
  end;

  function min(L, R : INTEGER) return INTEGER is
  begin
    if L < R then
      return L;
    else
      return R;
    end if;
  end;

  function log2(val : INTEGER) return natural is
    variable res : natural;
  begin
    for i in 0 to 31 loop
      if (val <= (2 ** i)) then
        res := i;
        exit;
      end if;
    end loop;
    return res;
  end function Log2;

  constant minWIDTH : integer := min(WIDTHA, WIDTHB);
  constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
  constant maxSIZE  : integer := max(SIZEA, SIZEB);
  constant RATIO    : integer := maxWIDTH / minWIDTH;

  -- An asymmetric RAM is modeled in a similar way as a symmetric RAM,
with an
  -- array of array object. Its aspect ratio corresponds to the port
with the
  -- lower data width (larger depth)
  type ramType is array (0 to maxSIZE - 1) of
std_logic_vector(minWIDTH - 1 downto 0);

  signal my_ram : ramType := (others => (others => '0'));

  signal readA : std_logic_vector(WIDTHA - 1 downto 0) := (others =>
'0');
  signal readB : std_logic_vector(WIDTHB - 1 downto 0) := (others =>
'0');
  signal regA  : std_logic_vector(WIDTHA - 1 downto 0) := (others =>
'0');
  signal regB  : std_logic_vector(WIDTHB - 1 downto 0) := (others =>
'0');

begin
  process(clkA)
  begin
```

```
    if rising_edge(clkA) then
     if enA = '1' then
      readA <= my_ram(conv_integer(addrA));
      if weA = '1' then
       my_ram(conv_integer(addrA)) <= diA;
      end if;
     end if;
     regA <= readA;
    end if;
  end process;

  process(clkB)
  begin
    if rising_edge(clkB) then
     for i in 0 to RATIO - 1 loop
      if enB = '1' then
       readB((i + 1) * minWIDTH - 1 downto i * minWIDTH) <=
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO))));
       if weB = '1' then
        my_ram(conv_integer(addrB & conv_std_logic_vector(i,
log2(RATIO)))) <= diB((i + 1) * minWIDTH - 1 downto i * minWIDTH);
       end if;
      end if;
     end loop;
     regB <= readB;
    end if;
  end process;

  doA <= regA;
  doB <= regB;

end behavioral;
```

True Dual-Port Asymmetric RAM Write First (Verilog)

```
// Asymmetric port RAM - TDP
// WRITE_FIRST MODE.
// asym_ram_tdp_write_first.v


module asym_ram_tdp_write_first (clkA, clkB, enaA, weA, enaB, weB,
addrA, addrB, diA, doA, diB, doB);
parameter WIDTHB = 4;
parameter SIZEB = 1024;
parameter ADDRWIDTHB = 10;
parameter WIDTHA = 16;
parameter SIZEA = 256;
parameter ADDRWIDTHA = 8;
```

```
input clkA;
input clkB;
input weA, weB;
input enaA, enaB;

input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
input [WIDTHB-1:0] diB;

output [WIDTHA-1:0] doA;
output [WIDTHB-1:0] doB;

`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
  if (value < 2)
    log2 = value;
  else
  begin
    shifted = value-1;
    for (res=0; shifted>0; res=res+1)
      shifted = shifted>>1;
    log2 = res;
  end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
localparam minWIDTH = `min(WIDTHA, WIDTHB);

localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHA-1:0] readA;
reg [WIDTHB-1:0] readB;

always @(posedge clkB)
begin
  if (enaB) begin
    if (weB)
      RAM[addrB] = diB;
    readB = RAM[addrB] ;
```

```
  end
end


always @(posedge clkA)
begin : portA
  integer i;
  reg [log2RATIO-1:0] lsbaddr ;
    for (i=0; i< RATIO; i= i+ 1) begin
      lsbaddr = i;
      if (enaA) begin

      if (weA)
        RAM[{addrA, lsbaddr}] = diA[(i+1)*minWIDTH-1 -: minWIDTH];

      readA[(i+1)*minWIDTH -1 -: minWIDTH] = RAM[{addrA, lsbaddr}];
    end
  end
end

assign doA = readA;
assign doB = readB;

endmodule
```

True Dual-Port Asymmetric RAM Write First (VHDL)

```
--Asymmetric RAM
--True Dual Port write first mode.
--asym_ram_tdp_write_first.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asym_ram_tdp_write_first is
  generic(
    WIDTHA     : integer := 4;
    SIZEA      : integer := 1024;
    ADDRWIDTHA : integer := 10;
    WIDTHB     : integer := 16;
    SIZEB      : integer := 256;
    ADDRWIDTHB : integer := 8
  );

  port(
    clkA   : in  std_logic;
    clkB   : in  std_logic;
    enA    : in  std_logic;
```

```
    enB   : in  std_logic;
    weA   : in  std_logic;
    weB   : in  std_logic;
    addrA : in  std_logic_vector(ADDRWIDTHA - 1 downto 0);
    addrB : in  std_logic_vector(ADDRWIDTHB - 1 downto 0);
    diA   : in  std_logic_vector(WIDTHA - 1 downto 0);
    diB   : in  std_logic_vector(WIDTHB - 1 downto 0);
    doA   : out std_logic_vector(WIDTHA - 1 downto 0);
    doB   : out std_logic_vector(WIDTHB - 1 downto 0)
  );

end asym_ram_tdp_write_first;

architecture behavioral of asym_ram_tdp_write_first is
  function max(L, R : INTEGER) return INTEGER is
  begin
    if L > R then
      return L;
    else
      return R;
    end if;
  end;

  function min(L, R : INTEGER) return INTEGER is
  begin
    if L < R then
      return L;
    else
      return R;
    end if;
  end;

  function log2(val : INTEGER) return natural is
    variable res : natural;
  begin
    for i in 0 to 31 loop
      if (val <= (2 ** i)) then
        res := i;
        exit;
      end if;
    end loop;
    return res;
  end function Log2;

  constant minWIDTH : integer := min(WIDTHA, WIDTHB);
  constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
  constant maxSIZE  : integer := max(SIZEA, SIZEB);
  constant RATIO    : integer := maxWIDTH / minWIDTH;

  -- An asymmetric RAM is modeled in a similar way as a symmetric RAM, with an
  -- array of array object. Its aspect ratio corresponds to the port with the
  -- lower data width (larger depth)
  type ramType is array (0 to maxSIZE - 1) of std_logic_vector(minWIDTH - 1
downto 0);
```

```
  signal my_ram : ramType := (others => (others => '0'));

  signal readA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
  signal readB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');
  signal regA  : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
  signal regB  : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');

begin
  process(clkA)
  begin
    if rising_edge(clkA) then
      if enA = '1' then
        if weA = '1' then
          my_ram(conv_integer(addrA)) <= diA;
          readA                       <= diA;
        else
          readA <= my_ram(conv_integer(addrA));
        end if;
      end if;
      regA <= readA;
    end if;
  end process;

  process(clkB)
  begin
    if rising_edge(clkB) then
      for i in 0 to RATIO - 1 loop
        if enB = '1' then
          if weB = '1' then
            my_ram(conv_integer(addrB & conv_std_logic_vector(i,
log2(RATIO)))) <= diB((i + 1) * minWIDTH - 1 downto i * minWIDTH);
          end if;
          -- The read statement below is placed after the write statement -- on
purpose
          -- to ensure write-first synchronization through the variable
          -- mechanism
          readB((i + 1) * minWIDTH - 1 downto i * minWIDTH) <=
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO))));
        end if;
      end loop;
      regB <= readB;
    end if;
  end process;

  doA <= regA;
  doB <= regB;

end behavioral;
```

Initializing RAM Contents

RAM can be initialized in following ways:

- Specifying RAM Initial Contents in the HDL Source Code
- Specifying RAM Initial Contents in an External Data File

Specifying RAM Initial Contents in the HDL Source Code

Use the signal default value mechanism to describe initial RAM contents directly in the HDL source code.

VHDL Coding Examples

```
type ram_type is array (0 to 31) of std_logic_vector(19 downto 0);
signal RAM : ram_type :=
(
 X”0200A”, X”00300”, X”08101”, X”04000”, X”08601”, X”0233A”,
X”00300”, X”08602”, X”02310”, X”0203B”, X”08300”, X”04002”,
X”08201”, X”00500”, X”04001”, X”02500”, X”00340”, X”00241”,
X”04002”, X”08300”, X”08201”, X”00500”, X”08101”, X”00602”,
X”04003”, X”0241E”, X”00301”, X”00102”, X”02122”, X”02021”, X”0030D”,
X"08201"
);
```

All bit positions are initialized to the same value:

```
type ram_type is array (0 to 127) of std_logic_vector (15 downto 0);
signal RAM : ram_type := (others => (others => '0'));
```

Verilog Coding Example

All addressable words are initialized to the same value.

```
reg [DATA_WIDTH-1:0] ram [DEPTH-1:0];
integer i;
initial for (i=0; i<DEPTH; i=i+1) ram[i] = 0;
end
```

Specifying RAM Initial Contents in an External Data File

Use the file read function in the HDL source code to load the RAM initial contents from an external data file.

- The external data file is an ASCII text file with any name.
- Each line in the external data file describes the initial content at an address position in the RAM.
- There must be as many lines in the external data file as there are rows in the RAM array. An insufficient number of lines is flagged.
- The addressable position related to a given line is defined by the direction of the primary range of the signal modeling the RAM.
- You can represent RAM content in either binary or hexadecimal. You cannot mix both.
- The external data file cannot contain any other content, such as comments.
- The following external data file initializes an 8 x 32-bit RAM with binary values:

```
00001110110000011001111011000110
00101011001011010101001000100011
01110100010100011000011100001111
01000001010000100101001110010100
00001001101001111111101000101011
00101101001011111110101010100111
11101111000100111000111101101101
10001111010010011001000011101111
00000001100011100011110010011111
11011111001110101011111001001010
11100111010100111110110011001010
11000100001001101100111100101001
10001011100101011111111111100001
11110101110110010000010110111010
01001011000000111001010110101110
11100001111111001010111010011110
01101111011010010100001101110001
01010100011011111000011000100100
11110000111101101111001100001011
10101101001111010100100100011100
01011100001010111111101110101110
01011101000100100111010010110101
11110111000100000101011101101101
11100111110001111010101100001101
01110100000011101111111000011111
00010011110101111000111001011101
01101110001111100011010101101111
10111100000000010011101011011011
11000001001101001101111100010000
00011111110010110110011111010101
```

```
01100100100000011100100101110000
10001000000100111011001010001111
11001000100011101001010001100001
10000000100111010011100111100011
11011111010010100010101010000111
10000000110111101000111110111011
10110011010111101111000110011001
00010111100001001010110111011100
10011100101110101111011010110011
01010011101101010001110110011010
01111011011100010101000101000001
10001000000110010110111001101010
11101000001101010000111001010110
11100011111100000111110101110101
01001010000000001111111101101111
00100011000011001000000010001111
10011000111010110001001011100100
11111111111011110101000101000111
11000011000101000011100110100000
01101101001011111010100011101001
10000111101100101001110011010111
11010110100100101110110010100100
01001111111001101101011111001011
11011001001101110110000100110111
10110110110111100101110011100110
10011100111001000010111111010110
00000000001011011111001010110010
10100110011010000010001000011011
11001010111111001001110001110101
00100001100010000111000101001000
00111100101111110001101101111010
11000010001010000000010100100001
11000001000110001101000101001110
10010011010100010001100100100111
```

Verilog Example

Use a `$readmemb` or `$readmemh` system task to load respectively binary-formatted or hexadecimal data.

```
reg [31:0] ram [0:63];

initial begin
  $readmemb("rams_20c.data", ram, 0, 63);
end
```

VHDL Example

Load the data as follows:

```
type RamType is array(0 to 7) of bit_vector(31 downto 0);
impure function InitRamFromFile (RamFileName : in string) return
RamType is
FILE RamFile : text is in RamFileName;
variable RamFileLine : line;
variable RAM : RamType;
begin
for I in RamType'range loop
readline (RamFile, RamFileLine);
read (RamFileLine, RAM(I));
end loop;
return RAM;
end function;

signal RAM : RamType := InitRamFromFile("rams_20c.data");
```

Initializing Block RAM (Verilog)

```
// Initializing Block RAM (Single-Port Block RAM)
// File: rams_sp_rom
module rams_sp_rom (clk, we, addr, di, dout);
input clk;
input we;
input [5:0] addr;
input [19:0] di;
output [19:0] dout;

reg [19:0] ram [63:0];
reg [19:0] dout;

initial
begin
  ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
  ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
  ram[57] = 20'h00300; ram[56] = 20'h08602; ram[55] = 20'h02310;
  ram[54] = 20'h0203B; ram[53] = 20'h08300; ram[52] = 20'h04002;
  ram[51] = 20'h08201; ram[50] = 20'h00500; ram[49] = 20'h04001;
  ram[48] = 20'h02500; ram[47] = 20'h00340; ram[46] = 20'h00241;
  ram[45] = 20'h04002; ram[44] = 20'h08300; ram[43] = 20'h08201;
  ram[42] = 20'h00500; ram[41] = 20'h08101; ram[40] = 20'h00602;
  ram[39] = 20'h04003; ram[38] = 20'h0241E; ram[37] = 20'h00301;
  ram[36] = 20'h00102; ram[35] = 20'h02122; ram[34] = 20'h02021;
  ram[33] = 20'h00301; ram[32] = 20'h00102; ram[31] = 20'h02222;
  ram[30] = 20'h04001; ram[29] = 20'h00342; ram[28] = 20'h0232B;
  ram[27] = 20'h00900; ram[26] = 20'h00302; ram[25] = 20'h00102;
```

```
  ram[24] = 20'h04002; ram[23] = 20'h00900; ram[22] = 20'h08201;
  ram[21] = 20'h02023; ram[20] = 20'h00303; ram[19] = 20'h02433;
  ram[18] = 20'h00301; ram[17] = 20'h04004; ram[16] = 20'h00301;
  ram[15] = 20'h00102; ram[14] = 20'h02137; ram[13] = 20'h02036;
  ram[12] = 20'h00301; ram[11] = 20'h00102; ram[10] = 20'h02237;
  ram[9]  = 20'h04004; ram[8]  = 20'h00304; ram[7] = 20'h04040;
  ram[6]  = 20'h02500; ram[5]  = 20'h02500; ram[4] = 20'h02500;
  ram[3]  = 20'h0030D; ram[2]  = 20'h02341; ram[1] = 20'h08201;
  ram[0]  = 20'h0400D;
end

always @(posedge clk)
begin
  if (we)
    ram[addr] <= di;
  dout <= ram[addr];
end

endmodule
```

Initializing Block RAM (VHDL)

```
-- Initializing Block RAM (Single-Port Block RAM)
-- File: rams_sp_rom.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_sp_rom is
  port(
    clk  : in  std_logic;
    we   : in  std_logic;
    addr : in  std_logic_vector(5 downto 0);
    di   : in  std_logic_vector(19 downto 0);
    do   : out std_logic_vector(19 downto 0)
  );
end rams_sp_rom;

architecture syn of rams_sp_rom is
  type ram_type is array (63 downto 0) of std_logic_vector(19 downto 0);
  signal RAM : ram_type := (X"0200A", X"00300", X"08101", X"04000", X"08601",
X"0233A",
                        X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                        X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                        X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                        X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                        X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                        X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                        X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                        X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                        X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                         X"0030D", X"02341", X"08201", X"0400D");
```

```
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if we = '1' then
        RAM(conv_integer(addr)) <= di;
      end if;
      do <= RAM(conv_integer(addr));
    end if;
  end process;

end syn;
```

Initializing Block RAM From an External Data File (Verilog)

```
// Initializing Block RAM from external data file
// Binary data
// File: rams_init_file.v

module rams_init_file (clk, we, addr, din, dout);
input clk;
input we;
input [5:0] addr;
input [31:0] din;
output [31:0] dout;

reg [31:0] ram [0:63];
reg [31:0] dout;

initial begin
$readmemb("rams_init_file.data",ram);
end

always @(posedge clk)
begin
  if (we)
     ram[addr] <= din;
  dout <= ram[addr];
end endmodule
```

Initializing Block RAM From an External Data File (VHDL)

```
-- Initializing Block RAM from external data file
-- File: rams_init_file.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
use std.textio.all;

entity rams_init_file is
 port(
  clk  : in  std_logic;
  we   : in  std_logic;
  addr : in  std_logic_vector(5 downto 0);
  din  : in  std_logic_vector(31 downto 0);
  dout : out std_logic_vector(31 downto 0)
 );
end rams_init_file;

architecture syn of rams_init_file is
 type RamType is array (0 to 63) of bit_vector(31 downto 0);

 impure function InitRamFromFile(RamFileName : in string) return
RamType is
  FILE RamFile : text is in RamFileName;
  variable RamFileLine : line;
  variable RAM         : RamType;
 begin
  for I in RamType'range loop
   readline(RamFile, RamFileLine);
   read(RamFileLine, RAM(I));
  end loop;
  return RAM;
 end function;

 signal RAM : RamType := InitRamFromFile("rams_init_file.data");
begin
 process(clk)
 begin
  if clk'event and clk = '1' then
   if we = '1' then
    RAM(conv_integer(addr)) <= to_bitvector(din);
   end if;
   dout <= to_stdlogicvector(RAM(conv_integer(addr)));
  end if;
 end process;

end syn;
```

Black Boxes

A design can contain EDIF files generated by:

- Synthesis tools

- Schematic text editors
- Any other design entry mechanism

These modules must be instantiated to be connected to the rest of the design.

Use `BLACK_BOX` instantiation in the HDL source code.

Vivado synthesis lets you apply specific constraints to these `BLACK_BOX` instantiations.

After you make a design a `BLACK_BOX`, each instance of that design is a `BLACK_BOX`.

Download the coding example files from: Coding Examples.

BLACK_BOX (Verilog)

```
// Black Box
// black_box_1.v
//
(* black_box *) module black_box1 (in1, in2, dout);
input in1, in2;
output dout;
endmodule

module black_box_1 (DI_1, DI_2, DOUT);
input DI_1, DI_2;
output DOUT;

black_box1 U1 (
                .in1(DI_1),
                .in2(DI_2),
                .dout(DOUT)
              );

endmodule
```

BLACK_BOX (VHDL)

```
-- Black Box
-- black_box_1.vhd
library ieee;
use ieee.std_logic_1164.all;

entity black_box_1 is
 port(DI_1, DI_2 : in  std_logic;
    DOUT       : out std_logic);
end black_box_1;
architecture rtl of black_box_1 is
 component black_box1
  port(I1 : in  std_logic;
     I2 : in  std_logic;
```

