cādence®

Stratus High-Level Synthesis Reference Guide

Product Version 18.1
April 2018

Contents

1

Preface

This preface provides a general introduction to this manual, and contains the following sections:

About this Document

This guide provides detailed reference information for all Cadence Stratus^{TM} High-Level Synthesis (Stratus HLS) directives, project file commands, command line switches, environment variables, and Extended SystemC (ESC) functions. This manual is intended as a companion to the and other Stratus HLS documentation.

This manual assumes that you have a good understanding of C++ and SystemC and a basic understanding of how to write, compile, and execute a program using either of these languages.

Other Resources

A number of resources are delivered with Stratus HLS, including the following guides:

- **Stratus High-Level Synthesis Installation Guide**. Provides information about how to install and configure Stratus HLS, how to obtain Stratus HLS license, and tool compatibility with other Cadence products.

- **Stratus High-Level Synthesis Release Notes**. Provides information about the new features and supported platforms specific to each release of Stratus HLS. To view the Release Notes, please go to http://support.cadence.com/StratusHLS.
- **Stratus High-Level Synthesis Getting Started Guide**. Provides a simple introduction to the Stratus Integrated Development Environment (Stratus IDE) and a guided introduction to using the Stratus High-Level Synthesis flow on a simple example.
- **Stratus High-Level Synthesis User Guide**. Detailed information on using Stratus HLS for behavioral synthesis, design exploration, and design verification; the Stratus HLS design project and environment; integrating Stratus HLS with other design and verification tools; effectively using Stratus HLS command-line options and synthesis directives; and report generation and analysis.
- **Stratus High-Level Synthesis Reference Guide**. Detailed reference information on directives, project file commands, command line switches, environment variables, and Extended SystemC (ESC) functions.

If you have comments or questions about Stratus documentation, please e-mail support@cadence.com.

Using the Documentation

Stratus HLS documentation is located within your installation directory and may be accessed by loading into your web browser. This top-level file provides access to all of the current documentation, including manuals, application notes, and HTML documents.

The manuals are provided in Adobe Acrobat (PDF and HTML) formats. To view the PDF files, you must use Acrobat 5.0 or higher version. They are formatted for easy on-screen viewing. If you would like to print the manuals, you may want to print two pages of documentation per sheet of paper. To do this, specify two Pages Per Sheet in your Printer Properties dialog box. Also, choose Fit to Paper as the Page Scaling option in the Acrobat Print dialog box.

The PDF manuals include hyperlinks to other manuals. These hyperlinks require the manuals to be in the same directory. Therefore, if you copy the manuals outside the directory, make sure you copy all of the manuals to the same directory. Otherwise, you may have broken links between the documents.

Typographical Conventions

This and all Stratus HLS manuals use visual cues to help you locate and interpret information easily. These cues are explained in Table 1-2.

Typeface	**Represents**
`courier font`	Indicates code. For example: `do burst_response keeping {`
`courier bold`	Used to highlight important sections of code, like actions. For example: **`do`** `burst_response keeping {`
bold	The bold font indicates keywords in descriptive text. For example, the following sentence contains keywords for the **show ini** command and the **get_symbol()**routine: You can display these settings with the **show ini setting** command or retrieve them within ***e***code with the **get_symbol()** routine.
italic	The italic font represents user-defined variables that you must provide. For example, the following line instructs you to type the "write cover" as it appears, and then the actual name of a file: **write cover** *filename*
[ ] square brackets	Square brackets indicate optional parameters. For example, in the following construct the keywords "list of" are optional: **var** *name*: [**list of**] *type*
[ ] bold brackets	Bold square brackets are required. For example, in the following construct you must type the bold square brackets as they appear: **extend** *enum-type-name*: **[**name,…**]**
construct, …	An item, followed by a separator (usually a comma or a semicolon) and an ellipsis is an abbreviation for a list of elements of the specified type. For example, the following line means you can type a list of zero or more names separated by commas. **extend** *enum-type-name*: **[**name,…**]**

|	The pipe character indicates alternative syntax or parameters. For example, the following line indicates that either the **bits** or **bytes** keyword should be used: **type** *scalar-type* (**bits** | **bytes**: *num*)
Specman>	Denotes the Specman XP prompt
C1>, C2>, …	Denotes the Verilog simulator prompt
>	Denotes the VHDL simulator prompt
%	Denotes the UNIX prompt

Customer Support

If you have a problem using Stratus HLS or the documentation, you can submit a customer case to Cadence Support. When doing so, please provide enough information about the problem so that it can be investigated efficiently. Describe the problem in full, give the version of the software you are using, and state the exact circumstances in which the problem occurs. In addition, provide the following files:

- elog
- ixcom log
- `irun.log` file
- SW versions (IES, SN, IUS, UXE, VIPCAT, XP firmware version, OS version)
- HW platform
- Test file
- Config file

This section contains:

- Cases
- Use Cadence Online Support

Cases

Stratus HLS cases are your way of giving feedback, asking questions, getting solutions, and reporting problems. Unless told otherwise, Cadence Support staff will respond to your case. If Cadence support cannot answer your question, Cadence research and development personnel will get involved.

It is important to specify the severity level of the service request as accurately as possible. There are three levels of severity:

- **Critical** – You cannot proceed without a solution to the issue.
- **Important** – You can proceed, but you need a solution to the issue.
- **Minor** – You prefer to have a solution, but you can wait for it.

You can request Support to increase the severity level of an issue. Therefore, do not use **Critical** unless resolution of an issue is absolutely necessary and urgently required.

Using Cadence Online Support

Cadence encourages you to submit cases using Cadence Online Support. With Cadence Online Support you can also track your open cases.

To use Cadence Online Support to submit a case:

1. If you do not yet have a Cadence Online Support account, go to http://support.cadence.com/ and click *Register Now* under the *New User* heading.

   You must provide a valid *HostID* for any Cadence product (VIP, Stratus HLS, Specman, Incisive Enterprise Manager, or other). The *HostID* is contained in the SERVER line of your Cadence product license file.

   If you already have a Cadence Online Support account, then you only need to update your Cadence Online Support preferences to include a valid HostID for a Cadence product.

2. Log in to Cadence Online Support, and on the upper left side of the page click *Create Case* under the *Cases* heading.

   A form is presented for submission of your service request. Select *Stratus HLS* in the *Product* list box and click *Continue*. Follow the online instructions to complete the Service

Request.

Creating Group Privileges in Cadence Online Support

Sometimes it is beneficial to view the cases of others on your project.
To create group privileges in Cadence Online Support:

1. Open a Cadence Online Support service request by clicking *Create Case* under the *Cases* heading.
2. Select *Stratus HLS* in the *Product* list box and click *Continue*.
3. Fill in the required fields in the form presented.
   Explain in the Stated Problem text box that you want to create a group of users.
4. In the *People to notify upon Case creation* field, include the email addresses of the users you want to have group privileges.

   Icon Each person receiving group privileges must have a Cadence Online Support account.

5. Click *Submit Case* to complete the case.

Visit http://www.cadence.com/support/Pages/default.aspx to learn more about Cadence Global Customer Support and the Support Offerings we provide. For more details about our support process, visit http://www.cadence.com/support/Pages/support_process.aspx.

Cadence Online Support

Your Cadence Online Support (formerly "SourceLink") account lets you download releases of Cadence products, search for solutions to common problems, receive announcements on product releases, submit service requests, and track your open service requests. For more information about opening a Support account, see Customer Support.

2

Rules and Conventions

Reserved Keywords

Stratus HLS's directives, project file commands, and other keywords are reserved for their designated use. In addition, a number of identifiers and other terms are also reserved for use as keywords. You may not use Stratus HLS identifiers in your designs where they would be visible to a directive instantiation, use them as `define_hls_config` names, or `#define` them.

The following types of items are reserved:

Reserved Items	For a list, see ...
Class names	
Synthesis Control Attributes	"Synthesis Control Attributes"
-D macro names	
#define macro names	
ESC functions	"ESC functions"
Global function names	
Global variable names	
Programs/scripts	"Programs/Scripts"
Project file commands	"Project File Commands"
Synthesis directives	"Synthesis Directives"

Reserved Keywords	**Description**
`AGGRESSIVE`	Stratus HLS identifier for `HLS_UNROLL_LOOP`
`ALL`	Stratus HLS identifier for `HLS_UNROLL_LOOP`
`BDW_* / bdw_*`	BDW API commands/functions
`BEFORE_UNROLL`	Stratus HLS identifier for `HLS_DPOPT_REGION`
`COMPACT`	Stratus HLS identifier for `HLS_MAP_ARRAY_INDEXES`
`COMPLETE`	Stratus HLS identifier for `HLS_UNROLL_LOOP`
`CONSERVATIVE`	Stratus HLS identifier for `HLS_UNROLL_LOOP`
`HLS_* / hls_*`	Stratus HLS directives/identifiers
`HLS::*`	Stratus HLS identifier explicitly defined in the "HLS" namespace
`hls::*`	Reserved for future use
`hls_internal_*`	Environment variables
`STRATUS_VERSION`	A macro defined in C++ in the `stratus_hls.h` file. This symbol will contain a symbol that identifies the major release of stratus as an integer. The format is `YYYYNN` where `YYYY` is the release year, and `NN` is the release number. For example, for Stratus 15.1, `STRATUS_VERSION` would be 201501. This definition can be used to make SystemC code dependent on the Stratus release, or to make it dependent on whether the code is being used instead of with Cynthesizer or C to Slicon Compiler.
`STRATUS`	A macro defined in C++ code by `stratus_hls`, `stratus_vlg`, and `Bdw_extract`, it may be tested to determine whether one of these is the tool currently processing the C++ code.
`STRATUS_HLS`	A macro defined in C++ code by `stratus_hls`, it may be tested to determine whether `stratus_hls` is the tool currently processing the C++ code.
`STRATUS_VLG`	A macro defined in C++ code by `stratus_vlg`, it may be tested to determine whether `stratus_vlg` is the tool currently processing the C++ code.
`CYNW_* / cynw_*`	Stratus HLS fixed/floating point and CynWare interface IP template classes

`dec`	Decimal designator for stream I/O
`ESC_* / esc_*`	Extended SystemC functions
`hex`	Hexadecimal designator for stream I/O (base 16, using digits 0 to 9 and letters A to F)
`MEM_DISTANCE`	Stratus HLS identifier for `HLS_CONSTRAIN_ARRAY_MAX_DISTANCE`
`NO_ALTS`	Stratus HLS identifier for `HLS_DPOPT_REGION`
`NO_CHAIN_IN`	Stratus HLS identifier for `HLS_DPOPT_REGION`
`NO_CHAIN_OUT`	Stratus HLS identifier for `HLS_DPOPT_REGION`
`NO_CONSTANTS`	Stratus HLS identifier for `HLS_DPOPT_REGION`
`NO_CSE`	Stratus HLS identifier for `HLS_DPOPT_REGION`
`NO_DCE`	Stratus HLS identifier for `HLS_DPOPT_REGION`
`NO_DEAD`	Stratus HLS identifier for `HLS_DPOPT_REGION`
`NO_TRIMMING`	Stratus HLS identifier for `HLS_DPOPT_REGION`
`oct`	Octal designator for stream I/O (base 8, using digits between 0 and 7)
`OFF`	Stratus HLS identifier for various directives
`ON`	Stratus HLS identifier for various directives
`SIMPLE`	Stratus HLS identifier for `HLS_INDEX_MAPPING`

Stratus HLS Identifier Conflicts

All identifiers defined by Stratus HLS are defined in a "`HLS`" namespace. Therefore, conflicts can arise if global variables match Stratus HLS identifiers. Any conflicts can be resolved by disambiguating the affected name with the "::" operator.
For example, say you add the global variable `OFF` to *saxo_light.cc*:

```
bool OFF;
```

and you have a `HLS_UNROLL_LOOP` directive and another expression that both use `OFF`:

```
HLS_UNROLL_LOOP( OFF, "dont unroll" );
if ( OFF ) {
...
```

When attempting to build the behavioral simulation model, Stratus HLS will flag the instances as ambiguous and will display the following message:

```
"saxo_light.cc", line 37: error: "OFF" is ambiguous
        HLS_UNROLL_LOOP( OFF, "dont unroll" );
                    ^
"saxo_light.cc", line 38: error: "OFF" is ambiguous
        if ( OFF ) {
             ^
```

You can alter the code with the "::" operator to solve this issue:

```
HLS_UNROLL_LOOP( OFF, "dont unroll" );
if ( ::OFF ) {
```

If `OFF` were declared as a member of `saxo_light` (rather than as a global variable), Stratus HLS would instead display the following message:

```
"saxo_light.cc", line 37: error: no instance of overloaded function
          "HLS_UNROLL_LOOP" matches the argument list
            argument types are: (bool, char [12])
                  HLS_UNROLL_LOOP( OFF, "dont unroll" );
                  ^
```

In this case, the data member `OFF` hides the Stratus HLS identifier `OFF`, so there is no ambiguity. However, there is a data type mismatch. Being more specific about the Stratus HLS variable solves this problem:

```
HLS_UNROLL_LOOP( HLS::OFF, "dont unroll" );
if ( OFF ) {
```

Of course, it is always legal (and safe) to always use full qualification on all of your identifiers, like this:

```
HLS_UNROLL_LOOP( HLS::OFF, "dont unroll" );
if ( saxo_light::OFF ) {
```

This amount of qualification, however, is usually unnecessary.

Boolean Values in Command Line Switches

All of the syntax for command line switches that require boolean type values are expressed in these examples using `yes` and `no`. Note that in Stratus HLS the following strings are acceptable as boolean values:

- To set a switch to TRUE you can use: `on, true, t, yes, y, 1`
- To set a switch to FALSE you can use: `off, false, f, no, n, 0`

Therefore, all of the following are correct in setting command line switch values:

```
set_attr comm_subexp_elim 1
set_attr default_protocol false
set_attr output_style_reset_all on
set_attr output_style_two_process_fsm n
```

Placement of Directives

In general, following is the recommended placement for directives in your SystemC design code:

- Directives must be inside a block of code that Stratus HLS is going to process.
- Directives to affect a block of code must be place immediately after (inside) the open curly brace for that block of code.
- Directives to affect a single variable may be placed anywhere where that variable is in scope.

Recommended placement can vary among directives. For placement information specific to a particular directive, refer to the reference material for the directive in the *Stratus HLS Reference Guide*.

3

List of Commands by Category

This chapter divides the Stratus HLS command, directives, and attributes into various categories. Within each list, the items are provided in alphabetical order with a brief description and the detailed information is provided in the subsequent sections.

Project File Commands

Following is an alphabetical list of the commands available for your project file.

Keyword	Description
`define_analysis_config`	Defines a configuration for a code analysis run.
`define_equiv_config`	Defines a configuration for an equivalence checking run.
`define_external_array_access`	Defines the presence of an external array connection.
`define_hls_config`	Defines a new `hls_config` to the module specified if a `hls_config` by that name does not already exist.
`define_ide_extras`	Defines which files are displayed in the *Project* pane of the Stratus IDE.
`define_io_config`	Defines a new `io_config` to the module specified if an `io_config` by that name does not already exist.
`define_hls_module`	Defines a module to be synthesized, with `hls_configs` for the various behavioral synthesis configurations.
`define_logic_synthesis_config`	Defines a configuration for a logic synthesis run.
`define_post_elab_tcl`	Defines a Tcl script to be run after the elaboration phase completes.
`define_post_optim_tcl`	Defines a Tcl script to be run after the optimization phase completes.

`define_post_sched_tcl`	Defines a Tcl script to be run after the scheduling phase completes.
`define_post_rtl_tcl`	Defines a Tcl script to be run after synthesis completes successfully, and RTL is produced.
`define_post_run_tcl`	Defines a Tcl script to be run after stratus_hls completes, regardless of the phase to which it has advanced.
`define_power_config`	Defines a configuration for a power analysis run.
`define_report`	Creates a report with specified parameters.
`define_setup_tcl`	Defines a Tcl script to be run immediately after stratus_hls begins.
`define_sim_config`	Defines a configuration for a simulation you want to run.
`define_system_module`	Specifies paths to C++ source files to compile and link into simulations.
`enable_code_coverage`	Turns on code coverage measurements in the Verilog simulator.
`enable_license_waiting`	By default, Stratus HLS will fail immediately if a license is not available for either Stratus HLS, or for a tool invoked from Stratus. The `enable_license_waiting` project command causes Stratus HLS and any third party tools to wait for licenses when a license is unavailable.
`enable_waveform_logging`	Specifies recording options for the simulation.
`extract_extra_ports`	Specifies connection of memory extra ports across levels of hierarchy.
`extract_memory`	Specifies extraction of memories across levels of hierarchy.
`set_analysis_options`	Default name/value pairs to be passed to the code analysis tool.
`set_equiv_options`	Default name/value pairs to be passed to the equivalence checking tool.
`set_logic_synthesis_options`	Default name/value pairs to be passed to the logic synthesis tool.

`set_part_options`	Specifies attributes for library modules from managed libraries such as `cynw_cm_float`.
`set_power_options`	Default name/value pairs to be passed to the power analysis tool.
`set_systemc_options`	Specifies the SystemC class library to be used for simulation.
`use_systemc_simulator`	The name of the SystemC simulator that will be executed to cause a simulation to run.
`use_hls_lib`	Directory containing a `stratus_hls` parts library, math library, or memory wrapper library.
`use_tech_lib`	A path to a *.lib* ASIC library technology file.
`use_verilog_simulator`	The name of the make target that will be executed to cause a simulation to run.

define_analysis_config

A project file command.

Syntax

`define_analysis_config` *`name modconfigs`* `[ -options` *`optvals`* `] [ -command bdw_runhal | bdw_runspyglass ] [-verilog_files` *`vlog_files`* `]`

Parameters

`name`
The name by which the code analysis configuration will be referred. This name is used as part of various targets in *Makefile.prj*.

`modconfigs`
One or more module config definitions where a module configs is a list containing the module name, an optional representation (RTL_V or GATES_V), and an hls_config name for the module.

`[ -options` *`optvals`* `]`
An optional list of name-value pairs to pass to the code analysis tool script in the form of Tcl variables.

`[ -command` *`bdw_runhal | bdw_runspyglass`* `]`
A command string that specifies the analysis tool. It overrides the string specified for the `analysis_command`. The default value is `bdw_runhal`.

`[ -verilog_files` *`vlog_files`* `]`
A list of one or more Verilog files that should be included in the analysis.

Description

The `define_analysis_config` command defines a configuration for a code analysis run. A separate work library directory is maintained for each `define_analysis_config` under *bdw_work/analysis/<name>*.

Each `define_analysis_config` must be given one or more module configs each of which is a list containing a `hls_module` an optional representation and a `hls_config`:

```
{hls_module [representation] hls_config}
```

By position, these are:

1. **hls_module**: The name specified in a `hls_module` command for the project. Each `hls_module` may appear in only one specified array. Any `hls_module` that does not appear in an

`define_analysis_config` command will not be included in your code analysis run. The wildcard character '`*`' can be used to specify the hls_config to use for any submodule that has not yet been named in the same command. The '`*`' character cannot be used in the first module configuration.

2. **representation**: A three element list can be passed to `define_analysis_config`. If a three element list is passed, the second element specifies the representation to be analyzed. This can be specified as `RTL_V or GATES_V`. If a two argument list is passed to `define_analysis_config` the representation defaults to RTL_V.
3. **hls_config**: The name of a `hls_config` specified for the module named in #1. This `hls_config` will be used when performing behavioral synthesis with stratus_hls for this module by placing a `-D<hls_config>=1` on the stratus_hls command line. The `hls_config`s available for each module are defined in their `define_hls_module` commands. The default, if there are only two entries in the list, is the first `hls_config` defined for the module.

Use `-all` as the `define_hls_config` string to define a code analysis Makefile rule for every `hls_config` in the `hls_module`, eliminating the need for individual `define_analysis_config` statements. As you add or remove `hls_configs` to your project, the `-all` option automatically causes the list of code analysis Makefile rules to grow or shrink, as needed. Using the `-all` option with different `define_analysis_configS` containing different options will cause a matrix of code analysis Makefile rules to be created.

It is common to specify only one module from the project in an `define_analysis_config`, however you can specify multiple modules. An `define_analysis_config` may have as many *hls_module/hls_config* arrays specified as there are `hls_modules` in the project; the order in which they are specified is not relevant. If the modules are nested, however, the first array specified must be the top module, and the remaining modules must be instantiated at some point below the first module in the netlist. After the top module is specified, the order of the remaining modules is not relevant, even if there are multiple levels of nesting.

Optional parameters can be added to the `define_analysis_config` command to affect the code analysis tool's configuration. These are:

- **`-options {name1 value1} {name2 value2}`**: The *options* option can be used to send arbitrary parameters to the code analysis tool script in the form of Tcl variables. These options override the options specified globally in the `set_analysis_options` command. Whenever the command for an `define_analysis_config` is executed, a variable is set for the child process for each name/value pair given in the *options* option.

In addition, two standard options may be set for the `define_analysis_config`. These options are identical to those listed in set_analysis_options.

- **`-command altCmdStr`**: The *analysis_cmd* option overrides the `analysis_command` atrribute for the `define_analysis_config` in which it appears. This option is useful for `define_analysis_config` that require a customized script.

- **`-verilog_files {path1 path2...}`**: When the GATES_V representation is specified, it is necessary to include additional Verilog files for the technology gates. The `-verilog_files` argument is as described in the `define_sim_config` command.

See also:

- The `analysis_command` attribute in Project Attributes
- "set_analysis_options"

Example 1

```
define_analysis_config SP {saxo_light C_BASIC}
                    -options {VAL1 10}
```

This defines a code analysis Makefile rule named `SP`. It will be implemented from the `saxo_light` module, where the RTL was generated by Stratus HLS with the `C_BASIC` directive set. The variable `VAL1` with a value of `10` will be sent to the code analysis tool.

Example 2

```
define_analysis_config SP_ALL {saxo_light -all}
```

This defines a separate code analysis Makefile rule for every `hls_config` in the `saxo_light` module.

define_equiv_config

A project file command.

Syntax

`define_equiv_config` `name <lsconfig> | < -spec [lsconfig_name | simconfig_name | module_config] -imp [lsconfig_name | simconfig_name | module_config] > [-bbox_mul int] [-constraint path] [-module name] [-reset_conditions conditions] [-no_gui] [-ctype sim | ls] [-options optvals] [-command command] [-verilog_files vlog_files]`

Parameters

`name`

Is the name of this equiv config. It can be used with `make` to run the equiv config using the command line `make equiv_name`.

`lsconfig`

Specifies the name of a logic synth config for which equivalence should be checked. This will check the equivalence between the RTL used as input to logic synthesis and the gate level verilog generated from logic synthesis.

`-spec`

Specifies that the following config defines the Verilog RTL to be used as the "specification" for JasperGold Sequential Equivalency Checking App (Jasper-SEC). It defines the baseline for the comparison. When this parameter is used, the `-imp` parameter must also be used, or an error is generated.

`-imp`

Specifies that the following config defines the Verilog RTL to be used as the "implementation" for Jasper-SEC that is to be compared to the baseline or "specification" config. When this parameter is used, the `-spec` parameter must also be used, or an error is generated.

`lsconfig_name`

Specifies the name of a logic synth config that defines the configuration for equivalence checking. If no `-spec/-imp` parameter pair is given then the gate-level implementation will be checked for equivalence with the RTL used for the logic synth config. If a `-spec/-imp` parameter pair is given, then the RTL defined by the `-spec` logic synth config will be checked for equivalence with the RTL defined by the `-imp` logic synth config.

`simconfig_name`

Specifies the name of a sim config that defines the RTL to be used for the `-spec` or `-imp` parameter with Jasper-SEC. The RTL for all of the V_RTL modules in the sim config will be used in the comparison.

`module_config`
Specifies a 2-element TCL list consisting of an hls module name and an hls module config, like this: `{dut BASIC}`. All of the RTL for the given module config will be used in the comparison by Jasper-SEC.

`-options`
This flag is used to specify a list of name-value pairs for the equiv config.

optvals

Specifies a list of name-value pairs as in the [options] argument to define_equiv_config

`command`
Is an override for the project equiv_command attribute. If `-command` is not provided and `-spec/-imp` pair is used then `bdw_runjaspersec` will be used by default.

`verilog_files`
Specifies a list of one or more Verilog files that should be included in the analysis.

`-bbox_mul` *`int`*
Specifies the threshold for black boxing multipliers for Jasper-SEC. The default is 128. The `-bbox_mul` parameter is only used for Jasper-SEC.

`-constraint` *`path`*
Specifies the path to a Tcl file containing code to set constraints for Jasper-SEC. The -constraint parameter is only used for Jasper-SEC.

`-module` *`name`*
Specifies the name of the module to be analyzed. The default is the top level module of the RTL.

`-reset_conditions` *`conditions`*
Specifies the reset conditions if needed for Jasper-SEC.

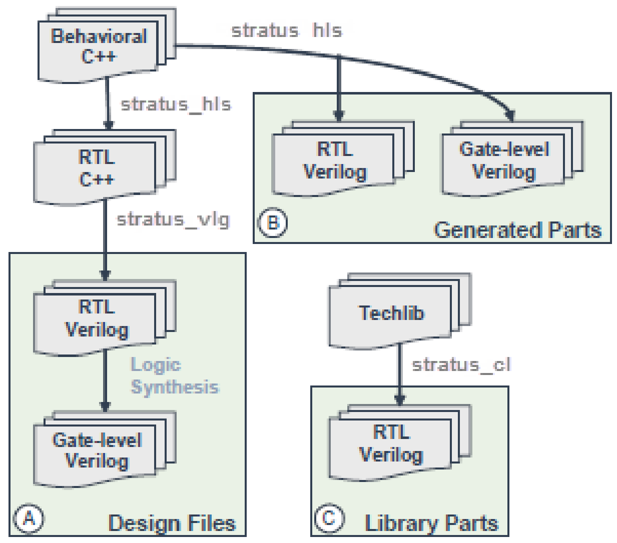
`-no_gui`
Specifies not to run the Jasper-SEC GUI when Jasper runs. The default is to run the Jasper-SEC GUI when the equivConfig is processed. The `-no_gui` parameter is only used for Jasper-SEC.

`-ctype`
This flag is used to force `define_equiv_config` to use either sim configs (`-ctype sim`) or ls configs (`-ctype ls`) when there are both sim configs and ls configs with the same name.

Description
The `define_equiv_config` command defines a configuration for an equivalence checking run. Defining an equivalence checking configuration is similar to defining a logic synthesis configuration.

An equivalence checking run requires a design that has been through gate-level synthesis, two

RTL-level designs, or a characterized library. Executing a `define_equiv_config` with a `define_logic_synthesis_config` will automatically execute the logic synthesis run associated with it, then compare the RTL and the gate-level implementation. The `define_equiv_config` must be inserted below the associated `define_logic_synthesis_config` in the project file. No such dependency is set up for characterized libraries. When comparing two RTL configurations with Jasper-SEC, logic synthesis does not need to be run, so no dependencies will be generated between the equiv config and the logic synthesis configs, but any needed hls_configs will be processed automatically.



There are three levels of equivalence checking:

- You can check all of the Stratus HLS RTL Verilog output for a design (A), including Stratus HLS-generated parts (B), against the output of logic synthesis. Library parts (C) are not included.
- You can check only parts generated by Stratus HLS (B), such as DpOpt parts or automatically-generated multiplexers. Library parts (C) are not included.
- You can check the characterized RTL library parts (C) against their gate-level counterparts.

An `define_equiv_config` defines which gate-level Verilog and RTL Verilog to compare, however the level of equivalence checking is primarily determined by which Makefile rule you run on the `define_equiv_config`. Your `define_equiv_config` commands will generally be structured the same regardless of whether you are checking all Verilog output, generated parts, or library parts.

All of the files generated by an equivalence checking run are written to the *bdw_work/equiv* directory. Data from each `define_equiv_config` will be stored in a separate directory named after the `define_equiv_config`.

Optional parameters can be added to the `define_equiv_config` command to affect the equivalence checking tool's configuration. These are:

- **`define_logic_synthesis_config`**: An `define_equiv_config` may or may not contain the name of a `define_logic_synthesis_config`.
    - If included, the `define_logic_synthesis_config` will provide most of the necessary information for the equivalence checking run, such as the paths to the required Verilog files and the location of the simulation data file. The `define_logic_synthesis_config` is required if you are checking Stratus HLS RTL Verilog output for a design against the output of logic synthesis.
    - If you are only checking characterized RTL library parts against their gate-level counterparts, the `define_logic_synthesis_config` may be omitted.
- **`-options {name1 value1}... {nameN valueN}`**: The *options* option can be used to send arbitrary parameters to the logic synthesis script in the form of Tcl variables. These options override the options specified globally in the `set_equiv_options` command. Whenever the command for a `define_equiv_config` is executed, a variable is set for the child process for each name/value pair given in the *options* option.

  In addition, standard options may be set for the `define_equiv_config`. The list of options is identical to those listed in "set_equiv_options".
- **`-command scriptCommand`**: The *command* option overrides the `equiv_command` attribute and for the `define_equiv_config` in which it appears. This option is useful for `define_equiv_configs` that require a customized script.

- **`-verilog_files {file1.v … fileN.v}`**: The `verilog_files` setting contains a list of Verilog files that will be included in the equivalence checking run. These must be simulation (*.v*) libraries, not synthesis (*.lib* or *.fdb*) libraries. Because gate-level simulation libraries are supplied by fabrication technology vendors and are not part of Stratus's characterized libraries, you will need to either supply the simulation library filename to `verilog_files` or make the library global to the project with the `verilog_files` attribute.

If you have a set of Verilog files that should be included in *all* equivalence checking runs, use the `verilog_files` attribute instead; see Project Attributes. Note that files specified using `verilog_files` are also added to all Verilog simulations in your project.

See also, Project Attributes.

If you are using the Genus Synthesis Solutions version 17.1 or later for the logic synthesis output, there is a command change that requires you to use Conformal Equivalence Checker version 15.2 or later.

Examples

This defines an equivalence checking Makefile rule named `E1` that will compare the output from the `BASIC` logic synthesis run against the RTL that it came from.

```
define_equiv_config E1 BASIC -verilog_files ../techlib/TSMC018/tsmc18.v
```

An example of using the new syntax to run equivalence checking between the BASIC and PIPE_1 configs of a module called dut looks like this in project.tcl:

```
    define_logicsynth_config BASIC {dut BASIC}
    define_logicsynth_config PIPE_1 {dut PIPE_1}

    define equiv_config E1 -spec BASIC -imp PIPE_1 -bbox_mul 64 -constraint
    const_input.tcl -reset_conditions "~rst" -command bdw_runjaspersec
```

Here is an example using two hls configs directly:

```
    define equiv_config E1 -spec {dut BASIC} -imp {dut PIPE_1} -constraint
    const_input.tcl -reset_conditions "~rst"
```

If the define_equiv_config specifies an `lsconfig_name` or a `simconfig_name` with `-spec` or `-imp`, and there exists both a sim config and an ls config with the same name, an error will be generated by bdw_makegen to tell the user to change either the name of one of the configs, or use the `-ctype` flag to force the use of either sim configs or ls configs.

define_external_array_access

A synthesis Tcl command.

Syntax

`define_external_array_access` `[-to` *`top_module_name`*`] [-from` *`sub_module_name`*`]`

Equivalent Directive

None

Parameters

`top_module_name`

Specifies the top module where the array itself is instantiated. The parent module where the array is instantiated can be either an `hls_module` or a `system_module`. That is, the array can be in the testbench, and never be synthesized.

`sub_module_name`

Specifies all the sub-modules that access an external array instantiated in the top module. Any number of sub-modules can access the external array. Note that module names and not instance names are used in this specification; so, if there are multiple instances of a given module, they are treated the same way.

Description

The `define_external_array_access` command allows you to define the presence of an external array connection. It establishes a processing order for modules when jobs involving multiple modules are executed. Synthesis is done bottom-up. It provides rules about allowable combinations of simulations to be enforced at the project level. For example, an RTL parent cannot have a BEH child if there are external array connections.

It is also supported to have multiple `define_external_array_access` commands in the same project file with different `-to` options.

Example 1

In the example, the array itself is instantiated in module `top`. And, there is a hierarchical connection from `top` to `sub1` through `hier`, and from `top` to `sub2` through `heir`.

```
# Definition of hls_modules
define_hls_module top top.cpp
define_hls_module hier hier.cpp
define_hls_module sub1 sub1.cpp
define_hls_module sub2 sub2.cpp
define_hls_config * BASIC

# Specification of the connection topology
define_external_array_access -to top -from hier.sub1 -from hier.sub2
```

define_hls_config

A project file command.

Syntax

`define_hls_config` `module name [ -io_config ioc ] [ attribute_values ] [ compiler_options ] [ -setup_tcl script] [ -post_elab_tcl script] [ -post_optim_tcl script] [ -post_sched_tcl script] [ -post_rtl_tcl script] [ -post_run_tcl script]`

Parameters

`modules`

Specifies the name of the `hls_module` to add the `define_hls_config` to. The wildcard character ('`*`') can be used to select multiple `hls_module`s to add the `define_hls_config` to (see examples below for more details). If a `hls_config` of the same name already exists for a `hls_module`, the `add_hls_config` has no effect for that module.

`name`

Specifies the name of the `hls_config` that will be added to the module.

`[ -io_config` *`ioc`* `]`

Specifies the `io_config` that this `define_hls_config` should be associated with. If omitted, the `define_hls_config` is associated with the *closest previous* `io_config` specified in either a `define_hls_module` command or with an `define_io_config` command.

`[ attribute_values ]`

Specifies synthesis control attributes that should be set when stratus_hls executes processes this `hls_config`. Each option is specified using `--<attribute>=<value>` syntax .Any number of options can be specified. Options specified in a `define_hls_config` take precedence over global attribute settings made in the project.tcl file.

`[ compiler_options ]`

Specifies C pre-processor options like -I and -D that should be set when stratus_hls processes this `hls_config`. The same syntax used for the C++ compiler, `-D<name>[=<value>]`, should be used. Any number of options can be specified. Options specified in a `define_hls_config` take precedence over global attribute settings made in the project.tcl file.

`[ -setup_tcl` *`script`*`]`

Specifies a Tcl command or script to be run immediately after stratus_hls begins. Overrides any define_setup_tcl command in the project file.

`[ -post_elab_tcl` *`script`*`]`

Specifies a Tcl command or script to be run after the elaboration phase completes. Overrides any define_post_elab_tcl command in the project file.

`[ -post_optim_tcl  `*`script`*`]`
Specifies a Tcl command or script to be run after the optimization phase completes. Overrides any define_post_optim_tcl command in the project file.

`[ -post_sched_tcl  `*`script`*`]`
Specifies a Tcl command or script to be run after the scheduling phase completes. Overrides any define_post_sched_tcl command in the project file.

`[ -post_rtl_tcl  `*`script`*`]`
Specifies a Tcl command or script to be run after stratus_hls has completed successfully and RTL has been written. Overrides any define_post_rtl_tcl command in the project file.

`[ -post_run_tcl `*`script`*`]`
Specifies a Tcl command or script to be run after stratus_hls completes, regardless of what phase it has run through. Overrides any `define_post_run_tcl` command in the project file.

For each *`tcl`* option, the *script* parameter can be a single Tcl command in quotes, or it can be a multi-line script surrounded by `{}`.

Placement
In the project file after all `hls_module` definitions that you want the `define_hls_config` to apply to

Description
The `define_hls_config` command can be used outside of a `hls_module` definition to add a new `hls_config` to the module specified if a `hls_config` by that name does not already exist. You can apply a single `define_hls_config` command to multiple `hls_module`s using a wildcard character ('*') to match more than one `hls_module` name that has been previously specified in the project file.

See also define_hls_module.

Example 1
```
define_hls_module modC modC.cc
define_hls_config modC BASIC -DFAST=1 --cse=on
```

This example results in adding a `hls_config` named *BASIC* to `hls_module` *modC.* The macro `FAST` is set to the value 1, and the `cse` synthesis control option is set to `on` during synthesis with Stratus HLS.

Example 2
```
define_hls_module M1 M1.cc
define_hls_module M2 M2.cc
define_hls_module C1 C1.cc
define_hls_config M* LAT10 --sched_effort=low
```

This example results in adding a `hls_config` named *LAT10* to `hls_module`s *M1* and *M2*. The name of `hls_module` *C1* does not match the glob 'M*' and therefore no `hls_config` was added to module *C1*.

Example 3

```
define_hls_module M1 M1.cc
define_hls_module M2 M2.cc
define_hls_config * LAT10
define_hls_module C1 C1.cc
```

This example results in adding a `hls_config` named *LAT10* to `hls_module`s *M1* and *M2*. The definition of `hls_module` *C1* comes after the `define_hls_config` command and therefore no *LAT10* `hls_config` is added to module *C1.*

Example 4

```
define_hls_module M1 M1.cc
define_io_config M1 PIN2
define_hls_config M1 CFG2
```

This example results in adding a `hls_config` named *CFG2* to `hls_module` *M1* associated with the *PIN2* `io_config`.

Example 5

```
define_hls_module M1 M1.cc
define_io_config M1 PIN1
define_io_config M1 PIN2
define_hls_config M1 CFG2_SA1 -io_config PIN1 --sched_effort=low
```

This example results in adding a `hls_config` named *CFG2_SA1* to `hls_module` *M1* associated with the *PIN1* `io_config`.

Example 6

```
define_hls_config M1 CFG1 -post_elab_tcl {
  pipeline_loop [find -loop MAIN_LOOP]
} -post_rtl_tcl {
  report summary
  report ops_by_state -silent -verbose
}
```

This example shows specification of a `-post_elab_tcl` option with a script that pipelines a loop, and a `-ops_by_state` option with a script that generates 2 reports.

define_hls_module

A project file command.

Syntax

`define_hls_module` *name source_files* [-template *template_or_namespace*] [-verilog_files *verilog_files*]

Parameters

name
Defines the name of the RTL module, and if -template is not specified, the name of the SystemC module.

source_files
Defines either a single path, or a list of paths to C++ source files for the module.

template
A string containing a C++ template instantiation or namespace-specific declaration if the module is templated or in a namespace

verilog_files
Specifies a single path to a verilog source file, or a list of paths.

Description

The `define_hls_module` command defines a module that is to undergo behavioral synthesis. Each `define_hls_module` must have a name and a path to the main source file. The main source file may call other source files, as needed.

In most cases, one or more hls_configs will be defined for each hls_module using the `define_hls_config` command. Synthesis is performed on hls_configs.

The options are as follows:

- **`verilog_files`**: The `verilog_files` setting contains a list of Verilog files that will be included in a co-simulation run with the module. These files will be placed on the Verilog compiler command line following the name of the module's synthesized *<module>_rtl.v* file, but before any `-y` arguments for libraries. Therefore, modules in these files will be used in preference to Verilog modules in `use_hls_lib` libraries.

To set multiple files, do one of the following:

- Specify multiple filenames in one `verilog_files` command (braces are required if more than one file is specified):

```
-verilog_files {file1.v file2.v}
```

- Specify a directory and file extension (if the files are in the same directory and have the same file extension):

```
-verilog_files /path/*.v
```

- Specify an individual file:

```
-verilog_files file.v
```

that contains `` `include`` statements of other Verilog files:

```
`include "/path1/file1.v"
`include "/path2/file2.v"
```

If you have a set of Verilog files that should be included in *all* Verilog simulations rather than for a specific module, use the `verilog_files` attribute instead; see Project Attributes.

- **`template`**: The `template` option specifies a template instantation that is to be used to build this `define_hls_module`. The -template option can also be used when the module is in a separate namespace. When a define_hls_module is defined by a templated `SC_MODULE`, an explicit set of template parameters or namespaces must be specified. For example, if a templated `SC_MODULE` is defined as follows:

```
template <typename T>
SC_MODULE(filter) {
```

and is instantiated using two different types in a parent module like this:

```
SC_MODULE(dut) {
   ...
   filter< sc_uint<8> > m_filter8;
   filter< sc_uint<16> > m_filter16;
   ...
```

then there must be two `define_hls_module` statements, each with a -`template` option specifying the template instantiation:

```
define_hls_module filter8 dut.cpp -template "filter< sc_uint<8> >"
   ...
define_hls_module filter16 dut.cpp -template "filter< sc_uint<16> >"
   ...
```

If an SC_MODULE is declared in a namespace as follows:

```
namespace my_designs {
    SC_MODULE(dut) {
        ...
    };
}
```

Then a -template argument must be used to identify the module in C++.

define_hls_module dut dut.cpp -template "my_designs::dut"

If multiple modules have the same name, but different namespaces, then a separate `define_hls_module` command must be added for each, using unique names for the hls_modules.

See also:

- define_hls_config
- define_io_config
- Stratus HLS targets in the *User Guide*
- Preparing for Behavioral Synthesis in the *User Guide*
- Using C++ templates with define_hls_modules in the *User Guide*

Example 1

```
define_hls_module mymod mymod.cc
define_hls_config mymod BASIC_C
define_hls_config mymod FLATTENED_C
```

This example defines a synthesizable module named `mymod` whose main source file is *mymod.cc*. It has two `define_hls_configs`: `BASIC_C` and `FLATTENED_C`. A directives header file for this module might look like this:

```
#if defined (BASIC_C)
<stratus_hls directives for BASIC_C>
#elif defined (FLATTENED_C)
<stratus_hls directives for FLATTENED_C>
#endif
```

A `define_sim_config` named A that contains the mymod module using the `FLATTENED_C` `define_hls_config` in Verilog RTL would look like:

```
define_sim_config A {mymod RTL_V FLATTENED_C}
```

A `define_logic_synthesis_config` named B that contains the `mymod` module using the `FLATTENED_C` `define_hls_config` would look like:

```
define_logic_synthesis_config B {mymod FLATTENED_C}
```

Example 2

```
define_hls_module dct { dct.cc algs.cc }
define_hls_module idct { idct.cc algs.cc }

define_hls_config * BASIC
define_hls_config * FAST
```

Sometimes a module to be synthesized includes more than a single source file. This example shows how you can list all of the source files in the `define_hls_module` command. Note that algs.cc file is specified in both modules. Also, the '*' wildcard character is used to avoid the necessity of listing the module in the hls_configs defined for it.

Example 3

project.tcl:

```
define_hls_module filter8 filter.h
   -template "my_filter< sc_uint<8> >"

define_hls_module filter16 filter.h
   -template "my_filter< sc_uint<16> >"
```

filter.h:

```
template <T>
SC_MODULE {
   sc_in_clk  clk;
   sc_in<bool> rst;
   sc_in<T> data;
   ...
```

This examples shows how an SC_MODULE using a C++ template can be specified in project.tcl. Two different `hls_modules` are defined, each with a different set of template parameters specified with the `-template` argument.

define_ide_extras

A project file command.

Syntax

`define_ide_extras [ -dirs dirs ] [ -files files ] [ -exclude files ]`

Parameters

[-dirs dirs]

Specifies one or more directories whose contents should be displayed.

[*-files* files]

Specifies one or more files whose contents should be displayed.

[*-exclude* files]

Specifies one or more files that should not be displayed.

Placement

In the project file.

Description

The `define_ide_extras` command can be used to change which files are displayed in the Project pane of the Stratus IDE. Stratus IDE will automatically select a set of files to be displayed based on the source files specified for hls_modules and system_modules in the project. However, you can cause additional files to be accessible as well using `define_ide_extras`.

The `-dirs` option specifies a directory whose contents should be displayed. The `-files` argument specifies files that should be displayed, and the `-exclude` argument specifies files that should be omitted. Wildcards are supported. In order to specify multiple files or directories, specify the paths as a Tcl list.

Example 1

```
    define_ide_extras -dirs ../shared -exclude {*.cc *.cpp}
```

Adds the contents of the `../shared` directory to the *Project* tree, but omits `.cc` and `.cpp` files.

Example 2

```
    define_ide_extras -files { notes.txt spec.txt }
```

Adds the `notes.txt` and `spec.txt` files to the *Projects* pane.

define_io_config

A project file command.

Syntax

`define_io_config` *`modules name`* `[` *`compiler_options`* `]`

Parameters

`name`

Specifies the name of the `io_config` that will be added to the module.

`modules`

Specifies the name of the `hls_module` and/or the `system_module` to add the `io_config` to. The wildcard character ('*') can be used to select multiple `hls_module`s and/or `system_module`s to add the `io_config` to (see examples below for more details). If a `io_config` of the same name already exists for module, the `define_io_config` has no effect for that module.

`[` *`compiler_options`* `]`

Specifies optional an arbitrary sequence of "dash" options that will be sent to the compiler for BEH sims and to `stratus_hls` for synthesis for configs that have this`io_config.`

Placement

In the project file after all `hls_module` and/or `system_module` definitions that you want the `define_io_config` to apply to.

Description

The `define_io_config` command adds a new `io_config` to the module specified if an `io_config` by that name does not already exist. You can apply a single `define_io_config` command to multiple modules using a wildcard character ('*') to match more than one module name that has been previously specified in the project file.

See also:

- define_hls_module
- define_system_module

Example 1

`hls_module` modA modA.cc

```
define_io_config modA PIN -DUSE_PIN=1
```

```
define_hls_config modA BASIC
define_hls_config modA PIPE
```

This example results in adding an `io_config` named *PIN* to the *modA* module using the `define_io_config` command. It then adds a `hls_configs` named *BASIC* and *PIPE* to the *modA* module. Both hls_configs are associated with the PIN io_config because it was the most recently defined io_config, and because they do not use the `-io_config` option to specify a different one. The `USE_PIN` macro will be set to `1` when compiling and running `stratus_hls` on configurations using the PIN io_config.

Example 2

```
define_hls_module M1 M1.cc
define_hls_module M2 M2.cc
define_io_config M* PIN
define_io_config M* TLM
define_hls_config M* BASIC -io_config PIN
```

This example results in adding 2 an `io_configs` named *PIN* and *TLM* to the *M1* and M2 modules using the `define_io_config` command. Also, the BASIC hls_config is added to M1 and M2, and will use the PIN io_config as specified with the `-io_config` option. If the `-io_config` option had not been specified, hls_config *BASIC* would have used the *TLM* io_config because it was the most recently defined.

define_logic_synthesis_config

A project file command.

Syntax

`define_logic_synthesis_config` `name mod_configs [ -options optvals ] [ -command command ] [ -verilog_files files ]`

Parameters

`name`

The name by which the logic synthesis configuration will be referred. This name is used as part of various targets in *Makefile.prj*.

`mod_configs`

A set of name-value pairs giving module name and hls_config.

`optvals`

Specifies a list of name-value pairs to pass to the logic synthesis script in the form of Tcl variables.

`command`

A command string that overrides the string specified for the `logic_synthesis_command` project attribute.

`files`

Specifies a path, or a list of paths to Verilog files to be included and passed to the logic synthesis tool.

Description

The `define_logic_synthesis_config` command defines a configuration for a logic synthesis run. A separate work library directory is maintained for each `define_logic_synthesis_config`. Ordinarily, each of these directories would contain a database for the logic synthesis tool.

Each `define_logic_synthesis_config` must have at least one *module_config*. Each *module_config* is a Tcl list specified with curly braces that contains 1 or 2 strings. By position, they are:

- **`hls_module`:** The name specified in a `define_hls_module` command for the project. Each `hls_module` may appear in only one *module_config*. Any `hls_module` that does not appear in a `define_logic_synthesis_config` command will not be included in your logic synthesis run. The wildcard character '`*`' can be used to specify the define_define_hls_config to use for any submodule that has not yet been named in the same command. The '`*`' character cannot be used in the first module configuration.

- **`hls_config`**: The name of an `hls_config` specified for the module named in #1. The RTL Verilog result from this hls_config will be the one processed by the logic synthesis tool.

  Optionally, you can use `-all` in place of the `hls_config` string to define a logic synthesis configuration for every `hls_config` in the `hls_module`, eliminating the need for individual `define_logic_synthesis_config` statements for each hls_config. As you add or remove `define_hls_configs` to your project, the `-all` option automatically causes the list of logic synthesis Makefile rules to grow or shrink, as needed. Using the `-all` option with different `define_logic_synthesis_config`s containing different options will cause a matrix of logic synthesis Makefile rules to be created.

It is common to specify only one module from the project in a `define_logic_synthesis_config`, however you can specify multiple modules. A `define_logic_synthesis_config` may have as many *module_config* values as there are `hls_modules` in the project; the order in which they are specified is not relevant. If the modules are nested, however, the first *module_config* defined must be the top module, and the remaining modules must be instantiated at some point below the first module in the netlist. After the top module is specified, the order of the remaining modules is not relevant, even if there are multiple levels of nesting.

Optional parameters can be added to the `define_logic_synthesis_config` command to affect the logic synthesis tool's configuration. These are:

- **`-options {name1 value1} {name2 value2} ...`**: The *options* option can be used to send arbitrary parameters to the logic synthesis script in the form of Tcl variables. These options override the options specified globally in the `set_logic_synthesis_options` command. Whenever the synthesis command for a `define_logic_synthesis_config` is executed, a variable is set for the child process for each name/value pair given in the *options* option.

  In addition, a number of standard options may be set for the `define_logic_synthesis_config`. The list of options is identical to those listed in set_logic_synthesis_options.

  There are also a number of FPGA-specific options for FPGA logic synthesis configurations.

- **`-command altCmdStr`**: The *`-command`* option overrides the `logic_synthesis_command` attribute for the `define_logic_synthesis_config` in which it appears. This option is useful for `define_logic_synthesis_config`s that require a customized script. For example, if you are defining an FPGA logic synthesis configuration, use this option to specify the script that will run the FPGA tool instead of the ASIC tool.

- **`-verilog_files {file1.v file2.v ...}`**: The `-verilog_files` *option* is used to pass a list of extra verilog files to the logic synthesis tool.

See also:

- Defining Logic Synthesis Targets in the *User Guide*
- Defining Logic Synthesis Configurations in the *User Guide*

Example 1

```
define_logic_synthesis_config BASIC_G {saxo_light C_BASIC}
```

This defines a logic synthesis Makefile rule for a gate-level module named `BASIC_G`. It is implemented from the `saxo_light` module, where the RTL was generated by Stratus HLS with the `C_BASIC` directive set.

Example 2

```
define_logic_synthesis_config BASIC_W    {saxo_light C_BASIC}
                       -options {BDW_LS_WIRELOAD_MODEL tsmc18_wl10}
                                {BDW_LS_NOGATES 1}
                       -command "ac_shell bdw_runbg2.tcl"
```

This defines a logic synthesis Makefile rule named `BASIC_W` for a gate-level module. It will be implemented from the `saxo_light` module, where the RTL was generated by Stratus HLS with the `C_BASIC` directive set. The `BDW_LS_WIRELOAD_MODEL` variable will be set to `tsmc18_wl10`, RTL versions of the stratus_hls generated parts will be used instead of gate-level versions, and an alternative command will be used to run the logic synthesis tool.

Example 3

```
define_logic_synthesis_config BASIC_ALL {saxo_light -all}
```

This defines a separate logic synthesis Makefile rule for every `define_define_hls_config` in the `saxo_light` module.

Example 4

```
define_logic_synthesis_config BASIC_ALL      {saxo_light -all}
define_logic_synthesis_config BASIC_NG_ALL   {saxo_light -all}
                                  -options {BDW_LS_NOGATES 1}
define_logic_synthesis_config BASIC_W_ALL {saxo_light -all}
                         -options {BDW_LS_WIRELOAD_MODEL tsmc18_wl10}
```

This defines a matrix of logic synthesis Makefile rules. For every `define_define_hls_config` in the `saxo_light` module, Stratus HLS will create Makefile rules for each of the three `define_logic_synthesis_config`s with their various settings. If the `saxo_light` module had three `define_define_hls_config`s, then Stratus HLS would create nine logic synthesis Makefile rules.

Example 5

```
define_logic_synthesis_config MULTI{M2 C_BASIC1} {M1 C_BASIC3} {M0 C_BASIC2}
```

This tells Stratus HLS to create a rule that will perform logic synthesis on three modules with three different behavioral synthesis configurations. If these modules are nested, then module M2 is taken to be the top module. All of the modules' Verilog files will be loaded during the logic synthesis run.

Example 6

```
define_logic_synthesis_config FPGA_FLAT_A {saxo_light C_FLATTENED} -options \
{BDW_LS_FPGA_TOOL quartus} \
{BDW_LS_FPGA_PART 5SGXEA7H3F35C3} \
{BDW_LS_CLK_PERIOD 20.0} -command bdw_runquartus
```

This defines a logic synthesis Makefile rule named `FPGA_FLAT_A`. It will be implemented from the `saxo_light` module, where the RTL was generated by Stratus HLS with the `C_FLATTENED` directive set. It will run Altera Quartus, establishing a variety of standard settings for the target FPGA device.

FPGA logic synthesis can be performed using Xilinx Vivado by defining the following rule:

```
define_logic_synthesis_config FPGA_FLAT {saxo_light C_FLATTENED} -options \
{BDW_LS_FPGA_TOOL vivado} \
{BDW_LS_FPGA_PART xa7a75tfgg484-2I} \
{BDW_LS_CLK_PERIOD 20.0} -command bdw_runvivado
```

This defines a logic synthesis Makefile rule named `FPGA_FLAT`. It will be implemented from the `saxo_light` module, where the RTL was generated by Stratus HLS with the `C_FLATTENED` directive set. It will run Xilinx Vivado, establishing a variety of standard settings for the target FPGA device.

Example 7

```
define_logic_synthesis_config MULTI{top C_BASIC} {* C_BASIC}
```

Specifies that module *top* should be synthesized using the `C_BASIC` `define_define_hls_config`, and that all other define_hls_modules are submodules that should also be synthesized using the `C_BASIC` define_hls_config.

define_post_elab_tcl

A project file command.

Syntax

`define_post_elab_tcl` [*tcl_scripts*]

Parameters

[*tcl_scripts*]

Specifies either a single Tcl command, or a list of Tcl commands.

Description

`define_post_elab_tcl` defines a Tcl script to be run after the elaboration phase completes. At the post_elab access point, all design objects will have been recognized, and Tcl commands can be executed to either analyze the contents of the design, or to make settings that affect how the design is synthesized. For example, the following post_elab Tcl will set the loop named main_alg to be pipelined, and print a message if it was not already set to be pipelined by directives:

```
define_post_elab_tcl {

 set loop [find -loop main_alg]

 if { ![get_attr $loop is_pipelined] } {

 puts "Pipelining loop [get_attr name $loop]"
 pipeline_loops $loop




 }




}
```

define_post_optim_tcl

A project file command.

Syntax

`define_post_optim_tcl [`*`tcl_scripts`*`]`

Parameters

`[`*`tcl_scripts`*`]`

Specifies either a single Tcl command, or a list of Tcl commands.

Description

`define_post_optim_tcl` defines a Tcl script to be run after the optimization phase completes. At this stage, all micro-architecture transforms have been completed, so it is not possible to specify Tcl commands that affect the way the design is synthesized. So, the post_optim access point is used mainly for reporting. For example, the following will cause the `arrays` and `ops_by_function` reports to be generated:

```
define_post_optim_tcl {

 report arrays -silent

 report ops_by_function -silent

}
```

define_post_sched_tcl

A project file command.

Syntax

`define_post_sched_tcl [`*`tcl_scripts`*`]`

Parameters

`[`*`tcl_scripts`*`]`

Specifies either a single Tcl command, or a list of Tcl commands.

Description

`define_post_sched_tcl` defines a Tcl script to be run after the scheduling phase completes. The post_sched access point is used mainly for reporting. For example, the following causes the latencies report to be printed after scheduling:

```
define_post_sched_tcl {
  report latencies -silent
}
```

define_post_rtl_tcl

A project file command.

Syntax

`define_post_rtl_tcl [`*`tcl_scripts`*`]`

Parameters

[*tcl_scripts*]

Specifies either a single Tcl command, or a list of Tcl commands.

Description

`define_post_rtl_tcl` defines a Tcl script to be run after synthesis completes successfully, and RTL is produced. The post_rtl access point is used primarily for reporting. For example, the following causes the `timing` and `area` reports to be generated whenever Stratus successfully produces RTL:

```
define_post_rtl_tcl {

  report timing -silent

  report area -silent

}
```

define_post_run_tcl

A project file command.

Syntax

`define_post_run_tcl [`*`tcl_scripts`*`]`

Parameters

`[`*`tcl_scripts`*`]`

Specifies either a single Tcl command, or a list of Tcl commands.

Description

`define_post_run_tcl` defines a Tcl script to be run after stratus_hls completes, regardless of the phase to which it has advanced.

define_power_config

A project file command.

Syntax

`define_power_config` `*name* [sim_config_info]+ [ -module *module* ] [ -options *optvals* ] [ -command *command* ]`

Parameters

`name`

The name by which the power analysis configuration will be referred. This name is used as part of various targets in *Makefile.prj*.

`sim_config_info`

The name of a simulation config to be used for simulation data or a list where the first element is the name of a `sim_config` and the following elements are parameters to be passed to the Joules `read_stimulus` command to specify how to generate frames from the stimulus file generated by the `sim_config`. Only Joules supports more than one `sim_config_info` list. All of the `sim_config`s must use the same module configs or an error will be reported. This is to ensure the stimulus files all use the same RTL in the simulations.

`module`

Specifies the top-level hls_module to be analyzed. Required if the `sim_config` has more than on RTL module

`optvals`

Specifies one or more name/value pairs specifying options for the power analysis. These options can override options set with `set_power_options`.

`command`

Optional setting that defines the command script to be run. Example script names would be `bdw_runjoules` and `bdw_runpt`. If no command is specified, Joules will be run and the version of Joules used will be the version that is included with the current Stratus HLS installation.

Location

The `define_power_config` should be located somewhere below the associated `sim_config` in the project file, since the sim_config named in the power config must already be declared.

Description

Stratus HLS's integration with works with Cadence Joules, Ansys PowerTheater™, and SpyGlass Power tools allowing you to obtain power estimates from Verilog RTL or gate-level simulations. Each power analysis run is defined using a `define_power_config` command. Defining a power analysis configuration is similar to defining a logic synthesis configuration.

A power analysis run requires existing simulation data. To log Verilog RTL or gate-level simulation data, enable FSDB or VCD logging using the `enable_waveform_logging` command in the *project.tcl* file. Executing a `define_power_config` will automatically run the simulation associated with it if necessary.

All of the files generated by a power analysis run are written to the *bdw_work/power* directory. Data from each `define_power_config` will be stored in a separate directory named after the `define_power_config`.

See also:

- set_power_options
- enable_waveform_logging

Examples

The following example defines a power analysis run named `P1` that will analyze simulation data from the `BASIC` simulation run:

```
define_power_config P1 BASIC
```

The following example define power config to run a power analysis on the fir module using the simulation data from the BASIC_V simulation config. Since no options are specified, the power analysis will be done on the fir module using all of the simulation data from the BASIC_V simulation config.

```
define_power_config BASIC_V_P BASIC_V -options {BDW_PWR_GCT_OPTS \ "-fanout root=5
branch=6 leaf=4"}
```

The following example uses 2 `sim_config`s and breaks one of them up into 100 equal sized frames. When the user runs "make power_BASIC_V_P3" from the command line or selects the BASIC_V_P3 power config in the `stratus_ide` and clicks the *Run* button, the necessary hls_configs will be synthesized and the BASIC_V_1 and BASIC_V_2 simulations will be run to generate the simulation data necessary for power analysis. Joules will then be run to generate the power reports for the module.

```
 define_power_config BASIC_V_P3 BASIC_V_1 {BASIC_V_2 -frame_count 100} -options
{BDW_LEF_LIB ../../mylibs/mylib.lef}
```

define_report

A project file command.

Syntax

`define_report` *`report_name`* `[-args` *`arg_list`*`] [-help_proc` *proc_name*] [-object *object_type*] [-repeat] [-report_proc *proc_name*] [-source *script_file*] [-summary *short descr*]

Parameters

`[-args` *`arg_list`*`]`

Specifies the arguments accepted. The format is identical to the format for Tcl proc arguments, and the arguments must match those in the specified list.

`[-help_proc` *`proc_name`*`]`

Specifies the name of a proc to call if the user specifies `-help` on report. If unspecified, a help command will be formulated automatically.

`[-object` *`object_type`*`]`

Specifies the type of object that is accepted. If omitted, any object is accepted. May specify a Tcl list.

`[-repeat]`

If specified, the report will be executed once for each object specified. If not specified, the proc will be executed once for each set of objects specified.

`[-report_proc` *`proc_name`*`]`

Specifies the name of the proc to call for the script. If not specified, calls a proc with the same name as the report.

`[-source` *`script_file`*`]`

Specifies a script to source. Not needed for scripts packaged with their `define_report` command that will be sourced by customers.

`[-summary` *`short descr`*`]`

Gives a short description string that will be printed with `report -help` when no report is specified.

The `define_report` command registers a reporting script so that it can be executed by name using the `report` command. The `define_report` command should be placed either in the project file, or in a file sourced from the project file. For example, a report named `ops_on_parts` that accepts `hls_config` objects can be registered as follows:

```
define_report ops_on_parts -object hls_config -report_proc print_ops_on_parts -source ops_on_parts.tcl
```

This report can be executed using the report command as follows:

```
> report ops_on_parts [find -hls_config BASIC]
```

The `report` command will call the `print_ops_on_parts` proc, passing the BASIC `hls_config` object as a parameter after sourcing the `ops_on_parts.tcl` file. Calls to the `report` command can appear anyplace Tcl is supported in Stratus.

define_setup_tcl

A project file command.

Syntax

`define_setup_tcl` [*tcl_scripts*]

Parameters

[*tcl_scripts*]

Specifies either a single Tcl command, or a list of Tcl commands.

Description

`define_setup_tcl` defines a Tcl script to be run immediately after `stratus_hls` begins. The design has not yet been loaded at the `setup` access point, but synthesis control attribute can be both read and written. For example, the following setup Tcl will set the `unroll_loops` synthesis control attribute to on, and print a message if it was not already on:

```
define_setup_tcl {

  if {[get_attr unroll_loops] != "on"} {

    set_attr unroll_loops on

    puts old_unroll [get_attr unroll_loops]
    }
}
```

define_sim_config

A project file command.

Syntax

`define_sim_config` *`name module_configs`* `[ -argv` *`args`* `] [` *`-io_config ioc`* `] [-nosdf] [-no_vendor_memories] [ -run_time` *`time`* `] [ -sys_files` *`paths`* `] [ -verilog_files` *`verilog_files`* `] [ -verilog_libs` *`vlog_libs`* `] [ -verilog_simulator` *`vlog_sim`* `] [ -verilog_top` *`top_modules`* `] [ -verilog_input_delay` *`input_delay`* `]`

Parameters

`name`
The name by which the simulation configuration will be referred. This name is used as part of various targets in *Makefile.prj*.

`module_configs`
A list of 1, 2 or 3-element lists specifying { [(BEH|RTL_V|GATES_V)] }.

`args`
A string containing a command line to pass to the simulation for the `sim_config` for access with `esc_argv()`.

`ioc`
The name of an io_config.

`time`
Specifies a simulation time in nano-seconds.

`paths`
Specifies a single path to a source file, or a list of paths.

`sys_files`
Specifies a single path to a C++ source file, or a list of paths.

`vlog_sim`
Specifies one of the supported Verilog simulators, or a user-defined simulator

`verilog_files`
Specifies a single path to a verilog source file, or a list of paths.

`verilog_top`
Specifies the name of a top module in Verilog, or a list of top modules in external files that should be loaded by the simulator for the `sim_config`.

`input_delay`

Specifies a delay to be applied to all inputs from SystemC to Verilog in nano-seconds.

Description

The `define_sim_config` command defines a configuration for a simulation that you want to run. The basic form for a `define_sim_config` is:

```
define_sim_config config_name {module_configuration} {module_configuration} ...
```

where a *`module_onfiguration`* is a Tcl list specified within curly braces that contains between 1 and 4 arguments:
{*module representation config instName*}

The arguments are specified in order as follows:

- **module**: The name specified in a `define_hls_module` or `define_system_module` command for the project. Wildcards are also supported for applying the same settings for multiple modules. The remaining three settings within the braces pertain to that module.

  - The `define_sim_config` command may have as many *configuration* groupings as there are modules in the project, with each module in a separate grouping.

  - Each module may appear in only one configuration in any given define_sim_config unless an instName is also provided for instance-specific simulation.

  - Any module that does not appear in a `define_sim_config` command will be simulated as a behavioral model for that `sim_config`.

- **representation**: This can be `BEH`, `RTL_V`, or `GATES_V`, indicating that the module is to be represented as behavioral C+, RTL Verilog, or gate-level Verilog, respectively.

  - The default, if no *representation* is specified, is `BEH`.

  - If `RTL_V` is specified, Verilog RTL models will be included in the simulation for both the design module itself and for each library component referenced by the design.

  - If `GATES_V` is specified, gate-level models will be included for each library element that has a gate-level representation.

    - If a `logic_synthesis_config` is specified with a `GATES_V` representation, the gate-level model that has undergone logic synthesis will be included in the simulation.

    - If an `hls_config` is specified with a `GATES_V` representation, the Verilog RTL model will be simulated with the gate-level library elements.

- **hls_config|system_config|logic_synthesis_config**: The name of a `hls_config`, `system_config` or `logic_synthesis_config` defined for the `module`.

The following table lists the available representation/configuration combinations:

Representation	Configuration	Simulation Uses
BEH	None	Original behavioral model
BEH	hls_config|system_config	Behavioral C++ version of the module
RTL_V	hls_config	RTL_V version of module + RTL_V hls_lib models
RTL_V	logic_synthesis_config	Not allowed
GATES_V	hls_config	RTL_V version of module + gate-level hls_lib models
GATES_V	logic_synthesis_config	Gate-level module + gate-level hls_lib models

- An `hls_config` can be specified with any of the three representations.

- If the representation is `RTL_V`, or `GATES_V`, the `hls_config` will be used when performing behavioral synthesis with stratus_hls on the module.
- If the representation is `BEH` and a `hls_config` is specified, then a behavioral model is built specifically for that `hls_config` where ordinarily the same behavioral model is used for all behavioral `define_sim_configs`. When this behavioral model is compiled, a `-D<hls_config>=1` is placed on the compiler command line so that the same `#ifdefs` that affect the module during behavioral synthesis will also affect the behavioral simulation model.
- If the representation is `BEH` and no `hls_config` is specified, the original behavioral model is used with no transformations.

- A `logic_synthesis_config` can be specified with the `GATES_V` representation. The `logic_synthesis_config`'s definition contains the `hls_config` used to produce the post-logic-synthesis gate-level module. The `logic_synthesis_config` definition must be located before the `define_sim_config` definition in the project file.

- If no `hls_config` or `logic_synthesis_config` is specified, and the representation is `RTL_V`, or `GATES_V`, the first `hls_config` defined in the module's`hls_module` setting is used.

- **inst_name**: If there is more than one instance of the module in the design, the optional *instName* argument allows you to define different `define_sim_config` settings for a named instance. The *configuration* for the instance must be specified in its own set of braces and must follow a set of braces that defines the default behavior for other instances in the module. For example, if there are three instances of module `M1` named `M1_a`, `M1_b`, `M1_c` and `define_sim_config INST {M1 BEH} {M1 RTL_V BASIC M1_b}` is specified, then instance `M1_b` will be simulated in RTL Verilog using the `BASIC hls_config` setting and all other instances of module M1 will be simulated in behavioral mode.

Several optional parameters can be added to the `define_sim_config` command to affect the simulation's configuration. These are:

- **- ioc**: If the `-io_config` option is specified, then any `hls_config` or `system_config` that is explicitly specified must have the named `io_config`. For modules that do not have a config specified, the default config will be the first config for that module which has the specified `io_config`. If the `-io_config` option is not specified, all explicitly specified configs must have the same io_config.
- **- verilog_files**: The optional `verilog_files` setting is used to add a set of Verilog source files to simulations that include at least one module configured as `RTL_V` or `GATES_V`. For example, adding -`verilog_files {module1.v module2.v}` to a `define_sim_config` command will result in the *module1.v* and *module2.v* files being compiled into that simulation. Braces are required only if more than one file is specified.

Gate-level simulations require models for the parts in the techlib and the path to the file(s) that contain the models is given with the `-verilog_libs` option. For example:

```
define_sim_config BASIC_GATES {saxo_light GATES_V LS_FLAT} -verilog_libs
TSMC_LIBS/20nm/verilog.v
```

If you have a set of Verilog files that should be included in *all* Verilog simulations rather than for a specific module, use the `verilog_files` attribute instead; see Project Attributes.

- **- verilog_modules**: This setting specifies the name(s) of additional top-level modules that may be required in the simulation. For example, some FPGA simulations require vendor-specific modules to be instantiated at the top level. Adding them here ensures that they are loaded by the simulator.
- **- verilog_libs**: This specifies the paths to simulation libraries for the `sim_config`. For Xilinx FPGA designs, `define_sim_config`s for post-synthesis simulations use the `unisims` version. Multiple paths may be specified in the same way as multiple paths in the `verilog_files` option, described above.
- **- sys_files**: The optional `sys_files` setting is used to add a set of C++ source files to a

`define_sim_config`. The given set of files will be compiled and linked into the simulation executable or shared library. This option provides a convenient method for using different testbench files in different simulations. For example, adding `[sys_files tb2.cc]` to a `define_sim_config` command will result in the *tb2.cc* file being compiled and linked into the SystemC simulation. Braces are required only if more than one file is specified.

