Vivado Design Suite User Guide

Synthesis

UG901 (v2014.4) December 19, 2014

Revision History

The following table shows the revision history for this document:

Date	Version	Revision
12/19/2014	2014.4	Corrected VHDL files and references. Further described "default_state" Removed "mandatory" on the -top option.
11/19/2014	2014.4	Changed DONT_TOUCH to = "yes" in Verilog Module Example, page 39. Added `dynpreaddmutladdr.v` and `dynpreaddmutladdr.vhd` to document and zip file.
10/08/2014	2014.3	Added supported VHDL, Verilog, and Mixed Language structures in Chapter 4, VHDL Support, Chapter 5, Verilog Language Support, and Chapter 6, Mixed Language Support. Added new figures of software throughout. Added note on No NCG support. Rewrote Setting a Bottom-Up Out-of-Context Flow in Chapter 1. Rewrote Manually Setting a Bottom-Up Flow and Importing Netlists in Chapter 1. Changed Tcl Script Example in Chapter 1. Added new strategies: Chapter 1, Using Synthesis Settings . Added DIRECT_ENABLE, DIRECT_RESET to Attributes. Changed descriptions on DONT_TOUCH, and KEEP. Changed description in Using Synthesis Attributes in XDC files. Added Pre-Adder in the DSP Block in Chapter 3. Using the Squarer in the UltraScale DSP Block in Chapter 3. Added eight code examples to ug901-vivado-synthesis-`examples.zip`.
06/04/2014	2014.2	Added a "none" value to FSM_ENCODING and FSM Components. Removed "clock" from IO_BUFF_TYPE. Added a readme to the ug901-code-examples.zip. Changed the description and figure in Viewing Reports, page 30 to match the reports available after running synthesis only.
04/23/2014	2014.1	Initial release for 2014.1.

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)

  IEEE Standard for SystemVerilog 2008

- Verilog

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

  IEEE Standard for Verilog Hardware Description Language (IEEE Std 2008)

- VHDL

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

- 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 6].

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 7]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, page 12.
- Queue the synthesis and implementation runs to launch sequentially or simultaneously with multi-processor machines. See Running Synthesis, page 16.
- Monitor synthesis or implementation progress, view log reports, and cancel runs. See Monitoring the Synthesis Run, page 24.

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]

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.

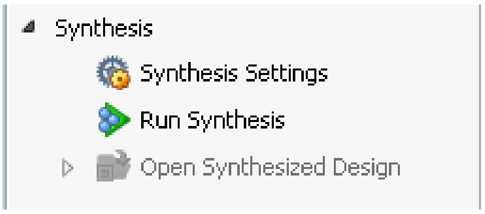
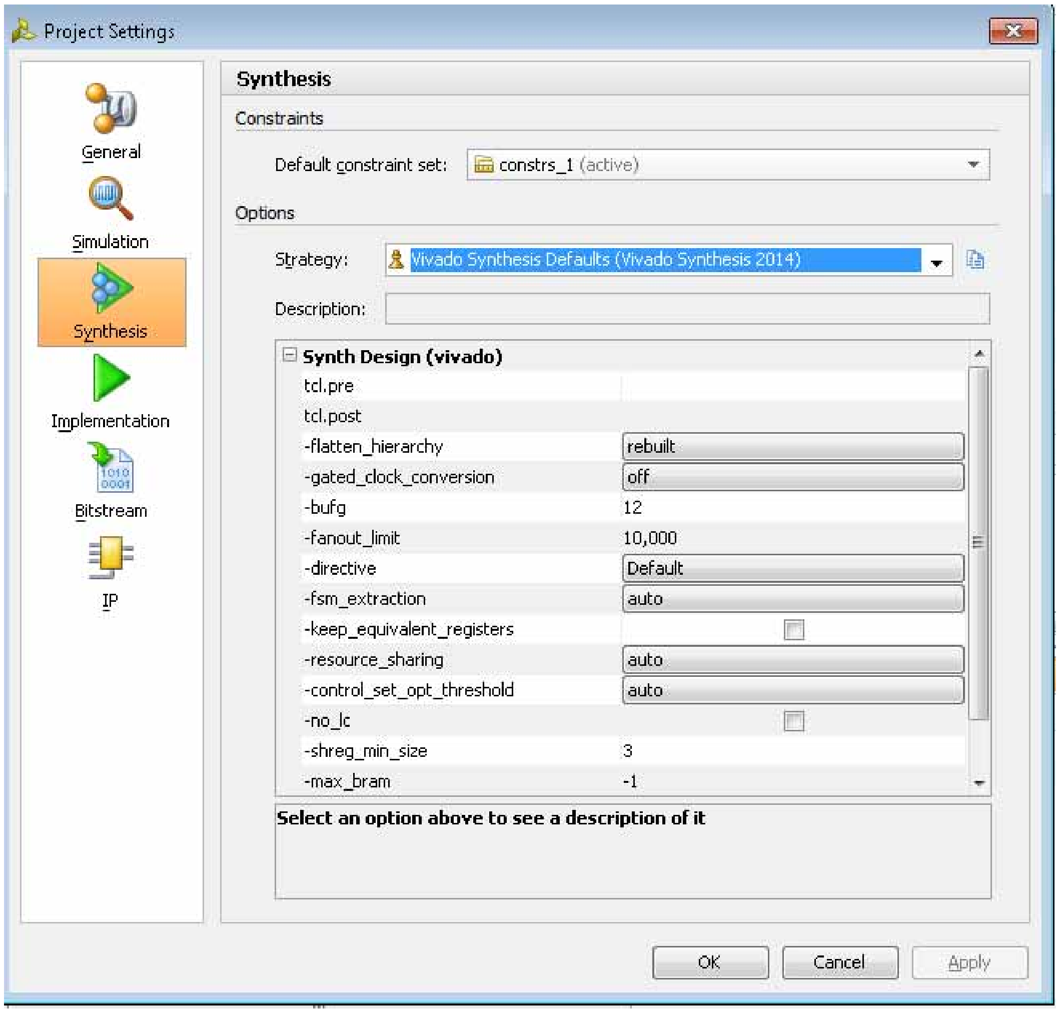


Figure 1-1: **Flow Navigator: Synthesis**

The Project Settings dialog box opens, as shown in Figure 1-2.



Synthesis

Constraints

Default constraint set: constrs_1 (active)

Options

Strategy: Vivado Synthesis Defaults (Vivado Synthesis 2014)

Description:

Synth Design (vivado)	
tcl.pre	
tcl.post	
-flatten_hierarchy	rebuilt
-gated_clock_conversion	off
-bufg	12
-fanout_limit	10,000
-directive	Default
-fsm_extraction	auto
-keep_equivalent_registers	☐
-resource_sharing	auto
-control_set_opt_threshold	auto
-no_lc	☐
-shreg_min_size	3
-max_bram	-1

Select an option above to see a description of it

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 *Vivado Design Suite User Guide: Using Constraints (UG903)* [Ref 8] for more information.

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 four different preconfigured strategies: **Vivado Synthesis Defaults**, **Flow_RuntimeOptimized**, **Flow_AreaOptimized_High**, and **Flow_PerfOptimized_High**.

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, page 12.

Table 1-1 lists the Run Strategy options, their default settings, and other options. Following the table is a full description of each option.

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

Options\Strategies	Default	Flow_RunTimeOptimized	Flow_Flow_AreaOptimized_High	Flow_PerfOptimized_High
–flatten_hierarchy	rebuilt	none	rebuilt	rebuilt
–gated_clock_conversion	off	off	off	off
–bufg	12	12	12	12
–fanout_limit	10,000	10,000	10,000	400
–directive	default	RunTimeOptimized	AreaOptimizedHigh	Default
–fsm_extraction	auto	off	auto	one_hot
-keep_equivalent_registers	unchecked	unchecked	unchecked	checked
-resource_sharing	auto	auto	auto	off
-control_set_opt_threshold	auto	auto	1	auto
-no_lc	unchecked	unchecked	unchecked	checked

Table 1-1: **Vivado Preconfigured Strategies *(Cont'd)***

Options\Strategies	Default	Flow_RunTimeOptimized	Flow_Flow_AreaOptimized_High	Flow_PerfOptimized_High
-shreg_min_size	3	3	3	5
-max_bram	-1	-1	-1	-1
-max_dsp	-1	-1	-1	-1

c. Select from the displayed **Options**, which are:

