



Intel[®] High Level Synthesis Compiler Pro Edition

Reference Manual

Updated for Intel[®] Quartus[®] Prime Design Suite: **19.4**



MNL-1083 | 2020.02.10

Latest document on the web: [PDF](#) | [HTML](#)



Contents

- 1. Intel® HLS Compiler Pro Edition Reference Manual..... 5**
- 2. Compiler..... 7**
 - 2.1. Intel HLS Compiler Pro Edition Command Options..... 7
 - 2.2. Using Libraries in Your Component..... 11
 - 2.3. Compiler Interoperability..... 11
 - 2.4. Intel HLS Compiler Pipeline Approach..... 13
- 3. C Language and Library Support..... 15**
 - 3.1. Supported C and C++ Subset for Component Synthesis..... 15
 - 3.2. C and C++ Libraries..... 15
 - 3.3. Templated and Overloaded Functions..... 17
 - 3.3.1. Templated Functions..... 17
 - 3.3.2. Overloaded Functions..... 18
 - 3.3.3. Function Name Mapping..... 18
 - 3.4. Compiler-Defined Preprocessor Macros..... 19
- 4. Component Interfaces..... 20**
 - 4.1. Component Invocation Interface..... 20
 - 4.1.1. Scalar Parameters..... 21
 - 4.1.2. Pointer and Reference Parameters..... 21
 - 4.1.3. Interface Definition Example: Component with Both Scalar and Pointer Arguments..... 21
 - 4.2. Avalon Streaming Interfaces..... 22
 - 4.3. Avalon Memory-Mapped Master Interfaces..... 25
 - 4.3.1. Memory-Mapped Master Testbench Constructor..... 26
 - 4.3.2. Implicit and Explicit Examples of Describing a Memory Interface..... 26
 - 4.3.3. Avalon Memory-Mapped Master Interfaces and Load-Store Units..... 28
 - 4.4. Slave Interfaces..... 33
 - 4.4.1. Control and Status Register (CSR) Slave..... 34
 - 4.4.2. Slave Memories..... 36
 - 4.5. Component Invocation Interface Control Attributes..... 37
 - 4.6. Unstable and Stable Component Parameters..... 38
 - 4.7. Global Variables..... 39
 - 4.8. Structs in Component Interfaces..... 39
 - 4.9. Reset Behavior..... 39
- 5. Component Memories (Memory Attributes)..... 41**
 - 5.1. Static Variables..... 46
- 6. Loops in Components..... 48**
 - 6.1. Loop Initiation Interval (`ii` Pragma)..... 49
 - 6.2. Loop-Carried Dependencies (`ivdep` Pragma)..... 50
 - 6.3. Loop Coalescing (`loop_coalesce` Pragma)..... 52
 - 6.4. Loop Unrolling (`unroll` Pragma)..... 53
 - 6.5. Loop Concurrency (`max_concurrency` Pragma)..... 54
 - 6.6. Loop Iteration Speculation (`speculated_iterations` Pragma)..... 54
 - 6.7. Loop Pipelining Control (`disable_loop_pipelining` Pragma)..... 56



6.8. Loop Interleaving Control (<code>max_interleaving</code> Pragma).....	57
7. Component Concurrency.....	58
7.1. Serial Equivalence within a Memory Space or I/O.....	58
7.2. Concurrency Control (<code>hls_max_concurrency</code> Attribute).....	58
7.3. Component Pipelining Control (<code>hls_disable_component_pipelining</code> Attribute)....	59
8. Arbitrary Precision Math Support.....	61
8.1. Declaring <code>ac_int</code> Data Types.....	63
8.1.1. Important Usage Information on the <code>ac_int</code> Data Type.....	64
8.2. Integer Promotion and <code>ac_int</code> Data Types.....	64
8.3. Debugging Your Use of the <code>ac_int</code> Data Type.....	65
8.4. Declaring <code>ac_fixed</code> Data Types.....	65
8.5. Declaring <code>ac_complex</code> Data Types.....	66
8.6. AC Data Types and Native Compilers.....	67
8.7. Declaring <code>hls_float</code> Data Types.....	67
8.7.1. Operators and Return Types Supported by the <code>hls_float</code> Data Type.....	69
9. Component Target Frequency.....	74
10. Systems of Tasks.....	75
10.1. Task Functions	75
10.2. Internal Streams.....	80
10.3. System of Tasks Simulation.....	81
11. Libraries.....	82
11.1. Object Libraries.....	83
11.2. Creating an Object Library.....	84
11.3. Creating Objects From HLS Code.....	85
11.3.1. Creating an Object File From HLS Code.....	85
11.3.2. Supported OpenCL Language Constructs.....	86
11.4. Creating Objects From RTL Code.....	87
11.4.1. RTL Modules and the HLS Pipeline.....	89
11.4.2. Creating an HLS-Library Object File from an RTL Module	99
11.5. Packaging Object Files Into a Library.....	100
12. Advanced Hardware Synthesis Controls.....	102
12.1. The <code>hls_fpga_reg()</code> Function.....	102
13. Intel High Level Synthesis Compiler Pro Edition Reference Summary.....	104
13.1. Intel HLS Compiler Pro Edition <code>i++</code> Command-Line Arguments.....	104
13.2. Intel HLS Compiler Pro Edition Header Files.....	106
13.3. Compiler-Defined Preprocessor Macros.....	109
13.4. Intel HLS Compiler Pro Edition Keywords.....	110
13.5. Intel HLS Compiler Pro Edition Simulation API (Testbench Only).....	110
13.6. Intel HLS Compiler Pro Edition Component Memory Attributes.....	112
13.7. Intel HLS Compiler Pro Edition Loop Pragmas.....	118
13.8. Intel HLS Compiler Pro Edition Scope Pragmas.....	123
13.9. Intel HLS Compiler Pro Edition Component Attributes.....	124
13.10. Intel HLS Compiler Pro Edition Component Default Interfaces.....	126
13.11. Intel HLS Compiler Pro Edition Component Invocation Interface Control Attributes...126	
13.12. Intel HLS Compiler Pro Edition Component Macros.....	128



- 13.13. Systems of Tasks API..... 130
 - 13.13.1. `ihc::stream` Class..... 133
- 13.14. Intel HLS Compiler Pro Edition Streaming Input Interfaces..... 134
- 13.15. Intel HLS Compiler Pro Edition Streaming Output Interfaces..... 139
- 13.16. Intel HLS Compiler Pro Edition Memory-Mapped Interfaces..... 143
- 13.17. Intel HLS Compiler Pro Edition Load-Store Unit Control..... 147
- 13.18. Intel HLS Compiler Pro Edition Arbitrary Precision Data Types..... 149
- A. Advanced Math Source Code Libraries..... 151**
 - A.1. Random Number Generator Library..... 151
 - A.2. Matrix Multiplication Library..... 152
- B. Supported Math Functions..... 154**
 - B.1. Math Functions Provided by the `math.h` Header File 154
 - B.2. Math Functions Provided by the `extendedmath.h` Header File..... 158
 - B.3. Math Functions Provided by the `ac_fixed_math.h` Header File..... 160
 - B.4. Math Functions Provided by the `hls_float.h` Header File..... 160
 - B.5. Math Functions Provided by the `hls_float_math.h` Header File..... 160
- C. Intel HLS Compiler Pro Edition Reference Manual Archives..... 162**
- D. Document Revision History of the Intel HLS Compiler Pro Edition Reference Manual..163**



1. Intel® HLS Compiler Pro Edition Reference Manual

The *Intel® HLS Compiler Pro Edition Reference Manual* provides reference information about the features supported by the Intel HLS Compiler Pro Edition. The Intel HLS Compiler is sometimes referred to as the i++ compiler, reflecting the name of the compiler command.

In this publication, `<quartus_installdir>` refers to the location where you installed Intel Quartus® Prime Design Suite.

The default Intel Quartus Prime Design Suite installation location depends on your operating system:

Windows C:\intelFPGA_pro\19.4

Linux /home/<username>/intelFPGA_pro/19.4

About the Intel HLS Compiler Documentation Library

Documentation for the Intel HLS Compiler is split across a few publications. Use the following table to find the publication that contains the Intel HLS Compiler information that you are looking for:

Table 1. Intel High Level Synthesis Compiler Documentation Library

Title and Description	PRO	STD
Release Notes Provide late-breaking information about the Intel HLS Compiler.	Link	Link
Getting Started Guide Get up and running with the Intel HLS Compiler by learning how to initialize your compiler environment and reviewing the various design examples and tutorials provided with the Intel HLS Compiler.	Link	Link
User Guide Provides instructions on synthesizing, verifying, and simulating intellectual property (IP) that you design for Intel FPGA products. Go through the entire development flow of your component from creating your component and testbench up to integrating your component IP into a larger system with the Intel Quartus Prime software.	Link	Link
<i>continued...</i>		



Title and Description	PRO	STD
<i>Reference Manual</i> Provides reference information about the features supported by the Intel HLS Compiler. Find details on Intel HLS Compiler command options, header files, pragmas, attributes, macros, declarations, arguments, and template libraries.	Link	Link
<i>Best Practices Guide</i> Provides techniques and practices that you can apply to improve the FPGA area utilization and performance of your HLS component. Typically, you apply these best practices after you verify the functional correctness of your component.	Link	Link
<i>Quick Reference</i> Provides a brief summary of Intel HLS Compiler declarations and attributes on a single two-sided page.	Link	Link

2. Compiler

2.1. Intel HLS Compiler Pro Edition Command Options

Use the Intel HLS Compiler command options to customize how the compiler performs general functions, customize file linking, or customize compilation.

Table 2. General Command Options

These i++ command options perform general compiler functions.

Command Option	Description
--debug-log	Instructs the compiler to generate a log file that contains diagnostic information. By default, the <code>debug.log</code> file is in the <code>a.prj</code> subdirectory within your current working directory. If you also include the <code>-o <result></code> command option, the <code>debug.log</code> file will be in the <code><result>.prj</code> subdirectory. If your compilation fails, the <code>debug.log</code> file is generated whether you set this option or not.
-h or --help	Instructs the compiler to list all the command options and their descriptions on screen.
-o <result>	Instructs the compiler to place its output into the <code><result></code> executable and the <code><result>.prj</code> directory. If you do not specify the <code>-o <result></code> option, the compiler outputs an <code>a.out</code> file for Linux and an <code>a.exe</code> file for Windows. Use the <code>-o <result></code> command option to specify the name of the compiler output. Example command: <code>i++ -o hlsoutput multiplier.c</code> Invoking this example command creates an <code>hlsoutput</code> executable for Linux and an <code>hlsoutput.exe</code> for Windows in your working directory.
-v	Verbose mode that instructs the compiler to display messages describing the progress of the compilation. Example command: <code>i++ -v hls/multiplier/multiplier.c</code> , where <code>multiplier.c</code> is the input file.
--version	Instructs the compiler to display its version information on screen. Command: <code>i++ --version</code>

Table 3. Command Options that Customize Compilation

These i++ command options perform compiler functions that impact the translation from source file to object file.

Option	Description
-c	Instructs the compiler to preprocess, parse, and generate object files (<code>.o/.obj</code>) in the current working directory. The linking stage is omitted. Example command: <code>i++ -march="Arria 10" -c multiplier.c</code>

continued...



Option	Description
	Invoking this example command creates a multiplier.o file and sets the name of the <result>.prj directory to multiplier.prj. When you later link the .o file, the -o option affects only the name of the executable file. The name of the <result>.prj directory remains unchanged from when the directory name was set by i++ -c command invocation.
--component <components>	Allows you to specify a comma-separated list of function names that you want the compiler to synthesize to RTL. Example command: i++ counter.cpp --component count To use this option, your component must be configured with C-linkage using the extern "C" specification. For example: <pre>extern "C" int myComponent(int a, int b)</pre> Using the component function attribute is preferred over using the --component command option to indicate functions that you want the compiler to synthesize.
-D<macro> [= <val>]	Allows you to pass a macro definition (<macro>) and its value (<val>) to the compiler. If you do not specify a value for <val>, its default value will be 1.
-g	Generate debug information (default).
-g0	Do not generate debug information.
--gcc-toolchain=<GCC_dir>	Specifies the path to a GCC installation that you want to use for compilation. This path should be the absolute path to the directory that contains the GCC lib, bin, and include folders. You should not need to use this if you configured your system as described in the Getting Started Guide.
--hyper-optimized-handshaking=[auto off]	This option applies to Intel Stratix® 10 devices only. Use this option to modify the handshaking protocol used in certain areas of your design. By default, the --hyper-optimized-handshaking option is set to auto. When you enable this optimization, the compiler adds pipeline registers to the handshaking paths of the stallable nodes. This optimization results in a higher f _{MAX} at the cost of increased area and latency due to the added registers. Disabling this optimization typically decreases area and latency at the cost of lower f _{MAX} . <i>Restriction:</i> This option applies only to designs targeting Intel Stratix 10. If you use this option when you target devices other than Intel Stratix 10 devices, the compiler exits with an error.
-I<dir>	Adds a directory (<dir>) to the end of the include path list.
-march=[x86-64 <FPGA_family> <FPGA_part_number>]	Instructs the compiler to compile the component to the specified architecture or FPGA family. The -march compiler option can take one of the following values: x86-64 Instructs the compiler to compile the code for an emulator flow. "<FPGA_family>" Instructs the compiler to compile the code for a target FPGA device family. The <FPGA_family> value can be any of the following device families: <ul style="list-style-type: none"> • Arria10 or "Arria 10" • Cyclone10GX or "Cyclone 10 GX" • Stratix10 or "Stratix 10" Quotation marks are required only if you specify a FPGA family name specifier that contains spaces

continued...



Option	Description
	<p><code><FPGA_part_number></code> Instructs the compiler to compile the code for a target device. The compiler determines the FPGA device family from the FPGA part number that you specify here.</p> <p>If you do not specify this option, <code>-march=x86-64</code> is assumed.</p> <p>If the parameter value that you specify contains spaces, surround the parameter value in quotation marks.</p>
<p><code>--quartus-compile</code></p>	<p>Compiles your HDL file with the Intel Quartus Prime compiler.</p> <p>Example command: <code>i++ --quartus-compile <input_files> -march="Arria 10"</code></p> <p>When you specify this option, the Intel Quartus Prime compiler is run after the HDL is generated. The compiled Intel Quartus Prime project is put in the <code><result>.prj/quartus</code> directory and a summary of the FPGA resource consumption and maximum clock frequency is added to the high level design reports in the <code><result>.prj/reports</code> directory.</p> <p>This compilation is intended to estimate the best achievable f_{MAX} for your component. Your component is not expected to cleanly close timing in the reports.</p>
<p><code>--quartus-seed <seed></code></p>	<p>Specifies the seed value that is used by Intel Quartus Prime project located in the <code><result>.prj/quartus</code> directory.</p> <p>This seed value is used by the Intel Quartus Prime Fitter for initial placement configuration when optimizing design placement to meet timing requirements (f_{MAX}).</p>
<p><code>--simulator <simulator_name></code></p>	<p>Specifies the simulator you are using to perform verification.</p> <p>This command option can take the following values for <code><simulator_name></code>:</p> <ul style="list-style-type: none"> modelsim none <p>If you do not specify this option, <code>--simulator modelsim</code> is assumed.</p> <p>Important: The <code>--simulator</code> command option only works in conjunction with the <code>-march</code> command option.</p> <p>The <code>--simulator none</code> option instructs the HLS compiler to skip the verification flow and generate RTL for the components without generating the corresponding test bench. If you use this option, the high-level design report (<code>report.html</code>) is generated more quickly but you cannot co-simulate your design. Without data from co-simulation, the report must omit verification statistics such as component latency.</p> <p>Example command: <code>i++ -march="<FPGA_family_or_part_number>" --simulator none multiplier.c</code></p>
<p><code>-ffp-contract=fast</code></p>	<p>Remove intermediate rounding and conversion when possible, except for code blocks fenced by <code>#pragma clang fp contract(off)</code>.</p> <p>To learn more, review the following tutorial: <code><quartus_installdir>/hls/examples/tutorials/best_practices/floating_point_ops</code></p>
<p><code>--fpc</code></p>	<p>This option is deprecated and will be removed in a future release. Use <code>-ffp-contract=fast</code> instead.</p> <p>Remove intermediate rounding and conversion when possible.</p> <p>Exception: Intermediate rounding and conversion is not removed in code blocks fenced by <code>#pragma clang fp contract(off)</code>.</p> <p>To learn more, review the following tutorial: <code><quartus_installdir>/hls/examples/tutorials/best_practices/floating_point_ops</code></p>
<p><code>-ffp-reassoc</code></p>	<p>Relax the order of floating point arithmetic operations, except for code blocks fenced by <code>#pragma clang fp reassoc(off)</code></p> <p>To learn more, review the following tutorial: <code><quartus_installdir>/hls/examples/tutorials/best_practices/floating_point_ops</code></p>
<p><code>--fp-relaxed</code></p>	<p>This option is deprecated and will be removed in a future release. Use <code>-ffp-reassoc</code> instead.</p> <p>Relax the order of floating point arithmetic operations.</p>

continued...

Option	Description
	<p>Exception: The order of floating point operations in code blocks fenced by <code>#pragma clang fp reassoc(off)</code> is not relaxed.</p> <p>To learn more, review the following tutorial: <code><quartus_installdir>/hls/examples/tutorials/best_practices/floating_point_ops</code></p>
<code>--daz</code>	Disable subnormal support in native IEEE-754 double-precision floating-point computations.
<code>--rounding={ieee faithful}</code>	<p>Control rounding scheme for native IEEE-754 double-precision adders, multipliers, and dividers.</p> <p>If you do not specify this option, adders and multipliers use IEEE-754 RNE rounding (0.5 ULP) and dividers use faithful rounding (1 ULP).</p> <p>The <code>--rounding</code> option can take one of the following values:</p> <p><code>ieee</code> All adders, multipliers, and dividers use IEEE-754 RNE rounding.</p> <p><code>faithful</code> All adders, multipliers, and dividers use faithful rounding.</p>
<code>--clock <clock target></code>	<p>Optimizes the RTL for the specified clock frequency or period.</p> <p>The <code><clock target></code> value must include a unit.</p> <p>For example:</p> <pre>i++ -march="Arria 10" test.cpp --clock 100MHz i++ -march="Arria 10" test.cpp --clock 10ns</pre>

Table 4. Command Options that Customize File Linking

These HLS command options specify compiler actions that impact the translation of the object file to the binary or RTL component.

Option	Description
<code>-ghdl</code>	<p>Logs all signals when running the verification executable. After running the executable, the simulator logs waveforms to the <code>a.prj/verification/vsim.wlf</code> file.</p> <p>For details about the ModelSim* waveform, see Debugging during Verification in <i>Intel High Level Synthesis Compiler Pro Edition User Guide</i>.</p>
<code>-L<dir></code>	(Linux only) Adds a directory (<code><dir></code>) to the end of the search path for the library files.
<code>-l<library></code>	(Linux only) Specifies the library file (<code>.a</code>) name when linking the object file to the binary. On Windows, you can list library files (<code>.lib</code>) on the command line without specifying any command options or flags.
<code>--x86-only</code>	<p>Creates only the testbench executable.</p> <p>The compiler outputs an <code><result></code> file for Linux or a <code><result>.exe</code> file for Windows. The <code><result>.prj</code> directory and its contents are not created.</p>
<code>--fpga-only</code>	<p>Creates only the <code><result>.prj</code> directory and its contents.</p> <p>The testbench executable file (<code><result>/<result>.exe</code>) is not created.</p> <p>Before you can co-simulate your hardware from a compilation output that uses this option, you must compile your testbench with the <code>--x86-only</code> option (or as part of a full compilation).</p>



2.2. Using Libraries in Your Component

Use libraries to reuse functions created by you or others without needing to know the function implementation details.

To use the functions in a library, you must have the C-header files (.h) for the library available. For object libraries, you must also have the object library archive file (.a on Linux systems or .lib on Windows systems) available.

Any object libraries that you use in your component must be built and used by the same version number Intel FPGA high-level design tool. For example, to compile your component with the Intel HLS Compiler Version 19.4, the libraries included in your component must have been created with a version 19.4 Intel FPGA high-level design tool. If you use a library with a different version, you get a version mismatch error when you compile your component.

To include a library in your component:

1. Review the header files corresponding to the library that you want to include in your component.

The header file shows you the functions available to call in the library and how to call the functions.

2. Include the header files in your component code.

For example, `#include "primitives.h"`

3. Compile your component with the Intel HLS Compiler as follows::

- For source code (that is, header-only) libraries, there is no additional library file name to specify.

For example, `i++ -march=arria10 MyComponent.cpp`

- For object libraries, ensure that you add the object library archive file name to the `i++` command.

For example, `i++ -march=arria10 MyComponent.cpp libprim.a`

Related Information

- [Libraries](#) on page 82
- [Advanced Math Source Code Libraries](#) on page 151
- [Arbitrary Precision Math Support](#) on page 61

2.3. Compiler Interoperability

The Intel High Level Synthesis Compiler is compatible with x86-64 object code compiled by supported versions of GCC or Microsoft Visual Studio. You can compile your testbench code with GCC or Microsoft Visual Studio, but generating RTL and cosimulation support for your component always requires the Intel HLS Compiler.

To see what versions of GCC and Microsoft Visual Studio the Intel HLS Compiler supports, see "[Intel High Level Synthesis Compiler Prerequisites](#)" in *Intel High Level Synthesis Compiler Getting Started Guide*.

The interoperability between GCC or Microsoft Visual Studio, and the Intel HLS Compiler lets you decouple your testbench development from your component development. Decoupling your testbench development can be useful for situations

where you want to iterate your testbench quickly with platform-native compilers (GCC/Microsoft Visual Studio), without having to recompile the RTL generated for your component.

To create only your testbench executable with the `i++` command, specify the `--x86-only` option.

You can choose to only generate RTL and cosimulation support for your component by linking the object file or files for your component with the Intel High Level Synthesis Compiler.

To generate only your RTL and cosimulation support for your component, specify the `--fpga-only` option.

To use a native compiler (GCC or Microsoft Visual Studio) to compile your Intel HLS Compiler code, you must point the native compiler to Intel HLS Compiler resources and libraries. The Intel HLS Compiler example designs contain build scripts (`Makefile` for Linux and `build.bat` for Windows) that you can use as examples of the required configuration. These scripts locate the Intel HLS Compiler installation, so you do not need to hard-code the locations in your build scripts.

GCC

The following instructions were tested with GCC compiler and C++ Libraries version 5.4.0.

To compile your Intel HLS Compiler code with GCC:

1. Add the Intel HLS Compiler header files to the `g++` command include path.
The header files are in the `quartus_installdir/hls/include` directory.
2. Add the HLS emulation library to the linker search path.
The emulation library is in the `quartus_installdir/hls/host/linux64/lib` directory.
3. Add the `hls_emul` library to the linker command (that is, specify `-lhls_emul.as` as a command option).
4. Ensure that you specify the `-std=c++14` option of the `g++` command.
5. If you are using HLS tasks in a system of tasks (`ihc::launch` and `ihc::collect`), specify the `-pthread` option of the `g++` command.
6. If you are using arbitrary precision datatypes, include the reference version instead of the FPGA-optimized version provided with the Intel HLS Compiler. You can use the `__INTELFPGA_COMPILER__` macro to control which version is included. For example, if you are using arbitrary precision integers, you can use the following macro code

```
#ifndef __INTELFPGA_COMPILER__
#include "HLS/ac_int.h"
#else
#include "ref/ac_int.h"
#endif
```



If you implement these steps, your `g++` command resembles the following example command:

```
g++ myFile.cpp -I"${HLS_INSTALL_DIR}/include" -L"${HLS_INSTALL_DIR}/host/  
linux64/lib" -lhls_emul -pthread -std=c++14
```

Microsoft Visual C++

The following instructions were tested with Microsoft Visual Studio 2017 Professional.

To compile your Intel HLS Compiler code with Microsoft Visual C++:

1. Add the Intel HLS Compiler header files to the compiler command include path.
The header files are in the `quartus_installdir\hls\include` directory.
2. Add the HLS emulation library to the linker search path.
The emulation library is in the `quartus_installdir\hls\host\windows64\lib` directory.
3. Add the `hls_emul` library to the linker command.
4. If you are using arbitrary precision datatypes, include the reference version instead of the FPGA-optimized version provided with the Intel HLS Compiler. You can use the `__INTELFPGA_COMPILER__` macro to control which version is included:

```
#ifdef __INTELFPGA_COMPILER__  
#include "HLS/ac_int.h"  
#else  
#include "ref/ac_int.h"  
#endif
```

Your Microsoft Visual C++ compiler command should resemble the following example command:

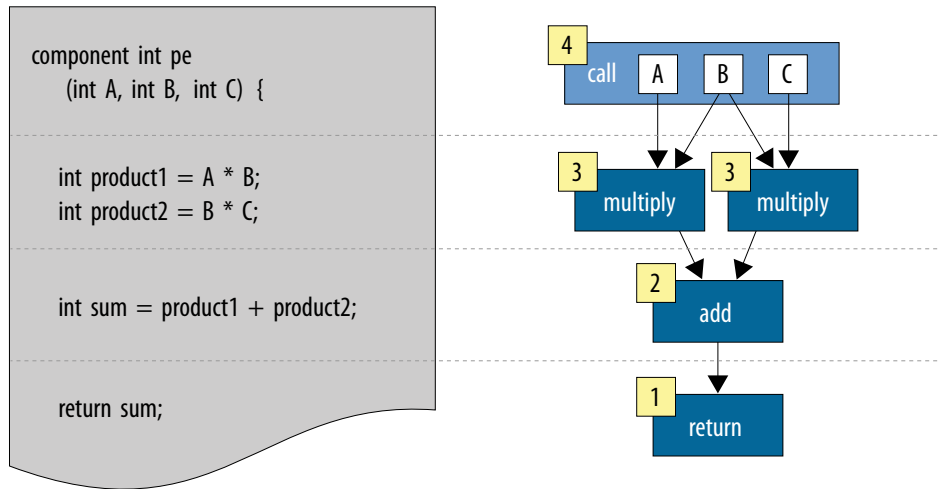
```
cl myFile.cpp /I "%HLS_INSTALL_DIR%\include" /nologo /EHsc /wd4068 /MD /stc:c+  
+14 /Zi  
/link "/libpath:%HLS_INSTALL_DIR%\host\windows64\lib" hls_emul.lib
```

2.4. Intel HLS Compiler Pipeline Approach

The Intel HLS Compiler attempts to pipeline functions as much as possible. Different stages of the pipeline might have multiple operations performed in parallel.

The following figure shows an example of the pipeline architecture generated by the Intel HLS Compiler. The numbered operations on the right side represent the pipeline implementation of the C++ code on the left side of the figure. Each box in the right side of the figure is an operation in the pipeline.

Figure 1. Example of Pipeline Architecture



With a pipelined approach, multiple invocations of the component can be simultaneously active. For example, the earlier figure shows that the first invocation of the component can be returning a result at the same time the fourth invocation of the component is called.

One invocation of a component advances to the its next stage in the pipeline only after all of the operations of its current stage are complete.

Some operations can stall the pipeline. A common example of operations that can stall a pipeline is a variable latency operation like a memory load or store operation. To support pipeline stalls, the Intel HLS Compiler propagates `ready` and `valid` signals through the pipeline to all operations that have a variable latency.

For operations that have a fixed latency, the Intel HLS Compiler can statically schedule the interaction between the operations and `ready` signals are not needed between the stages with fixed latency operations. In these cases, the compiler optimizes the pipeline to statically schedule the operations, which significantly reduces the logic required to implement the pipeline.

3. C Language and Library Support

3.1. Supported C and C++ Subset for Component Synthesis

The Intel HLS Compiler has several synthesis limitations regarding the supported subset of C99 and C++.

The compiler cannot synthesize code for dynamic memory allocation, virtual functions, function pointers, and C++ or C library functions except the supported math functions explicitly mentioned in the appendix of this document. In general, the compiler can synthesize functions that include classes, structs, functions, templates, and pointers.

While some C++ constructs are synthesizable, aim to create a component function in C99 whenever possible.

Important: These synthesis limitations do not apply to testbench code.

3.2. C and C++ Libraries

The Intel High Level Synthesis (HLS) Compiler provides a number of header files to provide FPGA implementations of certain C and C++ functions.

Table 5. Intel HLS Compiler Pro Edition Header Files Summary

HLS Header File	Description
HLS/hls.h	Required for component identification and component parameter interfaces.
HLS/math.h	Includes FPGA-specific definitions for the math functions from the <code>math.h</code> for your operating system.
HLS/extendedmath.h	Includes additional FPGA-specific definitions of math functions not in <code>math.h</code> .
HLS/ac_int.h	Provides FPGA-optimized arbitrary width integer support.
HLS/ac_fixed.h	Provides FPGA-optimized arbitrary precision fixed point support.
HLS/ac_fixed_math.h	Provides FPGA-optimized arbitrary precision fixed point math functions.
HLS/ac_complex.h	Provides FPGA-optimized complex number support.
HLS/hls_float.h	Provides FPGA-optimized arbitrary-precision IEEE IEEE 754 compliant floating-point number support.
<i>continued...</i>	

HLS Header File	Description
<code>HLS/hls_float_math.h</code>	Provides FPGA-optimized floating-point math functions.
<code>HLS/stdio.h</code>	Provides <code>printf</code> support for components so that <code>printf</code> statements work in x86 emulations, but are disabled in component when compiling to an FPGA architecture.
<code><iostream></code>	To use <code>cout</code> and <code>cerr</code> in your component, guard the statements with the <code>HLS_SYNTHESIS</code> macro.

math.h

To access functions in `math.h` from a component to be synthesized, include the "HLS/math.h" file in your source code. The header ensures that the components call the hardware versions of the math functions.

For more information about supported `math.h` functions, see [Supported Math Functions](#) on page 154.

stdio.h

Synthesized component functions generally do not support C and C++ standard library functions such as FILE pointers.

A component can call `printf` by including the header file `HLS/stdio.h`. This header changes the behavior of `printf` depending on the compilation target:

- For compilation that targets the x86-64 architecture (that is, `-march=x86-64`), the `printf` call behaves as normal.
- For compilation that targets the FPGA architecture (that is, `-march="<FPGA_family_or_part_number>"`), the compiler removes the `printf` call.

If you use `printf` in a component function without first including the `#include "HLS/stdio.h"` line in your code, you get an error message similar to the following error when you compile hardware to the FPGA architecture:

```
$ i++ -march="<FPGA_family_or_part_number>" --component dut test.cpp
Error: HLS gen_qsys FAILED.
See ./a.prj/dut.log for details.
```

You can use C and C++ standard library functions such as `fopen` and `printf` as normal in all testbench functions.

iostream

A component can use C++ standard output streams (`cout` or `cerr`) provided by the standard C++ header but you must guard any `cout` or `cerr` statements with the `HLS_SYNTHESIS` macro. This macro ensures that statements in a component work in x86 emulations (that is, `-march=x86-64`), but are disabled in the component when compiling it to an FPGA architecture (that is, `-march="<FPGA_family_or_part_number>"`). For example:

```
#include "HLS/hls.h"
#include <iostream>

component int debug_component (int a){
#ifdef HLS_SYNTHESIS
```




```

        std::cout << "input value: " << a << std::endl;
    #endif
        return a;
    }

```

If you attempt to use `cout` or `cerr` in a component function without guarding the line in your code with the `HLS_SYNTHESIS` macro, you get an error message similar to the following error when you compile hardware to the FPGA architecture:

```

$ i++ -march="<FPGA_family_or_part_number>" run.cpp
run.cpp:5: Compiler Error: Cannot synthesize std::cout used inside of a
component.
HLS Main Optimizer FAILED.

```

Related Information

[Supported Math Functions](#) on page 154

3.3. Templated and Overloaded Functions

You can use templating and overloading to create generalized function interfaces for your HLS components and HLS tasks. HLS components can be both templated and overloaded. HLS tasks can only be templated. You cannot overload an HLS task function.

Related Information

[Task Functions](#) on page 75

3.3.1. Templated Functions

Using a templated function as an HLS component differs from using the templated function as an HLS task.

Templated Functions as an HLS Component

When you create a template function, you must declare the variant of the function to synthesize into hardware.

For example, a templated `multadd` function might be useful in a system.

```

template <typename T, int MULT>
T multadd (T a, T b) {
    return MULT * (a + b);
}

```

To synthesize a version of this function into a component, you must declare the variant that you want to synthesize:

```

template component int multadd<int, 5>(int a, int b);

```

This declaration combined with the earlier template definition marks the `int` variant with `MULT=5` of the `multadd` function to be generated into a component. This component can now be invoked from the testbench.

Templated Functions as an HLS Task

If you want to use the function as a task in a system of tasks, use the `ihc::launch` and `ihc::collect` calls, and wrap the function that is called in parentheses.

For example, to use the `multadd` function template that was defined earlier as an HLS task, your HLS component code might look like the following code:

```
component void foo () {  
    int a, b;  
    ihc::launch((multadd<int, 5>), a, b);  
    int res = ihc::collect((multadd<int, 5>));  
}
```

If you forget to wrap the template function in parentheses, the Intel HLS Compiler generates error like these:

```
test.cpp:10:7: error: expected '>'  
    ihc::launch(multadd<int, 5>, a, b);  
                ^  
note: expanded from macro 'launch'  
#define launch(x, ...) _launch<decltype(x),x>(__VA_ARGS__)  
                ^  
test.cpp:10:27: error: expected unqualified-id  
    ihc::launch(multadd<int, 5>, a, b);
```

3.3.2. Overloaded Functions

HLS component functions can be overloaded, but HLS task functions cannot because the `ihc::launch` and `ihc::collect` calls cannot distinguish between overloaded variants of a task function.

To overload a component function, define multiple variants of the function.

For example:

```
component int mult (int a, int b) {  
    return a * b;  
}  
  
component float mult (float a, float b) {  
    return a * b;  
}
```

3.3.3. Function Name Mapping

The Intel HLS Compiler always generates unique function names to avoid name collisions that might occur for overloaded and templated functions.

A mapping of the full function declaration to the synthesized function name is provided in the summary page of the high-level design reports (`report.html`). The synthesized function name is used for all the other reports such as the loops report and area analysis.

The following example shows an example of this table in the report:



User-defined Function Name	Mapped Function Name
float add(float, float)	add
int add(int, int)	add_1
bar(int, float)	bar

3.4. Compiler-Defined Preprocessor Macros

The Intel HLS Compiler Pro Edition has a built-in macros that you can use to customize your code to create flow-dependent behaviors.

Table 6. Macro Definition for `__INTELFPGA_COMPILER__`

Tool Invocation	<code>__INTELFPGA_COMPILER__</code>
g++ or cl	Undefined
i++ -march=x86-64	1940
i++ -march="<FPGA_family_or_part_number>"	1940

Table 7. Macro Definition for `HLS_SYNTHESIS`

Tool Invocation	<code>HLS_SYNTHESIS</code>	
	Testbench Code	HLS Component Code
g++ or cl	Undefined	Undefined
i++ -march=x86-64	Undefined	Undefined
i++ -march="<FPGA_family_or_part_number>"	Undefined	Defined

4. Component Interfaces

Intel HLS Compiler generates a component interface for integrating your RTL component into a larger system. A component has two basic interface types: the component invocation interface and the parameter interface.

The *component invocation interface* is common to all HLS components and contains the return data (for nonvoid functions) and handshake signals for passing control to the component, and for receiving control back when the component finishes executing.

The *parameter interface* is the protocol you use to transfer data in and out of your component function. The parameter interface for your component is based on the parameters that you define in your component function signature.

4.1. Component Invocation Interface

For each function that you label as a component, the Intel HLS Compiler creates a corresponding RTL module. This RTL module must have top-level ports, or interfaces, that allow your overall system to interact with your HLS component.

By default, the RTL module for a component includes the following interfaces and data:

- A call interface that consists of `start` and `busy` signals. The call interface is sometimes referred to as the do stream.
- A return interface that consists of `done` and `stall` signals. The return interface is sometimes referred to as the return stream.
- Return data if the component function has a return type that is not `void`

See [Figure 2](#) on page 21 for an example component showing these interfaces.

Your component function parameters generate different RTL depending on their type. For details see the following sections:

- [Scalar Parameters](#) on page 21
- [Pointer and Reference Parameters](#) on page 21

You can also explicitly declare Avalon Streaming interfaces (`stream_in<>` and `stream_out<>` classes) and Avalon Memory-Mapped Master (`mm_master<>` classes) interfaces on component interfaces. For details see the following sections:

- [Avalon Streaming Interfaces](#) on page 22
- [Avalon Memory-Mapped Master Interfaces](#) on page 25

In addition, you can indicate the control signals that correspond to the actions of calling your component by using the component invocation interface arguments. For details, see [Component Invocation Interface Control Attributes](#) on page 37.



4.1.1. Scalar Parameters

Each scalar parameter in your component results in an input conduit that is synchronized with the component `start` and `busy` signals.

The inputs are read into the component when the external system pulls the `start` signal high and the component keeps the `busy` signal low.

For an example of how to specify a scalar parameters and how it is read in by a component, see the `a` argument in [Figure 2](#) on page 21 and [Figure 3](#) on page 22.

4.1.2. Pointer and Reference Parameters

Each pointer or reference parameter of a component results in an input conduit for the address. The input conduit is synchronized with the component `start` and `busy` signals. In addition to this input conduit, all pointers share a single Avalon Memory-Mapped (MM) master interface that the component uses to access system memory.

You can customize these pointer interfaces using the `mm_master<>` class.

Note: Explicitly-declared Avalon Memory-Mapped Master interfaces and Avalon Streaming interfaces are passed by reference.

For details about Avalon (MM) Master interfaces, see [Avalon Memory-Mapped Master Interfaces](#) on page 25.

4.1.3. Interface Definition Example: Component with Both Scalar and Pointer Arguments

The following design example illustrates the interactions between a component's interfaces and signals, and the waveform of the corresponding RTL module.

```
component int dut(int a, int* b, int i) {  
    return a*b[i];  
}
```

Figure 2. Block Diagram of the Interfaces and Signals for the Component `dut`

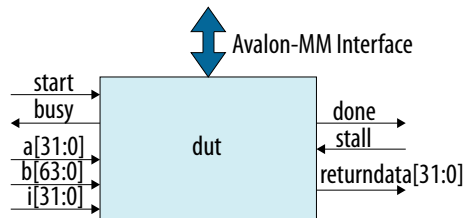
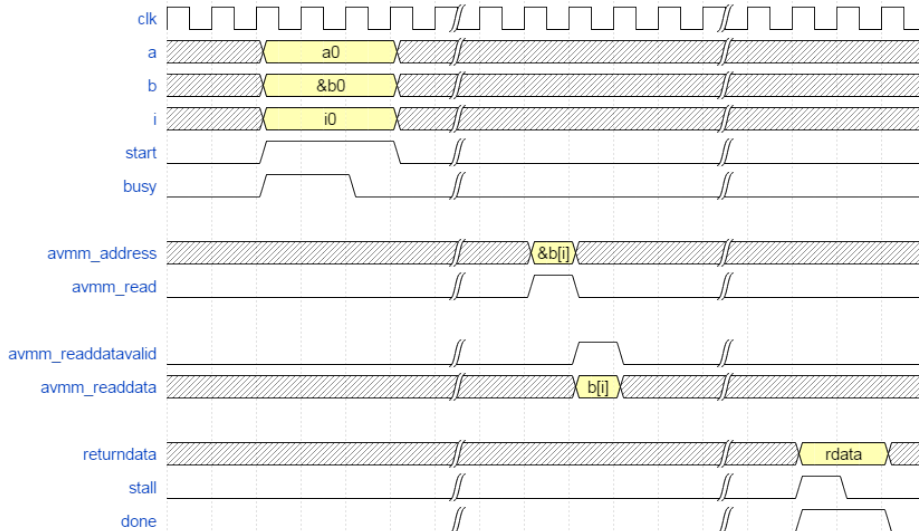


Figure 3. Waveform Diagram of the Signals for the Component dut

This diagram shows that the Avalon-MM read signal reads from a memory interface that has a read latency of one cycle and is non-blocking.



If the `dut` component raises the `busy` signal, the caller needs to keep the `start` signal high and continue asserting the input arguments. Similarly, if the component downstream of `dut` raises the `stall` signal, then `dut` holds the `done` signal high until the `stall` signal is de-asserted.

