Vivado Design Suite User Guide: Synthesis (UG901)

Mixed Language Support

Additional Resources and Legal Notices

Introduction

Synthesis is the process of transforming a Register Transfer Level (RTL) specified design into a gate-level representation. AMD Vivado™ synthesis is timing-driven and optimized for memory usage and performance. Vivado synthesis supports a synthesizeable subset of:

- SystemVerilog: IEEE Standard for SystemVerilog-Unified Hardware Design Specification, and Verification Language (IEEE Std 1800-2012)
- Verilog: IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364-2005)
- VHDL: IEEE Standard for VHDL Language (IEEE Std 1076-2002)
- VHDL 2008
- Mixed languages: Vivado supports a mix of VHDL, Verilog, and SystemVerilog.

In most instances, 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 *ISE to Vivado Design Suite Migration Guide* (UG911).

There are two ways to setup and run synthesis:

- Use Project Mode, selecting options from the Vivado Integrated Design Environment (IDE).
- 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) for more information about operation modes. This chapter covers both modes in separate subsections.

Navigating Content by Design Process

AMD Adaptive Computing documentation is organized around a set of standard design processes to help you find relevant content for your current development task. You can access the AMD Versal™ adaptive SoC design processes on the Design Hubs page. You can also use the Design Flow Assistant to better understand the design flows and find content that is specific to your intended design needs. This document covers the following design processes:

Hardware, IP, and Platform Development

Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the AMD Vivado™ timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:

  - Vivado Synthesis
  - Synthesis Attributes
  - Using Block Synthesis Strategies

Vivado Synthesis

Synthesis Methodology

The AMD 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 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 a strategy. A strategy is a configuration of command options that you can apply to design runs for synthesis or implementation. See Creating Run Strategies.
- Queue the synthesis and implementation runs to launch sequentially or simultaneously with multi-processor machines. See Running Synthesis.
- Monitor synthesis or implementation progress, view log reports, and cancel runs. See Monitoring the Synthesis Run.

Using Synthesis

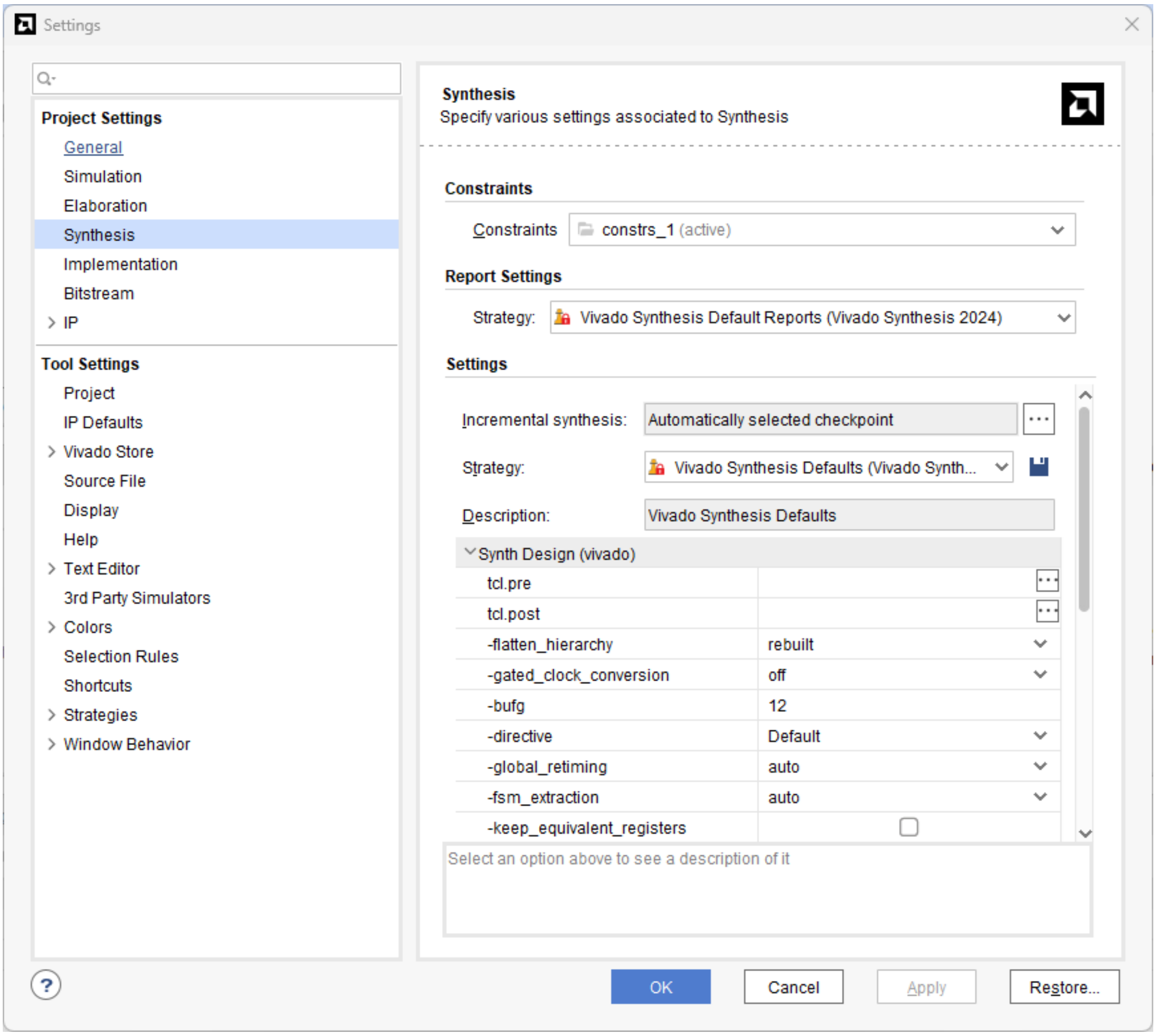
This section describes using the AMD Vivado™ IDE to set up and run Vivado synthesis. The corresponding Tcl Console commands follow most Vivado IDE procedures, and most Tcl commands link directly to the *Vivado Design Suite Tcl Command Reference Guide* (UG835). Additionally, there is more information regarding Tcl commands and using Tcl in the *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894).

✎ **Video:** See the following for more information: Vivado Design Suite QuickTake Video: Synthesis Options and Vivado Design Suite QuickTake Video: Synthesizing the Design.

Using Synthesis Settings

1. From the Flow Navigator, click Settings, select Synthesis, or select Flow >Settings > Synthesis Settings.

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



Synthesis
Specify various settings associated to Synthesis

Constraints
Constraints: constrs_1 (active)

Report Settings
Strategy: Vivado Synthesis Default Reports (Vivado Synthesis 2024)

Settings
Incremental synthesis: Automatically selected checkpoint
Strategy: Vivado Synthesis Defaults (Vivado Synth...
Description: Vivado Synthesis Defaults

Synth Design (vivado)	
tcl.pre	
tcl.post	
-flatten_hierarchy	rebuilt
-gated_clock_conversion	off
-bufg	12
-directive	Default
-global_retiming	auto
-fsm_extraction	auto
-keep_equivalent_registers	☐

Select an option above to see a description of it

2. Under the Constraints section of the Settings dialog box, select the Default Constraint Set as the active constraint set; 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

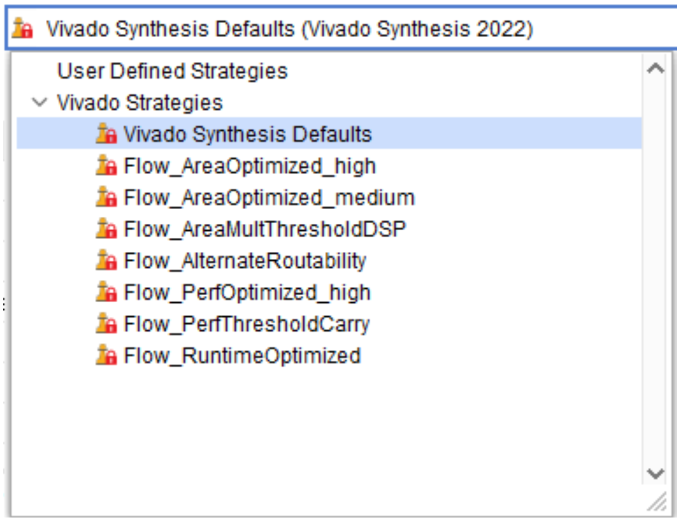
   > These constraints define pin placement and absolute, or relative, placement of cells such as block RAMs, LUTs, Flip-Flops, and device configuration settings.

   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) for more information about organizing constraints.
   New runs use the selected constraint set, and the Vivado synthesis targets this constraint set for design changes.
3. From the Options area: Select a Strategy from the drop-down menu where you can view and select a predefined synthesis strategy to use for the synthesis run.

There are different preconfigured strategies, as shown in the following figure.


Vivado Synthesis Defaults (Vivado Synthesis 2022)
User Defined Strategies
Vivado Strategies
Vivado Synthesis Defaults
Flow_AreaOptimized_high
Flow_AreaOptimized_medium
Flow_AreaMultThresholdDSP
Flow_AlternateRoutability
Flow_PerfOptimized_high
Flow_PerfThresholdCarry
Flow_RuntimeOptimized

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

For a list of all the strategies and their respective settings, see the directive option in the following list and see Vivado Preconfigured Strategies to see a matrix of strategy default settings.

4. Select from the displayed options:

   flatten_hierarchy

   Determines how Vivado synthesis controls hierarchy.

   none

   Instructs the synthesis tool to never flatten the hierarchy. The output of synthesis has the same hierarchy as the original RTL.

   full

   Instructs the tool to fully flatten the hierarchy leaving only the top level.

   rebuilt

   When set, rebuilt allows the synthesis tool to flatten the hierarchy, perform synthesis, and 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 using an RTL attribute to work. See GATED_CLOCK, 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 Vivado synthesis tool infers up to nine more BUFGs.

directive

Replaces the `-effort_level` option. When specified, this option runs Vivado synthesis with different optimizations. See Vivado Preconfigured Strategies for a list of all strategies and settings. Values are:

Default

Default settings. See Vivado Preconfigured Strategies.

RuntimeOptimized

Performs fewer timing optimizations and eliminates some RTL optimizations to reduce synthesis runtime.

AreaOptimized_high

Performs general area optimizations, including forcing ternary adder implementation, applying new thresholds for using carry chain in comparators, and implementing area-optimized multiplexers.

AreaOptimized_medium

Performs general area optimizations, including changing the threshold for control set optimizations, forcing ternary adder implementation, lowering multiplier threshold of inference into DSP blocks, moving shift register into block RAM, applying lower thresholds for use of CARRY chain in comparators, and also area optimized MUX operations.

AlternateRoutability

Set of algorithms to improve route-ability (less use of MUXFs and CARRYs)

AreaMapLargeShiftRegToBRAM

Detects large shift registers and implements them using dedicated block RAM.

AreaMultThresholdDSP

Lower threshold for dedicated DSP block inference.

FewerCarryChains

Higher operand size threshold to use LUTs instead of the carry chain.

LogicCompaction

Arranges CARRY chains and LUTs in such a way that it makes the logic more compact using fewer SLICES. This could have a negative effect on timing QoR.

PerformanceOptimized

Performs general timing optimizations, including logic level reduction at the expense of area.

PowerOptimized_high

Performs general timing optimizations including logic level increase at the expense of area.

PowerOptimized_medium

Performs general timing optimizations by lowering logic level reduction at the expense of area.

global_retiming

devices only. For controlling retiming select the `-global_retiming` option. This option <auto|on|off> provides an option to perform for intra-clock sequential paths by automatically moving registers (register balancing) across combinatorial gates or LUTs. It maintains the original behavior and latency of the circuit and does not require changes to the RTL sources. The default is auto. For Versaldevices retiming will performed when auto is set. For non-Versaldevices, retiming will not be performed when auto is set.

✎ **Note:** When retiming in OOC mode, registers that are driven by or that are driving ports are not retimed.

fsm_extraction

Controls how synthesis extracts and maps finite state machines. FSM_ENCODING describes the options in more detail.

keep_equivalent_registers

Prevents merging of registers with the same input logic.

resource_sharing

Sets the sharing of arithmetic operators between different signals. The values are auto, on, and off. The auto value sets performing resource sharing depend on the timing of the design.

control_set_opt_threshold

Sets the threshold for the clock to enable optimization to the lower number of control sets. The default is auto which means the tool chooses a value based on the device being targeted. Any positive integer value is supported. The given value is the number of fanouts necessary for the tool to move the control sets into the D logic of a register. If the fanout is higher than the value, the tool attempts to have that signal drive the `control_set_pin` on that register.

no_lc

When checked, this option turns off LUT combining.

no_srlextract

When checked, this option turns off SRL extraction for the full design so that they are implemented as simple registers.

shreg_min_size

Is the threshold for inference of SRLs. The default setting is 3. This sets the number of sequential elements that would result in the inference of an SRL for fixed delay chains (static SRL). Strategies define this setting as 5 and 10 also. See Vivado Preconfigured Strategies for a list of all strategies and settings.

max_bram

Describes the maximum number of block RAM allowed in the design. Often this is used when there are black boxes or third-party netlists in the design and allow the designer to save room for these netlists.

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

max_uram

Sets the maximum number of UltraRAM (AMD UltraScale+™ device block RAMs) blocks allowed in design. The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.

max_dsp

Describes the maximum number of block DSP allowed in the design. Often this is used when there are black boxes or third-party netlists in the design and allows room for these netlists. The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.

max_bram_cascade_height

Controls the maximum number of block RAM that can be cascaded by the tool. The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.

max_uram_cascade_height

Controls the maximum number of UltraScale+ device UltraRAM blocks that can be cascaded by the tool. The default setting of -1 indicates that the tool chooses the maximum number allowed for the specified part.

cascade_dsp

Controls how adders in sum DSP block outputs are implemented. By default, the sum of the DSP outputs is computed using the block built-in adder chain. The value tree forces the sum to be implemented in the fabric. The values are auto, tree, and force. The default is auto.

no_timing_driven

(Optional) Disables the default timing-driven synthesis algorithm. This results in a reduced synthesis runtime, but ignores the effect of timing on synthesis.

sfcu

Run synthesis in single-file compilation unit mode.

assert

Enable VHDL assert statements to be evaluated. A severity level of failure or error stops the synthesis flow and produces an error. A severity level of warning generates a warning.

debug_log

Prints out extra information in the synthesis log file for debugging purposes. The `-debug_log` should be added to the More Options field.

- 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 *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894) for more information about Tcl scripting.
Use the `DIRECTORY` property of the current project or current run to define the relative paths in your scripts.

5. Click Finish.

Tcl Commands to Get Property

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

Creating Run Strategies

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

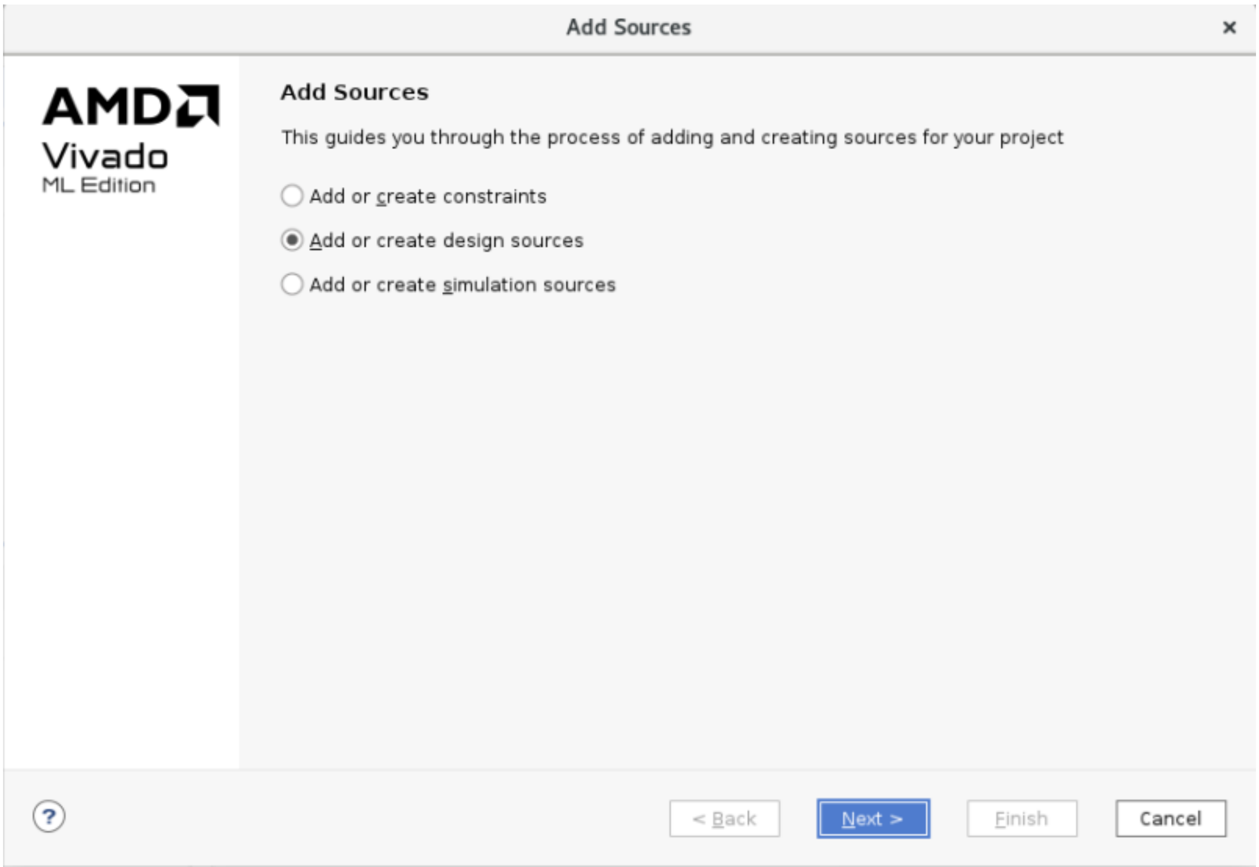
✎ **Video:** See the following for more information: Vivado Design Suite QuickTake Video: Creating and Managing Runs.

Select Settings from the Flow Navigator, select Synthesis, and select a Strategy from the Strategy drop-down list, shown in previous figure, and click OK.

Setting Synthesis Inputs

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

1. From the File menu or the Flow Navigator, select the Add Sources command to open the Add Sources wizard, shown in the following figure.



Add Sources

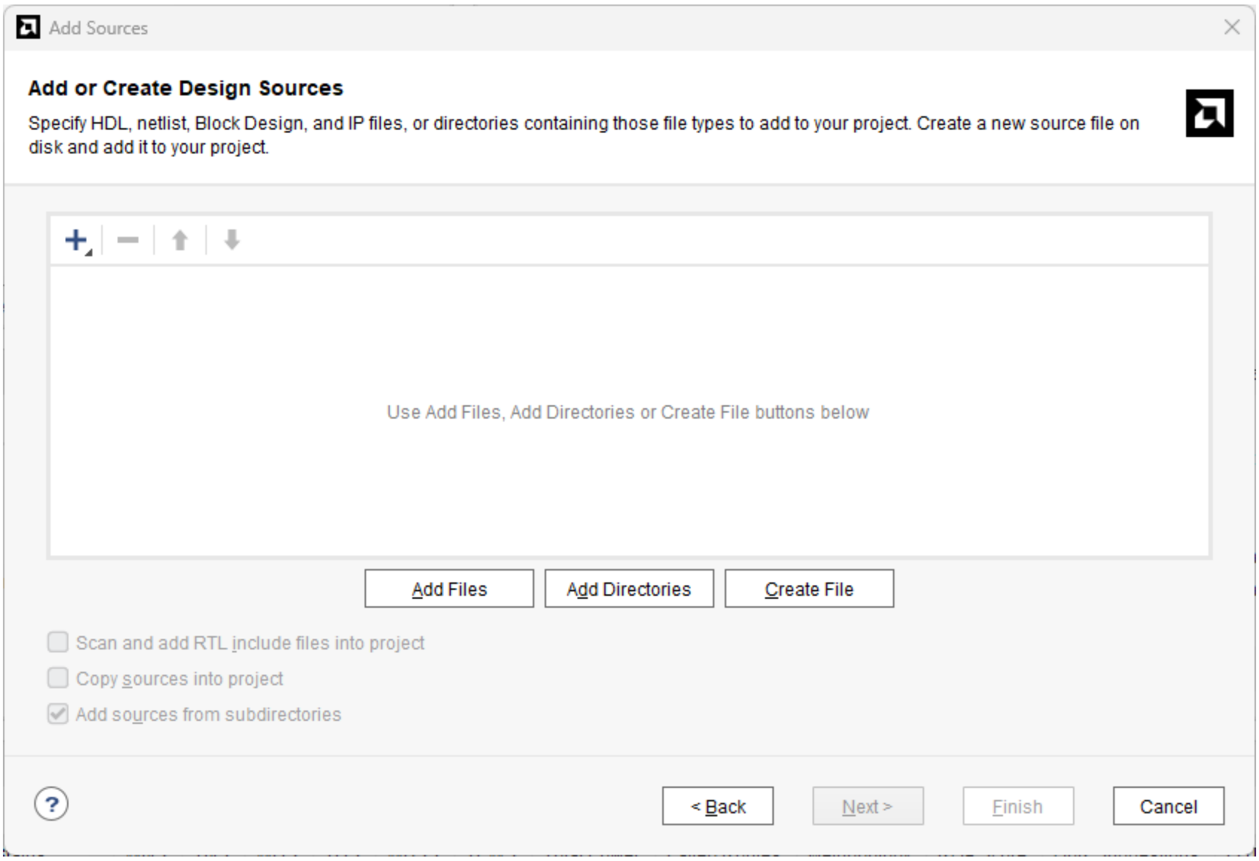
This guides you through the process of adding and creating sources for your project

○ Add or create constraints

● Add or create design sources

○ Add or create simulation sources

2. Select an option corresponding to the files to add, and click Next.
   The following figure shows the Add or Create Design Sources page that is displayed if Add or create design sources is selected.



Add or Create Design Sources

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

Use Add Files, Add Directories or Create File buttons below

☐ Scan and add RTL include files into project

☐ Copy sources into project

☑ Add sources from subdirectories

3. Add constraint, RTL, or other project files, click Finish.
   See *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) for

more information about creating RTL source projects.

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

The following chapters provide details on supported HDL constructs.

- HDL Coding Techniques
- VHDL Support
- VHDL-2008 Language Support
- Verilog Language Support
- SystemVerilog Support
- Mixed Language Support

Vivado synthesis also supports several RTL attributes that control synthesis behavior. Synthesis Attributes describes these attributes. For timing constraints, Vivado synthesis uses the XDC file.

Using Block Synthesis Strategies describes the available Block Synthesis Strategies.

‼ **Important:** Vivado Design Suite does not support the UCF format. See *ISE to Vivado Design Suite Migration Guide* (UG911) for the UCF to XDC conversion procedure.

Controlling File Compilation Order

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

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

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

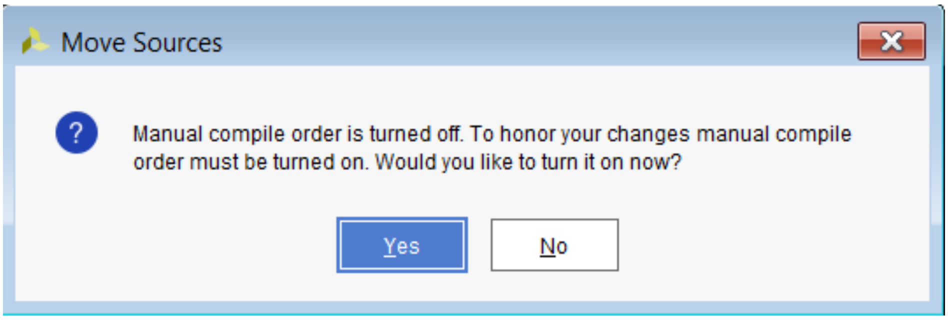
The default setting, Automatic Update and Compile Order, specifies that the tool manages the compilation order as shown in the Compilation Order window, and shows which modules are used and where they are in the hierarchy tree in the Hierarchy window.

The compilation order updates automatically as you change source files.

To modify the compile order before synthesis, select a file, and right-click Hierarchy Update > Automatic Update, Manual Compile Order so that the Vivado IDE can automatically determine the best top module for the design and allows for manual specification of the compilation order.

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

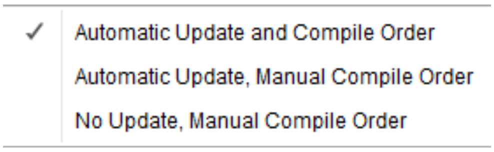
Figure: Move Sources Option



Manual compile order is turned off. To honor your changes manual compile order must be turned on. Would you like to turn it on now?

From the Sources window Compile order tab, drag, and drop files to arrange the compilation order, or use the menu Move Up or Move Down commands.
Other options are available from the Hierarchy Update context menu, as shown in the following figure.

Figure: Hierarchy Update Options



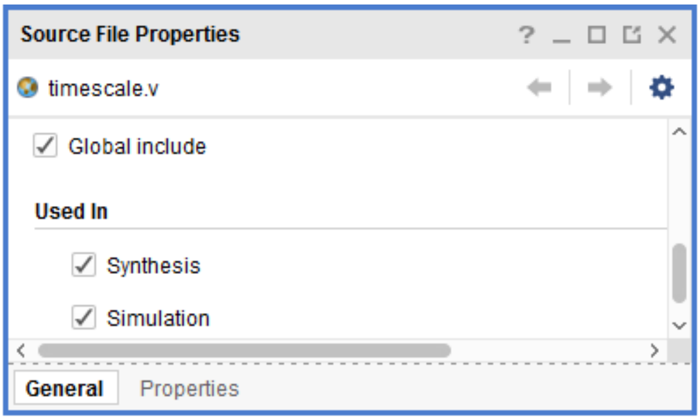
✓ Automatic Update and Compile Order
Automatic Update, Manual Compile Order
No Update, Manual Compile Order

See *Vivado Design Suite User Guide: Design Flows Overview* (UG892) for information about design flows.

Defining Global Include Files

The Vivado IDE supports designating one of more Verilog or Verilog Header source files as global ‘include files and processes those files before any other sources. Designs that use common header files might require multiple `include statements to be repeated across multiple Verilog sources used in the design.
To designate a Verilog or Verilog header file as a global `include file:

1. In the Sources window, select the file.
2. Check the Global include check box in the Source File Properties window, as shown in the following figure.



☑ Global include

Used In

☑ Synthesis

☑ Simulation

★ **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 *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893), for information about the Sources window.

RTL Linter

Vivado Synthesis includes an RTL linter that can analyze your code to determine if there are any code segments that, while legal, could cause issues with your design.

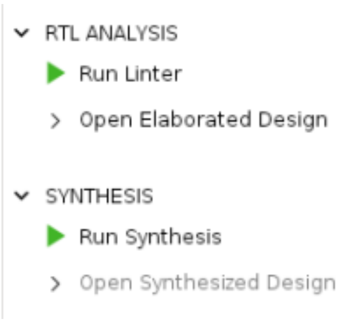
Running the Linter

There are two ways to run the linter. The first is the command line with the -lint option in synth_design :

```
synth_design -top <top_level> -part <part> -lint
```

The second way is with the IDE. In the RTL ANALYSIS section of the Project Manager, there is a Run Linter button that runs the linter with the current part and top level.

Figure: RTL Analysis

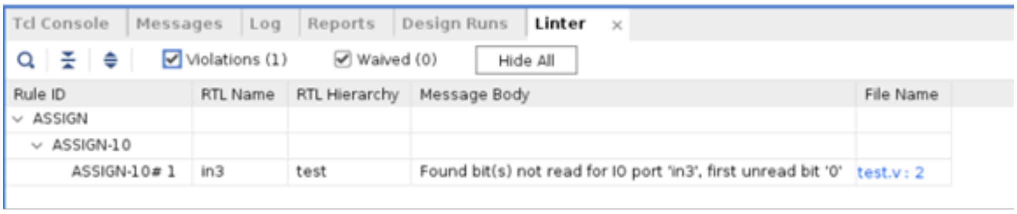


✎ **Note:** The RTL linter currently does not work on out-of-context runs.

Linter Output

After running the linter, the tool will create a new tab with the results.

Figure: Linter



Rule ID	RTL Name	RTL Hierarchy	Message Body	File Name
ASSIGN				
ASSIGN-10				
ASSIGN-10# 1	in3	test	Found bit(s) not read for IO port 'in3', first unread bit '0'	test.v : 2

The output will have a Rule ID, the RTL name of the signal or port with the problem, the hierarchy where this is found, and a message and file name of the potential problem. In the previous figure, there is a port called “in3” in test.v, that is never used.

Creating Waivers for the RTL Linter

Waivers are created so that the Linter can ignore certain conditions. These are created with the `create_waiver` command with the -type LINT option For example :

```
create_waiver -type LINT -id ASSIGN-1 -rtl_hierarchy x/y
```

This waiver suppresses reporting on any ASSIGN-1 issues in the x/y hierarchy. Waivers can use the following options in any combination.

id

Rule ID for the waiver.

rtl_name

The signal or port name.

rtl_hierarchy

The hierarchy in the design.

rtl_file

The file in question.

Once you get the correct waivers for your design, you can write these to a Tcl file for future use in your flow:

```
write_waivers -type LINT -file <filename>.tcl
```

List of Linter Rules

The RTL linter warns about the following types of issues.

Table: Linter Rules

Name	ID name	Description
Arithmetic overflow	ASSIGN-1	For arithmetic

Name	ID name	Description
		expressions when the target is not large enough to hold the full precision of the result.
Operands have mixed signs	ASSIGN-2	In an arithmetic operation, when operands have different signs. ✎ **Note:** Operands on mixed signs in Verilog are treated as unsigned.
Shifter overflow	ASSIGN-3	In a shift operation, when the shift value is larger than the size of the result. The result is in all 0's.
Signal bits not set	ASSIGN-5	When one or more bits of signal are not set.
Signal bits are not used	ASSIGN-6	When one ore more bits of a signal are not read from.
Multiple assignments in an array	ASSIGN-7	When one or more bits in an array have been assigned multiple times. Could lead to multi-driver issues later in flow.
Comparison of arrays of different sizes	ASSIGN-8	When signals of array types of different dimensions are directly compared.
IO bits are not set	ASSIGN-9	When one or more bits of an I/O are not set.
IO bits are not used	ASSIGN-10	When one or more bits of an I/O are not read.
Arithmetic operators are not mergeable	QOR-1	When two consecutive arithmetic operators cannot be merged.

Name	ID name	Description
Inferred latch	INFER-1	Latch is inferred instead of register, which can be unintended.

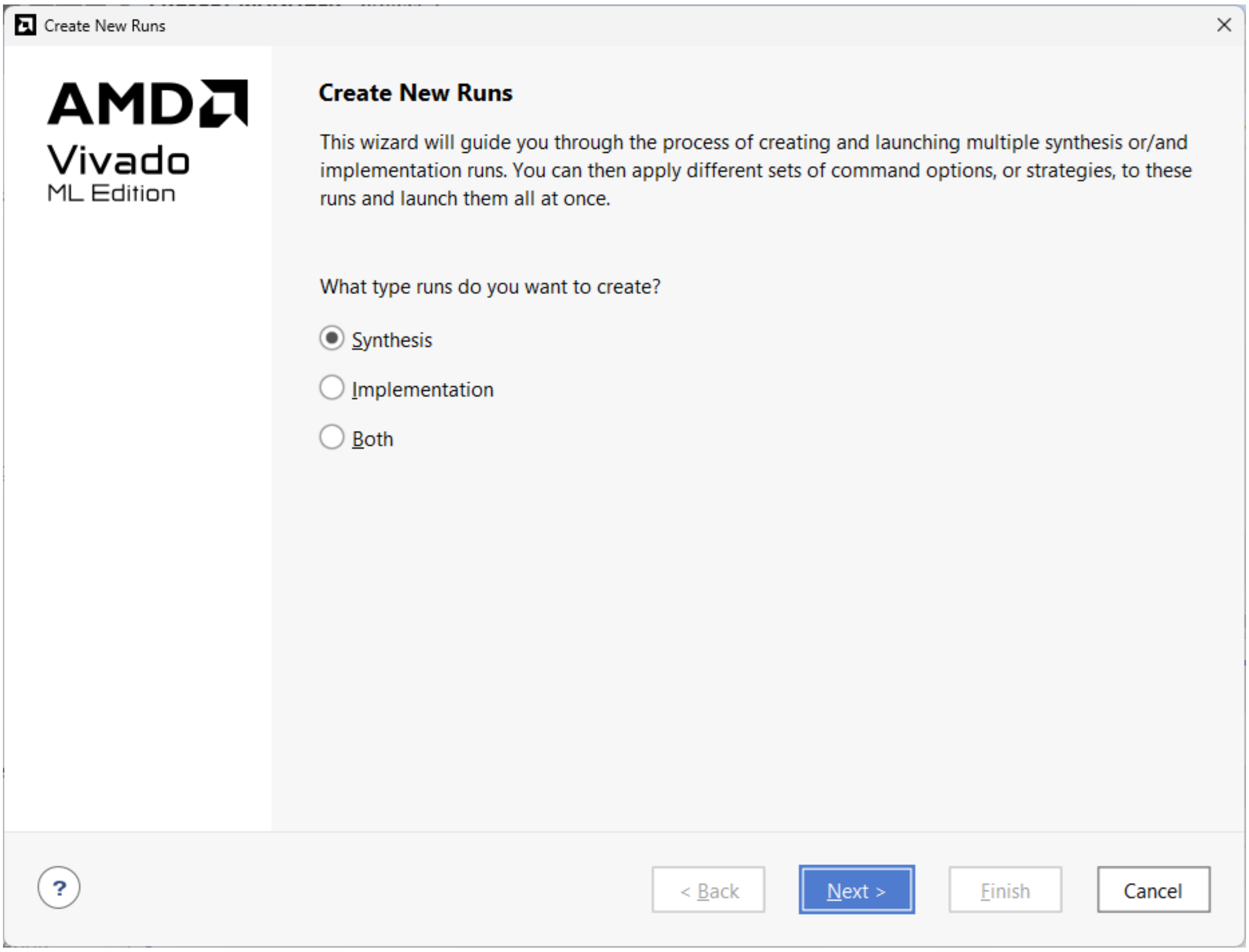
Running Synthesis

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

- AMD 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. Select Flow > Create Runs, or in Design Runs, click the Create Runs button ✚ to open the Create New Runs wizard. The Create New Runs dialog box opens, as shown in the following figure.



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.

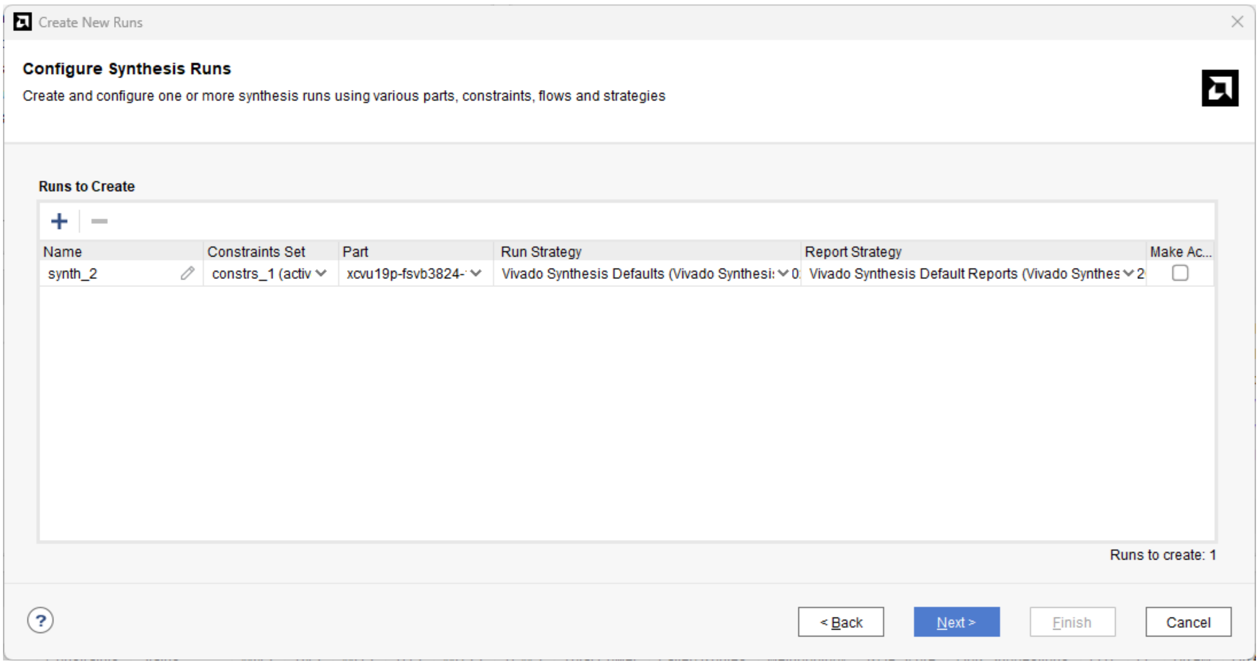
What type runs do you want to create?

● Synthesis

○ Implementation

○ Both

2. Select Synthesis, and click Next.
   The Configure Synthesis Runs opens, as shown in the following figure.



Configure Synthesis Runs

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

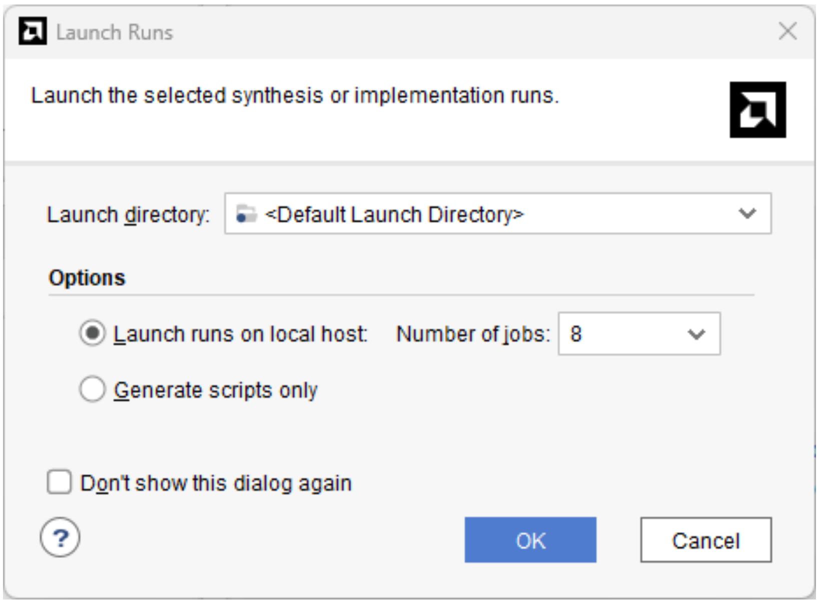
Runs to Create

Name	Constraints Set	Part	Run Strategy	Report Strategy	Make Ac...
synth_2	constrs_1 (activ	xcvu19p-fsvb3824-	Vivado Synthesis Defaults (Vivado Synthesi:	Vivado Synthesis Default Reports (Vivado Synthes	☐

Runs to create: 1

3. Click the Add button and configure the synthesis run with the Name, Constraints Set, Part, Strategy, and check Make Active, if you want this run to be the active run.
   The Vivado IDE contains a default strategy. You can set a specific name for the strategy run or accept the default name(s), which are numbered as synth_1, synth_2, and so forth. To create your own run strategy, see Creating Run Strategies.
   See *Vivado Design Suite User Guide: Using Constraints* (UG903)
   - For detailed information on constraints, see "About XDC Constraints."
   - For detailed information about constraint processing order, see "Constraint Files Order with IP Cores."