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 the *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]
```

- **–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, page 42, for more information.

- **–bufg**: Controls how many `BUFGs` the tool infers in the design. The Vivado design tools use this option when other `BUFGs` in the design netlists are not visible to the synthesis process.

The tool infers up to the amount specified, and tracks how many BUFGs are instantiated in the RTL. For example, if the -bufg option is set to 12 and there are three BUFGs instantiated in the RTL, the tool infers up to nine more BUFGs.

- **-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 the 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. Values are Default and RuntimeOptimized, which runs synthesis quickly and with less optimization.
- **-fsm_extraction**: Controls how synthesis extracts and maps finite state machines. FSM_ENCODING, page 40, describes the options in more detail.
- **-keep_equivalent_registers**: Prevents merging of registers with the same input logic. See KEEP, page 44 for more information.
- **-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.

TIP: *Register merging can be prevented using the* KEEP *attribute, which prevents 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. See KEEP, page 44 for more information.*

- **-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). See SHREG_EXTRACT, page 50 for more information.

- **-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_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.

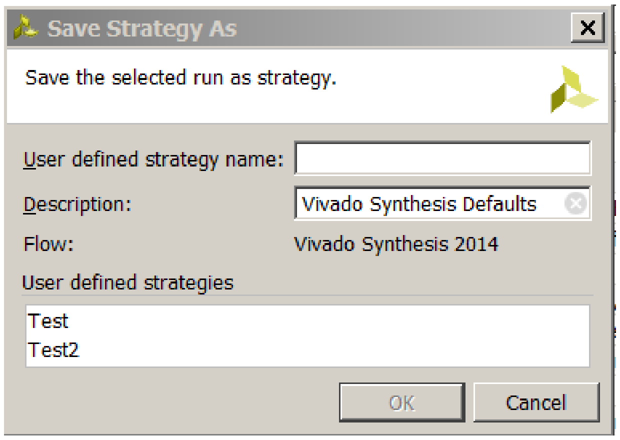
  Note: The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.

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. Strategies are tool- and version-specific. Each major release has version-specific options.

1. Select either the **Flow Navigator > Synthesis Settings**, button, or **Tools > Options > Project Settings**, and click the **Synthesis** button.
2. Click the **Save Strategy As** button. This button is also available on the **Strategy** option in **Synthesis Settings**.

   The Save Strategy As dialog box opens, as shown in Figure 1-3, page 12.



Save Strategy As

Save the selected run as strategy.

User defined strategy name:

Description: Vivado Synthesis Defaults

Flow: Vivado Synthesis 2014

User defined strategies

Test
Test2

Figure 1-3: **Save Strategy As Dialog Box**

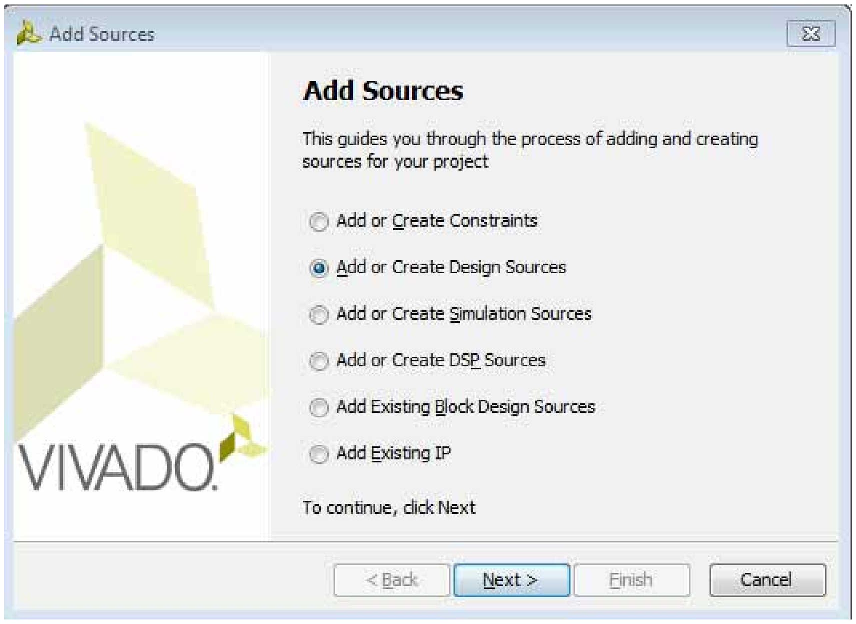
3. From the New Strategy dialog box, specify a **User defined strategy name**, a **Description**, the **Flow**.
4. After you set the options, click **OK**.

   The Save Strategies As dialog box displays **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, in the Flow Navigator, select the **Add Sources** command to open the Add Sources wizard, shown in the following figures.


Add Sources

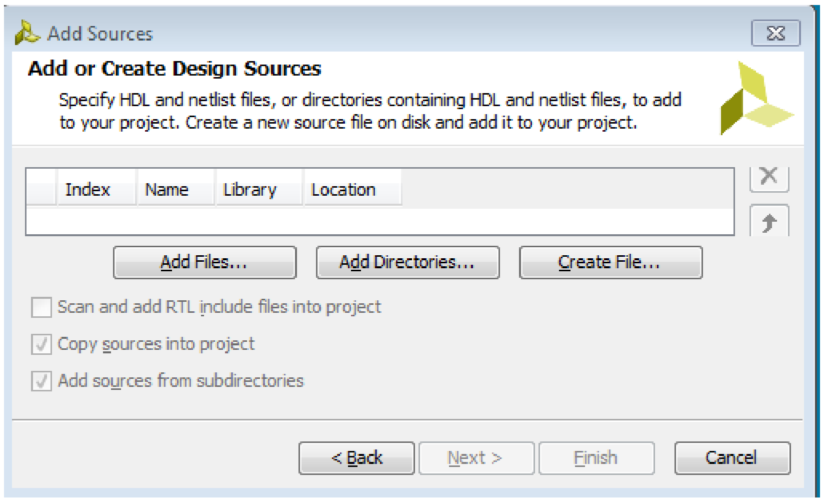
Add Sources

This guides you through the process of adding and creating sources for your project

( ) Add or Create Constraints
(●) Add or Create Design Sources
( ) Add or Create Simulation Sources
( ) Add or Create DSP Sources
( ) Add Existing Block Design Sources
( ) Add Existing IP

To continue, click Next

Figure 1-4: **Add Sources Dialog Box**


Add Sources

Add or Create Design Sources

Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.

Index	Name	Library	Location

[ ] Scan and add RTL include files into project
[✓] Copy sources into project
[✓] Add sources from subdirectories

Figure 1-5: **Add or Create Sources Dialog Box**

Add constraint, RTL, or other project files. See the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 2] for more information about the Add Source wizard.

The Vivado synthesis tool reads the subset of files that can be synthesized in VHDL, Verilog, or SystemVerilog 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, Verilog Language Support
- Chapter 6, Mixed Language Support
- Chapter 7, SystemVerilog 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 the ISE to Vivado Design Suite Migration Guide* (UG911) [Ref 6] *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 sets 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 options provided 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. Set **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.
2. From the Sources window popup menu, drag and drop files in the **Compile Order** window to arrange the compilation order, or use the menu **Move Up** or **Move Down** commands.

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

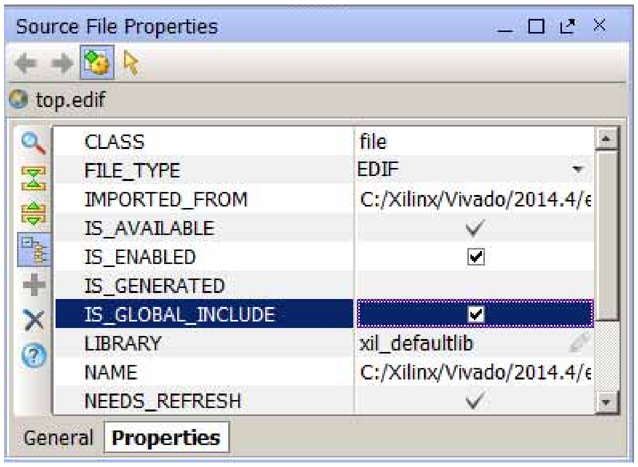
Defining Global Include Files

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 and open the popup menu.
2. Select the **Set Global Include** command, or use the **Global Include** checkbox in the Source File Properties window, as shown in the following figure.



CLASS	file
FILE_TYPE	EDIF
IMPORTED_FROM	C:/Xilinx/Vivado/2014.4/e
IS_AVAILABLE	✓
IS_ENABLED	✓
IS_GENERATED	
IS_GLOBAL_INCLUDE	✓
LIBRARY	xil_defaultlib
NAME	C:/Xilinx/Vivado/2014.4/e
NEEDS_REFRESH	✓

Figure 1-6: **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 the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 2], for information about the Sources window.

Running Synthesis

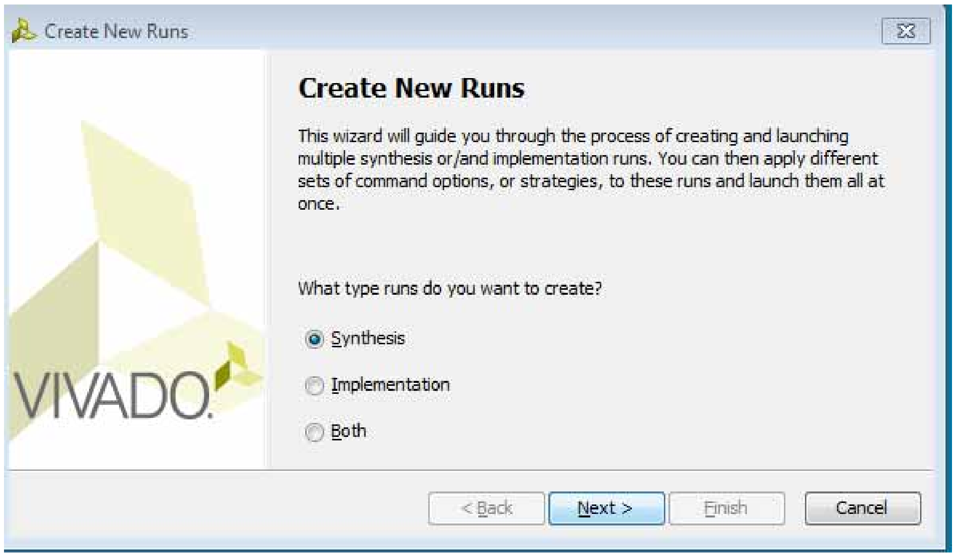
A *run* defines and configures aspects of the design that are used during synthesis. A synthesis run defines the:

- 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** to open the Create New Runs wizard.
2. Select **Synthesis**, and click **Next**.

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



Create New Runs

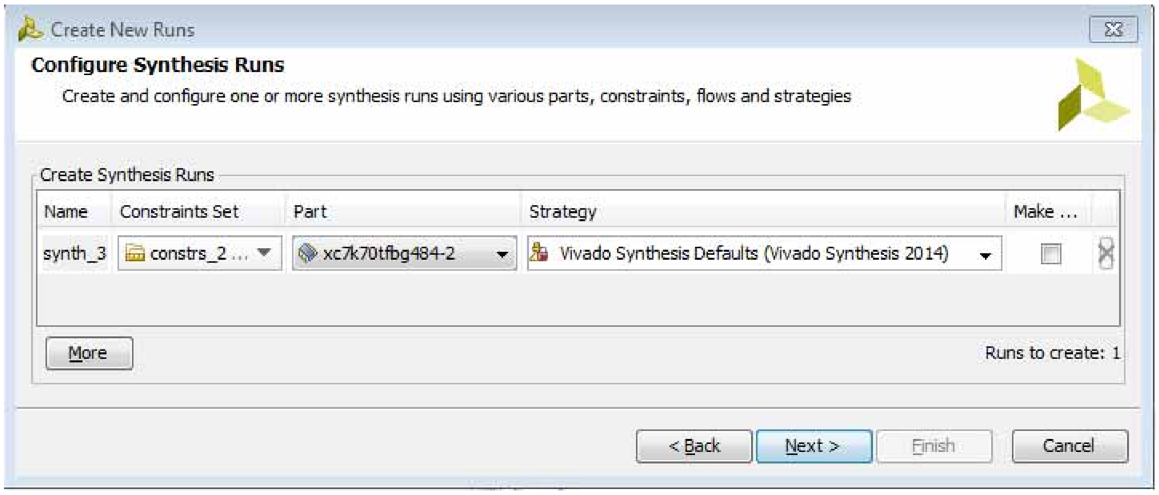
This wizard will guide you through the process of creating and launching multiple synthesis or/and implementation runs. You can then apply different sets of command options, or strategies, to these runs and launch them all at once.

What type runs do you want to create?

[✓ Synthesis] Implementation Both

Figure 1-7: **Create New Runs Dialog Box**

The Configure Synthesis Runs dialog box opens, as shown in Figure 1-8, page 17.



Create New Runs

Configure Synthesis Runs

Create and configure one or more synthesis runs using various parts, constraints, flows and strategies

Create Synthesis Runs

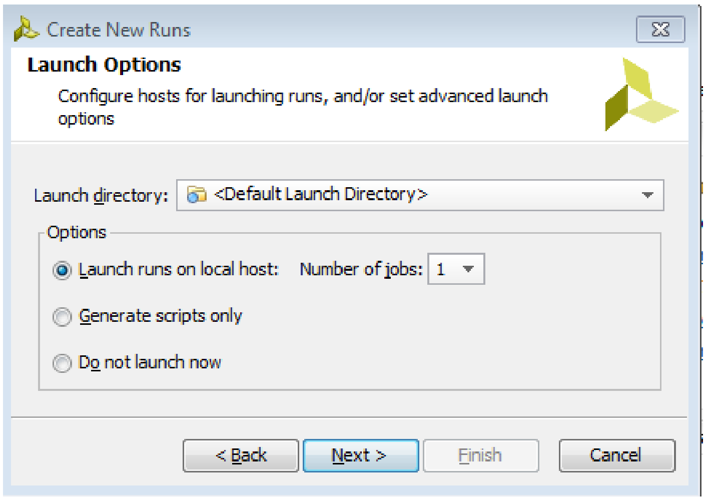
Name	Constraints Set	Part	Strategy	Make ...
synth_3	constrs_2 ...	xc7k70tfbg484-2	Vivado Synthesis Defaults (Vivado Synthesis 2014)	☐

Runs to create: 1

Figure 1-8: **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 Launch Options dialog box opens, as shown in the following figure.



Create New Runs

Launch Options

Configure hosts for launching runs, and/or set advanced launch options

Launch directory: <Default Launch Directory>

Options

[✓ Launch runs on local host:] Number of jobs: 1

Generate scripts only

Do not launch now

Figure 1-9: **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**: Lets you run 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**: Lets you launch the runs on a remote host (Linux only) and configure that host.

See "Appendix A" of 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**: Lets you generate 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.

The Choose Synthesis Strategies dialog box opens.

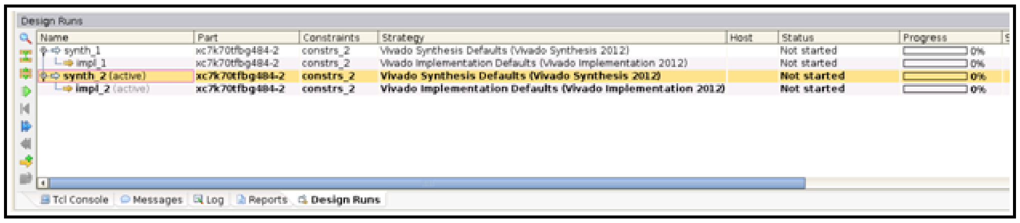
4. Select the **Constraints set, Part**, and **Strategy**, and click **Next**.

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.

For more information about constraints, see the *Vivado Design Suite User Guide: Using Constraints (UG903)* [Ref 8].

To create your own run strategy, see Creating Run Strategies, page 12.

After setting the Create New Runs wizard option and starting a run, you can see the results in the Design Runs window, as shown in the following figure.



Design Runs

Name	Part	Constraints	Strategy	Host	Status	Progress
synth_1	xc7k70tfbg484-2	constrs_2	Vivado Synthesis Defaults (Vivado Synthesis 2012)		Not started	0%
impl_1	xc7k70tfbg484-2	constrs_2	Vivado Implementation Defaults (Vivado Implementation 2012)		Not started	0%
synth_2 (active)	xc7k70tfbg484-2	constrs_2	Vivado Synthesis Defaults (Vivado Synthesis 2012)		Not started	0%
impl_2 (active)	xc7k70tfbg484-2	constrs_2	Vivado Implementation Defaults (Vivado Implementation 2012)		Not started	0%

Figure 1-10: **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 "Appendix A" of 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. 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 Module** option, shown in the following figure.

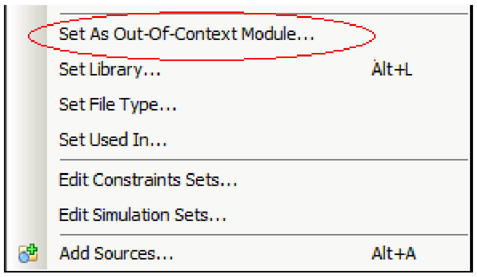
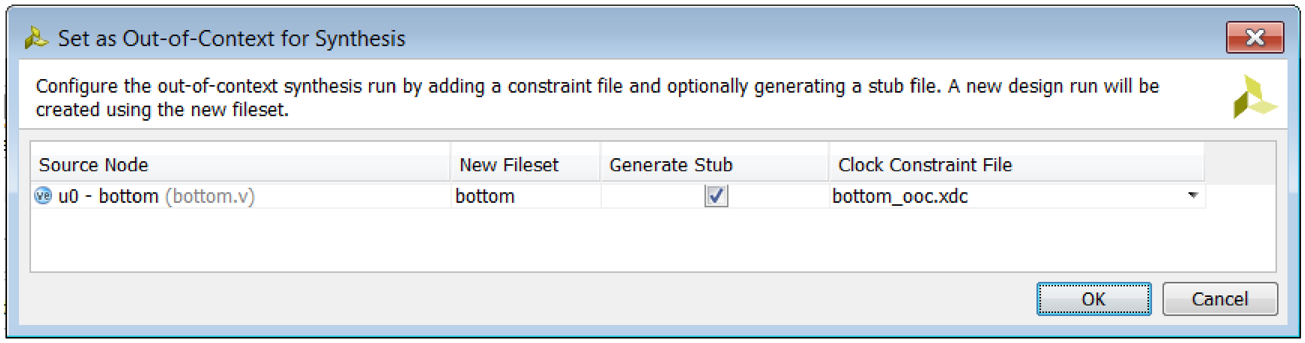


Figure 1-11: **Set As Out-of-Context Option**

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



Set as Out-of-Context for Synthesis

Configure the out-of-context synthesis run by adding a constraint file and optionally generating a stub file. A new design run will be created using the new fileset.

Source Node	New Fileset	Generate Stub	Clock Constraint File
u0 - bottom (bottom.v)	bottom	✓	bottom_ooc.xdc

Figure 1-12: **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:

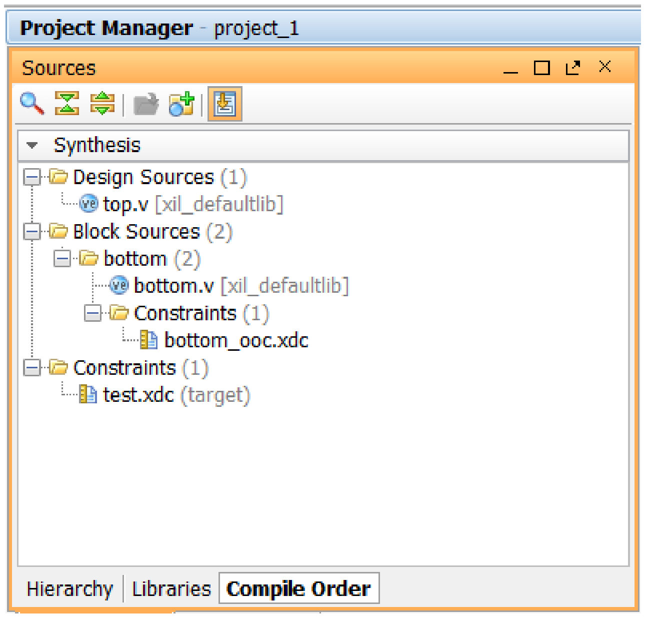
- **Source Node**: Module to which you are applying the OOC.
- **New Fileset**: Lists the Fileset name, which you can edit.
- **Generate Stub**: A checkbox that you can check to have the tool create a stub file.
- **Clock Constraint File**: Choose to have the tool create a new XDC template for you, or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

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.



Project Manager - project_1

Sources

Synthesis