```
      O  : out std_logic);
  end component;

  attribute black_box : string;
  attribute black_box of black_box1 : component is "yes";

begin
  U1 : black_box1 port map(I1 => DI_1, I2 => DI_2, O => DOUT);
end rtl;
```

FSM Components

Vivado Synthesis Features

- Specific inference capabilities for synchronous Finite State Machine (FSM) components.
- Built-in FSM encoding strategies to accommodate your optimization goals.
- FSM extraction is enabled by default.
- Use `-fsm_extraction off` to disable FSM extraction.

FSM Description

Vivado synthesis supports specification of Finite State Machine (FSM) in both Moore and Mealy form. An FSM consists of the following:

- A state register
- A next state function
- An outputs function

FSM Diagrams

The following diagram shows an FSM representation that incorporates Mealy and Moore machines.

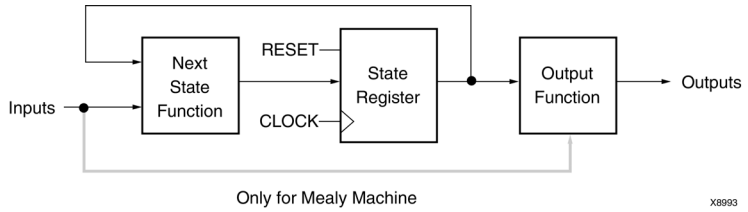

Figure 3-3: **FSM Representation Incorporating Mealy and Moore Machines Diagram**

The following diagram shows an FSM diagram with three processes.

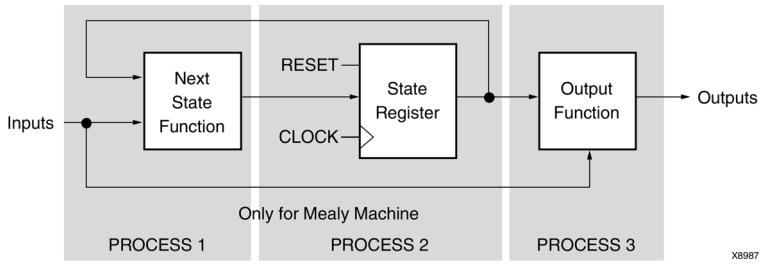

Figure 3-4: **FSM With Three Processes Diagram**

FSM Registers

- Specify a reset or power-up state for Vivado synthesis to identify a Finite State Machine (FSM) or set the value of `FSM_ENCODING` to `"none"`.
- The State Register can be asynchronously or synchronously reset to a particular state.

RECOMMENDED: *Use synchronous reset logic over asynchronous reset logic for an FSM.*

Auto State Encoding

When `FSM_ENCODING` is set to `"auto"`, the Vivado synthesis attempts to select the best-suited encoding method for a given FSM.

One-Hot State Encoding

One-Hot State encoding has the following attributes:

- Is the default encoding scheme for a state machine, up to 32 states.
- Is usually a good choice for optimizing speed or reducing power dissipation.
- Assigns a distinct bit of code to each FSM state.
- Implements the State Register with one flip-flop for each state.
- In a given clock cycle during operation, only one bit of the State Register is asserted.
- Only two bits toggle during a transition between two states.

Gray State Encoding

Gray State encoding has the following attributes:

- Guarantees that only one bit switches between two consecutive states.
- Is appropriate for controllers exhibiting long paths without branching.
- Minimizes hazards and glitches.

- Can be used to minimize power dissipation.

Johnson State Encoding

Johnson State encoding is beneficial when using state machines containing long paths with no branching (as in Gray State Encoding).

Sequential State Encoding

Sequential State encoding has the following attributes:

- Identifies long paths
- Applies successive radix two codes to the states on these paths.
- Minimizes next state equations.

FSM Example (Verilog)

```
// State Machine with single sequential block
//fsm_1.v
module fsm_1(clk,reset,flag,sm_out);
input clk,reset,flag;
output reg sm_out;

parameter s1 = 3'b000;
parameter s2 = 3'b001;
parameter s3 = 3'b010;
parameter s4 = 3'b011;
parameter s5 = 3'b111;

reg [2:0] state;

always@(posedge clk)
  begin
    if(reset)
      begin
        state <= s1;
        sm_out  <= 1'b1;
      end
  else
    begin
     case(state)
       s1: if(flag)
            begin
              state <= s2;
              sm_out <= 1'b1;
            end
           else
            begin
              state <= s3;
              sm_out <= 1'b0;
            end
       s2: begin state <= s4; sm_out <= 1'b0; end
```

```
        s3: begin state <= s4; sm_out <= 1'b0; end
        s4: begin state <= s5; sm_out <= 1'b1; end
        s5: begin state <= s1; sm_out <= 1'b1; end
      endcase
    end
 end
endmodule
```

FSM Example with Single Sequential Block (VHDL)

```
-- State Machine with single sequential block
-- File: fsm_1.vhd
library IEEE;
use IEEE.std_logic_1164.all;

entity fsm_1 is
  port(
    clk, reset, flag : IN  std_logic;
    sm_out           : OUT std_logic
  );
end entity;

architecture behavioral of fsm_1 is
  type state_type is (s1, s2, s3, s4, s5);
  signal state : state_type;
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if (reset = '1') then
        state  <= s1;
        sm_out <= '1';

      else
        case state is
          when s1 => if flag = '1' then
             state  <= s2;
             sm_out <= '1';

           else
             state  <= s3;
             sm_out <= '0';

           end if;
          when s2 => state <= s4;
            sm_out       <= '0';
          when s3 => state <= s4;
            sm_out       <= '0';
          when s4 => state <= s5;
            sm_out       <= '1';
          when s5 => state <= s1;
            sm_out       <= '1';
```

```
          end case;
        end if;
      end if;
    end process;

end behavioral;
```

FSM Reporting

The Vivado synthesis flags INFO messages in the log file, giving information about Finite State Machine (FSM) components and their encoding. The following are example messages:

```
INFO: [Synth 8-802] inferred FSM for state register 'state_reg' in module 'fsm_test'
INFO: [Synth 8-3354] encoded FSM with state register 'state_reg' using encoding 'sequential'
in module 'fsm_test'
```

ROM HDL Coding Techniques

Read-Only Memory (ROM) closely resembles Random Access Memory (RAM) with respect to HDL modeling and implementation. Use the `ROM_STYLE` attribute to implement a properly-registered ROM on block RAM resources. See ROM_STYLE for more information.

ROM Using Block RAM Resources (Verilog)

```
// ROMs Using Block RAM Resources.
// File: rams_sp_rom_1.v
//
module rams_sp_rom_1 (clk, en, addr, dout);
inputclk;
inputen;
input[5:0] addr;
output  [19:0] dout;

(*rom_style = "block" *) reg [19:0] data;

always @(posedge clk)
begin
  if (en)
    case(addr)
      6'b000000: data <= 20'h0200A;6'b100000: data <= 20'h02222;
      6'b000001: data <= 20'h00300;6'b100001: data <= 20'h04001;
      6'b000010: data <= 20'h08101;6'b100010: data <= 20'h00342;
      6'b000011: data <= 20'h04000;6'b100011: data <= 20'h0232B;
      6'b000100: data <= 20'h08601;6'b100100: data <= 20'h00900;
      6'b000101: data <= 20'h0233A;6'b100101: data <= 20'h00302;
      6'b000110: data <= 20'h00300;6'b100110: data <= 20'h00102;
```

```
        6'b000111: data <= 20'h08602;6'b100111: data <= 20'h04002;
        6'b001000: data <= 20'h02310;6'b101000: data <= 20'h00900;
        6'b001001: data <= 20'h0203B;6'b101001: data <= 20'h08201;
        6'b001010: data <= 20'h08300;6'b101010: data <= 20'h02023;
        6'b001011: data <= 20'h04002;6'b101011: data <= 20'h00303;
        6'b001100: data <= 20'h08201;6'b101100: data <= 20'h02433;
        6'b001101: data <= 20'h00500;6'b101101: data <= 20'h00301;
        6'b001110: data <= 20'h04001;6'b101110: data <= 20'h04004;
        6'b001111: data <= 20'h02500;6'b101111: data <= 20'h00301;
        6'b010000: data <= 20'h00340;6'b110000: data <= 20'h00102;
        6'b010001: data <= 20'h00241;6'b110001: data <= 20'h02137;
        6'b010010: data <= 20'h04002;6'b110010: data <= 20'h02036;
        6'b010011: data <= 20'h08300;6'b110011: data <= 20'h00301;
        6'b010100: data <= 20'h08201;6'b110100: data <= 20'h00102;
        6'b010101: data <= 20'h00500;6'b110101: data <= 20'h02237;
        6'b010110: data <= 20'h08101;6'b110110: data <= 20'h04004;
        6'b010111: data <= 20'h00602;6'b110111: data <= 20'h00304;
        6'b011000: data <= 20'h04003;6'b111000: data <= 20'h04040;
        6'b011001: data <= 20'h0241E;6'b111001: data <= 20'h02500;
        6'b011010: data <= 20'h00301;6'b111010: data <= 20'h02500;
        6'b011011: data <= 20'h00102;6'b111011: data <= 20'h02500;
        6'b011100: data <= 20'h02122;6'b111100: data <= 20'h0030D;
        6'b011101: data <= 20'h02021;6'b111101: data <= 20'h02341;
        6'b011110: data <= 20'h00301;6'b111110: data <= 20'h08201;
        6'b011111: data <= 20'h00102;6'b111111: data <= 20'h0400D;
    endcase
end

assign dout = data;

endmodule
```

ROM Inference on an Array (VHDL)

```
-- ROM Inference on array
-- File: roms_1.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity roms_1 is
  port(
    clk  : in  std_logic;
    en   : in  std_logic;
    addr : in  std_logic_vector(5 downto 0);
    data : out std_logic_vector(19 downto 0)
  );
end roms_1;

architecture behavioral of roms_1 is
  type rom_type is array (63 downto 0) of std_logic_vector(19 downto 0);
```

```
  signal ROM : rom_type := (X"0200A", X"00300", X"08101", X"04000", X"08601",
X"0233A",
                           X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                           X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                           X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                           X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                           X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                           X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                           X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                           X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                           X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                            X"0030D", X"02341", X"08201", X"0400D");
  attribute rom_style : string;
  attribute rom_style of ROM : signal is "block";

begin
  process(clk)
  begin
    if rising_edge(clk) then
      if (en = '1') then
        data <= ROM(conv_integer(addr));
      end if;
    end if;
  end process;

end behavioral;
```

VHDL Support

Introduction

This chapter describes the supported VHDL language constructs in Vivado synthesis and notes any exceptions to support.

VHDL compactly describes complicated logic, and lets you:

- Describe the structure of a system: how the system is decomposed into subsystems, and how those subsystems are interconnected.
- Specify the function of a system using familiar programming language forms.
- Simulate a system design before it is implemented and programmed in hardware.
- Produce a detailed, device-dependent version of a design to be synthesized from a more abstract specification.

For more information, see the *IEEE VHDL Language Reference Manual* (LRM).

Supported and Unsupported VHDL Data Types

Some VHDL data types are part of predefined packages. For information on where they are compiled, and how to load them, see VHDL Predefined Packages.

The type is defined in the IEEE `std_logic_1164` package.

Unsupported Data Types

VHDL supports the `real` type defined in the standard package for calculations only, such as the calculation of generics values.

IMPORTANT: *You cannot define a synthesizable object of type `real`.*

VHDL Data Types

VHDL Predefined Enumerated Types

Vivado synthesis supports the following predefined VHDL enumerated types for hardware description as listed in the following table.

Table 4-1: **VHDL Enumerated Type Summary**

Enumerated Type	Defined In	Allowed Values
bit	standard package	0 (logic zero) 1 (logic 1)
boolean	standard package	false true
std_logic	IEEE std_logic_1164 package	See the *std_logic Allowed Values*.

std_logic Allowed Values

Table 4-2: **std_logic Allowed Values**

Value	Meaning	What Vivado synthesis does
U	initialized	Not accepted by Vivado synthesis
X	unknown	Treated as don't care
0	low	Treated as logic zero
1	high	Treated as logic one
Z	high impedance	Treated as high impedance
W	weak unknown	Not accepted by Vivado synthesis
L	weak low	Treated identically to 0
H	weak high	Treated identically to 1
-	don't care	Treated as don't care

Supported Overloaded Enumerated Types

Table 4-3: **Supported Overloaded Enumerated Types**

Type	Defined In IEEE Package	SubType Of	Contains Values
`std_ulogic`	std_logic_1164	N/A	Same values as `std_logic` Does not contain predefined resolution functions
`X01`	std_logic_1164	std_ulogic	`X, 0, 1`
`X01Z`	std_logic_1164	std_ulogic	`X, 0, 1, Z`
`UX01`	std_logic_1164	std_ulogic	`U, X, 0, 1`
`UX01Z`	std_logic_1164	std_ulogic	`U, X, 0, Z`

VHDL User-Defined Enumerated Types

You can create your own enumerated types. User-defined enumerated types usually describe the states of a finite state machine (FSM).

VHDL User-Defined Enumerated Types Coding Example

```
type STATES is (START, IDLE, STATE1, STATE2, STATE3) ;
```

Supported VHDL Bit Vector Types

Table 4-4: **Supported VHDL Bit Vector Types**

Type	Defined In Package	Models
`bit_vector`	Standard	Vector of bit elements
`std_logic_vector`	IEEE std_logic_1164	Vector of std_logic elements

Supported VHDL Overloaded Types

Table 4-5: **Supported VHDL Overloaded Types**

Type	Defined In IEEE Package
`std_ulogic_vector`	std_logic_1164
`unsigned`	std_logic_arith
`signed`	std_logic_arith

VHDL Integer Types

The integer type is a predefined VHDL type. Vivado synthesis implements an integer on 32 bits by default. For a more compact implementation, define the exact range of applicable values, where: `type MSB is range 8 to 15.`

You can also take advantage of the predefined natural and positive types, overloading the integer type.

VHDL Multi-Dimensional Array Types

Vivado synthesis supports VHDL multi-dimensional array types.

RECOMMENDED: *Although there is no restriction on the number of dimensions, describe no more than three dimensions.*

Objects of multi-dimensional array type can be passed to functions and used in component instantiations. Objects of multi-dimensional array type that you can describe are: signals, constants, and variables.

Fully Constrained Array Type Coding Example

An array type must be fully constrained in all dimensions.

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB12 is array (11 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB12;
```

Array Declared as a Matrix Coding Example

You can declare an array as a matrix.

```
subtype TAB13 is array (7 downto 0,4 downto 0) of STD_LOGIC_VECTOR (8
downto 0);
```

Multi-Dimensional Array Signals and Variables Coding Examples

The following coding examples demonstrate the uses of multi-dimensional array signals and variables in assignments.

1. Make the following declarations:

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB05 is array (4 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB05;
signal WORD_A : WORD8;
signal TAB_A, TAB_B : TAB05;
signal TAB_C, TAB_D : TAB03;
constant CNST_A : TAB03 := (
("00000000","01000001","01000010","10000011","00001100"),
("00100000","00100001","00101010","10100011","00101100"),
("01000010","01000010","01000100","01000111","01000100"));
```

2. You can now specify:
   - A multi-dimensional array signal or variable:

     ```
     TAB_A <= TAB_B; TAB_C <= TAB_D; TAB_C <= CNST_A;
     ```

   - An index of one array:

     ```
     TAB_A (5) <= WORD_A; TAB_C (1) <= TAB_A;
     ```

   - Indexes of the maximum number of dimensions:

     ```
     TAB_A (5) (0) <= '1'; TAB_C (2) (5) (0) <= '0'
     ```

   - A slice of the first array

     ```
     TAB_A (4 downto 1) <= TAB_B (3 downto 0);
     ```

   - An index of a higher level array and a slice of a lower level array:

     ```
     TAB_C (2) (5) (3 downto 0) <= TAB_B (3) (4 downto 1); TAB_D (0) (4)
     (2 downto 0)
     \\ <= CNST_A (5 downto 3)
     ```

3. Add the following declaration:

   ```
   subtype MATRIX15 is array(4 downto 0, 2 downto 0) of STD_LOGIC_VECTOR
   (7 downto 0);
   signal MATRIX_A : MATRIX15;
   ```

4. You can now specify:
   - A multi-dimensional array signal or variable:

     ```
     MATRIXA <= CNST_A
     ```

   - An index of one row of the array:

     ```
     MATRIXA (5) <= TAB_A;
     ```

   - Indexes of the maximum number of dimensions

     ```
     MATRIXA (5,0) (0) <= '1';
     ```

Note: Indexes can be variable.

VHDL Record Types Code Example

- A field of a record type can also be of type Record.
- Constants can be record types.
- Record types cannot contain attributes.
- Vivado synthesis supports aggregate assignments to record signals.

The following code snippet is an example:

```
type mytype is record field1 : std_logic;
 field2 : std_logic_vector (3 downto 0);
end record;
```

VHDL Objects

VHDL objects include: Signals, Variables, Constants and Operators.

Signals

Declare a VHDL signal in:

- An architecture declarative part: Use the VHDL signal anywhere within that architecture.
- A block: Use the VHDL signal within that block.

Assign the VHDL signal with the <= signal assignment operator.

```
signal sig1 : std_logic;
sig1 <= '1';
```

Variables

A VHDL variable is:

- Declared in a process or a subprogram.
- Used within that process or subprogram.
- Assigned with the := assignment operator.

```
variable var1 : std_logic_vector (7 downto 0); var1 := "01010011";
```

Constants

You can declare a VHDL constant in any declarative region. The constant is used within that region. You cannot change the constant values after they are declared.