   After some constraints are processed for a project, those constraint attributes can become design properties. For more information about design properties, see the *Vivado Design Suite Properties Reference Guide* (UG912).
4. Click Next. The Launch Options page opens.



Launch the selected synthesis or implementation runs.

Launch directory: <Default Launch Directory>

Options

● Launch runs on local host: Number of jobs: 8

○ Generate scripts only

☐ Don't show this dialog again

5. In the Launch Options page, set the options as follows, click Next.
    - In the Launch Directory drop-down option, browse to and select the directory from which to launch the run.
    - In the Options area, choose one of the following:

      Launch runs on local host

      Runs the options from the machine on which you are working. The Number of jobs drop-down lets you specify how many runs to launch.

      ✎ **Note:** The number of jobs can significantly affect the amount of memory used by the Vivado tool. Turning this to a very high number could cause the tool to take up large amounts of memory depending on the sizes of the individual runs or OOC runs in the design. Using too much memory could lead to crashes in the tool.

      Launch runs on remote hosts (Linux only)

      Launches the runs on a remote host and configures that host. See *Vivado Design Suite User Guide: Implementation* (UG904), for more information about launching runs on remote hosts in Linux. Use the Configure Hosts button to configure the hosts from the dialog box.

      Launch runs on cluster

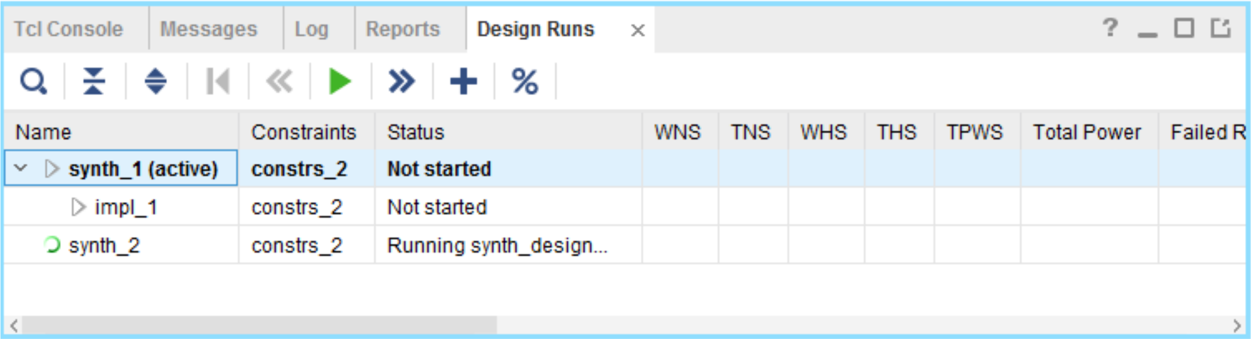
      Launches the runs on an external tool such as lsf. Hitting the settings button allows the configuration of that cluster tool.

      Generate scripts only

      Generates scripts to run later. Use runme.bat (Windows) or runme.sh (Linux) to start the run.

6. After setting the Create New Runs wizard option, click Finish in the Launch Runs summary.

   You can see the results in the Design Runs window, as shown in the following figure.



Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed R
synth_1 (active)	constrs_2	Not started							
impl_1	constrs_2	Not started							
synth_2	constrs_2	Running synth_design...							

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. Use the tree widgets in the window to expand, and collapse synthesis runs. 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, delete, or change properties on 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, right-click, and select the Make Active command from the pop-up 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, 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 *Vivado Design Suite User Guide: Implementation* (UG904), for more information about using remote hosts.

Setting a Bottom-Up, Out-of-Context Flow

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

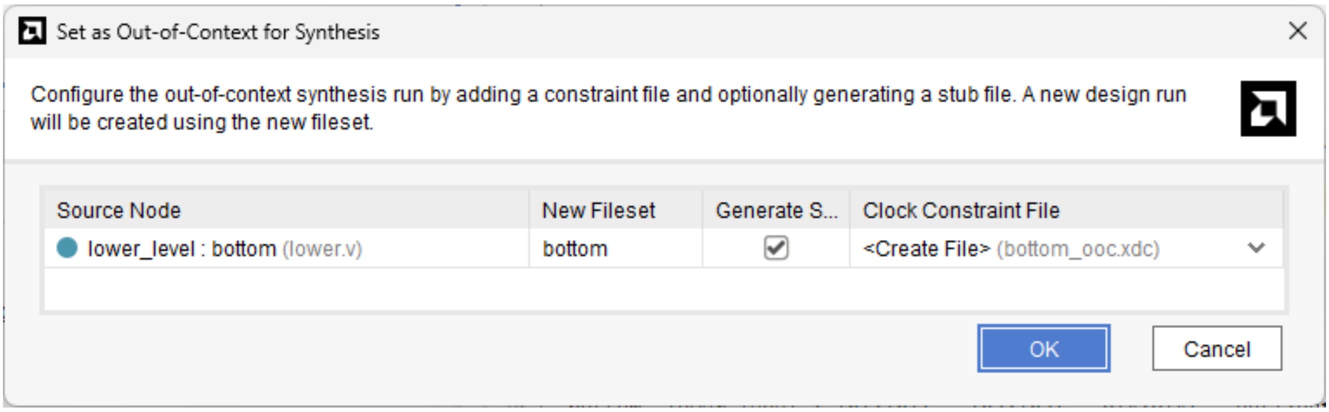
- Lower OOC modules run separately from the top level and have their own constraints.
- OOC modules can be run as needed.
- After you have run synthesis on an OOC module, it does not need to be rerun unless you change the RTL or constraints for that run.
- When the top level is run, the lower level OOC runs are treated as black boxes.

If any IP is synthesized in OOC mode, the top-level synthesis run infers a black box for these IP. Hence, users cannot reference objects, such as pins, nets, and cells, internal to the IP as part of the top-level synthesis constraints. During implementation, the netlists from the IP DCPs are linked with the netlist produced when synthesizing the top-level design files, and the Vivado Design Suite resolves the IP black boxes. The IP XDC output products generated during implementation are applied along with any user constraints. If any constraints reference items inside the IP, there are warnings during synthesis about this, but they can be resolved during implementation.

This can result in a large runtime improvement for the top level because synthesis no longer needs to improve 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, find that module in the hierarchy view, and right-click the Set As Out-Of-Context for Synthesis option, shown in the following figure, and click OK.

Figure: Set as Out-of-Context Synthesis Dialog Box



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 S...	Clock Constraint File
lower_level : bottom (lower.v)	bottom	☑	<Create File> (bottom_ooc.xdc)

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

Source Node

Module to which you apply the OOC.

New Fileset

Lists the New Fileset name, which you can edit.

Generate Stub

A checkbox that you can check to have the tool create a stub file.

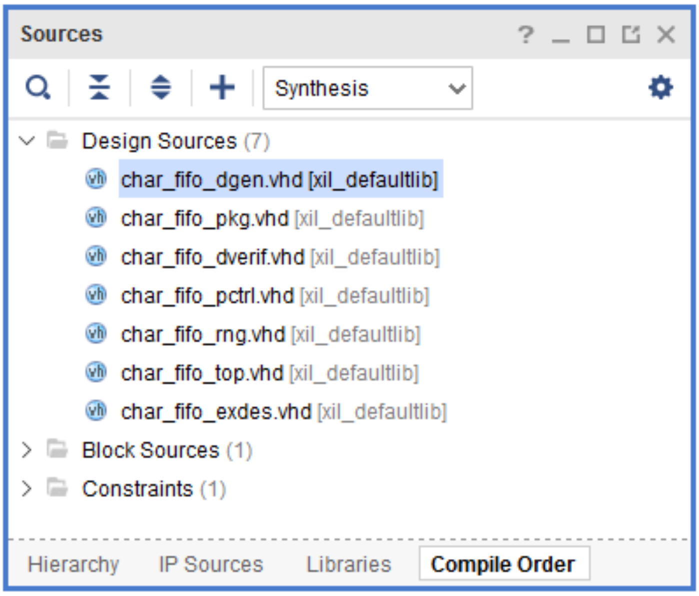
Clock Constraint File

Choose to have the tool create a new XDC template for you, or you can use the drop-down menu to copy an existing XDC file to this Fileset. This XDC file should have clock definitions for all your clock pins on the OOC module.

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

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

Figure: Compile Order Tab



Sources
Synthesis
Design Sources (7)
char_fifo_dgen.vhd [xil_defaultlib]
char_fifo_pkg.vhd [xil_defaultlib]
char_fifo_dverif.vhd [xil_defaultlib]
char_fifo_pctrl.vhd [xil_defaultlib]
char_fifo_rng.vhd [xil_defaultlib]
char_fifo_top.vhd [xil_defaultlib]
char_fifo_exdes.vhd [xil_defaultlib]
Block Sources (1)
Constraints (1)
Hierarchy
IP Sources
Libraries
Compile Order

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, described in Launching a Synthesis Run. This action sets the lower level as a top module and runs synthesis on that module without creating I/O buffers.

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

⚠ **CAUTION!** Do not use the Bottom-Up OOC flow when there are AMD IP in OOC mode in the lower levels of the OOC module. To have AMD 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

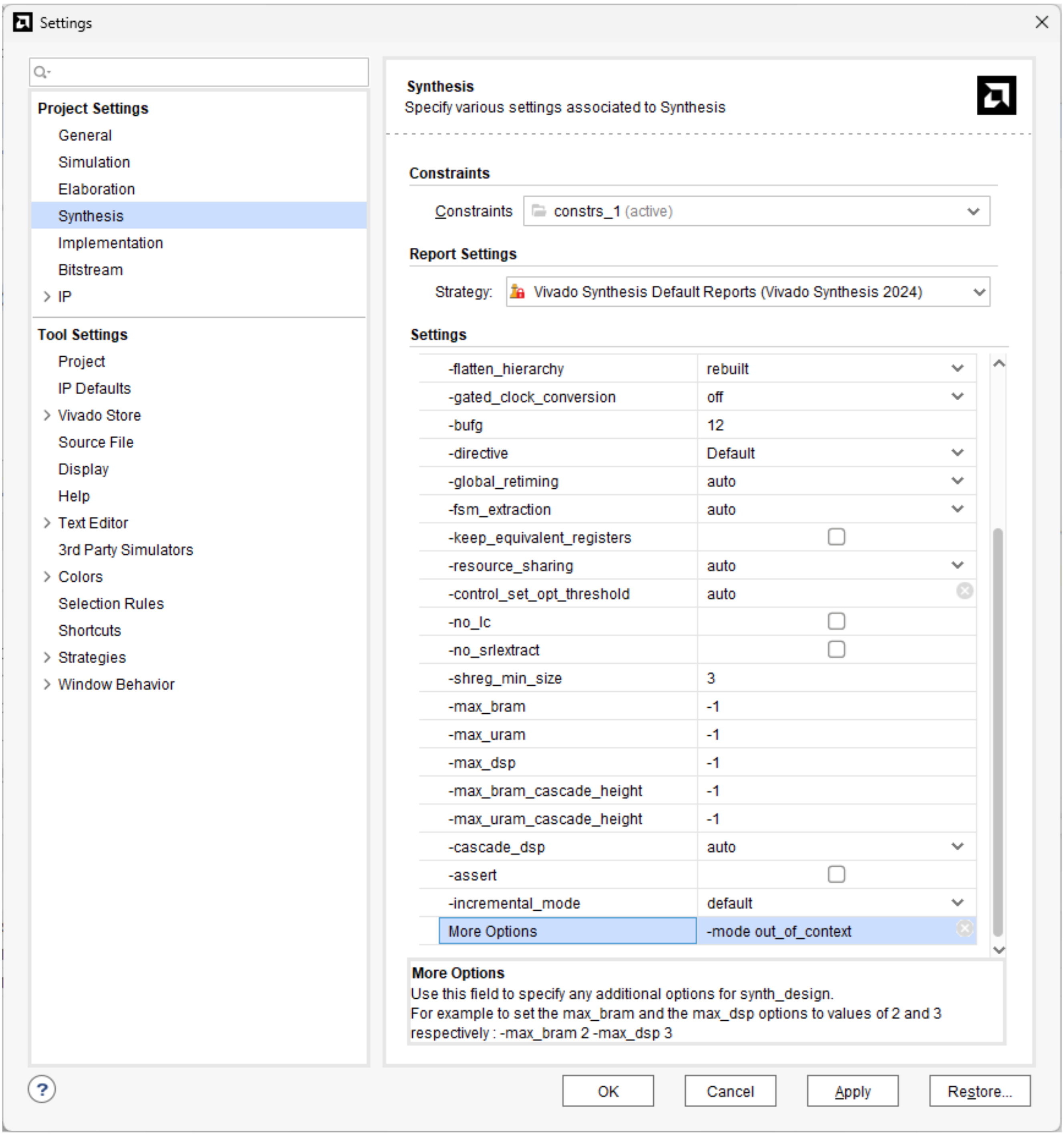
To manually run a bottom-up flow, instantiate a lower-level netlist or third-party netlist as a black box, and the Vivado tools fit that black box into the full design after synthesis completes. The following sections describe the process.

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

Creating a Lower-Level Netlist

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

Figure: Settings Dialog Box



Synthesis

Specify various settings associated to Synthesis

Constraints

Constraints: constrs_1 (active)

Report Settings

Strategy: Vivado Synthesis Default Reports (Vivado Synthesis 2024)

Settings

-flatten_hierarchy	rebuilt
-gated_clock_conversion	off
-bufg	12
-directive	Default
-global_retiming	auto
-fsm_extraction	auto
-keep_equivalent_registers	☐
-resource_sharing	auto
-control_set_opt_threshold	auto
-no_lc	☐
-no_srlextract	☐
-shreg_min_size	3
-max_bram	-1
-max_uram	-1
-max_dsp	-1
-max_bram_cascade_height	-1
-max_uram_cascade_height	-1
-cascade_dsp	auto
-assert	☐
-incremental_mode	default
More Options	-mode out_of_context

More Options

Use this field to specify any additional options for synth_design.
For example to set the max_bram and the max_dsp options to values of 2 and 3 respectively : -max_bram 2 -max_dsp 3

In the More Options section, you can type `-mode out_of_context` to have the tool not insert any I/O buffers in this level.

After you run synthesis, open the synthesized design and in the Tcl Console, type the `write_edif` Tcl command in the Tcl Console. The syntax is as follows:

```
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 by providing a description of the port in a lower-level to the Vivado tool. In the Setting a Bottom-Up, Out-of-Context Flow, this is referred to as a stub file.

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

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

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

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

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

Putting Together the Manual Bottom-Up Components

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

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

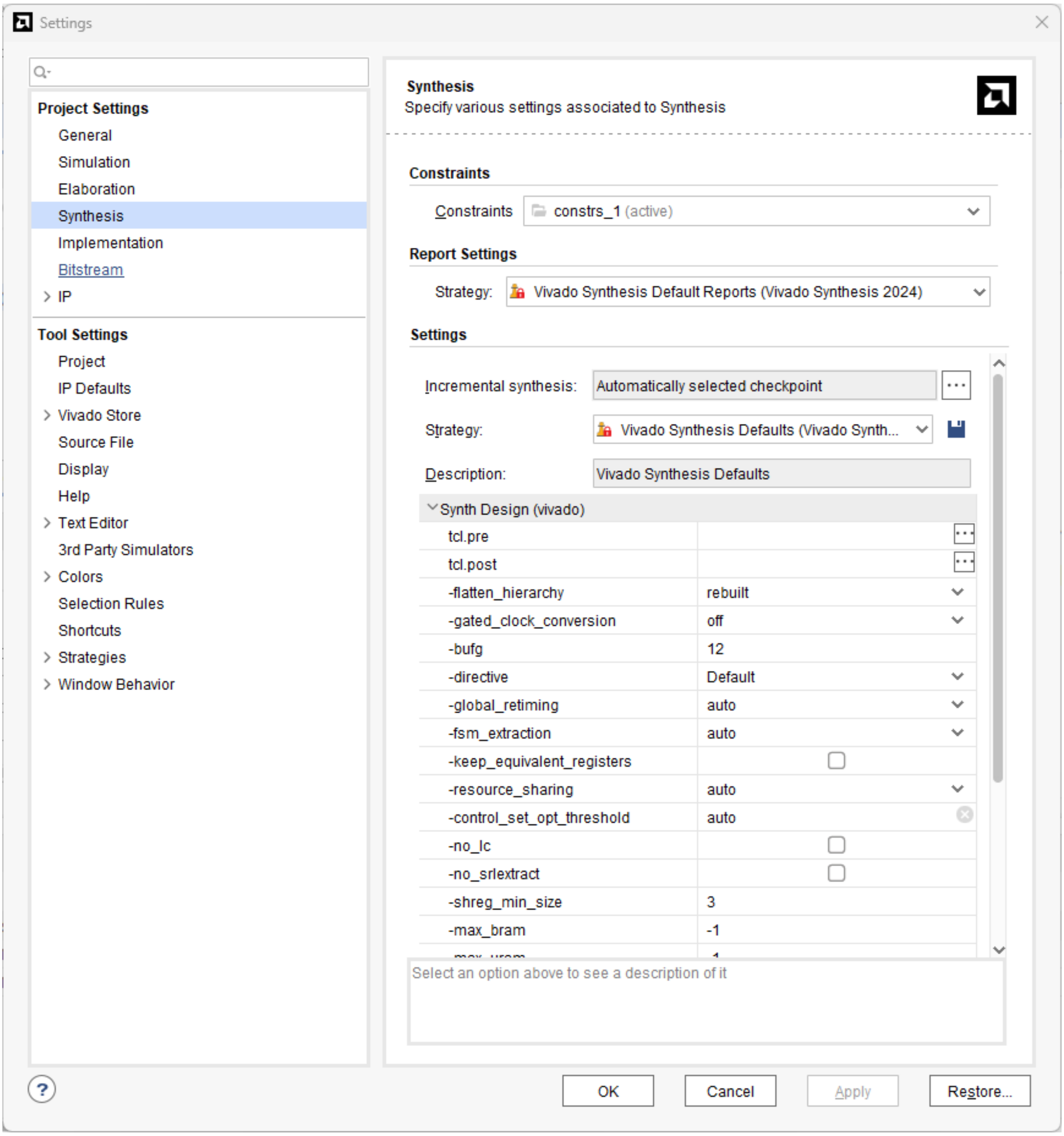
Incremental Synthesis

Vivado Synthesis can be run incrementally. In this flow, the tool puts incremental synthesis info in the generated DCP file that can be referenced in later runs. It detects when the design has changed and only re-runs synthesis on sections of the design that have changed. The key advantage of this flow is that the runtime is significantly reduced for designs with small changes. In addition, the QoR of the design fluctuates less when small changes are inserted into the RTL.

Setting Up Incremental Synthesis in Project Mode

You can set up Incremental Synthesis with a project in the Synthesis page of the Settings dialog box.

Figure: Synthesis Page of Settings Dialog Box



Synthesis

Specify various settings associated to Synthesis

Constraints

Constraints: constrs_1 (active)

Report Settings

Strategy: Vivado Synthesis Default Reports (Vivado Synthesis 2024)

Settings

Incremental synthesis: Automatically selected checkpoint

Strategy: Vivado Synthesis Defaults (Vivado Synth...

Description: Vivado Synthesis Defaults

Synth Design (vivado)

tcl.pre	
tcl.post	
-flatten_hierarchy	rebuilt
-gated_clock_conversion	off
-bufg	12
-directive	Default
-global_retiming	auto
-fsm_extraction	auto
-keep_equivalent_registers	☐
-resource_sharing	auto
-control_set_opt_threshold	auto
-no_lc	☐
-no_srlextract	☐
-shreg_min_size	3
-max_bram	-1

Select an option above to see a description of it

Note the following important settings:

Incremental Synthesis selection box

Use the Browse button to indicate if incremental synthesis uses a known checkpoint, the last checkpoint created (default), or if incremental synthesis is disabled.

incremental_mode Synth Design option

Describes how aggressive synthesis is with optimizations across partitions. The values are quick, default, aggressive, and off. Quick turns off most optimizations, aggressive turns on more and repeat synthesis on certain sections. Off tells synthesis not to use the incremental synthesis information in the DCP file.

Using Incremental Synthesis in Non-Project Mode

In project mode, the tool automatically reads in the last DCP file from when synthesis was the last run if running in default mode or any DCP that is specifically mentioned. In non-project mode, the reference DCP must be read before synthesis. The command for that is:

```
read_checkpoint -auto_incremental -incremental<path to dcp file>
```

Or

```
read_checkpoint -incremental <path to dcp file>
```

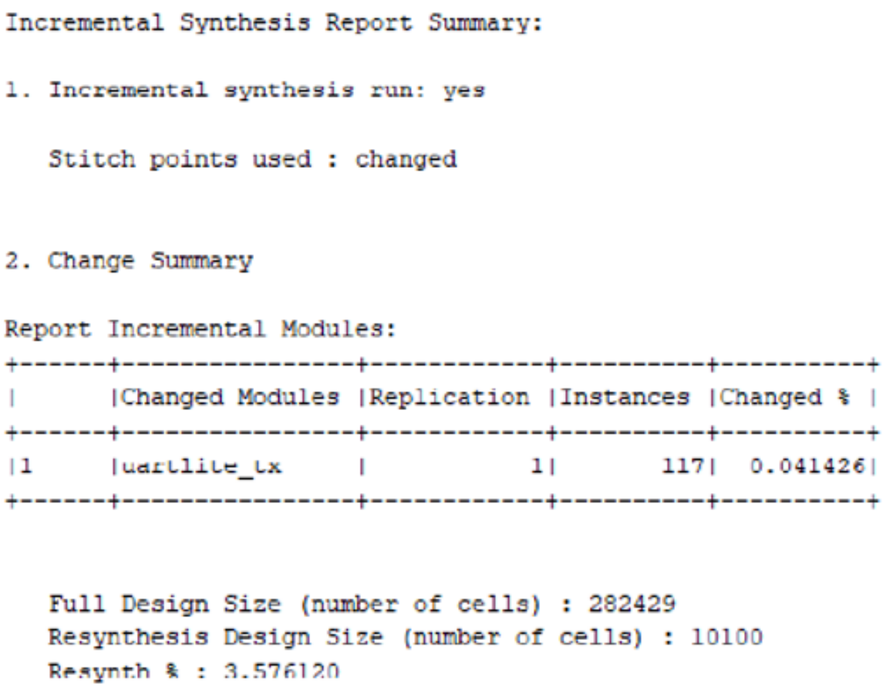
After this, run the `synth_design` command as normal.

✎ **Note:** The `-auto_incremental` option in `read_checkpoint` is the same as the default behavior in the IDE.

Interpreting the Log File

When the reference run is performed, the tool partitions out the design as it is performing synthesis. When the incremental run is started, it compares the elaborated design with the reference run and identifies the changed modules. The partitioning from the reference run is initialized in the incremental run. The partitions that contain the changed modules and the partitions that are affected by them are marked. Based on this, the tool performs synthesis on the marked partitions. The information on how much of the design and what parts of the design were re-synthesized can be found in the log file after the incremental run.
This information is in the “Incremental Synthesis Report Summary.” The following is an example of the report.

Figure: Incremental Synthesis Report Summary



```
Incremental Synthesis Report Summary:

1. Incremental synthesis run: yes

   Stitch points used : changed


2. Change Summary

Report Incremental Modules:
+------+----------------+------------+----------+----------+
|      |Changed Modules |Replication |Instances |Changed % |
+------+----------------+------------+----------+----------+
|1     |uartlite_tx     |           1|       117|  0.041426|
+------+----------------+------------+----------+----------+


   Full Design Size (number of cells) : 282429
   Resynthesis Design Size (number of cells) : 10100
   Resynth % : 3.576120
```

This section has information on which sections of the design changed and needed to be re-synthesized. In addition, it also has information on how much of the design changed from reference run to incremental run.

Re-Synthesizing the Full Design

There are some cases or types of designs that cause the flow to trigger a full re-synthesis of the design. These instances occur under the following conditions:

1. When changes to the top level of the hierarchy are made
2. When the Synthesis settings change
3. When small designs contain few partitions
4. When more than 50% of the partitions have a change

In addition, unusually large XDC files can trigger a re-synthesis of the full design. This improves in future releases.

✎ **Note:** Even though it is a Synthesis setting, `-mode out_of_context` does not trigger a full re-synthesis.

Using Third-Party Synthesis Tools with Vivado IP

The Vivado IP catalog is designed, constrained, and validated with the Vivado Design Suite synthesis.

✎ **Note:** Even though this is a synthesis setting, `-mode out_of_context` does not trigger a full resynthesis.

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

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

1. Create the IP customization in a managed IP project.
2. Generate the output products for the IP, including the synthesis design checkpoint (DCP).
   The Vivado IDE creates a stub HDL file, which is used in third-party synthesis tools to infer a black box for the IP (_stub.v | _stub.vhd). The stub file contains directives to prevent I/O buffers from being inferred; you might need to modify these files to support other synthesis tool directives.
3. Synthesize the design with the stub files for the AMD IP.
4. Use the netlist produced by the third-party synthesis tool and the DCP files for the AMD IP, run Vivado implementation. For more information, see *Vivado Design Suite User Guide: Designing with IP* (UG896).

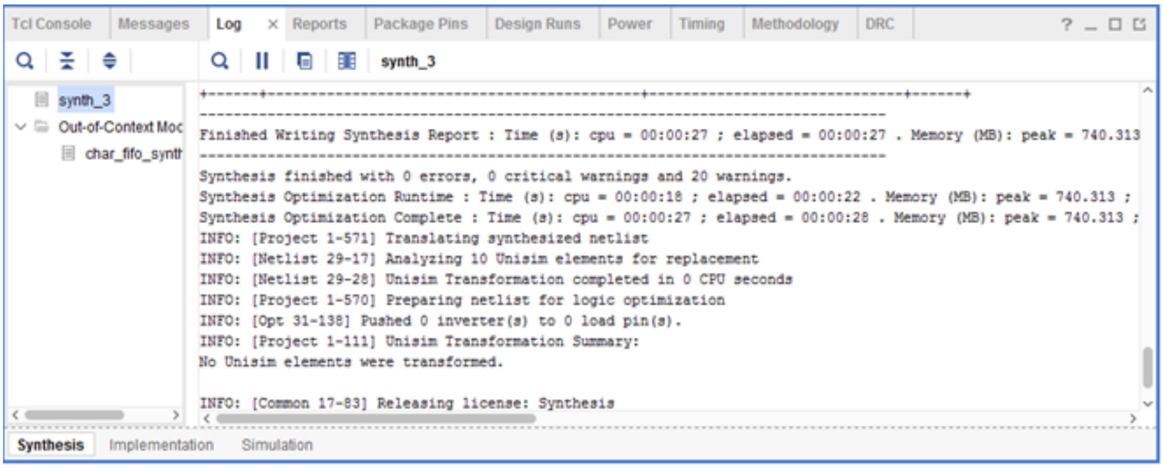
Moving Processes to the Background

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

Monitoring the Synthesis Run

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

Figure: Log Window



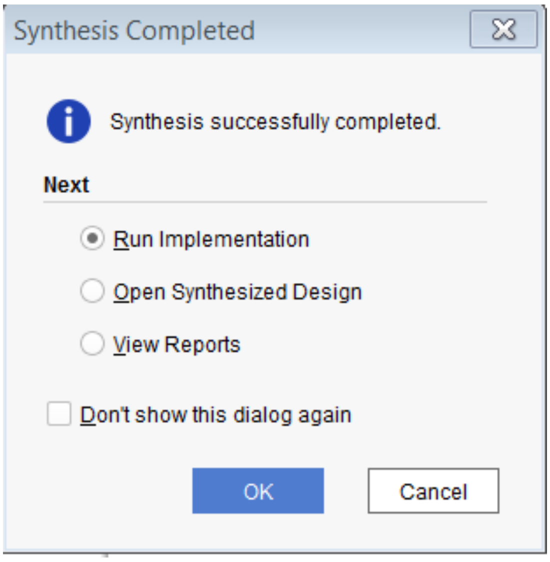
```
+------+----------------------------------------------+-------------------------------+------+
------------------------------------------------------------------------------------------------
Finished Writing Synthesis Report : Time (s): cpu = 00:00:27 ; elapsed = 00:00:27 . Memory (MB): peak = 740.313
------------------------------------------------------------------------------------------------
Synthesis finished with 0 errors, 0 critical warnings and 20 warnings.
Synthesis Optimization Runtime : Time (s): cpu = 00:00:18 ; elapsed = 00:00:22 . Memory (MB): peak = 740.313 ;
Synthesis Optimization Complete : Time (s): cpu = 00:00:27 ; elapsed = 00:00:28 . Memory (MB): peak = 740.313 ;
INFO: [Project 1-571] Translating synthesized netlist
INFO: [Netlist 29-17] Analyzing 10 Unisim elements for replacement
INFO: [Netlist 29-28] Unisim Transformation completed in 0 CPU seconds
INFO: [Project 1-570] Preparing netlist for logic optimization
INFO: [Opt 31-138] Pushed 0 inverter(s) to 0 load pin(s).
INFO: [Project 1-111] Unisim Transformation Summary:
No Unisim elements were transformed.

INFO: [Common 17-83] Releasing license: Synthesis
```

Flow After Synthesis Completion

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

Figure: Synthesis Completed Dialog Box



Synthesis Completed

Synthesis successfully completed.

Next

● Run Implementation

○ Open Synthesized Design

○ View Reports

☐ Don't show this dialog again

Select one of the options:

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 a Synthesized Design environment so that you can perform I/O pin planning, design analysis, and floorplanning.

View Reports

Opens the Reports window so you can view reports.

Don’t show this dialog again

Use the checkbox to stop this dialog box display.

★ **Tip:** You can revert to having the dialog box present by selecting Tools > Settings > Window Behavior.

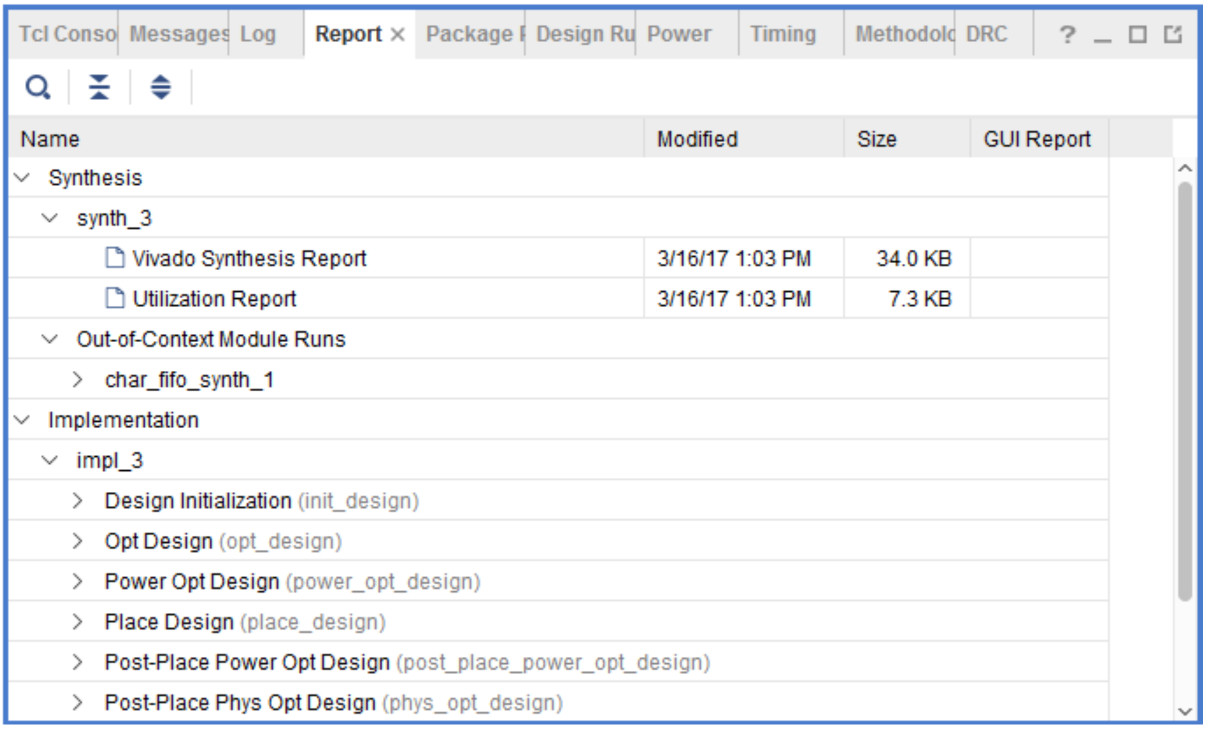
Analyzing Synthesis Results

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

✎ **Video:** See the following for more information: Vivado Design Suite QuickTake Video: Advanced Synthesis using Vivado.

Open the Reports view, shown in the following figure, and select a report for a specific run.

Figure: Synthesis Reports View



Name	Modified	Size	GUI Report
Synthesis			
synth_3			
Vivado Synthesis Report	3/16/17 1:03 PM	34.0 KB	
Utilization Report	3/16/17 1:03 PM	7.3 KB	
Out-of-Context Module Runs			
char_fifo_synth_1			
Implementation			
impl_3			
Design Initialization (init_design)			
Opt Design (opt_design)			
Power Opt Design (power_opt_design)			
Place Design (place_design)			
Post-Place Power Opt Design (post_place_power_opt_design)			
Post-Place Phys Opt Design (phys_opt_design)			

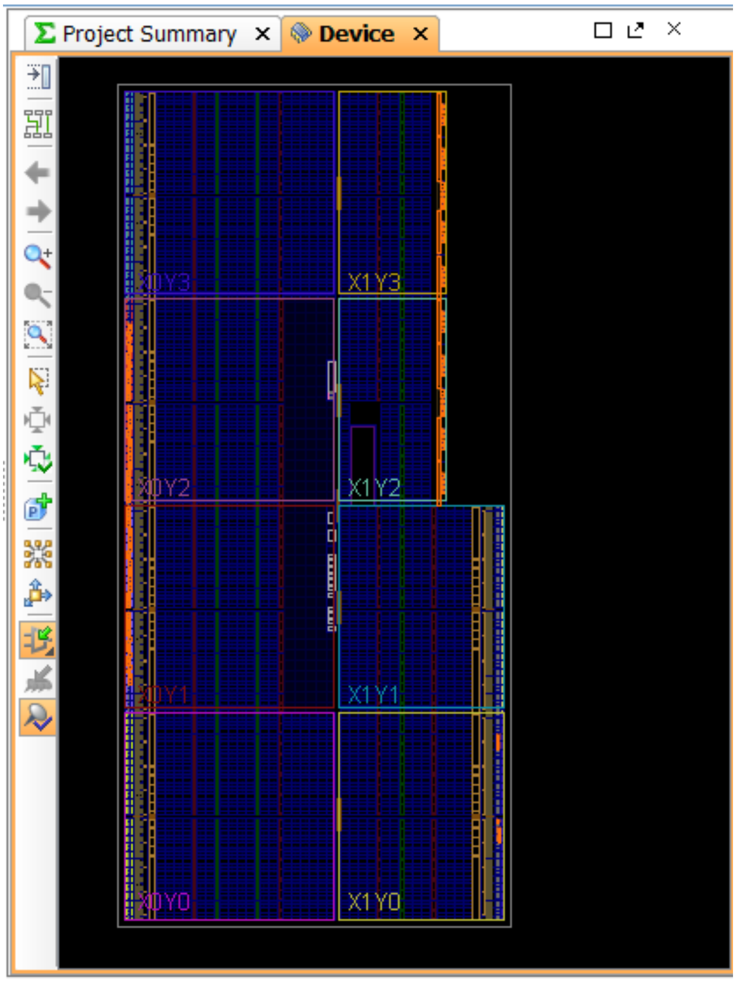
Using the Synthesized Design Environment

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

To open a synthesized design, select Open Synthesized Design from the Flow Navigator or the Flow menu.

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

Figure: Device Window



From this perspective, you can examine the design logic and hierarchy, view the resource usage and timing estimates, or run design rule checks (DRCs). For more information, see the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)

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.

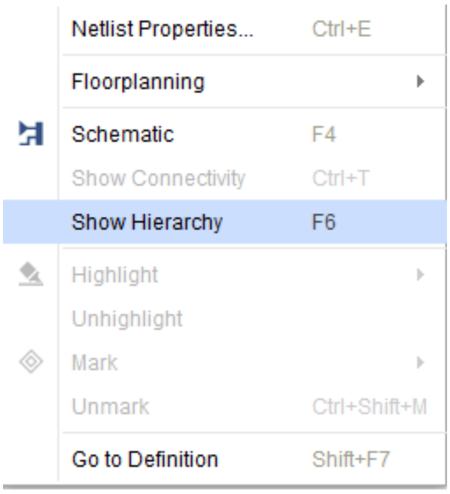
Exploring the Logic Hierarchy

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

Figure: Show Hierarchy Option



Exploring the Logical Schematic

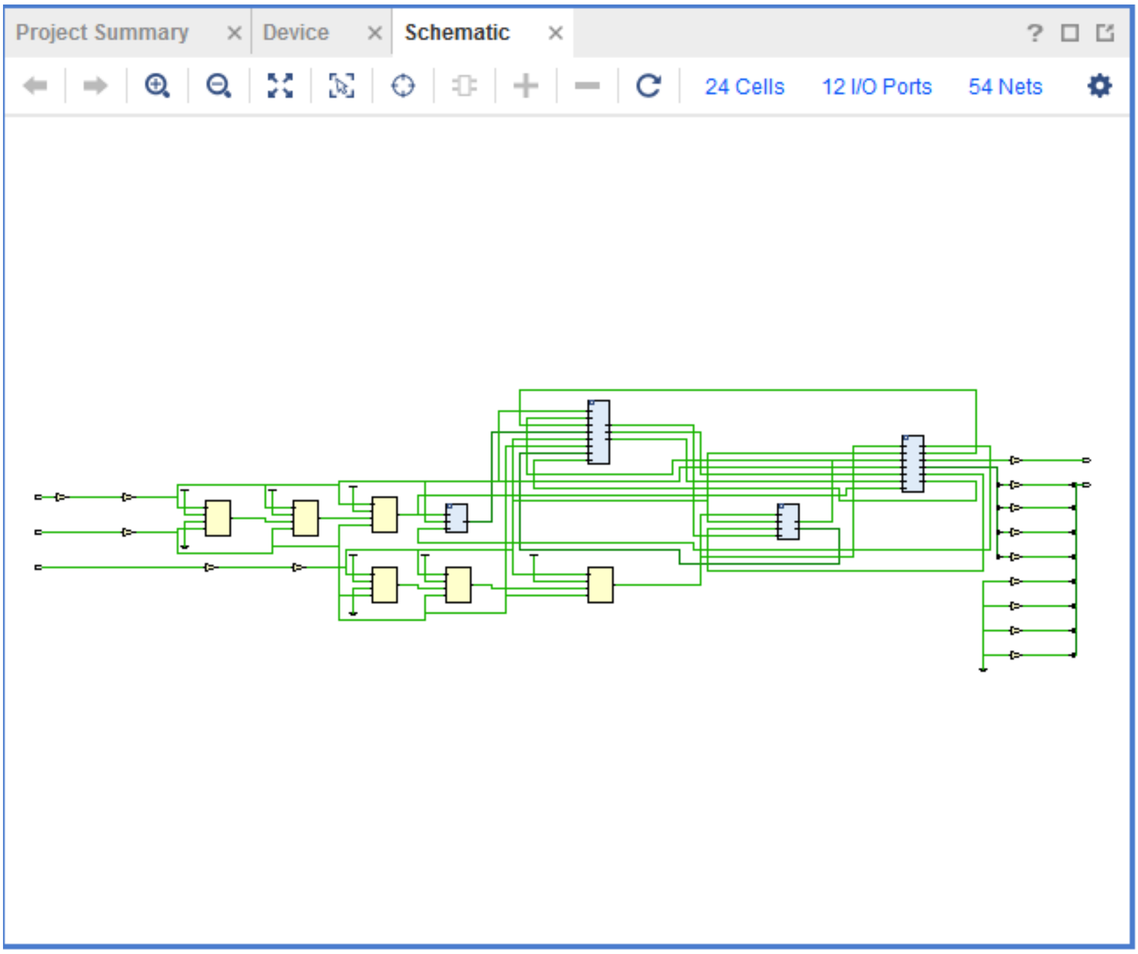
The Schematic window allows selective expansion and exploration of the logical design. You must select at least one logic object to open and display the Schematic window.

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, select one or more instances, nets, or timing paths, and 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: Schematic Window



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

Running Timing Analysis

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

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

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

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

Running Synthesis with Tcl

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

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

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