```
Design Sources (1)
    top.v [xil_defaultlib]
Block Sources (2)
    bottom (2)
        bottom.v [xil_defaultlib]
        Constraints (1)
            bottom_ooc.xdc
Constraints (1)
    test.xdc (target)
```

Figure 1-13: **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, page 19. This action sets the lower-level as a top module and runs synthesis on that module without creating I/O buffers.

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! *This flow should not be used 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.*

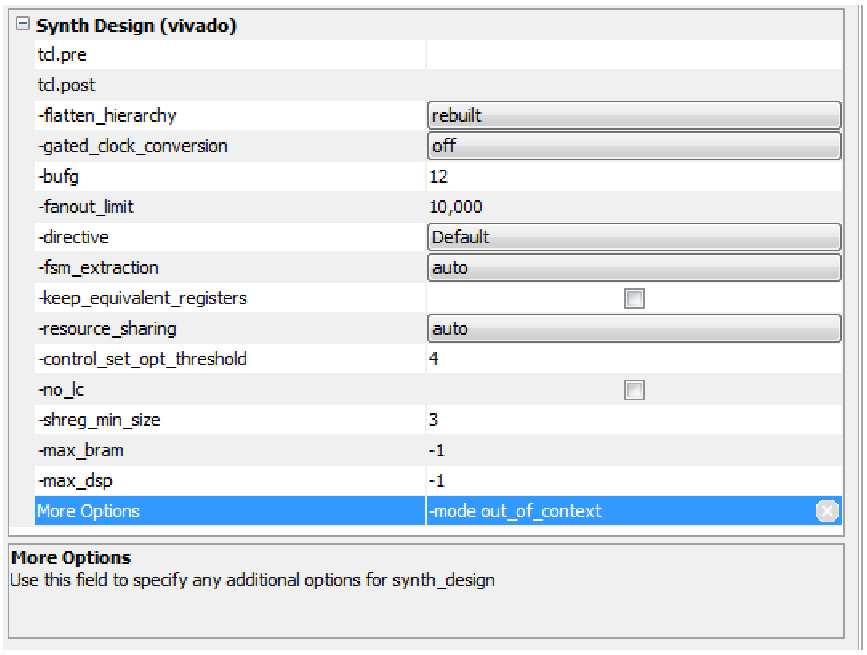
Manually Setting a Bottom-Up Flow and Importing Netlists

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 RTL, and 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 mode as shown in the following figure.



Synth Design (vivado)

tcl.pre	
tcl.post	
-flatten_hierarchy	rebuilt
-gated_clock_conversion	off
-bufg	12
-fanout_limit	10,000
-directive	Default
-fsm_extraction	auto
-keep_equivalent_registers	☐
-resource_sharing	auto
-control_set_opt_threshold	4
-no_lc	☐
-shreg_min_size	3
-max_bram	-1
-max_dsp	-1
More Options	-mode out_of_context

More Options
Use this field to specify any additional options for synth_design

Figure 1-14: **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 Tcl command.

Tcl Command to Create a Lower-Level Netlist

```
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, page 19, this was 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 your design is only from third-party netlists, 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 if you are running 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|.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.

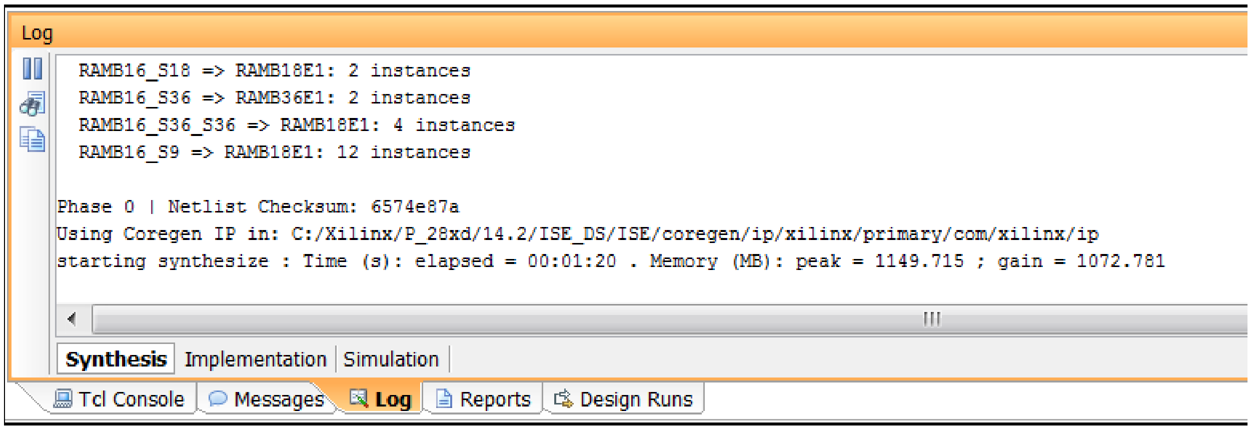
For more information, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 19].

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, shown in the following figure.



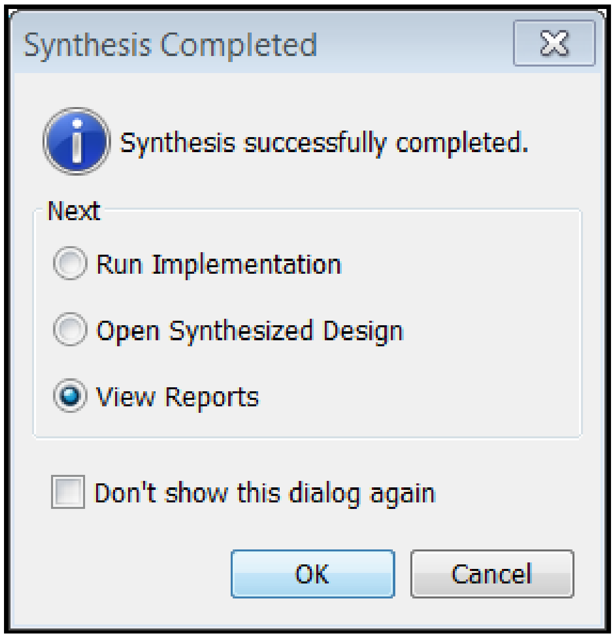
```
  RAMB16_S18 => RAMB18E1: 2 instances
  RAMB16_S36 => RAMB36E1: 2 instances
  RAMB16_S36_S36 => RAMB18E1: 4 instances
  RAMB16_S9 => RAMB18E1: 12 instances

Phase 0 | Netlist Checksum: 6574e87a
Using Coregen IP in: C:/Xilinx/P_28xd/14.2/ISE_DS/ISE/coregen/ip/xilinx/primary/com/xilinx/ip
starting synthesize : Time (s): elapsed = 00:01:20 . Memory (MB): peak = 1149.715 ; gain = 1072.781
```

Figure 1-15: **Log Window**

Following Synthesis

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



Synthesis Completed

Synthesis successfully completed.

Next

- Run Implementation
- Open Synthesized Design
- [✓ View Reports]

Don't show this dialog again

Figure 1-16: **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.