```
signal sig1 : std_logic_vector (5 downto 0);constant init0 :
std_logic_vector (5 downto 0) := "010111";sig1 <= init0;
```

Operators

Vivado synthesis supports VHDL operators.

Shift Operator Examples

Table 4-6: **Shift Operator Examples**

Operator	Example	Logically Equivalent To
SLL (Shift Left Logic)	sig1 <= A(4 downto 0) sll 2	sig1 <= A(2 downto 0) & "00";
SRL (Shift Right Logic)	sig1 <= A(4 downto 0) srl 2	sig1 <= "00" & A(4 downto 2);
SLA (Shift Left Arithmetic)	sig1 <= A(4 downto 0) srl 2	sig1 <= A(2 downto 0) & A(0) & A(0);
SRA (Shift Right Arithmetic)	sig1 <= A(4 downto 0) sra 2	sig1 <= <= A(4) & A(4) & A(4 downto 2);
ROL (Rotate Left)	sig1 <= A(4 downto 0) rol 2	sig1 <= A(2 downto 0) & A(4 downto 3);
ROR (Rotate Right)	A(4 downto 0) ror 2	sig1 <= A(1 downto 0) & A(4 downto 2);

VHDL Entity and Architecture Descriptions

VHDL Circuit Descriptions

A VHDL circuit description (design unit) consists of the following:

- Entity declaration: Provides the external view of the circuit. Describes objects visible from the outside, including the circuit interface, such as the I/O ports and generics.
- Architecture: Provides the internal view of the circuit, and describes the circuit behavior or structure.

VHDL Entity Declarations

The I/O ports of the circuit are declared in the entity. Each port has a:

- name
- mode (in, out, inout, buffer)
- type

Constrained and Unconstrained Ports

When defining a port, the port:

- Can be constrained or unconstrained.
- Are usually constrained.

- Can be left unconstrained in the entity declaration.
  - If ports are left unconstrained, their width is defined at instantiation when the connection is made between formal ports and actual signals.
  - Unconstrained ports allow you to create different instantiations of the same entity, defining different port widths.

RECOMMENDED: *Do not use unconstrained ports. Define ports that are constrained through generics. Apply different values of those generics at instantiation. Do not have an unconstrained port on the top-level entity.*

Array types of more than one-dimension are not accepted as ports.

The entity declaration can also declare VHDL generics.

Buffer Port Mode

RECOMMENDED: *Do not use buffer port mode.*

VHDL allows buffer port mode when a signal is used both internally, and as an output port when there is only one internal driver. Buffer ports are a potential source of errors during synthesis, and complicate validation of post-synthesis results through simulation.

NOT RECOMMENDED Coding Example WITH Buffer Port Mode

```
entity alu is
 port(
   CLK : in STD_LOGIC;
   A : inSTD_LOGIC_VECTOR(3 downto 0);
   B : inSTD_LOGIC_VECTOR(3 downto 0);
   C : buffer STD_LOGIC_VECTOR(3 downto 0));
end alu;

architecture behavioral of alu is
begin
 process begin
   if rising_edge(CLK) then
     C <= UNSIGNED(A) + UNSIGNED(B) UNSIGNED(C);
   end if;
 end process;
end behavioral;
```

Dropping Buffer Mode

RECOMMENDED: *Drop buffer port mode.*

In the previous coding example, signal C: was modeled with a buffer mode, and is used both internally and as an output port. Every level of hierarchy that can be connected to C must also be declared as a buffer.

To drop buffer mode:

1. Insert a dummy signal.
2. Declare port C as an output.

RECOMMENDED Coding Example WITHOUT Buffer Port Mode

```
entity alu is
port(
  CLK : in STD_LOGIC;
  A : in STD_LOGIC_VECTOR(3 downto 0);
  B : in STD_LOGIC_VECTOR(3 downto 0);
  C : out STD_LOGIC_VECTOR(3 downto 0));
end alu;
architecture behavioral of alu is
-- dummy signal
  signal C_INT : STD_LOGIC_VECTOR(3 downto 0);
begin
  C <= C_INT;
  process begin
    if rising_edge(CLK) then
      C_INT <= A and B and C_INT;
    end if;
  end process;
end behavioral;
```

VHDL Architecture Declarations

You can declare internal signals in the architecture. Each internal signal has a name and a type.

VHDL Architecture Declaration Coding Example

```
ibrary IEEE;
use IEEE.std_logic_1164.all;

entity EXAMPLE is
port (
  A,B,C : in std_logic;
  D,E : out std_logic );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
  signal T : std_logic;
```

```
begin
...
end ARCHI;
```

VHDL Component Instantiation

Component instantiation allows you to instantiate one design unit (component) inside another design unit to create a hierarchically structured design description.

To perform component instantiation:

1. Create the design unit (entity and architecture) modeling the functionality to be instantiated.
2. Declare the component to be instantiated in the declarative region of the parent design unit architecture.
3. Instantiate and connect this component in the architecture body of the parent design unit.
4. Map (connect) formal ports of the component to actual signals and ports of the parent design unit.

Elements of Component Instantiation Statement

Vivado synthesis supports unconstrained vectors in component declarations.

The main elements of a component instantiation statement are:

- Label: Identifies the instance.
- Association list: Introduced by the reserved `port map` keyword and ties formal ports of the component to actual signals or ports of the parent design unit. An optional association list is introduced by the reserved `generic map` keyword and provides actual values to formal generics defined in the component.

Component Instantiation (VHDL)

This coding example shows the structural description of a half-Adder composed of four nand2 components.

```
--
-- A simple component instantiation example
--   Involves a component declaration and the component instantiation
itself
--
-- instantiation_simple.vhd
--
entity sub is
  generic(
```

```
    WIDTH : integer := 4
  );
  port(
    A, B : in  BIT_VECTOR(WIDTH - 1 downto 0);
    O    : out BIT_VECTOR(2 * WIDTH - 1 downto 0)
  );
end sub;

architecture archi of sub is
begin
  O <= A & B;
end ARCHI;

entity instantiation_simple is
  generic(
    WIDTH : integer := 2);
  port(
    X, Y : in  BIT_VECTOR(WIDTH - 1 downto 0);
    Z    : out BIT_VECTOR(2 * WIDTH - 1 downto 0));
end instantiation_simple;

architecture ARCHI of instantiation_simple is
  component sub                          -- component declaration
    generic(
      WIDTH : integer := 2);
    port(
      A, B : in  BIT_VECTOR(WIDTH - 1 downto 0);
      O    : out BIT_VECTOR(2 * WIDTH - 1 downto 0));
  end component;

begin
  inst_sub : sub                         -- component instantiation
    generic map(
      WIDTH => WIDTH
    )
    port map(
      A => X,
      B => Y,
      O => Z
    );

end ARCHI;
```

Recursive Component Instantiation

Vivado synthesis supports recursive component instantiation.

Recursive Component Instantiation Example (VHDL)

```
--
-- Recursive component instantiation
--
-- instantiation_recursive.vhd
--
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

entity instantiation_recursive is
 generic(
  sh_st : integer := 4
 );
 port(
  CLK : in  std_logic;
  DI  : in  std_logic;
  DO  : out std_logic
 );
end entity instantiation_recursive;

architecture recursive of instantiation_recursive is
 component instantiation_recursive
  generic(
   sh_st : integer);
  port(
   CLK : in  std_logic;
   DI  : in  std_logic;
   DO  : out std_logic);
 end component;
 signal tmp : std_logic;
begin
 GEN_FD_LAST : if sh_st = 1 generate
  inst_fd : FD port map(D => DI, C => CLK, Q => DO);
 end generate;
 GEN_FD_INTERM : if sh_st /= 1 generate
  inst_fd : FD port map(D => DI, C => CLK, Q => tmp);
  inst_sstage : instantiation_recursive
   generic map(sh_st => sh_st - 1)
   port map(DI => tmp, CLK => CLK, DO => DO);
 end generate;
end recursive;
```

VHDL Component Configuration

A component configuration explicitly links a component with the appropriate model.

- A model is an entity and architecture pair.
- Vivado synthesis supports component configuration in the declarative part of the architecture. The following is an example:

```
for instantiation_list : component_name use
 LibName.entity_Name(Architecture_Name);
```

The following statement indicates that:

- All NAND2 components use the design unit consisting of entity NAND2 and architecture ARCHI.
- The design unit is compiled in the work library.

```
For all : NAND2 use entity work.NAND2(ARCHI);
```

The value of the top module name (`-top`) option in the `synth_design` command is the configuration name instead of the top-level entity name.

VHDL GENERICS

VHDL GENERICs have the following properties:

- Are equivalent to Verilog parameters.
- Help you create scalable design modelizations.
- Let you write compact, factorized VHDL code.
- Let you parameterize functionality such as bus size, and the number of repetitive elements in the design unit.

For the same functionality that must be instantiated multiple times, but with different bus sizes, you need describe only one design unit with generics. See the GENERIC Parameters Example.

Declaring Generics

You can declare generic parameters in the entity declaration part. Supported generics types are: `integer`, `boolean`, `string`, and `real`.

GENERIC Parameters Example

```
-- VHDL generic parameters example
--
-- generics_1.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity addern is
 generic(
  width : integer := 8
 );
 port(
  A, B : in  std_logic_vector(width - 1 downto 0);
  Y    : out std_logic_vector(width - 1 downto 0)
 );
end addern;

architecture bhv of addern is
begin
 Y <= A + B;
end bhv;

Library IEEE;
use IEEE.std_logic_1164.all;

entity generics_1 is
 port(
  X, Y, Z : in  std_logic_vector(12 downto 0);
  A, B    : in  std_logic_vector(4 downto 0);
  S       : out std_logic_vector(17 downto 0));
end generics_1;

architecture bhv of generics_1 is
 component addern
  generic(width : integer := 8);
  port(
   A, B : in  std_logic_vector(width - 1 downto 0);
   Y    : out std_logic_vector(width - 1 downto 0));
 end component;
 for all : addern use entity work.addern(bhv);

 signal C1     : std_logic_vector(12 downto 0);
 signal C2, C3 : std_logic_vector(17 downto 0);
begin
 U1 : addern generic map(width => 13) port map(X, Y, C1);
 C2 <= C1 & A;
```

```
  C3 <= Z & B;
  U2 : addern generic map(width => 18) port map(C2, C3, S);
end bhv;
```

VHDL Combinatorial Circuits

Combinatorial logic is described using concurrent signal assignments that you specify in the body of an architecture. You can describe as many concurrent signal assignments as are necessary; the order of appearance of the concurrent signal assignments in the architecture is irrelevant.

VHDL Concurrent Signal Assignments

Concurrent signal assignments are concurrently active and re-evaluated when any signal on the right side of the assignment changes value. The re-evaluated result is assigned to the signal on the left-hand side.

Supported types of concurrent signal assignments are: Simple Signal Assignment Example, and Concurrent Selection Assignment Example (VHDL).

Simple Signal Assignment Example

```
T <= A and B;
```

Concurrent Selection Assignment Example (VHDL)

```
-- Concurrent selection assignment in VHDL
--
-- concurrent_selected_assignment.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity concurrent_selected_assignment is
  generic(
    width : integer := 8);
  port(
    a, b, c, d : in  std_logic_vector(width - 1 downto 0);
    sel        : in  std_logic_vector(1 downto 0);
    T          : out std_logic_vector(width - 1 downto 0));
end concurrent_selected_assignment;

architecture bhv of concurrent_selected_assignment is
begin
```

```
  with sel select T <=
    a when "00",
    b when "01",
    c when "10",
    d when others;
end bhv;
```

Generate Statements

Generate statements include:

- `for-generate` statements
- `if-generate` statements

`for-generate` Statements

The `for-generate` statements describe repetitive structures.

`for-generate` Statement (VHDL)

In this coding example, the `for-generate` statement describes the calculation of the result and carry out for each bit position of this 8-bit Adder.

```
--
-- A for-generate example
--
-- for_generate.vhd
--
entity for_generate is
  port(
    A, B : in  BIT_VECTOR(0 to 7);
    CIN  : in  BIT;
    SUM  : out BIT_VECTOR(0 to 7);
    COUT : out BIT
  );
end for_generate;

architecture archi of for_generate is
  signal C : BIT_VECTOR(0 to 8);
begin
  C(0) <= CIN;
  COUT <= C(8);
  LOOP_ADD : for I in 0 to 7 generate
    SUM(I)   <= A(I) xor B(I) xor C(I);
    C(I + 1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
  end generate;
```

```
end archi;
```

if-generate Statements

An if-generate statement activates specific parts of the HDL source code based on a test result, and is supported for static (non-dynamic) conditions.

For example, when a generic indicates which device family is being targeted, the if-generate statement tests the value of the generic against a specific device family and activates a section of the HDL source code written specifically for that device family.

for-generate Nested in an if-generate Statement (VHDL)

In this coding example, a generic N-bit Adder with a width ranging between 4 and 32 is described with an if-generate and a for-generate statement.

```
-- A for-generate nested in a if-generate
--
-- if_for_generate.vhd
--
entity if_for_generate is
  generic(
    N : INTEGER := 8
  );
  port(
    A, B : in  BIT_VECTOR(N downto 0);
    CIN  : in  BIT;
    SUM  : out BIT_VECTOR(N downto 0);
    COUT : out BIT
  );
end if_for_generate;

architecture archi of if_for_generate is
  signal C : BIT_VECTOR(N + 1 downto 0);
begin
  IF_N : if (N >= 4 and N <= 32) generate
    C(0) <= CIN;
    COUT <= C(N + 1);
    LOOP_ADD : for I in 0 to N generate
      SUM(I)   <= A(I) xor B(I) xor C(I);
      C(I + 1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
    end generate;
  end generate;
end archi;
```

Combinatorial Processes

You can model VHDL combinatorial logic with a process, which explicitly assigns signals a new value every time the process is executed.

IMPORTANT: *No signals should implicitly retain its current value, and a process can contain local variables.*

Memory Elements

Hardware inferred from a combinatorial process does not involve any memory elements.

A memory element process is combinatorial when all assigned signals in a process are always explicitly assigned in all possible paths within a process block.

A signal that is not explicitly assigned in all branches of an `if` or `case` statement typically leads to a Latch inference.

IMPORTANT: *If Vivado synthesis infers unexpected Latches, review the HDL source code for a signal that is not explicitly assigned.*

Sensitivity List

A combinatorial process has a sensitivity list. The sensitivity list appears within parentheses after the `PROCESS` keyword. A process is activated if an event (value change) appears on one of the sensitivity list signals.

For a combinatorial process, this sensitivity list must contain:

- All signals in conditions (for example, `if` and `case`).
- All signals on the right-hand side of an assignment.

Missing Signals

Signals might be missing from the sensitivity list. If one or more signals is missing from the sensitivity list:

- The synthesis results can differ from the initial design specification.
- Vivado synthesis issues a warning message.
- Vivado synthesis adds the missing signals to the sensitivity list.

IMPORTANT: *To avoid problems during simulation explicitly add all missing signals in the HDL source code and re-run synthesis.*

Variable and Signal Assignments

Vivado synthesis supports VHDL variable and signal assignments. A process can contain local variables, which are declared and used within a process and generally not visible outside the process.

Signal Assignment in a Process Example

```
-- Signal assignment in a process
-- signal_in_process.vhd

entity signal_in_process is
 port(
   A, B : in  BIT;
   S    : out BIT
 );
end signal_in_process;

architecture archi of signal_in_process is
begin
 process(A, B)
 begin
   S <= '0';
   if ((A and B) = '1') then
     S <= '1';
   end if;
 end process;
end archi;
```

Variable and Signal Assignment in a Process Example (VHDL)

```
-- Variable and signal assignment in a process
-- variable_in_process.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity variable_in_process is
 port(
   A, B    : in  std_logic_vector(3 downto 0);
   ADD_SUB : in  std_logic;
```

```
    S       : out std_logic_vector(3 downto 0)
  );
end variable_in_process;

architecture archi of variable_in_process is
begin
  process(A, B, ADD_SUB)
    variable AUX : std_logic_vector(3 downto 0);
  begin
    if ADD_SUB = '1' then
      AUX := A + B;
    else
      AUX := A - B;
    end if;
    S <= AUX;
  end process;
end archi;
```

if-else Statements

The if-else and if-elsif-else statements use TRUE and FALSE conditions to execute statements.

- If the expression evaluates to TRUE, the if branch is executed.
- If the expression evaluates to FALSE, x, or z, the else branch is executed.
  - A block of multiple statements is executed in an if or else branch.
  - begin and end keywords are required.
  - if-else statements can be nested.

if-else Statement Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is port (
  a, b, c, d : in std_logic_vector (7 downto 0);
  sel1, sel2 : in std_logic;
  outmux : out std_logic_vector (7 downto 0));
end mux4;

architecture behavior of mux4 is begin
  process (a, b, c, d, sel1, sel2)
    begin
    if (sel1 = '1') then
    if (sel2 = '1') then
    outmux <= a;
```

```
  else outmux <= b;
   else
  end if;
   if (sel2 = '1') then outmux <= c;
    else
   outmux <= d;
  end if;
end if;
 end process;
end behavior;
```

case Statements

A case statement:

- Performs a comparison to an expression to evaluate one of several parallel branches.
- Evaluates the branches in the order in which they are written.
- Executes the first branch that evaluates to `TRUE`.

If none of the branches match, a `case` statement executes the default branch.

case Statement Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is port (
 a, b, c, d : in std_logic_vector (7 downto 0);
 sel : in std_logic_vector (1 downto 0);
 outmux : out std_logic_vector (7 downto 0));
end mux4;

architecture behavior of mux4 is begin
process (a, b, c, d, sel)
begin
   case sel is
   when "00" => outmux <= a;
   when "01" => outmux <= b;
   when "10" => outmux <= c;
   when others => outmux <= d; -- case statement must be complete
  end case;
  end process;
end behavior;
```

for-loop Statements

Vivado synthesis `for-loop` statements support:

- Constant bounds
- Stop test condition using the following operators: <, <=, >, and >=.
- Next step computations falling within one of the following specifications:
  - var = var + step
  - var = var - step

  Where:

  - `var` is the loop variable
  - `step` is a constant value
- Next and exit statements

for-loop Example

```
--
-- For-loop example
--
-- for_loop.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity for_loop is
  port(
    a     : in  std_logic_vector(7 downto 0);
    Count : out std_logic_vector(2 downto 0)
  );
end for_loop;

architecture behavior of for_loop is
begin
  process(a)
    variable Count_Aux : std_logic_vector(2 downto 0);
  begin
    Count_Aux := "000";
    for i in a'range loop
      if (a(i) = '0') then
        Count_Aux := Count_Aux + 1;
      end if;
    end loop;
    Count <= Count_Aux;
  end process;
end behavior;
```

VHDL Sequential Logic

A VHDL process is sequential (as opposed to combinatorial) when some assigned signals are not explicitly assigned in all paths within the process. The generated hardware has an internal state or memory (Flip-Flops or Latches).

RECOMMENDED: *Use a sensitivity-list based description style to describe sequential logic.*

Describing sequential logic using a process with a sensitivity list includes:

- The clock signal
- Any optional signal controlling the sequential element asynchronously (asynchronous set/reset)
- An if statement that models the clock event.

Sequential Process With a Sensitivity List Syntax

```
process (<sensitivity list>)
begin
  <asynchronous part>
  <clock event>
  <synchronous part>
end;
```

Asynchronous Control Logic Modelization

Modelization of any asynchronous control logic (asynchronous set/reset) is done before the clock event statement.

Modelization of the synchronous logic (data, optional synchronous set/reset, optional clock enable) is done in the `if` branch of the clock event.

Table 4-7: **Asynchronous Control Logic Modelization Summary**

Modelization of	Contains	Performed
Asynchronous control logic	Asynchronous set/reset	Before the clock event statement
Synchronous logic	Data Optional synchronous set/reset Optional clock enable	In the clock event `if` branch.

Clock Event Statements

Describe the clock event statement as:

- Rising edge clock:
  - `if rising_edge (clk) then`
- Falling edge clock:
  - `if falling_edge (clk) then`

Missing Signals

If any signals are missing from the sensitivity list, the synthesis results can differ from the initial design specification. In this case, Vivado synthesis issues a warning message and adds the missing signals to the sensitivity list.

IMPORTANT: *To avoid problems during simulation, explicitly add all missing signals in the HDL source code and re-run synthesis.*

VHDL Sequential Processes Without a Sensitivity List

Vivado synthesis allows the description of a sequential process using a `wait` statement. The sequential process is described without a sensitivity list.

The wait statement is the first statement and the condition in the wait statement describes the sequential logic clock.

IMPORTANT: *The same sequential process cannot have both a sensitivity list and a `wait` statement, and only one `wait` statement is allowed.*

Sequential Process Using a Wait Statement Coding Example

```
process begin
  wait until rising_edge(clk);
  q <= d;
end process;
```

Describing a Clock Enable in the wait Statement Example

You can describe a clock enable (`clken`) in the `wait` statement together with the clock.

```
process begin
  wait until rising_edge(clk) and clken = '1';
  q <= d;
end process;
```

Describing a Clock Enable After the Wait Statement Example

You can describe the clock enable separately, as follows:

```
process begin
  wait until rising_edge(clk);
  if clken = '1' then
    q <= d;
  end if;
end process;
```

Describing Synchronous Control Logic

You can use the same coding method as was shown to describe a clock enable to describe synchronous control logic, such as a synchronous `reset` or `set`.

IMPORTANT: *You cannot describe a sequential element with asynchronous control logic using a process without a sensitivity list. Only a process with a sensitivity list allows such functionality. Vivado synthesis does not allow the description of a Latch based on a wait statement. For greater flexibility, describe synchronous logic using a process with a sensitivity list.*

VHDL Initial Values and Operational Set/Reset

You can initialize registers when you declare them. The initialization value is a constant and can be generated from a function call.

Initializing Registers Example One

This coding example specifies a power-up value in which the sequential element is initialized when the circuit goes live and the circuit global reset is applied.