```
Description: Synthesize a design using Vivado Synthesis and open that
design

Syntax: synth_design                      [-name <arg>] [-part <arg>] [-
constrset <arg>] [-top <arg>]
                                          [-include_dirs <args>] [-generic
<args>]
                                          [-verilog_define <args>]
                                          [-flatten_hierarchy <arg>]
                                          [-gated_clock_conversion <arg>]
                                          [-directive <arg>] [-rtl] [-bufg
<arg>] [-no_lc]
                                          [-shreg_min_size <arg>] [-mode
<arg>]
                                          [-fsm_extraction <arg>][-
rtl_skip_mlo][-rtl_skip_ip]
                                          [-rtl_skip_constraints]
                                          [-keep_equivalent_registers] [-
resource_sharing <arg>]
                                          [-cascade_dsp <arg>] [-
control_set_opt_threshold <arg>]
                                          [-max_bram <arg>] [-max_uram <arg>]
                                          [-max_dsp <arg>] [-
max_bram_cascade_height <arg>]
                                          [-max_uram_cascade_height <arg>] [-
global_retiming]
                                          [-no_srlextract]
                                          [-assert] [-no_timing_driven] [-
sfcu] [-debug_log]
                                          [-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:
                                          zull, 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,
AreaOptimized_high,
                                          AreaOptimized_medium,
AlternateRoutability,
                                          AreaMapLargeShiftRegToBRAM,
                                          AreaMultThresholdDSP,
FewerCarryChains.
                                          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.
[-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,
user_encoding, auto.
                                          Default: auto
[-rtl_skip_mlo]                           Skip mandatory logic
optimization for RTL elaboration
                                          of the design; requires -rtl
option.
[-rtl_skip_ip]                            Exclude subdesign checkpoints
in the RTL elaboration of
                                          the design; requires -rtl
option.
[-rtl_skip_constraints]                   Do not load and validate
constraints against elaborated
                                          design; requires -rtl option.
[-srl_style]                              Static SRL Implementation
Style. Values: register,
                                          rl, srl_reg, reg_srl,
reg_srl_reg.
[-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
[-cascade_dsp]                            Controls how adders summing DSP
block outputs will be
                                          implemented. Value: auto, tree,
force.
                                          Default: auto
[-control_set_opt_threshold]              Threshold for synchronous
control set optimization to
                                          lower number of control sets.
Valid values are 'auto'
                                          and non-negative integers. The
higher the number, the
                                          more control set optimization
will be performed and
                                          fewer control sets will result.
To disable control set
                                          optimization completely, set to
0.
                                          Default: auto
[-max_bram]                               Maximum number of block RAM
allowed in design. (Note -1
                                          means that the tool will choose
the max number allowed
                                          for the part in question).
                                          Default: -1
[-max_uram]                               Maximum number of UltraRAM
blocks 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 in question).
                                          Default: -1
[-max_bram_cascade_height]                Controls the maximum number of
BRAM that can be
                                          cascaded by the tool. (Note -1
means that the tool will
                                          choose
                                          the max number allowed for the
part in question).
                                          Default: -1
[-max_uram_cascade_height]                Controls the maximum number of
UtraRAM that can be
                                          cascaded by the tool. (Note -1
means that the tool will
                                          choose the max number allowed
for the part in
                                          question).
                                          Default:1
[-global_retiming]                        Seeks to improve circuit
performance for intra-clock
                                          sequential paths by
automatically moving registers
                                          (register balancing) across
combinatorial gates or
                                          LUTs.It maintains the original
behavior and latency of
                                          the circuit and does not
require changes to the RTL
                                          sources.A value of "on" turns
on retiming, "off" turns
                                          off retiming and "auto" will
allow the tool to decide.
```

```
                                          "auto" will have retiming turns
  on for Vesal devices
                                          and off for non-Versal devices.
  Default: "auto"
  [-no_srlextract]                        Prevents the extraction of
  shift registers so that they
                                          get implemented as simple
  registers.
  [-assert]                               Enable VHDL assert statements
  to be evaluated. A
                                          severity level of failure will
  stop the synthesis flow
                                          and produce an error.
  [-no_timing_driven]                     Do not run in timing driven
  mode.
  [-sfcu]                                 Run in single-file compilation
  unit mode.
  [-debug_log]                            Print detailed log files for
  debugging.
  [-quiet]                                Ignore command errors.
  [-verbose]                              Suspend message limits during
  command
```

For the `-generic` option, special handling needs to happen with VHDL boolean and `std_logic_vector` types 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:

```
-generic my_gen=1‘b0
```

For `std_logic_vector`, the value for `0010` is:

```
-generic my_gen=4‘b0010
```

✎ **Note:** Overriding string generics or parameters is not supported.

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

A verbose version of the help is available in the *Vivado Design Suite Tcl Command Reference Guide* (UG835). 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

The following table shows the supported Tcl commands for Vivado timing constraints. The commands are linked to more information to the full description in the *Vivado Design Suite Tcl Command Reference Guide* (UG835).

Table: Supported Synthesis Tcl Commands

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

For details on these commands, see the following documents:

- *Vivado Design Suite Tcl Command Reference Guide* (UG835)
- *Vivado Design Suite User Guide: Using Constraints* (UG903)
- *Vivado Design Suite Tutorial: Using Constraints* (UG945)
- *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)

Multi-Threading in RTL Synthesis

On multiprocessor systems, RTL synthesis leverages multiple CPU cores by default (up to four) to speed up compile times.
The maximum number of simultaneous threads varies, depending on the number of processors available on the system, the OS, and the stage of the flow (see *Vivado*

Design Suite User Guide: Implementation (UG904)).

The `general.maxThreads` Tcl parameter, which is common to all threads in Vivado, gives you control to specify the number of threads to use when running RTL synthesis. For example:

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

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

Vivado Preconfigured Strategies

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

Table: Vivado Preconfigured Settings

Options\Strategies	Default	Flow_AreaOptimized_high	Flow_AreaOptimized_medium	Flow_AreaMultThresholdDSP	Flow_AlternateRoutability	Flow_PerfOptimized_high	Flow_PerfThresholdCarry	Flow_RuntimeOptimized
-flatten_hierarchy	rebuilt	rebuilt	rebuilt	rebuilt	rebuilt	rebuilt	rebuilt	none
-gated_clock_conversion	off	off	off	off	off	off	off	off
-bufg	12	12	12	12	12	12	12	12
-directive	Default	AreaOptimized_high	AreaOptimized_medium	AreaMultThresholdDSP	AlternateRoutability	PerformanceOptimized	FewerCarryChains	RunTimeOptimized
-retiming	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked
-fsm_extraction	auto	auto	auto	auto	auto	one_hot	auto	off
-keep_equivalent_registers	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	checked	unchecked
-resource_sharing	auto	auto	auto	auto	auto	off	off	auto
-control_set_opt_threshold	auto	1	1	auto	auto	auto	auto	auto
-no_lc	unchecked	unchecked	unchecked	unchecked	checked	checked	checked	unchecked
-no_srlextract	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked

Options\Strategies	Default	Flow_AreaOptimized_high	Flow_AreaOptimized_medium	Flow_AreaMultThresholdDSP	Flow_AlternateRoutability	Flow_PerfOptimized_high	Flow_PerfThresholdCarry	Flow_RuntimeOptimized
-shreg_min_size	3	3	3	3	10	5	3	3
-max_bram	-1	-1	-1	-1	-1	-1	-1	-1
-max_uram	-1	-1	-1	-1	-1	-1	-1	-1
-max_dsp	-1	-1	-1	-1	-1	-1	-1	-1
-max_b_cascade_height	-1	-1	-1	-1	-1	-1	-1	-1
-max_u_cascade_height	-1	-1	-1	-1	-1	-1	-1	-1
-cascade_dsp	auto	auto	auto	auto	auto	auto	auto	auto
-assert	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked	unchecked

Synthesis Attributes

Introduction

In the AMD Vivado™ Design Suite, Vivado synthesis can synthesize attributes of several types. In most cases, these attributes have the same syntax and behavior.

- If Vivado synthesis supports the attribute. It uses the attribute and creates a 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. Still, 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).

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

‼ **Important:** Care should be taken when putting this attribute on loadless signals. The attribute and signal might not be preserved. Attributes are case-insensitive, regardless of HDL.

ASYNC_REG Verilog Example

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

ASYNC_REG VHDL Examples

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

BLACK_BOX

The `BLACK_BOX` attribute is a useful debugging attribute directs 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.

CASCADE_HEIGHT

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

Often, the tool chooses to cascade the block RAMs that it creates. This attribute can be used to shorten the length of the chain. Place the attribute on the RAM in question, and you can place the attribute in the RTL files. A value of 0 or 1 for this attribute effectively turns off any cascading of block RAMs.

✎ **Note:** This attribute is only applicable to AMD UltraScale™ and AMD Versal™ architecture block RAMs and URAMs (UltraRAMs).

More information on `CASCADE_HEIGHT` attributes for UltraRAM is available in CASCADE_HEIGHT.

CASCADE_HEIGHT Verilog example

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

CASCADE_HEIGHT VHDL example

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

CLOCK_BUFFER_TYPE

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

By default, Vivado synthesis uses BUFGs for clock 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 in the RTL and 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;
```

CLOCK_BUFFER_TYPE XDC Example

```
set_property CLOCK_BUFFER_TYPE BUFG [get_ports clk]
```

CRITICAL_SIG_OPT

`CRITICAL_SIG_OPT` is used to optimize sequential loops by restructuring logic in the feedback path, so that timing-critical signals travel through the smallest number of logic levels. The attribute should be placed on sequential objects such as registers, that drive critical paths to their own inputs.

The optimization improves critical path timing, but at the expense of increased logic utilization as it involves Shannon decomposition. You must mark the sequential elements (registers) with a loop which have reasonable logic levels. It can cause resource overhead due to logic replication.

CRITICAL_SIG_OPT Verilog Example

```
(* CRITICAL_SIG_OPT = “true” *)  reg [3 : 0]  signal_name;
```

CRITICAL_SIG_OPT VHDL Example

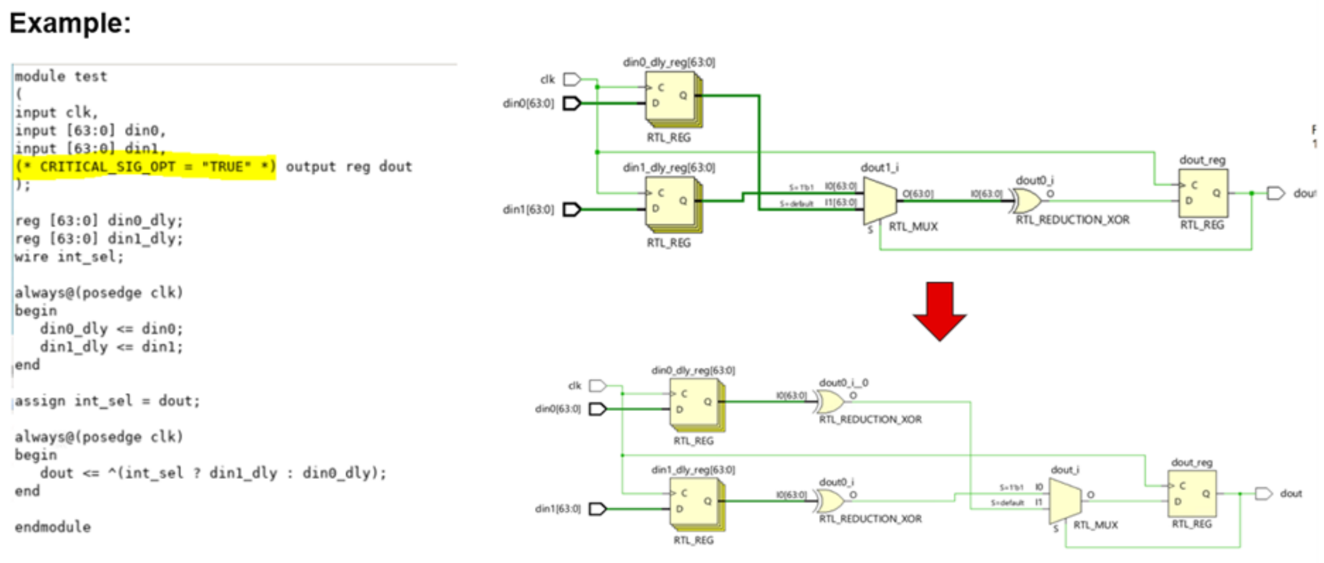
```
attribute CRITICAL_SIG_OPT: string;

attribute CRITICAL_SIG_OPT of signal_name: signal is "true"
```

CRITICAL_SIG_OPT XDC Example

```
set_property CRITICAL_SIG_OPT 1 [get_cells <registers>]
```

Figure: Example



Example:

```
module test
(
input clk,
input [63:0] din0,
input [63:0] din1,
(* CRITICAL_SIG_OPT = "TRUE" *) output reg dout
);

reg [63:0] din0_dly;
reg [63:0] din1_dly;
wire int_sel;

always@(posedge clk)
begin
   din0_dly <= din0;
   din1_dly <= din1;
end

assign int_sel = dout;

always@(posedge clk)
begin
   dout <= ^(int_sel ? din1_dly : din0_dly);
end

endmodule
```

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_ENABLE XDC Example

```
set_property direct_enable yes [get_nets -of [get_ports ena3]]
```

✎ **Note:** For XDC usage, this attribute only works on type net, so you must use the `get_nets` command for the object.

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;
```

DIRECT_RESET XDC Example

```
set_property direct_reset yes [get_nets -of [get_ports rst3]]
```

✎ **Note:**
For XDC usage, this attribute only works on type net, so you need to use the `get_nets` command for the object.

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 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.

In general, the `DONT_TOUCH` attribute should be set in RTL only. Signals that need to be kept can often be optimized before the XDC file is read. Therefore, setting this attribute in the RTL ensures that it is used. There is one use case where it is recommended that `DONT_TOUCH` is set in the XDC file. This would be when `DONT_TOUCH` is set to yes in the RTL, and it is desired to be taken out without having to change the RTL. In this case, setting `DONT_TOUCH` to no in XDC when that same signal has `DONT_TOUCH` set to yes in RTL effectively removes that attribute without having to change the RTL.

✎ **Note:** When using the XDC to remove a `DONT_TOUCH` that is set in RTL, you can end up getting warnings after synthesis when the implementation flow reads the same XDC but the signal in question has been optimized out. These warnings can be ignored. However, you can also bypass them by putting the `DONT_TOUCH` attributes in an XDC file marked as for synthesis only.

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

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

DSP_FOLDING

The `DSP_FOLDING` attribute controls whether the Vivado synthesis folds two MAC structures connected with an adder into one DSP primitive.
The values for `DSP_FOLDING` are:

yes

The tool converts MAC structures.

no

The tool does not convert MAC structures.

`DSP_FOLDING` is supported in RTL only. It should be placed on the module/entity/architecture of the logic that contains the MAC structures.

DSP_FOLDING Verilog Example

```
(* dsp_folding = "yes" *) module top .....
```

DSP_FOLDING VHDL Example

```
attribute dsp_folding : string;
attribute dsp_folding of my_entity : entity is "yes";
```

DSP_FOLDING_FASTCLOCK

The `DSP_FOLDING_FASTCLOCK` attribute tells the tool which port should become the new faster clock when using DSP folding.
The values for `DSP_FOLDING_FASTCLOCK` are:

yes
The tool uses this port to connect to the new clock.

no
The tool does not use this port.

`DSP_FOLDING_FASTCLOCK` is supported in RTL only. Place this attribute only on a port or a pin.

DSP_FOLDING_FASTCLOCK Verilog Example

```
(* dsp_folding_fastclock = "yes" *) input clk_fast;
```

DSP_FOLDING_FASTCLOCK VHDL Example

```
attribute dsp_folding_fastclock : string;
attribute dsp_folding_fastclock of clk_fast : signal is "yes";
```

EXTRACT_ENABLE

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

specified in the RTL, this attribute can tell the tool to move that enable to the CE pin of the flip-flop.
EXTRACT_ENABLE is placed on the registers and is supported in RTL and XDC. It can take boolean values of: yes and no.

EXTRACT_ENABLE Verilog Example

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

EXTRACT_ENABLE VHDL Example

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

EXTRACT_ENABLE XDC Example

```
set_property EXTRACT_ENABLE yes [get_cells my_reg]
```

EXTRACT_RESET

EXTRACT_RESET controls if registers infer resets. Typically, the Vivado tools extract or not extract resets based on heuristics that typically benefit the most designs. In cases where Vivado is not behaving in a desired way, this attribute overrides the default behavior of the tool. If an undesired synchronous reset goes to the flip-flop, this attribute can force it to the D input logic. Conversely, if the tool is not inferring a reset specified in the RTL, this attribute can command the tool to move that reset to the dedicated reset of the flop. This attribute can only be used with synchronous resets; asynchronous resets are not supported.
EXTRACT_RESET is placed on the registers and supported in the RTL and XDC. It can take the boolean values: yes or no. A value of no means that the reset does not go to the R pin of the register but is routed through logic to the D pin of the register. A value of yes means that the reset goes directly to the R pin of the register.

EXTRACT_RESET Verilog Example

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

EXTRACT_RESET VHDL Example

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

EXTRACT_RESET XDC Example

```
set_property EXTRACT_RESET yes [get_cells my_reg]
```

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 statemachine registers. The legal values for this are `one_hot`, `sequential`, `johnson`, `gray`, `user_encoding`, and none. The `auto` value is the default, and allows the tool to determine best encoding. The `user_encoding` value tells the tool to still infer a statemachine, but to use the encoding given in the RTL by the user.

The `FSM_ENCODING` attribute can be set in the RTL or the XDC.

FSM_ENCODING Verilog Example

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

FSM_ENCODING VHDL Example

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

FSM_SAFE_STATE

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

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

The legal values for `FSM_SAFE_STATE` are:

auto_safe_state

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. For this to work, a `default` or `others` state must be in the RTL.

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 Verilog Example

```
(* 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 IDE that instructs the tool to attempt the conversion.
- The `GATED_CLOCK` RTL attribute or XDC property 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 Tools > Settings > Project Settings > Synthesis.
2. In the Options area, set the `-gated_clock_conversion` option to one of the following values:

   off

   Disables the gated clock conversion.

   on

   Gated clock conversion occurs if the `gated_clock` attribute is set in the RTL code. This option gives you more control of the outcome.

   auto

   Gated clock conversion occurs if either of the following events are true:

   - The `gated_clock` attribute is set to YES.
   - The Vivado synthesis can detect the gate and there is a valid clock constraint set. This option lets the tool make decisions.

⚠ **CAUTION!** Care should be taken when using attributes like KEEP_HIERARCHY, DONT_TOUCH, and MARK_DEBUG. These attributes can interfere with gated clock conversion if placed on hierarchies or instances that need to change to support the conversion.

GATED_CLOCK Verilog Example

```
(* gated_clock = "yes" *) 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 "yes";
end test;
```

GATED_CLOCK XDC Example

```
set_property GATED_CLOCK yes [get_ports clk]
```

IOB

The IOB attribute controls 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 only in the RTL.

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. This attribute is only supported, and can only be set in the RTL.

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.

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

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

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

⚠ **CAUTION!** Take care when using KEEP attribute on loadless signals. Synthesis keeps those signals resulting in issues later in the flow.

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 Vivado synthesis to optimize. 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 hierarchy specified in the RTL, but for better Quality of Results (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 be set in the RTL and in XDC. If it is used in the XDC, it can only be put on the instance.

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 Architecture

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

KEEP_HIERARCHY XDC Example

On Instance

```
set_property keep_hierarchy yes [get_cells u0]
```

MARK_DEBUG

This attribute is applicable to net objects. 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 on the signal in question:

```
(* 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 on the signal in question. 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>]
```

XDC Syntax Example

Often, the use of `MARK_DEBUG` is on pins of hierarchies and can be used on any elaborated sequential element, such as RTL_REG. `MARK_DEBUG` attributes are intended to go on nets. It is recommended that you use both the `get_nets` and the `get_pins` command as shown, such as.

```
set_property MARK_DEBUG true [get_nets -of [get_pins\ hier1/hier2/
<flop_name>/Q]]
```

This recommended use ensures that the `MARK_DEBUG` goes onto the net connected to that pin, regardless of its name.

✎ **Note:** If a `MARK_DEBUG` is applied on a bit of a signal that was declared as a bit_vector, the whole bus gets the `MARK_DEBUG` attribute. In addition, if a `MARK_DEBUG` is placed on a pin of a hierarchy, the full hierarchy is kept.

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 that indicates what the fanout should be. A value of -1 tells the tool not to perform any replication.
This attribute only works on registers and combinatorial signals.

✎ **Note:** It replicates the register or the driver that drives the combinatorial signal to achieve the fan outputs, black boxes, EDIF files, and Native Generic Circuit (NGC) files are not supported with this attribute.

✎ **Recommended:** Using MAX_FANOUT attributes on global high fanout signals leads to sub-optimal replication in synthesis. For this reason, AMD recommends only using MAX_FANOUT inside the hierarchies on local signals with medium to low fanout.

MAX_FANOUT Verilog Example

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.

MAX_FANOUT XDC Examples

```
set_property MAX_FANOUT <value> [get_cells in1_int_reg]
```

```
set_property MAX_FANOUT <value> [get_nets top/my_net]
```

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
```

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

RAM_DECOMP

The `RAM_DECOMP` attribute instructs the tool to infer RTL RAMs that are too large to fit in a single block RAM primitive to use a more customizable configuration.
For example, a RAM specified as 2K x 36 would often be configured as two 2 K x 18 block RAMs arranged side by side. This is the configuration that yields the fastest design. By setting `RAM_DECOMP = "power"`, the RAM would instead be configured as two 1K x 36 block RAMs. This is more power-friendly because during a read or write, only the one RAM with the address being used is active. It comes at the cost of timing because Vivado synthesis must use address decoding. The `RAM_DECOMP` would force the second configuration of that RAM.
Alternatively, a value of *area* would force the configuration to be as small of an area as possible. This could also be a change from the fastest design.
The values accepted for `RAM_DECOMP` are "`power` and `area`".
This attribute can be set in either RTL or XDC. Place the attribute on the RAM instance itself.

RAM_DECOMP Verilog Example

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

RAM_DECOMP VHDL Example

```
attribute ram_decomp : string;
attribute ram_decomp of myram : signal is "power";
```

RAM_DECOMP XDC Example

```
set_property ram_decomp power [get_cells myram]
```

RAM_STYLE

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

block
: Instructs the tool to infer RAMB-type components.

distributed
: Instructs the tool to infer the LUT RAMs.

registers
: Instructs the tool to infer registers instead of RAMs.

ultra
: Instructs the tool to use the AMD UltraScale+™ URAM primitives.

mixed
: Instructs the tool to infer a combination of RAM types designed to minimize the amount of space that is unused.

auto
: Lets the synthesis tool decide how to implement the RAM. This value is mainly used by XPMs that must choose a value for `RAM_STYLE`. This is the same as the default behavior. That must choose a value for `RAM_STYLE`.

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

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

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.

RETIMING_BACKWARD

The `RETIMING_BACKWARD` attribute instructs the tool to move a register backward through logic closer to the sequential driving elements. Unlike the retiming global setting, this attribute is not timing-driven and works regardless of whether the retiming global setting is active or if there are even timing constraints. If the global retiming setting is active, the `RETIMING_BACKWARD` step happens first, and the global retiming can enhance that register to move further back the chain. However, it does not interfere with the attribute and moves the register back to the original location.

✎ **Note:** Cells with `DONT_TOUCH/MARK_DEBUG` attributes, cells with timing exceptions (`false_path, multicycle_path`), and user-instantiated cells block this attribute.

The `RETIMING_BACKWARD` attribute takes an integer as a value. This value describes the amount of logic the register is allowed to cross. Larger values allow the register to cross more logic. 0 would turn the attribute off.

RETIMING_BACKWARD Verilog Example

```
(*retiming_backward = 1 *) reg my_sig;
```

RETIMING_BACKWARD VHDL Example

```
attribute retiming_backward : integer;
attribute retiming_backward of my_sig : signal is 1;
```

RETIMING_BACKWARD XDC Example

```
set_property retiming_backward 1 [get_cells my_sig];
```

RETIMING_FORWARD

The `RETIMING_FORWARD` attribute instructs the tool to move a register forward through logic closer to the driven sequential elements. Unlike the retiming global setting, this attribute is not timing-driven and works regardless of whether the retiming global setting is active or if there are even timing constraints. If the global retiming setting is active, the `RETIMING_FORWARD` step happens first, and the global retiming can enhance that register to move further up the chain. However, it does not interfere with the attribute and moves the register back to the original location.

✎ **Note:** Cells with `DONT_TOUCH/MARK_DEBUG` attributes, cells with timing exceptions (`false_path, multicycle_path`), and user-instantiated cells block this attribute.

The `RETIMING_FORWARD` attribute takes an integer as a value. This value describes the amount of logic the register is allowed to cross. Larger values allow the register to cross more logic. 0 would turn the attribute off.

RETIMING_FORWARD Verilog Example

```
(* retiming_forward = 1 *) reg my_sig;
```

RETIMING_FORWARD VHDL Example

```
attribute retiming_forward : integer;
attribute retiming_forward of my_sig : signal is 1;
```

RETIMING_FORWARD XDC Example

```
set_property retiming_forward 1 [get_cells my_sig];
```

ROM_STYLE

`ROM_STYLE` instructs the synthesis tool on how to infer constant arrays into memory structures like Block RAMs. Accepted values are:

block
: Instructs the tool to infer RAMB-type components

distributed
: Instructs the tool to infer the LUT ROMs. Instructs the tool to infer constant arrays into distributed RAM (LUTRAM) resources. By default, the tool selects which ROM to infer based on heuristics that give the best results for the most designs.

ultra
: Instructs synthesis to use URAM primitives. (AMD Versal™ adaptive SoC parts only).

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.

RW_ADDR_COLLISION

The `RW_ADDR_COLLISION` attribute is for specific types of RAMs. When RAM is a simple dual port, and the read address is registered, Vivado synthesis infers a block RAM and sets the write mode to `WRITE_FIRST` for best timing. Also, if a design writes to the same address it is reading from, the RAM output is unpredictable. `RW_ADDR_COLLISION` overrides this behavior.
The values for `RW_ADDR_COLLISION` are:

auto
The default behavior as described previously.

yes
These inserts bypass logic so that when an address is read from the same time it is written to, the value of the input is seen on the output making the whole array behave as WRITE_FIRST.

no
This is when you do not care about timing or the collision possibility. In this case, the write mode is set to `NO_CHANGE` , resulting in power savings.

`RW_ADDR_COLLISION` is supported in RTL only.

RW_ADDR_COLLISION Verilog Example

```
(*rw_addr_collision = "yes" *) reg [3:0] my_ram [1023:0];
```

RW_ADDR_COLLISION VHDL Example

```
attribute rw_addr_collision : string;
attribute rw_addr_collision of my_ram : signal is "yes";
```

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 instructs the synthesis tool on how to infer SRLs 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 after the SRL.

block
The tool infers the SRL inside a block RAM.

Place SRL_STYLE on the signal declared for SRL. This attribute can be set in RTL and XDC. The 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.

✎ **Note:** Use care when using combinations of SRL_STYLE, SHREG_EXTRACT, and `-shreg_min_size`. The SHREG_EXTRACT attribute always takes precedence over the others. If SHREG_EXTRACT is set to "`no`" and SRL_STYLE is set to "`srl`", registers are used. The `-shreg_min_size`, being the global variable, always has the least amount of precedence. If an SRL of length 10 is set and SRL_STYLE is set to "`srl`", and `-shreg_min_size` is set to 20, the SRL is still inferred.

✎ **Note:** In the following examples, the SRLs are all created with buses where the SRL shifts from one bit to the next. If the code to use SRL_STYLE has many differently named signals driving each other, place the SRL_STYLE attribute on the last signal in the chain. This includes if the last register in the chain is in a different hierarchy level than the other registers. The attribute always goes on the last register in the chain.

SRL_STYLE Verilog Example

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

SRL_STYLE VHDL Example

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

SRL_STYLE XDC Example

```
set_property srl_style register [get_cells my_shifter_reg*]
```

TRANSLATE_OFF/TRANSLATE_ON OFF/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 can start with one of the following keywords:

- synthesis
- synopsy
- pragma
- xilinx

In newer versions of the tool, using a keyword has become optional. The tool works with translate_off/on or off/on in the comment.
`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 OFF/ON Verilog Example

```
// synthesis translate_off
Code....
// synthesis translate_on
// synthesis off
Code....
// synthesis on
```

TRANSLATE_OFF/TRANSLATE_ON OFF/ON VHDL Example

```
-- synthesis translate_off
Code...
-- synthesis translate_on
-- synthesis on
Code....
-- synthesis off
```

⚠ **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_DSP

`USE_DSP` instructs the synthesis tool how to deal with synthesis arithmetic structures. By default, unless there are timing concerns or threshold limits, synthesis attempts to infer `mults`, `mult-add`, `mult-sub`, and `mult-accumulate` type structures into DSP blocks.
Adders, subtracters, and accumulators can go into these blocks also, but by default are implemented with the logic instead of with DSP blocks. The `USE_DSP` attribute overrides the default behavior and force these structures into DSP blocks.
Accepted values are: "`logic`" , "`simd`" , "`yes`" , and "`no`" :

- The "`logic`" value is used specifically for XOR structures to go into the DSP primitives. For "`logic`" , this attribute can be placed on the module/architecture level only.
- The "`simd`" is used to instruct the tool to put SIMD structures (Single-instruction-multiple-data) into DSPs. See the templates for examples.
- The "`yes`" and "`no`" values instruct the tool to either put the logic into a DSP or not. These values can be placed in the RTL on signals, architecture, components, entities, and modules. The priority is:
  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_DSP Verilog Example

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

USE_DSP VHDL Example

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

Custom Attribute Support in Vivado

Vivado synthesis supports the use of *custom attributes* in RTL. Behavior synthesis of a custom attribute is unknown. Often, custom attributes are intended for use in other tools downstream from the synthesis process.

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

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

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

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

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

Example with Custom Attribute on Hierarchy (Verilog)

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

Example with Custom Attribute on Hierarchy (VHDL)

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

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

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

Example with Custom Attribute on a Signal (Verilog)

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

Example with Custom Attribute on a Signal (VHDL)

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

Using Synthesis Attributes in XDC files

Some synthesis attributes can also be set from an XDC file as well as the original RTL file. In general, attributes that are used in the end stages of synthesis and describe how synthesis- created logic is allowed in the XDC file. Attributes that are used towards the beginning of synthesis and affect the compiler are not allowed in the XDC.
For example, the KEEP and `DONT_TOUCH` attributes are not allowed in the XDC.
This is because, at the time the attribute is read from the XDC file, components that have the KEEP or `DONT_TOUCH` attribute might have already been optimized and would therefore not exist at the time the attribute is read. For that reason, those attributes must always be set in the RTL code. For more information on where to set specific attributes, see the individual attribute descriptions in this chapter.

✎ **Note:** At the time the XDC file is read, multi-bit signals exist as single nodes in Synthesis. Because of this, putting attributes on individual bits of a vector signal puts that attribute on all bits of the signal.

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

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

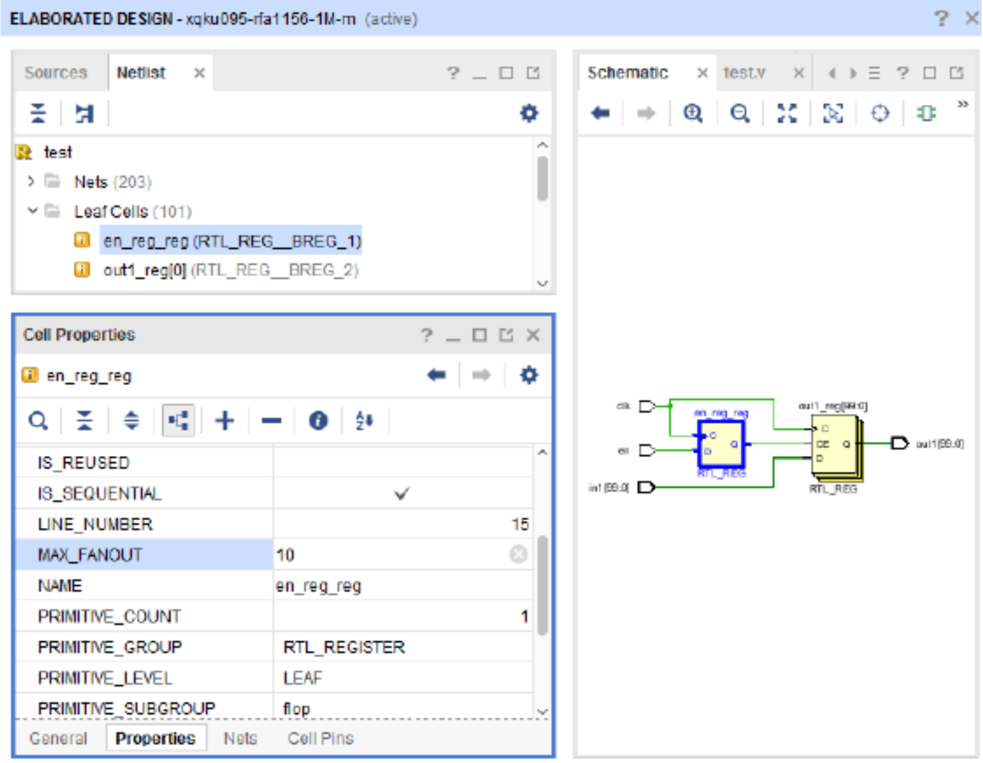
For example:

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

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

1. Open the elaborated design, shown in the following figure, 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, as shown in the following figure.



ELABORATED DESIGN - xqku095-rfa1156-1M-m (active)

test
- Nets (203)
- Leaf Cells (101)
    - en_reg_reg (RTL_REG__BREG_1)
    - out1_reg[0] (RTL_REG__BREG_2)

Cell Properties

en_reg_reg

IS_REUSED	
IS_SEQUENTIAL	✓
LINE_NUMBER	15
MAX_FANOUT	10
NAME	en_reg_reg
PRIMITIVE_COUNT	1
PRIMITIVE_GROUP	RTL_REGISTER
PRIMITIVE_LEVEL	LEAF
PRIMITIVE_SUBGROUP	flop

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.

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

✎ **Note:** If the same attribute is put on the same object in both the XDC and in RTL, but the values of the attributes are different, the XDC attribute is accepted, and the RTL attribute is ignored.

Synthesis Attribute Propagation Rules

Read each individual attribute for the rules on whether it should be placed on hierarchies or signals.
Generally, when an attribute is placed on a hierarchy, it affects only that boundary and not the items inside that hierarchy. For example, placing a `DONT_TOUCH` on a specific level affects that level only, and not the signals inside that level.
There are some exceptions to this rule. These are `DSP_FOLDING` , `RAM_STYLE`, `ROM_STYLE`, `SHREG_EXTRACT`, and `USE_DSP`. When these attributes are placed on a hierarchy, they also affect the signals inside that hierarchy.

✎ **Note:** For the Verilog syntax of having the attribute inside block comments, /* attr = value */, this attribute is attached to the next lexical item after the comment. If the comment is on its own line, the next item in the RTL, no matter how far down, gets the attribute. If the attribute is specified at the end of the file, the attribute gets attached to the module.

Using Block Synthesis Strategies

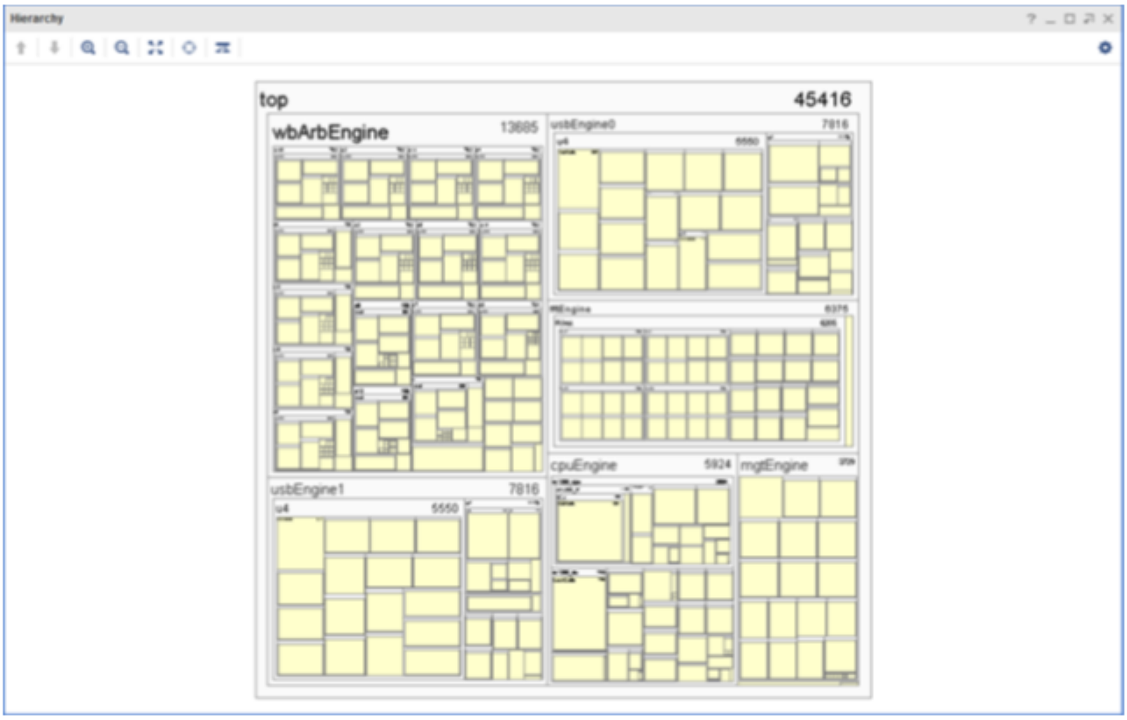
Overview

AMD Vivado™ synthesis comes with many strategies and global settings that you can use to customize how your design is synthesized. This Figure shows the available, pre-defined strategies in the Synthesis Settings, and Vivado Preconfigured Strategies provides a side-by-side comparison of the strategy settings.

You can override certain settings, such as `-retiming`, using attributes or XDC files in the RTL or XDC files for specific hierarchies or signals. However, in general, options affect the whole design.

As designs become more complex, the application of such settings can limit your design from reaching its full potential. Certain hierarchies in a design might work better with different options than others. The following figure shows a medium-sized design that has many different types of hierarchy.

Figure: Multiple Hierarchies within a Design



One option is to synthesize such hierarchies in out of context (OOC) mode; this is effective, but complicates the design flow. The OOC flow separates the hierarchies that are assigned to be synthesized in OOC mode, and runs them separately from the other parts of the design. This means that synthesis runs more than one time per design. Also, the OOC constraints must be separated from the constraints of the rest of the design, adding even more complexity.

The Block-Level Synthesis flow (`BLOCK_SYNTH`) uses a property that lets you use certain global settings and strategies on specific levels of hierarchy in a top-down flow that is differs from the top level of the full design.

Setting a Block-Level Flow

To set a Block-Level Synthesis flow (using the BLOCK_SYNTH property), you enter a Tcl property in the XDC file only. The command syntax is, as follows:

