Notice of Disclaimer
The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available “AS IS” and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of the Limited Warranties which can be viewed at http://www.xilinx.com/warranty.htm; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in Critical Applications: http://www.xilinx.com/warranty.htm#critapps.

© Copyright 2013 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.
# Revision History

The following table shows the revision history for this document:

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/30/2013</td>
<td>2013.3</td>
<td>Added FSM_SAFE_STATE attribute with examples. Made correction to KEEP_HIERARCHY VHDL example code. Made corrections to MAX_FANOUT and VHDL examples. Made correction to usage instruction for SHREG_EXTRACT. Added SRL_STYLE attribute with examples. Added section on Using Synthesis Attributes in XDC files.</td>
</tr>
<tr>
<td>12/18/2013</td>
<td>2013.4</td>
<td>Added list of supported languages for Vivado synthesis to the Introduction and updated Setting a Bottom-Up Flow Using the Out-of-Context Flow in Chapter 1, Vivado Synthesis. Added RTL and XDC usage details for various attributes, updated GATED_CLOCK Verilog Example, added information about static SRLs and precedence to SRL_STYLE, and updated Using Synthesis Attributes in XDC files. Added additional DONT_TOUCH Verilog Examples and DONT_TOUCH VHDL Examples in Chapter 2, Synthesis Attributes. Added SystemVerilog information to Targeting SystemVerilog for a Specific File and updated Enum Types in Chapter 3, SystemVerilog Support. Updated 32-Bit Dynamic Shift Registers VHDL Coding Example, added Simple Dual-Port Block RAM Examples, updated True-Dual-Port BRAM with Byte Write Enable Examples, added FIR Filters, and changed Multiply-Add and Multiply-Accumulate examples in Chapter 4, HDL Coding Techniques.</td>
</tr>
</tbody>
</table>
# Table of Contents

**Revision History** ................................................................. 3

**Chapter 1: Vivado Synthesis**
- Introduction ........................................................................... 6
- Synthesis Methodology ......................................................... 7
- Using Synthesis ..................................................................... 7
- Viewing Floorplanning and Utilization Reports ......................... 27
- Exploring the Logic ............................................................... 30
- Running Synthesis with Tcl .................................................. 36

**Chapter 2: Synthesis Attributes**
- Introduction ........................................................................... 39
- Supported Attributes ............................................................ 39

**Chapter 3: SystemVerilog Support**
- Introduction ........................................................................... 56
- Targeting SystemVerilog for a Specific File .............................. 56
- Data Types ............................................................................ 56
- Processes .............................................................................. 61
- Procedural Programming Assignments .................................... 63
- Tasks and Functions ............................................................... 65
- Modules and Hierarchy .......................................................... 66
- Interfaces ............................................................................. 67

**Chapter 4: HDL Coding Techniques**
- Introduction ........................................................................... 71
- Advantages of VHDL ............................................................. 71
- Advantages of Verilog ............................................................ 71
- Advantages of SystemVerilog ................................................ 72
- Flip-Flops, Registers, and Latches .......................................... 72
- Latches ................................................................................. 76
- 3-states ............................................................................... 78
- Shift Registers .................................................................... 81
Dynamic Shift Registers ............................................................. 85
Multipliers .............................................................................. 88
Multiply-Add and Multiply-Accumulate ..................................... 90
FIR Filters ............................................................................ 94
RAM HDL Coding Techniques ................................................ 96

Appendix A: Additional Resources
Xilinx Resources ..................................................................... 140
Solution Centers .................................................................... 140
Vivado Documentation ............................................................ 140
Introduction

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

- SystemVerilog
- Verilog
- VHDL
- Mixed languages

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

**IMPORTANT:** Vivado synthesis does not support UCF constraints. Migrate UCF constraints to XDC constraints. For more information, see the "UCF to XDC Constraints Conversion" in the ISE to Vivado Design Suite Migration Guide (UG911) [Ref 6].

There are two ways to setup and run synthesis:

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

See the **Vivado Design Suite User Guide: Design Flows Overview** (UG892) [Ref 7] for more information about operation modes. This chapter covers both modes in separate subsections.
Synthesis Methodology

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

Within the Vivado IDE, you can do the following:

- Create and save strategies. Strategies are configurations of command options, that you can apply to design runs for synthesis or implementation. See Creating Run Strategies, page 11.
- Queue the synthesis and implementation runs to launch sequentially or simultaneously with multi-processor machines. See Running Synthesis, page 16.
- Monitor synthesis or implementation progress, view log reports, and cancel runs. See Monitoring the Synthesis Run, page 23.

Using Synthesis

This section describes using the Vivado Integrated Design Environment (IDE) to set up and run Vivado synthesis. The corresponding Tcl Console commands are below each Vivado IDE procedure.

See the following guides for more information regarding Tcl commands, and using Tcl:

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

To set the synthesis options for the design, from the **Synthesis** section of the Flow Navigator:

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

   ![Flow Navigator: Synthesis](image1)

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

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

   ![Project Settings Dialog Box](image2)

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

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

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

      - Timing 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.


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

   - **TCL Command**: `-constrset <arg>`
b. From the Options area: Select a Strategy from the drop-down menu where you can view and select a predefined synthesis strategy to use for the synthesis run.

You can also define your own strategy. When you select a synthesis strategy, available Vivado strategy displays in the dialog box. You can override synthesis strategy settings by changing the option values as described in Creating Run Strategies, page 11.

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

Table 1-1: Vivado Default and RunTimeOptimized Strategies

<table>
<thead>
<tr>
<th>Options</th>
<th>Default</th>
<th>Flow_RunTimeOptimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>-flatten_hierarchy</td>
<td>rebuilt</td>
<td>none</td>
</tr>
<tr>
<td>-gated_clock_conversion</td>
<td>off</td>
<td>off</td>
</tr>
<tr>
<td>-bufg</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>-fanout_limit</td>
<td>10,000</td>
<td>10,000</td>
</tr>
<tr>
<td>-directive</td>
<td>default</td>
<td>RunTimeOptimized</td>
</tr>
<tr>
<td>-fsm_extraction</td>
<td>auto</td>
<td>off</td>
</tr>
<tr>
<td>-keep_equivalent_registers</td>
<td>unchecked</td>
<td>unchecked</td>
</tr>
<tr>
<td>-resource_sharing</td>
<td>checked (on)</td>
<td>auto</td>
</tr>
<tr>
<td>-control_set_opt_threshold</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-no_lc</td>
<td>unchecked</td>
<td>unchecked</td>
</tr>
<tr>
<td>-shreg_min_size</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

c. Select from the displayed Options, which are:

The tcl.pre and tcl.post options are hooks for Tcl files that run immediately before and after synthesis.

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

See the *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894) [Ref 4] for more information about Tcl scripting.
You can use the DIRECTORY property of the current project or current run to define the relative paths in your scripts:

- **Tcl Command:**
  - `tclcmd`: `get_property DIRECTORY [current_project]`
  - `tclcmd`: `get_property DIRECTORY [current_run]`

**–flatten_hierarchy:** Determines how Vivado synthesis controls hierarchy.

- **none:** Instructs the synthesis tool to never flatten the hierarchy. The output of synthesis has the exact same hierarchy as the original RTL.
- **full:** Instructs the tool to fully flatten the hierarchy leaving only the top level.
- **rebuilt:** When set, `rebuilt` allows the synthesis tool to flatten the hierarchy, perform synthesis, and then rebuild the hierarchy based on the original RTL.

This value allows the QoR benefit of cross-boundary optimizations, with a final hierarchy that is similar to the RTL for ease of analysis.

**–gated_clock_conversion:** Turns on and off the ability of the synthesis tool to convert the clocked logic with enables.

The use of gated clock conversion also requires the use of an RTL attribute to work. See [GATED_CLOCK, page 45](#), for more information.

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

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

**–fanout_limit:** Specifies the number of loads a signal must drive before it starts replicating logic. This global limit is a general guide, and when the tool determines it is necessary, it can ignore the option. If a hard limit is required, see the [MAX_FANOUT option described in the Chapter 2, Synthesis Attributes](#).

**Note:** The `–fanout_limit` switch does not impact control signals (such as set, reset, clock enable): use [MAX_FANOUT](#) to replicate these signals if needed.

**–directive:** Replaces the `effort_level` option. When specified, this option runs Vivado synthesis with different optimizations. Values are `Default` and `RuntimeOptimized`, which runs synthesis quickly and with less optimization.

**–fsm_extraction:** Controls how synthesis extracts and maps finite state machines. The [FSM_ENCODING, page 43](#) and [FSM Description, page 131](#) describe the options in more detail.
-**keep_equivalent_registers**: Prevents merging of registers with the same input logic. See **KEEP, page 46** for more information.

-**resource_sharing**: Sets the sharing of arithmetic operators between different signals. The values are **auto**, **on** and **off**.

The **auto** value sets performing resource sharing to depend on the timing of the design: **on** means that it is always on, and **off** means that it is always off.

-**control_set_opt_threshold**: Sets the threshold for clock enable optimization to the lower number of control sets. The default is 1.

The given value is the number of fanouts necessary for the tool to move the control sets into the \( \text{D} \) logic of a register. If the fanout is higher than the value, the tool attempts to have that signal drive the **control_set_pin** on that register.

-**no_lc**: When checked, this option turns off LUT combining.

**TIP**: Register merging can be prevented using the **KEEP** attribute, which prevents optimizations where signals are either optimized or absorbed into logic blocks. This attribute instructs the synthesis tool to keep the signal it was placed on, and that signal is placed in the netlist. See **KEEP, page 42** for more information.

-**shreg_min_size**: Is the threshold for inference of SRLs. The default setting is 3. This sets the number of sequential elements that would result in the inference of an SRL for fixed delay chains (static SRL). See **SHREG_EXTRACT, page 50** for more information.

**Creating Run Strategies**

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

1. To see the current strategies for the flow, select **Tools > Options** and click the **Strategies** button to open the strategies window,

2. In the **Flow** drop-down, select **Vivado Synthesis**. The options on the right are the same as those shown in the Synthesis Project Settings dialog box.

To create a custom strategy, do one of the following:

- Right-click the **User Defined Strategies > Create New Strategy**.
- Click the **Create New Strategy** button \( \text{➕} \) under the flow options to open the New Strategy dialog box, as shown in Figure 1-3.
From the New Strategy dialog box, name the strategy, set the strategy type, and specify the tool version. There is an input option for a meaningful description. After you set the options, 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, in the Flow Navigator, select the **Add Sources** command to open the Add Sources wizard, shown in Figure 1-4.

![Add Sources Wizard](image)

**Figure 1-4: Add Sources Wizard**

Add constraint, RTL, or other project files. See the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 4] for more information about the Add Source wizard.

The Vivado synthesis tool reads the subset of files that can be synthesized in VHDL, Verilog, or SystemVerilog options supported in the Xilinx tools. Chapter 3, *SystemVerilog Support*, provides details on the supported SystemVerilog constructs.
Vivado synthesis also supports several RTL attributes that control synthesis behavior. Chapter 2, Synthesis Attributes describes these attributes.

Vivado synthesis uses the XDC file for timing constraints.

**IMPORTANT:** Vivado Design Suite does not support the UCF format. See the ISE to Vivado Design Suite Migration Guide (UG911) [Ref 6] for the UCF to XDC conversion procedure.

**Controlling File Compilation Order**

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

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

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

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

- Manages the compilation order as shown in the Compilation Order window
- Shows which modules are used and where they are in the hierarchy tree in the Hierarchy window

The compilation order updates automatically as you change source files.

To modify the compile order before synthesis:

1. Set **Hierarchy Update > Automatic Update, Manual Compile Order** so that the Vivado IDE automatically determines the best top module for the design and allows manual specification of the compilation order.