```
signal arb_onebit   : std_logic := '0';
signal arb_priority : std_logic_vector(3 downto 0) := "1011";
```

Initializing Registers Example Two

This coding example combines power-up initialization and operational reset.

```
--
-- Register initialization
--   Specifying initial contents at circuit powes-up
--   Specifying an operational set/reset
--
-- File: VHDL_Language_Support/initial/initial_1.vhd
--
```

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

entity initial_1 is
 Port(
  clk, rst : in  std_logic;
  din      : in  std_logic;
  dout     : out std_logic
  );
end initial_1;

architecture behavioral of initial_1 is
 signal arb_onebit : std_logic := '1'; -- power-up to vcc
begin
 process(clk)
 begin
  if (rising_edge(clk)) then
   if rst = '1' then           -- local synchronous reset
    arb_onebit <= '0';
   else
    arb_onebit <= din;
   end if;
  end if;
 end process;

 dout <= arb_onebit;

end behavioral;
```

VHDL Functions and Procedures

Use VHDL functions and procedures for blocks that are used multiple times in a design. The content is similar to combinatorial process content

Declare functions and procedures in:

- The declarative part of an entity
- An architecture
- A package

A function or procedure consists of a declarative part and a body.

The declarative part specifies:

- `input` parameters, which can be unconstrained to a given bound.
- `output` and `inout` parameters (procedures only)

IMPORTANT: *Resolution functions are not supported except the function defined in the IEEE* `std_logic_1164` *package.*

Function Declared Within a Package Example (VHDL)

Download the coding example files from: Coding Examples.

This coding example declares an `ADD` function within a package. The `ADD` function is a single-bit `Adder` and is called four times to create a 4-bit `Adder`. The following example uses a function:

```
-- Declaration of a function in a package
--
-- function_package_1.vhd
--
package PKG is
  function ADD(A, B, CIN : BIT) return BIT_VECTOR;
end PKG;

package body PKG is
  function ADD(A, B, CIN : BIT) return BIT_VECTOR is
    variable S, COUT : BIT;
    variable RESULT  : BIT_VECTOR(1 downto 0);
  begin
    S      := A xor B xor CIN;
    COUT   := (A and B) or (A and CIN) or (B and CIN);
    RESULT := COUT & S;
    return RESULT;
  end ADD;
end PKG;

use work.PKG.all;

entity function_package_1 is
  port(
    A, B : in  BIT_VECTOR(3 downto 0);
    CIN  : in  BIT;
    S    : out BIT_VECTOR(3 downto 0);
    COUT : out BIT
  );
end function_package_1;

architecture ARCHI of function_package_1 is
```

```
  signal S0, S1, S2, S3 : BIT_VECTOR(1 downto 0);
begin
  S0   <= ADD(A(0), B(0), CIN);
  S1   <= ADD(A(1), B(1), S0(1));
  S2   <= ADD(A(2), B(2), S1(1));
  S3   <= ADD(A(3), B(3), S2(1));
  S    <= S3(0) & S2(0) & S1(0) & S0(0);
  COUT <= S3(1);
end ARCHI;
```

Procedure Declared Within a Package Example (VHDL)

The following example uses a procedure within a package:

```
-- Declaration of a procedure in a package
--
-- Download: procedure_package_1.vhd
--
package PKG is
  procedure ADD(
    A, B, CIN : in  BIT;
    C         : out BIT_VECTOR(1 downto 0));
end PKG;

package body PKG is
  procedure ADD(
    A, B, CIN : in  BIT;
    C         : out BIT_VECTOR(1 downto 0)) is
    variable S, COUT : BIT;
  begin
    S    := A xor B xor CIN;
    COUT := (A and B) or (A and CIN) or (B and CIN);
    C    := COUT & S;
  end ADD;
end PKG;

use work.PKG.all;

entity procedure_package_1 is
  port(
    A, B : in  BIT_VECTOR(3 downto 0);
    CIN  : in  BIT;
    S    : out BIT_VECTOR(3 downto 0);
    COUT : out BIT
  );
end procedure_package_1;

architecture ARCHI of procedure_package_1 is
```

```
begin
 process(A, B, CIN)
  variable S0, S1, S2, S3 : BIT_VECTOR(1 downto 0);
 begin
  ADD(A(0), B(0), CIN, S0);
  ADD(A(1), B(1), S0(1), S1);
  ADD(A(2), B(2), S1(1), S2);
  ADD(A(3), B(3), S2(1), S3);
  S    <= S3(0) & S2(0) & S1(0) & S0(0);
  COUT <= S3(1);
 end process;
end ARCHI;
```

Recursive Functions Example

Vivado synthesis supports recursive functions. This coding example models an `n!` function.

```
function my_func(x : integer) return integer is begin
 if x = 1 then return x;
 else return (x*my_func(x-1));
 end if;
end function my_func;
```

VHDL Assert Statements

Assert statements are not supported in with the -assert synthesis option.

CAUTION! *Care should be taken using asserts. Vivado can only support static asserts that do not create, or are created, by behavior. For example, performing as assert on a value of a constant or a parameter/generic works; however, as asset on the value of a signal inside an "if" statement will not work.*

VHDL Predefined Packages

Vivado synthesis supports the VHDL predefined packages as defined in the STD and IEEE standard libraries. The libraries are pre-compiled, and need not be user-compiled, and can be directly included in the HDL source code.

VHDL Predefined Standard Packages

VHDL predefined standard packages that are, by default, included, define the following basic VHDL types: `bit`, `bit_vector`, `integer`, `natural`, `real`, and `boolean`.

VHDL IEEE Packages

Vivado synthesis supports the some predefined VHDL IEEE packages, which are pre-compiled in the IEEE library, and the following IEEE packages:

- numeric_bit
  - Unsigned and signed vector types based on bit.
  - Overloaded arithmetic operators, conversion functions, and extended functions for these types.
- std_logic_1164:
  - `std_logic`, `std_ulogic`, `std_logic_vector`, and `std_ulogic_vector` types.
  - Conversion functions based on these types.
- `std_logic_arith` (Synopsys)
  - Unsigned and signed vector types based on `std_logic`.
  - Overloaded arithmetic operators, conversion functions, and extended functions for these types.
- `numeric_std`
  - Unsigned and signed vector types based on `std_logic`.
- Overloaded arithmetic operators, conversion functions, and extended functions for these types. Equivalent to `std_logic_arith`.
- `std_logic_unsigned` (Synopsys)
  - Unsigned arithmetic operators for `std_logic` and `std_logic_vector`
- `std_logic_signed` (Synopsys)
  - Signed arithmetic operators for `std_logic` and `std_logic_vector`
- `std_logic_misc` (Synopsys)
  - Supplemental types, subtypes, constants, and functions for the std_logic_1164 package, such as `and_reduce` and `or_reduce`.

VHDL Predefined IEEE Fixed Point and Floating Point Packages

The IEEE fixed-point package, `fixed_pkg` contains functions for fixed-point math, and is precompiled into the `ieee_proposed` library. Invoke this package using:

```
ieee.std_logic_1164.all;
ieee.numeric_std.all;
library ieee_proposed;
ieee_proposed.fixed_pkg.all;
```

The predefined IEEE floating-point package, `float_pkg`, contains functions for floating-point math, is precompiled into the `ieee_proposed` library.

Invoke this package as follows:

```
ieee.std_logic_1164.all;
ieee.numeric_std.all;
library ieee_proposed;
ieee_proposed.float_pkg.all;
```

VHDL Predefined IEEE Real Type and IEEE Math_Real Packages

VHDL predefined IEEE `real` type and IEEE `math_real` packages are supported only for calculations such as the calculation of generics values, and cannot be used to describe synthesizable functionality.

VHDL Real Number Constants

The following table describes the VHDL real number constants.

Table 4-8: **VHDL Real Number Constants**

Constant	Value	Constant	Value
math_e	E	math_log_of_2	ln2
math_1_over_e	1/e	math_log_of_10	ln10
math_pi	Π	math_log2_of_e	$\log_2$
math_2_pi	2π	math_log10_of_e	$\log_{10}$
math_1_over_pi	1/ π	math_sqrt_2	√2
math_pi_over_2	π/2	math_1_oversqrt_2	1/√2
math_pi_over_3	π/3	math_sqrt_pi	√π
math_pi_over_4	π/4	math_deg_to_rad	2π/360
math_3_pi_over_2	3π/2	math_rad_to_deg	360/2π

VHDL Real Number Functions

The following table describes VHDL real number functions:

Table 4-9: **VHDL Real Number Functions**

ceil(x)	realmax(x,y)	exp(x)	cos(x)	cosh(x)
floor(x)	realmin(x,y)	log(x)	tan(x)	tanh(x)
round(x)	sqrt(x)	log2(x)	arcsin(x)	arcsinh(x)
trunc(x)	cbrt(x)	log10(x)	arctan(x)	arccosh(x)
sign(x)	"**"(n,y)	log(x,y)	arctan(y,x)	arctanh(x)
"mod"(x,y)	"**"(x,y)	sin(x)	sinh(x)	

Defining Your Own VHDL Packages

You can define your own VHDL packages to specify:

- Types and subtypes
- Constants
- Functions and procedures
- Component declarations

Defining a VHDL package permits access to shared definitions and models from other parts of your project and requires the following:

- Package declaration: Declares each of the previously listed elements.
- Package body: Describes the functions and procedures declared in the package declaration.

Package Declaration Syntax

```
package mypackage is
  type mytype is record
    first : integer;
    second : integer;
  end record;
  constant myzero : mytype := (first => 0, second => 0);
  function getfirst (x : mytype) return integer;
end mypackage;

package body mypackage is
  function getfirst (x : mytype) return integer is
  begin
    return x.first;
  end function;
end mypackage;
```

Accessing VHDL Packages

To access a VHDL package:

1. Use a library clause to include the library in which the package is compiled. For example:

   ```
   library library_name;
   ```

2. Designate the package, or a specific definition contained in the package, with a use clause. For example: `use library_name.package_name.all`.
3. Insert these lines immediately before the entity or architecture in which you use the package definitions.

Because the `work` library is the default library, you can omit the library clause if the designated package has been compiled into this library.

VHDL Constructs Support Status

Vivado synthesis supports VHDL design entities and configurations except as noted in the following table.

Table 4-10: **VHDL Constructs and Support Status**

VHDL Construct	Support Status
VHDL Entity Headers	
Generics	Supported
Ports	Supported, including unconstrained ports
Entity Statement Part	**Unsupported**
VHDL Packages	
STANDARD	**Type TIME is not supported**
VHDL Physical Types	
TIME	Ignored
REAL	Supported, but only in functions for constant calculations.
VHDL Modes	
Linkage	**Unsupported**
VHDL Declarations	
Type	Supported for the following: • Enumerated types • Types with positive range having constant bounds • Bit vector types • Multi-dimensional arrays
VHDL Objects	
Constant Declaration	Supported except for deferred constant
Signal Declaration	Supported except for register and bus type signals.
Attribute Declaration	Supported for some attributes, otherwise skipped.
VHDL Specifications	
HIGHLOW	Supported
LEFT	Supported
RIGHT	Supported
RANGE	Supported
REVERSE_RANGE	Supported
LENGTH	Supported
POS	Supported

Table 4-10: **VHDL Constructs and Support Status *(Cont'd)***

VHDL Construct	Support Status
ASCENDING	Supported
Configuration	Supported only with the `all` clause for instances list. • If no clause is added, Vivado synthesis looks for the entity or architecture compiled in the default library.
Disconnection	**Unsupported**
Underscores	Object names can contain underscores in general (DATA_1), but Vivado synthesis does not allow signal names with leading underscores (_DATA_1).
VHDL Operators	
Logical Operators: and, or, nand, nor, xor, xnor, not	Supported
Relational Operators: =, /=, <, <=, >, >=	Supported
& (concatenation)	Supported
Adding Operators: +, -	Supported
*	Supported
/	Supported if the right operand is a constant power of 2, or if both operands are constant.
Rem	Supported if the right operand is a constant power of 2.
Mod	Supported if the right operand is a constant power of 2.
Shift Operators: sll, srl, sla, sra, rol, ror	Supported
Abs	Supported
**	Supported if the left operand is 2.
Sign: +, -	Supported
VHDL Operands	
Abstract Literals	Only integer literals are supported.
Physical Literals	**Ignored**
Enumeration Literals	Supported
String Literals	Supported
Bit String Literals	Supported
Record Aggregates	Supported
Array Aggregates	Supported
Function Call	Supported

Table 4-10: **VHDL Constructs and Support Status *(Cont'd)***

VHDL Construct	Support Status
Qualified Expressions	Supported for accepted predefined attributes.
Types Conversions	Supported
Allocators	**Unsupported**
Static Expressions	Supported
Wait Statement	
Wait on `sensitivity_list` until `boolean_expression`. See VHDL Combinatorial Circuits.	Supported with one signal in the sensitivity list and in the boolean expression. • Multiple `wait` statements are not supported. • `wait` statements for Latch descriptions are not supported.
Wait for `time_expression`. See VHDL Combinatorial Circuits.	**Unsupported**
Assertion Statement	Supported for static conditions only.
Signal Assignment Statement	Supported. Delay is **ignored**.
Variable Assignment Statement	Supported
Procedure Call Statement	Supported
If Statement	Supported
Case Statement	Supported
Loop Statements	
Next Statement	Supported
Exit Statement	Supported
Return Statement	Supported
Null Statement	Supported
Concurrent Statement	
Process Statement	Supported
Concurrent Procedure Call	Supported
Concurrent Assertion Statement	Ignored
Concurrent Signal Assignment	Supported. No after clause. No transport or guarded options, No waveforms UNAFFECTED is supported.Statement
Component Instantiation Statement	Supported
for-generate	Statement supported for constant bounds only
if-generate	Statement supported for static condition only

VHDL RESERVED Words

Table 4-11: **VHDL RESERVED Words**

RESERVED Words			
abs	access	after	alias
all	and	architecture	array
assert	attribute	begin	block
body	buffer	bus	case
component	configuration	constant	disconnect
downto	else	elsif	end
entity	exit	file	for
function	generate	generic	group
guarded	if	impure	in
inertial	inout	is	label
library	linkage	literal	loop
map	mod	nand	new
next	nor	not	null
of	on	open	or
others	out	package	port
postponed	procedure	process	pure
range	record	register	reject
rem	report	return	rol
ror	select	severity	signal
shared	sla	sll	sra
srl	subtype	then	to
transport	type	unaffected	units
until	use	variable	wait
when	while	with	xnor
xor			

VHDL-2008 Language Support

Introduction

Vivado synthesis supports a synthesizable subset of the VHDL-2008 standard. The following section describes the supported subset and the procedures to use it.

Setting up Vivado to use VHDL-2008

There are several ways to run VHDL-2008 files with Vivado.

The first way is to set this within the GUI. To do that go to the Source File Properties window. The set **File Type: VHDL 2008.** The Source File properties then sets that the file type to VHDL-2008, as shown in the following figure.

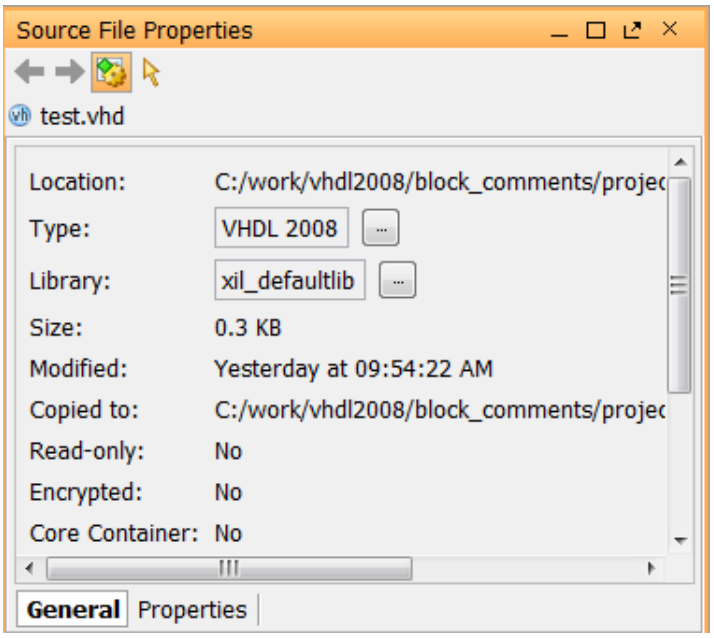

Figure 5-1: **Source File Properties**

You can also set files to VHDL-2008 from the Tcl Console with the following command:

- **Tcl Command**:
  `set_property FILE_TYPE {VHDL 2008} [get_files <file>.vhd]`

Finally, in the Non-Project or Tcl flow, the command for reading in VHDL has VHDL-2008 is as follows:

- **Tcl Command**: `read_vhdl -vhdl2008 <file>.vhd`

If you want to read in more than one file, you can either use multiple `read_vhdl` commands or multiple files with one command

- **Tcl Command**: `read_vhdl -vhdl2008 {a.vhd b.vhd c.vhd}`

Supported VHDL-2008 Features

Vivado supports the following VHDL-2008 features.

Operators

Matching Relational Operators

VHDL-2008 now provides relational operators that return bit or `std_logic` types. In the previous VHDL standard, the relational operators (`=`, `<`, `>=`...) returned boolean types. With the new types, code that needed to be written as:

```
if x = y then
    out1 <= '1';
else
    out1 <= '0';
end if;
```

Can now be written as:

```
out1 <= x ?= y;
```

The following table lists the relational operators supported in Vivado.

Table 5-1: **Supported Relational Operators**

Operator	Usage	Description
?=	x ?= y	x equal to y
?/=	x ?/= y	x not equal to y
?<	x ?< y	x less than y

Table 5-1: **Supported Relational Operators *(Cont'd)***

Operator	Usage	Description
?<=	x ?<= y	x less than or equal to y
?>	x ?> y	x greater than y
?>=	x ?>= y	x greater than or equal to y

Maximum and Minimum Operators

The new maximum and minimum operators in VHDL-2008 take in two different values and return the larger or smaller respectively. For example:

```
out1 <= maximum(const1, const2);
```

Shift Operators (rol, ror, sll, srl, sla and sra)

The `sla` and `sra` operators previously defined only bit and boolean elements. Now, the VHDL-2008 standard defines them in the signed and unsigned libraries.

Unary logical Reduction Operators

In the previous version of VHDL, operators such as `and`, `nand`, `or`, took two different values and then returned a bit or boolean value. For VHDL-2008, unary support has been added for these operators. They return the logical function of the input. For example, the code:

```
out1 <= and("0101");
```

would `AND` the 4 bits together and return `1`. The logical functions have unary support are: `and`, `nand`, `or`, `nor`, `xor`, and `xnor`.

Mixing Array and Scalar Logical Operators

Previously in VHDL, both of the operands of the logical operators needed to be the same size.