- **- argv**: The optional `-argv` setting allows a set of command line arguments to be specified for a particular sim_config. This option is overridden by a `BDW_HUB_ARGV` Makefile variable, described in Adding variables that affect Makefile.prj in the *User Guide*. The elements of the given string are accessible via the *esc_argc()* and *esc_argv()* functions in either standalone SystemC simulations or Verilog cosimulations. These are also accessible via the `argc/argv` parameters to `sc_main` for standalone SystemC simulations. For example, adding `-argv "-datafile config2.dat"` to a `define_sim_config` will result in *-datafile* being returned from a call to esc_argv(1), and *config2.dat* being returned from a call to `argv(2)`. This is described further in Passing arguments into a simulation in the *User Guide*.
- **- verilog_sim**: The optional `verilog_sim` setting overrides the `use_verilog_simulator` command for the `define_sim_config` in which it appears. This option is useful for `sim_configs` that require a customized simulation script; this is described further in use_verilog_simulator.
- **- nosdf**: Some post-logic-synthesis simulations require an *.sdf* file to provide timing information from synthesis; others require that the *.sdf* file not be loaded into the simulation. The `nosdf` option disables the generation of code in the toplevel Verilog file that would try to load the *.sdf* file. The use of this option depends upon the type of simulation you are performing:
    - In the ASIC flow, post-logic-synthesis simulations require the *.sdf* file. Do not use the `nosdf` option in this type of simulation.
    - For the FPGA flow, gate-level simulations can be performed after logic synthesis.

When a simulation is executed, the following variables are set to provide context to the simulation:

- **BDW_END_OF_SIM_CMD**: A string that contains the command specified in *project.tcl* using the `end_of_sim_command` attribute. You may wish to include this variable in a user-written simulator target as a final shell command after simulation has completed.
- **BDW_SIM_CONFIG**: Set to the name of the `define_sim_config` that is being executed.
- **BDW_SIM_CONFIG_DIR**: Set to the directory where `define_sim_config`-specific data is stored. This is ordinarily *bdw_work/define_sim_configs/<define_sim_config>*. A testbench might store a log file in this directory.

- **BDW_SIM_ENV_SETUP**: A string that contains shell commands for setting the variables that are guaranteed to be set during a simulation. This variable should be de-referenced before executing your own simulator command.
- **BDW_VLOG_DUT_FILES**: A string containing the paths to the Verilog files that need to be compiled for this simulation. Paths relative to the top-level Makefile may be given.
- **BDW_VLOG_LIBS**: A string containing `-y` arguments that reference all of the Verilog libraries used in this simulation.
- **BDW_VLOGSIM_ARGS**: A string containing all of the parameters that must be passed to the Verilog simulator at runtime. This includes options required for Hub cosimulation, along with any value specified earlier in your own Makefile for the same variable.
- **BDW_VLOGCOMP_ARGS**: A string containing all of the parameters that must be passed to the compile step for the Verilog simulator.
- **BDW_VLOGSIM_DEPS**: The targets on which the simulator target should depend.

See also:

- Defining simulation targets in the *User Guide*
- Preparing for Simulation in the *User Guide*

Example 1

```
define_sim_config HY_LOW_V {HYBRID RTL_V C_HY_LOW_TPT}
```

This simulation will run HYBRID in RTL Verilog mode, where the RTL was generated by Stratus HLS with the `C_HY_LOW_TPT` directive set. All other synthesizable modules will be simulated in behavioral mode.

Example 2

```
define_sim_config MIXED {HYBRID BEH C_HY_LOW_TPT} {SUBSYNTH RTL_V C_FAST}
```

This simulation will run `HYBRID` as a BEH model where the the SystemC is compiled with `-DC_HY_LOW_TPT=1`. The `SUBSYNTH` block will be simulated in Verilog that was synthesized with the `C_FAST` directive set. All other synthesizable modules will be simulated in behavioral mode compiled with no configuration-specific -D.

Example 3

```
define_logic_synthesis_config LS_FLAT {HYBRID C_BASIC}
define_sim_config GATES {HYBRID GATES_V LS_FLAT}
```

This simulation will run `HYBRID` as a gate-level Verilog model using the `LS_FLAT define_logic_synthesis_config`, which specifies that the RTL should be generated by stratus_hls with the `C_BASIC` directive set.

Example 4

```
define_system_module tb tb.cc
define_system_configs tb TB1
define_hls_module m1 m1.cc
define_hls_config m1 C1
define_sim_config SC1 {tb TB1} {m1 RTL_V C1}
```

This simulation will run the `C1 hls_config` for `hls_module m1` as a Verilog RTL model and `system_config TB1` for the tb `system_module`.

Example 5

```
define_system_module tb tb.cc
define_system_module main main.cc
define_system_config * TB1
define_hls_module m1 m1.cc
define_hls_module m2 m2.cc
define_hls_module m3 m3.cc
define_hls_config * C1
define_hls_config m1 C2
define_sim_config SC1 {* TB1} {m1 RTL_V C2} {* RTL_V C1}
```

This `define_sim_config` defines a simulation where:

- the two `system_modules`, *tb* and *main* use the *TB1* `system_config` .
- the `hls_module` *m1* use the *C2* hls_config and the `RTL_V` representation.
- the two `hls_modules`, *m2* and *m3* use the *C1* `hls_config` and the `RTL_V` *representation*.

Note that the wildcard in `{* RTL_V C1}` does not match module *m1* since it was specified earlier in the command. This makes the order significant. If the command has instead been:

```
define_sim_config SC1 {* TB1} {* RTL_V C1} {m1 RTL_V C2}
```

an error would be result because `{* RTL_V C1}` matches all modules, including module *m1*, which makes the `{m1 RTL_V C2}` specification a conflicting specification. As a rule, more specific specifications should appear first, followed by wildcards.

Example 6

```
# define a post-synthesis sim_config for Xilinx Vivado.
define_sim_config FPGA_SYN {saxo_light GATES_V FPGA_FLAT} \
-verilog_files ${XILDIR}/verilog/src/glbl.v -nosdf \
-verilog_modules glbl \
-verilog_libs ${XILDIR}/verilog/src/unisims
```

```
# define a gate level sim_config for Altera Quartus.
set family "stratixv"
define_sim_config FPGA_SYN_A { saxo_light GATES_V FPGA_FLAT_A } \
-verilog_files "$env(ALTERA)/eda/sim_lib/altera_primitives.v
$env(ALTERA)/eda/sim_lib/altera_lnsim.sv
$env(ALTERA)/eda/sim_lib/220model.v
$env(ALTERA)/eda/sim_lib/sgate.v
$env(ALTERA)/eda/sim_lib/cadence/${family}_atoms_ncrypt.v
$env(ALTERA)/eda/sim_lib/cadence/${family}_hssi_atoms_ncrypt.v
$env(ALTERA)/eda/sim_lib/${family}_hssi_atoms.v
$env(ALTERA)/eda/sim_lib/${family}_atoms.v"
```

Where `family` can be extracted by running by executing "`get_part_info -family <fpga_part>`" in the tcl prompt got by running "`quartus_sta -s`". Please note that the family name have to be converted to lower case and there should not be spaces in the family name (for example, "STRATIX V" should be used as "stratixv"). Thus user has to select the appropriate Altera family simulation files.

This example defines gate-level simulations for Vivado/Quartus FPGA tools. Both simulations use the `GATES_V` representation. The `verilog_libs` option specifies the path to Xilinx.

define_system_config

A project file command.

Syntax

`define_system_config` *`modules name [ -io_config`* `ioc`] [*options*]

Parameters

`modules`
Specifies the name of the `system_module` to add the `system_config` to. The wildcard character ('*') can be used to select multiple `system_module`s to add the `system_config` to (see examples below for more details). If a `system_config` of the same name already exists for a `system_module`, the `define_system_config` has no effect for that module.

`name`
Specifies the name of the `system_config` that will be added to the module.

`[ -io_config` `ioc`]
Specifies the io_config to use for this system_config (default the last io_config added to the module).

[`options`]
Specifies optional -D options that are specific for the given `system_config`.

Placement

In the project file after all `system_module` definitions that you want the `define_system_config` to apply to

Description

The `define_system_config` command can be used outside of a `system_module` definition to add a new `system_config` to the module specified if a `system_config` by that name does not already exist. You can apply a single `define_system_config` command to multiple `system_module`s using a wildcard character ('*') to match more than one `system_module` name that has been previously specified in the project file.

See also, define_system_module.

Example 1

```
define_system_module tb tb.cc
define_system_module top top.cc
define_system_config tb SYS_BEH
define_system_config top SYS_BEH
```

This example results in adding a `system_config` named *`SYS_BEH`* to `system_module` *tb* and *top.*

Example 2

```
define_system_module tb tb.cc
define_system_module top top.cc
define_system_config t* SYS_BEH
```

This example is another way to accomplish the same thing as example 1. This results in adding a `system_config` named *SYS_BEH* to `system_module` *tb* and *top.*

define_system_module

A project file command.

Syntax

`define_system_module` `[` *`name`* `]` *`source_files`*

Parameters

`name`

The name of the module and allows it to have configs defined for it.

`source_files`

List of one or more paths to C++ source files.

Description

The `define_system_module` command specifies files that are part of the SystemC system, but not synthesized. Each of the files specified in the `define_system_module` command will be compiled and linked into all simulations. Several `define_system_module` commands may be used.

There are 2 forms for the `define_system_module` command as shown above:

1. A simple form in which only source file names are specified. Each of the source files listed will be compiled and linked into the simulation. However, it is not possible to build different configurations of such modules in different define_sim_config commands.
2. A form similar to the `define_hls_module` command in which the module is given a name. In this form, only a single C++ source file name may be specified. The `define_system_module` may be configured differently in different `sim_configs`.

The second form of the `define_system_module` command may contain arguments are as follows:

- **define_system_config:** A `define_system_config` defines an individual behavioral synthesis configuration. A `define_hls_module` includes the names of one or more `hls_configs`. Each `define_hls_config` has a name and an optional set of -D and -I compiler directives.
- **ioc (or io_configs)**: An `ioc` describes the set of interfaces supported by the `system_module`.

You can add `io_configs` and `system_config`s outside of the `define_system_module` statement using the `define_io_config` and `define_system_config` command respectively. For more information on these commands, see "define_io_config" or "define_system_config".

Each source file specified in a `define_system_module` command will have C++ compilation performed on it as required using rules defined by Stratus HLS. The flags passed to the C++

compiler and linker can be altered by various user-settable Makefile variables. These are defined in *source_files* are generally *.cc* or *.cpp* files. Because they will not be synthesized by Stratus HLS, they may include and use any SystemC or C++ constructs.
source_files can also be *.c* files. However, if your ANSI C code includes code that is not C++ compatible, you must wrap each C function declaration in an `extern "C"` call. For more information, see Using system files with ANSI C code in the *User Guide* and Adding variables that affect Makefile.prj in the *User Guide.*

If you wish to compile non-synthesizable sources using rules in your own Makefile, then you should not use the `define_system_module` command for those files. Rather, you should add the relevant object files to the `BDW_EXTRA_OBJS` variable before including *Makefile.prj*. This will cause these objects to be linked into all simulation objects produced by *Makefile.prj*.

See also, System files in the *User Guide*.

Example 1

```
define_system_module testbench.cc checker.cc
```

This command will cause the *testbench.cc* and *checker.cc* files to be compiled as necessary and linked into all simulations. The following commands are equivalent to the line above:

```
define_system_module testbench testbench.cc
define_system_module checker checker.cc
```

Example 2

In this example, a testbench is configured to produce stalls only in some simulation configurations.

In *project.tcl*:

```
# A define_system_module that has 2 different system_configs.
define_system_module tb tb.cc
define_system_config tb Stall
define_system_config tb NoStall

# define_sim_configs that test both with and without stalling.
define_sim_config NoStall_C {tb NoStall} {dut RTL_V BASIC}
define_sim_config Stall_C {tb Stall} {dut RTL_V BASIC}
```

In *tb.cc*:

```
// Conditional code in the testbench based on the system_config.
for ( int i=0; i < N; i++ ) {
// Conditionally add waits to the sink in the testbench.
#  if defined(Stall)
     wait(4);
#  endif
```

```
    v = dout.get();
    printf(outfile, “out=%d\n”, (int)v);
}
```

Example 3

```
define_system_module tb tb.cc
define_system_config tb NoStall
define_system_config tb Stall
```

This example creates a `define_system_module` named *tb* and adds the *NoStall* and *Stall* `system_config`s to that module.

enable_code_coverage

A project file command.

Syntax

`enable_code_coverage` `[ -verilog ]`

Parameters

[-verilog]

Specifies that code coverage should be enabled for Verilog simulations.

Description

This command turns on code coverage analysis in Verilog simulations. As currently supported, this command will turn on the built-in code coverage analysis tools in the Verilog simulator being used.

- **ModelSim**: Turns on branch, condition, expression and statement coverage using the `-cover bces` option to the Verilog compiler. At the end of the simulation, a text report file is produced by running the command `coverage report -lines -zeros -byinstance -select bces`. The text output is placed in the file `bdw_work/sims/<sim_config>/coverage.txt`. Note: this command works with ModelSim version 6.0c or later.
- **NC-Sim/Incisive**: Turns on coverage using the `-coverage all` option to the `ncelab` command. This enables all supported code coverage types. During simulation, the `coverage -code` and `coverage -fsm` commands are used to record the actual data. After the simulation, the `iccr` command is run to generate a text report file named `bdw_work/sims/<sim_config>/coverage.txt`.
- **VCS and VCSi**: Turns on condition, FSM and line coverage (including continuous assignments) using the options `-cm cond+fsm+line -cm_line contassign` during compilation and simulation. The code coverage database is created in the directory `bdw_work/sims/<sim_config>/simv.cm`. After simulation, the cmView command is run to create text reports, which will be placed in separate text files in the directory `bdw_work/sims/<sim_config>/simv.cm/reports`. The coverage database remains available so you can run the cmView command manually after the simulation.

It is not possible to modify the above code coverage options and settings. If necessary, you can create a custom Makefile target as described in "use_verilog_simulator". Such a custom target can include the necessary code coverage options and commands.

enable_license_waiting

A project file command.

Syntax

```
enable_license_waiting [-all |-stratus | -off]
```

Description

By default, Stratus HLS will fail immediately if a license is not available for either Stratus HLS, or for a tool invoked from Stratus. The `enable_license_waiting` command causes Stratus HLS and any third party tools to wait for licenses when they are not immediately available. You can set the following values to the command:

- **`all`** - Enables license waiting for stratus_hls, and for any third party tools. This is the default if no arguments are specified.
- **`stratus`** - Enables license waiting only for stratus_hls.
- **`off`** - Disables license waiting

enable_waveform_logging

A project file command.

Syntax

enable_waveform_logging *(-vcd | -fsdb | -shm) [-scv] [-ports_only [depth]]*

Parameters

vcd | fsdb | shm
Either -vcd, -fsdb, or -shm must be specified.

scv
Options that determine how simulation trace data is logged.

depth
Specifies an optional hierarchical depth for logging below ports. Defaults to 1.

Description

The `enable_waveform_logging` command defines how simulation trace data is to be logged for later viewing in your schematic/waveform viewer. This logging is required for viewing waveform data.

The `enable_waveform_logging` command supports a number of options:

- **`-shm`**: Creates an Incisive compressed waveform database for all simulations. The database is created in the simulation output directory: *bdw_work/sims/<sim_config-name>/*<sim_config-name>.shm. By default, the database contains all ports and top level signals for each module in the simulation.

For `-shm` waveform logging format with the Joules integration, you must use Joules 16.11 or later.

- **`-vcd`**: Creates VCD (Value Change Dump) trace files for all simulations. Each simulation will produce a VCD trace file for all signals (including the internal signals in chained parts). The VCD file will be found in the simulation output directory: *bdw_work/sims/<sim_config_name>/systemc.vcd* for behavioral C++ simulations or *bdw_work/sims/<sim_config_name>/verilog.vcd* for RTL Verilog simulations.

  By default the trace files automatically include ports and top-level internal signals for synthesized parts as well as internal signals in chained parts. To dump additional signals, call `sc_trace` from inside your `SC_MODULE` constructors. To use the `view_* make` targets, make sure the file pointer you pass to `sc_trace` is the one that BDW opens.

```
sc_trace( esc_trace_file( esc_trace_vcd ), my_sc_sig, "my_sig_name" );
```

- **`-fsdb`**: Creates FSDB trace files for all RTL C++ or Verilog simulations for use with Verdi. This is the default setting. The FSDB file will be found in the simulation output directory: *bdw_work/sims/<sim_config_name>/systemc.fsdb* for behavioral C++ simulations or *bdw_work/sims/<sim_config_name>/verilog.fsdb* for RTL Verilog simulations.

  By default the trace files automatically include ports and top-level internal signals for synthesized parts as well as internal signals in chained parts. To dump additional signals, call `sc_trace` from inside your `SC_MODULE` constructors. To use the `view_* make` targets, make sure the file pointer you pass to `sc_trace` is the one that BDW opens.

  ```
  sc_trace( esc_trace_file( esc_trace_fsdb ), my_sc_sig, "my_sig_name" );
  ```

- **`-scv`**: When used with the `-fsdb` or `-shm` option, enables transaction logging for memories and interfaces. Transaction level logging allows waveform viewers to display transaction streams along with signal level waveforms. For more information on transaction level logging, see Transaction Level Logging in the *User Guide.*

- **`-ports_only[<nLevels>]`**:This option works with the `vcd` or `fsdb` option to limit the data being logged to ports, such as excluding internal signals or excluding ports on chained parts. *nLevels* is the number of levels of hierarchy (including the synthesized module) for which ports should be logged.

  Available *nLevels* settings are 0 through 3. If *`ports_only`* is specified without an *nLevels* setting, a setting of 1 is assumed. If no `ports_only` setting is specified, a setting of `0` is assumed.

  Only the ports for `SC_MODULE`s simulated with the BEH representation are logged. The internal signals of `BEH SC_MODULE`s are not logged.

The following table lists the data that is logged for both SystemC and Verilog for the various *nLevels* settings. Each item to be logged is represented by a letter in the accompanying illustration.

Data being logged	nLevels=0		nLevels=1		nLevels=2		nLevels=3	
	SysC	Ver	SysC	Ver	SysC	Ver	SysC	Ver

Ports on the DUT (a)	x	x	x	x	x	x	x	x
Internal signals for the DUT (b)	x	x		x		x		x
Ports on library parts in the DUT (c)		x		x		x		x
Ports on chained parts (d)		x		x	x	x	x	x
Internal signals for chained parts (e)	x	x				x		x
Ports on library parts inside chained parts (f)		x				x	x	x

Note that SystemC limits the width of signals that can be traced. If a signal or port in your design is wider than 1023 bits, Stratus HLS will emit a warning:

```
Simulation will not trace signal 'X' because it is wider than 1023 bits.
```

Simulation logging can be turned on and off by editing the project file to add, change, or remove the `enable_waveform_logging` command. For example, if you have the following command in your project file:

```
enable_waveform_logging -vcd
```

and you wish to run your next simulation without logging, you can edit the project file to contain:

```
#enable_waveform_logging -vcd
```

and logging will not be done during subsequent simulations. Re-compilation of source files is required only when adding or removing `enable_waveform_logging -fsdb`, since presence or absence of `enable_waveform_logging -fsdb` controls whether references to Synopsys runtime software can be safely present.

The `BDW_TRACE_SIM` environment variable can also be used to control whether logging is performed in a given simulation:

- If `BDW_TRACE_SIM` is set to `1`, `ON`, or `on`: if the project file contains a `enable_waveform_logging` command, logging will be done as specified by the `enable_waveform_logging` command.

- If `BDW_TRACE_SIM` is set to `0`, `OFF`, or `off`: no logging will be done, regardless of whether the project file contains a `enable_waveform_logging` command.

Note that the `BDW_TRACE_SIM` environment variable can never be used to enable logging when there is no `enable_waveform_logging` command in the project file.

Since re-compliation is required when adding or removing a `enable_waveform_logging -fsdb` command, we recommend use of `BDW_TRACE_SIM` for FSDB users.

See also:

- Specifying how simulation trace files are logged in the *User Guide*
- Checking schematic and waveform data in the *User Guide*

Example

```
enable_waveform_logging -fsdb -ports_only 1
```

extract_extra_ports

A project file command.

Syntax

`extract_extra_ports` `[ -from` *`modinst`* `] [ -to` *`module`* `[` *`config`* `] ] [-type` *`memtypes`* `]`

Parameters

`modinst`

Specifies either a single hls_module name, or a hierarchical path through modules, optionally followed by a wildcarded memory instance spec

`module`

Specifies a single hls_module name

`config`

Specifies an hls_config name to which the extraction will be done.

`memtypes`

Specifies the wildcarded memory type spec. ,

Placement

In the project file after the hls_module referenced in the command have been defined.

Description

The `extract_extra_ports` command specifies how extra ports on memories are to be connected across levels of hierarchy. For an overview of this feature,see Extra Ports on Memories in the *User Guide.*

A list of memory names can follow the client module specification. The names are names of items in the behavioral model. They can be names of arrays, or names of explicitly instantiated memories. Partial or complete wildcards are supported. To access fields of structs, use the same 'dot' syntax you would use in C++. For example, `s.arr` references the arr member of a struct or class instance named s.

If no instance names are given, any memory that matches the specified -`type` is extracted. If -`type` is specified, and instance names are specified, only items that match both the type and the instance name are extracted. It is an error to specify neither a type nor an instance name. If all memories are to have their extra ports extracted, specify * as an instance name.

Example 1

```
extract_extra_ports -from {* *}
```

Specifies that for all memories instantiated by all hls_modules, extra ports are to be connected to ports on the hls_module.

Example 2

```
extract_extra_ports -type SRAM64X8 -from {dut1 *} -from {dut2 *}
```

Connect extra ports for memories of type SRAM64x8 in `dut1` and `dut2`.

Example 3

```
extract_extra_ports -to top -from {hier::* *}
```

If `top` is a hls_module that instantiates module `hier`, which in turn instantiates several hls_modules, each of which instantiates memories, extract extra ports for all memories in all children of `hier`, connect them through the `hier` module, and bind them to signals in module `top`.

Example 4

```
extract_memory -to main -from {top::hier::* *}
```

Similar to the previous example, except module main is a system_module. This effectively connects extra ports in leaf-level modules up through ports on the topmost RTL module.

extract_memory

A project file command.

Syntax

```
extract_memory  [ -from modinst ] [ -to module [ config ] ] [-type memtypes ] [ -share
]
```

Parameters

modinst
Specifies either a single hls_module name, or a hierarchical path through modules, optionally followed by a wildcarded memory instance spec

module
Specifies a single hls_module name

config
Specifies an hls_config name to which the extraction will be done.

memtypes
Specifies wildcarded memory type spec. Contains one or more memory names. These are names of memories from a memory library, not the names of instances in an accessing module.

Placement

In the project file after the hls_Modules referenced in the command have been defined.

Description

The `extract_memory` command specifies how memories are to be extracted from child modules and instantiated and possibly shared in a parent module. For an overview of this feature, see Specifying extraction and sharing in the *User Guide.*

A list of memory names can follow the client module specification. The names are names of items in the behavioral model. They can be names of arrays, or names of explicitly instantiated memories. Partial or complete wildcards are supported. To access fields of structs, use the same 'dot' syntax you would use in C++. For example, `s.arr` references the arr member of a struct or class instance named s.

If no instance names are given, any memory that matches the specified `-type` is extracted. If `-type` is specified, and instance names are specified, only items that match both the type and the instance name are extracted. It is an error to specify neither a type nor an instance name.

The `-share` option can contain 2 or more `-from` element. This specifies that the clients in the `-from` elements share access to the resource in a mutually exclusive way. The number of `-from` elements in a `-share` element defines the number of clients sharing access to the extracted resources. If the -`share` element is omitted and -`from` elements appear directly with in the `extract_memory` command, there is no sharing.

See also:

- Specifying extraction and sharing in the *User Guide*

Example 1

```
extract_memory -to hier -share -from {sub1 mem*} -from {sub2 mem*}
```

Extract all memories mapped to arrays starting with `mem` in modules `sub1` and `sub2` to parent module hier, and share them.

Example 2

```
extract_memory -type SRAM64X8 -to hier \
                -share -from sub1 -from sub2
```

Extract all memories of type SRAM64X8 in modules `sub1` and `sub2` to parent module hier, and share them.

Example 3

```
extract_memory -to hier -from {sub1 mem*} -from {sub2 mem*}
```

Extract all memories mapped to arrays starting with `mem` in modules `sub1` and `sub2` to parent module hier, but do not share them.

Example 4

```
extract_memory [to top] \
     -share -from {hier::sub1 mem*} -from {hier::sub2 mem*}
```

Extract all memories mapped to arrays starting with `mem` in modules `sub1` and `sub2` to parent module `top` through intermediate module `hier`, and share them.

set_analysis_options

A project file command.

Syntax

`set_analysis_options` *`name_value_pairs`*

Parameters

`name_value_pairs`
Name/value pairs that specify variables for code analysis execution.

Description

The `set_analysis_options` command establishes name/value pairs that are sent to the code analysis tool script in the form of Tcl variables. Each name/value pair is separated by braces, as follows:

```
set_analysis_options {name1 value1} {name2 value2}
```

If an individual option has multiple values, then the values should be surrounded by quotes or braces, as follows:

```
set_analysis_options {name1 "value1 value2"}
```

or

```
set_analysis_options {name1 {value1 value2}}
```

The variables specified using the `set_analysis_options` command can also be overridden for individual `define_analysis_config` commands, as described in "define_analysis_config".

The `bdw_runhal` and `bdw_runspyglass` options are:

- **BDW_ANALYSIS_RULES_FILE**: Specifies that path to a "rules" file to be used instead of the default HAL rules file. It is supported by both HAL and Spyglass.
- **BDW_ANALYSIS_OPTIONS**: Specifies HAL options to be used instead of the default HAL options. This is supported by both HAL and Spyglass.
- **BDW_ANALYSIS_EXTRA_OPTIONS**: Specifies HAL options to be used in addition to the default HAL options. This is supported by both HAL and Spyglass.
- **BDW_ANALYSIS_OPT_FILE**: Specifies the path to a file containing HAL options to be used instead of the default HAL options. This option is applicable for HAL only
- **BDW_ANALYSIS_NCBROWSE**: If this option is set to 1, "ncbrowse" is launched on the report file when the analysis finishes. This option is applicable for HAL only.

- **BDW_ANALYSIS_POLICY**: Sets a Spyglass "policy" and is applicable for Spyglass only.

See your Spyglass documentation for more information on these settings.

In addition, you may send arbitrary values to your code analysis tool by establishing your own name/value pairs. In this case, you also need to customize the script to make use of the variables. If there is no setting for this command, then the defaults for the code analysis tool script will be used.

See also, define_analysis_config.

Example

The following sends a variable named VAL1 with a value of 10 to the code analysis tool script and sets the Spyglass rules file to *hls_spyg_rules.spq*.

```
set_analysis_options {VAL1 10} {BDW_ANALYSIS_RULES hls_spyg_rules.spq}
```

set_equiv_options

A project file command.

Syntax

`set_equiv_options` `name_value_pairs`

Parameters

`name_value_pairs`
Specifies the list of name/value pairs that specify variables for equivalence checker execution.

Description

The `set_equiv_options` command establishes name/value pairs that are sent to the equivalence checking tool script in the form of Tcl variables. Each name/value pair is separated by braces, as follows:

```
set_equiv_options {name1 value1} {name2 value2}
```

If an individual option has multiple values, then the values should be surrounded by quotes or braces, as follows:

```
set_equiv_options {name1 "value1 value2"}
```

or

```
set_equiv_options {name1 {value1 value2}}
```

The variables specified using the `set_equiv_options` command can also be overridden for individual `define_equiv_config` commands, as described in "define_equiv_config".

The `define_equiv_config` command currently supports one standard option:
ultra Set to `on` to tell the tool to use Conformal Ultra instead of Conformal ASIC.

In addition, you may send arbitrary values to your equivalence checking tool by establishing your own name/value pairs. In this case, you also need to customize the script to make use of the variables. If there is no setting for this command, then the defaults for the equivalence checking tool script will be used.

See also:

- "define_equiv_config"
- "Cadence Conformal Integration" application note

Example

The following sends a variable named VAL1 with a value of 10 to the equivalence checking script and instructs the tool to use Conformal Ultra instead of Conformal ASIC.

```
set_equiv_options          {ultra on} {VAL1 10}
```

set_logic_synthesis_options

A project file command.

Syntax

`set_logic_synthesis_options` *`name_value_pairs`*

Parameters

`name_value_pairs`

Name/value pairs that specify variables for logic synthesis execution.

Description

The `set_logic_synthesis_options` command establishes name/value pairs that are sent to the logic synthesis script in the form of Tcl variables. Each name/value pair is separated by braces, as follows:

```
set_logic_synthesis_options {name1 value1} {name2 value2}
```

If an individual option has multiple values, then the values should be surrounded by quotes or braces, as follows:

```
set_logic_synthesis_options {name1 "value1 value2"}
```

or

```
set_logic_synthesis_options {name1 {value1 value2}}
```

If there is no setting for this command, then the defaults for the logic synthesis tool will be used. The variables specified using the `set_logic_synthesis_options` command can also be overridden for individual `define_logic_synthesis_config` commands, as described in define_logic_synthesis_config.

Standard options. The `define_logic_synthesis_config` command supports a number of standard options:

- **BDW_LS_CLK_PERIOD**: Specifies a clock period to be used for logic synthesis that overrides the period used by Stratus HLS.
- **BDW_LS_CLK_GATING**: Specifies the clock gating to be used for logic synthesis. The following values can be set:
  - **NONE**: The default value. With this setting no clock gating will be inserted..
  - **REPORT_ONLY**: With this setting, no clock gating will be inserted, but a clock gating report will be generated.
  - **ON**: clock gating insertion will be enabled.

BDW_LS_DO_DISSOLVE: Controls how the logic synthesis tool handles hierarchy.

-

  A value of "0" (the default) instructs the logic synthesis tool to:
  - Maintain the "dont_touch" setting (if any) for parts created by Stratus HLS.
  - Perform in-place optimizations on instantiated parts.
  - Perform logic synthesis optimization with medium effort.

  A value of "1" instructs the logic synthesis tool to:
  - Remove the "dont_touch" setting (if any) on all parts created by Stratus HLS.
  - Dissolve (ungroup and flatten) the hierarchy.
  - Perform logic synthesis optimization with increased effort.
- **BDW_LS_INCREMENTAL**: When set to 1, specifies that an incremental compile should be performed by logic synthesis instead of a full compile. This option defaults to off ("0").
- **BDW_LS_LEFLIB**: Specifies the path to the ".lef" file for the library specified with `BDW_LS_TECHLIB` to be used for logic synthesis. This option is required by Oasys realTime and is not currently used by other synthesis scripts.
- **BDW_LS_LIB_OPT_FILE**: Specifies the name of an option file in TCL format that will be sourced by the logic synthesis scripts prior to loading libraries. This can be used to control which parts from the library are used or ignored. This is only implemented for the standard DC script bdw_rundc.tcl.
- **BDW_LS_LIBNAME**: Specifies the library (volcano) generated with Blast Create. This library name is required in addition to the library path specified with the `BDW_LS_TECHLIB` option.
- **BDW_LS_NO_COMPILE**: Set to 1 to tell the logic synthesis tool to skip the standard scripts compile step. This would be used in a BDW_LS_OPTIONS_FILE to override the standard scripts compile. This is only implemented in the DC script bdw_rundc.tcl.
- **BDW_LS_NOGATES**: Set to 1 to tell the tool to use RTL versions of the stratus_hls generated parts instead of gate-level versions.
- **BDW_LS_NOTOUCH_ALL**: A value of "1" instructs Stratus HLS to fill the `BDW_LS_NOTOUCH_PARTS` list with all gate-level parts. This option is off ("0") by default.
- **BDW_LS_NOTOUCH_PARTS**: A list of gate-level modules that the logic synthesis tool is not to touch during the logic synthesis run. You can define this list or instruct Stratus HLS to fill in the list using the `BDW_LS_NOTOUCH_ALL` setting. If you define the list, then the `BDW_LS_NOTOUCH_ALL` setting is ignored.

- **BDW_LS_OPCONDITIONS**: Defines the operating conditions to be used from the chosen library.
- **BDW_LS_OPTION_FILE**: Specifies the name of an option file in TCL format that will be sourced by the logic synthesis scripts to provide synthesis tool specific settings. The commands in this file will be executed just prior to the compile step in the standard bdw_rundc.tcl script. This could be used to either modify settings prior to compilation, or do a totally custom compile by using BDW_LS_NO_COMPILE to disble the compilation step in the standard bdw_rundc.tcl script.
- **BDW_LS_RLCLIB**: Specifies the path to the captable file for the library specified with BDW_LS_TECHLIBl to be used for logic synthesis. It is currently only supported by the Oasys realTime script.
- **BDW_LS_RT_PHYSICAL**: Force Oasys realTime to run physical optimization instead of virtual. The default is to run optimize with -virtual. Setting this to 1 will run optimize without -virtual.
- **BDW_LS_SIZE_ONLY**: A value of "1" instructs Stratus HLS to fill the `BDW_LS_SIZE_ONLY_PARTS` list with all gate-level parts. The set_size_only command will be used on all of the parts in this list. This option is off ("0") by default.
- **BDW_LS_SIZE_ONLY_PARTS**: A list of gate-level parts that the logic synthesis tool will set to size_only during the logic synthesis run. You can define this list or instruct Stratus HLS to fill in the list using the BDW_LS_SIZE_ONLY setting. If you define the list, then the BDW_LS_SIZE_ONLY setting is ignored. A part being in the BDW_LS_SIZE_ONLY_PARTS list overrides it being in the BDW_LS_NOTOUCH_PARTS list.
- **BDW_LS_TECHLIB**: The path to the technology library and memory models for the logic synthesis tool. The library extensions of the various tools are generally *.db* (Design Compiler) and *.lib* (Genus).
- **BDW_LS_WIRELOAD_MODEL**: Specifies the name of the wireload model to be used for logic synthesis.

Arbitrary values. In addition to the named options above, you may send arbitrary values to your logic synthesis tool by establishing your own name/value pairs.

See also:

- "Passing options to your logic synthesis tool" (*UG*)

Example 1

```
set_logic_synthesis_options  {BDW_LS_TECHLIB ../../techlib/lib/slow.lib}
```

Example 2

```
set_logic_synthesis_options  {BDW_LS_TECHLIB "../techlib/t18.lib ../t16.lib"}
```

Example 3

```
set_logic_synthesis_options  {BDW_LS_TECHLIB ../../techlib/lib/slow.lib} \
                        {BDW_LS_DO_DISSOLVE 1}
```

Example 4

```
set_logic_synthesis_options  {BDW_LS_TECHLIB tsmc18.volcano} \
                      {BDW_LS_LIBNAME mylib}
```

Blast Create requires both a `BDW_LS_TECHLIB` and `BDW_LS_LIBNAME` to be specified.

set_part_options

A project file command.

Syntax

`set_part_options` *`name_value_pairs`*

Parameters

`name_value_pairs`
A list of name-value pairs that specifies one or more library modules and the value the attribute should be given. It has the following format:

```
[<lib_spec>:][<mod_name>|<mod_name_wildcard>.]<attrib_name>
```

where `<lib_spec>` is one of:

```
<lib_name> | <path>/<lib>.bdl | <path>/<libname>.tcl
```

Description

The `set_part_options` command is used to specify attributes for library modules from managed libraries such as cynw_cm_float. The command can be used in a project.tcl file to enable or disable specific library modules, or to specify build parameters such as "latency" for specific library modules.

Each *<name>* in an `set_part_options` command specifies an attribute on either a single module, or several modules. The format for *<name>* shown above has 3 segments:

- *<lib_spec>*: Specifies the name of a use_hls_lib, or the path of a *.bdl* file. Optional. If omitted, can apply to all libraries referenced in the project.
- *<mod_name>|<mod_name_regexp>*: Specifies either the name of a module, or a Tcl regular expression for a module. Optional. If specified, only attributes on matching modules will be affected. If not specified, attributes on any module will be affected.
- *<attrib_name>*: The name of the attribute to be affected. The attributes that can be set are defined by the modules in the libraries.
- The values specified in a `set_part_options` command interact with other attribute specifications as follows:
- Higher precedence than:
  - Default attribute value specs for the library module.

- Lower precedence than:
    - Values specified on the command line with the `attrib_value` attribute.
    - The values specified on the stratus_hls command line for things like use_tech_lib and clock_period when modules are built on the fly.

See also:

- See The cynw_cm_float Library in the *User Guide*
- *"attrib_value" in Synthesis Control Attributes*

Example

```
# Specify that the cynw_cm_float_unit from the cynw_cm_float
# library should be enabled.
set_part_options { cynw_cm_float:cynw_cm_float_unit.enabled 1}


# Specify that latency 1 should be used for all floating
# point multipliers.
# This can be overridden by an explicit modConfig command.
set_part_options { cynw_cm_float:*mul*.latency 1}
```

set_power_options

A project file command.

Syntax

`set_power_options` `name_value_pairs`

Parameters

`name_value_pairs`
Name/value pairs that specify variables for power analysis execution.

Description

The `set_power_options` command establishes name/value pairs that are sent to the power analysis tool script in the form of Tcl variables. Each name/value pair is separated by braces, as follows:

```
set_power_options {name1 value1} {name2 value2}
```

If an individual option has multiple values, then the values should be surrounded by quotes or braces, as follows:

```
set_power_options {name1 "value1 value2"}
```

or

```
set_power_options {name1 {value1 value2}}
```

The variables specified using the `set_power_options` command can also be overridden for individual `define_power_config` commands, as described in "define_power_config".

The `set_power_options` command currently supports the following standard options:

- **BDW_VOLTAGE**: Specifies the operating voltage to be used instead of the voltage in the *.lib* file. This option is not valid for Cadence Joules.
- **BDW_PWR_CLK_NETS**: Specifies the path to a PowerTheater "clock nets" specification file. The path is relative to the project directory.
- **BDW_PWR_START**: Specifies the start time in ns of the simulation data to be included in the power analysis. The default is 0.
- **BDW_PWR_FINISH**: Specifies the finish time in ns of the simulation data to be included in the power analysis. The default is the end of the simulation data.

> For Joules integration, the value specified for `BDW_PWR_START` and `BDW_PWR_FINISH` must also specify the units in ns/ps. The format is `<number><unit>`. For example:
>
> ```
> set_power_options {BDW_PWR_START 1000ps} {BDW_PWR_FINISH 10000ps}
> ```
>
> For other tools, the unit suffix is not supported, and values are considered to be in nano-seconds.

- **BDW_LEF_LIB** (For Joules integration): Specifies the path to a LEF library to be used during the power analysis.
- **BDW_CAP_TABLE** (For Joules integration): Specifies the path to a cap_table file to be used during power analysis.
- **BDW_PWR_STIM_OPTS** (For Joules integration): Specifies a string of options to be added to the `read_stimulus` command. Stratus HLS automatically adds the `-file` option and if the BDW_PWR_START or BDW_PWR_FINISH options are specified, we will add the appropriate `-start` and `-end` options to `read_stimulus`. Any options specified in this setting will be added to the end of the `read_stimulus` command options Stratus HLS automatically provides.
- **BDW_PWR_GCT_OPTS** (For Joules integration): Specifies a string of options to be passed to the gen_clock_tree command for power analysis.
- **BDW_PWR_PLOT_PWR_OPTS** (For Joules integration): Allows you to pass parameters to the `plot_power_profile` command. It's value is a list of one or more lists of parameters. A `plot_power_profile` command will be executed for each list of parameters. The default format will be PNG and the default output file will be in *bdw_work/power/<config_name>/<module_name>_power_profile.png*.
- **BDW_PWR_PLOT_ACT_OPTS** (For Joules integration): This option is used to pass parameters to the plot_activity_profile command. It's value is a list of one or more lists of parameters. A `plot_acitvity_profile` command will be executed for each list of parameters. The default format will be PNG and the default output file will be in *bdw_work/power/<config_name>/<module_name>_activity_profile.png*.
- **BDW_PWR_OPTION_FILE**: Specifies the path to a file to a file containing power analysis tool options. For PowerTheater, this file will be included using the `-i` command line option to the tools. This option is not valid for Cadence Joules.

In addition, you may send arbitrary values to your power analysis tool by establishing your own name/value pairs. In this case, you also need to customize the script to make use of the variables. If there is no setting for this command, then the defaults for the power analysis tool script will be used.

See also:

- "define_power_config"
- "Sequence PowerTheater Integration" application note

Example

The following sends a variable named VAL1 with a value of 10 to the power analysis script and limits the simulation data to include in the power analysis run to 10,000ns.

```
set_power_options{BDW_PWR_FINISH 10000} {VAL1 10}
```

set_systemc_options

A project file command.

Syntax

`set_systemc_options [ -version 2.2 | 2.3 ] [ -gcc 4.1 | 4.4 | 4.8 | external ]`

Parameters

`-version`
Specifies the version of the SystemC class library to be used for simulation. Stratus HLS supports SystemC versions 2.2 and 2.3.

`-gcc`
Specifies the version of the gcc to be used for simulation. Stratus HLS supports gcc versions 4.1, 4.4, and 4.8. With "`-gcc external`" the gcc found on the user's $PATH is used.

Description

The `set_systemc_options` command must be used to specify the SystemC class library to be used for simulation, if the `builtin` option is specified using the `use_systemc_simulator` command.

See:

- use_systemc_simulator
- Integration Options in the *User Guide*

use_hls_lib

A project file command.

Syntax

`use_hls_lib` *`path`*

Parameters

`path`

Directory containing a parts library such as a memory library, interface library, or floating point library. This directory must contain a *.bdl* library catalog file. If a relative path is used, it is assumed to be relative to the directory containing the top-level Makefile.

Description

The `use_hls_lib` command must appear in *project.tcl* for each library that is to be used by Stratus HLS in the project.

You may specify an interface library and/or a memory wrapper library defined in Stratus IDE. Add a separate `use_hls_lib` command for each library; the order of library entries is irrelevant. Stratus HLS will choose from amongst all of the parts in the libraries.

See also:

- Specifying ASIC technology libraries in the User Guide
- Explicitly instantiated (or external) memories in the *User Guide*
- Libraries in the *User Guide*
- *The cynw_cm_float Library in the User Guide*

Example 1

```
use_hls_lib memlib
```

Use a memory library defined in Stratus IDE.

Example 2

```
use_hls_lib "[get_install_path]/share/stratus/cynware/cynw_cm_float"
```

Use the cynw_cm_float library in the project.

use_systemc_simulator

A project file command.

Syntax

`use_systemc_simulator  simname`

Parameters

simname
The name of the SystemC simulator that will execute during SystemC simulation.

Description

A SystemC simulation will be run for each define_sim_config in the project.

Stratus HLS supports the following SystemC simulators:

- **incisive** or **ncsc:** SystemC simulator and SimVision debugger from Cadence.
- **builtin**: The SystemC simulator shipped with Stratus HLS (the default).
- **external**: Any user customized simulator.

Cadence recommends to use the SystemC and GCC included with Stratus HLS rather than using any user customized environment.

See also:

- Defining a SystemC simulator in the *User Guide*
- Debugging your program in the *User Guide*

Example

```
use_systemc_simulator incisive
```

use_tech_lib

A project file command.

Syntax

`use_tech_lib` *`path`* `[ -verilog_files` *`vlog_files`* `]`

Parameters

`path`

The path to a technology library file. Either a relative or full path may be specified. Relative paths are assumed to be relative to the top-level Makefile.

`vlog_files`

The path, or a list of paths to Verilog files to load for gate-level simulations. Any files specified here will be automatically excluded from `power_config` runs.

Description

The specified library file provides information about the target technology for your project, including delay and area information. This *.lib* file is provided by your ASIC vendor (not with Stratus HLS) using the Liberty format standard. This file can be encrypted using the Cadence encryption (*.enc*) scheme accepted by Genus and also gzipped except when the cynw_cm_float library is used. When the cynw_cm_float library is used a plaintext file must be specified. See Types of Libraries in the *User Guide.*

The `use_tech_lib` is used by Stratus HLS's datapath optimization features and allows Stratus HLS to produce parts on-the-fly if it needs parts that are not already included in your `hls_lib`.

Only one technology library is required, but multiple libraries may be specified using the use of multiple `use_tech_lib` commands. In this case, all of the libraries will be provided to Stratus HLS. If there is any duplication of technology data in the libraries, then the data from the last library specified in the project file will be used.

If you have several variations of the target technology library (that is, for worst case or best case conditions, for commercial or military operations), you should use the same *.lib* file throughout your project:

- In the `use_tech_lib` command
- When performing logic synthesis on the Verilog RTL produced by Stratus HLS.

This will help you optimize for timing closure across the behavioral synthesis and logic synthesis processes.

See also:

- Specifying ASIC technology libraries in the *User Guide*
- Technology library in the *User Guide*
- "tech_lib" in Synthesis Control Attributes
- use_hls_lib

Example 1

```
use_tech_lib "/opt/use_tech_lib/ABC_1.2_10ns/tech.lib" \
       -verilog_files /opt/use_tech_lib/ABC_1.2.v
```

In this example a verilog file containing simulation models for the technology library has been specified.

Example 2

In this example, both of the libraries will be provided to Stratus HLS. If there is any duplication of technology data in the libraries, then the data from the *tech.lib* will take precedence.

```
use_tech_lib "/opt/use_tech_lib/ABC_1.2_10ns/other.lib"
use_tech_lib "/opt/use_tech_lib/ABC_1.2_10ns/tech.lib"
```

Example 3

In this example, one of the techlibs shipped with Stratus HLS is used out of the Stratus installation.

```
use_tech_lib  "
[get_install_path]/share/stratus/techlibs/GPDK045/gsclib045_svt_v4.4/gsclib045/timing/s
low_vdd1v2_basicCells.lib"
```

use_verilog_simulator

A project file command.

Syntax

use_verilog_simulator `simname`

Parameters

simname
The name of the Makefile target that will execute when a Verilog simulator must be executed.

Description

A Verilog simulation will be run for each `sim_config` that includes a module in the `RTL_V` or `GATES_V` format. The name specified in the `use_verilog_simulator` command is the name of the `make` target that will be executed to cause that simulation to run.

There are several simulators for which built-in support is provided. The choices are:

- **incisive** or **ncverilog**: Cadence Incisive.
- **vcs**: Synopsys VCS simulator.
- **vcsi**: Synopsys VCSi simulator.
- **mti**: Model Technologies Verilog simulator.

Each of these commands invokes the simulator in a predefined way. This will compile all of the Verilog files produced by the Stratus HLS tools, all of the Verilog models stored in libraries, and Verilog modules you have specified using either the `verilog_files` attribute or the `define_hls_module` command. A shared library containing your SystemC simulation will automatically be loaded, and cosimulation will automatically be started.

You may also specify the name of a target in your own Makefile as an argument for this command. Often, the commands used to prepare and compile Verilog source files and to begin a simulation involve steps you will need to customize yourself. You can easily achieve this by adding a target in your own top-level Makefile and then specifying that target's name in the `use_verilog_simulator` command.

Makefile.prj sets a number of Makefile variables that provide context for custom simulator targets. You can use these variables in your own Makefile rules after *Makefile.prj* is included. The available variables are:

- **BDW_END_OF_SIM_CMD**: Contains a string that contains the command specified in *project.tcl* using `end_of_sim_command` attribute. You may wish to include this variable in a user-written simulator target as a final shell command after simulation has completed.

- **BDW_SIM_CONFIG**: Set to the name of the `sim_config` that is being executed.
- **BDW_SIM_CONFIG_DIR:** Set to the directory where `sim_config`-specific data is stored. This is ordinarily *bdw_work/sim_configs/<sim_config>*.
- **BDW_SIM_ENV_SETUP:** A string that contains shell commands for setting the variables that are guaranteed to be set during a simulation. This variable should be de-referenced before executing your own simulator command.
- **BDW_VLOG_DUT_FILES:** A string containing the paths to the Verilog files that need to be compiled for this simulation. Paths relative to the top-level Makefile may be given.
- **BDW_VLOG_LIBS**: A string containing `-y` arguments that reference all of the Verilog libraries used in this simulation.
- **BDW_VLOGCOMP_ARGS**: A string containing all of the parameters that must be passed to the compile step for the Verilog simulator.
- **BDW_VLOGSIM_ARGS:** A string containing all of the parameters that must be passed to the Verilog simulator at runtime. This includes options required for Hub cosimulation, along with any value specified earlier in your own Makefile for the same variable.
- **BDW_VLOGSIM_DEPS**: The targets on which the simulator target should depend.

The `use_verilog_simulator` command is required if any of your simulation configurations involve cosimulation.

See also, Defining a Verilog simulator in the *User Guide.*

Example 1

```
use_verilog_simulator incisive
```

Causes the built-in rules for executing NC-Verilog to be executed for each `sim_config` that has a module simulated as Verilog.

Project Attributes

A project attribute is identified by its name and contains either a single value or a set of values. Types include integer (which can be unsigned, positive, and so on), double, boolean, string, list of strings, and so on.
The project commands `get_attr` and `set_attr` can be used to retrieve, list, and set values for the project attributes.

The following table lists all the Stratus HLS project attributes.

Project Attribute Name	Description
`analysis_command`	The string given by the `analysis_command` attribute will be executed in an operating system shell each time a analysis config is processed. If the `analysis_command` is a script, it must either include the appropriate shell interpreter as part of the command (for example, `perl mydcscript.pl`) or the script file must be self-executing. The path may be relative to the main project directory or found on your Unix search path. Stratus HLS ships with a self-executing script named `bdw_runhal`. This script reads from the BDW API, runs HAL, and generates a code analysis report. This is the default if `set_attr analysis_command` is omitted. However, you may use the `bdw_runspyglass` script instead by either using `set_attr analysis_command bdw_runspytlass` or by using `'-command bdw_runspyglass'` on the `define_analysis_config` command (see define_analysis_config). There are no parameters passed to this attribute. Rather, information about the run that is to be performed is passed using Tcl variables. The string specified using the `analysis_command` attribute can also be overridden for individual `define_analysis_config` commands by using the `define_analysis_config`'s *`command`* option.

`cachelib_dir`	Specifies the directory to be used for the project’s cachelib. The `cachelib=on|off` option specifies whether a cachelib should be used at all, so if `-cachelib=off` is specified, `--cachelib_dir` has no affect.
	
	This option can be specified at the project level as a mechanism for sharing a cachelib between multiple projects, or it can be used on individual `hls_configs` so that different cachelibs are used for different configurations.
	
	Note that concurrent access to a cachelib from multiple projects is safe, but that the sharing mechanism is not high performance, so sharing should be limited to low-concurrency applications.
	
	For example:
	`  # Specify a directory for the cachelib that is also specified for other projects.`
	`   set_attr cachelib_dir  ../shared_cachelib`
	
	`   # Use a different cachelib for different configs.`
	`   define_hls_config dut  FAST --cp=3.5  --cachelib_dir=cachelib35`
	`   define_hls_config dut  FAST --cp=4.5  --cachelib_dir=cachelib45`
	
	Note that using different cachelibs for different configs should be done only to improve performance and capacity. It will not have any functional affect. Parts in the cachelib are automatically tagged with the clock period, which will cause separate parts to be built for each clock period. Using separate cachelibs simply reduces the size of each cachelib, and supports reuse of cachelibs in a more modular way.

`cc_options`	The string specified with the `cc_options` attribute will be added to the C++ compiler command line when compiling all `systemModules` and `define_hls_modules`. The Makefile generated by `bdw_makegen` will add all basic required options for C++ compilation. This attribute can be used to specify project-specific options. The `BDW_EXTRA_CCFLAGS` Makefile variable can also be used to add options to C++ compiler command lines. Also see the `hls_cc_options` attribute for passing compiler options to `stratus_hls`.
`clock_gating_module`	The Verilog for the module specified with the `clock_gating_module` attribute will be found in the library defined by the `use_hls_lib` command and will not be synthesized by Stratus HLS.
`end_of_analysis_command`	The given string will be included in the `Makefile.prj` file after each `define_analysis_config` execution. A typical use of this attribute is additional checking of results after each analysis config is executed. Only one `end_of_analysis_command` is supported per project file. You can use the following options with the attribute to acquire information about which `define_analysis_config` has just been executed: • **BDW_ANALYSIS_CONFIG** Set to the name of the `define_analysis_config` that has just been executed. • **BDW_ANALYSIS_CONFIG_DIR** Set to the directory where `define_analysis_config`-specific data is stored. This is ordinarily `bdw_work/analysis/`*`<analysisConfig>`*`.`

`end_of_equiv_command`	The given string will be included in the `Makefile.prj` file after each `define_equiv_config` execution. A typical use of this command is additional checking of results after each equivalence config is executed. Only one `endOfEquivCommand` is supported per project file. You can use the following options with the attribute to acquire information about which `define_equiv_config` has just been executed: • **BDW_EQUIV_CONFIG** Set to the name of the `define_equiv_config` that has just been executed. • **BDW_EQUIV_CONFIG_DIR** Set to the directory where `define_equiv_config`-specific data is stored. This is ordinarily `bdw_work/equiv/`*`<equivConfig>`* .

`end_of_hls_command`	The given string will be included in the `Makefile.prj` file after each execution of `stratus_hls`. A typical use of this command is the execution of a report generation script that scans the report produced by `stratus_hls`. Only one `end_of_hls_command` is supported per project file. You can use the following options with the attribute to acquire information about the module that has just been synthesized: • **BDW_CRTL_FILE** Set to the path of the source file that was just produced by stratus_hls. • **BDW_MODULE** Set to the name of the module that has just been synthesized. • **BDW_HLS_CONFIG** Set to the name of the `define_hls_config` whose settings have just been used to synthesize the current module. • **BDW_HLS_CONFIG_DIR** Set to the path of the directory containing results from the `stratus_hls` execution that just completed. The path will be `bdw_work/modules/`*`<module_name>`*`/`*`<hls_config>`*. • **BDW_HLSLIB_DIRS** Set to a string containing several space-separated strings, each of which is the path to a library directory used in the project. This includes all of the libraries referenced with `hls_lib`, and the library produced by `stratus_hls` containing parts it created on the fly. The paths are relative to the project directory.

`end_of_logic_synthesis_command`	The `end_of_logic_synthesis_command` specifies a command that will execute after logic synthesis. The given string will be included in the `Makefile.prj` file after each execution of your logic synthesis tool. A typical use of this command is the execution of a report generation script that scans the report produced by logic synthesis. Only one `end_of_logic_synthesis_command` is supported per project file. You can use the following environment variables with the attribute to acquire information about the module that has just undergone logic synthesis: • **BDW_LS_CONFIG** Set to the name of the `define_logic_synth_config` whose settings have just been used during logic synthesis on the current module. • **BDW_LS_CONFIG_DIR** Set to the path of the directory containing results from the logic synthesis execution that just completed. The path will be `bdw_work/ logicsynth/<`*`logicSynthConfig`*`>.` • **BDW_LS_CONFIG_LOGS** Set to a string containing several space-separated strings, each of which is the path to either a *.bdr* file or a raw log file from one logic synthesis execution. The paths are relative to the project directory.

`end_of_power_command`	The given string will be included in the `Makefile.prj` file after each power analysis. A typical use of this command is additional power analysis after the 3rd party power analysis tool has completed. Only one `end_of_power_command` is supported per project file. You can use the following options with the attribute to acquire information about which power analysis has just been executed: • **BDW_POWER_CONFIG** Set to the name of the `define_power_config` that has just been executed. • **BDW_POWER_CONFIG_DIR** Set to the directory where `define_power_config`-specific data is stored. This is ordinarily `bdw_work/power/`*`<power_config>`* .
`end_of_sim_command`	The given string will be included in the `Makefile.prj` file after each simulation. A typical use of this command is the execution of a results analysis script that determines whether the simulation succeeded or not. Only one `end_of_sim_command` is supported per project file. You can use the following options with the attribute to acquire information about which simulation has just been executed: • **BDW_SIM_CONFIG** Set to the name of the `define_sim_config` that has just been executed. • **BDW_SIM_CONFIG_DIR** Set to the directory where `define_sim_config`-specific data is stored. This is ordinarily `bdw_work/sims/`*`<sim_config>`* .

`equiv_command`	The string given by the `equiv_command` attribute will be executed in an operating system shell each time a single module is processed for a `equiv_command`. If the `equiv_command` is a script, it must either include the appropriate shell interpreter as part of the command (for example, `perl mydcscript.pl`) or the script file must be self-executing. The path may be relative to the main project directory or found on your Unix search path. Stratus HLS ships with a self-executing script named `bdw_runconformal`. This script reads from the BDW API, runs Conformal, and generates an equivalence checking report. This is the default if the `equiv_command` attribute is omitted. The string specified using the `equiv_command` attribute can also be overridden for individual `define_equiv_config` commands by using the `define_equiv_config`'s *command* option.
`hls_options`	The `hls_options` is a read-only attribute that gives all of the options that will be placed on the `stratus_hls` command line to control a synthesis run. Individual options from this string should set using a `set_attr` command. C-compiler options can be set as a string using `hls_cc_options` . *For the list of available options, see " stratus_hls Command Line Switches ".*

`hls_cc_options`	Sets the C-compiler options that will be passed to `stratus_hls` for synthesis. For example, "`-I../include -DOPT2`". Note that the `hls_cc_options` attribute specifies a similar set of attributes for SystemC compilation. Only `-I`, `-D`, and `-U` compiler options can be specified in `hls_cc_options`. Options that are specific to gcc, like `-g`, will cause errors if specified in `hls_cc_options`. Options for gcc should be specified in the `cc_options` attribute. Other `stratus_hls` options may also be specified in `hls_cc_options`, including synthesis control options. `set_attr hls_cc_options` is cumulative. The options in an `hls_cc_options` are parsed into the equivalent `set_attr` command and applied at the time the attribute is set. After setting `hls_cc_options`, the attributes it set can be read with `get_attr`.
`lib_workdir`	By default, compiled libraries are written to `bdw_work/libs` and are recompiled as that directory is cleaned out for new working files. To maintain a static location for compiled libraries and save time by omitting unnecessary recompilation, specify a directory with the `lib_workdir` attribute. The `lib_workdir` can be shared across multiple projects. Any number of `hls_libs` can be compiled into the same `lib_workdir`; it is not necessary to create a separate `lib_workdir` for each library. However, the various libraries should use unique names to avoid naming conflicts between different library data. The `lib_workdir` attribute will only affect `hls_lib's` that are outside of the project directory. For `hls_lib` commands that reference libraries that are in the project directory, compiled libraries will be written to `bdw_work/libs/<libname>`.

`logic_synthesis_command`	The string given by the `logic_synthesis_command` attribute will be executed in an operating system shell each time a single module is processed for a `logic_synthesis_config`. If the `command_string` is a script, it must either include the appropriate shell interpreter as part of the command (for example, `perl mydcscript.pl`) or the script file must be self-executing. The path may be relative to the main project directory or found on your Unix search path. Stratus HLS provides a number of default scripts accessible via the following commands: • **bdw_rungenus**. Cadence Genus. • **bdw_rundc**. Synopsys Design Compiler. There are no command line parameters passed to this command. Rather, information about the run that is to be performed is passed using Tcl variables. The string specified using the logic_synthesis_command attribute can also be overridden for individual define_logic_synthesis_config commands by using the `-command` option. This is the recommended method for specifying a command for FPGA logic synthesis. See also, Defining a logic synthesis tool in the *User Guide*.

`makefile`	This setting defines the name of the file that will be generated when `bdw_makegen` processes the *project.tcl* file. If this attribute is not set, the output of `bdw_makegen` will be placed in a file named `Makefile.prj`.
	The `-o` command line argument to `bdw_makegen` can also be used to control the name of the output Makefile.
	For example:
	`set_attr makefile myproj.mak`
	This will cause `bdw_makegen` to produce a file named *myproj.mak*. This file would then be included in your own top-level Makefile. This is equivalent to using the `-o` command line option as follows:
	`bdw_makegen project.tcl -o myproj.mak`

`omit_single_field_ports`	Stratus HLS's default rules for generating the names of ports on RTL modules can be affected using this `omit_single_field_ports` attribute. When a struct or class type is used in a port or a modular interface, a port is added for each field in the struct. If the struct has only a single field, the name of that field is included in the port name by default. If the value of the `omit_single_field_ports` attribute is *true*, the names of single fields to be omitted from port names instead.
`power_command`	This indicates the integration script to be used for Stratus HLS makefile targets that include a power analysis run. For example: `set_attr power_command commandString` Ansys PowerArtist will be used by default if `power_command` is omitted.
`start_of_analysis_command`	The given string, containing a shell command, will be included in the `Makefile.prj` file before each `define_analysis_config` execution. Only one `start_of_analysis_command` is supported per project file. You can use the following options with the attribute to acquire information about which `define_analysis_config` has just been executed: • **BDW_ANALYSIS_CONFIG** Set to the name of the define_analysis_config that has just been executed. • **BDW_ANALYSIS_CONFIG_DIR** Set to the directory where define_analysis_config -specific data is stored. This is ordinarily `bdw_work/analysis/`*`<analysisConfig>`*.

`start_of_equiv_command`	The given string, containing a shell command, will be included in the `Makefile.prj` file before each `define_equiv_config` execution. Only one `start_of_equiv_command` is supported per project file. You can use the following options with the command to acquire information about which `equivConfig` has just been executed: • **BDW_EQUIV_CONFIG** Set to the name of the `define_equiv_config` that has will be executed. • **BDW_EQUIV_CONFIG_DIR** Set to the directory where `define_equiv_config`-specific data is stored. This is ordinarily `bdw_work/equiv/`*`<equiv_config>`* .

`start_of_hls_command`	The given string, containing a shell command, will be included in the `Makefile.prj` file before each execution of `stratus_hls`. Only one `start_of_hls_command` is supported per project file. You can use the following options with the command to acquire information about the module that will be synthesized: • **BDW_CRTL_FILE** Set to the path of the source file that will be produced by `stratus_hls`. • **BDW_MODULE** Set to the name of the module that will be synthesized. • **BDW_HLS_CONFIG** Set to the name of the `hls_config` whose settings will be used to synthesize the current module. • **BDW_HLS_CONFIG_DIR** Set to the path of the directory containing results from the `stratus_hls` execution . The path will be *bdw_work/modules/<module_name>/<hls_config>*. • **BDW_HLSLIB_DIRS** Set to a string containing several space-separated strings, each of which is the path to a library directory used in the project. This includes all of the libraries referenced with `hls_lib`, and the library produced by `stratus_hls` containing parts it created on the fly. The paths are relative to the project directory.

`start_of_logic_synthesis_command`	The `start_of_logic_synthesis_command` attribute specifies a command that will execute before logic synthesis. The given string will be included in the *`Makefile.prj`* file before each execution of your logic synthesis tool, but after the generation of the `.info` file. This enables the script to use information in the `.info` file or modify it if desired prior to execution for the logic synthesis script. Only one `start_of_logic_synthesis_command` is supported per project file. You can use the following environment variables with the command to acquire information about the module that will be synthesized: • **BDW_LS_CONFIG** Set to the name of the `define_logic_synthesis_config` whose settings will be used during logic synthesis on the current module. • **BDW_LS_CONFIG_DIR** Set to the path of the directory that will contain results from the logic synthesis execution. The path will be `bdw_work/ logicsynth/`*`<logic_synthesis_config>`*`.` • **BDW_LS_CONFIG_LOGS** Set to a string containing several space-separated strings, each of which is the path to either a *.bdr* file or a raw log file from one logic synthesis execution. The paths are relative to the project directory. For example, `set_attr start_of_logic_synthesis_command "bdw_shell prep_vlog.tcl"` This will execute the TCL script named `prep_vlog.tcl` in the project directory before each logic synthesis execution.

`start_of_power_command`	The given string, containing a shell command, will be included in the `Makefile.prj` file before each power analysis. Only one `start_of_power_command` is supported per project file. You can use the following options with the command to acquire information about which power analysis will be executed: • **BDW_POWER_CONFIG** Set to the name of the `define_power_config` that will be executed. • **BDW_POWER_CONFIG_DIR** Set to the directory where `define_power_config`-specific data is stored. This is ordinarily `bdw_work/power/`*`<power_config>`* . For example, `set_attr start_of_power_command "prep_pnr.csh"` This will cause the shell script `prep_pnr.csh` to be run before every place and route execution.
`start_of_sim_command`	The given string will be included in the `Makefile.prj` file before each simulation, but after the generation of the `siminfo` file. Only one `start_of_sim_command` is supported per project file. You can use the following options with the command to acquire information about which simulation will be executed: • **BDW_SIM_CONFIG** Set to the name of the `define_sim_config` that will be executed. • **BDW_SIM_CONFIG_DIR** Set to the directory where `define_sim_config`-specific data is stored. This is ordinarily `bdw_work/sims/`*`<sim_config>`* .

`verilog_dialect`	The given string determines the dialect of Verilog to be used. The default value is "`1995`." You can use the following options with the attribute: • **1995**Stratus shall generate Verilog 1995 compliant code in all Verilog files and inform all integrated consumers of Stratus-generated Verilog to expect Verilog 1995. • **systemverilog** Stratus shall generate SystemVerilog compliant code in all Verilog files and inform all integrated consumers of Stratus-generated Verilog to expect SystemVerilog.

`verilog_files`	This command can be used to specify additional Verilog files you would like included in every simulation. The files in this list will be presented to the Verilog compiler ahead of the `-y` arguments for component libraries. This allows modules in these files to be used in favor of modules in either characterized parts libraries or memory wrapper libraries.
	
	The files specified with `verilog_files` will be passed to any command that includes a `-verilog_files` option: `define_hls_module`, `define_equiv_config`, and `define_sim_config`. If a Verilog file should be specific to one `define_hls_module`, `define_equiv_config`, and `define_sim_config`, add the Verilog file to that command's `verilog_files` *or* `vlogFiles` option instead.
	
	Multiple files may be set in the following ways:
	
	• With multiple filenames in one `verilog_files` attribute:
	`set_attr verilog_files "file1.v file2.v"`
	
	• With multiple separate `verilogFiles` commands:
	`set_attr verilog_files "mypart1.v"`
	`set_attr verilog_files "mypart2.v"`
	
	• Using a wildcard character to specify a directory and file extension (if the files are in the same directory and have the same file extension):
	`set_attr verilog_files "/libpath/*.v"`
	
	• As an individual file:
	`set_attr verilog_files "test.v"`
	that contains `` `include `` statements of other Verilog files:
	`` `include "/libpath1/file1.v" ``
	`` `include "/libpath2/file2.v" ``

`verilog_libs`	This attribute can be used to specify additional Verilog libraries you would like to include in every simulation. Each of the directories specified will be passed to the Verilog simulator with a `-y` parameter. The format of Verilog libraries is such that each module must be in a separate file, and the base name of the file must be the same as the name of the module it contains. For example, `set_attr verilog_libs "..mylib1"` This causes `-y ../mylib1` to be added to each Verilog compilation command.
`verilog_top_modules`	Specifies either a Verilog module name, or a list of Verilog module names that are top-level modules in Verilog simulations. The source files for these modules can be specified using either the `verilog_files` project attribute, or the `-verilog_files` option to the `define_hls_config` or `define_sim_config` commands.
`verilog_top_wrapper`	A Verilog simulation in the Stratus HLS environment has a top-level module that instantiates other modules, including the SystemC testbench and the synthesized Verilog DUT. By default, this top-level module is called `top`. This name may conflict wit user-written modules (for example, the DUT or system modules). In such cases, the name of the top-level Verilog module may be changed by using this command. The `module_name` parameter must define a legal Verilog module name that is unique in the simulation.
`wait_for_licenses`	The `wait_for_licenses` project attribute gives the current license waiting setting. The value will be `all`, `stratus`, or `off`. The attribute is read-only, and can be changed using the `enable_license_waiting` command.

`worklib`	The `worklib` atrribute overrides the default of `bdw_work` as the root directory for all files produced by `bdw_makegen` Makefiles. The `worklib` attribute is useful for cases where a non-local directory is desired for object files, simulation executables, and log files. This is common when simulations are to be run on multiple platforms and only a subset of the paths on the network are accessible to all computers. For example, `set_attr worklib /net/sims/projABC/bdw_work` This example causes all of the files produced using `Makefile.prj` to be stored under the `/net/sims/projABC/bdw_work` directory.

The files with `verilog_files` will be presented to the Verilog compiler ahead of the `-y` arguments for component libraries. This enables modules in these files to be used in favor of modules in either characterized parts libraries or memory wrapper libraries.

Synthesis Control Attributes

Following is an alphabetical list and brief description of the available synthesis control attributes. These attributes can be set either at the project level, or on individual `hls_configs`. These attribute affect how synthesis is run by `stratus_hls`. Settings made for an `hls_config` override those made for the project.
Synthesis control attribute are set on the project using the `set_attr` command in `project.tcl`. For example:

```
set_attr clock_period        7.5
set_attr default_input_delay 1.5
```

Synthesis control attributes are most easily set on `hls_configs` in the `define_hls_config` command using `--<attrib>=<value>` syntax. For example:

```
define_hls_config dut C1 --default_input_delay=1.2  --balance_expr=delay
```

However, the `set_attr` command can also be used to set synthesis control attributes on `hls_configs`. The `find` command is useful for this purpose because it enables the attribute to be set on several `hls_configs` with a single command. For example:

```
define_hls_config dut BASIC1
define_hls_config dut BASIC2
define_hls_config dut PIPE1
define_hls_config dut PIPE2

set_attr sched_aggressive_1 on  [find -hls_config PIPE*]
set_attr cycle_slack 1.2 [find -hls_config *]
```

This will set the `sched_aggressive_1` attribute to on `PIPE1` and `PIPE2`, and will set `cycle_slack` to 1.2 on all four `hls_configs`.