```
set_property BLOCK_SYNTH.<option name> <value> [get_cells
<instance_name>]
```

Where:

- <option_name> is the option that you want to set.
- <value> is the value you assign to that option.
- <instance_name> is the hierarchical instance on which to set the option.

For example:

```
set_property BLOCK_SYNTH.MAX_LUT_INPUT 4 [get_cells fftEngine]
```

Set the property to an instance name, and *not* on an entity or module name. By using instance names, the Vivado synthesis tool is able to have more flexibility when there are modules/entities that are instantiated multiple times. In the provided example, the fftEngine instance is being set, so there are no LUT5 or LUT6 primitives.

✎ **Note:** By setting a `BLOCK_SYNTH` on an instance, you affect that instance and everything below that instance. For example, if `fftEngine` had other modules instantiated within it, those modules would also not have any LUT5s or LUT6s primitives.

✎ **Note:** In addition to affecting this instance, the BLOCK_SYNTH property also causes the hierarchy of this instance to be hardened. Be careful with this, especially if this hierarchy contains I/O buffers or is inferring input/output buffers.

When you put a `BLOCK_SYNTH` property on an instance, the instance gets that value for that specific option; all other options use the default values.
Multiple `BLOCK_SYNTH` properties can be set on the same instance to try out different combinations. For example, the following keeps equivalent registers, disables the FSM inference, and uses the `AlternateRoutability` strategy:

```
set_property BLOCK_SYNTH.STRATEGY {ALTERNATE_ROUTABILITY} [get_cells
mod_inst]
set_property BLOCK_SYNTH.KEEP_EQUIVALENT_REGISTER 1 [get_cells mod_inst]
set_property BLOCK_SYNTH.FSM_EXTRACTION {OFF} [get_cells mod_inst]
```

To prevent impacting instances under the instance that require a different property setting, you can nest `BLOCK_SYNTH` properties on multiple levels. If you only want this on one particular level, you can set it on that level, and on the subsequent levels, you can set the default values back, using the command as follows:

```
set_property BLOCK_SYNTH.MAX_LUT_INPUT 6 [get_cells fftEngine/newlevel]
```

If the new level is the only hierarchy under `fftEngine`, this command ensures that only `fftEngine` gets the `MAX_LUT_INPUT = 4` property. You can also put an entirely different set of options on this level as well, and not go back to the default.

✎ **Note:** When performing the block level flow, the tool keeps this design in a top-down mode meaning that the full design goes through synthesis. For the instance in question, Vivado synthesis preserves the hierarchy to ensure that the logic of that level does not blur and stays within that level. This could have a potential effect on QoR. For this reason, be careful when setting `BLOCK_LEVEL` properties. Only set them on instances you know need them.

Block-Level Flow Options

The block-level flow supports some of the predefined strategies that are in the tool as well. The strategies that are allowed are: `DEFAULT`, `AREA_OPTIMIZED`, `ALTERNATE_ROUTABILITY`, and `PERFORMANCE_OPTIMIZED`. The XDC constraint syntax is as follows:

```
set_property BLOCK_SYNTH.STRATEGY {<value>} [get_cells <inst_name>]
```

The following table lists the supported Vivado Block synthesis settings.

Table: Vivado Block Synthesis Settings

Option	Type	Values	Description
RETIMING	INTEGER	0/1	• 0 – Disable Retiming • 1 – Enable Retiming
ADDER_THRESHOLD	INTEGER	4-128	Changes the threshold for the size of an adder for synthesis to infer in a CARRY chain.

Option	Type	Values	Description
			- Higher numbers mean more LUTs. - Lower numbers mean more CARRY chains. The threshold is calculated by adding the sizes of the adder operands. The specified value should be >= sum of the input widths.
COMPARATOR_THRESHOLD	INTEGER	4-128	Changes the threshold for the size of a comparator for synthesis to infer in a CARRY chain. - Higher numbers mean more LUTs. - Lower numbers mean more CARRY chains.
SHREG_MIN_SIZE	INTEGER	3-32	Changes the threshold for the size of a register chain before synthesis infers SRL primitives.

Option	Type	Values	Description
			- Higher numbers mean more registers. - Lower numbers mean more SRLs.
FSM_EXTRACTION	STRING	OFF ONE_HOT SEQUENTIAL GRAY JOHNSON AUTO	Sets the encodings of state machines that the synthesis tool infers.
LUT_COMBINING	INTEGER	0/1	- 0 – Disable LUT combining - 1 – Enable LUT combining
CONTROL_SET_THRESHOLD	INTEGER	0-128	Controls the fanout needed on control signals before synthesis infers registers with control signals.

Option	Type	Values	Description
			- Higher numbers mean less logic on control signals and more on D input of flop. - Lower numbers mean more control signals and less logic on D input.
MAX_LUT_INPUT	INTEGER	4-6	- 4 – No LUT5 or LUT6 primitives are inferred - 5 – No LUT6 primitives are inferred - 6 – All LUTs can be inferred.
MUXF_MAPPING	INTEGER	0/1	- 0 – Disable MUXF7/F8/F9 inference - 1 – Enable MUXF7/F8/F9 inference
KEEP_EQUIVALENT_REGISTER	INTEGER	0/1	

Option	Type	Values	Description
			- 0 – Merges equivalent registers - 1 – Retains equivalent registers
PRESERVE_BOUNDARY	INTEGER	Any number	This option can be used with incremental synthesis. It is used to mark hierarchies that are known to change. Using this option can make the hierarchy static and allow the incremental flow to work. The value given does not matter because having this option set is sufficient.
LOGIC_COMPACTION	INTEGER	1	Arranges CARRY chains and LUTs in such a way that it makes the logic more compact using fewer SLICES.
SRL_STYLE	STRING	REGISTER SRL SRL_REG REG_SRL REG_SRL_REG	Sets the default implementation for inferred SRLs.

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 AMD devices.
- Templates are available from the AMD Vivado™ Design Suite Integrated Design Environment (IDE). To access the templates, in the Window Menu, select Language Templates.

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

Advantages of VHDL

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

Advantages of Verilog

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

Advantages of SystemVerilog

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

Flip-Flops, Registers, and Latches

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

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

Flip-Flops, Registers, and Latches are described with:

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

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

- The clock signal
- All asynchronous control signals

Flip-Flops and Registers Control Signals

Flip-Flops and Registers control signals include:

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

Coding Guidelines

- Do not asynchronously set or reset registers.
  - Control set remapping becomes impossible.
  - Sequential functionality in device resources, such as block RAM components and DSP blocks, can be set or reset synchronously only.
  - If you use asynchronously set or reset registers, you cannot leverage device resources or 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 can adversely affect area and performance.
- Avoid operational set/reset logic whenever possible. There can 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 penalizes circuit performance.

Flip-Flops and Registers Inference

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

FDCE
: D flip-flop with Clock Enable and Asynchronous Clear

FDPE
: D flip-flop with Clock Enable and Asynchronous Preset

FDSE
: D flip-flop with Clock Enable and Synchronous Set

FDRE
: D flip-flop with Clock Enable and Synchronous Reset

Flip-Flops and Registers Initialization

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

Flip-Flops and Registers Reporting

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

Flip-Flops and Registers Reporting Example

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

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

Flip-Flops and Registers Coding Examples

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

Register with Rising-Edge Coding Verilog Example

Filename: registers_1.v

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

module registers_1(d_in,ce,clk,clr,dout);
input [7:0] d_in;
input ce;
input clk;
input clr;
output [7:0] dout;
reg [7:0] d_reg;
always @ (posedge clk)
begin
if(clr)
d_reg <= 8'b0;
else if(ce)
d_reg <= d_in;
end
assign dout = d_reg;
endmodule
```

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

Filename: registers_1.vhd

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

library IEEE;
use IEEE.std_logic_1164.all;

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

Latches

The Vivado log file reports the type and size of recognized Latches.
Inferred Latches are often the result of HDL coding mistakes, such as incomplete if or case statements.
Vivado synthesis issues a warning for the instance shown in the following reporting example. This warning lets you verify that the inferred Latch functionality was intended.

Latches Reporting Example

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

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

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

Latch With Positive Gate and Asynchronous Reset Coding Verilog Example

Filename: latches.v

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

endmodule
```

Latch With Positive Gate and Asynchronous Reset Coding VHDL Example

Filename: latches.vhd

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

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

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

Tristates

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

Tristate Implementation

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

- An external pin of the circuit (OBUFT)
- An Internal bus (BUFT):
  - An inferred BUFT is converted automatically to logic realized in LUTs by Vivado synthesis.
  - When an internal bus inferring a BUFT is driving an output of the top module, the Vivado synthesis infers an OBUF.

Tristate Reporting Example

Tristate buffers are inferred and reported during synthesis.

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

Tristate Description Using Concurrent Assignment Coding Verilog Example

Filename: tristates_2.v

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

```
input T, I;
output O;
assign O = (~T) ? I: 1'bZ;
endmodule
```

Tristate Description Using Combinatorial Process Implemented with OBUFT Coding VHDL Example

Filename: tristates_1.vhd

```
-- Tristate Description Using Combinatorial Process
-- Implemented with an OBUFT (IO buffer)
-- File: tristates_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;
entity tristates_1 is
port(
T : in std_logic;
I : in std_logic;
O : out std_logic
);
end tristates_1;
architecture archi of tristates_1 is
begin
process(I, T)
begin
if (T = '0') then
O <= I;
else
O <= 'Z';
end if;
end process;
end archi;
```

Tristate Description Using Combinatorial Always Block Coding Verilog Example

Filename: tristates_1.v

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

```
reg O;

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

endmodule
```

Shift Registers

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

Static Shift Register Elements

A static Shift Register usually involves:

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

Shift Registers SRL-Based Implementation

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

- SRL16E
- SRLC32E

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

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

Shift Registers Coding Examples

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

32-Bit Shift Register Coding Example One (VHDL)

This coding example uses the concatenation coding style.

Filename: shift_registers_0.vhd

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

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

end shift_registers_0;

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

32-Bit Shift Register Coding Example Two (VHDL)

The same functionality can also be described as follows:

Filename: shift_registers_1.vhd

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

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

end shift_registers_1;

architecture archi of shift_registers_1 is
signal shreg : std_logic_vector(DEPTH - 1 downto 0);
begin
process(clk)
begin
if rising_edge(clk) then
if clken = '1' then
for i in 0 to DEPTH - 2 loop
shreg(i + 1) <= shreg(i);
end loop;
shreg(0) <= SI;
end if;
end if;
end process;
SO <= shreg(DEPTH - 1);
end archi;
```

8-Bit Shift Register Coding Example One (Verilog)

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

Filename: shift_registers_0.v

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

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

reg [WIDTH-1:0] shreg;

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

assign SO = shreg[WIDTH-1];

endmodule
```

32-Bit Shift Register Coding Example Two (Verilog)

Filename: shift_registers_1.v

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

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

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

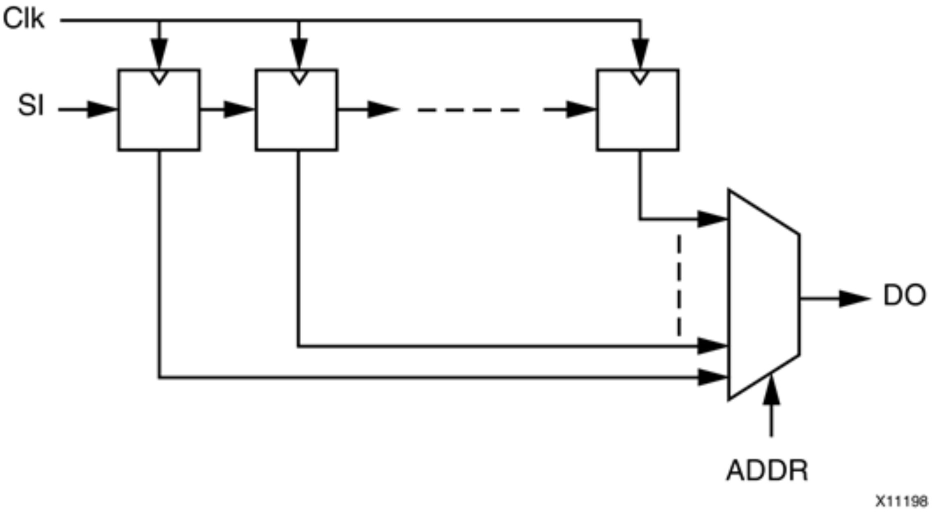
Dynamic Shift Registers

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. The following figure illustrates the functionality of the Dynamic Shift register.

Figure: Dynamic Shift Registers Diagram



Dynamic Shift Registers Coding Examples

Download the coding example files from Coding Examples.

32-Bit Dynamic Shift Registers Coding Verilog Example

Filename: dynamic_shift_registers_1.v

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

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

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

assign DO = data[SEL];

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

32-Bit Dynamic Shift Registers Coding VHDL Example

Filename: dynamic_shift_registers_1.vd

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

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

end dynamic_shift_register_1;

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

signal SRL_SIG : SRL_ARRAY;

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

DO <= SRL_SIG(conv_integer(A));

end rtl;
```

Multipliers

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

✎ **Recommended:** If you do not intend to use all most significant bits of a device, AMD 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 the `USE_DSP` 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, and after the multiplication.
When a Multiplier does not fit on a single DSP block, Vivado synthesis decomposes the macro to implement it. In that case, Vivado synthesis uses either of the following:

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

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

Multipliers Coding Examples

Unsigned 16x24-Bit Multiplier Coding Verilog Example

Filename: mult_unsigned.v

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

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

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

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

assign RES = M[3];

endmodule
```

Unsigned 16x16-Bit Multiplier Coding VHDL Example

Filename: mult_unsigned.vhd

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

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

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

Multiply-Add and Multiply-Accumulate

The following macros are inferred:

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

The macros are inferred by aggregation of:

- A Multiplier
- An Adder/Subtractor
- Registers

Multiply-Add and Multiply-Accumulate Implementation

During Multiply-Add and Multiply-Accumulate implementation:

- Vivado synthesis can implement an inferred Multiply-Add or Multiply-Accumulate macro on DSP block resources.
- Vivado synthesis attempts to take advantage of the pipelining capabilities of DSP blocks.
- Vivado synthesis pulls up to:
  - Two register stages are 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 DSP block if its implementation requires only a single DSP resource.
- If the macro exceeds the limits of a single DSP, Vivado synthesis does the following:
  - Processes it as two separate Multiplier and Accumulate macros.
  - Makes independent decisions on each macro.

Macro Implementation on DSP Block Resources

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

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

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

Complex Multiplier Examples

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

Complex Multiplier Verilog Example

Fully pipelined complex multiplier using three DSP blocks.

Filename: cmult.v

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

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

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

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

// Common factor (ar ai) x bi, shared for the calculations of the real
and imaginary final products
//
always @(posedge clk)
begin
addcommon <= ar_d - ai_d;
mult0 <= addcommon * bi_dd;
common <= mult0;
end

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

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

```
pi_int <= multi + commonr2;
end

assign pr = pr_int;
assign pi = pi_int;

endmodule // cmult
```

Complex Multiplier Examples (VHDL)

Fully pipelined complex multiplier using three DSP blocks.

Filename: cmult.vhd

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

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

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

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

begin
process(clk)
begin
```

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

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

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

-- Imaginary product
--
process(clk)
begin
if rising_edge(clk) then
ai_ddd <= ai_dd;
ai_dddd <= ai_ddd;
```

```
addi <= resize(br_ddd, BWIDTH + 1) + resize(bi_ddd, BWIDTH + 1);
multi <= addi * ai_dddd;
commonr2 <= common;
pi_int <= multi + commonr2;
end if;
end process;

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

end rtl;
```

Pre-Adders in the DSP Block

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

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

Filename: dynpreaddmultadd.v

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

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

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

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

// Output accumulation result
assign dynpreaddmultadd_out = p_reg;

endmodule // dynpreaddmultadd
```

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

Filename: dynpreaddmultadd.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 the output of the pre-adder.

Download the coding example files from Coding Examples .

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

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

Square of a Difference (Verilog)

Filename: squarediffmult.v

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

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

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

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

// Output result
assign square_out = p_reg;
```

```
endmodule // squarediffmult
```

Square of a Difference (VHDL)

Filename: squarediffmult.vhd

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

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

entity squarediffmult is
generic(
SIZEIN : natural := 16
);
port(
clk, ce, rst : in std_logic;
ain, bin : in std_logic_vector(SIZEIN - 1 downto 0);
square_out : out std_logic_vector(2 * SIZEIN + 1 downto 0)
);
end squarediffmult;

architecture rtl of squarediffmult is

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

begin
process(clk)
begin
if rising_edge(clk) then
if rst = '1' then
a_reg <= (others => '0');
b_reg <= (others => '0');
diff_reg <= (others => '0');
m_reg <= (others => '0');
p_reg <= (others => '0');
else
a_reg <= signed(ain);
b_reg <= signed(bin);
```

```
diff_reg <= resize(a_reg, SIZEIN + 1) - resize(b_reg, SIZEIN + 1);
m_reg <= diff_reg * diff_reg;
p_reg <= m_reg;
end if;
end if;
end process;

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

end rtl;
```

FIR Filters

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

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

Download the coding example files from Coding Examples .

8-Tap Even Symmetric Systolic FIR (Verilog)

Filename: sfir_even_symetric_systolic_top.v

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

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

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

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

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

endmodule

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

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

assign cascdata = datatwo;

always @(posedge clk)
begin
coeff <= coeffin;
data <= datain;
datatwo <= data;
dataz <= datazin;
preadd <= datatwo + dataz;
product <= preadd * coeff;
cascout <= product + cascin;
end

endmodule


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

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

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

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

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

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

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

endmodule // sfir_even_symmetric_systolic_top
```

8-Tap Even Symmetric Systolic FIR (VHDL)

Filename: sfir_even_symetric_systolic_top.vhd

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


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

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

```
architecture rtl of sfir_shifter is

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

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

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

end rtl;

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

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

architecture rtl of sfir_even_symmetric_systolic_element is

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

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

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

end rtl;

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

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

architecture rtl of sfir_even_symmetric_systolic_top is

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

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

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

begin

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

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

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

end rtl;
```

Convergent Rounding (LSB Correction Technique)

The DSP block primitive leverages a pattern detect circuitry to compute convergent rounding (either to even, or to odd).
The following are examples of the convergent rounding inference, which infers at the block full performance, and also infers a 2-input AND gate (1 LUT) to implement the LSB correction.

Rounding to Even (Verilog)

Filename: convergentRoundingEven.v

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

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

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

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

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

assign multadd = z1 + c + 1'b1;

always @(posedge clk)
begin
areg <= a;
breg <= b;
```

```
z1 <= areg * breg;
pattern_detect <= multadd[15:0] == pattern ? 1'b1 : 1'b0;
multadd_reg <= multadd;
end

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

endmodule // convergentRoundingEven
```

Rounding to Even (VHDL)

Filename: convergentRoundingEven.vhd

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

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

architecture beh of convergentRoundingEven is

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

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

constant pattern : signed(15 downto 0) := (others => '0');
constant c : signed := "0000000000000000000000000111111111111111";

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

begin

multadd <= z1 + c + 1;

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

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

end beh;
```

Rounding to Odd (Verilog)

Filename: convergentRoundingOdd.v

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

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

reg pattern_detect;
wire [15:0] pattern = 16'b1111111111111111;
wire [39:0] c = 40'b0000000000000000000000000111111111111111; // 15 ones

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

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

assign multadd = z1 + c;

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

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

endmodule // convergentRoundingOdd
```

Rounding to Odd (VHDL)

Filename: convergentRoundingOdd.vhd

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

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

architecture beh of convergentRoundingOdd is

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

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

constant pattern : signed(15 downto 0) := (others => '1');
constant c : signed := "0000000000000000000000000111111111111111";

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

begin

multadd <= z1 + c;

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

```
pattern_detect <= true;
else
pattern_detect <= false;
end if;
end if;
end process;

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

end beh;
```

RAM HDL Coding Techniques

Vivado synthesis can interpret various RAM coding styles, and maps them into distributed RAMs or block RAMs. This action does the following:

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

Download the coding example files from Coding Examples.

Choosing Between Distributed RAM and Dedicated Block RAM

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

Table: Distributed RAM versus Dedicated Block RAM

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

Whether to use distributed RAM or dedicated block RAM can depend upon the characteristics of the RAM described in the HDL source code, the availability of block

RAM resources, and whether you have forced a specific implementation style using RAM_STYLE attribute.

Memory Inference Capabilities

Memory inference capabilities include the following:

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

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

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

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

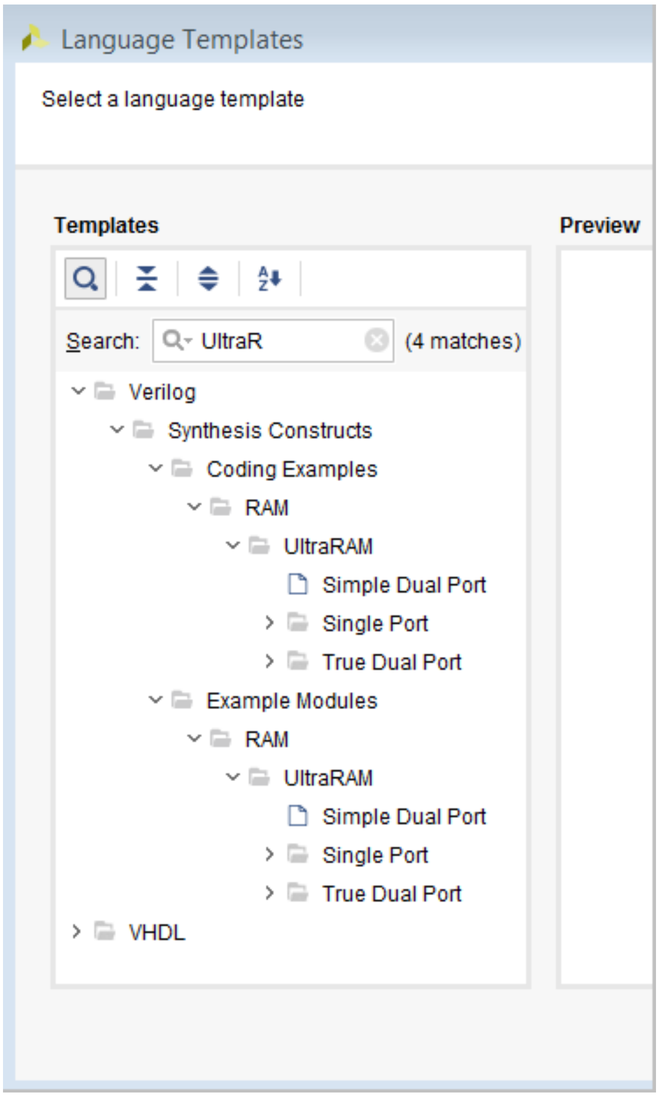
UltraRAM Coding Templates

UltraRAM is described in "Chapter 2, UltraRAM Resources" of the *UltraScale Architecture Memory Resources User Guide* (UG573) as follows:
UltraRAM is a single-clocked, two port, synchronous memory available in AMD UltraScale+™ devices. Because UltraRAM is compatible with the columnar architecture, multiple UltraRAMs can be instantiated and directly cascaded in an UltraRAM column for the entire height of the device. A column in a single clock region contains 16 UltraRAM blocks. Devices with UltraRAM include multiple UltraRAM columns distributed in the device. Most of the devices in the UltraScale+ family include UltraRAM blocks. For the available quantity of UltraRAM in specific device families, see the *UltraScale Architecture and Product Data Sheet: Overview* (DS890).
The following files are included in the Coding Examples:

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

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

Figure: UltraRAM Coding Templates



See the *UltraScale Architecture Memory Resources User Guide* (UG573) for more information.

Inferring UltraRAM in Vivado Synthesis

Overview of the UltraRAM Primitive

UltraRAM is a new dedicated memory primitive available in the UltraScale+ devices from AMD. This is a large memory that is designed to be cascaded for very large RAM blocks. For more information, see the *UltraScale Architecture Memory Resources User Guide* (UG573).

Description of the UltraRAM Primitive

The UltraRAM primitive is a dual port memory with a single clock. A single primitive is configured as 4 K x 72. The UltraRAM has two ports, which can access all 4 K of the RAM. This allows for a single port, simple dual port, and true dual-port behavior. There are also multiple pipeline registers for each port of the primitive. The UltraRAM has one clock, global enable, an output register reset, a write enable, and byte write enable support for control signals.

Differences between UltraRAM and Block RAM

There are a few notable differences between UltraRAM and block RAM to consider, as follows:

- The UltraRAM only has one clock, so while true dual port operation is supported, both ports are synchronous to each other.
- The aspect ratio of the UltraRAM is not configurable like block RAM, it is always configured as 4 K x 72.
- The resets on the output registers can only be reset to 0.
- The write modes (`read_first`, `write_first`, no_change) do not exist in this primitive. The regular UltraRAM behaves like no_change; however, if you describe `read_first` or `write_first` in RTL, the Vivado synthesis creates the correct logic.
- Finally, the `INIT` for RAM does not exist, the UltraRAM powers up in a 0 condition.

Using UltraRAM Inference

There are three ways of getting UltraRAM primitives, as follows:

Direct instantiation

Provides you the most control but is the hardest to perform.

XPM flow

Allows you to specify the type of RAM you want along with the behavior, but gives no access to the RTL.

Inference RAM

Is in the middle of the two, relatively easy, and gives more control to the user on how the RAM is created.

Attributes for Controlling UltraRAM

There are two attributes needed to control UltraRAM in Vivado synthesis: `RAM_STYLE` and `CASCADE_HEIGHT`.

RAM_STYLE

The RAM_STYLE attribute has a value called ultra. By default, Vivado synthesis uses a heuristic to determine what type of RAM to infer, URAM, block RAM or LUTRAM. If you want to force the RAM into an UltraRAM, you can use the RAM_STYLE attribute to tell Vivado synthesis to infer the URAM primitives.
More information is available in RAM_STYLE.

RAM_STYLE Verilog Example

```
(* ram_style = "ultra" *) 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 "ultra";
```

CASCADE_HEIGHT

When cascading multiple UltraRAMs (URAMs) together to create a larger RAM, Vivado synthesis limits the height of the chain to eight to provide flexibility to the place and route tool. To change this limit, you can use the `CASCADE_HEIGHT` attribute to change the default behavior.

✎ **Note:** This option is only applicable to UltraScale architecture block RAMs and URAMs.

CASCADE_HEIGHT Verilog Example

```
(* cascade_height = 16 *) reg [data_size-1:0] myram [2**addr_size-1:0];
```

CASCADE_HEIGHT VHDL Example

```
attribute cascade_height : integer;
attribute cascade_height of my_ram signal is 16;
```

In addition to the attributes that only affect the specific RAMs on which they are put, there is also a global setting which affects all RAMs in the design.
The Synthesis Settings menu has the `-max_uram_cascade_height` setting. The default value is -1 which means that the Vivado synthesis tool determines the best course of action, but this can be overridden by other values. In case of a conflict between the global setting and a `CASCADE_HEIGHT` attribute, the attribute is used for that specific RAM.

Inference Capabilities

The Vivado Synthesis tool can do many types of memories using the UltraRAM primitives. For examples, see the Coding Guidelines.

- In single port memory, the same port that reads the memory also writes to it. All three of the write modes for the block RAM are supported, but it should be noted that the UltraRAM itself acts like a `NO_CHANGE` memory. If `WRITE_FIRST` or `READ_FIRST` behavior is described in the RTL, the UltraRAM created is set in simple dual-port mode.
- In a simple dual port memory, one port reads from the RAM while another writes to it. Vivado synthesis can infer these memories into UltraRAM.

  ★ **Tip:** One stipulation is that both ports must have the same clock.
- In True Dual Port mode, both ports can read from and write to the memory. In this mode, only the `NO_CHANGE` mode is supported.

⚠ **CAUTION!** Care should also be taken when simulating the true dual port RAM. In the previous versions of block RAM, there was address collision that was taken care of by the simulation models; with the UltraRAM, it is different. In the UltraRAM, port A always happens before port B. If Port A has a write and Port B is a read from that address, the memory is written to and read from, but if Port A has the read and Port B has the write, the old value is seen during the read.

⚠ **CAUTION!** Be sure to never read and write to the same address during the same clock cycle on a true dual-port memory because the RTL and post-synthesis simulations could be different.

For both the simple dual-port memory and the true dual-port memory, the clocks have to be the same for both ports.

In addition to the different styles of RAMs, there are also a few other features of the UltraRAM that can be inferred. The RAM has a global enable signal that precedes the write enable. It has the standard write enable, and byte write enable support. The data output also has a reset like the previous block RAM; however, in this case, there is no SRVAL that can be set. Only resets of 0 are supported.

Pipelining the RAM

The UltraRAM (URAM) supports pipelining registers into the RAM. This becomes especially useful when multiple UltraRAMs are used to create a very large RAM. To fully pipeline the RAM, you must add extra registers to the RAM output in RTL. To calculate the number of pipeline registers to use, add together the number of rows and columns in the RAM matrix.

✎ **Note:** The tool does not create the pipeline registers for you; they must be in the RTL code for Vivado synthesis to make use of them.

The synthesis log file has a section about URAMs and how many rows and columns are used to create the RAM matrix. You can use this section to add pipeline registers in the RTL.

To calculate the number of rows and columns of the matrix yourself, remember that

the UltraRAM is configured as a 4 K x 72.
To calculate the number of rows, take your address space of the RAM in RTL and divide by 4 K. If this number is higher than the number specified by `CASCADE_HEIGHT`, remove the extra RAMs, and start them on a new column in the log.

Creating Pipeline Example 1: 8K x 72

In this example, 8 K divided by 4 K is two, so there are 2 rows. If the `CASCADE_HEIGHT` is set higher than 2, it is a 2 x 1 matrix. There should be three pipeline stages added to the output of the RAM (2 + 1).

Creating Pipeline Example 2 : 8K x 80

In this example, 8 K divided by 4 K is two, so there are two rows. The data space does not matter for this calculation, so the matrix would be two rows and 1 column resulting in three pipeline registers again.

✎ **Note:** The whole matrix is reproduced to get the extra 8 bits of data space needed to create the RAM, but that does not matter to the calculation of pipeline registers.

Creating Pipeline Example 3: 16K x 70 CASCADE_HEIGHT Set to 3

In this example, 16 K divided by 4 K is four; however, because the `CASCADE_HEIGHT` is 3, this would be a 3 x 2 matrix. This would result in 5 pipeline registers that can be used.

RAM HDL Coding Guidelines

Download the coding example files from Coding Examples .

Block RAM Read/Write Synchronization Modes

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

Read-first
Old content is read before new content is loaded.

Write-first
New content is immediately made available for reading Write-first is also known as read-through.

No-change
Data output does not change as new content is loaded into RAM.

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

Distributed RAM Examples

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

Dual-Port RAM with Asynchronous Read Coding Verilog Example

Filename: rams_dist.v

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

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

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

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

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

endmodule
```

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

Filename: rams_dist.vhd

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

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

entity rams_dist is
```

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

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

do <= RAM(to_integer(unsigned(a)));

end syn;
```

Single-Port Block RAMs

The following sections provide VHDL and Verilog coding examples for Single-Port Block RAM.

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

Filename: rams_sp_rf_rst.v

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

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

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

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

endmodule
```

Single Port Block RAM with Resettable Data Output (VHDL)

Filename: rams_sp_rf_rst.vhd

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

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

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

architecture syn of rams_sp_rf_rst is
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
signal ram : ram_type;
begin
```

```
process(clk)
begin
if clk'event and clk = '1' then
if en = '1' then -- optional enable
if we = '1' then -- write enable
ram(to_integer(unsigned(addr))) <= di;
end if;
if rst = '1' then -- optional reset
do <= (others => '0');
else
do <= ram(to_integer(unsigned(addr)));
end if;
end if;
end if;
end process;

end syn;
```

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

Filename: rams_sp_wf.v

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

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

```
end
endmodule
```

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

Filename: rams_sp_wf.vhd

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

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

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

end syn;
```

Single-Port RAM with Read First (VHDL)

Filename: rams_sp_rf.vhd

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

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

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

end syn;
```

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

Filename: rams_sp_nc.v

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

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

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

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

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

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

Filename: rams_sp_nc.vhd

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

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

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

architecture syn of rams_sp_nc 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(to_integer(unsigned(addr))) <= di;
else
do <= RAM(to_integer(unsigned(addr)));
end if;
end if;
end if;
end process;

end syn;
```

Simple Dual-Port Block RAM Examples

The following sections provide VHDL and Verilog coding examples for Simple Dual-Port Block RAM.