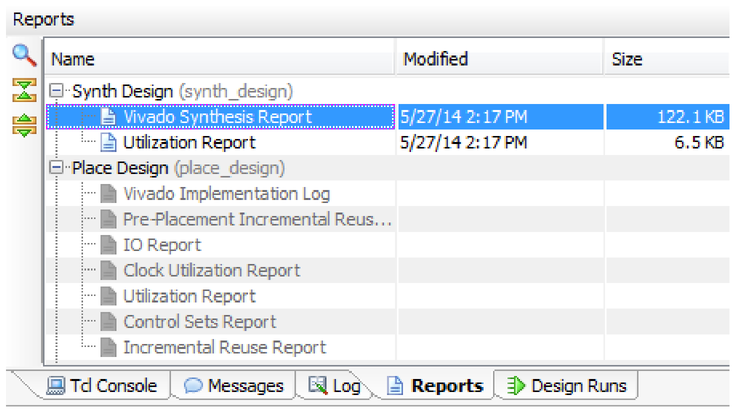
Use the **Don't show this dialog again** checkbox to stop this dialog box display.

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.

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



Reports

Name	Modified	Size
Synth Design (synth_design)		
Vivado Synthesis Report	5/27/14 2:17 PM	122.1 KB
Utilization Report	5/27/14 2:17 PM	6.5 KB
Place Design (place_design)		
Vivado Implementation Log		
Pre-Placement Incremental Reus...		
IO Report		
Clock Utilization Report		
Utilization Report		
Control Sets Report		
Incremental Reuse Report		

Figure 1-17: **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.

For more information, see the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 9].

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.

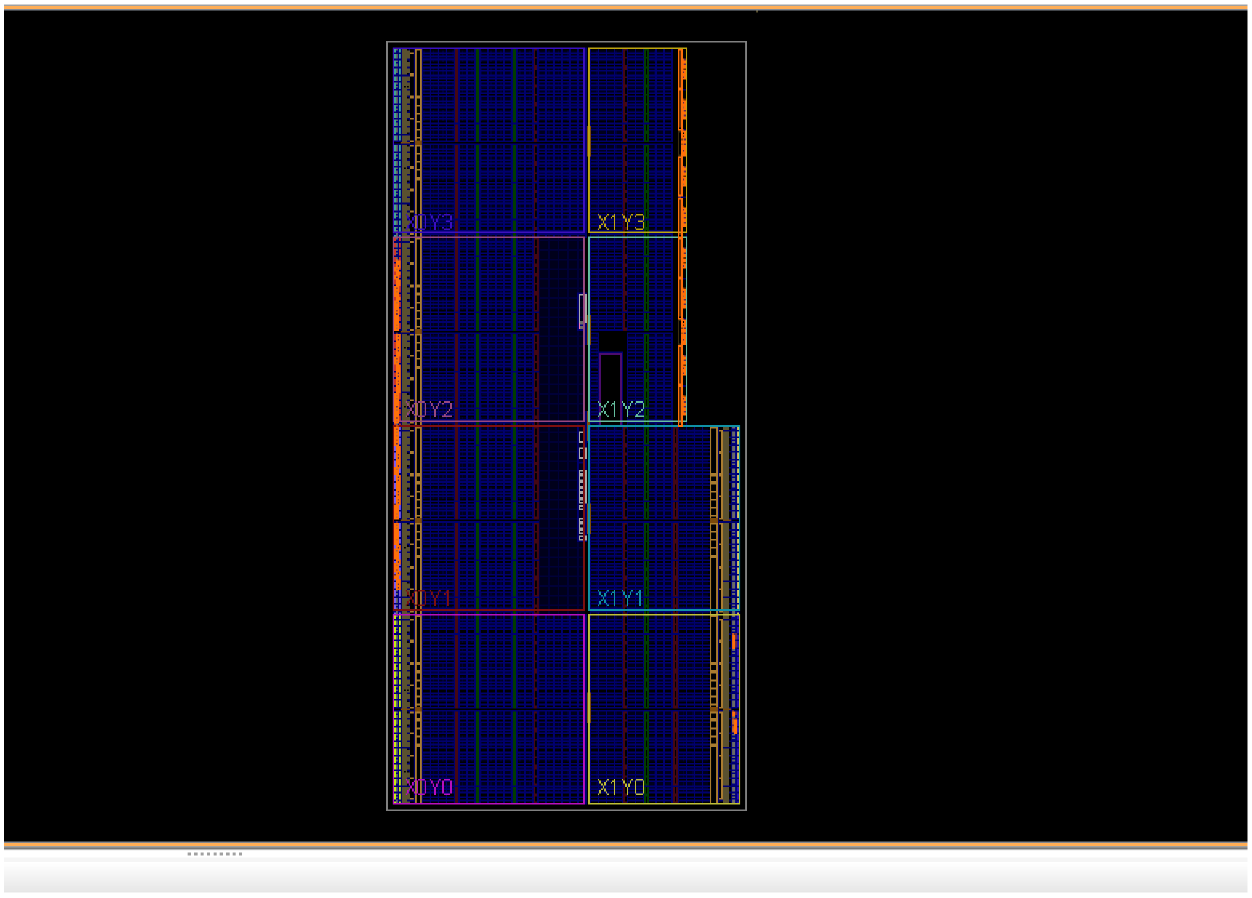
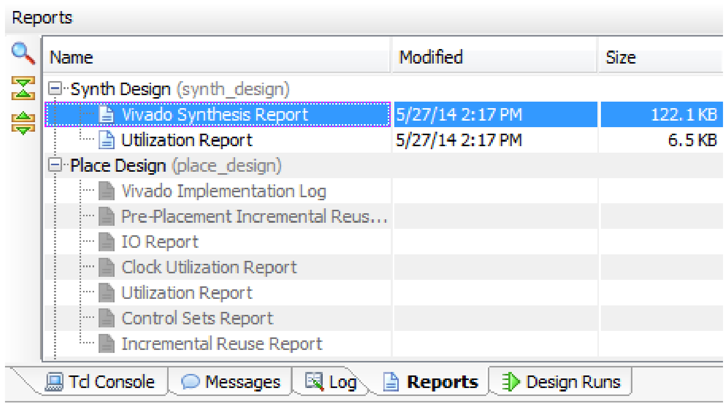


Figure 1-18: **Device Window**

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

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.



Reports

Name	Modified	Size
Synth Design (synth_design)		
Vivado Synthesis Report	5/27/14 2:17 PM	122.1 KB
Utilization Report	5/27/14 2:17 PM	6.5 KB
Place Design (place_design)		
Vivado Implementation Log		
Pre-Placement Incremental Reus...		
IO Report		
Clock Utilization Report		
Utilization Report		
Control Sets Report		
Incremental Reuse Report		

Figure 1-19: **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. Right-click and select **Show Hierarchy**, as shown in the following figure. Also, you can press **F6** to open the Hierarchy window.

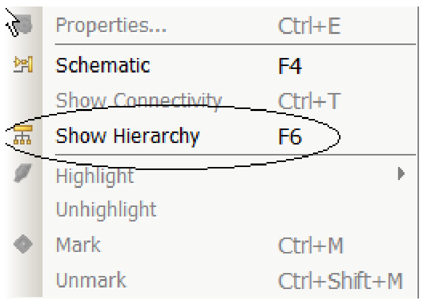


Figure 1-20: **Show Hierarchy Option**

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.

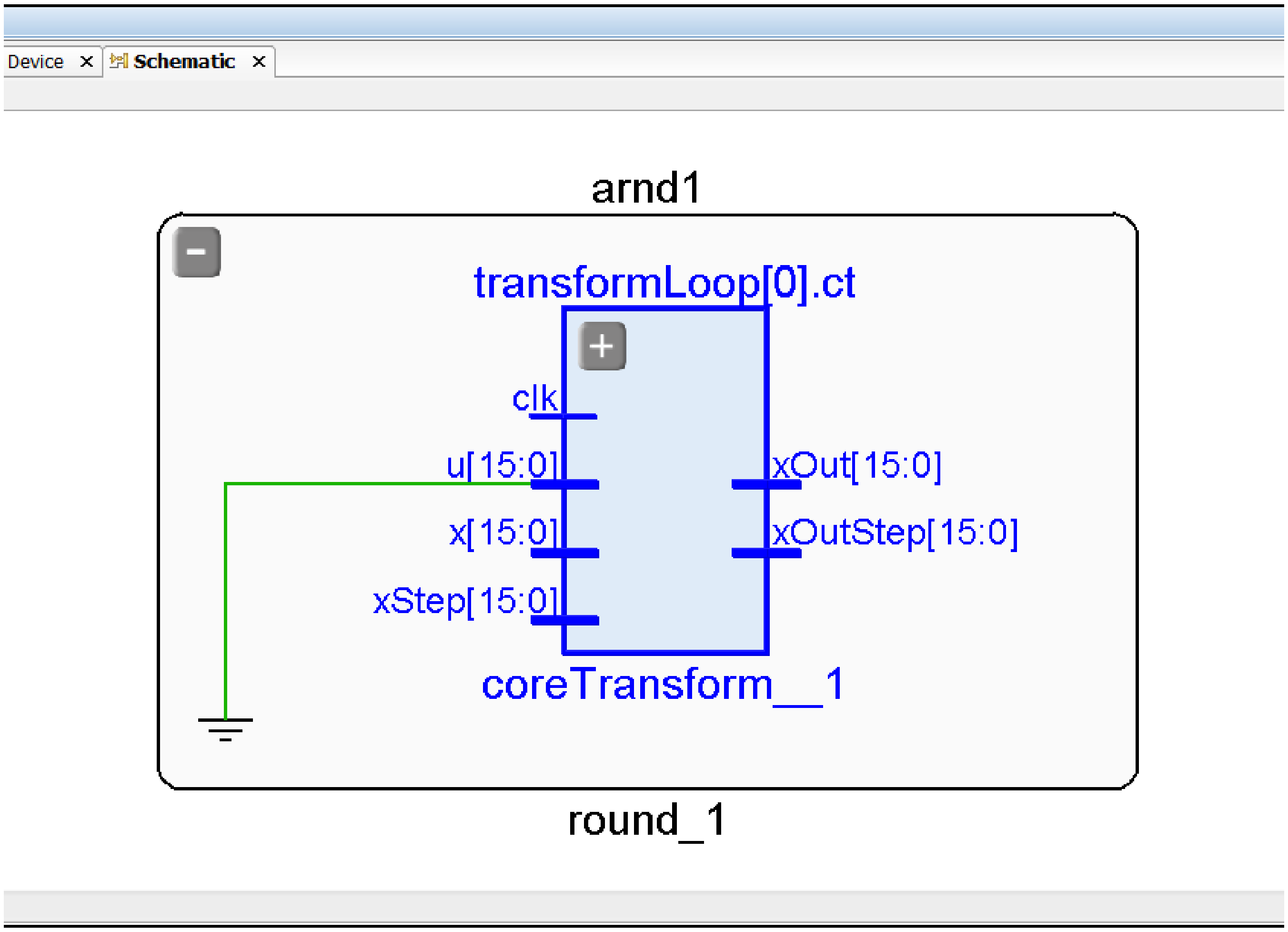


Figure 1-21: **Schematic Window**

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>]
              [-seu_max_util <arg>] [-flatten_hierarchy <arg>]
              [-gated_clock_conversion <arg>] [-directive <arg>] [-rtl]
              [-bufg <arg>] [-no_lc] [-fanout_limit <arg>]
              [-shreg_min_size <arg>] [-mode <arg>] [-fsm_extraction <arg>]
              [-keep_equivalent_registers] [-resource_sharing <arg>]
              [-control_set_opt_threshold <arg>] [-max_bram <arg>]
              [-max_dsp <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,
                                runtimeoptimized
                                Default: default
[-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
                                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_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
      [-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.*

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.

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

Table 1-2 shows the supported Tcl commands for Vivado timing constraints.

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 8]
- *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 9]
- *Vivado Design Suite Tutorial: Using Constraints* (UG945) [Ref 11]

Table 1-2: **Supported Synthesis Tcl Commands**

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

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 *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.

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 132.

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”;
```

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 if a 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 `"onehot"` encode, and that is in a `"0101"` state (which is an illegal for `"onehot"`), 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 window, 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 Verilog Example

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

GATED_CLOCK VHDL Example

```
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 specific ports are needed to be kept, 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 Verilog Example

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

KEEP VHDL Example

```
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 Verilog Example

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 VHDL Example

On Module:

```
attribute keep_hierarchy : string;
attribute keep_hierarchy of beh : architecture 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 Lab tools. 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]
```

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 Verilog Examples

On Signal:

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

MAX_FANOUT VHDL Example

```
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 accepted are:

- block: Instructs the tool to infer RAMB type components.
- distributed: Instructs the tool to infer the LUT RAMs.
- registers: Instructs the tool to infer registers instead of RAMs.

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. This can be set in the RTL or the XDC.

RAM_STYLE Verilog Example

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

RAM_STYLE VHDL Example

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

For more information about RAM coding styles, see RAM HDL Coding Techniques, page 85.

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 Verilog Example

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

ROM_STYLE VHDL Example

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

For information about coding for ROM, see ROM HDL Coding Techniques, page 138.

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 Verilog Example

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

SHREG_EXTRACT VHDL Example

```
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 currently 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, `mults`, `mult-add`, `mult-sub`, `mult-accumulate` type structures go 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 Verilog Example

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

USE_DSP48 VHDL Example

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

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, use the following syntax:

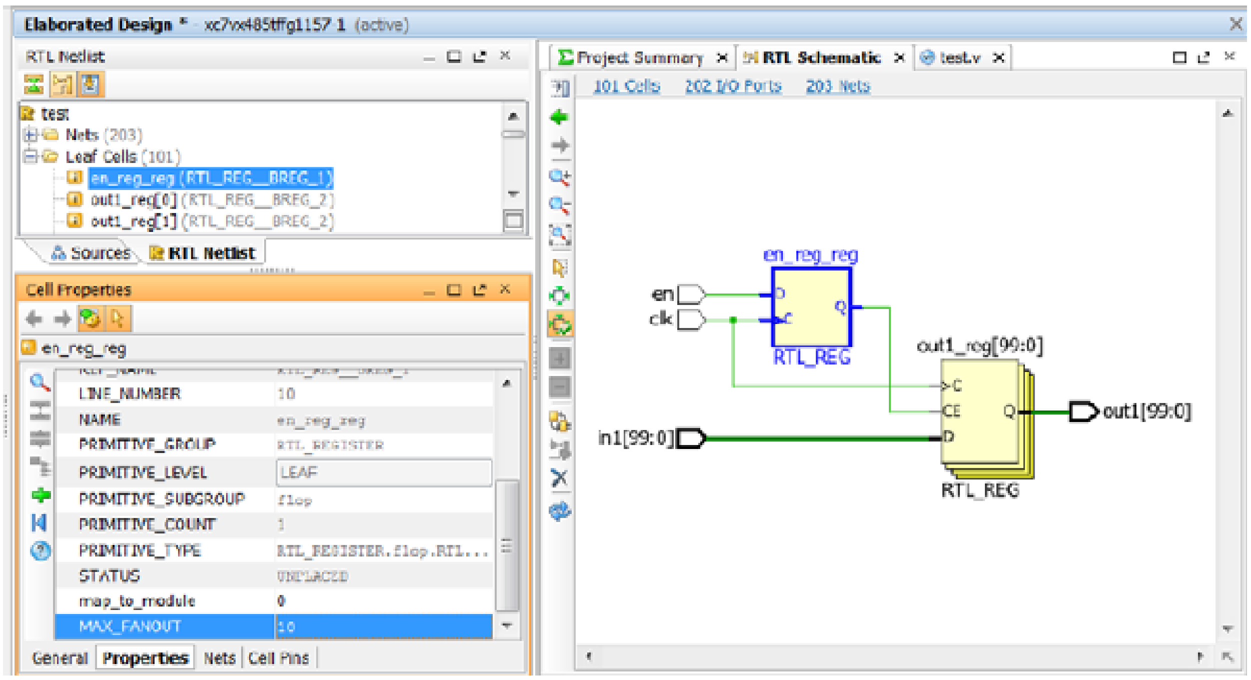
```
set_property <attribute> <value> <target>
```

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 (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.



LINE_NUMBER	10
NAME	en_reg_reg
PRIMITIVE_GROUP	RTL_REGISTER
PRIMITIVE_LEVEL	LEAF
PRIMITIVE_SUBGROUP	flop
PRIMITIVE_COUNT	1
PRIMITIVE_TYPE	RTL_REGISTER.flop.RTL...
STATUS	UNPLACED
map_to_module	0
MAX_FANOUT	10

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: https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

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, Unions, 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:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

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

```
-- 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 top 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 top;
architecture rtl of top is
begin
  process (clk) is
  begin
    if clr = '1' then
      dout <= "00000000";
    elsif rising_edge(clk) then
      if ce = '1' then
        dout <= d_in;
      end if;
    end if;
  end process;
end architecture rtl;
```

Register wiht Rising-Edge Verilog Coding Example

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

module top(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
```

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
-----+----+-----

===================================================================
```

Latches Coding Example

The following subsection provides a VHDL and coding example for latches with positive gates and asynchronous resets.

Latch With Positive Gate and Asynchronous Reset VHDL Coding Example

```
-- 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_2;

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;
```

Tri-States

- Tri-state buffers are usually modeled by:
    - A signal
    - 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:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

Tri-State Implementation

Inferred Tri-state 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`)

Tri-state Reporting Example

Tri-state buffers are inferred and reported during synthesis.

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

Tri-States Coding Examples

The following subsections provide VHDL and Verilog coding examples for tri-state.

Tri-State Description Using Concurrent Assignment (Verilog)

```
// Tristate Description Using Concurrent Assignment
// File: tristates_2.v
//
module v_three_st_2 (T, I, O);
    input  T, I;
    output O;

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

endmodule
```

Tri-State 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 example 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 example;

architecture archi of example 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;
```

Tri-State Description Using Combinatorial Process VHDL Coding Example

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

entity three_st_1 is
    port(T : in  std_logic;
         I : in  std_logic;
         O : out std_logic);
end three_st_1;

architecture archi of three_st_1 is
begin

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

end archi;
```

Tri-State Description Using Combinatorial Always Block Verilog Coding Example

```
// Tristate Description Using Combinatorial Always Block
// File: tristates_1.v
//
module v_three_st_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, page 68, the length of the propagation chain varies dynamically during circuit operation.

Download the coding example files from:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

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 VHDL Coding Example One

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 VHDL Coding Example Two

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;
```

32-Bit Shift Register Verilog Coding Example One

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 v_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
```

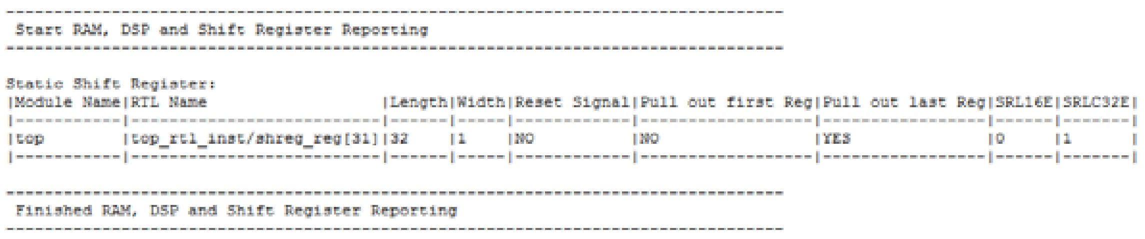
8-Bit Shift Register Verilog Coding Example Two

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

module v_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.

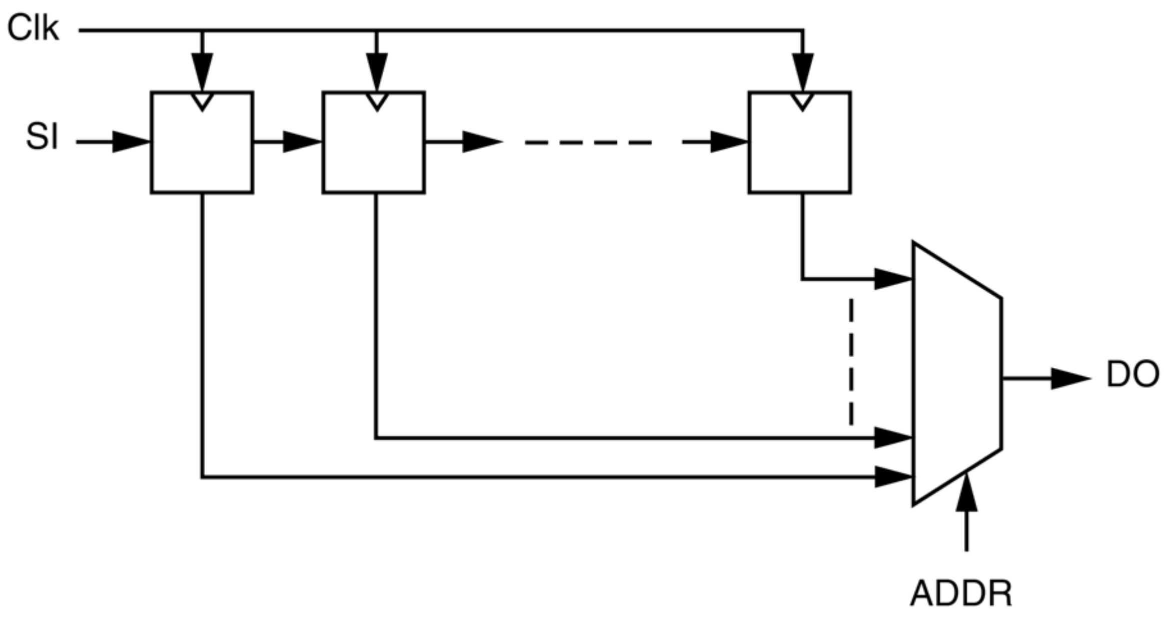


Figure 3-1: **Dynamic Shift Registers Diagram**

Dynamic Shift Registers

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

Download the coding example files from:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

32-Bit Dynamic Shift Registers VHDL Coding Example

```
-- 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 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;

architecture rtl of dynamic_shift_register 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;
```

32-Bit Dynamic Shift Registers Verilog Coding Example

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

module dynamic_shift_register (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
```

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. For more information on the `KEEP` attribute, see KEEP, page 44.

Multipliers Coding Examples

Unsigned 16x16-Bit Multiplier VHDL Coding Example

```
-- Unsigned 16x16-bit Multiplier
-- File: multipliers1.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity multipliers1 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 multipliers1;

architecture beh of multipliers1 is begin
  RES <= A * B;
end beh;
```

Unsigned 16x24-Bit Multiplier Verilog Coding Example

```
// Unsigned 16x24-bit Multiplier
//1 latency stage on operands
//3 latency stage after the multiplication
// File: multipliers2.v
//
module multipliers2 (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
```

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, page 44.

Multiply-Add and Multiply-Accumulate Coding Examples

The following subsections provide VHDL and Verilog coding examples for multiply add and multiply accumulate functions.

Download the coding example files from:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

Fully Pipelined Streaming Multiply Accumulate VHDL Coding Example One

```
-- Signed 40-bit streaming accumulator with 16-bit inputs
-- File: multipliers3.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity macc is
  generic (SIZEIN  : natural := 16;
           SIZEOUT : natural := 40);
  port (clk, ce, sload : in  std_logic;
        a, b           : in  signed (SIZEIN-1 downto 0);
        accum_out      : out signed (SIZEOUT-1 downto 0));
end entity;

architecture rtl of macc is
  -- Declare registers for intermediate values
  signal a_reg, b_reg          : signed (SIZEIN-1 downto 0);
  signal sload_reg             : std_logic;
  signal mult_reg              : signed (2*SIZEIN-1 downto 0);
  signal adder_out, old_result : signed (SIZEOUT-1 downto 0);

begin

  process (adder_out, sload_reg)
  begin
    if sload_reg = '1' then
      old_result <= (others => '0');
    else
      -- 'sload' is now active (=low) and opens the accumulation loop.
      -- The accumulator takes the next multiplier output in
      -- the same cycle.
      old_result <= adder_out;
    end if;
  end process;

  process (clk)
  begin
    if rising_edge(clk) then
      if ce = '1' then
        a_reg     <= a;
        b_reg     <= b;
        mult_reg  <= a_reg * b_reg;
        sload_reg <= sload;
```