Attributes
`abort_level [FATAL|ERROR|WARNING|NOTE|HINT|INFO]`
`allow_pipeline_loop_expansion [on	off]`
`attrib_value attrib_spec value`
`balance_expr [off|width|delay]`
`cachelib [on|off]`

`cap_table_file cap_table_file file_name`
`clock_period <`*`float`*`>`
`comm_subexp_elim [on|off]`
`constrain_multiport_mem_distance [on|off]`
`cycle_slack <`*`float`*`>`
`default_input_delay <`*`float`*`>`
`default_preserve_io [true|false]`
`default_protocol[on|off]`
`default_stable_input_delay <`*`float`*`>`
`dont_ungroup_name <part_name>[,<`*`partname`*`>]`
`dont_ungroup_type (none,reg_bank,div,mod)`
`dpopt_auto  [off|array|op|expr|all]`
`dpopt_effort [low|normal|high]`
`dpopt_with_enable [on|off]`
`eco_baseline <`*`string`*`>`
`eco_sharing [off|on]`
`fail_level [FATAL|ERROR|WARNING|NOTE|HINT|INFO]`
`fixed_reset [on|off]`
`flatten_arrays [none|all]`
`fpga_dsp_latency <`*`integer`*`>`
`fpga_dsp_min_widths [integer,integer	integer]`
`fpga_part <`*`string`*`>`
`fpga_tool <`*`string`*`>`
`fpga_use_dsp [off|on]`

`global_state_encoding [binary|one_hot]`
`ignore_cells <`*`quoted_string`*`>`
`inline_partial_constants [on|off]`
`interconnect_mode [wireload|ple]`
`lef_library file_name [file_name]`
`lsb_trimming [on|off]`
`message_detail [0|1|2]`
`message_level <severity>:<message_id_number>(,<message_id_number>)*`
`message_suppress <message_id_number>(,<message_id_number>)*`
`method_processing [translate|synthesize|dpopt]`
`number_of_routing_layers int`
`output_style_fp_rtl_same_arch [on|off]`
`output_style_fsm_increment [on|off]`
`output_style_mem [array|assigns|bin_file|hex_file|flat]`
`output_style_merge_cases [on|off]`
`output_style_multi_cycle_parts [rtl|generic]`
`output_style_mux [impl_case|expl_bool|expl_case|expl_case_dx]`
`output_style_parts [rtl|generic]`
`output_style_reset_all [on|off|excluding_dpopt]`
`output_style_reset_all_async [no|yes]`
`output_style_reset_all_sync [no|yes]`
`output_style_separate_behaviors [on|off]`
`output_style_separate_ memories [on|off]`

`output_style_starc <`*`rule_specification`*`>`
`output_style_structure_only [on|off]`
`output_style_two_part_fsm [on|off]`
`output_style_ungroup_parts [on|off]`
`part_effort [zero|low|med|high]`
`path_delay_limit [<`*`integer`*`>|off]`
`path_delay_limit_unshare_regs [on|off]`
`port_conns [named|positional]`
`power [on|off]`
`power_fsm [off|on]`
`power_clock_gating [off|on]`
`power_memory [off|on]`
`prints [on|off]`
`qrc_tech_file` *`file_name`*
`register_fsm_mux_selects [on|off]`
`retimed_parts [on|off]`
`rtl_annotation [ off | all | decl | id | op | stack | state ]`
`scale_of_cap_per_unit_len` *`float`*
`scale_of_res_per_unit_len` *`float`*
`sched_aggressive_1 [off|on]`
`sched_aggressive_2 [off|on]`
`sched_analysis [off|on_failure|always]`
`sched_asap [off|on]`

`sched_effort [low|medium|high]`
`sharing_effort_parts [low|high]`
`sharing_efforts_regs [low|high]`
`shift_trimming [none|standard|aggressive]`
`simple_index_mapping [off|on]`
`summary_level [INFO|HINT|NOTE|WARNING|ERROR|FATAL]`
`switch_optimizer [off|on]`
`timing_aggression [<integer>|off]`
`uarch_tcl`
`undef_func [none|warn|error]`
`unroll_loops [off|on]`
`verilog_dialect [systemverilog|1995]`
`version or v`
`Version or V`
`wireload <wireload_model>`

abort_level

Syntax

`abort_level [FATAL|ERROR|WARNING|NOTE|HINT|INFO]`

Description

This attribute controls the conditions under which Stratus HLS terminates execution.

The default value is `FATAL`--by default, Stratus HLS terminates immediately when a `FATAL` error occurs.

The `abort_level` attribute allows you to lower the threshold condition that causes immediate termination. Termination occurs immediately when Stratus HLS encounters a message at least as severe as the `abort_level`.

The available values, in order of decreasing severity are: `FATAL`, `ERROR`, `WARNING`, `NOTE`, `HINT`, and `INFO`. The default is `FATAL`.

- **`FATAL`**: Indicates that Stratus HLS is unable to continue processing the design and must terminate execution immediately.
- **`ERROR`**: Indicates that Stratus HLS is unable to produce correct results in the face of the condition described.In most circumstances, the program continues processing until the end of the current phase of program execution, but then terminates without producing design output files. See fail_level.
- **`WARNING`**: Indicates that Stratus HLS has detected a condition in which the results may not be as you intended or a condition that is especially anomalous and must be brought to your attention. The output is expected to be functionally correct.
- **`NOTE`**: Provides information that may be of particular significance.
- **`HINT`**: Provides suggestions for controls settings or design changes you might want to try for getting a better result.
- **`INFO`**: Provides normal information about the application's progress, decisions that have been made, and results.

Example

```
set_attr abort_level ERROR
```

allow_pipeline_loop_expansion

Syntax

```
allow_pipeline_loop_expansion [ on | off ]
```

Description

The `allow_pipeline_loop_expansion` attribute allows pipelined loops to be expanded to include the state before the loop. The default value is on.

Stratus will never schedule any ops inside of a pipeline before the first state.
The `allow_pipeline_loop_expansion` attribute determines how Stratus achieves this. If the attribute is set to on, and this transformation is applied, there will be no operations inside of a pipeline before the first state. If the attribute is set to off, or the transformation cannot be applied, Stratus will first attempt to move all operations appearing before the first state. If any of these operations cannot be moved, Stratus will insert a state at the entrance of the loop. If the new state violates a protocol, a WARNING will be issued.

attrib_value

Syntax

```
attrib_value attrib_spec <value>
```

Description

The `attrib_value` attribute is used to specify attributes for library modules from managed libraries such as `cynw_cm_float`. The attribute can be used to enable or disable specific library modules, or to specify build parameters such as "latency" for specific library modules. The `attrib_value` attribute should not be confused with the `get_attr` and `set_attr` commands.

`attrib_spec`
Specifies one or more library modules, and has the following format:

```
  [<lib_spec>:][<mod_name>|<mod_name_wildcard>.]
   <attrib_name>
```

where `<lib_spec>` is one of:

```
  <lib_name> | <path>/<lib>.bdl |
   <path>/<libname>.tcl
```

`value`
The value the attribute should be given.

The *`attrib_spec`* in an `attrib_value` attribute specifies an attribute on either a single module, or several modules. The format for *`attrib_spec`* shown above has three segments:

- ***`<lib_spec>`***: Specifies the name of a `hlsLib`, or the path of a *.bdl* file. Optional. If omitted, can apply to all libraries referenced in the project.
- ***`<mod_name>|<mod_name_regexp>`***: Specifies either the name of a module, or a Tcl regular expression for a module. Optional. If specified, only attributes on matching modules are affected. If not specified, attributes on any module are affected.
- ***`<attrib_name>`***: The name of the attribute to be affected. The attributes that can be set are defined by the modules in the libraries.

The values specified in an `attrib_value` attribute interact with other attribute specifications as follows:

- Higher precedence than:
  - Default attribute value specs for the library module.
  - Values specified in a project file with the `set_part_options` command.
- Lower precedence than:

- The values specified on the stratusHLS command line for things like `techlib` and `clock_period` when modules are built on the fly.

Example

```
# Specify that latency 1 should be used for all floating
# point multipliers, and 0 for all floating point adders.
define_hls_config * LAT1 \
    -attrib_value="cynw_cm_float:*mul*.latency 1" \
    -attrib_value="cynw_cm_float:*add*.latency 0"

# Specify that the cynw_cm_float_unit from the cynw_cm_float
# library should be enabled.
define_hls_config * USE_UNIT \
    -attrib_value="cynw_cm_float_unit.enabled 1"
```

balance_expr

Syntax

`balance_expr [off|width|delay]`

Description

The `balance_expr` attribute controls the global setting for balancing expressions of commutative operations. While balancing expressions, Stratus HLS restructure the expression to maximize opportunities for parallel execution. By default, this setting is `delay`.

- **off**: Disables global expression balancing.
- **width**: Instructs Stratus HLS to minimize the width of the data operations in each expression. In order to create smaller parts, Stratus combines operands with the same bit width as well as operands with similar bit widths.
- **delay**: Instructs Stratus HLS to minimize the delay of each expression. Stratus HLS attempts to minimize the delay at which the final result is available by estimating the time at which inputs are read and the delays through various functional units. This is an approximation, because Stratus makes several assumptions about the nature of delays through various functional units.

For more information about this feature, see Balancing Expressions.

Example

```
set_attr balance_expr width
define_hls_module HDP hdp.cc [define_hls_config BASIC balance_expr=off]
```

In this example, the `BASIC hls_config` is run with expression balancing disabled and any other `hls_config`s is with expression balancing set to minimize the width of data operations.

cachelib

Syntax

```
cachelib [on|off]
```

Description

The `cachelib` attribute line argument enables or disables use of a project-level cache for parts created on-the-fly by Stratus HLS. By default, the cache library is enabled (`on`); however, can be disabled by specifying `cachlib=off`.

The cache library stores synthesis results for parts created either by Stratus HLS for datapath elements of your design, or for a CellMath library like `cynw_cm_float`. Use of the project cache library can dramatically speed up iterative use of Stratus HLS and `cynw_cm_float`.

If the cache library is enabled, when a part is synthesized, it is stored in the cache library. If an identical part is required in the future, the copy stored in the cache is used instead of synthesizing it again. The same cache library is shared by all `hls_modules` and `hls_configs` in the same project.

For a part from the cache to be used, the part must perform exactly the same function, and it must have been synthesized in exactly the same environment, including:

- Clock period and cycle slack, either specified directly, or indirectly using `--timing_aggression`.
- The `use_tech_lib` command
- The `ignore_cells option` attribute
- The `wireload` attribute
- timing constraints placed on the part such as latency, input delay, and output delay
- The version of CellMath Designer used to create the part

If a part is used from the cache, Stratus HLS prints a message describing its metrics.

cap_table_file

Syntax

```
cap_table_file file_name
```

Description

This attribute (optional) specifies the capacitance table file for technology mapping while using the PLE mode. You can specify only one file. This file can be encrypted. Part building uses the capacitance table to derive the capacitance unit per length, resistance unit per length, and area unit per length.

For example, `set_attr cap_table_file my_cap_table.cap`

clock_period

Syntax

`clock_period <`*`float`*`>`

Description

This attribute is used to specify a clock period in the same time units used by the project's `tech_lib` for the operating frequency of the architecture to be generated by Stratus HLS. A number of data type float is used to define the value. The default value is `10.000`

Stratus HLS uses this clock period characterize datapath parts, and to scheduled parts into clock cycles. Note that the `cycle_slack` and `timing_aggression` options affect how much of the `clock_period` is available for scheduling operations.

See also, Specifying clock period in the *User Guide.*

Example 1

```
set_attr clock_period 10.0
```

comm_subexp_elim

Syntax

`comm_subexp_elim [on|off]`

Description

This switch is used to turn on or off common sub-expression elimination. The default value is `on`.

When this is turned on, Stratus HLS examines the SystemC source code for sub-expressions that are used in multiple locations. It then saves the value of one instance of the sub-expression in a temporary variable and use that value in place of the other instances of the sub-expression. For example, without common sub-expression elimination, the following set of assignments:

```
x = a + b;
y = a + b + c;
```

```
z = (a + b) * d;
```

produces hardware that calculates `a + b` three separate times. With common sub-expression elimination turned `on`, Stratus HLS instead implements this code as if it had been written:

```
tmp = a + b;
x = tmp;
y = tmp + c;
z = tmp * d;
```

The net effect on the design is difficult to predict as it varies for different design configurations. It can cause area and delay to increase, decrease, or remain unchanged. You need to experiment with this feature to determine whether is beneficial for your design configuration.

See, Eliminating Common Subexpressions in the *User Guide.*

Example

```
set_attr comm_subexp_elim on
define_hls_config dut BASIC --comm_subexp_elim=off
```

In this example, the `BASIC hls_config` will be run with common subexpression elimination disabled and any other `hls_config`s will be run with it enabled.

constrain_multiport_mem_distance

Syntax

```
constrain_multiport_mem_distance [on|off]
```

Description

Determines whether or not a `HLS_CONSTRAIN_ARRAY_MAX_DISTANCE` directive is required to schedule accesses to a multiport memory in a pipelined loop. The default value is `on`.

When set to `off`, Stratus HLS will not error out if no `HLS_CONSTRAIN_ARRAY_MAX_DISTANCE` directive is specified for a multiport memory accessed within a pipeline. However, if a `HLS_CONSTRAIN_ARRAY_MAX_DISTANCE` directive is specified, Stratus HLS will honor the constraint even if `constrain_multiport_mem_distance` is set to `off`.

When set to `on`, Stratus HLS terminates with an error if no `HLS_CONSTRAIN_ARRAY_MAX_DISTANCE` directive is specified for a multi-port memory accessed within a pipeline.

For example,

```
set_attr constrain_multiport_mem_distance off
```

cycle_slack

Syntax

```
cycle_slack <float>
```

Description

This option specifies an amount of time in each clock period that is to be reserved for use by downstream tools.The default value is `0.000`.

This setting may be used to make the timing calculations performed by Stratus more conservative (that is, easier to get through your logic synthesis tool). This setting is closely related to your `clock_period` setting and must be smaller than your `clock_period`.

`cycle_slack` takes a floating point value as an argument, and Stratus effectively subtracts that value from the clock period, leaving the slack portion of each clock cycle unused from a timing perspective.

This setting has en equal effect on every phase of tool execution and every timing path created. As a result, in many cases using `cycle_slack` may cause unnecessary degradation in design area. For designs that are missing timing, it is recommended that the `timing_aggression` flag be used instead, as it enables a more sophisticated and effective means of controlling Stratus's planned interaction with downstream logic synthesis tools.

`cycle_slack` may be used in conjunction with `timing_aggression`. The initial implementation of `timing_aggression` implicitly sets `cycle_slack` to `0` when `timing_aggression` is `0` or `off`. The implicit value of `cycle_slack` increases linearly to 50% of the clock period as the `timing_aggression` setting is lowered to `-10`, causing Stratus to aim for half the specified clock period at the most conservative setting. The implicit value of `cycle_slack` decreases linearly to -10% of the clock period as the `timing_aggression` value is raised to `+10`, effectively lengthening the target clock period by 10% at the most aggressive setting. These values may change in future releases as Cadence works to improve the utility of the `timing_aggression` control.

Example

```
set_attr clock_period 10.0
set_attr cycle_slack 1.5
```

The `clock_period` setting defines the clock period to be 10ns. However, with the `cycle_slack` setting, only 8.5ns of each clock cycle is actually used by Stratus HLS when performing timing calculations. If Stratus HLS chains parts together such that the timing exceeds 8.5ns, then it automatically inserts a clock cycle into the schedule. This setting makes it much easier to meet timing with your downstream tools, but results in a less aggressive implementation of your design.

See also:

- Adding cycle slack in the *User Guide*

- Specifying clock period in the *User Guide*

default_input_delay

Syntax

`default_input_delay <`*`float`*`>`

Description

The `default_input_delay` attribute provides project-wide and `hls_config`-specific control over input delays. It can be overridden globally with the `HLS_SET_DEFAULT_INPUT_DELAY` directive or overridden for a specific input with the `HLS_SET_INPUT_DELAY` directive.

By default, this attribute is not set.

For details on specifying input delays, see:

- HLS_SET_DEFAULT_INPUT_DELAY
- HLS_SET_INPUT_DELAY
- Specifying input delays in the *User Guide*

Example

```
set_attr default_input_delay 1.5
```

default_preserve_io

Syntax

```
default_preserver_io [true|false]
```

Description

The `default_preserve_io` attribute, when set, has the same effect as setting `preserve_io_signals` on all behaviors. The default value is false.

default_protocol

Syntax

```
default_protocol [on|off]
```

Description

The `default_protocol` attribute when set to `on` defines protocol regions for all of your I/O accesses to maintain protocols at the input and output design interfaces. This means that scheduling does not add clock cycles and guarantee relationships between I/O and user-written `wait()`s. The default value is `off`.

By default, Stratus HLS assumes that all regions are free-scheduling regions. This means that scheduling will add clock cycles as needed to minimize area while satisfying the performance constraints. To specify an exception to this rule, you can use the HLS_DEFINE_PROTOCOL directive.

default_stable_input_delay

Syntax

`default_stable_input_delay <float>`

Description

The `default_stable_input_delay` attribute provides project-wide and `hls_config`-specific control over input delays for inputs that are read within stable regions. An input can be specified as stable using the HLS_ASSUME_STABLE directive, or the assume_stable Tcl command. If an input is specified as stable, and if no specific delay value has been specified for it using the HLS_SET_INPUT delay directive or set_input_delay Tcl command, then its delay can be specified by the default_stable_input attribute.

A separate delay is often specified for stable inputs in cases where the input is known to become stable before the region in which which it is read. It is typical to specify a delay value of 0.0 for stable inputs to allow Stratus maximum flexibility in avoiding regsiters on these inputs. However, care must be taken when constraining logic synthesis when shorter inputs delays are specified since it may result in false timing violations.

By default, this attribute is not set. In this case, the delay for stable inputs where no specific delay has been specified will be given by the `default_input_delay` attribute or the `HLS_SET_DEFAULT_INPUT_DELAY` directive.

For details on specifying input delays, see:

- HLS_SET_DEFAULT_INPUT_DELAY
- HLS_SET_INPUT_DELAY
- Specifying input delays in the *User Guide*

Example

```
set_attr default_stable_input_delay 0.0
```

dont_ungroup_name

Syntax

```
dont_ungroup_name <part_name>[,<partname>]
```

Description

This option modifies the behavior of part ungrouping such that parts with specific names are instantiated as sub-modules rather than being ungrouped. One or more part names can be specified in a comma-separated list. The part names are module names, not instance names, and can include wildcards. By default, this attribute is not set.

Reasons for not ungrouping specific parts include:

- Reduction in the size of the main RTL file by instantiating large parts, or parts of which there are many instances, as separate sub-modules.
- Problems processing a part as part of the main RTL body by a downstream tool.
- The desire to the generic style of RTL for a specific part as specified by `output_style_parts=generic`. Ungrouped parts never use the generic style.
- The desire to use gate-level parts in logic synthesis for a particular part.

The names of parts can be read either in the *Parts* tab in the *Analysis* pane in Stratus IDE, or in the allocation table at the end of `stratus_hls.log`.

Note that the hls_module name is pre-pended to the name of `HLS_DPOPT_REGION` parts, in addition a timiming variant of _N (N=1,2, …) is post-pended. For that reason, in order to prevent ungrouping of a `HLS_DPOPT_REGION` part, you must either include the hls_module name prefix, or use a wildcard. For example, if a part specified with `HLS_DPOPT_REGION`(`DPOPT_DEFAULT`, `"mypart"`, "") in a hls_module named dut, the part name will be `dut_mypart`, not `mypart`. To prevent ungrouping of these parts, you can specify either `dont_ungroup_name=dut_mypart`, or `dont_ungroup_name='*mypart'`.

Note that when wildcards are used, each name must be surrounded with quotes to prevent expansion in the shell before `stratus_hls` is executed.

Example 1
Prevent ungrouping of a `HLS_DPOPT_REGION` part named `dut_filter`.

```
set_attr dont_ungroup_name dut_filter_*
```

Example 1
Prevent ungrouping of a `HLS_DPOPT_REGION` whose name is specified as `"filter"`, but that may be instantiated in several hls_modules, and which may have several _alt versions

```
set_attr dont_ungroup_name *_filter_*
```

Example 2

Prevent ungrouping of any part with Mul in the name.

```
set_attr dont_ungroup_name *Mul*
```

dont_ungroup_type

Syntax

```
dont_ungroup_type (none,reg_bank,div,mod)
```

Description

This attribute modifies the behavior of part ungrouping such that certain categories of parts are instantiated as submodules rather than being ungrouped. The default value is `none`.

The attribute accepts one or more of the following words, specified in a comma-separated list:

- **`none`**: No categories of parts (apart from categories that are never ungrouped, like memories) are omitted from ungrouping.
- **`reg_bank`**: Register bank parts created with the `HLS_MAP_TO_REG_BANK` feature are instantiated as explicit modules. This can be useful for very large register banks since logic synthesis and other tool runtimes can be long if large register banks are placed directly in the main RTL body.
- **`div`**: Divider parts, or `HLS_DPOPT_REGION` parts containing division are instantiated as explicit modules. The style of the output will be controlled by `output_style_parts` or `output_style_multi_cycle_parts` depending on whether the divide is single or multi-cycle, respectively.
- **`mod`**: Similar to `div`, but for the modulus operator.

dpopt_auto

Syntax

```
dpopt_auto  [off|array|op|expr|all]
```

Description

The `dpopt_auto` attribute enables automatic DpOpt pattern matching and part creation.The default value is `off`.

This option allows you to specify datapath optimization on a project-wide or `hls_config`-specific basis, rather than in individual code blocks as is accomplished with the `HLS_DPOPT_REGION` directive. With `dpopt_auto`, DpOpt examines the design to determine which library parts and/or datapath components would be most beneficial. It creates new parts that are characterized for area and timing and stores them in a library. It will optionally use them within the design based on decisions regarding pattern matching, scheduling, and allocation.

The default value is `off`.

The `array`, `op` and `expr` options for `dpopt_auto` determine which types of parts are created. You can use them individually or together to generate different sets of custom parts.

- **`off`**:Disables automatic datapath optimization.
- **`array`**: Creates DpOpt parts for read/write logic associated with non-constant accesses to flattened arrays. This option reduces the complexity and number of operations generated for non-constant accesses to flattened arrays and therefore increases the capacity of Stratus HLS and reduces the run-time. This can result in an increase in area due to the limited number of optimizations that can be peformed on the read/write logic after DpOpt is applied. However, this cannot be applied to arrays of signals.
- **`op`**: Generates parts for single operations (e.g., +~ & *) that are not exactly matched by a part in the library. This is usually a non-square part, such as:

```
sc_uint<3> a;
sc_uint<5> b;
sc_uint<8> c;
c = a * b;
```

Like the `expr` setting, `op` causes constant inputs to the operation to be included within the suggested part. Unlike `expr`, `op` does not cause the bottom bits of an operation to be trimmed.

- **`expr`**: Generates parts for whole expressions based on patterns that appear frequently in the design. These parts combine certain operations to take advantage of constant folding operations at the gate level, carry save and balancing optimizations, and removal of paths to unused bits. Specifically:
  - Constants are merged with operators they feed in order to take advantage of constant folding operations at the gate level.
  - Arithmetic operations are merged with other arithmetic operations and comparators to take advantage of carry save and balancing optimizations.
  - All operations are merged with constant right shift and range operations in order to allow for the removal of paths to unused bits of the result.

- **`all`**: Enables `array`, `op` and `expr` options. You can also enable these options by specifying `dpopt_auto=array,op,expr` (with no space after the commas).

The use of `dpopt_auto` usually results in a shorter latency. Based on the design itself, the area may increase or decrease.

Using `inline_partial_constants` or `lsb_trimming` in conjunction with `dpopt_auto` may produce undesirable results. These flags can make more targets for the `dpopt_auto` flag, resulting in large area increases in some (primarily floating point) configurations that have undergone significant part sharing.

`dpopt_auto` can also be affected by the `dpopt_effort` setting.

See also, Specifying Automatic Pattern Matching and Part Creation in the *User Guide.*

Example

```
set_attr dpopt_auto all
```

dpopt_effort

Syntax

```
dpopt_effort [low|normal|high]
```

Description

When using the `cynw_cm_float` CynWare library, Stratus HLS will create several timing variants of each type of floating point operation used in your design.The default value is `normal`.

There are two things that can be varied for each implementation:

- The *total delay* of the part. Longer delays have smaller areas. Pipeline stages are added if the total delay is longer than a clock cycle.
- The *input delay* for pipelined implementations. Having several input and output delay scenarios available allows Stratus HLS to explore more options for chaining other logic at the input and output of floating point operations.

The number of timing variants built can be controlled using the `dpopt_effort` attribute.

- `low`: Only the fastest possible.The number of total delay, input delays, variants is 1.
- `normal`: The fastest possible, and a delay rounded up to the next full clock period. The number of total delay, input delays is 2. The number of variants is 2-4.
- `high`: Add 2 intermediate delays between fastest, and next full clock period. The number of total delay, input delays is 4 & 3, respectively. The number of variants is 4 to 12.

A variant may be synthesized for each combination of total delay and input delay. When total delay + input delay is less than the clock period, the part is asynchronous. No more than one asynchronous part is synthesized for a given total delay.

If attribute values are used to specify explicit timing attributes, they affect how many parts are built as follows:

- If a `latency` is specified, all total delays used will be consistent with the given latency.
- If an `input_delay` is specified, only that input delay will be used.
- If a `max_delay` is specified, only one variant will be synthesized.

See also, The cynw_cm_float Library in the *User Guide.*

dpopt_with_enable

Syntax

```
dpopt_with_enable [on|off]
```

Description

This attribute controls the implementation of pipelined DpOpt parts.The default value is `off`.

When this option is set to `on`, Stratus HLS will change the way DpOpt parts are created. Each pipelined DpOpt part will have a new input, called `enable`, which will be active only when the data inputs to the DpOpt parts are active. The enable input (like the data inputs) is set under control of the FSM, as determined by Stratus HLS's scheduling engine. The enable input is combined with the stall input and connected to the enable input of the registers inside the DpOpt part.

The enable input, combined with the insertion of clock gating by the logic synthesis tool, means that the registers in the DpOpt will change only when the part is actively working on valid inputs. This can result in significant power savings depending on how often the DpOpt part is enabled during the design's operation and the number of registers within it.

Please note the following information:

- The option applies to all pipelined DpOpt parts used in a design. This includes implicitly created parts like multipliers, as well as custom parts created with `HLS_DPOPT_REGION`.
- The default value of the option is `off` because not all designs will benefit from this option. The enable pin and internal shift register add to the area of the design. If pipelined DpOpt parts have a small number of registers, or they are always active (determined by the algorithm characteristics), the power savings may not be significant. Take these factors into account when using this option. If this attribute is not set, the value set for the `--power` attribute is implied for power optimization.

- The logic synthesis tool may or may not recognize and add clock gating to the registers in the DpOpt part. This may be dependent on specific options to the clock-gating tools and their characteristics. You may have to modify your logic synthesis flows to obtain the full benefit of this option.

This option requires a Stratus-XL license. See also, Reducing the Power Consumption of DpOpt Parts in the *User Guide.*

eco_baseline

Syntax

`eco_baseline <string>`

Description

Specify the name of the baseline configuration against which this is an incremental change. By default, this attribute is not set.

eco_sharing

Syntax

`eco_sharing [off|on]`

Description

Set this attribute to `on` if you want unmatched OPs to be mapped to existing FUs. The default value is `off`.

fail_level

Syntax

`fail_level [FATAL|ERROR|WARNING|NOTE|HINT|INFO]`

Description

This attribute controls the conditions under which Stratus HLS will fail to write its RTL output files. Normally, when a message of `fail_level` severity or greater is issued, Stratus HLS will continue to the end of the current major phase of execution and then terminate. Consequently, the default behavior is to fail to write RTL output files if a `FATAL` or `ERROR` message has been issued (It is possible that some RTL output files will be written before an `ERROR` occurs.)

The `fail_level` attribute enables you to alter the threshold of conditions that will cause a failure.

You can use this switch to lower the threshold to `WARNING` or raise it to `FATAL`, but be aware that

`ERROR` messages indicate a condition under which Stratus HLS does not believe it can produce a correct result. So, the files produced in the face of an `ERROR` message should be treated as an invalid design.

The available values, in order of decreasing severity, are: `FATAL`, `ERROR`, `WARNING`, `NOTE`, `HINT`, and `INFO`. The default is `ERROR`.

- **`FATAL`**: Indicates that Stratus HLS is unable to continue processing the design and must terminate execution immediately. See abort_level.
- **`ERROR`**: Indicates that Stratus HLS is unable to produce correct results in the face of the condition described. In most circumstances, the program continues processing until the end of the current phase of program execution, but then terminates without producing design output files.
- **`WARNING`**: Indicates that Stratus HLS has detected a condition in which the results may not be as you intended or a condition that is especially anomalous and must be brought to your attention. The output is expected to be functionally correct.
- **`NOTE`**: Provides information that may be of particular significance.
- **`HINT`**: Provides suggestions for control settings or design changes you might want to try for getting a better result.
- **`INFO`**: Provides normal information about the application's progress, decisions that have been made, and results.

Example

```
set_attr fail_level WARNING
```

fixed_reset

Syntax

```
fixed_reset
```

Description

This attribute controls the treatment of the reset behavior of internal variables. The default value is `off`.

If set to `on`, then all assignments of constants to variables that appear in the reset section of the behavioral specification will appear as reset behavior of some synthesized register in the RTL. If set to `off` (the default), then such assignments may be rescheduled to some other point in the RTL's execution sequence. (**Note**: Assignments representing resetting output ports are never rescheduled.)

flatten_arrays

Syntax

```
flatten_arrays [none|all]
```

Description

The `flatten_arrays` attribute provides project-wide and `hls_config`-specific control over which arrays are flattened into registers and which arrays are instead mapped to memories.The default value is `none`.

The following values are accepted:

- **`none`**: Instructs Stratus HLS to not flatten any arrays. All arrays are mapped to memories.
- **`all`**: Instructs Stratus HLS to flatten all arrays into registers.

This setting can be overridden on individual arrays using the `HLS_FLATTEN_ARRAY`, and `HLS_MAP_TO_MEMORY` directives.

Example

```
1: typedef sc_uint<8> array_t[16];
2: array_t arr1, arr2, arr3, arr4;
3:
4: arr1[0] = in1;
5: arr2[1] = arr1[0];
6: arr3[in1.read()] = arr2[in2.read()];
7: arr4[0] = arr3[in3.read()];
```

where:

- `arr1` has only constant references (lines 4 and 5).
- `arr2` has a constant index left-hand side and variable index right-hand side reference (lines 5 and 6 respectively).
- `arr3` has a variable index reference on the left-hand side (line 6) and the right hand side (line 7).
- `arr4` has only a constant reference (line 7).

The various settings will cause the following to occur (assuming no `HLS_FLATTEN_ARRAY` or `HLS_MAP_TO_MEMORY` directives are specified):

- `--flatten_arrays=none`

  All arrays are mapped to memories.

- `--flatten_arrays=all`

  `arr1` and `arr4` are flattened into registers; `arr2` is flattened into registers and a read mux; `arr3` is flattened into registers, a read mux, and write decoder.

See also:

- Flattening Arrays in the *User Guide.*
- HLS_FLATTEN_ARRAY
- HLS_MAP_TO_MEMORY

fpga_dsp_latency

Syntax

```
fpga_dsp_latency [integer]
```

Description

This attribute is used to specify the latency that is used for mapping DSP slices during resource characterization. The default setting is 2.

This attribute affects the multipliers to be **auto-mapped** DSP parts (if `fpga_use_dsp` is `on`). Also, this attribute affects the multipliers specified by HLS_DPOPT_REGION(DPOPT_FPGA_USE_DSP) that do not have HLS_CONSTRAIN_REGION irrespective of whether `fpga_use_dsp` is `on` or `off`.

fpga_dsp_min_widths

Syntax

```
fpga_dsp_min_widths [integer,integer | integer]
```

Description

This attribute is used to specify the operand width of the multiplier that can be mapped onto DSP slices during resource characterization. The default setting is 12,12.

It will affect the multipliers to be **auto-mapped** DSP parts (if `fpga_use_dsp` is `on`). It does NOT affect the multipliers specified by HLS_DPOPT_REGION(DPOPT_FPGA_USE_DSP). All multipliers with operand width greater than or equal to the specified width will be allowed to be mapped to the DSP. When only one bit-width is specified, the bit_width of both the operands of the multiplier must be greater than or equal to the specified width.

For example:

- if `--fpga_dsp_min_widths=12,16`: multipliers 16x12, 12x16 or 13x17 will automatically be mapped to DSP parts.
- if `--fpga_dsp_min_widths=16`: multipliers 16x16, 16x17, 17x18 will autpmatically be mapped to DSP parts.
- `--fpga_dsp_min_widths=16` is equvalent to `--fpga_dsp_min_widths=16,16`

fpga_part

Syntax

`fpga_part <`*`string`*`>`

Description

This attribute is used to specify the target part for an FPGA `hls_config`. No default is set.

fpga_tool

Syntax

`fpga_tool <`*`string`*`>`

Description

This attribute is used to specify the fpga tool, which is used for resource characterization. The fpga tool specified must be able to synthesize into the part specified by `fpga_part`. The accepted values are Vivado and Quartus.

No default is set.

fpga_use_dsp

Syntax

`fpga_use_dsp [off|on]`

Description

In the FPGA mode, when specifying `fpga_tool` and `fpga_part` attributes, the `fpga_use_dsp` attribute enables automated mapping of multipliers into the DSP slices of the specified FPGA part. This option allows you to specify DSP usage on a project-wide or hls_config-specific basis. When `fpga_use_dsp = on`, the multipliers that satisfy the contraints of `fpga_dsp_min_widths` will automatically be mapped into DSP slices during resource characterization.

The default value is `off`.

In ASIC mode, the attribute fpga_use_dsp will be used to isolate the multiplier with the flip flop before or both the flip flop before and after. For more details, see Using fpga_use_dsp in ASIC Mode in the *Stratus HLS User Guide*.

global_state_encoding

```
Syntax
global_state_encoding [binary|one_hot]
```

Desription
This attribute specifies which encoding method is used to represent the state of the controlling FSM in the synthesized RTL. By default, the value is `binary` and a binary encoding of the states is used. However, may be switched to a one-hot encoding with the `one_hot` setting. The binary encoding will require fewer flip flops in the RTL, while the one-hot encoding will use more flip flops but may improve the timing and layout characteristics of the gate-level implementation.

ignore_cells

Syntax
```
ignore_cells <quoted_string>
```

Description
By default, Stratus HLS automatically picks up cells in a technology library when creating parts on-the-fly during a Stratus HLS run. However, there may be times when you want to ignore one or more cells in the library.

The `ignore_cells` attribute instructs DpOpt to ignore cells from the technology library when creating parts during a Stratus HLS run. The string is a space-delimited list of cells. Wildcards are allowed; for example, specify "*L" to ignore cells that end in "L" such as HDINVDL, HDNOR2DL, and HDOAI21DL.

The list of cells must be quoted. Note that when quotation marks are needed in an already quoted string, they need to be escaped with the backslash character ().

It is possible to ignore too many cells, effectively causing an incomplete technology library. In this case, you will receive an error message stating that information is missing from the technology library.

See also, Libraries in the *User Guide.*

inline_partial_constants

Syntax

`inline_partial_constants [on|off]`

Description

This attribute instructs Stratus HLS to remove constant portions of certain registers and wires, causing signals to be broken up into pieces more frequently. If a subset of the bits of all the definitions of a reference are constant and equal, then the constant bits are inlined into the reference, causing a concatenation of constants and bit selects. `inline_partial_constants` can make more targets for the `dpopt_auto` flag, which can reduce sharing and cause area to increase. This is only an issue if the absence of `inline_partial_constants` would have created an opportunity for significant sharing of DpOpt functional units.

The default value is `off`.

interconnect_mode

Syntax

`interconnect_mode [wireload|ple]`

Description

This attribute determines whether to use the wireload information contained in the *.lib* or the PLE information contained in the *.lef/.cap/.qrc* files when synthesizing parts. The default value is `wireload`.

The `interconnect_mode` attribute must be set explicitly. Stratus HLS will not infer the value of `interconnect_mode` from the specification of a LEF library

For example, `set_attr interconnect_mode wireload`

For more information, see:

- Libraries in the *Stratus HLS User Guide*
- Achieving Timing Closure in Logic Synthesis in the *Stratus HLS User Guide*.

lef_library file_name

Syntax

`lef_library file_name [file_name]`

Description

The attribute specifies comma separated list of files containing LEF libraries for technology mapping. Stratus HLS will use the wire resistance, capacitance, area, and site size information in the LEF library. You must specify this attribute if you specify the `interconnect_mode` attribute value as `PLE`. By default, this attribute is not set.

When you read in LEF libraries, the cell area defined in the LEF libraries will be used instead of the cell area specified in the timing library (*.lib*). The timing library area will be used if the physical libraries do not contain any cell definitions.

> Cells that are found in the timing library but not in the LEF library will not be used in part building.

lsb_trimming

Syntax

`lsb_trimming [on|off]`

Description

This attribute instructs Stratus HLS to remove unused least significant bits from variables.The default value is `off`.

When this switch is `on`, if the least significant bits of a variable are never used, then they are trimmed and all references to the variable are shifted right. This results in a greater number of bit selects of arithmetic expressions.Combined with the `dpopt_auto` flag, LSB trimming has the side effect of causing more distinct functional units to be created by DpOpt, which can reduce sharing and cause area to increase. This is only an issue if the absence of LSB trimming would have created an opportunity for significant sharing of DpOpt functional unit.

message_detail

Syntax

`message_detail [0|1|2]`

Description

Stratus HLS automatically writes information from your behavioral synthesis run to the `stratus_hls.log` file under the `bdw_work/modules/`*`<hls_module>`*`/`*`<hls_config>`* directory. The default value is `0`.

The `message_detail` attribute causes Stratus HLS to print additional information to the `stratus_hls.log` file according to the following settings:

- **0**: No additional information is printed.
- **1 (or higher)**: The following information is added:

  Normalization complexity metrics: This prints the number of nodes present prior to each normalization phase. It provides a comparative gauge of the complexity of the design being processed with different design scenarios.

  For example, a 32-bit Linux machine with 2GB of memory may begin to slow down while processing a design with 2 million nodes and may have trouble at 5 million nodes. An extremely large scenario likely indicates that there is either too much unrolling in the design or the design itself is too large and would benefit from being partitioned.
- **2 (or higher)** The following information is added:

  Intrinsic mux details: This is a detailed breakdown of how each RHS symbol reference contributes to the multiplexing structures inherent in the design.

The total amount of estimated implicit mux area is automatically written to the log regardless of the `message_detail` setting. The intrinsic mux area is calculated based on the structure of the behavioral description (pre-allocation).

Note that the `message_detail` attribute controls information written to the `stratus_hls.log` file, not information available in the Stratus Integrated Design Environment (Stratus IDE). The Stratus IDE will automatically include the following:

- Estimated implicit mux area.
- Detailed mux information that is similar, but not identical, to that in the log file is available in the Detailed Hardware Reports by (Ops by Muxing).

See also, Customizing log data in the *User Guide.*

Example 1

```
set_attr message_detail 1
```

Example 2

```
void dut::thread0()
{
while( true ) {
    dataIn();
    for( i = 0; i < TAPS; i++ ) {
      data_out_reg += i * coefs[i];
    }
    dataOut();
  }
}
```

If `message_detail` were set to 2, the following mux information would be reported for iteration interval `i` and output `data_out_reg`:

```
Estimated intrinsic mux area: 531
at dut.cc line 18 estimated mux area: 425
             Symbol:data_out_reg Mux inputs: 2 Width: 16 Mux area: 425
at dut.cc line 50 estimated mux area: 106
             Symbol:i Mux inputs: 2 Width: 4 Mux area: 106
```

Example 3

The following example shows normalization data as written to the log:

```
00116: Optimizing thread saxo_light::thread0
00305: 763 nodes
00306: Optimize: pass 1...
00305: 557 nodes
00306: Optimize: pass 2.
00305: 556 nodes
00306: Optimize: pass 3...
00305: 557 nodes
00306: Optimize: pass 4.
00305: 553 nodes
```

message_level

Syntax

`message_level <severity>:<message_id_number>(,<message_id_number>)*`

Description

Users may customize the reported severity of specific messages using the `message_level` attribute. This affects program behavior by eliminating generation of RTL output if a message is promoted to ERROR, or by causing stratus_hls to terminate immediately if a message is promoted to FATAL.

`<severity>` must be one of FATAL, ERROR, WARNING, NOTE, HINT, or INFO.
`<message_id_number>` must be the 5-digit number of a posted message (leading zeros are optional). A warning is issued of the given number is not a valid message ID number. Multiple message IDs may be specified in a comma-separated list with no white space.

The severity of messages that are FATAL or ERROR by default cannot be reduced, but other messages may be demoted to a less severe status and any message may be promoted to a status more severe than it's default.

See also, `message_suppress` in the *Reference Guide* and Customizing log data in the *User Guide.*

Example 1

```
set_attr message_level ERROR:01433
```

Promotes message 01433 from a WARNING to an ERROR. By default, this message appears as:

```
WARNING 01433: at saxo_light.h line 31
WARNING 01433.   In thread "thread0" output signal "DOUT" is not initialized in
WARNING 01433.   the reset state. Consider using --output_style_reset_all=on.
```

but with this setting, this message would appear as:

```
ERROR 01433: at saxo_light.h line 31
ERROR 01433.   In thread "thread0" output signal "DOUT" is not initialized in
ERROR 01433.   the reset state. Consider using --output_style_reset_all=on.
```

and, since it is now an error, stratus_hls will halt before emitting RTL Verilog if this message is posted.

Example 2

set_attr message_level INFO:860

demotes message 00860 from a NOTE to a plain informational message. By default, this message appears as:

NOTE 00860: Long int data types are being implemented with 64 bits.

After this attribute is set, it appears as:

00860: Long int data types are being implemented with 64 bits.

message_suppress

Syntax

```
message_suppress <message_id_number>(,<message_id_number>)*
```

Description

Users may suppress reporting of specific messages using the `message_suppress` attribute.

<message_id_number> must be the 5-digit number of a posted message. Any leading zeros in the message id are optional. A warning is issued of the given number is not a valid message ID number. Multiple message IDs may be specified in a comma-separated list with no white space.

FATAL and ERROR messages may not be suppressed.

See also, message_level in the *Reference Guide* and Customizing log data in the *User Guide.*

Example 1

```
set_attr message_suppress 00860,3029
```

keeps Stratus_HLS from emitting NOTE messages 00860 and 03029.

method_processing

Syntax

```
method_processing [translate| synthesize| dpopt]
```

Description

The `method_processing` attribute provides global control over the treatment of `SC_METHOD`s during synthesis. The default value is `synthesize`.

The following values are accepted:

- **`translate`**: Instructs Stratus HLS to perform minimal processing on the body of the `SC_METHOD`, performing a fairly direct construct-by-construct translation from SystemC to Verilog.
- **`synthesize`**: Instructs Stratus HLS to synthesize the contents of the `SC_METHOD` along with the rest of the design. The SC_METHOD will be processed through all of the stages of synthesis. Stratus HLS's full set of optimization features may be applied including `HLS_UNROLL_LOOP`, `HLS_DPOPT_REGION`, and `HLS_SCHED_AGRESSIVE_1`. The functionality of the `SC_METHOD` will be implemented using functional units from the library, just like `SC_CTHREAD`s. Timing analysis will be performed to ensure that the `SC_METHOD` functionality can be implemented within a single clock cycle.
- **`dpopt`**: Instructs Stratus HLS to implement the contents of the `SC_METHOD` as a customized DpOpt part. Timing analysis will be performed to ensure that the `SC_METHOD` functionality can be implemented within a single clock cycle.

See also, Synthesizing SC_METHODs in the User Guide.

number_of_routing_layers

Syntax

`number_of_routing_layers` *`int`*

Description

This attribute (optional) limits the number of routing layers that will be used to calculate the area, capacitance, and resistance per unit length when synthesizing parts. The attribute has no default value, which indicates that all routing layers will be used.

For example, `set_attr number_of_routing_layers 9`

> This value must be greater than zero and no more than the maximum number of routing layers as specified by the technology libraries. Any other value will produce an error.

output_style_fp_rtl_same_arch

Syntax

```
output_style_fp_rtl_same_arch [on|off]
```

Description

The `output_style_fp_rtl_same_arch` attribute affects the form of the RTL Verilog models produced for `cynw_cm_float` parts. The parts synthesized for `cynw_cm_float` by CellMath Designer may use several different architectures. Each of these architectures is functionally identical, but may have different area, delay, and power characteristics.

The default value is `off`.

By default, the RTL Verilog models produced may use a different architecture than was used when synthesizing gates. Though the two descriptions are equivalent, this architecure difference can make formal verification more challenging.

By specifying `set_attr output_style_fp_rtl_same_arch on`, the RTL Verilog models for `cynw_cm_float` parts will use the same architecture that was used when synthesizing gates. This makes formal verification of the parts much easier.

This option requires a Stratus-FloatingPoint license.

output_style_fsm_increment

Syntax

```
output_style_fsm_increment [on|off]
```

Description

The `output_style_fsm_increment` attribute lists all states in the two-process (binary encoded) RTL FSM. Specifying `on` uses a default to increment the next state. Specifying `off` explicitly lists all the states in the FSM.

The default value is `on`.

output_style_mem

Syntax

```
output_style_mem [array|assigns|bin_file|hex_file|flat]
```

Description

This attribute determines the style in which memory parts are expressed in the RTL code. It applies to RAM and ROM models that are created by `Stratus_hls`—either by initializing an uninitialized ROM model found in the library or by creating a new RAM or ROM model from scratch.

The default is:

```
set attr output_style_mem=array [--output_style_mem=hex_file]
```

There are two fundamental ways in which memory parts are modeled:

1. by using an array inside the part [and, in the case of a ROM, initializing that array in the module's constructor in the RTL C model and in an initial block in the RTL Verilog model].
2. by using a case statement to decode the address and reference the appropriate scalar variable [value].

The option values select between these two fundamental models and, in the case of a ROM, the manor in which the storage array is initialized (where applicable).

- `array`: The memory's storage is modeled with an array. In the case of a ROM, the array is initialized with an `HLS_INITIALIZE_ROM` directive and a hexadecimal data file.
- `assigns`: The memory's storage is modeled with an array. In the case of a ROM, the array is initialized by assigning literal values to each individual array element.
- `bin_file`: The memory's storage is modeled with an array. In the case of a ROM, the array is initialized with an `HLS_INITIALIZE_ROM` directive and a binary data file.
- `hex_file`: The memory's storage is modeled with an array. In the case of a ROM, the array is initialized with an `HLS_INITIALIZE_ROM` directive that reads data from a file, interpreting the data as hexadecimal numbers.
- `flat`: The memory's storage is modeled with a set of scalars. The appropriate scalar is selected with a switch [case] statement applied to the address. In the case of a RAM, the appropriate scalar variable is read or written in the case arms. In the case of a ROM, the constant values are represented directly as literals in the case statement arms.

When either `bin_file` or `hex_file` is specified and the corresponding array in the behavioral model is initialized with a `HLS_INITIALIZE_ROM` directive using a suitably formatted data file, then the ROM model will be initialized with exactly the same file.

Placement Example

```
set_attr output_style_mem hex_file
```

Output Examples

The body of a small ROM model using the assigns style might look like this:

```
struct initmem_rom8x4_ar : public sc_module {
sc_in CLK;
sc_in OEN0;
sc_out<8> > DOUT0;
sc_in<2> > A0;
initmem_rom8x4_ar( sc_module_name name );
SC_HAS_PROCESS(initmem_rom8x4_ar);
const sc_uint<8> ar[4];
void thread1();
};
initmem_rom8x4_ar::initmem_rom8x4_ar(

sc_module_name name) : sc_module(name)
{
((sc_uint<8>*) ar)[0LL] = 1ULL;
((sc_uint<8>*) ar)[1LL] = 1ULL;
((sc_uint<8>*) ar)[2LL] = 2ULL;
((sc_uint<8>*) ar)[3LL] = 3ULL;
SC_METHOD(thread1);
sensitive_pos << ( CLK );
dont_initialize();
}

void initmem_rom8x4_ar::thread1(){
DOUT0 = (ar)[A0.read()];
return ;
}
```

while a model of the same ROM using the `hex_file` style would look like this:

```
struct initmem_rom8x4_ar : public sc_module {
  sc_in CLK;
  sc_in OEN0;
  sc_out<8> > DOUT0;
  sc_in<2> > A0;
  initmem_rom8x4_ar( sc_module_name name );
  SC_HAS_PROCESS(initmem_rom8x4_ar);
  const sc_uint<8> ar[4];
  void thread1();
};

initmem_rom8x4_ar::initmem_rom8x4_ar(sc_module_name name) : sc_module(name)
{
  HLS_INITIALIZE_ROM( sc_uint<8> , ar, ENUMS::CYN_HEX,
                "bdw_work/modules/initmem/MEMD_SIMP_HF/

initmem_rom8x4_ar.memh",
                "initialize initmem_rom8x4_ar" );
  SC_METHOD(thread1);
  sensitive_pos << ( CLK );
  dont_initialize();
}
void initmem_rom8x4_ar::thread1(){
  DOUT0 = (ar)[A0.read()];
  return ;
}
```

and, finally, a model of the same ROM using the flat style would look like this:

```
struct initmem_rom8x4_ar : public sc_module {
  sc_in CLK;
  sc_in OEN0;
  sc_out<8> > DOUT0;
  sc_in<2> > A0;
  initmem_rom8x4_ar( sc_module_name name );
  SC_HAS_PROCESS(initmem_rom8x4_ar);
  void thread1();
};

initmem_rom8x4_ar::initmem_rom8x4_ar(sc_module_name name) : sc_module(name)
{
  SC_METHOD(thread1);
  sensitive_pos << ( CLK );
  dont_initialize();
}

void initmem_rom8x4_ar::thread1(){
  switch( (sc_uint<2>)(A0.read()) ) {
  case 0ULL:
    DOUT0 = (sc_uint<8> ) (1ULL);
    break;
```

```
      case 1ULL:
        DOUT0 = (sc_uint<8> ) (1ULL);
        break;
      case 2ULL:
        DOUT0 = (sc_uint<8> ) (2ULL);
        break;
      default:
        DOUT0 = (sc_uint<8> ) (3ULL);
        break;
      }
    }
```

See also:

- HLS_INITIALIZE_ROM
- Specifying Output Settings for Your RTL Code in the *User Guide*

output_style_merge_cases

Syntax

```
output_style_merge_cases [on|off]
```

Description

This attribute determines whether certain data switching functions are represented as a single thread containing nested case [switch] statements in the RTL, or as a collection of separate threads, each containing a single, simple case [switch] statement.

The default value is `on`. Specifying the options as `on` invokes the single thread, nested case format. Specifying the option as `off` invokes the multi-thread, simple case statement format.

Examples

This example shows the use of the default `on` mode, where a mux is represented as nested and `case` statements:

```
always @(posedge clk )
begin: thread1
  case ( global_state )
     2'd2: begin
           if( cycle4_state ) begin
               if( cycle3_state ) begin
               end
               else begin
                      valid <= 1'd1;
```

```
              end
          end
          else begin
               if( cycle3_state ) begin
                    valid <= 1'd0;
               end
               else begin
                  valid <= 1'd1;
               end
          end
       end
   endcase
end
```

Compare this with the representation produced with the option set to `off`.

```
always @(cycle4_state or valid )
begin: thread0
      if( cycle4_state ) begin
           wire_valid <= valid;
      end
      else begin
            wire_valid <= 1'd0;
      end
end

always @(posedge clk )
begin: thread1
    case ( global_state )
        2'd2: begin
               if( cycle3_state ) begin
                  valid <= wire_valid;
               end
               else begin
                  valid <= 1'd1;
               end
        end
   endcase
end
        endcase
end
```

output_style_multi_cycle_parts

Syntax

```
output_style_multi_cycle_parts [rtl|generic]
```

Description

Determines the output style of all multi-cycle parts instantiated using the `RTL_V` representation in the Verilog output of Stratus HLS. Setting this option to `rtl` results in a simple, RTL representation for all explicitly instantiated parts. Setting this option to `generic` results in a register accurate representation of the part that consists of Boolean equations and register definitions that match the functionality of the `GATES_V` part.

The default value is `generic`. This option affects only multi-cycle parts. To affect all parts, use the `output_style_parts` option (see output_style_parts). However, note that controlling the output style of multi-cycle parts is the most common requirement, so `output_style_multi_cycle_parts` is the more commonly used attribute.

For more information on the difference between the `rtl` and `generic` output styles for parts, see Output style settings for parts in the *User Guide.*

output_style_mux

Syntax

```
output_style_mux [impl_case|expl_bool|expl_case|expl_case_dx]
```

Description

This attribute determines the style in which register multiplexors are expressed in the RTL code. The default value is `impl_case`.

There are four different forms of output:

- **`impl_case`**: Instructs Stratus HLS to describe the multiplexors as possibly nested case statements, with the decode/control logic expressed implicitly in the case structure, switch expressions, and activation values.
- **`expl_bool`**: Instructs Stratus HLS to describe the multiplexors as expressions of relational and boolean operations.
- **`expl_case`**: Instructs Stratus HLS to describe the multiplexors as single-level, onehot case statements switching on a value whose computation is explicitly expressed in expressions of relational and boolean operations.
- **`expl_case_dx`**: Similar to `expl_case`, with the addition of a default clause to the `case` statement

that assigns x's to the register when no multiplexor input is selected.

In many cases, the alternate forms of output result in shorter runtimes and reduced area through the logic synthesis tools. However, they usually result in longer simulation runtimes, as well.

The three `output_style_*` synthesis control attributes (`output_style_mux`, `output_style_separate_memories`, and `output_style_two_part_fsm`) may be used together and set independently.

See also, Specifying Output Settings for Your RTL Code in the *User Guide.*

Examples

This example shows the use of the default `impl_case` mode, where control is represented as a series of nested `if` and `case` statements:

```
always @(posedge(clk))
  begin :thread7
    case (global_state)

      3'd1: begin
         case (vld_in)

           1'b1: begin
              reg_1 <= in1[7:4];
           end

         endcase

       end

       3'd2: begin
          reg_1 <= reg_3[3:0];
       end

     endcase
   end
```

The three other forms (`expl_bool`, `expl_case`, and `expl_case_dx`) use temporary variables to re-encode the control signals into a single control signal for a one-hot mux:

```
t_39 = (sc_uint<1>) (global_state.read() == 1ULL);
t_40 = (sc_uint<1>) (global_state.read() == 3ULL);
t_44 = vld_in.read() & t_39;
t_43 = ~t_40 & ~t_39 | ~vld_in.read() & t_39;
t_42 = t_40;
t_41 = ( (sc_uint<3> )(sc_bv<3>)((sc_bv<1>)(t_44),
(sc_bv<2>)((sc_bv<1>)(t_43),
(sc_bv<1>)(t_42))) );
```

Using relational and boolean operations, these three forms vary the way in which the final one-hot mux is represented.

`expl_bool` represents the one-hot mux as a series of `AND ( & )` and `OR ( | )` operations:

```
reg_3 <= {{ 4 {t_44}}, t_44} & desc_Add_5U_0_1_out1 |
   {{ 4 {t_43}},t_43} & reg_3 |
   {{ 4 {t_42}}, t_42} & {1'b0, in1[15:12]};
```

Both `expl_case` and `expl_case_dx` represent the one-hot mux as a `switch` statement:

```
case (t_41)

    3'd4: begin
       reg_3 <= {1'b0, in1[15:12]};
    end

    3'd2: begin
       reg_3 <= reg_3;
    end

    3'd1: begin
       reg_3 <= desc_Add_5U_0_1_out1;
    end

    default: begin
       reg_3 <= 5'bxxxxx;
    end
endcase
```

Note that `expl_case_dx` also includes the final `default` clause to assist logic synthesis tools in recognizing the control expression as one-hot. (`expl_case` does not include the `default` clause shown above.)

output_style_parts

Syntax

```
output_style_parts [rtl|generic]
```

Description

Determines the output style of all parts explicitly instantiated using the `RTL_V` representation in the Verilog output of Stratus HLS. The default value is `rtl`.

Setting this option to `rtl` results in a simple, RTL representation for all explicitly instantiated parts. This includes both asynchronous and multi-cycle parts. Note that by default, asynchronous parts are ungrouped, and are so are not explicilty instantiated, which means the `output_style_parts` attribute does not affect them. However, if `output_style_ungroup_parts` is set to `off`, then `output_style_parts` will be applied to asynchronous parts because they will be explicitly instantiated.

Setting this option to `generic` results in a register accurate representation of the part that consists of Boolean equations and register definitions that match the functionality of the `GATES_V` part.

The default value of `rtl` is the most common setting for this option as the `generic` option is mainly of value only for multi-cycle parts. And, to control output style for only multi-cycle parts, the `output_style_multi_cycle_parts` option can be used (see `output_style_multi_cycle_parts`).

For more information on the difference between the `rtl` and `generic` output styles for parts, see Output style settings for parts in the *User Guide.*

output_style_reset_all

Syntax

`output_style_reset_all [on|off`|excluding_dpopt]

Description

The `output_style_reset_all` attribute implies the setting for both `--output_style_reset_all_async` and `--output_style_reset_all_sync`. The default value is `off`, and does not force a reset of all registers.

Setting the value to `no` implies a "no" value for those options and a `yes` or `excluding_dpopt` value implies a "yes" value for those options. When `excluding_dpopt` is used, flip flops internal to pipelined datapath components are not tied to the reset pin.

output_style_reset_all_async

Syntax

`output_style_reset_all`_async [on|off]

Description

The `output_style_reset_all_async` forces all registers not already otherwise reset to be asynchronously reset to zero by the primary writing process’s asynchronous reset signal. Registers that implement variables explicitly reset in the behavior are connected to all reset signals to which the resetting thread is sensitive. This is done regardless of this option's setting.

The default is `off` (deos not reset all flip flops).

If `output_style_reset_all_async` is not set explicitly, the value is determined by the option that you set for the `output_style_reset_all` attribute. See output_style_reset_all.

output_style_reset_all_sync

Syntax

```
output_style_reset_all_sync [on|off]
```

Description

The `output_style_reset_all_sync` attribute forces all registers not already otherwise reset to be synchronously reset to zero by the primary writing process's synchronous reset signal. Registers that implement variables explicitly reset in the behavior are still connected to all reset signals to which the resetting thread is sensitive. This is done regardless of this option's setting.

The default is `off` (does not reset all flip flops).

If `output_style_reset_all_sync` is not set explicitly, the value is determined by the option that you set for the `output_style_reset_all` attribute. See output_style_reset_all.

output_style_separate_behaviors

Syntax

```
output_style_separate_behaviors [on|off]
```

Description

When set to `on`, this option directs Stratus HLS to separate all explicit logic corresponding to SC_CTHREAD, SC_THREAD and SC_METHOD into respective separate RTL modules. The default value is `off`.

The name of separated RTL modules will be decided based on following syntax:

```
< hls_module_name >_< thread_name >_thread|method
```

output_style_separate_ memories

Syntax

```
output_style_separate_ memories [on|off]
```

Description

When set to `on`, this switch directs Stratus HLS to separate on-chip memories from other logic by placing everything except memory instances in a nested module and placing all memory instances directly in the top-level module. The default value is `off`.

If the `output_style_structure_only on` synthesis control attribute is also specified, then all memories will be instantiated in a submodule within the top-level module. A single submodule is used for all memories. The submodule will contain all only memories.

The three `output_style_*` synthesis control attributes (`output_style_mux`, `output_style_separate_memories`, and `output_style_two_part_fsm`) may be used together and set independently.

See also:

- Specifying Output Settings for Your RTL Code in the U*ser Guide*
- Clock Domain Crossing (CDC) in the U*ser Guide*

output_style_starc

Syntax

`output_style_starc  <`*`rule_specification`*`>`

Description

This attribute determines which of a set of selected STARC rules will be enforced by Stratus HLS. Most STARC rules are followed by default and compliance with most rules is not controllable.

Stratus HLS can comply with other STARC rules only at the expense of circuit quality (usually an increase in area). Users may choose to direct Stratus HLS to comply with these rules using the `output_style_starc` synthesis control attribute. A few STARC rules are complied with by default but may be disabled for reasons of backward compatibility.

Some STARC rules concern the structure of the RTL code (for example, the presence or absence of default branches in switch statements) while others concern the details of the Verilog syntax (for example, the use of blocking or non blocking assignment operators). Because of this variance in the nature of the STARC rules, compliance with some rules is implemented by `stratus_hls`. STARC rule identifiers are the STARC rule number with a preceeding 'S'.

The STARC rule identifiers S2.2.2.2, S2.2.3.1, S2.3.1.1, S2.3.1.3, S2.8.1.4, S2.10.3.1, S2.10.3.2 are controllable by the user. Of these, the following are `on` by default (but can be turned `off` using `--output_style_starc=-<rule>`)

S2.2.2.2

S2.2.3.1

S2.3.1.1

One may also use the special word `all` as a short hand for all rules implemented by the application.

A single `output_style_starc` synthesis control attributes can be used to specify one or more STARC rule identifiers. Multiple identifiers must be separated by commas (',') and have no embedded white space. If the rule identifier is prefixed with a minus ('- '), then compliance with that rule is not enforced. If the rule identifier is prefixed with a plus ('+') or has no prefix, then compliance with that rule is enforced. Multiple `output_style_starc` synthesis control attributes may be specified and, if so, their effect is cumulative.

Examples

Enable enforcement of rules 2.8.1.4 and 2.10.3.2 and disable enforcement of rule 2.10.3.2 for all configurations:

```
set_attr output_style_starc "+S2.8.1.4,S2.10.3.2,-S2.10.3.2"
```

Enable enforcement of all optional rules handled by stratus_hls in one configuration and disable enforcement of all such rules in another:

```
define_hls_configs dut C_STARC \
--output_style_starc=+all
```

```
define_hls_config dut C_NOSTARC \
--output_style_starc=-all
```

output_style_structure_only

Syntax

```
output_style_structure_only [on|off]
```

Description

If `output_style_structure_only on` is specified, then the top-level RTL file produced for a `hls_module` will contain only structural submodules. A structural submodule is either a hls_module, or a module created by `stratus_hls` as a structural submodule. Any other item, including Verilog assign or always statements, or instantiations of library modules, is moved to a structural submodule created by `stratus_hls`. The default value is `off`.

This attribute is particularly useuful for module that instantiate several hls_modules, and also instantiate a modular interface that infers hardware, like a MEM::shared_2 memory wrapper. If `output_style_structure_only off` is specified (or if no option is given), then the RTL produced will include always blocks, assign statements, an instantiation of a memory, and instantiations of the child hls_modules. However, if `output_style_structure_only on` is specified, the RTL will contain instantiations of the hls_modules, and an instantiation of one or more structural submodules created by stratus_hls. These structural submodules will contain the assign statements, always blocks, and memory instantiations that would have been in the top-level RTL module otherwise.

There are several cases in which stratus_hls will create more than one structural submodule. These are:

- If the `output_style_separate_memories on` option is specified, then all memories will be instantiated in a submodule separate from non-memory items.
- If the design uses multple clock signals, then the logic will be separated into submodules by clock domain as follows:
    - Logic that is entirely within one clock domain is placed in the same submodule.
    - Logic that crosses clock domains where inputs are written in clock domain A and outputs are written in clock domain B, are placed in the same submodule. These submodules will ordinarily be connected only to the clock used to store its outputs.

A clock domain crossing is inferred any time separate symbols are used as clocks in the logic that will be placed into submodules. The frequency of the clocks does not impact the identification of separate clock domains for the purpose of selecting a submodule.

See also, Which variables are set for use in logic synthesis scripts in the *User Guide.*

Examples

```
// Module containing 2 child hls_modules, and a MEM::wrapper.
// I/O from the top module.


SC_MODULE(top) {
  sc_in_clk clk;
  sc_in<bool> rst;
  cynw_p2p<DT>::in din;
  cynw_p2p<DT>::out dout;


  // ::shared_2 memory wrapper used by both modules.
  MT::shared_2<> mem;

  // Submodules, and channel between them.
  sub1_wrapper sub1;
  sub2_wrapper sub2;
  cynw_p2p<DT> chan;


  SC_CTOR(top) {
    sub1.din(din);
    sub1.dout(chan);
    sub2.clk(clk);
    sub2.rst(rst);
    sub2.mem(mem.if1);
    sub2.din(chan);
    sub2.dout(dout);
    sub2.clk(clk);
    sub2.rst(rst);
    sub2.mem(mem.if2);
    mem.clk_rst(clk,rst);
  }
};


// The resultant Verilog module
// with output_style_structure_only on
module dut(clk, rst, din_busy, din_vld, din_data,
           dout_busy, dout_vld, dout_data);


  input clk;
  input rst;
  input din_vld;
  input [7:0] din_data;
  input dout_busy;
  output din_busy;
  output dout_vld;
  output [7:0] dout_data;
  wire sub1_mem_WE0;
  wire sub1_mem_CE0;
  wire chan_busy;
  wire chan_vld;
  wire mem_if2_WE0;
  wire mem_if2_CE0;
  wire[7:0] sub1_mem_DIN0;
  wire[7:0] sub1_mem_A0;
  wire[7:0] chan_data;
  wire[7:0] mem_if2_DIN0;
  wire[7:0] mem_intsigs_DOUT0;
  wire[7:0] mem_if2_A0;
  reg sub1_clk;
  reg sub1_rst;
```

```
    reg[7:0] sub1_mem_DOUT0;


    sub sub2(
      .clk(clk), .rst(rst),
      .din_busy(chan_busy), .din_vld(chan_vld),
      .din_data(chan_data), .dout_busy(dout_busy),
      .dout_vld(dout_vld), .dout_data(dout_data),
      .mem_WE0(mem_if2_WE0), .mem_CE0(mem_if2_CE0),
      .mem_DIN0(mem_if2_DIN0), .mem_DOUT0(mem_intsigs_DOUT0),
      .mem_A0(mem_if2_A0));


  sub sub1(
    .clk(sub1_clk), .rst(sub1_rst),
    .din_busy(din_busy), .din_vld(din_vld),
    .din_data(din_data), .dout_busy(chan_busy),
    .dout_vld(chan_vld), .dout_data(chan_data),
    .mem_WE0(sub1_mem_WE0), .mem_CE0(sub1_mem_CE0),
    .mem_DIN0(sub1_mem_DIN0), .mem_DOUT0(sub1_mem_DOUT0),
    .mem_A0(sub1_mem_A0));


  dut_logic_1 dut_logic_1_1(
    .mem_if2_A0(mem_if2_A0), .mem_if2_DIN0(mem_if2_DIN0),
    .clk(clk), .mem_intsigs_DOUT0(mem_intsigs_DOUT0),
    .mem_if2_CE0(mem_if2_CE0), .mem_if2_WE0(mem_if2_WE0));
  endmodule
```

output_style_two_part_fsm

Syntax

`output_style_two_part_fsm [on|off]`

Description

For all cycle-accurate architectures scheduled by Stratus HLS, a finite state machine (FSM) is created to control what happens in each clock cycle. Finite state machines can be written in a number of ways and downstream logic synthesis tools can be very touchy in the way they handle them.

The default value is `on`. Setting this switch to `on` will produce RTL that is best for logic synthesis tools. Setting it to `off` will produce RTL that is more human-readable.

This attribute allows the user to specify whether the finite state machine is written using one or two processes. In single-process FSMs, the asynchronous next state logic and synchronous next state assignment appear in a single `SC_METHOD`. In two-process FSMs, the synchronous and asynchronous logic is separated into individual `SC_METHOD`s.

Most logic synthesis tools prefer finite state machines written with two processes. This gives the tool's synthesis policy checker fewer decisions to make about FSM implementation, allows it to concentrate more on optimizing other areas of the design, and usually runs faster.

The three `output_style_*` synthesis control attributes (`output_style_mux`, `output_style_separate_memories`, and `output_style_two_part_fsm`) may be used together and set independently.

output_style_ungroup_parts

Syntax

`output_style_ungroup_parts [on|off]`

Description

Ungroups datapath parts used to construct the design so that their descriptions are included directly in the main RTL body instead of being instantiated as submodules. This can improve results in logic synthesis when ungrouping is not performed there. Ungrouping parts may affect final estimated area since additional optimizations are performed after ungrouping. The default value is `on`.

Some kinds of parts are never ungrouped. These are:

- Memories
- `cynw_cm_float` parts
- Multi-cycle (pipelined) datapath parts
- Explicitly instantiated submodules

It is possible to prevent ungrouping of a subset of the parts used by the design using the following synthesis control attributes:

- `dont_ungroup_type` : Do not ungroup the specified class of parts.
- `dont_ungroup_name`: Do not ungroup parts with specific names.

See the following attributes documented in this table:

- `dont_ungroup_type`
- `dont_ungroup_name`

parts_effort

Syntax

`parts_effort [zero|low|med|high]`

Description

The `part_effort` attribute is used to control the effort applied by Stratus HLS for characterizing resources. The default value is `high`.

Stratus HLS can characterize the following types of resources:

- **0 (fastest)**: This is the default. Applies high effort and a constraint of zero to find the fastest (and lowest latency) possible implementation.
- **1 (fast)**: Applies low effort (and low run-time) with a constraint of zero attempting to find a fast implementation without very high run-time.
- **4 (slow)**: Applies low effort (and low run-time) with a relaxed constrain (clock period) to find a small implementation without very high run-time.
- **Z (zero)**: The delay and area are 0, not a real resource. The latency is set to the minimum allowed for the resource.

The following effort levels for `parts_effort` define which resources are built:

- **high** (default): build _4 and _1 resources, build _0 resource if scheduling fails.
- **medium**: build _4 and _1 resources, fail scheduling rather than building _0.
- **low**: build _4, do not build _1 or _0 even if scheduling fails.
- **zero**: build _Z, do not build any other resources.

path_delay_limit

Syntax

`path_delay_limit [<integer>|off]`

Description

Like all other CAD tools, Stratus HLS estimates the delay of signals traveling through the electrical circuits that underlie the design and, as estimates, the values Stratus HLS uses will not be a perfect match with reality.The default value is `off`.