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

Filename: simple_dual_one_clock.v

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

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

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

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

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

endmodule
```

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

Filename: simple_dual_one_clock.vhd

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

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

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

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

```
RAM(conv_integer(addra)) := dia;
end if;
end if;
end if;
end process;

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

end syn;
```

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

Filename: simple_dual_two_clocks.v

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

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

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

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

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

```
begin
dob <= ram[addrb];
end
end

endmodule
```

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

Filename: simple_dual_two_clocks.vhd

```
-- 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;
```

True Dual-Port Block RAM Examples

The following sections provide VHDL and Verilog coding examples for True Dual-Port Block RAM.

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

Filename: ram_tdp_rf_rf.v

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

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

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

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

always @(posedge clkb)
begin
```

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

endmodule
```

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

Filename: ram_tdp_rf_rf.vhd

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

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

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

architecture syn of rams_tdp_rf_rf is
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
shared variable RAM : ram_type;
begin
process(CLKA)
begin
```

```
if CLKA’event and CLKA = ‘1’ then
if ENA = ‘1’ then
DOA <= RAM(to_integer(unsigned(ADDRA)));
if WEA = ‘1’ then
RAM(to_integer(unsigned(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(to_integer(unsigned(ADDRB)));
if WEB = ‘1’ then
RAM(to_integer(unsigned(ADDRB))) := DIB;
end if;
end if;
end if;
end process;

end syn;
```

Block RAM with Optional Output Registers (Verilog)

Filename: rams_pipeline.v

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

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

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

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

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

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

Block RAM with Optional Output Registers (VHDL)

Filename: rams_pipeline.vhd

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

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

architecture beh of rams_pipeline is
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
signal ram : ram_type;
signal do1 : std_logic_vector(15 downto 0);
signal do2 : std_logic_vector(15 downto 0);
begin
process(clk1)
begin
if rising_edge(clk1) then
if we = '1' then
ram(to_integer(unsigned(addr1))) <= di;
end if;
do1 <= ram(to_integer(unsigned(addr1)));
end if;
end process;

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

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

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

end beh;
```

Byte Write Enable (Block RAM)

AMD 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

Byte Write Enable—True Dual Port with Byte-Wide Write Enable (Verilog)

Filename: bytewrite_tdp_ram_rf.v

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

module bytewrite_tdp_ram_rf
#(
//--------------------------------------------------------------------------
---
parameter NUM_COL = 4,
parameter COL_WIDTH = 8,
parameter ADDR_WIDTH = 10,
// Addr Width in bits : 2 *ADDR_WIDTH = RAM Depth
parameter DATA_WIDTH = NUM_COL*COL_WIDTH // Data Width in bits
//-------------------------------------------------------------------------
) (
input clkA,
input enaA,
input [NUM_COL-1:0] weA,
input [ADDR_WIDTH-1:0] addrA,
input [DATA_WIDTH-1:0] dinA,
output reg [DATA_WIDTH-1:0] doutA,

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


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

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

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

doutB <= ram_block[addrB];
end
end

endmodule // bytewrite_tdp_ram_rf
```

Byte Write Enable—True Dual Port READ_FIRST Mode (VHDL)

Filename: bytewrite_tdp_ram_rf.vhd

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

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

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

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

end bytewrite_tdp_ram_rf;

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

begin
```

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

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

Byte Write Enable—WRITE_FIRST Mode (VHDL)

Filename: bytewrite_tdp_ram_wf.vhd

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

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

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

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

end bytewrite_tdp_ram_wf;

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

begin

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

end process;

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

Byte-Wide Write Enable—NO_CHANGE Mode (Verilog)

Filename: bytewrite_tdp_ram_nc.v

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

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


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

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

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


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

```
end
end
end
endgenerate

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

endmodule // bytewrite_tdp_ram_nc
```

Byte-Wide Write Enable—NO_CHANGE Mode (VHDL)

Filename: bytewrite_tdp_ram_nc.vhd

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

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

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

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

end bytewrite_tdp_ram_nc;

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

begin

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

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

Asymmetric RAMs

The following sections provide VHDL and Verilog coding examples for asymmetric RAMs.

✎ **Note:** Asymmetric RAMs with byte-write enables are not supported with RTL inference. Please use the XPM flow if this is needed.

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

Filename: asym_ram_sdp_read_wider.vhd

```
-- Asymmetric port RAM
-- Read Wider than Write
-- asym_ram_sdp_read_wider.vhd

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

entity asym_ram_sdp_read_wider is
generic(
WIDTHA : integer := 4;
SIZEA : integer := 1024;
ADDRWIDTHA : integer := 10;
WIDTHB : integer := 16;
SIZEB : integer := 256;
ADDRWIDTHB : integer := 8
);

port(
clkA : in std_logic;
clkB : in std_logic;
enA : in std_logic;
enB : in std_logic;
weA : in std_logic;
addrA : in std_logic_vector(ADDRWIDTHA - 1 downto 0);
addrB : in std_logic_vector(ADDRWIDTHB - 1 downto 0);
diA : in std_logic_vector(WIDTHA - 1 downto 0);
doB : out std_logic_vector(WIDTHB - 1 downto 0)
);

end asym_ram_sdp_read_wider;

architecture behavioral of asym_ram_sdp_read_wider is
function max(L, R : INTEGER) return INTEGER is
begin
if L > R then
return L;
else
return R;
end if;
end;

function min(L, R : INTEGER) return INTEGER is
begin
if L < R then
return L;
else
return R;
end if;
end;

function log2(val : INTEGER) return natural is
variable res : natural;
begin
for i in 0 to 31 loop
if (val <= (2 ** i)) then
res := i;
exit;
end if;
end loop;
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA, WIDTHB);
constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
constant maxSIZE : integer := max(SIZEA, SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

-- An asymmetric RAM is modeled in a similar way as a symmetric RAM, with
an
-- array of array object. Its aspect ratio corresponds to the port with
the
-- lower data width (larger depth)
type ramType is array (0 to maxSIZE - 1) of std_logic_vector(minWIDTH - 1
downto 0);
```

```
signal my_ram : ramType := (others => (others => '0'));

signal readB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');
signal regA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');

begin

-- Write process
process(clkA)
begin
if rising_edge(clkA) then
if enA = '1' then
if weA = '1' then
my_ram(conv_integer(addrA)) <= diA;
end if;
end if;
end if;
end process;

-- Read process
process(clkB)
begin
if rising_edge(clkB) then
for i in 0 to RATIO - 1 loop
if enB = '1' then
readB((i + 1) * minWIDTH - 1 downto i * minWIDTH) <=
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO))));
end if;
end loop;
regB <= readB;
end if;
end process;

doB <= regB;

end behavioral;
```

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

Filename: asym_ram_sdp_read_wider.v

```
// Asymmetric port RAM
// Read Wider than Write. Read Statement in loop
//asym_ram_sdp_read_wider.v
```

```
module asym_ram_sdp_read_wider (clkA, clkB, enaA, weA, enaB, addrA,
addrB, diA, doB);
parameter WIDTHA = 4;
parameter SIZEA = 1024;
parameter ADDRWIDTHA = 10;

parameter WIDTHB = 16;
parameter SIZEB = 256;
parameter ADDRWIDTHB = 8;
input clkA;
input clkB;
input weA;
input enaA, enaB;
input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
output [WIDTHB-1:0] doB;
`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
if (value < 2)
log2 = value;
else
begin
shifted = value-1;
for (res=0; shifted>0; res=res+1)
shifted = shifted>>1;
log2 = res;
end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
localparam minWIDTH = `min(WIDTHA, WIDTHB);

localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHB-1:0] readB;
```

```
always @(posedge clkA)
begin
if (enaA) begin
if (weA)
RAM[addrA] <= diA;
end
end


always @(posedge clkB)
begin : ramread
integer i;
reg [log2RATIO-1:0] lsbaddr;
if (enaB) begin
for (i = 0; i < RATIO; i = i+1) begin
lsbaddr = i;
readB[(i+1)*minWIDTH-1 -: minWIDTH] <= RAM[{addrB, lsbaddr}];
end
end
end
assign doB = readB;

endmodule
```

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

Filename: asym_ram_sdp_write_wider.v

```
// Asymmetric port RAM
// Write wider than Read. Write Statement in a loop.
// asym_ram_sdp_write_wider.v

module asym_ram_sdp_write_wider (clkA, clkB, weA, enaA, enaB, addrA,
addrB, diA, doB);
parameter WIDTHB = 4;
parameter SIZEB = 1024;
parameter ADDRWIDTHB = 10;

parameter WIDTHA = 16;
parameter SIZEA = 256;
parameter ADDRWIDTHA = 8;
input clkA;
input clkB;
input weA;
input enaA, enaB;
input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
output [WIDTHB-1:0] doB;
`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
if (value < 2)
log2 = value;
else
begin
shifted = value-1;
for (res=0; shifted>0; res=res+1)
shifted = shifted>>1;
log2 = res;
end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
localparam minWIDTH = `min(WIDTHA, WIDTHB);

localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHB-1:0] readB;

always @(posedge clkB) begin
if (enaB) begin
readB <= RAM[addrB];
end
end
assign doB = readB;

always @(posedge clkA)
begin : ramwrite
integer i;
reg [log2RATIO-1:0] lsbaddr;
```

```
for (i=0; i< RATIO; i= i+ 1) begin : write1
lsbaddr = i;
if (enaA) begin
if (weA)
RAM[{addrA, lsbaddr}] <= diA[(i+1)*minWIDTH-1 -: minWIDTH];
end
end
end

endmodule
```

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

Filename: asym_ram_sdp_write_wider.vhd

```
-- 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_write_wider is
generic(
WIDTHA : integer := 4;
SIZEA : integer := 1024;
ADDRWIDTHA : integer := 10;
WIDTHB : integer := 16;
SIZEB : integer := 256;
ADDRWIDTHB : integer := 8
);

port(
clkA : in std_logic;
clkB : in std_logic;
enA : in std_logic;
enB : in std_logic;
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_write_wider;

architecture behavioral of asym_ram_sdp_write_wider is
function max(L, R : INTEGER) return INTEGER is
begin
if L > R then
return L;
else
return R;
end if;
end;

function min(L, R : INTEGER) return INTEGER is
begin
if L < R then
return L;
else
return R;
end if;
end;

function log2(val : INTEGER) return natural is
variable res : natural;
begin
for i in 0 to 31 loop
if (val <= (2 ** i)) then
res := i;
exit;
end if;
end loop;
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA, WIDTHB);
constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
constant maxSIZE : integer := max(SIZEA, SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

-- An asymmetric RAM is modeled in a similar way as a symmetric RAM, with
an
-- array of array object. Its aspect ratio corresponds to the port with
the
-- lower data width (larger depth)
type ramType is array (0 to maxSIZE - 1) of std_logic_vector(minWIDTH - 1
downto 0);
```

```
signal my_ram : ramType := (others => (others => '0'));

signal 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;
```

True Dual Port Asymmetric RAM Read First (Verilog)

Filename: asym_ram_tdp_read_first.v

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


module asym_ram_tdp_read_first (clkA, clkB, enaA, weA, enaB, weB, addrA,
addrB, diA, doA, diB, doB);
parameter WIDTHB = 4;
parameter SIZEB = 1024;
parameter ADDRWIDTHB = 10;
parameter WIDTHA = 16;
parameter SIZEA = 256;
parameter ADDRWIDTHA = 8;
input clkA;
input clkB;
input weA, weB;
input enaA, enaB;

input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
input [WIDTHB-1:0] diB;

output [WIDTHA-1:0] doA;
output [WIDTHB-1:0] doB;

`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
if (value < 2)
log2 = value;
else
begin
shifted = value-1;
for (res=0; shifted>0; res=res+1)
shifted = shifted>>1;
log2 = res;
end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
localparam minWIDTH = `min(WIDTHA, WIDTHB);
localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHA-1:0] readA;
reg [WIDTHB-1:0] readB;

always @(posedge clkB)
begin
if (enaB) begin
readB <= RAM[addrB] ;
if (weB)
RAM[addrB] <= diB;
end
end


always @(posedge clkA)
begin : portA
integer i;
reg [log2RATIO-1:0] lsbaddr ;
for (i=0; i< RATIO; i= i+ 1) begin
lsbaddr = i;
if (enaA) begin
readA[(i+1)*minWIDTH -1 -: minWIDTH] <= RAM[{addrA, lsbaddr}];

if (weA)
RAM[{addrA, lsbaddr}] <= diA[(i+1)*minWIDTH-1 -: minWIDTH];
end
end
end

assign doA = readA;
assign doB = readB;

endmodule
```

True Dual Port Asymmetric RAM Read First (VHDL)

Filename: asym_ram_tdp_read_first_first.vhd

```
-- asymmetric port RAM
-- True Dual port read first
-- asym_ram_tdp_read_first_first.vhd
```

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

entity asym_ram_tdp_read_first is
generic(
WIDTHA : integer := 4;
SIZEA : integer := 1024;
ADDRWIDTHA : integer := 10;
WIDTHB : integer := 16;
SIZEB : integer := 256;
ADDRWIDTHB : integer := 8
);

port(
clkA : in std_logic;
clkB : in std_logic;
enA : in std_logic;
enB : in std_logic;
weA : in std_logic;
weB : in std_logic;
addrA : in std_logic_vector(ADDRWIDTHA - 1 downto 0);
addrB : in std_logic_vector(ADDRWIDTHB - 1 downto 0);
diA : in std_logic_vector(WIDTHA - 1 downto 0);
diB : in std_logic_vector(WIDTHB - 1 downto 0);
doA : out std_logic_vector(WIDTHA - 1 downto 0);
doB : out std_logic_vector(WIDTHB - 1 downto 0)
);

end asym_ram_tdp_read_first;

architecture behavioral of asym_ram_tdp_read_first is
function max(L, R : INTEGER) return INTEGER is
begin
if L > R then
return L;
else
return R;
end if;
end;

function min(L, R : INTEGER) return INTEGER is
begin
if L < R then
return L;
else
```

```
return R;
end if;
end;

function log2(val : INTEGER) return natural is
variable res : natural;
begin
for i in 0 to 31 loop
if (val <= (2 ** i)) then
res := i;
exit;
end if;
end loop;
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA, WIDTHB);
constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
constant maxSIZE : integer := max(SIZEA, SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

-- An asymmetric RAM is modeled in a similar way as a symmetric RAM, with an
-- array of array object. Its aspect ratio corresponds to the port with the
-- lower data width (larger depth)
type ramType is array (0 to maxSIZE - 1) of std_logic_vector(minWIDTH - 1 downto 0);

signal my_ram : ramType := (others => (others => '0'));

signal readA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal readB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');
signal regA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');

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

```
end if;
end process;

process(clkB)
begin
if rising_edge(clkB) then
for i in 0 to RATIO - 1 loop
if enB = '1' then
readB((i + 1) * minWIDTH - 1 downto i * minWIDTH) <=
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO))));
if weB = '1' then
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO)))) <=
diB((i + 1) * minWIDTH - 1 downto i * minWIDTH);
end if;
end if;
end loop;
regB <= readB;
end if;
end process;

doA <= regA;
doB <= regB;

end behavioral;
```

True Dual Port Asymmetric RAM Write First (Verilog)

Filename: asym_ram_tdp_write_first.v

```
// Asymmetric port RAM - TDP
// WRITE_FIRST MODE.
// asym_ram_tdp_write_first.v


module asym_ram_tdp_write_first (clkA, clkB, enaA, weA, enaB, weB, addrA,
addrB, diA, doA, diB, doB);
parameter WIDTHB = 4;
parameter SIZEB = 1024;
parameter ADDRWIDTHB = 10;
parameter WIDTHA = 16;
parameter SIZEA = 256;
parameter ADDRWIDTHA = 8;
input clkA;
input clkB;
input weA, weB;
input enaA, enaB;

input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
input [WIDTHB-1:0] diB;

output [WIDTHA-1:0] doA;
output [WIDTHB-1:0] doB;

`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
if (value < 2)
log2 = value;
else
begin
shifted = value-1;
for (res=0; shifted>0; res=res+1)
shifted = shifted>>1;
log2 = res;
end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
localparam minWIDTH = `min(WIDTHA, WIDTHB);

localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHA-1:0] readA;
reg [WIDTHB-1:0] readB;

always @(posedge clkB)
begin
if (enaB) begin
if (weB)
RAM[addrB] = diB;
readB = RAM[addrB] ;
```

```
end
end


always @(posedge clkA)
begin : portA
integer i;
reg [log2RATIO-1:0] lsbaddr ;
for (i=0; i< RATIO; i= i+ 1) begin
lsbaddr = i;
if (enaA) begin

if (weA)
RAM[{addrA, lsbaddr}] = diA[(i+1)*minWIDTH-1 -: minWIDTH];

readA[(i+1)*minWIDTH -1 -: minWIDTH] = RAM[{addrA, lsbaddr}];
end
end
end

assign doA = readA;
assign doB = readB;

endmodule
```

True Dual Port Asymmetric RAM Write First (VHDL)

Filename: asym_ram_tdp_write_first.vhd

```
--Asymmetric RAM
--True Dual Port write first mode.
--asym_ram_tdp_write_first.vhd

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

entity asym_ram_tdp_write_first is
generic(
WIDTHA : integer := 4;
SIZEA : integer := 1024;
ADDRWIDTHA : integer := 10;
WIDTHB : integer := 16;
SIZEB : integer := 256;
ADDRWIDTHB : integer := 8
);

port(
clkA : in std_logic;
clkB : in std_logic;
enA : in std_logic;
enB : in std_logic;
weA : in std_logic;
weB : in std_logic;
addrA : in std_logic_vector(ADDRWIDTHA - 1 downto 0);
addrB : in std_logic_vector(ADDRWIDTHB - 1 downto 0);
diA : in std_logic_vector(WIDTHA - 1 downto 0);
diB : in std_logic_vector(WIDTHB - 1 downto 0);
doA : out std_logic_vector(WIDTHA - 1 downto 0);
doB : out std_logic_vector(WIDTHB - 1 downto 0)
);

end asym_ram_tdp_write_first;

architecture behavioral of asym_ram_tdp_write_first is
function max(L, R : INTEGER) return INTEGER is
begin
if L > R then
return L;
else
return R;
end if;
end;

function min(L, R : INTEGER) return INTEGER is
begin
if L < R then
return L;
else
return R;
end if;
end;

function log2(val : INTEGER) return natural is
variable res : natural;
begin
for i in 0 to 31 loop
if (val <= (2 ** i)) then
res := i;
exit;
end if;
end loop;
```

```
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA, WIDTHB);
constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
constant maxSIZE : integer := max(SIZEA, SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

-- An asymmetric RAM is modeled in a similar way as a symmetric RAM, with
an
-- array of array object. Its aspect ratio corresponds to the port with
the
-- lower data width (larger depth)
type ramType is array (0 to maxSIZE - 1) of std_logic_vector(minWIDTH - 1
downto 0);

signal my_ram : ramType := (others => (others => '0'));

signal readA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal readB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');
signal regA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');

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

process(clkB)
begin
if rising_edge(clkB) then
for i in 0 to RATIO - 1 loop
if enB = '1' then
if weB = '1' then
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO)))) <=
diB((i + 1) * minWIDTH - 1 downto i * minWIDTH);
```

```
end if;
-- The read statement below is placed after the write statement -- on
purpose
-- to ensure write-first synchronization through the variable
-- mechanism
readB((i + 1) * minWIDTH - 1 downto i * minWIDTH) <=
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO))));
end if;
end loop;
regB <= readB;
end if;
end process;

doA <= regA;
doB <= regB;

end behavioral;
```

Initializing RAM Contents

RAM can be initialized in following ways:

- Specifying RAM Initial Contents in the HDL Source Code
- Specifying RAM Initial Contents in an External Data File

Specifying RAM Initial Contents in the HDL Source Code

Use the signal default value mechanism to describe initial RAM contents directly in the HDL source code.

VHDL Coding Examples

```
type ram_type is array (0 to 31) of std_logic_vector(19 downto 0);
signal RAM : ram_type :=
(
X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300",
X"08602", X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500",
X"04001", X"02500", X"00340", X"00241", X"04002", X"08300", X"08201",
X"00500", X"08101", X"00602", X"04003", X"0241E", X"00301", X"00102",
X"02122", X"02021", X"0030D", X"08201"
);
```

All bit positions are initialized to the same value:

```
type ram_type is array (0 to 127) of std_logic_vector (15 downto 0);
signal RAM : ram_type := (others => (others => '0'));
```

Verilog Coding Example

All addressable words are initialized to the same value.

```
reg [DATA_WIDTH-1:0] ram [DEPTH-1:0];
integer i;
initial for (i=0; i<DEPTH; i=i+1) ram[i] = 0;
end
```

Specifying RAM Initial Contents in an External Data File

Use the file read function in the HDL source code to load the RAM initial contents from an external data file.

- The external data file is an ASCII text file with any name.
- Each line in the external data file describes the initial content at an address position in the RAM.
- There must be as many lines in the external data file as there are rows in the RAM array. An insufficient number of lines is flagged.
- The addressable position related to a given line is defined by the direction of the primary range of the signal modeling the RAM.
- You can represent RAM content in either binary or hexadecimal. You cannot mix both.
- The external data file cannot contain any other content, such as comments.

The following external data file initializes an 8 x 32-bit RAM with binary values:

```
00001110110000011001111011000110
00101011001011010101001000100011
01110100010100011000011100001111
01000001010000100101001110010100
00001001101001111111101000101011
00101101001011111110101010100111
11101111000100111000111101101101
10001111010010011001000011101111
00000001100011100011110010011111
11011111001110101011111001001010
11100111010100111110110011001010
11000100001001101100111100101001
10001011100101011111111111100001
11110101110110010000010110111010
01001011000000111001010110101110
11100001111111001010111010011110
01101111011010010100001101110001
01010100011011111000011000100100
11110000111101101111001100001011
10101101001111010100100100011100
```

```
01011100001010111111101110101110
01011101000100100111010010110101
11110111000100000101011101101101
11100111110001111010101100001101
01110100000011101111111000011111
00010011110101111000111001011101
01101110001111100011010101101111
10111100000000010011101011011011
11000001001101001101111100010000
00011111110010110110011111010101
01100100100000011100100101110000
10001000000100111011001010001111
11001000100011101001010001100001
10000000100111010011100111100011
11011111010010100010101010000111
10000000110111101000111110111011
10110011010111101111000110011001
00010111100001001010110111011100
10011100101110101111011010110011
01010011101101010001110110011010
01111011011100010101000101000001
10001000000110010110111001101010
11101000001101010000111001010110
11100011111100000111110101110101
01001010000000001111111101101111
00100011000011001000000010001111
10011000111010110001001011100100
11111111111011110101000101000111
11000011000101000011100110100000
01101101001011111010100011101001
10000111101100101001110011010111
11010110100100101110110010100100
01001111111001101101011111001011
11011001001101110110000100110111
10110110110111100101110011100110
10011100111001000010111111010110
00000000001011011111001010110010
10100110011010000010001000011011
11001010111111001001110001110101
00100001100010000111000101001000
00111100101111110001101101111010
11000010001010000000010100100001
11000001000110001101000101001110
10010011010100010001100100100111
```

Verilog Code Example

```
reg [31:0] ram [0:63];

initial begin
$readmemb("rams_20c.data", ram, 0, 63);
end
```

VHDL Code Example

Load the data as follows:

```
type RamType is array(0 to 7) of bit_vector(31 downto 0);
impure function InitRamFromFile (RamFileName : in string) return RamType
is
FILE RamFile : text is in RamFileName;
variable RamFileLine : line;
variable RAM : RamType;
begin
for I in RamType'range loop
readline (RamFile, RamFileLine);
read (RamFileLine, RAM(I));
end loop;
return RAM;
end function;
signal RAM : RamType := InitRamFromFile("rams_20c.data");
```

Initializing Block RAM (Verilog)

Filename: rams_sp_rom.v

```
// Initializing Block RAM (Single-Port Block RAM)
// File: rams_sp_rom
module rams_sp_rom (clk, we, addr, di, dout);
input clk;
input we;
input [5:0] addr;
input [19:0] di;
output [19:0] dout;

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

initial
begin
ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
```

```
ram[57] = 20'h00300; ram[56] = 20'h08602; ram[55] = 20'h02310;
ram[54] = 20'h0203B; ram[53] = 20'h08300; ram[52] = 20'h04002;
ram[51] = 20'h08201; ram[50] = 20'h00500; ram[49] = 20'h04001;
ram[48] = 20'h02500; ram[47] = 20'h00340; ram[46] = 20'h00241;
ram[45] = 20'h04002; ram[44] = 20'h08300; ram[43] = 20'h08201;
ram[42] = 20'h00500; ram[41] = 20'h08101; ram[40] = 20'h00602;
ram[39] = 20'h04003; ram[38] = 20'h0241E; ram[37] = 20'h00301;
ram[36] = 20'h00102; ram[35] = 20'h02122; ram[34] = 20'h02021;
ram[33] = 20'h00301; ram[32] = 20'h00102; ram[31] = 20'h02222;
ram[30] = 20'h04001; ram[29] = 20'h00342; ram[28] = 20'h0232B;
ram[27] = 20'h00900; ram[26] = 20'h00302; ram[25] = 20'h00102;
ram[24] = 20'h04002; ram[23] = 20'h00900; ram[22] = 20'h08201;
ram[21] = 20'h02023; ram[20] = 20'h00303; ram[19] = 20'h02433;
ram[18] = 20'h00301; ram[17] = 20'h04004; ram[16] = 20'h00301;
ram[15] = 20'h00102; ram[14] = 20'h02137; ram[13] = 20'h02036;
ram[12] = 20'h00301; ram[11] = 20'h00102; ram[10] = 20'h02237;
ram[9] = 20'h04004; ram[8] = 20'h00304; ram[7] = 20'h04040;
ram[6] = 20'h02500; ram[5] = 20'h02500; ram[4] = 20'h02500;
ram[3] = 20'h0030D; ram[2] = 20'h02341; ram[1] = 20'h08201;
ram[0] = 20'h0400D;
end

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

endmodule
```

Initializing Block RAM (VHDL)

Filename: rams_sp_rom.vhd

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

entity rams_sp_rom is
port(
clk : in std_logic;
we : in std_logic;
addr : in std_logic_vector(5 downto 0);
di : in std_logic_vector(19 downto 0);
do : out std_logic_vector(19 downto 0)
);
end rams_sp_rom;

architecture syn of rams_sp_rom is
type ram_type is array (63 downto 0) of std_logic_vector(19 downto 0);
signal RAM : ram_type := (X"0200A", X"00300", X"08101", X"04000",
X"08601", X"0233A",
X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
X"0030D", X"02341", X"08201", X"0400D");

begin
process(clk)
begin
if rising_edge(clk) then
if we = '1' then
RAM(to_integer(unsigned(addr))) <= di;
end if;
do <= RAM(to_integer(unsigned(addr)));
end if;
end process;

end syn;
```

Initializing Block RAM From an External Data File (Verilog)

Filename: rams_init_file.v

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

module rams_init_file (clk, we, addr, din, dout);
input clk;
input we;
input [5:0] addr;
input [31:0] din;
```

```
output [31:0] dout;

reg [31:0] ram [0:63];
reg [31:0] dout;

initial begin
$readmemb("rams_init_file.data",ram);
end

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

✎ **Note:** The external file initializing the RAM needs to be in bit vector form. External files in integer or hex format do not work.

Initializing Block RAM From an External Data File (VHDL)

Filename: rams_init_file.vhd

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

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

entity rams_init_file is
port(
clk : in std_logic;
we : in std_logic;
addr : in std_logic_vector(5 downto 0);
din : in std_logic_vector(31 downto 0);
dout : out std_logic_vector(31 downto 0)
);
end rams_init_file;

architecture syn of rams_init_file is
type RamType is array (0 to 63) of bit_vector(31 downto 0);
```

```
impure function InitRamFromFile(RamFileName : in string) return RamType
is
FILE RamFile : text is in RamFileName;
variable RamFileLine : line;
variable RAM : RamType;
begin
for I in RamType'range loop
readline(RamFile, RamFileLine);
read(RamFileLine, RAM(I));
end loop;
return RAM;
end function;

signal RAM : RamType := InitRamFromFile("rams_init_file.data");
begin
process(clk)
begin
if clk'event and clk = '1' then
if we = '1' then
RAM(to_integer(unsigned(addr))) <= to_bitvector(din);
end if;
dout <= to_stdlogicvector(RAM(to_integer(unsigned(addr))));
end if;
end process;

end syn;
```

✎ **Note:**
The external file initializing the RAM needs to be in bit vector form. External files in integer or hex format do not work.

3D RAM Inference

RAMs Using 3D Arrays

The following examples show inference of RAMs using 3D arrays.

3D RAM Inference Single Port (Verilog)

filename: rams_sp_3d.sv

```
// 3-D Ram Inference Example (Single port)
// File:rams_sp_3d.sv
module rams_sp_3d #(
parameter NUM_RAMS = 2,
A_WID = 10,
D_WID = 32
)
(
input clk,
input [NUM_RAMS-1:0] we,
input [NUM_RAMS-1:0] ena,
input [A_WID-1:0] addr [NUM_RAMS-1:0],
input [D_WID-1:0] din [NUM_RAMS-1:0],
output reg [D_WID-1:0] dout [NUM_RAMS-1:0]
);

reg [D_WID-1:0] mem [NUM_RAMS-1:0][2**A_WID-1:0];
genvar i;

generate
for(i=0;i<NUM_RAMS;i=i+1)
begin:u
always @ (posedge clk)
begin
if (ena[i]) begin
if(we[i])
begin
mem[i][addr[i]] <= din[i];
end
dout[i] <= mem[i][addr[i]];
end
end
end
endgenerate

endmodule
```

3D RAM Inference Single Port (VHDL)

Filename: ram_sp_3d.vhd

```
-- 3-D Ram Inference Example (Single port)
-- Compile this file in VHDL2008 mode
-- File:rams_sp_3d.vhd

library ieee;
use ieee.std_logic_1164.all;
package mypack is
type myarray_t is array(integer range<>) of std_logic_vector;
type mem_t is array(integer range<>) of myarray_t;
end package;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mypack.all;
entity rams_sp_3d is generic (
NUM_RAMS : integer := 2;
A_WID : integer := 10;
D_WID : integer := 32
);
port (
clk : in std_logic;
we : in std_logic_vector(NUM_RAMS-1 downto 0);
ena : in std_logic_vector(NUM_RAMS-1 downto 0);
addr : in myarray_t(NUM_RAMS-1 downto 0)(A_WID-1 downto 0);
din : in myarray_t(NUM_RAMS-1 downto 0)(D_WID-1 downto 0);
dout : out myarray_t(NUM_RAMS-1 downto 0)(D_WID-1 downto 0)
);
end rams_sp_3d;

architecture arch of rams_sp_3d is
signal mem : mem_t(NUM_RAMS-1 downto 0)(2**A_WID-1 downto 0)(D_WID-1
downto 0);
begin
process(clk)
begin
if(clk’event and clk=’1’) then
for i in 0 to NUM_RAMS-1 loop
if(ena(i) = ‘1’) then
if(we(i) = ‘1’) then
mem(i)(to_integer(unsigned(addr(i)))) <= din(i);
end if;
dout(i) <= mem(i)(to_integer(unsigned(addr(i))));
end if;
end loop;
end if;
end process;

end arch;
```

3D RAM Inference Simple Dual Port (Verilog)

Filename: rams_sdp_3d.sv

```
// 3-D Ram Inference Example (Simple Dual port)
// File:rams_sdp_3d.sv
```

```
module rams_sdp_3d #(
parameter NUM_RAMS = 2,
A_WID = 10,
D_WID = 32
)
(
input clka,
input clkb,
input [NUM_RAMS-1:0] wea,
input [NUM_RAMS-1:0] ena,
input [NUM_RAMS-1:0] enb,
input [A_WID-1:0] addra [NUM_RAMS-1:0],
input [A_WID-1:0] addrb [NUM_RAMS-1:0],
input [D_WID-1:0] dina [NUM_RAMS-1:0],
output reg [D_WID-1:0] doutb [NUM_RAMS-1:0]
);

reg [D_WID-1:0] mem [NUM_RAMS-1:0][2**A_WID-1:0];
// PORT_A
genvar i;
generate
for(i=0;i<NUM_RAMS;i=i+1)
begin:port_a_ops
always @ (posedge clka)
begin
if (ena[i]) begin
if(wea[i])
begin
mem[i][addra[i]] <= dina[i];
end
end
end
end
endgenerate

//PORT_B
generate
for(i=0;i<NUM_RAMS;i=i+1)
begin:port_b_ops
always @ (posedge clkb)
begin
if (enb[i])
doutb[i] <= mem[i][addrb[i]];
end
end
endgenerate
```

```
endmodule
```

3D RAM Inference - Simple Dual Port (VHDL)

filename: rams_sdp_3d.vhd

```
-- 3-D Ram Inference Example ( Simple Dual port)
-- Compile this file in VHDL2008 mode
-- File:rams_sdp_3d.vhd

library ieee;
use ieee.std_logic_1164.all;
package mypack is
type myarray_t is array(integer range<>) of std_logic_vector;
type mem_t is array(integer range<>) of myarray_t;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mypack.all;
entity rams_sdp_3d is generic (
NUM_RAMS : integer := 2;
A_WID : integer := 10;
D_WID : integer := 32
);
port (
clka : in std_logic;
clkb : in std_logic;
wea : in std_logic_vector(NUM_RAMS-1 downto 0);
ena : in std_logic_vector(NUM_RAMS-1 downto 0);
enb : in std_logic_vector(NUM_RAMS-1 downto 0);
addra : in myarray_t(NUM_RAMS-1 downto 0)(A_WID-1 downto 0);
addrb : in myarray_t(NUM_RAMS-1 downto 0)(A_WID-1 downto 0);
dina : in myarray_t(NUM_RAMS-1 downto 0)(D_WID-1 downto 0);
doutb : out myarray_t(NUM_RAMS-1 downto 0)(D_WID-1 downto 0)
);
end rams_sdp_3d;

architecture arch of rams_sdp_3d is
signal mem : mem_t(NUM_RAMS-1 downto 0)(2**A_WID-1 downto 0)(D_WID-1
downto 0);
begin
process(clka)
begin
if(clka'event and clka='1') then
```

```
for i in 0 to NUM_RAMS-1 loop
if(ena(i) = '1') then
if(wea(i) = '1') then
mem(i)(to_integer(unsigned(addra(i)))) <= dina(i);
end if;
end if;
end loop;
end if;
end process;

process(clkb)
begin
if(clkb'event and clkb='1') then
for i in 0 to NUM_RAMS-1 loop
if(enb(i) = '1') then
doutb(i) <= mem(i)(to_integer(unsigned(addrb(i))));
end if;
end loop;
end if;
end process;

end arch;
```

3D RAM Inference True Dual Port (Verilog)

Filename: rams_tdp_3d.sv

```
// 3-D Ram Inference Example (True Dual port)
// File:rams_tdp_3d.sv
module rams_tdp_3d #(
parameter NUM_RAMS = 2,
A_WID = 10,
D_WID = 32
)
(
input clka,
input clkb,
input [NUM_RAMS-1:0] wea,
input [NUM_RAMS-1:0] web,
input [NUM_RAMS-1:0] ena,
input [NUM_RAMS-1:0] enb,
input [A_WID-1:0] addra [NUM_RAMS-1:0],
input [A_WID-1:0] addrb [NUM_RAMS-1:0],
input [D_WID-1:0] dina [NUM_RAMS-1:0],
input [D_WID-1:0] dinb [NUM_RAMS-1:0],
output reg [D_WID-1:0] douta [NUM_RAMS-1:0],
output reg [D_WID-1:0] doutb [NUM_RAMS-1:0]
);

reg [D_WID-1:0] mem [NUM_RAMS-1:0][2**A_WID-1:0];
// PORT_A
genvar i;
generate
for(i=0;i<NUM_RAMS;i=i+1)
begin:port_a_ops
always @ (posedge clka)
begin
if (ena[i]) begin
if(wea[i])
begin
mem[i][addra[i]] <= dina[i];
end
douta[i] <= mem[i][addra[i]];
end
end
end
endgenerate

//PORT_B
generate
for(i=0;i<NUM_RAMS;i=i+1)
begin:port_b_ops
always @ (posedge clkb)
begin
if (enb[i]) begin
if(web[i])
begin
mem[i][addrb[i]] <= dinb[i];
end
doutb[i] <= mem[i][addrb[i]];
end
end
end
endgenerate

endmodule
```

RAM Inference Using Structures and Records

The following examples show inference of RAMs using Structures and Records.

RAM Inference Single Port Structure (Verilog)

Filename: rams_sp_struct.sv

```
// RAM Inference using Struct in SV(Simple Dual port)
// File:rams_sdp_struct.sv
typedef struct packed {
logic [3:0] addr;
logic [27:0] data;
} Packet;


module rams_sdp_struct #(
parameter A_WID = 10,
D_WID = 32
)
(
input clk,
input we,
input ena,
input [A_WID-1:0] raddr, waddr,
input Packet din,
output Packet dout
);

Packet mem [2**A_WID-1:0];

always @ (posedge clk)
begin
if (ena) begin
if(we)
mem[waddr] <= din;
end
end

always @ (posedge clk)
begin
if (ena) begin
dout <= mem[raddr];
end
end
endmodule
```

RAM Inference Single Port Structure (VHDL)

Filename: rams_sp_record.vhd

```
-- Ram Inference Example using Records (Single port)
-- File:rams_sp_record.vhd

library ieee;
use ieee.std_logic_1164.all;
package mypack is
type Packet is record
addr : std_logic_vector(3 downto 0);
data : std_logic_vector(27 downto 0);
end record Packet;
type mem_t is array(integer range<>) of Packet;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mypack.all;
entity rams_sp_record is generic (
A_WID : integer := 10;
D_WID : integer := 32
);
port (
clk : in std_logic;
we : in std_logic;
ena : in std_logic;
addr : in std_logic_vector(A_WID-1 downto 0);
din : in Packet;
dout : out Packet
);
end rams_sp_record;

architecture arch of rams_sp_record is
signal mem : mem_t(2**A_WID-1 downto 0);
begin
process(clk)
begin
if(clk'event and clk='1') then
if(ena = '1') then
if(we = '1') then
mem(to_integer(unsigned(addr))) <= din;
end if;
dout <= mem(to_integer(unsigned(addr)));
end if;
end if;
end process;

end arch;
```

RAM Inference - Simple Dual Port Structure (SystemVerilog)

Filename: rams_sdp_struct.sv

```
// RAM Inference using Struct in SV(Simple Dual port)
// File:rams_sdp_struct.sv
typedef struct packed {
logic [3:0] addr;
logic [27:0] data;
} Packet;


module rams_sdp_struct #(
parameter A_WID = 10,
D_WID = 32
)
(
input clk,
input we,
input ena,
input [A_WID-1:0] raddr, waddr,
input Packet din,
output Packet dout
);

Packet mem [2**A_WID-1:0];

always @ (posedge clk)
begin
if (ena) begin
if(we)
mem[waddr] <= din;
end
end

always @ (posedge clk)
begin
if (ena) begin
dout <= mem[raddr];
end
end
endmodule
```

RAM Inference - Simple Dual Port Record (VHDL)

Filename: rams_sdp_record.vhd

```
-- Ram Inference Example using Records (Simple Dual port)
-- File:rams_sdp_record.vhd

library ieee;
use ieee.std_logic_1164.all;
package mypack is
type Packet is record
addr : std_logic_vector(3 downto 0);
data : std_logic_vector(27 downto 0);
end record Packet;
type mem_t is array(integer range<>) of Packet;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mypack.all;
entity rams_sdp_record is generic (
A_WID : integer := 10;
D_WID : integer := 32
);
port (
clk : in std_logic;
we : in std_logic;
ena : in std_logic;
raddr : in std_logic_vector(A_WID-1 downto 0);
waddr : in std_logic_vector(A_WID-1 downto 0);
din : in Packet;
dout : out Packet
);
end rams_sdp_record;

architecture arch of rams_sdp_record is
signal mem : mem_t(2**A_WID-1 downto 0);
begin
process(clk)
begin
if(clk'event and clk='1') then
if(ena = '1') then
if(we = '1') then
mem(to_integer(unsigned(waddr))) <= din;
end if;
end if;
end if;
end process;
```

```
process(clk)
begin
if(clk'event and clk='1') then
if(ena = '1') then
dout <= mem(to_integer(unsigned(raddr)));
end if;
end if;
end process;

end arch;
```

RAM Inference True Dual Port Structure (SystemVerilog)

Filename: rams_tdp_struct.sv

```
// RAM Inference using Struct in SV(True Dual port)
// File:rams_tdp_struct.sv
typedef struct packed {
logic [3:0] addr;
logic [27:0] data;
} Packet;

module rams_tdp_struct #(
parameter A_WID = 10,
D_WID = 32
)
(
input clka,
input clkb,
input wea,
input web,
input ena,
input enb,
input [A_WID-1:0] addra,
input [A_WID-1:0] addrb,
input Packet dina, dinb,
output Packet douta, doutb
);

Packet mem [2**A_WID-1:0];

always @ (posedge clka)
begin
if (ena)
begin
```

```
douta <= mem[addra];
if(wea)
mem[addra] <= dina;
end
end

always @ (posedge clkb)
begin
if (enb)
begin
doutb <= mem[addrb];
if(web)
mem[addrb] <= dinb;
end
end

endmodule
```

RAM Inference True Dual Port Record (VHDL)

Filename: rams_tdp_record.vhd

```
-- Ram Inference Example using Records (True Dual port)
-- File:rams_tdp_record.vhd

library ieee;
use ieee.std_logic_1164.all;
package mypack is
type Packet is record
addr : std_logic_vector(3 downto 0);
data : std_logic_vector(27 downto 0);
end record Packet;
type mem_t is array(integer range<>) of Packet;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mypack.all;
entity rams_tdp_record is generic (
A_WID : integer := 10;
D_WID : integer := 32
);
port (
clka : in std_logic;
clkb : in std_logic;
wea : in std_logic;
web : in std_logic;
ena : in std_logic;
enb : in std_logic;
addra : in std_logic_vector(A_WID-1 downto 0);
addrb : in std_logic_vector(A_WID-1 downto 0);
dina : in Packet;
dinb : in Packet;
douta : out Packet;
doutb : out Packet
);
end rams_tdp_record;

architecture arch of rams_tdp_record is
signal mem : mem_t(2**A_WID-1 downto 0);
begin

process(clka)
begin
if(clka'event and clka='1') then
if(ena = '1') then
douta <= mem(to_integer(unsigned(addra)));
if(wea = '1') then
mem(to_integer(unsigned(addra))) <= dina;
end if;
end if;
end if;
end process;


process(clkb)
begin
if(clkb'event and clkb='1') then
if(enb = '1') then
doutb <= mem(to_integer(unsigned(addrb)));
if(web = '1') then
mem(to_integer(unsigned(addrb))) <= dinb;
end if;
end if;
end if;
end process;

end arch;
```

Black Boxes

A design can contain EDIF files generated by:

- Synthesis tools
- Schematic text editors
- Any other design entry mechanism

These modules must be instantiated to be connected to the rest of the design. Use BLACK_BOX instantiation in the HDL source code.
Vivado synthesis lets you apply specific constraints to these BLACK_BOX instantiations. After you make a design a BLACK_BOX, each instance of that design is a BLACK_BOX.
Download the coding example files from Coding Examples.

Black Box Verilog Example

Filename: black_box_1.v

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

module black_box_1 (DI_1, DI_2, DOUT);
input DI_1, DI_2;
output DOUT;

black_box1 U1 (
.in1(DI_1),
.in2(DI_2),
.dout(DOUT)
);

endmodule
```

Black Box VHDL Example

Filename: black_box_1.vhd

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

entity black_box_1 is
port(DI_1, DI_2 : in std_logic;
DOUT : out std_logic);
end black_box_1;
architecture rtl of black_box_1 is
component black_box1
port(I1 : in std_logic;
I2 : in std_logic;
O : out std_logic);
end component;

attribute black_box : string;
attribute black_box of black_box1 : component is "yes";

begin
U1 : black_box1 port map(I1 => DI_1, I2 => DI_2, O => DOUT);
end rtl;
```

FSM Components

Vivado Synthesis Features

- Specific inference capabilities for synchronous Finite State Machine (FSM) components.
- Built-in FSM encoding strategies to accommodate your optimization goals.
- FSM extraction is enabled by default.
- Use `-fsm_extraction off` to disable FSM extraction.