4.2. Avalon Streaming Interfaces

A component can have input and output streams that conform to the Avalon Streaming (ST) interface specifications. These input and output streams are represented in the C source by passing references to `ihc::stream_in<>` and `ihc::stream_out<>` objects as function arguments to the component.

When you use an Avalon ST interface, you can serialize the data over several clock cycles. That is, one component invocation can read from a stream multiple times.

You cannot derive new classes from the stream classes or encapsulate them in other formats such as structs or arrays. However, you may use references to instances of these classes as references inside other classes, meaning that you can create a class that has a reference to a stream object as a data member.

A component can have multiple read sites for a stream. Similarly, a component can have multiple write sites for a stream. However, try to restrict each stream in your design to a single read site, a single write site, or one of each.

Note: Within the component, there is no guarantee on the order of execution of different streams unless a data dependency exists between streams.

For more information about streaming interfaces, refer to "[Avalon Streaming Interfaces](#)" in *Avalon Interface Specifications*.

Restriction: The Intel HLS Compiler does not support the Avalon ST `channel` or `error` signals.



Streaming Input Interfaces

Table 8. Intel HLS Compiler Pro Edition Streaming Input Interface Template Summary

Template Object or Parameter	Description
<code>ihc::stream_in</code>	Streaming input interface to the component.
<code>ihc::buffer</code>	Specifies the capacity (in words) of the FIFO buffer on the input data that associates with the stream.
<code>ihc::readyLatency</code>	Specifies the number of cycles between when the <code>ready</code> signal is deasserted and when the input stream can no longer accept new inputs.
<code>ihc::bitsPerSymbol</code>	Describes how the data is broken into symbols on the data bus.
<code>ihc::firstSymbolInHighOrderBits</code>	Specifies whether the data symbols in the stream are in big endian order.
<code>ihc::usesPackets</code>	Exposes the <code>startofpacket</code> and <code>endofpacket</code> sideband signals on the stream interface.
<code>ihc::usesEmpty</code>	Exposes the <code>empty</code> out-of-band signal on the stream interface.
<code>ihc::usesValid</code>	Controls whether a <code>valid</code> signal is present on the stream interface.

Table 9. Intel HLS Compiler Pro Edition Streaming Input Interface `stream_in` Function APIs

Function API	Description
<code>T read()</code>	Blocking read call to be used from within the component
<code>T read(bool& sop, bool& eop)</code>	Available only if <code>usesPackets<true></code> is set. Blocking read with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals.
<code>T read(bool& sop, bool& eop, int& empty)</code>	Available only if <code>usesPackets<true></code> and <code>usesEmpty<true></code> are set. Blocking read with out-of-band <code>startofpacket</code> , <code>endofpacket</code> , and <code>empty</code> signals.
<code>T tryRead(bool &success)</code>	Non-blocking read call to be used from within the component. The <code>success</code> bool is set to true if the read was valid. That is, the Avalon-ST <code>valid</code> signal was high when the component tried to read from the stream. The emulation model of <code>tryRead()</code> is not cycle-accurate, so the behavior of <code>tryRead()</code> might differ between emulation and co-simulation.
<code>T tryRead(bool& success, bool& sop, bool& eop)</code>	Available only if <code>usesPackets<true></code> is set. Non-blocking read with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals.
<code>T tryRead(bool& success, bool& sop, bool& eop, int& empty)</code>	Available only if <code>usesPackets<true></code> and <code>usesEmpty<true></code> are set. Non-blocking read with out-of-band <code>startofpacket</code> , <code>endofpacket</code> , and <code>empty</code> signals.
<code>void write(T data)</code>	Blocking write call to be used from the testbench to populate the FIFO to be sent to the component.
<code>void write(T data, bool sop, bool eop)</code>	Available only if <code>usesPackets<true></code> is set.

continued...

Function API	Description
	Blocking write call with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals.
<code>void write(T data, bool sop, bool eop, int empty)</code>	Available only if <code>usesPackets<true></code> and <code>usesEmpty<true></code> are set. Blocking write call with out-of-band <code>startofpacket</code> , <code>endofpacket</code> , and <code>empty</code> signals.

Streaming Output Interfaces

Table 10. Intel HLS Compiler Pro Edition Streaming Output Interface Template Summary

Template Object or Parameter	Description
<code>ihc::stream_out</code>	Streaming output interface from the component.
<code>ihc::readylatency</code>	Specifies the number of cycles between when the <code>ready</code> signal is deasserted and when the input stream can no longer accept new inputs.
<code>ihc::bitsPerSymbol</code>	Describes how the data is broken into symbols on the data bus.
<code>ihc::firstSymbolInHighOrderBits</code>	Specifies whether the data symbols in the stream are in big endian order.
<code>ihc::usesPackets</code>	Exposes the <code>startofpacket</code> and <code>endofpacket</code> sideband signals on the stream interface.
<code>ihc::usesEmpty</code>	Exposes the <code>empty</code> out-of-band signal on the stream interface.
<code>ihc::usesReady</code>	Controls whether a <code>ready</code> signal is present.

Table 11. Intel HLS Compiler Pro Edition Streaming Output Interface `stream_out` Function APIs

Function API	Description
<code>void write(T data)</code>	Blocking write call from the component
<code>void write(T data, bool sop, bool eop)</code>	Available only if <code>usesPackets<true></code> is set. Blocking write with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals.
<code>void write(T data, bool sop, bool eop, int empty)</code>	Available only if <code>usesPackets<true></code> and <code>usesEmpty<true></code> are set. Blocking write with out-of-band <code>startofpacket</code> , <code>endofpacket</code> , and <code>empty</code> signals.
<code>bool tryWrite(T data)</code>	Non-blocking write call from the component. The return value represents whether the write was successful.
<code>bool tryWrite(T data, bool sop, bool eop)</code>	Available only if <code>usesPackets<true></code> is set. Non-blocking write with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals. The return value represents whether the write was successful. That is, the downstream interface was pulling the <code>ready</code> signal high while the HLS component tried to write to the stream.
<code>bool tryWrite(T data, bool sop, bool eop, int empty)</code>	Available only if <code>usesPackets<true></code> and <code>usesEmpty<true></code> are set.

continued...



Function API	Description
	Non-blocking write with out-of-band <code>startofpacket</code> , <code>endofpacket</code> , and <code>empty</code> signals. The return value represents whether the write was successful.
<code>T read()</code>	Blocking read call to be used from the testbench to read back the data from the component
<code>T read(bool &sop, bool &eop)</code>	Available only if <code>usesPackets<true></code> is set. Blocking read call to be used from the testbench to read back the data from the component with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals.
<code>T read(bool &sop, bool &eop, int &empty)</code>	Available only if <code>usesPackets<true></code> and <code>usesEmpty<true></code> are set. Blocking read call to be used from the testbench to read back the data from the component with out-of-band <code>startofpacket</code> , <code>endofpacket</code> , and <code>empty</code> signals.

Related Information

[Avalon Interface Specifications](#)

4.3. Avalon Memory-Mapped Master Interfaces

A component can interface with an external memory over an Avalon Memory-Mapped (MM) Master interface. You can specify the Avalon MM Master interface implicitly using a function pointer argument or reference argument, or explicitly using the `mm_master<>` class defined in the "HLS/hls.h" header file. Describe a customized Avalon MM Master interface in your code by including a reference to an `mm_master<>` object in your component function signature.

Each `mm_master` argument of a component results in an input conduit for the address. That input conduit is associated with the component start and busy signals. In addition to this input conduit, a unique Avalon MM Master interface is created for each address space. Master interfaces that share the same address space are arbitrated on the same interface.

For more information about Avalon MM Master interfaces, refer to "[Avalon Memory-Mapped Interfaces](#)" in *Avalon Interface Specifications*.

Table 12. Intel HLS Compiler Pro Edition Memory-Mapped Interfaces Summary

Template Object or Parameter	Description
<code>ihc::mm_master</code>	The underlying pointer type.
<code>ihc::dwidth</code>	The width of the memory-mapped data bus in bits
<code>ihc::awidth</code>	The width of the memory-mapped address bus in bits.
<code>ihc::aspace</code>	The address space of the interface that associates with the master.
<code>ihc::latency</code>	The guaranteed latency from when a read command exits the component when the external memory returns valid read data.
<code>ihc::maxburst</code>	The maximum number of data transfers that can associate with a read or write transaction.
<code>ihc::align</code>	The alignment of the base pointer address in bytes.
<i>continued...</i>	

Template Object or Parameter	Description
<code>ihc::readwrite_mode</code>	The port direction of the interface.
<code>ihc::waitrequest</code>	Adds the <code>waitrequest</code> signal that is asserted by the slave when it is unable to respond to a read or write request.
<code>getInterfaceAtIndex</code>	This testbench function is used to index into an <code>mm_master</code> object.

Related Information

[Avalon Interface Specifications](#)

4.3.1. Memory-Mapped Master Testbench Constructor

For components that use an instance of the Avalon Memory-Mapped (MM) Master class (`mm_master<>`) to describe their memory interfaces, you must create an `mm_master<>` object in the testbench for each `mm_master` argument.

To create an `mm_master<>` object, add the following constructor in your code:

```
ihc::mm_master<int, ... > mm(void* ptr, int size, bool use_socket=false);
```

where the constructor arguments are as follows:

- `ptr` is the underlying pointer to the memory in the testbench
- `size` is the total size of the buffer in bytes
- `use_socket` is the option you use to override the copying of the memory buffer and have all the memory accesses pass back to the testbench memory

By default, the Intel HLS Compiler copies the memory buffer over to the simulator and then copies it back after the component has run. In some cases, such as pointer-chasing in linked lists, copying the memory buffer back and forth is undesirable. You can override this behavior by setting `use_socket` to `true`.

Note: When you set `use_socket` to `true`, only Avalon MM Master interfaces with 64-bit wide addresses are supported. In addition, setting this option increases the run time of the simulation.

4.3.2. Implicit and Explicit Examples of Describing a Memory Interface

Optimize component code that describes a memory interface by specifying an explicit `mm_master` object.

Implicit Example

The following code example arbitrates the load and store instructions from both pointer dereferences to a single interface on the component's top-level module. This interface will have a data bus width of 64 bits, an address width of 64 bits, and a fixed latency of 1.

```
#include "HLS/hls.h"
component void dut(int *ptr1, int *ptr2) {
    *ptr1 += *ptr2;
    *ptr2 += ptr1[1];
}

int main(void) {
    int x[2] = {0, 1};
```



```
int y = 2;

dut(x, &y);

return 0;
}
```

Explicit Example

This example demonstrates how to optimize the previous code snippet for a specific memory interface using the explicit `mm_master` class. The `mm_master` class has a defined template, and it has the following characteristics:

- Each interface is given a unique ID that infers two independent interfaces and reduces the amount of arbitration within the component.
- The data bus width is larger than the default width of 64 bits.
- The address bus width is smaller than the default width of 64 bits.
- The interfaces have a fixed latency of 2.

By defining these characteristics, you state that your system returns valid read data after exactly two clock cycles and that the interface never stalls for both reads and writes, but the system must be able to provide two different memories. A unique physical Avalon MM master port (as specified by the `aspace` parameter) is expected to correspond to a unique physical memory. If you connect multiple Avalon MM Master interfaces with different physical Avalon MM master ports to the same physical memory, the Intel HLS Compiler cannot ensure functional correctness for any memory dependencies.

```
#include "HLS/hls.h"

typedef ihc::mm_master<int, ihc::dwidth<256>,
                      ihc::awidth<32>,
                      ihc::aspace<1>,
                      ihc::latency<2> > Master1;
typedef ihc::mm_master<int, ihc::dwidth<256>,
                      ihc::awidth<32>,
                      ihc::aspace<4>,
                      ihc::latency<2> > Master2;

component void dut(Master1 &mm1, Master2 &mm2) {
    *mm1 += *mm2;
    *mm2 += mm1[1];
}

int main(void) {
    int x[2] = {0, 1};
    int y = 2;

    Master1 mm_x(x, 2*sizeof(int), false);
    Master2 mm_y(&y, sizeof(int), false);

    dut(mm_x, mm_y);

    return 0;
}
```

4.3.3. Avalon Memory-Mapped Master Interfaces and Load-Store Units

When your component uses one or more Avalon Memory-Mapped (MM) Master interfaces, the Intel HLS Compiler inserts load-store units (LSUs) between the interface and the rest of your component. The type of LSU inserted depends on the inferred memory access pattern and other memory attributes.

The Intel HLS Compiler also tries to minimize the number of LSUs created by coalescing multiple load/store operations into wider load/store operations. Multiple LSUs can share a memory interface.

Typically, the Intel HLS Compiler creates burst-coalesced LSUs for variable-latency MM Master interfaces and pipelined LSUs for fixed-latency MM Master interfaces.

For details about the types of the LSUs and when the Intel HLS Compiler typically instantiates them, see [Load-Store Unit Types](#) on page 28 and [Memory-Access Coalescing and Load-Store Units](#) on page 32.

If your design contains one or more variable-latency Avalon MM Master interfaces (for example, if you interface with off-chip memory), you can control the LSU type to improve the performance and resource utilization of your design.

Use the high-level design reports to determine what types of LSUs your component has, and then you can apply these LSU controls as needed to achieve the component performance that you want.

Table 13. Intel HLS Compiler Pro Edition Load-Store Unit Control Summary

Template Object/Parameter/Function	Description
<code>ihc::lsu</code>	The underlying LSU class template object
<code>ihc::style</code>	Specifies the type of load-store unit.
<code>ihc::static_coalescing</code>	Explicitly allows or prevents static coalescing of a load/store operation with other load/store operations.
<code>load</code>	Loads data from memory into the LSU.
<code>store</code>	Stores data from the LSU into memory.

Related Information

[Control LSUs For Your Variable-Latency MM Master Interfaces](#)

4.3.3.1. Load-Store Unit Types

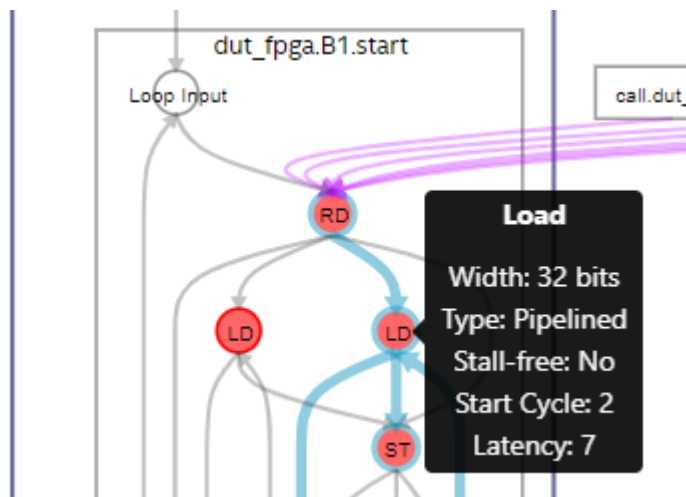
The Intel HLS Compiler determines the types of load-store units (LSUs) to instantiate and whether to coalesce memory accesses based on from the memory access pattern that the compiler infers.

The Intel HLS Compiler instantiates the following the types of LSUs:

- Burst-coalesced LSUs** The Intel HLS Compiler typically instantiates burst-coalesced LSUs for accessing variable-latency Avalon MM Master interfaces.
- Nonaligned burst-coalesced LSUs**
- Pipelined LSUs** The Intel HLS Compiler typically instantiates pipelined LSUs for accessing fixed-latency Avalon MM Master interfaces.
- Never-stall pipelined LSUs**

Click LSUs in the Graph Viewer (in the High-Level Design Reports) to see which types of LSU the compiler instantiated for your component.

Figure 4. Example of LSU Information Provided in the Graph Viewer



Burst-Coalesced Load-Store Units

A burst-coalesced LSU buffers contiguous memory requests until the largest possible burst can be made. For noncontiguous memory requests, a burst-coalesced LSU flushes the buffer between requests.

While a burst-coalesced LSU provides efficient, variable-latency access to memories outside of your component, a burst-coalesced LSU requires a considerable amount of FPGA resources.

The following code example results in the Intel HLS Compiler instantiating two burst-coalesced LSUs:

```
#include "HLS/hls.h"

component void
burst_coalesced(ihc::mm_master<int, ihc::dwidth<64>, ihc::awidth<32>,
ihc::aspace<1>, ihc::latency<0>> &in,
ihc::mm_master<int, ihc::dwidth<64>, ihc::awidth<32>,
ihc::aspace<2>, ihc::latency<0>> &out,
int i) {
int value = in[i / 2]; // Burst-coalesced LSU
out[i] = value; // Burst-coalesced LSU
}
```

Depending on the memory access pattern and other attributes, the compiler might modify a burst-coalesced LSU to be a nonaligned burst-coalesced LSU.

Nonaligned Burst-coalesced LSUs

When a burst-coalesced LSU can access a memory that is not aligned to the external memory word size, the Intel HLS Compiler creates a nonaligned burst-coalesced LSU. Nonaligned LSUs typically require more FPGA resources to implement than aligned LSUs. The throughput of a nonaligned LSU might be reduced if it receives many unaligned requests.

The following code example results in a nonaligned burst-coalesced LSU:

```
#include "HLS/hls.h"

struct State {
    int x;
    int y;
    int z;
};

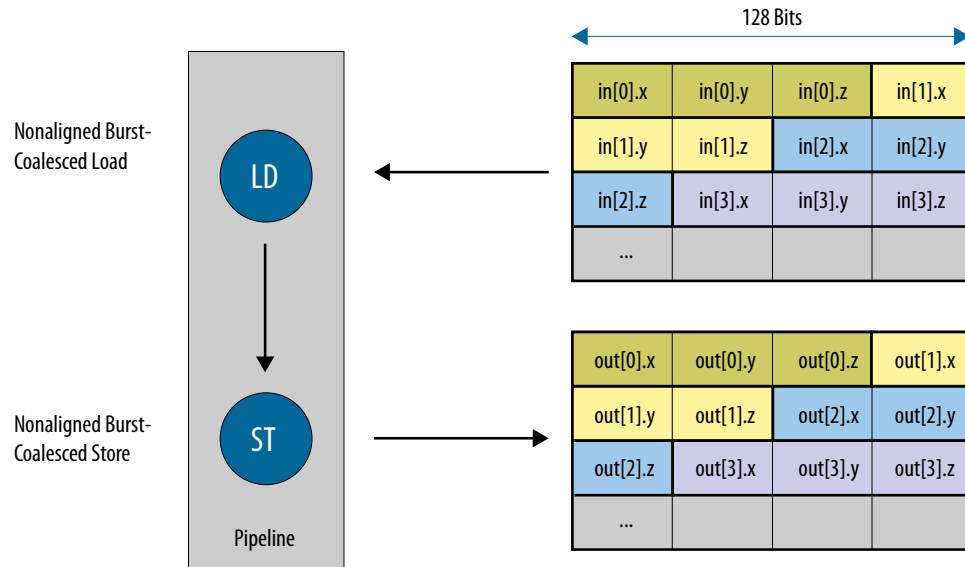
component void
static_coalescing(ihc::mm_master<State, ihc::dwidth<128>, ihc::awidth<32>,
                 ihc::aspace<1>, ihc::latency<0>> &in,
                 ihc::mm_master<State, ihc::dwidth<128>, ihc::awidth<32>,
                 ihc::aspace<2>, ihc::latency<0>> &out,
                 int i) {
    out[i] = in[i]; // Two Nonaligned Burst-coalesced LSUs
}
```

The figure that follows (Figure 5 on page 30) shows the external memory contents for the previous code example and the nonaligned burst-coalesced LSUs in the component pipeline.

The data type that is read and written is a 96-bit-wide struct. The external memory width is 128 bits. This difference between the read/write data width and the external memory width forces some of the memory requests to span two consecutive memory words.

A nonaligned burst-coalesced LSU can detect that discrepancy and serve such memory requests as needed while still buffering contiguous requests until the largest possible burst can be made.

Figure 5. Nonaligned Memory Accesses



Pipelined Load-Store Units

In a pipelined LSU, requests are submitted when they are received and no buffering occurs. Pipelined LSUs are also used for accessing local memories inside your component.



You can tell the compiler to instantiate pipelined LSUs for variable-latency MM Master interfaces. However, variable-latency interface access with pipelined LSUs might reduce throughput.

Memory accesses are pipelined, so multiple requests can be in flight at the same time. If there is no arbitration between the LSU and the memory interfaces, and the interface is fixed latency, a never-stall pipelined LSU is created.

The following code example results in the Intel HLS Compiler instantiating four pipelined LSUs:

```
#include "HLS/hls.h"

component void
pipelined(ihc::mm_master<int, ihc::dwidth<64>, ihc::awidth<32>,
          ihc::aspace<1>, ihc::latency<2>> &in,
          ihc::mm_master<int, ihc::dwidth<64>, ihc::awidth<32>,
          ihc::aspace<1>, ihc::latency<2>> &out,
          int gi, int li) {
    int lmem[1024];

    int res = in[gi]; // Pipelined LSU
    for (int i = 0; i < 4; i++) {
        lmem[li - i] = res; // Pipelined LSU
        res >>= 1;
    }

    res = 0;
    for (int i = 0; i < 4; i++) {
        res ^= lmem[li - i]; // Pipelined LSU
    }

    out[gi] = res; // Pipelined LSU
}
```

Never-Stall Pipelined LSUs

If a pipelined LSU is connected to a memory inside the component or to a fixed-latency MM Master interface without arbitration, a never-stall LSU is created because all accesses to the memory take a fixed number of cycles that are known to the compiler.

The following code example results in the Intel HLS Compiler instantiating three never-stall pipelined LSUs for accessing array lmem.

```
#include "HLS/hls.h"

component void
neverstall(ihc::mm_master<int, ihc::dwidth<128>, ihc::awidth<32>,
           ihc::aspace<1>, ihc::latency<0>> &in,
           ihc::mm_master<int, ihc::dwidth<128>, ihc::awidth<32>,
           ihc::aspace<1>, ihc::latency<0>> &out,
           int gi, int li) {
    int lmem[1024];
    for (int i = 0; i < 1024; i++)
        lmem[i] = in[i]; // Pipelined never-stall LSU

    out[gi] = lmem[li] ^ lmem[li + 1]; // Pipelined never-stall LSU
}
```

4.3.3.2. Memory-Access Coalescing and Load-Store Units

The Intel HLS Compiler sometimes coalesces multiple memory accesses into a wider memory access to save on the number of LSUs instantiated.

When the compiler coalesces the memory accesses, it is referred to as static coalescing because the coalescing occurs at compile time. This static coalescing contrasts with the dynamic coalescing done by a burst-coalesced LSU.

The compiler typically attempts to static coalescing when it detects multiple memory operations that access consecutive locations in memory. This coalescing is usually beneficial because it reduces the number of LSUs that compete for a shared memory interface.

The compiler coalesces memory accesses only up to the width of the memory interface that is being accessed. For an external memory interface, the maximum width is predetermined by the properties of the external memory that you are accessing. For a component (internal) memory interface, the maximum width can be set by the compiler based on the memory geometry that the compiler creates. For more details about component memories, see [Component Memories \(Memory Attributes\)](#) on page 41.

For the following code example, the Intel HLS Compiler statically coalesces the four 4-byte-wide load operations into one 16-byte-wide load operations. A similar coalescing is done for the the four store operations. Coalescing the load and store operations reduces the number of required accesses to the Avalon MM Master interfaces by 4.

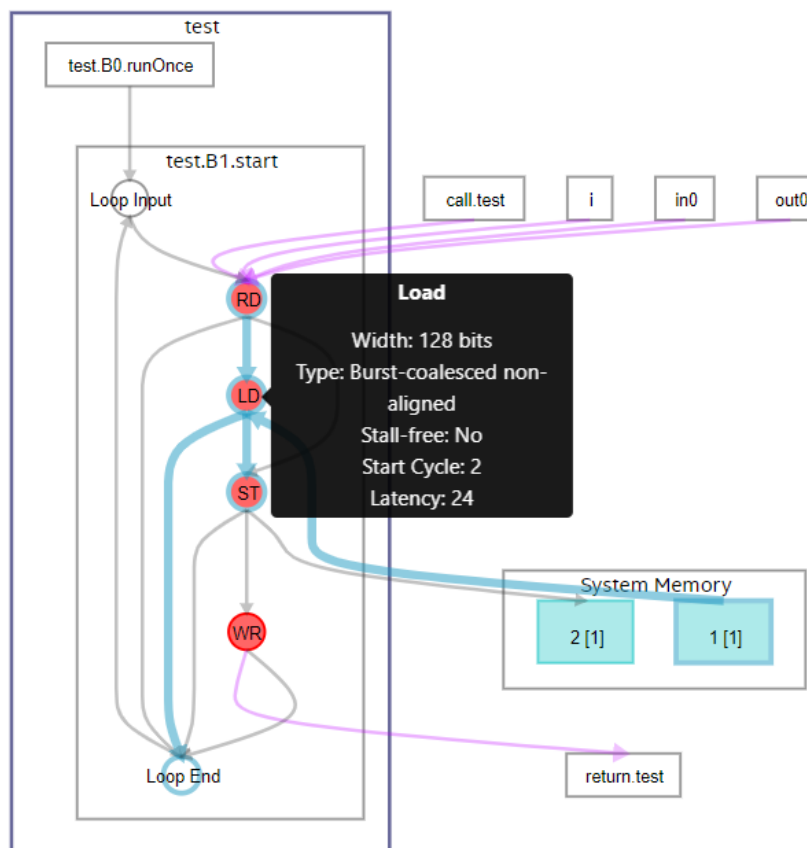
```
#include "HLS/hls.h"

component void
static_coalescing(ihc::mm_master<int, ihc::dwidth<128>, ihc::awidth<32>,
                  ihc::aspace<1>, ihc::latency<0>> &in,
                  ihc::mm_master<int, ihc::dwidth<128>, ihc::awidth<32>,
                  ihc::aspace<2>, ihc::latency<0>> &out,
                  int i) {
    // Four loads statically coalesced into one 16 bytes wide load
    int a1 = in[3 * i + 0];
    int a2 = in[3 * i + 1];
    int a3 = in[3 * i + 2];
    int a4 = in[3 * i + 3];

    // Four stores statically coalesced into one 16 bytes wide store
    out[3 * i + 0] = a4;
    out[3 * i + 1] = a3;
    out[3 * i + 2] = a2;
    out[3 * i + 3] = a1;
}
```

The Graph Viewer in the High-Level Design Reports for this example show that the design only has one load and one store, each of width 128 bit.

Figure 6. Graph Viewer Showing Coalesced Memory Accesses



4.4. Slave Interfaces

The Intel HLS Compiler can implement two different types of slave interface for you component: a control-and-status register (CSR) slave interface and a slave memory interface. In general, use the CSR slave interface to pass scalar values to your component and use the slave memory interface to pass arrays to and from your component.

Slave interfaces are implemented as Avalon Memory Mapped (Avalon MM) Slave interfaces. For details about the Avalon MM Slave interfaces, see "[Avalon Memory-Mapped Interfaces](#) in *Avalon Interface Specifications*.

Table 14. Types of Slave Interfaces

Slave Type	Associated Slave Interface	Read/Write Behavior	Synchronization	Read Latency	Controlling Interface Data Width
Register	The component CSR slave.	The component cannot update these registers from the	Synchronized with the component start signal.	Fixed value of 1.	Always 64 bits

continued...

Slave Type	Associated Slave Interface	Read/Write Behavior	Synchronization	Read Latency	Controlling Interface Data Width
		datapath, so you can read back only data that you wrote in.			
Memory (M20K/MLAB)	Dedicated slave interface on the component.	The component reads from this memory and updates it as it runs. Updates from the component datapath are visible in memory.	Reads and writes to slave memories from outside of the component should occur only when your component is not executing . You might experience undefined component behavior if outside slave memory accesses occur when your component is executing. The undefined behavior can occur even if a slave memory access is to a memory address that the component does not access.	Fixed value that is dependent on the component memory access pattern and any attributes or pragmas that you set. See the Function Viewer report in the High-Level Design Report (<code>report.html</code>) for the read latency of a specific slave memory argument.	The data width is a multiple of the slave data type, where the multiple is determined by coalescing the internal accesses.

4.4.1. Control and Status Register (CSR) Slave

A component can have a maximum of one CSR slave interface, but more than one argument can be mapped into this interface.

Any arguments that are labeled as `hls_avalon_slave_register_argument` are located in this memory space. The resulting memory map is described in the automatically generated header file `<results>.prj/components/<component_name>_csr.h`. This file also provides the C macros for a master component to interact with the slave component. Examples of master components include Nios® II soft processors and Intel Acceleration Stack host applications.

The control and status registers (that is, function call and return) of an `hls_avalon_slave_component` attribute are implemented in this interface.

You do not need to use the `hls_avalon_slave_component` attribute to use the `hls_avalon_slave_register_argument` attribute.

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/interfaces/mm_slaves`

Example code of a component with a CSR slave:

```
#include "HLS/hls.h"

struct MyStruct {
    int f;
    double j;
    short k;
};

hls_avalon_slave_component
component MyStruct mycomp_xyz (hls_avalon_slave_register_argument int y,
    hls_avalon_slave_register_argument MyStruct struct_argument,
    hls_avalon_slave_register_argument unsigned long long mylong,
    hls_avalon_slave_register_argument char char_arg
```



```

    ) {
    return struct_argument;
}

```

Generated C header file for the component mycomp_xyz:

```

/* This header file describes the CSR Slave for the mycomp_xyz component */

#ifndef __MYCOMP_XYZ_CSR_REGS_H__
#define __MYCOMP_XYZ_CSR_REGS_H__

/*****
 * Memory Map Summary
 *****/

/*
Register Address | Access | Register Contents (64-bits) | Description
-----|-----|-----|-----
0x0 | R | {reserved[62:0], busy[0:0]} | Read the busy status of the component
0 - the component is ready to accept a new start
1 - the component cannot accept a new start
0x8 | W | {reserved[62:0], start[0:0]} | Write 1 to signal start to the component
0x10 | R/W | {reserved[62:0], interrupt_enable[0:0]} | 0 - Disable interrupt, 1 - Enable interrupt
0x18 | R/Wclr | {reserved[61:0], done[0:0], interrupt_status[0:0]} | Signals component completion done is read-only and interrupt_status is write 1 to clear
0x20 | R | {returndata[63:0]} | Return data (0 of 3)
0x28 | R | {returndata[127:64]} | Return data (1 of 3)
0x30 | R | {returndata[191:128]} | Return data (2 of 3)
0x38 | R/W | {reserved[31:0], y[31:0]} | Argument y
0x40 | R/W | {struct_argument[63:0]} | Argument struct_argument (0 of 3)
0x48 | R/W | {struct_argument[127:64]} | Argument struct_argument (1 of 3)
0x50 | R/W | {struct_argument[191:128]} | Argument struct_argument (2 of 3)
0x58 | R/W | {mylong[63:0]} | Argument mylong
0x60 | R/W | {reserved[55:0], char_arg[7:0]} | Argument char_arg

NOTE: Writes to reserved bits will be ignored and reads from reserved bits will return undefined values.
*/

/*****
 * Register Address Macros
 *****/

```

```

/* Byte Addresses */
#define MYCOMP_XYZ_CSR_BUSY_REG (0x0)
#define MYCOMP_XYZ_CSR_START_REG (0x8)
#define MYCOMP_XYZ_CSR_INTERRUPT_ENABLE_REG (0x10)
#define MYCOMP_XYZ_CSR_INTERRUPT_STATUS_REG (0x18)
#define MYCOMP_XYZ_CSR_RETURNDATA_0_REG (0x20)
#define MYCOMP_XYZ_CSR_RETURNDATA_1_REG (0x28)
#define MYCOMP_XYZ_CSR_RETURNDATA_2_REG (0x30)
#define MYCOMP_XYZ_CSR_ARG_Y_REG (0x38)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_0_REG (0x40)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_1_REG (0x48)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_2_REG (0x50)
#define MYCOMP_XYZ_CSR_ARG_MYLONG_REG (0x58)
#define MYCOMP_XYZ_CSR_ARG_CHAR_ARG_REG (0x60)

/* Argument Sizes (bytes) */
#define MYCOMP_XYZ_CSR_RETURNDATA_0_SIZE (8)
#define MYCOMP_XYZ_CSR_RETURNDATA_1_SIZE (8)
#define MYCOMP_XYZ_CSR_RETURNDATA_2_SIZE (8)
#define MYCOMP_XYZ_CSR_ARG_Y_SIZE (4)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_0_SIZE (8)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_1_SIZE (8)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_2_SIZE (8)
#define MYCOMP_XYZ_CSR_ARG_MYLONG_SIZE (8)
#define MYCOMP_XYZ_CSR_ARG_CHAR_ARG_SIZE (1)

/* Argument Masks */
#define MYCOMP_XYZ_CSR_RETURNDATA_0_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_RETURNDATA_1_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_RETURNDATA_2_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_ARG_Y_MASK (0xffffffff)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_0_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_1_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_2_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_ARG_MYLONG_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_ARG_CHAR_ARG_MASK (0xff)

/* Status/Control Masks */
#define MYCOMP_XYZ_CSR_BUSY_MASK (1<<0)
#define MYCOMP_XYZ_CSR_BUSY_OFFSET (0)

#define MYCOMP_XYZ_CSR_START_MASK (1<<0)
#define MYCOMP_XYZ_CSR_START_OFFSET (0)

#define MYCOMP_XYZ_CSR_INTERRUPT_ENABLE_MASK (1<<0)
#define MYCOMP_XYZ_CSR_INTERRUPT_ENABLE_OFFSET (0)

#define MYCOMP_XYZ_CSR_INTERRUPT_STATUS_MASK (1<<0)
#define MYCOMP_XYZ_CSR_INTERRUPT_STATUS_OFFSET (0)
#define MYCOMP_XYZ_CSR_DONE_MASK (1<<1)
#define MYCOMP_XYZ_CSR_DONE_OFFSET (1)

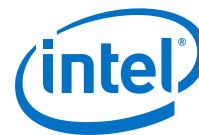
#endif /* __MYCOMP_XYZ_CSR_REGS_H__ */

```

4.4.2. Slave Memories

By default, component functions access parameters that are passed by reference through an Avalon Memory-Mapped (MM) Master interface. An alternative way to pass parameters by reference is to use an Avalon MM Slave interface, which exists inside the component.

Having a pointer argument generate an Avalon MM Master interface on the component has two potential disadvantages:



- The master interface has a single port. If the component has multiple load-store sites, arbitration on that port might create storable logic.
- Depending on the system in which the component is instantiated, other masters might use the memory bus while the component is running and create undesirable stalls on the bus.

Because a slave memory is internal to the component, the HLS compiler can create a memory architecture that is optimized for the access pattern of the component such as creating banked memories or coalescing memories.

Slave memories differ from component memories because they can be accessed from an Avalon MM Master outside of the component. Component memories are by definition restricted to the component and cannot be accessed outside the component.

You can explicitly control the structure of your slave memories by applying memory arguments to slave memory variable declarations.

Important: **Reads and writes to slave memories from outside of the component should occur only when your component is not executing.** You might experience undefined component behavior if outside slave memory accesses occur when your component is executing. The undefined behavior can occur even if a slave memory access is to a memory address that the component does not access.

A component can have many slave memory interfaces. Unlike slave register arguments that are grouped together in the CSR slave interface, each slave memory has a separate interface with separate data buses. The slave memory interface data bus width is determined by the width of the slave type. If the internal accesses to the memory have been coalesced, the slave memory interface data bus width might be a multiple of the width of the slave type.

Component Macro	Description
<code>hls_avalon_slave_memory_argument</code>	Implement the argument, in on-chip memory blocks, which can be read from or written to over a dedicated slave interface.

4.5. Component Invocation Interface Control Attributes

The component invocation interface refers to the control signals that correspond to actions of calling the function. All unstable component argument inputs are synchronized according to this component invocation protocol. A component argument is unstable if it changes while there is live data in the component (that is, between pipelined function invocations).

Table 15. Intel HLS Compiler Component Invocation Interface Control Attribute Summary

Control Attribute	Description
<code>hls_avalon_streaming_component</code>	This is the default component invocation interface.
<i>continued...</i>	

Control Attribute	Description
	The component uses <code>start</code> , <code>busy</code> , <code>stall</code> , and <code>done</code> signals for handshaking.
<code>hls_avalon_slave_component</code>	The <code>start</code> , <code>done</code> , and <code>returndata</code> (if applicable) signals are registered in the component slave memory map.
<code>hls_always_run_component</code>	The <code>start</code> signal is tied to 1 internally in the component. There is no <code>done</code> signal output.
<code>hls_stall_free_return</code>	If the downstream component never stalls, the <code>stall</code> signal is removed by internally setting it to 0.

Related Information

[Control and Status Register \(CSR\) Slave](#) on page 34

4.6. Unstable and Stable Component Parameters

If you do not specify the intended behavior for a parameter, the default behavior of an argument is unstable. An unstable argument can change while there is live data in the component (that is, between pipelined function invocations).

You can declare an interface argument to be stable with the `hls_stable_argument` attribute. A stable interface argument is an argument that does not change while your component executes, but the argument might change between component executions.

You can mark the following the interface arguments as stable:

- Scalar (conduit) arguments
- Pointer interface arguments
 - The address conduit input is stable. The associated Avalon MM Master interface is not affected.
- Pass-by-reference arguments
 - The address conduit input is stable. The associated Avalon MM Master interface is not affected.
- Avalon Memory-Mapped (MM) Master interface arguments
 - The address conduit input is stable. The associated Avalon MM Master interface is not affected.
- Avalon Memory-Mapped (MM) Slave register interface arguments

The following interface arguments cannot be marked as stable:

- Avalon Memory-Mapped (MM) Slave memory interface arguments
- Avalon Streaming interface arguments

You might save some FPGA area in your component design when you declare an interface argument as stable because there is no need to carry the data with the pipeline.

You cannot have two component invocations in flight with different stable arguments between the two component invocations.



Attribute	Description
<code>hls_stable_argument</code>	A stable argument is an argument that does not change while there is live data in the component (that is, between pipelined function invocations).

4.7. Global Variables

Components can use and update C++ global variables. If you access a global variable in your component function, it is implemented as an Avalon Memory-Mapped (MM) Master interfaces, like a pointer parameter.

If you access more than one global variable, each global variable uses the same Avalon MM Master interface, which results in stallable arbitration. If you use pointers and non-constant global memory accesses, then the pointers and global memory accesses all share the same Avalon MM Master interface.

In addition to the Avalon MM Master interface, each global variable that the component uses has an input conduit that must be supplied with the address of the global variable in system memory. The input conduit arguments that are generated in the RTL are named `@<global variable name>`. Input conduits generated for pointer arguments omit the `@` are named for the corresponding pointer argument.

If your global variable is declared as `const`, then no Avalon MM Master interface and no additional input conduit is generated. Therefore, global variables declared as `const` use significantly less FPGA area than modifiable global variable.

4.8. Structs in Component Interfaces

Review the `interface_structs.sv` file in your `<a.prj>/components/<component_name>` folder to see information about the padding and packed-ness of the implementation interfaces for the structs in your component.

The `interface_structs.sv` file contains the Verilog-style definitions of the structs found on your component interface.

4.9. Reset Behavior

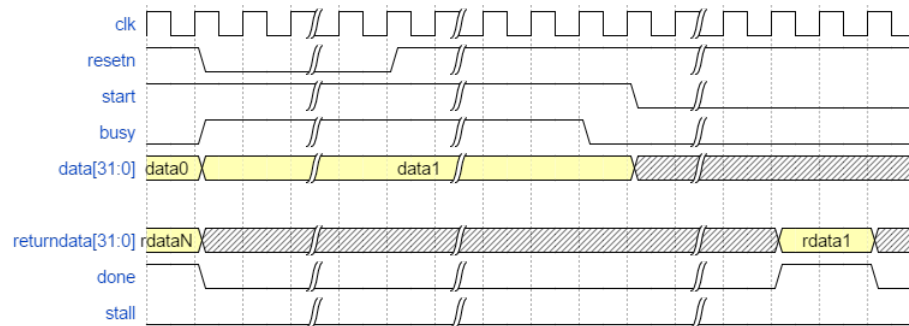
For your HLS component, the reset assertion can be asynchronous but the reset deassertion must be synchronous.

The reset assertion and deassertion behavior can be generated from an asynchronous reset input by using a reset synchronizer, as described in the following example Verilog code:

```
reg [2:0] sync_resetrn;
always @(posedge clock or negedge resetrn) begin
    if (!resetrn) begin
        sync_resetrn <= 3'b0;
    end else begin
        sync_resetrn <= {sync_resetrn[1:0], 1'b1};
    end
end
```

This synchronizer code is used in the example Intel Quartus Prime Pro Edition project that is generated for your components included in an i++ compile.

When the reset is asserted, the component holds its `busy` signal high and its `done` signal low. After the reset is deasserted, the component holds its `busy` signal high until the component is ready to accept the next invocation. All component interfaces (slaves, masters, and streams) are valid only after the component `busy` signal is low.



Simulation Component Reset

You can check the reset behavior of your component during simulation by using the `ihc_hls_sim_reset` API. This API returns 1 if the reset was exercised (that is, if the reset is called during hardware simulation of the component). Otherwise, the API returns 0.

Call the API as follows:

```
int ihc_hls_sim_reset(void);
```

During x86 emulation of your component, the `ihc_hls_sim_reset` API always returns 0. You cannot reset a component during x86 emulation.

5. Component Memories (Memory Attributes)

The Intel High Level Synthesis (HLS) Compiler builds a hardware memory system using FPGA memory resources (such as block RAMs) for any local, constant, static variable or array, and slave memory declared in your code.

Memory accesses are mapped to load-store units (LSUs), which transact with the hardware memory through its ports. The Intel HLS Compiler sometimes statically coalesces multiple memory accesses to a component memory into one wider memory access in order to save on the number of LSUs instantiated. LSUs for component memory are always pipelined LSUs.

If two or more LSUs need to be scheduled during the same cycle, the compiler might create stallable arbitration logic. Stallable arbitration logic appears red in the Component Viewer (in the High-Level Design Reports).

For more details about LSUs instantiated by the Intel HLS Compiler, see [Load-Store Unit Types](#) on page 28. For details about coalescing memory accesses to save on instantiated LSUs, see [Memory-Access Coalescing and Load-Store Units](#) on page 32.

Figure 7. A Basic Memory Configuration Inferred by the Intel HLS Compiler

The following diagram shows a basic memory configuration:

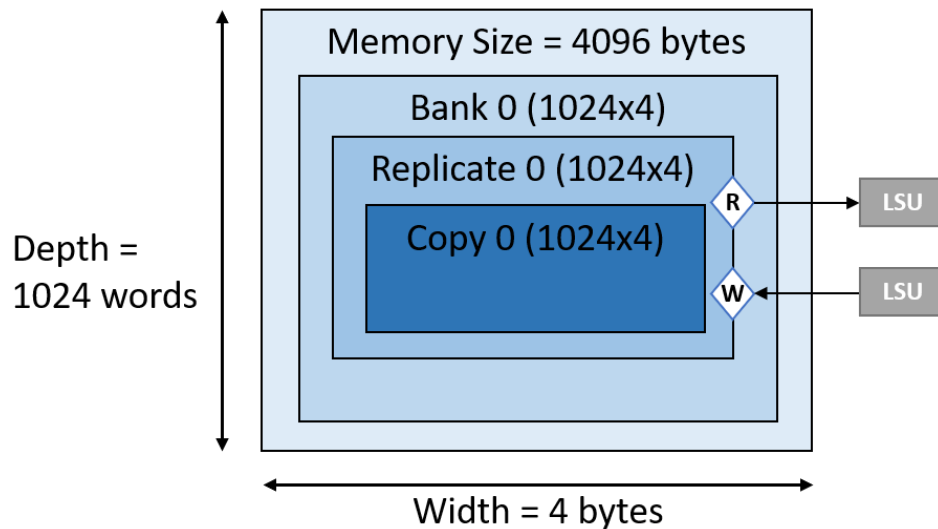


Figure 8. A Memory System With Two Memory Banks

The contents of a memory system can be partitioned into one or more memory banks, such that each bank contains a subset of data contained in the hardware memory:

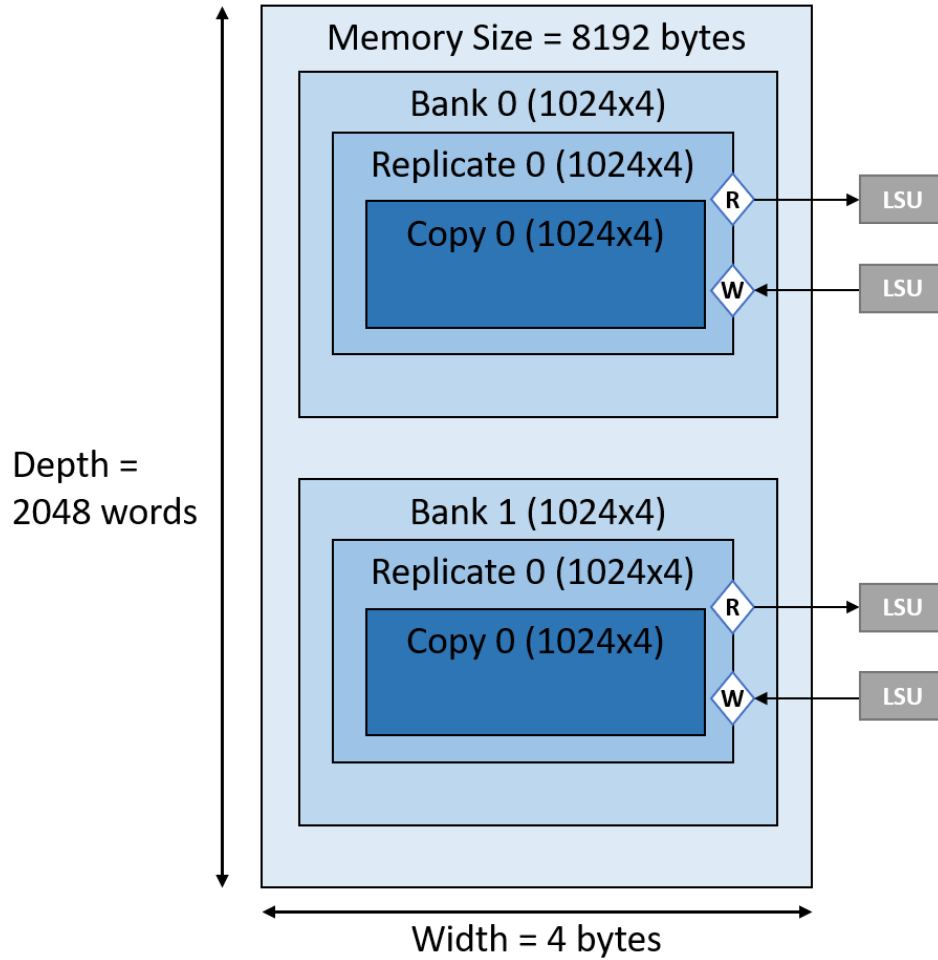
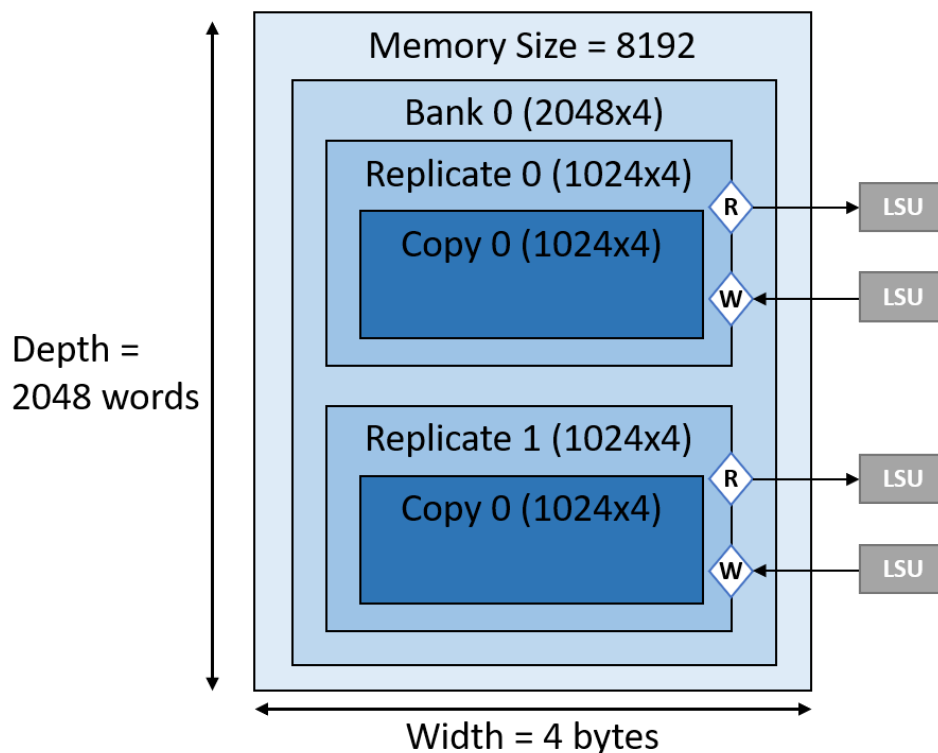


Figure 9. A Memory System With Two Replicates

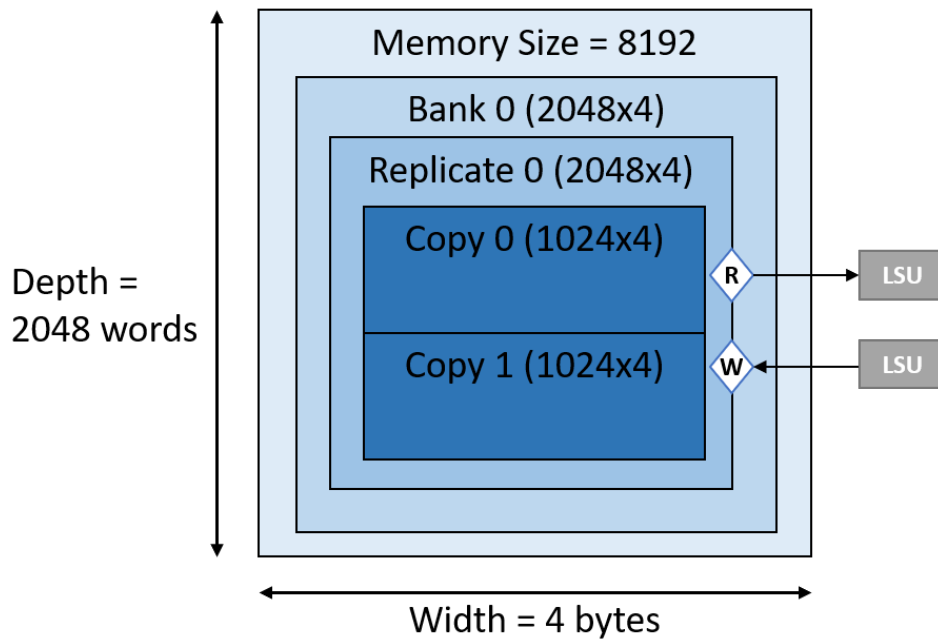
A memory bank can contain one or more memory replicates. The compiler might create memory replicates to create more read ports. Having more read ports allows faster access to your memory system if you have many read operations.

The replicates in a memory bank contain identical data and you can read from the replicates simultaneously. A replicate can have two or four access ports, depending on whether the replicate is clocked at the same frequency (single pumped) or twice the frequency (double pumped) of the component. All ports in replicates can be accessed concurrently. The number of ports in a memory bank depends on the number of replicates that the bank contains.



A replicate can also contain one or more private copies to support multiple concurrent loop iterations.

Figure 10. A Memory System With Two Private Copies



The Intel HLS Compiler can control the geometry and configuration parameters of the hardware memories that it builds. The compiler tries to create stall-free memory accesses. That is, the compiler tries to give memory reads and writes contention-free access to a memory port. A memory system is stall-free if all reads and writes in the memory system are contention-free.

The compiler tries to create a minimum-area stall-free memory system. If you want a different area-performance trade off, use the component memory attributes to specify your own memory system configuration and override the memory system inferred by the compiler.

Component Memory Attributes

Apply the component memory attributes to local, constant, and static variables, or arrays in your component to customize the on-chip memory architecture of the component memory system and lower the FPGA area utilization of your component. You can also apply memory attributes to slave memories and struct data members.

These component memory attributes are defined in the "HLS/hls.h" header file, which you can include in your code.

Table 16. Intel HLS Compiler Pro Edition Component Memory Attributes Summary

Memory Attribute	Description
<code>hls_register</code>	Forces a variable or array to be carried through the pipeline in registers. A register variable can be implemented either exclusively in flip-flops (FFs) or in a mix of FFs and RAM-based FIFOs.
<code>hls_memory</code>	Forces a variable or array to be implemented as embedded memory.

continued...



Memory Attribute	Description
<code>hls_memory_impl</code>	Forces a variable or array to be implemented as embedded memory of a specified type.
<code>hls_singlepump</code>	Specifies that the memory implementing the variable or array must be clocked at the same rate as the component accessing the memory.
<code>hls_doublepump</code>	Specifies that the memory implementing the variable or array must be clocked at twice the rate as the component accessing the memory.
<code>hls_numbanks</code>	Specifies that the memory implementing the variable or array must have a defined number of memory banks.
<code>hls_bankwidth</code>	Specifies that the memory implementing the variable or array must have memory banks of a defined width.
<code>hls_bankbits</code>	Forces the memory system to split into a defined number of memory banks and defines the bits used to select a memory bank.
<code>hls_numports_readonly_writeonly</code>	This memory attribute is deprecated. Use <code>hls_max_replicates</code> instead. Specifies that the memory implementing the variable or array must have a defined number of read and write ports.
<code>hls_simple_dual_port_memory</code>	Specifies that the memory implementing the variable or array should have no port that services both reads and writes.
<code>hls_merge (depthwise)</code>	Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a depth-wise manner.
<code>hls_merge (widthwise)</code>	Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a width-wise manner.
<code>hls_init_on_reset</code>	Forces the static variables inside the component to be initialized when the component <code>reset</code> signal is asserted.
<code>hls_init_on_powerup</code>	Sets the component memory implementing the static variable to initialize on power-up when the FPGA is programmed.
<code>hls_max_concurrency</code>	Specifies the memory has a defined maximum number of private copies to allow concurrent iterations of a loop at any given time.
<code>hls_max_replicates</code>	Specifies that the memory implementing the variable or array has no more than the specified number of replicates to enable simultaneous reads from the datapath

Struct Datatypes and Memory Attributes

You can apply memory attributes to `struct` member variables in the `struct` declaration. If you also apply memory attributes to the object instantiation of a `struct` variable, the attributes on the instantiation override the attributes from the declaration.

The following code example applies memory attributes to both a declaration and an instantiation:

```
struct State {
    int array[100] hls_memory;
    int reg[4] hls_register;
};
component int test(..) {
    struct State S1;
    struct State S2 hls_memory;
    // some uses
}
```

For this example code, the compiler splits `S1` into two variables, `S1.array[100]` (implemented in memory) and `S1.reg[4]` (implemented in registers). However, the compiler ignores the attributes applied at the `struct` declaration for object `S2` because the `S2` object has the `hls_memory` attribute applied at instantiation.

Constraints on Attributes for Memory Banks

The properties of memory banks constrain how you can divide component memory into banks with the memory bank attributes.

The relationship between the following properties is constrained:

- The number of bytes in your array that you want to access at one time (S). If you are accessing a local variable, this value represents the size (in bytes) of the local variable.
- The number of memory banks specified by `hls_numbanks` attribute (N_{banks}).
- The width (in bytes) of the memory banks specified by `hls_bankwidth` attribute (W).
- The number of memory bank-select bits specified by `hls_bankbits` attribute. That is, $n+1$ when you specify b_0, b_1, \dots, b_n as the bank-select bits (N_{bits}).

These attributes are subject to the following constraints:

- $N_{\text{banks}} \times W = S$
The number of bytes accessed concurrently (or size of a local variable) is equal to the number of memory banks it uses times the width of the memory banks.
- N_{banks} must be a power of 2 value.
- $N_{\text{banks}} = 2^{N_{\text{bits}}}$
 N_{bits} bank-selection bits that are required to address N_{banks} number of memory banks.

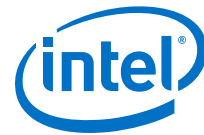
Values that you specify for the `hls_numbanks`, `hls_bankwidth`, and `hls_bankbits` attributes must meet these constraints. For attributes that you do not specify, the Intel HLS Compiler infers values for the attributes following these constraints.

5.1. Static Variables

The HLS compiler supports function-scope static variables with the same semantics as in C and C++.

Function-scope static variables are initialized to the specified values on reset. In addition, changes to these variables are visible across component invocations, making function-scope static variables ideal for storing state in a component.

To initialize static variables, the component requires extra logic, and the component might take some time to exit the reset state while this logic is active.



Static Variable Initialization

Unlike a typical program, you can control when the static variables in your component are initialized, if they are implemented as memories. A static variable can be initialized either when your component is powered up or when your component is reset.

Initializing a static variable when a component is powered up resembles a traditional programming model where you cannot reinitialize the static variable value after the program starts to run.

Initializing a static variable when a component is reset initializes the static variable each time each time your component receives a `reset` signal, including on power up. However, this type of static variable initialization requires extra logic. This extra logic can affect the start-up latency and the FPGA area needed for your component.

You can explicitly set the static variable initialization by adding one of the following attributes to your static variable declaration:

hls_init_on_reset The static variable value is initialized after the component is reset.

Add this attribute to your static variable declaration as shown in the following example:

```
static char arr[128] hls_init_on_reset;
```

This is the default behavior for initializing static variables. You do not need to specify the `hls_init_on_reset` keyword with your static variable declaration to get this behavior.

For example, the static variable in the following example is initialized when the component is reset:

```
static int arr[64];
```

hls_init_on_powerup The static variable is initialized only on power up. This initialization uses a memory initialization file (`.mif`) to initialize the memory, which reduces the resource utilization and start-up latency of the component.

Add this keyword to your static variable declaration as shown in the following example:

```
static char arr[128] hls_init_on_powerup;
```

Some static variables might not be able to take advantage of this initialization because of the complexity of the static variables (for example, an array of structs). In these cases, the compiler returns an error.

For a demonstration of initializing static variables, review the tutorial in `<quartus_installdir>/hls/examples/tutorials/component_memories/static_var_init`.

For information about resetting your component, see [Reset Behavior](#) on page 39.

6. Loops in Components

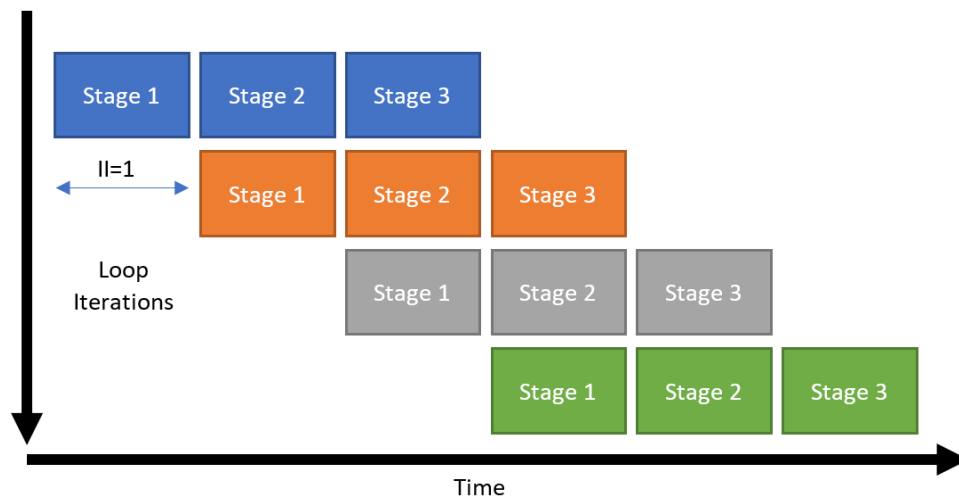
The Intel HLS Compiler Pro Edition attempts to pipeline loops to maximize throughput of the various components that you define.

Loop Pipelining

Pipelining loops enables the Intel HLS Compiler Pro Edition to execute subsequent iterations of a loop in a pipeline-parallel fashion. Pipeline-parallel execution means that multiple iterations of the loop, at different points in their executions, are executing at the same time. Because all stages of the loop are always active, pipelining loops helps maximize usage of the generated hardware.

Figure 11. Pipelined loop with three stages and four iterations

In this figure, one stage is the logic that runs during one clock cycle.



There are some cases where pipelining is not possible at all. In other cases, a new iteration of the loop cannot start until N cycles after the previous iteration.

The number of cycles for which a loop iteration must wait before it can start is called the initiation interval (II) of the loop. This loop pipelining status is captured in the high level design report (`report.html`). In general, an II of 1 is desirable.

A common case where $II > 1$ is when a part of the loop depends in some way on the results of the previous iteration of the same loop. The circuit must wait for these loop-carried dependencies to be resolved before starting a new iteration of the loop. These loop-carried dependencies are indicated in the optimization report.



In the case of nested loops, $II > 1$ for an outer loop is not considered a significant performance limiter if a critical inner loop carries out the majority of the work. One common performance limiter is if the HLS compiler cannot statically compute the trip count of an inner loop (for example, a variable inner loop trip count). Without a known trip count, the compiler cannot pipeline the outer loop.

For more information about loop pipelining, see [Pipeline Loops](#) in *Intel High Level Synthesis Compiler Best Practices Guide*.

Compiler Pragmas Controlling Loop Pipelining

The Intel HLS Compiler has several pragmas that you can specify in your code to control how the compiler pipelines your loops.

Loop pragmas must immediately precede the loop that the pragma applies to. You cannot have a loop pragma before elements such as labels on loops. The following table shows examples of how to apply loop pragmas correctly.

Incorrect (produces a compile-time error)	Correct
<pre>#pragma ivdep TEST_LOOP: for(int idx = 0; idx < counter; idx+ +) {...}</pre>	<pre>TEST_LOOP: #pragma ivdep for(int idx = 0; idx < counter; idx++) {...}</pre>

Table 17. Intel HLS Compiler Pro Edition Loop Pragmas Summary

Pragma	Description
<code>disable_loop_pipelining</code>	Prevents compiler from pipelining a loop,
<code>ii</code>	Forces a loop to have a loop initiation interval (II) of a specified value.
<code>ivdep</code>	Ignores memory dependencies between iterations of this loop.
<code>loop_coalesce</code>	Tries to fuse all loops nested within this loop into a single loop.
<code>max_concurrency</code>	Limits the number of iterations of a loop that can simultaneously execute at any time.
<code>max_interleaving</code>	Controls whether iterations of a pipelined inner loop in a loop nest from one invocation of the inner loop can be interleaved in the component data pipeline with iterations from other invocations of the inner loop.
<code>speculated_iterations</code>	Specifies the number of clock cycles that a loop exit condition can take to compute.
<code>unroll</code>	Unrolls the loop completely or by a number of times.

6.1. Loop Initiation Interval (`ii` Pragma)

The initiation interval, or II, is the number of clock cycles between the launch of successive loop iterations. Use the `ii` pragma to direct the Intel High Level Synthesis (HLS) Compiler to attempt to set the initiation interval (II) for the loop that follows the pragma declaration. If the compiler cannot achieve the specified II for the loop, then the compilation errors out.

You might want to increase the II of a loop to get an f_{MAX} improvement in your component. A loop is a good candidate to have the `ii` pragma applied to increase its loop II if the loop meets any of the following conditions:

- The loop is not critical to the throughput of your component.
- The running time of the loop is small compared to other loops it might contain.

You can also apply the `ii` pragma to force a loop to an II of 1 and accept a possible f_{MAX} penalty.

To specify a loop initiation interval for a loop, specify the pragma before the loop as follows:

```
#pragma ii <desired_initiation_interval>
```

The `<desired_initiation_interval>` parameter is required and is an integer that specifies the number of clock cycles to wait between the beginning of execution of successive loop iterations.

Example

Consider a case where your component has two distinct sequential pipelineable loops: an initialization loop with a low trip count and a processing loop with a high trip count and no loop-carried memory dependencies. In this case, the compiler does not know that the initialization loop has a much smaller impact on the overall throughput of your design. If possible, the compiler attempts to pipeline both loops with an II of 1.

Because the initialization loop has a loop-carried dependence, it will have a feedback path in the generated hardware. To achieve an II with such a feedback path, some clock frequency might be sacrificed. Depending on the feedback path in the main loop, the rest of your design could have run at a higher operating frequency.

If you specify `#pragma ii 2` on the initialization loop, you tell the compiler that it can be less aggressive in optimizing II for this loop. Less aggressive optimization allows the compiler to pipeline the path limiting the f_{max} and could allow your overall component design to achieve a higher f_{max} .

The initialization loop takes longer to run with its new II. However, the decrease in the running time of the long-running loop due to higher f_{max} compensates for the increased length in running time of the initialization loop.

6.2. Loop-Carried Dependencies (`ivdep` Pragma)

When compiling your components, the HLS compiler generates hardware to avoid any data hazards between load and store instructions to component memories, slave memories, and external memories (through Avalon-MM master interfaces). In particular, read-write dependencies can limit performance when they exist across loop iterations because they prevent the compiler from beginning a new loop iteration before the current iteration finishes executing its load and store instructions. You have the option to guarantee to the HLS compiler that there are no implicit memory dependencies across loop iterations in your component by adding the `ivdep` pragma in your code.

The `ivdep` pragma tells the compiler that a memory dependency between loop iterations can be ignored. Ignoring the dependency saves area and lowers the loop initiation interval (II) of the affected loop because the hardware required for avoiding data hazards is no longer required.



You can provide more information about loop dependencies by adding the `safelen(N)` clause to the `ivdep` pragma. The `safelen(N)` clause specifies the maximum number of consecutive loop iterations without loop-carried memory dependencies. For example, `#pragma ivdep safelen(32)` indicates to the compiler that there are a maximum of 32 iterations of the loop before loop-carried dependencies might be introduced. That is, while `#pragma ivdep` promises that there are no implicit memory dependency between any iteration of this loop, `#pragma safelen(32)` promises that the iteration that is 32 iterations away is the closest iteration that could be dependent on this iteration.

To specify that accesses to a particular memory array inside a loop will not cause loop-carried dependencies, add the line `#pragma ivdep array (array_name)` before the loop in your component code. The array specified by the `ivdep` pragma must be one of the following items:

- a component memory array
- a pointer argument
- a pointer variable that points to a component memory
- a reference to an `mm_master` object

If the specified array is a pointer, the `ivdep` pragma also applies to all arrays that may alias with specified pointer. The array specified by the `ivdep` pragma can also be an array or a pointer member of a struct.

Caution: Incorrect usage of the `ivdep` pragma might introduce functional errors in hardware.

Use Case 1:

If all accesses to memory arrays inside a loop do not cause loop-carried dependencies, add `#pragma ivdep` before the loop.

```
1 // no loop-carried dependencies for A and B array accesses
2 #pragma ivdep
3 for(int i = 0; i < N; i++) {
4     A[i] = A[i + N];
5     B[i] = B[i + N];
6 }
```

Use Case 2:

You may specify `#pragma ivdep array (array_name)` on particular memory arrays instead of all array accesses. This pragma is applicable to arrays, pointers, or pointer members of structs. If the specified array is a pointer, the `ivdep` pragma applies to all arrays that may alias with the specified pointer.

```
1 // No loop-carried dependencies for A array accesses
2 // Compiler inserts hardware that reinforces dependency constraints for B
3 #pragma ivdep array(A)
4 for(int i = 0; i < N; i++) {
5     A[i] = A[i - X[i]];
6     B[i] = B[i - Y[i]];
7 }
8
9 // No loop-carried dependencies for array A inside struct
10 #pragma ivdep array(S.A)
11 for(int i = 0; i < N; i++) {
12     S.A[i] = S.A[i - X[i]];
13 }
14
```

```

15 // No loop-carried dependencies for array A inside the struct pointed by S
16 #pragma ivdep array(S->X[2][3].A)
17 for(int i = 0; i < N; i++) {
18     S->X[2][3].A[i] = S.A[i - X[i]];
19 }
20
21 // No loop-carried dependencies for A and B because ptr aliases
22 // with both arrays
23 int *ptr = select ? A : B;
24 #pragma ivdep array(ptr)
25 for(int i = 0; i < N; i++) {
26     A[i] = A[i - X[i]];
27     B[i] = B[i - Y[i]];
28 }
29
30 // No loop-carried dependencies for A because ptr only aliases with A
31 int *ptr = &A[10];
32 #pragma ivdep array(ptr)
33 for(int i = 0; i < N; i++) {
34     A[i] = A[i - X[i]];
35     B[i] = B[i - Y[i]];
36 }

```

6.3. Loop Coalescing (loop_coalesce Pragma)

Use the `loop_coalesce` pragma to direct the Intel HLS Compiler to coalesce nested loops into a single loop without affecting the loop functionality. Coalescing loops can help reduce your component area usage by directing the compiler to reduce the overhead needed for loop control.

Coalescing nested loops also reduces the latency of the component, which could further reduce your component area usage. However, in some cases, coalescing loops might lengthen the critical loop initiation interval path, so coalescing loops might not be suitable for all components.

To coalesce nested loops, specify the pragma as follows:

```
#pragma loop_coalesce <loop_nesting_level>
```

The `<loop_nesting_level>` parameter is optional and is an integer that specifies how many nested loop levels that you want the compiler to attempt to coalesce. If you do not specify the `<loop_nesting_level>` parameter, the compiler attempts to coalesce all of the nested loops.

For example, consider the following set of nested loops:

```

for (A)
  for (B)
    for (C)
      for (D)
        for (E)

```

If you place the pragma before loop (A), then the loop nesting level for these loops is defined as:

- Loop (A) has a loop nesting level of 1.
- Loop (B) has a loop nesting level of 2.
- Loop (C) has a loop nesting level of 3.
- Loop (D) has a loop nesting level of 4.
- Loop (E) has a loop nesting level of 3.



Depending on the loop nesting level that you specify, the compiler attempts to coalesce loops differently:

- If you specify `#pragma loop_coalesce 1` on loop (A), the compiler does not attempt to coalesce any of the nested loops.
- If you specify `#pragma loop_coalesce 2` on loop (A), the compiler attempts to coalesce loops (A) and (B).
- If you specify `#pragma loop_coalesce 3` on loop (A), the compiler attempts to coalesce loops (A), (B), (C), and (E).
- If you specify `#pragma loop_coalesce 4` on loop (A), the compiler attempts to coalesce all of the loops [loop (A) - loop (E)].

Example

The following simple example shows how the compiler coalesces two loops into a single loop.

Consider a simple nested loop written as follows:

```
#pragma loop_coalesce
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    sum[i][j] += i+j;
```

The compiler coalesces the two loops together so that they run as if they were a single loop written as follows:

```
int i = 0;
int j = 0;
while(i < N){

  sum[i][j] += i+j;
  j++;

  if (j == M){
    j = 0;
    i++;
  }
}
```

6.4. Loop Unrolling (unroll Pragma)

The Intel HLS Compiler supports the `unroll` pragma for unrolling multiple copies of a loop.

Example code:

```
1 #pragma unroll <N>
2 for (int i = 0; i < M; ++i) {
3   // Some useful work
4 }
```

In this example, *N* specifies the unroll factor, that is, the number of copies of the loop that the HLS compiler generates. If you do not specify an unroll factor, the HLS compiler unrolls the loop fully. You can find the unroll status of each loop in the high level design report (`report.html`).

6.5. Loop Concurrency (`max_concurrency` Pragma)

You can use the `max_concurrency` pragma to increase or limit the concurrency of a loop in your component. The concurrency of a loop is how many iterations of that loop can be in progress at one time. By default, the Intel HLS Compiler tries to maximize the concurrency of loops so that your component runs at peak throughput.

To achieve maximum concurrency in loops, sometimes private copies of component memory have to be created to break dependencies on the underlying hardware that prevent the loop from being fully pipelined.

You can see the number of private copies created for your component memories in the High Level Design report (`report.html`) for your component:

- In the Details pane of the Loop analysis report as a message that says that the maximum number of simultaneous executions has been limited to N.
- In the Bank view of your component memory in the Function Memory Viewer, where it graphically shows the number of private copies.

Creating private copies of component memory in this case is not the same as replicating memory in order to increase the number of ports.

If you want to exchange some performance for component memory savings, apply `#pragma max_concurrency <N>` to the loop. When you apply this pragma, the number of private copies changes and controls the number of iterations entering the loop, as shown in the following example:

```
#pragma max_concurrency 1
for (int i = 0; i < N; i++) {
    int arr[M];
    // Doing work on arr
}
```

You can control the number of private copies created for a component memory accessed within a loop by using the `hls_max_concurrency` memory attribute. For details, see [hls_max_concurrency Memory Attribute](#).

You can also control the concurrency of your component by using the `hls_max_concurrency` component attribute. For more information about the `hls_max_concurrency(N)` component attribute, see [Concurrency Control \(hls_max_concurrency Attribute\)](#).

6.6. Loop Iteration Speculation (`speculated_iterations` Pragma)

With `speculated_iterations` pragma control, you can adjust the number of speculated iterations for a loop. Speculated iterations are loop iterations that are initiated while the loop exit condition is being calculated. Adjusting the number of speculated iterations can help enable more efficient loop pipelining in your component.

Typically, the exit condition for a loop iteration must be evaluated before it is known whether to start the next loop iteration or continue into the rest of the function. This requirement means that the loop initiation interval (II) cannot be lower than the number of cycles required to compute the exit condition. Speculated iterations can help lower the loop II because operations within the loop can occur in the function pipeline at the same time as the exit condition is evaluated.



For any speculated iteration, instructions with side effects outside of the loop (like writing to memory or a stream) are not completed until the loop exit condition for the iteration has been evaluated. For loop iterations that are in flight but incomplete when the loop exit condition is met, side effect data is discarded.

The Intel HLS Compiler determines the number of speculated iterations on a per-loop basis. You can see the number of speculated iterations for a loop in the Loop Analysis Report in the High Level Design Report (`report.html`).

While speculated iterations can improve loop II, they occupy the pipeline until they are completed. A new loop invocation cannot start until all of the speculated iterations have completed. For example, the next iteration of an outer loop cannot start until all the speculated iterations of an inner loop have completed.

For loops where the exit condition calculation is a bottleneck (as shown in the Loop Analysis Report), consider increasing the number of speculated iterations with the `speculated_iterations` pragma. Increasing the number of speculated iterations might not improve the loop II if other bottlenecks in the loop are found.

For frequently invoked loops with a low latency loop body (for example, an inner loop with a short trip count), you might want to use the `speculated_iterations` pragma to reduce the number of speculated iterations to reduce the overhead of your design. However, setting the number of speculated iterations too low might increase the loop II because there is not enough time to evaluate the exit condition.

The following example shows how you can change the characteristics of a pipelined loop with the `speculated_iterations` pragma.

```
include <HLS/hls.h>

component void unopt_int_cube_root (int *dst, int N) {
    int m = 0;
    // The exit condition which has 2 multiplies and a compare is most critical
    // in loop feedback path. The compiler choice of 4 speculated iterations
    // results in II=2 because the exit condition takes 7 cycles: each
    // multiplication takes 3 cycles and the comparison takes 1 cycle. Four
    // speculated iterations times two-cycle II gives 8 cycles to cover this
    // evaluation.

    while (m*m*m < N) {
        m += 1;
    }
    dst[0] = m;
}

component void opt_int_cube_root (int *dst, int N) {
    int m = 0;
    // Increasing to 7 speculated iterations to cover the 7 cycle exit condition
    // calculation allows us to achieve II=1
    #pragma speculated_iterations 7
    while (m*m*m < N) {
        m += 1;
    }
    dst[0] = m;
}

component void unopt2_int_cube_root (int *dst, int N) {
    int m = 0;
    // by setting to pragma to 0, user can verify that the II has increased to 7
    // which matches the exit condition bottleneck
    #pragma speculated_iterations 0
    while (m*m*m < N) {
        m += 1;
    }
}
```

```

}
dst[0] = m;
}

```

The Loop Analysis Report for these components looks like the following example:

Loops Analysis		<input checked="" type="checkbox"/> Show fully unrolled loops		
	Pipelined	II	Speculated Iterations	Details
Component: unopt_int_cube_root (spec.cpp:6)				
unopt_int_cube_root.B1.start (Component invocation)	No	n/a	n/a	Out-of-order inner loop
unopt_int_cube_root.B2 (spec.cpp:9)	Yes	2	4	Data dependency
Component: opt_int_cube_root (spec.cpp:15)				
opt_int_cube_root.B1.start (Component invocation)	No	n/a	n/a	Out-of-order inner loop
opt_int_cube_root.B2 (spec.cpp:20)	Yes	1	7	
Component: unopt2_int_cube_root (spec.cpp:26)				
unopt2_int_cube_root.B1.start (Component invocation)	No	n/a	n/a	Out-of-order inner loop
unopt2_int_cube_root.B2 (spec.cpp:31)	Yes	7	0	Data dependency

When you click the line with `unopt2_int_cube_root.B2 (spec.cpp:31)` in the Loop Analysis Report, the Details pane shows the following information:

Details

unopt_int_cube_root.B2:

- Compiler failed to schedule this loop with smaller II due to data dependency on variable(s):
 - `m (spec.cpp:7)`
- **Most critical loop feedback path during scheduling:**
 - **3.00 clock cycles 32-bit Integer Multiply Operation (spec.cpp:9)**
 - **3.00 clock cycles 32-bit Integer Multiply Operation (spec.cpp:9)**

6.7. Loop Pipelining Control (`disable_loop_pipelining` Pragma)

When the loop iterations effectively execute sequentially due to loop-carried dependencies, use the `disable_loop_pipelining` pragma to generate a simple sequential datapath and avoid loop resource hardware duplication. The simpler datapath and lack of resource duplication in hardware reduces the FPGA area utilization of your component.

Use the **Loop Analysis** section of the high-level design reports (`report.html`) to help determine if you should apply this pragma to your loops.

In the following example, the Intel HLS Compiler fails to schedule the loop with a small loop initiation interval (II) because of a memory dependency. Pipelining this loop is unlikely to have any benefit to your component throughput or performance.

```

#pragma disable_loop_pipelining
for (int i = 1; i < N; i++) {
    int j = a[i-1];
    // Memory dependency induces a high-latency loop feedback path
    a[i] = foo(j)
}

```

You can also disable pipelining the datapath of your entire component with the `hls_disable_component_pipelining` component attribute. For more information about this attribute, see [Component Pipelining Control \(hls_disable_component_pipelining Attribute\)](#) on page 59.



6.8. Loop Interleaving Control (`max_interleaving` Pragma)

The Intel HLS Compiler Pro Edition tries to maximize the throughput and hardware resource occupancy of pipelined inner loops in a loop nest by issuing new inner loop iterations as frequently as possible (minimizing the loop initiation interval). When the compiler cannot achieve a loop II of 1 for an inner loop, the compiler configures the loop nest to interleave iterations of one invocation of the inner loop with iterations of other invocations of the inner loop.

In cases where this interleaving would not yield a performance benefit, limiting or restricting the amount of interleaving can result in reduced FPGA area utilization.

To limit the number of interleaved invocations of an inner loop that can be executed simultaneously. Annotate the inner loop with the `max_interleaving` pragma. The annotated loop must be contained inside another pipelined loop.

The required parameter (`n`) specifies an upper bound on the degree of interleaving allowed. That is, how many invocations of the containing loop can execute the annotated loop at a given time.

Specify the `max_interleaving` pragma in one of the following ways:

- `#pragma max_interleaving 1`
The compiler restricts the annotated (inner) loop to be invoked only once per outer loop iteration. That is, all iterations of the inner loop travel the pipeline before the next invocation of the inner loop can occur.
- `#pragma max_interleaving 0`
The compiler allows the pipeline to contain a number simultaneous invocations of the inner loop equal to the loop initiation interval (II) of the inner loop. For example, an inner loop with an II of 2 can have iterations from two invocations in the pipeline at a time.

This behavior is the default behavior for the compiler if you do not specify the `max_interleaving` pragma.

In the following code snippet, the compiler restricts the pipelined execution of the `i` loop. A new invocation of the `i` loop corresponds only to subsequent iteration of the `j` loop.

```
// Loop j is pipelined with ii=1
for (int j = 0; j < M; j++) {
    int a[N];
    // Loop i is pipelined with ii=2
    #pragma max_interleaving 1
    for (int i = 1; i < N; i++) {
        a[i] = foo(i)
    }
    ...
}
```

7. Component Concurrency

The Intel HLS Compiler assumes that you want a fully pipelined data path in your component. In the C++ implementation, think of a fully pipelined data path as calling a function multiple times before the first call has returned (see also [Figure 11](#) on page 48 and [Intel HLS Compiler Pipeline Approach](#) on page 13). The behavior of multiple component invocations within the synthesized data path is subject to the concurrency model, so the Intel HLS Compiler might not be able to deliver a component with a component initiation interval (II) of 1, or even any pipelining.

The Intel HLS Compiler provides you with the `hls_max_concurrency` component attribute to help you control the maximum concurrency of your component.

7.1. Serial Equivalence within a Memory Space or I/O

Within a single memory space or I/O (stream read/write, Avalon-MM interface read/write, or component invocation input and return), every invocation of the component (that is, every cycle where the `start` signal is asserted and the component holds the `busy` signal low) on the component invocation interface behaves as though the previous invocation was fully executed.

When visualizing a single shared memory space, think of multiple function calls as executing sequentially, one after another. This way, when the component asserts the `done` signal, the results of a component invocation in hardware are guaranteed to be visible to both the next component invocation and the external system.

The HLS compiler leverages pipeline parallelism to execute component invocations and loop iterations in parallel if the associated dependencies allow for parallel execution. Because the HLS compiler generates hardware that keeps track of dependencies across component invocations, it can support pipeline parallelism while guaranteeing serial equivalence across memory spaces. Ordering between independent I/O instructions is not guaranteed.

7.2. Concurrency Control (`hls_max_concurrency` Attribute)

You can use the `hls_max_concurrency` component attribute to increase or limit the maximum concurrency of your component. The concurrency of a component is the number of invocations of the component that can be in progress at one time. By default, the Intel HLS Compiler tries to maximize concurrency so that the component runs at peak throughput.

You can control the maximum concurrency of your component by adding the `hls_max_concurrency` attribute immediately before you declare your component, as shown in the following example:

```
#include "HLS/hls.h"

hls_max_concurrency(3)
```

Intel Corporation. All rights reserved. Agilix, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.



```
component void foo ( /* arguments */ ){  
    // Component code  
}
```

The Intel HLS Compiler sets the component initiation interval (II) to 1 in the following cases:

- At the component level, the Intel HLS compiler does not automatically create private copies of component memory to increase the throughput. If your component invocation uses a non-static component memory system, the next invocation cannot start until the previous invocation has finished all of its accesses to and from that component memory. This limitation is shown in the Loop analysis report as load-store dependencies on the component memory. Adding the `hls_max_concurrency(N)` attribute to the component creates private copies of the component memory so that you can have multiple invocations of your component in progress at the same time.

For finer-grained control of which component memories to create local copies of, use the `hls_max_concurrency` memory attribute. For details, see [hls_max_concurrency Memory Attribute](#).

- In some cases, the compiler reduces concurrency to save a great deal of area. In these cases, the `hls_max_concurrency(N)` attribute can increase the concurrency from 1.
- This attribute can also accept a value of 0. When this attribute is set to 0, the component should be able to accept new invocations as soon as the downstream datapath frees up. Only use this value when you see loop initiation interval (II) issues (such as extra bubbles) in your component, because using this attribute can increase the component area.

You can also control the concurrency of loops in components with the `max_concurrency(N)` pragma. For more information about the `max_concurrency(N)` pragma, see [Loop Concurrency \(max_concurrency Pragma\)](#) on page 54.

7.3. Component Pipelining Control ([hls_disable_component_pipelining Attribute](#))

If running simultaneous invocations of your component does not improve throughput, or if you do not intend to invoke your component repeatedly, avoid extra FPGA area utilization by using the `hls_disable_component_pipelining` component attribute.

When you specify the `hls_disable_component_pipelining`, the Intel HLS Compiler generates a simpler, sequential datapath for your component.

You apply the attribute as shown in the following example:

```
#include "HLS/hls.h"  
  
hls_disable_component_pipelining  
component void baz ( /* arguments */ ){  
    // component code  
}
```



You can also disable pipelining the datapath of a only a loop in your component with the `disable_loop_pipelining` pragma. For more information about this pragma see [Loop Pipelining Control \(disable_loop_pipelining Pragma\)](#) on page 56.

8. Arbitrary Precision Math Support

The Intel HLS Compiler Pro Edition supports a range of FPGA-optimized arbitrary-precision data types that are defined in header files that you can include in your designs.

Some of these header files are based on the Algorithmic C (AC) data types that Mentor Graphics* provides under the Apache license. For more information about the Algorithmic C data types, refer to *Mentor Graphics Algorithmic C (AC) Datatypes*, which is available as a part of your Intel HLS Compiler installation:

`<quartus_installdir>/hls/include/ref/ac_datatypes_ref.pdf.`

The Intel HLS Compiler also supports arbitrary-precision IEEE 754 compliant floating point data types that is not based on the AC data types.

The Intel HLS Compiler supports the following arbitrary precision data types:

Table 18. Arbitrary Precision Data Types Supported by the Intel HLS Compiler Pro Edition

Data Type	Intel Header File	Description
ac_int	HLS/ac_int.h	Arbitrary-width integer support To learn more, review the following tutorials: <ul style="list-style-type: none"> <code><quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_int_basic_ops</code> <code><quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_int_overflow</code> <code><quartus_installdir>/hls/examples/tutorials/best_practices/struct_interfaces</code>
ac_fixed	HLS/ac_fixed.h	Arbitrary-precision fixed-point number support To learn more, review the tutorial: <code><quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_constructor</code>
	HLS/ac_fixed_math.h	Support for some nonstandard math functions for arbitrary-precision fixed-point data types To learn more, review the tutorial: <code><quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_math_library</code>
ac_complex	HLS/ac_complex.h	Complex number support
hls_float	HLS/hls_float.h	Arbitrary-precision floating-point number support
	HLS/hls_float_math.h	Support for commonly used exponential, logarithmic, power, and trigonometric functions. To learn more, review the following tutorials: <ul style="list-style-type: none"> <code><quartus_installdir>/hls/examples/tutorials/hls_float/1_reduced_doubl</code> <code><quartus_installdir>/hls/examples/tutorials/hls_float/2_explicit_arithmetic</code> <code><quartus_installdir>/hls/examples/tutorials/hls_float/3_conversions</code>

Intel Corporation. All rights reserved. Agilix, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

The Intel HLS Compiler also supports some nonstandard math functions for the following data types when you include an additional header file:

- `ac_fixed` data type
Include the `HLS/ac_fixed_math.h` header file
- `hls_float` data type
Include the `HLS/hls_float_math.h` header file

Advantages of Arbitrary Precision Data Types

The arbitrary precision data types have the following advantages over using standard C/C++ data types in your components:

- You can achieve narrower data paths and processing elements for various operations in the circuit.
- The data types ensure that all operations are carried out in a size guaranteed not to lose any data. However, you can still lose data if you store data into a location where the data type is too narrow.

Limitations of AC Data Types

The AC data types have the following limitations:

- Multipliers are limited to generating 512-bit results.
- Dividers are limited to consuming a maximum of 64 bits.
- The FPGA-optimized header files provided by the Intel HLS Compiler are not compatible with GCC or MSVC. When you use the Intel HLS Compiler header files, you cannot use GCC or MSVC to compile your testbench. Both your component and testbench must be compiled with the Intel HLS Compiler.

To compile AC data types with GCC or MSVC, use the reference AC data types headers also provided with the Intel HLS Compiler. For details, see [AC Data Types and Native Compilers](#) on page 67.

Limitations of the Intel HLS Compiler Arbitrary Precision Floating Point Data Type

The `hls_float` data type has the following limitations:

- FP optimization into constants that was previously done for floats and doubles is not done for `hls_float`.
- A limited set of math functions is supported. For details, see [Operators and Return Types Supported by the `hls_float` Data Type](#) on page 69.
- The `hls_float` header files provided by the Intel HLS Compiler are not compatible with GCC or MSVC. When you use the Intel HLS Compiler header files, you cannot use GCC or MSVC to compile your testbench. Both your component and testbench must be compiled with the Intel HLS Compiler.
- The high-level design reports do not show bit widths for the `hls_float` data type.
- Constant initialization works only with rounding mode RZERO

Related Information

[AC Datatypes at HLSLibs](#)



8.1. Declaring ac_int Data Types

The HLS compiler package includes an `ac_int.h` header file to provide arbitrary precision integer support in your component.

1. Include the `ac_int.h` header file in your component in the following manner:

```
#ifndef __INTELFPGA_COMPILER__
#include "HLS/ac_int.h"
#else
#include "ref/ac_int.h"
#endif
```

2. After you include the header file, declare your `ac_int` variables in one of the following ways:
 - Template-based declaration
 - `ac_int<N, true> var_name; //Signed N bit integer`
 - `ac_int<N, false> var_name; //Unsigned N bit integer`
 - Predefined types up to 63 bits
 - `intN var_name; //Signed N bit integer`
 - `uintN var_name; //Unsigned N bit integer`

Where *N* is the total length of the integer in bits.

Restriction: If you want to initialize an `ac_int` variable to a value larger than 64 bits, you must use the `bit_fill` or `bit_fill_hex` utility function. For details see "2.3.14 Methods to Fill Bits" in *Mentor Graphics Algorithmic C (AC) Datatypes*, which is available as <quartus_installdir>/hls/include/ref/ac_datatypes_ref.pdf.

The following code example shows the use of the `bit_fill` or `bit_fill_hex` utility functions:

```
typedef ac_int<80,false> i80_t;
i80_t x;
x.bit_fill_hex("a9876543210fedcba987"); // member function
x = ac::bit_fill_hex<i80_t>("a9876543210fedcba987"); // global
function
int vec[] = { 0xa987, 0x6543210f, 0xedcba987 };
x.bit_fill(vec); // member function
x = bit_fill<i80_t>(vec); // global function
// inlining the constant array
x.bit_fill( (int [3]) { 0xa987,0x6543210f,0xedcba987 } ); // member
function
x = bit_fill<i80_t>( (int [3]) { 0xa987,0x6543210f,0xedcba987 } ); //
global function
```

For a list of supported operators and their return types, see "Chapter 2: Arbitrary-Length Bit-Accurate Integer and Fixed-Point Datatypes" in *Mentor Graphics Algorithmic C (AC) Datatypes*, which is available in the following file:
<quartus_installdir>/hls/include/ref/ac_datatypes_ref.pdf.

8.1.1. Important Usage Information on the `ac_int` Data Type

The `ac_int` datatype has a large number of API calls that are documented in the `ac_int` documentation included in the Intel HLS Compiler installation package. For more information on AC datatypes, refer to *Mentor Graphics Algorithmic C (AC) Datatypes*, which is available as `<quartus_installdir>/hls/include/ref/ac_datatypes_ref.pdf`.

The `ac_int` datatype automatically increases the size of the result of the operation to guarantee that the intermediate operations never overflow. However, the HLS compiler automatically truncates or extends the result to the size of the specified destination container, so ensure that the storage variable for your computation is large enough.

The HLS compiler installation package includes a number of examples in the tutorials. Refer to the tutorials in `<quartus_installdir>/hls/example/tutorials/ac_datatypes` for some of the recommended practices.

8.2. Integer Promotion and `ac_int` Data Types

The rules of integer promotion when you use `ac_int` data types are different from standard C/C++ rules. Your component design should account for these differing rules.

Depending on the data type of the operands, integer promotion is carried out differently:

- Both operands are standard integer types (`int`, `short`, `long`, `unsigned char`, or `signed char`):
If both operands are of standard integer type (for example `char` or `short`) operations, integers are promoted following the C/C++ standard. That is, the operation is carried out in the data type and size of the largest operand, but at least 32 bits. The expression returns the result in the larger data type.
- Both operands are `ac_int` data types:
If both operands are `ac_int` data types, operations are carried out in the smallest `ac_int` data type needed to contain all values. For example, the multiplication of two 8-bit `ac_int` values is carried out as an 16-bit operation. The expression returns the result in that type.
- One operand is a standard integer type and one operand is an `ac_int` type:
If the expression has one standard data type and one `ac_int` type, the rules for `ac_int` data type promotion apply. The resulting expression type is always an `ac_int` data type. For example, if you add a `short` data type and an `ap_int<16>` data type, the resulting data type is `ac_int<17>`.

In C/C++, literals are by default an `int` data type, so when you use a literal without any casting, the expression type is always at least 32 bits. For example, if you have code like following code snippet, the comparison is carried out in 32 bits:

```
ac_int<5, true> ap;  
...  
if (ap < 4) {  
...  
}
```




If the operands are signed differently and the unsigned type is at least as large as the signed type, the operation is carried out as an unsigned operations. Otherwise, the unsigned operand is converted to a signed operand.

For example, if you have code like the following snippet, the `-1` value expands to a 32-bit negative number (`0xffffffff`) while the `uint3` value is a positive 32-bit number `7` (`0x00000007`):

```
uint3 x = 7;
if (x != -1) {
    // FAIL
}
```

8.3. Debugging Your Use of the `ac_int` Data Type

The "HLS/ac_int.h" header file provides you with tools to help check `ac_int` operations and assignments for overflow in your component when you run an x86 emulation of your component: `DEBUG_AC_INT_WARNING` and `DEBUG_AC_INT_ERROR`.

When you use the `DEBUG_AC_INT_WARNING` and `DEBUG_AC_INT_ERROR` macros, you cannot declare `constexpr ac_int` variables or `constexpr ac_int` arrays.

Table 19. Intel HLS Compiler `ac_int` Debugging Tools Summary

Tool	Description
<code>DEBUG_AC_INT_WARNING</code>	Emits a warning for each detected overflow.
<code>DEBUG_AC_INT_ERROR</code>	Emits a message for the first overflow that is detected and then exits the component with an error.

After you use these tools to determine that your component has overflows, run the `gdb` debugger on your component to run the program again and step through the program to see where the overflows happen.

Review the `ac_int_overflow` tutorial in `<quartus_installdir>/hls/example/tutorials/ac_datatypes` to learn more.

8.4. Declaring `ac_fixed` Data Types

The HLS compiler package includes an `ac_fixed.h` header file for arbitrary precision fixed-point support.

1. Include the `ac_fixed.h` header file in your component in the following manner:

```
#ifndef __INTELFPGA_COMPILER__
#include "HLS/ac_fixed.h"
#else
#include "ref/ac_fixed.h"
#endif
```

2. After you include the header file, declare your `ac_fixed` variables as follows:
 - `ac_fixed<N, I, true, Q, O> var_name; //Signed fixed-point number`

```
— ac_fixed<N, I, false, Q, O> var_name; //Unsigned fixed-point number
```

Where the template attributes are defined as follows:

N The total length of the fixed-point number in bits.

I The number of bits used to represent the integer value of the fixed-point number.

The difference of $N-I$ determines how many bits represent the fractional part of the fixed-point number.

Q The quantization mode that determines how to handle values where the generated precision (number of decimal places) exceeds the number of bits available in the variable to represent the fractional part of the number.

For a list of quantization modes and their descriptions, see "2.1. Quantization and Overflow" in *Mentor Graphics Algorithmic C (AC) Datatypes*, which is available in the following file:

`<quartus_installdir>/hls/include/ref/ac_datatypes_ref.pdf.`

O The overflow mode that determines how to handle values where the generated value has more bits than the number of bits available in the variable.

For a list of overflow modes and their descriptions, see "2.1. Quantization and Overflow" in *Mentor Graphics Algorithmic C (AC) Datatypes*, which is available in the following file:

`<quartus_installdir>/hls/include/ref/ac_datatypes_ref.pdf.`

For a list of supported operators and their return types, see "Chapter 2: Arbitrary-Length Bit-Accurate Integer and Fixed-Point Datatypes" in *Mentor Graphics Algorithmic C (AC) Datatypes*, which is available in the following file:

`<quartus_installdir>/hls/include/ref/ac_datatypes_ref.pdf.`

8.5. Declaring ac_complex Data Types

The HLS compiler package includes an `ac_complex.h` header file for complex number support.

1. Include the `ac_complex.h` header file in your component in the following manner:

```
#ifndef __INTELFPGA_COMPILER__
#include "HLS/ac_complex.h"
#else
#include "ref/ac_complex.h"
#endif
```

2. After you include the header file, declare your `ac_complex` variables according to the data type of your complex number.

The underlying data type can be `ac_int`, `ac_fixed`, `hls_float`, and standard C integer or floating-point data types.



For a list of supported operators and their return types, see "4. Complex Datatype" in *Mentor Graphics Algorithmic C (AC) Datatypes*, which is available in the following file: `<quartus_installdir>/hls/include/ref/ac_datatypes_ref.pdf`.

8.6. AC Data Types and Native Compilers

The reference version of the Mentor Graphics Algorithmic C (AC) data types is also provided with the Intel HLS Compiler. Do not use these reference header files in your component if you want to compile your component with an FPGA target.

Use the reference header files for AC data types to confirm functional correctness in your component when you are compiling your component with native compilers (g++ or MSVC).

If you use the reference header files and compile your component to an FPGA target, your component can compile successfully but your component QoR will be poor.

All of your code must use the same header files (either the reference header files or the FPGA-optimized header files). For example, your code cannot use the reference header files in your testbench and, at the same time, use the FPGA-optimized header file in your component code.

The following reference header files are provided with the Intel HLS Compiler:

AC data type	Reference Header File	Description
ac_int	ref/ac_int.h	Arbitrary width integer support
ac_fixed	ref/ac_fixed.h	Arbitrary precision fixed-point number support
ac_complex	ref/ac_complex.h	Arbitrary precision complex number support

8.7. Declaring hls_float Data Types

The Intel HLS Compiler Pro Edition includes the `hls_float.h` header file for arbitrary-precision floating-point number support. The floating-point representation for `hls_float` data types adopts the same IEEE standard as native C++ `float` and `double` types.

An `hls_float` variable carries an explicit sign bit and an arbitrary number of bits for the exponent and mantissa.

Due to the differences in the internal math implementations and rounding errors, the results from `hls_float` operations might not always be bit-accurate to those produced by C++ native floating-point types with the same exponent and mantissa bit widths. However, these results are validated against the infinitely accurate results.

1. Include the `hls_float.h` header file in your component in the following manner:

```
#include "HLS/hls_float.h"
```

2. After you include the header file, declare your `hls_float` variables as follows:

```
hls_float<exponent_width, mantissa_width[,rounding_mode]>
```

Where the template attributes are defined as follows:



exponent_width, mantissa_width

The bit-width of the exponent and mantissa of the floating-point variable.

The `hls_float` data type supports the following *exponent_width, mantissa_width* combinations:

Table 20. Exponent- and Mantissa-Width Combinations Supported by the `hls_float` Data Type

5, 10	8, 7	8, 10	8, 17	8, 23
8, 26	10, 35	11, 44	11, 52	15, 63

Some of these width combinations map to some commonly used floating-point formats:

Floating-Point Format	<i>exponent_width, mantissa_width</i> Setting
IEEE 754 half-precision (binary16)	5, 10
bfloat16	8, 7
IEEE 754 single-precision (binary32)	8, 23
IEEE 754 double-precision (binary 64)	11, 52
80-bit extended precision ⁽¹⁾	15, 63

rounding_mode

Optional parameter to specify the IEEE 754 rounding mode used when converting between data types.

Set the rounding mode with one of the following values:

- `ihc::fp_config::FP_Round::RNE`
Round to nearest, tie to even
This rounding mode is more accurate (0.5 ULP), but requires more FPGA area.
- `ihc::fp_config::FP_Round::RZERO`
Round towards zero
This rounding mode is less accurate (1 ULP) and requires less FPGA area.

If you do not set this parameter, the Intel HLS Compiler uses the `ihc::FP_Round::RNE` rounding mode.

The `hls_float` data type supports a limited set of math operations. For details, see [Operators and Return Types Supported by the `hls_float` Data Type](#) on page 69.

⁽¹⁾ Not a bit-to-bit mapping.

80-bit extended precision has one explicit bit of fraction that is dropped when converting it to `hls_float<15, 63>`.



8.7.1. Operators and Return Types Supported by the `hls_float` Data Type

The `hls_float` data type supports all overloaded math operators and a limited set of the math functions provided by the Intel HLS Compiler Pro Edition. For some math operators, you can control the precision of the output by using templated versions of the functions.

Important: Due to the differences in the internal math implementations and rounding errors, the results from `hls_float` operations might not always be bit-accurate to those produced by C++ native floating-point types with the same exponent and mantissa bit widths. However, these results are validated against the infinitely accurate results.

Supported Math Functions

In addition to supporting all overloaded math operators, the Intel HLS Compiler supports the following additional math functions for the `hls_float` data type through the `HLS/hls_float_math.h` header file:

- Exponential and logarithmic functions^(*):
 - `ln`, `log2`, `log10`, `ln(1+x)`
 - `ex`, `2x`, `10x`, `ex-1`
- Advanced functions^(*):
 - reciprocal
 - `reciprocal_sqrt`
 - `sqrt(*)`
 - cube root
 - `hypot` (hypotenuse)
- Power functions^(*):
 - `pow`, `powr`, `pown`
- Trigonometric functions^(*):
 - `sin`, `cos`, `sincos`
 - `sinpi`, `cospi`
 - `asin`, `asinpi`, `acos`, `acospi`, `atan`, `atanpi`, `atan2`

Conversion Rules

You can convert between different sizes of `hls_float` data types through assignment or by using the `convert_to()` function. For example,

```
hls_float<8, 32> myFloat = ...;
hls_float<3, 18> myFloat2 = myFloat; // use rounding rules defined by hls_float
type
hls_float <3, 18>myFloat3 = myFloat.convert_to<3, 18,
ihc::fp_config::FP_Round::RZERO>();
// use rounding rules defined in convert_to() function call
```

^(*) Not supported for `hls_float<15, 63>` precision variables.



To convert between native types (for example, `float`, `double`) and `hls_float` data types, assign to or from the types. Type conversion in an assignment occurs according to the rules in the [Table 21](#) on page 70 table that follows.

For two `hls_float` variables in a binary operation, the `hls_float` variable with the larger exponent bit-width is considered to be the "larger" variable. If the two variables have the same exponent bit width, the variable with the larger mantissa bit-width is considered to be the larger variable. The operands are then unified to the "larger" type before the binary operation occurs.

Native floating point data types and `hls_float` data types are converted to `hls_float` data types according to the rules in the [Table 21](#) on page 70 table that follows.

The Intel HLS Compiler also provides some operations that leave the precision of input types untouched and provide control over the output precision. For details, see [Operations With Explicit Precision Controls](#) on page 70.

Table 21. Default Conversion Rules for `hls_float` Variables

Data Type	From <code>hls_float</code> To Data Type	From Data Type To <code>hls_float</code>
<code>hls_float</code> with higher representable range	Keep exponent equivalent. The mantissa is rounded according to the rounding mode of the target <code>hls_float</code> (with the higher representable range).	<code>+-Inf</code> if the source of the conversion is out of the representable range. Otherwise, keep exponent equivalent. The mantissa is rounded according to the rounding mode of the target <code>hls_float</code> (with the smaller representable range).
<code>float</code>	Convert original <code>hls_float</code> to <code>hls_float<8, 23></code> with earlier <code>hls_float</code> rule, then bit-cast to <code>float</code>	Bit-cast <code>float</code> to <code>hls_float<8, 23></code> , and then convert to target <code>hls_float</code> precision using the <code>hls_float</code> to <code>hls_float</code> rules described earlier.
<code>double</code>	Convert original <code>hls_float</code> to <code>hls_float<11, 52></code> with earlier <code>hls_float</code> rule, then bit-cast to <code>double</code>	Bit-cast <code>double</code> to <code>hls_float<11, 52></code> , and then convert to target <code>hls_float</code> precision using the <code>hls_float</code> to <code>hls_float</code> rules described earlier.
<code>long double</code> (emulation only) (Linux only)	Convert original <code>hls_float</code> to <code>hls_float<15, 63></code> with earlier <code>hls_float</code> rule, then insert a 1-bit 1 to the MSB of fraction bits to get an approximate equivalent of 80-bit representation of <code>long double</code>	Drop the explicit 1 fraction bit to convert <code>long double</code> to 79-bit <code>hls_float<15, 63></code>
<code>long double</code> (emulation only) (Windows only)	Same as <code>double</code>	Same as <code>double</code>
C++ native integer types	Truncate towards zero Converting from <code>hls_float</code> that is larger than range of integer type is undefined behavior.	Round to nearest, tie breaks to even. If the integer value is too large, the <code>hls_float</code> value saturates to plus infinity.

Operations With Explicit Precision Controls

The Intel HLS Compiler provides the following operations that leave the precision of input `hls_float`-type variables untouched and let you control the output precision:

Rounding Mode Control For `hls_float` to `hls_float` Conversions



Syntax `convert_to<output_exponent_width,
output_mantissa_width, rounding_mode>`

Description Use this method to override the rounding mode set for an `hls_float` variable when you are converting the variable to different precision.

By default, `hls_float` to `hls_float` conversions use the rounding mode that you specified when you declared the variable.

Multiplication

Syntax `ihc::hls_float< output_exponent_width,
output_mantissa_width > ::mul <accuracy_setting],
[subnormal_setting]> (hls_float_a, hls_float_b)`

Where the optional parameters are defined as follows:

subnormal_setting Optional parameter to specify whether input and output number are flushed to zero when carrying out basic binary operations explicitly.

Set this parameter with one of the following values:

- `ihc::fp_config::FP_Subnormal::ON`
Input and output numbers in the subnormal range are preserved.
The target FPGA device must have subnormal support,
Subnormal support might require more FPGA area.
- `ihc::fp_config::FP_Subnormal::OFF`
Input or output numbers in the subnormal range are flushed to zero.
- `ihc::fp_config::FP_Subnormal::AUTO`
With this setting, the Intel HLS Compiler enables subnormal support only when it is directly supported by the target FPGA device and it does incur any extra FPGA area overhead.

If you do not set this parameter, the Intel HLS Compiler uses the `ihc::FP_Subnormal::AUTO` subnormal setting.

accuracy_setting Optional parameter that influences trade-offs between the accuracy of the result due to different rounding decisions in the intermediary calculations and the FPGA area utilized by the generated hardware. Floating-point operations with less accurate results typically use fewer logic elements.

For example, a divider with a high accuracy might use 20% more FPGA area than divider with low accuracy. The low accuracy divider has a higher error bound [1 unit of least precision (ULP)] than a high accuracy divider (0.5 ULP).

Set this parameter with one of the following values:

- `ihc::fp_config::FP_Accuracy::LOW`
- `ihc::fp_config::FP_Accuracy::HIGH`

If you do not set this parameter, the Intel HLS Compiler uses the

`ihc::fp_config::FP_Accuracy::HIGH` accuracy setting.

Description This math function supplements the basic multiplication operation performed by the multiplication (*) operator.

Multiplies `hls_float_a` and `float_b` without changing the input types, and outputs an `hls_float` at the specified precision.

Addition/Subtraction/Division

Syntax `ihc::hls_float< output_exponent_width, output_mantissa_width > ::add <[optional parameters]> (hls_float_a, hls_float_b)`

`ihc::hls_float< output_exponent_width, output_mantissa_width > ::sub <[optional parameters]> (hls_float_a, hls_float_b)`

`ihc::hls_float< output_exponent_width, output_mantissa_width > ::div <[optional parameters]> (hls_float_a, hls_float_b)`

Description These math functions supplement the basic math operations performed by the addition/subtraction/division (+/ -//) operators.

Adds/Subtracts/Divides `hls_float_a` and `hls_float_b` by first casting `hls_float_a` and `hls_float_b` to the specified `hls_float` precision. The operation and output are at the specified precision.

You can also specify the *optional parameters* that are the *accuracy_setting* and *subnormal_setting* parameters described earlier.

Comparison Operators

Comparison operators (>, <, ==, !=, >=, <=) are subject to the conversion rules described earlier.

The == and != operators impose a bit-wise comparison of the casted values.



Comparisons with NaN always return false.

Additional `hls_float` Functions

The `hls_float` data type also has the following additional functions:

Table 22.

Function	Description
Getters and Setters	
<code>hls_float::get_exponent</code> <code>hls_float::set_exponent</code>	Gets/sets the exponent value of the <code>hls_float</code> variable.
<code>hls_float::get_mantissa</code> <code>hls_float::set_mantissa</code>	Gets/sets the mantissa value of the <code>hls_float</code> variable.
<code>hls_float::get_sign</code> <code>hls_float::set_sign</code>	Gets/sets the sign bit of the <code>hls_float</code> variable.
Special Constants	
<code>hls_float<e,m>::nan()</code>	Constant used to assign the <code>hls_float</code> variable a value of NaN.
<code>hls_float<e,m>::pos_inf()</code>	Constant used to assign the <code>hls_float</code> variable a value of $+\infty$.
<code>hls_float<e,m>::neg_inf()</code>	Constant used to assign the <code>hls_float</code> variable a value of $-\infty$.
Value Queries	
<code>hls_float::is_nan()</code>	Returns <code>true</code> if the value of the <code>hls_float</code> variable is NaN.
<code>hls_float::is_inf()</code>	Returns <code>true</code> if the value of the <code>hls_float</code> variable is $\pm\infty$.
<code>hls_float::is_zero()</code>	Returns <code>true</code> if the value of the <code>hls_float</code> variable is zero.
Special Functions	
<code>hls_float::next_after(next_val)</code>	Returns the next representable value towards <code>next_val</code> .

9. Component Target Frequency

You can specify component target frequency either in the `i++` command by specifying the `--clock` option or by using the `hls_scheduler_target_fmax_mhz` component attribute. The component attribute takes priority over the command option.

For details about the `--clock` option, see [Command Options Affecting Linking](#) on page 106.

For details about the `hls_scheduler_target_fmax_mhz` component attribute, see [hls_scheduler_target_fmax_mhz Component Attribute](#) on page 125.

The two options for setting target frequency are functionally equivalent except their scopes differ:

- The `--clock` option applies to all components compiled with the invocation of the `i++` command that contains the `--clock` option.
- The `hls_scheduler_target_fmax_mhz` component attribute applies only to the component that has the attribute.

To learn more about the attribute and how it interacts with the loop pragma, review the following tutorial:

```
<quartus_installdir>/hls/examples/tutorials/best_practices/
set_component_target_fmax
```

If you use both the `i++` command `--clock` option and the `hls_scheduler_target_fmax_mhz` component attribute, the component attribute takes priority. For example, you can compile the following code with the `i++ ... --clock=300MHz` command:

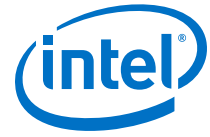
```
component int test1(){
  ...
}

hls_scheduler_target_fmax_mhz(200)
component int test2(){
  ...
}
```

The compiler schedules component `test1` at 300 MHz (from the command option) and component `test2` at 200 MHz (from the component attribute).

Note: Setting the target f_{MAX} determines the pipelining effort at the compilation stage. Compiling with Quartus Prime software reports the achievable f_{MAX} value for your components. This value is often different from the value you specified.

You can lower the `--clock` value to reduce the latency of your design at the expense of reducing the f_{MAX} of your component.



10. Systems of Tasks

Your component design might contain operations that you want to run asynchronously from the main flow of your component. The Intel HLS Compiler Pro Edition lets you define these asynchronous activities in task functions. These task functions, along with the component that invokes them, constitute a system of tasks.

The `component` keyword marks a single function and its subfunctions as a component. Within this component function, directly-called functions are in-lined while functions that use the systems of tasks API calls (`ihc::launch` and `ihc::collect`) generate hardware outside the component datapath and behave like an asynchronous call.

The function tagged with the `component` keyword marks the boundary of a system of tasks. Your external system can interact with all the interfaces that the component exposes.

Implementing your design as a system of tasks instead of a monolithic component can be useful in situations where expressing coarse-grained thread-level parallelism is needed. For example, a system of tasks is useful in the following situations:

- Improving the performance of operations like executing loops in parallel
- Reducing FPGA area utilization by sharing an expensive compute block with different parts of your component

Table 23. Intel HLS Compiler System of Tasks Summary

Function	Description
<code>ihc::launch</code>	Marks a function as an Intel HLS Compiler task for hardware generation, and launches the task function asynchronously.
<code>ihc::collect</code>	Synchronizes the completion of the specified task function in the component.
<code>ihc::stream</code>	Allows streaming communication between different task functions.
<code>ihc::launch_always_run</code>	Launches a task function at component power-on or reset and continuously executes the function.

10.1. Task Functions

The Intel HLS Compiler Pro Edition implements task functions in way similar to HLS component functions, but with some additional constraints.

Scalar Parameters and Return Values

Like HLS components, the scalar parameters and return value for an HLS task are implemented as conduits and the hand-shaking is implemented as a simple `stall/valid` handshake. The `ihc::launch` and `ihc::collect` calls connect directly to the HLS task function `do` and `return` streams.

In the High Level Design Report (`report.html`), the `ihc::launch` and `ihc::collect` calls appear as blocking streaming write and streaming read operations.

Interaction with External Systems

Task functions can use a global instance of the `ihc::stream_in` class to take an input from the external system, or a global instance of the `ihc::stream_out` class to provide output to the external system.

The global `ihc::stream_in` and `ihc::stream_out` streams must be declared outside of any struct variables, and they cannot be declared in arrays.

Communication Between HLS Task Functions

For two task functions to communicate with each other, connect them with a global `ihc::stream` object (instead of the `ihc::stream_in` and `ihc::stream_out` objects).

The global `ihc::stream` object must be declared outside of any struct variables, and it cannot be declared in an array.

The `ihc::stream` object has an API very similar to the `ihc::stream_in` and `ihc::stream_out` classes. However, since these streams always require handshaking, the API does not support the parameters `ihc::usesReady` or `ihc::usesValid`. They do support `tryRead` and `tryWrite` API functions.

The `ihc::stream` objects can have both of their endpoints within the system of tasks. This includes within the same function as well. For an example of using an `ihc::stream` within a single function as a FIFO, see the following tutorial:

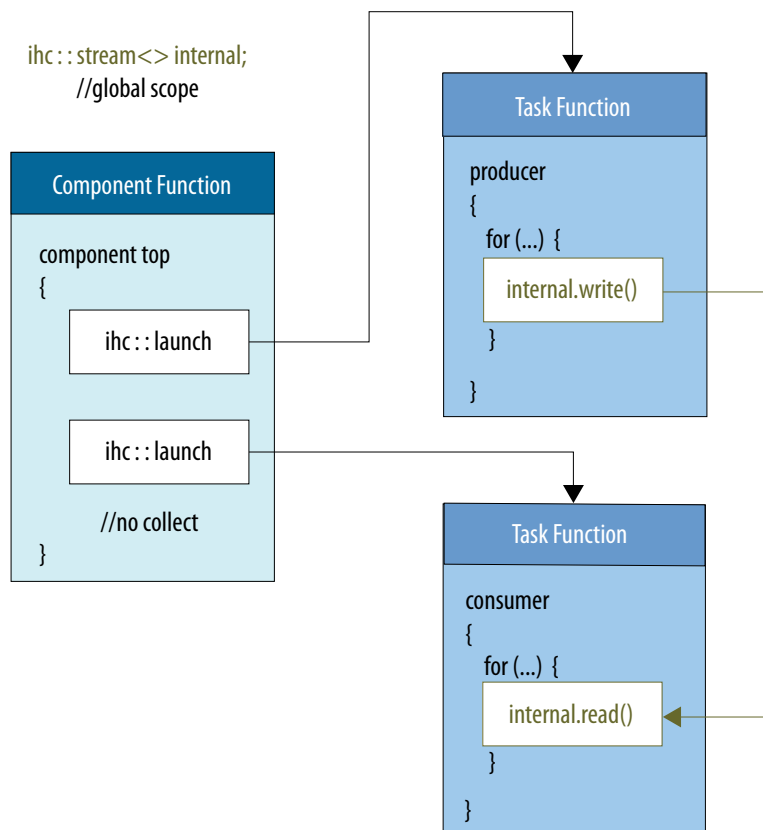
```
<quartus_installdir>/hls/examples/tutorials/system_of_tasks/  
internal_stream
```

If an instance of the `ihc::stream` class has only one endpoint within the system of tasks, it is treated as if it were a `ihc::stream_in` or `ihc::stream_out` class based on its usage within the system, so it can be used interchangeably with `ihc::stream_in` or `ihc::stream_out` (provided that the limitations do not affect the design). An `ihc::stream` object can be used for multiple tasks to communicate with one another. See the following tutorial:

```
<quartus_installdir>/hls/examples/tutorials/system_of_tasks/  
parallel_loops
```

The following diagram shows how you might use the `ihc::stream` object to communicate between task functions:

Figure 12. Example of a Systems of Tasks using Internal Streams



HLS Task Function Restrictions

HLS task functions are subject to the following restrictions:

- Task functions cannot be shared between multiple components.
- All read sites and write sites for a stream must be within the same function (component or task).
- A task function can be launched (with `ihc::launch`) only from one component function or task function. The launching function and the collecting function can be different functions but they must part of the same component system of tasks.
- A task function can be collected (with `ihc::collect`) only from one component or task function. The collecting function and the launching function can be different functions but they must part of the same component system of tasks.
- No guarantee of execution order is provided between independent I/O instructions, even at the task level.

The `ihc::launch` and `ihc::collect` calls to a particular task function are executed in order.

Any stream accesses to that task from the current function are executed in instruction order only with respect to `ihc::launch` and `ihc::collect` calls to the corresponding function.

Figure 13. Example 1 of a Valid `ihc::launch/ihc::collect` Sequence

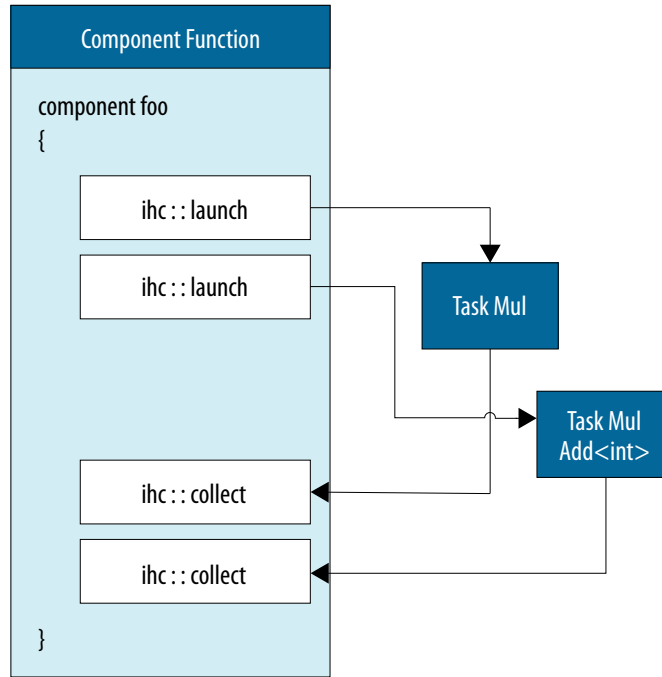


Figure 14. Example 2 of a Valid `ihc::launch/ihc::collect` Sequence

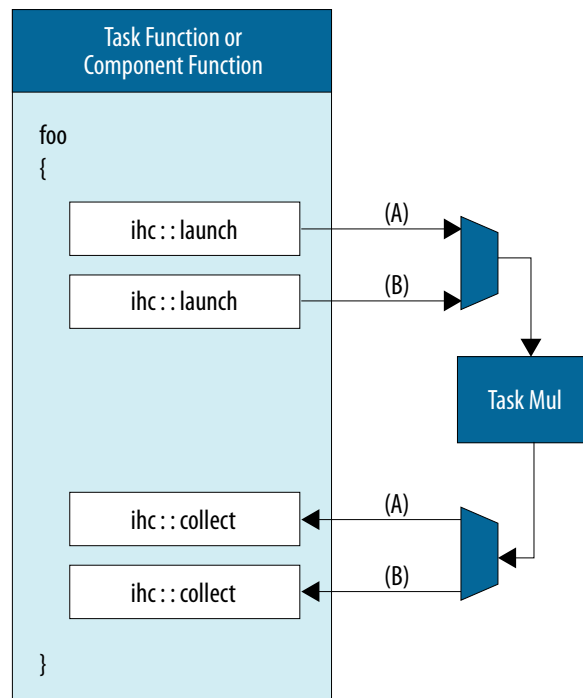


Figure 15. Example 3 of a Valid `ihc::launch/ihc::collect` Sequence

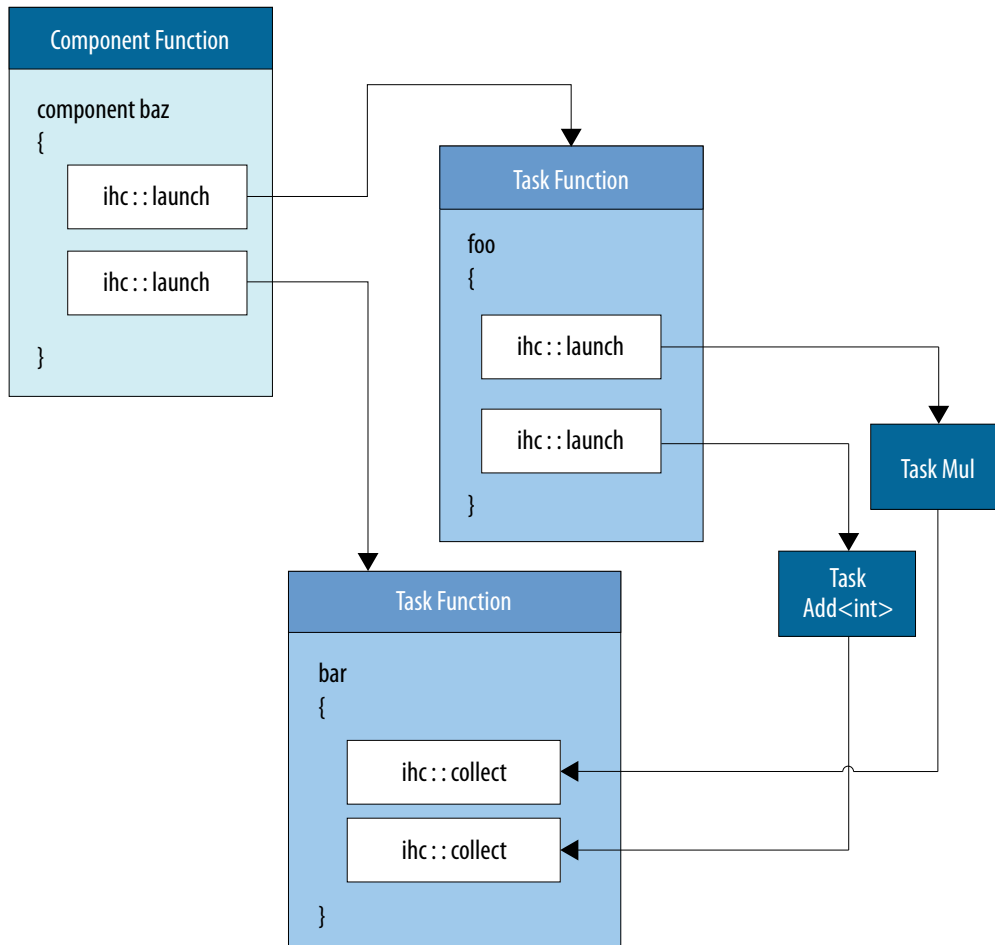
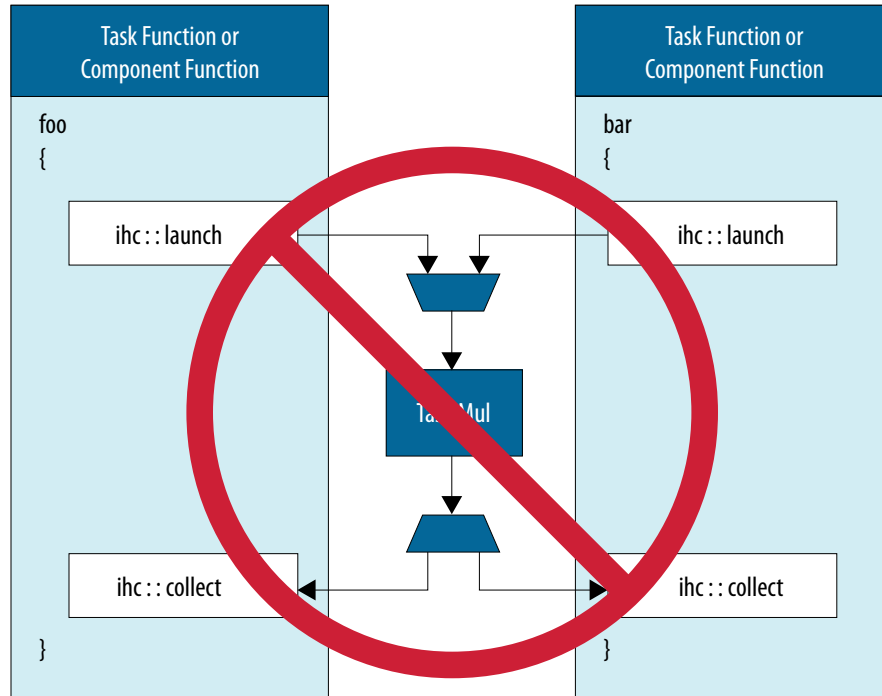


Figure 16. Example of an Invalid `ihc::launch/ihc::collect` Sequence



Task Attributes

You can use the following function-level attributes on an HLS task function:

- `hls_max_concurrency`
- `hls_component_ii`
- `hls_scheduler_target_fmax_mhz`
- `hls_disable_component_pipelining`

In addition to these function attributes, you can use any HLS attributes and pragmas within your HLS task functions. For example, you can use attributes and pragmas like `#pragma ii`, `#pragma ivdep`, `hls_memory`, and `hls_register`.

You cannot use component macros or component invocation interface control attributes when you define HLS task functions. For example, you cannot use `hls_avalon_slave_register_argument`, `hls_conduit_argument`, `hls_stall_free_return`, or `hls_avalon_streaming_component`

10.2. Internal Streams

You can use the HLS `ihc::stream` object as a FIFO in a single task or component.

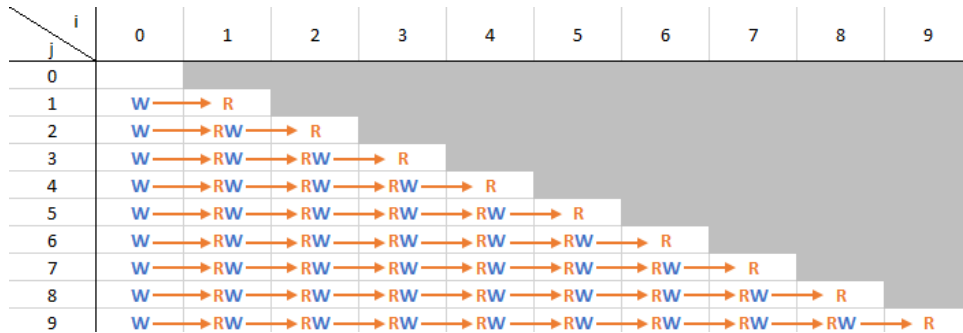
For an example of using the HLS tasks `ihc::stream` object as a FIFO, review the tutorial in `<quartus_installdir>/hls/examples/tutorials/system_of_tasks/internal_stream`.



To help you understand the tutorial better, review the following diagram showing a store-load dependency:

This diagram is simplified from the tutorial. It shows 10 iterations, while the tutorial goes through 32 iterations.

In the diagram, i is the index of the outer loop and j is the index of the inner loop.



Each iteration of the outer loop reads all the values written by the previous loop iteration and writes one less value to the buffer. The internal stream outperforms the array in this design because array must allocate enough space to store written values before the values are read, but an internal stream does not need to allocate this space.

In addition, the trip count of the inner loop decreases by one in each outer loop, so the space claimed by array is never filled after the first iteration, which wastes area.

10.3. System of Tasks Simulation

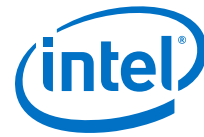
When you simulate a system of tasks design where the completion of a task function is not synchronized with an `ihc::collect` call, use the `ihc_hls_set_component_wait_cycle` testbench API function to allow output from that task function to be returned after the component function finishes running.

If you do not use this function in your testbench, the latency of some task functions might make your simulation output inaccurate.

For an example of a valid systems of task design where the completion of a task function is not synchronized with an `ihc::collect` call, see [Example 3 of a Valid `ihc::launch/ihc::collect` Sequence](#).

Table 24. Intel HLS Compiler Testbench API for System of Tasks

Function	Description
<code>ihc_hls_set_component_wait_cycle</code>	This function tells the simulation process to continue running for a specified number of cycles after the <code>done</code> signal for the specified component is observed.



11. Libraries

With libraries, you can reuse functions without knowing the underlying hardware design or implementation details. Libraries can be created with Intel FPGA high-level design tools including the Intel HLS Compiler and the Intel FPGA SDK for OpenCL*, either from code initially targeting that tool or from RTL code.

The Intel HLS Compiler supports two types of libraries:

- [Object libraries](#)
- [Source code libraries](#)

Object Libraries

An object library is a single platform-specific archive file that contains one or more object files. An object file contains implementations of one or more functions. The object and library files use the same formats as the operating system that you compile your Intel HLS Compiler code on, with additional sections that carry HLS-specific information.

On Linux platforms, an object library is a `.a` archive file that contains `.o` object files. On Windows platforms, a library is a `.lib` archive file that contains `.obj` object files.

An object library includes one or more function signature files that you include in your component source code so that your component can call the functions provided by the library. A function signature file is a C-style header file (`.h`) that declares the signatures of the functions that are provided in an object library.

Object libraries can be created from RTL or high-level source code.

Source Code Libraries

A source code library is a C-style header file that contains a source code library. You include this header file in your component source code, and the header file code is compiled along with your component.

You can use C++ templates to make your source code library more customizable.

The Intel HLS Compiler provides some source code libraries that provide you with FPGA-optimized code for some commonly-used algorithms.

For details about source code libraries included with the Intel HLS Compiler, see the following sections:

- [Arbitrary Precision Math Support](#) on page 61
- [Advanced Math Source Code Libraries](#) on page 151



11.1. Object Libraries

An object library is a single platform-specific archive file that contains one or more object files, each of which contains implementations of one or more functions.

The object and library files use the same formats as the operating system that you compile your Intel HLS Compiler code on, with additional sections that carry additional library information. On Linux platforms, a library is a `.a` archive file that contains `.o` object files. On Windows platforms, a library is a `.lib` archive file that contains `.obj` object files.

You can call the functions in the library from your component without needing to know the hardware design or the implementation details underlying the functions in the library. Add the library to the `i++` command line when you compile your component.

You can create a library from your HLS C++ code source files or register transfer level (RTL) language source files. You can target the library for use with one of the following Intel high-level design products:

- Intel HLS Compiler Pro Edition
- Intel FPGA SDK for OpenCL Pro Edition

To create a library from your HLS code that targets the Intel FPGA SDK for OpenCL, you must have the Intel FPGA SDK for OpenCL Pro Edition installed. The version of the SDK must be same as your version of Intel HLS Compiler.

Creating a library is a two-step process

1. Each object file is generated from input source files with the `fpga_crossgen` command.

The required input source files depend on the type of source code you are creating the object from.

An object is effectively an intermediate representation of your source code with both a CPU representation and an FPGA representation of your code.

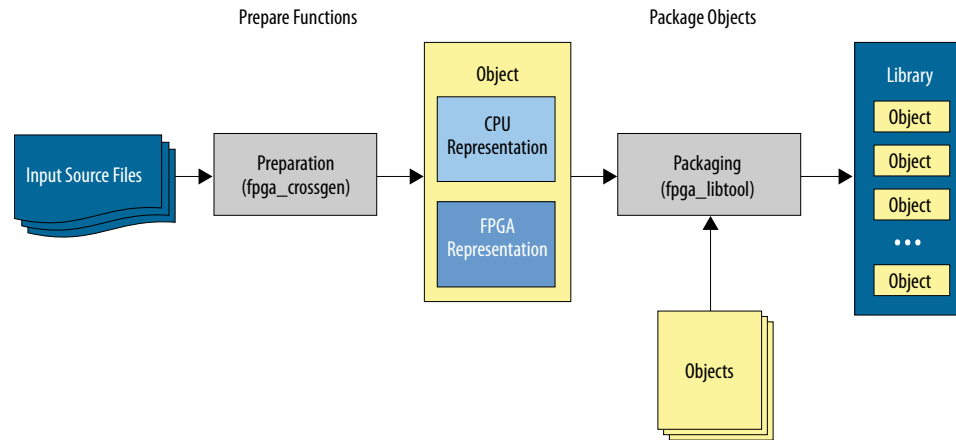
An object can be targeted for use with only one Intel high-level design product. If you want to target more than one high-level design product, you must generate a separate object for each target product.

2. Object files are collected into a library file with the `fpga_libtool` command

Objects created from different types of source code can be collected into a library, provided all objects target the same high-level design product.

Libraries must be built and used by the same version number Intel FPGA high-level design tool. For example, to compile your component with the Intel HLS Compiler Version 19.4, the libraries included in your component must have been created with a version 19.4 Intel FPGA high-level design tool.

Figure 17. High-Level View of the Library Creation Process



11.2. Creating an Object Library

Creating an object library is a multistep process where you create the objects that you want to include in a library and then collect the objects into a library file.

To create an object library:

1. Create the objects for your library with the `fpga_crossgen` command. You can create your objects from a variety of sources:
 - Create an object from HLS code.
For details, see [Creating Objects From HLS Code](#) on page 85.
 - Create an object from RTL code.
For details, see [Creating Objects From RTL Code](#) on page 87.
 - Create an object from OpenCL code.
For details, see [Creating Library Objects From OpenCL Code](#) in the *Intel FPGA SDK for OpenCL Pro Edition Programming Guide*.
2. Collect the objects into an object library with the `fpga_libtool` command.
For details, see [Packaging Object Files Into a Library](#) on page 100.

For example, if you wanted to create a Linux HLS object library called `foobar` from HLS code in a file called `foo.cpp` and OpenCL code in a file called `bar.cl`, run the following commands:

```
fpga_crossgen foo.cpp -target hls -o foo.o
fpga_crossgen bar.cl -target hls -o bar.o
fpga_libtool -target hls -create foobar.a foo.o bar.o
```

You can use the resulting library (`foobar.a`) in your component by including the header file or files that you created for the library (for example `,foobar.a`, or `foo.h` and `bar.h`) in your component.



When you compile your component, specify the library in the option of the `i++` command. For example, to compile a component `baz.cpp` that uses the `foobar.a` library, issue the following command:

```
i++ baz.cpp foobar.a
```

11.3. Creating Objects From HLS Code

You can create a library from object files from your HLS source code. An HLS-based object file contains code for CPU execution (testbench and emulation) and FPGA execution. A library can contain multiple object files.

Restriction: Creating object files from HLS code is supported only on Linux operating systems.

You can create object files for use with different Intel high-level design tools from the same HLS source code.

Depending on the target high-level design tool, your source code might require adjustments to support tool-specific data types or constructs.

Intel HLS Compiler

No additional work is needed in your HLS source code when you use the code to create objects for Intel HLS Compiler libraries.

Intel FPGA SDK for OpenCL

The Intel FPGA SDK for OpenCL supports language constructs that are not natively supported by C++. Your component might need modifications to support those constructs.

The Intel HLS Compiler supports a limited set of OpenCL language constructs through the `ocl_types.h` header file. For details, review [Supported OpenCL Language Constructs](#) on page 86.

Restriction: You cannot use systems of tasks in components intended for use in an OpenCL library object.

To create an object from your HLS code that targets the Intel FPGA SDK for OpenCL, you must have the Intel FPGA SDK for OpenCL Pro Edition installed. The version of the SDK must be the same as your version of Intel HLS Compiler.

11.3.1. Creating an Object File From HLS Code

Use the `fpga_crossgen` command to create objects for your library from your HLS code. An object created from HLS code contains information required both for emulating the functions in the object and synthesizing the hardware for the object functions.

Restriction: Creating object files from HLS code is supported only on Linux operating systems.

The `fpga_crossgen` command creates one object file from one input source file. The object created can be used only libraries that target the same Intel high-level design tool.

Objects are assigned the same version number as the version number of your Intel HLS Compiler installation. Libraries can contain only objects with the same version number, and can only be used with Intel high-level design tools with the same version number.

1. Create a library object with the following command:

```
fpga_crossgen <source_file> --target target_HLD_tool [-o <object_file_name>]
```

Where the command parameters are defined as follows:

- *target_HLD_tool*

The target Intel high-level design tool for this library. This parameter can have one of the following values:

- *hls*

Target this object to be included in libraries for components developed with the Intel HLS Compiler.

Objects built for the Intel HLS Compiler are created as operating system specific object files (.o on Linux). You cannot use objects created on one operating system with the Intel HLS Compiler running on a different operating system.

- *aoc*

Target this object to be included in libraries for kernels developed with the Intel FPGA SDK for OpenCL.

Objects built for the Intel FPGA SDK for OpenCL are not operating system specific. The objects are created as Intel FPGA SDK for OpenCL object files (.aoco).

You must have the Intel FPGA SDK for OpenCL Pro Edition installed to use this option. The version of the SDK must be the same as your version of Intel HLS Compiler.

If you do not specify an object file name with the `-o` option, the object file name defaults to be the same name as the source file name.

After you generated all objects that you want to include in your library, collect the objects in the library with the `fpga_libtool` command. For details, see [Packaging Object Files Into a Library](#) on page 100.

11.3.2. Supported OpenCL Language Constructs

If you are using the Intel HLS Compiler to develop libraries to use with the Intel FPGA SDK for OpenCL, you might need access to OpenCL language constructs that are not typically available natively from C++ language elements. The Intel HLS Compiler provides support for some OpenCL language constructs through the `ocl_types.h` header file.

All basic OpenCL data types (`double`, `float`, `long long`, `long`, `int`, `short`, `char` and `bool`) are supported without needing the `ocl_types.h` header file.

Add OpenCL language construct support by adding the following code to your component:

```
#include "HLS/ocl_types.h"
```



The OpenCL types header file adds support for the following OpenCL language constructs:

- [OpenCL address space qualifiers](#)
- Arbitrary precision integers (up to 64 bits)
- OpenCL vector data types

Note: The size of the `long` data type is 64 bits in OpenCL and for the Intel HLS Compiler on Linux systems. For Intel HLS Compiler on Windows systems, the size of `long` is 32 bits.

OpenCL Address Space Qualifiers

The `ocl_types.h` header file adds macros to support defining pointer in different OpenCL address spaces as follows:

OpenCL Address Space Qualifier	Intel HLS Compiler Macro
<code>__global</code>	<code>OCL_ADDRSP_GLOBAL</code>
<code>__local</code>	<code>OCL_ADDRSP_LOCAL</code>
<code>__constant</code>	<code>OCL_ADDRSP_CONSTANT</code>
<code>__private</code>	<code>OCL_ADDRSP_PRIVATE</code>

Arbitrary Precision Integers

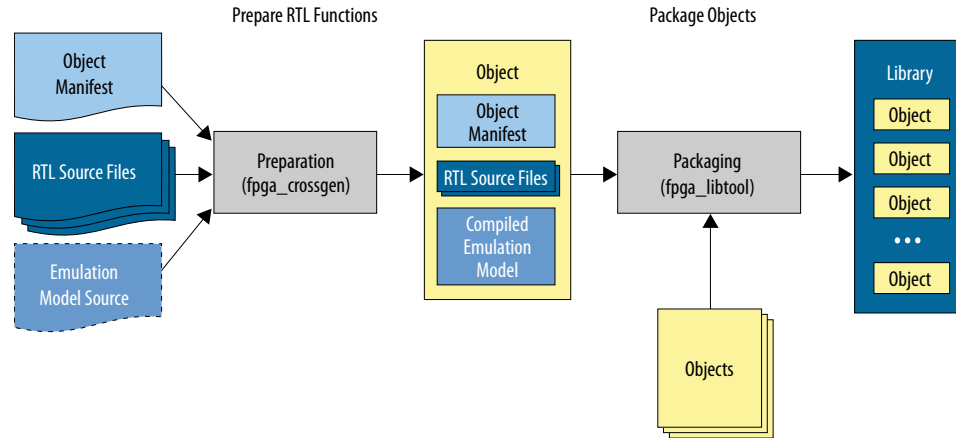
The `ocl_types.h` header file supports the OpenCL `intX_t` and `uintX_t` data types up to 64 bits. That is, you can use `int1_t - int64_t` and `uint1_t - uint64_t` in your component.

11.4. Creating Objects From RTL Code

You can create a library from object files that package register transfer level (RTL) language source files. An RTL-based object file also contains an object manifest file (in XML format) that identifies the functions that are callable in the object file. A library can contain multiple RTL-based objects.

Creating a library from RTL code is a two-step process. First, each object file is created from the RTL source and emulation models as described in the object manifest file with the `fpga_crossgen` command. Then, one or more object files are collected into an HLS library file with the `fpga_libtool` command.

Figure 18. High-Level View of the Library Creation Process from RTL



To create a library from RTL code, you need to create the following files and components:

Table 25. Files and Components Required for Creating a Library From RTL Code

File or Component	Description
RTL-based Functions	
RTL module source files	Verilog (.v), System Verilog (.sv), or VHDL (.vhd) files and accompanying memory initialization files (.mif or .hex) that define the RTL modules in the library. You cannot use additional files such as Intel Quartus Prime IP File (.qip), Synopsys Design Constraints File (.sdc), or Tcl Script File (.tcl).
Object manifest file	An XML (.xml) file that describes the properties of the callable functions available in the RTL module. The Intel HLS Compiler uses these properties to integrate the RTL module in an HLS library into the component pipeline.
RTL module function signature file	A C-style header file (.h) that declares the signatures of the functions that are implemented by the RTL module and described in the RTL module properties file. Use this header file in your HLS component source code so that your component can call the functions provided in the HLS library.
HLS emulation model files	C++ files (.cpp and .h) that contain code that is functionally equivalent to the RTL component and has the same function signatures as the RTL component. The emulation model is used only for component emulation. Co-simulations use the RTL provided in the library.

11.4.1. RTL Modules and the HLS Pipeline

HLS libraries allows you to use RTL modules that are written in Verilog, SystemVerilog, or VHDL inside HLS components. The Intel HLS Compiler integrates the RTL modules into the HLS pipeline architecture.

Consider using HLS libraries in the following situations:

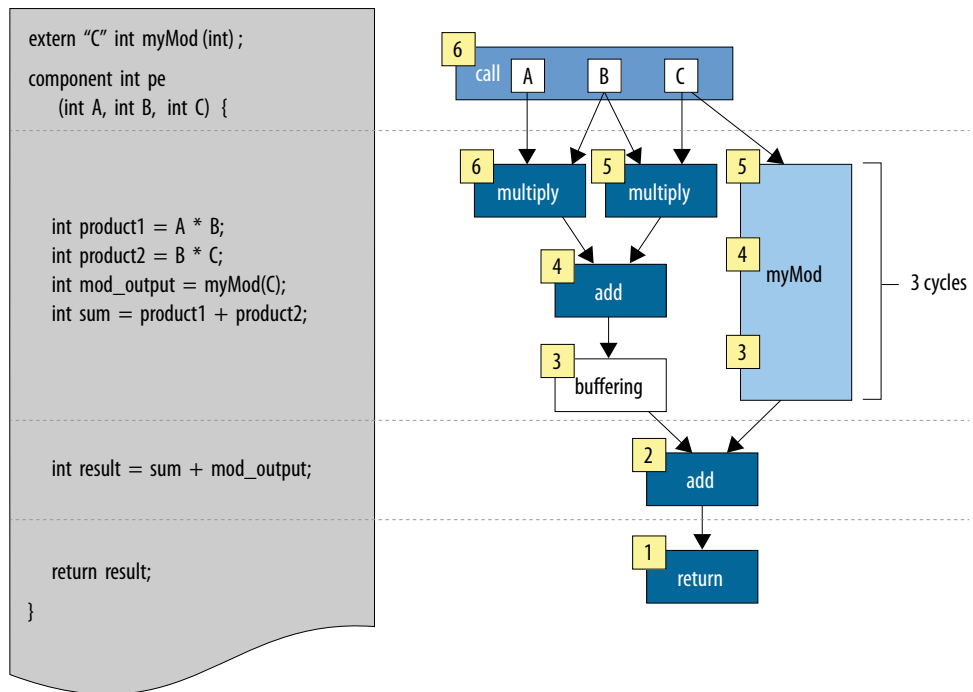
- You want to use optimized and verified RTL modules in HLS components without rewriting the modules as C++ functions.
- You want to implement HLS component functionality that you cannot express effectively in C++.

11.4.1.1. Integration of an RTL Module into the HLS Pipeline

When you specify an HLS library during component compilation, the Intel HLS Compiler integrates the RTL module within the library into the overall component pipeline.

The following figure shows how an HLS library called `myMod` might be integrated into the example pipeline described in [Intel HLS Compiler Pipeline Approach](#) on page 13.

Figure 19. Example of Pipeline Architecture That Integrates an HLS Library



The depicted RTL module has a latency of 3 cycles. Since the multiply and add operations have a latency of just one cycle, the compiler inserts buffering to balance the latency of the parallel data paths in the pipeline. A balanced latency allows the invocations of the HLS component to execute without stalling the pipeline.

Specifying the latency of the RTL module in the HLS library object manifest file allows the HLS compiler to balance the pipeline latencies in the HLS component. The pipeline integration protocol uses ready/valid handshaking, so the latency of the RTL module

can be variable. However, the variability in the latency should be small to maximize performance. In addition, specify the latency in the HLS library object manifest file for the object in the HLS library so that the RTL module experiences a good approximation of the actual latency in steady state.

Note: You must specify the RTL module latency correctly in the HLS library object manifest file, or you get bad quality of results (QoR) for your component.

11.4.1.2. RTL Module Interfaces

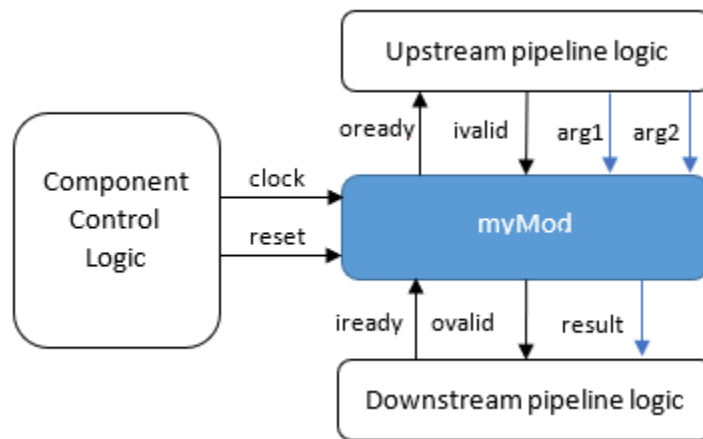
For an RTL module to properly interact with other compiler-generated operations, you must support a simple ready/valid handshaking protocol at both the input and the output of an RTL module.

An RTL module must use a single streaming interface. That is, a single pair of `ready` and `valid` logic must control all the inputs.

You have the option to provide the necessary streaming ports but declare the RTL module as stall-free. In this case, you do not have to implement proper stall behavior because the Intel HLS Compiler creates a wrapper for your module.

You must handle `invalid` signals properly if your RTL module has an internal state. For more information, see [Stall-Free RTL](#) on page 97.

Consider the following interfaces for the RTL module `myMod`:



In this diagram, `myMod` interacts with the upstream module through data signals, `arg1` and `arg2`, and control signals, `ivalid` (input) and `oready` (output). The `ivalid` control signal equals 1 (`ivalid = 1`) if and only if data signal `arg1` and data signal `arg2` contain valid data. When the control signal `oready` equals 1 (`oready = 1`), it indicates that the `myMod` RTL module can process the data signals `arg1` and `arg2` if they are valid (that is, `ivalid = 1`). When `ivalid = 1` and `oready = 0`, the upstream module holds the values of `ivalid`, `arg1`, and `arg2` in the next clock cycle.



The `myMod` module interacts with the downstream pipeline logic through the data signal `result` and the control signals, `ovalid` (output) and `iready` (input). The `ovalid` control signal equals 1 (`ovalid = 1`) if and only if the data signal `result` contains valid data. When the `iready` control signal equals 1 (`iready = 1`), the downstream module can process the data signal `result` if it is valid. When `ovalid = 1` and `iready = 0`, the `myMod` RTL module must hold the valid of the `ovalid` and `result` signals in the next clock cycle.

11.4.1.3. RTL Reset and Clock Signals

Resets and clocks of RTL modules are connected to the same clock and reset drivers as the rest of the HLS pipeline.

Because of the common clock and reset drivers, an RTL module runs in the same clock domain as the HLS component that is integrating the RTL module. The module reset input is asserted whenever the HLS component is reset.

11.4.1.3.1. Intel Stratix 10 Design-Specific Reset Requirements for Stall-Free and Stallable RTL Modules

When you create an RTL module for Intel Stratix 10 HLS designs, ensure that the module satisfies specific logic reset requirements.

Reset Requirements for Stall-Free RTL Modules

A stall-free RTL module is a fixed-latency module for which the Intel HLS Compiler can optimize away stall logic.

- When creating a stall-free RTL module for an Intel Stratix 10 design, use synchronous clear signals only.
- After **deassertion** of the reset signal to the stall-free RTL module, the module must be operational within 15 clock cycles. If the reset signal is pipelined within the module, this requirement limits the reset pipelining to no more than 15 stages.

Reset Requirements for Stallable RTL Modules

A stallable RTL module has a variable latency, and it relies on backpressured input and output interfaces to function correctly.

- When creating a stallable RTL module for an Intel Stratix 10 design, use synchronous clear signals only.
- After **assertion** of the reset signal to the stallable RTL module, the module must deassert its `oready` and `ovalid` interface signals within 40 clock cycles.
- After **deassertion** of the reset signal to the stallable RTL module, the module must be fully operational within 40 clock cycles. The module signals its readiness by asserting the `oready` interface signal.

11.4.1.4. Object Manifest File Syntax

The HLS library object manifest file is an XML file that maps the RTL modules in a library object to functions that can be called by your HLS code. The Intel HLS Compiler uses the properties defined in the manifest file to integrate an RTL module into the component pipeline.

The following example show a simple object manifest file for an RTL module that implements a double-precision square root function. The RTL module is implemented in VHDL with a Verilog wrapper.

The following object manifest file is for an RTL module named `my_fp_sqrt_double` (line 2) that implements a callable function with a C interface named `my_sqrtfd` (line 2).

```

1: <RTL_SPEC>
2:   <FUNCTION name="my_sqrtfd" module="my_fp_sqrt_double">
3:     <ATTRIBUTES>
4:       <IS_STALL_FREE value="yes"/>
5:       <IS_FIXED_LATENCY value="yes"/>
6:       <EXPECTED_LATENCY value="31"/>
7:       <CAPACITY value="31"/>
8:       <HAS_SIDE_EFFECTS value="no"/>
9:       <ALLOW_MERGING value="no"/>
10:      <PARAMETER name="WIDTH" value="64"/>
11:    </ATTRIBUTES>
12:    <INTERFACE>
13:      <AVALON port="clock" type="clock"/>
14:      <AVALON port="resetn" type="resetn"/>
15:      <AVALON port="ivalid" type="ivalid"/>
16:      <AVALON port="iready" type="iready"/>
17:      <AVALON port="ovalid" type="ovalid"/>
18:      <AVALON port="oready" type="oready"/>
19:      <INPUT port="datain" width="64"/>
20:      <OUTPUT port="dataout" width="64"/>
21:    </INTERFACE>
22:    <REQUIREMENTS>
23:      <FILE name="my_fp_sqrt_double_s5.v" />
24:      <FILE name="fp_sqrt_double_s5.vhd" />
25:    </REQUIREMENTS>
26:    <RESOURCES>
27:      <ALUTS value="2057"/>
28:      <FFS value="3098"/>
29:      <RAMS value="15"/>
30:      <MLABS value="43"/>
31:      <DSPS value="1.5"/>
32:    </RESOURCES>
33:  </FUNCTION>
34: </RTL_SPEC>

```

Table 26. Elements and Attributes in the Object Manifest File

XML Element	Description
RTL_SPEC	Top-level element in the object manifest file. There can only be one such top-level element in the file.
FUNCTION	Element that defines the HLS function that the RTL module implements. The name attribute within the FUNCTION element specifies the function name. You might have multiple FUNCTION elements, each declaring a different function that you can call from the HLS component. The same RTL module can implement multiple functions by specifying different parameters. To use the same module with different parameter combinations, create a separate FUNCTION tag for each parameter combination.

continued...



XML Element	Description
ATTRIBUTES	<p>Element that contains other XML elements that describe various characteristics (for example, latency) of the RTL module.</p> <p>The example RTL module takes one PARAMETER setting named WIDTH, which has a value of 64.</p> <p>See Table 27 on page 93 for more details other ATTRIBUTES-specific elements.</p> <p>If you create multiple RTL-based functions using different modules or use the same RTL module with different PARAMETER settings, you must create a separate FUNCTION element for each function.</p>
INTERFACE	<p>Element that contains other XML elements that describe the RTL module interface.</p> <p>The example object manifest file shows the streaming interface signals that every RTL module must provide (that is, clock, resetn, ivalid, iready, ovalid, and oready).</p> <p>The resetn signal is active low. Its synchronicity depends on the target device:</p> <p><i>Intel Arria® 10</i> The resetn signal is asynchronous to the clock signal.</p> <p><i>Intel Stratix 10</i> The resetn signal is synchronous to the clock signal. For more information about reset signal timing, see Intel Stratix 10 Design-Specific Reset Requirements for Stall-Free and Stallable RTL Modules on page 91.</p> <p>The signal names must match the ones specified in the RTL module properties file. An error occurs during library creation if a signal name is different in the RTL code and the RTL module properties file.</p>
REQUIREMENTS	<p>Element that specifies one or more RTL resource files (that is, .v, .sv, .vhdl, .hex, and .mif). The specified paths to these files are relative to the location of the object manifest file. Each RTL resource file becomes part of the associated Platform Designer component that corresponds to the entire HLS component.</p> <p>HLS libraries do not support .qip files.</p>
RESOURCES	<p>Optional element that specifies an estimate of the FPGA resources that the RTL module uses. If you do not specify this element, the estimated FPGA resources that the RTL module uses defaults to zero in the HLS resource estimation report.</p>

11.4.1.4.1. XML Elements for ATTRIBUTES

In the RTL module properties file of the RTL module within an HLS library, there are XML elements under ATTRIBUTES that you can specify to set module characteristics.

Table 27. XML Elements for the RTL module properties file ATTRIBUTES Element

XML Element	Description
IS_STALL_FREE	<p>Instructs the Intel HLS Compiler to remove all stall logic around the RTL module.</p> <p>Set IS_STALL_FREE to "yes" to indicate that the module does not generate stalls internally and it cannot properly handle incoming stalls. The module ignores the stall input.</p> <p>If you set IS_STALL_FREE to "no", the module must properly handle all stall and valid signals.</p> <p>If you set IS_STALL_FREE to "yes", you must also set IS_FIXED_LATENCY to "yes". Also, if the RTL module has an internal state, it must properly handle ivalid=0 inputs.</p>
IS_FIXED_LATENCY	<p>Indicates whether the RTL module has a fixed latency.</p>

continued...

XML Element	Description
	<p>Set <code>IS_FIXED_LATENCY</code> to "yes" if the RTL module always takes a known number of clock cycles to compute its output. The value you assign to the <code>EXPECTED_LATENCY</code> element specifies the number of clock cycles.</p> <p>The safe value for <code>IS_FIXED_LATENCY</code> is "no". When you set <code>IS_FIXED_LATENCY="no"</code>, the <code>EXPECTED_LATENCY</code> value must be at least 1.</p> <p>For a given RTL module, you may set <code>IS_FIXED_LATENCY</code> to "yes" and <code>IS_STALL_FREE</code> to "no". Such a module produces its output in a fixed number of clock cycles and handles stall signals properly.</p>
EXPECTED_LATENCY	<p>Specifies the expected latency of the RTL module.</p> <p>If you set <code>IS_FIXED_LATENCY</code> to "yes", set the <code>EXPECTED_LATENCY</code> value to be the exact latency of the module. Otherwise, the Intel HLS Compiler generates incorrect hardware.</p> <p>For a module with variable latency, the Intel HLS Compiler balances the pipeline around this module to the <code>EXPECTED_LATENCY</code> value that you specify. For modules that can stall and require use of signals such as <code>iready</code>, the <code>EXPECTED_LATENCY</code> value must be set to at least 1.</p> <p>The specified value and the actual latency might differ for a module with variable latency, which might affect the number of stalls inside the pipeline. However, the resulting hardware is functionally correct.</p>
CAPACITY	<p>Specifies the number of multiple inputs that this module can process simultaneously. You must specify a value for <code>CAPACITY</code> if you also set <code>IS_STALL_FREE="no"</code> and <code>IS_FIXED_LATENCY="no"</code>. Otherwise, you do not need to specify a value for <code>CAPACITY</code>.</p> <p>If <code>CAPACITY</code> is strictly less than <code>EXPECTED_LATENCY</code>, the Intel HLS Compiler automatically inserts capacity-balancing FIFO buffers after this module when necessary.</p> <p>A conservative but safe value for <code>CAPACITY</code> is 1.</p>
HAS_SIDE_EFFECTS	<p>Indicates whether the RTL module has side effects. Modules that have internal states or communicate with external memories are examples of modules with side effects.</p> <p>Set <code>HAS_SIDE_EFFECTS</code> to "yes" to indicate that the module has side effects. Specifying <code>HAS_SIDE_EFFECTS</code> to "yes" ensures that optimization efforts do not remove calls to modules with side effects.</p> <p>Stall-free modules with side effects (that is, <code>IS_STALL_FREE="yes"</code> and <code>HAS_SIDE_EFFECTS="yes"</code>) must properly handle <code>ivalid=0</code> input cases because the module might receive invalid data occasionally.</p> <p>A conservative but safe value for <code>HAS_SIDE_EFFECTS</code> is "yes".</p>
ALLOW_MERGING	<p>This attribute is reserved for future use.</p> <p>To prevent unexpected behavior, always set this attribute as <code><ALLOW_MERGING value="no" /></code>.</p>
PARAMETER	<p>Specifies the value of an RTL module parameter.</p> <p><code>PARAMETER</code> attributes:</p> <ul style="list-style-type: none"> • <code>name</code> Specifies the name of the RTL module parameter. • <code>value</code> Specifies a decimal numeric value for the parameter. <p>The value for an RTL module parameter can be specified using either a <code>value</code> or a <code>type</code> attribute.</p>

11.4.1.4.2. XML Elements for INTERFACE

In the RTL module properties file of the RTL module within an HLS library, there are XML elements under `INTERFACE` that define aspects of the RTL module interface.

The RTL module cannot access the memories of the HLS component.



Table 28. Mandatory XML Elements for the RTL module properties file INTERFACE Element

XML Element	Description
INPUT	<p>Specifies the input parameter of the RTL module that receives the value of a call argument with the RTL-based function is called.</p> <p>INPUT attributes:</p> <ul style="list-style-type: none"> port Specifies the port name of the RTL module. width Specifies the width of the port in bits. The width of the port must match the size of the C datatype corresponding to the call argument. The port width is always a multiple of 8 bits. <p>All call arguments must be passed by value. You cannot use reference, pointer, and array type arguments.</p>
OUTPUT	<p>Specifies the output parameter of the RTL module that represents the return value of functions based on this module.</p> <p>OUTPUT attributes:</p> <ul style="list-style-type: none"> port Specifies the port name of the RTL module. width Specifies the width of the port in bits. The width of the port must match the size of the C datatype of the function return value. The port width is always a multiple of 8 bits. <p>The return value cannot be a pointer.</p>

11.4.1.4.3. XML Elements for RESOURCES

In the RTL module properties file of the RTL module within an HLS library, there are optional elements under `RESOURCES` that you can define to specify the estimated FPGA resource utilization of the module. If you do not specify a particular element, it is assigned a default value of zero in the report estimates.

Table 29. XML Elements for the RTL module properties file RESOURCES Element

XML Element	Description
ALUTS	Specifies the number of combinational adaptive look-up tables (ALUTs) that the module uses.
FFS	Specifies the number of dedicated logic registers that the module uses.
RAMS	Specifies the number of block RAMs that the module uses.
DSPS	Specifies the number of digital signal processing (DSP) blocks that the module uses.
MLABS	Specifies the number of memory logic arrays (MLABs) that the module uses. This value is equal to the number of adaptive logic modules (ALMs) that is used for memory divided by 10 because each MLAB consumes 10 ALMs.

11.4.1.5. Mapping HLS Datatypes to RTL Signals

All supported composite datatypes are represented by wide input or output signals. Typically, the components of a composite datatype are presented with the first-declared value or value of lowest index in the low-order bits of the signal.

Arrays

In C++, arrays are passed as a pointer to the memory in which the array is stored.

The Intel HLS Compiler does not support pointer parameters for RTL modules. However, C++ allows you to pass a struct by value, so you can declare a struct datatype that has an array as one of its members and declare your function to accept an argument of this struct-type by value.

Structs

You can use both packed and unpacked structs as call arguments and return values in your HLS components and tasks. The members of a struct are presented as slices of the input signal, with the first-declared struct member in the lowest-order bits of the input signal.

- **Unpacked Structs**

When your struct declaration is not packed, the layout of the input signal corresponding to the struct datatype is determined by C language-specific padding rules that cause the Intel HLS Compiler to insert padding bytes before struct members that require a specific alignment.

You should use packed structs as arguments to your RTL modules unless there is a specific reason to conform to a particular padded struct layout.

- **Packed Structs**

If the struct type is declared as packed, member values start on an 8-bit boundary.

The Intel HLS Compiler does not insert padding bytes to align struct members on platform-defined boundaries. The second-declared member always starts in the next highest byte after high-order byte of the first-declared struct member.

- **System Verilog Structs**

If you are developing an RTL module in System Verilog, you can declare a System Verilog struct type that corresponds to the C++ struct type that is mapped to the input signal of your RTL module.

The declaration order of the struct members is reversed in the System Verilog declaration because it specifies how the member signals should be concatenated to produce the composite signal. In a System Verilog concatenation expression, the bits are specified from high to low. That is, the last byte of the C++ struct type must be listed first in the System Verilog signal concatenation.

You can compile your emulation models as HLS components to obtain an `interface_structs.v` file that contains declarations of the System Verilog struct types corresponding to the struct-type arguments of those functions. For details, see the following tutorial:

```
<quartus_installdir>/hls/examples/tutorials/libraries/  
rtl_struct_mapping
```

- **Pointers in Structs**

You cannot use struct types that have reference or pointer members as arguments to or return values from RTL-based functions.

11.4.1.6. HLS Emulation Models for RTL-Based Functions

For an RTL-based function, write C++ code that serves as an emulation model for that function. This model is used when you run your component in emulation mode.

The emulation model is not used when you co-simulate your component; co-simulations use RTL extracted from the library.



Important: If your function uses `static` variables to hold internal state, the emulation is equivalent to the RTL functionality only if the function is called from only one place in the HLS component.

This behavior is different because on CPUs all calls to the function share the same state variables. On FPGAs, the RTL module is instantiated once for each location in the HLS component where the function is called, and these instances do not share state.

11.4.1.7. Potential Incompatibility between RTL Modules and Partial Reconfiguration

When you create an HLS library using RTL modules, you might encounter Partial Reconfiguration-related issues.

Consider a situation where you create and verify your library on a device that does not support Partial Reconfiguration (PR). If you then use the library RTL module inside a PR region, the module might not function correctly after PR.

To ensure that the RTL modules function correctly on a device that uses PR:

- The RTL modules do not use memory logic array blocks (MLABs) with initialized content.
- The RTL modules do not make any assumptions regarding the power-up values of any logic.

For complete PR coding guidelines, refer to [Creating a Partial Reconfiguration Design](#) in the *Partial Reconfiguration User Guide*.

11.4.1.8. Stall-Free RTL

The Intel HLS Compiler can optimize hardware resource usage and performance by not placing stall logic around an RTL module with fixed latency.

If you have an RTL module with a fixed latency that you want integrated into your component pipeline without surrounding stall logic, ensure that you set attributes in the object manifest file (`.xml`) as follows:

1. Specify a value for the `EXPECTED_LATENCY` attribute (under the `FUNCTION` element) so that the latency equals the number of pipeline stages in the module.

Important: An inaccurate `EXPECTED_LATENCY` value causes the RTL module to be out of sync with the rest of the pipeline, and can lead to functionally incorrect results.

2. Set the `IS_STALL_FREE` attribute under the `FUNCTION` element to "yes".

This setting instructs the Intel HLS Compiler to avoid placing stall logic around the RTL module. This setting also tells the compiler that the RTL module produces a result after the number cycles specified in the `EXPECTED_LATENCY` attribute after accepting input values. The stall free logic produces a result every cycle but the result is delayed by the number cycles specified in the `EXPECTED_LATENCY` attribute.

For RTL modules with a fixed latency, the output signals (`ovalid` and `oready`) can have constant high values, and the input ready signal (`iready`) can be ignored.

A stall-free RTL module might receive an invalid input signal (`ivalid` is low). In this case, the module must produce invalid data on the output `EXPECTED_LATENCY` cycles after the cycle in which the input was invalid. For a stall-free RTL module without an internal state, you might find it convenient to propagate the invalid input through the module. If the module has an internal state, that state should not be affected by data inputs that are not accompanied by `ivalid = 1`.

11.4.1.9. RTL Module Restrictions and Limitations for HLS Libraries

RTL modules that you want to include in an HLS library are subject to some restrictions and limitations to ensure that the library works consistently across different user designs.

RTL Module Restrictions

When you create an RTL module, ensure that it operates within the following restrictions:

- The RTL module must work correctly at any clock frequency that passes timing analysis.
- Data input and output sizes must match the sizes of the arguments and return value declared in the RTL module function signature (`.h`) file. The input and output sizes must always be the size of a C++ standard type: `char`, `short`, `int`, `long`, `float`, or `double`.

For example, if you work with 24-bit values inside an RTL module, declare inputs to be 32 bits and declare the function signature to accept the `uint` data type. In the RTL module, accept the 32-bit input but discard the top 8 bits.

- RTL modules cannot connect to external I/O signals. All input and output signals must come from the HLS component that uses the library.
- An RTL module must have a `clock` port, a `resetn` port, and handshaking ports to support the data input and output interfaces. The handshake signal must be named `ivalid`, `ovalid`, `iready`, and `oready`.
- Every function call that corresponds to an RTL module instantiation is completely independent of other instantiations. No hardware is shared.
- An RTL module must receive all its inputs at the same time. A single `ivalid` input signifies that all inputs contain valid data.

RTL-Based Object Limitations

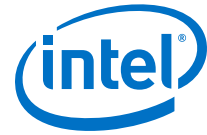
Using RTL modules in HLS libraries has the following limitations:

- You can only set RTL module parameters in the object manifest file (`.xml`) file. To use the same module with multiple parameter combinations, create a separate `FUNCTION` tag for each parameter combination.
- Pass data inputs to the RTL module only by value through the HLS component code.

You cannot pass streams, pointers, or references as input to an RTL module.

For streaming data, extract data from the stream first in your component and then pass the extracted scalar data to the RTL module in the HLS library.

Passing data inputs to an RTL module as pointers or references causes a fatal error in the Intel HLS Compiler.



- Names of RTL module source files cannot conflict with the names of objects in other libraries or in file names of Intel HLS Compiler IP.

When you create a library, choose RTL module names that are unlike to conflict with other libraries or compiler IP. For example, prefix the name of your RTL modules with the name of your library.

If there is a naming conflict, the Intel Quartus Prime compilation of the HLS component might fail or result in a functionally-incorrect FPGA image.

- Names of the RTL module and its signals cannot conflict with reserved names defined by any of the supported RTL languages: Verilog, System Verilog, and VHDL.
- The Intel HLS Compiler does not support `.qip` files. You must manually parse nested `.qip` files to create a flat list of RTL files.

11.4.2. Creating an HLS-Library Object File from an RTL Module

Before an RTL module can be included in a library intended for use in an Intel HLS Compiler design, create a platform-specific object (`.o` files on Linux, `.obj` files on Windows) from the RTL module. Use the `fpga_crossgen` command to create the object.

For instructions on creating an OpenCL library object file from RTL, see “[Packaging an RTL Component for an OpenCL Library](#)” in the *Intel FPGA SDK for OpenCL Pro Edition Programming Guide*. Before you can create an HLS library object from an RTL module, ensure that the functions in your RTL module are functionally correct and that you have the following files ready:

- RTL module source files
These files are the the Verilog (`.v`), System Verilog (`.sv`), or VHDL (`.vhd1`) files and the accompanying memory initialization files (`.mif` or `.hex`) that define the RTL modules.
- RTL object manifest file
This XML file describes the callable interfaces of your RTL modules. Review [Object Manifest File Syntax](#) on page 92 for details about what to include in this XML file.
- HLS emulation model file
These C++ files (`.cpp` and `.h`) provide an emulation model for the RTL module that allows you to emulate your component when it includes an HLS library that contains this RTL module. Full hardware compilations use the RTL source files.
- RTL module function signature file
This C-style header file (`.h`) declares the signatures of the functions that are implemented by the RTL module and described in the object manifest file. Include this file in you HLS component code for the component to call the functions provided by the RTL modules packaged in the object.

- After you have the files ready, create the HLS library object with the following command:

```
fpga_crossgen <object_manifest_file_name> --target hls --emulation_model  
<emulation_model_file_location> [-o <object_file_name>]
```

Where `<object_manifest_file_name>` is the full path of the RTL object manifest (`.xml`) file including the file name. This path can be a full or relative path.

If you do not specify an object file name with the `-o` option, the object file name defaults to be the same name as the object manifest file name. That is, an object manifest file named `manifest.xml` results in an object file named `manifest.o` (on Linux) or `manifest.obj` (on Windows).

The output of the command is a platform-specific object file (`.o` on Linux, `.obj` on Windows). The platform of the object file is determined by the platform where you run the `fpga_crossgen` command. When you run the command on Linux, you get a `.o` object file. When you run the command on Windows, you get a `.obj` object file.

Important: Each object created with the `fpga_crossgen` command is assigned a compiler version number. You can package only object with the same version number into a library, and a library can be used only with a target compiler (For example, `i++`) with the same version number.

11.5. Packaging Object Files Into a Library

Collect object files in a library file so that others can incorporate the library into their projects and call the functions that are contained in the objects in the library. Package object files into a library with the `fpga_libtool` command.

Before you package object files into a library, ensure that you have the path information for all of the object files that you want to include in the library. All objects to be packaged in the library must have the same version number. This library can be used only by an Intel high-level design tool with the same version number.

The `fpga_libtool` command creates libraries encapsulated in operating system specific archive files (`.a` on Linux, `.lib` on Windows). Create the HLS library file with the following command:

```
fpga_libtool --target target_HLD_tool --create library_name[.a | .lib  
| .aoclib] object_file_1 [object_file_2 ... object_file_n]
```

Where the command parameters are defined as follows:



- *target_HLD_tool*

The target Intel high-level design tool for this library. This parameter can have one of the following values:

- *hls*

Target this library for components developed with the Intel HLS Compiler.

Libraries built for the Intel HLS Compiler are encapsulated in operating system specific archive files (*.a* on Linux, *.lib* on Windows). You cannot use HLS libraries created on one operating system with the Intel HLS Compiler running on a different operating system.

- *aoc*

Target this library for kernels developed with the Intel FPGA SDK for OpenCL.

Libraries built for the Intel FPGA SDK for OpenCL are not operating system specific. The objects are created as Intel FPGA SDK for OpenCL object files (*.aoclib*).

You must have the Intel FPGA SDK for OpenCL Pro Edition installed to use this option. The version of the SDK must be the same as your version of Intel HLS Compiler.

library_name

The name of the library file.

Specify the file extension of the library files as follows, depending on the target high-level design tool:

- Intel HLS Compiler

Specify the operating-system specific archive extension: *.a* for Linux-platform libraries and *.lib* for Windows-platform libraries.

- Intel FPGA SDK for OpenCL

Specify *.aoclib* as the file extension for an OpenCL library.

OpenCL libraries are not operating-system specific.

You can specify one or more object files to include in the library.

For example, the following command packages three Linux-platform objects (*prim1.o*, *prim2.o*, and *prim3.o*) into an HLS library called *libdemo*:

```
fpga_libtool --create libdemo.a prim1.o prim2.o prim3.o --target hls
```

12. Advanced Hardware Synthesis Controls

12.1. The `hls_fpga_reg()` Function

In some cases, explicitly asking the compiler to insert a register stage between the operand and the return value of the function call can help improve the performance of your component. Use the `hls_fpga_reg()` function to insert at least one register between the operand and return value of the function call.

Typically, you do not need to use this function to achieve the performance from your component that you want.

To use the `hls_fpga_reg()` function effectively, you must know about how portions of the data path are placed on FPGA devices, and you typically use the `hls_fpga_reg()` function for the following purposes:

- Breaking the critical paths between spatially distant portions of a data path, such as between processing elements of a large systolic array.
- Reducing the pressure on placement and routing efforts caused by spatially distinct portions of the kernel implementation.

The Intel HLS Compiler does not provide feedback about where you should add `hls_fpga_reg()` function calls. Use Intel Quartus Prime software to determine where you should insert the calls to address specific aspects of component performance.

Syntax

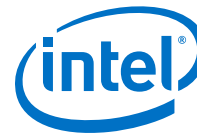
```
T hls_fpga_reg(T op)
```

where *T* can be any sized type.

Description

The `hls_fpga_reg()` function directs the Intel HLS Compiler to insert at least one hardware pipelining register on the signal path that assigns the operand to the return value. This built-in function operates as an assignment, where the operand is assigned to the return value. The assignment has no implicit semantic or functional meaning beyond a standard assignment.

Functionally, you can consider the `hls_fpga_reg()` function to be always optimized away by the compiler.



Usage Notes You can nest `hls_fpga_reg()` function calls to increase the minimum number of registers that are inserted on the assignment path. Because each function call guarantees the insertion of at least one register stage, the number of nested calls provides a lower limit on the number of registers.

For example, the following code snippet tells the compiler to insert at least two registers on the assignment path.

```
int out=hls_fpga_reg(hls_fpga_reg(in));
```

The compiler might insert more than two registers on the path.

13. Intel High Level Synthesis Compiler Pro Edition Reference Summary

13.1. Intel HLS Compiler Pro Edition i++ Command-Line Arguments

Use the i++ command-line arguments to affect how your component is compiled and linked.

General i++ Command Options

Option	Description
--debug-log	Generate the compiler diagnostics log.
-h, --help	List compiler command options along with brief descriptions.
-oresult	Place compiler output into the <result> executable and the <result>.prj directory.
-v	Display messages describing the progress of the compilation.
--version	Display compiler version information.

Command Options Affecting Compiling

Option	Default Value	Description
-c		Preprocess, parse, and generate object files.
--component <i>component name</i>		Comma-separated list of function names to synthesize to RTL. To use this option, your component must be configured with C-linkage using the extern "C" specification. For example: <pre>extern "C" int myComponent(int a, int b)</pre> Using the component function attribute is preferred over using the --component command option to indicate functions that you want the compiler to synthesize.
-Dmacro [=val]		Define a <macro> with <val> as its value.
-g		Generate debug information (default option).
-g0		Do not generate debug information.
--gcc-toolchain=<GCC_dir>		Specifies the path to a GCC installation that you want to use for compilation. This path should be the absolute path to the directory that contains the GCC lib, bin, and include folders.
--hyper-optimized-handshaking=[auto off]	auto	This option applies to Intel Stratix 10 devices only. Use this option to modify the handshaking protocol used in certain areas of your design.

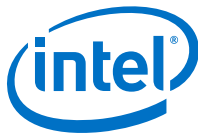
continued...

Intel Corporation. All rights reserved. Agilix, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.



Option	Default Value	Description
<code>-Idir</code>		Add directory <code><dir></code> to the end of the main include path.
<code>-march=[x86-64 FPGA_family FPGA_part_number]</code>	x86-64	Generate code for an emulator flow (x86-64) or for the specified FPGA family or FPGA part number.
<code>--quartus-compile</code>		Run the HDL generated through Intel Quartus Prime to generate accurate f_{MAX} and area estimates. Your component is not expected to cleanly close timing.
<code>--quartus-compile <seed></code>		Specifies the Fitter seed to use when your component is compiled to hardware by Intel Quartus Prime.
<code>--simulator simulator_name</code>	modelsim	Specifies the simulator you are using to perform verification. This command option can take the following values for <code><simulator_name></code> : <code>modelsim</code> Use ModelSim for component verification. <code>none</code> Disable verification. That is, generate RTL for components without the test bench. If you do not specify this option, <code>--simulator modelsim</code> is assumed.
<code>-ffp-contract=fast</code>		Remove intermediate rounding and conversion when possible, except for code blocks fenced by <code>#pragma clang fp contract(off)</code> . To learn more, review the following tutorial: <code><quartus_installdir>/hls/examples/tutorials/best_practices/floating_point_ops</code>
<code>--fpc</code>		This option is deprecated and will be removed in a future release. Use <code>-ffp-contract=fast</code> instead. Remove intermediate rounding and conversion when possible. Exception: Intermediate rounding and conversion is not removed in code blocks fenced by <code>#pragma clang fp contract(off)</code> . To learn more, review the following tutorial: <code><quartus_installdir>/hls/examples/tutorials/best_practices/floating_point_ops</code>
<code>-ffp-reassoc</code>		Relax the order of floating point arithmetic operations, except for code blocks fenced by <code>#pragma clang fp reassoc(off)</code> To learn more, review the following tutorial: <code><quartus_installdir>/hls/examples/tutorials/best_practices/floating_point_ops</code>
<code>--fp-relaxed</code>		This option is deprecated and will be removed in a future release. Use <code>-ffp-reassoc</code> instead. Relax the order of floating point arithmetic operations. Exception: The order of floating point operations in code blocks fenced by <code>#pragma clang fp reassoc(off)</code> is not relaxed. To learn more, review the following tutorial: <code><quartus_installdir>/hls/examples/tutorials/best_practices/floating_point_ops</code>
<code>--daz</code>		Disable subnormal support in double-precision floating-point computations.
<code>--rounding=[ieee faithful]</code>		Control rounding scheme for double-precision adders, multipliers, and dividers. If you do not specify this option, adders and multipliers use IEEE-754 RNE rounding (0.5 ULP) and dividers uses faithful rounding (1 ULP). The <code>-rounding</code> option can take one of the following values: <code>ieee</code> All adders, multipliers, and dividers use IEEE-754 RNE rounding. <code>faithful</code> All adders, multipliers, and dividers use faithful rounding.
<code>--clock clock target</code>	240 MHz	Optimize the RTL for the specified clock frequency or period. The <code>clock target</code> value must include a unit.



Option	Default Value	Description
		For example: <pre>i++ -march="Arria 10" test.cpp --clock 100MHz i++ -march="Arria 10" test.cpp --clock 10ns</pre>

Command Options Affecting Linking

Option	Default Value	Description
-ghdl		Enable full debug visibility and logging of all HDL signals in simulation.
-Ldir		(Linux only) Add directory <dir> to the list of directories to be searched for library files specified with the -l option.
-llibrary		(Linux only) Search the library name <library> when linking.
--x86-only		Create only the testbench executable (<result>.out/<result>.exe).
--fpga-only		Create only the <result>.prj directory and its contents.

13.2. Intel HLS Compiler Pro Edition Header Files

Coding your component to be compiled by the Intel HLS Compiler requires you to include the `hls.h` header file. Other header files provided with the Intel HLS Compiler provide FPGA-optimized implementations of certain C and C++ functions.

Table 30. Intel HLS Compiler Pro Edition Header Files Summary

HLS Header File	Description
<code>HLS/hls.h</code>	Required for component identification and component parameter interfaces.
<code>HLS/math.h</code>	Includes FPGA-specific definitions for the math functions from the <code>math.h</code> for your operating system.
<code>HLS/extendedmath.h</code>	Includes additional FPGA-specific definitions of math functions not in <code>math.h</code> .
<code>HLS/ac_int.h</code>	Provides FPGA-optimized arbitrary width integer support.
<code>HLS/ac_fixed.h</code>	Provides FPGA-optimized arbitrary precision fixed point support.
<code>HLS/ac_fixed_math.h</code>	Provides FPGA-optimized arbitrary precision fixed point math functions.
<code>HLS/ac_complex.h</code>	Provides FPGA-optimized complex number support.
<code>HLS/hls_float.h</code>	Provides FPGA-optimized arbitrary-precision IEEE IEEE 754 compliant floating-point number support.
<code>HLS/hls_float_math.h</code>	Provides FPGA-optimized floating-point math functions.
<code>HLS/stdio.h</code>	Provides <code>printf</code> support for components so that <code>printf</code> statements work in x86 emulations, but are disabled in component when compiling to an FPGA architecture.
<iostream>	To use <code>cout</code> and <code>cerr</code> in your component, guard the statements with the <code>HLS_SYNTHESIS</code> macro.

`hls.h` Header File

Syntax `#include "HLS/hls.h"`



Description Required for component identification and component parameter interfaces.

math.h Header File

Syntax `#include "HLS/math.h"`

Description Includes FPGA-specific definitions for the math functions from the `math.h` for your operating system.

To learn more, review the following tutorial:
`<quartus_installdir>/hls/examples/tutorials/
best_practices/single_vs_double_precision_math.`

extendedmath.h Header File

Syntax `#include "HLS/extendedmath.h"`

Description Includes additional FPGA-specific definitions of math functions not in `math.h`.

To learn more, review the following design:
`<quartus_installdir>/hls/examples/QRD.`

ac_int.h Header File

Syntax `#include "HLS/ac_int.h"`

Description Intel HLS Compiler version of `ac_int` header file.

Provides FPGA-optimized arbitrary width integer support.

To learn more, review the following tutorials:

- `<quartus_installdir>/hls/examples/tutorials/
ac_datatypes/ac_int_basic_ops`
- `<quartus_installdir>/hls/examples/tutorials/
ac_datatypes/ac_int_overflow`
- `<quartus_installdir>/hls/examples/tutorials/
best_practices/struct_interfaces`

ac_fixed.h Header File

Syntax `#include "HLS/ac_fixed.h"`

Description Intel HLS Compiler version of the `ac_fixed` header file.

Provides FPGA-optimized arbitrary precision fixed point support.

To learn more, review the following tutorial:

`<quartus_installdir>/hls/examples/tutorials/
ac_datatypes/ac_fixed_constructor.`

ac_fixed_math.h Header File

Syntax `#include "HLS/ac_fixed_math.h"`

Description Intel HLS Compiler version of the `ac_fixed_math` header file.

Provides FPGA-optimized arbitrary precision fixed point math functions.

To learn more, review the following tutorial:

`<quartus_installdir>/hls/examples/tutorials/
ac_datatypes/ac_fixed_math_library.`

ac_complex.h Header File

Syntax `#include "HLS/ac_complex.h"`

Description Intel HLS Compiler version of the `ac_complex` header file.

Provides FPGA-optimized complex math functions.

hls_float.h Header File

Syntax `#include "HLS/hls_float.h"`

Description Header file to provide FPGA-optimized arbitrary-precision IEEE 754 compliant floating-point number support.

To learn more, review the following tutorials:

- `<quartus_installdir>/hls/examples/tutorials/
hls_float/1_reduced_doubl`
- `<quartus_installdir>/hls/examples/tutorials/
hls_float/2_explicit_arithmetic`
- `<quartus_installdir>/hls/examples/tutorials/
hls_float/3_conversions`

hls_float_math.h Header File

Syntax `#include "HLS/hls_float_math.h"`

Description Header file to provide math functions for `hls_float` data types.

To learn more, review the following tutorials:



- `<quartus_installdir>/hls/examples/tutorials/hls_float/1_reduced_doubl`
- `<quartus_installdir>/hls/examples/tutorials/hls_float/2_explicit_arithmetic`
- `<quartus_installdir>/hls/examples/tutorials/hls_float/3_conversions`

stdio.h Header File

Syntax `#include "HLS/stdio.h"`

Description Provides `printf` support for components so that `printf` statements work in x86 emulations, but are disabled in component when compiling to an FPGA architecture.

Standard C++ <iostream> Header File

Syntax `#include <iostream>`

Description To use the C++ standard output streams (`cout` and `cerr`) provided by the standard `<iostream>` header, you must guard any standard output statements with the `HLS_SYNTHESIS` macro.

This macro ensures that statements in a component work in x86 emulations but are disabled in the component when compiling to an FPGA architecture.

13.3. Compiler-Defined Preprocessor Macros

The has a built-in macros that you can use to customize your code to create flow-dependent behaviors.

Table 31. Macro Definition for `__INTELFPGA_COMPILER__`

Tool Invocation	<code>__INTELFPGA_COMPILER__</code>
<code>g++</code> or <code>cl</code>	Undefined
<code>-march=x86-64</code>	
<code>-march="<FPGA_family_or_part_number>"</code>	

Table 32. Macro Definition for `HLS_SYNTHESIS`

Tool Invocation	<code>HLS_SYNTHESIS</code>	
	Testbench Code	HLS Component Code
<code>g++</code> or <code>cl</code>	Undefined	Undefined
<code>-march=x86-64</code>	Undefined	Undefined
<code>-march="<FPGA_family_or_part_number>"</code>	Undefined	Defined

13.4. Intel HLS Compiler Pro Edition Keywords

Table 33. Intel HLS Compiler Pro Edition Keywords

Feature	Description
component	Indicates that a function is a component. Example: <pre>component void foo()</pre>

13.5. Intel HLS Compiler Pro Edition Simulation API (Testbench Only)

Table 34. Intel HLS Compiler Pro Edition Simulation API (Testbench only) Summary

Function	Description
<code>ihc_hls_enqueue</code>	This function enqueues one invocation of an HLS component.
<code>ihc_hls_enqueue_noret</code>	This function enqueues one invocation of an HLS component. This function should be used when the return type of the HLS component is void.
<code>ihc_hls_component_run_all</code>	This function pushes all enqueued invocations of a component into the component in the HDL simulator as quickly as the component can accept new invocations.
<code>ihc_hls_sim_reset</code>	This function sends a reset signal to the component during automated simulation.
<code>ihc_hls_set_component_wait_cycle</code>	This function tells the simulation process to continue running for a specified number of cycles after the done signal for the specified component is observed.

`ihc_hls_enqueue` Function

Syntax `ihc_hls_enqueue(void* retptr, void* funcptr, /*function arguments*/)`

Description This function enqueues one invocation of an HLS component. The return value is stored in the first argument which should be a pointer to the return type. The component is not run until the `ihc_hls_component_run_all()` is invoked.

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/usability/enqueue_call`.

`ihc_hls_enqueue_noret` Function

Syntax `ihc_hls_enqueue_noret(void* funcptr, /*function arguments*/)`



Description This function enqueues one invocation of an HLS component. This function should be used when the return type of the HLS component is void. The component is not run until the `ihc_hls_component_run_all()` is invoked.

To learn more, review the tutorial: <quartus_installdir>/hls/examples/tutorials/usability/enqueue_call.

ihc_hls_component_run_all Function

Syntax `ihc_hls_component_run_all (void* funcptr)`

Description This function accepts a pointer to the HLS component function. When run, all enqueued invocations of the component will be pushed into the component in the HDL simulator as quickly as the component can accept new invocations.

To learn more, review the tutorial: <quartus_installdir>/hls/examples/tutorials/usability/enqueue_call.

ihc_hls_sim_reset Function

Syntax `int ihc_hls_sim_reset(void)`

Description This function sends a reset signal to the component during automated simulation. It returns 1 if the reset was exercised or 0 otherwise.

To learn more, review the tutorial: <quartus_installdir>/hls/examples/tutorials/component_memories/static_var_init.

ihc_hls_set_component_wait_cycle Function

Syntax `ihc_hls_set_component_wait_cycle(<component function name>, <# of wait cycles>)`

Description This function tells the simulation process to continue running for a specified number of cycles after the done signal for the specified component is observed. This delay can enable task functions with a higher latency than the component function to successfully return their output during simulation.

Use this function when you simulate a design that uses a system of tasks where the completion of a task function is not synchronized with an `ihc::collect` call.

Simulation API Code Example

```
component int foo(int val) {  
    // function definition  
}  
  
component void bar (int val) {
```

```

// function definition
}
int main() {
// .....
int input = 0;
int res[5];
ihc_hls_enqueue(&res, &foo, input);
ihc_hls_enqueue_noret(&bar, input);
input = 1;
ihc_hls_enqueue(&res, &foo, input);
ihc_hls_enqueue_noret(&bar, input);
ihc_hls_component_run_all(&foo);
ihc_hls_component_run_all(&bar);
}

```

13.6. Intel HLS Compiler Pro Edition Component Memory Attributes

Use the component memory attributes to control the on-chip component memory architecture of your component.

Table 35. Intel HLS Compiler Pro Edition Component Memory Attributes Summary

Memory Attribute	Description
<code>hls_register</code>	Forces a variable or array to be carried through the pipeline in registers. A register variable can be implemented either exclusively in flip-flops (FFs) or in a mix of FFs and RAM-based FIFOs.
<code>hls_memory</code>	Forces a variable or array to be implemented as embedded memory.
<code>hls_memory_impl</code>	Forces a variable or array to be implemented as embedded memory of a specified type.
<code>hls_singlepump</code>	Specifies that the memory implementing the variable or array must be clocked at the same rate as the component accessing the memory.
<code>hls_doublepump</code>	Specifies that the memory implementing the variable or array must be clocked at twice the rate as the component accessing the memory.
<code>hls_numbanks</code>	Specifies that the memory implementing the variable or array must have a defined number of memory banks.
<code>hls_bankwidth</code>	Specifies that the memory implementing the variable or array must have memory banks of a defined width.
<code>hls_bankbits</code>	Forces the memory system to split into a defined number of memory banks and defines the bits used to select a memory bank.
<code>hls_numports_readonly_writeonly</code>	This memory attribute is deprecated. Use <code>hls_max_replicates</code> instead. Specifies that the memory implementing the variable or array must have a defined number of read and write ports.
<code>hls_simple_dual_port_memory</code>	Specifies that the memory implementing the variable or array should have no port that services both reads and writes.
<code>hls_merge (depthwise)</code>	Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a depth-wise manner.
<code>hls_merge (widthwise)</code>	Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a width-wise manner.
<code>hls_init_on_reset</code>	Forces the static variables inside the component to be initialized when the component <code>reset</code> signal is asserted.
<i>continued...</i>	



Memory Attribute	Description
<code>hls_init_on_powerup</code>	Sets the component memory implementing the static variable to initialize on power-up when the FPGA is programmed.
<code>hls_max_concurrency</code>	Specifies the memory has a defined maximum number of private copies to allow concurrent iterations of a loop at any given time.
<code>hls_max_replicates</code>	Specifies that the memory implementing the variable or array has no more than the specified number of replicates to enable simultaneous reads from the datapath

`hls_register` Memory Attribute

<i>Syntax</i>	<code>hls_register</code>
<i>Constraints</i>	N/A
<i>Default Value</i>	Based on the memory access pattern inferred by the compiler.
<i>Description</i>	Forces a variable or array to be implemented as registers. To learn more, review the following tutorial: <code><quartus_installdir>/hls/examples/tutorials/ best_practices/swap_vs_copy.</code>

`hls_memory` Memory Attribute

<i>Syntax</i>	<code>hls_memory</code>
<i>Constraints</i>	N/A
<i>Default Value</i>	Based on the memory access pattern inferred by the compiler.
<i>Description</i>	Forces a variable or array to be implemented as embedded memory. To learn more, review the following tutorial: <code><quartus_installdir>/hls/tutorials/ component_memories/memory_implementation.</code>

`hls_memory_impl` Memory Attribute

<i>Syntax</i>	<code>hls_memory_impl("type")</code>
<i>Constraints</i>	N/A
<i>Default Value</i>	Based on the memory size and memory access pattern inferred by the compiler.
<i>Description</i>	Forces a variable or array to be implemented as embedded memory of the specified type. The <i>type</i> parameter can be one of the following values:



BLOCK_RAM Implement the variable or array as memory blocks, such as M20K memory blocks.

MLAB Implement the variable or array as memory logic array blocks (MLABs).

To learn more, review the following tutorial:
<quartus_installdir>/hls/tutorials/
component_memories/memory_implementation.

hls_singlepump Memory Attribute

Syntax hls_singlepump

Constraints N/A

Default Value Based on the memory access pattern inferred by the compiler.

Description Specifies that the memory implementing the variable or array must be clocked at the same rate as the component accessing the memory.

To learn more, review the following tutorial:
<quartus_installdir>/hls/examples/QRD.

hls_doublepump Memory Attribute

Syntax hls_doublepump

Constraints N/A

Default Value Based on the memory access pattern inferred by the compiler.

Description Specifies that the memory implementing the variable or array must be clocked at twice the rate of the component accessing the memory.

To learn more, review the following tutorial:
<quartus_installdir>/hls/tutorials/
component_memories/memory_bank_configuration.

hls_numbanks Memory Attribute

Syntax hls_numbanks(*N*)

Constraints This attribute is subject to constraints outlined in [Constraints on Attributes for Memory Banks](#) on page 46.

Default Value Based on the memory access pattern inferred by the compiler.



Description Specifies that the memory implementing the variable or array must have N banks, where N is a power-of-two constant number.

To learn more, review the following tutorial:
<quartus_installdir>/hls/tutorials/
component_memories/memory_geometry.

hls_bankwidth Memory Attribute

Syntax `hls_bankwidth(N)`

Constraints This attribute is subject to constraints outlined in [Constraints on Attributes for Memory Banks](#) on page 46.

Default Value Based on the memory access pattern inferred by the compiler.

Description Specifies that the memory implementing the variable or array must have banks that are N bytes wide, where N is a power-of-two constant number.

To learn more, review the following tutorial:
<quartus_installdir>/hls/tutorials/
component_memories/memory_geometry.

hls_bankbits Memory Attribute

Syntax `hls_bankbits(b_0, b_1, \dots, b_n)`

Constraints This attribute is subject to constraints outlined in [Constraints on Attributes for Memory Banks](#) on page 46.

Default Value Based on the memory access pattern inferred by the compiler.

Description Forces the memory system to split into 2^{n+1} banks, with $\{b_0, b_1, \dots, b_n\}$ forming the bank-select bits.

Important: b_0, b_1, \dots, b_n must be consecutive, positive integers. You can specify the consecutive, positive integers in ascending or descending order.

If you do not specify the `hls_bankwidth(N)` attribute along with this attribute, then b_0, b_1, \dots, b_n are mapped to array index bits 0 to $n-1$ in the memory bank implementation.

To learn more, review the following tutorial:
<quartus_installdir>/hls/tutorials/
component_memories/memory_geometry.

`hls_numports_readonly_writeonly` Memory Attribute

<i>Syntax</i>	<code>hls_numports_readonly_writeonly(<i>M</i>, <i>N</i>)</code>
<i>Constraints</i>	N/A
<i>Default Value</i>	Based on the memory access pattern inferred by the compiler.
<i>Description</i>	<p>This memory attribute is deprecated. Use <code>hls_max_replicates</code> instead.</p> <p>Specifies that the memory implementing the variable or array must have <i>M</i> read ports and <i>N</i> write ports, where <i>M</i> and <i>N</i> are constant numbers greater than zero.</p>

`hls_simple_dual_port_memory` Memory Attribute

<i>Syntax</i>	<code>hls_simple_dual_port_memory</code>
<i>Constraints</i>	N/A
<i>Default Value</i>	N/A
<i>Description</i>	<p>Specifies that the memory implementing the variable or array should have no port that services both reads and writes.</p> <p>To learn more, review the following tutorial: <<code>quartus_installdir</code>>/hls/tutorials/ component_memories/memory_bank_configuration.</p>

`hls_merge (depthwise)` Memory Attribute

<i>Syntax</i>	<code>hls_merge("mem_name", "depth")</code>
<i>Constraints</i>	N/A
<i>Default Value</i>	N/A
<i>Description</i>	<p>Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a depth-wise manner.</p> <p>All variables with same <<code>mem_name</code>> label specified in their <code>hls_merge</code> attribute are merged into the same memory system.</p> <p>To learn more, review the following tutorial: <<code>quartus_installdir</code>>/hls/tutorials/ component_memories/memory_merging.</p>



hls_merge (widthwise) Memory Attribute

<i>Syntax</i>	<code>hls_merge("mem_name", "width")</code>
<i>Constraints</i>	N/A
<i>Default Value</i>	N/A
<i>Description</i>	<p>Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a width-wise manner.</p> <p>All variables with same <code><mem_name></code> label specified in their <code>hls_merge</code> attribute are merged into the same memory system.</p> <p>To learn more, review the following tutorial: <code><quartus_installdir>/hls/tutorials/component_memories/memory_merging.</code></p>

hls_init_on_reset Memory Attribute

<i>Syntax</i>	<code>hls_init_on_reset</code>
<i>Constraints</i>	N/A
<i>Default Value</i>	Default behavior for static variables.
<i>Description</i>	<p>Forces the static variable inside the component to be initialized when the component <code>reset</code> signal is asserted. This requires an additional write port to the component memory implemented and can increase the power-up latency when the component is reset.</p> <p>To learn more, review the following tutorial: <code><quartus_installdir>/hls/examples/tutorials/component_memories/static_var_init.</code></p>

hls_init_on_powerup Memory Attribute

<i>Syntax</i>	<code>hls_init_on_powerup</code>
<i>Constraints</i>	N/A
<i>Default Value</i>	N/A
<i>Description</i>	<p>Sets the component memory implementing the static variable to initialize on power-up when the FPGA is programmed. When the component is reset, the component memory is not reset back to the initialized value of the static.</p>



To learn more, review the following tutorial:
<quartus_installdir>/hls/examples/tutorials/
component_memories/static_var_init.

hls_max_concurrency Memory Attribute

Syntax `hls_max_concurrency(N)`

Constraints N/A

Default Value N/A

Description Specifies that the memory can have a maximum N private copies to allow N concurrent iterations of a loop at any given time, where N is rounded up to the nearest power of 2.

Apply this attribute only when the scope of a variable (through its declaration or access pattern) is limited to a loop. If the loop has the `max_concurrency` pragma applied to it, the number of private copies created is the lesser of the `hls_max_concurrency` memory attribute value and the `max_concurrency` pragma value.

hls_max_replicates Memory Attribute

Syntax `hls_max_replicates(N)`

Constraints N/A

Default Value N/A

Description Specifies that the memory implementing the variable or array has no more than the N replicates to enable simultaneous reads from the datapath.

To learn more, review the following tutorial:
<quartus_installdir>/hls/tutorials/
component_memories/memory_bank_configuration.

13.7. Intel HLS Compiler Pro Edition Loop Pragmas

Use the Intel HLS Compiler loop pragmas to control how the compiler pipelines the loops in your component.

Table 36. Intel HLS Compiler Pro Edition Loop Pragmas Summary

Pragma	Description
<code>disable_loop_pipelining</code>	Prevents compiler from pipelining a loop,
<code>ii</code>	Forces a loop to have a loop initiation interval (II) of a specified value.
<i>continued...</i>	



Pragma	Description
<code>ivdep</code>	Ignores memory dependencies between iterations of this loop.
<code>loop_coalesce</code>	Tries to fuse all loops nested within this loop into a single loop.
<code>max_concurrency</code>	Limits the number of iterations of a loop that can simultaneously execute at any time.
<code>max_interleaving</code>	Controls whether iterations of a pipelined inner loop in a loop nest from one invocation of the inner loop can be interleaved in the component data pipeline with iterations from other invocations of the inner loop.
<code>speculated_iterations</code>	Specifies the number of clock cycles that a loop exit condition can take to compute.
<code>unroll</code>	Unrolls the loop completely or by a number of times.

disable_loop_pipelining Loop Pragma

Syntax `#pragma disable_loop_pipelining`

Description Tells the compiler to not pipeline this loop.

Disable loop pipelining for a loop when the loop-carried dependencies cause the loop iterations to effectively execute sequentially. With loop pipelining disabled, the Intel HLS Compiler can generate a simpler datapath and reduce the FPGA area utilization of your component.

Example:

```
#pragma disable_loop_pipelining
for (int i = 1; i < N; i++) {
    int j = a[i-1];
    // Memory dependency induces a high-latency loop feedback path
    a[i] = foo(j)
}
```

ii Loop Pragma

Syntax `#pragma ii N`

Description Forces the loop to which you apply this pragma to have a loop initiation interval (II) of $\langle N \rangle$, where $\langle N \rangle$ is a positive integer value.

Forcing a loop II value can have an adverse effect on the f_{MAX} of your component because using this pragma to get a lower loop II combines pipeline stages together and creates logic with a long propagation delay.

Using this pragma with a larger loop II inserts more pipeline stages and can give you a better component f_{MAX} value.

Example:

```
#pragma ii 2
for (int i = 0; i < 8; i++) {
    // Loop body
}
```

ivdep Loop Pragma

Syntax `#pragma ivdep safelen(N) array(array_name)`

Description Tells the compiler to ignore memory dependencies between iterations of this loop.

It can accept an optional argument that specifies the name of the array. If `array` is not specified, all component memory dependencies are ignored. If there are loop-carried dependencies, your generated RTL produces incorrect results.

The `safelen` parameter specifies the dependency distance. The dependency distance is the number of iterations between successive load/stores that depend on each other. It is safe to not include `safelen` is only when the dependence distance is infinite (that is, there are no real dependencies).

Example:

```
#pragma ivdep safelen(2)
for (int i = 0; i < 8; i++) {
  // Loop body
}
```

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/best_practices/loop_memory_dependency`.

loop_coalesce Loop Pragma

Syntax `#pragma loop_coalesce N`

Description Tells the compiler to try to fuse all loops nested within this loop into a single loop. This pragma accepts an optional value `N` which indicates the number of levels of loops to coalesce together.

```
#pragma loop_coalesce 2
for (int i = 0; i < 8; i++) {
  for (int j = 0; j < 8; j++) {
    // Loop body
  }
}
```

max_concurrency Loop Pragma

Syntax `#pragma max_concurrency N`

Description This pragma limits the number of iterations of a loop that can simultaneously execute at any time.



This pragma is useful mainly when private copies of are created to improve the throughput of the loop. This is mentioned in the details pane for the loop in the Loop Analysis pane and the Bank view of the Function Memory Viewer of the high level design report (`report.html`).

This can occur only when the scope of a component memory (through its declaration or access pattern) is limited to this loop. Adding this pragma can be used to reduce the area that the loop consumes at the cost of some throughput.

Example:

```
// Without this pragma,  
// multiple private copies  
// of the array "arr"  
#pragma max_concurrency 1  
for (int i = 0; i < 8; i++) {  
    int arr[1024];  
    // Loop body  
}
```

max_interleaving Loop Pragma

Syntax `#pragma max_interleaving <option>`

Description *<option>* This pragma controls whether iterations of a pipelined inner loop in a loop nest from one invocation of the inner loop can be interleaved in the component data pipeline with iterations from other invocations of the inner loop.

By default, the Intel HLS Compiler tries to interleave a number of simultaneous invocations of the inner loop equal to the loop initiation interval (II) of the inner loop. For example, an inner loop with an II of 2 can have iterations from two invocations in the pipeline at a time.

In cases where the interleaving of loop iterations from different loop invocations does not yield a performance benefit, limiting or restricting the amount of interleaving can result in reduced FPGA area utilization.

Supported values for *<option>*:

- 1
The compiler restricts the annotated (inner) loop to be invoked only once per outer loop iteration. That is, all iterations of the inner loop travel the pipeline before the next invocation of the inner loop can occur.
- 0
Use the default interleaving behavior.

Example:

```
// Loop j is pipelined with ii=1  
for (int j = 0; j < M; j++) {  
    int a[N];  
}
```

```
// Loop i is pipelined with ii=2
#pragma max_interleaving 1
for (int i = 1; i < N; i++) {
    a[i] = foo(i)
}
...
}
```

speculated_iterations Loop Pragma

Syntax `speculated_iterations N`

Description This pragma specifies the number of loop iterations to wait before considering a loop exit condition. That is, you estimate that a loop takes at least N loop iterations before the exit condition is met.

If you specify a value that is too low, then the loop II increases to accommodate the iterations required to determine whether the loop exit condition is met.

Example:

```
component int loop_speculate (int N) {
    int m = 0;
    // The exit path has 2 multiplies and
    // compare is most critical in loop feedback path
    #pragma speculated_iterations 2
    while (m*m*m < N) {
        m += 1;
    }
    return m;
}
```

unroll Loop Pragma

Syntax `#pragma unroll N`

Description This pragma unrolls the loop completely or by $\langle N \rangle$ times, where $\langle N \rangle$ is optional and is a positive integer value.

Example:

```
#pragma unroll 8
for (int i = 0; i < 8; i++) {
    // Loop body
}
```

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/best_practices/resource_sharing_filter`.



13.8. Intel HLS Compiler Pro Edition Scope Pragmas

Use the Intel HLS Compiler scope pragmas to influence the rounding of floating-point operations and the ordering of arithmetic operations in your component at finer grain than the `i++` command options.

Table 37. Intel HLS Compiler Pro Edition Scoped Pragmas Summary

Pragma	Description
<code>fp contract</code>	Controls the removal of intermediate rounding and conversion when possible within the code block that this pragma is applied to.
<code>fp reassoc</code>	Controls the relaxing of the order of floating point arithmetic operations within the code block that this pragma is applied to.

`fp contract` Scoped Pragma

Syntax `#pragma clang fp contract(state)`

Description This pragma controls whether the compiler can contract floating-point multiply and add or subtract operations into a single fused multiply-add (FMA), and controls whether the compiler skip intermediate rounding and conversions.

If multiple occurrences of this pragma affect the same scope, the pragma with the narrowest scope takes precedence.

The *state* parameter can be one of the following values:

- `off`
Turns off any permissions to fuse instructions into FMAs.
The effect of the `-ffp-contract=fast i++` command flag is suppressed for instructions within the scope of the pragma.
- `fast`
Allows the fusing of `mult` and `add` instructions into an FMA, but might violate the language standard.
For instructions with the scope of this pragma, the same optimizations as `-ffp-contract=fast i++` command flag are enabled.

`fp reassoc` Scoped Pragma

Syntax `#pragma clang fp reassoc(state)`

Description This pragma controls whether the compiler can relax the order of floating point operations requested by the source code. With some reordering, the compiler can optimize the hardware structure which improves the performance of your component.

If multiple occurrences of this pragma affect the same scope, the pragma with the narrowest scope takes precedence.

The *state* parameter can be one of the following values:

- `on`
Enables the effect of the `-ffp-reassoc i++` command flag for instructions within the scope of the pragma.
- `off`
The effect of the `-ffp-reassoc i++` command flag is suppressed for instructions within the scope of the pragma.

13.9. Intel HLS Compiler Pro Edition Component Attributes

Table 38. Intel HLS Compiler Component Attributes Summary

Attribute	Description
<code>hls_component_ii</code>	Force the component to which you apply this attribute to have a specified component initiation interval (II).
<code>hls_disable_component_pipelining</code>	Prevents the creation of the pipelined component datapath. Multiple invocations of this component now occur sequentially and not simultaneously.
<code>hls_max_concurrency</code>	Request more copies of the component memory so that the component can run multiple invocations in parallel.
<code>hls_scheduler_target_fmax_mhz</code>	Specify the target clock frequency of your component.

`hls_component_ii` Component Attribute

Syntax `hls_component_ii(<N>)`

Description Forces the component to which you apply this attribute to have a component initiation interval (II) of `<N>`, where `<N>` is a positive integer value.

This can have an adverse effect on the f_{MAX} of your component because using this attribute to get a lower II combines pipeline stages together and creates logic with a long propagation delay.

Using this attribute with a larger II inserts more pipeline stages and can give you a better component f_{MAX} value.

`hls_disable_component_pipelining` Component Attribute

Syntax `hls_disable_component_pipelining`

Description Tells the compiler to not create a pipelined datapath for the component. An unpipelined component datapath can save FPGA area utilization in some cases.

Use this attribute when a pipelined datapath does not improve your component throughput or when the component is not invoked repeatedly.



Example

```
#include "HLS/hls.h"

hls_disable_component_pipelining
component void baz ( /* arguments */ ){
    // component code
}
```

hls_max_concurrency Component Attribute

Syntax hls_max_concurrency(<N>)

Description In some cases, the concurrency of a component is limited to 1. This limit occurs when the generated hardware cannot be shared across component invocations. For example, when using component memories for a non-static variable.

You can use this attribute to request more copies of the component memory so that the component can run multiple invocations in parallel.

This attribute can accept any non-negative whole number, including 0.

Value greater than 0 A value greater than 0 indicates how many copies of the component memory to instantiate as well as how many component invocations can be in flight at once.

Value equal to 0 Setting hls_max_concurrency to a value of 0 is useful in cases when there is no component memory but the component still has a poor dynamic loop initiation interval (II) even if you believe your component II should be 1. You can review the II for loops in your component in the high level design report.

To learn more, review the design example:

`<quartus_installdir>/hls/examples/inter_decim_filter.`

Example

```
hls_max_concurrency(2)
component void foo(ihc::stream_in<int> &data_in,
    ihc::stream_out<int> &data_out) {
    int arr[N];
    for (int i = 0; i < N; i++) {
        arr[i] = data_in.read();
    }
    // Operate on the data and modify in place
    for (int i = 0; i < N; i++) {
        data_out.write(arr[i]);
    }
}
```

hls_scheduler_target_fmax_mhz Component Attribute

Syntax hls_scheduler_target_fmax_mhz(<N>)



Description Apply the `hls_scheduler_target_fmax_mhz` component attribute to have the compiler target a specific f_{MAX} value. Specify the target f_{MAX} value in MHz.

The component is not guaranteed to close timing at the specified frequency, and any tasks in a system of tasks use the same clock regardless of having different scheduling targets.

13.10. Intel HLS Compiler Pro Edition Component Default Interfaces

Table 39. Intel HLS Compiler Default Interfaces

Interface	Description
Component invocation interface (component call and return)	The component call is implemented as an interface consisting of the component <code>start</code> and <code>busy</code> conduits. The component return is also implemented as an interface that includes the component <code>done</code> and <code>stall</code> signals.
Scalar parameter interface (passed by value)	Scalar parameters are implemented as input conduits that are synchronized with the component invocation interface.
Pointer parameter interface (passed by reference)	Pointer parameters are implemented as an implicit Avalon Memory-Mapped Master (<code>mm_master</code>) interface with the default parametrization. By default, the base address is treated as a scalar parameter so it is implemented as a conduit that is synchronized to the component invocation interface. A memory mapped interface is also exposed on the component.

13.11. Intel HLS Compiler Pro Edition Component Invocation Interface Control Attributes

Table 40. Intel HLS Compiler Component Invocation Interface Control Attribute Summary

Control Attribute	Description
<code>hls_avalon_streaming_component</code>	This is the default component invocation interface. The component uses <code>start</code> , <code>busy</code> , <code>stall</code> , and <code>done</code> signals for handshaking.
<code>hls_avalon_slave_component</code>	The <code>start</code> , <code>done</code> , and <code>returndata</code> (if applicable) signals are registered in the component slave memory map.
<code>hls_always_run_component</code>	The <code>start</code> signal is tied to 1 internally in the component. There is no <code>done</code> signal output.
<code>hls_stall_free_return</code>	If the downstream component never stalls, the <code>stall</code> signal is removed by internally setting it to 0.

`hls_avalon_streaming_component` Invocation Control Attribute

Description This is the default component invocation interface.



This attribute follows the Avalon-ST protocol for both the function call and the return streams. The component consumes the unstable arguments when the `start` signal is asserted and the `busy` signal is deasserted. The component produces the return data when the `done` signal is asserted.

*Top-Level
Module Ports*

- Function call:
 - `start`
 - `busy`
- Function return:
 - `done`
 - `stall`

Example

```
component hls_avalon_streaming_component void foo(/*component arguments*/)
```

hls_avalon_slave_component Invocation Control Attribute

Description

The `start`, `done`, and `returndata` (if applicable) signals are registered in the component slave memory map.

These component must take either slave, stream, or stable arguments. If you do not specify these types of arguments, the compiler generates an error message when you compile this component.

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/interfaces/mm_slaves`.

*Top-Level
Module Ports*

- Avalon-MM slave interface
- `irq_done` signal

Example

```
component hls_avalon_slave_component void foo(/*component arguments*/)
```

hls_always_run_component Invocation Control Attribute

Description

The `start` signal is tied to 1 internally in the component. There is no `done` signal output. The control logic is optimized away when Intel Quartus Prime compiles the generated RTL for your FPGA.

Use this protocol when the component datapath relies only on explicit streams for data input and output.

IP verification does not support components with this component invocation protocol.

Top-Level Module Ports None

Example

```
component hls_always_run_component void foo(/*component arguments*/)
```

hls_stall_free_return Invocation Control Attribute

Description If the downstream component never stalls, the `stall` signal is removed by internally setting it to 0.

This feature can be used with the `hls_avalon_streaming_component`, `hls_avalon_slave_component`, and `hls_always_run_component` arguments. This attribute can be used to specify that the downstream component is stall free.

Top-Level Module Ports N/A

Example

```
component hls_stall_free_return int dut(int a, int b)
{ return a * b; }
```

13.12. Intel HLS Compiler Pro Edition Component Macros

Table 41. Intel HLS Compiler Component Macros Summary

Macro	Description
<code>hls_conduit_argument</code>	Implement the argument as an input conduit that is synchronous to the component call (start and busy).
<code>hls_avalon_slave_register_argument</code>	Implement the argument as a register that can be read from and written to over an Avalon-MM slave interface.
<code>hls_avalon_slave_memory_argument</code>	Implement the argument, in on-chip memory blocks, which can be read from or written to over a dedicated slave interface.
<code>hls_stable_argument</code>	A stable argument is an argument that does not change while there is live data in the component (that is, between pipelined function invocations).

hls_conduit_argument Component Macro

Syntax `hls_conduit_argument`

Description This is the default interface for scalar arguments.

The compiler implements the argument as an input conduit that is synchronous to the component's call (start and busy).

Example

```
component void foo(
  hls_conduit_argument int b)
```




hls_avalon_slave_register_argument Component Macro

Syntax hls_avalon_slave_register_argument

Description The compiler implements the argument as a register that can be read from and written to over an Avalon-MM slave interface. The argument will be read into the component's pipeline, similar to the conduit implementation. The implementation is synchronous to the start and busy interface.

Changes to the value of this argument made by the component data path will not be reflected on this register.

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/interfaces/mm_slaves`.

Example

```
component void foo(  
    hls_avalon_slave_register_argument int b)
```

hls_avalon_slave_memory_argument Component Macro

Syntax hls_avalon_slave_memory_argument(*N*)

Description The compiler implements the argument, where *N* specifies the size of the memory in bytes, in on-chip memory blocks, which can be read from or written to over a dedicated slave interface. The generated memory has the same architectural optimizations as all other internal component memories (such as banking or coalescing).

If the compiler performs static coalescing optimizations, the slave interface data width is the coalesced width. This attribute applies only to a pointer argument.

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/interfaces/mm_slaves`.

Example

```
component void foo(  
    hls_avalon_slave_memory_argument(128*sizeof(int)) int *a)
```

hls_stable_argument Component Macro

Syntax hls_stable_argument

Description A stable argument is an argument that does not change while there is live data in the component (that is, between pipelined function invocations).

Changing a stable argument during component execution results in undefined behavior; each use of the stable argument might be the old value or the new value, but with no guarantee of consistency. The same variable in the same invocation can appear with multiple values.

Using stable arguments, where appropriate, might save a significant number of registers in a design.

Stable arguments can be used with conduits, mm_master interfaces, and slave_registers.

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/interfaces/stable_arguments`.

Example

```
component int dut(
  hls_stable_argument int a,
  hls_stable_argument int b) {
  return a * b;}
```

13.13. Systems of Tasks API

Systems of tasks are available only for Intel HLS Compiler Pro Edition.

Table 42. Intel HLS Compiler System of Tasks Summary

Function	Description
<code>ihc::launch</code>	Marks a function as an Intel HLS Compiler task for hardware generation, and launches the task function asynchronously.
<code>ihc::collect</code>	Synchronizes the completion of the specified task function in the component.
<code>ihc::stream</code>	Allows streaming communication between different task functions.
<code>ihc::launch_always_run</code>	Launches a task function at component power-on or reset and continuously executes the function.

`ihc::launch` Function

Syntax `ihc::launch(<function>, <function_argument_list>)`

Where the function parameters are defined as follows:

- *<function>*
The name of the function that you are calling as an Intel HLS Compiler task in your component.
If the task function is a templated function, wrap the function handle in a pair of parenthesis. For example:
`ihc::launch((foo<int>)), arg_a, arg_b);`
- *<function_argument_list>*
The list of arguments to pass to the task function.
This list must match the arguments (in names and types) that the task function expects.

Description The `ihc::launch` API function identifies a function as Intel HLS Compiler task for hardware generation. Calling this function starts the task function asynchronously.



If the task function cannot accept a new thread, the `ihc::launch` function can block the function that calls the `ihc::launch` function.

The list of arguments that supply the `ihc::launch` API function must match (in names and types) the list of arguments expected by the task function.

`ihc::collect` Function

Syntax `ihc::collect(function)`

Where the function parameters are defined as follows:

- `<function>`

The name of the Intel HLS Compiler task function to synchronize the completion of.

If the task function is a templated function, wrap the function handle in a pair of parenthesis. For example:

```
ihc::collect((foo<int>);
```

Description The `ihc::collect` API function synchronizes the completion of the specified task function in the component.

For a non-void task function, the `ihc::collect` API function collects the result from the specified task function.

For a void task function, the `ihc::collect` API function synchronizes against the `done` signal of the task function.

The number of `ihc::collect` calls for a task function must match the number of `ihc::launch` calls for the same task function to flush all of the calls to the task.

Special Case: If you do not use `ihc::collect` at all, the compiler optimizes and ties-off the return stream of the task to be stall free and ignores any data on the return stream. Other streaming interfaces can still back-pressure the task function. Additionally, the caller might finish before the task function.

`ihc::launch_always_run` Function

Syntax `ihc::launch_always_run<function>()`

Where the parameters are defined as follows:

- `function`

The name of the function that you are calling as a continuously-executing Intel HLS Compiler task in your component.

Description Use the `ihc::launch_always_run` API function to continuously execute a task function, much like an invoking a component with the `hls_always_run_component` invocation interface argument.

The task launches at the power-on or the reset of the component instead of at a specific point in the datapath.

The task function that you provide to this API must match this prototype:

```
void function(void)
```

Your task function must have no function arguments and no return value. You should communicate with your task function through global streams or by using compile-time constant template parameters.

Example The following example shows a simple use of the `ihc::launch_always_run` function.

```
ihc::stream<int> in_stream, out_stream;

template <ihc::stream<int> &inStream,
         ihc::stream<int> &outStream>

void my_task()
{
    int x = inStream.read();
    x *= 2;
    outStream.write(x);
}

component void foo()
{
    ihc::launch_always_run<my_task<in_stream, out_stream>>();
}
```

Intel HLS Compiler System of Tasks Code Example

The following code example illustrates how you can use the systems of tasks API.

```
int mul(int a, int b)
{
    return a * b;
}

Template<typename T>
T add(T a, T b)
{
    return a + b;
}

component int foo(int a, int b)
{
    ihc::launch(mul, a, b);
    ihc::launch((add<int>), a, b);
    int prod = ihc::collect(mul);
    int sum = ihc::collect(add<int>);
    return sum + prod;
}
```



Related Information

Intel HLS Compiler Pro Edition Component Invocation Interface Control Attributes on page 126

13.13.1. `ihc::stream` Class

Table 43. Intel HLS Compiler Systems of Tasks Streaming Interface Template Summary

Template Object or Parameter	Description
<code>ihc::stream</code>	Streaming interface to the component or task function.
<code>ihc::buffer</code>	Specifies the capacity (in words) of the FIFO buffer on the input data that associates with the stream.
<code>ihc::usesPackets</code>	Exposes the <code>startofpacket</code> and <code>endofpacket</code> sideband signals on the stream interface.

`ihc::stream` Template Object

Syntax `ihc::stream<datatype, template arguments>`

Valid Values Any valid C++ datatype

Default Value N/A

Description Streaming interface to the component or task.
The width of the stream data bus is equal to a width of `sizeof(datatype)`.

`ihc::buffer` Template Parameter

Syntax `ihc::buffer<value>`

Valid Values Non-negative integer value.

Default Value 0

Description The capacity, in words, of the FIFO buffer on the input data that associates with the stream.

`ihc::usesPackets` Template Parameter

Syntax `ihc::usesPackets<value>`

Valid Values true or false

Default Value false



Description Exposes the `startofpacket` and `endofpacket` sideband signals on the stream interface, which can be accessed by the packet based reads/writes.

Intel HLS Compiler System of Tasks Streaming Interface `stream` Function APIs

Table 44. Intel HLS Compiler Streaming Input Interface `stream` Function APIs

Function API	Description
<code>T read()</code>	Blocking read call to be used from within the component or task
<code>T read(bool& sop, bool& eop)</code>	Available only if <code>usesPackets<true></code> is set. Blocking read with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals.
<code>T tryRead(bool &success)</code>	Non-blocking read call to be used from within the component or task. The <code>success</code> bool is set to true if the read was valid.
<code>T tryRead(bool& success, bool& sop, bool& eop)</code>	Available only if <code>usesPackets<true></code> is set. Non-blocking read with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals.
<code>void write(T data)</code>	Blocking write call from the component or task.
<code>void write(T data, bool sop, bool eop)</code>	Available only if <code>usesPackets<true></code> is set. Blocking write with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals.
<code>bool tryWrite(T data)</code>	Non-blocking write call from the component or task. The return value represents whether the write was successful.
<code>bool tryWrite(T data, bool sop, bool eop)</code>	Available only if <code>usesPackets<true></code> is set. Non-blocking write with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals. The return value represents whether the write was successful.

13.14. Intel HLS Compiler Pro Edition Streaming Input Interfaces

Use the `stream_in` object and template arguments to explicitly declare Avalon Streaming (ST) input interfaces. You can also use the `stream_in` Function APIs.

Table 45. Intel HLS Compiler Pro Edition Streaming Input Interface Template Summary

Template Object or Parameter	Description
<code>ihc::stream_in</code>	Streaming input interface to the component.
<code>ihc::buffer</code>	Specifies the capacity (in words) of the FIFO buffer on the input data that associates with the stream.
<code>ihc::readyLatency</code>	Specifies the number of cycles between when the <code>ready</code> signal is deasserted and when the input stream can no longer accept new inputs.
<code>ihc::bitsPerSymbol</code>	Describes how the data is broken into symbols on the data bus.
<code>ihc::firstSymbolInHighOrderBits</code>	Specifies whether the data symbols in the stream are in big endian order.
<i>continued...</i>	



Template Object or Parameter	Description
<code>ihc::usesPackets</code>	Exposes the <code>startofpacket</code> and <code>endofpacket</code> sideband signals on the stream interface.
<code>ihc::usesEmpty</code>	Exposes the <code>empty</code> out-of-band signal on the stream interface.
<code>ihc::usesValid</code>	Controls whether a <code>valid</code> signal is present on the stream interface.

`ihc::stream_in` Template Object

Syntax `ihc::stream_in<datatype, template parameters>`

Valid Values Any valid C++ datatype

Default Value N/A

Description Streaming input interface to the component.

The width of the stream data bus is equal to a width of `sizeof(datatype)`.

The testbench must populate this buffer (stream) fully before the component can start to read from the buffer.

To learn more, review the following tutorials:

- `<quartus_installdir>/hls/examples/tutorials/interfaces/explicit_streams_buffer`
- `<quartus_installdir>/hls/examples/tutorials/interfaces/explicit_streams_packets_empty`
- `<quartus_installdir>/hls/examples/tutorials/interfaces/explicit_streams_packet_ready_valid`
- `<quartus_installdir>/hls/examples/tutorials/interfaces/explicit_streams_ready_latency`
- `<quartus_installdir>/hls/examples/tutorials/interfaces/multiple_stream_call_sites`

`ihc::buffer` Template Parameter

Syntax `ihc::buffer<value>`

Valid Values Non-negative integer value.

Default Value 0

Description The capacity, in words, of the FIFO buffer on the input data that associates with the stream. The buffer has latency. It immediately consumes data, but this data is not immediately available to the logic in the component.

If you use the `tryRead()` function to access this stream and the stream read is scheduled within the first cycles of operation, the first (or more) calls to the `tryRead()` function might return `false` in co-simulation (and therefore in hardware).

Review the function viewer in the Graph Viewer of the High Level Design Reports to see when operations are scheduled in your component. If you see this behavior, use the blocking `read()` function to ensure consistency between emulation and co-simulation.

This parameter is available only on input streams.

`ihc::readyLatency` Template Parameter

Syntax `ihc::readyLatency<value>`

Valid Values Non-negative integer value between 0-8.

Default Value 0

Description The number of cycles between when the `ready` signal is deasserted and when the input stream can no longer accept new inputs.

`ihc::bitsPerSymbol` Template Parameter

Syntax `ihc::bitsPerSymbol<value>`

Valid Values A positive integer value that evenly divides by the data type size.

Default Value Datatype size

Description Describes how the data is broken into symbols on the data bus.

Data is broken down according to how you set the `ihc::firstSymbolInHighOrderBits` declaration. By default, data is broken down in little endian order.

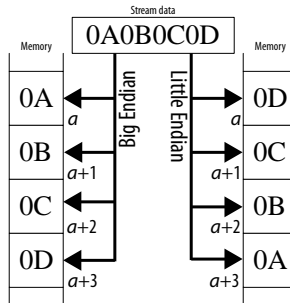
`ihc::firstSymbolInHighOrderBits` Template Parameter

Syntax `ihc::firstSymbolInHighOrderBits<value>`

Valid Values `true` or `false`

Default Value `false`

Description Specifies whether the data symbols in the stream are in big endian order.



ihc::usesPackets Template Parameter

Syntax ihc::usesPackets<value>

Valid Values true or false

Default Value false

Description Exposes the startofpacket and endofpacket sideband signals on the stream interface, which can be accessed by the packet based reads/writes.

ihc::usesEmpty Template Parameter

Syntax ihc::usesEmpty<value>

Valid Values true or false

Default Value false

Description Exposes the empty out-of-band signal on the stream interface.

Use this declaration only with streams that read more than one data symbol per clock cycle.

The empty signal indicates the number of symbols on the data bus that do not represent valid data during the final stream read of a packet.

You can control whether the empty symbols are in the low-order bits or high-order bits with the ihc::firstSymbolInHighOrderBits declaration.

ihc::usesValid Template Parameter

Syntax ihc::usesValid<value>

Valid Values true or false



Default Value true

Description Controls whether a valid signal is present on the stream interface. If false, the upstream source must provide valid data on every cycle that ready is asserted.

This is equivalent to changing the stream read calls to tryRead and assuming that success is always true.

If set to false, buffer and readyLatency must be 0.

Intel HLS Compiler Pro Edition Streaming Input Interface stream_in Function APIs

Table 46. Intel HLS Compiler Pro Edition Streaming Input Interface stream_in Function APIs

Function API	Description
T read()	Blocking read call to be used from within the component
T read(bool& sop, bool& eop)	Available only if usesPackets<true> is set. Blocking read with out-of-band startofpacket and endofpacket signals.
T read(bool& sop, bool& eop, int& empty)	Available only if usesPackets<true> and usesEmpty<true> are set. Blocking read with out-of-band startofpacket, endofpacket, and empty signals.
T tryRead(bool &success)	Non-blocking read call to be used from within the component. The success bool is set to true if the read was valid. That is, the Avalon-ST valid signal was high when the component tried to read from the stream. The emulation model of tryRead() is not cycle-accurate, so the behavior of tryRead() might differ between emulation and co-simulation.
T tryRead(bool& success, bool& sop, bool& eop)	Available only if usesPackets<true> is set. Non-blocking read with out-of-band startofpacket and endofpacket signals.
T tryRead(bool& success, bool& sop, bool& eop, int& empty)	Available only if usesPackets<true> and usesEmpty<true> are set. Non-blocking read with out-of-band startofpacket, endofpacket, and emptysignals.
void write(T data)	Blocking write call to be used from the testbench to populate the FIFO to be sent to the component.
void write(T data, bool sop, bool eop)	Available only if usesPackets<true> is set. Blocking write call with out-of-band startofpacket and endofpacket signals.
void write(T data, bool sop, bool eop, int empty)	Available only if usesPackets<true> and usesEmpty<true> are set. Blocking write call with out-of-band startofpacket, endofpacket, and empty signals.



Intel HLS Compiler Streaming Input Interfaces Code Example

The following code example illustrates both `stream_in` declarations and `stream_in` function APIs.

```
// Blocking read
void foo (ihc::stream_in<int> &a) {
    int x = a.read();
}

// Non-blocking read
void foo_nb (ihc::stream_in<int> &a) {
    bool success = false;
    int x = a.tryRead(success);

    if (success) {
        // x is valid
    }
}

int main() {
    ihc::stream_in<int> a;
    ihc::stream_in<int> b;
    for (int i = 0; i < 10; i++) {
        a.write(i);
        b.write(i);
    }
    foo(a);
    foo_nb(b);
}
```

13.15. Intel HLS Compiler Pro Edition Streaming Output Interfaces

Use the `stream_out` object and template arguments to explicitly declare Avalon Streaming (ST) output interfaces. You can also use the `stream_out` Function APIs.

Table 47. Intel HLS Compiler Pro Edition Streaming Output Interface Template Summary

Template Object or Parameter	Description
<code>ihc::stream_out</code>	Streaming output interface from the component.
<code>ihc::readylatency</code>	Specifies the number of cycles between when the <code>ready</code> signal is deasserted and when the input stream can no longer accept new inputs.
<code>ihc::bitsPerSymbol</code>	Describes how the data is broken into symbols on the data bus.
<code>ihc::firstSymbolInHighOrderBits</code>	Specifies whether the data symbols in the stream are in big endian order.
<code>ihc::usesPackets</code>	Exposes the <code>startofpacket</code> and <code>endofpacket</code> sideband signals on the stream interface.
<code>ihc::usesEmpty</code>	Exposes the <code>empty</code> out-of-band signal on the stream interface.
<code>ihc::usesReady</code>	Controls whether a <code>ready</code> signal is present.

`ihc::stream_out` Template Object

Syntax `ihc::stream_out<datatype, template parameter>`

Valid Values Any valid POD (plain old data) C++ datatype.

Default Value N/A

Description Streaming output interface from the component. The testbench can read from this buffer once the component returns.

To learn more, review the following tutorials:

- `<quartus_installdir>/hls/examples/tutorials/interfaces/ explicit_streams_buffer`
- `<quartus_installdir>/hls/examples/tutorials/interfaces/ explicit_streams_packets_empty`
- `<quartus_installdir>/hls/examples/tutorials/interfaces/ explicit_streams_packet_ready_valid`
- `<quartus_installdir>/hls/examples/tutorials/interfaces/ explicit_streams_ready_latency`
- `<quartus_installdir>/hls/examples/tutorials/interfaces/ mulitple_stream_call_sites`

`ihc::readylatency` Template Parameter

Syntax `ihc::readylatency<value>`

Valid Values Non-negative integer value (between 0-8)

Default Value 0

Description The number of cycles between when the `ready` signal is deasserted and when the sink can no longer accept new inputs.

Conceptually, you can view this parameter as an almost ready latency on the input FIFO buffer for the data that associates with the stream.

`ihc::bitsPerSymbol` Template Parameter

Syntax `ihc::bitsPerSymbol<value>`

Valid Values Positive integer value that evenly divides the data type size.

Default Value Datatype size

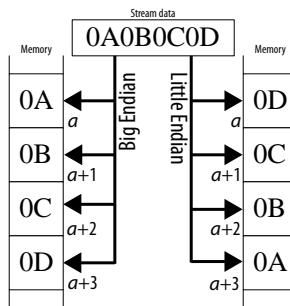
Description Describes how the data is broken into symbols on the data bus.

Data is broken down according to how you set the `ihc::firstSymbolInHighOrderBits` declaration. By default, data is broken down in little endian order.



ihc::firstSymbolInHighOrderBits Template Parameter

- Syntax** `ihc::firstSymbolInHighOrderBits<value>`
- Valid Values** true or false
- Default Value** false
- Description** Specifies whether the data symbols in the stream are in big endian order.



ihc::usesPackets Template Parameter

- Syntax** `ihc::usesPackets<value>`
- Valid Values** true or false
- Default Value** false
- Description** Exposes the startofpacket and endofpacket sideband signals on the stream interface, which can be accessed by the packet based reads/writes.

ihc::usesEmpty Template Parameter

- Syntax** `ihc::usesEmpty<value>`
- Valid Values** true or false
- Default Value** false
- Description** Exposes the empty out-of-band signal on the stream interface.
- Use this declaration only with streams that write more than one data symbol per clock cycle.
- The empty signal indicates the number of symbols on the data bus that do not represent valid data during the final stream write of a packet.



You can control whether the empty symbols are in the low-order bits or high-order bits with the `ihc::firstSymbolInHighOrderBits` declaration.

ihc::usesReady Template Parameter

Syntax `ihc::usesReady<value>`

Valid Values `true` or `false`

Default Value `true`

Description Controls whether a ready signal is present. If `false`, the downstream sink must be able to accept data on every cycle that valid is asserted. This is equivalent to changing the stream read calls to `tryWrite` and assuming that `success` is always `true`.

If set to `false`, `readyLatency` must be 0.

Intel HLS Compiler Pro Edition Streaming Output Interface `stream_out` Function APIs

Table 48. Intel HLS Compiler Pro Edition Streaming Output Interface `stream_out` Function APIs

Function API	Description
<code>void write(T data)</code>	Blocking write call from the component
<code>void write(T data, bool sop, bool eop)</code>	Available only if <code>usesPackets<true></code> is set. Blocking write with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals.
<code>void write(T data, bool sop, bool eop, int empty)</code>	Available only if <code>usesPackets<true></code> and <code>usesEmpty<true></code> are set. Blocking write with out-of-band <code>startofpacket</code> , <code>endofpacket</code> , and <code>empty</code> signals.
<code>bool tryWrite(T data)</code>	Non-blocking write call from the component. The return value represents whether the write was successful.
<code>bool tryWrite(T data, bool sop, bool eop)</code>	Available only if <code>usesPackets<true></code> is set. Non-blocking write with out-of-band <code>startofpacket</code> and <code>endofpacket</code> signals. The return value represents whether the write was successful. That is, the downstream interface was pulling the <code>ready</code> signal high while the HLS component tried to write to the stream.
<code>bool tryWrite(T data, bool sop, bool eop, int empty)</code>	Available only if <code>usesPackets<true></code> and <code>usesEmpty<true></code> are set. Non-blocking write with out-of-band <code>startofpacket</code> , <code>endofpacket</code> , and <code>empty</code> signals. The return value represents whether the write was successful.

continued...



Function API	Description
T read()	Blocking read call to be used from the testbench to read back the data from the component
T read(bool &sop, bool &eop)	Available only if usesPackets<true> is set. Blocking read call to be used from the testbench to read back the data from the component with out-of-band startofpacket and endofpacket signals.
T read(bool &sop, bool &eop, int &empty)	Available only if usesPackets<true> and usesEmpty<true> are set. Blocking read call to be used from the testbench to read back the data from the component with out-of-band startofpacket, endofpacket, and empty signals.

Intel HLS Compiler Streaming Output Interfaces Code Example

The following code example illustrates both `stream_out` declarations and `stream_out` function APIs.

```
// Blocking write
void foo (ihc::stream_out<int> &a) {
    static int count = 0;
    for(int idx = 0; idx < 5; idx++){
        a.write(count++); // Blocking write
    }
}

// Non-blocking write
void foo_nb (ihc::stream_out<int> &a) {
    static int count = 0;
    for(int idx = 0; idx < 5; idx++){
        bool success = a.tryWrite(count++); // Non-blocking write
        if (success) {
            // write was successful
        }
    }
}

int main() {
    ihc::stream_out<int> a;
    foo(a); // or foo_nb(a);

    // copy output to an array
    int outputData[5];
    for (int i = 0; i < 5; i++) {
        outputData[idx] = a.read();
    }
}
```

13.16. Intel HLS Compiler Pro Edition Memory-Mapped Interfaces

Use the `mm_master` object and template arguments to explicitly declare Avalon Memory-Mapped (MM) Master interfaces for your component.

Table 49. Intel HLS Compiler Pro Edition Memory-Mapped Interfaces Summary

Template Object or Parameter	Description
<code>ihc::mm_master</code>	The underlying pointer type.
<code>ihc::dwidth</code>	The width of the memory-mapped data bus in bits
<i>continued...</i>	

Template Object or Parameter	Description
<code>ihc::awidth</code>	The width of the memory-mapped address bus in bits.
<code>ihc::aspace</code>	The address space of the interface that associates with the master.
<code>ihc::latency</code>	The guaranteed latency from when a read command exits the component when the external memory returns valid read data.
<code>ihc::maxburst</code>	The maximum number of data transfers that can associate with a read or write transaction.
<code>ihc::align</code>	The alignment of the base pointer address in bytes.
<code>ihc::readwrite_mode</code>	The port direction of the interface.
<code>ihc::waitrequest</code>	Adds the <code>waitrequest</code> signal that is asserted by the slave when it is unable to respond to a read or write request.
<code>getInterfaceAtIndex</code>	This testbench function is used to index into an <code>mm_master</code> object.

`ihc::mm_master` Template Object

Syntax `ihc::mm_master<datatype, template parameter>`

Valid values Any valid C++ datatype

Default Value Default interface for pointer arguments.

Description The underlying pointer type. Pointer arithmetic performed on the master object conforms to this type. Dereferences of the master results in a load-store site with a width of `sizeof(datatype)`. The default alignment is aligned to the size of the datatype.

You can use multiple template arguments in any combination as long the combination of arguments describes a valid hardware configuration.

Example:

```
component int dut(
  ihc::mm_master<int,
    ihc::aspace<2>, ihc::latency<3>,
    ihc::awidth<10>, ihc::dwidth<32>
  > &a)
```

To learn more, review the following tutorials:

- `<quartus_installdir>/hls/examples/tutorials/interfaces/pointer_mm_master`
- `<quartus_installdir>/hls/examples/tutorials/interfaces/mm_master_testbench_operators`

`ihc::dwidth` Template Parameter

Syntax `ihc::dwidth<value>`

Valid Values 8, 16, 32, 64, 128, 256, 512, or 1024



Default Value 64
Description The width of the memory-mapped data bus in bits.

ihc::awidth Template Parameter

Syntax `ihc::awidth<value>`
Valid Values Integer value in the range 1 – 64
Default Value 64
Description The width of the memory-mapped address bus in bits.

This value affects only the width of the Avalon MM Master interface. The size of the conduit of the base address pointer is always set to 64-bits.

ihc::aspace Template Parameter

Syntax `ihc::aspace<value>`
Valid Values Integer value greater than 0.
Default Value 1
Description The address space of the interface that associates with the master. Each unique *value* results in a separate Avalon MM Master interface on your component. All masters with the same address space are arbitrated within the component to a single interface. As such, these masters must share the same template parameters that describe the interface.

ihc::latency Template Parameter

Syntax `ihc::latency<value>`
Valid Values Non-negative integer value
Default Value 1
Description The guaranteed latency from when a read command exits the component when the external memory returns valid read data. If this latency is variable (such as when accessing DRAM), set it to 0.

ihc::maxburst Template Parameter

Syntax `ihc::maxburst<value>`



<i>Valid Values</i>	Integer value in the range 1 – 1024
<i>Default Value</i>	1
<i>Description</i>	<p>The maximum number of data transfers that can associate with a read or write transaction. This value controls the width of the burstcount signal.</p> <p>For fixed latency interfaces, this value must be set to 1.</p> <p>For more details, review information about burst signals and the burstcount signal role in "Avalon Memory-Mapped Interface Signal Roles" in <i>Avalon Interface Specifications</i>.</p>

ihc::align Template Parameter

<i>Syntax</i>	<code>ihc::align<value></code>
<i>Valid Values</i>	Integer value greater than the alignment of the datatype
<i>Default Value</i>	Alignment of the datatype
<i>Description</i>	<p>The alignment of the base pointer address in bytes.</p> <p>The Intel HLS Compiler uses this information to determine how many simultaneous loads and stores this pointer can permit.</p> <p>For example, if you have a bus with 4 32-bit integers on it, you should use <code>ihc::dwidth<128></code> (bits) and <code>ihc::align<16></code> (bytes). This means that up to 16 contiguous bytes (or 4 32-bit integers) can be loaded or stored as a coalesced memory word per clock cycle.</p> <p><i>Important:</i> The caller is responsible for aligning the data to the set value for the align argument; otherwise, functional failures might occur.</p>

ihc::readwrite_mode Template Parameter

<i>Syntax</i>	<code>ihc::readwrite_mode<value></code>
<i>Valid Values</i>	readwrite, readonly, or writeonly
<i>Default Value</i>	readwrite
<i>Description</i>	The port direction of the interface. Only the relevant Avalon master signals are generated.

ihc::waitrequest Template Parameter

<i>Syntax</i>	<code>ihc::waitrequest<value></code>
---------------	--



Valid Values true or false

Default Value false

Description Adds the `waitrequest` signal that is asserted by the slave when it is unable to respond to a read or write request. For more information about the `waitrequest` signal, see "[Avalon Memory-Mapped Interface Signal Roles](#)" in *Avalon Interface Specifications*.

getInterfaceAtIndex Testbench Function

Syntax `getInterfaceAtIndex(int index)`

Description This testbench function is used to index into an `mm_master` object. It can be useful when iterating over an array and invoking a component on different indicies of the array. This function is supported only in the testbench.

Code Example

```
int main() {
// .....
for(int idx = 0; idx < N; idx++) {
    dut(src_mm.getInterfaceAtIndex(idx));
}
// .....
}
```

13.17. Intel HLS Compiler Pro Edition Load-Store Unit Control

For variable-latency Avalon Memory-Mapped (MM) Master interfaces (`ihc::latency<0>`), you can control the type of load-store unit (LSU) with the `ihc::lsu` template object and the corresponding `load()` and `store()` functions.

Table 50. Intel HLS Compiler Pro Edition Load-Store Unit Control Summary

Template Object/Parameter/Function	Description
<code>ihc::lsu</code>	The underlying LSU class template object
<code>ihc::style</code>	Specifies the type of load-store unit.
<code>ihc::static_coalescing</code>	Explicitly allows or prevents static coalescing of a load/store operation with other load/store operations.
<code>load</code>	Loads data from memory into the LSU.
<code>store</code>	Stores data from the LSU into memory.

ihc::lsu Template Object

Syntax `ihc::lsu<template arguments>`

Valid Values N/A.

Default Value N/A.

Description The underlying LSU class object.

To learn more, review the following tutorial:
<quartus_installdir>/hls/examples/tutorials/
best_practices/lsu_control

ihc::style Template Parameter

Syntax `ihc::style<LSU_type>`

Valid Values *LSU_type* can be one of the following values:

- BURST_COALESCED
- PIPELINED

Default Value BURST_COALESCED

Description Specifies the type of load-store unit to create.

A burst-coalesced LSU buffers requests until the largest possible burst can be made.

A pipelined LSU submits requests as they are received.

ihc::static_coalescing Template Parameter

Syntax `ihc::static_coalescing<value>`

Valid Values true or false

Default Value true

Description Specifies whether to allow or prevent static coalescing of the load/store operation with other load/store operations.

load Function

Syntax `load(<memory_location>)`

Parameters The <memory_location> argument specifies the memory location to load data into the LSU from.

Return Type Object of same type as the base type of the argument specified for <memory_location>.

Description The load function loads data from a memory location specified by the <memory_location> argument and returns the data that the argument points to.



store Function

Syntax `store(<memory_location>, <value_to_store>)`

Parameters The `<memory_location>` argument specifies the memory location to store data coming from the LSU.

The `<value_to_store>` argument is the value from the LSU to store in memory. The type is the same as the pointer base type.

Return Type None.

Description The store function stores data in the LSU to a memory location specified by the `<memory_location>` argument.

13.18. Intel HLS Compiler Pro Edition Arbitrary Precision Data Types

Table 51. Arbitrary Precision Data Types Supported by the Intel HLS Compiler Pro Edition

Data Type	Intel Header File	Description
ac_int	HLS/ac_int.h	Arbitrary-width integer support To learn more, review the following tutorials: <ul style="list-style-type: none"> <code><quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_int_basic_ops</code> <code><quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_int_overflow</code> <code><quartus_installdir>/hls/examples/tutorials/best_practices/struct_interfaces</code>
ac_fixed	HLS/ac_fixed.h	Arbitrary-precision fixed-point number support To learn more, review the tutorial: <code><quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_constructor</code>
	HLS/ac_fixed_math.h	Support for some nonstandard math functions for arbitrary-precision fixed-point data types To learn more, review the tutorial: <code><quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_math_library</code>
ac_complex	HLS/ac_complex.h	Complex number support
hls_float	HLS/hls_float.h	Arbitrary-precision floating-point number support
	HLS/hls_float_math.h	Support for commonly used exponential, logarithmic, power, and trigonometric functions. To learn more, review the following tutorials: <ul style="list-style-type: none"> <code><quartus_installdir>/hls/examples/tutorials/hls_float/1_reduced_doubl</code> <code><quartus_installdir>/hls/examples/tutorials/hls_float/2_explicit_arithmetic</code> <code><quartus_installdir>/hls/examples/tutorials/hls_float/3_conversions</code>

Table 52. Intel HLS Compiler ac_int Debugging Tools Summary

Tool	Description
DEBUG_AC_INT_WARNING	Emits a warning for each detected overflow.
DEBUG_AC_INT_ERROR	Emits a message for the first overflow that is detected and then exits the component with an error.

DEBUG_AC_INT_WARNING ac_int Debugging Tool

Macro Syntax `#define DEBUG_AC_INT_WARNING`

If you use this macro, declare it in your code before you declare `#include HLS/ac_int.h`.

i++ Command Option Syntax `-D DEBUG_AC_INT_WARNING`

Description Enables runtime tracking of `ac_int` data types during x86 emulation (the `-march=x86-64` option, which the default option, of the `i++` command).

This tool uses additional resources for tracking the overflow and empty constructors, and emits a warning for each detected overflow.

To learn more, review the tutorial:
`<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_int_overflow.`

DEBUG_AC_INT_ERROR ac_int Debugging Tool

Macro Syntax `#define DEBUG_AC_INT_ERROR`

If you use this macro, declare it in your code before you declare `#include HLS/ac_int.h`.

i++ Command Option Syntax `-D DEBUG_AC_INT_ERROR`

Description Enables runtime tracking of `ac_int` data types during x86 emulation of your component (the `-march=x86-64` option, which the default option, of the `i++` command).

This tool uses additional resources to track the overflow and empty constructors, and emits a message for the first overflow that is detected and then exits the component with an error.

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_int_overflow`

A. Advanced Math Source Code Libraries

The Intel HLS Compiler Pro Edition comes with templated source code libraries that help speed the development of your components by providing you with FPGA-optimized code for some commonly-used algorithms.

The Intel HLS Compiler provides the following libraries:

Library	Description	Header file
Random number generator	Generate random integers or floating point numbers that follow a uniform distribution, or random floating point numbers that follow a Gaussian distribution	HLS/rand_lib.h
Matrix multiplication	Multiply two 2-D matrices.	HLS/matrix_mult.h

A.1. Random Number Generator Library

The random number generator source code library provided with the Intel HLS Compiler Pro Edition gives you FPGA-optimized random number generator template classes that you can add to your component without needing to write your own.

The Random Number Generator Library and Cryptography

The use of these pseudo-random number generator (PRNG) algorithms are not recommended for cryptographic purposes. The PRNGs included in this library are not cryptographically-secure pseudo-random number generators (CSPRNGs) and should not be used for cryptography. CSPRNG algorithms are designed so that no polynomial-time algorithm (PTA) can compute or predict the next bit in the pseudo-random sequence, nor is there a PTA that can predict past values of the CSPRNG; these algorithms do not achieve this purpose. Additionally, these algorithms have not been reviewed nor are they recommended for use as a PRNG component of a CSPRNG, even if the input values are from a non-deterministic entropy source with an appropriate entropy extractor.

Table 53. Properties of Values That Can Be Generated by the Intel HLS Compiler Random Number Generator Library

Value distribution	Value type	Value range	Generation method
Uniform	Integer	$[-2^{31}, 2^{31}-1]$	Tausworthe Generator
	Floating point	$[0, 1)$ (non-inclusive)	Tausworthe Generator
Gaussian	Floating point	$[0, 1)$	Central limit theorem (CLT) (Default)
			Box-Muller

Header File

To include the random number generator library in your component, add the following line to your component:

```
#include "HLS/rand_lib.h"
```

The header file is self-documented. You can review the header file to learn how to use the random number generator library in your component.

Random Number Object Declarations

Declare random number objects in your components as follows. In all cases, specifying **<seed_value>** is optional.

- Uniform distribution integer random number

```
static RNG_Uniform<int> <object_name>(<seed_value>)
```

- Uniform distribution floating point random number

```
static RNG_Uniform<float> <object_name>(<seed_value>)
```

- Gaussian distribution floating point random number (CLT method)

```
static RNG_Gaussian<float> <object_name>(<seed_value>)
```

or

```
static RNG_Gaussian<float, ihc::GAUSSIAN_CLT> <object_name>(<seed_value>)
```

- Gaussian distribution floating point random number (Box-Muller method)

```
static RNG_Gaussian<float, ihc::GAUSSIAN_BOX_MULLER>  
<object_name>(<seed_value>)
```

A.2. Matrix Multiplication Library

The matrix multiplication source code library provided with the Intel HLS Compiler Pro Edition gives you an FPGA-optimized templated source code library to perform matrix multiplication of two matrices stored in a 2-D array.

When you use the matrix multiplication library, you can affect the number of DSP blocks and RAM blocks by controlling the dot product vector size and the number of matrix elements read at one time. Increasing the dot product vector size can achieve better latency, but at the cost of using more DSP blocks and other FPGA resources.

Header File

To include the matrix multiplication library in your component, add the following line to your component:

```
#include "HLS/matrix_mult.h"
```

The header file is self-documented. You can review the header file to learn how to use the matrix multiplication library in your component.



Template Arguments

The matrix multiplication library multiplies two 2-D matrices, A and B. The resulting product is returned in a third matrix, C. The matrix multiplication library has the following template arguments:

<i>T</i>	The data type of the matrix elements (For example, <code>int</code> , <code>float</code> , <code>long</code> , <code>double</code>).
<i>t_rowsA</i>	The number of rows in matrix A.
<i>t_colsA</i>	The number of columns in matrix A. This value also the number of rows in matrix B.
<i>t_colsB</i>	The number of columns in matrix B.
<i>DOT_VEC_SIZE</i>	The number of DSP blocks to use in a single computation. This value must be a factor of <i>t_colsA</i> . You can achieve better component latency by increasing this value. However, you use more FPGA area to achieve this. Keeping this value low lowers your FPGA resource usage, but increases the latency.
<i>BLOCK_SIZE</i>	The number of elements to read at one time from matrix A. The default value of <i>BLOCK_SIZE</i> is the value of <i>DOT_VEC_SIZE</i> . You can reduce this number if the bandwidth needed by matrix A is lower than the value of <i>DOT_VEC_SIZE</i> , but it must remain a factor of <i>DOT_VEC_SIZE</i> .
<i>RUNNING_SUM_MULT_L</i>	This parameter can be adjusted to try and improve the f_{MAX} of a component that uses this library. Review the header file for a detailed description of this argument and its effects.

B. Supported Math Functions

The Intel HLS Compiler has built-in support for generating efficient IP out of standard math functions present in the `math.h` C header file. The compiler also has support for some math functions that are not supported by the `math.h` header file, and these functions are provided in `extendedmath.h` C header file.

To use the Intel implementation of `math.h` for Intel FPGAs, include `HLS/math.h` in your function by adding the following line:

```
#include "HLS/math.h"
```

To use the nonstandard math functions that are optimized for Intel FPGAs, include `HLS/extendedmath.h` in your function by adding the following line:

```
#include "HLS/extendedmath.h"
```

The `extendedmath.h` header is compatible only with Intel HLS Compiler. It is not compatible with GCC or Microsoft Visual Studio.

If your component uses arbitrary precision fixed-point datatypes provided in the `ac_fixed.h` header, you use some of the datatypes with some math functions by including the following line:

```
#include "HLS/ac_fixed_math.h"
```

To see examples of how to use the math functions provided by these header files, review the following tutorial: `<quartus_installdir>/hls/examples/tutorials/best_practices/single_vs_double_precision_math`.

If your component uses the `hls_float` arbitrary precision floating point data type, add the following line to add support for math functions:

```
#include "HLS/hls_float_math.h"
```

B.1. Math Functions Provided by the `math.h` Header File

The Intel HLS Compiler Pro Edition supports a subset of functions that are present in your native compiler through the `HLS/math.h` header file.

For each `math.h` function listed below, "●" indicates that the HLS compiler supports the function; "X" indicates that the function is not supported.

The math functions supported on Linux operating systems might differ from the math functions supported on Windows operating systems. Review the comments in the `HLS/math.h` header file to see which math functions are supported on the different operating systems.



Table 54. Trigonometric Functions

Trigonometric Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
cos	cosf	•
sin	sinf	•
tan	tanf	•
acos	acosf	•
asin	asinf	•
atan	atanf	•
atan2	atan2f	•

Table 55. Hyperbolic Functions

Hyperbolic Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
cosh	coshf	•
sinh	sinhf	•
tanh	tanhf	•
acosh	acoshf	•
asinh	asinhf	•
atanh	atanhf	•

Table 56. Exponential and Logarithmic Functions

Exponential or Logarithmic Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
exp	expf	•
frexp	frexpf	•
ldexp	ldexpf	•
log	logf	•
log10	log10f	•
modf	modff	•
exp2	exp2f	•
exp10 (Linux only)	exp10f (Linux only) ^(*)	•
expm1	expm1f	•
ilogb	ilogbf	•
log1p	log1pf	•
log2	log2f	•

continued...

^(*) For Windows, support for this function is in the `extendedmath.h` header file



Exponential or Logarithmic Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
logb	logbf	•
scalbn	scalbnf	X
scalbln	scalblnf	X

Table 57. Power Functions

Power Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
pow	powf	•
sqrt	sqrtf	•
cbrt	cbrtf	•
hypot	hypotf	•

Table 58. Error and Gamma Functions

Error or Gamma Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
erf	erff	•
erfc	erfcf	•
tgamma	tgammaf	•
lgamma	lgammaf	•
lgamma_r (Linux only) ^(*)	lgamma_rf (Linux only) ^(*)	•

Table 59. Rounding and Remainder Functions

Rounding or Remainder Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
ceil	ceilf	•
floor	floorf	•
fmod	fmodf	•
trunc	truncf	•
round	roundf	•
lround	lroundf	X
llround	llroundf	X
rint	rintf	•
lrint	lrintf	X
llrint	llrintf	X
nearbyint	nearbyintf	X
remainder	remainderf	•
remquo	remquof	•



Table 60. Floating-Point Manipulation Functions

Floating-Point Manipulation Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
copysign	copysignf	•
nan	nanf	X
nextafter	nextafterf	•
nexttoward	nexttowardf	X

Table 61. Minimum, Maximum, and Difference Functions

Minimum, Maximum, or Difference Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
fdim	fdim	•
fmax	fmax	•
fmin	fmin	•

Table 62. Other Functions

Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
fabs	fabsf	•
fma	fmaf	•

Table 63. Classification Macros

Classification Macro		Supported ?
Double-precision floating point function	Single-precision floating point function	
fpclassify (Linux only)	fpclassifyf (Linux only)	•
isfinite	isfinitef	•
isinf	isinf	•
isnan	isnanf	•
isnormal (Linux only)	isnormalf (Linux only)	•
signbit (Linux only)	signbitf (Linux only)	•

Table 64. Comparison Macros

Comparison Macro		Supported ?
Double-precision floating point function	Single-precision floating point function	
isgreater	isgreaterf	X
isgreaterequal	isgreaterequalf	X
isless	islessf	X

continued...

Comparison Macro		Supported ?
Double-precision floating point function	Single-precision floating point function	
islessequal	islessequalf	X
islessgreater	islessgreaterf	X
isunordered (Linux only)	isunorderedf (Linux only)	•

B.2. Math Functions Provided by the `extendedmath.h` Header File

The Intel HLS Compiler Pro Edition supports an additional subset of math functions through the `HLS/extendedmath.h` header file.

For each `extendedmath.h` function listed below, "•" indicates that the Intel HLS Compiler Pro Edition supports the function; "X" indicates that the function is not supported.

The math functions supported on Linux operating systems might differ from the math functions supported on Windows operating systems. Review the comments in the `HLS/extendedmath.h` header file to see which math functions are supported on the different operating systems.

Table 65. Extended Math Functions

Extended Math Functions		Supported ?
Double-precision floating point function	Single-precision floating point function	
<code>sincos</code>	<code>sincosf</code>	•
<code>acospi</code>	<code>acospif</code>	•
<code>asinpi</code>	<code>asinpif</code>	•
<code>atanpi</code>	<code>atanpif</code>	•
<code>cospi</code>	<code>cospif</code>	•
<code>sinpi</code>	<code>sinpif</code>	•
<code>tanpi</code>	<code>tanpif</code>	•
<code>pown</code>	<code>pownf</code>	•
<code>powr</code>	<code>powrf</code>	•
<code>rsqrt</code>	<code>rsqrtf</code>	•

Table 66. Exponential and Logarithmic Functions

Exponential or Logarithmic Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
<code>exp10</code> (Windows only)	<code>exp10f</code> (Windows only) (*)	•

(*) For Linux, support for this function is in the `math.h` header file



Table 67. Error and Gamma Functions

Error or Gamma Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
lgamma_r (Windows only) ^(*)	lgamma_rf (Windows only) ^(*)	•

Table 68. Minimum, Maximum, and Difference Functions

Minimum, Maximum, or Difference Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
maxmag	maxmagf	•
minmag	minmagf	•

Table 69. Other Functions

Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
fract	fractf	•
mad	madf	•
oclnan	oclnanf	•
rootn	rootnf	•

Table 70. Classification Macros

Classification Macro		Supported ?
Double-precision floating point function	Single-precision floating point function	
isordered	isorderedf	•

In addition, the `HLS/extendedmath.h` header file supports the following versions of the `popcount` function:

Table 71. Popcount function

Data type	Function
Unsigned char	popcountc
Unsigned short	popcountS
Unsigned int	popcount
Unsigned long	popcountl
Unsigned long long	popcountll

To see an example of how to use the math functions provided by the `extendedmath.h` header file and how to override a math function in the header file so that you can compile your design with GCC or Microsoft Visual Studio, review the following example design: `<quartus_installdir>/hls/examples/QRD`.

B.3. Math Functions Provided by the `ac_fixed_math.h` Header File

Adding the `ac_fixed_math.h` header file adds support for the following arbitrary precision fixed-point (`ac_fixed`) datatype functions:

- `sqrt_fixed`
- `reciprocal_fixed`
- `reciprocal_sqrt_fixed`
- `sin_fixed`
- `cos_fixed`
- `sincos_fixed`
- `sinpi_fixed`
- `cospi_fixed`
- `sincospi_fixed`
- `log_fixed`
- `exp_fixed`

For details about inputs type restrictions, input value limits, and output type propagation rules, review the comments in the `ac_fixed_math.h` header file.

B.4. Math Functions Provided by the `hls_float.h` Header File

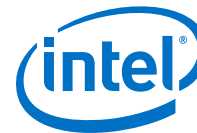
Adding the `hls_float.h` header file adds support for the following arbitrary precision floating point (`hls_float.h`) data type functions:

- Arithmetic operators: `+`, `-`, `*`, `/`
- Arithmetic assignment operators: `=`, `+=`, `-=`, `*=`, `/=`
- Comparison operators: `>`, `<`, `==`, `!=`, `>=`, `<=`
- Unary operators: `+`, `-`, `abs()`
- Explicit functions: `add(a, b)`, `sub(a, b)`, `mul(a, b)`, `div(a, b)`

B.5. Math Functions Provided by the `hls_float_math.h` Header File

Adding the `hls_float_math.h` header file adds support for the following arbitrary precision floating point (`hls_float_math.h`) data type functions:

- square root: `ihc_sqrt`
- cube root: `ihc_cbrt`
- reciprocal (inverse): `ihc_recip`
- reciprocal (inverse) square root: `ihc_rsqr`
- hypotenuse: `ihc_hypot`
- e^x : `ihc_exp`
- e^{x-1} : `ihc_expm1`
- 2^x : `ihc_exp2`



- 10^x : `ihc_exp10`
 - $\ln(x)$: `ihc_log`
 - $\log_2(x)$: `ihc_log2`
 - $\log_{10}(x)$: `ihc_log10`
 - $\ln(1+x)$: `ihc_log1p`
 - x^y : `ihc_pow`
Both x and y are `hls_float` data types
 - x^n : `ihc_pown`
 x is `hls_float` data type, n is `ac_int` data type
 - x^y : `ihc_powr`
Both x and y are `hls_float` data types. With the restriction that $x \geq 0 \rightarrow x^y$, $x \geq 0$, undefined behavior if $x < 0$
 - `sin`: `ihc_sin`
 - `sinpi`: `ihc_sinpi`
 - `cos`: `ihc_cos`
 - `cospi`: `ihc_cospi`
 - `sincos`: `ihc_sincos`
 - `sincospi`: `ihc_sincospi`
 - `arcsin`: `ihc_asin`
 - `arcsinpi`: `ihc_asinpi`
 - `arccos`: `ihc_acos`
 - `arccospi`: `ihc_acospi`
 - `arctan`: `ihc_atan`
 - `arctanpi`: `ihc_atanpi`
 - `arctan(x/y)`: `ihc_atan2`
Both x and y are `hls_float` data types
- For details about inputs type restrictions, input value limits, and output type propagation rules, review the comments in the `hls_float_math.h` header file.



C. Intel HLS Compiler Pro Edition Reference Manual Archives

Intel HLS Compiler Version	Title
19.4	Intel HLS Compiler Pro Edition Reference Manual
19.3	Intel HLS Compiler Reference Manual
19.2	Intel HLS Compiler Reference Manual
19.1	Intel HLS Compiler Reference Manual
18.1.1	Intel HLS Compiler Reference Manual
18.1	Intel HLS Compiler Reference Manual
18.0	Intel HLS Compiler Reference Manual
17.1.1	Intel HLS Compiler Reference Manual
17.1	Intel HLS Compiler Reference Manual

Intel Corporation. All rights reserved. Agilix, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.




ISO
9001:2015
Registered

D. Document Revision History of the Intel HLS Compiler Pro Edition Reference Manual

Document Version	Intel HLS Compiler Pro Edition Version	Changes
2020.02.10	19.4	<ul style="list-style-type: none"> Corrected the syntax of the <code>ihc::launch_always_run</code> system of tasks API function in Systems of Tasks API on page 130.
2020.01.27	19.4	<ul style="list-style-type: none"> Corrected the spelling of the <code>-ffp-contract=fast</code> and <code>-ffp-reassoc</code> command options in the following sections: <ul style="list-style-type: none"> Intel HLS Compiler Pro Edition Command Options on page 7 Intel HLS Compiler Pro Edition i++ Command-Line Arguments on page 104 Intel HLS Compiler Pro Edition Scope Pragmas on page 123
2019.12.16	19.4	<ul style="list-style-type: none"> Removed information about Intel HLS Compiler Standard Edition. For reference information for the Intel HLS Compiler Standard Edition, see Intel HLS Compiler Standard Edition Reference Manual. Added <code>ihc::launch_always_run</code> to Systems of Tasks API on page 130. Added descriptions of <code>--daz</code> and <code>--rounding</code> command options to Intel HLS Compiler Pro Edition Command Options on page 7 and Intel HLS Compiler Pro Edition i++ Command-Line Arguments on page 104 Updated Operators and Return Types Supported by the hls_float Data Type on page 69 with addition details about type conversion to and from the <code>hls_float</code> data type. Added Creating an Object Library on page 84. Updated Object Manifest File Syntax on page 92 with information about <code>resetn</code> signal handling. Renamed Intel HLS Compiler Pro Edition Quick Reference to Intel HLS Compiler Pro Edition Reference Summary.

Document Revision History for Intel HLS Compiler Reference Manual

Previous versions of the *Intel HLS Compiler Reference Manual* contained information for both Intel HLS Compiler Standard Edition and Intel HLS Compiler Pro Edition.

Document Version	Intel Quartus Prime Version	Changes
2019.09.30	19.3	<ul style="list-style-type: none">  Expanded and reorganized information about HLS libraries into a new chapter that starts with Object Libraries on page 83.  Added information about arbitrary precision floating point number support to Declaring hls_float Data Types on page 67.  Added Intel HLS Compiler Pro Edition Scope Pragmas on page 123.

continued...



Document Version	Intel Quartus Prime Version	Changes
		<ul style="list-style-type: none"> • PRO For variable-latency Avalon Memory-Mapped (MM) Master interfaces, added information about load-store unit control to Intel HLS Compiler Pro Edition Load-Store Unit Control on page 147. • PRO Added the <code>--ffp-reassoc</code> and <code>--ffp-contract=fast</code> options to Intel HLS Compiler Pro Edition i++ Command-Line Arguments on page 104 • Revised Component Memories (Memory Attributes) on page 41 (formerly <i>Local Variables in Components</i>). • PRO Added information about the <code>hls_max_replicates</code> memory attribute to the following sections: <ul style="list-style-type: none"> — Component Memories (Memory Attributes) on page 41 — Intel HLS Compiler Pro Edition Component Memory Attributes on page 112 • PRO Deprecated the <code>hls_numports_readonly_writeonly</code> memory attribute throughout this document. Use <code>hls_max_replicates</code> instead. • PRO Added information about the <code>max_interleaving</code> loop pragma to the following sections: <ul style="list-style-type: none"> — Loop Interleaving Control (max_interleaving Pragma) on page 57 — Intel HLS Compiler Pro Edition Loop Pragmas on page 118 • PRO Added information about the <code>hls_fpga_reg</code> function to Advanced Hardware Synthesis Controls on page 102. • PRO Removed the restriction that task functions cannot have pointer or reference arguments from Task Functions on page 75. • In Intel HLS Compiler Pro Edition Component Memory Attributes on page 112, revised the description of the default value of the <code>hls_bankbits</code> memory attribute. • In Intel HLS Compiler Pro Edition Component Memory Attributes on page 112, removed references to the <code>bank_bits</code> tutorial. This tutorial has been removed.
2019.09.10	19.2	<ul style="list-style-type: none"> • Corrected typo in the description of the <code>-c</code> option in Intel HLS Compiler Pro Edition Command Options on page 7. The sentence that began, "When you later compile the <code>.o</code> file..." has been corrected to say, "When you later link the <code>.o</code> file".
2019.07.01	19.2	<ul style="list-style-type: none"> • PRO Added information about datapath pipelining control to the following sections: <ul style="list-style-type: none"> — Loop Pipelining Control (disable_loop_pipelining Pragma) on page 56 — Intel HLS Compiler Pro Edition Loop Pragmas on page 118 — Component Pipelining Control (hls_disable_component_pipelining Attribute) on page 59 — Intel HLS Compiler Pro Edition Component Attributes on page 124 • Revised and update the following topics about supported math functions: <ul style="list-style-type: none"> — Math Functions Provided by the math.h Header File on page 154 — Math Functions Provided by the extendedmath.h Header File on page 158

continued...



Document Version	Intel Quartus Prime Version	Changes
2019.06.04	19.1	<ul style="list-style-type: none"> PRO In Slave Memories on page 36, clarified the use of memory attributes for slave memories. In Component Memories (Memory Attributes) on page 41, clarified memory attributes support in Intel HLS Compiler Pro Edition and Intel HLS Compiler Standard Edition.
2019.05.03	19.1	<ul style="list-style-type: none"> PRO Added information about the <code>ihc_hls_set_component_wait_cycle</code> testbench API function to the following sections: <ul style="list-style-type: none"> System of Tasks Simulation on page 81 Intel HLS Compiler Pro Edition Simulation API (Testbench Only) on page 110 Updated diagrams in Task Functions on page 75. Updated diagram in Intel HLS Compiler Pipeline Approach on page 13. Updated diagram in Creating Objects From RTL Code on page 87. Updated diagram in Integration of an RTL Module into the HLS Pipeline on page 89.
2019.04.01	19.1	<ul style="list-style-type: none"> PRO Added information about developing your system with HLS tasks in Systems of Tasks on page 75. PRO Added information about templated and overloaded functions in Templated and Overloaded Functions on page 17. PRO Added information about arbitrary precision complex number (<code>ac_complex</code>) support to Arbitrary Precision Math Support on page 61. Updated Compiler Interoperability on page 11 with details about how to use GCC and Microsoft Visual Studio to compile your component. Added information about the compiler pipeline approach in Intel HLS Compiler Pipeline Approach on page 13. In Intel HLS Compiler Pro Edition Command Options on page 7, corrected <code>--gcc-toolchain</code> option syntax. In Intel HLS Compiler Pro Edition Command Options on page 7, updated the description of the <code>--quartus-compile</code> to indicate that your component is not expected to close timing when you compile your component with this option. Updated the following sections with information about the <code>--hyper-optimized-handshaking</code> option of the <code>i++</code> command: <ul style="list-style-type: none"> Intel HLS Compiler Pro Edition Command Options on page 7 Intel HLS Compiler Pro Edition i++ Command-Line Arguments on page 104 Updated Loop-Carried Dependencies (ivdep Pragma) on page 50 to indicate that arrays specified by the <code>ivdep</code> loop pragma can now be a reference to an <code>mm_master</code> object. Revised and reorganized Intel High Level Synthesis Compiler Pro Edition Reference Summary on page 104. In Declaring ac_int Data Types on page 63, revised the advice for initializing an <code>ac_int</code> variable to a value larger than 64 bits. To initialize this size of <code>ac_int</code> variable, use the <code>bit_fill</code> or <code>bit_fill_hex</code> utility functions.
2019.01.03	18.1.1	<ul style="list-style-type: none"> Fixed typos in table headings in Compiler-Defined Preprocessor Macros on page 19.
2018.12.24	18.1.1	<ul style="list-style-type: none"> Removed information about the "HLS/iostream" header file. The function provided by this header file is replaced by using the standard C++ <code>iostream</code> header and the <code>HLS_SYNTHESIS</code> macro. Added description of the <code>HLS_SYNTHESIS</code> macro to C and C++ Libraries on page 15.

continued...



Document Version	Intel Quartus Prime Version	Changes
2018.12.24	18.1	<ul style="list-style-type: none"> Updated Slave Interfaces on page 33 and Quick Reference with information about slave memory reads and writes that come from outside of the component. Added information about conduit creation and address spaces to Avalon Memory-Mapped Master Interfaces on page 25.
2018.09.24	18.1	<ul style="list-style-type: none"> PRO The Intel HLS Compiler has a new front end. For a summary of the changes introduced by this new front end, see <i>Improved Intel HLS Compiler Front End</i> in the Intel HLS Compiler Version 18.1 Release Notes. PRO The <code>--promote-integers</code> flag and the <code>best_practices/integer_promotion</code> tutorial are no longer supported in Pro Edition because integer promotion is now done by default. The flag and tutorial are still supported in Standard Edition. Components invoked with the <code>hls_avalon_slave_component</code> argument must take slave or stable arguments. If the component arguments are not slave or stable arguments, compiling the component generates an error message. The description of the <code>hls_avalon_slave_component</code> argument in Component Invocation Interface Control Attributes on page 37 and Quick Reference now reflects that requirement. In Loops in Components on page 48, clarified the pragma statements that apply to loops must immediately precede the loop that the pragma applies to. In Declaring ac_int Data Types on page 63, added initialization requirement for <code>ac_int</code> variables larger than 64 bits. You must use <code>ac::init_array</code> constructors to initialize <code>ac_int</code> variables larger than 64 bits. In Static Variables on page 46, removed the restriction on applying memory attributes to file-scoped static variables. Both file-scoped and function-scoped static variables can have memory attributes applied to them.
2018.07.08	18.0	<ul style="list-style-type: none"> In Static Variables on page 46, highlighted paragraph that says that memory attributes applied to static variables work only if the static variable is declared within the component function. In Control and Status Register (CSR) Slave on page 34, corrected a typo. The sentence "You do not need to use the <code>hls_avalon_slave_component</code> attribute to use the <code>hls_avalon_slave_component</code> attribute" was corrected to say "You do not need to use the <code>hls_avalon_slave_component</code> attribute to use the <code>hls_avalon_slave_register_argument</code> attribute".
2018.05.07	18.0	<ul style="list-style-type: none"> Starting with Intel Quartus Prime Version 18.0, the features and devices supported by the Intel HLS Compiler depend on what edition of Intel Quartus Prime you have. Intel HLS Compiler publications now use icons to indicate content and features that apply only to a specific edition as follows: <ul style="list-style-type: none"> PRO Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Pro Edition. STD Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Standard Edition. PRO Corrected the code example in Intel HLS Compiler Streaming Input Interfaces Code Example. The corrected line is <code>int x = a.tryRead(success); (was int x = a.tryRead(&success);</code>.

continued...



Document Version	Intel Quartus Prime Version	Changes
		<ul style="list-style-type: none"> PRO Added <code><quartus_installdir>/hls/examples/tutorials/interfaces/ explicit_streams_packets_empty</code> to list of tutorials in Table 45 on page 134 and Quick Reference. PRO Added <code>ihc::firstSymbolInHighOrderBits</code> and <code>ihc::usesEmpty</code> to the list of stream interface declarations in Table 45 on page 134 and Quick Reference. Also, revised the description of the <code>ihc::bitsPerSymbol</code> declaration to include the effect of the <code>ihc::firstSymbolInHighOrderBits</code> declaration. STD Added a footnote to the <code>-march MAX10</code> option in Command Options about a prerequisite required before you synthesize your component IP for Intel MAX[®] 10 devices. Added new topic AC Data Types and Native Compilers on page 67 describing use of reference AC datatype headers with the Intel HLS Compiler. Advanced Math Source Code Libraries on page 151 added to document Intel HLS Compiler libraries. The following Intel HLS Compiler libraries were added: <ul style="list-style-type: none"> PRO Random Number Generator Library on page 151 PRO Matrix Multiplication Library on page 152
2017.12.22	17.1.1	<ul style="list-style-type: none"> Updated <code>hls_avalon_slave_memory_argument(N)</code> description in Slave Memories on page 36 to include the description that the parameter value <i>N</i> is the size of the memory in bytes. Updated Table 19 on page 65 and Table 52 on page 150 to indicate that the <code>ac_int</code> debug macros have the following restrictions: <ul style="list-style-type: none"> You must declare the macros in your code before you declare <code>#include HLS/ac_int.h</code>. The <code>ac_int</code> debugging tools work only for x86 emulation of your component. Updated <code>-march "<FPGA_family>"</code> options in Intel HLS Compiler Pro Edition Command Options on page 7 to include FPGA family options without a space. Revised the description of the <code>ihc::align</code> argument in ihc::align Template Parameter on page 146 in Quick Reference. The same information also appears in Avalon Memory-Mapped Master Interfaces on page 25.
2017.11.06	17.1	<ul style="list-style-type: none"> Updated Intel HLS Compiler Pro Edition Command Options on page 7 as follows: <ul style="list-style-type: none"> Revised description of <code>-c i++</code> command option. Added descriptions of the <code>--x86-only</code> and <code>--fpga-only i++</code> command options. Updated Supported Math Functions on page 154 as follows: <ul style="list-style-type: none"> Noted that the <code>HLS/extendedmath.h</code> header file is supported only by the Intel HLS Compiler, not by the GCC or MSVC compilers. Added <code>popcount</code> to the list functions supported by the <code>HLS/extendedmath.h</code> header file. Expanded list of functions provided by <code>HLS/extendedmath.h</code> to explicitly list double-precision and single-precision floating point versions of the functions. Added a list of <code>popcount</code> function variations available for different data types. Updated Arbitrary Precision Math Support on page 61 to include restriction that the Intel arbitrary precision header files cannot be compiled with GCC.

continued...



Document Version	Intel Quartus Prime Version	Changes
		<ul style="list-style-type: none"> Added the <code>ihc::readwrite_mode</code> Avalon-MM interface to Avalon Memory-Mapped Master Interfaces on page 25 and Quick Reference. Added the <code>ihc::waitrequest</code> Avalon-MM interface to Avalon Memory-Mapped Master Interfaces on page 25 and Quick Reference. Added the <code>hls_stall_free_return</code> macro and <code>stall_free_return</code> attribute to Unstable and Stable Component Parameters on page 38 and Quick Reference. Reorganized the overall structure of the book, breaking up chapter 1 into smaller chapters and changing the order of the chapters. Updated mentions of the HLS or i++ installation directory to use the Intel Quartus Prime Design Suite installation directory as the starting point. Moved the following content to <i>Intel High Level Synthesis Compiler Best Practices Guide</i>: <ul style="list-style-type: none"> Moved "Avoid Pointer Aliasing" section to "Avoid Pointer Aliasing".
2017.06.23	—	<ul style="list-style-type: none"> Updated Static Variables on page 46 to add information about static variable initialization and how to control it. Minor changes and corrections.
2017.06.09	—	<ul style="list-style-type: none"> Revised Declaring ac_int Data Types on page 63 for changes in how to include <code>ac_int.h</code>. Revised Arbitrary Precision Math Support on page 61 to clarify support for Algorithmic C datatypes. Removed all mentions of <code>--device</code> compiler option. This option has been replaced by the changed function of the <code>-march</code> compiler option. See Table 3 on page 7 for details about the changed function of the <code>-march</code> compiler option. Updated the generated C header file for the component <code>mycomp_xyz</code> in Control and Status Register (CSR) Slave on page 34. Added information about structs in component interfaces to Component Interfaces on page 20. Revised C and C++ Libraries on page 15 with updates to <code>iostream</code> behavior. Added information about math functions supported by <code>extendedmath.h</code> header file to Supported Math Functions on page 154.
2017.02.03	—	<ul style="list-style-type: none"> In <i>Scalar Parameters and Avalon Streaming Interfaces</i>, updated information in the <i>Available Scalar Parameters for Avalon-ST Interfaces</i> table. In <i>Pointer Parameters, Reference Parameters, and Avalon Memory-Mapped Master Interfaces</i>, updated information in the <i>Available Template Arguments for Configuration of the Avalon-MM Interface</i> table. Added new information to <i>Global Variables</i> about area usage and optimizing for global constants, pointers, and variables.
2016.11.30	—	<ul style="list-style-type: none"> In <i>HLS Compiler Command Options</i>, modified the table <i>Command Options that Customize Compilation</i> in the following manner: <ul style="list-style-type: none"> Removed the <code>--rtl-only</code> command option and its description because it is no longer in use. Added the <code>--simulator <name></code> command option and its description. Remove the <code>-g</code> command option because the HLS compiler now generates debug information in reports by default for both Windows and Linux. In addition, debug data is available by default in final binaries for Linux. In <i>Pointer Parameters, Reference Parameters, and Avalon Memory-Mapped Master Interfaces</i>, added information on the <code>altera::align<value></code> template argument in the table.

continued...



Document Version	Intel Quartus Prime Version	Changes
		<ul style="list-style-type: none"> • Added the topics <i>Memory-Mapped Test Bench Constructor</i> and <i>Implicit and Explicit Examples of Creating a Memory-Mapped Master Test Bench</i>. • In <i>Usage Examples of Component Invocation Protocol Macros</i>, replaced component invocation protocol attributes in the code examples with their corresponding macros. • Added the line <code>#include "HLS/hls.h"</code> to the code snippets in the following sections: <ul style="list-style-type: none"> — <i>Usage Examples of Interface Synthesis Macros</i> — <i>Usage Examples of Component Invocation Protocol Macros</i> • Added the topic <i>Arbitrary Precision Integer Support</i> to introduce the <code>ac_int</code> datatype and the Intel-provided <code>ac_int.h</code> header file. Included the following subtopics: <ul style="list-style-type: none"> — <i>Defining the ac_int Datatype in Your Component for Arbitrary Precision Integer Support</i> — <i>Important Usage Information on the ac_int Datatype</i> • Updated the content in <i>Area Minimization and Control of On-Chip Memory Architecture</i>: <ul style="list-style-type: none"> — Replaced the <code>numreadports(n)</code> and <code>numwriteports(n)</code> entries the <i>Attributes for Controlling On-Chip Memory Architecture</i> table with a single <code>numports_readonly_writeonly(m,n)</code> entry. — Added information on the <code>hls_simple_dual_port_memory</code> macro. — Added information on the <code>hls_merge ("label", "direction")</code> and the <code>hls_bankbits(b0, b1, ..., bn)</code> attributes. • Added example use cases for the <code>hls_merge("label", "direction")</code> and the <code>hls_bankbits(b0, b1, ..., bn)</code> attributes. • Added the topic <i>Relationship between hls_bankbits Specifications and Memory Address Bits</i> to explain the derivation of a memory address in the presence of the <code>hls_bankbits</code> and <code>hls_bankwidth</code> attributes.
2016.09.12	—	Initial release.