Variations in the parameters given to the tools that follow Stratus HLS in the design flow, not to mention the choices of tools to use, are beyond Stratus HLS's control but can radically affect the accuracy of its estimates. For this reason, Stratus HLS has an input parameter to establish a variable upper bound of the allowable length of timing paths, true or false, in your design.

The `path_delay_limit` attribute allows you to specify a limit for the length of datapaths that can be used in RTL implementations. The integer value represents a percentage of the defined clock period.

This instructs Stratus HLS to limit datapaths to those that are smaller than or equal to `clock_period*<integer>/100`. For example, if the clock period is 10ns and the integer is set to 110, then Stratus HLS will only allow datapaths that are smaller than or equal to 11ns (that is, it will eliminate datapaths that are longer than 11ns). The length of datapaths is not limited to any specific value if this switch is set to `off`.

Logic synthesis tools are sometimes able to optimize long datapaths and produce a final result that is smaller than would be produced if Stratus HLS were more conservative in its design. On the other hand, long datapaths increase the risk that the logic synthesis tool will not be able to make timing closure. This switch allows you to control the balance between these two factors.

Cadence's research has shown that a value of 140 is usually an appropriate value for this flag. However it is recommended that instead of using this control, you instead use `timing_aggression`, which provides a simpler and more effective control for helping designs to meet timing.

`path_delay_limit` may be used in conjunction with `timing_aggression`. The initial implementation of `timing_aggression` implicitly sets `path_delay_limit` to 140 when `timing_aggression` is 0. The implicit value of `path_delay_limit` decreases linearly to 40 as the `timing_aggression` setting is lowered to -10, causing Stratus HLS to attempt to limit data path delays to 40% of the specified clock period. The implicit value of `path_delay_limit` increases linearly to 320 as the `timing_aggression` value is raised to +9, allowing Stratus HLS to emit data paths whose delay is up to 320% of the specified clock period. The implicit value of `path_delay_limit` is `off` when `timing_aggression` is set to +10 or "off". These values may change in future releases as Cadence works to improve the utility of the `timing_aggression` control.

path_delay_limit_unshare_regs

Syntax

```
path_delay_limit_unshare_regs [on|off]
```

This attribute is used in conjunction with the `path_delay_limit` attribute. If `path_delay_limit` is set to `off`, then the `path_delay_limit_unshare_regs` attribute will have no effect. The default value is `on`.

If the `path_delay_limit` attribute is set to an integer, then the `path_delay_limit_unshare_regs` adds extra effort to help Stratus HLS achieve the datapath delay limit.

Specifically, this means that Stratus HLS considers unsharing registers to help meet the datapath delay limit. Registers are timing start and end points; so, the false paths arising from a register share will not grow much beyond the effective clock cycle. This means that unsharing registers will not have the same potential to improve the datapath delay as the optimizations performed by the `path_delay_limit` attribute.

However, by unsharing registers the muxing in front of the registers can be reduced, which can help in meeting the datapath delay limit.

See path_delay_limit.

port_conns

Syntax

```
port_conns [named|positional]
```

Description

This attribute allows Stratus HLS users to generate RTL where all module instantiations express their port connections using named or positional notation. Depending on the downstream synthesis tool requirements, or just the overall readability of the generated code, this can be set accordingly.

Below are examples of the RTL C++ generated by Stratus HLS for each case. In this code excerpt is the instantiation of one of the chained parts built by Stratus HLS. With named port connections, the instantiation is expressed in separate lines for each port connection, explicitly naming the port of the chained part and the signal to which it connects.

```
struct modname;
modname::modname(sc_module_name name) : sc_module(name)
{
SC_HAS_PROCESS( modname);
modname_chain1_1 = new modname_chain1("modname_chain1_1");
modname_chain1_1->out2(modname_chain1_1_out2);
modname_chain1_1->out3(modname_chain1_1_out3);
modname_chain1_1->in1(modname_chain1_1_in1);
modname_chain1_1->in2(modname_chain1_1_in2);
...
}
```

However, with the port connections set to positional, the port connections are listed in a single line, simply listing the signal connections as arguments in the same order as the chained part's port declarations.

```
struct modname;
modname::modname(sc_module_name name) : sc_module(name)
{
```

```
SC_HAS_PROCESS(modname);
  modname_chain1_1 = new modname_chain1("modname_chain1_1");
  (*modname_chain1_1)(modname_chain1_1_out2,

modname_chain1_1_out3,
modname_chain1_1_in1, modname_chain1_1_in2 );
  ...
}
```

Note that the SystemC class libraries do not support positional port connections for modules with more than 64 ports. Because of this, Stratus HLS will always generate named port connections when it encounters such modules.

power

Syntax

`power [on|off]`

Description

This attribute turns on all the power reduction features including `--power_fsm`, `--power_clock_gating`, `--power_memory`, and `--dpopt_with_enable` as a group. The value set for `--power` is passed on to each of those options unless the other option is, itself explicitly set. The default value is `off`.

Stratus HLS includes power optimization capabilities as a separately licensed option. Specifically, Stratus HLS Low Power adds three capabilities to help generate lower power designs.

- Adding clock gating conditions to as many registers as possible throughout the design
- Reducing switching in the FSM's output logic
- Optimizing memory power by reducing the number of speculative read accesses.

Low Power report file

Details of the clock gating and FSM optimizations are available in the Stratus HLS Low Power report file. This file can be found in the hls_config's reports directory.
`bdw_work/modules/<module>/<hls_config>/reports/power.rpt`

This option requires a Stratus-XL license.

power_fsm

Syntax

`power_fsm [off|on]`

Description

This attribute applies specific transforms to the controller state machine encoding with the aim of reducing power consumption. The default value is `off`.

It also generates a power analysis report, which can be found in the hls_config's reports directory.
`bdw_work/modules/<module>/<hls_config>/reports/power.rpt`

If this attribute is set `on`, Stratus HLS enables state encoding power optimizations. If this attribute is set `off` the basic state encoding is used. If this attribute is not set, the value set for the `--power` attribute is implied for power optimization.

This option requires a Stratus-XL license.

power_clock_gating

Syntax

```
power_clock_gating [off|on]
```

Description

This attribute inserts logic for clock gating and other transforms to limit switching activity. The default value is `off`.

It also generates a power analysis report, which can be found in the hls_config's reports directory.
`bdw_work/modules/<module>/<hls_config>/reports/power.rpt`

If this attribute is set `on`, Stratus HLS allows insertion of special gating logic. This may increase area. If this attribute is set `off` special gating logic is not enabled. If this attribute is not set, the value set for the `--power` attribute is implied for power optimization.

This option requires a Stratus-XL license.

power_memory

Syntax

```
power_memory [off|on]
```

Description

Setting this attribute `on` inserts logic to control unnecessary switching on memory control line. This may increase area. The default value is `off`.

It also generates a power analysis report, which can be found in the hls_config's reports directory.
`bdw_work/modules/<module>/<hls_config>/reports/power.rpt`

If this attribute is set `off`, special memory control logic is not applied. If this attribute is not set, the value set for the `--power` attribute is implied for power optimization.

This option requires a Stratus-XL license.

prints

Syntax

```
prints [on|off]
```

Description

By default, Stratus HLS converts `printf()` statements in your behavioral code into `$write()` statements in your Verilog RTL code. This is useful for debugging RTL simulation results, but is not needed for logic synthesis and may actually hinder equivalence checking and processes for other downstream tools. The default value is `on`.

Stratus HLS's `prints` synthesis control attribute allows you to turn `printf()` conversion on and off. When this option is set to `off`, these statements are not emitted. Specifying `prints off` to stratus_hls tells it to omit `printf()` statements from the RTL C code.

Note that even when `prints` is `on` to stratus_hls, `cout` statements are never transferred into RTL code:

```
printf("output=%d\n",data); // OK
cout << "output=" << data << endl; // Not transferred into RTL code
```

See also Disabling printf() statement conversion in the *Stratus HLS User Guide.*

qrc_tech_file

Syntax

```
qrc_tech_file file_name
```

Description

This attribute (optional) specifies the QRC technology file from where the process and extraction information must be read. You can specify only one file. If you read in both a QRC technology file and a capacitance table file, an error will be produced. By default, this attribute is not set.

For example, `set_attr qrc_tech_file my_tech_file.qrc`.

register_fsm_mux_selects

Syntax

```
register_fsm_mux_selects [on|off]
```

Description

The `register_fsm_mux_selects` attribute optimizes the timing path for mux selects driven by the FSM by registering a pre-decoded select signal immediately directly in front of the mux select. If set to no, the mux selects will be driven directly from the FSM which can result in a savings in register area at the expense of timing closure. Often the logic synthesis tools can meet timing with --`register_fsm_mux_select off` and therefore save area in your design.

retimed_parts

Syntax

```
retimed_parts [on|off]
```

Description

This attribute determines whether or not retimed parts are used. The default setting is `on`, retimed parts are allowed.

The attribute controls whether retimed parts are enabled in the design. When set to `off`, this forces all parts characterized to have a latency of `0`. This may result in extra characterization steps to attempt to meet the constraint and if no characterization is possible for a resource then this may result in a failure. In the event of a failure, you will need to split large DPOPT parts into smaller sub-parts (potentially including changing `dpopt_auto` settings) or to relax the constraint. Individual `DPOPT regions` can be constrained using `HLS_CONSTRAIN_REGION` for a more targeted approach. See dpopt_auto and HLS_CONSTRAIN_REGION.

rtl_annotation

Syntax

```
rtl_annotation [ off | all | decl | id | op | stack | state ]
```

Description

This attribute provides the specified level of annotation in the RTL with links to the SystemC source code. The default setting is `off`, no added annotation. It accepts one or more values, separated by commas. The supported options are:

- off: No annotation
- all: All supported annotation
- decl: The port, signal and variable SystemC declarations associted with reg, wire and port Verilog declarations are listed
- id: Only an object identifier is included. No human-readable annotation is desired, but support

for later annotation with bdw_annotate_rtl is desired. It is recommended that you always set this option to get detailed annotations without having to re-synthesize your HLS designs.

- op: The source location of operations associated with resources and assignments are described.
- stack: For each item with a source location, the call stack is also shown.
- state: For FSM states registers, and for state-dependent assignments, information about the associated SystemC control flow is included.

All options are cumulative. For example, if all,op is specified, this is the same as all. Options can be removed using a - prefix. For example --rtl_annotation=all,-stack will include all options except stack.

For more details about annotations created in the RTL, see bdw_annotate_rtl.

scale_of_cap_per_unit_len

Syntax

```
scale_of_cap_per_unit_len float
```

Description

This attribute (optional) specifies the scale for the wire capacitance value. This attribute is used as multiplier to modify the capacitance values to resolve minor discrepancies between the default, detail, and sign-off extractors in the Encounter® place and route tool when using PLE mode. The default value is `1.0`.

For example, `set_attr scale_of_cap_per_unit_len 0.8`.

scale_of_res_per_unit_len

Syntax

```
scale_of_res_per_unit_len float
```

Description

This attribute (optional) specifies the scale for the wire resistance value. This attribute is used as multiplier to modify the resistance values to resolve minor discrepancies between the default, detail, and sign-off extractors in the Encounter® place and route tool when using PLE mode. The default value is `1.0`.

For example, `set_attr scale_of_res_per_unit_len 0.8`.

sched_aggressive_1

Syntax

`sched_aggressive_1 [off|on]`

Description

The default scheduling algorithm in Stratus HLS is intended for designs that are primarily datapath designs that have a minimal amount of control logic. For designs that contain a significant amount of control logic, the `sched_aggressive_1` attribute line option can be used to improve QoR. The default value is `off`.

The `sched_aggressive_1` attribute uses a more aggressive scheduling algorithm that results in:

1. Chaining across control flow boundaries.
2. Scheduling of non data dependent control flow statements in parallel.

This option also uses a more aggressive timing model that may result in a schedule with a reduced latency. However, there are a couple of limitations with this option:

- Stratus HLS is unable to determine that control flow branches are mutually exclusive and is unable to share resources that reside in different branches. For example, consider the following statement:

```
if( sel )
  X = A * B;
else
  X = C * D;
```

  When using `sched_aggressive_1`, Stratus HLS is unable to schedule the two multiply operations in a single multiply functional unit in a single cycle.

- Stratus HLS is unable to apply this scheduling algorithm to control flow statements that contain accesses to memories that are not flattened.

This option may result in RTL with reduced latency based on the amount of control flow in the design. This option may or may not result in increased Stratus HLS run times and a larger memory requirement during execution.

`sched_aggressive_1` may be overridden on individual blocks using the `HLS_REMOVE_CONTROL` directive.

The `stratus_hls.log` file will contain messages that indicate which conditionals to which `sched_aggressive_1` is or is not applied.

sched_aggressive_1=auto

This turns on sched_aggressive_1 on a conditional-by-conditional basis.

Note that `sched_aggressive_1` is applied during normalization while `sched_aggressive_2` is applied during scheduling, so any conditional that is transformed by `sched_aggressive_1` will not be available for transformation by `sched_aggressive_2`.

The following rules determine how the `sched_aggressive_1` transformation is applied when using `sched_aggressive_1=auto`.

Rule #1: `sched_aggressive_1` is applied to a conditional if the conditional is in a pipeline.

This implies that `sched_aggressive_1=auto` is almost equivalent to `sched_aggressive_1=on` for designs where a `HLS_PIPELINE_LOOP` directive is applied to a while(1) loop that encloses all of the code in the thread except for the reset code. The reset code outside of the main while(1) loop would use the following rules to determine if `sched_aggressive_1` is applied.

Rule #2: `sched_aggressive_1` is applied to a conditional if the conditional contains only a single assignment.

For instance the following would have `sched_aggressive_1` applied:

```
if (cond)
   x = a + b;
else
   x = a - b;
```

The following would not have `sched_aggressive_1` applied unless **Rule #1** or **Rule #3** apply:

```
if (cond)
    { x = a + b; y = b + c; }
else
    { x = a - b; y = b - c; }
```

Rule #3: `sched_aggressive_1` is applied if all the terms on the right-hand-side of the assignments in the conditional are constant.

For instance the following would have `sched_aggressive_1` applied:

```
if (cond)
    { x = 1; y = 2; }
else
    { x = 3; y = 4; }
```

and the following would not have `sched_aggressive_1` applied unless **Rule #1** applies:

```
if (cond)
    { x = 1; y = b + c; }
else
    { x = 3; y = 4; }
```

sched_aggressive_1=on

This turns `sched_aggressive_1` globally for the entire design.

See also:

- Scheduling aggressively in the *User Guide*
- HLS_REMOVE_CONTROL

sched_aggressive_2

Syntax

```
sched_aggressive_2 [off|on]
```

Description

The `sched_aggressive_2` attribute allows Stratus HLS be more aggressive at chaining operations across control flow boundaries. This can result in a schedule with reduced latency at the expense of a possible increase in area. Unlike `sched_aggressive_1`, this attribute cannot schedule control flow in parallel.

Since `sched_aggressive_1` cannot be applied to control flow statements that contain non-flattened memories, it can be advantageous to use both the `sched_aggressive_1` and `sched_aggressive_2` options simultaneously.

See also, Scheduling Aggressively in the *User Guide.*

sched_analysis

Syntax

```
sched_analysis [off|on_failure|always]
```

Description

The `sched_analysis` attribute is used to control generation of a scheduling report describing the multi-cycle circuit paths in the design. For each path in the report, a description is given of the control states on the path, the chain of operations in each control state with their individual delay values, and the reason each control state was added to the FSM.

With the default value, `on_failure`, this report is only generated when stratus_hls is unable to satisfy the timing constraints the user has imposed on the design. In this case, only the paths for which the constraints could not be satisfied are reported and the report is written to stdout as well as to the *stratus_hls.log* file.

When sched_analysis is set to `always`, a report is generated even if stratus_hls succeeds in scheduling the design. If `stratus_hls` is unable to satisfy the timing constraints for a particular `SC_CTHREAD`, the behavior is the same as when sched_analysis is set to `on_failure`. If `stratus_hls` is successful in scheduling a thread, a report covering *all* paths in the thread is generated, but written only to a file, `<output_dir>/scheduling_reports/<thread>.critical_paths.rep`, not to stdout or to the *stratus_hls.log* file.

A report is never generated when sched_analysis is set to `off`.

Report Format

The report file is organized in three sections:

1. A list of all operations in each state
2. A list of all the paths in the thread
3. A listing of the call stack information for each previously listed operation

The operations list in section 1 gives, for each operation,

- A unique identifier
- The location in the original source code from which the operation was derived (when applicable)
- The hardware resource to which the operation has been bound
- The delay allotted for the operation

The paths list in section 2 gives, for each path

- A cycle-by-cycle breakdown of the path, showing each operation on the path in the cycle in which it has been schedule. The operation descriptions are as in section 1 with the addition of a cumulative delay time for completion of the operation with that cycle.
- A cycle-by-cycle rational, giving a reason the cycle was created in the FSM

The call stacks listed in section 3 give the context by which this operation was reached. This is useful to disambiguate multiple paths that may include the same source code operator but represent instances of that operation from different calls to the containing subroutine.

sched_asap

Syntax

```
sched_asap [off|on]
```

Description

The `sched_asap` attribute is used to create RTL with minimum latency. When using this option, Stratus HLS will create the shortest possible schedule regardless of the number of functional units needed. The default value is `off`.

This attribute is most useful for quickly determining the minimum latency that can be achieved for a given set of constraints. This will almost always result in RTL with increased area but decreased latency.

Stratus HLS will create the shortest possible schedule while meeting all `HLS_CONSTRAIN_LATENCY` constraints specified for the block. Even with `sched_asap` enabled, Stratus HLS will never schedule fewer cycles than are specified as the minimum acceptable latency (*mincycles*) in the `HLS_CONSTRAIN_LATENCY` (*mincycles,maxcycles,"string"*) constraint.

See also:

- Scheduling as Fast as Possible in the *User Guide*
- Scheduling Aggressively in the *User Guide*

sched_effort

Syntax

```
sched_effort [low|medium|high]
```

Description

The `sched_effort` attribute is used to control the effort applied during scheduling stage of high level synthesis to minimize the area of the design. The default value is high.

The supported options are:

- high: minimize resource requirements and explore the schedule to minimize area.
- medium: minimize resource counts for resources considered sharable, operations where the muxing cost of performing a share would outweigh the area saving are ignored during this exploration.
- low: perform a fast scheduling pass where no attempt is made to minimize resources. This option also implies sched_asap=on unless the user explicitly overrides that setting.

Stratus will emit a note message reflecting how many operations in the thread will be considered sharable and would be subject to resource minimization in a sched_effort medium configuration.

sharing_effort_parts

Syntax

```
sharing_effort_parts [low|high]
```

Description

The `sharing_effort_parts` attribute is used to control the effort applied during allocation to minimize the area used by resources in the design using sharing.

The default value is `high`, which enables all resource sharing. Setting the option to `low` disables all resource sharing. See sharing_efforts_regs.

sharing_efforts_regs

Syntax

`sharing_efforts_regs [low|high]`

Description

The `sharing_effort_regs` attribute is used to control the effort applied during allocation to minimize the area used by registers in the design using sharing.

The default value is `high`, which enables all register sharing. Setting the option to `low` disables all register sharing. See sharing_effort_parts.

shift_trimming

Syntax

`shift_trimming [none|standard|aggressive]`

Description

This attribute sets the level of trimming that occurs on shift magnitude expressions (that is, RHS of shift operations). More aggressive trimming often results in lower area, but may lead to simulation mismatches with respect to a behavioral model compiled with gcc. This applies to right (>>) and left (<<) shift operators.The default value is `standard`.

There are three settings:

- **none**: Do not trim the shift magnitude expression. The shift magnitude expression is left at its native width.
- **standard**: Trimming for C/C++ and `sc_int`/`sc_uint` on the LHS will be performed so that the results of shifts in RTL code match the results of the same code at the behavioral level. This means that a negative RHS will have its bit representation treated as if it was an unsigned value of a width that allows the shifting of the entire LHS value. For instance, if the LHS is a long long and the shift value is -3 then the resulting shift will act as if the shift was 61 bits. So signed RHS instances will be sized so that they can shift the entire range of the target. For `sc_bigint` and `sc_biguint` values the RHS is assumed never to be negative in the trimming process as documented in the aggressive option

- **aggressive**: Trimming C/C++, `sc_int`/`sc_uint` and `sc_bigint`/`sc_bigunit` on the LHS is done assuming that the RHS will never be negative. If the size of the shift hardware generated for a design appears to be larger than it needs to be, using aggressive trimming may reduce it. However, if a RHS can be negative a simulation mismatch may occur.

simple_index_mapping

Syntax

```
simple_index_mapping [off|on]
```

Description

The `simple_index_mapping` attribute is used to instruct Stratus HLS to simplify the conversion of multi-dimensional indices into a memory address, replacing the multiplication and add operations with catenation. The default value is `off`.

This is equivalent to using the `SIMPLE` setting for the `HLS_MAP_ARRAY_INDEXES` directive, as described in Mapping Array Indices to Memory Addresses in the *User G*uide.

summary_level

Syntax

```
summary_level [INFO|HINT|NOTE|WARNING|ERROR|FATAL]
```

Description

Upon termination, Stratus HLS posts a summary of the messages produced during execution. This summary contains the number of instances of each message that exceeds a given threshold of severity. The default level is `WARNING`, meaning all `WARNING`s, `ERROR`s, and `FATAL` error messages are listed.

The `summary_level` attribute allows you to adjust the threshold to include less severe messages or exclude some of the more severe messages. For example, you can set the option to `HINT` to include all types of messages except `INFO`.

The message types in order of severity are: `INFO`, `HINT`, `NOTE`, `WARNING`, `ERROR`, and `FATAL`.

- **INFO**: Informs the user about normal processing of the design. This applies to all messages not labeled with one of the severity labels below. Note that the "INFO" label is not posted with these messages.
- **HINT**: Provides a suggestion for user action that may lead to a more satisfactory result.
- **NOTE**: Calls out a message that may be of special significance to the user because it relates

to the area or latency of the design or is otherwise a little beyond the ordinary.

- **WARNING**: Indicates that Stratus HLS has detected a condition in which the results may not be as the user intended or a condition that is especially anomalous and must be brought to the user's attention. The output is expected to be functionally correct.
- **ERROR**: Indicates that Stratus HLS is unable to produce correct results in the face of the condition described. It will continue processing until the end of the current phase of program execution, but will then terminate without producing design output files.
- **FATAL**: Indicates that Stratus HLS is unable to continue processing the design and must terminate execution immediately.

switch_optimizer

Syntax

`switch_optimizer [off|on]`

Description

During behavioral synthesis, Stratus HLS's switch optimizer performs a static analysis of the control paths in the design and simplifies the control structures where possible. It removes branches of switches that it determines to be unreachable and simplifies switch expressions. This optimization is controlled using the `switch_optimizer` attribute, which is `on` by default.

This optimization can improve Stratus HLS's ability to process designs containing complex control structures. For example, if the optimization removes the need to unroll a loop, the resulting design may be smaller due to the elimination of hardware that implements unneeded functionality. In addition, because the resulting design contains fewer operators and fewer control branches, tool runtime and memory footprint are reduced, which can dramatically improve scalability. These benefits will likely be more pronounced on designs with complex control structures.

There are three types of switch optimization: switch simplification, implicit switch constant propagation, and cascaded switch reduction.

- **Switch simplification**: Expressions in the `switch` conditional are analyzed to determine if constants can be folded into the case values themselves, or if the case value can be eliminated. The width of the resulting expression is also analyzed to determine if cases can be eliminated. This optimization can reduce QoR, latency, or both depending on other settings in your project.

  For example, consider the following:

  ```
      sc_uint<1> x;
  ```

```
switch( x + 3 ) {
case 0:
  a();
  break;
case 1:
  b();
  break;
case 2:
  c();
  break;
case 3:
  d();
  break;
case 4:
  e();
  break;
}
```

In this case, it is possible to know that `mode` can never be greater than 3 when called from this loop due to the bit width of the loop index. Thus, cases 5, 6, 7, and 8 can be eliminated. In addition, it is possible to observe that `mode+1` can never be equal to 0 because the loop index is unsigned. Thus, case 0 can be eliminated.

Stratus HLS can determine that default case can be eliminated, because the width of the resulting expression is 1. It also determines that case 0 and 1 can be eliminated, because they can never be executed. Stratus HLS changes case 3 and 4 to case 0 and 1, respectively, simplifying the `switch` statement to:

```
switch(x) {
case 0:
  d();
  break;
case 1:
  e();
  break;
}
```

- **Implicit switch constant propagation**: This optimization performs a static analysis of the control paths in the design and replaces variable references with constants in control paths where the value of the variable can be determined. Within the exclusive execution of a `switch` statement branch, the value of the `switch` expression is constant and equal to the case value of the switch. For example, consider the following:

```
void fn( sc_int<4> mode ) {
  switch( mode ) {
  . . .
  case 3 :
    x = mode + 5;
    . . .
    break;
  . . .
  }
}
```

Stratus HLS changes this to:

```
void fn( sc_int<4> mode ) {
  switch( mode ) {
  . . .
  case 3 :
    x = 8;
    . . .
    break;
  . . .
  }
}
```

- **Cascaded switch reduction**: When the target of each branch of a `switch` statement is also a `switch` statement, and conditional expression of each of those switches is the same, the switches are merged into a single switch, with a new conditional consisting of the concatenation of the two expressions. This simplifies control logic and can reduce the number of global states with comparable benefit in the FSM.

  Consider the following example:

```
sc_uint<1> M1;
sc_uint<1> M2;
   switch( M1 ) {
   case 0 :
     switch( M2 ) {
     case 0 :
       a();
       break;
     case 1 :
       b();
```

```
                  break;
                }
              case 1 :
                switch( M2 ) {
                case 0 :
                  c();
                  break;
              case 1 :
                d();
                break;
              }
```

Stratus HLS concatenates the expressions to create the following:

```
sc_uint<2> M = (M1,M2);
  switch( M ) {
  case 0 :
    a();
    break;
  case 1 :
    b();
    break;
  case 2 :
    c();
    break;
  case 3 :
    d();
    break;
  }
```

See also, Optimizing Switch Statements from Conditional Branch Expressions in the *User Guide.*

timing_aggression

Syntax

`timing_aggression [<integer>|off]`

Description

This attribute provides a simple means of controlling the balance between a design that is conservative from a timing point of view but may be somewhat larger than necessary versus one that may produce a smaller result but will require more work on the part of logic synthesis tools to meet timing. The default value is `off`.

The `timing_aggression` attribute may be set to a value from `-10.0` to `10.0` , inclusive, or `off` . Lower values cause Stratus HLS to produce a design that has more conservative timing characteristics—ones that may be larger but should meet timing through logic synthesis with relative ease. Higher values cause Stratus HLS to pack more logic into each clock cycle, possibly producing a smaller result but requiring logic synthesis tools to work harder to meet timing.

Note that setting `timing_aggression` to `off` is different from setting it to `0`. The former disables the internal timing optimization features associated with the `timing_aggression` feature, and the latter enables these features at a median level of optimization.

Cadence recommends that you begin by applying `timing_aggression=0` to your designs, as this will provide good overall management of path delay, false paths, and overall timing aggression. In the event that a design is experiencing difficulty meeting timing in logic synthesis, smaller (negative) values should be used. If a design is subject to stringent area requirements, you may wish to experiment with larger (positive) values for `timing_aggression`, up to the limits of your logic synthesis tool's capacity to meet timing.

The `timing_aggression` switch is a "big picture" control that affects a number of settings within Stratus HLS, some of which have their own specific controls. Those controls remain in place and, when specified together with `timing_aggression`, the more specific controls take precedence in that particular area. In some instances, you may wish to manipulate the more specific controls to reach points in the design space that are not otherwise achievable. These controls are:

- `cycle_slack`
- `path_delay_limit`

Stratus HLS will post a note-style message if `timing_aggression` is used in conjunction with one of these more specific controls.

See also, Achieving Timing Closure in Logic Synthesis in the *User Guide.*

uarch_tcl

Syntax

```
uarch_tcl
```

Description

The `uarch_tcl` attribute is an option to the `define_hls_config` command that defines a single Tcl command or a list of Tcl commands that are executed during the uarch control phase of synthesis for that module.

undef_func

Syntax

```
undef_func [none|warn|error]
```

Description

The `undef_func` attribute determines the action Stratus HLS will take if a function invocation that has no definition is detected in the input SystemC model. The default value is `warn`.

It can be set to one of the following

- **none**: Do not perform undefined function detection.
- **warn**: Emit a warning when an undefined function is detected.
- **error**: Emit an error when an undefined function is detected.

unroll_loops

Syntax

```
unroll_loops [off|on]
```

Description

The `unroll_loops` attribute is a global loop unrolling setting. The default value is `off`. When this option is enabled, Stratus HLS will automatically completely unroll every loop in the design that has a statically determinable number of iterations. This option affects only those loops that do not have a local `HLS_UNROLL_LOOP` setting. When both the `HLS_UNROLL_LOOP` directive and `unroll_loops` attribute are specified, `HLS_UNROLL_LOOP` takes precedence.

See also:

- Unrolling Loops in the *User Guide*
- "HLS_UNROLL_LOOP"

Example 1

```
set_attr unroll_loops on
```

Example 2

The following loop is always unrolled regardless of the `unroll_loops` setting:

```
for ( i = 0; i < 8; i++ ) {
   HLS_UNROLL_LOOP(ON,"loop1");
}
```

Example 3

The following loop is never unrolled regardless of the `unroll_loops` setting:

```
for ( i = 0; i < 8; i++ ) {
   HLS_UNROLL_LOOP(OFF,"loop2");
}
```

Example 4

The following loop is unrolled if `--unroll_loops` is on and not unrolled if `--unroll_loops` is off:

```
for ( i = 0; i < 8; i++ ) {
// No directive
}
```

Example 5

The following loop is never unrolled regardless of the `unroll_loops` setting, because it cannot be determine how many times it executes:

```
while ( in1.read() != 0 ) {
}
```

verilog_dialect

Syntax

`verilog_dialect [systemverilog|1995]`

Description

The default option for this attribute is `1995`. With this setting, Stratus emits Verilog 1995-compliant code in all Verilog files and inform all integrated consumers of Stratus-generated Verilog to expect Verilog 1995.

If the verilog_dialect project attribute is set to `systemverilog`, Stratus HLS emits SystemVerilog RTL. This includes:

1. Guarding against use of SystemVerilog keywords as identifiers, and
2. Use of the `@*` construct to specify the sensitivity of all combinational always blocks.

version or v

Syntax

`version or v`

Description

This attribute prints version information for the current version of Stratus HLS and exits. It does not run Stratus HLS.

Version or V

Syntax

`Version or V`

Description

This attribute prints version information and a build date, then checks for the availability of a license and exits. It does not run Stratus HLS.

wireload

Syntax

`wireload <`*`wireload_model`*`>`

Description

This attribute is used to specify the wireload model to be used by DpOpt when creating parts during a Stratus HLS run. For example, you may or may not want to force the use of the same wireload model for both library characterization and DpOpt parts created on the fly.

When no wireload model is specified, DpOpt will not use a wireload model.

You can specify the wireload model to use during logic synthesis using the `BDW_LS_WIRELOAD_MODEL` option for `define_logic_synthesis_options` in the project file.

Specifying a wireload model allows for the creation of RTL that will more accurately reflect real circuit characteristics and improve design convergence throughout the downstream tools and design flow.

The `wireload` attribute will be ignored if `interconnect_mode` is set to `ple`.

See also:

- Libraries in the *User Guide*
- Standard options for logic synthesis in the *User Guide*

`[ off | all | decl | id | op | stack | state ]`

Stratus will never schedule any ops inside of a pipeline before the first state.
The `allow_pipeline_loop_expansion` attribute determines how Stratus achieves this. If the attribute is set to on and this transformation is applied, there will be no operations inside of a pipeline before the first state. If the attribute is set to off, or the transformation cannot be applied, Stratus will first attempt to move all operations appearing before the first state. If this fails because some of these operations cannot be moved, Stratus will insert a state at the entrance of the loop. If the new state violates a protocol, a WARNING will be issued.

Synthesis Directives

Following is an alphabetical list of Stratus HLS directives.

Most of the Stratus HLS directive has an associated region command that takes precedence over the directive. For more information, see Synthesis Tcl Commands.

Directive	Reason
`HLS_ASSUME_STABLE`	Identifies an input (or group of inputs) that is stable within a block.
`HLS_BREAK_ARRAY_DEPENDENCY`	Allows you to break the inter-iteration array dependencies for a given array or loop.
`HLS_BREAK_PROTOCOL`	Identifies a specific location in the control flow where clock cycles may be added by scheduling.
`HLS_COALESCE_LOOP`	Specifies Stratus HLS to coalesce its enclosing loop.
`HLS_CONSTRAIN_ARRAY_MAX_DISTANCE`	Permits pipelined Stratus HLS designs to employ dual-port memories.
`HLS_CONSTRAIN_LATENCY`	Constrains a code block to be scheduled in specified number of clock cycles.

HLS_CONSTRAIN_REGION	• Specifies default delays on module inputs within a DpOpt part. • Specifies delays on module inputs within a DpOpt part. • Constrains a code block within a DpOpt part to be scheduled in specified number of clock cycles. • Constrains the stable time for module outputs within a DpOpt part.
HLS_DEFINE_FLOATING_PROTOCOL	Atomic-transaction directives that permit Stratus HLS to treat protocol as an atomic operation.
HLS_DEFINE_PROTOCOL	Specifies code blocks that should remain cycle-accurate.
HLS_DEFINE_STALL_LOOP	Causes a pipeline stall to be inferred for a loop.
HLS_DPOPT_REGION	Implements a code block as an optimized gate-level part.
HLS_EXPOSE_PORT	Controls whether a port in a `HLS_METAPORT` class will be placed on the RTL module.
HLS_FLATTEN_ARRAY	Controls array flattening on a local basis.
HLS_INITIALIZE_ROM	Controls the initialization of const arrays.
HLS_INLINE_MODULE	Identifies a module whose contents should be synthesized as part of the parent module that instantiates the module.
HLS_INVERT_DIMENSIONS	Allows you to use the `INVERT_DIMS` type of mapping to memory addresses.
HLS_MAP_ARRAY_INDEXES	Maps array indices to memory addresses.
HLS_MAP_TO_MEMORY	Maps an array to a memory.
HLS_MAP_TO_REG_BANK	Maps an array to a register bank.
HLS_METAPORT	Controls the recognition of a class as a metaport.

`HLS_NAME`	An alternative to using a C++ label before the curly-brace region or loop.
`HLS_PIPELINE_LOOP`	Controls loop pipelining.
`HLS_PRESERVE_IO_SIGNALS`	Prevents input and output signals from being optimized away because no other process uses them.
`HLS_PRESERVE_SIGNAL`	Preserves an `sc_signal<>` through the synthesis process.
`HLS_REMOVE_CONTROL`	Replaces control flow branching with data selection (multiplexors).
`HLS_SCHEDULE_REGION`	Extracts the behavioral code in the region and schedules it as a separate FSM.
`HLS_SEPARATE_ARRAY`	Separates a multi-dim array into several separate arrays.
`HLS_SET_CLOCK_PERIOD`	Specifies an alternate clock period to be used for a specific clock signal.
`HLS_SET_DEFAULT_INPUT_DELAY`	Specifies delays on module inputs.
`HLS_SET_DEFAULT_OUTPUT_DELAY`	Controls how Stratus HLS specifies the `default_output_delay` for all threads.
`HLS_SET_DEFAULT_OUTPUT_OPTIONS`	Specifies the kind of circuit Stratus HLS will build for an output.
`HLS_SET_INPUT_DELAY`	Specifies delays on named module inputs.
`HLS_SET_IS_BOUNDED`	Allow reads and writes of an input or output within boundaries.
`HLS_SET_OUTPUT_DELAY`	Controls the way Stratus HLS specifies the required delay for the output.
`HLS_SET_OUTPUT_OPTIONS`	Controls how Stratus HLS specifies the timing semantics for outputs.
`HLS_SET_STALL_VALUE`	Specifies a constant value that should be written to an output port or internal signal during a pipeline stall condition.

`HLS_SPLIT_ADD`	Specifies a constant value that is used to split adders/subtrators.
`HLS_SPLIT_MULTIPLY`	Specifies a constant value that is used to split multipliers.
`HLS_UNROLL_LOOP`	Controls loop unrolling on a local basis.

HLS_ASSUME_STABLE

A Stratus HLS directive.

Syntax

`HLS_ASSUME_STABLE``(` *`io`* `[,` *`_name`* `] );`

or

`HLS_ASSUME_STABLE``(` *`first_io, last_io`* `[,` *`_name`* `] );`

Equivalent Command

`assume_stable`

Parameters

`io`

The identifier for the input port or inter-process input.

`first_io`

The identifier for the input port or inter-process input that is the first to be affected, in declarative order

`last_io`

The identifier for the input port or inter-process input that is the last to be affected, in declarative order.

`name`

The name of the delay used for reporting purposes.

Recommended placement

Place immediately following the curly brace that encloses the affected block of code.

Description

The `HLS_ASSUME_STABLE` directive instructs Stratus HLS to keep the specified port access inside the block in which the `HLS_ASSUME_STABLE` is used. The *port* can be either an input port or an interprocess signal between threads.

This allows the input read to move anywhere within the block but not outside the block. This directive reduces the need for registering the port inside the block and may therefore reduce area.

`HLS_ASSUME_STABLE` is a block-based directive. When used at the top of a block, the directive allows an input read to be copied and moved anywhere within the containing block.

Reading an input within an HLS_ASSUME_STABLE block will usually prevent it from being immediately registered. However, Stratus will perform timing analysis to determine if a stable input needs to be registered based on its input delay, and the delay of downstream datapath. This analysis may show that the input needs to be registered before it is used. To prevent registering, you may specify an `HLS_SET_INPUT_DELAY` of 0 for the stable inputs. If you specify the `--message_detail=1` option to Stratus HLS, a report is printed describing why a stable input was registered. In addition, the `default_stable_input_delay` synthesis control attribute may be used to establish a default for all stable inputs.

Multiple regions in which it is safe to move input reads of a particular input can be expressed by using `HLS_ASSUME_STABLE` directives for the same input in multiple blocks.
When a design contains a large number of signals that must be marked with `HLS_ASSUME_STABLE`, it is convenient to use the form accepting two inputs. This form specifies the first and last of the input ports or inter-process inputs that should be treated as stable. Any input ports or interprocess inputs declared between the two that are named are also considered stable.

It is likely that your inputs are not stable before system reset. Therefore, your `HLS_ASSUME_STABLE` directive should be placed within the `SC_CTHREAD`'s main `while(1)` block, often within a more restrictive block within that `while(1)` loop, as appropriate.

See also:

- Specifying stable inputs in the *Stratus HLS User Guide*
- `default_stable_input_delay` in the *Stratus HLS Reference Guide*.

Example 1

To demonstrate the effect of `HLS_ASSUME_STABLE`, we'll show how input handling differs from inputs constrained with `HLS_DEFINE_PROTOCOL`. The inputs in the following example are read in a `HLS_DEFINE_PROTOCOL` block:

```
{HLS_DEFINE_PROTOCOL("read");
 wait();
 a_in = a.read();
 b_in = b.read();
}
rslt = (a_in * b_in) + a+in;
```

If it is not possible to compute both the multiplication and the addition in a single clock cycle, the design is scheduled like this:

```
Cycle 1:
    tmp = a.read() * b.read();
    a_reg = a.read();
Cycle 2:
    rslt = tmp + a_reg;
```

Because input `a` is read in a `HLS_DEFINE_PROTOCOL` block, it is assumed to not be valid in any other cycle. So, the value of `a` must be stored in a register until it is used in the subsequent cycle.

If you know that input `a` is held stable by an external register while the algorithm executes, you can use `HLS_ASSUME_STABLE` to eliminate this register. You can also avoid using intermediate variables to store input values by eliminating the separate `HLS_DEFINE_PROTOCOL` block. Consider the following version of the same algorithm:

```
{
   HLS_ASSUME_STABLE(a, "stable input a");
   HLS_ASSUME_STABLE(b, "stable input b");
   rslt = (a.read() * b.read()) + a.read();
}
```

With the same part timing, this can be scheduled as:

```
Cycle 1:
    tmp = a.read() * b.read();
Cycle 2:
    rslt = tmp + a.read();
```

Notice that `a_reg` has been eliminated, and that input a is being read when it is needed because the `HLS_ASSUME_STABLE` directive tells Stratus HLS that its safe to read inputs a and b at any time in the block.

Example 2

In the following example, each of `p1`, `p2`, and `p3` are considered stable within the block marked by `HLS_ASSUME_STABLE` since all three declarations fall within the `p1-p3` range:

```
// Declaration of module and ports.
SC_MODULE(dut) {
  sc_in<bool> p1;
  sc_in<bool> p2;
  sc_in<bool> p3;
  ...

// Code accessing p1 through p3 as stable inputs.
   {
     HLS_ASSUME_STABLE( p1, p3, "stable" );
     z = f(p1, p2, p3);
   }
```

HLS_BREAK_ARRAY_DEPENDENCY

A Stratus HLS directive.

Syntax

```
HLS_BREAK_ARRAY_DEPENDENCY( < array_id >);
```

Equivalent Command

break_array_dependency

Parameters

array_id

Identifies the array to break inter-iteration dependencies.

Recommended placement

Place immediately following the curly brace that encloses the affected block of code.

Description

The `HLS_BREAK_ARRAY_DEPENDENCY` directive allows you to break any dataflow connections between memory accesses within the specified block scope containing the directive. Break array directives specified in loop body breaks dependency across all iterations, if loop is unrolled; otherwise, it breaks dependency across one iteration. It also breaks any dataflow connections between memory accesses within the specified block scope containing the break array directive. Memory writes can occur simultaneously with any other access. Adjacent break array directives maintains initiation interval cycle distance between last operation in boundaries.

Example 1

```
while(){
  HLS_BREAK_ARRAY_DEPENDENCY(array);
  address = in.read();
  v1 = array[address];
  array[address+1] = v2;
}
```

In the above example, by default, without break array dependency, is scheduled in two cycles. And, with break array dependency, it can be scheduled in one cycle.

Example 2

```
while(){
    address = in.read();
```

```
        v1 = array[ address ] ; //Stmt1
        array[ address]  = v2;  //Stmt2
        for(...){
            HLS_BREAK_ARRAY_DEPENDENCY(array);
            array[ i ] = v3;         //Stmt3
            array [ i + 1] = v4;   //Stmt4
         }
    }
```

In the above example, `Stmt1` and `Stmt2` get scheduled one cycle apart as in the normal flow. `Stmt3` and `Stmt4` get scheduled in the same cycle due to the break dependency directive. Furthermore, `Stmt2` and `Stmt1` have a memory dataflow dependency (internal stratus dependency) to `Stmt3` and `Stmt4`. This dataflow ensures that `Stmt1` and `Stmt2` do not move around the inner `for()` loop or that `Stmt3` and `Stmt4` do not move outside their containing `for(...)` loop.

HLS_BREAK_PROTOCOL

A Stratus HLS directive.

Syntax

`HLS_BREAK_PROTOCOL` `(` *`min_lat`*`,` *`max_lat`* `[,`*`label`* `]);`

or

`HLS_BREAK_PROTOCOL``(` *`label`* `);`

Equivalent Command

break_protocol

Parameters

`min_lat`
The minimum acceptable latency for a block of code.

`max_lat`
The maximum acceptable latency for a block of code. The following are some special values that can be used for the *max_lat* value:

- **`HLS_INFINITE`**: Place no upper bound on the latency.
- **`HLS_ACHIEVABLE`**: Constrain the latency to the minimum achievable latency.

`label`
The name of the latency constraint used for reporting purposes.

Recommended placement

Place immediately following the curly brace that encloses the affected block of code.

Description

With `--default_protocol=true`, the user can specify exceptions to the every-region-is-a-protocol-region rule by using the `HLS_BREAK_PROTOCOL` directive. This identifies a point in the control flow where scheduling is allowed to add cycles. The `HLS_BREAK_PROTOCOL` directive takes parameters that determine how many cycles are allowed to be added by scheduling.

Note that the `HLS_BREAK_PROTOCOL` directive is not a region directive like `HLS_CONSTRAIN_LATENCY`. That is to say it does not apply to the entire code region in which it appears. Rather, it identifies a specific location in the control flow where clock cycles may be added by scheduling.

If the `min_lat` and `max_lat` values are omitted and a label is specified as in `HLS_BREAK_PROTOCOL("my latency")`, the latency achieved by Stratus HLS will be printed to the log file.

See also:

- Describing I/O Protocol in the *User Guide*

Example

```
void t()
    {
        // reset behavior
        in_rdy.write(false);
        out_vld.write(false);
        out_data.write(0);
        wait();

        while (1)
        {
            sc_uint<24> data;
            sc_uint<19> X=0;
            sc_uint<8> A;
            sc_uint<8> B;
            sc_uint<8> C;

            wait();
            in_rdy.write(true);
            do { wait(); } while( !in_vld.read() );
            data = in_data.read();
            A = data.range(23,16);
            B = data.range(15,8);
            C = data.range(7,0);
            in_rdy.write(false);

#if defined(BREAK_PROTOCOL)
            HLS_BREAK_PROTOCOL("computation");
#endif
            X = (A+B+C) * (A-B) + (B*C);
            do { wait(); } while (!out_rdy.read());
            out_data.write(X);
            out_vld.write(true);
            wait();
            out_vld.write(false);
            }

        }
```

In this example, by default, every region in the thread will be considered a free-scheduling region. This will have the effect of making all the port read()s and write()s produce warnings of the form:

```
WARNING 01165: at design.cpp line 62

WARNING 01165. Ignoring wait statement outside of a protocol block
```

and

```
WARNING 01177: at design.cpp line 34

WARNING 01177. Reference to I/O signal, in_rdy, is not in a protocol block.
```

If, however the design is run with `--default_protocol=true`, these warnings will not appear. You should never ignore warning 01165 and warning 01177. In this example, if the clock cycle is very short relative to the process technology, it may not be possible to perform the mathematical computation within a single cycle. In that case, scheduling will fail because there is only one `wait()` statement between the input data read and the output data write. Adding the `HLS_BREAK_PROTOCOL` directive will permit scheduling to add clock cycles at that point and achieve a valid schedule.

HLS_COALESCE_LOOP

A Stratus HLS directive.

Syntax

`HLS_COALESCE_LOOP` ***`( [ option ] );`***

Equivalent Command

coalesce_loops

Parameters

`option`

Determines how loop coalescing is affected by the directive. The default value is CONSERVATIVE. You can use any of the following options:

- **`ON`**: The loop will be coalesced with its parent loop.
- **`OFF`**: The loop will not be coalesced even if a parent loop specified `HLS_COALESCE_LOOP(ALL)`.
- **`CONSERVATIVE`**: The loop will be coalesced with any parent loop that is pipelined. Any intervening loops will also be coalesced.
- **`ALL`**: All child loops that are not explicitly unrolled will be coalesced with the loop.

Recommended placement

Place immediately following the curly brace that encloses the loop to be affected.

Description

The `HLS_COALESCE_LOOP` directive controls how Stratus HLS performs loop coalescing relative to its enclosing loop. Loop coalescing is an alternative to loop unrolling, primarily used for loops nested within pipelined loops. Loop coalescing allows some loops to be integrated with the enclosing pipelined loop instead of being unrolled. Unlike loop unrolling, the loop's contents are not duplicated.

Loop coalescing is useful for the following applications:

- When an inner loop contains many iterations, and would causes scalability problems if it was unrolled.
- When an inner loop has an indeterminate number of iterations, and cannot be unrolled.

Loops coalescing is appropriate for tightly nested loops near the top of a pipelined loop. More details can be found in the *User Guide.*

See also:

- HLS_UNROLL_LOOP

- Coalescing Loops in the *User Guide*
- Pipelining Loops in the *User Guide*

HLS_CONSTRAIN_ARRAY_MAX_DISTANCE

A Stratus HLS directive.

Syntax

`HLS_CONSTRAIN_ARRAY_MAX_DISTANCE``( `*`array_id`*` [ , `*`dist`*` ] [, _name_] );`

or

`HLS_CONSTRAIN_ARRAY_MAX_DISTANCE``( ext_mem_port1_id, ext_mem_port2_id_ , dist  [, _name_] );`

Equivalent Command

constrain_array_max_distance

Parameters

`array_id`

The name of the array that will be mapped to a dual-port memory or register bank in which the ports should be constrained.

`ext_mem_port1_id, ext_mem_port2_id`

The name of each port of the external dual-port memory that should be constrained.

`dist`

is the maximum distance constraint. If _dist_ is omitted, then it is set to -1 indicating that the distance can be arbitrarily large.

Recommended placement

Place immediately following the associated `HLS_PIPELINE_LOOP` directive.

Description

The `HLS_CONSTRAIN_ARRAY_MAX_DISTANCE` directive permits pipelined Stratus HLS designs to employ dual-port memories. It should appear after the `HLS_PIPELINE_LOOP` directive in the desired loop.

In order to guarantee a "safe" and functionally correct schedule in pipelined loops that perform both read and write operations on a dual-port memory, you must determine a maximum distance (*max_distance*) to guarantee that write operations and subsequent read operations are scheduled in the correct order.

The maximum distance is the maximum number of clock cycles that can safely be allowed to transpire between a write operation and a subsequent read operation within a single loop iteration. A number of factors contribute to the determination of the maximum distance, including:

- The address gap.
- The initiation interval of the pipelined loop.

The address gap is the difference in addresses between reads and writes within an iteration, defined as:

```
GAP = address(write) - address(read)
```

within a single iteration. The maximum distance is defined as:

```
max_distance = (II * GAP) - 1
```

where "II" is the initiation interval.
For instructions on how to determine the maximum distance, please see the application note titled "Using Dual-port Memories in Pipelined Loops." Once you determine a suitable maximum distance, add it to the `HLS_CONSTRAIN_ARRAY_MAX_DISTANCE` directive.

The maximum distance specified works in conjunction with the initiation interval in the `HLS_PIPELINE_LOOP` directive and the minimum address gap employed in a design. By remaining mindful of the effects these design parameters have on the maximum distance, you can develop your code so as to aid Stratus HLS in producing highly efficient pipelined designs.

In general, the more iterations that take place between a write and a subsequent read, the easier it will be to schedule the read (as more cycles will have intervened, allowing the write more time to complete). Hence, designs with "wide" minimum address gaps—that is, when the data being read comes from iterations that are relatively long ago—will have larger maximum distances. Larger maximum distances lend more scheduling flexibility to Stratus, and therefore will tend to produce better results.

Similarly, iteration intervals greater than 1 have a multiplicative effect on increasing the maximum distance. While increasing the interval figure can drastically increase the maximum distance, it is important to remain mindful of the impact that increased intervals will have on overall throughput.

Caution: If a pipelined loop with dual-port memory accesses is presented to Stratus without the necessary `HLS_CONSTRAIN_ARRAY_MAX_DISTANCE` directive, Stratus HLS will be unable to schedule the design, and will exit with an error. In this case, Stratus HLS will also indicate the smallest usable value for the maximum distance that would allow the design to be scheduled.

NOTE: you can prevent Stratus HLS from terminating with an error if no `HLS_CONSTRAIN_ARRAY_MAX_DISTANCE()` directive is specified by setting --constrain_multiport_mem_distance=no. For more details, see "constrain_multiport_mem_distance" in Synthesis Control Attributes.

It is very important to note that Stratus's suggested maximum distance *will NOT necessarily be the correct maximum distance* to use for a design. Stratus is merely indicating the smallest value for the maximum distance for which RTL could theoretically be produced. If this value were still too small for the actual design parameters (that is, the initiation interval and the differences in read/write addresses seen within loop iterations), the RTL may attempt to perform reads and writes in the wrong order.

Stratus HLS does not verify maximum distances for correctness. Until you have determined your own safe maximum distance (using the method described in the application note), Stratus HLS's suggested maximum distance cannot be assumed to be correct.

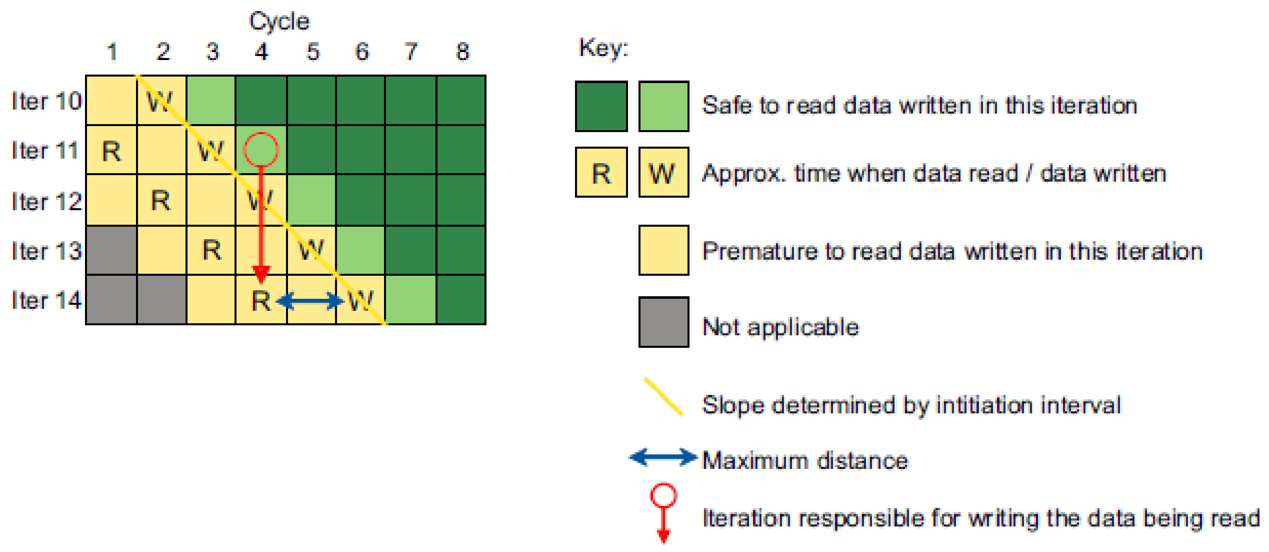
See also:

- HLS_PIPELINE_LOOP
- Using Dual-Port Memories in Pipelined Loops in the *User Guide*

Example 1

```
while (1) {
  HLS_PIPELINE_LOOP(HARD_STALL, 1, "pipe");
  HLS_CONSTRAIN_ARRAY_MAX_DISTANCE( mem, 2, "pipeline_constrain");
  y = mem[ i - 3 ];
  mem[ i ] = in1;
  i++;
}
```

This example employs an initiation interval of 1, maximum distance of 2, and gap of 3. The critical measure of safety in this and other pipelined designs is to ensure that the red circle (representing the data being read) lies in the green portion of the diagram (that is, above the yellow line that indicates that it is premature to try to read the data written by each iteration in those cycles.)

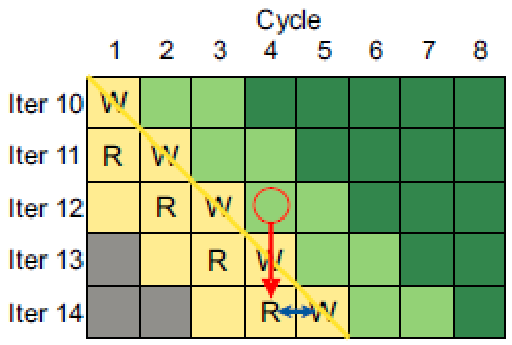

Cycle
1 2 3 4 5 6 7 8
Iter 10
Iter 11
Iter 12
Iter 13
Iter 14
R
W
Key:
Safe to read data written in this iteration
Approx. time when data read / data written
Premature to read data written in this iteration
Not applicable
Slope determined by intitiation interval
Maximum distance
Iteration responsible for writing the data being read

Example 2

```
while (1) {
  HLS_PIPELINE_LOOP(HARD_STALL, 1, "pipe");
  HLS_CONSTRAIN_ARRAY_MAX_DISTANCE( mem, 1, "pipeline_constrain");
```

```
    y = mem[ i - 2 ];
    mem[ i ] = in1;
    i++;
}
```

This example shows the effect of changing the maximum distance to 1 and the gap to 2.



HLS_CONSTRAIN_LATENCY

A Stratus HLS directive.

Syntax

`HLS_CONSTRAIN_LATENCY``( `*`min_lat, max_lat [,label ]);`*

or

`HLS_CONSTRAIN_LATENCY``( label );`

Equivalent Command

constrain_latency

Parameters

`min_lat`

The minimum acceptable latency for a block of code.

`max_lat`

The maximum acceptable latency for a block of code or `HLS_INFINITE` or `HLS_ACHIEVABLE`.

`label`

The name of the latency constraint used for reporting purposes.

Recommended placement

Place immediately following the curly brace that encloses the code block to be constrained.

Description

The `HLS_CONSTRAIN_LATENCY` directive is used to specify the minimum and maximum acceptable latency for a block of code. The block can be a loop body, an `if/else` or switch branch, or straight line code enclosed by braces ({ }).

The first two arguments must be integer literals (for example, 3, 16). In addition, there are some special values that can be used for the max_lat value. These are:

- **`HLS_INFINITE`** : Place no upper bound on the latency.
- **`HLS_ACHIEVABLE`**: Constrain the block to the minimum achievable latency.

If the *min_lat* and *max_lat* values are omitted and a label is specified as in `HLS_CONSTRAIN_LATENCY("my latency")`, the latency achieved by Stratus HLS for the block will be printed to the log file. Note that using either this limit-free form, or using `HLS_INFINITE` can affect the result achieved by stratus_hls because the presence of the latency constraint affects code motion rules.

Constraints may not overlap even if they do not conflict. Additionally, latency blocks may not contain a loop, unless the loop is completely unrolled. If a latency constraint is required for a block containing a loop (as might be the case with nested loops), then the block must be subdivided.

In the absence of a `HLS_CONSTRAIN_LATENCY` constraint, Stratus HLS will attempt to create the smallest design possible, as measured by area units in your library.

See also, Constraining latency in the *User Guide.*

Example 1

The following blocks of code are constrained to execute in exactly 10 cycles, no more than 10 cycles, and no less than 10 cycles, respectively.

```
{
   HLS_CONSTRAIN_LATENCY(10,10,"lat1"); // exactly 10 cycles
   ...
}
{
   HLS_CONSTRAIN_LATENCY(0,10,"lat2"); // no more than 10 cycles
   ...
}
{
   HLS_CONSTRAIN_LATENCY(10,HLS_INFINITE,"lat3"); // no less than 10 cycles
   ...
}
```

Example 2

Following are examples of illegal latency constraint use:

```
// ILLEGAL USE OF LATENCY CONSTRAINTS
{
   ...
   HLS_CONSTRAIN_LATENCY(3,3,"lat1"); // ILLEGAL - not first statement
   ...
}
// ILLEGAL USE OF LATENCY CONSTRAINTS
{
   HLS_CONSTRAIN_LATENCY(0,8,"lat2"); // ILLEGAL - contains inner block
   ...
   {
      HLS_CONSTRAIN_LATENCY(0,3,"lat3"); // ILLEGAL - constraints overlap
      ...
   }
}
// ILLEGAL USE OF LATENCY CONSTRAINTS
{
   HLS_CONSTRAIN_LATENCY(0,10,"lat4"); // ILLEGAL - contains loop
   ...
   for (...) {
     ...
   }
   ...
}
```

Example 3

Following are examples of legal latency constraint use:

```
// LEGAL USE OF LATENCY CONSTRAINTS
{
   {
      HLS_CONSTRAIN_LATENCY(0,10,"lat1"); // LEGAL
      ...
   }
   for (...) {
      HLS_CONSTRAIN_LATENCY(0,3,"lat2"); // LEGAL
      ...
   }
   {
      HLS_CONSTRAIN_LATENCY(0,10,"lat3"); // LEGAL
      ...
   }
}
// LEGAL USE OF LATENCY CONSTRAINT WITH UNROLLED LOOP
{
   HLS_CONSTRAIN_LATENCY(0,10,"lat4"); // LEGAL
      ...
   for (...) {
      HLS_UNROLL_LOOP(COMPLETE,8,"shift"); // LEGAL
      ...
   }
   ...
}
```

Example 4

Constrain latency to the minimum achievable:

```
    while (1) {
     HLS_CONSTRAIN_LATENCY( 0, HLS_ACHIEVABLE, "main lat" );
     ...
    }
```

Example 5

Constrain latency to no less than 2 cycles:

```
while (1) {
 HLS_CONSTRAIN_LATENCY( 2, HLS_INFINITE, "main lat" );
 ...
}
```

Example 6
Do not constrain the latency of the body of the loop, but print the latency achieved. Note that adding this directive this may result in a different design because it may impose code motion rules that prevent operations from function `f()` from being moved outside of the loop.

```
for ( int i=0; i < 16; i++ )  {
 HLS_CONSTRAIN_LATENCY( "loop lat" );
 out[i] = f(in[i]);
}
```

HLS_CONSTRAIN_REGION

A Stratus HLS subdirective.

Syntax

`HLS_CONSTRAIN_REGION(` ***`minlat`*****`,`** ***`maxlat`*** ***`[,`*** ***`indel`*****`,`** **`maxdel ] );`**

Equivalent Command

constrain_region

Parameters

`minlat`

The minimum latency of the resource created for the region. The minimum is 0 for `HLS_DPOPT_REGION`s and 1 for `HLS_SCHEDULE_REGION`s.

`maxlat`

The maximum latency of the resource created for the region. `HLS_INFINITE` to indicate no constraint, and `HLS_ACHIEVABLE` to indicate the fastest achievable latency.

`indel`

The input delay constraint. This amount of time will be reserved at the start of the clock cycle before the resource is scheduled if the used when building the resource. Optional, or -1 to indicate no constraint.

`maxdel`

The maximum delay amount in the same time units used by the project's `tech_lib`. This is an optional parameter that specifies the maximum time after which the resources outputs must become stable.

Recommended placement

Place immediately following the associated `HLS_DPOPT_REGION` or `HLS_SCHEDULE_REGION` directive.

Description

The `HLS_CONSTRAIN_REGION` directive can be used to control the latency and timing of the resource created for either an `HLS_DPOPT_REGION` or an `HLS_SCHEDULED_REGION`. This optional directive should be placed adjacent to a `HLS_CONSTRAIN_REGION` or `HLS_DPOPT_REGION` directive. Two kinds of constraints can be specified:

1. **Latency constraints**. These specify the latency of the resource created for the region. For a DPOPT region, this is the number of internal register stages. For a scheduled region, it is the *acyclic* latency through the region. That is, the latency not considering the number of iterations of rolled up loops in the region.

2. **Timing constraints**. These specify timing constraints at the input and output of the resource created for the region. Input delays are measured from a clock edge until the first stage of logic begins, and are only relevant for resources with latency. The max output delay specifies the time after which the outputs will become stable after the last clock edge, or for asynchronous resources, the maximum total delay for the resource.

These constraints are enforced as limits when the resource is created for the region. The actual timing and latency of the resource is used when scheduling operations using that resource in the accessing thread.

If only latency constraints are required, the timing constraint parameters can be omitted. If only timing constraints are required, values must still be supplied for the latency constraints since they appear first.

If no `HLS_CONSTRAIN_REGION` has specified, or if values are omitted from the directive, the following assumptions are made:

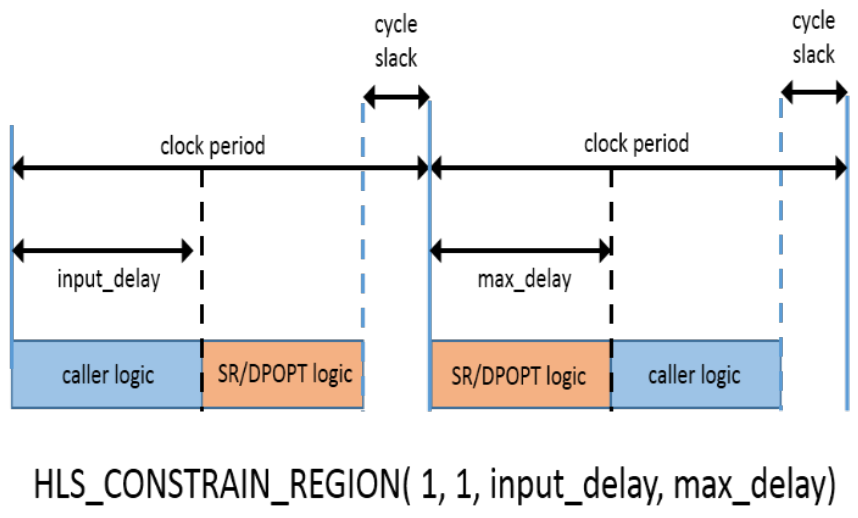
- The input delay is clk->q, making the entire clock cycle available to the first stage of logic for the region.
- There is no maximum delay. Clock cycles will be added as required, and the final stage can contain any amount of logic.
- For `HLS_SCHEDULE_REGION`, the latency will be the shortest achievable latency, and at least 1 cycle.
- For `HLS_DPOPT_REGION`, several latencies may be achieved with the one giving the best overall results selected. The latency may be 0.

`HLS_CONSTRAIN_REGION` is similar to `HLS_CONSTRAIN_LATENCY`, except that `HLS_CONSTRAIN_LATENCY` sets the latency of the design/algorithm while `HLS_CONSTRAIN_REGION` sets the latency of a part created for the algorithm. Multiple parts may be scheduled in parallel. `HLS_CONSTRAIN_REGION` does not constrain the design, but it may be nested within `HLS_CONSTRAIN_LATENCY` blocks that are being used to constrain the design.

The diagram below describes how an `HLS_CONSTRAIN_REGION` directive applies to a scheduled region, or DPOPT region with a latency of 1.

The *caller logic* box show which parts of the clock cycle logic scheduled in the accessing thread can occupy. The *SR/DPOPT logic* boxes show which part of the clock cycle logic scheduled for the region being constrained can be scheduled.

Example 1



This example constrains the `mypart` DPOPT resource to have a latency of exactly 1 with an input delay of 1.5 and an max output delay of 2.5.

```
{
  HLS_DPOPT_REGION("mypart");
  HLS_CONSTRAIN_REGION( 1, 1, 1.5, 2.5 );
  out = f(in);
}
```

Example 2

This example constrains the `mypart` resource to be asynchronous with a max delay of 3.5.

```
{
  HLS_DPOPT_REGION("mypart");
  HLS_CONSTRAIN_REGION( 0, 0, 0, 2.5 );
  out = f(in);
}
```

Example 3

This example constrains the `cube` scheduled region to have a latency of exactly 3 to encourage the multipliers to be shared.

```
sc_uint<8> cube ( sc_uint<8> v ) {
  HLS_SCHEDULE_REGION();
  HLS_CONSTRAIN_REGION( 3, 3 );
  return v * v * v;
}
```

Example 4

This example constrains the `getin` scheduled region to have a latency of 1, no input delay limit, and a max delay in the final stage of 2.5.

```
sc_uint<8> getin ( sc_uint<8> v ) {
  HLS_SCHEDULE_REGION();
  HLS_CONSTRAIN_REGION( 1, 1, 0.0, 2.5 );

  return din.get() + v;
}
```

See also:

- HLS_DPOPT_REGION
- HLS_SCHEDULE_REGION
- Controlling latency and delays within DpOpt parts in the *User Guide.*

HLS_DEFINE_FLOATING_PROTOCOL

A Stratus HLS directive.

Syntax

`HLS_DEFINE_FLOATING_PROTOCOL``(` *`setup`*`,` *`delay`*`,` *`ii`*`,` *`context`*`,` *`flags`*`,` *`address`*`,` *`name`* `);`

Equivalent Command

define_floating_protocol

Parameters

`setup`

Specifies the time at the end of the first clock cycle of the transaction during which outputs are required to be stable.

`delay`

Specifies the delay after which the transaction's outputs can be assumed to be stable and can be safely read.

`ii`

Initiation interval for transactions.

`context`

This may be a variable, the address of a variable, or an integer.

`flags`

Flags that affect scheduling of the transaction operations.

`address`

Specifies an array index value that is used to identify accesses to conflicting addresses. If your interface has a single scalar value that could be used like an array index, specify it as the address value. Otherwise, specify an integer 0.

name

Provides a suffix for the name used in reporting to refer to this block.

Recommended placement

Place in the block of code that defines the transaction. This is often at the top of a function that defines a transaction.

Description

The `HLS_DEFINE_FLOATING_PROTOCOL` directive defines a block containing a fixed-length I/O protocol that Stratus HLS can schedule in parallel with similar protocols on other interfaces. This directive is typically used to define transaction functions, such as memory read or write operations. For example:

```
sc_uint<8> read_mem1( sc_uint<10> a ) {
   HLS_DEFINE_FLOATING_PROTOCOL( 1.2, 2.5, 1, &mem1_context, HLS_MEM_READ_FP,
a, "read_mem1" );
   mem1_addr = a;
   mem1_ce = 0;
   mem1_we = 1;
   wait();
   mem1_ce = 1;
   wait();
   return mem1_dout.read();
}
```

The `HLS_DEFINE_FLOATING_PROTOCOL` directive will cause Stratus HLS to treat the `read_mem1` function in the following way:

- Each call to `read_mem1()` from an `SC_CTHREAD` will be scheduled as a transaction. This means that the memory read transaction can be *moved* by Stratus HLS's scheduling engine as required to meet latency constraints.
- The fourth (*context*) parameter, mem1, defines a context for the transaction; it identifies the interface with which this transaction interacts. Stratus HLS will ensure that two transactions with the same context will never be scheduled in a conflicting way. However, it will allow transaction functions for different contexts to be overlapped without conflict. It is often convenient to use a variable, or the address of a variable as a context, but an integer can be used as well.
- The third (*pipe_init*) parameter, `1`, specifies that the transaction has an initiation interval of 1. This allows Stratus HLS to schedule a new transaction on this interface every clock cycle.
- There are two calls to `wait()` in the function. When executed from an `SC_CTHREAD`, this will execute a fixed-length protocol that accesses a resource with a latency of 1 (for example, a memory component with a latency of 1).
- The first two parameters, `1.2` and `2.5`, specify *setup* and *delay* timing respectively. These values reflect the timing requirements of the external component being interacted with. They are used by Stratus HLS to determine how much logic can be chained in the states when the transaction begins and ends.

The result is a fixed-length I/O protocol that can be scheduled by Stratus HLS much like it schedules access to algorthmic resources. Contrast this with I/O performed in a

`HLS_DEFINE_PROTOCOL` block. I/O in a `HLS_DEFINE_PROTOCOL` block cannot be executed in parallel with other `HLS_DEFINE_PROTOCOL` blocks and cannot be moved in the schedule to meet latency constraints. This difference makes `HLS_DEFINE_FLOATING_PROTOCOL` the best choice for accessing resources with fixed-length protocols, but a bad choice for defining handshaking protocols.

The `HLS_DEFINE_FLOATING_PROTOCOL` directive has the following parameters:

- **setup and delay**: Similar to an internal memory within a module, if a *setup* time is specified, it is possible for Stratus HLS to safely create a chain of logic into the state when the transaction is started. If a *delay* is given, it is possible for Stratus HLS to create a chain of logic in the state where the outputs of the transaction are read. If a value of 0 is specified, Stratus HLS will always place a register at the design outputs (for *setup*) or design inputs (for *delay*). The units used must match the units from the `hls_lib` being used for the design.

  The special value `HLS_CALC_TIMING` indicates that the timing for the transaction is not known and should be calculated from the timing known from other sources for the signals or ports used in the transaction.

  If a positive setup time is given, then that setup time is used as an output delay to constrain outputs from Stratus HLS during logic synthesis. If a positive delay time is given, then that delay value is used as an input delay to constrain the inputs to Stratus HLS during logic synthesis. No constraints are emitted for logic synthesis when a delay value of zero is specified.
- **ii**: The *pipe_init* parameter is the initiation interval. It defines the time after an operation is begun that another can be started on another atomic-transaction block with the same context. If *pipe_init* is zero, then operations cannot be pipelined. The initiation interval is used by Stratus HLS to prevent the starting of an atomic transaction before a preceding atomic transaction operation with the same context has executed its initiation interval. Operations with the same context may be overlapped, but there will never be two operations from the same context scheduled within their specified initiation intervals.
- **context**: The *context* parameter defines which atomic transactions represent different operations on the same interface. This is used to determine which operations may be scheduled in the same cycle. For convenience, the context is often the "this" parameter of the port class. It may also be a string or an integer.
- **flags**: Possible values are:
  - **HLS_DEFAULT_FP**: No options specified.

- **HLS_MEM_READ_FP**: The operation should be treated as a memory read.
- **HLS_MEM_WRITE_FP**: The operation should be treated as a memory read.
- **HLS_UNSTALLABLE_FP**: Specify that the resource being communicated with does not handle pipeline stalls, which will result in a stall fifo being inserted automatically.
- **HLS_CHAIN_MEM_IN**: Allows chaining of logic in the state in which the operation is begun, if possible given the *setup* parameter value. This is the default.
- **HLS_CHAIN_MEM_OUT**: Allows chaining of logic in the state in which the operation's outputs are read, if possible given the *delay* parameter value. This is the default.
- **HLS_CHAIN_MEM_IO**: Specifies both `HLS_ASYNC_MEM_IN` and `HLS_ASYNC_MEM_OUT`.
- **HLS_IGNORE_BOUNDARIES**: If this flag is set, the memory read or write operation will not be limited in movement by boundaries. Ordinarily, an external memory read or write operation is limited in movement by the boundaries that surround it. By default, a boundary is created by each `HLS_DEFINE_PROTOCOL` block.
- **HLS_NO_CHAIN_MEM_IN**: Causes registers to be used at the outputs from the accessing thread, independent of the *setup* parameter value.
- **HLS_NO_CHAIN_MEM_OUT**: Causes registers to be used at the inputs to the accessing thread, independent of the *delay* parameter value.
- **HLS_NO_CHAIN_MEM_IO**: Specifies both `HLS_REG_MEM_IN` and `HLS_REG_MEM_OUT`.
- **HLS_ALLOW_SPEC_READS**: Allows speculative reads to be scheduled. This is the default.
- **HLS_NO_SPEC_READS**: Prevents operations from being executed before enclosing conditions have been met. For example, if a memory read operation is used as follows:

```
if (en)
    v = read_mem1(a);
```

  this option will prevent the operation from being scheduled before the `if (en)` condition can be tested. This may make it more difficult to meet scheduling constraints, but will have a positive effect on power consumption.

- **address**: The *address* parameter identifies the variable that specifies the array index value. Stratus HLS uses this to perform dataflow analysis amongst memory operations. Specification of an *address* parameter allows Stratus HLS to consider the address when analyzing the required order of memory operations. If no such value exists for an application, specify 0.
- **name**: The *name* parameter provides a suffix for the name used in reporting to refer to this

block. This parameter can safely be left as an empty string.

Latency

The protocol for an atomic transaction block is specified as a series of wire ops and calls to `wait()`. For example:

```
data_type get( const address_type& address )
{
   HLS_DEFINE_FLOATING_PROTOCOL( 0.5, 1.2, 1, this, HLS_MEM_READ_FP |
HLS_UNSTALLABLE_FP, address, "" );
   RW = 1; // 1 means read, 2 means write.
   ADDR = address;
   CE = true;
   wait(1);// One clock for memory to receive inputs
   CE = false;// Deassert after 1 cycle
   wait(1);// One clock for us to latch the data
      return DIN.read();// Read the data
}
```

The total number of `wait()`s in the block is used by Stratus HLS to determine the latency of the operation for the purposes of scheduling. The latency will be *N-1* where *N* is the number of `wait()`s in the block directive. For example, a `get()` function containing two `wait()`s models the protocol for accessing a memory with latency of 1. If Stratus HLS chains into the memory operation, it will consume *N-1* cycles in the schedule. If the memory operation is not chained into, it will consume *N* cycles in the schedule. Note that Stratus HLS will only chain into a memory operation if a *setup* time is specified for it in the atomic transaction directive.

The use of `HLS_MEM_READ_FP | HLS_UNSTALLABLE_FP` specifies that this operation should be scheduled similar to a memory read operations, and that a stall fifo should be added if the accessing thread stalls.

Limitations on Movement of Transactions

A specified transaction block can be moved in the FSM by Stratus HLS's scheduling engine to meet latency targets. However, there are limits to where the transaction can be moved:

- **Dataflow**: The dataflow into and out of the transaction must be met. For example:

```
sc_uint<8> v = read_mem1( a + offset );
     dout.put(v+1);
```

The memory read transaction cannot be scheduled until the `a+offset` operation has been scheduled. This is a limit caused by input dataflow restrictions. The memory read must also be scheduled before the `v+1` operation. This is a limit caused by output dataflow restrictions.

- **Boundaries**: Boundaries are locations in the FSM. They are defined implicitly by `HLS_DEFINE_PROTOCOL` blocks. A transaction operation will, by default, be prevented from moving past a boundary. For example:

```
{HLS_DEFINE_PROTOCOL("sync");
   do {wait();} while (!sync);
}
sc_uint<8> v = read_mem1( a );
dout.put(v+1);
{HLS_DEFINE_PROTOCOL("write");
   wait();
   dout.write(v);
}
```

There are two boundaries defined here. One is defined by the `sync` `HLS_DEFINE_PROTOCOL` block, and the other is defined by the `write` `HLS_DEFINE_PROTOCOL` block. The `read_mem1()` transaction will not be moved any earlier than the end of the `sync` block, nor any later than the beginning of the `write` block. Note that there is no dataflow preventing the operation from moving before the `sync` block.

This behavior can be disabled by specifying the `HLS_IGNORE_BOUNDARIES` option in the flags option of the directive.

- **Conditions**: A condition enclosing a transaction may or may not limit where the transaction can move. For example:

```
if (en) {
  v = mem1_read(a);
}
```

A transaction defined with `HLS_DEFINE_FLOATING_PROTOCOL` is assumed to have a side effect, and so is never moved outside of an enclosing condition. Transactions defined with `HLS_DEFINE_FLOATING_PROTOCOL` is assumed to be movable outside of enclosing conditions. This is a speculative execution of the transaction. Speculative execution can be prevented using the `HLS_NO_SPEC_READS` flag.

Assertion of Outputs after the Initiation Interval

If an initiation interval is specified in an atomic transaction directive, Stratus HLS can schedule an operation to begin before a preceding operation with the same context has completed. The initiation interval determines how far after the start of one operation another can begin.

In the example shown in "Latency", the initiation interval is 1, but there are two `wait()`s. This will allow an operation to begin every clock cycle with consecutive operations overlapping. This protocol asserts the CE signal in the first cycle, and deasserts it in the last cycle. This causes conflicting values to be written by earlier and later operations which are overlapped. Stratus HLS resolves this conflict by giving precedence to values written within the initiation interval. In this example, this results in CE continuing to be asserted throughout a string of pipelined operations. This rule implies that outputs should be written after the initiation interval only when they are deassertions of signals that can safely be treated as weak assertions and thus be eliminated by overlapping operations.

Limitations of Atomic-Transaction Directives

If `HLS_DEFINE_FLOATING_PROTOCOL` is used, the following restrictions apply to the code within the block containing the directive:

- Only wire operations are permitted. That is, no arithmetic or logical operations that would require functional-units are permitted.
- No conditionals are permitted.
- No loops are permitted.
- The block must contain at least one `wait()`.
- Any number of additional `wait()` statements is permitted.
- Wire operations, including bit slice and concatenation are permitted.
- Reading and writing of ports or inter-thread signals is permitted.
- Reading and writing of values coming into the block, or otherwise within scope, is permitted, including member variables of the port class.
- Placement of wire operations between `wait()` statements is permitted. This allows signals to be asserted at the start, in the middle, or at the end of the transaction.

These limitations permit modeling of protocols with fixed latency and no branching.

HLS_DEFINE_PROTOCOL

A Stratus HLS directive.

Syntax

`HLS_DEFINE_PROTOCOL``(` *`name`* `);`

Equivalent Command

define_protocol

Parameters

`name`

A string that uniquely identifies the directive.

Recommended placement

Place immediately following the curly brace that encloses the affected block of code.

Explanation

Stratus HLS permits the mixture of manually-scheduled behavior containing explicit `wait()` statements with behavior whose schedule will be determined automatically by Stratus HLS. Manually-scheduled behavior is used to maintain cycle accuracy for I/O protocol. You specify this behavior using a `CCOL` directive.

Cycle-accurate I/O protocol must be contained in braces ({ }), and the first statement of that code block must be a call to `HLS_DEFINE_PROTOCOL`. `HLS_DEFINE_PROTOCOL` takes a string argument (*block_name*) that is used for reporting.

The presence of `HLS_DEFINE_PROTOCOL` signifies that port reads and writes within the brace-enclosed block (and any functions it calls) should remain cycle-accurate. This implies that any operations in the protocol block must be able to be implemented within the time specified in the protocol. It is up to the user to make sure operations contained in a protocol block are schedulable. If they are not, Stratus HLS will produce an error message.
Protocol blocks may contain conditional jumps out of the protocol block (such as via a `break` or `return` statement), but there should not be any port access in the basic block containing the jump.

See also:

- Describing I/O Protocol in the *User Guide*

Example 1

```
{
  HLS_DEFINE_PROTOCOL( "inputs");
  a = in.read();
  wait(1);
  b = in.read();
  wait(1); // 2-cycle input protocol
}
{
  // Internal behavioral code
}
{
  HLS_DEFINE_PROTOCOL( "outputs" );
  out.write( x );
  wait(1);
  out.write( y );
  wait(1);
  out.write( z );
  wait(1);// 3-cycle output protocol
}
```

This example shows how protocol blocks (`inputs` and `outputs`) can coexist with the internal behavioral code.

Example 2

```
sc_uint<8> io_function() {
    sc_uint<8> value;
    HLS_DEFINE_PROTOCOL("io");

    do {
        wait( );
        value = data_port.read();
    }while ( valid_port.read() != true );

    if( value > 10)
    {
        data_port_out = 1;
        wait();
        return 0; // IO and wait statement are contained
                  // in basic block that jumps outside protocol block.
                  // This will have unexpected results.
    }
    return value; // This is okay.
}
```

In this example, the I/O and `wait()` statements jump outside the protocol block. Because there is also port access in the block containing the jump, unexpected results can occur.

Example 3

```
{
  HLS_DEFINE_PROTOCOL( "bad1" );
  a = aIn.read();
  b = bIn.read();
  c = cIn.read();
  d = dIn.read();
  wait();
  zOut = a * b * c * d; // illegal if not schedulable!
}
```

In this example, it is unlikely that the protocol block would be schedulable, because it would require the operation `a*b*c*d` to be implemented in a single cycle. In this case, Stratus HLS would produce an error message saying that `HLS_DEFINE_PROTOCOL` block `bad1` cannot be scheduled.

HLS_DEFINE_STALL_LOOP

A Stratus HLS directive.

Syntax

HLS_DEFINE_STALL_LOOP([*type*] [*name*]);

Equivalent Command

define_stall_loops

Parameters

type

Specifies for which loops the stall is to be inferred. The value of *type* can be ON, OFF, ALL, or AGGRESSIVE. The default type is ALL.

name

The name of the directive used for reporting purposes (optional).

Recommended placement

Place in the body of the loop for which stalls are to be inferred.

Description

The `HLS_DEFINE_STALL_LOOP` directive causes a pipeline stall to be inferred for a loop, even if the loop is outside of a pipeline. Only loops that will implicitly infer stalls within pipelines can have stalls inferred using `HLS_DEFINE_STALL_LOOP`. For example, the following will infer a stall both inside and outside a pipeline:

```
do {
  HLS_DEFINE_STALL_LOOP(ALL);
  wait();
} while (!rdy.read());
```

Loops of the following form will also infer a stall:

```
while (!rdy.read()) {
  HLS_DEFINE_STALL_LOOP(ALL);
  wait();
}
```

The only supported value for the *options* parameter is `ALL`. The *options* parameter can be omitted and will default to `ALL`. Future releases may support other values for this parameter.

There are two advantages of inferring a pipeline stall for a loop outside of a pipeline:

- Control is removed from the design's FSM, which will allow more design elements to be executed in parallel.

  Stratus HLS's scheduling engine will not schedule two loops from the same thread in parallel. Nor will it schedule a conditional branch in parallel with a loop. Inferring a pipeline stall effectively removes a loop from the FSM (the busy loop). This removes restrictions from scheduling since a pipeline stall can be scheduled in parallel with either another loop or a conditional branch. This can improve latencies, sometimes significantly.

- It can make the output register safely sharable.

  The `HLS_SET_OUTPUT_DELAY` directive can be used to make an output port's register sharable, but only if it's properly protected by a stalling protocol. The `HLS_DEFINE_STALL_LOOP` directive can be used for this purpose. See Making registered outputs sharable in the *User G*uide.

  See also:

- Pipeline stalling in the *User Guide*
- Making registered outputs sharable in the *User Guide*

Example

```
void mod::thread0()
{
  while () {
    ...
    y = f(x);

    // Infer a stall outside of a pipeline.
    dout = y;
    vld = 1;
    do {
      HLS_DEFINE_STALL_LOOP();
      wait();
    } while (busy.read());
    vld = 0;
}
```

HLS_DPOPT_REGION

A Stratus HLS directive.

Syntax

```
HLS_DPOPT_REGION( [flags] [, name ]);
```

Equivalent Command

dpopt_region

Parameters

flags

Configurations that control how parts are created with DpOpt.

name

The name of the custom datapath component. In other words, the datapath component will be placed in a module named as the specified *name*. *name* must be a legal identifier in both C++ and Verilog and must be no more than 200 characters in length. The name can be omitted, in which case Stratus HLS automatically creates a name. If the region is a function, the function name will be used. If the region is a loop, the loop name will be used. Otherwise, names will be created using "dpopt_%d" format.

Recommended placement

Place immediately following the curly-brace that encloses the code block to be optimized.

Description

Stratus HLS's datapath optimization technology is controlled by the use of the `HLS_DPOPT_REGION` directive.

There are several flags that may be included as the *config* parameter in the `HLS_DPOPT_REGION` directive. These flags may be included individually or together using the bitwise-OR operator.

The flags are:

- **BEFORE_UNROLL**: Applies DpOpt to a block of code before loop unrolling is performed. This can significantly improve Stratus HLS scalability and runtime by encapsulating large portions of a designs data flow into a single DpOpt part very early in the synthesis process. It has the potential to increase the area of the design because the data flow encapsulated in the DpOpt part before loop unrolling is not available for downstream optimizations such as constant propagation and dead code elimination. However, if you plan to use `NO_ALTS`, it is very likely that `BEFORE_UNROLL` will produce the same results with improved runtime and capacity.
- **DPOPT_DEFAULT**: Specifies the default settings for `HLS_DPOPT_REGION`. The default is to

produce only for minimum delay (`OPTIM_DELAY`).

- **DPOPT_FPGA_USE_DSP**: Enables (optionally) mapping of DpOpt parts into the DSP slices of the specified FPGA part by controlling the `use_dsp` property at the individual resource level. This flag does not have any impact during ASIC mode.
- **NO_ALTS**: Implies NO_CONSTANTS | NO_CSE | NO_DCE | NO_TRIMMING.
- **NO_CHAIN_IN**: Disables chaining of additional logic into the inputs of the generated part. Scheduling will place the part at the beginning of a clock cycle with registers immediately in front of the part. The registers will not be included within the generated part and will therefore be available for register sharing in allocation. Part sharing in allocation may result in a mux being placed between the register and the inputs of the generated part.
- **NO_CHAIN_OUT**: Disables chaining of outputs of generated part into additional logic. Scheduling will place the part at the end of a clock cycle with registers immediately following the part. The registers will not be included within the generated part and will therefore be available for register sharing in allocation. Register sharing in allocation may result in a mux being placed between the outputs of the generated part and the register.
- **NO_CONSTANTS**: Instructs DpOpt to not propagate constants into newly created parts. This value is used if there will be multiple instances of the generated block and each will not have the same constant inputs.
- **NO_CSE**: Disables common subexpression elimination in DpOpt parts in order to prevent alternate parts from being created.
- **NO_DCE**: Disables dead code elimination in DpOpt parts in order to prevent alternate parts from being created.
- **NO_TRIMMING**: Instructs DpOpt to not trim input or output port widths of the newly created part. This value is used if there will be multiple instances of the generated block that will use different input/output widths, up to the maximum.

Stratus HLS automatically unrolls loops within `HLS_DPOPT_REGION` blocks if the number of iterations can be determined. No loop unrolling directives are necessary.

Array flattening is implied with the use of `HLS_DPOPT_REGION`.

The behavior of datapath optimization is also affected by the following:

- The `dpopt_effort` attribute in Synthesis Control Attributes
- HLS_CONSTRAIN_REGION
- HLS_CONSTRAIN_LATENCY

See also:

- HLS_SCHEDULE_REGION
- HLS_UNROLL_LOOP
- Specifying DpOpt Region in the *User Guide*

Restrictions on DpOpt use

Following are restrictions on the behavior inside of a `HLS_DPOPT_REGION` block:

- The code cannot access memories. (Accesses to flattened arrays are acceptable.)
- *Note:* As of v2.4, there are no longer restrictions on code that contains `for` loops in `HLS_DPOPT_REGION` blocks. Stratus HLS automatically unrolls loops within `HLS_DPOPT_REGION` blocks if the number of iterations can be determined. No loop unrolling directives are necessary. The result can then be fed to DpOpt.
- The code cannot jump to outside the block. (Constructs such as `goto`, `break`, and `continue` will violate this restriction unless these move execution to elsewhere in the `HLS_DPOPT_REGION` block.)

As an example of the final restriction, note that the first example below violates this, while the second does not:

```
switch (a) {
  case 1:
    {
       HLS_DPOPT_REGION("VIOLATION");
       ...
       if (...) {
         ...
         break; // violates restriction, jumps outside this block
       }
       ...
    }
  ...
}
{
  HLS_DPOPT_REGION("OK");
  switch (a) {
     case 1: ...
             break; // ok, jumps elsewhere in HLS_DPOPT_REGION
     case 2: ...
             break; // ok, jumps elsewhere in HLS_DPOPT_REGION
    default: ...
            break; // ok, jumps elsewhere in HLS_DPOPT_REGION
  }
}
```

Any other synthesizable behavior is legal, including function calls. Of course, the bodies of any functions called in this block must adhere to the `HLS_DPOPT_REGION` coding restrictions mentioned above.

- *name* must be legal as an identifer in both SystemC and Verilog. It must start with an letter and contain only letters, digits, and underscore (that is, it must match the pattern [A-Za-z][A-Za-z0-9_]*). Furthermore, it must not be a reserved word of C++, SystemC, or Verilog, it must not contain double underscores (_*), and it must not end in an underscore (*).

Example 1

```
{
  HLS_DPOPT_REGION("myPart");
  a = b * c + d * 17;
  tmp = b + d;
  if (e) {
    a = a + tmp; // value of "a" is defined
  }
}
...
... = a; // value of "a" is used, "tmp" is never used
```

This will create a gate-level part named `myPart` with four inputs (of the types of `b`, `c`, `d`, `e`) and one output (of the same type as `a`). In this case, `a` is the only output since it is the only variable whose value is defined in this block and used elsewhere.

Note that since this directive is applied to code between the surrounding braces, the input for DpOpt need not be separated as a function. This directive can be applied to an arbitrary block of code simply by bounding it with braces.

Example 2

```
{
   HLS_DPOPT_REGION("myPart");
   HLS_CONSTRAIN_REGION( 2, 3, HLS_CLOCK_PERIOD-10 );
   a = b * c + d * 17;
   tmp = b + d;
   if (e) {
     a = a + tmp; // value of "a" is defined
   }
}
```

This will create a part named `myPart` with 10ns worth of combinational logic between the inputs (`a`, `b`, `c`, `d`) and the first register internal to the part. There will then be combinational logic for a full clock cycle up to the second internal register, and then another full cycle of combinational logic up to the third register if DpOpt decides to use a third set of registers. After the last set of registers, there will be 5ns of combinational logic before the part's output port (`a`).

Example 3

```
{
   HLS_DPOPT_REGION("part1");
   for ( i = 0; i < 8; i++ ) {
     // This is unrolled because it's in a DPOPT_INLINE part
   }
   for ( i = 0; i < variable; i++ ) {
      // This is not unrolled because Stratus HLS does not know how
      // many times it iterates.
      // You will get multiple dpopt_inline parts to the
      // functionality before, within, and after this loop.
   }
   for ( i = 0; i < variable; i++ ) {
      HLS_UNROLL_LOOP(OFF,"loop1");
      // This loop will not be unrolled.
   }
}
```

This example highlights Stratus HLS loop unrolling behavior for `HLS_DPOPT_REGION` blocks. Stratus HLS will automatically unroll the `for` loop within *mypart1*. Stratus HLS would have unrolled the second for loop because it is also within the `HLS_DPOPT_REGION`, however Stratus HLS cannot determine the number of iterations. Stratus HLS will not unroll the third `for` loop because of the explicit `OFF` setting.

None of the loops in this example are affected by the `--unroll_loops` flag.

Example 4

```
   for (int i=0; i < 16; i++ ) {
       HLS_DPOPT_REGION(NO_ALTS, "myPart");
       HLS_UNROLL_LOOP(ON);
       out[i] = in[i] * i;
   }
}
```

This example creates a part for the body of the loop, and uses the NO_ALTS option to ensure that an identical part is used for every iteration of the unrolled loop. If this option had not been used, since the size of variable i will be different for different iterations, different width parts would have been created which would have prevented the parts from being shared.

Example 5

```
   for (int i=0; i < 16; i++ ) {
       HLS_DPOPT_REGION(BEFORE_UNROLL, "myPart");
       HLS_UNROLL_LOOP(ON);
       out[i] = in[i] * i;
   }
}
```

This example has a similar effect to the previous one except it uses the `BEFORE_UNROLL` option to achieve it. `BEFORE_UNROLL` has the added benefit of providing better scalability because the dpopt part is extracted before the loop is unrolled, which reduces the internal size of the design being processed by Stratus HLS. Also, using `BEFORE_UNROLL`, you can be assured that parts for each iteration are sharable because the loop iterator will not be a constant at the time the part is created.

Example 6

```
for (size_t i = 0; i < 16; i++) {
    HLS_DPOPT_REGION( DPOPT_FPGA_USE_DSP, "multiply", "");
    HLS_CONSTRAIN_REGION(1,1, 100.0 , 100.0 );
    sum += data[i] *  coef[i];
  }
}
```

This example creates a part for the body of the loop, and uses the `DPOPT_FPGA_USE_DSP` option to ensure that DSP slices of the specified FPGA part is used for a latency of 1. The blocks of code are constrained to a latency of 1 cycle.

Example 7

```
for (size_t i = 0; i < 16; i++) {
    HLS_DPOPT_REGION( NO_CHAIN_IN | DPOPT_FPGA_USE_DSP, "multiply", "");

    HLS_CONSTRAIN_REGION(1,1, 100.0 , 100.0 );
    sum += data[i] *  coef[i];
  }
}
```

This example has a similar effect to the previous one, it creates a part with latency 1, but ensures that registers will immediately drive the inputs such that FPGA synthesis tools will be able to map into a DSP slice using both input and internal registers optimally, for a latency of 2 cycles.

Example 8

```
for (size_t i = 0; i < 16; i++) {
    HLS_DPOPT_REGION( NO_CHAIN_IN | NO_CHAIN_OUT | DPOPT_FPGA_USE_DSP, "multiply", "");
    HLS_CONSTRAIN_REGION(2,2, 100.0 , 100.0 );
    sum += data[i] *  coef[i];
  }
}
```

This example instructs Stratus HLS to synthesize a DPOPT part with latency of 2, and ensures that registers will be connected to both inputs and outputs of the part when it is scheduled, this will allow an FPGA synthesis tool to map into a DSP slice which can utilize 4 cycles of latency, described as one stage at the input and 3 stages at the output.

HLS_EXPOSE_PORT

A Stratus HLS directives.

Syntax

HLS_EXPOSE_PORT(*on_off, port_name*);

Equivalent Command

None

Parameters

on_off
If `ON`, the given port will be exported to the RTL module.

port_name
The name of the port to be affected.

Recommended placement

Place within a struct or class definition.

Description

The `HLS_EXPOSE_PORT` directive controls whether a port in a `HLS_METAPORT` class will be placed on the RTL module. Use `HLS_EXPOSE_PORT` to expose an individual port or multiple ports. This directive must appear within a struct or class definition, and can affect ports declared within that class or its base classes. Explicit port names can refer to either ports in the class itself, or ports declared in the base class.

Settings are applied in the following order:

- Default settings based on each port's position in a `HLS_METAPORT` or `HLS_INLINE_MODULE` class.
- Directives in the class in which declarations appear.
- Directives in derived classes (which may affect their base classes).
- Directives with the value `ON`.
- Directives with the value `OFF`.

The `HLS_EXPOSE_PORT` directive is used most commonly used when a class is marked with both `HLS_INLINE_MODULE` and `HLS_METAPORT`. For more information on using `HLS_EXPOSE_PORT` with this design pattern, see "Special Precautions when a Class Is both a `HLS_INLINE_MODULE` and a `HLS_METAPORT`" in Creating Active Modular Interfaces.

See also:

- HLS_INLINE_MODULE
- Combining HLS_METAPORT and HLS_INLINE_MODULE in the *User Guide*

HLS_FLATTEN_ARRAY

A Stratus HLS directive.

Syntax

`HLS_FLATTEN_ARRAY``(` *`array_id`* `[,` *`option`* `] );`

Equivalent Command

flatten_arrays

Parameters

`array_id`

The name of the array to be flattened.

`option`

Whether or not to flatten the array. The options are DEFAULT_FLATTEN (default option), DPOPT_FLATTEN, or DONT_FLATTEN. However, note that DPOPT_FLATTEN cannot be applied to arrays of signals.

Recommended placement

- Place in the module constructor for arrays that are data members of the module.
- Place in the module constructor for arrays that are (`const`) global variables.
- Place immediately after the array declaration for arrays that are automatic variables in functions.

Description

The `HLS_FLATTEN_ARRAY` directive is used to implement an array as individual registers rather than as a memory. The `HLS_FLATTEN_ARRAY` directive may be applied to any array, whether it is a local array, class member, or global array. This directive overrides global array flattening options (for example, `flatten_arrays=none`).

If an array is not specified to be flattened, either because the value of the `flatten_arrays` attribute does not select it or because `HLS_FLATTEN_ARRAY( array_name, DONT_FLATTEN)` has been specified for it, Stratus HLS will implement the array with an RTL memory component. Access to the memory component is through read/write ports, and few memories have more than three ports, so the number of concurrent accesses to the memory in a given clock cycle is very limited. Using a memory component usually results in the smallest RTL area for an array.

If an array is specified to be flattened, Stratus HLS will transform the array into a set of discrete registers, one register per word in the array. This transformation allows read and write access to every word in the array every clock cycle, but usually results in a larger area than a memory since one flip-flop is required for each bit in the array.

Array flattening is most beneficial when all indices used to access the array are constant (or statically determinable, as in a loop index variable in a loop that is completely unrolled). Writing to a flattened array with a variable index implies a potentially large construct to determine which individual value is to be written. This will be a large control construct if DEFAULT_FLATTEN is specified, a large combinational custom datapath component if DPOPT_FLATTEN is specified. Similarly, reading from a flattened array with a variable index implies a potentially large multiplexor to select the appropriate value. Either situation can significantly degrade the quality of results. Variably indexed array writes, in particular, also have the potential to trigger very large increases in Stratus HLS's runtime and memory requirements when DEFAULT_FLATTEN is specified (depending on the size of the array and the patterns in which it is accessed).

With DEFAULT_FLATTEN, variable array writes on the left-hand side of an assignment create control (switch statements) with a write to the appropriate register in each branch. Array reads create muxes with the appropriate register on each of the mux inputs. Flattening large memories, especially with variable writes and DEFAULT_FLATTEN, may cause Stratus HLS to run out of memory and crash due to the large amount of control being created.

With DPOPT_FLATTEN, variable array writes on the left-hand side of an assignment are implemented with a single, potentially large, custom combinational logic element that steers the written data to the selected element of the flattened array. This can substantially reduce Stratus HLS runtime and the risk of running out of memory at the possible expense of increased RTL area.

Note that the HLS_MAP_TO_REG_BANK is an alternative to DPOPT_FLATTEN for arrays with variable indexes.

See also:

- The " flatten_arrays" attribute in Synthesis Control Attributes
- Flattening Arrays in the *User G*uide

Example 1

```
SC_MODULE( saxo_light ) {
       .
       .
       .
   // shift regs (16. format)
   sc_int<16> l[TAPS]; // left
   sc_int<16> r[TAPS]; // right


   // coefficients (8.8 format)
   sc_int<16> cw[TAPS]; // tweeters
   sc_int<16> ct[TAPS]; // woofers
   // constructor
SC_CTOR( saxo_light ) {
     HLS_FLATTEN_ARRAY(l);
     HLS_FLATTEN_ARRAY(r);
     HLS_FLATTEN_ARRAY(cw);
     HLS_FLATTEN_ARRAY(ct);
     SC_CTHREAD( thread0, CLK.pos() );
     reset_signal_is(RSTN, 0 );
   }
};
```

The code above is a partial view of a class defined in C++. This class has four member variables that are arrays. In the example, each of these arrays is specified to be flattened.

Example 2

```
SC_MODULE (flat_example) {
.
  sc_uint<16> mymem[16]; //definition of a 16x16 memory variable
.
SC_CTOR(flat_example) {
.
  HLS_FLATTEN_ARRAY(mymem,DPOPT_FLATTEN );
.
for (i=0, i<16, i++) {
  HLS_UNROLL_LOOP(OFF);
  mymem[i] = i;
}
```

The code snippets above shows the definition of an array variable, a directive to flatten that variable, and a `for` loop to fill the memory, which will not be unrolled. Because the loop is not unrolled, the array indexes will not be constants, and DPOPT_FLATTEN may be helpful in making the array accesses more efficient.

HLS_INITIALIZE_ROM

A Stratus HLS directive.

Syntax

`HLS_INITIALIZE_ROM( type, array, file, [ format ] );`

Parameters

type
The type of words in the array without the const.

array
The target array in the design.

file
The path to and name of the data file.

format
The kind of data in the file: `HLS_BIN`, `HLS_DEC`, `HLS_FLT`, or `HLS_HEX`. (These identifiers are defined in the `HLS::ENUMS` namespace.)

Recommended placement

- Place in the module constructor for arrays that are data members of the module.
- Place in the module constructor for arrays that are (`const`) global variables.
- Place immediately after the array declaration for arrays that are automatic variables in functions.

Description

This directive may be used to initialize a ROM from a text file.
Files used to initialize ROMs with this directive have the following format (which is similar to that used with the Verilog `$readmemh` or `$readmemb` statements):

- Values in the file are separated by white space and/or new lines.
- Underscore (_) characters may be embedded in the values in the file.
- 'x' and 'z' characters are interpreted as '0' with a WARNING.
- C++ style comments may be embedded in the file as shown below.

```
/* block comment */
// end of line comment
```

- After underscore characters and comments have been removed, and after 'x' and 'z' characters are replaced with 0's, the remaining strings must be:

- Binary strings if `HLS_BIN` has been specified.
- C-format decimal strings if `HLS_DEC` has been specified.
- C-format floating point strings if `HLS_FLT` has been specified.
- C-format hexadecimal strings if `HLS_HEX` has been specified.

Here, "C-format" is defined as "acceptable to `scanf'`s `%d`, `%f`, and `%x` format specifiers, respectively."

The number of values in the initialization text file should exactly match the number of elements in the array.

To use `HLS_INITIALIZE_ROM` with the `HLS_BIN`, `HLS_DEC`, or `HLS_HEX` file type specifications, it must be legal to assign a long long int to a scalar variable of the type of the array elements. For example, to use:

```
<>const my_type my_array[16];
HLS_INITIALIZE_ROM( my_type, my_array, HLS_DEC,
"my_array_data.dec", "init my_array" );
```

it must be legal to write:

```
long long int value;
my_type element;
element = value;
```

Similarly, it must be legal to assign a double to a scalar variable of the type of the array elements when using `HLS_FLT`.

Note that the `HLS_INITIALIZE_ROM` directive affects the behavioral-level behavior as well as stratus_hls, so you must be sure that `HLS_INITIALIZE_ROM` is called dynamically before any reference to the array. In other words, if you can't place `HLS_INITIALIZE_ROM` in the constructor, we recommend that it is the first statement in the scope of the const array.

`HLS_INITIALIZE_ROM` may be used to initialize `const` class members or automatic (local) variables in addition to global variables. Problems with initializing const class members via aggregate assignment, the "traditional" approach, were the major stumbling block to the use of such variables, so the `HLS_INITIALIZE_ROM` directive has the benefit of enabling more modular designs.

`HLS_INITIALIZE_ROM` is also useful with multidimensional arrays (which don t exist in Verilog). Values will be read from the file and assigned to array elements sequentially, with the right most array index varying the fastest. For example, given the `sc_uint<8> ar[2][3][4]` array, `ar[0][0][0]` will get the first value in the file, `ar[0][0][1]` the second value in the file, `ar[0][0][3]` the fourth, `ar[0][1][0]` the fifth, and so on.

ROMs may also be initialized in the `const` array declaration with an assignment statement that lists the initial values. For information, see Inferring ROMs in the *User Guide*.

See also:

- Inferring Memories from Multi-Dimensional Arrays in the *User Guide*
- Initializing ROMs in the *User Guide*

Example

```
SC_MODULE( dut ) {
   const sc_int<15> my_rom_array[200];
   ...
   SC_CTOR( dut ) {
     HLS_INITIALIZE_ROM( sc_int<15>, my_rom_array, HLS_HEX, /
          "my_rom_array_data.memh", "init" );
   }
}
```

HLS_INLINE_MODULE

A Stratus HLS directive.

Syntax

`HLS_INLINE_MODULE;`

Equivalent Command

None

Parameters

None.

Recommended placement

Place immediately following the curly brace that encloses the class definition.

Description

The `HLS_INLINE_MODULE` directive identifies a module whose contents should be synthesized as part of the parent module that instantiates the module.

A class is called an inline module if it contains a `HLS_INLINE_MODULE` directive and is derived, either directly or indirectly, from `sc_module`.

Inline modules provide a convenient way of packaging up synthesizable behavior in a reusable class due to the following:

- The inline module can be connected to the outer module using standard SystemC port binding techniques.
- The inline module simulates as a separate `SC_MODULE` in a behavioral simulation.
- The contents of the inline module (`SC_THREAD`s, `SC_CTHREAD`s, component instantiations, and signals) are synthesized along with the parent module. The inline module is effectively flattened.
- The inline module can contain member functions that can either be called directly from threads in the parent module, or indirectly through `sc_port<sc_interface>` bindings. This makes it easy to integrate the inline module with behavioral code in a module that uses it.

An inline module should be incorporated into the system according to the following rules:

- There should be no `define_hls_module` command in `project.tcl` for an inline module. This means that no wrapper module will be created for an inline module, `stratus_hls` executions will not be performed separately for inline modules, and inline module names will not appear in other `project.tcl` constructs such as `define_sim_config`s.

- An inline module should be instantiated directly in its parent module. Wrapper modules are not used for inline modules.
- The inline module can be connected to its parent module either through SystemC port bindings, or by direct access of the parent module to members and functions in the inline module.
- An `sc_port<>` on either a parent module or a sibling inline module can be bound to a corresponding `sc_interface` on an inline module where the `sc_interface` has an arbitrary type.
- The bodies of the functions that provide the behavior for the inline module must be made visible to the module that instantiates it. This means that either the inline module's functions must be defined inline with its class definition, or the source file containing the function bodies must be included into the parent module's source file.

Reports will be produced for each `SC_CTHREAD` and `SC_METHOD` in each inline module that's instantiated in a synthesized module. The composite reports for the module will include the hardware used by the threads and methods in the inline modules.

Stratus HLS produces a single, integrated body of RTL code containing all of the hardware required to implement the main module and all inline modules it instantiates.

Component instantiations. Any explicit instantiation of a non-inline module in an inline module will result in an equivalent instantiation in the parent module.

An explicit instantiation of an inline module in an inline module will result in the inlining of both modules into the parent module. Instantiation of inline modules within inline modules is supported to an arbitrary depth.

Signals. An `sc_signal` declared in an inline module that 1) carries information either between threads or methods within the inline module, or 2) carries information between the inline module and its parent module will be instantiated in the parent module using a unique name.

Ports. Ports on an inline module may be bound to ports on a parent module, signals in a parent module, or signals in the inline module.

`sc_in<T>`, `sc_out<T>`, and `sc_port<IF>` ports are all supported on inline modules. `sc_port<IF>` ports are not supported on synthesizable non-inline modules.

Signal-level port bindings on an inline module result in a wired connection in the flattened model between the signal and port being connected.

If an `sc_port<IF>` is bound to a module that implements interface `IF`, calls to functions in interface IF through the port are handled like an ordinary function call by Stratus HLS. That is, the contents of the function is inlined into the calling thread or method.

Port inheritance rules. Inline modules are often used to add `SC_METHOD`s or `SC_CTHREAD`s to modular interfaces, so port inheritance rules are important for inline modules.

By default, ports declared in an inline module class are not inherited by the instantiating module.

If a single subclass of the inline module is a metaport class, then the metaport subclass is inherited by the instantiating module:

- In this case, the metaport subclass is placed on the wrapper module rather than the inline module.
- This technique also makes it easy to use the metaport subclass on its own for applications where the behavior of the inline module is not needed or is undesirable, such as for hierarchical port connections.

If a metaport is instantiated in an inline module, that metaport is never inherited by the parent of the inline module.

Directive propagation. If a directive appears in the constructor of an inline module, it will affect the threads and methods in that module, and potentially those in inline modules it instantiates.

For any thread or method in an inline module, directives will affect it according to the following precedence rules:

- In the thread or method function body.
- In the constructor of the inline module in which the thread or methods is started.
- In the constructor of a parent module that instantiates the inline module.

Example

```
SC_MODULE(M) {
   HLS_INLINE_MODULE;
   sc_in_clk clk;
   sc_in<bool> din;
   ...
};
```

See also:

- Using Modular Interfaces in the *User Guide*
- Combining HLS_METAPORT and HLS_INLINE_MODULE in the *User Guide*

HLS_INVERT_DIMENSIONS

A Stratus HLS directive.

Syntax

`HLS_INVERT_DIMENSIONS``(` *`array`* `);`

Equivalent Command

invert_dimensions

Parameters

`array`

The array to be processed.

Description

The `HLS_INVERT_DIMENSIONS` directive inverts the order of the dimensions of a multi-dimensional array for processing by Stratus HLS. A common application is one where inner dimensions will be constant, while outer dimensions will be variable. Using `HLS_INVERT_DIMENSIONS` makes it practical to use `HLS_SEPARATE_ARRAY`.

Example

```
sc_uint<8> a[16][16];
HLS_INVERT_DIMENSIONS( a );
HLS_SEPARATE_ARRAY( a );

for ( int i=0; i < 16; i++ ) {
  for ( int j=0; j < 16; j++ ) {
     HLS_UNROLL_LOOP();
     a[i][j] = f(i,j);
  }
}
```

In this example, since only the inner loop is unrolled, j will be a constant, while i will be a variable. In order to apply `HLS_SEPARATE_ARRAY`, the outer dimensions of the array must have constant indices. By using `HLS_INVERT_DIMENSIONS`, we achieve that requirement without re-writing the code.

HLS_MAP_ARRAY_INDEXES

A Stratus HLS directive.

Syntax

`HLS_MAP_ARRAY_INDEXES( array, option );`

Equivalent Command

map_array_indexes

Parameters

array

The name of the multi-dimensional array to be affected.

option

The type of mapping to use—either COMPACT or SIMPLE.

Recommended placement

- Place in the module constructor for arrays that are data members of the module.
- Place in the module constructor for arrays that are (`const`) global variables.
- Place immediately after the array declaration for arrays that are automatic variables in functions.

Description

The `HLS_MAP_ARRAY_INDEXES` directive allows you to control how multi-dimensional indices are mapped to memory addresses.

Stratus HLS normally follows this convention, which is equivalent to specifying the `COMPACT` setting for the `HLS_MAP_ARRAY_INDEXES` directive.

C++ defines that a multidimensional array access is converted to a memory offset through a sequence of multiplication and addition operations. For example:

```
sc_uint<8> myarray[2][3][4];
... myarray[x][y][z] ...
```

is implemented as if it had been written:

```
sc_uint<8> myarray[2*3*4];
... myarray[ ( ( ( x * 3 ) + y ) * 4 ) + z ]...
```

You can use the `SIMPLE` setting for `HLS_INVERT_DIMENSIONS` to instruct Stratus HLS to simplify the conversion of multi-dimensional indices into a memory address, replacing the multiplication and add operations with catenation.

For example, the following:

```
sc_uint<8> myarray[2][3][4];
```

HLS_INVERT_DIMENSIONS(myarray, SIMPLE);

```
... myarray[x][y][z] ...
```

will implement the design is if written:

```
sc_uint<8> myarray[2][4][4];
... myarray[ (sc_uint<1>) x ][ (sc_uint<2>) y ][ (sc_uint<2> z ] ...
```

and calculate the one-dimensional index as:

```
... myarray[ ( (sc_uint<1>) x, (sc_uint<2>) y, (sc_uint<2>) z ) ] ...
```

That is, the individual index values are trimmed or padded to the "proper" width of an index of that dimension and catenated to form the final index. This is likely to reduce the combinational area of your design at the possible expense of unused memory locations.

The `SIMPLE` index mapping can also be applied from the command line. For more information, see "--simple_index_mapping" in the Reference Guide.

Note: Use of this directive can lead to incorrect RTL if the behavior uses indices that are larger than those of the declaration. For example:

```
... myarray[0][0][17] ...
```

is legal C++, equivalent to:

```
... myarray[1][1][1] ...
```

However, when `SIMPLE` index mapping is specified, Stratus HLS will trim the 17 bits to two bits, producing a reference equivalent to:

```
... myarray[0][0][1] ...
```

Stratus HLS will produce a warning in situations where the width of the index is adjusted. However the following:

```
... myarray[0][(sc_uint<2>) 3][0] ...
```

has the same problem, in that the middle index is "out of range," but Stratus HLS will not produce a warning message as the expression is of the correct width as written.

Another option

You can also get much of the benefit of specifying `SIMPLE` index mapping without the risk of creating a simulation mismatch by simply ensuring that all of your array declarations (excluding the leftmost) are powers of two. That is, in our example above, change the declaration of `myarray` from:

```
sc_uint<8> myarray[2][3][4];
```
to:
```
sc_uint<8> myarray[2][4][4];
```

and leave the index mapping mode at the default setting of `COMPACT`. Stratus HLS will implement the C++ standard address calculations as described above, but will find that it can replace the multiplications with constant shift operations at a considerable savings in combinational area.

Inverting dimensions

You can also use the `HLS_MAP_ARRAY_INDEXES` directive to invert the dimensions of a multi-dimensional array. For example, if you have an array declared and accessed like this:

```
sc_uint<8> a[8][3];
...
x = a[i][j];
```

it will be effectively the same as if it had been:

```
sc_uint<8> a[3][8];
...
x = a[j][i];
```

This can reduce the complexity of indexing similar to the `SIMPLE` option, but without wasting storage space. It can also be used with the `HLS_SEPARATE_ARRAY` directive to separate arrays by inner dimensions rather than outer dimensions.

You can specify more than one `HLS_INDEX_MAPPING` directive for the same array as long as one directive specifies either `SIMPLE` or `COMPACT` mapping.

See also:

- Mapping Array Indices to Memory Addresses in the *User Guide*
- Inverting Array Dimensions in the *User Guide*
- HLS_SEPARATE_ARRAY
- "simple_index_mapping" in Synthesis Control Attributes

HLS_MAP_TO_MEMORY

A Stratus HLS directive.

Syntax

`HLS_MAP_TO_MEMORY``( < `*`array`*` > [, < `*`mem_name`*` > [, < `*`clk`*` >] [, < `*`port_num`*` >, < `*`port_num`*` >, ... ] ] );`

Equivalent Command

map_to_memory

Parameters

`array`

A reference to the array to be bound.

`mem_name`

A string giving the name of the memory to which the array will be bound. The memory must be present in a library referenced in the Stratus HLS project. Optional. If omitted, the array is mapped to a memory that is selected by Stratus HLS. The `mem_name` parameter should be specified as a quoted string, and not a C++ identifier. For example, “RAM32X8” rather than RAM32X8. In cases where a C++ type name is the only thing available, such as in a module templated on the memory type, the `HLS_TYPE_TO_STR()` macro can be used. See Example 3.

`clock`

Specifies a clock that should be connected to memory instantiations. If no clock is specified, the clock for the accessing SC_CTHREAD is used automatically.

`port_num`

Specifies a subset of memory access ports that can be used by the accessing thread or module. Port indexes start at 1, and the order is defined by the order of ports in the memory editor in Stratus IDE. Up to 4 ports may be specified. If no ports are specified, all ports are accessible.

Recommended placement

- Place in the module constructor for arrays that are data members of the module.
- Place in the module constructor for arrays that are (`const`) global variables.
- Place immediately after the array declaration for arrays that are automatic variables in functions.
- Place in the body of a thread for which specific memory ports must be allocated. In this case, there may be multiple directives for the same array in different threads, each specifying different ports.

Description

The `HLS_MAP_TO_MEMORY` directive allows you to specify that an array is to be mapped to a memory, and to optionally specify which type of memory to use, which memory ports to use, and which clock should be bound to the memory. This is useful when a global array flattening option, like `--flatten_arrays=all` is specified, but it is desired to map a specific array to a memory. It also provides a way to make sure the memory is of the desired area and dimension. The specified memory type must be in one of the parts libraries specified in an `hls_lib` command for your project.

During behavioral synthesis, the impact of array binding directives can be seen in the array disposition reports that immediately follow the allocation report.

If the memory has the "Half Speed" or "Quarter Speed" options selected, you must also specify an alternate clock to connect to the memory in the RTL model. This is done by specifying the clock port as the optional third parameter to `HLS_MAP_TO_MEMORY`.

The `HLS_MAP_TO_MEMORY` directive overrides the global `--flatten_arrays` setting for the arrays on which it is applied.

The `HLS_MAP_TO_MEMORY` can be used to control which memory ports are used by an individual thread or module. This is useful in the following two scenarios:

1. When multiple threads in the same module access the same array, and that array is mapped to a multi-port memory, you may wish to limit each thread to use a separate port. To accomplish this, an `HLS_MAP_TO_MEMORY` directive should be placed in the body of each thread, and a different port number should be specified for each thread.
2. When multiple modules access the same array using the *external arrays* feature, you may wish to restrict each module to accessing a specific port on a multi-port memory. In this case, an `HLS_MAP_TO_MEMORY` directive should be placed in the constructor of each sub-module, with a different port specified in each module.

Note that explicit port specifications are not necessary if the accessing threads or modules naturally align with the capabilities of the ports. For example, if the memory has one write-only-port and one read-only port, and if one thread or module contains only writes, while the other contains only reads, there is no need to specify ports explicitly with `HLS_MAP_TO_MEMORY`: Stratus HLS will automatically access only the required ports.

For multiple clock domain designs in which threads with different clocks access the same memory, memory ports should be explicitly allocated to each thread using the port index arguments. Stratus HLS will automatically connect the appropriate clock to the appropriate memory port based on the clocks used by the accessing threads. If threads from with different clocks access the same memory port, Stratus HLS will issue an error. Note that the *`clk`* argument should *not* be used in this kind of design unless access to the memory is made from only one clock domain.

See also:

- Binding Arrays to Memories in the *User Guide*
- The "`flatten_arrays`" attribute in Synthesis Control Attributes

Example 1

This example binds the `my_ram4x8` array to a `ram4x8` memory.

```
SC_MODULE( dut ) {
   sc_uint<8> my_ram4x8[4];
   ...
   SC_CTOR( dut ) {
      HLS_MAP_TO_MEMORY( my_ram4x8, "ram4x8" );
   }
}
```

Example 2

This example causes array `my_ram4x8` to be bound to a memory that is selected by Stratus HLS.

```
SC_MODULE( dut ) {
   sc_uint<8> my_ram4x8[4];
   ...
   SC_CTOR( dut ) {
      HLS_MAP_TO_MEMORY( my_ram4x8 );
   }
}
```

Example 3

Because the memory name is specified as a string, using `HLS_MAP_TO_MEMORY` in a templated class where the memory type is specified as a template parameter requires special attention. This example shows how to use the `HLS_TYPE_TO_STR()` macro to solve the problem.

```
template <typename MEM>
class my_ip {
   MEM::data_type arr[MEM::mem_size];
   ...
   my_ip {
      HLS_MAP_TO_MEMORY( arr, HLS_TYPE_TO_STR(MEM) );
   }
}
```

The `HLS_TYPE_TO_STR()` macro accepts a type name and returns a string constant containing the name of the type.

Example 4

This example uses a half-speed clock for the mem array because it is mapped to a memory model that has the half-speed option selected.

```
SC_MODULE( dut ) {

   sc_in_clk clk;  // Clock for SC_CTHREAD
   sc_in_clk hclk; // Clock for memory.
   sc_uint<8> mem[1024];
   ...
   SC_CTOR( dut ) {

      SC_CTHREAD( thread, clk.pos() );
      …
      HLS_MAP_TO_MEMORY( mem, “HALF_SPEED_RAM”, hclk );
   }
}
```

Example 5

This example shows how ports can be explicitly allocated to sub-modules that access an external array. `HLS_MAP_TO_MEMORY` is specified in the constructor of each module, so the mapping of port applies only to accesses within that module.

```
  SC_MODULE(sub1) {
    sub1(   sc_module_name& name,
            int             arr[1024] )

         // Connection to array member.
         : m_arr( arr )

    {
        SC_CTHREAD( write_proc, clk.pos() );

         // Mapping to port #1.
        HLS_MAP_TO_MEMORY( m_arr, "RAM1024X32", 1 );
    }
  };

   SC_MODULE(sub2) {
    ...
    sub2(   sc_module_name& name,
            int             arr[1024] )
        // Connection to array member.
        : m_arr( arr )

    {
        SC_CTHREAD( read_proc, clk.pos() );


        // Mapping to port #2.
        HLS_MAP_TO_MEMORY( m_arr, "RAM1024X32", 2 );
    }
};
```

HLS_MAP_TO_REG_BANK

A Stratus HLS directive.

Syntax

HLS_MAP_TO_REG_BANK(*array*);

Equivalent Command

map_to_reg_bank

Parameters

array
The name of the array to be bound.

Recommended placement

- Place in the module constructor for arrays that are data members of the module.
- Place immediately after the array declaration for arrays that are automatic variables in functions.

Description

The `HLS_MAP_TO_REG_BANK` directive allows you to specify that an array is to be mapped to a dedicated bank or registers.

The `HLS_MAP_TO_REG_BANK` directive overrides the global `--flatten_arrays` setting for the arrays on which it is applied.

See also:

- Binding Arrays to Auto-Generated Register Banks in the *User Guide*
- " flatten_arrays" in Synthesis Control Attributes

Example

```
SC_MODULE( dut ) {
   sc_uint<8> my_regs[4];
   ...
   SC_CTOR( dut ) {
      HLS_MAP_TO_REG_BANK( my_regs );
   }
}
```

This example binds the `my_regs` array to a dedicated register bank component.

HLS_METAPORT

A Stratus HLS directive.

Syntax

`HLS_METAPORT`

Equivalent Command

None

Parameters

None

Recommended placement

Place immediately following the curly brace that encloses the class definition.

Description

The `HLS_METAPORT` directive identifies a module containing other ports that should be treated as though it were a single port.

A class containing the `HLS_METAPORT` directive is called a metaport class. A metaport contains members that are ordinary SystemC ports (sc_in, sc_out, or sc_port).

When a metaport is instantiated on an sc_module, it is treated as a single, modular unit that represents an interface:

- Only the metaport class, and not the individual ports in the metaport, are placed on the wrapper module.
- A single binding call is made to connect all of the metaport's member ports to the members of a compatible channel class. Details of this binding are specified in a member function of the metaport.

Stratus HLS inherits the member ports of a metaport onto the RTL module. Bindings between the metaport and the individual RTL ports are handled in the wrapper module.

Port inheritance rules: By default, all member ports in a metaport are inherited by the module class that instantiates it.

If a class contains both the `HLS_METAPORT` and `HLS_INLINE_MODULE` directives, then ports are inherited by default. Note that this is not the case when `HLS_INLINE_MODULE` alone appears in a class.

Inheritance for individual ports or ranges of ports can be specified using the `HLS_EXPOSE_PORT` directive.

For each member port of a metaport, a port is added to the RTL module the naming convention

`<metaport_name>_<member_port_name>` where `<metaport_name>` is the name of the metaport instance in the parent module, and `<member_port_name>` is the name of the port in the metaport.

If the datatype of any member port is a struct or class type, then a port is added for each member of the struct or class. In this case, `<member_port_name>` is replaced for each port with `<struct_name>_<field_name>`, where `<struct_name>` is the name of the struct or class typed member and `<field_name>` is the name of a field in the struct. If `<field_name>` has a struct or class type, then this convention continues recursively.

Wrapper modules always contain metaport instances, not individual port declarations. The wrapper module automatically handles a number of metaport

connection tasks:

- When an RTL or gate-level module is instantiated within the wrapper, the individual ports in the metaports are bound to the corresponding signals in the
- flattened module.
- When a behavioral model is instantiated within the wrapper, the metaports are bound together.
- If there are ny un-exposed ports on the metaport (`HLS_EXPOSE_PORT( on_off, port_name )`), those ports in the metaport on the wrapper are bound to dummy signals to avoid unconnected port errors from SystemC.

Modular port replacement: Because a metaport encapsulates an entire interface in a single class, different implementations can be substituted by replacing the metaport class. This is feasible as long as the set of classes that will be interchanged meets the following requirements:

- Each class must support the same binding syntax for connection in parent modules. It is normal for each version of the metaport class to bind to a different channel class, but the syntax used to perform the binding must be consistent.
- Each class must support the same API. Any calls made into the modular interface from a thread or method within the accessing module must be the same amongst the classes. The semantics of the functions in the API must also be consistent.

The `io_config` project command can be used to control which interfaces are used in which configurations.

Applications for modular port replacement include:

- Switching between TLM and PIN level interfaces.
- Changing the bandwidth of a pin-level interface.

See also, Using Modular Interfaces in the *User Guide*.

HLS_NAME

A Stratus HLS directive.

Syntax

```
HLS_NAME ( name );
```

Equivalent Command

None

Parameters

`name`

specifies a name to the object within which the directive appears. This name can be used to identify the object from Tcl.

Recommended placement

Place inside a curly-brace region in C++ source code.

Description

The `HLS_NAME` directive is an alternative to using a C++ label before the curly-brace region or loop.

HLS_PIPELINE_LOOP

A Stratus HLS directive.

Syntax

`HLS_PIPELINE_LOOP``(` *`pipe_type [, _ii ] [, name ]`*`)`

or

`HLS_PIPELINE_LOOP``( name )`

Equivalent Command

pipeline_loops

Parameters

`pipe_type`

The pipelining type (supported options are: `HARD_STALL` or `SOFT_STALL`)

`pipeline_ii`

Default value is 1 to *ii*

`name`

The name of the pipeline for reporting purposes.

Recommended placement

Place immediately following the curly brace that encloses the pipelined loop.

Description

The `HLS_PIPELINE_LOOP` directive tells Stratus HLS to pipeline its enclosing loop. The *name* is a string that Stratus HLS can use to refer to the pipeline during reporting; this name must be unique to the project. The *`HARD_STALL`* *type* specifies that the later stages of the loop will be inactive as the loop fills and the earlier stages of the loop will be inactive as the loop drains. The *`SOFT_STALL type`* specifies that the loop should be pipelined to support both pipelining filling during an output stall and pipeline draining during in input stall.

If no initiation interval is specified, it defaults to 1.

Not all loops can be successfully pipelined. For more information, see Coding considerations for pipelining in the *User Guide*.

Stratus HLS automatically unrolls loops within pipelines if the number of iterations can be determined. No loop unrolling directives are necessary.

If your pipelined design employs dual-port memories, you must also define a `HLS_CONSTRAIN_ARRAY_MAX_DISTANCE` directive. See

HLS_CONSTRAIN_ARRAY_MAX_DISTANCE.

> The stall propagation between the pipeline stages is implemented using a combinational path. If the pipeline has many stages, the length of this path may make it difficult to close timing in logic synthesis.

See also:

- Pipelining loops in the *User Guide*
- HLS_UNROLL_LOOP

Example 1

```
while (1) {
  HLS_PIPELINE_LOOP( HARD_STALL, 2,"pipeline" );
  for ( i = 0; i < 8; i++ ) {
      // This loop is unrolled because it is in a pipelined loop.
  }

  int n_iters = in1.read();
  for ( int j=0; j < n_iters; j++ ) {
      // This loop cannot be unrolled because it does not have a
      // statically determinably number of iterations, so it will prevent pipelining.
  }

  while ( in1.read() != 0 ) {
      // This loop will be interpreted as a pipeline stall.
      wait();
   }
}
```

This example highlights Stratus HLS loop behavior within pipelined blocks. Stratus HLS will automatically unroll the `for` loop within *pipeline*. Stratus cannot unroll the second for() loop because it does not have a fixed number of iterations. This will prevent pipelining. The while() loop will be interpreted as a pipeline stall, and is supported in a pipeline.

Neither of the loops in this example are affected by the `unroll_loops` flag.

Example 2

```
while (1) {
  HLS_PIPELINE_LOOP( "pipeline" );
  ...
```

In this example, the *pipe_type* and *ii* parameters are defaulted to `HARD_STALL` and 1, respectively.

Additional examples can be found in Pipelining Loops in the User Guide.

HLS_PRESERVE_IO_SIGNALS

A Stratus HLS directive.

Syntax

`HLS_PRESERVE_IO_SIGNALS``( [` *`label`* `] );`

Equivalent Command

preserve_io_signals

Parameters

`label`

A text string used for reporting purposes.

Recommended placement

Place in the body of the `SC_METHOD` or `SC_CTHREAD` whose output ports and signals are to be preserved. Alternately, the directive may be placed in the module constructor to apply it to all threads and methods in the module.

Description

The `HLS_PRESERVE_IO_SIGNALS` directive prevents sc_signals variables written by a particular `SC_METHOD` or `SC_CTHREAD` from being optimized away when no other process reads those signals. It is useful in applications where a post-processing step will connect something to the signals rather than a a process in the same SystemC model. This is sometimes the case when using the Extra Ports feature of Stratus HLS memory models. The `HLS_PRESERVE_IO_SIGNALS` directive also prevents registers associated with both sc_out and sc_signal variables from being shared, or sliced to remove constant bits. Using this directive will produce a simpler mapping between sc_signal and sc_out in SystemC and registers in Verilog.

The `HLS_PRESERVE_IO_SIGNALS` directive sets the `default_preserve_io` behavior attribute and has the same precedence as that attribute. It can be overwritten by `set_attr default_preserve_io $behavior` in post_elab_tcl.

`HLS_PRESERVE_IO_SIGNALS` is an alternative to specifying individual signals with the `HLS_PRESERVE_SIGNAL` directive when a large number of signals needs to be preserved. However, it can have unintended side effects since signals that are not specifically named can be affected, so it should be used with care.

See also:

- HLS_PRESERVE_SIGNAL
- Extra Ports Tab in the *User Guide*

HLS_PRESERVE_SIGNAL

A Stratus HLS directive.

Syntax

`HLS_PRESERVE_SIGNAL``(` *`signal_id`* `[ ,` *`always`* `]);`

Equivalent Command

preserve_signals

Parameters

`signal_``id`

The name of an sc_signal or sc_out variable that is to be preserved.

`always`

If true, the signal will be preserved even if it lacks both a reader and a writer. Defaults to true.

Recommended placement

Place in the module constructor of the SC_MODULE that instantiates the signal.

Description

The `HLS_PRESERVE_SIGNAL` directive will preserve an sc_signal through the synthesis process. If the always parameter is *true*, the signal will be preserved even if it does not have both a reader and a writer in the BEH model. If always if *false*, the signal will be retained in favor of other signals asynchronously connected to it, but if it does not have both a reader and a writer in the design, it will be removed. Signals that are neither read nor written will *never* be preserved, even if HLS_PRESERVE_SIGNAL is used on them: the signal must be referenced somewhere in the model other than the directive. The `HLS_PRESERVE_SIGNAL` directive can also be used with with sc_out and sc_signal variables to prevent the register associated with it from being shared, or sliced to remove constant bits. Using this directive will produce a simpler mapping between sc_signal and sc_out in SystemC and registers in Verilog.

The `HLS_PRESERVE_SIGNAL` directive sets the `preserve_io` behavior_io attribute, and has the same precedence as that attribute. It can be overwritten by a `set_attr preserve_io $beh_io` in post_elab_tcl.

`HLS_PRESERVE_SIGNAL` is useful in several applications:

- If a signal will be connected at the RTL level to code inserted by a post-synthesis process, it needs to be retained in the RTL, and not have its name changed. Without `HLS_PRESERVE_SIGNAL`, such a signal, and logic connected to it, would be removed.
- For design debugging purposes, forcing a signal to remain with its name unchanged can make waveforms more understandable. Without `HLS_PRESERVE_SIGNAL`, if there is another signal that is asynchronously connected, Stratus HLS may eliminate the wire.

- Stratus may share registers for output ports and signals if they have identical logic. It may also *slice* these registers to avoid storing constant bits in them. This can make debugging, and some downstream processes awkward, and the `HLS_PRESERVE_SIGNAL` directive will prevent this.
- When using the memory Extra Ports feature, signals connected to memory extra ports may also be connected to logic in the BEH model. Without `HLS_PRESERVE_SIGNAL`, it will appear that the signal lacks either readers or writers, and it will be removed from the RTL.

See also:

- esc_trace
- Extra Ports Tab in the *User Guide*

Example

```
SC_MODULE(dut) {
  sc_in_clk clk;
  sc_in<bool> rst;
  ...
  sc_uint<8> mem[1024]; // Memory with extra port.
  sc_in< bool > NEXP;// Input inverted before extra port.
  sc_signal< bool > EXP; // Signal bound to extra port on memory.

  SC_CTOR(dut) {
    ...
    // Method that will write to extra port.
    SC_METHOD(inv_EXP);
    sensitive << NEXP;

    // Directive to prevent removal of logic driving EXP.
    HLS_PRESERVE_SIGNAL(EXP);
  }
...
  void inv_EXP() {
    EXP.write( !NEXP.read() );
  }
  ...
```

In this example, presume that mem is mapped to a memory with an extra port that is bound to the `EXP` signal. The `inv_EXP` method inverts the `NEXP` input and writes it to `EXP`, which will be bound to the extra port. However, since it appears that no other thread reads `EXP`, Stratus HLS may optimize away that logic. The `HLS_PRESERVE_SIGNAL(EXP)` directive prevents that, and preserves the logic writing to the extra port.

HLS_REMOVE_CONTROL

A Stratus HLS directive.

Syntax

`HLS_REMOVE_CONTROL( [ type ] [ , label ] );`

Equivalent Command

remove_control

Parameters

type

Types are `ON` (enable) or `OFF` (disable) the directive on the block.The default is `ON`.

label

An optional message used for reporting purposes.

Recommended placement

Place immediately following the curly brace that encloses the affected block of code.

Description

The `HLS_REMOVE_CONTROL` directive controls whether Stratus HLS represents conditionality in the design as branches in the FSM or as characteristics of the datapath. By default, Stratus represents conditionality as characteristics of the datapath wherever it can do so. Certain types of operations, such as operations with side effects, can cause the conditionality to be represented as branches in the FSM instead.

`OFF` can be applied by the user to direct Stratus to represent conditionality as branches in the FSM for all control statements that enclose this directive.

`string` is an optional message used for reporting purposes.

The HLS_REMOVE_CONTROL directive should be placed at the beginning of the code block.

Example

```
if ( cond ) {
  HLS_REMOVE_CONTROL(OFF,"string");
  ...
}
```

HLS_SCHEDULE_REGION

A Stratus HLS directive.

Syntax

```
HLS_SCHEDULE_REGION( ( flags , II, name );

HLS_SCHEDULE_REGION( name );

HLS_SCHEDULE_REGION();
```

Equivalent Command

schedule_region

Parameters

flags

Specifies options. Multiple options may be ORed together:

- **`REGION_DEFAULT`**: (has value 0).
- **`NO_CHAIN_IN`**: Causes registered inputs on the scheduled region.
- **`NO_CHAIN_OUT`**: Causes registered outputs on the scheduled region.

II

Specifies the initiation interval. If the `II` is >0 the scheduled region is pipelined. An `II`>1 is supported only if the latency of the scheduled region is constrained to <= `II` using `HLS_CONSTRAIN_REGION`. If the `II` is unspecified, the scheduled region has the same initiation interval as the loop from which it is accessed, or 1 if a latency constraint <= `II` is not specified. An `II` less than the `II` of the accessing thread may be specified.

name

A string that will be used to name the module created for the scheduled region. If a name is not specified, the name of the function containing the scheduled region will be used if the scheduled region covers the entire function. A default name will be constructed otherwise.

Recommended placement

Place immediately following the curly-brace that encloses the code block to be scheduled separately.

Description

The `HLS_SCHEDULE_REGION` directive provides a way to have Stratus HLS segregate part of a design's functionality for separate processing with minimal modification to the source code structure. The operations in the region are processed by high-level synthesis as if they were in a separate SC_THREAD, producing an independent finite state machine in RTL. This FSM can implement the operations with a fixed latency, with a variable latency, or in a pipelined manner. `HLS_SCHEDULE_REGION` allows most constructs supported by Stratus HLS.

Similar to the `HLS_DPOPT_REGION` directive, `HLS_SCHEDULE_REGION` can improve scalability and tool runtimes, and can produce better quality results. `HLS_SCHEDULE_REGION` is often used in place of `HLS_DPOPT_REGION` when the region has any of the following attributes:

- Contains I/O or modular interface accesses
- Contains accesses to arrays mapped to memories
- Has a variable latency
- Has internal resource sharing opportunities
- Has a long latency
- Has internal loops that you do not wish to unroll

Considerations for Using HLS_SCHEDULE_REGION

The following are some of the points to consider when using a scheduled region:

- **Pipelining**: For `HLS_SCHEDULE_REGION`, whether the implementation is pipelined can be controlled with the initiation interval parameter of the directive. If an initiation interval is specified in the `HLS_SCHEDULE_REGION` directive, it must match the initiation interval of the accessing pipelined loop. If no initiation interval is specified and the region is accessed from a pipelined loop, it is implicitly given the same initiation interval as the pipeline. If there are loop-carried dependencies in the accessing pipelined loop that pass through the scheduled region, as long as there are no other dependencies elsewhere in the same pipeline, the LCD will be moved into the FSM for the `HLS_SCHEDULED_REGION` so that the latency of the region can be longer than its initiation interval. All scheduled regions have a latency of at least 1 cycle.
- **I/O**: Scheduled regions may contain I/O statements, including I/O statements in loops and conditionals, as long as all invocations of the scheduled region specify the same ports and/or signals. Invocations of scheduled regions that contain I/O will not be executed concurrently with other scheduled regions that contain I/O. They will execute sequentially in the order in which they are invoked. Scheduling also ensures that the calling thread does not access a port or signal during the execution of a scheduled region that accesses the same port or signal.
- **Memory Access**: Scheduled regions may contain access to arrays mapped to memories, including accesses in loops and conditionals, as long as all invocations of the scheduled region specify the same arrays. Scheduled regions may not contain access to arrays mapped to REG_BANKs. Scheduling ensures that a scheduled region does not access a memory

concurrently with the calling thread, or with any other scheduled region.

- **Sharing of instances**: Within a scheduled region, instances of datapath components may be shared just like any thread. However, instances of datapath components within a scheduled region will not be shared with any thread or any other scheduled region.
- **Handling of control flow**: Using `HLS_SCHEDULE_REGION`, control flow is treated just as it is in an SC_CTHREAD. Loops are only unrolled only if a synthesis control setting is made to cause them to be unrolled. Also, control flow can be left in place to achieve variable latency.
- **Constraining timing and latency:** The `HLS_CONSTRAIN_REGION` directive can be used to control the latency and timing of an `HLS_SCHEDULED_REGION`. This optional directive should be placed in the region as the `HLS_CONSTRAIN_REGION` directive. For more information, see HLS_CONSTRAIN_REGION.

  If no `HLS_CONSTRAIN_REGION` has specified, or if values are omitted from the directive, the following assumptions are made:

  - The input delay is clk->q, making the entire clock cycle available to the first stage of logic for the region.
  - There is no maximum delay. Clock cycles will be added as required, and the final stage can contain any amount of logic.
  - The latency will be the shortest achievable latency, and at least 1 cycle.

See also:

- HLS_DPOPT_REGION
- HLS_CONSTRAIN_REGION

Restrictions on Using HLS_SCHEDULE_REGION

The following types of regions cannot be implemented with `HLS_SCHEDULE_REGION`:

- Regions in functions called from module constructors.
- Regions in inline modules (the `HLS_INLINE_MODULE` directive).
- Regions in metaports (the `HLS_METAPORT` directive).
- Regions in functions whose return type is pointer to user defined class or struct.
- Regions in functions that have parameters whose type is “pointer to pointer”.
- Regions in functions called from SC_METHODs.

- Regions within an `HLS_DPOPT_REGION`.
- Regions that contain other scheduled regions, or call functions containing other scheduled regions. That is, hierarchical scheduled regions are not supported.
- Regions in functions whose parameters include a reference to part of an array.
- Regions that contain protocol that are accessed from a protocol region within a pipelined loop.
- Regions that perform I/O in a protocol region, but that do not contain at least 1 wait.
- Regions with a specified initiation interval that does not match the initiation interval of an accessing pipelined loop.
- A region with variable latency, or that can stall, that is called from within a pipelined loop that can stall. That is, either the scheduled region can stall, or the accessing thread can stall, but not both of them.
- Regions in functions that access unflattened arrays or ports where different calls to the function specify different arrays or ports. That is, if there are multiple calls to a function containing a scheduled region, all calls must specify the same unflattened arrays or ports.
- Regions that access unflattened arrays or write to ports and are accessed from a pipelined loop where the same arrays or ports are written by either the accessing thread, or another scheduled region in the same pipeline. That is, a scheduled region in a pipeline must have exclusive access to the unflattened arrays and ports it accesses.
- Regions in functions that access separated arrays, except where the array is declared within the function.
- Regions in functions that access arrays mapped to register banks, except where the array is declared within the function.
- Regions in functions that are called from more than one thread.

HLS_SEPARATE_ARRAY

A Stratus HLS directive.

Syntax

HLS_SEPARATE_ARRAY(*array*, [, *ndims*]);

Equivalent Command

separate_arrays

Parameters

array

The name of a multi-dimensional array

ndims

The number of outer dimensions that will be separated. Optional. Defaults to 1.

Recommended placement

- Place in the module constructor for arrays that are data members of the module.
- Place immediately after the array declaration for arrays that are automatic variables in functions.

Description

The `HLS_SEPARATE_ARRAY` directive specifies that a multi-dimensional array should be processed as an *array of arrays* mapped to multiple hardware resources instead of being mapped to a single hardware resource with a single dimension. The directive is useful for both arrays mapped to memories, and arrays mapped to register banks, but is, in general not applicable to flattened arrays. By associating a multi-dimensional array with multiple hardware resource, you can increase the bandwidth for access.

The `HLS_SEPARATE_ARRAY` directive specifies that the outer dimensions of a multi-dimensional array should be used to create discrete arrays for the inner arrays. For example

```
sc_uint<8> a[2][16]
HLS_SEPARATE_ARRAY( a );
...
for ( i=0; i < 16; i++ ) {
   rslt += a[0][i] * a[1][i];
}
```

If `a` is mapped to a memory, 2 separate memory instances will be allocated, one for `a[0]`, and one for `a[1]`, each 16-words wide. This allows the memory reads to be done concurrently.

Prior to version 4.3.0, Stratus requires that the indexes that select amongst the separate arrays be constants. Starting with version 4.3.0 the indexes that select amongst the separate arrays can be either constant or variable (non-constant).

By default, `HLS_SEPARATE_ARRAY` will use the outermost dimension to determine how arrays are separated. So, the following declaration:

```
sc_uint<8> a[2][3][4];
HLS_SEPARATE_ARRAY( a );
```

would be processed as 2 arrays, each containing 12 8-bit words.

This can be changed by specifying a value for the second parameter to `HLS_SEPARATE_ARRAY`. This parameter specifies how many outer dimensions should be used to separate the array. It defaults to 1. If we change the example like this:

```
sc_uint<8> a[2][3][4];
HLS_SEPARATE_ARRAY( a, 2 );
```

The outer 2 dimensions will specify how to separate the array, and it would be processed as 6 arrays, each containing 4 8-bit words.

You can also combine `HLS_SEPARATE_ARRAY` with `HLS_INVERT_DIMENSIONS(a)` to specify that the inner dimensions should be used to separate the arrays. So if we have this:

```
sc_uint<8> a[2][3][4];
HLS_SEPARATE_ARRAY( a, 2 );
HLS_INVERT_DIMENSIONS( a );
```

it would be equivalent to writing this:

```
sc_uint<8> a[4][3][2];
HLS_SEPARATE_ARRAY( a, 2 );
```

which would result in 12 arrays, each containing 2 8-bit words.

See also:

- Separating Multidimensional Arrays in the *User Guide*
- HLS_INVERT_DIMENSIONS

Example #1

```
sc_uint<8> a[2][3][4];
HLS_SEPARATE_ARRAY( a );
```

Gives 2 arrays, each 12 X 8-bit words.

Example #2

```
sc_uint<8> a[2][3][4];
HLS_SEPARATE_ARRAY( a, 2 );
```

Gives 6 arrays, each 4 X 8-bit words.

Example #3

```
sc_uint<8> a[2][3][4];
HLS_SEPARATE_ARRAY( a );
HLS_INVERT_DIMENSIONS(a)
```

Gives 4 arrays, each 6 X 8-bit words.

Example #4

```
sc_uint<8> a[2][3][4];
HLS_SEPARATE_ARRAY( a, 2 );
HLS_INVERT_DIMENSIONS( a )
```

Gives 12 arrays, each 2 X 8-bit words.

HLS_SET_ARE_BOUNDED

A Stratus HLS directive.

Syntax

`HLS_SET_ARE_BOUNDED` `(`*`first_signal`*`,` *`last_signal`*`, [`*`label`*`]);`

Equivalent Command

None

Parameters

`first-signal`

The first port or signal in the module that should be made bounded.

`last-signal`

The last port or signal in the module that should be made bounded.

`label`

An optional string to be associated with the directive.

Recommended placement

Place either in constructor of the class in which the values are declared, or in the bodies of each thread that accesses the values.

Description

The `HLS_SET_ARE_BOUNDED` directive has the same effect as the `HLS_SET_IS_BOUNDED` directive, except it can be used to apply to a set of port or signal declarations without naming all of them. Given a pair of port or signal names from an `SC_MODULE`, the directive will apply to them, and any other ports or signals that are declared between them in the same `SC_MODULE`.

The `HLS_SET_ARE_BOUNDED` directive fills a similar role to the `HLS_ASSUME_STABLE` directive with several differences:

- It can be used with both inputs and outputs.
- The `HLS_ASSUME_STABLE` directive is specified within a C++ block in which the input reads can safely be scheduled, while the location of the `HLS_SET_ARE_BOUNDED` directive is not important since the limits are separately specified by boundaries.
- Because the `HLS_SET_ARE_BOUNDED` directive is not block-specific, it is more suitable for use with modular interfaces because the directive can be included within the modular interface source code, and need not appear in the thread accessing the interface.

The `HLS_SET_ARE_BOUNDED` form of the directive provides a convenience for specifying the directive to several inputs or outputs at one time so long as they are declared together.

The boundary mechanism is also used to control the limits on movement of explicit memory reads and writes. For a general discussion of boundaries, see Limits on movement of external memory operations in the *User Guide.*

See also:

- HLS_SET_IS_BOUNDED
- HLS_ASSUME_STABLE
- HLS_SET_OUTPUT_DELAY
- Specifying stable inputs in the *User Guide*

Example

```
SC_MODULE(dut) {
 sc_in<bool> in_valid;
 sc_in<bool> out_ready;
 sc_in< sc_uint<8> > in_data1;
 sc_in< sc_uint<8> > in_data2;
 sc_in< sc_uint<8> > in_data3;
 sc_out< sc_uint<8> > out_data1;
 sc_out< sc_uint<8> > out_data2;
 sc_out< sc_uint<8> > out_data3;

SC_CTOR(dut) {
  // Specify that the in_data2, in_data3, out_data1, and out_data2 will
  // obey boundary restrictions, but that in_data1 and out_data3 will not.
  HLS_SET_ARE_BOUNDED( in_data2, out_data2 );
}
```

HLS_SET_CLOCK_PERIOD

A Stratus HLS directive.

Syntax

HLS_SET_CLOCK_PERIOD(***signal_id, period***);

Equivalent Command

set_clock_period

Parameters

signal_id

The name of an `sc_in` or `sc_in_clk` that is used as a clock, and that should be assumed to have an alternate clock period.

period

The period that Stratus should assume for the specified clock.

Recommended placement

Place in the module's constructor for the module containing `SC_METHOD`s or `SC_CTHREAD`s that use the referenced clock.

Description

The `HLS_SET_CLOCK_PERIOD` is used primarily with Clock Domain Crossing (CDC) circuits to specify a period for a clock that is different to the period specified to Stratus HLS for processing of the module.

Stratus HLS uses the specified clock period when performing timing analysis on individual `SC_CTHREAD`s and `SC_METHOD`s. It also uses the clock period as a constraint when selecting and constructing datapath components. In a design that has multiple clock domains, some threads and methods may require one clock, while other threads and methods require another clock. For a given thread or method, Stratus HLS will use the default clock period (for example, the one specified with `--clock_period`, or using a `hlsLib`). For threads and methods that should use a different clock period, the `HLS_SET_CLOCK_PERIOD` directive should be used.

The `HLS_SET__CLOCK_PERIOD` directive specifies the clock period for a clock port on the design. Any thread or method that uses that clock port will use the specified clock period. For example:

```
SC_MODULE(M) {
   sc_in_clk clk1;
   sc_in_clk clk2;
   sc_in<bool> rst1;
   sc_in<bool> rst2;
   SC_CTOR(M) {
     // Thread that uses clk1.
     SC_CTHREAD( t, clk1.pos() );
     reset_signal_is( rst1, 0 );

    // Method that uses clk2.
     HLS_SET_CLOCK_PERIOD( clk2, 13.0 );
     SC_METHOD( m );
     sensitive << clk2.pos();
     sensitive << rst2.neg();
}
...
```

The `HLS_SET_CLOCK_PERIOD` directive will have the following effect on any thread or method that uses the clock it references:

- When timing analysis is being performed for the thread or method, the alternate clock period is used. This can affect both scheduling and allocation.
- No pipelined library modules will be used in a thread whose clock period is longer than the default clock period.
- No library module whose delay is longer than the alternate clock period will be used for the thread or method.

Alternate clock periods do not affect the construction of library components.

It is permissible to specify more than one `HLS_SET_CLOCK_PERIOD` to support designs with more than 2 clock domains.

Cadence recommends the following practices using `HLS_SET_CLOCK_PERIOD` with CDC designs:

- Large algorithmic `SC_CTHREAD`s should not use alternate clock periods. If an `SC_CTHREAD` contains a significant amount of logic, and must use a different period to other `SC_CTHREAD`s, it should be placed into a separate `SC_MODULE`.
- The alternate clock period should be longer than the default clock period. This is not a requirement, but since library modules all use the default clock period, this practice will avoid problems and restrictions in using library modules in alternate-clocked threads and methods.
- Optimally, alternate clocks should only be used with `SC_METHOD`s and modular interfaces that contain clock domain crossing circuits.
- Since the built-in Stratus logic synthesis integration scripts do not process alternate clocks in a special way, it is recommended that you use custom logic synthesis scripts for production designs that use `HLS_SET_CLOCK_PERIOD`. All periods are available to custom logic synthsis scripts in the `BDW_LS_CLOCK_PERIODS` variable. See Which variables are set for use in logic synthesis scripts in the *User Guide*.

See also:

- "clock_period" in the Synthesis Control Attributes section
- Clock Domain Crossing (CDC) in the *User Guide*

Example

It is common to use `HLS_SET_CLOCK_PERIOD` with CDC interfaces generated with Stratus's Generated Interface feature. The example below specifies that the reader's clock period is 13, while the writer's clock period is the default period.

```
// Reference the cdc_if interface and submodules.
#include "cdc_if.h"
#include "writer_wrap.h"
#include "reader_wrap.h"

SC_MODULE(M) {
  // Writer-side clock and reset.
  sc_in_clk wclk;
  sc_in<bool> wrst;
  // Reader-side clock and reset.
  sc_in_clk rclk;
  sc_in<bool> rrst;

  // Hierarchical connections.
  cynw_p2p<DT,ioConfig>::base_in din;
  cynw_p2p<DT,ioConfig>::base_out dout;

  // CDC interface, and accessing modules.
  cdc_if::chan<ioConfig> m_cdc;
  writer_wrapper m_writer;
  reader_wrapper m_reader;

  SC_CTOR(M) {
    // Connect each submodule the the appropriate clock,
    // and to the CDC interface.
    m_writer.clk(wclk);
    m_writer.rst(wrst);
    m_writer.din( din );
    m_writer.dout( m_cdc.input );

    m_reader.clk(rclk);
    m_reader.rst(rrst);
    m_reader.din( m_cdc.output );
    m_reader.dout( dout );

    // use the w_clk_rst and r_clk_rst APIs to connect
    // both clocks and resets to the CDC interface.
    m_cdc.w_clk_rst( wclk, wrst );
    m_cdc.r_clk_rst( rclk, rrst );

    // Specify the clock period for the reader.
    // This should be the same period used as the default
    // period when processing the reader module.
    HLS_SET_CLOCK_PERIOD( rclk, 13.0 );
}
....
```

HLS_SET_CYCLE_SLACK

A Stratus HLS directive.

Syntax

`**HLS_SET_CYCLE_SLACK**( *time* [, *label* ] );`

Equivalent Command

set_cycle_slack

Parameters

time

The anount of slack to apply, in techlib time units.

label

An optional label for messaging.

Recommended placement

Place at the top of a thread or method function to affect only that process, or in the module's constructor to affect all processes in the module. May also be places after an `HLS_SCHEDULE_REGION` directive to affect the slack used when scheduling the region.

Description

The `HLS_SET_CYCLE_SLACK` sets the amount of cycle slack to be used when scheduling a specific thread. `HLS_SET_CYCLE_SLACK` overrides the `cycle_slack` synthesis control attribute. The specified slack value is subtracted from the clock period to give an effective clock period for the process. Positive values will result in a more conservative schedule, and negative values will result in a more aggressive schedule. The slack value specified with `HLS_SET_CYCLE_SLACK` will not affect the clock period used for characterizing resources.

HLS_SET_DEFAULT_INPUT_DELAY

A Stratus HLS directive.

Syntax

`HLS_SET_DEFAULT_INPUT_DELAY` `(` *`time`* `);`

Equivalent Command

set_default_input_delay

Parameters

`time`

The delay amount in the same time units used by the project's hls_lib and tech_lib.

Recommended placement

Place in the module's constructor to affect inputs read in all `SC_CTHREAD`s and `SC_METHOD`s in the module. Place immediately following the curly brace at the beginning of each `SC_CTHREAD` or `SC_METHOD` function to affect just that process.

Description

This directive specifies an input delay that will be used for all input ports for which no `HLS_SET_INPUT_DELAY` directive has been specified. Its effect is as though a `HLS_SET_INPUT_DELAY` directive had been specified for all input ports.

The value specified in `HLS_SET_DEFAULT_INPUT_DELAY` will only affect input ports to the design. It will not affect internal signals that are inputs to a thread. The input delays for internal signals will be calculated directly by Stratus. It will also not affect ports whose access falls in the scope of a `HLS_DEFINE_FLOATING_PROTOCOL` directive. The delay given in a `HLS_SET_INPUT_DELAY` directive will be used to constrain the input during logic synthesis as well as during behavioral synthesis.

Input delays can also be specified via the `--default_input_delay` command line option, which has the same effect as specifying the `HLS_SET_DEFAULT_INPUT_DELAY` directive. Both `HLS_SET_DEFAULT_INPUT_DELAY` and `HLS_SET_INPUT_DELAY` override `--default_input_delay`.

See also:

- HLS_SET_INPUT_DELAY
- "default_input_delay" in Synthesis Control Attributes
- Controlling Inputs in the User Guide

Example

```
HLS_SET_DEFAULT_INPUT_DELAY (1, "all inputs");
HLS_SET_INPUT_DELAY(data_in, 5, "data_in"); // override default
                                        // constraint for data_in input
```

This example shows the use of both input delay directives. This is useful when most, but not all, of your input ports have the same delay. Here, `data_in` has an input delay of 5 and all others have an input delay of 1.

HLS_SET_DEFAULT_OUTPUT_DELAY

A Stratus HLS directive.

Syntax

`HLS_SET_DEFAULT_OUTPUT_DELAY``( output_delay );`

Equivalent Command

set_default_output_delay

Parameters

`output_delay`
The delay amount in the same time units used by the project's tech_lib.

Recommended placement

Place in the module's constructor to affect outputs written in all `SC_CTHREAD`s and `SC_METHOD`s in the module. Place immediately following the curly brace at the beginning of each `SC_CTHREAD` or `SC_METHOD` function to affect just that process.

Description

The `HLS_SET_DEFAULT_OUTPUT_DELAY` directive specifies the delay that will be used by default for any asynchronous output, unless an `HLS_SET_OUTPUT_DELAY` directive has been specified for the output. The *output_delay* gives the time at the end of the clock cycle that must be reserved for the readers of output signals. That is, it specifies the setup time of the downstream module. Stratus HLS will ensure that the output becomes stable in time to meet this requirement. This directive does not affect either registered outputs or asynchronous inter-thread outputs.

See also:

- HLS_SET_OUTPUT_DELAY
- Controlling outputs in the *User Guide*

HLS_SET_DEFAULT_OUTPUT_OPTIONS

A Stratus HLS directive.

Syntax

`HLS_SET_DEFAULT_OUTPUT_OPTIONS( option );`

Equivalent Command

set_default_output_options

Parameters

option

The option specifying the default handling of outputs from the thread or module.

Recommended placement

Place in the module's constructor to affect outputs written in all `SC_CTHREAD`s and `SC_METHOD`s in the module. Place immediately following the curly brace at the beginning of each `SC_CTHREAD` or `SC_METHOD` function to affect just that process.

Description

The `HLS_SET_DEFAULT_OUTPUT_OPTIONS` directive specifies the timing semantics for outputs where no `HLS_SET_OUTPUT_OPTIONS` directive is given. The parameter values have the same meaning as those for `HLS_SET_OUTPUT_OPTIONS`.

See also:

- HLS_SET_OUTPUT_DELAY
- Controlling outputs in the *User Guide*

Example

This example shows how `HLS_SET_DEFAULT_OUTPUT_OPTIONS` can be used to specify the handling of all outputs except one that is specified with `HLS_SET_OUTPUT_OPTIONS`.

```
SC_MODULE(dut) {
  sc_in_clk clk;
  sc_in<bool> rst;
  sc_out<bool> out_vld;
  sc_out< sc_uint<8> > dout1;
  sc_out< sc_uint<8> > dout2;
  SC_CTOR(dut) {
    SC_CTHREAD(t,clk.pos());
    reset_signal_is( rst, 0 );
  }
  void t() {
    // All outputs will be ASYNC_NO_HOLD
    // except out_vld, which will be ASYNC_HOLD.
    HLS_SET_DEFAULT_OUTPUT_OPTIONS( ASYNC_NO_HOLD );
    HLS_SET_OUTPUT_OPTIONS( out_vld, ASYNC_HOLD );
    HLS_SET_DEFAULT_OUTPUT_DELAY( 2.5 );
    ...
  }
};
```

HLS_SET_INPUT_DELAY

A Stratus HLS directive.

Syntax

`HLS_SET_INPUT_DELAY``(` *`signal_id, delay`*`);`

Equivalent Command

set_input_delay

Parameters

`signal_id`

The identifier for the input port or process input.

`delay`

The delay amount in the same time units used by the project's tech_lib.

Recommended placement

Place in the module's constructor to affect inputs read in all `SC_CTHREAD`s and `SC_METHOD`s in the module. Place immediately following the curly brace at the beginning of each `SC_CTHREAD` or `SC_METHOD` function to affect just that process.

Description

This directive specifies the delay after the rising clock edge at which Stratus HLS will assume data on the specified input to be valid. Stratus HLS will attempt to chain this port directly into a functional unit without introducing a register. A register may still be introduced in certain cases depending on the clock period, the input delay, and the characteristics of the available library parts.

The value specified in `HLS_SET_INPUT_DELAY` can be used on either `sc_in<>` ports on the module, or on inter-process variables or `sc_signals<>` that are written by one process and read by another. It should only be used to constrain the input delays of ports or signals read in a `HLS_DEFINE_PROTOCOL` block. It should not be used for ports or signals whose access falls in the scope of a `HLS_DEFINE_FLOATING_PROTOCOL` directive or `HLS_ASSUME_STABLE` directive.

For inter-thread signals or variables, if timing is specified by `HLS_SET_INPUT_DELAY`, that value is taken as the input delay constraint for the process in which the directive appears. The input delay value for the process that reads the signal also establishes a delay requirement for the process that writes the signal. If no `HLS_SET_INPUT_DELAY` is specified for an inter-thread signal, then an appropriate input delay value is calculated by Stratus HLS based on the delay characteristics of the process that writes the signal.

The delay given in a `HLS_SET_INPUT_DELAY` directive for an `sc_in<>` port will be used to constrain the input during logic synthesis as well as during behavioral synthesis.

Input delays can also be specified via the `--default_input_delay` command line option, which has the same effect as specifying the `HLS_SET_DEFAULT_INPUT_DELAY` directive. `--default_input_delay` can be overridden globally with `HLS_SET_DEFAULT_INPUT_DELAY` or overridden for a specific input with `HLS_SET_INPUT_DELAY`.

If no input delay is specified for an input, Stratus HLS will print a WARNING and will assume that the input delay is the same as the setup delay for a basic registers in the technology library or characterized library.

See also:

- HLS_SET_DEFAULT_INPUT_DELAY
- "default_input_delay" in Synthesis Control Attributes
- Specifying input delays in the *User Guide*

Example 1
The following example specifies an input delay of 0.5ns for both the `data0_in` and `data1_in` inputs:

```
void mod::thread0()
{
  HLS_SET_INPUT_DELAY(data0_in,0.5,"data0_delay");
  HLS_SET_INPUT_DELAY(data1_in,0.5,"data1_delay");
  while( true )
  {
    {
      HLS_DEFINE_PROTOCOL("Input");
      data0 = data0_in.read();
      data1 = data1_in.read();
    }
    {
      data_out = data0 * data1;
    }
    {
      HLS_DEFINE_PROTOCOL("write");
      data_out.write( data_out );
      wait(1);
    }
  }
}
```

The equivalent circuit looks like this:

The latency of the block in this case is going to be "N+1" clock cycles, where N is the latency of the algorithm (the multiplier in this case).

HLS_SET_IS_BOUNDED

Stratus HLS directive.

Syntax

`HLS_SET_IS_BOUNDED``(` *`signal_id`* `);`

Equivalent Command

set_is_bounded

Parameters

`signal_id`

The identifier for the port or interprocess input or output.

Recommended placement

Place either in constructor of the class in which the values are declared, or in the bodies of each thread that accesses the values.

Description

The `HLS_SET_IS_BOUNDED` directive specifies that an I/O signal can be safely accessed outside of a `HLS_DEFINE_PROTOCOL` block. It is called a *bounded* value because I/O reads and writes are restricted to take place between boundaries elsewhere in the thread. Each `HLS_DEFINE_PROTOCOL` block defines a boundary. So, while reads and writes of a `HLS_SET_IS_BOUNDED` do not need to appear inside a `HLS_DEFINE_PROTOCOL` block, limitations are placed on where the I/O can be scheduled based on surrounding `HLS_DEFINE_PROTOCOL` blocks.

The `HLS_SET_IS_BOUNDED` directive fills a similar role to the `HLS_ASSUME_STABLE` directive with several differences:

- It can be used with both inputs and outputs.
- The `HLS_ASSUME_STABLE` directive is specified within a C++ block in which the input reads can safely be scheduled, while the location of the `HLS_SET_IS_BOUNDED` directive is not important since the limits are separately specified by boundaries.
- Because the `HLS_SET_IS_BOUNDED` directive is not block-specific, it is more suitable for use with modular interfaces because the directive can be included within the modular interface source code, and need not appear in the thread accessing the interface.

The `HLS_SET_ARE_BOUNDED` form of the directive provides a convenience for specifying the directive to several inputs or outputs at one time so long as they are declared together.

The boundary mechanism is also used to control the limits on movement of explicit memory reads and writes. For a general discussion of boundaries, see Limits on movement of external memory operations in the *User Guide.*

The `HLS_SET_IS_BOUNDED` directive tends to produce unregistered input reads, and registered output writes, but with no shifting of output values through pipelines. Input reads will be unregistered as long as computations using the input value can occur within the same boundaries. Output writes are registered by default, but can be made asynchronous using the `HLS_SET_OUTPUT_DELAY` directive.

See also:

- Specifying stable inputs in the *User Guide*
- HLS_ASSUME_STABLE
- HLS_SET_OUTPUT_DELAY

Example 1

```
SC_MODULE(dut) {
  sc_in<bool> in_valid;
  sc_in<bool> out_ready;
  sc_in< sc_uint<8> > in_data;
  sc_out< sc_uint<8> > out_data;

SC_CTOR(dut) {
  // Specify that the in_data input
  // and the out_data output can be performed
  // safely between the boundaries formed
  // by the surrounding HLS_DEFINE_PROTOCOL blocks.
  HLS_SET_IS_BOUNDED( in_data );
  HLS_SET_IS_BOUNDED( out_data );
}
void thread {
  while (1) {
    // The end of this HLS_DEFINE_PROTOCOL forms
    // the uppper boundary.
    {HLS_DEFINE_PROTOCOL("wait_in");
     do {wait();} while (!in_valid.read());
    }

    // more code...

    // Portion of the thread that accesses in_data and out_data.
    out_data = f( in_data.read() );

    // more code...

    // The start of this HLS_DEFINE_PROTOCOL block forms
    // the lower boundary.
    {HLS_DEFINE_PROTOCOL("wait_out");
     do {wait();} while (!out_ready.read());
    }
}
```

In this example, the in_data input and the out_data output are identified as bounded values. The boundaries are defined implicitly by the `HLS_DEFINE_PROTOCOL` blocks before and after the reads and writes of the values. Note that the reads and writes of in_data and out_data are not in `HLS_DEFINE_PROTOCOL` blocks. If, for example, the loop is scheduled in 10 cycles, Stratus HLS is permitted to schedule the read of in_data at any time after in_valid is true, and before out_ready is true. This allows the input read to be moved close to its use in the thread, and for input registers to be eliminated. Similarly, as soon as the out_data value is calculated, it can be written.

It is important that the `HLS_SET_IS_BOUNDED` directive only be used when the design's I/O protocol supports reading and writing of the values at any time within the region defined by the boundaries.

HLS_SET_OUTPUT_DELAY

A Stratus HLS directive.

Syntax

`HLS_SET_OUTPUT_DELAY(` ***`signal_id`*****`,`** ***`output_delay`*** **`);`**

Equivalent Command

set_output_delay

Parameters

`signal_id`

The name of an `sc_out` port, or an `sc_signal` or member variable written in one thread and read in another.

`output_delay`

The double that specifies the external delay (setup time) on an async output. The units of the delay are the same as the units of the tech_lib being used to process the design.

Recommended placement

Place in the module's constructor to affect outputs written in all `SC_CTHREAD`s and `SC_METHOD`s in the module. Place immediately following the curly brace at the beginning of each `SC_CTHREAD` or `SC_METHOD` function to affect just the outputs written by that process.

Description

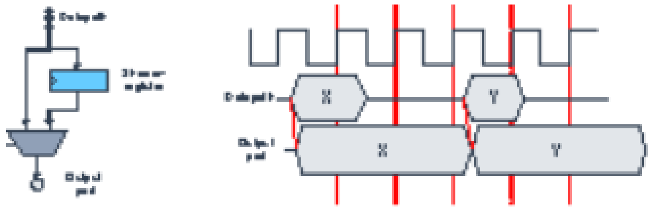
The `HLS_SET_OUTPUT_DELAY` directive controls the way Stratus HLS specifies the required delay for the output. This directive is only relevant when either the `HLS_SET_OUTPUT_OPTIONS` or `HLS_SET_DEFAULT_OUTPUT_OPTIONS` directive has been used to specify that the output is asynchronous using one of the `ASYNC_*` codes. The *delay* gives the time at the end of the clock cycle that must be reserved for the reader of the signal. That is, it specifies the setup time of the downstream module. Stratus HLS will ensure that the output becomes stable in time to meet this requirement.

See also:

- HLS_SET_DEFAULT_OUTPUT_OPTIONS
- Controlling outputs in the *User Guide*

Example 1

```
HLS_SET_OUTPUT_OPTIONS( ouput, ASYNC_HOLD );
HLS_SET_OUTPUT_DELAY( output, 2.5 );
```



In this example, the port *output* has been specified to be asynchronous using `HLS_SET_OUTPUT_OPTIONS` directive, and its output delay constraint has been set to 2.5. This means that Stratus will reserve 2.5 ns at the end of the clock cycle for downstream processing. If the clock period is 10 ns, and a 1 ns cycle slack has been specified, that means the output will be guaranteed to be stable (10.0 - 1.0 - 2.5) = 6.5 ns after the preceding clock edge.

HLS_SET_OUTPUT_OPTIONS

A Stratus HLS directive.

Syntax

`HLS_SET_OUTPUT_OPTIONS``(` *`signal_id, option`* `);`

Equivalent Command

set_output_options

Parameters

`signal_id`

The name of an `sc_out` port, or an `sc_signal` or member variable written in one thread and read in another.

`options`

The option specifying the timing options for the output.

Recommended placement

Place in the module's constructor to affect outputs written in all `SC_CTHREAD`s and `SC_METHOD`s in the module. Place immediately following the curly brace at the beginning of each `SC_CTHREAD` or `SC_METHOD` function to affect just the outputs written by that process.

Description

The `HLS_SET_OUTPUT_OPTIONS` directive controls the way Stratus HLS specifies the kind of circuit Stratus HLS will build for an output. The allowable values are:

- **SYNC_HOLD (default)**: The output will be registered, and its value will be held stable between writes to the output. This is the option used if no `HLS_SET_OUTPUT_OPTIONS` directive is specified for an output.
- **SYNC_NO_HOLD**:The output will be registered, but its value will only be held stable for a single cycle, or longer if that cycle is extended by a pipeline stall. This places restrictions on the protocol used when writing the output, but it also makes the output register available to be shared during other cycles, potentially reducing design area.
- **ASYNC_HOLD**: The output will be updated asynchronously, but its value will be held stable between assertions. A backing output register will be used to guarantee the stability of the output unless the output is written in every state.
- **ASYNC_NO_HOLD**: The output will be updated asynchronously, and its value is only guaranteed to remain stable for one cycle. An output register will never be used for this output.

- **ASYNC_STALL_NO_HOLD**: The output will be updated asynchronously, and its value is only guaranteed to remain stable for one cycle. However, if a pipeline stall occurs during that cycle, the value will be held stable during the stall. An output register will only be used to guarantee stability during stalls in cases where there is an asynchronous path from an input to the output.

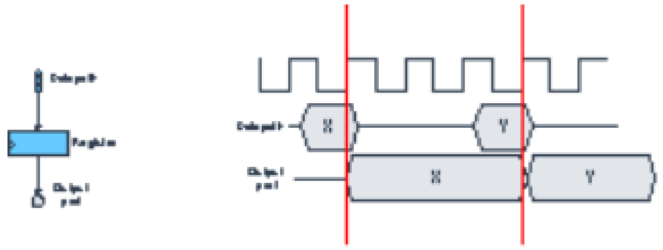
Note that when one of the `ASYNC_*` options is used on an `sc_out` port, it is recommended to also use the `HLS_OUTPUT_DELAY` directive to specify a timing constraint for the output.

See also:

- HLS_SET_DEFAULT_OUTPUT_OPTIONS
- Controlling outputs in the *User Guide*

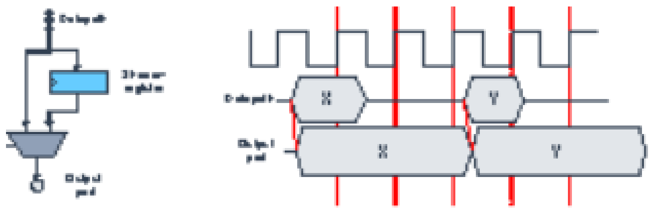
Example 1

```
HLS_SET_OUTPUT_OPTIONS( output, SYNC_HOLD );
```



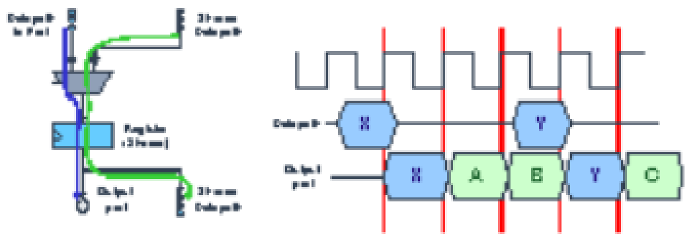
Example 2

```
HLS_SET_OUTPUT_OPTIONS( output, ASYNC_HOLD );
HLS_SET_OUTPUT_DELAY( output, 2.5 );
```



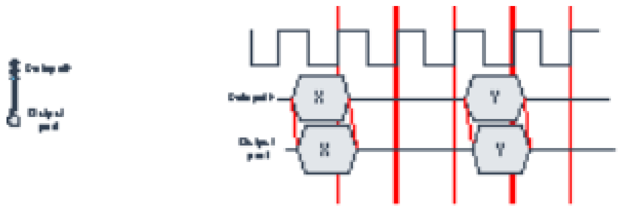
Example 3

```
HLS_SET_OUTPUT_OPTIONS( output, SYNC_NO_HOLD );
```



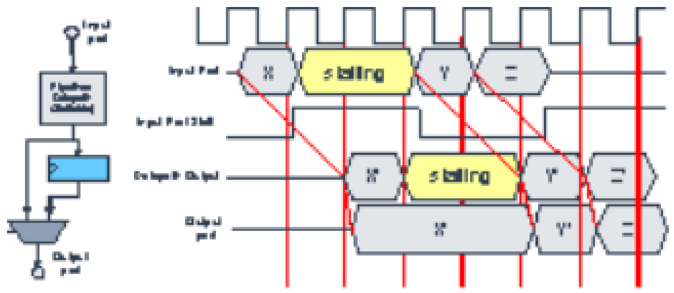
Example 4

```
HLS_SET_OUTPUT_OPTIONS( output, ASYNC_NO_HOLD );
HLS_SET_OUTPUT_DELAY( output, 2.5 );
```



Example 5

```
HLS_SET_OUTPUT_OPTIONS( output, ASYNC_STALL_NO_HOLD );
HLS_SET_OUTPUT_DELAY( output, 2.5 );
```



HLS_SET_STALL_VALUE

A Stratus HLS directive.

Syntax

`HLS_SET_STALL_VALUE``( signal_id, value, [global=true]);`

Equivalent Command

set_stall_value

Parameters

`signal_id`
The identifier for an output port or inter-process signal.

`value`
The integer value that the given signal will be set to during a stall.

global
A boolean value that is true if the stall value should apply globally for all stalls.

Recommended placement

Place in the module's constructor, or in the body of the `SC_CTHREAD` that writes *signal* for global stalls, and in the body of the thread where it should take effect for local stalls.

Description

The `HLS_SET_STALL_VALUE` directive specifies a constant value that should be written to an output port or internal signal during a pipeline stall condition. A pipeline stall is inferred from a busy loop as described in Pipeline stalling in the *User Guide*.

When a pipeline for an `SC_CTHREAD` is stalled, its FSM does not advance, and none of its registers will change. Because register values do not change, the protocol outputs written from the `SC_CTHREAD` will also not change. This may cause violations of the signal-level protocol. The `HLS_SET_STALL_VALUE` can be used to cause a port or signal to be asserted to a particular value when the state machine is stalled. For example an `in_rdy` port might be set to 0 during a stall condition to prevent an upstream module from writing values during the stall.

`HLS_SET_STALL_VALUE` may be used for both `sc_out<>` ports, and internal `sc_signal<>s`. Only ports or signals with scalar types (e.g. `sc_uint<>`, `bool`, etc.) are supported. The value specified must be a constant that is compatible with the port or signal.

Note that `HLS_SET_STALL_VALUE` does not affect whether or not the specified signal is registered. If `HLS_SET_STALL_VALUE` is specified for a registered signal or output, its value will be applied when a stall condition is active at a clock edge. On the first clock edge after the stall, normal operation of the FSM will resume, and the register will be set to the value it would have received if a stall had not occurred.

If `HLS_SET_STALL_VALUE` is applied to an unregistered port or signal, its value will be updated asynchronously as soon as the stall condition occurs, and will return to its normal value as soon as the stall condition goes away. See HLS_SET_OUTPUT_DELAY for information on how to specify that a port or signal should be updated asynchronously.

In pipelines using the `HARD_STALL` option, any stall condition will affect all stages in the pipeline concurrently. This is called a *global* stall. However, if the `SOFT_STALL` option is used, then each pipeline state may stall independently. In `SOFT_STALL` pipelines, it may be necessary to specify that a stall value be applied only in certain stages of a pipeline. In such cases, the `HLS_SET_STALL_VALUE` directive should be placed within the pipeline, at the point in the control flow where it should take effect. This is often adjacent to an assignment to the signal affected by the directive.

See also:

- Pipeline stalling in the *User Guide*
- Overriding values of signals during pipeline stalls in the *User Guide*

Example 1

The following example has a hard output stall, and a sampled input in a HARD_STALL pipeline. When a stall occurs on the output, the `HLS_SET_STALL_VALUE` directive will cause the `in_rdy` signal to be set to 0 in order to prevent the upstream module from writing values during the pipeline stall. Because the pipeline is `HARD_STALL`, all stages will stall concurrently, so the `HLS_SET_STALL_VALUE` can be global.

```
SC_MODULE(M) {
  sc_in_clk clk;
  sc_in<bool> rst;
  sc_in<bool> in_vld;
  sc_out<bool> in_rdy;
  sc_in< sc_uint<8> > in_data;
  sc_out<bool> out_vld;
  sc_in<bool> out_rdy;
  sc_out< sc_uint<8> > out_data;
  SC_CTOR(M) {
     SC_CTHREAD(t,clk.pos());
     reset_signal_is(rst,0);
}
void t() {
  HLS_SET_DEFAULT_INPUT_DELAY(1,"");
  HLS_SET_STALL_VALUE( in_rdy, 0 );
  { HLS_DEFINE_PROTOCOL("reset");
```

```
    in_rdy = 0;
    out_vld = 0;
    wait();
  }
while (1) {
   sc_uint<8> d, q, vld;

   // The thread is pipelined II=1 with HARD_STALL
   HLS_PIPELINE_LOOP(HARD_STALL,1,"pipe");

   // If an output stall occurs while in_rdy is 1,
   // the HLS_SET_STALL_VALUE() directive will cause
   // it to be set to 0 during the stall.
   { HLS_DEFINE_PROTOCOL("input");
     in_rdy = 1;
     wait(1);
     vld = in_vld.read();
     d = in_data.read();
     in_rdy = 0;
   }
   q = f(d);

   // Hard stall on the output.
   // This causes the stall condition, which affects all stages concurrently.
   // When the stall condition "out_rdy--0" is in effect,
   // the HLS_SET_STALL_VALUE will be applied.
   { HLS_DEFINE_PROTOCOL("output");
     out_vld = vld;
     out_data = q;
     do { wait(); } while (out_rdy.read()==0);
     out_vld = 0;
   }
  }
 }
};
```

Example 2

The following example has a hard output stall, and a sampled input in a `SOFT_STALL` pipeline. When a stall occurs on the output, it will affect different stages of the pipeline at different times, so it is necessary to specify the stage in which the `HLS_SET_STALL_VALUE` directive should have its effect. For that reason, the directive has been moved from the top of the thread to be next to the `in_rdy=0` assignment.

```
SC_MODULE(M) {
  sc_in_clk clk;
  sc_in<bool> rst;
  sc_in<bool> in_vld;
  sc_out<bool> in_rdy;
  sc_in< sc_uint<8> > in_data;
  sc_out<bool> out_vld;
  sc_in<bool> out_rdy;
  sc_out< sc_uint<8> > out_data;
  SC_CTOR(M) {
     SC_CTHREAD(t,clk.pos());
     reset_signal_is(rst,0);
}
void t() {
  HLS_SET_DEFAULT_INPUT_DELAY(1,"");
  { HLS_DEFINE_PROTOCOL("reset");
    in_rdy = 0;
    out_vld = 0;
    wait();
  }
while (1) {
   sc_uint<8> d, q, vld;

   // The thread is pipelined II=1 with SOFT_STALL.
   HLS_PIPELINE_LOOP(SOFT_STALL,1,"pipe");

   // If an output stall occurs while in_rdy is 1, that stall will ripple
   // through the pipe, and eventually case this stage to stall.
   // The local HLS_SET_STALL_VALUE() directive will cause
   // it to be set to 0 when the stall affects this stage.
   { HLS_DEFINE_PROTOCOL("input");
     HLS_SET_STALL_VALUE( in_rdy, 0, false );
     in_rdy = 1;
     wait(1);
     vld = in_vld.read();
     d = in_data.read();
     in_rdy = 0;
   }
   q = f(d);

   // Hard stall on the output.
   // When the stall condition "out_rdy--0" is in effect,
   // the HLS_SET_STALL_VALUE will be applied only when the stall affects.
   // the stage where the directive appears.
   { HLS_DEFINE_PROTOCOL("output");
     out_vld = vld;
```

```
        out_data = q;
        do { wait(); } while (out_rdy.read()==0);
        out_vld = 0;
      }
    }
  }
};
```

HLS_SPLIT_ADD

A Stratus HLS directive.

Syntax

`HLS_SPLIT_ADD(<input_width>, <name>);`

Equivalent Command

split_add

Parameters

input_width

An integer value specifying the maximum split width. The value must be more than 2 to apply the splitting operation. A value of 0, locally turns of the splitting operation.

name

The <name> is a user-defined identifier for the region to split ops.

Recommended placement

Place in the body of the SC_CTHREAD.

Description

The `HLS_SPLIT_ADD` directive specifies a constant value that is used to split adders/subtrators having greater bit-width than `input_width` so that operand width will be within the `input_width size`. The `HLS_SPLIT_ADD` directive inside a DPOPT region is ignored during ASIC flow.

Example 1

The following example has 2 adders. The adder that has `HLS_SPLIT_ADD` being applied would be split.

```
{
HLS_SPLIT_ADD (3, "split_the_add");
c = a2 + a3;
}

b = a1 + a2;
```

HLS_SPLIT_MULTIPLY

A Stratus HLS directive.

Syntax

`HLS_SPLIT_MULTIPLY`*`(<input_width>, <name>);`*

Equivalent Command

split_multiply

Parameters

`input_width`

An integer value specifying the maximum split width. The value must be more than 2 to apply the splitting operation. A value of 0, locally turns of the splitting operation.

`name`

Specifies a user-defined identifier for the region to split ops.

Recommended placement

Place in the body of the SC_CTHREAD.

Description

The `HLS_SPLIT_MULTIPLY` directive specifies a constant value that is used to split multipliers having greater bit-width than `input_width` so that operand width will be within the `input_width size`. The `HLS_SPLIT_MULTIPLY` directive inside a `DPOPT` region is ignored during ASIC flow.

Example 1

The following example has 2 multipliers. The multiplier that has `HLS_SPLIT_MULTIPLY` being applied would be split.

```
{
  HLS_SPLIT_MULTIPLY (3, "mult_split");
  c = a2 * a3;
}

  b = a1 * a2;
```

HLS_UNROLL_LOOP

A Stratus HLS directive.

Syntax

`HLS_UNROLL_LOOP``( [ `*`type`*`, [ `*`count`*`,]] [,`*`name`*`])`

Equivalent Command

unroll_loops

Parameters

`type`

The unrolling type: `ON`, `ALL`, `OFF`, `COMPLETE`, `AGGRESSIVE`, or `CONSERVATIVE`. The default type is `ON`.

`count`

The number of times the loop body should be replicated.

`name`

A unique name for the unroll used for reporting purposes.

Recommended placement

Place immediately following the curly brace that encloses the loop to be unrolled.

Description

The `HLS_UNROLL_LOOP` directive can completely or partially unroll a loop, resulting in multiple hardware instances in the cycle-accurate design generated by Stratus HLS. Unrolling a loop is a powerful transformation that allows parallelism to be exploited in the design.

Stratus HLS automatically unrolls loops within pipelines (that is, in `HLS_PIPELINE_LOOP` blocks) and DpOpt parts (that is, in `HLS_DPOPT_REGION` blocks).

Unless the `unroll_loops` synthesis control option is set to `on`, other types of loops are not unrolled. In this case, if loop unrolling is desired, the `HLS_UNROLL_LOOP` statement should be the first statement of the loop. If a loop cannot be unrolled as specified, Stratus HLS will generate a warning at runtime.

When you unroll a loop, you'll often need to flatten arrays that are access from inside the body of the loop. The array must have one port for each time the loop is unrolled.

There are 2 basic versions of this directive: one in which the loop unroll count is specified, and one where it is not. The *type* parameter determines which form is required.

The unroll count is not required for the following types:

- **ON**: Completely unrolls the current loop if the number of iterations can be determined.

- **ALL**: Completely unrolls the current loop and all inner loops if the number of iterations can be determined. Unrolling of inner loops can be prevented with the `OFF` setting.
- **OFF**: Disables unrolling of the current loop. This is useful when the global `--unroll_loops` feature is in use, but you want to stop a specific loop from unrolling. The `OFF` setting also prevents a nested loop from being unrolled when the ALL setting is used on the outer loop. OFF affects only the specified loop, not inner loops.

The unroll count is required for the following types:

- **COMPLETE**: With `COMPLETE` unrolling, the loop logic is completely replicated when the *value* is equal to the total number of loop iterations. Using `COMPLETE` unrolling will result in the largest synthesized area, but often the minimum latency and highest throughput for the loop. A partial unroll is possible if you specify a *value* lower than the loop iteration number.
- **CONSERVATIVE**: Conservative loop unrolling is used when the total number of loop iterations is unknown or variable in the design, and the loop itself contains a "loop termination." A loop termination typically uses the C/C++ `break` command inside the loop on a certain condition. Stratus HLS `CONSERVATIVE` loop unrolling can produce parallelism in such a loop (which normally would not be possible) by including the loop termination condition in each replicated iteration. In this case, the *value* defines the number of iteration replicates that can individually process data until the `break` condition is hit. Adjusting this value will vary the synthesized area vs. throughput.
- **AGGRESSIVE**: Aggressive loop unrolling allows Stratus HLS to ignore any loop termination conditions and introduce as much parallelism as defined by the *value*. The use of `AGGRESSIVE` loop unrolling can result in synthesized logic that operates differently than the SystemC model, so caution must be used when considering `AGGRESSIVE` loop unrolling. Specifically, `AGGRESSIVE` unrolling should only be used when the total number of loop iterations is known and the *value* is a partial multiple of the total number of iterations. In other words, in a `for` loop with `m` iterations and `n` as the *value*, `m modulo n = 0`.

The *count* is the number of times the loop is to be unrolled (i.e., how much the loop is parallelized). Typically this number is a partial multiple of the total number of loop iterations. For example, a `for` loop that has 16 total iterations might typically be unrolled eight (one-half) or four (one-quarter) times. You may need to try several *values* to find the optimal area vs. throughput tradeoff.

If no parameters are specified, or if only a label parameter is specified, the default type is ON and the count must be automatically determined by Stratus HLS.

If Stratus HLS cannot unroll a loop, it will provide a warning. Circumstances stopping a loop from unrolling include an indeterminate number of loop iterations and the explicit use of the `OFF` setting for `HLS_UNROLL_LOOP`.

Following is the order of precedence of the various loop unrolling types:

Unrolling type	**Description**
`HLS_UNROLL_LOOP(OFF)`	Highest precedence of loop unrolling settings.
`HLS_UNROLL_LOOP(ON/ALL/COMPLETE/CONSERVATIVE/AGGRESSIVE);` also unrolling from `HLS_PIPELINE_LOOP/HLS_DPOPT_REGIO`N	Next highest precedence. Note that loops within `HLS_PIPELINE_LOOP` and `HLS_DPOPT_REGION` blocks are automatically unrolled by Stratus HLS; they do not require an unroll setting.
`unroll_loops`	Lowest precedence.

See also:

- Unrolling Loops in the *User Guide*
- "unroll_loops" in Synthesis Control Attributes

Example 1

```
for ( i=0; i<4; i++ ){
  HLS_UNROLL_LOOP ( ON, "shift" );
  r[i]=i;
}
```

This is a simple example of complete loop unrolling via the `ON` setting. After unrolling, the loop is totally eliminated.

```
r[0]=0;
r[1]=1;
r[2]=2;
r[3]=3;
```

Since all accesses to `r` are now constants, r can be flattened. Consequently, all of the accesses can be done in parallel. This behavior can also be expressed in this manner:

```
for ( i=0; i<4; i++ ){
  HLS_UNROLL_LOOP ( COMPLETE, 4, "shift" );
  r[i]=i;
}
```

Example 2

```
for ( i = 0; i < 8; i++ ) {
   HLS_UNROLL_LOOP(ALL,"loop1");
   // This is unrolled because the ALL directive unrolls this and all
   // enclosed loops.
   for ( j = 0; j < 8; j++ ) {
      HLS_UNROLL_LOOP(OFF,"loop2");
      // This is not unrolled because of the OFF directive.
      for ( k = 0; k < 8; k++ )
           // No unroll directive.
           // This is unrolled because of the ALL directive for loop1.
      }
   }
}
```

This example demonstrates the use of `HLS_UNROLL_LOOP` on nested loops. *loop1* is unrolled because the `ALL` setting is applied to it and the iteration count is determinable. *loop2* would have been unrolled, except it is explicitly turned OFF. The third loop has no `HLS_UNROLL_LOOP` setting, but is unrolled because 1) the `ALL` setting in *loop1* unrolls inner loops, 2) the `OFF` setting in *loop2* does not affect inner loops, and 3) the iteration count is determinable.

Example 3

```
for(i=0;i<4;i++){
  HLS_UNROLL_LOOP(AGGRESSIVE, 3, "mem_init");
  r[i]=i;
}
```

This example demonstrates the problems that can arise if `AGGRESSIVE` unrolling is used and the `HLS_UNROLL_LOOP` *value* is not a partial multiple of the total number of iterations. In this trivial example a dual-ported memory could be used. However, if the loop is unrolled three times it becomes:

```
for(i=0;i<4;i=i+3){
  r[i]=i;
  // no check for loop termination!
  r[i+1]=i+1;
  // no check for loop termination!
  r[i+2]=i+2;
}
```

In this case, during the second of the iterations, memory locations 5 and 6 would be written (which is not desired).

Synthesis Tcl Commands

This section lists the synthesis Tcl commands used in Stratus HLS.
You can specify a set of Tcl commands to `stratus_hls` that will be executed in the uarch control phase of synthesis. These can include commands from the Control API. At this point, you have access through the API to the specific objects in the design, such as the specific loops and arrays in the design. The Control API can be used to set attributes on these objects to control the micro-architecture.

Every synthesis Tcl command has an equivalent source code directive (synthesis directive) that is set in the SystemC Source file (`.cpp` file). For more information, see Controlling Micro-Architecture in the *User Guide*.

Region Command	Description
assume_stable	Identifies an input (or group of inputs) that is stable within a block. See also HLS_ASSUME_STABLE.
break_array_dependency	Allows you to break the inter-iteration array dependencies for a given array or loop.
break_protocol	Identifies a specific location in the control flow where clock cycles may be added by scheduling. See also HLS_BREAK_PROTOCOL.
coalesce_loops	Specifies how a loop will be coalesced with other loops in the same nest. See also HLS_COALESCE_LOOP.
constrain_array_max_distance	Permits access to dual ported memories in pipelines, and specifies the maximum safe distance between reads and writes. See also HLS_CONSTRAIN_ARRAY_MAX_DISTANCE.
constrain_latency	Constrains a code block to be scheduled in specified number of clock cycles. See also HLS_CONSTRAIN_LATENCY.
constrain_region	Constrains the latency, input delay, and output delay of the resource created for either a scheduled region or DpOpt region. See also HLS_CONSTRAIN_REGION.
define_floating_protocol	Defines a fixed-latency protocol as an atomic transaction that can be scheduled as an operation. See also HLS_DEFINE_FLOATING_PROTOCOL.

define_protocol	Specifies that the region should remain cycle-accurate. See alsoHLS_DEFINE_PROTOCOL.
dpopt_region	Defines an optimized datapath component for the region that can be scheduled as an operation. See also HLS_DPOPT_REGION.
flatten_arrays	Controls array flattening on a local basis. See also HLS_FLATTEN_ARRAY.
invert_dimensions	Inverts the dimensions for a multi-dimensional array, usually to cause outer indexes to be constants. See also HLS_INVERT_DIMENSIONS.
map_array_indexes	Controls how memory indexes are calculated for multi-dimensional arrays. See also HLS_MAP_ARRAY_INDEXES.
map_to_memory	Maps an array to a memory. See also HLS_MAP_TO_MEMORY.
map_to_reg_bank	Maps an array to a register bank. See also HLS_MAP_TO_REG_BANK.
pipeline_loops	Controls loop pipelining. See also HLS_PIPELINE_LOOP.
preserve_io_signals	Prevents input and output signals from being optimized away because no other process uses them. See also HLS_PRESERVE_IO_SIGNALS.
preserve_signals	Preserves an `sc_signal<>` through the synthesis process. See also HLS_PRESERVE_SIGNAL.
remove_control	Controls whether FSM-based control or mux-based control is used. See also HLS_REMOVE_CONTROL.
schedule_region	Provides a way to have Stratus HLS segregate part of a design's functionality for separate processing with minimal modification to the source code structure. See also HLS_SCHEDULE_REGION.
separate_arrays	Separates that a multi-dimensional array should be implemented as several separate arrays. See also HLS_SEPARATE_ARRAY.

set_clock_period	Specifies an alternate clock period to be used for a specific clock signal. See also HLS_SET_CLOCK_PERIOD.
set_cycle_slack	Specifies an alternate cycle slack for a specific thread.
set_default_input_delay	Specifies the default delay assumed for thread inputs. See also HLS_SET_DEFAULT_INPUT_DELAY.
set_default_output_delay	Controls the default output delay for asynchronous thread outputs. See also HLS_SET_DEFAULT_OUTPUT_DELAY.
set_default_output_options	Specifies the default timing semantics for thread outputs. See also HLS_SET_DEFAULT_OUTPUT_OPTIONS.
set_input_delay	Specifies the delay assumed for specific thread inputs. See also HLS_SET_INPUT_DELAY.
set_is_bounded	Specifies that input reads can be moved to avoid registers, limited by surrounding boundaries. See also HLS_SET_IS_BOUNDED.
set_output_delay	Specifies the output delay assumed for specific asynchronous thread outputs. See also HLS_SET_OUTPUT_DELAY.
set_output_options	Specifies the timing semantics for specific thread outputs. See also HLS_SET_OUTPUT_OPTIONS.
set_stall_value	Specifies a constant value that should be written to an output port or internal signal during a pipeline stall condition. See also HLS_SET_STALL_VALUE.
split_add	Specifies a constant value that should be used for splitting a large adders/subtractors. See also HLS_SPLIT_ADD.
split_multiply	Specifies a constant value that should be used for splitting a large multiplier. See also HLS_SPLIT_MULTIPLY.
unroll_loops	Controls loop unrolling on a local basis. See also HLS_UNROLL_LOOP.

assume_stable

A synthesis Tcl command.

Syntax

`assume_stable` `[ -name` *`name`* `] [`*`-untimed`*`]` *`region_or_loop_id io_or_list_of*`*

Equivalent Directive

HLS_ASSUME_STABLE

Parameters

`name`

A name to be used in messages.

-untimed

If specified, any datapath driving the input is calculated in an asynchronous way that does not affect the schedule of the accessing thread.

`region_or_loop_id`

Specifies the region or loop in which the inputs are stable. This must appear first.

`io_or_list_of`

Specifies one or more I/O symbols for the enclosing behavior. These do not need to be full object paths, but can be.

Description

The `assume_stable` command identifies a set of inputs, and a region in which they can be assumed to be stable. Any read of the inputs within that region will not result in the input being registered, even if the computation that uses the input appears later in the region.

break_array_dependency

A synthesis Tcl command.

Syntax

`break_array_dependency region_id array_id_or_list_of`

Equivalent Directive

HLS_BREAK_ARRAY_DEPENDENCY

Parameters

`region_id`

Specifies the ID of the region/loop/behavior containing the array.

`array_id_or_list_of`

Specifies the ID(s) of array(s) to break inter-iteration dependencies.

Description

The `break_array_dependency` command breaks the inter-iteration array dependencies for a given region, loop, or behavior.

break_protocol

Synthesis Tcl command.

Syntax

`break_protocol [-before | -after] [-latency lat | [-min_lat min] [-max_lat max]] [ -name string ] basic_block_ids`

Parameters

`[-after]`
Specifies that the protocol is to be broken after a basic block. This is the default option.

`[-before]`
Specifies that the protocol is to be broken before a basic block.

[-latency *lat*]
Specifies the latency that is applied to both min and max value.

`[-min_lat min]`
Specifies the minimum acceptable latency for a basic block. The default value is 0.

`[-max_lat max]`
Specfies the maximum acceptable latency for a basic block. If omitted, the maximum latency is infinite.

`[-name string ]`
If specified, *`string`* identifies the constraint implied by the command in all reports. If not specified, the default *`string`* is `break_protocol`, which means that `break_protocol` is the default name for constraints implied by the break_protocol command.

`basic_block_ids`
List of basic block IDs.

Description

The `break_protocol` command will find a single basic block that is after all specified basic blocks, or before all specified basic blocks depending on whether `-before` or `-after` is specified. So, if a user specifies `-after`, for a given set of basic blocks, the user is saying, "I want to break protocol after all these basic blocks have finished."

Example 1 : Break after a wait() that has a label:

Tcl:

```
    break_protocol -after [find -basic_block -if {state_name == "MYSTATE"}]
```

SystemC:

```
wait(1);
if (setup_tmp) {
  COEFF_READY = 1;   // output write
  MYSTATE: wait(1);
  // Will insert break here.
  COEFF_READY = 0; // output write
}
```

Example 2 : Break after all operations on the 'DOUT_rdy' input:

Tcl:

```
break_protocol -after [get_attr operations.basic_block [find -behavior_io DOUT_rdy]]
```

SystemC:

```
bool tmp_rdy;
do {
    tmp_rdy = DOUT_rdy.read();
    // Will break here.
    wait(1);
} while (!tmp_rdy);
DOUT_vld = 1;           // output write
DOUT = tmp_dout;
wait(1);
DOUT_vld = 0;
```

Example 3 : Break before any operation on DOUT_vld in a loop named MAIN_LOOP

Tcl:

```
break_protocol -before [filter -within [find -loop MAIN_LOOP] \
      [get_attr operations.basic_block [find -behavior_io DOUT_rdy]]]
```

SystemC:

```
DOUT_vld = 0;
// Won't break here.
wait();
MAIN: while (1) {
  bool tmp_rdy;
  do {
      tmp_rdy = DOUT_rdy.read();
```

```
        wait(1);
    } while (!tmp_rdy);

    // Will break here.
    DOUT_vld = 1;           // output write
    DOUT = tmp_dout;
    wait(1);
    DOUT_vld = 0;
```

Example 4 : Break after a region named "compute"

Tcl:

```
break_protocol - after -min_lat 2 [find -region COMPUTE -return basic_blocks]
```

SystemC:

```
while (true) {
    PDT sum=1;
    {
        HLS_DPOPT_REGION( NO_ALTS, "COMPUTE", "");
        for (PDT i=0; i< 4; i++) {
            sum =  sum * i * m_a.read() ;
        }
    }
    // Will break here.
    m_x.write(sum);
    wait();
}
```

Example 5 : Break at the start of a loop named LOOP1

Tcl:

```
break_protocol -before [find -loop LOOP1 -return header]
```

SystemC:

```
// Will break here.
LOOP1: for ( int i=0; i<16; i++) {
    if(cur < limit) {
        sum += vals[limit];
    }
}
```

coalesce_loops

A synthesis Tcl command.

Syntax

`coalesce_loops [-type` *`option`* `] [ -name` *`name`* `]` *`loop_id_or_list_of`*

Equivalent Directive

HLS_COALESCE_LOOP

Parameters

`option`

Determines how loop coalescing is affected by the directive. The default value is CONSERVATIVE. You can use any of the following options:

- **ON**: The loop will be coalesced with its parent loop.
- **OFF**: The loop will not be coalesced even if a parent loop specified `HLS_COALESCE_LOOP(ALL)`.
- **CONSERVATIVE**: The loop will be coalesced with any parent loop that is pipelined. Any intervening loops will also be coalesced.
- **ALL**: All child loops that are not explicitly unrolled will be coalesced with the loop.

`name`

A name to be used in messages.

`loop_id_or_list_of`

The IDs of the loops to be affected.

DescriptionThe `coalesce_loops` command controls how Stratus HLS performs loop coalescing relative to the specified loop. Loop coalescing is an alternative to loop unrolling, primarily used for loops nested within pipelined loops. Loop coalescing allows some loops to be integrated with the enclosing pipelined loop instead of being unrolled. Unlike loop unrolling, the loop's contents are not duplicated.

Loop coalescing is useful for the following applications:

- When an inner loop contains many iterations, and would causes scalability problems if it was unrolled.
- When an inner loop has an indeterminate number of iterations, and cannot be unrolled.

Loops coalescing is appropriate for tightly nested loops near the top of a pipelined loop. More details can be found in the *User Guide.*

See also:

- HLS_UNROLL_LOOP
- Coalescing Loops in the *User Guide*
- Pipelining Loops in the *User Guide*

constrain_array_max_distance

A synthesis Tcl command.

Syntax

`constrain_array_max_distance` `[ -name name ] [ -max dist ]` *`array_id loop_id`*

Equivalent Directive

HLS_CONSTRAIN_ARRAY_MAX_DISTANCE

Parameters

name

A name to be used in messages..

`dist`
Specifies the maximum distance constraint.

`array_id`
The ID of the array that will be mapped to a dual-port memory in which the ports should be constrained.

`loop_id`
The ID of the loop that will be mapped to a dual-port memory in which the ports should be constrained.

Description

The `constrain_array_max_distance` command permits pipelined Stratus HLS designs to employ dual-port memories to both a loop and an array.

constrain_latency

A synthesis Tcl command.

Syntax

`constrain_latency` `[-name name] [-`*`min_lat min_lat`*`] [-`*`max_lat max_lat`*`]` *`region_or_loop_id`*

Equivalent Directive

HLS_CONSTRAIN_LATENCY

Parameters

name

The name of the latency constraint used for reporting purposes.

`min_lat`
The minimum acceptable latency for the region. The default value is 0.

`max_lat`
The maximum acceptable latency for the region. If omitted, the maximum latency is infinite.

`region_or_loop_id`
The ID of the region or loop for which the latency must be specified.

Description

The `constrain_latency` command is used to specify the minimum and maximum acceptable latency for a region. .

constrain_region

A synthesis Tcl command.

Syntax

`constrain_region` *`[-min_lat minlat] [-max_lat maxlat] [-input_delay indel ] [ -max delay maxdel ] _function_or_region_id_or_list_of_`*

Equivalent Directive

HLS_CONSTRAIN_REGION

Parameters

`min_lat`
The minimum acceptable internal register states the generated part can contain.

`max_lat`
The maximum acceptable internal register states the generated part can contain.

`input_delay`
The input delay constraint used when building the resource.

`max del`
The maximum delay at the output of the parts in the same time units used by the project's tech_lib.

`_function_or_region_id_or_list_of_`
The ID of the function or region to be affected.

Description

The `constrain_region` is used to specify latency and timing constraints for the part created for a scheduled region or DpOpt region. The specified regions must also be scheduled or DpOpted. This command is optional, and is used only when specific constraints are required.

See also

- dpopt_region
- HLS_DPOPT_REGION

define_floating_protocol

A synthesis Tcl command.

Syntax

```
define_floating_protocol [ -name name ] [-context _context_] [-setup_time _setup_] [-
delay _delay] [-initiation_interval _ii_] [-flags _flags_] _region_id_
```

Equivalent Directive

HLS_DEFINE_FLOATING_PROTOCOL

Parameters

name

Provides a suffix for the name used in reporting to refer to this block.

`context`
This may be an object ID, a string, or an integer.

`setup_time`
Specifies the time at the end of the first clock cycle of the transaction during which outputs are required to be stable.

`delay`
Specifies the delay after which the transaction's outputs can be assumed to be stable and can be safely read.

`initiation_interval`
Initiation interval for transactions.

`flags`
Flags that affect scheduling of the transaction operations.

`_region_id_`
Specifies the region that is to be defined as a floating protocol.

Description

The `define_floating_protocol` commands defines a block containing a fixed-length I/O protocol that Stratus HLS can schedule in parallel with similar protocols on other interfaces. This command is typically used to define transaction functions, such as memory read or write operations.

define_protocol

A synthesis Tcl command.

Syntax

```
define_protocol [ -name name ] region_id
```

Equivalent Directive

HLS_DEFINE_PROTOCOL

Parameters

name

A string to be used in messages about the protocol region.

region_id

The ID of the region that should be treated as protocol.

Description

Stratus HLS permits the mixture of manually-scheduled behavior containing explicit `wait()` statements with behavior whose schedule will be determined automatically by Stratus HLS. Manually-scheduled behavior is used to maintain cycle accuracy for I/O protocol.

dpopt_region

A synthesis Tcl command.

Syntax

`dpopt_region [ -flags flags ] [ -name name ] function_or_region_or_loop_id_or_list_of`

Equivalent Directive

HLS_DPOPT_REGION

Parameters

flags

Configurations that control how parts are created with DpOpt. Multiple flags can be specified in a Tcl list.

name

The name of the custom datapath component. In other words, the datapath component will be placed in a module named as the specified *name*. *name m*ust be a legal identifier in both C++ and Verilog. The name can be omitted, in which case Stratus HLS automatically creates a name. If the region is a function, the function name will be used. If the region is a loop, the loop name will be used. Otherwise, names will be created using "dpopt_%d" format.

`function_or_region_or_loop_id_or_list_of`

The region or regions that are to have a DpOpt part created for them.

Description

Stratus HLS's datapath optimization technology is controlled by the use of the `dpopt_region` command.

There are several flags that may be included as the config parameter in the command. These flags may be included individually or together in a Tcl list.

The flags are:

- **BEFORE_UNROLL**: Applies DpOpt to a block of code before loop unrolling is performed. This can significantly improve Stratus HLS scalability and runtime by encapsulating large portions of a designs data flow into a single DpOpt part very early in the synthesis process. It has the potential to increase the area of the design because the data flow encapsulated in the DpOpt part before loop unrolling is not available for downstream optimizations such as constant propagation and dead code elimination. However, if you plan to use `NO_ALTS`, it is very likely that `BEFORE_UNROLL` will produce the same results with improved runtime and capacity.

- **DPOPT_DEFAULT**: Specifies the default settings for `dpopt_region`. The default is to produce only for minimum delay (`OPTIM_DELAY`).
- **NO_ALTS**: Implies NO_CONSTANTS | NO_CSE | NO_DCE | NO_TRIMMING.
- **NO_CHAIN_IN**: Disables chaining of additional logic into the inputs of the generated part. Scheduling will place the part at the beginning of a clock cycle with registers immediately in front of the part. The registers will not be included within the generated part and will therefore be available for register sharing in allocation. Part sharing in allocation may result in a mux being placed between the register and the inputs of the generated part.
- **NO_CHAIN_OUT**: Disables chaining of outputs of generated part into additional logic. Scheduling will place the part at the end of a clock cycle with registers immediately following the part. The registers will not be included within the generated part and will therefore be available for register sharing in allocation. Register sharing in allocation may result in a mux being placed between the outputs of the generated part and the register.
- **NO_CONSTANTS**: Instructs DpOpt to not propagate constants into newly created parts. This value is used if there will be multiple instances of the generated block and each will not have the same constant inputs.
- **NO_CSE**: Disables common subexpression elimination in DpOpt parts in order to prevent alternate parts from being created.
- **NO_DCE**: Disables dead code elimination in DpOpt parts in order to prevent alternate parts from being created.
- **NO_TRIMMING**: Instructs DpOpt to not trim input or output port widths of the newly created part. This value is used if there will be multiple instances of the generated block that will use different input/output widths, up to the maximum.

flatten_arrays

A synthesis Tcl command.

Syntax

```
flatten_arrays [ -type flatten_type ] array_id_or_list_of
```

Equivalent Directive

HLS_FLATTEN_ARRAY

Parameters

flatten_type
One of the following:
DEFAULT_FLATTEN: Each array member it treated like a separate variable.

DPOPT_FLATTEN: Specifies that DpOpt parts are to be created for any logic required for variable access to the arrays. However, note that `-dpopt_access` cannot be applied to arrays of signals.

DONT_FLATTEN: Prevents flattening. The arrays will be mapped to a memory from the project's memory library.

`array_id_or_list_of`
The IDs of the arrays to be affected.

Description

The **flatten_arrays** command is used to implement an array as individual variables rather than as a memory or register bank. This command may be applied to any array, whether it is a local array, class member, or global array.

For more information, see HLS_FLATTEN_ARRAY.

invert_dimensions

A synthesis Tcl command.

Syntax

`invert_dimensions` *`array_id_or_list_of`*

Equivalent Directive

HLS_INVERT_DIMENSIONS

Parameters

`array_id_or_list_of`
The IDs of arrays to be processed.

Description

The `invert_dimensions` command inverts the order of the dimensions of a multi-dimensional array for processing by Stratus HLS. A common application is one where inner dimensions will be constant, while outer dimensions will be variable.

map_array_indexes

A synthesis Tcl command.

Syntax

`map_array_indexes` *`option array_id_or_list_of`*

Equivalent Directive

HLS_MAP_ARRAY_INDEXES

Parameters

`option`

The type of mapping to use—either COMPACT or SIMPLE.

`array_id_or_list_of`

The IDs of the multi-dimensional arrays to be affected.

Description

The `map_array_indexes` command allows you to control how multi-dimensional indices are mapped to memory addresses.

map_to_memory

A synthesis Tcl command.

Syntax

`map_to_memory` `[ -`*`mem_type mem_type`*` ] [-ports {<port_num> <port_num> ...}] [-thread <thread>] [ -`*`clock module_io`*` ]` *`array_id_or_list_of`*

Equivalent Directive

HLS_MAP_TO_MEMORY

Parameters

`mem_type`

A string giving the type of the memory to which the array will be bound. The memory must be present in a library referenced in the Stratus HLS project. Optional. If omitted, the array is mapped to a memory that is selected by Stratus HLS.

`port_num`

A single integer, or a Tcl list of integers. Specifies a subset of memory access ports that can be used by the accessing thread or module. Port indexes start at 1, and the order is defined by the order of ports in the memory editor in Stratus IDE. Up to 4 ports may be specified. If no ports are specified, all ports are accessible.

`thread`

This optional parameter specifies a thread for which access will be limited to the given ports. You cannot specify -thread without -ports. If no thread is specified, the command pertains to all threads in the module. This allows different ports to be allocated to different threads. If there are multiple `map_to_memory` commands for the same array (but different threads) they must all specify the same memory type.

`clock`

Specifies a clock other than the thread's clock for the accessing thread that should be bound to the memory.

`array_id_or_list_of`

The IDs of the arrays to be bound.

Description

The `map_to_memory` command allows you to specify that an array is to be mapped to a memory, and to optionally specify which type of memory to use, which memory ports to use, and which clock should be bound to the memory. This is useful when a global array flattening option, like `--flatten_arrays=all` is specified, but it is desired to map a specific array to a memory. It also provides a way to make sure the memory is of the desired area and dimension. The specified memory type must be in one of the parts libraries specified in an `hls_lib` command for your project.

During behavioral synthesis, the impact of array binding directives can be seen in the array disposition reports that immediately follow the allocation report.

If the memory has the "Half Speed" or "Quarter Speed" options selected, you must also specify an alternate clock to connect to the memory in the RTL model. This is done by specifying a`-clock` option to the `map_to_memory` command.

Similar to the `HLS_MAP_TO_MEMORY` directive, the `map_to_memory` command can be used to control which memory ports are used by an individual thread or module. This is useful in the following two scenarios:

1. When multiple threads in the same module access the same array, and that array is mapped to a multi-port memory, you may wish to limit each thread to use a separate port. To accomplish this, a `map_to_memory` command with a `-thread` option should be called for each thread, and a different port number should be specified for each thread.
2. When multiple modules access the same array using the *external arrays* feature, you may wish to restrict each module to accessing a specific port on a multi-port memory. In this case, a `map_to_memory` command should be called without a `-thread` option for each `hls_module`, specifying the required ports in each command.

Note that explicit port specifications are not necessary if the accessing threads or modules naturally align with the capabilities of the ports. For example, if the memory has one write-only-port and one read-only port, and if one thread or module contains only writes, while the other contains only reads, there is no need to specify ports explicitly with `map_to_memory`: Stratus HLS will automatically access only the required ports.

For multiple clock domain designs in which threads with different clocks access the same memory, memory ports should be explicitly allocated to each thread using the `-ports` option. Stratus HLS will automatically connect the appropriate clock to the appropriate memory port based on the clocks used by the accessing threads. If threads from with different clocks access the same memory port, Stratus HLS will issue an error. Note that the *`-clock`* option should *not* be used in this kind of design unless access to the memory is made from only one clock domain.

map_to_reg_bank

A synthesis Tcl command.

Syntax

`map_to_reg_bank` *`array_id_or_list_of`*

Equivalent Directive

HLS_MAP_TO_REG_BANK

Parameters

`array_id_or_list_of`

The IDs of the array to be bound.

Description

The `map_to_reg_bank` command allows you to specify that an array is to be mapped to a dedicated bank of registers.

pipeline_loops

A synthesis Tcl command.

Syntax

`pipeline_loops` `[-type` *`pipe_type]`* *`[ -initiation_interval ii ]`* `[ -`
`name` *`name`* `]` *`loop_id_or_list_of`*

Equivalent Directive

HLS_PIPELINE_LOOP

Parameters

`type`
The pipelining type, either HARD_STALL or SOFT_STALL.

`initiation_interval`
The initiation interval of the pipeline. If unspecified, defaults to 1. Note that the SOFT_STALL option only supports an initiation interval of 1.

`name`
The name of the pipeline for reporting purposes.

`loop_id_or_list_of`
The ID of the loop to be pipelined.

Description

The `pipeline_loops` command tells Stratus HLS to pipeline its enclosing loop. The *name* is a string that Stratus HLS can use to refer to the pipeline during reporting; this name must be unique to the project.

preserve_io_signals

A synthesis Tcl command.

Syntax

`preserve_io_signals` *`behavior_id_or_list_of`*

Equivalent Directive

HLS_PRESERVE_IO_SIGNALS

Parameters

`behavior_id_or_list_of`

The IDs of the behaviors to be affected.

Description

The `preserve_io_signals` command prevents sc_signals variables read or written by a particular `SC_METHOD` from being optimized away when no other process writes of reads those signals. It is useful in applications where a post-processing step will connect something to the signals rather than a a process in the same SystemC model. This is sometimes the case when using the Extra Ports feature of Stratus HLS memory models. The `preserve_io_signals` command also prevents registers associated with both sc_out and sc_signal variables from being shared, or sliced to remove constant bits. Using this directive will produce a simpler mapping between sc_signal and sc_out in SystemC and registers in Verilog.

preserve_signals

A synthesis Tcl command.

Syntax

`preserve_signals` `-always=[yes|no]` *`signal_id_or_list_of`*

Equivalent Directive

HLS_PRESERVE_SIGNAL

Parameters

`-always=[yes|no]`
By default, the value of always is `yes`; therefore, all signals listed will be preserved even if it is unused.

`signal_id_or_list_of`
The ID of an behavior_io that is to be preserved.

Description

The `preserve_signals` command will preserve an sc_signal through the synthesis process. By default, the `-always` parameter is `yes` and the signal(s) will be preserved even if it does not have readers and writers in the BEH model. If you set `-always` to `no` the signal will be retained in favor of other signals asynchronously connected to it, but if it is unusued by the design, it will be removed.
The `HLS_PRESERVE_SIGNAL` directive can also be used with with sc_out and sc_signal variables to prevent the register associated with it from being shared, or sliced to remove constant bits. Using this directive will produce a simpler mapping between sc_signal and sc_out in SystemC and registers in Verilog.

remove_control

A synthesis Tcl command.

Syntax

`remove_control [-type type ] [-name name] region_id`

Equivalent Directive

HLS_REMOVE_CONTROL

Parameters

type
Type are `ON` (enable) or `OFF` (disable) the directive on the block.The default is `ON`.

name
An optional message used for reporting purposes.

region_id
The region to be affected. For the ON option, all branches within the specified region have control removal turned on. For the OFF option, all branches that control the execution of the specified region have FSM-based control.

Description

ON instructs Stratus HLS to turn all of the conditional statements within the block and any nested blocks into datapath control instead of state machine control. `OFF` instructs Stratus HLS to not do this for neither the control statement that immediately enclose this directive, nor control statements that enclose those control statements.

When used in nested statements, the `OFF` setting overrides the `ON` setting.

schedule_region

A synthesis Tcl command.

Syntax

`schedule_region [-flags <`*`options`*`>] [-ii <`*`init_interval`*`>] [-name <`*`name`*`>] <region_ids>`

Equivalent Directive

HLS_SCHEDULE_REGION

Parameters

`options`
Configurations that control how parts are created for the scheduled region. Multiple flags can be specified in a Tcl list.

ii
Specifies the initiation interval. If it is > 0 the scheduled region is pipelined. If it is unspecified, the schedule region has the same initiation interval as the loop from which it is accessed.

`name`
A string that will be used to name the module created for the scheduled region.

`region_ids`
The region or regions that are to have a scheduled part created for them.

Description

The `schedule_region` command provides a way to have Stratus HLS segregate part of a design's functionality for separate processing with minimal modification to the source code structure. The operations in the region are processed by high-level synthesis as if they were in a separate SC_THREAD, producing an independent finite state machine in RTL. This FSM can implement the operations with a fixed latency, with a variable latency, or in a pipelined manner. `schedule_region` allows most constructs supported by Stratus HLS.

Similar to the `dpopt_region` command, `schedule_region` can improve scalability and tool runtimes, and can produce better quality results. It is often used in place of `dpopt_region` when the region has any of the following attributes:

- Contains I/O or modular interface accesses
- Contains accesses to arrays mapped to memories

- Has a variable latency
- Has internal resource sharing opportunities
- Has a long latency
- Has internal loops that you do not wish to unroll

separate_arrays

A synthesis Tcl command.

Syntax

`separate_arrays [ -num_dims` *`ndims`* `]` *`array_id_or_list_of`*

Equivalent Directive

HLS_SEPARATE_ARRAY

Parameters

`ndims`

The number of outer dimensions that will be separated. Optional. Defaults to 1.

`array_id_or_list_of`

The ID of the multi-dimensional arrays to be separated.

Description

The `separate_arrays` command specifies that a multi-dimensional array should be processed as an *array of arrays* mapped to multiple hardware resources instead of being mapped to a single hardware resource with a single dimension. The directive is useful for both arrays mapped to memories, and arrays mapped to register banks, but is, in general not applicable to flattened arrays. By associating a multi-dimensional array with multiple hardware resource, you can increase the bandwidth for access.

set_clock_period

A synthesis Tcl command.

Syntax

`set_clock_period` `-period` *`clock_period`* *`clock_port`*

Equivalent Directive

HLS_SET_CLOCK_PERIOD

Parameters

`clock_period`

The period that Stratus should assume for the specified clock.

`clock_port`

The name of an sc_in or sc_in_clk for which an alternate clock period is to be set. The name of the port can ordinarily be specified, but if the current virtual directory is not the hls_config being synthesized, a *module_io* object path must be specified.

Description

The `set_clock_period` is used primarily with Clock Domain Crossing (CDC) circuits to specify a period for a clock that is different to the period specified to Stratus HLS for processing of the module. Stratus HLS uses the specified clock period when performing timing analysis on individual `SC_CTHREAD`s and `SC_METHOD`s. It also uses the clock period as a constraint when selecting and constructing datapath components. In a design that has multiple clock domains, some threads and methods may require one clock, while other threads and methods require another clock.

set_cycle_slack

A synthesis Tcl command.

Syntax

`set_cycle_slack` `-slack` *`slack_time`* *`behavior_id_or_list_of`*

Equivalent Directive

HLS_SET_CYCLE_SLACK

Parameters

slack_time
The amount of slack time subtracted from the available clock period for the given behaviors. The available time for scheduling operations is calculated as `clock_period - cycle_slack`: positive values make less time available for scheduling operations,and negative values make more time available. Units are the units used by the `tech_lib`.

`behavior_id_or_list_of`
The IDs of behaviors whose cycle slack is to be set.

Description

The `set_cycle_slack` command specifies an alternate cycle slack value to be used for a specific thread. This overrides the default cycle slack value set by the `cycle_slack` synthesis control attribute.

set_default_input_delay

A synthesis Tcl command.

Syntax

`set_default_input_delay` `-delay` *`input_delay`* *`behavior_id_or_list_of`*

Equivalent Directive

HLS_SET_DEFAULT_INPUT_DELAY

Parameters

`delay`

The delay that will be reserved at the start of clock cycles before inputs are read. The same time units used by the project's tech_lib are assumed.

`behavior_id_or_list_of`

The IDs of the behaviors whose default input delay is to be set.

Description

This is a mechanism for setting an alternate default input delay for a specific thread. It takes precedence over the global `default_input_delay` attribute. The `default_input_delay` applies only to module inputs and not inter-thread signals.

set_default_output_delay

A synthesis Tcl command.

Syntax

`set_default_output_delay` `-delay` *`output_delay`* *`behavior_id_or_list_of`*

Equivalent Directive

HLS_SET_DEFAULT_OUTPUT_DELAY

Parameters

`output_delay`

The delay amount in the same time units used by the project's tech_lib.

`behavior_id_or_list_of`

The IDs of the behaviors whose default output delay is to be set.

Description

The `set_default_output_delay` command specifies the delay that will be used by default for any asynchronous output, unless an `HLS_SET_OUTPUT_DELAY` directive or `set_output_delay` command has been specified for the output. The *output_delay* gives the time at the end of the clock cycle that must be reserved for the readers of output signals. That is, it specifies the setup time of the downstream module. Stratus HLS will ensure that the output becomes stable in time to meet this requirement.

See also

- set_default_output_options
- set_output_delay

set_default_output_options

A synthesis Tcl command.

Syntax

`set_default_output_options` `-options options` *`behavior_id_or_list_of`*

Equivalent Directive

HLS_SET_DEFAULT_OUTPUT_OPTIONS

Parameters

`options`
The option specifying the default handling of outputs from the thread or module. Multiple options may be specified using a Tcl list.

`behavior_id_or_list_of`
Specifies the IDs of the behaviors to be affected.

Description

The `set_default_output_options` command specifies the timing semantics for outputs where no `set_output_options` command or `HLS_SET_OUTPUT_OPTIONS` directive is given.

See also

- set_default_output_delay
- set_output_options

set_input_delay

A synthesis Tcl command.

Syntax

`set_input_delay` `-delay` *`input_delay`* *`signal_id_or_list_of`*

Equivalent Directive

HLS_SET_INPUT_DELAY

Parameters

`input_delay`
The delay amount in the same time units used by the project's tech_lib.

`signal_id_or_list_of`
The ID of a behavior_io whose input delay is to be set.

Description

This command specifies the delay after the rising clock edge at which Stratus HLS will assume data on the specified input to be valid. Stratus HLS will attempt to chain this port directly into a functional unit without introducing a register. This setting overrides any default input delay specified.

set_is_bounded

A synthesis Tcl command.

Syntax

`set_is_bounded` *`signal_id_or_list_of`*

Equivalent Directive

HLS_SET_IS_BOUNDED

Parameters

`signal_id_or_list_of`
The ID of a behavior_io that is to be bounded.

Description

The `set_is_bounded` command specifies that one or more behavior I/O's can be safely accessed outside of a protocol region. It is called a *bounded* value because I/O reads and writes are restricted to take place between boundaries elsewhere in the thread. Each protocol region defines a boundary. So, while reads and writes do not need to appear inside a protocol region, limitations are placed on where the I/O can be scheduled based on surrounding protocol regions.

set_output_delay

A synthesis Tcl command.

Syntax

`set_output_delay -delay output_delay signal_id_or_list_of`

Equivalent Directive

HLS_SET_OUTPUT_DELAY

Parameters

`output_delay`
The double that specifies the external delay (setup time) on an async output. The units of the delay are the same as the units of the `tech_lib` being used to process the design.

`signal_id_or_list_of`
The ID of a behavior_io whose output delay is to be set.

Description

The `set_output_delay` command controls the way Stratus HLS specifies the required delay for the output. This command is only relevant when either the `set_output_options` or `set_default_output_options` command has been used to specify that the output is asynchronous using one of the `ASYNC_*` codes. The *delay* gives the time at the end of the clock cycle that must be reserved for the reader of the signal. That is, it specifies the setup time of the downstream module. Stratus HLS will ensure that the output becomes stable in time to meet this requirement.

set_output_options

A synthesis Tcl command.

Syntax

`set_output_options -options options signal_id_or_list_of`

Equivalent Directive

HLS_SET_OUTPUT_OPTIONS

Parameters

options
The option specifying the timing semantics options for the output. Multiple options may be specified using a Tcl list.

signal_id_or_list_of
The ID of a behavior_io to be affected.

Description

The `set_output_options` command controls the way Stratus HLS specifies the kind of circuit Stratus HLS will build for an output. The allowable values are:

- **SYNC_HOLD (default)**: The output will be registered, and its value will be held stable between writes to the output. This is the option used if no other options are specified.
- **SYNC_NO_HOLD**:The output will be registered, but its value will only be held stable for a single cycle, or longer if that cycle is extended by a pipeline stall. This places restrictions on the protocol used when writing the output, but it also makes the output register available to be shared during other cycles, potentially reducing design area.
- **ASYNC_HOLD**: The output will be updated asynchronously, but its value will be held stable between assertions. A backing output register will be used to guarantee the stability of the output unless the output is written in every state.
- **ASYNC_NO_HOLD**: The output will be updated asynchronously, and its value is only guaranteed to remain stable for one cycle. An output register will never be used for this output.
- **ASYNC_STALL_NO_HOLD**: The output will be updated asynchronously, and its value is only guaranteed to remain stable for one cycle. However, if a pipeline stall occurs during that cycle, the value will be held stable during the stall. An output register will only be used to guarantee stability during stalls in cases where there is an asynchronous path from an input to

the output.

Note that when one of the `ASYNC_*` options is used on an output.

set_stall_value

A synthesis Tcl command.

Syntax

`set_stall_value -value` *`value signal_id`*

Equivalent Directive

HLS_SET_STALL_VALUE

Parameters

`value`

An integer value specifying the value that the given signal will be set to during a stall.

`signal_id`

The ID of a behavior_io that is to be affected.

Description

The `set_stall_value` command specifies a constant value that should be written to an output port or internal signal during a pipeline stall condition. A pipeline stall is inferred from a busy loop as described in "Pipeline stalling" in the *User Guide*.

Note that there is no mechanism for defining a *non-global* stall value using the set_stall_value command. The `HLS_SET_STALL_VALUE` directive must be used to define local stall values.

split_add

A synthesis Tcl command

Syntax

```
split_add [-max_width <input_width>]  region_id
```

Equivalent Directive

HLS_SPLIT_ADD

Parameters

input_width

An integer value specifying the maximum split width. Optional parameter

region_id

The region in which adds should be split

Description

The `split_add` command specifies a constant value that is used to split adders/subtractors having greater bit-width than `input_width` so that operand width will be within the `input_width size`. Only add operations in the specified region are affected.

split_multiply

A synthesis Tcl command.

Syntax

```
split_multiply [-max_width input_width]  region_id
```

Equivalent Directive

HLS_SPLIT_MULTIPLY

Parameters

`input_width`
An integer value specifying the maximum split width. Optional argument.

region_id
The region in which multiplies should be split

Description

The `split_multiply` command specifies a constant value that is used to split multipliers having greater bit-width than `input_width` so that operand width will be within the `input_width size`. Only multiply operations in the specified region are affected.

unroll_loops

A synthesis Tcl command.

Syntax

```
unroll_loops [-type type ] [-count count ] [-name name ] loop_id_or_list_of
```

Equivalent Directive

HLS_UNROLL_LOOP

Parameters

type
The unrolling type: `ON`, `ALL`, `OFF`, `COMPLETE`, `AGGRESSIVE`, or `CONSERVATIVE`. The default type is `ON`.

`count`
The number of times the loop body should be replicated. If omitted, the loop will be unrolled fully if the number of loop iterations can be automatically calculated.

name
A unique name for the unroll used for reporting purposes.

loop_id_or_list_of
The IDs of the loops that are to be affected.

Description

The `unroll_loops` command can completely or partially unroll a loop, resulting in multiple hardware instances in the cycle-accurate design generated by Stratus HLS. Unrolling a loop is a powerful transformation that allows parallelism to be exploited in the design.

Scripting Utility Commands

This section lists all the scripting utility commands that can be used in Stratus HLS.

Scripting Utility Commands	**Description**
`close_project`	Close the currently open project.
`cont`	During interactive runs of stratus_hls, tells the tool to run until the next access point.
`filter`	Filter a list of objects based on the value of an attribute
`find`	Find a single or multiple objects for a name pattern
`get_attr`	Returns the value of the attribute.
`get_fanin`	Returns input connections in a CFG, DFG, or netlist.
`get_fanout`	Returns outgoing connections in a CFG, DFG, or netlist.
`get_hls_config`	Returns the object ID of the hls_config currently being synthesized.
`get_hls_module`	Returns the hls_module object currently being synthesized.
`get_install_path`	Returns the filepath to the installation directory of the running Stratus HLS executable.
`get_phase`	Returns a string describing which phase of the HLS process is currently being executed.
`get_project`	Returns the object ID of the current project.
`get_stacks`	Returns a collection of call stacks for various design objects.
`get_timing_paths`	Returns a collection of timing paths described by a collection of nodes in each timing path.
`get_version`	Returns a list of version numbers in the following order: major release, minor release, build number.
`help`	Returns information about the specified command.

`list_attr`	Returns the list of attribute names of the specified object.
`load_results`	Loads results for the specified snapshot or hls_config to make them available for analysis.
`make_link`	Creates a text string that is recognized as markup in a Stratus report.
`open_project`	Opens the project defined by the given project.tcl file.
`quit`	Terminates a stratus_hls run at an access point.
`report`	Print one of several reports.
`run`	Executes the given config.
`set_attr`	Sets the value of an object's attribute. If no object is specified, the current scope is used.
`trim_paths`	Shortens file system paths and object paths to make them more readable.
`un_link`	Returns a string without annotations created by `make_link`.
`unload_results`	Unloads any results loaded for the specified hls_config or snapshot.
`vcd`	Sets the current virtual directory for identifying objects by a relative object ID.
`vls`	Lists information about a virtual directory but it does not return a value.
`vpwd`	Returns the current virtual directory

close_project

A scripting utility command.

Syntax

```
close_project
```

Description

Closes the currently open project. The command is not strictly required before existing a script or opening a different project, but if a project is only needed temporariliy by the script, `close_project` provides a way to return memory.

cont

A scripting utility command.

Syntax

`cont [-through` *phase*`]`

Parameter

`[-through` *phase*`]`

Specifies the phase through which to run.

Description

The `cont` command is only useful during interactive runs of stratus_hls. With no options, it will run until the next snapshot point, or the tool exit. If a `-through` option is given, it will run through that phase and break.

filter

A scripting utility command.

Syntax

`filter attrib_name value_pattern object_path_list` *`search_options`*

Parameters

`attrib_name`

The name of an attribute to filter on.

`value_pattern`

A pattern to match in each attribute

`object_path_list`

A list of object paths to be filtered

`search_options`

- `-func` *`funcname`*: In a function with the given name, or a function called from function with that name. Any object where the `call_stack` command would include a function whose name matches the specified name is returned. Unlike `call_stack`, the behavior function name is included in the search. That means that any object either directly in the function, or in a function called from the function is returned. Wildcards are supported.
- `-if` expr: Evaluates an expression for each candidate as described in the examples below.
- `-omit`: Objects that match the specification are omitted from the results.
- `-relatives_in` *`snapshot_name`*: For each object found, finds any associated object in the named snapshot.
- `-return` *`attrib_expression`*: Returns not the object found but an attribute value as specified from each object found.
- `-source` *`file`*`[:line[:column]]`: An item with a source location at this file[:line[:column]]. This will find things based on source position. Any subset of the *`file`*`:line:column` can be specified, and any items with matching src_links will be included in the result. There is no support for wildcards or line/column ranges.
- `-unique`: No duplicates are returned.
- `-within` *object_type*: Operations hierarchically within the specified object. Accepts behavior, loop, region, or basic_block.

Description

The filter command is usually used together with the find command to filter a set of objects by the value of one of their attributes. Given an attribute name, a pattern to match with the values of that attribute, and a list of objects to read the attribute from, the filter command returns a subset of the list of objects containing only those objects that match the pattern.

For example, in order to find all behavior objects that are threads rather than methods or functions, the filter command can be used as follows:

set threads [filter is_thread true [find -behavior *]]

Another use is to find objects by the the names they had in the original source code. Each object has a unique name within its scope, and that name must be a legal C++ identifier. However, the original name given to an object in the source code is saved. The uniquified, filtered name is available in the 'name' attribute for the object, and the original name is available in the original_name attribute. Objects can be found through their original name through the `filter` command. This command finds all objects for a given attribute value. For example, to find all loops that are named `FOR1`:

```
set loops [filter original_name FOR1 [find -loop *]]
```

See also find for ***`search_options`*** **Usage Examples**.

find

A scripting utility command.

Syntax

`find [-ignore_case] [-root path] [-object_type] [name_pattern]` *`search_options`*

Parameters

`[-ignore_case]`

Ignores case when matching root path or name pattern.

`[-root path]`

Specifies the starting directory for the search (the default is the current working directory or the current design). You may use the `*` and `?` wildcards; however, the `/` separator is not matched with wildcards.

`[-object_type]`

Specifies the type of object for which you want to search: hls_module, hls_config, array, behavior, behavior_io, region, loop, module_io, basic_block, op, resource, snapshot, lib, sc_instance, rtl_module, rtl_instance, rtl_signal, rtl_process, lib_module, lib_module_io, message, report, sim_config, logic_synth_config, analysis_config, power_config, equiv_config, * (all objects).

`[name_pattern]`

Specifies the glob-style name pattern for matching object names. Defaults to `*` if not specified. You may use the `*` and `?` wildcards; however, the `/` separator is not matched with wildcards. Stratus HLS tries to match `name_pattern` to objects assuming scopes are specified in the pattern. If no objects are found, Stratus HLS tries to match objects assuming scopes are not specified in the pattern.

`search_options`

- `-func` *`funcname`*: In a function with the given name, or a function called from function with that name. Any object where the `call_stack` command would include a function whose name matches the specified name is returned. Unlike `call_stack`, the behavior function name is included in the search. That means that any object either directly in the function, or in a function called from the function is returned. Wildcards are supported.
- `-if` expr: Evaluates an expression for each candidate as described in the examples below.
- `-omit`: Objects that match the specification are omitted from the results.
- `-relatives_in` *`snapshot_name`*: For each object found, finds any associated object in the named snapshot.
- `-return` *`attrib_expression`*: Returns not the object found but an attribute value as specified

from each object found.

- `-source` *`file`*`[:line[:column]]`: An item with a source location at this file[:line[:column]]. This will find things based on source position. Any subset of the *`file`*`:line:column` can be specified, and any items with matching src_links will be included in the result. There is no support for wildcards or line/column ranges.
- `-unique`: No duplicates are returned.
- `-within` *object_type*: Operations hierarchically within the specified object. Accepts behavior, loop, region, or basic_block.

Description

The find command is used to find objects based on their type, and their names. The search begins at either the current virtual directory, or the directory specified in the -root option. An object is matched if it is below the starting point for the search in object tree, matches the type specified by -op, and has a name that matches the specified pattern. A list of full object paths for the matched objects is returned. These objects can then be accessed with other Tcl functions to query their state, or make settings.

For example, in a uarch_tcl proc, the following script will cause all loops to be unrolled:

```
foreach loop [find -loop *] {
unroll_loops $loop
}
```

This example relies on the current virtual directory to start the search, which is automatically set to the directory containing the description of the design contents by Stratus. The following example prints the names of all regions in each thread in the module.

```
foreach beh [find -behavior *] {
  if { [get_attr is_thread $beh] } {
    puts "Thread [get_attr name $beh]"
    foreach region [find -region * -root $beh] {
      puts " Region [get_attr name $region]"
    }
  }
}
```

Behavior objects include both threads and methods, so we need to use the is_thread attribute on the behavior objects returned from find so that we report on only the threads. This shows that the values returned from the find command are object identifiers that can be used with other Tcl commands that accept objects, like get_attr. Notice that the second find, the one that finds regions in each thread, specified -root $beh. This causes only regions within that behavior to be returned.

The find command is often combined with the filter command to search for things using an attribute on an object, and not just its name. So, for example, we can simplify the script above using filter as follows:

```
foreach beh [filter is_thread true [find -behavior *]] {
  puts "Thread [get_attr name $beh]"
  foreach region [find -region * -root $beh] {
    puts " Region [get_attr name $region]"
  }
}
```

The filter command accepts an attribute name, a pattern matched against values of that attribute, and a list of objects on which to test that attribute value. We use the find command to supply that list. So, the find command returns all behaviors, then the filter command returns on the behaviors for which the is_thread attribute has the value true.
The find command can also be used to find things based on a name pattern. For example, if we wanted to find all of the regions that begin with the string "DP", we could used:

```
find -region DP*
```

The original_name attribute is often a useful way or finding thing based on the name specified in the source code. Each object has a unique name that is often based on an associated name in the source code. In cases where multiple objects were given the same name in the source code, each is automatically given a uniquified name. This can make things hard to identify with find. The original_name attribute stores the specified name, so, in the following situation:

```
IO_LOOP:
for ( ... ) {}

...

IO_LOOP:
for ( ... ) {}
```

we have 2 loops, perhaps in different parts of the code, that both have the label IO_LOOP. Stratus will give one the name IO_LOOP, and the other the name IO_LOOP_1. You can find them both using their original names as follows:

```
filter original_name IO_LOOP [find -loop *]
```

search_options Usage Examples

`-if` expr

The **`-if`** option is a tool for evaluating matches based on attribute values other than `name`. The `-if` option supports expressions of the form:

```
{attrib_name} [rel_op [value_expr]]
```

where:

- `{attrib_name}`: specifies a chain of one or more attribute names, separated by '.'. The value of the last attribute in the chain is used to evaluate the option, and the value of each preceding attribute is used to select an object stored in an attribute of that name.
- *rel_op*: is one of =, =, <, <=, >, >=.
- *value_expr*: is a value of the type of the last attribute in the chain or a glob.

For example, to find operations whose `kind` attribute is `mul`, you could use:

```
find -op * -if { .kind == "mul" }
```

or you can use the chaining feature to find operations implemented on modules while name begins with Mul:

```
find -op * -if { .rtl_instances.module.name == "Mul*" }
```

To find operators where the output width of the operator is >= 32:

```
find -op * -if {.width >= 32}
```

If multiple -if expressions are specified, they are 'anded'. For example, to find all array writes that do not have constant addresses:

```
find -op * -if {.kind == array_write} -if {.predict_const == false}
```

If the comparison operator and value are omitted, it requires only that the attribute be set. For example, to return only rtl_instances where there is an instance's module:

```
find -rtl_instance * -if { .module }
```

When evaluating attribute chains, each non-final attribute must contain object identifiers. If an attribute contains multiple object IDs, all of them are evaluated until one matches the expression. If any value matches the expression, the condition is satisfied.

-omit

The **-omit** option changes the behavior of both `find` and `filter` such that matched objects are not included in the results. The `[-object_type]` argument interacts with `-omit` such that if `*` is specified, all objects are still matched, but if any other value is given, only non-matching objects are matched. The `-omit` option does not affect the `-return` or `-relative_in` options, which are applied after matching to change the result.

For example:

```
find -op * -source dut.h -omit
```

will find all ops, but omit those whose source file is dut.h. This is equivalent to:

```
find -region "dpopt*" -omit
```

that finds all regions whose names do not begin with "dpopt."

-relatives_in *snapshot_name*

The `-relatives_in` option finds an related object or list of objects in a snapshot. For example, if `-op` is used in the `rtl` snapshot, then `find -op $opname -relatives_in elab`, or `filter -relatives_in elab$op` will find the original operation in the `elab` snapshot from which `$op` was derived. If the specified snapshot is not yet loaded, it will be loaded implicitly if the snapshot exists.

Similarly, if `$op` is an operation in a loaded `elab` snapshot, `filter -relatives_in rtl $op` will find the ops from the final result that originated from `$op`.

> `-snapshot` (not `-in_snapshot`) is also a valid option since it is used to find snapshots themselves.

-return *attr_expression*

The `-return` option allows values other than the object found to be returned. For example, to return a unique list of all module names used in rtl_instances in the design:

```
find -rtl_instance * -return .module.name -unique
```

-within *objects*

This is designed to operate similarly to the Design Contents view in Stratus IDE. It accepts several kinds of hierarchical containers that are represented as objects in snapshots. This can include behaviors, loops, regions, or basic_blocks. Items like this have nesting that is not represented in the object hierarchy. For example, consider the following loop nest:

```
OUTER:
for ( int i=0; i < N; ++ {
  val1 = din.get();
  for ( int j=0; j < M; j++ ) {
    val2 = din.get();
     {HLS_DPOPT_REGION("compute");
      reg = val1 * const + val2;
```

```
        }
      }
    }
```

Executing `find -op * -within [find -loop OUTER]` will return all of the ops in the loop nest. while executing `find -op * -within [find -region compute]` will return the `*` and `+` in the dpopt block.

If multiple objects are given, an object will match if it is within only one of them. This is useful for applications using wildcards. For example:

```
find -op * -within [find -region "calc_part*"]
```

will find all ops within any instance of a dpopted region whose name begins with `calc_part`.

get_attr

A scripting utility command.

Syntax

```
get_attr attribute_name [-unique] [object_ids]
```

Parameters

`attribute_name`

Specifies the name of the attribute whose value you want to retrieve. May be an *attribute chain*, separated by `'.'`.

`[-unique]`

If multiple strings are returned, either because there are multiple `object_ids`, or because the attribute has multiple values, duplicate values are removed from the result.

`[object_id]`

Identifies one ore more objects for which the value of the specified attribute is to be retrieved. If multiple objects are specified, the attribute value on each object is returned in a list.

Description

Returns the value of the attribute.

For a project `my_project`, to get the value of the attribute `cc_options`, you would use:

```
get_attr cc_options project:my_project
```

To store this in a Tcl variable named `cf`, you would use:

```
set cf [get_attr cc_options project:my_project]
```

`attribute_name` also accepts attribute chains as is supported with the `find` command's `-if` and `-return` options. This supports attributes that reference other objects to be accessed without nesting get_attr commands. For example:

```
    # Get the multipler op on line 47.
    set mul [find -op -source dut.cpp:47 -if {.kind == mul}]

    # Print the name of the module its implemented on.
    puts "Multiplier at [get_attr src_links $mul] implemented on module [get_attr
    rtl_instances.module.name $mul]"
```

get_fanin

A scripting utility command.

Syntax

`get_fanin [-no_loopback]  [-loopback_only] [-ancillary_only] [-no_ancillary] [-port` *num*`] [-exclude` *exclude_objects*`]` *base_object_or_list_of*

Parameters

`[-no_loopback]`

This means only forward arcs in the graph should be followed. No LCD connection in DFG, no loop restarts in CFG, and no self connections in netlists will be followed.

`[-loopback_only]`

This means only loopback connections should be followed. Its main value is to check whether it returns `unset`, meaning there are none.

`[-ancillary_only]`

Includes only ancillary connections.

`[-no_ancillary]`

Omits connections that are not for transmitting dataflow or control. For DFG, these are connections present only to keep operators in order, like links between array operations. For netlist, these are clock and reset connections. Order-only connections are never classified as loopback connections.

`[-port` `num`]

Gives connections for the given port index.

`[-exclude` *`exclude_objects`*`]`

The specified objects are excluded from the results. This can be used as a simple mechanism of avoiding infinite recursion in scripts that follow dataflow through multiple layers.

`base_object_or_list_of`

One or more objects from which to start the search. Must not mix objects from different graphs.

Description

The intent of the `get_fanin` and `get_fanout` functions is to provide a simple mechanism for graph traversal that does not require dealing with the details of ports, bit ranges, etc. This omits some details, but makes it much easier to use.

The `get_fanin` and `get_fanout` functions are used to traverses directed graphs. The three kinds of directed graphs in the Stratus object model are:

- DFG: Nodes are `op` of `behavior_io`. Returns `op`.
- CFG: Nodes are `basic_block`. Returns `basic_block`.
- Netlist. Nodes are:
    - `rtl_instance`: Returns the `rtl_instance` or `rtl_process` that reads or writes the connected wire, or the non-wire `rtl_signal` that is connected.
    - `rtl_signal`: For `get_fanin` of signals written by an `rtl_process`, returns the inputs to the `rtl_process`. If access to the `rtl_process` is required, use the `rtl_signals_writer` attribute. Otherwise, returns the `rtl_instance` or `rtl_process` that reads or writes the signal.
    - `rtl_process`: Returns the readers or writers of the `rtl_signals` read or written by the rtl_process.

When traversing netlists, `rtl_signal` objects representing asynchronous wires are, in general, not returned, but instead the readers or writers of those wires are returned. The emphasis is on identifying connected non-wire objects.

When traversing DFGs, ops with kind==wire are never returned: the op on the other end of the wire is returned. Ops with kind==reg are also never returned because they are not part of the dataflow graph. If the output of an `op` is stored in a register, the `reg_op` attribute on the `op` gives access to the operation that describes storage of in a register. If an `op` with kind==reg is given to `get_fanin` or `get_fanout`, the connections for the associated `op` are returned.

The command accepts either one object of these types, or a set of objects. The return value is another set of objects from the same graph. All objects must be from the same graph.

get_fanout

A scripting utility command.

Syntax

`get_fanout [-no_loopback] [-loopback_only] [-ancillary_only] [-no_ancillary] [-exclude` *exclude_object*`]` *base_object_or_list_of*

Parameters

`[-no_loopback]`

This means only forward arcs in the graph should be followed. No LCD connection in DFG, no loop restarts in CFG, and no self connections in netlists will be followed.

`[-loopback_only]`

This means only loopback connections should be followed. Its main value is to check whether it returns `unset`, meaning there are none.

`[-ancillary_only]`

Includes only ancillary connections.

`[-no_ancillary]`

Omits connections that are not for transmitting dataflow or control. For DFG, these are connections present only to keep operators in order, like links between array operations. For netlist, these are clock and reset connections. Order-only connections are never classified as loopback connections.

`[-port` `num`]

Gives connections for the given port index.

`[-exclude` *`exclude_objects`*`]`

The specified objects are excluded from the results. This can be used as a simple mechanism of avoiding infinite recursion in scripts that follow dataflow through multiple layers.

`base_object_or_list_of`

One or more objects from which to start the search. Must not mix objects from different graphs.

Description

The intent of the `get_fanin` and `get_fanout` functions is to provide a simple mechanism for graph traversal that does not require dealing with the details of ports, bit ranges, etc. This omits some details, but makes it much easier to use.

The `get_fanin` and `get_fanout` functions are used to traverses directed graphs. The three kinds of directed graphs in the Stratus object model are:

- DFG: Nodes are `op` of `behavior_io`. Returns `op`.
- CFG: Nodes are `basic_block`. Returns `basic_block`.
- Netlist. Nodes are:
    - `rtl_instance`: Returns the `rtl_instance` or `rtl_process` that reads or writes the connected wire, or the non-wire `rtl_signal` that is connected.
    - `rtl_signal`: For `get_fanin` of signals written by an `rtl_process`, returns the inputs to the `rtl_process`. If access to the `rtl_process` is required, use the `rtl_signals_writer` attribute. Otherwise, returns the `rtl_instance` or `rtl_process` that reads or writes the signal.
    - `rtl_process`: Returns the readers or writers of the `rtl_signals` read or written by the rtl_process.

When traversing netlists, `rtl_signal` objects representing asynchronous wires are, in general, not returned, but instead the readers or writers of those wires are returned. The emphasis is on identifying connected non-wire objects.

When traversing DFGs, ops with kind==wire are never returned: the op on the other end of the wire is returned. Ops with kind==reg are also never returned because they are not part of the dataflow graph. If the output of an `op` is stored in a register, the `reg_op` attribute on the `op` gives access to the operation that describes storage of in a register. If an `op` with kind==reg is given to `get_fanin` or `get_fanout`, the connections for the associated `op` are returned.

The command accepts either one object of these types, or a set of objects. The return value is another set of objects from the same graph. All objects must be from the same graph.

get_hls_config

A scripting utility command.

Syntax

```
get_hls_config [-name]
```

Parameters

```
[-name]
```

Specifies the name of the hls_config whose value you want to retrieve.

Description

Returns the object ID of the hls_config currently being synthesized. The full object path is returned by default, and only the leaf name is returned if the -name option is specified. The -name option makes this command easy to use in conditionals.

get_hls_module

A scripting utility command.

Syntax

`get_hls_module [-name]`

Parameter

`[-name]`

Returns the hls_module object currently being synthesized. By default, the full object path is returned. However, if `-name` is specified, then only the leaf name is returned, to make it easy to use in conditionals.

get_phase

A scripting utility command.

Syntax

```
get_phase
```

Description

Returns a string describing which phase of the HLS process is currently being executed or has just completed. Values are:

- **elab**: The elaboration phase of an HLS job.
- **optim**: The optimization phase of an HLS job.
- **post_run**: After an HLS job has completed, but before the tool has exited, no matter what stage it reached.
- **project**: The project file is being executed to define the project's contents.
- **report**: During a `report` or `rtl_annotation` command.
- **rtl**: After RTL has been produced by an HLS job.
- **sched**: The scheduling phase of an HLS job.
- **setup**: The setup phase of an HLS job.
- **shell**: HLS is not running. This is the state when a script is being executed by bdw_shell.

get_install_path

A scripting utility command.

Syntax

```
get_install_path
```

Description

Returns the filepath to the installation directory of the running Stratus HLS executable.

get_project

A scripting utility command.

Syntax

```
get_project
```

Description

Returns the object ID of the current project.

get_stacks

A scripting utility command.

Syntax

```
get_stacks [options] [object_ID_or_list_of]
```

where the object IDs may be any object that has a `src_links` attribute, which includes most objects that are members of snapshots.

Parameters

`[options] may be one or more of:`

- `-common`: Returns the portion of the stack that is common to all src_links for all objects.
- `-no_ip`: Omits any portions of stacks that are in IP source files, like cynw_fixed.h.
- `-omit_tail`: Omits the final location that is in src_links so that only the function calls are returned.
- `-single`: Returns only a single stack for each object, even if there is more than one available.

Description

The `get_stacks` command returns a list of call stacks, each of which is a list of four-element, comma-separated call stack node records. In many simple cases, `get_stacks` returns only a single call stack.

Each call stack node has the format:

```
<function>,<,file>,<line>,<column>
```

For example, consider the following code:

```
140: sc_uint<8> addone( sc_uint<8> val ) {
141: return val + 1;
142: }
143: sc_uint<8> compute( sc_uint<8> val ) {
144: return addone( val * 3 );
145: }
146: void thread() {
  ...
175: rslt1 = compute(din1);
176: rslt2 = compute(din2);
  ... }
```

The call stack for the addition operator in the `addone` function would look something like:

```
{ thread0,dut.cpp,175,12 compute,dut,cpp,144,14 addone,dut.cpp,141,16}
```

There are two addition operators after function inlining, and each has its own call stack. The second addition operator would have the following call stack:

```
{ thread0,dut.cpp,176,12 compute,dut.cpp,144,14 addone,dut.cpp,141,16}
```

Note that both addition operators would have the same `src_links` attribute value of `dut.cpp:141:16`. That is, they can be distinguished by their calls stacks. The `src_links` entry is included in the call stack by default, but if the `-omit_tail` option is specified, it is left out so that only locations of function calls are included. For example:

```
{ thread0,dut.cpp,176,12 compute,dut.cpp,144,14}
```

Call stacks can be iterated over as Tcl lists, and each list element can be analyzed using the Tcl `split` command to produce a 4-element list. For example:

```
foreach stack [get_stacks $op] {
  foreach frame [split $stack ,] {
    puts "Frame: func=[lindex $frame 0], file=[lindex $frame 1], line=[lindex $frame
2], col=[lindex $frame 3]"
  }
}
```

> Unlike the `src_links` attribute, commas are used as delimiters rather than colons since function names may contains colons.

For some objects, the `src_links` attribute contains more than one entry. For example a dpopt op would have `src_links` for all ops contained in it. In this case, a separate stack is returned by `get_stacks` for each entry in src_links. This results in a "list of lists" being returned by `get_stacks`.

If multiple objects are given as options, then stacks will be returned for all of them. In this case, a "list of lists" will be returned as well.

The list of lists will never contain duplicates.

> The size of the returned list is not necessarily the same as the number of nodes passed to `get_stacks`. If you have a list of nodes and need to correlate stacks with those nodes, a separate call to `get_stacks` must be made for each node.

The `-common` option returns only the leading portion of each call stack that is common. In our example above, that would be an empty list since the first entry is different for each. However, if `get_stacks` is called for both the `*` in `compute()` and the `+` in `addone()`, then -common would result in only the following being returned:

```
{ thread0,dut.cpp,176,12 compute,dut.cpp,144,14}
```

get_timing_paths

A scripting utility command.

Syntax

`get_timing_paths [`*`options`*`] [`*`hls_config_object`*`]`

Parameters

The *options* may be one or more of:

- `-from`: An rtl_instance, rtl_signal, or rtl_process from which the path originates.
- `-n`: The max number of paths to report.
- `-stage`: one of sched, alloc or rtl.
- `-through`: An rtl_instance, rtl_signal, or rtl_process through which the path goes.
- `-to`: An rtl_instance, rtl_signal, or rtl_process at which the path terminates.

Description

The value returned is a "list of lists," where each element in the outer list is a timing path, and each inner list contains strings that describe nodes in a timing path. The order of each inner list is the order of the timing path. Each timing path node is encoded in a string that contains six comma-separated fields:

1. A human-readable identifier for the timing path node.
2. A kind code that is one of port, resource, wire, always, mux, const, or reg.
3. A unit delay that this node adds to the path as a floating point number.
4. A cumulative delay to this point in the path.
5. A detail string describing something about the node. For example, 'state' for a register.
6. An object ID corresponding to the node.

These fields correspond to the columns in the report from `bdw_report_long_paths`, and the content should be expected to be the same among equivalent uses of `get_timing_paths` and `bdw_report_long_paths`, which should both be equivalent to timing paths reported by Stratus IDE.

Some fields may be omitted:

- There may not always be an object ID available.
- There may not always be a detail string available.

The cumulative delay is provided only for convenience. The cumulative delay of a list entry is always the sum of the delays of all previous entries.

The following example demonstrates how the get_timing_paths command can be used:

```
set paths [get_timing_paths -n 10]
set ipath 0
set fmtStrStr " %-50s %-10s %10s %10s %-8s"

foreach path $paths {
    puts "Path #$ipath"
    foreach node $path {
      set fields [split $node ","]
      set id [lindex $fields 0]
      set kind [lindex $fields 1]
      set delay [lindex $fields 2]
      set cumm [lindex $fields 3]
      set detail [lindex $fields 4]
      set obj [lindex $fields 5]
      puts " [format $fmtStrStr $id $kind $delay $cumm $detail]"
    }
    incr ipath
}
```

get_version

A scripting utility command.

Syntax

```
get_version
```

Description

Returns a list of version numbers in the following order: major release, minor release, build number. For example, "`15.10-p001 (031802)`" where `15` is the release year, `01` is the major release with the year, `001` is a minor release number, and `031802` is a unique build number.

help

A scripting utility command.

Syntax

`help [`*`command`*`]`

Parameter

`[`*`command`*`]`

Can either be a command name or a glob that may match multiple commands. Specifying no options is the same as specifying `*`. If a glob is given and matches multiple commands, then a table is printed that displays matching commands and their summaries. If a full command name is given, help for that command is printed, including an overview, as well as a list of parameters and their functions.

Description

The content of the help is parsed from the HTML docs shipped with the product in the *Stratus HLS Reference Guide*. The set of commands available for help is defined by the *Stratus HLS Reference Guide*.

A `-h` option is also available on all commands such that `command -h` is equivalent to `help command`.

list_attr

A scripting utility command.

Syntax

```
list_attr [-l] [object_id]
```

Parameters

`[-l]`

If specified, it causes all of the attributes of the given object and their values to be printed.

`[object_id]`

Identifies the object for which to retrieve the attribute names. The default is the current scope.

Description

Returns the list of attribute names of the specified object.

load_results

A scripting utility command.

Syntax

`load_results [-basic] [-snapshot *name*] *hls_config_or_snapshots*`

Parameters

`[-basic]`

Causes only a subset of the results to be loaded. Does not include `hls_module`, `snapshot`, `operation`, or `basic_block`.

`[-snapshot]`

Loads the given snapshot given a simple name. Object given must be an `hls_config`.

`hls_config_or_snapshots`

The list of configs or snapshots whose results are to be loaded.

make_link

A scripting utility command.

Syntax

```
make_link [-object object_id] [-source path] [text]
```

Parameters

`[-object object_id]`

Specifies an object path that should be used for linking.

`[-source path]`

Specifies a path, relative to the project directory, that should be used for linking.

`text`

Specifies the text that will appear as a link. This can contain any characters.

Description

The `make_link` command creates a text string that is recognized as markup in a Stratus report. The `text` string given will be displayed in the report in Stratus IDE, and in the plain-text version of the report. Stratus IDE will offer cross-linking from the text based on the specified object or file path.

open_project

A scripting utility command.

Syntax

`open_project [`*`project_tcl_path`*`]`

Parameter

`[`*`project_tcl_path`*`]`

The `open_project` command loads the project defined by the given project.tcl file. Any project that was already loaded is first unloaded.There is no support for having multiple projects loaded at the same time in the same script. If no argument is specified, a file named `project.tcl` in the current directory is loaded.

quit

A scripting utility command.

Syntax

```
quit
```

The `quit` command is only useful during interactive runs of stratus_hls. When stopped at a snapshot point, it causes the tool to exit. An error status is logged unless `cont` is executed from the `rtl` snapshot.

report

A scripting utility command.

Syntax

`report [report_name] [options] object_id_or_list_of [-silent] [-help]`

Parameters

[report_name]

Specifies the name of a registered report. If not specified, `-list` must have been specified.

`[options]`

Specifies options defined in `define_report` for the report using `-opt value` syntax. Any order is supported. If no default was given in `define_report`, the option must be specified.

object_id_or_list_of

Specifies one or more object IDs. If `define_report` specified `-repeat`, then the report is printed once for each object. Otherwise, all objects are sent to the report.

`[-silent]`

If specified, the report is not printed to the terminal; it is only printed to the report file.

`[-help]`

If a report name is given, prints help for it. Otherwise, prints the names of available reports.

Description

All reports are printed to the log for associated config object. If objects are children of configs, then a unique report file will be generate in the config reports directory. If multiple config objects are specified, a copy of the report will be generated for each config. If the same report is generated more than once for a config object, the previous report is overwritten. Reports are deleted when the associated config is cleaned.

run

A scripting utility command.

Syntax

`run [-clean] [-i] [-through` *`phase`*`] [`*`config_object_id`*`]`

Parameters

`[-clean]`

Cleans the results before executing.

`[-i]`

Runs interactively. Only for hls_config.

`[-through` *`phase`*`]`

Runs only through the specified phase. Only for * hls_config.

`[`*`config_object_id`*`]`

Is an hls_config, sim_config, logic_synth_config, power_config, equiv_config or analysis_config.

Description

Starts a run of the given config. For hls_configs, interactive mode may be specified, in which case, the tool will stop at a Tcl shell prompt at a snapshot point. If a `-through` option is specified interactive mode, the tool will stop after that phase has been executed. Without `-through`, interactive mode will stop at the first snapshot point. In non-interactive ode, `-through` will cause the tool to run only until the specified phase is complete.

There is no support for running multiple jobs at one time, and there is no support for starting background jobs.

If a job is already running for the given config, an error will be generated if `-run` is given in any way.

Makefile dependencies will be used to determine if the job must be executed. However, if `-clean` is specified, any previous results are always removed and the job is always run.

After the run completes, the shell's virtual directory is changed to the config that was executed.For interactive runs, the results of the run are loaded and available for access by Tcl commands in the shell. For non-interactive runs, results must be explicitly loaded with a `load_results` command.

set_attr

A scripting utility command.

Syntax

```
set_attr attribute_name attribute_value [object_id]
```

Parameters

`attribute_name`

Specifies the name of the attribute to be set.

`attribute_value`

Specifies the new value of the attribute.

`[object_id]`

Identifies the object whose attribute is to be set. The default is the current scope.

Description

Sets the value of an object’s attribute. If no object is specified, the current scope is used.

trim_paths

A scripting utility commmand.

Syntax

`trim_paths [-file] [-object] [-tail] [-root` *`prefix`*`]` *`string`*

Parameters

`[-file]`

Specifies that only non-object paths, which are assumed to be file system paths, will be trimmed.

`[-object]`

Specifies that only valid object paths will be trimmed.

`[-tail]`

Specifies that trimming will remove everything except the final segment of the path.

`[-root` *`prefix`*`]`

Specifies a specific prefix that should be removed. Defaults to the snapshot above the current directory.

`string`

Specifies the string that is to be filtered. It may contain file system paths, or object paths.

Description

The `trim_paths` utility shortens file system paths and object paths to make them more readable. It returns a string with paths trimmed as follows:

- If the specified `root` path is present it is removed. The default for `root` for object paths is the snapshot directory above the current directory, which results in paths being trimmed to only the portion within the current snapshot. The default for `root` for file system paths is the current directory. However, the `root` may be any path string.
- If the leading segment of the path is the current Stratus installation directory, then the entire directory portion of the path is removed such that only the file name remains. This results in paths that are part of the product being trimmed to the filename only.
- The object type is left in object paths.

By default, both file system paths, and object paths are trimmed. This can be changed by specifying `-object` or `file`, which causes the filter to be applied to either only object paths or only non-object paths, respectively.

The `-tail` option causes all affected paths to be reduced to only the leaf name. The object type is removed from all object paths by this option.

The `trim_paths` command is useful in a number of situations:

```
# To shorten paths in call stacks.
set stack [trim_paths [get_stacks $op]]

# To shorten paths in 'src_links' attribute values
puts "Op: kind=[get_attr kind $op] : source=[trim_paths src_links $op]"

# To print relative object paths that can be used in other snapshots:
foreach op [find -op -if {kind == "mul"}] {
  puts "Mul op: [trim_paths $op]"
}

# To print only object names given an array of object paths:
puts "Loop names: [trim_paths -tail [find -loop]]"
```

un_link

A scripting utility command.

Syntax

```
un_link text
```

Description

Given a string that contains an annotation created with `make_link`, the `un_link` command returns the string without the annotations. Strings without annotations are simply returned.

unload_results

A scripting utility commmand.

Syntax

`unload_results [`*`hls_config_or_snapshot`*`]`

Description

Unloads any results for the specified hls_config or snapshot. If no object is specified, unloads all results. It is not necessary to call `unload_results` before exiting a post-process script, but it is recommended if analyzing multiple configs when finished with a given config in order to save memory. It is not necessary to call `unload_results` when running live.

vcd

A scripting utility command.

Syntax

```
vcd object_path
```

Parameters

`object_path`
Identifies the object path to be used as the current scope.

Description

Sets the current virtual directory for identifying objects by a relative object ID. See also the section on virtual directories.

vls

A scripting utility command.

Syntax

`vls [-attribute] [-l] [-R] [path]`

Parameters

`[-attribute]`

Lists the attributes.

`[-l]`

If specified, it causes all of the attributes of the given object and their values to be printed.

`[-R]`

Recursive listing.

`[path]`

The virtual paths for which objects are listed. The current virtual directory is the default.

Description

This command displays information about a virtual directory but it does not return a value. The variable myvar is empty after the following Tcl command:

```
set myvar [vls]
```

The command displays a virtual directory in the following way:

- A line with the directory followed by a colon, if multiple directories have been specified
- A line for the current directory './'
- A line for each sub directory

The `vls` output of the following example does not start with a line with the directory followe by a colon, because only one directory is specified.

```
vls [find -hls_config conf1]/behaviors
./
thread1
...
```

The output of the following example of a vls command for multiple directories shows how the line of the directory precedes the lines for its contents:

```
vls [find -hls_config conf1]/*
hls_config:proj1/dut/conf1.behaviors:
```

```
thread1
...
```

```
hls_config:proj1/dut/conf1.module_terms:
in1
...
```

The `-attribute` command option results in a listing of attributes for all objects:

- A line with the path of the node
- A line with the string "Attributes:"
- A number of lines that display the names and the values of each attribute

Here is an example:

```
vls -attribute [find -hls_config conf1]/behaviors
hls_config:proj1/dut/conf1.behaviors:
thread1
  Attributes:
    cycle_slack = unset
    ...
...
```

The `vls` command shows a recursive listing if the `-R` option is specified, by listing the contents of each sub directory after the line for a sub directory. The paths of all directories are relative to the path arguments of the command. Here is an example:

```
vls -R [find -hls_config conf1]
./
behaviors
behaviors/thread1
behaviors/thread1/regions
behaviors/thread1/regions/reset
...
```

vpwd

A scripting utility command.

Syntax

```
pwd
```

Description

Returns the current virtual directory.

Known Limitation

For objects with multiple path-RegBinding and ResBinding?, this command returns the native object path.

Programs/Scripts

Following is an alphabetical list of the Stratus HLS programs and scripts.

Program/Process	Description
`bdw_annotate_rtl`	Re-annotates RTL files for the given module or hls_config pair.
`bdw_export`	Exports Verilog files.
`bdw_ifgen`	Re-generates an interface library
`bdw_ls`	Lists the contents of a library
`bdw_makegen`	Generates a `makefile.prj` file from your `project.tcl` file.
`bdw_memgen`	Generates memories.
`bdw_rm`	Removes parts from a library
`bdw_runconformal`	Integrates Stratus HLS and Conformal.
`bdw_runhal`	Integrates Stratus HLS and HAL.
`bdw_runjaspersec` `bdw_runjaspersec.tcl`	Integrates Stratus HLS and Cadence JasperGold® Sequential Equivalency Checking (SEC) App. bdw_runjaspersec.tcl is the Tcl script used by Jasper to do the actual equivalence comparison.
`bdw_runjoules`	Integrates Stratus HLS and Cadence Joules.
`bdw_runpt`	Integrates Stratus HLS and PowerTheater.
`bdw_runquartus`	Integrates Stratus HLS and Altera Quartus tool.
`bdw_runspyglass`	Integrates Stratus HLS and Spyglass.
`bdw_runvivado`	Integrates Stratus HLS and Xilinx Vivado tool.
`bdw_scan_modules`	Finds templated `define_hls_modules` and emits `define_hls_module` statements
`bdw_versions`	Lists and verifies the versions of tools in your environment.

`stratus`	Enables interaction with a Stratus project using Tcl.

bdw_annotate_rtl

A Stratus HLS program.

Location

bdw_annotate_rtl

Syntax

```
bdw_annotate_rtl [-m[odule] <module>] [-c[onfig] <hlsConfig>] [-p[project]
<projectFile>] [-proc <proc_name>] [options]
```

Parameters

`[-m[odule] <module>]`

The name of the module on which to report.

`[-c[onfig] <hlsConfig>]`

The name of the hlsConfig on which to report.

`[-p[project] <projectFile>]`

User-defined project file. A project file may be specified in a -project arg, but if none is given, project.tcl will be assumed.

`[-proc <proc_name>]`

The annotation proc to use. If this is not specified, the `proc` specified in the project file will be used.

`[options]`

`-all`

All available annotations.

`-decl`

Declarations to add to annotation.

`-op`

Operation information annotation.

-stack

Call stack annotation.

-state

Control flow information to be added to annotation.

Description

The `bdw_annotate_rtl` command results in a re-annotation of the RTL files for the given module/hls_config pair. All non-resource Verilog files will be annotated. Any existing annotations will be removed, and new ones added. The files will remain functionally identical; only the comments will be changed.

The types of Tcl objects for which annotations will be included are:

- **rtl_instance**: On the line above a component instantiation, assign statement, or always block for a resource, or ungrouped resource.
- **op**: On the line above an assignment at the input or output of the resource, or on the line above a case or if statement that describes the branch.
- **rtl_process**: On the line above an always block containing control that is not an ungrouped resource.
- **basic_block**: On the line above assignments that are state-specific, but that are not operation-specific. This includes the FSM.

Object identifiers will be shown relative to the `rtl` snapshot. For example:

```
// rtl_instance:dut/mem64x16_r_2
assign Add24_2_out = s_reg_24 + din;

// op:thread0/OP27
s_reg_24
```

rtl_instance

For ungrouped resources, a detailed comment is generated only for the first instance in the file. So, if a submodule is ungrouped to multiple always blocks, one detailed comment will be shown. For all others, the following will be shown:

```
// rtl_instance:<verilog_module>/<instance_name>
// This resource is split across multiple concurrent processes.
// See line XXX
```

id

```
// rtl_instance:<verilog_module>/<instance_name>
```

appears above a submodule instantiation, assign statement, or always block for an ungrouped instance.

op

Prints the function, output widths, and input widths, and the name of the module. Inputs or outputs that are signed have an `S` following the width.

If there are operations associated with the instance, the information that would be printed for each of those operations appears in sequence after other information printed that is not operation-specific. A comment preceds the operation data that says how may ops. For example:

```
// rtl_instance:dut/Add32_4
// Resource=Add32, Function=add: Inputs=32S,32S Outputs: 32S
// Implements 2 operations:
// dut.cpp:149:14
// dut.cpp:519:19 assign x = a + b;
```

decl

No content for `rtl_instance`.

state

No content for `rtl_instance`.

stack

No content for `rtl_instance`.

op

An `op:` annotation is generated for assignments within a case or if statement that implements a sharing or control flow mux. This may be the definition of the input of a resource that implements the op, or the definition of a value for a register.

id

```
// op:<behavior_name>/<op_name>
```

This content will be added for all options except `off`.

op

Prints the source location of the operation, and if there is an associated declared object.

```
case (global_state)
5'd03: begin
  // op:thread1/OP47
  // dut.cpp:827
  Add23_4_in1
```

Registers, IOs, and resource input defs may all be decribed this way.

decl

If the target of the operation has declaration information, the source location of the declaration is shown. This would include the name and original location of a variable that is specific to this op.

```
case (global_state)
 5'd03: begin
  // op:thread1/OP47
  // dut.cpp:234
  // variable: accum, dut.h:47
  sreg_24
```

The prefix will be one of `variable`, `signal`, or `port`.

state

The state in which the op occurs is described, including its stage in a pipeline. The location of a state is described, depending on what is known about it. If it is:

- defined by a user-entered wait, the location of that wait is shown.
- a scheduler-added wait, the nearest preceding control flow structure is shown. This may be an explicit wait, a loop header, or a branch that has a source location. If the state begins more than one cycle after this construct, "N cycles after" is prepended.
- in a pipeline, the stage and cycle of the pipeline are shown.

```
case (global_state)
 5'd03: begin
  // op:thread1/OP47
  // dut.cpp:234
  // cycle# 27: 1 cycle after wait() at dut.cpp:722
  sreg_24
```

stack

If the source location for the op is inside a function called from the original behavior, its call stack is shown:

```
5'd05: begin
  // op:thread1/OP32
  // dut.cpp:211
  // in function calc() called from dut.cpp:792
  // in function alg() called from dut.cpp:899
  sreg_24
```

rtl_process

id

```
// rtl_process:<verilog_module>/<process_name>
```

will appear above each non-resource mux described by an always block. The name will be the name that appears on the left-hand side of assignments in the block.

op

Describes what the mux is. This may include both sharing and control flow.

```
// rtl_process:dut/drive_sreg_24
// Sharing 2 operations on register sreg_24
always @(posedge clk) begin {
  if (!arst) begin
    sreg_24
```

Special purpose muxes may get more specific annotations. For example:

- State register threads: describes the thread it is for in a global_state, and the pipeline it is for in a cycle_state.

decl

If the mux is for a port or signal such that each op within the mux would have the same decl info, the annotation shows the info described under `op` annotations for **decl**.

state

No content for muxes.

stack

No content for muxes.

basic_block

A `basic_block` annotation is added for cases where there is a state-specific defintion that is not associated with any particular op. This mainly occurs in state machine register threads themselves. The basic block identified will always have the `defines_state` attribute set to `true`, meaning that a state is initiated by this basic block in the control flow. It can be used by the annotation generator to access information about both the state itself, and about related control flow.

id

```
    // basic_block:<behavior_name>/<bb_name>
```

This content will be added for all options except `off`.

op

No content for `basic_block:`.

decl

No content for `basic_block:`.

state

Describes the state using the nearest available landmark in the behavior. The format is described in the **state** entry for **op**.

```
    case (global_state)
     5'd03: begin
      // basic_block:<behavior_name>/<bb_name>
      // cycle# 27: 1 cycle after wait() at dut.cpp:722
      global_state_next
```

stack

No content for `basic_block`.

bdw_export

A Stratus HLS program.

Location

bdw_export

Syntax

```
bdw_export
  [-proj name]
  [-prefix module_prefix [-keep_filenames] ]
  [-keep_toplevel]
  [-match_top]
  [-exclude_mems]
  [-exclude_timescale]
  [-combined_file name]
  [-module name]
  ( -simconfig name [-full] [-headers] | -lsconfig name | -config name )
  <output_dir>
```

Parameters

-proj project_file

Specifies your project file name. This defaults to *project.tcl* and is not ordinarily needed.

-prefix <module_prefix>

This optional setting adds a prefix to exported modules. The `module_prefix` will be prepended to the names of all modules defined in the exported files. By default, `module_prefix` will also be prepended to the names of those files. To instead keep the names of exported files the same as the original ones, use `-keep_filenames`.

Instantiations of modules defined in exported files will use the prefixed module names, but the instance names will be left unchanged.

-keep_toplevel

This option is used with `-prefix`. Normally `-prefix` will add a prefix to all modules. This flag forces `bdw_export` to leave the name of the top level module in its original form.

-match_top

This option forces `bdw_export` to name the file containing the top level module to match the name of the top level module instead of being named <top_level_module_name>_rtl.v.

-exclude_mems

This option prevents memory models generated with `bdw_memgen` from being copied to the output directory.

-exclude_timescale

This option will remove the timescale directive from all the exported verilog files.

-combined_file <name>

This option will cause all of the exported files to be concatenated into one file <output_dir>/<name>. For the generic library flow, the cynw_lib.v and cynw_generic.v files will still be separate files in <output_dir>.

-module <name>

This option specifies the name of a module to be exported instead of exporting all modules in the sim config or logic synthesis config.

-simconfig name [-full] [-headers] | -lsconfig name | -config name

This specifies the name of the `sim_config` , `logic_synthesis_config or hls_config` whose files you wish to export. Stratus HLS will export files to the given directory. The -config option will export the given hls_config for all hls_modules in the project unless the -module option is also specified.

-full

This option only applies when exporting a simconfig. It causes export of all of the verilog files normally exported, plus all of the verification wrappers and `irun` command files for each exported module. The following `irun` command files are provided:

- ***SC_<module_name>.f***: The file would contain all the `irun` options needed to compile and include the systemC model in a systemC testbench.
- ***VL_WRAP_<module_name>.f***: The file would contain the necessary `irun` options to include the verilog verification wrappers and the systemC model in a (System)Verilog testbench.
- ***SC_WRAP_<module_name>.f***: The file would contain the `irun` options to include the new systemC wrappers and the dut *V_RTL* files in a systemC testbench.
- ***VL_<module_name>.f***: The file would contain the `irun` options to include just the dut *V_RTL* files in a (System)Verilog testbench.

`-headers`
This option only applies when exporting a simconfig and will cause `bdw_export` to copy the Stratus headers to <export_dir>/include so that the exported files can be used without access to a Stratus installation.

`output_dir`
The directory to which the output files will be written. The directory must already exist. Files will be overwritten in the directory, but the directory will not be cleaned before new files are written to it.

Description

The `bdw_export` command exports all of the Verilog files for a single `sim_config` or `logic_synthesis_config` to a single directory that is outside the project directory. These files could then be used to feed a downstream Verilog-only flow. That is, there will be no SystemC in the exported code, and no cosimulation facility for it.

For a `sim_config`, `bdw_export` exports all Verilog files that would be used in the simulation run, including design models, library models, and extra Verilog files specified in the global `verilog_files` setting in *project.tcl* or using the `sim_config`'s -verilog_files setting.

For a `logic_synthesis_config`, `bdw_export` exports all Verilog files that would be passed to your logic synthesis tool, including design models, library models, and extra Verilog files specified via the `logic_synthesis_config`'s `-verilog_files` setting.

Example 1

Consider the following `sim_config`:

```
define_sim_config A {saxo_light GATES_V C_BASIC}
```

This defines a configuration for the RTL_V version of `saxo_light` and the GATES_V version of any library parts used. To export using these settings, do the following:

```
bdw_export -simconfig A <output_dir>
```

Example 2

Here is an example that exports the RTL_V representations of both the dut and libraries

```
sim_config B {saxo_light RTL_V C_BASIC}
bdw_export -simconfig B <output_dir>
```

Example 3

Consider the following `logic_synth_config`:

```
define_logic_synthesis_config LS_X {saxo_light C_BASIC} -options {BDW_LS_NOGATES 1}
```

Because the `BDW_LS_NOGATES` option is used, this `logic_synthesis_config` will use RTL_V versions of the design and library parts. To export using these settings, do the following:

```
bdw_export -lsconfig LS_X <output_dir>
```

If you wish to export the GATES_V version of the library parts instead, you should create a new logic_synthesis_config without the `BDW_LS_NOGATES` option and export it like this:

```
define_logic_synthesis_config LS_Y {saxo_light C_BASIC}
bdw_export -lsconfig LS_Y <output_dir>
```

Example 4

The following example exports files for the `BASIC_V` configuration and prepends `PJ01_` to the names of the exported files and all modules defined in those files:

```
bdw_export -proj project.tcl -simconfig BASIC_V -prefix PJ01_ exportDir
```

Example 5

The following example exports files for the `BASIC` hls_config of module dut and prepends `PJ01_` to the names of the exported files and all modules defined in those files:

```
bdw_export -config BASIC -module dut -prefix PJ01_ exportDir
```

bdw_ifgen

A Stratus HLS program.

Syntax

```
bdw_ifgen <project_file> <bdl_file> [<interface_name>]
```

Parameters

`project_file`

The path to a project file. Usually project.tcl.

`bdl_file`

The path to a .bdl file for an interface library.

`interface_name`

An optional name of an interface to generate. If omitted, all interfaces in the library are generated.

Description

The `bdw_ifgen` program can be used to re-generate the Cynware Generated Interfaces in a specific interface library. This can be useful when a new release of Stratus is installed, and you wish to update your interface models. It is also useful for sharing interface libraries. Since an interface is entirely described by the .bdl file for its library, the .bdl file can be shared between projects, or checked into revision control. Then, interface can be generated from the .bdl file as needed using `bdw_ifgen`. A `Stratus_HLS_XL` license is required to execute `bdw_ifgen`. For more information, see Using Generated Interfaces in the *User Guide*.

Example

```
bdw_ifgen project.tcl iflib/iflib.bdl
```

This command will generate all interfaces in the 'iflib' library.

bdw_ls

A Stratus HLS command line utility.

Syntax

`bdw_ls <library_path> [<filter_string>] [options]`

Parameters

`library_path`

The path to a directory containing a library .bdl file, or the path to the .bdl file.

`filter_string`

An optional glob style string for filtering the parts selected by name. The filter is case insensitive.

`options`

bdw_ls supports a number of options to control part selection and display formatting.

Part selection options

`-bdate <time_string>`

Select parts that were built at or before the specified time

`-adate <time_string>`

Select parts that were built at or after the specified time

`-bmin N`

Select parts that were built at or before N minutes ago.

`-amin N`

Select parts that were built at or after N minutes ago.

`-bday N`

Select parts that were built at or before N days ago.

`-aday N`

Select parts that were built at or after N days ago.

Display Format options

`-d or -date`

Display the build date for selected parts.

`-m or -metrics`

Display the metrics for selected parts.

`-s or -sources`

Display the source files for selected parts.

`-p or -params`

Display the build parameters for selected parts.

`-l or listall`
Display all information for selected parts.

`-1`
Display information for one part per line.

Description

The `bdw_ls` command line utility can be used to display the contents of a Stratus HLS library.

See also:

- "bdw_rm"

Example

```
bdw_ls Cynw_std_T018_10ns "add" -m
Add1
  Metrics
        area 26.6112
        delay 0.2153589
        setupTime 0.0
        combArea 26.6112
        seqArea 0.0
        bits 0
        latency 0
Add_ECLA1
  Metrics
        area 26.6112
        delay 0.2153589
        setupTime 0.0
        combArea 26.6112
        seqArea 0.0
        bits 0
        latency 0
.
.
.
```

This command displays a list of all parts in the Cynw_std_T018_10ns library.

bdw_makegen

A Stratus HLS program.

Syntax

`bdw_makegen <`*`project_file`*`>`

Parameters

`project_file`

The name of the project file (usually *project.tcl*) that contains settings and configurations for your project.

Description

Stratus HLS provides a single project file where you will define your project, such as your modules, simulation configurations, and logic synthesis configurations. This project file (named *project.tcl*) contains information about the structure of your project and all of the different explorations you wish to perform on your design module.

The file generated will be *Makefile.prj* (unless you have specified a different Makefile name in the project file).

There are a small number of options to `bdw_makegen`:

- **-o <makefile>** Specifies the name of the Makefile to generate. This overrides both the default of *Makefile.prj*, and any Makefile name specified using the `makefile` command in *project.tcl*. (For consistency, we will always refer to this file as *Makefile.prj* in documentation.)
- **-args <argString>** The `-args` option allows arguments to be passed to the *project.tcl* file. Each token that follows the `-args` option will be accessible from Tcl via the `makegenArgs` array variable. For example, if the following command is used to run `bdw_makegen`:

```
bdw_makegen project.tcl -args -product image_processor
```

Then the tokens following `-args` can be accessed from *project.tcl* using the `makegenArgs` array:

```
foreach arg $makegenArgs {
puts $arg
}
```

This will result in the following output:

```
-product
image_processor
```

See also, Creating Your Makefile in the *User Guide.*

bdw_memgen

A Stratus HLS program.

Syntax

```
bdw_memgen [<part_name>]
```

Parameters

`part_name`

The path to and name of the memory model (*.bdm*) file you want to re-generate.

Description

The Stratus HLS Memory Model Generator (`bdw_memgen`) generates memory models from memory descriptions created in Stratus IDE. The memory models are generated automatically by Stratus IDE when they are entered or edited. However, it is sometimes necessary to re-generate the memory models using the bdw_memgen command line utility. For example, when a new release of Stratus is installed, it is advisable to re-generate memory models. This can be done with bdw_memgen as follows:

```
bdw_memgen memlib/*.bdm
```

where `memlib` is the name of the memory library referenced by the `hls_lib` command in project.tcl. There is one `.bdm` file for each memory model in the library.

For more information about memory models, see Explicitly instantiated (or external) memories in the *User Guide*.

bdw_rm

A Stratus HLS command line utility.

Syntax

```
bdw_rm [<library_path>] <part_name_list>
```

Parameters

library_path

The path to the library to operate on. If no library path is given, bdw_rm will look for a .bdl file in the current directory.

part_name-list

A space separated list of part names to be removed from the library. May include wildcards within quotes.

Description

The `bdw_rm` command line utility can be used to remove one or more parts from a Stratus library.

See also:

- "bdw_ls"

Example

```
bdw_rm 'bdw_ls "*add*" -bday 2'
```

This command will remove all parts from the library in the current directory that contain "add" in their name and were built at or before 2 days ago.

bdw_runvivado

A Stratus HLS script.

Location

`bdw_runvivado`

Description

The `bdw_runvivado` script collects information about the design from Stratus HLS, sets user-defined variable values, and runs Xilinx Vivado tool.

bdw_runconformal

A Stratus HLS script.

Location

`bdw_runconformal`

Description

The `bdw_runconformal` script collects information about the design from Stratus HLS, sets user-defined variable values, runs Conformal, and generates an equivalence checking report. This is the default if an `end_of_equiv_command` setting is omitted from the project file.

See also, Project Attributes.

bdw_runjaspersec, bdw_runjaspersec.tcl

A Stratus HLS script.

Location

`bdw_runjaspersec`

Description

The `bdw_runjaspersec` script does some initial processing and then runs jasper with the appropriate input. bdw_runjaspersec.tcl is a Tcl script used by Jasper-SEC to do the actual equivalence comparison.

Syntax

bdw_runjaspersec: -h[elp] -b[box_mul] <value> -s[pec] <ls_config | sim_config | hls_config> -i[mp] <ls_config | sim_config | hls_config> [-no_gui] [-m[odule] top_module] -constraint <file_path>

Parameters

-h
Help message.

`-b[box_mul]`
Specify threshold for blackboxing multipliers. The default is 128-bit.

-s`[pec]`
Specify sim_config or ls_config or hls_config of spec_design.

`-i[mp]`
Specify sim_config or ls_config or hls_config of imp_design.

`-n | -no_gui`
Specify running Jasper-SEC without GUI. The default is running with GUI.

-r `| -reset_conditions`
Specify reset conditions.

Examples

Here is an example of the use of bdw_runjaspersec to compare the RTL of to logic synth configs:

```
bdw_runjaspersec -module dut -spec BASIC -imp LOW_POWER -reset_conoditions "~rst"
```

The -spec and -imp values do not contain "ls_" or "sim_" so they are assumed to be module configs of the dut module. If the define_equiv_config has a `-spec/-imp` parameter pair defined, then makegen.tcl will add the necessary parameters from the equivConfig to the command line to run `bdw_runjaspersec`.

The following define_equiv_config command:

```
define_equiv_config E1 -spec BASIC -imp PIPE_1 -bbox_mul 64 -no_gui -module dut -
reset_conditions "~rst" -command bdw_runjaspersec
```

causes the following command line to be written to makefile.prj for running Jasper-SEC:

```
bdw_runjaspersec -spec ls_BASIC -imp ls_PIPE_1 -bbox_mul 64 -reset_conditions "~rst"
-no_gui -module dut
```

bdw_runjoules

A Stratus HLS script.

Location

`bdw_runjoules`

Description

The `bdw_runjoules` script collects information about the design from Stratus HLS, sets user-defined variable values, runs Joules, and generates a power estimation report. This is the default if a `power_command` project attribute is omitted from the project file.

bdw_runpt

A Stratus HLS script.

Location

`bdw_runpt`

Description

The `bdw_runpt` script collects information about the design from Stratus HLS, sets user-defined variable values, runs PowerTheater, and generates a power estimation report.

bdw_runspyglass

A Stratus HLS script.

Location

`bdw_runspyglass`

Description

The `bdw_runspyglass` script collects information about the design from Stratus HLS, sets user-defined variable values, runs Spyglass, and generates a code analysis report.

See also:

- The `analysis_command` attribute in Project Attributes
- "Atrenta Spyglass Integration" application note

bdw_scan_modules

A Stratus HLS command line utility.

Syntax

```
bdw_scan_modules [-D<var>[=<value>]] [-I<include_path>]
                       [-quiet] [-short_names]
                       [-o <out_path>]
                       <c++_file> [{<c++_file>}]
```

Parameters

`var/value`
A pre-processor definition either with or without a value.

`include_path`
A directory in which to look for #included files.

`out_path`
An optional path to which `define_hls_module` statements are written.

`quiet`
Prevents informational messages from being printed.

`short_names`
Uses a uniquifying integer suffix rather than the names of each template parameter when creating `hls_module` names.

`c++_file`
A list of one or more C++ source files that should be scanned.

Location

bdw_scan_modules

Description

The `bdw_scan_modules` command line utility finds unique instantiations of `SC_MODULE`s and emits a `define_hls_module` statement for each. This utility is intended for designs that uses templated `SC_MODULE`s because it finds each unique sets of template parameters used with each `SC_MODULE` and generates the appropriate `[template]` option with the `define_hls_module` statement for each one.

The output of `bdw_scan_modules` can be manually copied and pasted into a project.tcl file, and modified by hand from there by adding `io_configs` or `hls_configs`. Or, the output can be placed in a file and included in project.tcl using the Tcl `source` command. If the generated `define_hls_module` statements are sourced, the `define_io_config` and `define_hls_config` commands can be used to configure the modules in the project.tcl file.

By default, `bdw_scan_modules` will generate a unique module name for each templated `define_hls_module` based on its template parameters. For example, if the template instantiation is:

```
filter< sc_uint<8> > m_filter;
```

then the `hls_module` statement generated will be:

```
define_hls_module filter_sc_uint_8 dut.cpp \
             -template "filter sc_uint<8>"
```

If you are copying the output of `bdw_scan_modules` into your project.tcl file, you can change the names to something more to your liking. The only requirement is that the `define_hls_module` statements in your project have unique names.

You can also use the `-short_names` option to generate names that are uniquified only with an integer suffix rather than the names of the template parameters. The the example above, `-short_names` would produce:

```
define_hls_module filter_1 dut.cpp -template "filter< sc_uint<8> >"
```

Multiple source files can be placed on the `bdw_scan_modules` command line. If a module's member functions are separated across multiple files, then all of those files must either be specified on the command line, or `#included` from a file specified on the command line. This allows `bdw_scan_modules` to include all the required source files for a module in its associated `define_hls_module` statement.

See also:

- define_io_config
- define_hls_config
- Using C++ templates with hls_modules in the *User Guide*
- *Using templated hls_modules to design a switching fabric* in the Stratus Knowledge Base.

Example #1

```
// filter.h
template <typename T>
SC_MODULE(filter) {
  sc_in_clk clk;
  sc_in<bool> rst;
  sc_in<T> din;
  sc_out<T> dout;
  SC_CTOR(filter);
  ...
};
```

```
// dut.cpp
```

```
#include "systemc.h"
#include "filter.h"
SC_MODULE(dut) {
  sc_in_clk clk;
  sc_in<bool> rst;
  sc_in< sc_uint<8> > din8;
  sc_out< sc_uint<8> > dout8;
  sc_in< sc_uint<8> > din16;
  sc_out< sc_uint<8> > dout16;


  filter< sc_uint<8> > m_filter8;
  filter< sc_uint<16> > m_filter16


  SC_CTOR(dut);
  ...
};
```

Execute command:

```
> bdw_scan_modules dut.cpp
```

Produces:

```
define_hls_module dut dut.cpp
define_hls_module filter_sc_uint_8 dut.cpp \
             -template "filter< sc_uint<8> >"
define_hls_module filter_sc_uint_16 dut.cpp \
             -template "filter< sc_uint<16> >"
```

Example #2

Using the same source code as Example #1, if the command is instead:

```
> bdw_scan_modules -short_names dut.cpp
```

then the result is:

```
define_hls_module dut dut.cpp
define_hls_module filter_1 dut.cpp -template "filter< sc_uint<8> >"
define_hls_module filter_2 dut.cpp -template "filter< sc_uint<16> >"
```

bdw_versions

A Stratus HLS script.

Location

`bdw_versions`

Syntax

`bdw_versions [-simulator (mti|ncverilog|vcs)]`

Parameters

`-simulator (mti|ncverilog|vcs)`
This optional setting checks your current operating system, compiler, and simulator version and emits a message when the combination is unsupported.

Description

When used without the `-simulator` option, the `bdw_versions` script will return a detailed listing of the versions of tools in the environment, and the versions of tools used to configure the Stratus HLS kit in the current environment. The `bdw_versions` script will also detect any incompatibilities and issue error messages as required. `bdw_versions` is useful for debugging problems when the setup of the environment is suspected.

When used with the `-simulator` option, the `bdw_versions` script verifies that your combination of operating system, compiler, and simulator vendor/version is supported by Stratus HLS. This is also automatically checked at runtime. If an unsupported combination is detected, a warning message is emitted. For a list of supported operating system, compiler, and simulator combinations, see *Installation Guide*.

Example #1

```
bdw_versions
```

This command checks for compatibility of the SystemC, gcc, and Stratus HLS installations accessed by your current shell and prints a report.

Example #2

```
bdw_versions -simulator vcs
```

This command checks your current VCS version, operating system version, and compiler version against supported Stratus HLS combinations.

bdw_runquartus

A Stratus HLS script.

Location

`bdw_runquartus`

Description

The `bdw_runquartus` script collects information about the design from Stratus HLS, sets user-defined variable values, and runs Altera Quartus tool.

bdw_runhal

A Stratus HLS script.

Location

`bdw_runhal`

Description

The `bdw_runhal` script collects information about the design from Stratus HLS, sets user-defined variable values, runs HAL, and generates a code analysis report. This is the default script used if the `analysis_command` project attribute setting is omitted from the project file.

See also:

- The `analysis_command` attribute in Project Attributes
- set_analysis_options

stratus

A Stratus HLS shell utility.

Syntax

The stratus command has the following syntax:

```
stratus [-batch] [-execute "command"]+ [-files "tcl_files"] [script_args]
```

Parameters

`[-batch]`

Exits after executing `-files`.

`[-execute "command"]`

`-execute s`pecifies a Tcl command to be executed before loading `-files`.

`[-files "tcl_files"]`

Loads the specified files. Multiple files should be specified in a quoted string.

`[-help]`

Prints this help. See help.

`[script_args]`

Any additional arguments are available to scripts via Tcl $argv.

If the `-batch` option is omitted a Stratus interactive shell is opened after any files specified in `-files` have been sourced. If there is a file in the current directory named `project.tcl`, it is silently loaded, which initializes the shell with the project tree.

A script can be made into an executable file by adding *execute* permissions to the file in Linux, and by adding the following lined to the top of the file:

#!/bin/sh

#\

```
 exec stratus -batch -files "$0" "$@"

# Tcl script begins here...
```

ESC functions

Following is an alphabetical list of the commands available.

Keyword	Description
`esc_argc`	Returns the number of command line arguments.
`esc_argv`	Returns the entire set of command line arguments or the command line argument at a given index.
`esc_initialize`	Initializes cosimulation functionality.
`esc_log_fail`	Indicates that a simulation has failed a test.
`esc_log_latency`	Logs the latency of a computation.
`esc_log_message`	Logs a message with a condition code.
`esc_log_pass`	Indicates that a simulation has passed a test.
`esc_normalize_to_ps`	Converts an `sc_time` to double with picosecond time units.
`esc_printf`	Provides `printf` functionality.
`esc_rand`	Generates pseudo-random numbers.
`esc_srand`	Seeds the pseudo-random number generator.
`esc_trace`	Causes a signal to be traced in BEH and RTL sims
`esc_version`	Returns the ESC version.

esc_argc

An ESC function.

Syntax
int **esc_argc** ( )

Parameters
None.

Description

The `esc_argc()` function returns the number of command line parameters passed to either a standalone SystemC program or a cosimulation. These parameters are ordinarily specified using the `-argv` option to `define_sim_config`. For more information, see esc_argv.

In order for `esc_argc()` to function properly, `esc_initialize()` must have been called from `sc_main`. For more information, see "Initializing and terminating the simulation".

Example

```
int sc_main( int argc, char* argv[] )
 {
    esc_initialize( argc, argv );
    esc_elaborate();
    sc_start(0);
 }
```

esc_argv

An ESC function.

Syntax

const char** **esc_argv** ( )
const char* **esc_argv** (int *index*)

Parameters

index
The index of the argument to be accessed. For a standalone program, index 0 gives the name of the executable file and index 1 is the first argument value. For a cosimulation, index 0 is an empty string and index 1 is the first value in the `argv` option string.

Description

The `esc_argv()` function allows command line arguments to be accessed through a common function for either standalone or cosimulation execution. The `esc_argv()` function returns command line arguments in one of two forms:

- With no *index* parameter, `esc_argv()` returns an array of strings for the entire set of command line arguments. This makes it suitable for direct replacement of the argv parameter in an existing C program.
- With the *index* parameter, the `esc_argv()` function returns the individual command line argument at the given index.

The values returned by `esc_argv()` come from the `argv` value passed to `sc_main` and from there to `sc_initialize()`. These values are ordinarily specified using the `-argv` option to the `define_sim_config` command.

See also:

- Passing arguments into a simulation in the *User Guide*.

Example

If the `define_sim_configs` are set up as follows:

```
define_sim_config SCB1 {M1 BEH} -argv "infile1.dat outfile1.dat"
define_sim_config SC1 {M1 RTL_V C1} -argv "infile1.dat outfile1.dat"
define_sim_config SC2 {M1 RTL_V C2} -argv "infile2.dat outfile2.dat"
define_sim_config SC2_V {M1 RTL_V C2} -argv "infile2.dat outfile2.dat"
```

and *argc/argv* are added to `sc_main`:

```
sc_main(int argc, char* argv[])
{
  esc_initialize(argc, argv);
  esc_elaborate();
  sc_start();
}
```

the arguments may be accessed as follows:

```
void esc_elaborate()
{
   infile = fopen(esc_argv(1),"r");
   outfile = fopen(esc_argv(2),"w");
}
```

In this example, the first parameter to `escArgv()` gives the name of the file opened as `infile`, and the second parameter gives the name of the file opened as `outfile`. So, in `sim_config SC2`, `infile` will be *infile2.dat*, and `outfile` will be *outfile2.dat*.

esc_initialize

An ESC function.

Syntax

int **esc_initialize** (int *argc,* char * *argv[ ]*)

Parameters

argc
The number of command line arguments.

argv
The array of pointers to *char* strings of command line arguments.

Description

The `esc_initialize()` function should be called from `sc_main` to provide arguments to the Stratus HLS runtime simulation platform. In addition, the use of `esc_initialize()` permits the command line arguments `argc` and `argv` to be accessed from within your testbench using the `esc_argc()` and `esc_argv()` functions, respectively.

`esc_initialize()` also enables the use of the simulation results logging functions. For more information, see Organizing the sc_main function with ESC functions in the *User Guide.*

Example

```
sc_main(int argc, char* argv[])
{
  esc_initialize(argc, argv);
  esc_elaborate();
  sc_start(1000);
}
```

esc_log_fail

An ESC function.

Syntax

bool **esc_log_fail**();

Parameters

None.

Description

The `esc_log_fail()` function marks a simulation log file as having failed a test. This status is available in the Stratus Integrated Design Environment (Stratus IDE) and in custom reports using the BDW Tcl API.

Example

```
esc_log_fail();
```

esc_log_latency

An ESC function.

Syntax

bool **esc_log_latency** (const char* *modName*,
unsigned long *latency*,
const char* *label=0*
);

bool **esc_log_latency** (unsigned long *latency*,
const char* *label=0*
);

bool **esc_log_latency** (const char * *modName*,
unsigned long *minValue*,
unsigned long *maxValue*,
double *mean_value*,
const char * *label=0*
);

bool **esc_log_latency** (unsigned long *minValue*,
unsigned long *maxValue*,
double *mean_value*,
const char * *label=0*
);

Parameters

modName
The name of the module that issued the message.

latency
The latency in clock cycles.

min_latency
The minimum latency in clock cycles.

max_latency
The maximum latency in clock cycles.

mean_latency
The mean latency in clock cycles.

label=0
An optional name for the latency. This allows you to log a variety of different latencies, if desired. If no label is specified, "0" is assumed.

Description

The `esc_log_latency()` function allows you to log the latency of a computation. This status is available in the Stratus IDE and in custom reports via the BDW Tcl API. For more information, see Logging latency of the computation in the *User Guide*.

Latencies can be reported either globally for the design, or for individual modules. Notice that 2 of the overloads of the function have a modName parameter, and 2 do not. If no modName is specified, the latency is assumed to pertain to the entire simulation rather than a specific module.

If `esc_log_latency()` is called more than once for the same module or for the entire design, all reported latencies are logged and displayed in Stratus IDE.

Examples

```
esc_log_latency("dut1", latency);
```

This will report the latency specific to the *dut1* module.

```
esc_log_latency(latency);
```

This will report the latency for the entire design.

```
esc_log_latency("design", 3, 6, 4.5, "my_latency");
```

This will report the min, max, and mean latency number of the design to a latency named `my_latency`.

esc_log_message

An ESC function.

Syntax

esc_log_message (const char* *modName*,

int *conditionCode*,

const char* *formatStr*, ...);

Parameters

modName

The name of the module that issued the message.

conditionCode

One of the following: `esc_note`, `esc_warning`, `esc_error`, or `esc_fatal`.

formatStr

The format string for the message.

Description

The `esc_log_message()` function can be called to log a message with one of the condition codes to the current simulation log file. The possible condition codes are `esc_note`, `esc_warning`, `esc_error`, and `esc_fatal`. This status is available in the Stratus IDE and in custom reports using the BDW Tcl API.

Example

```
esc_log_message("module",esc_warning, "Severity Alert",);
```

This logs the message "Severity Alert" under Messages in the simulation execution report.

esc_log_pass

An ESC function.

Syntax
bool **esc_log_pass**();

Parameters
None.

Description
The `esc_log_pass()` function marks a simulation log file as having passed a test. This status is available in the Stratus IDE and in custom reports via the BDW Tcl API.

Example
```
esc_log_pass();
```

esc_normalize_to_ps

An ESC function.

Syntax

double **esc_normalize_to_ps** (sc_time & *t*)

Parameters

t

An `sc_time` with any time unit that will be converted.

Description

Converts an `sc_time` to double with picosecond time units. `esc_normalize_to_ps()` is useful when you need numbers to represent simulation times that are not affected by the simulator's time resolution as `sc_time_stamp` is.

esc_printf

An ESC function.

Syntax

esc_printf (const char* *formatStr*, ...);

Parameters

formatStr
The format string for the message.

Description

The `esc_printf()` function is identical to the system `printf()` function, except that the output is routed through Stratus HLS. For a cosimulation, this results in the message being routed out of the logic simulator's transcript. In a standalone simulation, the system `printf()` function is called.

For additional parameters and usage information, see your system `printf()` documentation.

esc_rand

An ESC function.

Syntax

int **esc_rand**(void)

Parameters

None.

Description

The `esc_rand()` function is identical to the Unix `rand()` function, except the sequence of numbers returned by `esc_rand()` is platform-independent. By contrast, the sequence of numbers returned by `rand()` is platform-dependent, so testbenches that include `rand()` calls are not portable between Solaris and Linux if comparison is done against a golden file.

The `esc_rand()` function returns the next element in the sequence of pseudo-random numbers. The `esc_rand()` function may be called before it to seed the pseudo-random number generator. If `esc_srand()` is not called, a default seed will be chosen. Both of these functions require that `esc_initialize()` be called.

See also:

- esc_srand
- esc_initialize

Example

Following is an example of a thread that generates random data for a testbench:

```
void tb::injection_thread()
{
  reset();
  esc_srand( CURRENT_SEED ); // Seed the random number generator.

  for(int p=0 ; p < LEN ; p++) {
    int i = esc_rand(); // Get a random integer from the sequence.
    dut_in.write( i ); // Send it to the device under test.
  }
}
```

esc_srand

An ESC function.

Syntax
void **esc_srand**(unsigned int *seed*)

Parameters
seed
The seed value for the pseudo-random number generator.

Description
The `esc_srand()` function is used to specify the seed value for the `esc_rand()` function. Otherwise, `esc_srand()` is identical to the Unix `srand()` function. These functions allow you to generate platform-independent numbers, allowing testbenches that include them to be portable between Solaris and Linux if comparison is done against a golden file.

The `esc_srand()` function may be omitted before `esc_rand()`. If `esc_srand()` is not called, a default seed will be chosen. Both of these functions require that `esc_initialize()` be called.

See also:

- esc_rand
- esc_initialize

esc_trace

An ESC function.

Syntax

void **esc_trace** (<signal> [, <always>])

Parameters

signal
The name of an `sc_signal` variable, or an array of `sc_signal` variables. Cannot be a non-signal variable.
always
Boolean value. If true, the signal is traced even if it is not used, which may affect RTL QOR.

Description

The `esc_trace()` function is similar to the SystemC `sc_trace()` function, but it has the following extensions:

- The signals will be traced in the signal log file that is managed automatically as part of the `enable_waveform_logging` feature.
- The signals will only be traced if enable_waveform_logging is on.
- The names of the signals will be preserved in the RTL generated for hls_modules that call `esc_trace()`, either directly, or indirectly through an instantiated modular interface.
- An optional second parameter controls whether the signal will be preserved and traced in RTL even if it isn't used by the instantiating module.

The `esc_trace()` function is particularly useful in `HLS_INLINE_MODULE`s that are part of modular interfaces. When a `HLS_INLINE_MODULE` is instantiated, the signals within it are not automatically traced in BEH simulations by the `enable_waveform_logging` feature.

By adding calls to `esc_trace()` to the `HLS_INLINE_MODULE`, the signals will be traced when an instantiating module is simulated at the BEH level. The author of the modular interface can control which signals should be visible to users by calling `esc_trace()` for only a subset of its signals.

The best place to call `esc_trace()` from in an inline module is from the `start_of_simulation()` member function. This is a virtual function defined for sc_modules that can be overloaded by user-defined modules. It is called after all of the design has been elaborated (e.g. all instances created, and port bindings completed) and as simulation is about to begin. As such, it is a good place to put tracing calls. The `start_of_simulation()` function is also included in the behavioral synthesis of instantiating modules, so the `esc_trace()` function can have an effect on synthesis of the module to ensure that the signal names are preserved.

See also:

- HLS_PRESERVE_SIGNAL
- HLS_INLINE_MODULE

Example

Following is an example that shows how sc_trace() can be used in the channel class for a modular interface:

```
// inline module definition.
template <typename T>
SC_MODULE(mychan)
{
  HLS_INLINE_MODULE;
  sc_signal<bool> rdy;
  sc_signal<bool> vld;
  sc_signal<T> data
};
void start_of_simulation()
{
  esc_trace( rdy );
  esc_trace( vld );
  esc_trace( data );
}
T get() {
  ...
}
void put( T& v ) {
}
// hls_module that instantiates the mychan<> module.
SC_MODULE(dut) {
  sc_in_clk clk;
  sc_in<bool> rst;
  ...
  mychan< sc_uint<8> > chan;
  ...
};
```

The esc_trace() calls have 2 effects:

- The rdy, vld, and data signals will be traced in BEH simulation of module dut whenever enable_waveform_logging is enabled for the simulation.
- The rdy, vld, and data signals' names will be preserved and traced in the RTL for module dut. The names will be chan_rdy, chan_vld, and chan_data.

Because the optional second parameter to esc_trace() allowed to take it default value of false, if for any reason the rdy, vld, or data signals are unused by dut, they will not be traced in RTL simulations. If the following calls had been made instead:

```
void start_of_simulation()
{
  esc_trace( rdy, true );
  esc_trace( vld, true );
  esc_trace( data, true );
}
```

Then the signals would be preserved in the RTL even if they are unused. This may have a negative impact on QOR.

esc_version

An ESC function.

Syntax

const char* **esc_version** ( )

Parameters

None.

Description

The `esc_version()` function is used to determine the current version of Stratus HLS. It returns a const char* of the form:

`Stratus version - Y.MM - pNNNN (BBBB)` where Y is the release year, MM is the major release number, NNNN is the minor release number, and BBBB is a unique build identifier.

Global Variables

Global variables will be set to appropriate values during the behavioral synthesis process. They can be used in the calculation of value used in directives.
For example:

```
HLS_SET_DEFAULT_INPUT_DELAY( HLS_CLOCK_PERIOD / 4.0 );
```

will set the default input delay to 1/4 of the clock period used during synthesis.These variables are not set to any meaningful value during behavioral simulations.

Global Variable	Description
HLS_CYCLE_SLACK_VALUE	The change in the clock period available for synthesis, either using the `cycle_slack` attribute, or the `timing_aggression` attribute.
HLS_CLOCK_PERIOD	The clock period specified in the `clock_period` attribute.
HLS_FU_CLOCK_PERIOD	The clock period value adjusted by the cycle slack value.
HLS_INITIATION_INTERVAL	The initiation interval of the pipelined loop in which the directive appears, or 0 if it is not in a pipelined loop.
HLS_REG_DELAY	The clk-to-q delay for a typical flip-flop in the tech library.
HLS_REG_SETUP_TIME	The d-to-clk setup time or a typical flip-flop in the tech library.

4

Object Attributes

Stratus HLS organizes all of the settings for a project in a unified object model. The object model includes objects defined in project.tcl such as hls_modules and hls_configs as well as objects defined in SystemC source such as loops and regions. Each object has a pre-defined set of attributes that control the synthesis process.
This chapter list some of the pre-defined object attributes that control the synthesis process.

analysis_config Object Attributes

An analysis_config objectexists for each define_analysis command in the project file.

config_specs	**Type**: List of strings. Contains a list of `hls_config` objects that are covered by this config.
job_status	**Type**: Enumeration. One of the following • `unrun`: The config has not been executed. • `running`: The config is being processed. • `complete_ok`: The config completed successfully. This does not mean the analysis found no errors. • `complete_error`: The config completed with an error.
last_run	**Type**: String. Specifies the date and time of last execution.
name	**Type**: String. Specifies the name of the config.

passed	**Type**: Boolean. True if the analysis found no errors, false if errors were found, and `unset` if no status is available.
results_available	**Type**: Boolean. True if the config has been run and results can be loaded.
runtime	**Type**: Integer. Specifies the elapsed time for the last run of this config in seconds.
type	Type: String. Value is `analysis_config`.

array Object Attributes

An array object represents an array in the SystemC source before it is flattened or mapped to a resource. All arrays contain only scalar objects, so an array of structs or classes will result in an array for every structure field.

data_is_signed	**Type**: Boolean. Read only. True if data words are signed.
data_width	**Type**: Integer. Read only. The width of each word in this array.
data_words	**Type**: Integer, or array of integer if multi-dim. Read only. The number of words in this array.
extracted_to	**Type**: String. Read only. Specifies the name of the parent module to which this memory is extracted as controlled by the `extract_memory` command.
flatten_type	**Type**: One of DEFAULT_FLATTEN, DPOPT_FLATTEN, NO_FLATTEN. The default value is FLATTEN, if global `flatten_arrays`; else, NO_FLATTEN. Specifies whether and how to flatten this array. This attribute is set by the `flatten_arrays`, `map_to_memory`, `map_to_reg_bank` commands.
index_simple	**Type**: Boolean. The default value is 0. Read only. This attribute is set by the `map_array_indexes` command.
invert_dims	**Type**: Boolean. The default value is 0. Read only. This attribute is set by the `invert_dimensions` command.
is_extracted	**Type**: Boolean. Read-only. True if the array has been extracted to a parent module as specified by the `extract_memory` project command.

map_type	**Type**: One of MAP_MEMORY, MAP_REG_BANK. Specifies whether to map this array to a memory or `reg_bank`. This attribute is set by the `map_to_memory` and `map_to_reg_bank` commands.
memory_name	**Type**: String (name of memory type). The default value is "". Specifies the name of the memory to which the array must be mapped. This attribute is set by the `map_arrays` command.
name	**Type**: String. Read-only. The name of the array variable.
operations	**Type**: List of object paths. Specifies all operations in any behavior in the same snapshot that access the array.
parent	**Type**: Object path. Specifies the path of the parent object.
rtl_instances	**Type**: An object path or a list of paths. Specifies one or more object paths to rtl_instance objects that are mapped to this array. • Only available at `rtl` snapshot. • This is a convenience link and it could be derived by following `rtl_instances` on each operation.
separate_dims	**Type**: Integer. The default value is 0 (means no separation). This attribute is set by the `separate_arrays` command.
scope	**Type**: Object path. Specifies the path of the parent scope.
src_links	**Type:** String or list of strings Specifies the location in the source code where the array is declared.

use_clock	**Type**: String. Default is "" Specifies the clock for the accessing thread, or the alternative clock specified to be used with the associated memory in the `map_to_memory` command.
will_be_flattened	**Type**: Boolean. True at the `elab` snapshot if the array will be flattened during the `optim` phase. False at all other times.

Stratus organizes all of the settings for a project in a unified object model. The object model includes objects defined in `project.tcl` such as `hls_modules` and `hls_configs` as well as objects defined in SystemC source such as loops and regions. Each object has a pre-defined set of attributes that control the synthesis process.

basic_block Object Attributes

A `basic_block` represents a node in a CFG. A `basic_block` is uniquely associated with an FSM state and may or may not define a state. A `basic_block` is always within a loop and may be within a region. All loops have a `basic_block` as a header and some `basic_blocks` are loop headers. All operations are uniquely associated with a `basic_block`.

defines_state	**Type**: Boolean. True if the basic block begins a clocked state.
fsm_encoding	**Type**: List of state encodings. Each state encoding has the form `<state_reg>,<value>` where <state_reg> is an `rtl_signal:` object path for a state register. The state register `value` associated with the FSM state follows the `rtl_signal`, separated by a comma. For example, `rtl_signal:design/dut/BASIC/design/global_state1,14`. More than one encoding is returned in cases where both a `global_state` and one or more `cycle_state` registers are involved. Only set for basic_blocks where `defines_state` is true.
is_loop_header	**Type**: Boolean. True if the basic block is the loop header of the loop given by the `loop` attribute.
latency_in_loop	**Type**: Integer of pair of integers. Specifies the number of cycles from the start of the enclosing loop that the basic block will start. If the latency is fixed, a single number is returned. If the latency is variable, a pair of numbers is returned giving the min and max. The value for a `basic_block` that is a loop header is 1 if the block also defines a state, and 0 if it does not. For pipelined loops, `latency_in_loop` is always a single value, and when combined with the loop's `init_interval` attribute, it can be used to calculate the pipeline stage and cycle of the `basic_block`. Latencies are *acyclic* latencies, meaning that they are evaluated as though all loops had exactly one iteration.
loop	**Type**: Object path. Specifies the loop that this `basic_block` is immediately part of. For basic blocks where `is_loop_header` is true, this is the loop for which it is the header.

operations	**Type**: List of object identifiers. Specifies all operation objects that are in the basic block. The order of the list is not significant.
region	**Type**: Object path. Specifies the region that the `basic_block` is immediately within, or `undef`, if none.
src_links	**Type**: String. Specifies the source location of the `wait()`, if `defines_state` is true, and the state was defined by a wait in a protocol.
state_name	**Type**: String. Specifies the name associated with the state in the `basic_block` where `define_state` is true. This may have been specified directly by a label or generated by Stratus.
states	**Type**: List of object paths of type basic_block. Specifies the `basic_block` objects that define states that this basic block is part of. There can be more than one in cases where state boundaries are branched around. For `basic_blocks` where `defines_state` is true, the list contains the `basic_block` itself.

behavior Object Attributes

A behavior is the internal representation of an SC_CTHREAD, an SC_METHOD, or a function that has not been inlined.
Each behavior object has the attributes listed below, and also all of the attributes for region objects.

all region attributes	See region Object Attributes.
bits	**Type**: Integer. Specifies the number of register bits in the part.
clock	**Type:** Object path The behavior_io object that is the clock for this behavior. For functions or async SC_METHODs, returns an empty string.
comb_area	**Type**: Real number of square microns or luts. Specifies the combinational area.
cycle_slack	**Type**: Real. The default value is `unset` This attribute is set by the `set_cycle_slack` command.
default_input_delay	**Type**: Real. The default value is `unset`. Specifies the default input delay of a behavior. This attribute is set by the `set_default_input_delay` command.
default_output_delay	**Type**: Real. The default value is clock period. Specifies the default output delay of a behavior. This attribute is set by the `set_default_output_delay` command.
default_output_options	**Type**: One of SYNC_HOLD, SYNC_NO_HOLD, ASYNC_HOLD, ASYNC_NO_HOLD. The default value is `unset`. Specifies the default output options of a behavior.This attribute is set by the `set_default_output_options` command.
default_preserve_io	**Type**: Boolean Specifies whether the sc_signals written by this behavior must be preserved even if they are not used. If not specified, the default value will be the value of the `default_preserve_io` synthesis control attribute.

is_thread	**Type:** Boolean. Read-only. True if the behavior is an SC_CTHREAD
is_method	**Type:** Boolean. Read-only. True if the behavior is an SC_METHOD
is_function	**Type:** Boolean. Read-only. True if the behavior is a non-inlined function.
is_default_protocol	**Type:** Boolean. True if the behavior is to be scheduled in a cycle accurate way by default.
name	**Type**: String. Read-only. The name of this behavior.
parent	**Type**: Object path. Specifies the path of the parent object.
resets	**Type:** List of strings The resets for this behavior, encoded as strings of the form <port>,<polarity>,<sync|async>. One triplet for each reset for the process, in order of priority. For example `{"behavior_io:filter/filter1/BASIC/thread1/arst,0,async" "behavior_io:filter/filter1/BASIC/thread1/srst,1,sync"}` for a process with one active low async reset, and one active high sync reset.
root_loop	**Type**: String containing an object path. The object path of the the "root" loop for the behavior. This is a loop that does not appear in the source code but that contains all top-level loops in the behavior.
scope	**Type**: Object path. Specifies the path of the parent scope.
seq_area	**Type**: Real number of square microns or luts. Specifies the sequential area.

src_links	**Type:** string or list of strings. Read only. Describes the location of the behavior in the source code.
total_area	**Type**: Real number of square microns or luts. Specifies the total area of the part.

behavior_io Object Attributes

A `behavior_io` represents I/O from the associated behavior, either with module ports, inter-thread signals, or function parameters. If the type of the I/O contain a struct, class or array, there will be a separate `behavior_io` object for each leaf-level object.

direction	**Type**: String, either `input` or `output`. Read only. Specifies whether the I/O is read or written by the behavior.
input_delay	**Type**: Real. Default `unset`. The input delay set by the `set_input_delay` for this input. If no `input_delay` has been set specifically for this input, UNSET will be returned and not the default input delay.
is_async_reset	**Type**: Boolean. Read only. True if this input is used as an async reset by a thread or method in the behavior.
is_bounded	**Type**: Boolean. Read only. True if reads and writes of this I/O are limited by surrounding boundaries. This attribute is set by the `set_is_bounded` command.
is_reset	**Type**: Boolean. Read only. True if this input is used as a reset by a method or thread in the behavior.
is_signed	**Type**: Boolean. Read only. True if the data type of this I/O is signed.
module_io	**Type**: Object path. Specifies the path to the `module_io object` associated with this `behavior_io`.
name	**Type**: String. Read only. The name of this I/O.
operations	**Type**: List of object paths. Specifies all operations in any behavior in the same snapshot that access the port or signal.

output_delay	**Type**: Real. Default `unset`. The output delay set by the `set_output_delay` for this output. If no output delay has been set specifically for this output, UNSET will be returned and not the default output delay.
output_options	**Type**: one of SYNC_HOLD, SYNC_NO_HOLD, ASYNC_HOLD, ASYNC_NO_HOLD. This attribute is set by the `set_output_options` command. If no output options have been set specifically for this input, `unset` will be returned and not the default output options.
parent	**Type**: Object path. Specifies the path of the parent object.
preserve_io	**Type**: Boolean. True if this I/O will be preserved in the RTL even if it is not accessed. This attribute is set by the HLS_PRESERVE_SIGNAL directive. If it is not specified, its default value will be the value of the `default_preserve_io` attribute on the associated behavior.
rtl_signals	**Type**: An object path, or a list of paths. Specifies one or more object paths to `rtl_signal` objects. • Only available at `rtl` snapshot. • The signals may be registers, wires, or ios. • This is a convenience link and it could be derived by following `rtl_instances` on each operation.
scope	**Type**: Object path. Specifies the path of the parent scope.
src_links	**Type:** String or list of strings. Describes the declaration location in the SystemC sources.

stall_value	**Type**: Integer. The default value is `unset`. The value that the output will be given during stalls. This attribute is set by the `set_stall_value` command.
width	**Type:** Integer. Read only. The width of the data type of the I/O.

equiv_config Object Attributes

An equiv_config objectexists for each define_equiiv_config command in the project file.

config_specs	**Type**: List of strings. Contains a list of the `hls_config` objects analyzed by this config.
job_status	Type: Enumeration. One of the following: • `unrun`: The config has not been executed. • `running`: The config is being processed. • `complete_ok`: The config completed successfully. This does not mean equivalence was proven. • `complete_error`: The config completed with an error.
last_run	**Type**: String. Specifies the date and time of last execution.
name	**Type**: String. Specifies the name of the config.
passed	**Type**: Boolean. True if equivalence was proven; false if a negative case was logged. `unset` if no status was logged.
results_available	**Type**: Boolean. True if the config has been run, and results can be loaded.
runtime	**Type**: Integer. Specifies the elapsed time for the last run of this config in seconds.
type	**Type**: String. equiv_config

hls_config Object Attributes

The project filehas attributes for the metrics for the results of synthesis.

results_available	**Type**: Boolean. True if the config has been run and results can be loaded.
last_phase	**Type**: String. One of `elab`, `optim`, `sched`, or `rtl`, indicating the last phase that has been completed. If the config has not been processed, the value will be `unset`.
job_status	**Type**: Enumeration. The value will be one of the following: • `unrun`: The config has not been executed. • `running`: The config is being processed. • `complete_ok`: The config completed successfully. • `complete_error`: The config completed with an error.
runtime	**Type**: Integer. Specifies the elapsed time for the last run of this config in seconds.
last_run	**Type**: String. Specifies the date and time of last execution.
version	**Type**: String. Specifies the version of Stratus used for the last execution.
bits	**Type**: Integer. Specifies the number of register bits in the part.
total_area	**Type**: Real number of square microns or luts. Specifies the total area of the part.
seq_area	**Type**: Real number of square microns or luts. Specifies the sequential area.

comb_area	**Type**: Real number of square microns or luts. Specifies the combinational area.

lib Object Attributes

A lib object is a container for library modules. There is a `lib` object for each `use_hls_lib` command in the project file. There is not a `lib` object for the cachelib.

name	**Type**: String. Specifies the base name of the .bdl file for the library. • This is `stratus_hls` for the on-the-fly library.
path	Type: String. Specifies the path to the library, relative to the project.

lib_module Object Attributes

A `lib_module` object is created for each module for which there is an `rtl_instance`. There is also a l`ib_module` object for each module in an `hls_lib` used by the project.

bits	**Type**: Integer. Specifies the number of register bits in the part.
comb_area	**Type**: Real number of square microns or luts. Specifies the combinational area.
delay	**Type**: Real number in techlib units. Specifies the worst case output delay.
function	**Type**: String. Describes the function with an enumerated value. If the module performs only one function, then a single word is returned. If the module performs several functions, then a list of the functions is returned. For example, a DPOPT module that performs 2 multiplications and an add would return `"mul mul add"`. The names returned are as described for the `kind` attribute of `op` objects for datapath modules. For other kinds of modules, one of the following is returned: `buffer, partition, RAM, ROM, reg_bank, scheduled_region, clock_gate`.
init_interval	**Type**: Integer. Specifies the initiation interval for the module if it is pipelined, or `0` if it is not.

kind	**Type**: String. One of: • `clock_gate` • `function` • `mux` • `partition` • `ram` • `rom` • `regbank` • `scheduled`
latency	**Type**: Integer. Specifies the latency of the module. `0` for async.
name	**Type**: String. Specifies the name of the module.
params	**Type**: Tcl list of 2-element lists. Specifies a name-value pair for each setting for building or configuring the module. These include things like `tech_lib`, `clock_period`, `input_delay`, etc., and for `cynw_cm_float`, things like E and M.
rtl_file	**Type**: String. Specifies the path to the Verilog RTL source for the module, relative to the project if it is a project library, or the library if it is not.
seq_area	**Type**: Real number of square microns or luts. Specifies the sequential area.
setup_time	**Type**: Real number in techlib units. Specifies the worst case setup time for parts with latency, `unset` for async.

total_area	**Type**: Real number of square microns or luts. Specifies the total area of the part.

lib_module_io Object Attributes

A `lib_module_io` is analogous to a `module_io`, but it describes a port on a `lib_module` or `rtl_module` rather than an `hls_config`. A separate kind of object is used so that `find -module_io` does not find library module ports. Other than that, `lib_module_io` has all the same attributes as does a `module_io`. See module_io Object Attributes

logic_synth_config Object Attributes

A logic_synth_config objectexists for each define_logic_synth_config command in the project file.

bits	**Type**: Integer. Specifies the number of flip flops.
comb_area	**Type**: Real number of square microns or luts. Specifies the combinational area.
config_specs	**Type**: List of strings. Contains a list of `hls_config` objects for all modules processed by this config.
job_status	**Type**: Enumeration. One of the following: • `unrun`: The config has not been executed. • `running`: The config is being processed. • `complete_ok`: The config completed successfully. This does not mean timing was met. • `complete_error`: The config completed with an error.
last_run	**Type**: String. Specifies the date and time of last execution.
name	**Type**: String. Specifies the name of the config.
results_available	**Type**: Boolean. True if the config has been run, and results can be loaded.
runtime	**Type**: Integer. Specifies the elapsed time for the last run of this config in seconds.
seq_area	**Type**: Real number of square microns or luts. Specifies the sequential area.

slack	**Type**: Real number in library units. Specifies the worst case slack.
total_area	**Type**: Real number of square microns or luts. Specifies the total area.
type	**Type**: String. `logic_synth_config`

loop Object Attributes

A loop object represents a for, while, or do loop in the SystemC source code.
Each loop object has the attributes listed below, and also all of the attributes for region objects.

all region attributes	See region Object Attributes.
basic_blocks	**Type**: List of object identifiers. The `basic_block` objects that are either directly within this loop or within a nested loop.
header	**Type**: Object identifier. Specifies the basic block that is the header for this loop.
is_coalesced	**Type**: Boolean. Read only. This attribute is set by the `coalesce_loops` command.
is_pipelined	**Type**: Boolean. Read only. This attribute is set by the `pipeline_loops` command.
iterations	**Type**: Integer. Specifies the number of iterations of the loop if known and fixed.
latency_in_loop	**Type**: Integer or pair of integers. Specifies the number of cycles from the start of the enclosing loop to the start of the loop. The semantics are similar to the `latency_in_loop` attribute on `basic_block`.
lcds	**Type**: List of comma-separated pairs of object paths. Each LCD is defined by a source op object and a sink op. Each of these is represented by a comma-separated pair of object paths. For example, `op:design/BASIC/elab/thread1/OP14,op:design/BASIC/elab/thread1/OP27`
name (not inherited)	**Type:** String. Read only. The name of the loop given by a label outside the loop, a directive immediately inside the loop, or by the naming convention `loop_<N>_<M>` where N and M are the order of the loop in its loop nest.

nested_loops	**Type:** List of strings. Read only. A list of paths to nested loops. Empty for innermost loops.
outer_loop	**Type:** String. An object path for the parent loop, or `unset` for the root loop.
parent	**Type**: Object path. Specifies the path of the parent object.
pipeline_ii	**Type**: Integer. This attribute is set by the `pipeline_loops` command.
pipeline_type	**Type**: Enumeration. One of HARD_STALL, SOFT_STALL. The default value is HARD_STALL. This attribute is set by the `pipeline_loops` command.
scheduled_latency	**Type**: Integer. Specifies the scheduled latency of the loop. `unset` at `optim` or earlier.
scope	**Type**: Object path. Specifies the path of the parent scope.
src_links	**Type:** String. Describes the location of the loop in the source code.
unroll_count	**Type**: Integer. The default value is -1 (means calculate). This attribute is set by the `unroll_loops` command.
unroll_type	**Type**: Enumeration. One of ON, OFF, ALL, AGGRESSIVE, CONSERVATIVE. The default value is UNDEF (means depends on context). This attribute is set by the `unroll_loops` command.
will_be_unrolled	**Type**: Boolean. True if the loop will be unrolled during optimization. Never true in the `optim` or later snapshot.

message Object Attributes

A `message` object will be present for each numbered message that appears in the Stratus log file. Messages are not snapshotted. They always include all the messages produced in the run.

code	**Type**: Integer. The integer code printed with the message.
name	**Type**: String. Specifies an arbitrary name given to each message.
phase	**Type**: String. Specifies the execution phase during which the message was reported.
severity	**Type**: Enumeration. One of FATAL, ERROR, WARNING, or NOTE.
src_links	**Type**: String. Specifies a source location associated with the message or `unset`.
text	**Type**: String. The message text.

module_io Object Attributes

A module_io represents a scalar port or signal in a module. If the type of the port or signal contain a struct, class or array, there will be a separate module_io object for each leaf-level object.

behavior_ios	**Type**: List of behavior_io objects. Contains each behavior_io object in the same hls_config for this port or signal.
clock_period	**Type**: Real. The alternate clock period set for this clock signal using the `set_clock_period` command. If the port or signal is a clock, and does not have an alternate period set for it, the default clock period is given. For non-clock ports or signals, the value is `unset`.
direction	**Type**: Enumeration. Read only. Value is `input`, `output` or `none` (for signals, which do not have a direction). The direction of the port.
is_clock	**Type**: Boolean. Read only. True if the port or signal is a clock port or signal.
is_signal	**Type**: Boolean. True for inter-process signals, false for ports.
is_signed	**Type:** Boolean. Read only. True if the type of the port or signal is signed.
name	**Type:** String. Read only. The name of the port or signal.
parent	**Type**: Object path. Specifies the path of the parent object.
scope	**Type**: Object path. Specifies the path of the parent scope.

src_links	**Type**: String or list of strings. Gives the location in the SystemC source where the port or signal is declared.
width	**Type**: Integer. Read only. The number of bits in the port or signal's data type.

op Object Attributes

An `op` object (short for *operation*) represents an individual activity in a behavior. Operations are the nodes in the DFG, and they are connected to the CFG by being uniquely associated with a basic_block. Once synthesis is complete, an operation may have an implementation on one or more RTL objects.

after_unroll	**Type**: Enumeration. Predicts the constant-ness of the operation after loops are unrolled. • `const`: The value depends only on constants, and loop LCDs whose initial values are constants, and is predicted to be either constant or optimized away by the `optim` phase. • `const_index`: The operation has an index input that is `const` and accesses an array that was not declared `const`. • `variable`: The operations value is not constant, even after loop unrolling. • `unset`: No constant-ness was calculated. This is always the case for the `optim` or later phases.
basic_block	**Type**: Object path. Specifies the path to the basic block that contains the operation.
behavior_objects	**Type**: An opject path, or a list of paths. If the operation operates on another object in the design, returns their paths. Object types are: • behavior_io • array
const_value	**Type**: String. Specifies the value if the op's type is `const`. `unset` if not `const`.
decl_links	**Type**: String. Specifies variable names and locations for variables related to the operation.

input_widths	**Type**: Tcl list of integers. An integer for each functional input giving its width. • The order of inputs is not defined but can be used with the `-port` option of the `get_inputs` command to find upstream connections for the input at the associated position. • Control inputs, such as tags, are not included.

kind	`**Type**: Enumeration.`
	`Describes the kind of operation. Values are:`
	• `none`
	• `input`
	• `output`
	• `const`
	• `lcd_mux`
	• `mux`
	• `wire`
	• `sub`
	• `uminus`
	• `not`
	• `sqrt`
	• `deref`
	• `add`
	• `and`
	• `eq`
	• `mul`

- `ne`
- `or`
- `xor`
- `div`
- `ge`
- `gt`
- `le`
- `lt`
- `rs`
- `ls`
- `mod`
- `array_read`
- `array_write`
- `var_range`
- `func_call`
- `dpopt`
- `sched_region`
- `floating`
- `floating_read`
- `floating_write`
- `rb`
- `rb_read`
- `rb_write`
- `reg_read`
- `reg_write`
- `branch`

label	**Type**: String. A descriptive label for the operation. For operators, gives a symbol like `*` or `+`. For storage and I/O operations, gives `<obj>:read` or `<obj>:write`. For dpopt and schedule region, gives the name of the region.
name	**Type**: String. A unique name for the op.
output_widths	**Type**: Tcl list of integers. An integer for each output giving its width. Most operations have a single output but some, like scheduled regions, may have more than one.
reg_op	**Type**: An object path. If the operation's output is stored in a register, gives the `op` node with `kind==reg` that describes storage in the register. `reg` kind `op` nodes are present in the `sched` and `rtl` snapshots, but are never returned by `get_fanin` or `get_fanout`, so this attribute is provided to access register storage ops that are part of the dataflow.
rtl_instances	**Type**: An object path, or a list of paths. One or more object paths to `rtl_instance` objects. Only available at `rtl` snapshot.
rtl_signals	**Type**: An object paht, or a list of paths. One or more object paths to `rtl_signal` objects. • Only available at `rtl` snapshot. • The signals may be registers, wires, or ios. Note that these are `rtl_signal`, not `behavor_io`, so they represent netlist items.

src_links	**Type**: String. The source file location of the op.

power_config Object Attributes

A power_config objectexists for each define_power_config command in the project file.

config_specs	**Type**: List of strings. Contains a list of the `hls_config` objects analyzed by this config.
job_status	**Type**: Enumeration. One of the following: `unrun`: The config has not been executed. `running`: The config is being processed. `complete_ok`: The config completed successfully. `complete_error`: The config completed with an error.
last_run	**Type**: String. Specifies the date and time of last execution.
name	**Type**: String. Specifies the name of the config.
results_available	**Type**: Boolean. True if the config has been run, and results can be loaded.
runtime	**Type**: Integer. Specifies the elapsed time for the last run of this config in seconds.
type	**Type**: String. `power_config`

region Object Attributes

Any of the following is termed as a region in Stratus HLS:

- Every loop implicitly has a region that can be accessed through the loop's name.
- Every behavior implicitly has a region with the same name as the behavior.
- If a C++ label appears immediately before a curly-brace block, it names that block as a region object.
- If a pair of C++ labels appears in the same block, a region is defined for each pair where the first dominates the second. The names of these regions are: from__to_
- If a block directive (for example, HLS_DEFINE_PROTOCOL_PROTOCOL and so on) has a string label argument, that label can be used to identify the corresponding region.

A C++ curly-brace block may or may not have a name. If a block doesn't have a name, it will not be accessible as a region object

basic_blocks	**Type**: List of `basic_block` object identifiers. Specifies all the basic blocks that are part of a region. When regions overlap, a given basic block may appear in more than one `basic_blocks` attribute.
entry	**Type**: Object identifier. Specifies all `basic_block` objects that are at the beginning of this region.
fp_context	**Type**: String or integer. Multiple floating protocol regions with the same fp_context will not be scheduled concurrently. This attribute is set by the `define_floating_protocol` command.
fp_delay	**Type**: Real. The default value is 0.0. The output delay for the device being accessed by the protocol. This attribute is set by the `define_floating_protocol` command.

fp_flags	**Type**: Any combination of HLS_MEM_READ_FP, HLS_MEM_WRITE_FP, HLS_UNSTALLABLE_FP, HLS_IGNORE_BOUNDARIES, HLS_NO_CHAIN_MEM_IN, HLS_NO_CHAIN_MEM_OUT, HLS_CHAIN_MEM_IN, HLS_CHAIN_MEM_OUT, HLS_NO_SPEC_READS, HLS_ALLOW_SPEC_READS. The default value is 0. This attribute is set by the `define_floating_protocol` command.
fp_ii	**Type**: Integer. The default value is 1. The initiation interval for the floating protocol. This attribute is set by the `define_floating_protocol` command.
fp_setup_time	**Type**: Real. The default value is 0.0. The setup time for the device being accessed by the protocol. This attribute is set by the `define_floating_protocol` command.
is_control_removed	**Type**: Boolean. True if conditional control will be removed and explicit muxes created for this region. This is set by the `remove_control` command.
is_dpopt	**Type**: Boolean. Default is false. True if the region is to be dpopt'ed. This attribute is set by the `dpopt_region` and `schedule_region` commands. Can be set to false to disable DPOPT for a region that contain an HLS_DPOPT_REGION directive.
is_fp	**Type**: Boolean. Read only. This attribute is set by the `define_floating_protocol` command.
is_latency_constrained	**Type**: Boolean. Read only. True if a latency constraint exists for this region.
is_protocol	**Type**: Boolean. The default value is 0. This attribute is set by the `define_protocol` command. Note that a region that is within another protocol region but does not define its own protocol region will be false.

is_scheduled	**Type**: Boolean. The default value is 0. True if the region is to be scheduled separately using the behavioral synthesis flow. This attribute is set by the `dpopt_region` and `schedule_region` commands.
max_latency	**Type**: Integer. The default value is -1 (means no max). This attribute is set by the `constrain_latency` command.
min_latency	**Type**: Integer. The default value is 0. This attribute is set by the `constrain_latency` command.
name	**Type**: String. Read-only. The name of the region.
outer_region	**Type**: Object identifier. Gives all the `region, loop` or `behavior` objects that this region is immediately within.
parent	**Type**: Object path. Specifies the path of the parent object.
region_flags	**Type**: Enumeration. • one of (DPOPT_DEFAULT, REGION_DEFAULT, BEFORE_INLINE, BEFORE_UNROLL, AFTER_UNROLL) • any of (NO_CONSTANTS, NO_CSE, NO_DCE, NO_DEAD, NO_ENABLE, NO_TRIMMING, WITH_ENABLE, NO_CHAIN_IN, NO_CHAIN_OUT, DPOPT_DISABLE, DPOPT_FPGA_USE_DSP) The `DPOPT_FPGA_USE_DSP` attribute can be used only with the `dpopt_region` command. This attribute is set by the `dpopt_region` and `schedule_region` commands.

region_ii	**Type**: `Integer. Read only.` For scheduled regions, this attribute is set via the `schedule_region` command. For dpopt regions, the value is always 1. For other regions, the value is 0.
region_input_delay	**Type**: Real. The default value is 0.0. Specifies the input delay after which execution of the resource created for the region will begin. Synthesis of the resource created for the region will be constrained by this value. This attribute is set by the `constrain_region` command.
region_max_delay	**Type**: Real. The default value is the clock period. Specifies the delay after which the output of the resource created for this region must be stable. Synthesis of the resource created for the region will be constrained by this value. This attribute is set by the `constrain_region` command.
region_max_latency	**Type**: Integer. The default value is -1 (means no limit). Specifies the maximum latency for the resource created for this region. This attribute is set by the `constrain_region` command.
region_min_latency	**Type**: Integer. The default value is 0. Specifies the minimum latency for the resource created for this region. This attribute is set by the `constrain_region` command.
scheduled_latency	**Type**: Integer. Gives the number of cycles achieved by scheduling for a latency-constrained region. Only available in `sched` and `rtl` snapshots.
scope	**Type**: Object path. Specifies the path of the parent scope.
src_links	**Type**: String or list of strings. Gives a string describing the location of the start and end of this region in the source code.

stable_inputs	**Type**: array of paths to `behavior_terms`. Gives behavior inputs that can be assumed to be stable within this region. This attribute is set by the `assume_stable` command.
untimed_inputs	**Type**: array of paths to `behavior_terms`. Gives the set of stable behavior inputs that may take longer than 1 cycle to initialize, but that will be stable throughout the execution of the region. This attribute is set by the `assume_stable` command.

rtl_instance Object Attributes

An `rtl_instance` represents an instantiation of a submodule in an `rtl_module`. It may represent a real instance, or an instance that has been ungrouped. Access to input and output signals is given by the `get_fanin` and `get_fanout` functions.

behavior	**Type**: Object path. Specifies the behavior object from which this was inferred.
kind	**Type**: Enumeration. Specifies the kind of module of which this is an instance. One of: • `function` • `mux` • `memory` • `reg_bank` • `partition` • `hls_module` • `black_box`
decl_links	**Type**: String. Gives the declaration location for an associated explicit module intantiation, or array declaration in SystemC.
module	**Type**: Object path. Specifies the path to the object of which this instance is one. It may be any of: • `rtl_module`: For a structural partition for this `hls_module`. • `lib_module`: For an instance of a library module. • `unset`: For an instance of a black-boxed module.
name	**Type**: String. Specifies the name of the instance in the RTL.

operations	Type: List of object paths. Contains operations in the schedule for this object.
rtl_module	**Type**: Object path. Specifies the `rtl_object` that contains the instance in its netlist.
src_links	**Type**: String. Specifies source locations of mapped operations. Would be the same as getting operations and getting their src_links.
ungrouped_to	**Type**: Object path, or array of object paths to `rtl_process`. Populated if the instance was ungrouped, and there is no actual instance in Verilog. Contains the `rtl_process` objects that implement the instance.

rtl_module Object Attributes

An `rtl_module` represents a structural partition in the RTL output of `stratus_hls` for an `hls_config`. There is an `rtl_module` for the topmost module itself, and for modules created for purposes such as scheduled regions or `--output_style_separate_behaviors`. The other RTL objects, `rtl_instance`, `rtl_signal`, and `rtl_process` are owned by the `rtl_module` in which they appear. When hierarchy exists, an `rtl_instance` also exists for the `rtl_module`, though the current version of `stratus_hls` never created more than one instance of a particular `rtl_module`. This makes an `rtl_module` `a` container for netlists in Stratus output.

The parent/child relationship of an `rtl_module` to the objects it contains should be used to find those objects. There is no attribute that lists the contents of an `rtl_module`.

behavior	**Type**: Object path. An object path for a behavior for which this `rtl_module` was created. This may be a scheduled region, or it ma be an SC_CTHREAD or SC_METHOD if `output_style_separate_behaviors` was used.
function	**Type**: String. Describes their purpose, for hierarchical instances: • `partition`: created for options like `--output_style_separate_memories` and `--output_style_separate_behaviors` • `reg_bank`: a register bank • `scheduled_region`: scheduled region
is_top	**Type**: Boolean. True for the top RTL module.
name	**Type**: String. Specifies the name of the module.
rtl_file	**Type**: String. The path to the Verilog RTL source file, relative to the project directory.

rtl_instances	**Type**: List of object paths. Contains the rtl_instance for each instantiation of this object. • In practice, there will never be more than 1. • The top-level module has no instance.

rtl_process Object Attributes

Represents an always block or assign statement in the RTL.

behavior	**Type**: Object path. Specifies the behavior object from which this was inferred.
clock	**Type**: Object path. Specifies the path to `rtl_signal` object for the process' clock, or an empty string for async processes.
name	**Type**: String. Specifies the name of the process in the RTL, or empty string.
operations	**Type**: List of object paths. Contains operations in the schedule for this object. For ungrouped processes, this is the set of operations for the `rtl_instance`. Otherwise, it is the set of operations for the driven `rtl_signal`.
resets	**Type:** List of strings The resets for this process, encoded as strings of the form <port>,<polarity>,<sync|async>. One triplet for each reset for the process, in order of priority. For example `{"rtl_signal:filter/filter1/BASIC/rtl/filter1/arst,0,async" "rtl_signal:filter/filter1/BASIC/rtl/filter1/srst,1,sync"}` for a process with one active low async reset, and one active high sync reset.
rtl_module	**Type**: Object path. Specifies the `rtl_object` that contains the process in its netlist.
signal_written	**Type**: Object path. The `rtl_signal` written by this process. Though `get_fanout` can be used to traverse netlists in general, `get_fanout` for an rtl_process returns the readers of the written signal, not the written signal itself. This attribute provides a mechanism for getting that `rtl_signal` if it is needed.

src_links	**Type**: String. Specifies source locations of mapped operations. Would be the same as getting operations and getting their src_links.
ungrouped_from	**Type**: Object path. If this process is the result of ungrouping an `rtl_instance`, contains its path.

rtl_signal Object Attributes

An `rtl_signal` represents a port, wire, or register in an RTL module.

behavior	**Type**: Object path. Specifies the behavior object from which this was inferred.
decl_links	**Type**: String. Specifies variable names and locations for variables related to the signal.
is_register	**Type**: Boolean. True if written by a clocked thread.
is_signed	**Type**: Boolean. True if the signal is signed value.
is_state	**Type**: Boolean. True if the signal is part of FSM control.
module_io	**Type**: Object path. Specifies the `rtl_object` that contains the signal in its netlist.
name	**Type**: String. Specifies the name of the object in the RTL.
operations	**Type**: List of object paths. Contains operations in the schedule for this object. Registers, wires and ports can have read and write operations.
rtl_module	**Type**: Object path. Specifies the `rtl_object` that contains the signal in its netlist.
src_links	**Type**: String. Specifies source locations of mapped operations.Would be the same as getting operations and getting their src_links.

width	**Type**: Integer. Specifies the width of the signal.
writer	**Type**: Object path. Specifies the `rtl_process` or `rtl_instance` that writes the signal. Though `get_fanin` can be used to traverse netlists in general, `get_fanin` for an rtl_signal returns the inputs to a writing rtl_process, not the rtl_process itself. This attribute provides a mechanism for getting the `rtl_process` if it is needed.

sc_instance Object Attributes

An `sc_instance` object represents an instantiation of a module in a SystemC model. Only non-inlined modules are represented. The module may be another `hls_module`, a library module, or a black-block module. `sc_instance` objects are only present in pre-`rtl` snapshots. In the `rtl` snapshot, there will ordinarily be a corresponding `rtl_instance`.

arrays	**Type**: List of object paths. Specifies the array objects for the external arrays bound to this instance.
kind	**Type**: Enumeration. One of: • `hls_module` • `lib_module` • `black_box`
module	**Type**: Object path. If hls_module or lib kind, gives the path to the module's object.
module_ios	**Type**: List of object paths. Specifies the module_io objects for the ports and signals bound to this instance.
name	**Type**: String. Specifies the name of the variable from which the instance is accessed.
src_links	**Type**: String. Specifies the source location of the instantiation.

sim_config Object Attributes

A sim_config objectexists for each define_sim_config command in the project file.

config_specs	**Type**: List of strings.
	
	Contains list of comma-separated pairs giving each hls_config object and its representation in the sim_config. Representation values are BEH, RTL_V, or GATES_V. For example:
	
	`{hls_config:design/dut/BASIC,RTL_V`
	`hls_config:design/compute/BASIC,RTL_V}`

job_status	**Type**: Enumeration. One of the following: • `unrun` : The config has not been executed. • `running` : The config is being processed. • `complete_ok` : The config completed successfully. This does not mean the simulation passed. • `complete_error`: The config completed with an error.
last_run	**Type**: String. Specifies the date and time of last execution.
name	**Type**: String. Specifies the name of the config.
passed	**Type**: Boolean. True if a pass was logged, false if a failure was logged, and `unset` if no status was logged.
results_available	**Type**: Boolean. True if the config has been run, and results can be loaded.
runtime	**Type**: Integer. Specifies the elapsed time for the last run of this config in seconds.

type	**Type**: String. sim_config

snapshot Object Attributes

A `snapshot` object represents a snapshot in an `hls_config`. The `snapshot` object will exist in any design that has been run past the point that would create the snapshot. A snapshot will exist before it has been loaded.

name	**Type**: Enumeration. One of the following: • `elab` • `optim` • `sched` • `rtl`
is_loaded	**Type**: Boolean. True if the snapshot has been loaded.

5

Index