FSM Description

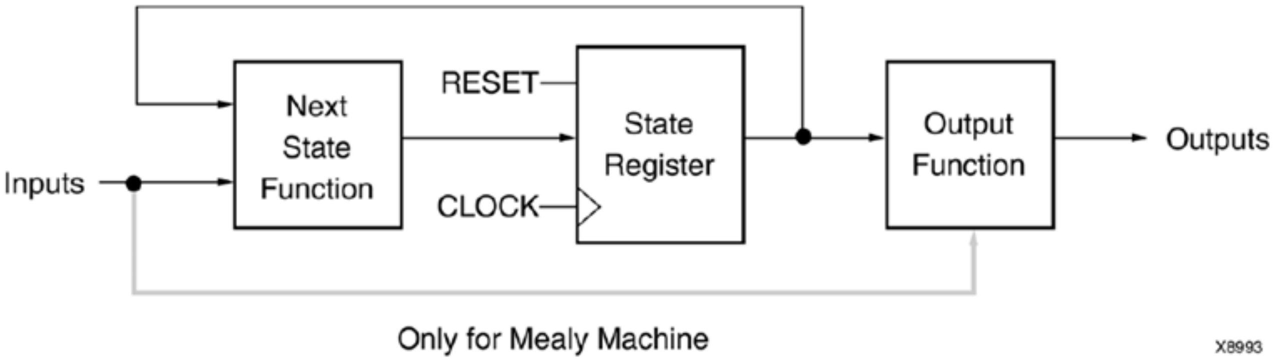
Vivado synthesis supports specification of Finite State Machine (FSM) in both Moore and Mealy form. An FSM consists of the following:

- A state register
- A next state function
- An outputs function

FSM Diagrams

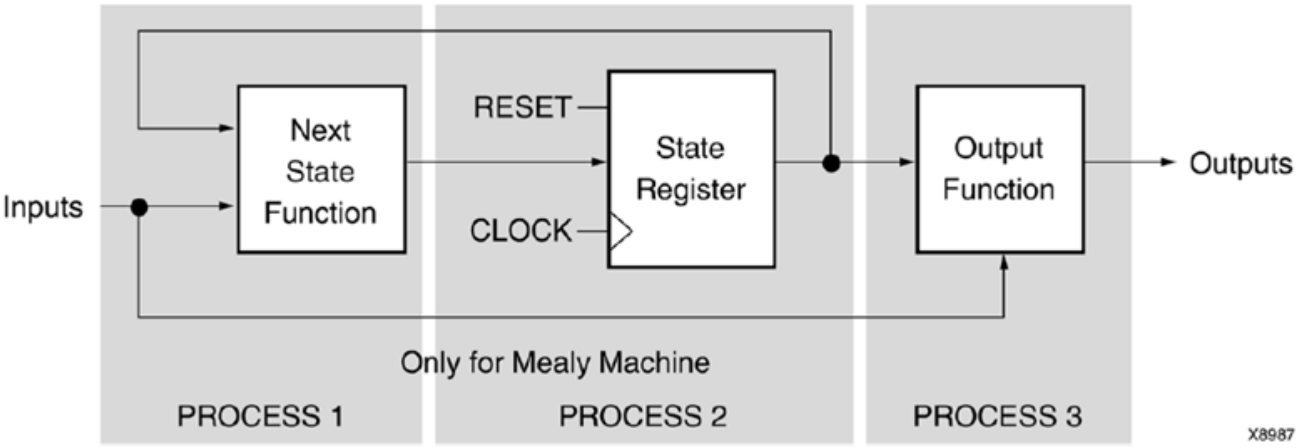
The following diagram shows an FSM representation that incorporates Mealy and Moore machines.

Figure: FSM Representation Incorporating Mealy and Moore Machines Diagram



The following diagram shows an FSM diagram with three processes.

Figure: 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.

✎ **Note:** Use synchronous reset logic over asynchronous reset logic for an FSM.

Auto State Encoding

When `FSM_ENCODING` is set to "`auto`", the Vivado synthesis attempts to select the best-suited encoding method for a given FSM.

One-Hot State Encoding

One-Hot State encoding has the following attributes:

- Is the default encoding scheme for a state machine, up to 32 states.
- Is usually a good choice for optimizing speed or reducing power dissipation.
- Assigns a distinct bit of code to each FSM state.
- Implements the State Register with one flip-flop for each state.
- In a given clock cycle during operation, only one bit of the State Register is asserted.
- Only two bits toggle during a transition between two states.

Gray State Encoding

Gray State encoding has the following attributes:

- Guarantees that only one bit switches between two consecutive states.
- Is appropriate for controllers exhibiting long paths without branching.
- Minimizes hazards and glitches.
- Can be used to minimize power dissipation.

Johnson State Encoding

Johnson State encoding is beneficial when using state machines containing long paths with no branching (as in Gray State Encoding).

Sequential State Encoding

Sequential State encoding has the following attributes:

- Identifies long paths
- Applies successive radix two codes to the states on these paths.
- Minimizes next state equations.

FSM Example (Verilog)

Filename: fsm_1.v

```
// State Machine with single sequential block
//fsm_1.v
module fsm_1(clk,reset,flag,sm_out);
input clk,reset,flag;
output reg sm_out;

parameter s1 = 3'b000;
parameter s2 = 3'b001;
parameter s3 = 3'b010;
parameter s4 = 3'b011;
parameter s5 = 3'b111;
```

```
reg [2:0] state;

always@(posedge clk)
begin
if(reset)
begin
state <= s1;
sm_out <= 1'b1;
end
else
begin
case(state)
s1: if(flag)
begin
state <= s2;
sm_out <= 1'b1;
end
else
begin
state <= s3;
sm_out <= 1'b0;
end
s2: begin state <= s4; sm_out <= 1'b0; end
s3: begin state <= s4; sm_out <= 1'b0; end
s4: begin state <= s5; sm_out <= 1'b1; end
s5: begin state <= s1; sm_out <= 1'b1; end
endcase
end
end
endmodule
```

FSM Example with Single Sequential Block (VHDL)

Filename: fsm_1.vhd

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

entity fsm_1 is
port(
clk, reset, flag : IN std_logic;
sm_out : OUT std_logic
);
```

```
end entity;

architecture behavioral of fsm_1 is
type state_type is (s1, s2, s3, s4, s5);
signal state : state_type;
begin
process(clk)
begin
if rising_edge(clk) then
if (reset = '1') then
state <= s1;
sm_out <= '1';

else
case state is
when s1 => if flag = '1' then
state <= s2;
sm_out <= '1';

else
state <= s3;
sm_out <= '0';

end if;
when s2 => state <= s4;
sm_out <= '0';
when s3 => state <= s4;
sm_out <= '0';
when s4 => state <= s5;
sm_out <= '1';
when s5 => state <= s1;
sm_out <= '1';
end case;
end if;
end if;
end process;

end behavioral;
```

FSM Reporting

The Vivado synthesis flags INFO messages in the log file, giving information about Finite State Machine (FSM) components and their encoding. The following are example messages:

```
INFO: [Synth 8-802] inferred FSM for state register 'state_reg' in module
'fsm_test'
```

```
INFO: [Synth 8-3354] encoded FSM with state register 'state_reg' using
encoding 'sequential' in module 'fsm_test'
```

ROM HDL Coding Techniques

Read-only memory (ROM) closely resembles random access memory (RAM) with respect to HDL modeling and implementation. Use the ROM_STYLE attribute to implement a properly-registered ROM on block RAM resources. See ROM_STYLE for more information.

ROM Using Block RAM Resources (Verilog)

Filename: rams_sp_rom_1.v

```
// ROMs Using Block RAM Resources.
// File: rams_sp_rom_1.v
//
module rams_sp_rom_1 (clk, en, addr, dout);
input clk;
input en;
input [5:0] addr;
output [19:0] dout;

(*rom_style = "block" *) reg [19:0] data;

always @(posedge clk)
begin
if (en)
case(addr)
6'b000000: data <= 20'h0200A; 6'b100000: data <= 20'h02222;
6'b000001: data <= 20'h00300; 6'b100001: data <= 20'h04001;
6'b000010: data <= 20'h08101; 6'b100010: data <= 20'h00342;
6'b000011: data <= 20'h04000; 6'b100011: data <= 20'h0232B;
6'b000100: data <= 20'h08601; 6'b100100: data <= 20'h00900;
6'b000101: data <= 20'h0233A; 6'b100101: data <= 20'h00302;
6'b000110: data <= 20'h00300; 6'b100110: data <= 20'h00102;
6'b000111: data <= 20'h08602; 6'b100111: data <= 20'h04002;
6'b001000: data <= 20'h02310; 6'b101000: data <= 20'h00900;
6'b001001: data <= 20'h0203B; 6'b101001: data <= 20'h08201;
6'b001010: data <= 20'h08300; 6'b101010: data <= 20'h02023;
6'b001011: data <= 20'h04002; 6'b101011: data <= 20'h00303;
6'b001100: data <= 20'h08201; 6'b101100: data <= 20'h02433;
6'b001101: data <= 20'h00500; 6'b101101: data <= 20'h00301;
6'b001110: data <= 20'h04001; 6'b101110: data <= 20'h04004;
6'b001111: data <= 20'h02500; 6'b101111: data <= 20'h00301;
6'b010000: data <= 20'h00340; 6'b110000: data <= 20'h00102;
```

```
6'b010001: data <= 20'h00241; 6'b110001: data <= 20'h02137;
6'b010010: data <= 20'h04002; 6'b110010: data <= 20'h02036;
6'b010011: data <= 20'h08300; 6'b110011: data <= 20'h00301;
6'b010100: data <= 20'h08201; 6'b110100: data <= 20'h00102;
6'b010101: data <= 20'h00500; 6'b110101: data <= 20'h02237;
6'b010110: data <= 20'h08101; 6'b110110: data <= 20'h04004;
6'b010111: data <= 20'h00602; 6'b110111: data <= 20'h00304;
6'b011000: data <= 20'h04003; 6'b111000: data <= 20'h04040;
6'b011001: data <= 20'h0241E; 6'b111001: data <= 20'h02500;
6'b011010: data <= 20'h00301; 6'b111010: data <= 20'h02500;
6'b011011: data <= 20'h00102; 6'b111011: data <= 20'h02500;
6'b011100: data <= 20'h02122; 6'b111100: data <= 20'h0030D;
6'b011101: data <= 20'h02021; 6'b111101: data <= 20'h02341;
6'b011110: data <= 20'h00301; 6'b111110: data <= 20'h08201;
6'b011111: data <= 20'h00102; 6'b111111: data <= 20'h0400D;
endcase
end

assign dout = data;

endmodule
```

ROM Inference on an Array (VHDL)

Filename: roms_1.vhd

```
-- 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 AMD 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 language forms.
- Simulate a system design before it is implemented and programmed in hardware.
- Produce a detailed, device-dependent version of a design to be synthesized from a more abstract specification.

For more information, see the IEEE VHDL Language Reference Manual (LRM).

Supported and Unsupported VHDL Data Types

Some VHDL data types are part of predefined packages. For information on where

they are compiled, and how to load them, see VHDL Predefined Packages.
The type is defined in the IEEE `std_logic_1164` package.

Unsupported Data Types

VHDL supports the real type defined in the standard package for calculations only, such as the calculation of generics values.

‼ **Important:** You cannot define a synthesizable object of type `real`.

VHDL Data Types

VHDL Predefined Enumerated Types

Vivado synthesis supports the following predefined VHDL enumerated types.

Table: 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 std_logic Allowed Values.

std_logic Allowed Values

Table: std_logic Allowed Values

Value	Meaning	What Vivado synthesis does
U	initialized	Not accepted by Vivado synthesis
X	unknown	Treated as do not care
0	low	Treated as logic zero.
1	high	Treated as logic one
Z	high impedance	Treated as high impedance

Value	Meaning	What Vivado synthesis does
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 do not care

Supported Overloaded Enumerated Types

Table: 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).

User-Defined Enumerated Types Coding Example (VHDL)

```
type STATES is (START, IDLE, STATE1, STATE2, STATE3) ;
```

Supported VHDL Types

Table: Supported VHDL Bit Vector Types

Type	Defined In Package	Models
bit_vector	Standard	Vector of bit elements

Type	Defined In Package	Models
std_logic_vector	IEEE std_logic_1164	Vector of std_logic elements

Table: Supported VHDL Overloaded Types

Type	Defined In IEEE Package
std_ulogic_vector	std_logic_1164
unsigned	numeric_std
signed	numeric_std
unsigned	std_logic_arith (Synopsys)
signed	std_logic_arith (Synopsys)

VHDL Integer Types

The integer type is a predefined VHDL type. Vivado synthesis implements an integer on 32 bits by default. For a more compact implementation, define the exact range of applicable values, where type MSB is range 8 to 15.
You can also take advantage of the predefined natural and positive types, overloading the integer type.

VHDL Multi-Dimensional Array Types

Vivado synthesis supports VHDL multi-dimensional array types.

✎ **Recommended:** Although there is no restriction on the number of dimensions, describe no more than three dimensions.

Objects of multi-dimensional array type can be passed to functions and used in component instantiations. Objects of multi-dimensional array type that you can describe are signals, constants, and variables.

Fully Constrained Array Type Coding Example

An array type must be fully constrained in all dimensions.

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB12 is array (11 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB12;
```

Array Declared as a Matrix Coding Example

You can declare an array as a matrix.

```
subtype TAB13 is array (7 downto 0,4 downto 0) of STD_LOGIC_VECTOR (8
downto 0);
```

Multi-Dimensional Array Signals and Variables Coding Examples

1. Make the following declarations:

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB05 is array (4 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB05;
signal WORD_A : WORD8;
signal TAB_A, TAB_B : TAB05;
signal TAB_C, TAB_D : TAB03;
constant CNST_A : TAB03 := (
("00000000","01000001","01000010","10000011","00001100"),
("00100000","00100001","00101010","10100011","00101100"),
("01000010","01000010","01000100","01000111","01000100"));
```

2. You can now specify:
   - A multi-dimensional array signal or variable:

     ```
     TAB_A <= TAB_B; TAB_C <= TAB_D; TAB_C <= CNST_A;
     ```

   - An index of one array:

     ```
     TAB_A (5) <= WORD_A; TAB_C (1) <= TAB_A;
     ```

   - Indexes of the maximum number of dimensions:

     ```
     TAB_A (5) (0) <= '1'; TAB_C (2) (5) (0) <= '0'
     ```

   - A slice of the first array

     ```
     TAB_A (4 downto 1) <= TAB_B (3 downto 0);
     ```

   - An index of a higher level array and a slice of a lower level array:

     ```
     TAB_C (2) (5) (3 downto 0) <= TAB_B (3) (4 downto 1); TAB_D (0)
     (4) (2 downto 0)
     \\ <= CNST_A (5 downto 3)
     ```

3. Add the following declaration:

```
subtype MATRIX15 is array(4 downto 0, 2 downto 0) of STD_LOGIC_VECTOR (7 downto 0);
signal MATRIX_A : MATRIX15;
```

4. You can now specify:

- A multi-dimensional array signal or variable:

  ```
  MATRIXA <= CNST_A
  ```

- An index of one row of the array:

  ```
  MATRIXA (5) <= TAB_A;
  ```

- Indexes of the maximum number of dimensions

  ```
  MATRIXA (5,0) (0) <= '1';
  ```

✎ **Note:** Indexes can be variable.

VHDL Record Types Code Example

- A field of a record type can also be of type Record.
- Constants can be record types.
- Record types cannot contain attributes.
- Vivado synthesis supports aggregate assignments to record signals. The following code snippet is an example:

```
type mytype is record field1 : std_logic; field2 : std_logic_vector (3
downto 0);
end record;
```

VHDL Objects

VHDL objects include: Signals, Variables, Constants, and Operators.

Signals

Declare a VHDL signal in:

- An architecture declarative part: Use the VHDL signal anywhere within that architecture.
- A block: Use the VHDL signal within that block.

Assign the VHDL signal with the <= signal assignment operator.

```
signal sig1 : std_logic;
sig1 <= '1';
```

Variables

A VHDL variable is:

- Declared in a process or a subprogram.
- Used within that process or subprogram.
- Assigned with the := assignment operator.

```
variable var1 : std_logic_vector (7 downto 0); var1 := "01010011";
```

Constants

You can declare a VHDL constant in any declarative region. The constant is used within that region. You cannot change the constant values after they are declared.

```
signal sig1 : std_logic_vector(5 downto 0);constant init0 :
std_logic_vector (5 downto 0) := "010111";sig1 <= init0;
```

Operators

Vivado synthesis supports VHDL operators.

Shift Operator Examples

Table: 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) sla 2	sig1 <= A(2 downto 0) & A(0) & A(0);
SRA (Shift Right Arithmetic)	sig1 <= A(4 downto 0) sra 2	sig1 <= <= A(4) & A(4) & A(4 downto 2);
ROL (Rotate Left)	sig1 <= A(4 downto 0) rol 2	sig1 <= A(2 downto 0) & A(4 downto 3);
ROR (Rotate Right)	A(4 downto 0) ror 2	sig1 <= A(1 downto 0) & A(4 downto 2);

VHDL Entity and Architecture Descriptions

VHDL Circuit Descriptions

A VHDL circuit description (design unit) consists of the following:

- Entity declaration: Provides the external view of the circuit. Describes objects visible from the outside, including the circuit interface, such as the I/O ports and generics.
- Architecture: Provides the internal view of the circuit, and describes the circuit behavior or structure.

VHDL Entity Declarations

The I/O ports of the circuit are declared in the entity. Each port has a:

- name
- mode (in, out, inout, buffer)
- type

Constrained and Unconstrained Ports

When defining a port, the port:

- Can be constrained or unconstrained.
- Are usually constrained.
- Can be left unconstrained in the entity declaration.
    - If ports are left unconstrained, their width is defined at instantiation when the connection is made between formal ports and actual signals.
    - Unconstrained ports allow you to create different instantiations of the same entity, defining different port widths.

✎ **Recommended:** Do not use unconstrained ports. Define ports that are constrained through generics. Apply different values of those generics at instantiation. Do not have an unconstrained port on the top-level entity.

Array types of more than one-dimension are not accepted as ports. The entity declaration can also declare VHDL generics.

Buffer Port Mode

✎ **Recommended:** Do not use buffer port mode.

VHDL allows buffer port mode when a signal is used both internally, and as an output port when there is only one internal driver. Buffer ports are a potential source of errors during synthesis, and complicate validation of post-synthesis results through simulation.

NOT RECOMMENDED Coding Example WITH Buffer Port Mode

```
entity alu is
port(
CLK : in STD_LOGIC;
A : inSTD_LOGIC_VECTOR(3 downto 0);
B : inSTD_LOGIC_VECTOR(3 downto 0);
C : buffer STD_LOGIC_VECTOR(3 downto 0));
end alu;

architecture behavioral of alu is
begin
process begin
if rising_edge(CLK) then
C <= UNSIGNED(A) + UNSIGNED(B) UNSIGNED(C);
end if;
end process;
end behavioral;
```

Dropping Buffer Port 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.

Component Instantiation (VHDL)

Filename: instantiation_simple.vhd

This coding example shows the structural description of a half-Adder composed of four nand2 components.

```
--
-- A simple component instantiation example
-- Involves a component declaration and the component instantiation
itself
--
-- instantiation_simple.vhd
--
entity sub is
generic(
WIDTH : integer := 4
);
port(
A, B : in BIT_VECTOR(WIDTH - 1 downto 0);
O : out BIT_VECTOR(2 * WIDTH - 1 downto 0)
);
end sub;

architecture archi of sub is
begin
O <= A & B;
end ARCHI;

entity instantiation_simple is
generic(
WIDTH : integer := 2);
port(
X, Y : in BIT_VECTOR(WIDTH - 1 downto 0);
Z : out BIT_VECTOR(2 * WIDTH - 1 downto 0));
end instantiation_simple;

architecture ARCHI of instantiation_simple is
component sub -- component declaration
generic(
WIDTH : integer := 2);
port(
A, B : in BIT_VECTOR(WIDTH - 1 downto 0);
O : out BIT_VECTOR(2 * WIDTH - 1 downto 0));
end component;

begin
inst_sub : sub -- component instantiation
generic map(
WIDTH => WIDTH
)
port map(
A => X,
B => Y,
O => Z
);

end ARCHI;
```

Recursive Component Instantiation

Vivado synthesis supports recursive component instantiation.

Recursive Component Instantiation Example (VHDL)

Filename: instantiation_recursive.vhd

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

entity instantiation_recursive is
generic(
sh_st : integer := 4
);
port(
CLK : in std_logic;
DI : in std_logic;
DO : out std_logic
);
end entity instantiation_recursive;
```

```
architecture recursive of instantiation_recursive is
component instantiation_recursive
generic(
sh_st : integer);
port(
CLK : in std_logic;
DI : in std_logic;
DO : out std_logic);
end component;
signal tmp : std_logic;
begin
GEN_FD_LAST : if sh_st = 1 generate
inst_fd : FD port map(D => DI, C => CLK, Q => DO);
end generate;
GEN_FD_INTERM : if sh_st /= 1 generate
inst_fd : FD port map(D => DI, C => CLK, Q => tmp);
inst_sstage : instantiation_recursive
generic map(sh_st => sh_st - 1)
port map(DI => tmp, CLK => CLK, DO => DO);
end generate;
end recursive;
```

VHDL Component Configuration

A component configuration explicitly links a component with the appropriate model.

- A model is an entity and architecture pair.
- Vivado synthesis supports component configuration in the declarative part of the architecture. The following is an example:

```
for instantiation_list : component_name use
LibName.entity_Name(Architecture_Name);
```

The following statement indicates that:

- All NAND2 components use the design unit consisting of entity NAND2 and architecture ARCHI.
- The design unit is compiled in the work library.

```
For all : NAND2 use entity work.NAND2(ARCHI);
```

The value of the top module name (`-top`) option in the `synth_design` command is the configuration name instead of the top-level entity name.

VHDL GENERICS

VHDL GENERICs have the following properties:

- Are equivalent to Verilog parameters.
- Help you create scalable design modelizations.
- Let you write compact, factorized VHDL code.
- Let you parameterize functionality such as bus size, and the number of repetitive elements in the design unit.

For the same functionality that must be instantiated multiple times, but with different bus sizes, you need describe only one design unit with generics. See the GENERIC Parameters Example.

Declaring Generics

You can declare generic parameters in the entity declaration part. Supported generics types are: `integer`, `boolean`, `string`, and `real`.

GENERIC Parameters Example

Filename: generics_1.vhd

```
-- VHDL generic parameters example
--
-- generics_1.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity addern is
generic(
width : integer := 8
);
port(
A, B : in std_logic_vector(width - 1 downto 0);
Y : out std_logic_vector(width - 1 downto 0)
);
end addern;

architecture bhv of addern is
begin
Y <= A + B;
end bhv;

Library IEEE;
use IEEE.std_logic_1164.all;

entity generics_1 is
```

```
port(
X, Y, Z : in std_logic_vector(12 downto 0);
A, B : in std_logic_vector(4 downto 0);
S : out std_logic_vector(17 downto 0));
end generics_1;

architecture bhv of generics_1 is
component addern
generic(width : integer := 8);
port(
A, B : in std_logic_vector(width - 1 downto 0);
Y : out std_logic_vector(width - 1 downto 0));
end component;
for all : addern use entity work.addern(bhv);

signal C1 : std_logic_vector(12 downto 0);
signal C2, C3 : std_logic_vector(17 downto 0);
begin
U1 : addern generic map(width => 13) port map(X, Y, C1);
C2 <= C1 & A;
C3 <= Z & B;
U2 : addern generic map(width => 18) port map(C2, C3, S);
end bhv;
```

✎ **Note:**
When overriding generic values during instantiation, splitting up different array elements is not supported.

For example, if there is a generic `my_gen` defined as an array, as follows, it does not work:

```
my_gen(1) => x,
my_gen(0) => y
```

Instead, it should be set as follows:

```
my_gen => (x,y)
```

VHDL Combinatorial Circuits

Combinatorial logic is described using concurrent signal assignments that you specify in the body of an architecture. You can describe as many concurrent signal assignments as are necessary; the order of appearance of the concurrent signal assignments in the architecture is irrelevant.

VHDL Concurrent Signal Assignments

Concurrent signal assignments are concurrently active and re-evaluated when any signal on the right side of the assignment changes value. The re-evaluated result is assigned to the signal on the left-hand side.
Supported types of concurrent signal assignments are: Simple Signal Assignment Example, and Concurrent Selection Assignment Example (VHDL).

Simple Signal Assignment Example

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

Concurrent Selection Assignment Example (VHDL)

Filename: concurrent_selected_asssignment.vhd

```
-- Concurrent selection assignment in VHDL
--
-- concurrent_selected_assignment.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity concurrent_selected_assignment is
generic(
width : integer := 8);
port(
a, b, c, d : in std_logic_vector(width - 1 downto 0);
sel : in std_logic_vector(1 downto 0);
T : out std_logic_vector(width - 1 downto 0));
end concurrent_selected_assignment;

architecture bhv of concurrent_selected_assignment is
begin
with sel select T <=
a when "00",
b when "01",
c when "10",
d when others;
end bhv;
```

Generate Statements

Generate statements include:

- `for-generate` statements
- `if-generate` statements

Using for-generate Statements

The for-generate statements describe repetitive structures.

Example of for-generate Statement (VHDL)

Filename: for-generate.vhd

In this coding example, the `for-generate` statement describes the calculation of the result and carries 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;
```

Using 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.

Example of for-generate Nested in an if-generate Statement (VHDL)

Filename: if_for_generate.vhd

In this coding example, a generic N-bit adder with a width ranging between 4 and 32 is described with an `if-generate` and a `for-generate` statement.

```
-- A for-generate nested in a if-generate
--
-- if_for_generate.vhd
--
entity if_for_generate is
generic(
N : INTEGER := 8
);
port(
A, B : in BIT_VECTOR(N downto 0);
CIN : in BIT;
SUM : out BIT_VECTOR(N downto 0);
COUT : out BIT
);
end if_for_generate;

architecture archi of if_for_generate is
signal C : BIT_VECTOR(N + 1 downto 0);
begin
IF_N : if (N >= 4 and N <= 32) generate
C(0) <= CIN;
COUT <= C(N + 1);
LOOP_ADD : for I in 0 to N generate
SUM(I) <= A(I) xor B(I) xor C(I);
C(I + 1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
end generate;
end generate;
end archi;
```

Combinatorial Processes

You can model VHDL combinatorial logic with a process, which explicitly assigns signals a new value every time the process is executed.

‼ **Important:** No signals should implicitly retain its current value, and a process can contain local variables.

Memory Elements

Hardware inferred from a combinatorial process does not involve any memory elements.
A memory element process is combinatorial when all assigned signals in a process are

always explicitly assigned in all possible paths within a process block.
A signal that is not explicitly assigned in all branches of an if or case statement typically leads to a Latch inference.

‼ **Important:** If Vivado synthesis infers unexpected Latches, review the HDL source code for a signal that is not explicitly assigned.

Sensitivity List

A combinatorial process has a sensitivity list. The sensitivity list appears within parentheses after the `PROCESS` keyword. A process is activated if an event (value change) appears on one of the sensitivity list signals. For a combinatorial process, this sensitivity list must contain:

- All signals in conditions (for example, `if` and `case`).
- All signals on the right-hand side of an assignment.

Missing Signals

Signals might be missing from the sensitivity list. If one or more signals is missing from the sensitivity list:

- The synthesis results can differ from the initial design specification.
- Vivado synthesis issues a warning message.
- Vivado synthesis adds the missing signals to the sensitivity list.

‼ **Important:** To avoid problems during simulation, explicitly add all missing signals in the HDL source code and re-run synthesis.

Variable and Signal Assignments

Vivado synthesis supports VHDL variable and signal assignments. A process can contain local variables, which are declared and used within a process and generally not visible outside the process.

Signal Assignment in a Process Example

Filename: signal_in_process.vhd

```
-- Signal assignment in a process
-- signal_in_process.vhd

entity signal_in_process is
port(
A, B : in BIT;
S : out BIT
);
end signal_in_process;
```

```
architecture archi of signal_in_process is
begin
process(A, B)
begin
S <= '0';
if ((A and B) = '1') then
S <= '1';
end if;
end process;
end archi;
```

Variable and Signal Assignment in a Process Example (VHDL)

Filename: variable_in_process.vhd

```
-- 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;
```

Using 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.

Example of if-else Statement (VHDL)

```
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;
```

Using 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.

Example of case Statement (VHDL)

```
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;
```

Using for-loop Statements

Vivado synthesis `for-loop` statements support:

- Constant bounds
- Stop test condition using the following operators: <, <=, >, and >=.
- Next step computations falling within one of the following specifications:
  - var = var + step
  - var = var - step

    Where:
    - `var` is the loop variable
    - `step` is a constant value
- Next and exit statements

Example of for-loop Statement (VHDL)

Filename: for_loop.vhd

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

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

architecture behavior of for_loop is
begin
process(a)
variable Count_Aux : std_logic_vector(2 downto 0);
begin
Count_Aux := "000";
for i in a'range loop
if (a(i) = '0') then
Count_Aux := Count_Aux + 1;
end if;
end loop;
Count <= Count_Aux;
end process;
end behavior;
```

VHDL Sequential Logic

A VHDL process is sequential (as opposed to combinatorial) when some assigned signals are not explicitly assigned in all paths within the process. The generated hardware has an internal state or memory (Flip-Flops or Latches).

✎ **Recommended:** Use a sensitivity-list based description style to describe sequential logic.

Describing sequential logic using a process with a sensitivity list includes:

- The clock signal
- Any optional signal controlling the sequential element asynchronously (asynchronous set/reset)
- An if statement that models the clock event.

Sequential Process With a Sensitivity List Syntax

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

Asynchronous Control Logic Modelization

Modelization of any asynchronous control logic (asynchronous set/reset) is done before the clock event statement.
Modelization of the synchronous logic (data, optional synchronous set/reset, optional clock enable) is done in the if branch of the clock event.

Table: Asynchronous Control Logic Modelization Summary

Modelization Type	Contains	Performed
Asynchronous control logic	Asynchronous set/reset	Before the clock event statement
Synchronous logic	Data Optional synchronous set/reset Optional clock enable	In the clock event if branch.

Clock Event Statements

Describe the clock event statement as:

- Rising edge clock:

  ```
  if rising_edge (clk) then
  ```

- Falling edge clock:

  ```
  if falling_edge (clk) then
  ```

Missing Signals

If any signals are missing from the sensitivity list, the synthesis results can differ from the initial design specification. In this case, Vivado synthesis issues a warning message and adds the missing signals to the sensitivity list.

‼ **Important:** To avoid problems during simulation, explicitly add all missing signals in the HDL source code and re-run synthesis.

VHDL Sequential Processes Without a Sensitivity List

Vivado synthesis allows the description of a sequential process using a wait statement. The sequential process is described without a sensitivity list.
The wait statement is the first statement and the condition in the wait statement describes the sequential logic clock.

‼ **Important:** The same sequential process cannot have both a sensitivity list and a wait statement, and only one wait statement is allowed.

Sequential Process Using a Wait Statement Coding Example (VHDL)

```
process begin
wait until rising_edge(clk);
q <= d;
end process;
```

Describing a Clock Enable in the wait Statement Example (VHDL)

You can describe a clock enable (`clken`) in the `wait` statement together with the clock.

```
process begin
wait until rising_edge(clk) and clken = '1';
q <= d;
end process;
```

Describing a Clock Enable After the Wait Statement Example (VHDL)

You can describe the clock enable separately, as follows:

```
process begin
wait until rising_edge(clk);
if clken = '1' then
q <= d;
end if;
end process;
```

Describing Synchronous Control Logic

You can use the same coding method as was shown to describe a clock enable to describe synchronous control logic, such as a synchronous reset or set.

‼ **Important:** You cannot describe a sequential element with asynchronous control logic using a process without a sensitivity list. Only a process with a sensitivity list allows such functionality. Vivado synthesis does not allow the description of a Latch based on a wait statement. For greater flexibility, describe synchronous logic using a process with a sensitivity list.

VHDL Initial Values and Operational Set/Reset

You can initialize registers when you declare them. The initialization value is a constant and can be generated from a function call.

Initializing Registers Example One (VHDL)

This coding example specifies a power-up value in which the sequential element is initialized when the circuit goes live and the circuit global reset is applied.

```
signal arb_onebit : std_logic := '0';
signal arb_priority : std_logic_vector(3 downto 0) := "1011";
```

Initializing Registers Example Two (VHDL)

Filename: initial_1.vhd

This coding example combines power-up initialization and operational reset.

```
--
-- Register initialization
-- Specifying initial contents at circuit powes-up
-- Specifying an operational set/reset
--
-- File: VHDL_Language_Support/initial/initial_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity initial_1 is
Port(
clk, rst : in std_logic;
din : in std_logic;
dout : out std_logic
);
end initial_1;

architecture behavioral of initial_1 is
signal arb_onebit : std_logic := '1'; -- power-up to vcc
```

```
begin
process(clk)
begin
if (rising_edge(clk)) then
if rst = '1' then -- local synchronous reset
arb_onebit <= '0';
else
arb_onebit <= din;
end if;
end if;
end process;

dout <= arb_onebit;

end behavioral;
```

VHDL Functions and Procedures

Use VHDL functions and procedures for blocks that are used multiple times in a design. The content is similar to combinatorial process content
Declare functions and procedures in:

- The declarative part of an entity
- An architecture
- A package

A function or procedure consists of a declarative part and a body. The declarative part specifies:

- `input` parameters, which can be unconstrained to a given bound.
- `output` and `inout` parameters (procedures only)

‼ **Important:** Resolution functions are not supported except the function defined in the IEEE `std_logic_1164` package.

Function Declared Within a Package Example (VHDL)

Filename: function_package_1.vhd

Download the coding example files from Coding Examples .
This coding example declares an `ADD` function within a package. The `ADD` function is a single-bit Adder and is called four times to create a 4-bit Adder. The following example uses a function:

```
-- Declaration of a function in a package
--
```

```
-- function_package_1.vhd
--
package PKG is
function ADD(A, B, CIN : BIT) return BIT_VECTOR;
end PKG;

package body PKG is
function ADD(A, B, CIN : BIT) return BIT_VECTOR is
variable S, COUT : BIT;
variable RESULT : BIT_VECTOR(1 downto 0);
begin
S := A xor B xor CIN;
COUT := (A and B) or (A and CIN) or (B and CIN);
RESULT := COUT & S;
return RESULT;
end ADD;
end PKG;

use work.PKG.all;

entity function_package_1 is
port(
A, B : in BIT_VECTOR(3 downto 0);
CIN : in BIT;
S : out BIT_VECTOR(3 downto 0);
COUT : out BIT
);
end function_package_1;

architecture ARCHI of function_package_1 is
signal S0, S1, S2, S3 : BIT_VECTOR(1 downto 0);
begin
S0 <= ADD(A(0), B(0), CIN);
S1 <= ADD(A(1), B(1), S0(1));
S2 <= ADD(A(2), B(2), S1(1));
S3 <= ADD(A(3), B(3), S2(1));
S <= S3(0) & S2(0) & S1(0) & S0(0);
COUT <= S3(1);
end ARCHI;
```

Procedure Declared Within a Package Example (VHDL)

Filename: procedure_package_1.vhd

The following example uses a procedure within a package:

```
-- Declaration of a procedure in a package
```

```
--
-- Download: procedure_package_1.vhd
--
package PKG is
procedure ADD(
A, B, CIN : in BIT;
C : out BIT_VECTOR(1 downto 0));
end PKG;

package body PKG is
procedure ADD(
A, B, CIN : in BIT;
C : out BIT_VECTOR(1 downto 0)) is
variable S, COUT : BIT;
begin
S := A xor B xor CIN;
COUT := (A and B) or (A and CIN) or (B and CIN);
C := COUT & S;
end ADD;
end PKG;

use work.PKG.all;

entity procedure_package_1 is
port(
A, B : in BIT_VECTOR(3 downto 0);
CIN : in BIT;
S : out BIT_VECTOR(3 downto 0);
COUT : out BIT
);
end procedure_package_1;

architecture ARCHI of procedure_package_1 is
begin
process(A, B, CIN)
variable S0, S1, S2, S3 : BIT_VECTOR(1 downto 0);
begin
ADD(A(0), B(0), CIN, S0);
ADD(A(1), B(1), S0(1), S1);
ADD(A(2), B(2), S1(1), S2);
ADD(A(3), B(3), S2(1), S3);
S <= S3(0) & S2(0) & S1(0) & S0(0);
COUT <= S3(1);
end process;
end ARCHI;
```

Recursive Functions Example (VHDL)

Vivado synthesis supports recursive functions. This coding example models an n! function.