```
          -- Store accumulation result into a register
          adder_out <= old_result + mult_reg;
        end if;
      end if;
    end process;

    -- Output accumulation result
    accum_out <= adder_out;

  end rtl;
```

Fully Pipelined Streaming Multiply Accumulate VHDL Coding Example Two

```
-- Unsigned 16x16-bit Multiplier
-- File: multipliers1.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity multipliers1 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 multipliers1;

architecture beh of multipliers1 is begin
  RES <= A * B;
end beh;
```

Complex Multiplier Example - VHDL

Fully pipelined complex multiplier using three DSP48 blocks.

Download the coding example files from:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

```
-- Complex Multiplier (pr+i.pi) = (ar+i.ai)*(br+i.bi)
-- File: complex_multipliers1.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_ddd : 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-Adder in the DSP Block

When coding for inference and targeting the DSP block, it's recommended to use signed arithmetic and it's 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 Configured as Subtractor followed by a Multiplier (Verilog)

```
module presubmult  # (parameter SIZEIN = 16)
    (input clk, ce, rst,
     input  signed [SIZEIN-1:0] a, b, c,
     output signed [2*SIZEIN:0] presubmult_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] m_reg, p_reg;

    always @(posedge clk)
      if (rst)
        begin
           a_reg   <= 0;
           b_reg   <= 0;
```

```
          c_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;
            add_reg <= a - b;
            m_reg   <= add_reg * c_reg;
            p_reg   <= m_reg;
         end

   // Output accumulation result
   assign presubmult_out = p_reg;

endmodule // presubmult
```

Pre-Adder Configured as Subtractor Followed by a Multiplier (VHDL)

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

entity presubmult is
  generic (AWIDTH : natural := 12;
           BWIDTH : natural := 16;
           CWIDTH : natural := 17);
  port (clk  : 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);
        pout : out std_logic_vector(BWIDTH+CWIDTH downto 0));
end presubmult;

architecture rtl of presubmult 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 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);
        add  <= resize(a, BWIDTH+1) - resize(b, BWIDTH+1);
        mult <= add * c;
        p    <= mult;
      end if;
    end process;

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

  end rtl;
```

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

Filename: dynperaddmultadd.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)
       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

     // Output accumulation result
     assign dynpreaddmultadd_out = p_reg;

  endmodule // dynpreaddmultadd
```

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

Filename: `dynperaddmultadd.vhd`

```
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:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

The following is an example of the square of a difference; this can be used to replace advantageously calculations on absolute values on differences. It fits into a single DSP block and runs at full speed.

Square of a Difference (Verilog)

```
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)
      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

      // Output result
      assign square_out = p_reg;

   endmodule // squarediffmult
```

Square of a difference (VHDL)

```
   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:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

8-Tap Even Symmetric Systolic FIR (Verilog - top -)

```
// sfir_even_symmetric_systolic_top.v

module 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 // top
```

8-Tap Even Symmetric Systolic FIR (Verilog - shifter -)

```
//sfir_shifter.v
(* 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_shifter
```

8-Tap Even Symmetric Systolic FIR (Verilog - Processing Element -)

```
//sfir_even_symmetric_systolic_element.v
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
   // sfir_even_symmetric_systolic_element
```

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:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

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 you have 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, page 49.

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.

RAM HDL Coding Guidelines

Download the coding example files from:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

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.

Single-Port RAM with Asynchronous Read VHDL Coding Example

```
-- Single-Port RAM with Asynchronous Read (Distributed RAM)
-- File: rams_04.vhd

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

 entity rams_04 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_04;

 architecture syn of rams_04 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;
```

Dual-Port RAM with Asynchronous Read Verilog Coding Example

```
// Dual-Port RAM with Asynchronous Read (Distributed RAM)
// File: rams_09.v

module v_rams_09 (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 Block RAMs

Single-Port Block RAM Read-First Mode VHDL Coding Example

```
-- Single-Port Block RAM Read-First Mode
-- rams_01.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_01 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_01;

architecture syn of rams_01 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 Coding Example

```
-- 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_02a 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_02a;

architecture syn of rams_02 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 Write-First Mode Verilog Coding Example

```
// Single-Port Block RAM Write-First Mode (recommended template)
// File: rams_02a.v
//

module v_rams_02a (clk, we, en, addr, di, do);

input clk;
input we;
input en;
input [9:0] addr;
input [15:0] di;
output [15:0] do;
reg[15:0] RAM [1023:0];
reg[15:0] do;

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

Single-Port Block RAM No-Change Mode VHDL Coding Example

```
-- Single-Port Block RAM No-Change Mode
-- File: rams_03.vhd
--

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

entity rams_03 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_03;

architecture syn of rams_03 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;
        else
```

```
                do <= RAM( conv_integer(addr));
            end if;
        end if;
    end if;
    end process;

    end syn;
```

Single-Port Block RAM No-Change Mode Verilog Coding Example

```
// Single-Port Block RAM No-Change Mode
// File: rams_03.v

module v_rams_03 (clk, we, en, addr, di, do);

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

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

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

Simple Dual-Port Block RAM Examples

Simple Dual-Port Block RAM with Single Clock VHDL Coding Example

```
-- 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 Single Clock Verilog Coding Example

```
// 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 Dual Clocks VHDL Coding Example

```
-- 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;
```

Simple Dual-Port Block RAM with Dual Clocks Verilog Coding Example

```
// 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
```

True Dual-Port Block RAM Examples

Dual-Port Block RAM with Two Write Ports VHDL Coding Example

```
-- Dual-Port Block RAM with Two Write Ports
-- Correct Modelization with a Shared Variable
-- File: rams_16b.vhd

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

entity rams_16b 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_16b;

architecture syn of rams_16b 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;
```

Dual-Port Block RAM with Two Write Ports Verilog Coding Example

```
// Dual-Port Block RAM with Two Write Ports
// File: rams_16.v

module v_rams_16 (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
```

Block RAM with Resettable Data Output VHDL Coding Example

```
-- Block RAM with Resettable Data Output
-- File: rams_18.vhd

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

entity rams_18 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_18;

architecture syn of rams_18 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;
```

Block RAM with Resettable Data Output Verilog Coding Example

```
// Block RAM with Resettable Data Output
// File: rams_18.v

module v_rams_18 (clk, en, we, rst, addr, di, do);

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

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

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

```
                do <= ram[addr];
        end
end

endmodule
```

Block RAM with Optional Output Registers VHDL Coding Example

```
-- Block RAM with Optional Output Registers
-- File: rams_19.vhd
library IEEE;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rams_19 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_19;

architecture beh of rams_19 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;
```

Block RAM with Optional Output Registers Verilog Coding Example

```
// Block RAM with Optional Output Registers
// File: rams_19.v

module v_rams_19 (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
```

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

Single-Port Block RAM with Byte Write Enable VHDL Coding Example

This coding example uses generics and a `for-loop` construct for a compact and easily changeable configuration of the desired number and width of write columns.

```
-- Single-Port BRAM with Byte Write Enable
--  2x8-bit write
--  Read-First mode
--  Single-process description
--  Compact description of the write with a for-loop statement
--  Column width and number of columns easily configurable
-- bytewrite_ram_1b.vhd
 library ieee;
 use ieee.std_logic_1164.all;
 use ieee.std_logic_unsigned.all;
 entity bytewrite_ram_1b is generic (
 SIZE: integer := 1024;
 ADDR_WIDTH : integer := 10;
 COL_WIDTH : integer := 8;
 NB_COL: integer := 4);

 port (
 clk : in std_logic;
 we: in std_logic_vector(NB_COL-1 downto 0);
 addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
 di: in std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
 do: out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0));

 end bytewrite_ram_1b;
 architecture behavioral of bytewrite_ram_1b is type ram_type is array
(SIZE-1 downto 0)
 of std_logic_vector (NB_COL*COL_WIDTH-1 downto 0);
 signal RAM : ram_type := (others => (others => '0'));

 begin

 process (clk)
 begin
 if rising_edge(clk) then
     do <= RAM(conv_integer(addr));
     for i in 0 to NB_COL-1 loop
         if we(i) = '1' then
             RAM(conv_integer(addr))((i+1)*COL_WIDTH-1 downto i*COL_WIDTH)
<= di((i+1)*COL_WIDTH-1 downto i*COL_WIDTH);
         end if;
     end loop;
 end if;
 end process;

 end behavioral;
```

Single-Port Block RAM with Byte Write Enable Verilog Coding Example

This coding example uses parameters and a generate-for construct.

```
// Single-Port BRAM with Byte-wide Write Enable
//4x9-bit write
//Read-First mode
//Single-process description
//Compact description of the write with a generate-for
//   statement
//Column width and number of columns easily configurable
//
// bytewrite_ram_1b.v
```

```
//

module v_bytewrite_ram_1b (clk, we, addr, di, do);

parameter SIZE = 1024;
parameter ADDR_WIDTH = 10;
parameter COL_WIDTH = 8;
parameter NB_COL = 4;

inputclk;
input[NB_COL-1:0]we;
input[ADDR_WIDTH-1:0]addr;
input[NB_COL*COL_WIDTH-1:0] di;
output reg [NB_COL*COL_WIDTH-1:0] do;

reg[NB_COL*COL_WIDTH-1:0] RAM [SIZE-1:0];

always @(posedge clk)
begin
    do <= RAM[addr];
end

generate genvar i;
for (i = 0; i < NB_COL; i = i+1)
begin
always @(posedge clk)
begin
    if (we[i])
        RAM[addr][(i+1)*COL_WIDTH-1:i*COL_WIDTH] <=
di[(i+1)*COL_WIDTH-1:i*COL_WIDTH];
    end
end
endgenerate

endmodule
```

True-Dual-Port BRAM with Byte Write Enable Examples

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 BWWE_RAMB_RF_TDP is
```

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

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

end BWWE_RAMB_RF_TDP;

architecture byte_wr_ram_rf of BWWE_RAMB_RF_TDP 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—READ_FIRST Mode Verilog

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

module v_bytewrite_tdp_ram_readfirst
 #(
//---------------------------------------------------------------------
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 // v_bytewrite_tdp_ram_readfirst
```

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

 port (
   clka: instd_logic;
   ena : instd_logic;
   wea : instd_logic_vector(NB_COL-1 downto 0);
   addra : instd_logic_vector(ADDR_WIDTH-1 downto 0);
   dia : instd_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
   doa : out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
```

```
    clkb: instd_logic;
    enb : instd_logic;
    web : instd_logic_vector(NB_COL-1 downto 0);
    addrb : instd_logic_vector(ADDR_WIDTH-1 downto 0);
    dib : instd_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
    dob : out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0)
   );

end BWWE_RAMB_WF_TDP;

architecture byte_wr_ram_wf of BWWE_RAMB_WF_TDP 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 Write Enable—WRITE_FIRST Mode Verilog

```
// True-Dual-Port BRAM with Byte-wide Write Enable
//      Write-First mode
// File: HDL_Coding_Techniques/rams/bytewrite_tdp_ram_wf.v
//
// ByteWide Write Enable, - WRITE_FIRST mode template - Vivado recomended
module v_bytewrite_tdp_ram_writefirst
  #(

//--------------------------------------------------------------------------
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];
                  doutA[i*COL_WIDTH +: COL_WIDTH]  <= dinA[i*COL_WIDTH +:
COL_WIDTH] ;
               end else begin
                  doutA[i*COL_WIDTH +: COL_WIDTH]  <=
ram_block[addrA][i*COL_WIDTH +: COL_WIDTH] ;
               end
            end
         end
      end
   endgenerate

   // 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];
                  doutB[i*COL_WIDTH +: COL_WIDTH]  <= dinB[i*COL_WIDTH +:
COL_WIDTH] ;
               end else begin
                  doutB[i*COL_WIDTH +: COL_WIDTH]  <=
ram_block[addrB][i*COL_WIDTH +: COL_WIDTH] ;
               end
            end
         end
      end
   endgenerate

endmodule // v_bytewrite_tdp_ram_writefirst
```

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

 port (
   clka: instd_logic;
   ena : instd_logic;
   wea : instd_logic_vector(NB_COL-1 downto 0);
   addra : instd_logic_vector(ADDR_WIDTH-1 downto 0);
```

```
    dia : instd_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
    doa : out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
    clkb: instd_logic;
    enb : instd_logic;
    web : instd_logic_vector(NB_COL-1 downto 0);
    addrb : instd_logic_vector(ADDR_WIDTH-1 downto 0);
    dib : instd_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
    dob : out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0)
   );

end BWWE_RAMB_NC_TDP;

architecture byte_wr_ram_nc of BWWE_RAMB_NC_TDP 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;
```

Byte 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 v_bytewrite_tdp_ram_nochange
  #(
    //---------------------------------------------------------------
    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 // v_bytewrite_tdp_ram_nochange
```

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 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;

architecture behavioral of asym_ram_sdp 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 (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- VHDL

```
-- Asymmetric port RAM
-- Write Wider than Read
asym_ram_sdp_write_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 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;
    weB    : in  std_logic;
    addrA  : in  std_logic_vector(ADDRWIDTHA-1 downto 0);
    addrB  : in  std_logic_vector(ADDRWIDTHB-1 downto 0);
    diB    : in  std_logic_vector(WIDTHB-1 downto 0);
    doA    : out std_logic_vector(WIDTHA-1 downto 0)
    );

end asym_ram_sdp;

architecture behavioral of asym_ram_sdp 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

  -- read process
    process (clkA)
    begin
      if rising_edge(clkA) then
        if enA = '1' then
          readA <= my_ram(conv_integer(addrA));
        end if;
        regA <= readA;
      end if;
    end process;

  -- Write 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;
        end if;
        end loop;
        regB <= readB;
      end if;
    end process;

    doA <= regA;

  end behavioral;
```

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 (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
```

True Dual-Port Asymmetric Ram Read 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 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;

architecture behavioral of asym_ram_tdp 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;
```

True Dual-Port Asymmetric Ram Read First Verilog

```
// Asymetric RAM - TDP
// READ_FIRST MODE.
asym_ram_tdp_read_first.v


module asym_ram_tdp (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 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 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;

architecture behavioral of asym_ram_tdp 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;
```

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 (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
```

RAM Initial 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 Examples

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:

```
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
```

VHDL Coding 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");
```

Verilog Coding 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
```

Initializing Block RAM VHDL Coding Example

```
-- Initializing Block RAM (Single-Port Block RAM)
-- File: rams_20a.vhd
 library ieee;
 use ieee.std_logic_1164.all;
 use ieee.std_logic_unsigned.all;

 entity rams_20a 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_20a;

 architecture syn of rams_20a 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 Verilog Coding Example

```
// Initializing Block RAM (Single-Port Block RAM)
// File: rams_20a.v
module v_rams_20a (clk, we, addr, di, do);
input clk;
input we;
input [5:0] addr;
input [19:0] di;
output [19:0] do;

reg [19:0] ram [63:0];
reg [19:0] do;

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;
    do <= ram[addr];
end

endmodule
```

Initializing Block RAM From an External Data File VHDL Coding Example

```
-- Initializing Block RAM from external data file
-- File: rams_20c.vhd

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

 entity rams_20c 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_20c;

 architecture syn of rams_20c 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_20c.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;
```

Initializing Block RAM From an External Data File Verilog Coding Example

```
// Initializing Block RAM from external data file
// Binary data
// File: rams_20c.v

module v_rams_20c (clk, we, addr, din, dout);
input clk;
input we;
input [5:0] addr;
input [19:0] din;
output [19:0] dout;

reg [19:0] ram [0:63];
reg [19:0] dout;

initial begin
$readmemh("v_rams_20c.data",ram);
end

always @(posedge clk)
begin
    if (we)
        ram[addr] <= din;
    dout <= ram[addr];
end endmodule
```

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:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

BLACK_BOX Verilog Example

```
// Black Box
// black_box_1.v
//
module black_box1 (in1, in2, dout);
input in1, in2;
output dout;
endmodule

module test_black_box (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 Example

```
-- Black Box
-- black_box_1.vhd
library ieee;
use ieee.std_logic_1164.all;

entity test_black_box is
port(DI_1, DI_2 : in std_logic;
DOUT : out std_logic);
end test_black_box;
architecture rtl of test_black_box 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.

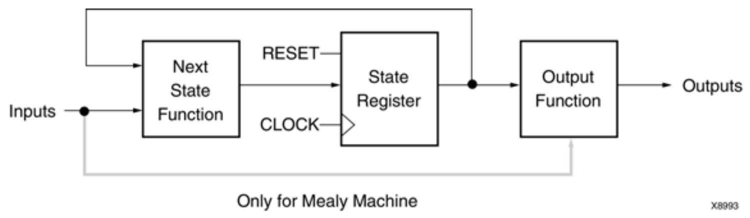


Figure 3-2: **FSM Representation Incorporating Mealy and Moore Machines Diagram**

The following diagram shows an FSM diagram with three processes.

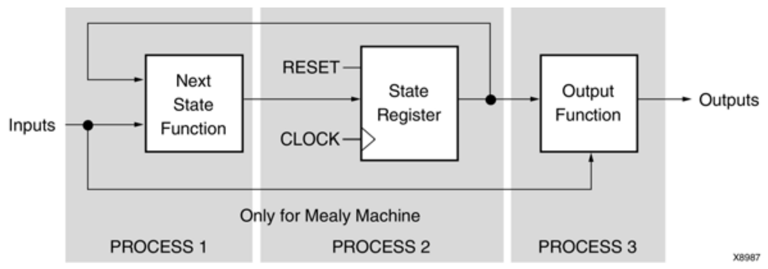


Figure 3-3: **FSM With Three Processes Diagram**

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 Verilog Example

```
// State Machine with single sequential block
//fsm_1.v
module fsm_test(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 VHDL Example

State Machine with Single Sequential Block - VHDL example

```
-- State Machine with single sequential block
-- File: fsm_1.vhd
library IEEE;
use IEEE.std_logic_1164.all;

entity fsm_test is
port ( clk, reset, flag : IN std_logic;
sm_out: OUT std_logic);
end entity;

architecture behavioral of fsm_test 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, page 50 for more information.

ROM Coding Examples

Description of a ROM with a VHDL Constant Coding Example

```
-- 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;
```

ROM Using Block RAM Resources Verilog Coding Example

```
// ROMs Using Block RAM Resources.
// File: rams_21a.v
//
module v_rams_21a (clk, en, addr, do);
inputclk;
inputen;
input[5:0] addr;
output  [19:0] do;

(*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 do = data;

endmodule
```

Dual-Port ROM VHDL Coding Example

```
-- 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).

See Xilinx Answer 62005 for documentation on VHDL 2008 support.

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, page 170.

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. The `type MSB` is a range of 8 to 15 bits. 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, Xilinx recommends that you 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
- 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) &
SRA (Shift Right Arithmetic)	sig1 <= A(4 downto 0) sra 2	sig1 <= <= A(4) & A(4) & A(4
ROL (Rotate Left)	sig1 <= A(4 downto 0) rol 2	sig1 <= A(2 downto 0) & A(4
ROR (Rotate Right)	A(4 downto 0) ror 2	sig1 <= A(1 downto 0) & A(4

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 ports, 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

```
library 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.

VHDL Component Instantiation Coding Example

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 top 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 top;

    architecture ARCHI of top 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 Coding Example

```
    --
    -- Recursive component instantiation
    --
    instantiation_recursive.vhd
    --
    library ieee;
    use ieee.std_logic_1164.all;
    library unisim;
    use unisim.vcomponents.all;

    entity single_stage is
        generic (
            sh_st: integer:=4);
        port (
            CLK : in std_logic;
      DI : in std_logic;
```

```
  DO : out std_logic );
end entity single_stage;

architecture recursive of single_stage is
    component single_stage
        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: single_stage
            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 `GENERIC`s 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 VHDL GENERIC Parameters Coding Example, page 154.

Declaring Generics

You can declare generic parameters in the entity declaration part. Supported generics types are:

- `integer`
- `boolean`
- `string`
- `real`

VHDL GENERIC Parameters Coding 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 top 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 top;

architecture bhv of top 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
- Selected (`with-select-when`)
- Conditional (`when-else`)

Simple Signal Assignment Coding Example

```
T <= A and B;
```

Concurrent Selection Assignment Coding Example

```
-- 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;
```

Concurrent Conditional Assignment (`when-else`) Coding Example

```
--
-- A concurrent conditional assignment (when-else)
--
-- File: concurrent_conditional_assignment.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity concurrent_conditional_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_conditional_assignment;

architecture bhv of concurrent_conditional_assignment is
begin
    T <= a when sel = "00" else
         b when sel = "01" else
         c when sel = "10" else
         d;
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 Coding Example

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 Coding Example

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 Coding 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 Coding Example

```
    -- 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 Coding 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 Coding 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 supports `for-loop` statements for constant bounds and stops the test condition using any of the following operators:

- `<`
- `<=`
- `>`
- `>=`

The next step computations within one of the following specifications are one of the following:

- `var` = var + step
- `var` = var - step

  Where: `var` is the `loop` variable and `step` is a constant value that can be `next` and `exit` statements.

`for-loop` Coding Example

```
--
-- For-loop example
--
for_loop.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity countzeros is
    port (
        a : in std_logic_vector (7 downto 0);
        Count : out std_logic_vector (2 downto 0) );
end countzeros;

architecture behavior of countzeros is
   begin
    process (a)
    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.

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.

Sequential Process With a Sensitivity List Syntax

```
process (<sensitivity list>)
begin
<asynchronous part>
<clock event>
<synchronous part>
end;
```

Clock Event Statements

Describe the clock event statement as:

```
rising edge clock
If clk’event and clk = ‘1’ then
falling edge clock
If clk’event and clk = ‘0’ then
```

For greater clarity, use the VHDL’93 IEEE standard `rising_edge` and `falling_edge` functions.

```
rising edge clock
if clk'event and clk='1' then

falling edge clock
if clk'event and clk='0' 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 Coding 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 Coding 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. For example, loading initial values from an external data file:

- Cannot depend on earlier initial values.
- Can be a parameter value propagated to a register.

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";
```

Operationally Initializing Sequential Elements

To initialize sequential elements operationally, describe the set/reset values and the local control logic, and assign a value to a register when the register reset line goes to the appropriate value. See the following coding example. See Flip-Flops, Registers, and Latches in Chapter 3 for more information about the following:

- Advantages and disadvantages of operational `set/reset`
- Asynchronous versus synchronous `set/reset`

Download the coding example files from:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

Initializing Registers VHDL Coding 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;
```

Default Initial Values on Memory Elements

Because every memory element must come up in a known state, Vivado synthesis does not apply IEEE standards for initial values in some cases. For example: In the previous coding example, if `arb_onebit` is not initialized to `1`, Vivado synthesis assigns it a default of `0` as its initial state.

IMPORTANT: *Vivado synthesis does not follow the IEEE standard, where* `U` *is the default for* `std_logic`.

Initialization

Initialization is the same for both registers and RAM components.

Whenever possible, Vivado synthesis adheres whenever possible to the IEEE VHDL standard when initializing signal values.

If no initial values are supplied in the VHDL code, Vivado synthesis uses the default values (where possible) listed in the Vivado synthesis column of the following table.

Table 4-8: **VHDL Initial Values in IEEE and Vivado Synthesis**

Type	IEEE	Vivado Synthesis
Bit	0	0
std_logic	U	0
bit_vector (3 downto 0)	0	0
std_logic_vector (3 downto 0)	0	0
integer (unconstrained)	`integer'left`	`integer'left`
integer range 7 downto 0	`integer'left = 7`	`integer'left = 7 (coded as 111)`
integer range 0 to 7	`integer'left = 0`	`integer'left = 0 (coded as 000)`
Boolean	`FALSE`	`FALSE (coded as 0)`
enum (S0,S1,S2,S3)	`type'left = S0`	`type'left = S0 (coded as 000)`

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 VHDL Coding Example

Download the coding example files from:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

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 EXAMPLE is
    port (
        A,B : in BIT_VECTOR (3 downto 0);
        CIN : in BIT;
        S : out BIT_VECTOR (3 downto 0);
        COUT : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE 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 VHDL Coding Example

The following example uses a procedure:

```
-- 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 EXAMPLE is
    port (
        A,B : in BIT_VECTOR (3 downto 0);
        CIN : in BIT;
        S : out BIT_VECTOR (3 downto 0);
        COUT : out BIT );
```

```
end EXAMPLE;

architecture ARCHI of EXAMPLE 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 VHDL Coding 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

VHDL assert statements help you debug your design, and let you detect undesirable conditions, such as bad values for:

- Generics, constants, and generate conditions
- Parameters in called functions

IMPORTANT: *For any failed condition in an assert statement, depending on the severity level, Vivado synthesis either Issues a warning message, or supports the assert statement only with static condition.*

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`
- `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 the following types:
    - std_logic_1164: `std_logic`, `std_ulogic`, `std_logic_vector`, and `std_ulogic_vector` types.
- Conversion functions based on the following 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 the following 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 4-9: **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 4-10: **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 Syntax 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 has been 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 4-11: **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

Table 4-11: **VHDL Constructs and Support Status *(Cont'd)***

VHDL Construct	Support Status
VHDL Declarations	
Type	Supported for the following: • Enumerated types • Types with positive range having constant bounds • Bit vector types • Multi-dimensional arrays
VHDL Objects	**Support Status**
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
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.

Table 4-11: **VHDL Constructs and Support Status *(Cont'd)***

VHDL Construct	Support Status
Rem	Supported if the right operand is a constant power.
Mod	Supported if the right operand is a constant power.
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
Qualified Expressions	Supported for accepted predefined attributes.
Types Conversions	Supported
Allocators	**Unsupported**
Static Expressions	Supported
VHDL Statements	
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

Table 4-11: **VHDL Constructs and Support Status *(Cont'd)***

VHDL Construct	Support Status
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 Statement	Supported. • No after clause. • No transport or guarded options, • No waveforms • UNAFFECTED is supported.
Component Instantiation Statement	Supported
for-generate	Statement supported for constant bounds only
if-generate	Statement supported for static condition only

VHDL RESERVED Words

Table 4-12: **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

Table 4-12: **VHDL RESERVED Words *(Cont'd)***

RESERVED Words *(Cont'd)*			
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			

Verilog Language Support

Introduction

This chapter describes the Vivado® synthesis support for Verilog Hardware Description Language (VHDL).

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: 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)
- IEEE 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 5-1: **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 5-2: **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 5-3: **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 Verilog Coding 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 Verilog Coding 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 Verilog Coding 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 Coding Example

Coding examples are included in this chapter.

Download the coding example files from:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

```
// 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
```

Verilog Parameters and `Generate-For` Coding Example

This coding example illustrates how to control the creation of repetitive elements using parameters and generate-for constructs. For more information, see Generate Loop Statements.

Download the coding example files from:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

```
//
// 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:

- 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:

- 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 Coding 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 Coding 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

Unsigned integers in case item expressions can cause unpredictable results.

Integer Handling in Verilog Case Statements Coding Example

In the following coding example, the case item expression 4 is an unsigned integer that causes unpredictable results. To resolve this issue, size the case item expression 4 to 3 bits.

```
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 Verilog Concatenations

Unsigned integers in Verilog concatenations can cause unpredictable results. If you use an expression that results in an unsigned integer:

- 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 and meta comments include:

- Verilog-2001 Attributes
- Verilog 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 5-4: **Verilog Constructs**

Verilog Constants	
Constant	**Support Status**
Integer	Supported
Real	Supported
String	Unsupported
Verilog Data Types	
Data Type	**Support Status**
Net types: • tri0 • tri1 • trireg	Unsupported
All Drive strengths	Ignored
Real and realtime registers	Unsupported
All Named events	Unsupported
Drive Strength	Ignored

Table 5-4: **Verilog Constructs**

Verilog Constants	
Constant	**Support Status**
Delay	Ignored
Verilog Procedural Assignments	
Procedural Assignment	**Support Status**
assign	Supported with limitations. See *Assign and Deassign Statements.*
deassign	Supported with limitations. See *Assign and Deassign 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
Verilog Design Hierarchies	
Design Hierarchy	**Support Status**
Module definition	Supported
Macromodule definition	Unsupported
Hierarchical names	Unsupported
Defparam	Supported
Array of instances	Supported
Verilog Compiler Directives	
Compiler Directive	**Support Status**
`celldefine `endcelldefine	Ignored
`default_nettype	Supported
`define	Supported
`ifdef `else `endif	Supported

Table 5-4: **Verilog Constructs**

Verilog Constants	
Constant	**Support Status**
\`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 5-5: **System Tasks and Status**

System Task or Function	Status	Comment
$display	Supported	Escape sequences are limited to %d, %b, %h, %o, %c, and %s
$fclose	Supported	
$fdisplay	Ignored	
$fgets	Supported	
$finish	Supported	$finish is supported for statically never active conditional branches only
$fopen	Supported	
$fscanf	Supported	Escape sequences are limited to %b and %d
$fwrite	Ignored	
$monitor	Ignored	
$random	Ignored	
$readmemb	Supported	
$readmemh	Supported	
$signed	Supported	

Table 5-5: **System Tasks and Status *(Cont'd)***

System Task or Function	Status	Comment
$stop	Ignored	
$strobe	Ignored	
$time	Ignored	
$unsigned	Supported	
$write	Supported	Escape sequences are limited to %d, %b, %h, %o, %c and %s
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 Syntax Example

The syntax for reporting the value of a binary constant in decimal is:

```
parameter c = 8’b00101010;
  initial begin
$display ("The value of c is %d", c);
  end
```

Verilog Log File Example

Vivado synthesis writes the following content to the log file:

```
Analyzing top module <example>. c =
8'b00101010
"foo.v" line 9: $display : The value of c is 42
```

Verilog Primitives

Vivado synthesis supports Verilog gate-level primitives except as shown in the following table. Vivado synthesis does not support Verilog switch-level primitives, such as:

`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 Coding Example

```
and U1 (out, in1, in2); bufif1 U2 (triout, data, trienable);
```

Unsupported Verilog Gate Level Primitives

The following gate-level primitives are not support in Vivado synthesis.

Table 5-6: **Unsupported Primitives**

Primitive	Status
pulldown and pullup	Unsupported
drive strength and delay	Ignored
Arrays of primitives	Unsupported

Verilog Reserved Keywords

Keywords marked with an asterisk (*) are reserved by Verilog, but Vivado synthesis does not support them. The following table lists the reserved keywords.

Table 5-7: **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

Table 5-7: **Verilog Reserved Keywords *(Cont'd)***

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.

Initial Values

Variable Declarations Coding Example

```
reg [3:0] arb_priority;
wire [31:0] arb_request;
wire signed [8:0] arb_signed;
```

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 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 file as an FDP or FDC Flip-Flop.

Initial Values Coding Example One

```
reg arb_onebit = 1'b0;
reg [3:0] arb_priority = 4'b1011;
```

Initial Values Coding 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 Coding 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 Coding 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

Data Types

Multi-Dimensional Array Verilog Coding 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 Verilog Coding 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];
```

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 Verilog 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 Verilog Data Types Coding 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 5-8: **Supported Operators**

Arithmetic	Logical	Relational	Conditional
+	&	<	?
-	&&	==	
*	|	===	
**	||	<=	
/	^	>=	
%	~	>=	
	~^	!=	
	^~	!==	
	<<	>	
	>>		
	<<<		
	>>>		

Supported Expressions

Table 5-9: **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

Table 5-9: **Supported Expressions *(Cont'd)***

Expression	Symbol	Status
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
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

Table 5-9: **Supported Expressions *(Cont'd)***

Expression	Symbol	Status
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 5-10: **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 signalled 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 Coding Example One

```
module example (A, B, O);
input     A, B;
output O;
assign O = A & B;
endmodule
```

Behavioral Verilog Module Declaration Coding Example Two

```
module example ( input     A, input   B output O
):

assign O = A & B;

endmodule
```

Module Instantiation

- A Behavioral Verilog module instantiation statement:
    - 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.

- A Behavioral Verilog module is instantiated in another module. See the following coding example.

Behavioral Verilog Module Instantiation Coding 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.
- For the creation of Latches, make sure that all assigned signals in a combinatorial process are always explicitly assigned in all paths of the process statements.
- 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.

- If-else statements use true-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 Coding 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 Coding Example

```
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
case (sel)
  2'b00: outmux = a;
  2'b01: outmux = b;
  2'b10: outmux = c;
  default: 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.

Disable Statements

Disable statements are not supported.

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.
- While `loops` can have disable statements. The `disable` statement is used inside a labeled block, as shown in the following code snippet:

  ```
  disable <blockname>
  ```

While Loop Coding 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 Coding 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 Coding 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 Coding 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 Coding 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;
endmodule
```

Assign and Deassign Statements

Vivado synthesis does not support assign and deassign 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 Coding Examples

For update information, see "Coding Examples" in the Introduction.

Tasks and Functions Coding 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 Coding 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 Coding 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

Vivado synthesis supports function calls to calculate constant values.

Constant Functions Coding 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
```

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 Coding Example One

```
reg a;
a = #10 (b | c);
```

Blocking Procedural Assignment Syntax Coding 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 Coding 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.

Constants

Non-Blocking Procedural Assignment Coding Example Two

This coding example shows how to use a non-blocking procedural assignment.

```
if (in1) out <= 1’b1;
else out <= in2;
```

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 Expressions Example

The following constant expressions represent the same value.

- `4'b1010`
- `4'o12`
- `4'd10`
- `4'ha`

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:
    - 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.

Include Files

Macros Coding Example One

```
'define myzero 0
assign mysig = 'myzero;
```

Macros Coding Example Two

```
'ifdef MYVAR
module if_MYVAR_is_declared;
...
endmodule
'else
module if_MYVAR_is_not_declared;
...
endmodule
'endif
```

- Verilog allows you to separate HDL source code into more than one file.
- Use either file inclusion method or design project file method to reference the additional files.

RECOMMENDED: Xilinx® recommends the design project file method.

File Inclusion Method

IMPORTANT: *Xilinx does not recommend the file inclusion method.*

- 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 Coding 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);
else
end

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
```

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.
- 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:
   a. User-specified list of unified logical libraries.
   b. 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`.

Accepted Coding Example

```
ff #(.init(2'b01)) u1 (.sel(sel), .din(din), .dout(dout));
```

NOT Accepted Coding 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. If the module name is not all lowercase, use case to preserve the module case.
4. 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 directions
  - 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.

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 Node Properties window, change the Type from **Verilog** to **SystemVerilog**, and click **Apply**.

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 for behavior. They can 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, union] 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 this:

```
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

A union is a data type comprising multiple data types. Only one data type is used. This is useful in cases where the data type changes depending on how it is used. The following code snippet is an example:

```
typedef union {int i; logic [7:0] j} my_union;
my_union sig1;
my_union sig2;
sig1.i = 32;  //sig1 gets the int format
sig2.j = 8’b00001111;  //sig2 get the 8bit logic format.
```

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 one of 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.

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.

Answer Records

- SystemVerilog 2008 Language support: AR#62005

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 Migration Guide* (UG911)
7. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
8. *Vivado Design Suite User Guide: Using Constraints* (UG903)

9. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)
10. *Vivado Design Suite Tutorial: Design Analysis and Closure Techniques* (UG938)
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)

Download the coding example files from:
https://secure.xilinx.com/webreg/clickthrough.do?cid=371104

Vivado Design Suite QuickTake Video Tutorials

Vivado Design Suite Documentation

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