VHDL-2008 supports using logical operators when one of the operands is an array and one is a scalar. For example, to `AND` one bit with all the bits of a vector, the following code was needed:

```
out1(3) <= in1(3) and in2;
out1(2) <= in1(2) and in2;
out1(1) <= in1(1) and in2;
out1(0) <= in1(0) and in2;
```

This can now be replaced with the following:

```
out1<= in1 and in2;
```

Statements

If-else- If and Case Generate

Previously in VHDL, `if-generate` statements took the form of the following:

```
if condition generate
--- statements
end generate;
```

An issue appears if you want to have different conditions; you would need to write multiple generates and be very careful with the ordering of the generates. VHDL-2008 now offers `if-else-if` generate statements.

```
if condition generate
---statements
else if condition2 generate
---statements
else generate
---statements
end generate;
```

In addition, VHDL-2008 also offers `case-generate` statements:

```
case expressions generate
   when condition ->
   statements
    when condition2 ->
   statements
end generate;
```

Sequential Assignments

VHDL-2008 allows sequential signal and variable assignment with conditional signals. For example, a register with an enable would be written as the following:

```
process(clk) begin
  if clk'event and clk='1' then
    if enable then
     my_reg <= my_input;
    end if;
  end if;
end process;
```

With VHDL-2008, this can now be written as the following:

```
process(clk) begin
     if clk'event and clk='1' then
          my_reg <= my_input when enable else my_reg;
```

```
      end if;
end process;
```

case? Statements

With VHDL-2008, the case statement has a way to deal with explicit don't care assignments. When using `case?`, the tool now evaluates explicit `don't cares`, as in the following example:

```
process(clk) begin
if clk'event and clk='1' then
      case? my_reg is
  when "11--" => out1 <= in1;
  when "000-" => out1 <= in2;
  when "1111" => out1 <= in3;
  when others => out1 <= in4;
      end case?
end if;
end process;
```

Note: For this statement to work, the signal in question must be assigned an explicit `don't care`.

select? Statements

Like the case, the select statement now has a way to deal with explicit `don't care` assignments. When using the `select?` statement, the tool now evaluates explicit `don't cares`, for example:

```
process(clk) begin
if clk'event and clk='1' then
  with my_reg select?
  out1 <= in1 when "11--",
      in2 when "000-",
      in3 when "1111",
      in4 when others;
end if;
end process;
```

Note: For this statement to work, the signal in question must be assigned an explicit `don't care`.

Slices in Aggregates

VHDL-2008 allows you to form an array aggregate and then assign it to multiple places all in one statement.

For example if `in1` where defined as a `std_logic_vector(3 downto 0)`:

```
(my_reg1, my_reg2, enable, reset) <= in1;
```

This example assigns all four signals to the individual bits of `in1`:

- `my_reg1 gets in1(3)`
- `my_reg2 gets in1(2)`
- `enable is in1(1)`
- `reset is in1(0)`

In addition, these signals can be assigned out of order, as shown in the following example:

```
(1=> enable, 0 => reset, 3 => my_reg1, 2 => my_reg2) <= in1;
```

Types

Unconstrained Element Types

Previously in VHDL, types and subtypes had to be fully constrained in the declaration of the type.

In VHDL-2008, it allowed to be unconstrained and the constraining happens with the objects that are of that type; consequently, types and subtypes are more versatile. For example:

```
subtype my_type is std_logic_vector;

signal my_reg1 : my_type (3 downto 0);
signal my_reg2 : my_type (4 downto 0);
```

In previous versions of VHDL, the preceding example would have been done with 2 subtypes.

Now, in VHDL-2008 this can be accomplished with one type. This can even be done for arrays, as shown in the following example:

```
type my_type is array (natural range <>) of std_logic_vector;
signal : mytype(1 downto 0)(9 downto 0);
```

boolean_vector/integer_vector

VHDL-2008 supports new predefined array types. Vivado supports `boolean_vector` and `integer_vector`. These types are defined as follows:

```
type boolean_vector is array (natural range <>) of boolean
type integer_vector is array (natural range <>) of integer
```

Miscellaneous

Reading Output Ports

In previous versions of VHDL, it was illegal to use signals declared as `out` for anything other than an output.

So if you wanted to assign a value to an output, and also use that same signal for other logic, you would either have to declare a new signal and have that drive the output and the other logic, or switch from an `out` to a `buffer` type.

VHDL-2008 lets you use output values, as shown in the following example:

```
entity test is port(
in1 : in std_logic;
clk : in std_logic;
out1, out2 : out std_logic);
end test;
```

And then later in the architecture:

```
process(clk) begin
if clk'event an clk='1' then
  out1 <= in1;
  my_reg <= out1;  -- THIS WOULD HAVE BEEN ILLEGAL in VHDL.
  out2 <= my_reg;
end if;
end process;
```

Expressions in Port Maps

VHDL-2008 allows the use of functions and assignments within the port map of an instantiation. One useful way this is used is in converting signals from one type to another, as shown in the following example:

```
U0 : my_entity port map (clk => clk, in1 => to_integer(my_signal)…
```

In the case above, the entity, `my_entity` had a port called `in1` that was of type `integer`, but in the upper-level, the signal, `my_signal` was of type `std_logic_vector`.

Previously in VHDL, you would have to create a new signal of type `integer` and do the conversion outside of the instantiation, and then assign that new signal to the port map.

In addition to type conversion, you can put logic into the port map, as shown in the following example:

```
U0 : my_entity port map (clk => clk, enable => en1 and en2 …
```

In this case the lower-level has an `enable` signal. On the top-level that `enable` is tied to the `AND` of two other signals.

Previously in VHDL, this, like the previous example would have needed a new signal and assignment, but in VHDL-2008 can be accomplished in the port map of the instantiation.

process (all)

In VHDL, when listing items in the sensitivity list of a process statement for combinational logic, it was up to the designer to make sure all the items read by the process statement were listed. If any were missed, there would be Warning messages and possible latches inferred in the design.

With VHDL-2008, you can use the `process(all)` statement that looks for all the inputs to the process and then creates the logic.

```
process(all) begin
enable <= en1 and en2;
end process;
```

Referencing Generics in Generic Lists

VHDL-2008 allows generics to reference other generics, as shown in the following example:

```
entity my_entity is generic (
gen1 : integer;
gen2 : std_logic_vector(gen1 - 1 downto 0));
```

In previous versions of VHDL, having the length of `gen2` be controlled by `gen1` was illegal.

Relaxed Return Rules for Function Return Values

In previous versions of VHDL, the return expression of a function needed be same type as was declared in the functions return type of the function. In VHDL-2008, the rules are relaxed to allow the return expression to be implicitly converted to the return type. For example:

```
subtype my_type1 is std_logic_vector(9 downto 0);
subtype my_type2 is std_logic_vector(4 downto 0);

function my_function (a,b : my_type2) return my_type1 is
begin
return (a&b);
end function;
```

Because concatenation is not static, this would return an error or warning in VHDL; however, it is allowed with VHDL-2008.

Extensions to Globally Static and Locally Static Expressions

In VHDL, expressions in many types of places needed to be static. For example, using concatenation would not have returned a static value and when used with an operator or function that needed a static value resulting in an error. VHDL-2008 allows for more expressions, like concatenation to return static values, thereby allowing for more flexibility.

Static Ranges and Integer Expressions in Range Bounds

In VHDL, it was possible to declare an object by using the range of another object. For example:

```
for I in my_signal'range…
```

This would require that the `range` of `my_signal'` be fixed, but if `my_signal` was declared as an unconstrained type, this would result in an error. VHDL-2008 now allows this by getting the range at the time of elaboration.

Block Comments

In VHDL, comments "--" were required for each line that had a comment. In VHDL-2008, there is support for blocks of comments using the /* and */ lines.

```
process(clk) begin
  if clk'event and clk='1' then
  /* this
  is
  a block
  comment */
  out1 <= in1;
  end if;
end process;
```

Verilog Language Support

Introduction

This chapter describes the Vivado® synthesis support for the Verilog Hardware Description Language.

Verilog Design

Complex circuits are often designed using a top-down methodology.

- Varying specification levels are required at each stage of the design process. For example, at the architectural level, a specification can correspond to a block diagram or an Algorithmic State Machine (ASM) chart.
- A block or ASM stage corresponds to a register transfer block in which the connections are N-bit wires, such as:
  - Register
  - Adder
  - Counter
  - Multiplexer
  - Interconnect logic
  - Finite State Machine (FSM)
- Verilog allows the expression of notations such as ASM charts and circuit diagrams in a computer language.

Verilog Functionality

Verilog provides both behavioral and structural language structures. These structures allow the expression of design objects at high and low levels of abstraction.

- Designing hardware with Verilog allows the use of software concepts such as:
  - Parallel processing
  - Object-oriented programming
- Verilog has a syntax similar to C and Pascal.
- Vivado synthesis supports Verilog as IEEE 1364.
- Verilog support in Vivado synthesis allows you to describe the global circuit and each block in the most efficient style.
  - Synthesis is performed with the best synthesis flow for each block.
  - Synthesis in this context is the compilation of high-level behavioral and structural Verilog HDL statements into a flattened gate-level netlist. The netlist can then be used to custom program a programmable logic device such as a Virtex® device.
  - Different synthesis methods are used for:
    - Arithmetic blocks
    - Interconnect logic
    - Finite State Machine (FSM) components

For information about basic Verilog concepts, see the *IEEE Verilog HDL Reference Manual*.

Verilog-2001 Support

Vivado synthesis supports the following Verilog-2001 features.

- Generate statements
- Combined port/data type declarations
- ANSI-style port list
- Module parameter port lists
- ANSI C style task/function declarations
- Comma-separated sensitivity list
- Combinatorial logic sensitivity
- Default nets with continuous assigns

- Disable default net declarations
- Indexed vector part selects
- Multi-dimensional arrays
- Arrays of net and real data types
- Array bit and part selects
- Signed reg, net, and port declarations
- Signed-based integer numbers
- Signed arithmetic expressions
- Arithmetic shift operators
- Automatic width extension past 32 bits
- Power operator
- N sized parameters
- Explicit in-line parameter passing
- Fixed local parameters
- Enhanced conditional compilation
- File and line compiler directives
- Variable part selects
- Recursive Tasks and Functions
- Constant Functions

For more information, see:

- Sutherland, Stuart. Verilog 2001: *A Guide to the New Features of the Verilog Hardware Description Language (2002)*
- I*EEE Standard Verilog Hardware Description Language Manual* (IEEE Standard1364-2001)

Verilog-2001 Variable Part Selects

Verilog-2001 lets you use variables to select a group of bits from a vector.

Instead of being bounded by two explicit values, the variable part select is defined by the starting point of its range and the width of the vector. The starting point of the part select can vary. The width of the part select remains constant.

Table 6-1: **Variable Part Selects Symbols**

Symbol	Meaning
+ (plus)	The part select increases from the starting point.
- (minus)	The part select decreases from the starting point

Variable Part Selects Verilog Coding Example

```
reg [3:0] data;
reg [3:0] select; // a value from 0 to 7
wire [7:0] byte = data[select +: 8];
```

Structural Verilog

Structural Verilog descriptions assemble several blocks of code and allow the introduction of hierarchy in a design.

Table 6-2: **Basic Concepts of Hardware Structure**

Concept	Description
Component	Building or basic block
Port	Component I/O connector
Signal	Corresponds to a wire between components

Table 6-3: **Verilog Components**

Item	View	Describes
Declaration	External	What is seen from the outside, including the component ports
Body	Internal	The behavior or the structure of the component

- A component is represented by a design module.
- The connections between components are specified within component instantiation statements.
- A component instantiation statement:
  - Specifies an instance of a component occurring within another component or the circuit
  - Is labeled with an identifier.
  - Names a component declared in a local component declaration.
  - Contains an association list (the parenthesized list). The list specifies the signals and ports associated with a given local port.

Built-In Logic Gates

Verilog provides a large set of built-in logic gates.

- The logic gates are instantiated to build larger logic circuits.
- The set of logical functions described by the built-in logic gates includes:
  - AND
  - OR
  - XOR
  - NAND
  - NOR
  - NOT

2-Input XOR Function Example

In this coding example, each instance of the built-in modules has a unique instantiation name such as:

- a_inv
- b_inv
- out

```
module build_xor (a, b, c);
input a, b;
output c;
wire c, a_not, b_not;

not a_inv (a_not, a);
not b_inv (b_not, b);
and a1 (x, a_not, b);
and a2 (y, b_not, a);
or out (c, x, y);
endmodule
```

Half-Adder Example

This coding example shows the structural description of a half-Adder composed of four, 2-input nand modules.

```
module halfadd (X, Y, C, S);
input X, Y;
output C, S;
wire S1, S2, S3;
```

```
nand NANDA (S3, X, Y);
nand NANDB (S1, X, S3);
nand NANDC (S2, S3, Y);
nand NANDD (S, S1, S2);
assign C = S3;
endmodule
```

Instantiating Pre-Defined Primitives

The structural features of Verilog allow you to design circuits by instantiating pre-defined primitives such as: gates, registers, and Xilinx® specific primitives such as `CLKDLL` and `BUFG`.

These primitives are additional to those included in Verilog, and are supplied with the Xilinx Verilog libraries (`unisim_comp.v`).

Instantiating an FDC and a BUFG Primitive Example

The `unisim_comp.v` library file includes the definitions for `FDC` and `BUFG`.

```
module example (sysclk, in, reset, out);
input sysclk, in, reset;
output out;
reg out;
wire sysclk_out;

FDC register (out, sysclk_out, reset, in); //position based
referencing
BUFG clk (.O(sysclk_out),.I(sysclk)); //name based referencing
```

Verilog Parameters

Verilog parameters:

- Allow you to create parameterized code that can be easily reused and scaled.
- Make code more readable, more compact, and easier to maintain.
- Describe such functionality as:
  - Bus sizes
  - The amount of certain repetitive elements in the modeled design unit
- Are constants. For each instantiation of a parameterized module, default parameter values can be overridden.
- Are the equivalent of VHDL generics. Null string parameters are not supported.

Use the Generics command line option to redefine Verilog parameters defined in the top-level design block. This allows you to modify the design without modifying the source code. This feature is useful for IP core generation and flow testing.

Verilog Parameters Example

Coding examples are included in this chapter. Download the coding example files from: Coding Examples.

```
// A Verilog parameter allows to control the width of an instantitated
// block describing register logic
//
//
// File:parameter_1.v
//
module myreg (clk, clken, d, q);

    parameter SIZE = 1;

    input                  clk, clken;
    input      [SIZE-1:0]  d;
    output reg [SIZE-1:0]  q;

  always @(posedge clk)
    begin
        if (clken)
            q <= d;
    end

endmodule

module parameter_1 (clk, clken, di, do);

    parameter SIZE = 8;

    input                  clk, clken;
    input      [SIZE-1:0]  di;
    output     [SIZE-1:0]  do;

    myreg #8 inst_reg (clk, clken, di, do);

endmodule
```

Parameter and `Generate-For` *Example*

The following coding example illustrates how to control the creation of repetitive elements using parameters and generate-for constructs. For more information, see Generate Loop Statements.

```
//
// A shift register description that illustrates the use of
parameters and
// generate-for constructs in Verilog
//
// File: parameter_generate_for_1.v
//
module parameter_generate_for_1 (clk, si, so);

parameter SIZE = 8;

input   clk;
input   si;
output  so;

reg [0:SIZE-1]  s;

assign so = s[SIZE-1];

always @ (posedge clk)
 s[0] <= si;

genvar i;
generate
  for (i = 1; i < SIZE; i = i+1)
  begin : shreg
      always @ (posedge clk)
      begin
          s[i] <= s[i-1];
      end
  end
endgenerate

endmodule
```

Verilog Parameter and Attribute Conflicts

Verilog parameter and attribute conflicts can arise because of the following:

- Parameters and attributes can be applied to both instances and modules in the Verilog code.
- Attributes can also be specified in a constraints file.

Verilog Usage Restrictions

Verilog usage restrictions in Vivado synthesis include the following:

- Case Sensitivity

- Blocking and Non-Blocking Assignments
- Integer Handling

Case Sensitivity

Vivado synthesis supports Verilog case sensitivity despite the potential of name collision.

- Because Verilog is case-sensitive, the names of modules, instances, and signals can theoretically be made unique by changing capitalization.
  - Vivado synthesis can synthesize a design in which instance and signal names differ only by capitalization.
  - Vivado synthesis errors out when module names differ only by capitalization.
- Do not rely on capitalization alone to make object names unique. Capitalization alone can cause problems in mixed language projects.

Blocking and Non-Blocking Assignments

Vivado synthesis supports blocking and non-blocking assignments.

- Do not mix blocking and non-blocking assignments.
- Although Vivado synthesis synthesizes the design without error, mixing blocking and non-blocking assignments can cause errors during simulation.

Unacceptable Example One

Do not mix blocking and non-blocking assignments to the same signal.

```
always @(in1)
begin
  if (in2)
    out1 = in1;

end else
  out1 <= in2;
```

Unacceptable Example Two

Do not mix blocking and non-blocking assignments for different bits of the same signal.

```
if (in2)
begin
  out1[0] = 1'b0;
  out1[1] <= in1;
end else begin
  out1[0] = in2;
  out1[1] <= 1'b1;
end
```

Integer Handling

Vivado synthesis handles integers differently from other synthesis tools in some situations. In those instances, the integers must be coded in a particular way.

Integer Handling in Verilog Case Statements

Unsized integers in case item expressions can cause unpredictable results.

Integer Handling in Verilog Case Statements Example

In the following coding example, the case item expression 4 is an unsized integer that causes unpredictable results. To resolve this issue, size the case item expression 4 to 3 bits, as shown in the following example:

```
reg [2:0] condition1; always @(condition1) begin
  case(condition1)
  4 : data_out = 2; // Generates faulty logic
  3'd4 : data_out = 2; // Does work
  endcase
end
```

Integer Handling in Concatenations

Unsigned integers in Verilog concatenations can cause unpredictable results. If you use an expression that results in an unsized integer, it does the following:

- Assign the expression to a temporary signal.
- Use the temporary signal in the concatenation.

```
reg [31:0] temp;
assign temp = 4'b1111 % 2;
assign dout = {12/3,temp,din};
```

Verilog-2001 Attributes and Meta Comments

Verilog-2001 Attributes

- Verilog-2001 attributes pass specific information to programs such as synthesis tools.
- Verilog-2001 attributes are generally accepted.
- Specify Verilog-2001 attributes anywhere for operators or signals, within module declarations and instantiations.
- Although the compiler might support other attribute declarations, Vivado synthesis ignores them.
- Use Verilog-2001 attributes to:
  - Set constraints on individual objects, such as:
    - Module
    - Instance
    - Net
  - Set the following synthesis constraints:
    - Full Case
    - Parallel Case

Verilog Meta Comments

- Verilog meta comments are understood by the Verilog parser.
- Verilog meta comments set constraints on individual objects, such as:
  - Module
  - Instance
  - Net
- Verilog meta comments set directives on synthesis:
  - `parallel_case` and `full_case`
  - `translate_on` and `translate_off`
  - All tool specific directives (for example, `syn_sharing`)

Verilog Meta Comment Support

Vivado synthesis supports:

- C-style and Verilog style meta comments:
  - C-style

```
/* ...*/
```

- C-style comments can be multiple line:
  - Verilog style

```
// ...
```

Verilog style comments end at the end of the line.

- Translate Off and Translate On

```
// synthesis translate_on
// synthesis translate_off
```

- Parallel Case

```
// synthesis parallel_case full_case
// synthesis parallel_case
// synthesis full_case
```

- Constraints on individual objects

Verilog Meta Comment Syntax

```
// synthesis attribute [of] ObjectName [is] AttributeValue
```

Verilog Meta Comment Syntax Examples

```
// synthesis attribute RLOC of u123 is R11C1.S0
// synthesis attribute HUSET u1 MY_SET
// synthesis attribute fsm_extract of State2 is "yes"
// synthesis attribute fsm_encoding of State2 is "gray"
```

Verilog Constructs

The following table lists the support status of Verilog constructs in Vivado synthesis.

Table 6-4: **Verilog Constructs**

Verilog Constants	
Constant	**Support Status**
Integer	Supported
Real	Supported
String	**Unsupported**
Verilog Data Types	
Net types: • tri0 • tri1 • trireg	**Unsupported**
All Drive strengths	Ignored
Real and realtime registers	**Unsupported**
All Named events	**Unsupported**
Delay	Ignored
Verilog Procedural Assignments	
assign	Supported with limitations. See Assign and De-assign Statements.
deassign	Supported with limitations. See Assign and De-assign Statements
force	**Unsupported**
release	**Unsupported**
forever statements	**Unsupported**
repeat statements	Supported, but repeat value must be constant
for statements	Supported, but bounds must be static
delay (#)	Ignored
event (@)	**Unsupported**
wait	**Unsupported**
named events	**Unsupported**
parallel blocks	**Unsupported**
specify blocks	Ignored
disable	Supported except in For and Repeat Loop statements

Table 6-4: **Verilog Constructs *(Cont'd)***

Verilog Constants	
Constant	**Support Status**
Verilog Design Hierarchies	
module definition	Supported
macromodule definition	**Unsupported**
hierarchical names	**Unsupported**
defparam	Supported
array of instances	Supported
Verilog Compiler Directives	
`celldefine `endcelldefine	Ignored
`default_nettype	Supported
`define	Supported
`ifdef `else `endif	Supported
`undef, `ifndef, `elsif	Supported
`include	Supported
`resetall	Ignored
`timescale	Ignored
`unconnected_drive `nounconnected_drive	Ignored
`uselib	**Unsupported**
`file, `line	Supported

Verilog System Tasks and Functions

Vivado synthesis supports system tasks or function as shown in the following table. Vivado synthesis ignores unsupported system tasks.

Table 6-5: **System Tasks and Status**

System Task or Function	Status	Comment
$display	**Not Supported**	
$fclose	Supported	
$fdisplay	Ignored	
$fgets	Supported	
$finish	Ignored	
$fopen	Supported	
$fscanf	Supported	Escape sequences are limited to %b and %d

Table 6-5: **System Tasks and Status *(Cont'd)***

System Task or Function	Status	Comment
$fwrite	Ignored	
$monitor	Ignored	
$random	Ignored	
$readmemb	Supported	
$readmemh	Supported	
$signed	Supported	
$stop	Ignored	
$strobe	Ignored	
$time	Ignored	
$unsigned	Supported	
$write	Supported	Escape sequences are limited to %d, %b, %h, %o, %c and %s.
$clog2	Supported	This is supported with SystemVerilog only.
all others	Ignored	

Using Conversion Functions

Use the following syntax to call `$signed` and `$unsigned` system tasks on any expression.

```
$signed(expr) or $unsigned(expr)
```

- The return value from these calls is the same size as the input value.
- The sign of the return value is forced regardless of any previous sign.

Loading Memory Contents With File I/O Tasks

Use the `$readmemb` and `$readmemh` system tasks to initialize block memories.

- Use `$readmemb` for binary representation.
- Use `$readmemh` for hexadecimal representation.
- Use index parameters to avoid behavioral conflicts between Vivado synthesis and the simulator.

```
$readmemb("rams_20c.data",ram, 0, 7);
```

Supported Escape Sequences

- %h
- %d
- %o
- %b
- %c
- %s

Verilog Primitives

Vivado synthesis supports Verilog gate-level primitives except as shown in Table 6-6.

Vivado synthesis does not support Verilog switch-level primitives, such as the following:

```
cmos, nmos, pmos, rcmos, rnmos, rpmos rtran, rtranif0, rtranif1,
tran, tranif0, tranif1
```

Gate-Level Primitive Syntax

```
gate_type instance_name (output, inputs,...);
```

Gate-Level Primitive Example

```
and U1 (out, in1, in2); bufif1 U2 (triout, data, trienable);
```

Unsupported Verilog Gate Level Primitives

The following gate-level primitives are not supported in Vivado synthesis.

Table 6-6: **Unsupported Primitives**

Primitive	Status
pulldown and pullup	Unsupported
drive strength and delay	Ignored
Arrays of primitives	Unsupported

Verilog Reserved Keywords

The following table lists the reserved keywords. Keywords marked with an asterisk (*) are reserved by Verilog, but Vivado synthesis does not support them.

Table 6-7: **Verilog Reserved Keywords**

always	and	assign	automatic
begin	buf	bufif0	bufif1
case	casex	casez	cell*
cmos	config*	deassign	default
defparam	design*	disable	edge
else	end	endcase	endconfig*
endfunction	endgenerate	endmodule	endprimitive
endspecify	endtable	endtask	event
for	force	forever	fork
function	generate	genvar	highz0
highz1	if	ifnone	incdir*
include*	initial	inout	input
instance*	integer	join	larger
liblist*	library*	localparam	macromodule
medium	module	nand	negedge
nmos	nor	noshow-cancelled*	not
notif0	notif1	or	output
parameter	pmos	posedge	primitive
pull0	pull1	pullup	pulldown
pulsestyle-_ondetect*	pulsestyle- _onevent*	rcmos	real
realtime	reg	release	repeat
rnmos	rpmos	rtran	rtranif0
rtranif1	scalared	show-cancelled*	signed
small	specify	specparam	strong0
strong1	supply0	supply1	table
task	time	tran	tranif0
tranif1	tri	tri0	tri1
triand	trior	trireg	use*
vectored	wait	wand	weak0
weak1	while	wire	wor
xnor	xor		

Behavioral Verilog

Vivado synthesis supports the behavioral Verilog Hardware Description Language (HDL), except as otherwise noted.

Variables in Behavioral Verilog

- Variables in behavioral Verilog are declared as an integer.
- These declarations are used in test code only. Verilog provides data types such as `reg` and `wire` for actual hardware description.
- The difference between reg and wire depends on whether the variable is given its value in a procedural block (`reg`) or in a continuous assignment (`wire`).
  - Both `reg` and `wire` have a default width of one bit (scalar).
  - To specify an N-bit width (vectors) for a declared reg or wire, the left and right bit positions are defined in square brackets separated by a colon.
  - In Verilog-2001, `reg` and `wire` data types can be signed or unsigned.

Variable Declarations Example

```
reg [3:0] arb_priority;
wire [31:0] arb_request;
wire signed [8:0] arb_signed;
```

Initial Values

Initialize registers in Verilog-2001 when they are declared.

- The initial value:
  - Is a constant.
  - Cannot depend on earlier initial values.
  - Cannot be a function or task call.
  - Can be a parameter value propagated to the register.
  - Specifies all bits of a vector.
- When you assign a register as an initial value in a declaration, Vivado synthesis sets this value on the output of the register at global reset or power up.

- When a value is assigned in this manner:
  - The value is carried in the Verilog file as an `INIT` attribute on the register.
  - The value is independent of any local reset.

Assigning an Initial Value to a Register

Assign a set/reset (initial) value to a register.

- Assign the value to the register when the register reset line goes to the appropriate value. See the following coding example.
- When you assign the initial value to a variable:
  - The value is implemented as a Flip-Flop, the output of which is controlled by a local reset.
  - The value is carried in the Verilog file as an FDP or FDC Flip-Flop.

Initial Values Example One

```
reg arb_onebit = 1'b0;
reg [3:0] arb_priority = 4'b1011;
```

Initial Values Example Two

```
always @(posedge clk)
begin
  if (rst)
    arb_onebit <= 1'b0;
end
```

Arrays of Reg and Wire

Verilog allows arrays of `reg` and `wire`.

Arrays Example One

This coding example describes an array of 32 elements. Each element is 4-bits wide.

```
reg [3:0] mem_array [31:0];
```

Arrays Example Two

This coding example describes an array of 64 8-bit wide elements. These elements can be assigned only in structural Verilog code.

```
wire [7:0] mem_array [63:0];
```

Multi-Dimensional Arrays

Vivado synthesis supports multi-dimensional array types of up to two dimensions.

- Multi-dimensional arrays can be:
  - Any net
  - Any variable data type
- Code assignments and arithmetic operations with arrays.
- You cannot select more than one element of an array at one time.
- You cannot pass multi-dimensional arrays to:
  - System tasks or functions
  - Regular tasks or functions

Multi-Dimensional Array Example One

This coding example describes an array of 256 x 16 wire elements of 8-bits each. These elements can be assigned only in structural Verilog code.

```
wire [7:0] array2 [0:255][0:15];
```

Multi-Dimensional Array Example Two

This coding example describes an array of 256 x 8 register elements, each 64 bits wide. These elements can be assigned in behavioral Verilog code.

```
reg [63:0] regarray2 [255:0][7:0];
```

Data Types

The Verilog representation of the `bit` data type contains the following values:

- 0 = logic zero
- 1 = logic one
- x =unknown logic value
- z = high impedance

Supported Data Types

- net
  - wire
- registers
  - reg
  - integer
- constants
  - parameter
  - Multi-dimensional arrays (memories)

Net and Registers

Net and Registers can be either:

- Single bit (scalar)
- Multiple bit (vectors)

Behavioral Data Types Example

This coding example shows sample Verilog data types found in the declaration section of a Verilog module.

```
wire net1; // single bit net
reg r1; // single bit register
tri [7:0] bus1; // 8 bit tristate bus
reg [15:0] bus1; // 15 bit register
reg [7:0] mem[0:127]; // 8x128 memory register
parameter state1 = 3'b001; // 3 bit constant
parameter component = "TMS380C16"; // string
```

Legal Statements

Vivado synthesis supports behavioral Verilog legal statements.

- The following statements (variable and signal assignments) are legal:
  - variable = expression
  - if (condition) statement
  - else statement
  - case (expression), for example:

```
expression: statement
...
default: statement
endcase
```

    - for (variable = expression; condition; variable = variable + expression) statement
    - while (condition) statement
    - forever statement
    - functions and tasks
- All variables are declared as integer or reg.
- A variable cannot be declared as a wire.

Expressions

Behavioral Verilog expressions include:

- Constants
- Variables with the following operators:
    - arithmetic
    - logical
        - bitwise
        - logical
    - relational
    - conditional

Logical Operators

The category (bitwise or logical) into which a logical operator falls depends on whether it is applied to an expression involving several bits, or a single bit.

Supported Operators

Table 6-8: **Supported Operators**

Arithmetic	Logical	Relational	Conditional
+	&	<	?
-	&&	==	
*	|	===	
**	||	<=	

Table 6-8: **Supported Operators *(Cont'd)***

Arithmetic	Logical	Relational	Conditional
/	^	>=	
%	~	>=	
	~^	!=	
	^~	!==	
	<<	>	
	>>		
	<<<		
	>>>		

Supported Expressions

Table 6-9: **Supported Expressions**

Expression	Symbol	Status
Concatenation	{}	Supported
Replication	{{}}	Supported
Arithmetic	+, -, *,**	Supported
Division	/	Supported only if the second operand is a power of 2, or both operands are constant.
Modulus	%	Supported only if second operand is a power of 2.
Addition	+	Supported
Subtraction	-	Supported
Multiplication	*	Supported
Power	**	Supported: • Both operands are constants, with the second operand being non-negative. • If the first operand is a 2, then the second operand can be a variable. • Vivado synthesis does not support the real data type. Any combination of operands that results in a real type causes an error. • The values X (unknown) and Z (high impedance) are not allowed.
Relational	>, <, >=, <=	Supported

Table 6-9: **Supported Expressions *(Cont'd)***

Expression	Symbol	Status
Logical Negation	!	Supported
Logical AND	&&	Supported
Logical OR	||	Supported
Logical Equality	==	Supported
Logical Inequality	!=	Supported
Case Equality	===	Supported
Case Inequality	!==	Supported
Bitwise Negation	~	Supported
Bitwise AND	&	Supported
Bitwise Inclusive OR	|	Supported
Bitwise Exclusive OR	^	Supported
Bitwise Equivalence	~^, ^~	Supported
Reduction AND	&	Supported
Reduction NAND	~&	Supported
Reduction OR	|	Supported
Reduction NOR	~|	Supported
Reduction XOR	^	Supported
Reduction XNOR	~^, ^~	Supported
Left Shift	<<	Supported
Right Shift Signed	>>>	Supported
Left Shift Signed	<<<	Supported
Right Shift	>>	Supported
Conditional	?:	Supported
Event OR	or, ','	Supported

Evaluating Expressions

The (===) and (!==) operators in the following table:

- Are special comparison operators.
- Are used in simulation to see if a variable is assigned a value of (x) or (z).
- Are treated as (==) or (!=) by synthesis.

Evaluated Expressions Based On Most Frequently Used Operators

Table 6-10: **Evaluated Expressions Based On Most Frequently Used Operators**

a b	a==b	a===b	a!=b	a!==b	a&b	a&&b	a|b	a||b	a^b
0 0	1	1	0	0	0	0	0	0	0
0 1	0	0	1	1	0	0	1	1	1
0 x	x	0	x	1	0	0	x	x	x
0 z	x	0	x	1	0	0	x	x	x
1 0	0	0	1	1	0	0	1	1	1
1 1	1	1	0	0	1	1	1	1	0
1 x	x	0	x	1	x	x	1	1	x
1 z	x	0	x	1	x	x	1	1	x
x 0	x	0	x	1	0	0	x	x	x
x 1	x	0	x	1	x	x	1	1	x
x x	x	1	x	0	x	x	x	x	x
x z	x	0	x	1	x	x	x	x	x
z 0	x	0	x	1	0	0	x	x	x
z 1	x	0	x	1	x	x	1	1	x
z x	x	0	x	1	x	x	x	x	x
z z	x	1	x	0	x	x	x	x	x

Blocks

Vivado synthesis supports some block statements.

- Block statements:
  - Group statements together.
  - Are designated by begin and end keywords.
  - Execute the statements in the order listed within the block.
- Vivado synthesis supports sequential blocks only.
- Vivado synthesis does not support parallel blocks.
- All procedural statements occur in blocks that are defined inside modules.
- The two kinds of procedural blocks are:
  - initial block
  - always block
- Verilog uses `begin` and `end` keywords within each block to enclose the statements. Because initial blocks are ignored during synthesis, only `always` blocks are described.

- `always` blocks usually take the following format. Each statement is a procedural assignment line terminated by a semicolon.

```
always
begin
statement
.... end
```

Modules

A Verilog design component is represented by a module. Modules must be declared and instantiated.

Module Declaration

- A Behavioral Verilog module declaration consists of:
  - The module name
  - A list of circuit I/O ports
  - The module body in which you define the intended functionality
- The end of the module is signaled by a mandatory `endmodule` statement.

Circuit I/O Ports

- The circuit I/O ports are listed in the module declaration.
- Each circuit I/O port is characterized by:
  - A name
  - A mode: `Input`, `Output`, `Inout`
  - Range information if the port is of `array` type.

Behavioral Verilog Module Declaration Example One

```
module example (A, B, O);
  input A, B;
    output O;
    assign O = A & B;
endmodule
```

Behavioral Verilog Module Declaration Example Two

```
module example ( input A, inputB, output O
);

  assign O = A & B;
endmodule
```

Module Instantiation

A behavioral Verilog module instantiation statement does the following:

- Defines an instance name.
- Contains a port association list. The port association list specifies how the instance is connected in the parent module. Each element of the port association list ties a formal port of the module declaration to an actual net of the parent module.
- Is instantiated in another module. See the following coding example.

Behavioral Verilog Module Instantiation Example

```
module top (A, B, C, O); input  A, B, C; output O;
  wire  tmp;

  example inst_example (.A(A), .B(B), .O(tmp));

  assign O = tmp | C;

endmodule
```

Continuous Assignments

Vivado synthesis supports both explicit and implicit continuous assignments.

- Continuous assignments model combinatorial logic in a concise way.
- Vivado synthesis ignores delays and strengths given to a continuous assignment.
- Continuous assignments are allowed on wire and tri data types only.

Explicit Continuous Assignments

Explicit continuous assignments start with an assign keyword after the net has been separately declared.

```
wire mysignal;
...
assign mysignal = select ? b : a;
```

Implicit Continuous Assignments

Implicit continuous assignments combine declaration and assignment.

```
wire misignal = a | b;
```

Procedural Assignments

- Behavioral Verilog procedural assignments:
  - Assign values to variables declared as reg.
  - Are introduced by always blocks, tasks, and functions.
  - Model registers and Finite State Machine (FSM) components.
- Vivado synthesis supports:
  - Combinatorial functions
  - Combinatorial and sequential tasks
  - Combinatorial and sequential always blocks

Combinatorial Always Blocks

Combinatorial logic is modeled efficiently by Verilog time control statements:

- Delay time control statement [#]
- Event control time control statement [@]

Delay Time Control Statement

The delay time control statement [# (pound)] is:

- Relevant for simulation only.
- Ignored for synthesis.

Event Control Time Control Statement

The following statements describe modeling combinatorial logic with the event control time control statement [@ (at)].

- A combinatorial always block has a sensitivity list appearing within parentheses after `always@`.
- An always block is activated if an event (value change or edge) appears on one of the sensitivity list signals.
- The sensitivity list can contain:
  - Any signal that appears in conditions, such as if or case.
  - Any signal appearing on the right-hand side of an assignment.
- By substituting an @ (at) without parentheses for a list of signals, the `always` block is activated for an event in any of the `always` block's signals as described.
- In combinatorial processes, if a signal is not explicitly assigned in all branches of if or case statements, Vivado synthesis generates a Latch to hold the last value.
- The following statements are used in a process:
  - variable and signal assignments
  - if-else statements
  - case statements
  - for-while loop statements
  - function and task calls

if-else Statements

Vivado synthesis supports `if-else` statements.

- The `if-else` statements use true and false conditions to execute statements.
  - If the expression evaluates to true, the first statement is executed.
  - If the expression evaluates to false, x, or z, the else statement is executed.
- A block of multiple statements is executed using `begin` and `end` keywords.
- if-else statements can be nested.

if-else Statement Example

This coding example uses an `if-else` statement to describe a Multiplexer.

```
module mux4 (sel, a, b, c, d, outmux);
  input [1:0] sel;
  input [1:0] a, b, c, d;
  output [1:0] outmux;
  reg [1:0] outmux;

  always @(sel or a or b or c or d)
begin
  if (sel[1])
  if (sel[0])
  outmux = d;
  else
  else
  outmux = c;
    if (sel[0])
    outmux = b;
  end endmodule
  else
    outmux = a;
```

Case Statements

Vivado synthesis supports case statements.

- A case statement performs a comparison to an expression to evaluate one of several parallel branches.
  - The case statement evaluates the branches in the order they are written.
  - The first branch that evaluates to true is executed.
  - If none of the branches matches, the default branch is executed.
- Do not use unsized integers in case statements. Always size integers to a specific number of bits. Otherwise, results can be unpredictable.
- `casez` treats all z values in any bit position of the branch alternative as a don't care.
- `casex` treats all x and z values in any bit position of the branch alternative as a don't care.
- The question mark (?) can be used as a don't care in either the `casez` or `casex` case statements.

Multiplexer Case Statement Example (Verilog)

```
// Multiplexer using case statement
module mux4 (sel, a, b, c, d, outmux);
input [1:0] sel;
input [1:0] a, b, c, d;
output [1:0] outmux;
reg [1:0] outmux;

always @ *
   begin
     case(sel)
      2'b00 : outmux = a;
       2'b01 : outmux = b;
       2'b10 : outmux = c;
       2'b11 : outmux = d;
      endcase
   end
endmodule
```

Avoiding Priority Processing

- The case statement in the previous coding example evaluates the values of `input sel` in priority order.
- To avoid priority processing:
  - Use a parallel-case Verilog attribute to ensure parallel evaluation of the `input sel`.
  - Replace the case statement with:

```
(* parallel_case *) case(sel)
```

For and Repeat Statements

Vivado synthesis supports for and repeat statements. When using `always` blocks, repetitive or bit slice structures can also be described using a `for` statement, or a `repeat` statement.

For Statements

The `for` statement is supported for constant bound, and stop test condition using the following operators: <, <=, >, >=.

The `for` statement is supported also for next step computation falling in one of the following specifications:

- `var` = var + step
- `var` = var - step

  Where:

  - `var` is the loop variable
  - `step` is a constant value

Repeat Statements

The repeat statement is supported for constant values only.

While Loops

When using `always` blocks, use `while` loops to execute repetitive procedures.

- A `while` loop:
  - Is not executed if the test expression is initially false.
  - Executes other statements until its test expression becomes false.
- The test expression is any valid Verilog expression.
- To prevent endless loops, use the `-loop_iteration_limit` option.
- A `while` loop can have disable statements. The `disable` statement is used inside a labeled block, as shown in the following code snippet:

  ```
  disable <blockname>
  ```

While Loop Example

```
parameter P = 4; always @(ID_complete) begin : UNIDENTIFIED
integer i; reg found; unidentified = 0; i = 0;
found = 0;
while (!found && (i < P))
  begin
  found = !ID_complete[i];
  unidentified[i] = !ID_complete[i];
  i = i + 1;
end
```

Sequential always Blocks

Vivado synthesis supports sequential `always` blocks.

- Describe a sequential circuit with an `always` block and a sensitivity list that contains the following edge-triggered (with `posedge` or `negedge`) events:
  - A mandatory clock event
  - Optional set/reset events (modeling asynchronous set/reset control logic)
- If no optional asynchronous signal is described, the `always` block is structured as follows:

```
always @(posedge CLK)
begin
<synchronous_part>
end
```

- If optional asynchronous control signals are modeled, the `always` block is structured as follows:

```
always @(posedge CLK or posedge ACTRL1 or à )
begin
if (ACTRL1)
  <$asynchronous part>
else
  <$synchronous_part>
end
```

Sequential always Block Example One

This coding example describes an 8-bit register with a rising-edge clock. There are no other control signals.

```
module seq1 (DI, CLK, DO);
  input [7:0] DI;
  input CLK;
  output [7:0] DO;
  reg [7:0] DO;

  always @(posedge CLK) DO <= DI ;
    endmodule
```

Sequential Always Block Example Two

The following code example adds an active-High asynchronous reset.

```
module EXAMPLE (DI, CLK, ARST, DO);
  input [7:0] DI;
  input CLK, ARST;
  output [7:0] DO;
  reg [7:0] DO;

always @(posedge CLK or posedge ARST)
if (ARST == 1'b1)
  DO <= 8'b00000000;
else
  DO <= DI;
endmodule
```

Sequential `always` Block Example Three

The following code example describes an active-High asynchronous reset and an active-Low asynchronous set:

```
module EXAMPLE (DI, CLK, ARST, ASET, DO);
  input [7:0] DI;
  input CLK, ARST, ASET;
  output [7:0] DO;
  reg [7:0] DO;

always @(posedge CLK or posedge ARST or negedge ASET)
if (ARST == 1'b1)
  DO <= 8'b00000000;
else if (ASET == 1'b1) DO <= 8'b11111111;
else

   DO <= DI;
endmodule
```

Sequential `always` Block Example Four

The following code example describes a register with no asynchronous set/reset, and a synchronous reset.

```
module EXAMPLE (DI, CLK, SRST, DO);
  input [7:0] DI;
  input CLK, SRST;
  output [7:0] DO;
  reg [7:0] DO;

always @(posedge CLK)
```

```
if (SRST == 1'b1)
  DO <= 8'b00000000;
else
  DO <= DI;
```

Assign and De-assign Statements

Vivado synthesis does not support assign and de-assign statements.

Assignment Extension Past 32 Bits

If the expression on the left-hand side of an assignment is wider than the expression on the right-hand side, the left-hand side is padded to the left according to the following rules:

- If the right-hand expression is signed, the left-hand expression is padded with the sign bit.
- If the right-hand expression is unsigned, the left-hand expression is padded with 0 (zero).
- For unsized x or z constants only, the following rule applies:

  If the value of the right-hand expression's leftmost bit is z (high impedance) or x (unknown), regardless of whether the right-hand expression is signed or unsigned, the left-hand expression is padded with that value (z or x, respectively).

Tasks and Functions

- When the same code is used multiple times across a design, using tasks and functions:
  - Reduces the amount of code.
  - Facilitates maintenance.
- Tasks and functions must be declared and used in a module. The heading contains the following parameters:
  - Input parameters (only) for functions.
  - Input/output/inout parameters for tasks.
- The return value of a function is declared either signed or unsigned. The content is similar to the content of the combinatorial `always` block.

Tasks and Functions Example One

```
//
// An example of a function in Verilog
//
// File: functions_1.v
//
module functions_1 (A, B, CIN, S, COUT);
    input [3:0] A, B;
    input CIN;
    output [3:0] S;
    output COUT;
    wire [1:0] S0, S1, S2, S3;

    function signed [1:0] ADD;
        input A, B, CIN;
        reg S, COUT;
        begin
            S = A ^ B ^ CIN;
            COUT = (A&B) | (A&CIN) | (B&CIN);
            ADD = {COUT, S};
        end
    endfunction

    assign S0   = ADD (A[0], B[0], CIN),
           S1   = ADD (A[1], B[1], S0[1]),
           S2   = ADD (A[2], B[2], S1[1]),
           S3   = ADD (A[3], B[3], S2[1]),
           S    = {S3[0], S2[0], S1[0], S0[0]},
           COUT = S3[1];

endmodule
```

Tasks and Functions Example Two

In this coding example, the same functionality is described with a task.

```
// Verilog tasks
// tasks_1.v
//
module tasks_1 (A, B, CIN, S, COUT);
input [3:0] A, B;
input CIN;
output [3:0] S;
output COUT;
reg [3:0] S;
reg COUT;
reg [1:0] S0, S1, S2, S3;
```

```
task ADD;
input A, B, CIN;
output [1:0] C;
reg [1:0] C;
reg S, COUT;
  begin
    S = A ^ B ^ CIN;
    COUT = (A&B) | (A&CIN) | (B&CIN);
    C = {COUT, S};
  end
endtask

always @(A or B or CIN)
begin
  ADD (A[0], B[0], CIN, S0);
  ADD (A[1], B[1], S0[1], S1);
  ADD (A[2], B[2], S1[1], S2);
  ADD (A[3], B[3], S2[1], S3);
  S = {S3[0], S2[0], S1[0], S0[0]};
  COUT = S3[1];
end

endmodule
```

Recursive Tasks and Functions

Verilog-2001 supports recursive tasks and functions.

- Use recursion with the `automatic` keyword only.
- The number of recursions is automatically limited to prevent endless recursive calls. The default is 64.
- Use `-recursion_iteration_limit` to set the number of allowed recursive calls.

Recursive Tasks and Functions Example

```
function automatic [31:0] fac;
input [15:0] n;
if (n == 1)
 fac = 1;

else

 fac = n * fac(n-1); //recursive function call
endfunction
```

Constant Functions and Expressions

Vivado synthesis supports function calls to calculate constant values.

Constants are assumed to be decimal integers.

- Specify constants in binary, octal, decimal, or hexadecimal.
- To specify constants explicitly, prefix them with the appropriate syntax.

Constant Functions Example

```
// A function that computes and returns a constant value
//
// functions_constant.v
//
module functions_constant (clk, we, a, di, do);
    parameter ADDRWIDTH = 8;
    parameter DATAWIDTH = 4;
    input clk;
    input we;
    input  [ADDRWIDTH-1:0] a;
    input  [DATAWIDTH-1:0] di;
    output [DATAWIDTH-1:0] do;

    function integer getSize;
        input addrwidth;
        begin
            getSize = 2**addrwidth;
        end
    endfunction

    reg [DATAWIDTH-1:0] ram [getSize(ADDRWIDTH)-1:0];

    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
    end
    assign do = ram[a];

endmodule
```

Constant Expressions Example

The following constant expressions represent the same value.

- `4'b1010`
- `4'o12`
- `4'd10`
- `4'ha`

Blocking and Non-Blocking Procedural Assignments

Blocking and non-blocking procedural assignments have time control built into their respective assignment statements.

- The pound sign (#) and the at sign (@) are time control statements.
- These statements delay execution of the statement following them until the specified event is evaluated as true.
- The pound (#) delay is ignored for synthesis.

Blocking Procedural Assignment Syntax Example One

```
reg a;
a = #10 (b | c);
```

Blocking Procedural Assignment Syntax Example Two (Alternate)

```
if (in1) out = 1'b0;
else out = in2;
```

This assignment blocks the current process from continuing to execute additional statements at the same time, and is used mainly in simulation.

Non-Blocking Procedural Assignment Syntax Example One

```
variable <= @(posedge_or_negedge_bit) expression;
```

Non-blocking assignments evaluate the expression when the statement executes, and allow other statements in the same process to execute at the same time. The variable change occurs only after the specified delay.

Non-Blocking Procedural Assignment Example Two

This coding example shows how to use a non-blocking procedural assignment.

```
if (in1) out <= 1'b1;
else out <= in2;
```

Verilog Macros

Verilog defines macros as follows:

```
'define TESTEQ1 4'b1101
```

The defined macro is referenced later, as follows:

```
if (request == 'TESTEQ1)
```

The `'ifdef` and `'endif` constructs do the following:

- Determine whether a macro is defined.
- Define conditional compilation.

If the macro called out by `'ifdef` is defined, that code is compiled.

- If the macro has not been defined, the code following the `'else` command is compiled.
- The `'else` is not required, but `'endif` must complete the conditional statement.

Use the Verilog Macros command line option to define (or redefine) Verilog macros.

- Verilog Macros let you modify the design without modifying the HDL source code.
- Verilog Macros is useful for IP core generation and flow testing.

Macro Example One

```
'define myzero 0
assign mysig = 'myzero;
```

Macro Example Two

```
'ifdef MYVAR
module if_MYVAR_is_declared;
...
endmodule
'else
module if_MYVAR_is_not_declared;
...
endmodule
'endif
```

Include Files

Verilog allows you to separate HDL source code into more than one file. To reference the code in another file, use the following syntax in the current file:

```
'include "path/file-to-be-included"
```

- The path is relative or absolute.
- Multiple `'include` statements are allowed in the same Verilog file. This makes your code more manageable in a team design environment in which different files describe different modules.
- To allow the file in your `'include` statement to be recognized, identify the directory in which it resides to Vivado synthesis, add the file to your project directory. Vivado synthesis searches the project directory by default.
- Include a relative or absolute path in the `'include` statement. This path points the Vivado tools to a directory other than the project directory. Use Verilog Include file search paths, `include_dirs`. This option points Vivado synthesis directly to the include file directory.
- If the include file is required for the Vivado Design Suite to construct the design hierarchy, the file must reside in the project directory, or be referenced by a relative or absolute path. The file need not be added to the project.

Behavioral Verilog Comments

Behavioral Verilog comments are similar to the comments in such languages as C++.

One-Line Comments

One-line comments start with a double forward slash (//).

```
// This is a one-line comment.
```

Multiple-Line Block Comments

Multiple-line block comments start with /* and end with */.

```
/* This is a multiple-line comment.
*/
```

Generate Statements

Behavioral Verilog generate statements:

- Allow you to create:
  - Parameterized and scalable code.
  - Repetitive or scalable structures.
  - Functionality conditional on a particular criterion being met.
- Are resolved during Verilog elaboration.
- Are conditionally instantiated into your design.
- Are described within a module scope.
- Start with a `generate` keyword.
- End with an `endgenerate` keyword.

Structures Created Using Generate Statements

Structures likely to be created using a generate statement include:

- Primitive or module instances
- Initial or always procedural blocks
- Continuous assignments
- Net and variable declarations
- Parameter redefinitions
- Task or function definitions

Supported Generate Statements

Vivado synthesis supports all Behavioral Verilog generate statements:

- generate-loop (`generate-for`)
- generate-conditional (`generate-if-else`)
- generate-case (`generate-case`)

Generate Loop Statements

Use a `generate-for` loop to create one or more instances that can be placed inside a module.

Use the `generate-for` loop the same way you use a normal Verilog for loop, with the following limitations:

- The `generate-for` loop index has a `genvar` variable.
- The assignments in the `for` loop control refers to the `genvar` variable.
- The contents of the `for` loop are enclosed by begin and end statements.
- The begin statement is named with a unique qualifier.

Generate Loop Statement 8-Bit Adder Example

```
generate genvar i;
  for (i=0; i<=7; i=i+1)
  begin : for_name
    adder add (a[8*i+7 : 8*i], b[8*i+7 : 8*i], ci[i], sum_for[8*i+7 :
8*i], c0_or[i+1]);
  end
endgenerate
```

Generate Conditional Statements

A `generate-if-else` statement conditionally controls which objects are generated.

- Each branch of the `if-else` statement is enclosed by begin and end statements.
- The begin statement is named with a unique qualifier.

Generate Conditional Statement Coding Example

This coding example instantiates two different implementations of a multiplier based on the width of data words.

```
generate
  if (IF_WIDTH < 10)
  begin : if_name
    multiplier_imp1 # (IF_WIDTH) u1 (a, b, sum_if);
  end
else
  begin : else_name
    multiplier_imp2 # (IF_WIDTH) u2 (a, b, sum_if);
  end
endgenerate
```

Generate Case Statements

A generate-case statement conditionally controls which objects are generated under which conditions.

- Each branch in a generate-case statement is enclosed by begin and end statements.
- The begin statement is named with a unique qualifier.

Behavioral Verilog Generate Case Statements Coding Example

This coding example instantiates more than two different implementations of an `adder` based on the width of data words.

```
generate
case (WIDTH)
1:
 begin : case1_name
adder #(WIDTH*8) x1 (a, b, ci, sum_case, c0_case);
 end
2:
 begin : case2_name
adder #(WIDTH*4) x2 (a, b, ci, sum_case, c0_case);
     end default:
begin : d_case_name
adder x3 (a, b, ci, sum_case, c0_case);
endcase endgenerate
end
```

SystemVerilog Support

Introduction

Vivado® synthesis supports the subset of SystemVerilog RTL that can be synthesized. These data types are described in the following sections.

Targeting SystemVerilog for a Specific File

By default, the Vivado synthesis tool compiles `*.v` files with the Verilog 2005 syntax and `*.sv` files with the SystemVerilog syntax.

To target SystemVerilog for a specific `*.v` file in the Vivado IDE:

1. Right-click the file, and select **Source Node Properties**.
2. In the Source File Properties window, change the File Type from **Verilog** to **SystemVerilog**, and click **OK**.

Tcl Command to Set Properties

Alternatively, you can use the following Tcl command in the Tcl Console:

```
set_property file_type SystemVerilog [get_files <filename>.v]
```

The following sections describe the supported SystemVerilog types in the Vivado IDE.

Data Types

The following data types are supported, as well as the mechanisms to control them.

Declaration

Declare variables in the RTL as follows:

```
[var] [DataType] name;
```

Where:

- `var` is optional and implied if not in the declaration.
- `DataType` is one of the following:
  - `integer_vector_type`: `bit`, `logic`, or `reg`
  - `integer_atom_type`: `byte`, `shortint`, `int`, `longint`, `integer`, or `time`
  - `non_integer_type`: `shortreal`, `real`, or `realtime`
  - `struct`
  - `enum`

Integer Data Types

SystemVerilog supports the following integer types:

- `shortint`: 2-state 16-bit signed integer
- `int`: 2-state 32-bit signed integer
- `longint`: 2-state 64-bit signed integer
- `byte`: 2-state 8-bit signed integer
- `bit`: 2-state, user defined vector size
- `logic`: 4-state user defined vector size
- `reg`: 4-state user-defined vector size
- `integer`: 4-state 32-bit signed integer
- `time`: 4-state 64-bit unsigned integer

4-state and 2-state refer to the values that can be assigned to those types, as follows:

- 2-state allows 0s and 1s.
- 4-state also allows X and Z states.

X and Z states cannot always be synthesized; therefore, items that are 2-state and 4-state are synthesized in the same way.

CAUTION! *Take care when using 4-state variables: RTL versus simulation mismatches could occur.*

- The types `byte`, `shortint`, `int`, `integer`, and `longint` default to signed values.
- The types `bit`, `reg`, and `logic` default to unsigned values.

Real Numbers

Synthesis supports real numbers; however, they cannot be used to create logic. They can only be used as parameter values. The SystemVerilog-supported real types are:

- `real`
- `shortreal`
- `realtime`

Void Data Type

The `void` data type is only supported for functions that have no return value.

User-Defined Types

Vivado synthesis supports user-defined types, which are defined using the `typedef` keyword. Use the following syntax:

```
typedef data_type type_identifier {size};
```

or

```
typedef [enum, struct] type_identifier;
```

Enum Types

Enumerated types can be declared with the following syntax:

```
enum [type] {enum_name1, enum_name2...enum_namex} identifier
```

If no type is specified, the `enum` defaults to `int`. Following is an example:

```
enum {sun, mon, tues, wed, thurs, fri, sat} day_of_week;
```

This code generates an `enum` of `int` with seven values. The values that are given to these names start with 0 and increment, so that, `sun = 0` and `sat = 6`.

To override the default values, use code as in the following example:

```
enum {sun=1, mon, tues, wed, thurs, fri, sat} day_of week;
```

In this case, `sun` is 1 and `sat` is 7.

The following is another example how to override defaults:

```
enum {sun, mon=3, tues, wed, thurs=10, fri=12, sat} day_of_week;
```

In this case, `sun=0`, `mon=3`, `tues=4`, `wed=5`, `thurs=10`, `fri=12`, and `sat=13`.

Enumerated types can also be used with the `typedef` keyword.

```
typedef enum {sun,mon,tues,wed,thurs,fri,sat} day_of_week;
day_of_week my_day;
```

The preceding example defines a signal called `my_day` that is of type `day_of_week`. You can also specify a range of `enums`. For example, the preceding example can be specified as:

```
enum {day[7]} day_of_week;
```

This creates an enumerated type called `day_of_week` with seven elements as follows: `day0`, `day1`...`day6`.

Following are other ways to use enumerated types:

```
enum {day[1:7]} day_of_week;  // creates day1,day2...day7
enum {day[7] = 5} day_of_week; //creates day0=5, day1=6... day6=11
```

Constants

SystemVerilog gives three types of elaboration-time constants:

- `parameter`: Is the same as the original Verilog standard and can be used in the same way.
- `localparam`: Is similar to `parameter` but cannot be overridden by upper-level modules.
- `specparam`: Is used for specifying delay and timing values; consequently, this value is not supported in Vivado synthesis.

There is also a run-time constant declaration called `const`.

Type Operator

The type operator allows parameters to be specified as data types, which allows modules to have different types of parameters for different instances.

Casting

Assigning a value of one data type to a different data type is illegal in SystemVerilog. However, a workaround is to use the cast operator ('). The cast operator converts the data type when assigning between different types. The usage is:

```
casting_type'(expression)
```

The `casting_type` is one of the following:

- `integer_type`
- `non_integer_type`
- `real_type`
- constant unsigned number
- user-created signing value type

Aggregate Data Types

In aggregate data types there are *structures* and *unions*, which are described in the following subsections.

Structures

A structure is a collection of data that can be referenced as one value, or the individual members of the structure. This is similar to the VHDL concept of a record. The format for specifying a structure is:

```
struct {struct_member1; struct_member2;...struct_memberx;}
structure_name;
```

Unions

Unions are ***not supported*** in Vivado synthesis.

Packed and Unpacked Arrays

Vivado synthesis supports both packed and unpacked arrays:

```
logic [5:0] sig1;  //packed array
logic sig2 [5:0]; //unpacked array
```

Data types with predetermined widths do not need the packed dimensions declared:

```
integer sig3;  //equivalent to logic signed [31:0] sig3
```

Processes

Always Procedures

There are four `always` procedures:

- `always`
- `always_comb`
- `always_latch`
- `always_ff`

The procedure `always_comb` describes combinational logic. A sensitivity list is inferred by the logic driving the `always_comb` statement.

For `always` you must provide the sensitivity list. The following examples use a sensitivity list of `in1` and `in2`:

```
always@(in1 or in2)
out1 = in1 & in2;
always_comb out1 = in1 & in2;
```

The procedure `always_latch` provides a quick way to create a latch. Like `always_comb`, a sensitivity list is inferred, but you must specify a control signal for the latch enable, as in the following example:

```
always_latch
 if(gate_en) q <= d;
```

The procedure `always_ff` is a way to create Flip-Flops. Again, you must specify a sensitivity list:

```
always_ff@(posedge clk)
out1 <= in1;
```

Block Statements

Block statements provide a mechanism to group sets of statements together. Sequential blocks have a `begin` and `end` around the statement. The block can declare its own variables, and those variables are specific to that block. The sequential block can also have a name associated with that block. The format is as follows:

```
begin [: block name]
[declarations]
[statements]
end [: block name]

begin : my_block
logic temp;
temp = in1 & in2;
out1 = temp;
end : my_block
```

In the previous example, the block name is also specified after the `end` statement. This makes the code more readable, but it is not required.

Note: Parallel blocks (or fork join blocks) are *not* supported in Vivado synthesis.

Procedural Timing Controls

SystemVerilog has two types of timing controls:

- Delay control: Specifies the amount of time between the statement its execution. This is not useful for synthesis, and Vivado synthesis ignores the time statement while still creating logic for the assignment.
- Event control: Makes the assignment occur with a specific event; for example, `always@(posedge clk)`. This is standard with Verilog, but SystemVerilog includes extra functions.

The logical `or` operator is an ability to give any number of events so that any event triggers the execution of the statement. To do this, use either a specific `or`, or separate with commas in the sensitivity list. For example, the following two statements are the same:

```
always@(a or b or c)
always@(a,b,c)
```

SystemVerilog also supports the implicit `event_expression @*`. This helps to eliminate simulation mismatches caused because of incorrect sensitivity lists, for example:

```
Logic always@* begin
```

Operators

Vivado synthesis supports the following SystemVerilog operators:

- Assignment operators

  (=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, >>>=)
- Unary operators (+, -, !, ~, &, ~&, |, ~|, ^, ~^, ^~)
- Increment/decrement operators (++, --)
- Binary operators (+, -, *, /, %, ==, ~=, ===, ~==, &&, ||, **, < , <=, >, >=, &, |, ^, ^~, ~^, >>, <<, >>>, <<<)

  Note: A**B is supported if A is a power of 2 or B is a constant.
- Conditional operator (? :)
- Concatenation operator ({…})

Signed Expressions

Vivado synthesis supports both signed and unsigned operations. Signals can be declared as unsigned or signed. For example:

```
logic [5:0] reg1;
logic signed [5:0] reg2;
```

Procedural Programming Assignments

Conditional if-else Statement

The syntax for a conditional if-else statement is:

```
if (expression)
  command1;
else
  command2;
```

The else is optional and assumes a latch or flip-flop depending on whether or not there was a clock statement. Code with multiple if and else entries can also be supported, as shown in the following example:

```
If (expression1)
   Command1;
else if (expression2)
  command2;
else if (expression3)
  command3;
else
  command4;
```

This example is synthesized as a priority `if` statement.

- If the first expression is found to be `TRUE`, the others are not evaluated.
- If unique or priority `if-else` statements are used, Vivado synthesis treats those as `parallel_case` and `full_case`, respectively.

Case Statement

The syntax for a `case` statement is:

```
case (expression)
  value1: statement1;
  value2: statement2;
  value3: statement3;
  default: statement4;
endcase
```

The default statement inside a `case` statement is optional. The values are evaluated in order, so if both `value1` and `value3` are true, `statement1` is performed.

In addition to `case`, there are also the `casex` and `casez` statements. These let you handle don't cares in the values (`casex`) or tri-state conditions in the values (`casez`).

If unique or priority case statements are used, Vivado synthesis treats those as `parallel_case` and `full_case` respectively.

Loop Statements

Several types of loops that are supported in Vivado synthesis and SystemVerilog. One of the most common is the `for` loop. Following is the syntax:

```
for (initialization; expression; step)
statement;
```

A `for` loop starts with the initialization, then evaluates the expression. If the expression evaluates to 0, it stops, else if the expression evaluates to 1, it continues with the statement. When it is done with the statement, it executes the step function.

- A `repeat` loop works by performing a function a stated number of times. Following is the syntax:

  ```
  repeat (expression)
    statement;
  ```

  This syntax evaluates the expression to a number, then executes the statement the specified number of times.

- The `for-each` loop executes a statement for each element in an array.

- The `while` loop takes an expression and a statement and executes the statement until the expression is `false`.
- The `do-while` loop performs the same function as the while loop, but instead it tests the expression after the statement.
- The `forever` loop executes all the time. To avoid infinite loops, use it with the break statement to get out of the loop.

Tasks and Functions

Tasks

The syntax for a `task` declaration is:

```
task name (ports);
  [optional declarations];
  statements;
endtask
```

Following are the two types of tasks:

- **Static task**: Declarations retain their previous values the next time the task is called.
- **Automatic task**: Declarations do not retain previous values.

CAUTION! *Be careful when using these tasks; Vivado synthesis treats all tasks as automatic.*

Many simulators default to static tasks if the static or automatic is not specified, so there is a chance of simulation mismatches. The way to specify a task as automatic or static is the following:

```
task automatic my_mult... //or
task static my_mult ...
```

Functions (Automatic and Static)

Functions are similar to tasks, but return a value. The format for a function is:

```
function data_type function_name(inputs);
 declarations;
 statements;
endfunction : function_name
```

The final `function_name` is optional but does make the code easier to read.

Because the function returns a value, it must either have a return statement or specifically state the function name:

```
function_name = ....
```

Like tasks, functions can also be automatic or static.

CAUTION! *Vivado synthesis treats all functions as automatic. However, some simulators might behave differently. Be careful when using these functions with third-party simulators.*

Modules and Hierarchy

Using modules in SystemVerilog is very similar to Verilog, and includes additional features as described in the following subsections.

Connecting Modules

There are three main ways to instantiate and connect modules:

- The first two are by ordered list and by name, as in Verilog.
- The third is by named ports.

If the names of the ports of a module match the names and types of signals in an instantiating module, the lower-level module can by hooked up by name. For example:

```
module lower (
 output [4:0] myout;
 input clk;
 input my_in;
 input [1:0] my_in2;
... ...
endmodule
//in the instantiating level.
lower my_inst (.myout, .clk, .my_in, .my_in2);
```

Connecting Modules with Wildcard Ports

You can use wildcards when connecting modules. For example, from the previous example:

```
// in the instantiating module
lower my_inst (.*);
```

This connects the entire instance, as long as the upper-level module has the correct names and types.

In addition, these can be mixed and matched. For example:

```
lower my_inst (.myout(my_sig), .my_in(din), .*);
```

This connects the `myout` port to a signal called `my_sig`, the `my_in` port to a signal called `din` and `clk` and `my_in2` is hooked up to the `clk` and `my_in2` signals.

Interfaces

Interfaces provide a way to specify communication between blocks. An interface is a group of nets and variables that are grouped together for the purpose of making connections between modules easier to write. The syntax for a basic interface is:

```
interface interface_name;
parameters and ports;
items;
endinterface : interface_name
```

The `interface_name` at the end is optional but makes the code easier to read. For an example, see the following code:

```
module bottom1 (
  input clk,
  input [9:0] d1,d2,
  input s1,
  input [9:0] result,
  output logic sel,
  output logic [9:0] data1, data2,
  output logic equal);

//logic//

endmodule

module bottom2  (
  input clk,
  input sel,
  input [9:0] data1, data2,
  output logic [9:0] result);

//logic//

endmodule

module top (
  input clk,
  input s1,
```

```
    input [9:0] d1, d2,
    output equal);

    logic [9:0] data1, data2, result;
    logic sel;

    bottom1 u0 (clk, d1, d2, s1, result, sel, data1, data2, equal);
    bottom2 u1 (clk, sel, data1, data2, result);
  endmodule
```

The previous code snippet instantiates two lower-level modules with some signals that are common to both.

These common signals can all be specified with an interface:

```
  interface my_int
    logic sel;
    logic [9:0] data1, data2, result;
  endinterface : my_int
```

Then, in the two bottom-level modules, you can change to:

```
  module bottom1 (
    my_int int1,
    input clk,
    input [9:0] d1, d2,
    input s1,
    output logic equal);
```

and

```
    module bottom2 (
    my_int int1,
    input clk);
```

Inside the modules, you can also change how you access `sel`, `data1`, `data2`, and `result`. This is because, according to the module, there are no ports of these names. Instead, there is a port called `my_int`. This requires the following change:

```
  if (sel)
  result <= data1;
  to:
  if (int1.sel)
  int1.result <= int1.data1;
```

Finally, in the top-level module, the interface must be instantiated, and the instances reference the interface:

```
  module top(
    input clk,
    input s1,
    input [9:0] d1, d2,
    output equal);
    my_int int3();  //instantiation
  bottom1 u0 (int3, clk, d1, d2, s1, equal);
    bottom2 u1 (int3, clk);
endmodule
```

Modports

In the previous example, the signals inside the interface are no longer expressed as inputs or outputs. Before the interface was added, the port `sel` was an output for `bottom1` and an input for `bottom2`.

After the interface is added, that is no longer clear. In fact, the Vivado synthesis engine does not issue a warning that these are now considered bidirectional ports, and in the netlist generated with hierarchy, these are defined as `inouts`. This is not an issue with the generated logic, but it can be confusing.

To specify the direction, use the `modport` keyword, as shown in the following code snippet:

```
interface my_int;
        logic sel;
        logic [9:0] data1, data2, result;

        modport b1 (input result, output sel, data1, data2);
        modport b2 (input sel, data1, data2, output result);
endinterface : my_int
```

Then, in the bottom modules, use when declared:

```
module bottom1 (
  my_int.b1 int1,
```

This correctly associates the inputs and outputs.

Miscellaneous Interface Features

In addition to signals, there can also be tasks and functions inside the interface. This lets you create tasks specific to that interface. Interfaces can be parameterized. In the previous example, `data1` and `data2` were both 10-bit vectors, but you can modify those interfaces to be any size depending on a parameter that is set.

Packages

Packages provide an additional way to share different constructs. They have similar behavior to VHDL packages. Packages can contain functions, tasks, types, and enums. The syntax for a package is:

```
package package_name;
  items
endpackage : package_name
```

The final `package_name` is not required, but it makes code easier to read. Packages are then referenced in other modules by the import command. Following is the syntax:

```
import package_name::item or *;
```

The `import` command must include items from the package to import or must specify the whole package.

Mixed Language Support

Introduction

Vivado synthesis supports VHDL and Verilog mixed language projects except as otherwise noted.

Mixing VHDL and Verilog

- Mixing VHDL and Verilog is restricted to design unit (cell) instantiation.
  - A Verilog module can be instantiated from VHDL code.
  - A VHDL entity can be instantiated from Verilog code.
  - No other mixing between VHDL and Verilog is supported. For example, you cannot embed Verilog source code directly in VHDL source code.
- In a VHDL design, a restricted subset of VHDL types, generics, and ports is allowed on the boundary to a Verilog module.
- In a Verilog design, a restricted subset of Verilog types, parameters, and ports is allowed on the boundary to a VHDL entity or configuration.
- Vivado synthesis binds VHDL design units to a Verilog module during HDL elaboration.
- The VHDL and Verilog files that make up a project are specified in a unique HDL project file.

Instantiation

- Component instantiation based on default binding is used for binding Verilog modules to a VHDL design unit.
- For a Verilog module instantiation in VHDL, Vivado synthesis does not support:
  - Configuration specification
  - Direct instantiation
  - Component configurations

VHDL and Verilog Libraries

- VHDL and Verilog libraries are logically unified.
- The default work directory for compilation is available to both VHDL and Verilog.
- Mixed language projects accept a search order for searching unified logical libraries in design units (cells). Vivado synthesis follows this search order during elaboration to select and bind a VHDL entity or a Verilog module to the mixed language project.

VHDL and Verilog Boundary Rules

The boundary between VHDL and Verilog is enforced at the design unit level.

- A VHDL entity or architecture can instantiate a Verilog module. See Instantiating VHDL in Verilog in the following section.
- A Verilog module can instantiate a VHDL entity. See Instantiating Verilog in VHDL.

Instantiating VHDL in Verilog

To instantiate a VHDL design unit in a Verilog design:

1. Declare a module name with the same as name as the VHDL entity that you want to instantiate (optionally followed by an architecture name).
2. Perform a normal Verilog instantiation.

Limitations (VHDL in Verilog)

Vivado synthesis has the following limitations when instantiating a VHDL design unit in a Verilog module:

- The only VHDL construct that can be instantiated in a Verilog design is a VHDL entity.
  - No other VHDL constructs are visible to Verilog code.
  - Vivado synthesis uses the entity-architecture pair as the Verilog-VHDL boundary.
- Use explicit port association. Specify formal and effective port names in the port map.
- All parameters are passed at instantiation, even if they are unchanged.
- The parameter override is named and not ordered. The parameter override occurs through instantiation, not through `defparams`.

Binding

Vivado synthesis performs binding during elaboration. During binding:

1. Vivado synthesis searches for a Verilog module with the same name as the instantiated module in the following:
   - User-specified list of unified logical libraries
   - User-specified order
2. Vivado synthesis ignores any architecture name specified in the module instantiation.
3. If Vivado synthesis finds the Verilog module, Vivado synthesis binds the name.
4. If Vivado synthesis does not find the Verilog module:
   - Vivado synthesis treats the Verilog module as a VHDL entity.
   - Vivado synthesis searches for the first VHDL entity matching the name using a case sensitive search for a VHDL entity in the user-specified list of unified logical libraries or the user-specified order.

   Note: This assumes that a VHDL design unit was stored with extended identifier.

Limitations (Verilog from VHDL)

Vivado synthesis has the following limitations when instantiating a VHDL design unit from a Verilog module:

- Use explicit port association. Specify formal and effective port names in the port map.
- All parameters are passed at instantiation, even if they are unchanged.
- The parameter override is named and not ordered. The parameter override occurs through instantiation, and not through `defparams`.

Acceptable Example

```
ff #(.init(2'b01)) u1 (.sel(sel), .din(din), .dout(dout));
```

NOT Acceptable Example

```
ff u1 (.sel(sel), .din(din), .dout(dout));
defparam u1.init = 2'b01;
```

Instantiating Verilog in VHDL

To instantiate a Verilog module in a VHDL design:

1. Declare a VHDL component with the same name as the Verilog module to be instantiated.
2. Observe case sensitivity.
3. Instantiate the Verilog component as if you were instantiating a VHDL component.
    - Binding a component to a specific design unit from a specific library by using a VHDL configuration declaration is not supported. Only the default Verilog module binding is supported.
    - The only Verilog construct that can be instantiated in a VHDL design is a Verilog module. No other Verilog constructs are visible to VHDL code.
    - During elaboration, Vivado synthesis treats all components subject to default binding as design units with the same name as the corresponding component name.
    - During binding, Vivado synthesis treats a component name as a VHDL design unit name and searches for it in the logical library work.
        - If Vivado synthesis finds a VHDL design unit, Vivado synthesis binds it.
        - If Vivado synthesis does not find a VHDL design unit:

          Vivado synthesis treats the component name as a Verilog module name and searches for it using a case sensitive search. Then Vivado synthesis selects and binds the first Verilog module matching the name.

Because libraries are unified, a Verilog cell with the same name as a VHDL design unit cannot exist in the same logical library.

A newly-compiled cell or unit overrides a previously-compiled cell or unit.

Generics Support

Vivado synthesis supports the following VHDL generic types and their Verilog equivalents for mixed language designs.

- integer
- real
- string
- boolean

Port Mapping

Vivado synthesis supports port mapping for VHDL instantiated in Verilog and Verilog instantiated in VHDL.

Port Mapping for VHDL Instantiated in Verilog

When a VHDL entity is instantiated in a Verilog module, formal ports can have the following characteristics:

- Allowed directions
  - in
  - out
  - inout
- Unsupported directives
  - buffer
  - linkage
- Allowed data types
  - bit
  - bit_vector
  - std_logic
  - std_ulogic
  - std_logic_vector
  - std_ulogic_vector

Port Mapping for Verilog Instantiated in VHDL

When a Verilog module is instantiated in a VHDL entity or architecture, formal ports can have the following characteristics:

- Allowed directions are: `input`, `output`, and `inout`.
- Allowed data types are: `wire` and `reg`
- Vivado synthesis *does not* support:
  - Connection to bidirectional pass options in Verilog.
  - Unnamed Verilog ports for mixed language boundaries.

Use an equivalent component declaration to connect to a case sensitive port in a Verilog module. Vivado synthesis assumes Verilog ports are in all lowercase.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx® Support website at: www.xilinx.com/support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

References

Vivado Documentation

1. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
2. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
3. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
4. *Vivado Design Suite User Guide: Using the Tcl Scripting Capabilities* (UG894)
5. *Vivado Design Suite User Guide: Implementation* (UG904)
6. *Vivado Design Suite User Guide: Hierarchical Design* (UG905)
7. *Vivado Design Suite Migration Guide* (UG911)
8. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
9. *Vivado Design Suite User Guide: Using Constraints* (UG903)
10. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)

11. *Vivado Design Suite Tutorial: Using Constraints* (UG945)
12. *Vivado Design Suite Properties Reference Guide* (UG912)
13. *7 Series DSP48E1 Slice User Guide* (UG479)
14. *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899)
15. *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
16. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
17. *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907)
18. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
19. *Vivado Design Suite User Guide: Designing with IP* (UG896)
20. *Vivado Design Suite Tutorial: Creating and Packaging Custom IP* (UG1119)
21. *UltraScale Architecture Memory Resources User Guide* (UG573)

Vivado Design Suite Documentation

Synthesis Coding Examples

Coding Examples

Vivado QuickTake Videos

QuickTake Video: Synthesis Options

QuickTake Video: Creating and Managing Runs

QuickTake Video: Advanced Synthesis using Vivado

Vivado Design Suite QuickTake Video Tutorials

Training Resources

Xilinx provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related training resources:

1. Essentials of FPGA Design
2. Static Timings Analysis and Design Contraints

Please Read: Important Legal Notices