```
function my_func(x : integer) return integer is begin
if R > 1 then
return (R*my_func(R-1));
else
return R;
end if;
end function my_func;
```

VHDL Assert Statements

Assert statements are supported with the -assert synthesis option.

⚠ **CAUTION!** Care should be taken using asserts. Vivado can only support static asserts that do not create or are created by behavior. For example, performing as assert on a value of a constant or an operator/generic works; however, as an asset on the value of a signal inside an if statement does not work.

VHDL Predefined Packages

Vivado synthesis supports the VHDL predefined packages as defined in the STD and IEEE standard libraries. The libraries are pre-compiled, and need not be user-compiled, and can be directly included in the HDL source code.

VHDL Predefined Standard Packages

VHDL predefined standard packages that are by default included, define the following basic VHDL types: `bit`, `bit_vector`, `integer`, `natural`, `real`, and `boolean`.

VHDL IEEE Packages

Vivado synthesis supports the some predefined VHDL IEEE packages, which are pre-compiled in the IEEE library, and the following IEEE packages:

- `numeric_bit`
    - Unsigned and signed vector types based on bit.
    - Overloaded arithmetic operators, conversion functions, and extended functions for these types.
- `std_logic_1164`
    - `std_logic`, `std_ulogic`, `std_logic_vector`, and `std_ulogic_vector` types.
    - Conversion functions based on these types.
- `numeric_std`
    - Unsigned and signed vector types based on `std_logic`.
    - Overloaded arithmetic operators, conversion functions, and extended functions for these types. Equivalent to `std_logic_arith`.
- `fixed_pkg`
    - For fixed variable and pin types.
    - `use ieee.fixed_pkg.all;`
- `float_pkg`
    - For floating variable and pin types.
    - `use ieee.float_pkg.all;`

VHDL Legacy Packages

- `std_logic_arith` (Synopsys)
    - Unsigned and signed vector types based on `std_logic`.
    - Overloaded arithmetic operators, conversion functions, and extended functions for these types.
- `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 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: VHDL Real Number Constants

Constant	Value	Constant	Value

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	log2
math_2_pi	2π	math_log10_of_e	log10
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: VHDL Real Number Functions

ceil(x)	realmax(x,y)	exp(x)	cos(x)	cosh(x)
floor(x)	realmin(x,y)	log(x)	tan(x)	tanh(x)
round(x)	sqrt(x)	log2(x)	arcsin(x)	arcsinh(x)
trunc(x)	cbrt(x)	log10(x)	arctan(x)	arccosh(x)
sign(x)	**(n,y)	log(x,y)	arctan(y,x)	arctanh(x)
mod(x,y)	**(x,y)	sin(x)	sinh(x)	

Defining Your Own VHDL Packages

You can define your own VHDL packages to specify:

- Types and subtypes
- Constants
- Functions and procedures
- Component declarations

Defining a VHDL package permits access to shared definitions and models from other parts of your project and requires the following:

Package declaration
Declares each of the previously listed elements.

Package body
Describes the functions and procedures declared in the package declaration.

Package Declaration Syntax

```
package mypackage is
type mytype is record
first : integer;
second : integer;
end record;
constant myzero : mytype := (first => 0, second => 0);
function getfirst (x : mytype) return integer;
end mypackage;

package body mypackage is
function getfirst (x : mytype) return integer is
begin
return x.first;
end function;
end mypackage;
```

Accessing VHDL Packages

To access a VHDL package:

1. Use a library clause to include the library in which the package is compiled. For example: `library library_name;`
2. Designate the package, or a specific definition contained in the package, with a use clause. For example: `use library_name.package_name.all`.
3. Insert these lines immediately before the entity or architecture in which you use the package definitions.

Because the `work` library is the default library, you can omit the library clause if the designated package has been compiled into this library.

VHDL Constructs Support Status

Vivado synthesis supports VHDL design entities and configurations except as noted in the following table.

Table: VHDL Constructs and Support Status

VHDL Construct	Support Status

VHDL Construct	Support Status
VHDL Entity Headers	
Generics	Supported
Ports	Supported, including unconstrained ports
Entity Statement Part	Unsupported
VHDL Packages	Supported
VHDL Physical Types	
TIME	Supported, but only in functions for constant calculations.
REAL	Supported, but only in functions for constant calculations.
VHDL Modes	
Linkage	Unsupported
VHDL Declarations	
Type	Supported for the following: • Enumerated types • Types with positive range having constant bounds • Bit vector types • Multi-dimensional arrays
VHDL Objects	
Constant Declaration	Supported except for deferred constant
Signal Declaration	Supported except for register and bus type signals.
Attribute Declaration	Supported for some attributes, otherwise skipped.
VHDL Specifications	
HIGHLOW	upported
LEFT	Supported

VHDL Construct	Support Status
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.
Rem	Supported if the right operand is a constant power of 2.
Mod	Supported if the right operand is a constant power of 2.

VHDL Construct	Support Status
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
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.

VHDL Construct	**Support Status**
Signal Assignment Statement	Supported. Delay is ignored.
Variable Assignment Statement	Supported
Procedure Call Statement	Supported
If Statement	Supported
Case Statement	Supported
Loop Statements	
Next Statements	Supported
Exit Statements	Supported
Return Statements	Supported
Null Statements	Supported
Concurrent Statements	
Process Statement	Supported
Concurrent Procedure Call	Supported
Concurrent Assertion Statements	Ignored
Concurrent Signal Assignments	Supported, except after clause, transport or guarded options, or waveforms. UNAFFECTED is supported.
Component Instantiation Statements	Supported
for-generate	Statement supported for constant bounds only
if-generate	Statement supported for static condition only

VHDL RESERVED Words

Table: VHDL RESERVED Words

abs	access	after	alias
all	and	architecture	array

assert	attribute	begin	block
body	buffer	bus	case
component	configuration	constant	disconnect
downto	else	elsif	end
entity	exit	file	for
function	generate	generic	group
guarded	if	impure	in
inertial	inout	is	label
library	linkage	literal	loop
map	mod	nand	new
next	nor	not	null
of	on	open	or
others	out	package	port
postponed	procedure	process	pure
range	record	register	reject
rem	report	return	rol
ror	select	severity	signal
shared	sla	sll	sra
srl	subtype	then	to
transport	type	unaffected	units
until	use	variable	wait
when	while	with	xnor

VHDL-2008 Language Support

Introduction

AMD Vivado™ synthesis supports a synthesizable subset of the VHDL-2008 standard. The following section describes the supported subset and the procedures to use it.

Setting up Vivado to use VHDL-2008

There are several ways to run VHDL-2008 files with Vivado. You can go to the Source File Properties window and set Type: VHDL 2008 from the drop-down of available file types. The Vivado tool sets the file type to VHDL-2008.
You can also set files to VHDL-2008 with the set_property command in the Tcl Console. The syntax is as follows:

```
set_property FILE_TYPE {VHDL 2008} [get_files <file>.vhd]
```

Finally, in the Non-Project or Tcl flow, the command for reading in VHDL has VHDL-2008 as follows:

```
read_vhdl -vhdl2008 <file>.vhd
```

If you want to read in more than one file, you can either use multiple `read_vhdl` commands or multiple files with one command, as follows:

```
read_vhdl -vhdl2008 {a.vhd b.vhd c.vhd}
```

Supported VHDL-2008 Features

Vivado supports the following VHDL-2008 features.

Operators

Matching Relational Operators

VHDL-2008 now provides relational operators that return bit or std_logic types. In the previous VHDL standard, the relational operators (=, <, >=...) returned boolean types. With the new types, code that needed to be written as:

```
if x = y then
out1 <= '1';
else
out1 <= '0';
end if;
```

Can now be written as:

```
out1 <= x ?= y;
```

The following table lists the relational operators supported in Vivado.

Table: Supported Relational Operators

Operator	Usage	Description
?=	x ?= y	x equal to y
?/=	x ?/= y	x not equal to y
?<	x ?< y	x less than y
?<=	x ?<= y	x less than or equal to y
?>	x ?> y	x greater than y
?>=	x ?>= y	x greater than or equal to y

Maximum and Minimum Operators

The new maximum and minimum operators in VHDL-2008 take in two different values and return the larger or smaller respectively. For example:

```
out1 <= maximum(const1, const2);
```

Shift Operators (rol, ror, sll, srl, sla, and sra)

The `sla`, and `sra` operators previously defined only bit and boolean elements. Now, the VHDL-2008 standard defines them in the signed and unsigned libraries.

Unary Logical Reduction Operators

In the previous version of VHDL, operators such as and, nand, or, took two different values and returned a bit or boolean value. For VHDL-2008, unary support has been added for these operators. They return the logical function of the input. For example, the code:

```
out1 <= and("0101");
```

would `AND` the 4 bits together and return `0` . The logical functions have unary support are: and, nand, `or`, `nor`, `xor`, and `xnor`.

Mixing Array and Scalar Logical Operators

Previously in VHDL, both of the operands of the logical operators needed to be the same size.
VHDL-2008 supports using logical operators when one of the operands is an array and one is a scalar. For example, to `AND` one bit with all the bits of a vector, the following code was needed:

```
out1(3) <= in1(3) and in2;
out1(2) <= in1(2) and in2;
out1(1) <= in1(1) and in2;
out1(0) <= in1(0) and in2;
```

This can now be replaced with the following:

```
out1<= in1 and in2;
```

Statements

If-else- If and Case Generate

Previously in VHDL, `if-generate` statements took the form of the following:

```
if condition generate
--- statements
end generate;
```

An issue appears if you want to have different conditions; you would need to write multiple generates and be very careful with the ordering of the generates. VHDL-2008 supports `if-else-if` generate statements.

```
if condition generate
---statements
elsif condition2 generate
---statements
else generate
---statements
end generate;
```

In addition, VHDL-2008 also offers `case-generate` statements:

```
case expressions generate
when condition =>
statements
when condition2 =>
statements
end generate;
```

Sequential Assignments

VHDL-2008 allows sequential signal and variable assignment with conditional signals. For example, a register with an enable would be written as the following:

```
process(clk) begin
if clk'event and clk='1' then
if enable then
my_reg <= my_input;
end if;
end if;
end process;
```

With VHDL-2008, this can now be written as the following:

```
process(clk) begin
if clk'event and clk='1' then
my_reg <= my_input when enable else my_reg;
end if;
end process;
```

Using case? Statements

With VHDL-2008, the case statement has a way to deal with explicit don’t care assignments. When using `case?` , the tool now evaluates explicit `don’t care` terms, as in the following example:

```
process(clk) begin
if clk'event and clk='1' then
case? my_reg is
when "01--" => out1 <= in1;
when "000-" => out1 <= in2;
when "1111" => out1 <= in3;
when others => out1 <= in4;
end case?;
end if;
end process;
```

✎ **Note:** For this statement to work, the signal in question must be assigned an explicit `don’t care`.

Using select? Statements

Like the case, the select statement now has a way to deal with explicit `don’t care` assignments. When using the `select?` statement, the tool now evaluates explicit `don’t care` terms, for example:

```
process(clk) begin
if clk'event and clk='1' then
with my_reg select?
out1 <= in1 when "11--",
in2 when "000-",
in3 when "1111",
in4 when others;
```

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

✎ **Note:** For this statement to work, the signal in question must be assigned an explicit `don’t care`.

Using Slices in Aggregates

VHDL-2008 allows you to form an array aggregate and assign it to multiple places all in one statement.
For example if in1 where defined as

```
std_logic_vector(3 downto 0) :
(my_reg1, my_reg2, enable, reset) <= in1;
```

This example assigns all four signals to the individual bits of `in1`:
my_reg1 gets in1(3)
my_reg2 gets in1(2)
enable is in1(1)
reset is in1(0)
In addition, these signals can be assigned out of order, as shown in the following example:

```
(1=> enable, 0 => reset, 3 => my_reg1, 2 => my_reg2) <= in1;
```

Types

Unconstrained Element Types

Previously, in VHDL, types and subtypes had to be fully constrained in declaring the type. In VHDL-2008, it is allowed to be unconstrained, and the constraining happens with the objects that are of that type; consequently, types and subtypes are more versatile. For example:

```
subtype my_type is std_logic_vector;
signal my_reg1 : my_type (3 downto 0);
signal my_reg2 : my_type (4 downto 0);
```

In previous versions of VHDL, the preceding example was done with 2 subtypes. Now, in VHDL-2008, this can be accomplished with one type. This can even be done for arrays, as shown in the following example:

```
type my_type is array (natural range <>) of std_logic_vector;
signal : mytype(1 downto 0)(9 downto 0);
```

Using boolean_vector and integer_vector Array Types

VHDL-2008 supports new predefined array types. Vivado supports `boolean_vector` and `integer_vector`. These types are defined as follows:

```
type boolean_vector is array (natural range <>) of boolean
type integer_vector is array (natural range <>) of integer
```

Miscellaneous

Reading Output Ports

In previous versions of VHDL, it was illegal to use signals declared as `out` for anything other than an output.
So if you wanted to assign a value to an output, and also use that same signal for other logic, you would either have to declare a new signal and have that drive the output and the other logic, or switch from an `out` to a `buffer` type.
VHDL-2008 lets you use output values, as shown in the following example:

```
entity test is port(
in1 : in std_logic;
clk : in std_logic;
out1, out2 : out std_logic);
end test;
```

And later in the architecture:

```
process(clk) begin
if clk'event an clk='1' then
out1 <= in1;
my_reg <= out1; -- THIS WOULD HAVE BEEN ILLEGAL in VHDL.
out2 <= my_reg;
end if;
end process;
```

Expressions in Port Maps

VHDL-2008 allows the use of functions and assignments within the port map of an instantiation. One useful way this is used is in converting signals from one type to another, as shown in the following example:

```
U0 : my_entity port map (clk => clk, in1 => to_integer(my_signal)...
```

In the previous case, the entity, `my_entity` had a port called `in1` that was of type integer, but in the upper-level, the signal, `my_signal` was of type `std_logic_vector`.
Previously in VHDL, you would have to create a new signal of type `integer` and do the conversion outside of the instantiation, and assign that new signal to the port map.
In addition to type conversion, you can put logic into the port map, as shown in the

following example:

```
U0 : my_entity port map (clk => clk, enable => en1 and en2 ...
```

In this case, the lower-level has an `enable` signal. On the top level that `enable` is tied to the `AND` of two other signals.
Previously in VHDL, this, like the previous example, would have needed a new signal and assignment, but in VHDL-2008 can be accomplished in the port map of the instantiation.

Using the process (all) Statement

In VHDL, when listing items in the sensitivity list of a process statement for combinational logic, it was up to the designer to make sure all the items read by the process statement were listed. If any were missed, there would be Warning messages and possible latches inferred in the design.
With VHDL-2008, you can use the `process(all)` statement that looks for all the inputs to the process and creates the logic.

```
process(all) begin
enable <= en1 and en2;
end process;
```

Referencing Generics in Generic Lists

VHDL-2008 allows generics to reference other generics, as shown in the following example:

```
entity my_entity is generic (
gen1 : integer;
gen2 : std_logic_vector(gen1 - 1 downto 0));
```

In previous versions of VHDL, having the length of gen2 be controlled by gen1 was illegal.

Generics in Packages

VHDL-2008 supports putting a generic in a package and being able to override that generic when the package is declared. For example:

```
package my_pack is
generic(
length : integer);

subtype my_type is std_logic_vector(length-1 downto 0);
end package my_pack;
```

This declares a subtype of std_logic_vector but does not specify the length. The calling VHDL file specifies what the length should be when the package is instantiated:

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

package my_pack1 is new work.my_pack generic map (length => 5);
package my_pack2 is new work.my_pack generic map (length => 3);
use work.my_pack1.all;
use work.my_pack2.all;

library ieee;
use ieee.std_logic_1164.all;

entity test is port (
clk : in std_logic;
in1 : in work.my_pack1.my_type;
in2 : in work.my_pack2.my_type;
out1 : out work.my_pack1.my_type;
out2 : out work.my_pack2.my_type);
end test;
```

This code uses the same package to declare two different subtypes and be able to use them.

Generic Types in Entities

VHDL-2008 supports undefined types in the generic statement for an entity. For example:

```
entity my_entity is
generic (type my_type);

port (in1 : in std_logic;
out1 : out my_type);
end entity my_entity;
```

This would declare an entity with an undetermined type, and the RTL that instantiates my_entity would look like:

```
my_inst1 : entity work.my_entity(beh) generic map (my_type => std_logic)
port map ...
my_inst2 : entity work.my_entity(beh) generic map (my_type =>
std_logic_vector(3 downto 0)) port map ...
```

The previous code instantiates my_entity twice, but in one case, out1 is a bit, and in the other case, out1 is a 4-bit vector.

Functions in Generics

In VHDL-2008, you can declare undefined functions inside of entities. For example

```
entity bottom is
generic (
function my_func (a,b : unsigned) return unsigned);
port ...
......
end entity bottom;
```

Later in the architecture of the entity:

```
process(clk) is
begin
if rising_edge(clk) then
y <= my_func(a,b);
end if;
end process;
```

This uses the my_func function, inside of the entity, but it still has not defined what this function actually accomplishes. That is defined as when the bottom is instantiated in an upper-level RTL.

```
inst_bot1 : bottom
generic map (
my_func => my_func1 )
port map ...
```

This ties the function my_func1 that was declared in a VHDL file or a package file to the generic function my_func. As long as my_func1 has two inputs called a and b that are both unsigned, it is able to work.

Relaxed Return Rules for Function Return Values

In previous versions of VHDL, the return expression of a function needed be same type as was declared in the functions return type of the function. In VHDL-2008, the rules are relaxed to allow the return expression to be implicitly converted to the return type. For example:

```
subtype my_type1 is std_logic_vector(9 downto 0);
subtype my_type2 is std_logic_vector(4 downto 0);

function my_function (a,b : my_type2) return my_type1 is
begin
return (a&b);
end function;
```

Because concatenation is not static, this would return an error or warning in VHDL; however, it is allowed with VHDL-2008.

Extensions to Globally Static and Locally Static Expressions

In VHDL, expressions in many types of places needed to be static. For example, using concatenation would not have returned a static value and when used with an operator or function that needed a static value resulting in an error. VHDL-2008 allows for more expressions, like concatenation to return static values, thereby allowing for more flexibility.

Static Ranges and Integer Expressions in Range Bounds

In VHDL, it was possible to declare an object by using the range of another object. For example:

```
for I in my_signal'range...
```

This would require that the range of my_signal be fixed, but if my_signal was declared as an unconstrained type, this would result in an error. VHDL-2008 now allows this by getting the range at the time of elaboration.

Block Comments

In VHDL, comments “ -- “ were required for each line that had a comment. In VHDL-2008, there is support for blocks of comments using the /* and */ lines.

```
process(clk) begin
if clk'event and clk='1' then
/* this
is
a block
comment */
out1 <= in1;
```

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

VHDL-2008 RESERVED Words

Table: VHDL-2008 RESERVED Words

abs	access	after	alias
all	and	architecture	array
assert	assume	assume_+guarantee	attribute
begin	block	body	buffer
bus	case	component	configuration
constant	context	cover	default
disconnect	downto	else	elsif
end	entity	exit	fairness
file	for	force	function
generate	generic	group	guarded
if	impure	in	inertial
inout	is	label	library
linkage	literal	loop	map
mod	nand	new	next
nor	not	null	of
on	open	or	others
out	package	parameter	port
postponed	procedure	process	property
protected	pure	range	record
register	reject	release	rem
report	restrict	restrict_guarantee	return
rol	ror	select	sequence
severity	signal	shared	sla

sll	sra	srl	strong
subtype	then	to	transport
type	unaffected	units	until
use	variable	vmode	vprop
vunit	wait	when	while
with	xnor	xor	

VHDL-2019 Language Support

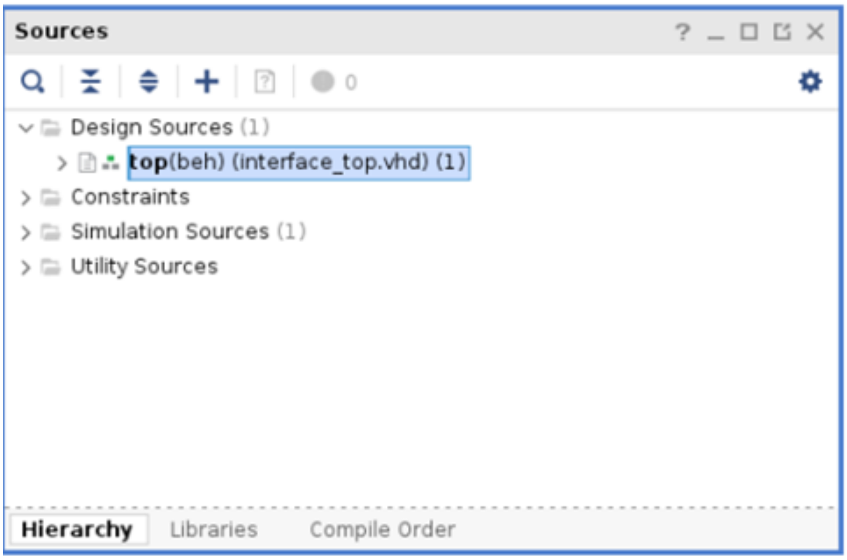
Introduction

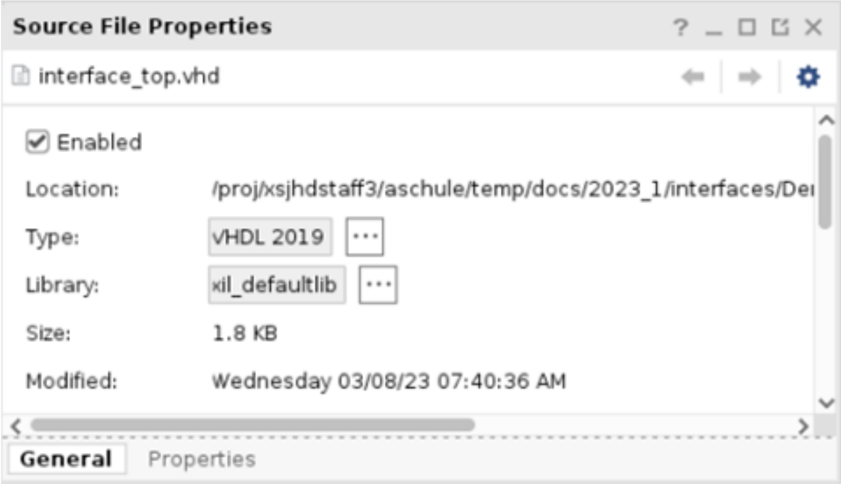
AMD Vivado™ synthesis supports a synthesizable subset of the VHDL-2019 standard. The following section describes what is supported and how to use it.

Setting up Vivado to use VHDL-2019

There are a few ways to setup Vivado to compile VHDL files as VHDL-2019. The first is in the IDE. Go to the Source file Properties window and set the Type to VHDL-2019 from the drop-down of available file types.

Figure: Source File Properties





☑ Enabled

Location: /proj/xsjhdstaff3/aschule/temp/docs/2023_1/interfaces/De

Type: VHDL 2019

Library: xil_defaultlib

Size: 1.8 KB

Modified: Wednesday 03/08/23 07:40:36 AM

You can also set files to VHDL-2019 with the `set_property` command in the Tcl Console. The syntax is as follows:

```
set_property FILE_TYPE {VHDL 2019} [get_files <file.vhd>]
```

For the non-project or Tcl flow, the command for reading in VHDL-2019 is:

```
read_vhdl -vhdl2019 <file.vhd>
```

If you want to read in more than one file, you can either use multiple `read_vhdl` commands or multiple files with one command, as follows:

```
read_vhdl -vhdl2019 {a.vhd b.vhd c.vhd}
```

Supported VHDL-2019 Features

Vivado supports the following VHDL-2019 features.

✎ **Note:** This is a relatively new language for Vivado. New features in this language will be added each release and will be added to the documentation

---

Interfaces

VHDL-2019 Interfaces are implemented using the `record` and `view` keywords. The type record is used to set up the interface, for example:

```
type data is record
     A : std_logic_vector(3 downto 0);
     B : std_logic_vector(3 downto 0);
     C : std_logic_vector(3 downto 0);
end record data;
```

Then the view acts like the SystemVerilog modport to indicate which signal act like inputs and which act like outputs:

```
view TxView of data is
     A : in;
     B : in;
     C : out;
end view TxView;
```

Then these views can be used for the port declarations of the hierarchies:

```
entity my_ent is
Port (
     Int_1 : view TxView;
     Int_2 : view RxView
);
end entity my_ent;
```

Conditional Identifiers

Vivado synthesis supports conditional identifiers that can be used to control synthesis based on the tool or the version.
For example, the following code creates an extra output if the tool is a synthesis tool :

```
entity my_ent is port(
clk : in std_logic;
in1, in2 : in std_logic;
`if TOOL_TYPE = "SYNTHESIS" then
new_out : out std_logic;
`end if
out1 : out std_logic;
```

Vivado synthesis supports the following conditional identifiers.

Table: Conditional Identifiers

Identifier	Value
VHDL_VERSION	Same as in VHDL compile version
TOOL_TYPE	“SYNTHESIS”
TOOL_VENDOR	“AMD/XILINX”
TOOL_NAME	“Vivado”
TOOL_EDITION	“ML Editions”
TOOL_VERSION	Current tool version

✎ **Note:** The values for these identifiers are case sensitive.

64-bit Integers

For VHDL-2019, integer types are 64-bit instead of 32-bit. This is automatic and the RTL does not need to be changed to take advantage of this.

Verilog Language Support

Introduction

This chapter describes the AMD Vivado™ synthesis support for the Verilog Hardware Description Language.
Coding examples are included in this chapter. Download the coding example files from Coding Examples.

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 be used to custom-program a programmable logic device such as a Virtex device.
    - Different synthesis methods are used for:
- Arithmetic blocks
- Interconnect logic
- Finite State Machine (FSM) components

For information about basic Verilog concepts, see the IEEE Verilog HDL Reference Manual.

Verilog-2001 Support

Vivado synthesis supports the following Verilog-2001 features.

- Generate statements
- Combined port/data type declarations
- ANSI-style port list
- Module operator 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.
The following table lists the variable part selects symbols.

Table: 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. The following table lists the concepts of hardware structure and their descriptions.

Table: Basic Concepts of Hardware Structure

Concept	Description
Component	Building or basic block
Port	Component I/O connector
Signal	Corresponds to a wire between components

The following table lists the Verilog Components, the view, and what the components describe.

Table: 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, which are instantiated to build larger logic circuits. The set of logical functions described by the built-in logic gates includes:

- AND
- OR
- XOR
- NAND
- NOR
- NOT

2-Input XOR Function Example

In this coding example, each instance of the built-in modules has a unique instantiation name such as `a_inv`, `b_inv`, and `out`.

```
module build_xor (a, b, c);
input a, b;
output c;
wire c, a_not, b_not;

not a_inv (a_not, a);not b_inv (b_not, b);and a1 (x, a_not, b);and a2 (y,
b_not, a);or out (c, x, y);
endmodule
```

Half-Adder Example

This coding example shows the structural description of a half-Adder composed of four, 2-input nand modules.

```
module halfadd (X, Y, C, S);
input X, Y;
```

```
output C, S;
wire S1, S2, S3;

nand NANDA (S3, X, Y);nand NANDB (S1, X, S3);nand NANDC (S2, S3, Y);nand
NANDD (S, S1, S2);assign C = S3;
endmodule
```

Instantiating Pre-Defined Primitives

The structural features of Verilog allow you to design circuits by instantiating pre-defined primitives such as: gates, registers, and AMD-specific primitives such as `CLKDLL` and `BUFG`.
These primitives are additional to those included in Verilog, and are supplied with the AMD Verilog libraries (unisim_comp.v).

Instantiating an FDC and a BUFG Primitive Example

The unisim_comp.v library file includes the definitions for FDC and BUFG .

```
module example (sysclk, in, reset, out);
input sysclk, in, reset;
output out;
reg out;
wire sysclk_out;

FDC register (out, sysclk_out, reset, in); //position based referencing
BUFG clk (.O(sysclk_out),.I(sysclk)); //name based referencing
```

Verilog Parameters

Verilog parameters do the following:

- 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 operator 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.

Parameters Example (Verilog)

Download the coding example files from Coding Examples.

Filename: parameter_1.v

```
// A Verilog parameter allows to control the width of an instantitated
// block describing register logic
//
//
// File:parameter_1.v
//
module myreg (clk, clken, d, q);

parameter SIZE = 1;

input clk, clken;
input [SIZE-1:0] d;
output reg [SIZE-1:0] q;

always @(posedge clk)
begin
if (clken)
q <= d;
end

endmodule

module parameter_1 (clk, clken, di, do);

parameter SIZE = 8;

input clk, clken;
input [SIZE-1:0] di;
output [SIZE-1:0] do;

myreg #8 inst_reg (clk, clken, di, do);

endmodule
```

Parameter and Generate-For Example (Verilog)

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

Filename: parameter_generate_for_1.v

```
//
// A shift register description that illustrates the use of parameters
and
// generate-for constructs in Verilog
//
// File: parameter_generate_for_1.v
//
module parameter_generate_for_1 (clk, si, so);

parameter SIZE = 8;

input clk;
input si;
output so;

reg [0:SIZE-1] s;

assign so = s[SIZE-1];

always @ (posedge clk)
s[0] <= si;

genvar i;
generate
for (i = 1; i < SIZE; i = i+1)
begin : shreg
always @ (posedge clk)
begin
s[i] <= s[i-1];
end
end
endgenerate

endmodule
```

Verilog Parameter and Attribute Conflicts

Verilog parameter and attribute conflicts can arise because of the following:

- Parameters and attributes can be applied to both instances and modules in the Verilog code.
- Attributes can also be specified in a constraints file.

Verilog Usage Restrictions

Verilog usage restrictions in Vivado synthesis include the following:

- Case Sensitivity
- Blocking and Non-Blocking Assignments
- Integer Handling

Case Sensitivity

Vivado synthesis supports Verilog case sensitivity despite the potential of name collision.

- Because Verilog is case-sensitive, the names of modules, instances, and signals can theoretically be made unique by changing capitalization.
  - Vivado synthesis can synthesize a design in which instance and signal names differ only by capitalization.
  - Vivado synthesis errors out when module names differ only by capitalization.
- Do not rely on capitalization alone to make object names unique. Capitalization alone can cause problems in mixed language projects.

Blocking and Non-Blocking Assignments

Vivado synthesis supports blocking and non-blocking assignments.

- Do not mix blocking and non-blocking assignments.
- Although Vivado synthesis synthesizes the design without error, mixing blocking and non-blocking assignments can cause errors during simulation.

For more information about the Verilog format for Vivado simulation, see *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Unacceptable Example One

Do not mix blocking and non-blocking assignments for different bits of the same signal.

```
always @(in1)
begin
if (in2)
out1 = in1;

end else
out1 <= in2;
```

Unacceptable Example Two

Do not mix blocking and non-blocking assignments for different bits of the same signal.

```
if (in2)
begin
out1[0] = 1'b0;
out1[1] <= in1;
end else begin
out1[0] = in2;
out1[1] <= 1'b1;
end
```

Integer Handling

Vivado synthesis handles integers differently from other synthesis tools in some situations. In those instances, the integers must be coded in a particular way.

Integer Handling in Verilog Case Statements

Unsized integers in case item expressions can cause unpredictable results.

Integer Handling in Verilog Case Statements Example

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

```
reg [2:0] condition1; always @(condition1) begin
case(condition1)
4 : data_out = 2; // Generates faulty logic
3'd4 : data_out = 2; // Does work
endcase
end
```

Integer Handling in Concatenations

Unsigned integers in Verilog concatenations can cause unpredictable results. If you use an expression that results in an unsized integer, it does the following:

- Assign the expression to a temporary signal.
- Use the temporary signal in the concatenation.

```
reg [31:0] temp;
assign temp = 4'b1111 % 2;
assign dout = {12/3,temp,din};
```

Verilog-2001 Attributes and Meta Comments

Verilog-2001 Attributes

- Verilog-2001 attributes pass specific information to programs such as synthesis tools.
- Verilog-2001 attributes are generally accepted.
- Specify Verilog-2001 attributes anywhere for operators or signals, within module declarations and instantiations.
- Although the compiler might support other attribute declarations, Vivado synthesis ignores them.
- Use Verilog-2001 attributes to set constraints on:
  - Individual objects, such as:
    - Module
    - Instance
    - Net
  - Set the following synthesis constraints:
    - Full Case
    - Parallel Case

Verilog Meta Comments

- Verilog meta comments are understood by the Verilog parser.
- Verilog meta comments set constraints on individual objects, such as:
  - Module
  - Instance
  - Net
- Verilog meta comments set directives on synthesis:
  - `parallel_case` and `full_case`
  - `translate_on` and `translate_off`
  - All tool specific directives (for example, `syn_sharing`)

Verilog Meta Comment Support

Vivado synthesis supports:

- C-style and Verilog style meta comments:
  - C-style

    ```
    /* ...*/
    ```
- C-style comments can be multiple line:
  - Verilog style

    ```
    // ...
    ```

Verilog style comments end at the end of the line.

- Translate Off and Translate On

```
// synthesis translate_on
// synthesis translate_off
```

- Parallel Case

```
// synthesis parallel_case full_case
// synthesis parallel_case
// synthesis full_case
```

- Constraints on individual objects

Verilog Meta Comment Syntax

```
// synthesis attribute [of] ObjectName [is] AttributeValue
```

Verilog Meta Comment Syntax Examples

```
// synthesis attribute RLOC of u123 is R11C1.S0
// synthesis attribute HUSET u1 MY_SET
// synthesis attribute fsm_extract of State2 is "yes"
// synthesis attribute fsm_encoding of State2 is "gray"
```

Verilog Constructs

The following table lists the support status of Verilog constructs in Vivado synthesis.

Table: Verilog Constructs

Verilog Constants	Support Status
Constant	
Integer	Supported
Real	Supported
String	Unsupported
Verilog Data Types	
Net types: • tri0 • tri1 • trireg	Unsupported

Verilog Constants	Support Status
- wand - wor	Supported
All Drive strengths	Ignored
Real and realtime registers	Unsupported
All Named events	Unsupported
Delay	Ignored
Verilog Procedural Assignments	
assign	Supported with limitations. See Using assign and deassign Statements.
deassign	Supported with limitations. See Using 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
Verilog Design Hierarchies	
module definition	Supported
macromodule definition	Unsupported

Verilog Constants	Support Status
hierarchical names	Supported [1]
defparam	Supported
array of instances	Supported
configurations	Supported
Verilog Compiler Directives	
`celldefine `endcelldefine	Ignored
`default_nettype	Supported
`define	Supported
`ifdef `else `endif	Supported
`undef, `ifndef, `elsif	Supported
`include	Supported
`resetall	Ignored
`timescale	Ignored
`unconnected_drive `nounconnected_drive	Ignored
`uselib	Unsupported
`file, `line	Supported

1. The processing for hierarchical names is done post-elaboration. Because of this, the connections are not seen in the elaborated view. They only start appearing in the post-synthesis view.

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: System Tasks and Status

System Task or Function	Status	Comment
$display	Limited Support	

System Task or Function	Status	Comment
$fclose	Not Supported	
$fdisplay	Ignored	
$fgets	Not Supported	
$finish	Ignored	
$fopen	Ignored	
$fscanf	Ignored	Escape sequences are limited to %b and %d
$fwrite	Ignored	
$monitor	Ignored	
$random	Ignored	
$readmemb	Supported	
$readmemh	Supported	
$signed	Supported	
$stop	Ignored	
$strobe	Ignored	
$time	Ignored	
$unsigned	Supported	
$write	Not Supported	
$clog2	Supported	This is supported with SystemVerilog only.
$floor	Limited Support	For parameters only.
$ceil	Limited Support	For parameters only.
$rtoi	Supported	
$itor	Supported	
$bits	Supported	
$bitstoreal	Supported	
$realtobits	Supported	

System Task or Function	Status	Comment
$bitstoshortreal	Supported	
$shortrealtobits	Supported	
$unpacked_dimensions	Supported	
$dimensions	Supported	
$left	Supported	
$right	Supported	
$low	Supported	
$high	Supported	
$increment	Supported	
$size	Supported	
$countones	Supported	
$countbits	Supported	
$onehot	Supported	
$onehot0	Supported	
$isunknown	Supported	
$asin	Supported	
$acos	Supported	
$atan	Supported	
$atan2	Supported	
$sinh	Supported	
$cosh	Supported	
$tanh	Supported	
$sin	Supported	
$asinh	Supported	
$cos	Supported	
$ascosh	Supported	

System Task or Function	Status	Comment
$tan	Supported	
$ln	Supported	
$atanh	Supported	
$log10	Supported	
$exp	Supported	
$sqrt	Supported	
$hypot	Supported	
$pow	Supported	
$fatal	Supported	
$warning	Supported	
$error	Supported	
$info	Supported	
all others	Ignored	

Using Conversion Functions

Use the following syntax to call `$signed` and `$unsigned` system tasks on any expression.

```
$signed(expr) or $unsigned(expr)
```

- The return value from these calls is the same size as the input value.
- The sign of the return value is forced regardless of any previous sign.

Loading Memory Contents With File I/O Tasks

Use the `$readmemb` and `$readmemh` system tasks to initialize block memories.

- Use `$readmemb` for binary representation.
- Use `$readmemh` for hexadecimal representation.
- Use index parameters to avoid behavioral conflicts between Vivado synthesis and the simulator.

```
$readmemb("rams_20c.data",ram, 0, 7);
```

Supported Escape Sequences

- %h
- %d
- %o
- %b
- %c
- %s

Verilog Primitives

Vivado synthesis supports Verilog gate-level primitives except as shown in the Table 1. Vivado synthesis does not support Verilog switch-level primitives, such as the following:

```
cmos, nmos, pmos, rcmos, rnmos, rpmos rtran, rtranif0, rtranif1, tran,
tranif0, tranif1
```

Gate-Level Primitive Syntax

```
gate_type instance_name (output, inputs,...);
```

Gate-Level Primitive Example

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

Unsupported Verilog Gate Level Primitives

The following table lists the gate-level primitives that are not supported in Vivado synthesis.

Table: Unsupported Primitives

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

Verilog Reserved Keywords

The following table lists the reserved keywords. Keywords marked with an asterisk (*) are reserved by Verilog and are not supported by Vivado synthesis.

Table: Verilog Reserved Keywords

always	and	assign	automatic
begin	buf	bufif0	bufif1
case	casex	casez	cell*
cmos	config*	deassign	default
defparam	design*	disable	edge
else	end	endcase	endconfig*
endfunction	endgenerate	endmodule	endprimitive
endspecify	endtable	endtask	event
for	force	forever	fork
function	generate	genvar	highz0
highz1	if	ifnone	incdir*
include*	initial	inout	input
instance*	integer	join	larger
liblist*	library*	localparam	macromodule
medium	module	nand	negedge
nmos	nor	noshow-cancelled*	not
notif0	notif1	or	output
parameter	pmos	posedge	primitive
pull0	pull1	pullup*	pulldown*
pulsestyle_ondetect*	pulsestyle_onevent*	rcmos	real
realtime	reg	release	repeat
rnmos	rpmos	rtran	rtranif0
rtranif1	scalared	show-cancelled*	signed
small	specify	specpa	strong0
strong1	supply0	supply1	table
task	time	tran	tranif0

tranif1	tri	tri0	tri1
triand	trior	trireg	use*
vectored	wait	wand	weak0
weak1	while	wire	wor
xnor	xor		

Behavioral Verilog

Vivado synthesis supports the behavioral Verilog Hardware Description Language (HDL), except as otherwise noted.

Variables in Behavioral Verilog

- Variables in behavioral Verilog are declared as an integer.
- These declarations are used in test code only. Verilog provides data types such as `reg` and `wire` for actual hardware description.
- The difference between reg and wire depends on whether the variable is given its value in a procedural block (`reg`) or in a continuous assignment (`wire`).
  - Both `reg` and `wire` have a default width of one bit (scalar).
  - To specify an N-bit width (vectors) for a declared reg or wire, the left and right bit positions are defined in square brackets separated by a colon.
  - In Verilog-2001, `reg` and `wire` data types can be signed or unsigned.

Variable Declarations Example

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

Initial Values

Initialize registers in Verilog-2001 when they are declared.

- The initial value:
  - Is a constant.
  - Cannot depend on earlier initial values.
  - Cannot be a function or task call.
  - Can be a parameter value propagated to the register.
  - Specifies all bits of a vector.
- When you assign a register as an initial value in a declaration, Vivado synthesis sets this value on the output of the register at global reset or power up.
- When a value is assigned in this manner:
  - The value is carried in the Verilog file as an `INIT` attribute on the register.
  - The value is independent of any local reset.

Assigning an Initial Value to a Register

Assign a set/reset (initial) value to a register.

- Assign the value to the register when the register reset line goes to the appropriate value. See the following coding example.
- When you assign the initial value to a variable:
  - The value is implemented as a Flip-Flop, the output of which is controlled by a local reset.
  - The value is carried in the Verilog file as an FDP or FDC Flip-Flop.

Initial Values Example One

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

Initial Values Example Two

```
always @(posedge clk) begin
if (rst)
arb_onebit <= 1'b0;
end
```

Arrays of Reg and Wire

Verilog allows arrays of `reg` and `wire`.

Arrays Example One

This coding example describes an array of 32 elements. Each element is 4 bits wide.

```
reg [3:0] mem_array [31:0];
```

Arrays Example Two

This coding example describes an array of 64 8-bit wide elements. These elements can be assigned only in structural Verilog code.

```
wire [7:0] mem_array [63:0];
```

Multi-Dimensional Arrays

Vivado synthesis supports multi-dimensional array types of up to two dimensions.

- Multi-dimensional arrays can be:
  - Any net
  - Any variable data type
- Code assignments and arithmetic operations with arrays.
- You cannot select more than one element of an array at one time.
- You cannot pass multi-dimensional arrays to:
  - System tasks or functions
  - Regular tasks or functions

Multi-Dimensional Array Example One

This coding example describes an array of 256 x 16 wire elements of 8 bits each. These elements can be assigned only in structural Verilog code.

```
wire [7:0] array2 [0:255][0:15];
```

Multi-Dimensional Array Example Two

This coding example describes an array of 256 x 8 register elements, each 64-bits wide. These elements can be assigned in behavioral Verilog code.

```
reg [63:0] regarray2 [255:0][7:0];
```

Data Types

The Verilog representation of the `bit` data type contains the following values:

- 0 = logic zero
- 1 = logic one
- x =unknown logic value
- z = high impedance

Supported Data Types

- net
  - wire
  - wand
  - wor
- registers
  - reg
  - integer
- constants
  - parameter
  - Multi-dimensional arrays (memories)

Net and Registers

Net and Registers can be either:

- Single bit (scalar)
- Multiple bit (vectors)

Behavioral Data Types Example

This coding example shows sample Verilog data types found in the declaration section of a Verilog module.

```
wire net1; // single bit net
reg r1; // single bit register
tri [7:0] bus1; // 8 bit tristate bus
reg [15:0] bus1; // 15 bit register
reg [7:0] mem[0:127]; // 8x128 memory register
parameter state1 = 3'b001; // 3 bit constant
parameter component = "TMS380C16"; // string
```

Legal Statements

Vivado synthesis supports behavioral Verilog legal statements.