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

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

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

Verilog typically requires that you place an `include statement at the top of any Verilog source file that references content from a another Verilog or header file. Designs that use common header files might require multiple `include statements to be repeated across multiple Verilog sources used in the design.

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

1. In the Sources window, select the file and open the popup menu.
2. Select the Set Global Include command, or use the Global Include checkbox in the Source Node Properties window, as shown in Figure 1-5, page 15.
TIP: In Verilog, reference header files that are specifically applied to a single Verilog source (for example, a particular `define macro), with an `include statement instead of marking it as a global `include file.

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

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

- Xilinx device to target during synthesis
- Constraint set to apply
- Options to launch single or multiple synthesis runs
- Options to control the results of the synthesis engine

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

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

2. Select **Synthesis**, **Implementation**, or **Both**, and Click **Next**.

   The Create New Runs dialog box opens, as shown in Figure 1-6.

3. Select one of the options: **Synthesis**, **Implementation**, or **Both**, and click **Next**.

   The Launch Options dialog box opens, as shown in Figure 1-7, page 17.

![Create New Runs Dialog Box](image)
4. In the Launch Options dialog box, set the options, as follows, then click **Next**.

   - In the **Launch Directory** drop-down menu, browse to, and select the directory from which to launch the run.

   - In the **Options** area, choose one of the following:
     - **Launch Runs on Local Host**: Lets you run the options from the machine on which you are working. The **Number of jobs** drop-down lets you specify how many runs to launch.
     - **Launch Runs on Remote Hosts**: Lets you launch the runs on a remote host (Linux only) and configure that host.

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

     - **Generate scripts only**: Lets you generate scripts to run later. Use `runme.bat` (Windows) or `runme.sh` (Linux) to start the run.

     - **Do not launch now**: Lets you save the settings that you defined in the previous dialog boxes and launch the runs at a later time.

   The Choose Synthesis Strategies dialog box opens.

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

   The Vivado IDE contains a default strategy. You can set a specific name for the strategy run or accept the default name(s), which are numbered as `synth_1`, `synth_2`, and so forth.
For more information about constraints, see the *Vivado Design Suite User Guide: Using Constraints (UG903)* [Ref 8].

To create your own run strategy, see *Creating Run Strategies, page 11*. After setting the Create New Runs wizard option and starting a run, you can see the results in the Design Runs window, as shown in Figure 1-8.

**Using the Design Runs Window**

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

If the Design Runs window is not already displayed, select **Window > Design Runs** to open the Design Runs window.

A synthesis run can have multiple implementation runs. To expand and collapse synthesis runs, use the tree widgets in the window.

The Design Runs window reports the run status, (when the run is not started, is in progress, is complete, or is out-of-date).

Runs become out-of-date when you modify source files, constraints, or project settings. To reset or delete specific runs, right-click the run and select the appropriate command.

**Setting the Active Run**

Only one synthesis run and one implementation run can be active in the Vivado IDE at any time. All the reports and tab views display the information for the active run. The Project Summary window only displays compilations, resource, and summary information for the active run.

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

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

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

The first two options start the active synthesis run. The third option opens the Launch Selected Runs window. Here, you can select to run on local host, run on a remote host, or generate the scripts to be run.

See "Appendix A" of the Vivado Design Suite User Guide: Implementation (UG904) [Ref 5], for more information about using remote hosts.

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

Setting a Bottom-Up Flow Using the Out-of-Context Flow

You can set a bottom-up flow by selecting any HDL object to run as a separate Out-of-Context module as follows:

1. Right-click the object, and select Set as Out-of-Context Module, as shown in Figure 1-9.

Figure 1-9: Set As Out-of-Context Option
2. In the Set as Out-of-Context Module dialog box, do the following:
   a. In the Block File Set Name field, enter a name for the out-of-context run.
   b. Optionally, use the **Use auto-generated blackbox stub for top level synthesis** option to create a stub file for use with the synthesis run that instantiates the out-of-context module.

   **Note:** The stub file shows the inputs and outputs for the black-box attribute set. By default, this option is selected. If you deselect this option, you must create your own stub file or component statement for the run.

   ![Set as Out-of-Context Module Dialog Box](image)

   **Figure 1-10:** Set as Out-Of-Context Module Dialog Box

3. When you set a flow to **Out-of-Context**, a new run is set up in the tool. To run the option, right-click and select **Launch Runs**, as described in Launching a Synthesis Run, page 19. **Figure 1-11**, page 20, shows the Launch Runs option.

   ![Launch Runs Option](image)

   **Figure 1-11:** Launch Runs Option
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 for later use. The stub file is the lower-level with inputs and outputs and the black-box attribute set. When you run the top-level module again, the bottom-up synthesis inserts the stub file into the flow and compiles the lower-level as a black box. The implementation run inserts the lower-level netlist, thus completing the design.

**CAUTION!** Be careful with this option when the lower-level netlist has parameters or generics that control behavior. When implemented as a bottom-up synthesis, this option uses the default values of parameters or generics. If the module is instantiated more than once, or with overriding parameters, it could result in incorrect logic.

**Note:** This flow is used predominately with Vivado Design Suite IP. For more information, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 9].

Alternatively, you can set the `-mode` to `out_of_context` in the Synthesis Settings > More Options view, as shown in Figure 1-12, page 21. This setting ensures that synthesis does not insert I/O buffers into that module.

![Synthesis Constraint Options](image)

**Figure 1-12:** Synthesis Constraint Options

### Setting Up a Manual Bottom-Up Flow

To manually run a bottom-up flow, instantiate a lower-level netlist as a black box and the Vivado tools fit that black box into the full design after synthesis completes. The following subsections describe the process.
Creating a Lower-Level Netlist

To create a lower-level netlist, set up a project with that netlist as the top level. Before you running synthesis, set the Out-Of-Context mode as shown in Figure 1-12. After you run synthesis, open the Synthesized Design, and in the Tcl Console, type:

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

Instantiating the Lower-Level Netlist in a Design

To run the top-level design with the lower-level netlist, instantiate the lower-level as a black box. To do so, you must provide a description of the port in lower-level to the Vivado tool.

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

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

```vhdl
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:

```verilog
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, and then add the lower-level netlists to the Vivado project.

The Vivado tool inserts the components into the flow after synthesis is run on the top-level, and the Placer and Router tools run as per normal.

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 Figure 1-13. The messages that show in this window during synthesis are also the messages included in the synthesis log file.

![Log Window](image)

Figure 1-13: Log Window

Following Synthesis

After the run is complete, the Synthesis Completed dialog box opens, as shown in Figure 1-14.
Using Synthesis

Select an option:

- **Run Implementation**: Launches implementation with the current Implementation Project Settings.

- **Open Synthesized Design**: Opens the synthesized netlist, the active constraint set, and the target device into Synthesized Design environment, so you can perform I/O pin planning, design analysis, and floorplanning.

- **View Reports**: Opens the Reports window so you can view reports.

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

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

Analyzing Synthesis Results

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

Open the **Reports** view, as shown in Figure 1-15, and select a report for a specific run to see details of the run.
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.

See the following for more information:

- *Vivado Design Suite Tutorial: Design Analysis and Closure Techniques* (UG938) [Ref 10]

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

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

With a synthesized design open, the Vivado IDE opens floorplanning displays, as shown in Figure 1-16, page 26.
From this perspective, examine the design logic and hierarchy, view the resource utilization and timing estimates, or run Design Rule Checks (DRCs).

Figure 1-16:  Floorplan Layout
Viewing Floorplanning and Utilization Reports

When you open the synthesized device view, the Resource estimates display graphically as an expandable hierarchical tree. As each resources type displays, expand it to view each level of logic hierarchy.

Utilization Reports

The Utilization window opens when you open the synthesized design Device window. You can see the reports (depending on your design):

- Summary, by table or chart
- Slice Logic by:
  - LUTS as logic or memory
  - Slice Registers as Flip-Flops or Latches
  - MUXes
- Memory by:
  - Block RAM Tile, RAM type
- DSP
- I/O and GTX Specifics
- Clocking
- Specific Features
- Primitives

Figure 1-17, page 28 shows the Utilization Summary report by chart.
Viewing Reports for Logic Instances

The Vivado IDE provides estimates of the number of device resources contained in the design. To display reports for any logic instance, including the top-level, use the Cell Properties window. Select a top-module or any instance module in the Netlist window, as shown in Figure 1-18.

If the Netlist or Cell Properties do not display, right-click the module, and select Cell Properties from the popup menu. In the Netlist or Cell Properties window, click a view, shown in Figure 1-19, page 29.
The design information in the Instance Properties window varies based on the instance type.

In Figure 1-19, for clkgen, the listings are:

- **General**: Provides the **Name**, **Cell**, and **Type** of the selected instance.
- **Attributes**: Lists file attributes.
- **Instance Pins**: Lists the instance pins by **ID**, **Name**, **Dir**, **BEL Pin**, and **Net**.
- **Aliases**: Lists any aliases.
- **Nodes**: Lists nodes by **ID**, **Name**, **Base Tile**, **Net**, and **Net Conflicts**.
- **PipNodes**: **ID**, **Node**, **Pip**, **Base Tile**, **Net**, and **Net Conflicts**.
- **Connectivity**: Provides **ID** and **Name**.
- **Power**: Shows **Signal Rate** and **%High** settings that you can adjust. A legend becomes available if the settings are calculated.
Exploring the Logic

The Vivado IDE provides several logic exploration perspectives:

- 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.

All windows cross-probe to present the most useful information.

Exploring the Logic Hierarchy

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

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

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

To open the Hierarchy window:

1. In the Netlist window, right-click to bring up the context menu.
2. Select **Show Hierarchy**, as shown in Figure 1-20. You can also press **F6** to open the Hierarchy window.

![Figure 1-20: Show Hierarchy Option](image)
Exploring the Logical Schematic

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

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

To open the Schematic window:

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

The window opens with the selected logic displayed, as shown in **Figure 1-21**.

![Schematic Window](image)

**Figure 1-21**:  **Schematic Window**

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

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

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

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

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

Using the Report Timing Summary Command

To perform a timing analysis, from the Flow Navigator select Netlist Analysis > Synthesized Design > Report Timing Summary. The Report Timing Summary dialog box opens, as shown in Figure 1-22, page 33.
From the Options view, select the options as follows, then click OK:

- **Results name**: Lets you name the report results.
- **Report**: Gives the option of setting the Path delay type to Max, Min, or Min_Max and checkbox options to Report unconstrained paths or Report datasheet.
- **Path Limits**: Selection options are:
  - Maximum number of paths per clock or path group
  - Maximum number of worst paths per endpoint
• **Path Display**: Selection options are:
  - A field to enter **Display paths with slack less than**: with a checkbox for Use default (1e+30).
  - A field to enter **Significant Digits**.

• **Command**: displays the current `report_timing` command encompassing the selected options, and checkboxes to **Open in a new tab** or **Open in Timing Analysis layout**.

**Figure 1-23** shows the Report Timing Summary Advanced view.

![Advanced Report Timing Summary Settings](image)

**Figure 1-23**: Advanced Report Timing Summary Settings

The Advanced view provides the following options:

• **Pins**: A checkbox to **Show input pins in path**.

• **File Output**: A checkbox to write results and a selection field. Options are available to **Overwrite** or **Append** the file.

• **Miscellaneous**: Options are as follows:
  - A Checkbox for **Ignore command errors (quiet mode)**
  - A Checkbox for **Suspend message limits during command execution**.
Figure 1-24 shows the Timer Settings view.

The Timer Settings options are:

- **Interconnect**: A drop-down lists the options of estimated or none.
- **Speed grade**: The options drop-down lists the available speed grades for the targeted device.
- **Multi-Corner Configuration**: Select the Corner name and Delay type.
- A checkbox lets you **Disable flight delays**.

**Additional Reporting Options**

To display a graphical view of device resource estimates, open a synthesized design by clicking **Tools > Report**.

Drop-down menu options show the available reports, as shown in *Figure 1-15, page 25*.
Running Synthesis with Tcl

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

- **Tcl Command**: `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:

![Tools > Report Options](image)

*Figure 1-25: Tools > Report Options*

See the following documents for more information regarding reporting:

- *Vivado Design Suite Tutorial: Design Analysis and Closure Techniques* (UG938) [Ref 10]
The following snippet is an example of the -help output:

```
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]
[-fanout_limit <arg>] [-mode <arg>] [-fsm_extraction <arg>]
[-keep_equivalent_registers] [-resource_sharing <arg>]
[-control_set_opt_threshold <arg>] [-quiet] [-verbose]
```

Returns: design object

Usage:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-name]</td>
<td>Design name</td>
</tr>
<tr>
<td>[-part]</td>
<td>Target part</td>
</tr>
<tr>
<td>[-constrset]</td>
<td>Constraint fileset to use</td>
</tr>
<tr>
<td>[-top]</td>
<td>Specify the top module name</td>
</tr>
<tr>
<td>[-include_dirs]</td>
<td>Specify verilog search directories</td>
</tr>
</tbody>
</table>
| [-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>] |
| [-gated_clock_conversion] | Convert clock gating logic to flop enable. Values: off, on, auto Default: off |
| [-directive]    | Synthesis directive. Values: default, runtimeoptimized Default: default |
| [-rtl]          | Elaborate and open an rtl design |
| [-bufg]         | Max number of global clock buffers used by synthesis. Default: 12 |

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

For boolean, the value for FALSE is as follows:

```
-generc my_gen=1'b0
```

For std_logic vector the value for 0010 is:

```
-generc my_gen=4'b0010
```

**IMPORTANT:** Overriding string generics or parameters is not supported.
A verbose version of the help is available in the Vivado Design Suite: Tcl Command Reference Guide (UG835) [Ref 3]. To determine any Tcl equivalent to a Vivado IDE action, run the command in the Vivado IDE and review the content in the TCL Console or the log file.

The following is an example synth_design Tcl script:

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

### Setting Constraints

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

For details on these commands, see the following documents:

- Vivado Design Suite Tcl Command Reference Guide (UG835) [Ref 3]
- Vivado Design Suite User Guide: Using Constraints (UG903) [Ref 8]
- Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906) [Ref 9]
- Vivado Design Suite Tutorial: Design Analysis and Closure Techniques (UG938) [Ref 10]
- Vivado Design Suite Tutorial: Using Constraints (UG945) [Ref 11]

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

<table>
<thead>
<tr>
<th>Command Type</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timing Constraints</strong></td>
<td><img src="table1-2.png" alt="Timing Constraints Commands" /></td>
</tr>
<tr>
<td><strong>Object Access</strong></td>
<td><img src="table1-2.png" alt="Object Access Commands" /></td>
</tr>
</tbody>
</table>
Synthesis Attributes

Introduction

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

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

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

Supported Attributes

ASYNC_REG

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

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

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

This attribute can be placed on any register; values are FALSE (default) and TRUE. This attribute can be set in the RTL or the XDC.
ASYNC_REG Verilog Example

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

ASYNC_REG VHDL Example

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

BLACK_BOX

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

BLACK_BOX Verilog Example

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

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

BLACK_BOX VHDL Example

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

For more information regarding coding style for Black Boxes, see Black Boxes in Appendix 1.

BUFFER_TYPE

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

By default, Vivado synthesis uses IBUF/BUFG or BUFGPs for clocks and IBUFs for inputs.

Supported values are:

- ibuf: For clock ports where a IBUF/BUFG pair is not wanted. In this case only, the IBUF is inferred for the clock.
- none: Indicates that no input or output buffers are used. A none value on a clock port results in no buffers.

The BUFFER_TYPE attribute can be placed on any top-level port. It can only be set in the RTL. It is not currently supported in the XDC.
Supported Attributes

BUFFER_TYPE Verilog Example

(* buffer_type = "none" *) input in1;  //this will result in no buffers
(* buffer_type = "ibuf" *) input clk1;  //this will result in a clock with no bufg

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 buffer_type : string;
attribute buffer_type of in1 : signal is "none";
end test;

DONT_TOUCH

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

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

The values for DONT_TOUCH are TRUE/FALSE or yes/no. This attribute can be placed on any signal, module, entity, or component.

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

IMPORTANT: Replace KEEP and KEEP_HIERARCHY attributes with DONT_TOUCH.

DONT_TOUCH Verilog Examples

Wire Example

(* dont_touch = "true" *) wire sig1;
assign sig1 = in1 & in2;
assign out1 = sig1 & in2;
Module Example

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

Instance Example

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

DONT_TOUCH VHDL Examples

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;

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;
architecture rtl of top is
attribute dont_touch : string;
attribute dont_touch of rtl : architecture is "true|yes";
beg
... end rtl;

Verilog Example on a Module

(* DONT TOUCH = "yes" *) module test (<port list>);

Verilog Example on an Instance

(* DONT TOUCH = "yes"*) my_mod U0 (<port hookups>);

VHDL Examples

Entity Example

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

Component Example

architecture 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";

FSM_ENCODING

FSM_ENCODING controls encoding on the state machine. Typically, the Vivado tools choose an encoding protocol for state machines based on heuristics that do the best for the most designs. Certain design types work better with a specific encoding protocol.
FSM_ENCODING can be placed on the state machine registers. The legal values for this are "one_hot", "sequential", "johnson", "gray", and "auto". The "auto" value is the default, and allows the tool to determine best encoding. This attribute can be set in the RTL or the XDC.

FSM_ENCODING Verilog Example

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

FSM_ENCODING VHDL Example

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

FSM_SAFE_STATE

FSM_SAFE_STATE tells synthesis to insert logic into the state machine that detects if a there is an illegal state and then puts it into a known state on the next clock cycle. For example, if there were a state machine with a "onehot" encode, and it got into the "0101" state (which is an illegal for "onehot"), the state machine would be able to recover.

FSM_SAFE_STATE can be placed on the state machine registers. The legal values for this are "reset_state" and "power_on_state". Applying the value "reset_state" forces the state machine into the reset state, and "power_on_state" forces it into the power-on state. This attribute can only be set in the RTL. It is not currently supported in the XDC.

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

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

IMPORTANT: This attribute can only be controlled through RTL.

GATED_CLOCK

Vivado synthesis allows the conversion of gated clocks. The two items to use to perform this conversion are:

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

To control the switch:

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

This attribute can only be set in the RTL.

GATED_CLOCK Verilog Example

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

GATED_CLOCK VHDL Example

entity test is port (  
in1, in2 : in std_logic_vector(9 downto 0);  
en : in std_logic;  
clk : in std_logic;  
)
Supported Attributes

out1 : out std_logic_vector( 9 downto 0));
attribute gated_clock : string;
attribute gated_clock of clk : signal is "true";
end test;

IOB

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

IOB Verilog Example

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

IOB VHDL Example

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

KEEP

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

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

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

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

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

- When you have a `MAX_FANOUT` attribute on one signal and a `KEEP` attribute on a second signal that is driven by the first; the `KEEP` attribute on the second signal would not allow fanout replication.

- With a `RAM STYLE= "block"`, when there is a `KEEP` on the register that would need to become part of the RAM, the `KEEP` attribute prevents the block RAM from being inferred.

The supported `KEEP` values are:

- `true`: Keeps the signal.
- `false`: Allows the Vivado synthesis to optimize, if it determines that it should. `false` does not force the tool to remove the signal. The default value is `false`.

This attribute can be placed on any signal, register, or wire.

**RECOMMENDED:** Xilinx recommends setting 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**

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

**KEEP VHDL Example**

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

**KEEP_HIERARCHY**

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

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

**KEEP_HIERARCHY Verilog Example**

**On Module:**

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

**On Instance:**

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

**KEEP_HIERARCHY VHDL Example**

**On Module:**

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

**On Instance:**

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

**MAX_FANOUT**

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

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

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

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

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

On Signal:

(* max_fanout = 50 *) reg sig1;

MAX_FANOUT VHDL Example

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

Note: In VHDL Vivado synthesis, max_fanout is an integer.

PARALLEL_CASE (Verilog Only)

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

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

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

RAM_STYLE

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

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

By default, the tool selects which RAM to infer, based upon heuristics that give the best results for most designs. Place this attribute on the array that is declared for the RAM. This can be set in the RTL or the XDC.

RAM_STYLE Verilog Example

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

**RAM_STYLE VHDL Example**

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

For more information about RAM coding styles, see [RAM HDL Coding Guidelines, page 105](#).

**ROM_STYLE**

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

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

This can be set in the RTL and the XDC.

**ROM_STYLE Verilog Example**

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

**ROM_STYLE VHDL Example**

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

For information about coding for ROM, see [ROM HDL Coding Techniques in Appendix 1](#).

**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;
```
**Supported Attributes**

**SRL_STYLE**

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

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

Place SRL_STYLE on the signal declared for SRL. This attribute can only be set in RTL. It is not currently supported in the XDC. In addition, this attribute can only be used on static SRLs. The indexing logic for dynamic SRLs is located within the SRL component itself. Therefore, the logic cannot be created around the SRL component to look up addresses outside of the component.

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

**SRL_STYLE Verilog Examples**

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

**SRL_STYLE VHDL Examples**

```vhdl
attribute srl_style : string;
attribute srl_style of my_srl : signal is "reg_srl_reg";
```
**TRANSLATE_OFF/TRANSLATE_ON**

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

- synthesis
- synopsys
- pragma

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

This attribute can only be set in the RTL.

**TRANSLATE_OFF/TRANSLATE_ON Verilog Example**

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

**TRANSLATE_OFF/TRANSLATE_ON VHDL Example**

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

---

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

---

**USE_DSP48**

USE_DSP48 instructs the synthesis tool how to deal with synthesis arithmetic structures. By default, mults, mult-add, mult-sub, mult-accumulate type structures go into DSP48 blocks. Adders, subtractors, and accumulators can also go into these blocks but by default, are implemented with the fabric instead of with DSP48 blocks. The USE_DSP48 attribute overrides the default behavior and force these structures into DSP48 blocks.
Accepted values are yes and no. This attribute can be placed in the RTL on signals, architectures and components, entities and modules. The priority is as follows:

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

If the attribute is not specified, the default behavior is for Vivado synthesis to determine the correct behavior.

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

**USE_DSP48 Verilog Example**

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

**USE_DSP48 VHDL Example**

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

**Using Synthesis Attributes in XDC files**

Some synthesis attributes can also be set from an XDC file as well as the original RTL file. In general, attributes that are used in the end stages of synthesis and describe how synthesis creates logic are 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, please see the individual attribute descriptions in this chapter.

To specify synthesis attributes in XDC, use the following syntax:

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

For example:

```
set_property MAX_FANOUT 15 [get_cells in1_int_reg]
```
In addition, you can set these attributes in the elaborated design as follows:

1. Open the elaborated design (Figure 2-2), 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.

2. In the Cell Properties window, click the Properties tab, and do one of the following:
   - Modify the property.
   - If the property does not exist, right-click, select Add Properties, and select the property from the window that appears, or click the + sign.

3. After the properties are set, select File > Save Constraints.

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

Introduction

Vivado® synthesis supports the subset of SystemVerilog RTL that can be synthesized. These data types are described in the following sections.

Targeting SystemVerilog for a Specific File

By default, the Vivado synthesis tool compiles *.v files with the Verilog 2005 syntax and *.sv files with the SystemVerilog syntax.

To target SystemVerilog for a specific *.v file in the Vivado IDE:

1. Right-click the file, and select Source Node Properties.
2. In the Source Node Properties window, change the Type from Verilog to SystemVerilog, and click Apply.

Alternatively, you can use the following Tcl command in the Tcl Console:

```
set_property file_type SystemVerilog [get_files <filename>.v]
```

The following sections describe the supported SystemVerilog types in the Vivado IDE.

Data Types

The following data types are supported, as well as the mechanisms to control them.

Declaration

Declare variables in the RTL as follows:

```
[var] [DataType] name;
```
Where:

- Var is optional and implied if not in the declaration.
- DataType is one of the following:
  - integer_vector_type: bit, logic, or reg
  - integer_atom_type: byte, shortint, int, longint, integer, or time
  - non_integer_type: shortreal, real, or realtime
  - struct
  - enum

**Integer Data Types**

SystemVerilog supports the following integer types:

- shortint: 2-state 16-bit signed integer
- int: 2-state 32-bit signed integer
- longint: 2-state 64-bit signed integer
- byte: 2-state 8-bit signed integer
- bit: 2-state, user defined vector size
- logic: 4-state user defined vector size
- reg: 4-state user-defined vector size
- integer: 4-state 32-bit signed integer
- time: 4-state 64-bit unsigned integer

4-state and 2-state refer to the values that can be assigned to those types, as follows:

- 2-state allows 0s and 1s.
- 4-state also allows X and Z states.

X and Z states cannot always be synthesized; therefore, items that are 2-state and 4-state are synthesized in the same way.

**CAUTION!** Take care when using 4-state variables: RTL versus simulation mismatches could occur.

- The types byte, shortint, int, integer, and longint default to signed values.
- The types bit, reg, and logic default to unsigned values.
Real Numbers

Synthesis supports real numbers; however, they cannot be used for behavior. They can be used as parameter values. The SystemVerilog-supported real types are:

- real
- shortreal
- realtime

Void Data Type

The void data type is only supported for functions that have no return value.

User-Defined Types

Vivado synthesis supports user-defined types, which are defined using the typedef keyword. Use the following syntax:

```plaintext
typedef data_type type_identifier {size};
```
or

```plaintext
typedef [enum, struct, union] type_identifier;
```

Enum Types

Enumerated types can be declared with the following syntax:

```plaintext
enum [type] {enum_name1, enum_name2...enum_nameX} identifier
```

If no type is specified, the enum defaults to int. Following is an example:

```plaintext
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:

```plaintext
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:

```plaintext
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.

```plaintext
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:

```plaintext
typedef enum {day[7]} day_of_week;
```

This creates an enumerated type called `day_of_week` with seven elements called `day0`, `day1...day6`.

Following are other ways to use this:

```plaintext
typedef enum {day[1:7]} day_of_week;  // creates day1, day2...day7
typedef enum {day[7] = 5} day_of_week; // creates day0=5, day1=6... day6=11
```

### Constants

SystemVerilog gives three types of elaboration-time constants:

- `parameter`: Is the same as the original Verilog standard and can be used in the same way.
- `localparam`: Is similar to `parameter` but cannot be overridden by upper-level modules.
- `specparam`: Is used for specifying delay and timing values; consequently, this value is `not supported` in Vivado synthesis.

There is also a run-time constant declaration called `const`.

### Type Operator

The type operator allows parameters to be specified as data types, which allows modules to have different types of parameters for different instances.

### Casting

Assigning a value of one data type to a different data type is illegal in SystemVerilog. However, a workaround is to use the cast operator (`'`). The cast operator converts the data type when assigning between different types. The usage is:

```plaintext
casting_type'(expression)
```

The `casting_type` is one of the following:

- `integer_type`
- `non_integer_type`
o real_type
  o constant unsigned number
  o user-created signing value type

Aggregate Data Types

In aggregate data types there are structures and unions, which are described in the following subsections.

Structures

A structure is a collection of data that can be referenced as one value, or the individual members of the structure. This is similar to the VHDL concept of a record. The format for specifying a structure is:

    struct {struct_member1; struct_member2;...struct_memberx;} structure_name;

Unions

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

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

Packed and Unpacked Arrays

Vivado synthesis supports both packed and unpacked arrays:

    logic [5:0] sig1;  //packed array
    logic sig2 [5:0]; //unpacked array

Data types with predetermined widths do not need the packed dimensions declared:

    integer sig3;  //equivalent to logic signed [31:0] sig3
Processes

Always Procedures

There are four always procedures:

- always
- always_comb
- always_latch
- always_ff

The procedure *always_comb* describes combinational logic. A sensitivity list is inferred by the logic driving the *always_comb* statement.

For always you must provide the sensitivity list. The following examples use a sensitivity list of *in1* and *in2*:

```vhdl
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:

```vhdl
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:

```vhdl
always_ff@(posedge clk)
out1 <= in1;
```
Block Statements

Block statements provide a mechanism to group sets of statements together. Sequential blocks have a `begin` and `end` around the statement. The block can declare its own variables, and those variables are specific to that block. The sequential block can also have a name associated with that block. The format is as follows:

```
begin [: block name]
[declarations]
[statements]
end [: block name]
```

```
begin : my_block
logic temp;
temp = in1 & in2;
out1 = temp;
end : my_block
```

In the previous example, the block name is also specified after the `end` statement. This makes the code more readable, but it is not required.

*Note:* Parallel blocks (or fork join blocks) are *not* supported in Vivado synthesis.

Procedural Timing Controls

SystemVerilog has two types of timing controls:

- **Delay control**: Specifies the amount of time between the statement its execution. This is not useful for synthesis, and Vivado synthesis ignores the time statement while still creating logic for the assignment.

- **Event control**: Makes the assignment occur with a specific event; for example, `always@(posedge clk)`. This is standard with Verilog, but SystemVerilog includes extra functions.

The logical `or` operator is an ability to give any number of events so that any one of them will trigger the execution of the statement. To do this, use either a specific `or`, or separate with commas in the sensitivity list. For example, the following two statements are the same:

```
always@(a or b or c)
always@(a,b,c)
```

SystemVerilog also supports the implicit `event_expression @*`. This helps to eliminate simulation mismatches caused because of incorrect sensitivity lists, for example:

```
Logic always@* begin
```
Operators

Vivado synthesis supports the following SystemVerilog operators:

- **Assignment operators**
  
  \(=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<=, >>>=\)

- **Unary operators**
  
  \(+, -, !, ~, &!, ~&, |, ~|, ^, ^!, ^~\)

- **Increment/decrement operators**
  
  \(++\), \(--\)

- **Binary operators**
  
  \(+, -, *, /, \%, ==, \!, ===, \!,==, &&, \|, **, <, <=, >, >=, &!, |, ^, ^!, ^~\)

  **Note:** \(A^{**B}\) is supported if \(A\) is a power of 2 or \(B\) is a constant.

- **Conditional operator**
  
  \(?\) : \)

- **Concatenation operator**
  
  \{\} \)

Signed Expressions

Vivado synthesis supports both signed and unsigned operations. Signals can be declared as unsigned or signed. For example:

```verilog
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:

```verilog
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:

```verilog
If (expression1)
  Command1;
else if (expression2)
  command2;
else if (expression3)
  command3;
else
```
command4;

This example is synthesized as a priority if statement. So, 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 allow you to handle don’t cares in the values (casex) or 3-state conditions in the values (casez).

If unique or priority case statements are used, Vivado synthesis treats those as parallel_case and full_case respectively.

**Loop Statements**

Several types of loops that are supported in Vivado synthesis and SystemVerilog. One of the most common is the for loop. Following is the syntax:

```
for (initialization; expression; step)
  statement;
```

A for loop starts with the initialization, then evaluates the expression. If the expression evaluates to 0, it stops, else if the expression evaluates to 1, it continues with the statement. When it is done with the statement, it executes the step function.

- A repeat loop works by performing a function a stated number of times. Following is the syntax:

  ```
  repeat (expression)
  statement;
  ```

  This works by evaluating the expression to a number and then executing the statement that many 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:

```plaintext
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:

```plaintext
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:

```plaintext
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 say in the statement:

```plaintext
function_name = ....
```
Like tasks, functions can also be automatic or static. Vivado synthesis treats all functions as automatic. However, some simulators might behave differently. Be careful when using these functions.

---

**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:

```verilog
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 hooking up modules. For example, from the previous example:

```verilog
    // in the instantiating module
    lower my_inst (.*);
```

This hooks up 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:

```verilog
    lower my_inst (.myout(my_sig), .my_in(din), .*);
```

This hooks up the myout port to a signal called my_sig, the my_in port to a signal called din and clk and my_in2 is hooked up to the clk and my_in2 signals.
Interfaces

Interfaces provide a way to specify communication between blocks. An interface is a group of nets and variables that are grouped together for the purpose of making connections between modules easier to write.

The syntax for a basic interface is:

```
interface interface_name;
parameters and ports;
items;
endinterface : interface_name
```

The `interface_name` at the end is optional but makes the code easier to read. For an example, see the following code:

```
module bottom1 (  
    input clk,  
    input [9:0] d1,d2,  
    input s1,  
    input [9:0] result,  
    output logic sel,  
    output logic [9:0] data1, data2,  
    output logic equal);
  //logic//  
endmodule

module bottom2 (  
    input clk,  
    input sel,  
    input [9:0] data1, data2,  
    output logic [9:0] result);
  //logic//  
endmodule

module top (  
    input clk,  
    input s1,  
    input [9:0] d1, d2,  
    output equal);
  logic [9:0] data1, data2, result;  
  logic sel;
  bottom1 u0 (clk, d1, d2, s1, result, sel, data1, data2, equal);  
  bottom2 u1 (clk, sel, data1, data2, result);
endmodule
```

The previous code snippet instantiates two lower-level modules with some signals that are common to both. These common signals can all be specified with an interface:
interface my_int
logic sel;
logic [9:0] data1, data2, result;
endinterface : my_int

Then in the two bottom-level modules, you can change to:

module bottom1(
  my_int int1,
  input clk,
  input [9:0] d1, d2,
  input s1,
  output logic equal);
and:
module bottom2(
  my_int int1,
  input clk);

Inside the modules, you can also change how you access sel, data1, data2, and result. This is because, according to the module, there are no ports of these names. Instead, there is a port called my_int. This requires the following change:

if (sel)
result <= data1;

to:

if (int1.sel)
int1.result <= int1.data1;

Finally, in the top level, the interface needs to 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:

```vhdl
interface my_int;
    logic sel;
    logic [9:0] data1, data2, result;

    modport b1 (input result, output sel, data1, data2);
    modport b2 (input sel, data1, data2, output result);
endinterface : my_int
```

Then in the bottom modules, use when declared:

```vhdl
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 example above, `data1` and `data2` were both 10-bit vectors, but those can be modified 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:

```vhdl
package package_name;
  items
endpackage : package_name
```

The final `package_name` is not required, but it makes code easier to read.

Packages are then referenced in other modules by the import command. Following is the syntax:

```vhdl
import package_name::item or *;
```

The `import` command must include items from the package to import or must specify the whole package.
HDL Coding Techniques

Introduction

Hardware Description Language (HDL) coding techniques let you:

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

Advantages of VHDL

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

Advantages of Verilog

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

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

Flip-Flops, Registers, and Latches

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

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

Flip-Flops, Registers and Latches are described with:

- sequential process (VHDL)
- always block (Verilog)
- always_ff for Flip Flops, always_latch for Latches (SystemVerilog)

The process or always block sensitivity list should list:

- The clock signal
- All asynchronous control signals

Flip-Flops and Registers Control Signals

Flip-Flops and Registers control signals include:

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

• Do not set or reset Registers asynchronously.
  - 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.
  - You will be unable to leverage device resources, or they will be configured sub-optimally.
• Do not describe Flip-Flops with both a set and a reset.
  - No Flip-Flop primitives feature both a set and a reset, whether synchronous or asynchronous.
  - Flip-Flop primitives featuring both a set and a reset may adversely affect area and performance.
• Avoid operational set/reset logic whenever possible. There may be other, less expensive, ways to achieve the desired effect, such as taking advantage of the circuit global reset by defining an initial content.
• Always describe the clock enable, set, and reset control inputs of Flip-Flop primitives as active-High. If they are described as active-Low, the resulting inverter logic will penalize circuit performance.

Flip-Flops and Registers Inference

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

• FDCE: D-Flip Flop with Clock Enable and Asynchronous Clear
• FDPE: D Flip Flop with Clock Enable and Asynchronous Preset
• FDSE: D Flip flop with Clock Enable and Synchronous Set
• FDRE: D Flip flop with Clock Enable and Synchronous Reset

Flip-Flops and Registers Initialization

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

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

Flip-Flops and Registers Reporting Example

```
RTL Component Statistics

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

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

Flip-Flops and Registers Coding Examples

The following subsections provide VHDL and Verilog examples of coding for Flip-Flops and registers.

Code examples are located at:

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip
Flip-Flops and Registers VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

--File: HDL_Coding_Techniques/registers/register6,vhd
-- Flip-Flop with
-- Rising-edge Clock
-- Active-high Synchronous Clear
-- Active-high Clock Enable
library IEEE;
use IEEE.std_logic_1164.all;

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

Flip-Flops and Registers Verilog Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

// File: HDL_Coding_Techniques/registers/registers_6.v
// 8-bit Register with
// Rising-edge Clock
// Active-high Synchronous Clear
// Active-high Clock Enable
module top(d_in,ce,clk,clr,dout);
input [7:0] d_in;
input ce;
input clk;
input clr;
output [7:0] dout;
reg [7:0] d_reg;

always @(posedge clk)
begain
if(clr)
    d_reg <= 8'b0;
else if(ce)
    d_reg <= d_in;
end
assign dout = d_reg;
endmodule
Latches

Code examples are located at: http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

Latches Reporting

- 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 reporting example below. This warning allows you to verify that the inferred Latch functionality was intended.

Latches Reporting Example

```plaintext
========================================================================= *
*                     Vivado.log                                        *
========================================================================= *
WARNING: [Synth 8-327] inferring latch for variable 'Q_reg'
========================================================================= Report
Cell Usage:

|Cell|Count
-----+----+-----
|LD  |    1
-----+----+-----
```

Latches Coding Examples

The following subsections provide VHDL and Verilog coding examples for latches with positive gates and asynchronous resets.
Latch With Positive Gate and Asynchronous Reset VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

-- File: HDL_Coding_Techniques/latches/latches_2.vhd

-- Latch with Positive Gate and Asynchronous Reset
--
library ieee;
use ieee.std_logic_1164.all;

entity latches_2 is
  port(G, D, CLR : in std_logic;
       Q : out std_logic);
end latches_2;

architecture archi of latches_2 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;
3-states

• 3-state buffers are usually modeled by:
  ° A signal
  ° An if-else construct
• This applies whether the buffer drives:
  ° An internal bus, or
  ° An external bus on the board on which the device resides
• The signal is assigned a high impedance value in one branch of the if-else

Code examples are located at:

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

3-states Implementation

Inferred 3-state buffers are implemented with different device primitives when driving an:

• Internal bus (BUFT)
  ° A BUFT inferred is converted automatically to a logic realized in LUTs by Vivado synthesis
  ° When an internal Bus inferring a BUFT is driving an output of the top module
    Vivado synthesis is inferring an OBUF
• External pin of the circuit (OBUFT)

3-states Reporting

3-state buffers are inferred and reported during synthesis.

3-state Reporting Example

=========================================================================*Vivado log file*========================================================================

Report Cell Usage:
---------+-+-------------------------
|Cell |Count |
---------+-+-------------------------
1 |OBUFT| 1
---------+-+-------------------------

=========================================================================
3-states Coding Examples

The following subsections provide VHDL and Verilog coding examples for 3-state.

3-state Description Using Combinatorial Process VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

```vhdl
-- File: HDL_Coding_Techniques/tristates/tristates_2.vhd
-- Tristate Description Using Combinatorial Process
-- Implemented with an OBUFT (IO buffer)
--
library ieee;
use ieee.std_logic_1164.all;

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

architecture archi of three_st_1 is
begin
  process (I, T)
  begin
    if (T='0') then
      O <= I;
    else
      O <= 'Z';
    end if;
  end process;
end archi;
```

3-state Description Using Concurrent Assignment VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

```vhdl
//File: HDL_Coding_Techniques/tristates/tristates_1.v
// Tristate Description Using Concurrent Assignment
//
library ieee;
use ieee.std_logic_1164.all;

entity three_st_2 is
  port(T : in std_logic;
       I : in std_logic;
       O : out std_logic);
end three_st_2;

architecture archi of three_st_2 is
begin
  O <= I when (T='0') else 'Z';
end archi;
```
3-state Description Using Combinatorial Process VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

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

entity example is
  generic (
    WIDTH : integer := 8
  );
  port(
    T : in std_logic;
    I : in std_logic_vector(WIDTH-1 downto 0);
    O : out std_logic_vector(WIDTH-1 downto 0));
end example;

architecture archi of example is
  signal S : std_logic_vector(WIDTH-1 downto 0);
begin
  process (I, T)
  begin
    if (T = '1') then
      S <= I;
    else
      S <= (others => 'Z');
    end if;
  end process;
  O <= not(S);
end archi;
```

3-state Description Using Combinatorial Always Block Verilog Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

```verilog
// Tristate Description Using Combinatorial Always Block
module v_three_st_1 (T, I, O);
  input T, I;
  output O;
  reg O;
  always @(T or I)
  begin
    if (~T)
      O = I;
    else
      O = 1'bZ;
  end
endmodule
```
### 3-state Description Using Concurrent Assignment Verilog Coding Example

```verilog
//
// Tristate Description Using Concurrent Assignment
//
module v_three_st_2 (T, I, O);
  input T, I;
  output O;

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

Code examples are located at:

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

### Static Shift Register Elements

A static Shift Register usually involves:

- A clock
- An optional clock enable
- A serial data input
- A serial data output
Shift Registers SRL-Based Implementation

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

- SRL16E
- SRLC32E

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

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

Shift Registers Coding Examples

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

32-Bit Shift Register VHDL Coding Example One

This coding example uses the concatenation coding style.

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

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

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_0 is
  generic (DEPTH : integer := 32);
  port (clk   : in std_logic;
         clken : in std_logic;
         SI    : in std_logic;
         SO    : out std_logic);
end shift_registers_0;

architecture archi of shift_registers_0 is
  signal shreg: std_logic_vector(DEPTH-1 downto 0);
begin
  process (clk)
```


begin
    if rising_edge(clk) then
        if clken = '1' then
            shreg <= shreg(DEPTH-2 downto 0) & SI;
        end if;
    end if;
end process;

SO <= shreg(DEPTH-1);
end archi;

32-Bit Shift Register VHDL Coding Example Two

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

The same functionality can also be described as follows:

-- File: HDL_Coding_Techniques/shift_registers/shift_registers_1.vhd
-- 32-bit Shift Register
--    Rising edge clock
--    Active high clock enable
--    for loop-based template
--
library ieee;
use ieee.std_logic_1164.all;

definition of registers
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
    definition of signals
signal shreg: std_logic_vector(DEPTH-1 downto 0);

begin

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

    SO <= shreg(DEPTH-1);
end archi;
32-Bit Shift Register Verilog Coding Example One

This coding example uses a concatenation to describe the Register chain.
http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

```verilog
-- File: HDL_Coding_Techniques/shift_registers/shift_registers_0.vhd
-- 8-bit Shift Register
-- Rising edge clock
-- Active high clock enable
-- Concatenation-based template
module v_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
```

8-Bit Shift Register Verilog Coding Example Two

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

```verilog
// File: HDL_Coding_Techniques/shift_registers/shift_registers_1.v
// 32-bit Shift Register
// Rising edge clock
// Active high clock enable
// For-loop based template
module v_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];1
endmodule
```
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.
Dynamic Shift Registers

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

32-Bit Dynamic Shift Registers VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

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

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

end dynamic_shift_register;
```
architecture rtl of dynamic_shift_register is
    type SRL_ARRAY is array (DEPTH-1 downto 0) of std_logic;
    signal SRL_SIG : SRL_ARRAY;

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

    DO <= SRL_SIG(conv_integer(A));
end rtl;

32-Bit Dynamic Shift Registers Verilog Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

//File: HDL_Coding_Techniques/dynamic_shift_registers/dynamic_shift_registers_1.v
// 32-bit dynamic shift register.

module dynamic_shift_register (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
Multipliers

Vivado synthesis infers Multiplier macros from multiplication operators in the source code.

- The resulting signal width equals the sum of the two operand sizes. For example, multiplying a 16-bit signal by an 8-bit signal produces a result of 24 bits.

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

Code examples are located at:

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

Multipliers Implementation

Multiplier macros can be implemented on:

- Slice logic
- DSP blocks

The implementation choice is:

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

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

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

DSP Block Implementation

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

- On the multiplication operands
- After the multiplication

When a Multiplier does not fit on a single DSP block, Vivado synthesis decomposes the macro to implement it. In that case, Vivado synthesis uses either of the following:
• Several DSP blocks
• A hybrid solution involving both DSP blocks and slice logic

Use the KEEP attribute to restrict absorption of Registers into DSP blocks. For example, if a Register is present on an operand of the multiplier, place KEEP on the output of the Register to prevent the Register from being absorbed into the DSP block. For more information on the KEEP attribute, see KEEP in Appendix 2.

**Multipliers Coding Examples**

**Unsigned 16x16-Bit Multiplier VHDL Coding Example**

```vhdl
-- -- Unsigned 16x16-bit Multiplier
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity multipliers1 is
  generic (
    WIDTHA : integer := 16;
    WIDTHB : integer := 16
  );
  port(
    A   : in std_logic_vector(WIDTHA-1 downto 0);
    B   : in std_logic_vector(WIDTHB-1 downto 0);
    RES : out std_logic_vector(WIDTHA+WIDTHB-1 downto 0));
end multipliers1;

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

**Unsigned 16x24-Bit Multiplier Verilog Coding Example**

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

Multiply-Add and Multiply-Accumulate

The following macros are inferred:

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

The macros are inferred by aggregation of:

- A Multiplier
- An Adder/Subtractor
- Registers

Multiply-Add and Multiply-Accumulate Implementation

During Multiply-Add and Multiply-Accumulate implementation:

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

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

• In default mode, Vivado synthesis:
  - Implements Multiply-Add and Multiply-Accumulate Macros.
  - Takes into account DSP block resources availability in the targeted device.
  - Can use 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 KEEP attribute to restrict absorption of Registers into DSP blocks. For example, to exclude a Register present on an operand of the Multiplier from absorption into the DSP block, apply KEEP on the output of the Register. For more information about the KEEP attribute, see KEEP, page 42.
Multiply-Add and Multiply-Accumulate Coding Examples

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

Fully Pipelined Streaming Multiply Accumulate VHDL Coding Example

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

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

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

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

  process (clk)
  begin
    if rising_edge(clk) then
      if ce = '1' then
        Send Feedback
```


a_reg     <= a;
b_reg     <= b;
mult_reg  <= a_reg * b_reg;
sload_reg <= sload;
-- Store accumulation result into a register
adder_out <= old_result + mult_reg;
end if;
end if;
end process;

-- Output accumulation result
accum_out <= adder_out;

end rtl;

Fully Pipelined Streaming Multiply Accumulate Verilog Coding Example

module macc  # (SIZEIN = 16, SIZEOUT = 40)
            (input clk, ce, sload,
             input signed [SIZEIN-1:0]  a, b,
             output signed [SIZEOUT-1:0] accum_out);

// Declare registers for intermediate values
reg signed [SIZEIN-1:0]  a_reg, b_reg;
reg                      sload_reg;
reg signed [2*SIZEIN:0]  mult_reg;
reg signed [SIZEOUT-1:0] adder_out, old_result;

always @(adder_out or sload_reg)
  if (sload_reg)
    old_result <= 0;
  else
    // 'sload' is now active (=low) and opens the accumulation loop.
    // The accumulator takes the next multiplier output in
    // the same cycle.
    old_result <= adder_out;

always @(posedge clk)
  if (ce)
    begin
      a_reg <= a;
b_reg <= b;
mult_reg <= a_reg * b_reg;
sload_reg <= sload;
    end


FIR Filters

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

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

A code example is located at:

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

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

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

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

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

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

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

  sfir_shifter #(dsize, nbtap) shifter_inst0 (clk, datain, shifterout);
generate
  genvar I;
  for (I=0; I<nbtap; I=I+1)
    if (I==0)
      sfir_even_symmetric_systolic_element #(dsize) fte_inst0 (clk, h[I], datain, shifterout, {32{1'b0}}, arraydata[I], arrayprod[I]);
    else
      sfir_even_symmetric_systolic_element #(dsize) fte_inst (clk, h[I], arraydata[I-1], shifterout, arrayprod[I-1], arraydata[I], arrayprod[I]);
  endgenerate
endmodule // top

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

(* dont_touch = "yes" *)
module sfir_shifter #(parameter dsize = 16, nbtap = 4)
  (input clk, [dsize-1:0] datain, output [dsize-1:0] dataout);
(* srl_style = "srl_register" *)
  reg [dsize-1:0] tmp [0:2*nbtap-1];
  integer i;

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

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

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

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] casdata, output reg signed [2*dsize-1:0] cascout);

  // Processing logic
  // ...
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 cascdat = datatwo;

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

demodule // sfir_even_symmetric_systolic_element

---

RAM HDL Coding Techniques

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

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

Code examples are located at:

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

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.
Whether to use distributed RAM or dedicated block RAM can depend upon:

- The characteristics of the RAM you have described in the HDL source code
- Whether you have forced a specific implementation style using \textit{ram\_style} attribute. See \textit{RAM\_STYLE, page 45}. for more information.
- Availability of block RAM resources

\textbf{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

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Action & Distributed RAM & Dedicated Block RAM \\
\hline
Write & Synchronous & Synchronous \\
\hline
Read & Asynchronous & Synchronous \\
\hline
\end{tabular}
\caption{Distributed RAM versus Dedicated Block RAM}
\end{table}
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 in order to accommodate the described data widths

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

**RAM HDL Coding Guidelines**

RAM HDL coding guidelines include:

- Block RAM Read/Write Synchronization modes
- Distributed RAM examples
- Single port ram coding example
  - Read first mode
  - Write first mode
  - No change mode
- Simple Dual port ram coding example
  - Single clock
  - Dual clock
- True Dual port ram coding example
  - Single clock
  - Dual clock
- Byte wide write enable coding example
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 subsections provide VHDL and Verilog coding examples for distributed RAM.

Single-Port RAM with Asynchronous Read VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

```vhdl
--File: HDL_Coding_Techniques/rams/rams_04.v
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_04 is
  port (clk : in std_logic;
        we : in std_logic;
        a  : in std_logic_vector(5 downto 0);
        di : in std_logic_vector(15 downto 0);
        do : out std_logic_vector(15 downto 0));
end rams_04;

architecture syn of rams_04 is
  type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
  signal RAM : ram_type;
begin

  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(a)) <= di;
      end if;
    end if;
  end process;

  do <= RAM(conv_integer(a));

end syn;
```
Dual-Port RAM with Asynchronous Read Verilog Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

    // Dual-Port RAM with Asynchronous Read (Distributed RAM)
    module v_rams_09 (clk, we, a, dpra, di, spo, dpo);
    input clk;
    input we;
    input [5:0] a;
    input [5:0] dpra;
    input [15:0] di;
    output [15:0] spo;
    output [15:0] dpo;
    reg [15:0] ram [63:0];

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

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

    endmodule

Single-Port Block RAMs

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

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

    -- Single-Port Block RAM Read-First Mode
    -- File: HDL_Coding_Techniques/rams/rams_04.v

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

    entity rams_01 is
        port (clk : in std_logic;
              we   : in std_logic;
              en   : in std_logic;
              addr : in std_logic_vector(9 downto 0);
              di   : in std_logic_vector(15 downto 0);
              do   : out std_logic_vector(15 downto 0));
    end rams_01;

    architecture syn of rams_01 is
        type ram_type is array (1023 downto 0) of std_logic_vector (15 downto 0);
        signal RAM: ram_type;
    begin

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

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

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

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

-- Single-Port Block RAM Read-First Mode
-- File: HDL_Coding_Techniques/rams/rams_01.vhd

module v_rams_01 (clk, en, we, addr, di, do);

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

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

   always @(posedge clk)
   begin
      if (en)
      begin
         if (we)
            RAM[addr]<=di;
            do <= RAM[addr];
      end
   end
endmodule
Single-Port Block RAM Write-First Mode VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

-- File: HDL_Coding_Techniques/rams/rams_02a.vhd
-- Single-Port Block RAM Write-First Mode (recommended template)
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_02a is
port (clk : in std_logic;
we   : in std_logic;
en   : in std_logic;
addr : in std_logic_vector(9 downto 0);
di   : in std_logic_vector(15 downto 0);
do   : out std_logic_vector(15 downto 0));
end rams_02a;

architecture syn of rams_02a is
begin

process (clk)
begin
if clk'event and clk = '1' then
if en = '1' then
if we = '1' then
RAM(conv_integer(addr)) <= di;
do <= di;
else
do <= RAM(conv_integer(addr));
end if;
end if;
end if;
end process;
end syn;
Single-Port Block RAM Write-First Mode Verilog Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

// Single-Port Block RAM Write-First Mode (recommended template)
//File: HDL_Coding_Techniques/rams/rams_02a.v
//
module v_rams_02a (clk, we, en, addr, di, do);

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

always @(posedge clk)
begin
  if (en)
    begin
      if (we)
        begin
          RAM[addr] <= di;
          do <= di;
        end
      else
        do <= RAM[addr];
    end
end
endmodule
Single-Port Block RAM No-Change Mode VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

-- Single-Port Block RAM No-Change Mode
-- File: HDL_Coding_Techniques/rams/rams_03.vhd

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

entity rams_03 is
  port (clk : in std_logic;
        we   : in std_logic;
        en   : in std_logic;
        addr : in std_logic_vector(9 downto 0);
        di   : in std_logic_vector(15 downto 0);
        do   : out std_logic_vector(15 downto 0));
end rams_03;

architecture syn of rams_03 is
  type ram_type is array (1023 downto 0) of std_logic_vector (15 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if clk'event and clk = '1' then
      if en = '1' then
        if we = '1' then
          RAM(conv_integer(addr)) <= di;
        else
          do <= RAM( conv_integer(addr));
        end if;
      end if;
    end if;
  end process;
end syn;
Single-Port Block RAM No-Change Mode Verilog Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

// Single-Port Block RAM No-Change Mode
module v_rams_03 (clk, we, en, addr, di, do);
input clk;
input we;
input en;
input [9:0] addr;
input [15:0] di;
output [15:0] do;
reg [15:0] RAM [1023:0];
reg [15:0] do;
always @(posedge clk)
begin
  if (en)
  begin
    if (we)
      RAM[addr] <= di;
    else
      do <= RAM[addr];
  end
endmodule

Simple Dual-Port Block RAM Examples

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

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

-- Simple Dual-Port Block RAM with One Clock
-- Correct Modelization with a Shared Variable
-- File: HDL_Coding_Techniques/rams/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);
  reg [15:0] RAM [1023:0];
  reg [15:0] do;
  always @(posedge clk)
  begin
    if (ena)
    begin
      if (wea)
        RAM[addra] <= dia;
      else
        do <= RAM[addra];
    end
  end
endmodule
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
    doa <= RAM(conv_integer(addrb));
end if;
end if;
end process;
end syn;

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

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

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

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

always @(posedge clk) begin if (ena)
begin
    if (wea)
        ram[addra] <= dia;
    end
end
always @(posedge clk) begin if (enb)
    dob <= ram[addrb];
end
endmodule
Simple Dual-Port Block RAM with Dual Clocks VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

-- Simple Dual-Port Block RAM with Two Clocks
-- Correct Modelization with a Shared Variable
-- File: HDL_Coding_Techniques/rams/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));
en end simple_dual_two_clocks;

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

process (clka)
begin
if clka'event and clka = '1' then
 if ena = '1' then
  if wea = '1' then
   RAM(conv_integer(addra)) := dia;
  end if;
 end if;
end if;
end process;

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

end syn;
Simple Dual-Port Block RAM with Dual Clocks Verilog Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

// Simple Dual-Port Block RAM with Two Clocks
// File: HDL_Coding_Techniques/rams/simple_dual_two_clocks.v
//
module simple_dual_two_clocks (clka,clkb,ena,enb,wea,addr,addrb,dia,dob);

input clka,clkb,ena,enb,wea;
input [9:0] addr,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[addr] <= dia;
end
end
always @(posedge clkb) begin if (enb)
begin
dob <= ram[addrb];
end
end
endmodule

True Dual-Port Block RAM Examples

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

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

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

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

entity rams_16b is
port(clka : in std_logic;
clkb : in std_logic;
en : in std_logic;
enb : in std_logic;
wea : in std_logic;
web : in std_logic;
addr : 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 entity rams_16b;
dib : in std_logic_vector(15 downto 0);
doa : out std_logic_vector(15 downto 0);
dob : out std_logic_vector(15 downto 0);
end rams_16b;
architecture syn of rams_16b is
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
shared variable RAM : ram_type;
begin

process (CLKA)
begin
if CLKA'event and CLKA = '1' then
 if ENA = '1' then
  DOA <= RAM(conv_integer(ADDRA));
  if WEA = '1' then
   RAM(conv_integer(ADDRA)) := DIA;
  end if;
 end if;
end if;
end process;

process (CLKB)
begin
if CLKB'event and CLKB = '1' then
 if ENB = '1' then
  DOB <= RAM(conv_integer(ADDRB));
  if WEB = '1' then
   RAM(conv_integer(ADDRB)) := DIB;
  end if;
 end if;
end if;
end process;
end syn;
Dual-Port Block RAM with Two Write Ports Verilog Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

// Dual-Port Block RAM with Two Write Ports
// File: HDL_Coding_Techniques/rams/rams_16.v
module v_rams_16 (clka, clkb, ena, enb, wea, web, addra, addrb, dia, dib, doa, dob);
input clka, clkb, ena, enb, wea, web;
input [9:0] addra, addrb;
input [15:0] dia, dib;
output [15:0] doa, dob;
reg [15:0] ram[1023:0];
reg [15:0] doa, dob;

always @(posedge clka) begin if (ena)
begin
if (wea)
ram[addra] <= dia;
doa <= ram[addra];
end
end
always @(posedge clkb) begin if (enb)
begin
if (web)
ram[addrb] <= dib;
dob <= ram[addrb];
end
end
endmodule

Block RAM with Re-Set Able Data Output VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

-- Block RAM with Resettable Data Output
-- File: HDL_Coding_Techniques/rams/rams_18.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_18 is
port (clk : in std_logic;
en : in std_logic;
we : in std_logic;
rst : in std_logic;
addr : in std_logic_vector(9 downto 0);
di : in std_logic_vector(15 downto 0);
do : out std_logic_vector(15 downto 0));
end rams_18;

architecture syn of rams_18 is
type ram_type is array (1023 downto 0) of std_logic_vector (15 downto 0);
signal ram : ram_type;
begin
process (clk)
begin
if clk'event and clk = '1' then
    if en = '1' then -- optional enable
        if we = '1' then -- write enable
            ram(conv_integer(addr)) <= di;
        end if;
        if rst = '1' then -- optional reset
            do <= (others => '0');
        else
            do <= ram(conv_integer(addr));
        end if;
    end if;
end if;
end process;
end syn;

Block RAM with Re-set Able Data Output Verilog Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

// Block RAM with Resettable Data Output
// File: HDL_Coding_Techniques/rams/rams_18.v
module v_rams_18 (clk, en, we, rst, addr, di, do);

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

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

    always @(posedge clk)
    begin
        if (en) // optional enable
            begin
                if (we) // write enable
                    ram[addr] <= di;
                if (rst) // optional reset
                    do <= 0;
                else
                    do <= ram[addr];
            end
    end
endmodule
Block RAM with Optional Output Registers VHDL Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

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

entity rams_19 is
port (clk1, clk2 : in std_logic;
we, en1, en2 : in std_logic;
addr1 : in std_logic_vector(9 downto 0);
addr2 : in std_logic_vector(9 downto 0);
di    : in std_logic_vector(15 downto 0);
res1  : out std_logic_vector(15 downto 0);
res2  : out std_logic_vector(15 downto 0));
end rams_19;

architecture beh of rams_19 is

type ram_type is array (1023 downto 0) of std_logic_vector (15 downto 0);

signal ram : ram_type;
signal do1 : std_logic_vector(15 downto 0);
signal do2 : std_logic_vector(15 downto 0);

begin

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

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

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

process (clk2)
begin
if rising_edge(clk2) then
  if en2 = '1' then

res2 <= do2;
end if;
end if;
end process;
end beh;

Block RAM with Optional Output Registers Verilog Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_uq_examples.zip

// Block RAM with Optional Output Registers
// File: HDL_Coding_Techniques/rams/rams_19.v
module v_rams_19 (clk1, clk2, we, en1, en2, addr1, addr2, di, res1, res2);

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

always @(posedge clk1)
begin
  if (we == 1'b1)
  begin
    RAM[addr1] <= di;
    do1 <= RAM[addr1];
  end
end
always @(posedge clk2)
begin
  do2 <= RAM[addr2];
end
always @(posedge clk1)
begin
  if (en1 == 1'b1)
  begin
    res1 <= do1;
  end
end
always @(posedge clk2)
begin
  if (en2 == 1'b1)
  begin
    res2 <= do2;
  end
end
endmodule
**Byte Write Enable (Block RAM)**

Xilinx supports byte write enable in block RAM.

Use byte write enable in block RAM to:

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

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

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

Vivado synthesis inference lets you take advantage of the block RAM byte write enable feature.

The described RAM is implemented on block RAM resources, using the byte write enable capability, provided that the following requirements are met:

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

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

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

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

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

RAM HDL Coding Techniques

NB_COL : integer := 4;

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

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

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

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

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

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_uq_examples.zip

//File: HDL_Coding_Techniques/rams/vivado_ram_templates_bwwe.v
// Single-Port BRAM with Byte Write Enable
//  4x9-bit write
//  Read-First mode
//  Single-process description
//  Compact description of the write with a generate-for statement
//  Column width and number of columns easily configurable
module v_bytewrite_ram_1b (clk, we, addr, di, do);

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

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

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

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

True-Dual-Port BRAM with Byte Write Enable Examples

Byte Write Enable—READ_FIRST Mode VHDL

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

-- READ_FIRST ByteWide WriteEnable Block RAM Template

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

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

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

end BWWE_RAMB_RF_TDP;

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

begin

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

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

Byte Write Enable—READ_FIRST Mode Verilog

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

// True-Dual-Port BRAM with Byte Write Enable
// Read-First mode (2 variants)
// Write-First mode
// No-Change mode
// File: HDL_Coding_Techniques/rams/bytewrite_tdp_ram.v
module v_bytewrite_tdp_ram_readfirst
#

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

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

  // CORE_MEMORY
  reg [DATA_WIDTH-1:0] ram_block [{2**ADDR_WIDTH}-1:0];

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

  // Port-B Operation:
  always @(posedge clkB) begin
    if(enaB) begin
      for(i=0;i<NUM_COL;i=i+1) begin
        if(weB[i]) begin
          ram_block[addrB][i*COL_WIDTH +: COL_WIDTH] <= dinB[i*COL_WIDTH +:
          COL_WIDTH];
        end
      end
      doutB <= ram_block[addrB];
    end
  end
endmodule // v_bytewrite_tdp_ram_readfirst
Byte Write Enable—Alternate READ_FIRST Mode

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

```
// Bytewrite Enable,- Alternate READ_FIRST mode template - Vivado
//module v_bytewrite_tdp_ram_readfirst2
#
//paramaters 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
#

// 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
endgenerate

always @ (posedge clkA) begin
  if(enaA) begin
    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
generate

always @ (posedge clkB) begin
    if(enaB) begin
        doutB <= ram_block[addrB];
    end
end
endmodule // v_bytewrite_tdp_ram_readfirst2

Byte Write Enable—WRITE_FIRST Mode VHDL

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

-- Byte Write Enable - WRITE_FIRST mode template Vivado recommended
module v_bytewrite_tdp_ram_writefirst
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

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

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

architecture byte_wr_ram_wf of BWWE_RAMB_WF_TDP is

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

begin

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

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

Byte Write Enable—WRITE_FIRST Mode Verilog

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

// Byte Write Enable, - WRITE_FIRST mode template - Vivado recommended
module v_bytewrite_tdp_ram_writefirst
#
//--------------------------------------------------------------------------
// parameter NUM_COL = 4,
// parameter COL_WIDTH = 8,
// parameter ADDR_WIDTH = 10, // Addr Width in bits: 2**ADDR_WIDTH= RAM Depth
// parameter DATA_WIDTH = NUM_COL*COL_WIDTH // Data Width in bits
//--------------------------------------------------------------------------
)
  (input clkA,
   input enaA,
   input [NUM_COL-1:0] weA,
   input [ADDR_WIDTH-1:0] addrA,
   input [DATA_WIDTH-1:0] dinA,
   output reg [DATA_WIDTH-1:0] doutA,
   input clkB,
   input enaB,
input [NUM_COL-1:0] weB,
input [ADDR_WIDTH-1:0] addrB,
input [DATA_WIDTH-1:0] dinB,
output reg [DATA_WIDTH-1:0] doutB
);

// CORE_MEMORY
reg [DATA_WIDTH-1:0] ram_block [{2**ADDR_WIDTH}-1:0];
// PORT-A Operation:
genvar i;
for(i=0;i<NUM_COL;i=i+1) begin
  always @(posedge clkA) begin
    if(enaA) begin
      if(weA[i]) begin
        ram_block[addrA][i*COL_WIDTH +: COL_WIDTH] <= dinA[i*COL_WIDTH +: COL_WIDTH];
doutA[i*COL_WIDTH +: COL_WIDTH]  <= dinA[i*COL_WIDTH +: COL_WIDTH];
      end else begin
        doutA[i*COL_WIDTH +: COL_WIDTH]  <= ram_block[addrA][i*COL_WIDTH +: COL_WIDTH];
      end
    end
  end
end
generate

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

endmodule // v_bytewrite_tdp_ram_writefirst
Byte Write Enable—NO_CHANGE Mode VHDL

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

-- NO_CHANGE ByteWide WriteEnable Block RAM Template
module v_bytewrite_tdp_ram_nochange(
    library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_unsigned.all;

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

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

    architecture byte_wr_ram_nc of BWWE_RAMB_NC_TDP is

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

        begin

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

--- End of VHDL Code ---
end if;
end process;

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

Byte Write Enable—NO_CHANGE Mode Verilog

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

```verilog
//bytereEnable, - NO_CHANGE mode template - Vivado recommended
//module v_byterewrite_tdp_ram_nochange(
//-------------------------------------------------------------
parameter   NUM_COL =   4,
parameter   COL_WIDTH = 8,
parameter   ADDR_WIDTH = 10, // Addr Width in bits : 2**ADDR_WIDTH = RAM Depth
parameter   DATA_WIDTH = NUM_COL*COL_WIDTH // Data Width in bits
//---------------------------------------------------------------
) ( // CORE_MEMORY
  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
);

// 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
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+=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
    endgenerate
always @ (posedge clkB) begin
    if(enaB) begin
        if (~weB)
            doutB <= ram_block[addrB];
    end
end
endmodule // v_bytewrite_tdp_ram_nochange

**RAM Initial Contents**

RAM can be initialized in the following ways:

- Specifying RAM Initial Contents in the HDL Source Code
- Specifying RAM Initial Contents in an External Data File

**Specifying RAM Initial Contents in the HDL Source Code**

Use the signal default value mechanism to describe initial RAM contents directly in the HDL source code.
VHDL Coding Examples

type ram_type is array (0 to 31) of std_logic_vector(19 downto 0);
signal RAM : ram_type :=
  (
    X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
    X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
    X"00340", X"00241", X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
    X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021", X"0030D", X"08201"
  );

All bit positions are initialized to the same value.

type ram_type is array (0 to 127) of std_logic_vector (15 downto 0);
signal RAM : ram_type := (others => (others => '0'));

Verilog Coding Examples

All addressable words are initialized to the same value.

    reg [DATA_WIDTH-1:0] ram [DEPTH-1:0];
    integer i;
    initial for (i=0; i<DEPTH; i=i+1) ram[i] = 0;
end

Specifying RAM Initial Contents in an External Data File

Use the file read function in the HDL source code to load the RAM initial contents from an external data file.

• The external data file is an ASCII text file with any name.
• Each line in the external data file describes the initial content at an address position in the RAM.
• There must be as many lines in the external data file as there are rows in the RAM array. An insufficient number of lines is flagged.
• The addressable position related to a given line is defined by the direction of the primary range of the signal modeling the RAM.
• You can represent RAM content in either binary or hexadecimal. You cannot mix both.
• The external data file cannot contain any other content, such as comments.
• The following external data file initializes an 8 x 32-bit RAM with binary values:

    00001111000011110000111100001111
    010010100100001100000010000100
    0000000000111110000000010000100
    11111101010000011100010000100100
    00000111000011110000111100001111
    010010100100001100000010000100
    0000000000111110000000010000100
    11111101010000011100010000100100
**VHDL Coding Example**

Load the data as follows:

```vhdl
type RamType is array(0 to 7) of bit_vector(31 downto 0);

impure function InitRamFromFile (RamFileName : in string) return RamType is
FILE RamFile : text is in RamFileName;
variable RamFileLine : line;
variable RAM : RamType;
begin
for I in RamType'range loop
readline (RamFile, RamFileLine);
read (RamFileLine, RAM(I));
end loop;
return RAM;
end function;

signal RAM : RamType := InitRamFromFile("rams_20c.data");
```

**Verilog Coding Example**

Use a `$readmemb` or `$readmemh` system task to load respectively binary-formatted or hexadecimal data.

```verilog
reg [31:0] ram [0:63];
initial begin
$readmemb("rams_20c.data", ram, 0, 63);
end
```

**Initializing Block RAM VHDL Coding Example**

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

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

entity rams_20a is
port (clk : in std_logic;
we   : in std_logic;
addr : in std_logic_vector(5 downto 0);
di   : in std_logic_vector(19 downto 0);
do   : out std_logic_vector(19 downto 0));
end rams_20a;

architecture syn of rams_20a is

type ram_type is array (63 downto 0) of std_logic_vector (19 downto 0);
signal RAM : ram_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
X"08201", X"00500", X"40001", X"02500", X"00340", X"00241",
X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
```
X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
X"0030D", X"02341", X"08201", X"0400D";

begin
process (clk)
begin
if rising_edge(clk) then
  if we = '1' then
    RAM(conv_integer(addr)) <= di;
  end if;
  do <= RAM(conv_integer(addr));
end if;
end process;
end syn;

Initializing Block RAM Verilog Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

// Initializing Block RAM (Single-Port Block RAM)
// File: HDL_Coding_Techniques/rams/rams_20a.v
module v_rams_20a (clk, we, addr, di, do);
input clk;
input we;
input [5:0] addr;
input [19:0] di;
output [19:0] do;
reg [19:0] ram [63:0];
reg [19:0] do;

initial
begin
  ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
  ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
  ram[57] = 20'h00300; ram[56] = 20'h08602; ram[55] = 20'h02310;
  ram[54] = 20'h0203B; ram[53] = 20'h08300; ram[52] = 20'h04002;
  ram[51] = 20'h08201; ram[50] = 20'h00500; ram[49] = 20'h04001;
  ram[48] = 20'h02500; ram[47] = 20'h00340; ram[46] = 20'h00241;
  ram[45] = 20'h04002; ram[44] = 20'h08300; ram[43] = 20'h08201;
  ram[42] = 20'h00500; ram[41] = 20'h08101; ram[40] = 20'h0602;
  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[18] = 20'h00301; ram[17] = 20'h04004; ram[16] = 20'h00301;
ram[0] = 20'h0400D;
end

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

Initializing Block RAM From an External Data File VHDL Coding Example

-- Initializing Block RAM from external data file
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;

entity rams_20c is
    port(clk : in std_logic;
         we  : in std_logic;
         addr : in std_logic_vector(5 downto 0);
         din  : in std_logic_vector(31 downto 0);
         dout : out std_logic_vector(31 downto 0));
end rams_20c;

architecture syn of rams_20c is

    type RamType is array(0 to 63) of bit_vector(31 downto 0);
    impure function InitRamFromFile (RamFileName : in string) return RamType is
        FILE RamFile: text is in RamFileName;
        variable RamFileLine : line;
        variable RAM : RamType;
    begin
        for I in RamType'range loop
            readline (RamFile, RamFileLine);
            read (RamFileLine, RAM(I));
        end loop;
        return RAM;
    end function;

    signal RAM : RamType := InitRamFromFile("rams_20c.data");

begin
    process (clk)
    begin
        if clk'event and clk = '1' then
            if we = '1' then
                RAM(conv_integer(addr)) <= to_bitvector(din);
            end if;
            dout <= to_std_logic_vector(RAM(conv_integer(addr)));
        end if;
    end process;
end syn;
Initializing Block RAM From an External Data File Verilog Coding Example

```verilog
// Initializing Block RAM from external data file
// Binary data
//
module v_rams_20c (clk, we, addr, din, dout);
  input clk;
  input we;
  input [5:0] addr;
  input [19:0] din;
  output [19:0] dout;

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

  initial begin
    $readmemh("rams_20c.data",ram);
  end

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

Black Boxes

A design can contain EDIF or NGC 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`.

Codes examples are available in
http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip
BLACK_BOX Verilog Example

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

BLACK_BOX VHDL Example

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

FSM Components

Code examples are located at:

http://www.xilinx.com/tpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

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.
The following diagram shows an FSM diagram with three processes.

**Figure 4-3:** FSM Representation Incorporating Mealy and Moore Machines Diagram

**State Registers**

- Specify a reset or power-up state for Vivado synthesis to identify a Finite State Machine (FSM).
- The State Register can be asynchronously or synchronously reset to a particular state.
- Xilinx recommends using synchronous reset logic over asynchronous reset logic for an FSM.

**Auto State Encoding**

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 state machine up to 32 states.
- Is usually a good choice for optimizing speed or reducing power dissipation.
- Assigns a distinct bit of code to each FSM state.
- Implements the State Register with one flip-flop for each state.
- In a given clock cycle during operation, only one bit of the State Register is asserted.
• Only two bits toggle during a transition between two states.

**Gray State Encoding**

Gray State encoding has the following attributes:

• Guarantees that only one bit switches between two consecutive states.
• Is appropriate for controllers exhibiting long paths without branching.
• Minimizes hazards and glitches.
• Can be used to minimize power dissipation.

**Johnson State Encoding**

Johnson State encoding is beneficial when using state machines containing long paths with no branching (as in Gray State Encoding).

**Sequential State Encoding**

Sequential State encoding has the following attributes:

• Identifies long paths
• Applies successive radix two codes to the states on these paths.
• Minimizes next state equations.

**FSM Verilog Example**

```verilog
// State Machine with single sequential block
module fsm_test(clk, reset, flag, sm_out);
input clk, reset, flag;
output reg sm_out;

parameter s1 = 2'b00;
parameter s2 = 2'b01;
parameter s3 = 2'b10;
parameter s4 = 2'b11;

reg [1:0] 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 <= 'b1;
end
else
begin
state <= s3;
sm_out <= 'b0;
end
s2: begin state <= s4; sm_out <= 'b0; end
s3: begin state <= s4; sm_out <= 'b0; end
s4: begin state <= s1; sm_out <= 'b1; end
endcase
end
endmodule

FSM VHDL Example

-- State Machine with single sequential block
library IEEE;
use IEEE.std_logic_1164.all;

entity fsm_test is
port ( clk, reset, flag : IN std_logic;
sm_out: OUT std_logic);
end entity;
architecture behavioral of fsm_test is
type state_type is (s1,s2,s3,s4); 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 <= s1; sm_out <= '1';

Send Feedback
end case;
end if;
end if;
end process;
end behavioral;

FSM Reporting

The Vivado synthesis flags INFO messages in the log file, giving information about Finite State Machine (FSM) components and their encoding. The following are example messages:

INFO: [Synth 8-802] inferred FSM for state register 'state_reg' in module 'fsm_test'
INFO: [Synth 8-3354] encoded FSM with state register 'state_reg' using encoding 'sequential' in module 'fsm_test'

ROM HDL Coding Techniques

Read-Only Memory (ROM) closely resembles Random Access Memory (RAM) with respect to HDL modeling and implementation. Use the ROM_STYLE attribute to implement a properly-registered ROM on block RAM resources. See ROM_STYLE, page 46 for more information.
ROM Coding Examples

Description of a ROM with a VHDL Constant Coding Example

http://www.xilinx.com/txpatches/pub/documentation/misc/vivado_synthesis_ug_examples.zip

-- Description of a ROM with a VHDL constant
-- File: HDL_Coding_Techniques/rams/roms_constant.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity roms_constant is
  port (clk : in std_logic;
        en   : in std_logic;
        addr : in std_logic_vector(6 downto 0);
        data : out std_logic_vector(19 downto 0));
end roms_constant;

architecture syn of roms_constant is

  type rom_type is array (0 to 127) of std_logic_vector (19 downto 0);
  constant 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"0030D", X"02341", X"08201", X"0400D",
      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"04112", 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"00870", 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"FF036", X"00301", X"00102", X"02237", X"04040", X"00304",
      X"04078", X"01110", X"02500", X"02500", X"0030D", X"02341", X"08201", X"0410D"
    );

begin

  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (en = '1') then
        data <= ROM(conv_integer(addr));
      end if;
    end if;
  end process;

end syn;
ROM Using Block RAM Resources Verilog Coding Example

// ROMs Using Block RAM Resources.
// Verilog code for a ROM with registered output (template 1)
module v_rams_21a (clk, en, addr, data);
input clk;
input en;
input [5:0] addr;
output reg [19:0] data;
(* ram_style = "block" *)
reg [19:0] data_tmp;
always @ (posedge clk)
begin
if(en)
  data <= data_tmp;
end
always @ * begin
  case(addr)
    6'b000000: data_tmp = 20'h0200A;
    6'b100000: data_tmp = 20'h02222;
    6'b000001: data_tmp = 20'h00300;
    6'b100001: data_tmp = 20'h04001;
    6'b000010: data_tmp = 20'h08101;
    6'b100010: data_tmp = 20'h00342;
    6'b000011: data_tmp = 20'h00300;
    6'b100011: data_tmp = 20'h0232B;
    6'b000100: data_tmp = 20'h08601;
    6'b100100: data_tmp = 20'h00900;
    6'b000101: data_tmp = 20'h0233A;
    6'b100101: data_tmp = 20'h00302;
    6'b000110: data_tmp = 20'h08300;
    6'b100110: data_tmp = 20'h02023;
    6'b000111: data_tmp = 20'h04001;
    6'b100111: data_tmp = 20'h04002;
    6'b001000: data_tmp = 20'h02310;
    6'b101000: data_tmp = 20'h00900;
    6'b001001: data_tmp = 20'h00241;
    6'b101001: data_tmp = 20'h02137;
    6'b001010: data_tmp = 20'h04002;
    6'b101010: data_tmp = 20'h02036;
  default: data_tmp = 20'h00303;
  endcase
end
endmodule
6'b010011: data_tmp = 20'h08300;
6'b110011: data_tmp = 20'h00301;
6'b010100: data_tmp = 20'h08201;
6'b110100: data_tmp = 20'h00102;
6'b010101: data_tmp = 20'h00500;
6'b110101: data_tmp = 20'h02237;
6'b010110: data_tmp = 20'h08101;
6'b110110: data_tmp = 20'h004004;
6'b010111: data_tmp = 20'h00602;
6'b110111: data_tmp = 20'h00304;
6'b011000: data_tmp = 20'h04003;
6'b111000: data_tmp = 20'h04040;
6'b011001: data_tmp = 20'h0241E;
6'b111001: data_tmp = 20'h02500;
6'b011010: data_tmp = 20'h00301;
6'b111010: data_tmp = 20'h02500;
6'b011011: data_tmp = 20'h00102;
6'b111011: data_tmp = 20'h02500;
6'b011100: data_tmp = 20'h02122;
6'b111100: data_tmp = 20'h0030D;
6'b011101: data_tmp = 20'h02021;
6'b111101: data_tmp = 20'h02341;
6'b011110: data_tmp = 20'h00301;
6'b111110: data_tmp = 20'h08201;
6'b011111: data_tmp = 20'h00102;
6'b111111: data_tmp = 20'h0400D;
endcase
end
end module
Dual-Port ROM VHDL Coding Example

-- A dual-port ROM
-- Implementation on LUT or BRAM controlled with a ram_style constraint
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity roms_dualport is
port (clk: in std_logic;
ena, enb : in std_logic
addra, addrb : in std_logic_vector(5 downto 0);
dataa, datab : out std_logic_vector(19 downto 0));
end roms_dualport;

architecture behavioral of roms_dualport 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 ram_style : string;
-- attribute ram_style of ROM : signal is "distributed";

begin

process (clk)
begin
if rising_edge(clk) then
if (ena = '1') then
    dataa <= ROM(conv_integer(addra));
end if;
end if;
end process;

process (clk)
begin
if rising_edge(clk) then
if (enb = '1') then
    datab <= ROM(conv_integer(addrb));
end if;
end if;
end process;

end behavioral;
Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx® Support website at: www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see: www.xilinx.com/company/terms.htm.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Vivado Documentation

10. Vivado Design Suite Tutorial: Design Analysis and Closure Techniques (UG938)
13. *7 Series DSP48E1 Slice User Guide* (UG479)
14. *Vivado® Design Suite Documentation:*
   