- The following statements (variable and signal assignments) are legal:
    - variable = expression
    - if (condition) statement
    - else statement
    - case (expression), for example:

      ```
      expression: statement
      ...
      default: statement
      endcase
      ```

    - for (variable = expression; condition; variable = variable + expression) statement
    - while (condition) statement
    - forever statement
    - functions and tasks
- All variables are declared as integer or reg.
- A variable cannot be declared as a wire.

Expressions

Behavioral Verilog expressions include:

- Constants
- Variables with the following operators:
    - arithmetic
    - logical
        - bitwise
        - logical
    - relational
    - conditional

Logical Operators

The category (bitwise or logical) into which a logical operator falls depends on whether it is applied to an expression involving several bits, or a single bit.

Supported Operators

Table: Supported Operators

Arithmetic	Logical	Relational	Conditional
+	&	<	?
-	&&	==	
*	|	===	

Arithmetic	Logical	Relational	Conditional
**	||	<=	
/	^	>=	
%	~	>=	
	~^	!=	
	^~	!==	
	<<	>	
	>>		
	<<<		
	>>>		

Supported Expressions

Table: Supported Expressions

Expression	Symbol	Status
Concatenation	{}	Supported
Replication	{{}}	Supported
Arithmetic	+, -, *,**	Supported
Division	/	Supported
Modulus	%	Supported
Addition	+	Supported
Subtraction	-	Supported
Multiplication	*	Supported
Power	**	Supported:

Expression	Symbol	Status
		- Both operands are constants, with the second operand being non-negative. - If the first operand is a 2, 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

Expression	Symbol	Status
Reduction AND	&	Supported
Reduction NAND	~&	Supported
Reduction OR	|	Supported
Reduction NOR	~|	Supported
Reduction XOR	^	Supported
Reduction XNOR	~^, ^~	Supported
Left Shift	<<	Supported
Right Shift Signed	>>>	Supported
Left Shift Signed	<<<	Supported
Right Shift	>>	Supported
Conditional	?:	Supported
Event OR	or, ','	Supported

Evaluating Expressions

The (===) and (!==) operators in the following table are:

- Special comparison operators.
- Used in simulation to see if a variable is assigned a value of (x) or (z).
- Treated as (==) or (!=) by synthesis.

See *Vivado Design Suite User Guide: Logic Simulation* (UG900) for more information about Verilog format for Vivado simulation.

Evaluated Expressions Based on Most Frequently Used Operators

Table: 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

a b	a==b	a===b	a!=b	a!==b	a&b	a&&b	a|b	a||b	a^b
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, as follows:

- Block statements group statements together. They are designated by begin and end keywords. Block statements 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 and *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
- End with an `endmodule` statement.

Circuit I/O Ports

- The circuit I/O ports are listed in the module declaration.
- Each circuit I/O port is characterized by:
  - A name
  - A mode: `Input`, `Output`, `Inout`
  - Range information if the port is of `array` type.

Behavioral Verilog Module Declaration Example One

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

Behavioral Verilog Module Declaration Example Two

```
module example ( input A, inputB, output O
);

assign O = A & B;
endmodule
```

Module Instantiation

A behavioral Verilog module instantiation statement does the following:

- Defines an instance name.
- Contains a port association list. The port association list specifies how the instance is connected in the parent module. Each element of the port association list ties a formal port of the module declaration to an actual net of the parent module.
- Is instantiated in another module. See the following coding example.

Behavioral Verilog Module Instantiation Example

```
module top (A, B, C, O); input A, B, C; output O;
wire tmp;

example inst_example (.A(A), .B(B), .O(tmp));

assign O = tmp | C;

endmodule
```

Continuous Assignments

Vivado synthesis supports both explicit and implicit continuous assignments.

- Continuous assignments model combinatorial logic in a concise way.
- Vivado synthesis ignores delays and strengths given to a continuous assignment.
- Continuous assignments are allowed on wire and tri data types only.

Explicit Continuous Assignments

Explicit continuous assignments start with an assign keyword after the net has been separately declared.

```
wire mysignal;
...
assign mysignal = select ? b : a;
```

Implicit Continuous Assignments

Implicit continuous assignments combine declaration and assignment.

```
wire misignal = a | b;
```

Procedural Assignments

- Behavioral Verilog procedural assignments:
    - Assign values to variables declared as reg.
    - Are introduced by always blocks, tasks, and functions.
    - Model registers and Finite State Machine (FSM) components.
- Vivado synthesis supports:
    - Combinatorial functions
    - Combinatorial and sequential tasks
    - Combinatorial and sequential always blocks

Combinatorial Always Blocks

Combinatorial logic is modeled efficiently by Verilog time control statements:

- Delay time control statement [#]
- Event control time control statement [@]

Delay Time Control Statement

The delay time control statement [# (pound)] is:

- Relevant for simulation only.
- Ignored for synthesis.

For more information on Verilog format for Vivado simulation, see *Vivado Design Suite User Guide: Logic Simulation* (UG900).

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 a * (asterisk) in the parentheses for a list of signals, the `always` block is activated for an event in any of the `always` block's signals as described.
- In combinatorial processes, if a signal is not explicitly assigned in all branches of if or case statements, Vivado synthesis generates a latch to hold the last value.
- The following statements are used in a process:
  - variable and signal assignments
  - if-else statements
  - case statements
  - for-while loop statements
  - function and task calls

Using if-else Statements

Vivado synthesis supports `if-else` statements.

- The `if-else` statements use true and false conditions to execute statements.
  - If the expression evaluates to true, the first statement is executed.
  - If the expression evaluates to false, x, or z, the else statement is executed.
- A block of multiple statements is executed using `begin` and end keywords.
- if-else statements can be nested.

Example of if-else Statement

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;
else
outmux = a;
end
endmodule
```

Case Statements

Vivado synthesis supports case statements.

- A case statement performs a comparison to an expression to evaluate one of several parallel branches.
    - The case statement evaluates the branches in the order they are written.
    - The first branch that evaluates to true is executed.
    - If none of the branches matches, the default branch is executed.
- Do not use unsized integers in case statements. Always size integers to a specific number of bits. Otherwise, results can be unpredictable.
- `casez` treats all z values in any bit position of the branch alternative as a don't care.
- `casex` treats all x and z values in any bit position of the branch alternative as a don't care.
- The question mark (?) can be used as a don't care in either the `casez` or `casex` case statements.

Multiplexer Case Statement Example (Verilog)

Filename: top.v

```
// Multiplexer using case statement
module mux4 (sel, a, b, c, d, outmux);
input [1:0] sel;
input [1:0] a, b, c, d;
output [1:0] outmux;
reg [1:0] outmux;

always @ *
begin
case(sel)
2'b00 : outmux = a;
2'b01 : outmux = b;
2'b10 : outmux = c;
2'b11 : outmux = d;
endcase
end
endmodule
```

Avoiding Priority Processing

- The case statement in the previous coding example evaluates the values of `input sel` in priority order.
- To avoid priority processing:
  - Use a parallel-case Verilog attribute to ensure parallel evaluation of the `input sel`.
  - Replace the case statement with:

    ```
    (* parallel_case *) case(sel)
    ```

For and Repeat Statements

Vivado synthesis supports `for` and `repeat` statements. When using `always` blocks, repetitive or bit slice structures can also be described using a `for` statement, or a `repeat` statement.

Using 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.

Using While Loops

When using `always` blocks, use `while` loops to execute repetitive procedures.

- A `while` loop:
  - Is not executed if the test expression is initially false.
  - Executes other statements until its test expression becomes false.
- The test expression is any valid Verilog expression.
- To prevent endless loops, use the `-loop_iteration_limit` option.
- A `while` loop can have `disable` statements. The `disable` statement is used inside a labeled block, as shown in the following code snippet:

```
disable <blockname>
```

Example of While Loop

```
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
```

Using 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 Examples

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
```

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
```

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;
elsif (ASET == 1'b1) DO <= 8'b11111111;
else

DO <= DI;
endmodule
```

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
```

Using 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 Examples

Filename: functions_1.v

```
//
// 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
```

Filename: task_1.v

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
```

Using 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.

Example of Recursive Tasks and Functions

```
function automatic [31:0] fac;
input [15:0] n;
if (n == 1)
fac = 1;

else

fac = n * fac(n-1); //recursive function call
endfunction
```

Using Constant Functions and Expressions

Vivado synthesis supports function calls to calculate constant values. Constants are assumed to be decimal integers.

- Specify constants in binary, octal, decimal, or hexadecimal.
- To specify constants explicitly, prefix them with the appropriate syntax.

Example of Constant Functions

Filename: functions_contant.v

```
// 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
```

Example of Constant Expressions

The following constant expressions represent the same value.

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

Using Blocking and Non-Blocking Procedural Assignments

Blocking and non-blocking procedural assignments have time control built into their respective assignment statements.

- The pound sign (`#`) and the at sign (`@`) are time control statements.
- These statements delay execution of the statement following them until the specified event is evaluated as true.
- The pound (`#`) delay is ignored for synthesis.

Blocking Procedural Assignment Syntax Example One

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

Blocking Procedural Assignment Syntax Example Two (Alternate)

```
if (in1) out = 1'b0;
else out = in2;
```

This assignment blocks the current process from continuing to execute additional statements at the same time, and is used mainly in simulation.
For more information regarding Verilog format for Vivado simulation, see *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Non-Blocking Procedural Assignment Syntax Example One

```
variable <= @(posedge_or_negedge_bit) expression;
```

Non-blocking assignments evaluate the expression when the statement executes, and allow other statements in the same process to execute at the same time. The variable change occurs only after the specified delay.

Non-Blocking Procedural Assignment Example Two

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

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

Verilog Macros

Verilog defines macros as follows:

```
`define TESTEQ1 4'b1101
```

The defined macro is referenced later, as follows:

```
if (request == 'TESTEQ1)
```

The `` `ifdef `` and `` `endif `` constructs do the following:

- Determine whether a macro is defined.
- Define conditional compilation.

If the macro called out by `` `ifdef `` is defined, that code is compiled.

- If the macro has not been defined, the code following the `` `else `` command is compiled.
- The `` `else `` is not required, but `` `endif `` must complete the conditional statement.

Use the Verilog Macros command line option to define (or redefine) Verilog macros.

- Verilog Macros let you modify the design without modifying the HDL source code.
- Verilog Macros is useful for IP core generation and flow testing.

Macro Example One

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

Macro Example Two

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

✎ **Note:** When synthesis runs, Vivado automatically sets the SYNTHESIS macro. So, when using ‘ifdef SYNTHESIS, it is triggered during the synthesis run.

Include Files

Verilog allows you to separate HDL source code into more than one file. To reference the code in another file, use the following syntax in the current file.

```
`include <path/file-to-be-included>
```

The previous line takes the contents of the file to be included and inserts it all into the current file at the line with the `` `include ``.
The path can be a relative or an absolute path. In the case of a relative path, the Verilog compiler looks in two different places for the file to be included.

- The first is relative to the file with the `` `include `` statement. The compiler looks there, and if it can find the file, it inserts the contents of the file there.
- The second place it looks is relative to the `-include_dirs` option in the Verilog options section of the General settings.

Multiple `` `include `` statements are allowed in the same Verilog file.

Behavioral Verilog Comments

Behavioral Verilog comments are similar to the comments in such languages as C++.

One-Line Comments

One-line comments start with a double forward slash (//).

```
// This is a one-line comment.
```

Multiple-Line Block Comments

Multiple-line block comments start with /* and end with */.

```
/* This is a multiple-line comment.
*/
```

Generate Statements

Behavioral Verilog generate statements:

- Allow you to create:
    - parameterized and scalable code.
    - Repetitive or scalable structures.
    - Functionality conditional on a particular criterion being met.
- Are resolved during Verilog elaboration.
- Are conditionally instantiated into your design.
- Are described within a module scope.
- Start with a `generate` keyword.
- End with an `endgenerate` keyword.

Structures Created Using Generate Statements

Structures likely to be created using a generate statement include:

- Primitive or module instances
- Initial or always procedural blocks
- Continuous assignments
- Net and variable declarations
- parameter redefinitions
- Task or function definitions

Supported Generate Statements

Vivado synthesis supports all Behavioral Verilog generate statements:

- generate-loop (generate-for)
- generate-conditional (generate-if-else)
- generate-case (generate-case)

Generate Loop Statements

Use a generate-for loop to create one or more instances that can be placed inside a module.
Use the generate-for loop the same way you use a normal Verilog for loop, with the following limitations:

- The generate-for loop index has a genvar variable.
- The assignments in the for loop control refers to the genvar variable.
- The contents of the for loop are enclosed by begin and end statements.
- The begin statement is named with a unique qualifier.

Generate Loop Statement 8-Bit Adder Example

```
generate genvar i;
for (i=0; i<=7; i=i+1)
begin : for_name
adder add (a[8*i+7 : 8*i], b[8*i+7 : 8*i], ci[i], sum_for[8*i+7 : 8*i],
c0_or[i+1]);
end
endgenerate
```

Generate Conditional Statements

A `generate-if-else` statement conditionally controls which objects are generated.

- Each branch of the `if-else` statement is enclosed by begin and end statements.
- The begin statement is named with a unique qualifier.

Generate Conditional Statement Coding Example

This coding example instantiates two different implementations of a multiplier based on the width of data words.

```
generate
if (IF_WIDTH < 10)
begin : if_name
multiplier_imp1 # (IF_WIDTH) u1 (a, b, sum_if);
end
else
begin : else_name
multiplier_imp2 # (IF_WIDTH) u2 (a, b, sum_if);
```

```
end
endgenerate
```

Generate Case Statements

A generate-case statement conditionally controls which objects are generated under which conditions.

- Each branch in a generate-case statement is enclosed by begin and end statements.
- The begin statement is named with a unique qualifier.

Behavioral Verilog Generate Case Statements Coding Example

This coding example instantiates more than two different implementations of an adder based on the width of data words.

```
generate
case (WIDTH)
1:
begin : case1_name
adder #(WIDTH*8) x1 (a, b, ci, sum_case, c0_case);
end
2:
begin : case2_name
adder #(WIDTH*4) x2 (a, b, ci, sum_case, c0_case);
end default:
begin : d_case_name
adder x3 (a, b, ci, sum_case, c0_case);
end
endcase
endgenerate
```

SystemVerilog Support

Introduction

AMD Vivado™ synthesis supports the subset of SystemVerilog RTL that can be synthesized. The following sections describe those data types.

Targeting SystemVerilog for a Specific File

By default, the Vivado synthesis tool compiles *.v files with the Verilog 2001 syntax and *.sv files with the SystemVerilog syntax.

To target SystemVerilog for a specific *.v file in the Vivado IDE, right-click the file, and select Source Node Properties. In the Source File Properties window, change the File Type to SystemVerilog, and click OK.

Tcl Command to Set Properties

Alternatively, you can use the following Tcl command in the Tcl Console:

```
set_property file_type SystemVerilog [get_files <filename>.v]
```

The following sections describe the supported SystemVerilog types in the Vivado IDE.

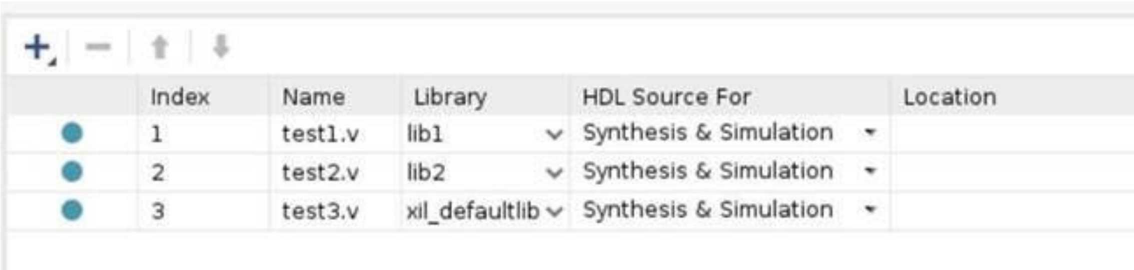
Compilation Units

System Verilog supports both single file and multiple file compilation through use of Compilation units.
A compilation unit is a collection of one or more SV source files compiled together. Every compilation unit is associated with single library. The compilation unit scope is a scope that is local to a global compilation unit, the scope has all the declarations that lie outside of any other design scope. Generally functions, tasks, parameter, nets, variables, and user defined types declared outside the module, interface, package or program come under the compilation unit scope.
For example, consider the following design.
In Tcl mode

```
read_verilog -lib lib1 {test1.sv }
read_verilog -lib lib2 {test2.sv }
read_verilog test3.sv
```

Or IDE

Figure: IDE



	Index	Name	Library	HDL Source For	Location
●	1	test1.v	lib1	Synthesis & Simulation	
●	2	test2.v	lib2	Synthesis & Simulation	
●	3	test3.v	xil_defaultlib	Synthesis & Simulation	

In the previous case, if test1.sv has declarations in the compilation unit scope such as params, typedefs, and so on. Following is an example.

```
Parameter P1 =2; // parameter declared out of module scope
module test1 (<port list>)
...
```

```
...
endmodule
```

and read the files as mentioned previously. Compiler unit scope starts with reading file test1.sv under lib1, but while reading test2.sv with lib2 would be illegal because compilation unit should be associated with single library. This can be addressed by following ways:
In Tcl mode, putting all the files in a single library.

```
read_verilog -lib lib1 {test1.sv}
read_verilog -lib lib1 {test2.sv}
read_verilog test3.sv
```

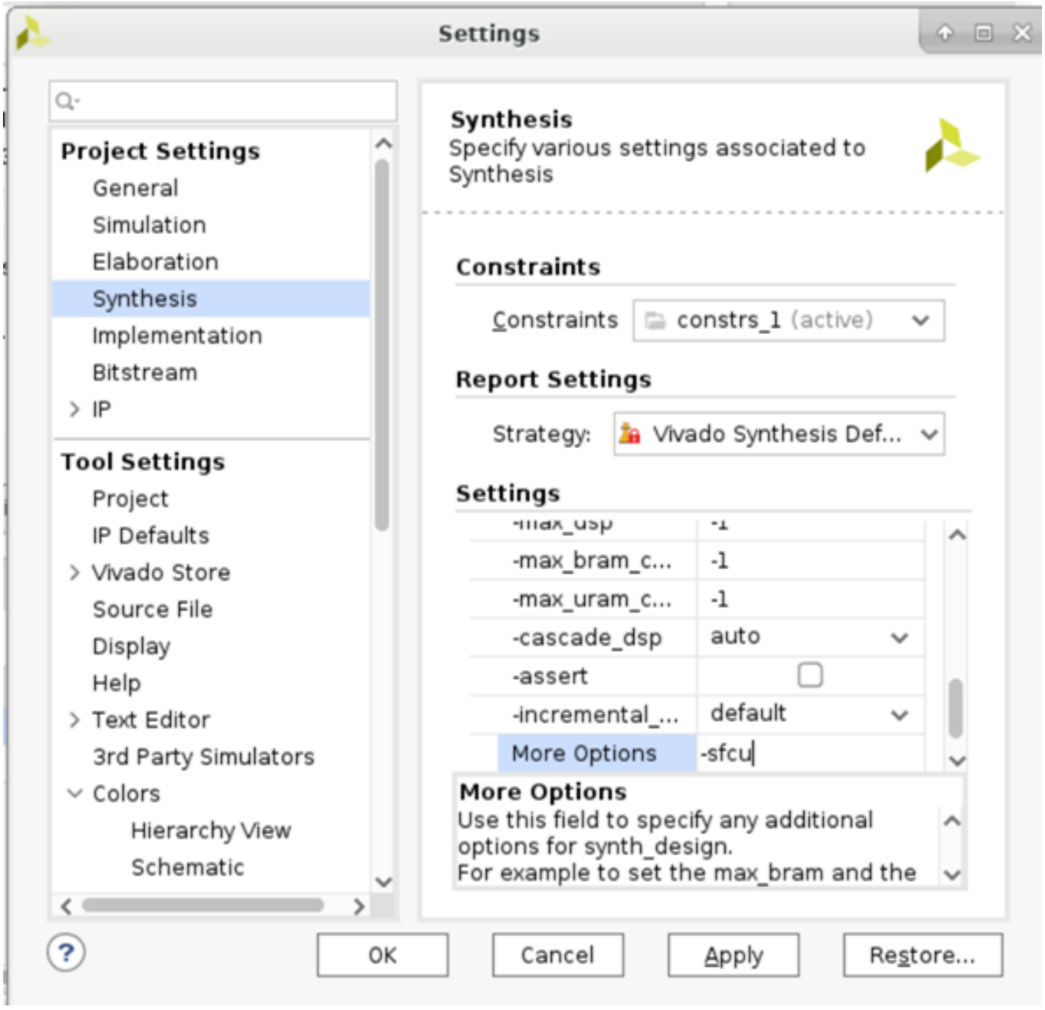
or not declaring libraries at all

```
read_verilog {test1.sv }
read_verilog {test2.sv}
read_verilog test3.sv
```

or (single file compilation unit mode)

```
read_verilog -lib lib1 {test1.sv}
read_verilog -lib lib2 {test2.sv}
read_verilog test3.sv
synth_design -top <top_name> -sfcu
```

Figure: Settings



Synthesis
Specify various settings associated to Synthesis

Constraints

Constraints: constrs_1 (active)

Report Settings

Strategy: Vivado Synthesis Def...

Settings

-max_dsp	-1
-max_bram_c...	-1
-max_uram_c...	-1
-cascade_dsp	auto
-assert	☐
-incremental_...	default
More Options	-sfcu

More Options
Use this field to specify any additional options for synth_design.
For example to set the max_bram and the

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.

See *Vivado Design Suite User Guide: Logic Simulation* (UG900) for more information about Verilog format for simulation.

Real Numbers

Synthesis supports real numbers; however, they cannot be used to create logic. They can only be used as parameter values. The SystemVerilog-supported real types are:

- real
- shortreal
- realtime

Void Data Type

The void data type is only supported for functions that have no return value.

User-Defined Types

Vivado synthesis supports user-defined types, which are defined using the typedef keyword. Use the following syntax:

```
typedef data_type type_identifier {size};
```

or

```
typedef [enum, struct] type_identifier;
```

Enum Types

Enumerated types can be declared with the following syntax:

```
enum [type] {enum_name1, enum_name2...enum_namex} identifier
```

If no type is specified, the `enum` defaults to `int`. Following is an example:

```
enum {sun, mon, tues, wed, thurs, fri, sat} day_of_week;
```

This code generates an `enum` of `int` with seven values. The values that are given to these names start with 0 and increment, so that, `sun = 0` and `sat = 6`.
To override the default values, use code as in the following example:

```
enum {sun=1, mon, tues, wed, thurs, fri, sat} day_of week;
```

In this case, `sun` is 1 and `sat` is 7.
The following is another example how to override defaults:

```
enum {sun, mon=3, tues, wed, thurs=10, fri=12, sat} day_of_week;
```

In this case, `sun=0`, `mon=3`, `tues=4`, wed=5, `thurs=10`, `fri=12`, and `sat=13`.
Enumerated types can also be used with the typedef keyword.

```
typedef enum {sun,mon,tues,wed,thurs,fri,sat} day_of_week; day_of_week
my_day;
```

The preceding example defines a signal called `my_day` that is of type `day_of_week`.
You can also specify a range of `enums`. For example, the preceding example can be specified as:

```
enum {day[7]} day_of_week;
```

This creates an enumerated type called `day_of_week` with seven elements as follows: day`0`, day1...day6.
Following are other ways to use enumerated types:

```
enum {day[1:7]} day_of_week; // creates day1,day2...day7
enum {day[7] = 5} day_of_week; //creates day0=5, day1=6... day6=11
```

Constants

SystemVerilog gives three types of elaboration-time constants:

- `parameter`: Is the same as the original Verilog standard and can be used in the same way.
- `localparameter`: 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 runtime 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 single section of data that can be referenced in different ways. The format for specifying a union is:

```
typedef union packed {union_member1; union_member2...union_memberx;}
unions_name;
```

Packed and Unpacked Arrays

Vivado synthesis supports both packed and unpacked arrays:

```
logic [5:0] sig1; //packed array logic sig2 [5:0]; //unpacked array
```

Data types with predetermined widths do not need the packed dimensions declared:

```
integer sig3; //equivalent to logic signed [31:0] sig3
```

Processes

Always Procedures

There are four `always` procedures:

- always
- always_comb
- always_latch
- always_ff

The procedure `always_comb` describes combinational logic. A sensitivity list is inferred by the logic driving the `always_comb` statement.
For `always` you must provide the sensitivity list. The following examples use a sensitivity list of `in1` and `in2`:

```
always@(in1 or in2)
out1 = in1 & in2;
always_comb out1 = in1 & in2;
```

The procedure `always_latch` provides a quick way to create a latch. Like `always_comb`, a sensitivity list is inferred, but you must specify a control signal for the latch enable, as in the following example:

```
always_latch
if(gate_en) q <= d;
```

The procedure `always_ff` is a way to create Flip-Flops. Again, you must specify a sensitivity list:

```
always_ff@(posedge clk)
out1 <= in1;
```

Block Statements

Block statements provide a mechanism to group sets of statements together. Sequential blocks have a begin and end around the statement. The block can declare its own variables, and those variables are specific to that block. The sequential block can also have a name associated with that block. The format is as follows:

```
begin [: block name]
[declarations]
[statements]
end [: block name]
begin : my_block logic temp;
temp = in1 & in2; out1 = temp;
end : my_block
```

In the previous example, the block name is also specified after the end statement. This makes the code more readable, but it is not required.

✎ **Note:** Parallel blocks (or fork join blocks) are not supported in Vivado synthesis.

Procedural Timing Controls

SystemVerilog has two types of timing controls:

- Delay control: Specifies the amount of time between the statement its execution. This is not useful for synthesis, and Vivado synthesis ignores the time statement while still creating logic for the assignment.
- Event control: Makes the assignment occur with a specific event; for example, `always@(posedge clk)`. This is standard with Verilog, but SystemVerilog includes extra functions.

The logical `or` operator is an ability to give any number of events so that any event triggers the execution of the statement. To do this, use either a specific `or`, or separate with commas in the sensitivity list. For example, the following two statements are the same:

```
always@(a or b or c)
always@(a,b,c)
```

SystemVerilog also supports the implicit `event_expression @*`. This helps to eliminate simulation mismatches caused because of incorrect sensitivity lists.
For example:

```
Logic always@* begin
```

See *Vivado Design Suite User Guide: Logic Simulation* (UG900) for the Verilog format for simulation.

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;
elsif (expression2)
command2;
elsif (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 statements 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 and evaluates the expression. If the expression evaluates to 0, it stops and executes; otherwise, if it 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, and 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 to make connections between modules is 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
```

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`. According to the module, this is because 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
```

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 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 specify the whole package.

SystemVerilog Constructs

The following table lists the SystemVerilog constructs. Constructs that are not supported are shaded in gray.

Table: SystemVerilog Constructs

Construct	Status
Data type	
Singular and aggregate types	Supported
Nets and variables	Supported

Construct	Status
Variable declarations	Supported
Vector declarations	Supported
2-state (two-value) and 4-state (four-value) data types	Supported
Signed and unsigned integer types	Supported
User-defined types	Supported
Enumerations	Supported
Defining new data types as enumerated types	Supported
Enumerated type ranges	Supported
Type checking	Supported
Enumerated types in numerical expressions	Supported
Enumerated type methods	Supported
Type parameters	Supported
Type operator	Supported
Cast operator	Supported
Bitstream casting	Supported
Const constants	Supported
$cast dynamic casting	Not Supported
Real, shortreal, and realtime data types	Supported
Aggregate data types	
Structures	Supported
Packed/Unpacked structures	Supported
Assigning to structures	Supported
Packed arrays	Supported
Unpacked arrays	Supported

Construct	Status
Operations on arrays	Supported
Multidimensional arrays	Supported
Indexing and slicing of arrays	Supported
Array assignments	Supported
Arrays as arguments to subroutines	Supported
Array manipulation methods (those that do not return queue type)	Not Supported
Array querying functions	Not Supported
Unpacked unions	Supported
Tagged unions	Not Supported(1)
Packed unions	Supported
Processes	
Combinational logic always_comb procedure	Supported
Implicit always_comb sensitivities	Supported
Latched logic always_latch procedure	Supported
Sequential blocks	Supported
Sequential logic always_ff procedure	Supported
Iff event qualifier	Supported
Aliases	Supported
Conditional event controls	Not Supported
Parallel blocks	Not Supported
Procedural timing controls	Not Supported
Sequence events	Not Supported
Assignment statement	
The continuous assignment statement	Supported

Construct	Status
Variable declaration assignment (variable initialization)	Supported
Assignment-like contexts	Supported
Array assignment patterns	Supported
Structure assignment patterns	Supported
Unpacked array concatenation	Supported
Net aliasing	Not Supported
Operators and expressions	
$error, $warning, $info	Supported only within initial blocks, and can only be used to evaluate constant expressions; for example, parameters.
Aggregate expressions	Supported
Arithmetic expressions with unsigned and signed types	Supported
Assignment operators	Supported
Assignment within an expression	Supported
Concatenation operators	Supported
Constant expressions	Supported
Increment and decrement operators	Supported
Operations on logic (4-state) and bit (2-state) types	Supported
Wildcard equality operators	Supported
Concatenation of stream_expressions	Supported
Operators with real operands	Not Supported
Re-ordering of the generic stream	Not Supported
Set membership operator	Not Supported
Streaming concatenation as an assignment target (unpack)	Supported

Construct	Status
Streaming dynamically sized data	Not Supported
Procedural programming statement	
Case statement violation reports and multiple processes	Supported
Loop statements	Supported
Unique-if, unique0-if and priority-if	Supported
Assert Statements	Not Supported
If statement violation reports and multiple processes	Not Supported
Jump statements	Not Recommended
Pattern matching conditional statements	Not Supported
Set membership case statement	Not Supported
unique-case, unique0-case, and priority-case	Supported
Violation reports generated by unique-if, unique0-if, and priority-if constructs	Not Supported
Tasks	
Coverage control functions	Not Supported
Static and Automatic task	Supported
Tasks memory usage and concurrent activation	Not Supported
Functions	
Return values and void functions	Supported
Static and Automatic function	Supported
Constant function	Supported
Background process spawned by function call	Not Supported
Virtual Functions	Not Supported

Construct	Status
Subroutine calls and argument passing	
Argument binding by name	Supported
Default argument value	Supported
Pass by reference	Supported
Pass by value	Supported
Optional argument list	Not Supported
Compiler Directives	
	Supported
Modules and Hierarchy	
Default port values	Supported
External modules	Supported
Module instantiation syntax	Supported
Member selects	Supported
Overriding module parameters	Supported
Top-level modules and $root	Supported
Binding auxiliary code to scopes or instances	Not Supported
Hierarchical names	Supported
Upwards name referencing	Not Supported
Interfaces	
Interface syntax	Supported
Modport expressions	Supported
Parameterized interfaces	Supported
Ports in interfaces	Supported
Array of interface	Supported
Clocking blocks and modports	Not Supported

Construct	Status
Dynamic Arrays	Not Supported
Example of exporting tasks and functions	Not Supported
Example of multiple task exports	Not Supported
Interfaces and specify blocks	Not Supported
Nested interface	Not Supported
Virtual interfaces	Not Supported
Packages	
Package declarations	Supported
Referencing data in packages	Supported
Using packages in module headers	Supported
Exporting imported names from packages	Supported
The std built-in package	Not Supported
Generate constructs	
	Supported
config statements	
	Supported
Class	
Instances	Supported
Member and method access	Supported
Constructors	Supported
Static class member and methods	Supported
Access using 'this' and 'super'	Supported
Object assignment	Supported
Inheritance	Supported
Data hiding and encapsulation	Supported

Construct	Status
Scope and resolution operator (::)	Supported
Nested classes	Supported
Objects inside structs	Supported
Virtual Classes	Not Supported
Abstract classes	Not Supported
Assignment with base class object	Not Supported
Object comparison with NULL	Not Supported

1. If used, tagged is ignored, and the tool produces a warning message.

Mixed Language Support

Introduction

AMD Vivado™ synthesis supports VHDL and Verilog mixed language projects except as otherwise noted.

Mixing VHDL and Verilog

The VHDL and Verilog files that make up a project are specified in a unique HDL project file. The rules for mixing VHDL and Verilog are, as follows:

- Mixing VHDL and Verilog is restricted to design unit (cell) instantiation.
- A Verilog module can be instantiated in VHDL code and a VHDL entity can be instantiated in 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. See VHDL and Verilog Boundary Rules.
- Vivado synthesis binds VHDL design units to a Verilog module during HDL elaboration.

Instantiation

For instantiation, the following rules apply:

- 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

Instantiating VHDL in Verilog

To instantiate a VHDL design unit in a Verilog design, do the following:

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.

Instantiating Verilog in VHDL

To instantiate a Verilog module in a VHDL design, do the following:

1. Declare a VHDL component with the same name as the Verilog module to be instantiated. VHDL direct entity instantiation is not supported when instantiating a Verilog module.
2. Observe case sensitivity.
3. Instantiate the Verilog component as if you were instantiating a VHDL component.
    - Binding a component to a specific design unit from a specific library by using a VHDL configuration declaration is not supported. Only the default Verilog module binding is supported.
    - The only Verilog construct that can be instantiated in a VHDL design is a Verilog module. No other Verilog constructs are visible to VHDL code.
    - During elaboration, Vivado synthesis treats all components subject to default binding as design units with the same name as the corresponding component name.
    - During binding, Vivado synthesis treats a component name as a VHDL design unit name and searches for it in the logical library work.
        - If Vivado synthesis finds a VHDL design unit, Vivado synthesis binds it.
        - If Vivado synthesis does not find a VHDL design unit, it treats the component name as a Verilog module name and searches for it using a case sensitive search. 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.

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 override is named and not ordered. The parameter override occurs through instantiation, not through `defparam`.

Acceptable Example

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

Unacceptable Example

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

Verilog in VHDL

Vivado synthesis has the following limitations when instantiating a Verilog module in a VHDL design unit:

- 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 `defparam`.
- Only component instantiation is supported when instantiating a Verilog module in VHDL. Direct entity instantiation is not supported.

VHDL and Verilog Libraries

For libraries with mixed VHDL and Verilog, libraries are handled as follows:

- 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

VHDL and Verilog boundary rules are, as follows:

- The boundary between VHDL and Verilog is enforced at the design unit level.
- A VHDL entity or architecture can instantiate a Verilog module. See Instantiating VHDL in Verilog in the following section.
- A Verilog module can instantiate a VHDL entity. See Instantiating Verilog in VHDL.

Binding

Vivado synthesis performs binding during elaboration. During binding, the following actions occur:

1. Vivado synthesis searches for a Verilog module with the same name as the instantiated module with a user-specified list of unified logical libraries and with a user-specified order.
2. Vivado synthesis ignores any architecture name specified in the module instantiation.
3. If Vivado synthesis finds the Verilog module, synthesis binds the name.
4. If Vivado synthesis does not find the Verilog module, it treats the Verilog module as a VHDL entity, and 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. This assumes that a VHDL design unit is stored with an extended identifier.

Generics Support

Vivado synthesis supports the following VHDL generic types and their Verilog equivalents for mixed language designs: integer, real, string, boolean.

Port Mapping

Vivado synthesis supports port mapping for VHDL instantiated in Verilog and Verilog instantiated in VHDL.

Port Mapping for VHDL Instantiated in Verilog

When a VHDL entity is instantiated in a Verilog module, formal ports can have the following characteristics:

- Allowed directions: `in`, `out`, `inout`
- Unsupported directives: `buffer`, `linkage`
- Allowed data types: `bit`, `bit_vector`, `std_logic`, `std_ulogic`, `std_logic_vector`, `std_ulogic_vector`

Port Mapping for Verilog Instantiated in VHDL

When a Verilog module is instantiated in a VHDL entity or architecture, formal ports can have the following characteristics:

- Allowed directions are: `input`, `output`, and `inout`.
- Allowed data types are: `wire` and `reg`
- Vivado synthesis does not support:
    - Connection to bidirectional pass options in Verilog.
    - Unnamed Verilog ports for mixed language boundaries.

Use an equivalent component declaration to connect to a case sensitive port in a Verilog module. Vivado synthesis assumes Verilog ports are in all lowercase.

Additional Resources and Legal Notices

Finding Additional Documentation

Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to https://docs.amd.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select Help > Documentation and Tutorials.
- On Windows, click the Start button and select Xilinx Design Tools > DocNav.
- At the Linux command prompt, enter `docnav`.

✎ **Note:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the Design Hubs View tab.
- Go to the Design Hubs web page.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Support.

References

These documents provide supplemental material useful with this guide:

Vivado Documentation

1. *UltraScale Architecture and Product Data Sheet: Overview* (DS890)
2. *7 Series DSP48E1 Slice User Guide* (UG479)
3. *UltraScale Architecture Memory Resources User Guide* (UG573)
4. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
5. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
6. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
7. *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894)
8. *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
9. *Vivado Design Suite User Guide: Designing with IP* (UG896)
10. *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899)
11. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
12. *Vivado Design Suite User Guide: Using Constraints* (UG903)
13. *Vivado Design Suite User Guide: Implementation* (UG904)
14. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)
15. *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907)
16. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
17. *ISE to Vivado Design Suite Migration Guide* (UG911)
18. *Vivado Design Suite Properties Reference Guide* (UG912)
19. *Vivado Design Suite Tutorial: Using Constraints* (UG945)
20. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
21. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
22. *Vivado Design Suite Tutorial: Creating, Packaging Custom IP* (UG1119)

23. Vivado Design Suite Documentation

Synthesis Coding Examples

Coding Examples

Training Resources

1. Vivado Design Suite QuickTake Video: Synthesis Options
2. Vivado Design Suite QuickTake Video: Creating and Managing Runs
3. Vivado Design Suite QuickTake Video: Advanced Synthesis using Vivado
4. Vivado Design Suite QuickTake Video Tutorials

Revision History

The following table shows the revision history for this document.

Section	**Revision Summary**
12/11/2024 Version 2024.2	
CRITICAL_SIG_OPT	Added new topic
List of Linter Rules	Added a new topic
Running Synthesis with Tcl	Updated the topic
Example of if-else Statement	Updated the topic
Conditional Identifiers	Updated the table in the topic
06/28/2024 Version 2024.1	
Targeting SystemVerilog for a Specific File	Updated the topic
VHDL-2008 RESERVED Words	Updated the topic
Asymmetric RAMs	Updated the topic
RAM_DECOMP	Updated the topic
MAX_FANOUT XDC Examples	Updated the topic
SystemVerilog Constructs	Updated the topic
Using Synthesis Settings	Updated the topic

Section	Revision Summary
Running Synthesis with Tcl	Updated the topic

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright