# Contents

1. Intel® HLS Compiler Reference Manual................................. 5

2. Compiler............................................................................... 6
   2.1. Intel HLS Compiler Command Options............................. 6
   2.2. Compiler Interoperability.................................................... 10
   2.3. Intel HLS Compiler Pipeline Approach............................. 12

3. C Language and Library Support............................................. 14
   3.1. Supported C and C++ Subset for Component Synthesis........ 14
   3.2. C and C++ Libraries......................................................... 14
   3.3. Templated and Overloaded Functions................................. 16
       3.3.1. Templated Functions..................................................... 16
       3.3.2. Overloaded Functions.................................................. 17
       3.3.3. Function Name Mapping............................................... 17
   3.4. Compiler-Defined Preprocessor Macros............................... 18
   3.5. Arbitrary Precision Math Support......................................... 18
       3.5.1. Declaring `ac_int` Datatypes in Your Component.......... 19
       3.5.2. Integer Promotion and `ac_int` Datatypes.................... 21
       3.5.3. Debugging Your Use of the `ac_int` Datatype.................. 22
       3.5.4. Declaring `ac_fixed` Datatypes in Your Component........ 22
       3.5.5. Declaring `ac_complex` Datatypes in Your Component..... 23
       3.5.6. AC Datatypes and Native Compilers.............................. 23

4. Component Interfaces............................................................ 25
   4.1. Component Invocation Interface.......................................... 25
       4.1.1. Scalar Parameters....................................................... 26
       4.1.2. Pointer and Reference Parameters............................... 26
       4.1.3. Interface Definition Example: Component with Both Scalar and Pointer Arguments.................................................. 26
   4.2. Avalon Streaming Interfaces............................................... 27
   4.3. Avalon Memory-Mapped Master Interfaces............................ 30
       4.3.1. Memory-Mapped Master Testbench Constructor................ 31
       4.3.2. Implicit and Explicit Examples of Describing a Memory Interface.......................................................... 31
   4.4. Slave Interfaces............................................................... 32
       4.4.1. Control and Status Register (CSR) Slave........................ 33
       4.4.2. Slave Memories......................................................... 36
   4.5. Component Invocation Interface Arguments................................ 37
   4.6. Unstable and Stable Component Arguments............................. 37
   4.7. Global Variables............................................................. 38
   4.8. Structs in Component Interfaces............................................ 38
   4.9. Reset Behavior............................................................... 39

5. Local Variables in Components (Memory Attributes).................. 40
   5.1. Static Variables.............................................................. 42

6. Loops in Components............................................................ 44
   6.1. Loop Initiation Interval (`iiPragma`)..................................... 45
A.12. Systems of Tasks API ................................................................. 105
  A.12.1. ihc::stream Class ................................................................. 107
A.13. Intel HLS Compiler Streaming Input Interfaces ...................................... 108
A.14. Intel HLS Compiler Streaming Output Interfaces ...................................... 113
A.15. Intel HLS Compiler Memory-Mapped Interfaces ...................................... 118
A.16. Intel HLS Compiler AC Datatypes .................................................. 121

B. Supported Math Functions ........................................................................ 124
  B.1. Math Functions Provided by the math.h Header File .......................... 124
  B.2. Math Functions Provided by the extendedmath.h Header File .......... 128
  B.3. Math Functions Provided by the ac_fixed_math.h Header File .......... 129
1. Intel® HLS Compiler Reference Manual

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

The features and devices supported by the Intel HLS Compiler depend on what edition of Intel Quartus® Prime you have. The following icons indicate content in this publication that applies only to the Intel HLS Compiler provided with a certain edition of Intel Quartus Prime:

- PRO Indicates that a feature or content applies only to Intel HLS Compiler Pro Edition.
- STD Indicates that a feature or content applies only to Intel HLS Compiler Standard Edition.

In this publication, <quartus_installdir> refers to the location where you installed Intel Quartus Prime Design Suite. The Intel High Level Synthesis (HLS) Compiler is installed as part of your Intel Quartus Prime Design Suite installation.

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

- PRO Windows C:\intelFPGA_pro\19.1
  Linux /home/<username>/intelFPGA_pro/19.1
- STD Windows C:\intelFPGA_standard\19.1
  Linux /home/<username>/intelFPGA_standard/19.1
2. Compiler

2.1. Intel HLS Compiler Command Options

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

Table 1. General Command Options
These i++ command options perform general compiler functions.

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

Table 2. Command Options that Customize Compilation
These i++ command options perform compiler functions that impact the translation from source file to object file.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>Instructs the compiler to preprocess, parse, and generate object files (.o/.obj) in the current working directory. The linking stage is omitted. Example command: i++ -march=&quot;Arria 10&quot; -c multiplier.c</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Invoking this example command creates a multiplier.o file and sets the name of the .prj directory to multiplier.prj. When you later compile the .o file, the -o option affects only the name of the executable file. The name of the .prj directory remains unchanged from when the directory name was set by i++ -c command invocation.</td>
<td></td>
</tr>
<tr>
<td><strong>--component &lt;components&gt;</strong></td>
<td>Allows you to specify a comma-separated list of function names that you want to 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 &quot;C&quot; specification. For example: extern &quot;C&quot; int myComponent(int a, int b) Using the component function attribute is preferred over using the --component command option to indicate functions that you want the compiler to synthesize.</td>
</tr>
<tr>
<td><strong>-D&lt;macro&gt;[=[val]]</strong></td>
<td>Allows you to pass a macro definition (&lt;macro&gt;) and its value (&lt;val&gt;) to the compiler. If you do not a specify a value for &lt;val&gt;, its default value will be 1.</td>
</tr>
<tr>
<td><strong>-g</strong></td>
<td>Generate debug information (default).</td>
</tr>
<tr>
<td><strong>--gcc-toolchain=&lt;GCC_dir&gt;</strong></td>
<td>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.</td>
</tr>
<tr>
<td>**--hyper-optimized-handshaking=[auto</td>
<td>off]**</td>
</tr>
<tr>
<td><strong>-I&lt;dir&gt;</strong></td>
<td>Adds a directory (&lt;dir&gt;) to the end of the include path list.</td>
</tr>
<tr>
<td>**--march=[x86-64</td>
<td>&lt;FPGA_family&gt;</td>
</tr>
</tbody>
</table>

**continued...**
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>• CycloneV or &quot;Cyclone V&quot;</td>
<td></td>
</tr>
<tr>
<td>• MAX10 or &quot;MAX 10&quot;*(1)</td>
<td></td>
</tr>
<tr>
<td>• StratixV or &quot;Stratix V&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Quotation marks are required only if you specify a FPGA family name specifier that contains spaces

`<FPGA_part_number>`

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.

If you do not specify this option, `-march=x86-64` is assumed.

If the parameter value that you specify contains spaces, surround the parameter value in quotation marks.

(1) If you develop your component IP for Intel MAX® 10 devices and you want to integrate your component IP into a system that you are developing in Intel Quartus Prime, ensure that the Intel Quartus Prime settings file (.qsf) for your system contains one of the following lines:

- set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE IMAGE WITH ERAM"
- set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE COMP IMAGE WITH ERAM"

When you compile the component IP for an Intel MAX 10 devices with Intel HLS Compiler, the generated Intel Quartus Prime example project contains all of the required QSF settings for your component. However, the Intel Quartus Prime project for the system into which you integrate your component might not have the required QSF setting.
### Option Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--promote-integers</td>
<td>Instructs the compiler to use additional FPGA resources to mimic g++ integer promotion. Integer promotion occurs when all integer operations are carried out in 32 bits even if the largest operand is smaller than 32 bits. The default behavior is to carry out integer operations in the size of the largest operand. Refer to the [path to i++ installation]/examples/tutorials/best_practices/integer_promotion design example for usage information on the --promote-integers command option. In Pro Edition, the compiler always promotes integers for standard types. Use the ac_int datatypes if you want smaller (or larger) datatypes.</td>
</tr>
<tr>
<td>--quartus-compile</td>
<td>Compiles your HDL file with the Intel Quartus Prime compiler. Example command: i++ --quartus-compile &lt;input_files&gt; -march=&quot;Arria 10&quot; 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 &lt;result&gt;.prj/quartus directory and a summary of the FPGA resource consumption and maximum clock frequency is added to the high level design reports in the &lt;result&gt;.prj/reports directory. 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.</td>
</tr>
<tr>
<td>--simulator &lt;simulator_name&gt;</td>
<td>Specifies the simulator you are using to perform verification. This command option can take the following values for &lt;simulator_name&gt;: • modelsim • none If you do not specify this option, --simulator modelsim is assumed. <strong>Important:</strong> The --simulator command option only works in conjunction with the -march command option. The --simulator none 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 (report.html) 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. Example command: i++ -march=&quot;&lt;FPGA_family_or_part_number&gt;&quot; --simulator none multiplier.c</td>
</tr>
</tbody>
</table>

### Table 3. 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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--clock &lt;clock_spec&gt;</td>
<td>Optimizes the RTL for the specified clock frequency or period.</td>
</tr>
<tr>
<td>--fpc</td>
<td>Removes intermediate rounding and conversion whenever possible. To see an example of when and how to use this option, review the tutorial in &lt;quartus_installdir&gt;/hls/examples/tutorials/best_practices/floating_point_ops.</td>
</tr>
<tr>
<td>--fp-relaxed</td>
<td>Relaxes the order of arithmetic operations. To see an example of how to use this option, review the tutorial in &lt;quartus_installdir&gt;/hls/examples/tutorials/best_practices/floating_point_ops.</td>
</tr>
<tr>
<td>-ghdl</td>
<td>Logs all signals when running the verification executable. After running the executable, the simulator logs waveforms to the a.prj/verification/vsim.wlf file. For details about the ModelSim* waveform, see Debugging during Verification in <em>Intel High Level Synthesis Compiler User Guide</em>.</td>
</tr>
<tr>
<td>-L&lt;dir&gt;</td>
<td>(Linux only) Adds a directory (&lt;dir&gt;) to the end of the search path for the library files.</td>
</tr>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-l &lt;library&gt;</td>
<td>(Linux only) Specifies the library file (.a) name when linking the object file to the binary. On Windows, you can list library files (.lib) on the command line without specifying any command options or flags.</td>
</tr>
<tr>
<td>--x86-only</td>
<td>Creates only the testbench executable. The compiler outputs an &lt;result&gt; file for Linux or a &lt;result&gt;.exe file for Windows. The &lt;result&gt;.prj directory and its contents are not created.</td>
</tr>
<tr>
<td>--fpga-only</td>
<td>Creates only the &lt;result&gt;.prj directory and its contents. The testbench executable file (&lt;result&gt;/&lt;result&gt;.exe) is not created. Before you can co-simulate your hardware from a compilation output that uses this option, you must compile your testbench with the --x86-only option (or as part of a full compilation).</td>
</tr>
</tbody>
</table>

### 2.2. 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.

STD With Microsoft Visual Studio, you can compile only code that does not explicitly use the Avalon®-Streaming interface.

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.

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`), add the `pthread` library to the linker 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:

```cpp
#ifdef __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 2015 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:

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

If you implement these steps, your Microsoft Visual C++ compiler command resembles the following example command:

```cmd
cl myFile.cpp /I "%HLS_INSTALL_DIR%\include" /nologo /EHsc /wd4068 /MD /link " /libpath:%HLS_INSTALL_DIR%\host\windows64\lib\ hls_emul.lib"
```

### 2.3. 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**

```cpp
component int pe (int A, int B, int C) {
    int product1 = A * B;
    int product2 = B * C;
    int sum = product1 + product2;
    return sum;
}
```

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 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 4. Intel HLS Compiler Header Files Summary

<table>
<thead>
<tr>
<th>HLS Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLS/hls.h</td>
<td>Required for component identification and component parameter interfaces.</td>
</tr>
<tr>
<td>HLS/math.h</td>
<td>Includes FPGA-specific definitions for the math functions from the math.h for your operating system.</td>
</tr>
<tr>
<td>HLS/extendedmath.h</td>
<td>Includes additional FPGA-specific definitions of math functions not in math.h.</td>
</tr>
<tr>
<td>HLS/ac_int.h</td>
<td>Provides FPGA-optimized arbitrary width integer support.</td>
</tr>
<tr>
<td>HLS/ac_fixed.h</td>
<td>Provides FPGA-optimized arbitrary precision fixed point support.</td>
</tr>
<tr>
<td>HLS/ac_fixed_math.h</td>
<td>Provides FPGA-optimized arbitrary precision fixed point math functions.</td>
</tr>
<tr>
<td>HLS/stdio.h</td>
<td>Provides printf support for components so that printf statements work in x86 emulations, but are disabled in component when compiling to an FPGA architecture.</td>
</tr>
<tr>
<td>&lt;iostream&gt;</td>
<td>To use cout and cerr in your component, guard the statements with the HLS_SYNTHESIS macro.</td>
</tr>
</tbody>
</table>
**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 124.

**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:

```bash
$ 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:

```cpp
#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 124](#)

### 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 56](#)

#### 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.

```c
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:

```c
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:

```c
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:

```c
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__)
^
```

### 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:

```c
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:
3.4. Compiler-Defined Preprocessor Macros

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

Table 5. Macro Definition for __INTELFPGA_COMPILER__

<table>
<thead>
<tr>
<th>Tool Invocation</th>
<th><strong>INTELFPGA_COMPILER</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>g++ or cl</td>
<td>Undefined</td>
</tr>
<tr>
<td>i++ -march=x86-64</td>
<td>1910</td>
</tr>
<tr>
<td>i++ -march=&quot;&lt;FPGA_family_or_part_number&gt;&quot;</td>
<td>1910</td>
</tr>
</tbody>
</table>

Table 6. Macro Definition for HLS_SYNTHESIS

<table>
<thead>
<tr>
<th>Tool Invocation</th>
<th>HLS_SYNTHESIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>g++ or cl</td>
<td>Testbench Code</td>
</tr>
<tr>
<td>i++ -march=x86-64</td>
<td>Undefined</td>
</tr>
<tr>
<td>i++ -march=&quot;&lt;FPGA_family_or_part_number&gt;&quot;</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

3.5. Arbitrary Precision Math Support

The Algorithmic C (AC) datatypes are a collection of header files that Mentor Graphics* provides under the Apache license. Intel developed optimized versions of the AC datatypes to allow the Intel HLS Compiler to generate efficient hardware on Intel FPGAs for these datatypes. For more information on Algorithmic C datatypes, refer to Mentor Graphics Algorithmic C (AC) Datatypes, which is available as <quartus_installdir>/hls/include/ref/ac_datatypes_ref.pdf.

The Intel HLS Compiler supports the following AC datatypes:

Table 7. AC Datatypes Supported by the HLS Compiler

<table>
<thead>
<tr>
<th>AC Datatype</th>
<th>Intel Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac_int</td>
<td>HLS/ac_int.h</td>
<td>Arbitrary width integer support</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To learn more, review the following tutorials:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &lt;quartus_installdir&gt;/hls/examples/tutorials/ac_datatypes/ac_int_basic_ops</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &lt;quartus_installdir&gt;/hls/examples/tutorials/ac_datatypes/ac_int_overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &lt;quartus_installdir&gt;/hls/examples/tutorials/best_practices/struct_interfaces</td>
</tr>
<tr>
<td>ac_fixed</td>
<td>HLS/ac_fixed.h</td>
<td>Arbitrary precision fixed-point support</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To learn more, review the tutorial: &lt;quartus_installdir&gt;/hls/examples/tutorials/ac_datatypes/ac_fixed_constructor</td>
</tr>
<tr>
<td></td>
<td>HLS/ac_fixed_math.h</td>
<td>Support for some nonstandard math functions for arbitrary precision fixed-point datatypes</td>
</tr>
</tbody>
</table>

continued...
AC Datatypes

<table>
<thead>
<tr>
<th>AC Datatype</th>
<th>Intel Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac_complex</td>
<td>HLS/ac_complex.h</td>
<td>Arbitrary precision complex number support</td>
</tr>
</tbody>
</table>

The Intel HLS Compiler also supports some nonstandard math functions for the `ac_fixed` datatype when you include the HLS/ac_fixed_math.h header file.

**Advantages of AC Datatypes**

The AC datatypes have the following advantages over using standard C/C++ datatypes in your components:

- You can achieve smaller datapaths and processing elements for various operations in the circuit.
- The datatypes 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 datatype is too small.

**Limitations of AC Datatypes**

The AC datatypes have the following limitations:

- Multipliers are limited to generating 512-bit results.
- Dividers are limited to a maximum of 64 bits.
- The Intel header files are not compatible with GCC or MSVC. When you use the Intel 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 datatypes with GCC or MSVC, use the reference AC datatypes headers also provided with the Intel HLS Compiler. For details, see AC Datatypes and Native Compilers on page 23.

**Related Information**

AC Datatypes Download page on the Mentor Graphics website

### 3.5.1. Declaring ac_int Datatypes in Your Component

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:

```c
#ifdef __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:

```c
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`.

### 3.5.1.1. Important Usage Information on the `ac_int` Datatype

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.
### 3.5.2. Integer Promotion and ac\_int Datatypes

The rules of integer promotion when you use ac\_int datatypes are different from standard C/C++ rules. Your component design should account for these differing rules.

Depending on the datatype 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 datatype and size of the largest operand, but at least 32 bits. The expression returns the result in the larger datatype.

- **Both operands are ac\_int datatypes**:
  
  If both operands are ac\_int datatypes, operations are carried out in the smallest ac\_int datatype 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 datatype and one ac\_int type, the rules for ac\_int datatype promotion apply. The resulting expression type is always an ac\_int datatype. For example, if you add a `short` datatype and an `ap\_int<16>` datatype, the resulting datatype is `ac\_int<17>`.

In C/C++, literals are by default an `int` datatype, 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:

```c
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 operation. 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):

```c
uint3 x = 7;
if (x != -1) {
    // FAIL
}
```
3.5.3. Debugging Your Use of the ac_int Datatype

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 8. Intel HLS Compiler ac_int Debugging Tools Summary

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG_AC_INT_WARNING</td>
<td>Emits a warning for each detected overflow.</td>
</tr>
<tr>
<td>DEBUG_AC_INT_ERROR</td>
<td>Emits a message for the first overflow that is detected and then exits the component with an error.</td>
</tr>
</tbody>
</table>

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.

3.5.4. Declaring ac_fixed Datatypes in Your Component

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:

```c
#ifdef __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.
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.

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.

### 3.5.5. **PRO** Declaring ac_complex Datatypes in Your Component

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

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

```c
#ifdef __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 datatype of your complex number.

   The underlying datatype can be `ac_int`, `ac_fixed`, and standard C integer or floating-point datatypes.

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.

### 3.5.6. AC Datatypes and Native Compilers

The reference version of the Mentor Graphics Algorithmic C (AC) datatypes 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 datatypes 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:

<table>
<thead>
<tr>
<th>AC Datatype</th>
<th>Reference Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac_int</td>
<td>ref/ac_int.h</td>
<td>Arbitrary width integer support</td>
</tr>
<tr>
<td>ac_fixed</td>
<td>ref/ac_fixed.h</td>
<td>Arbitrary precision fixed-point support</td>
</tr>
<tr>
<td>ac_complex</td>
<td>ref/ac_complex.h</td>
<td>Arbitrary precision complex datatype support</td>
</tr>
</tbody>
</table>
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 26 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 26
- **Pointer and Reference Parameters** on page 26

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 27
- **Avalon Memory-Mapped Master Interfaces** on page 30

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 Arguments** on page 37.
4.1.1. Scalar Parameters

Each scalar argument 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 26 and Figure 3 on page 27.

4.1.2. Pointer and Reference Parameters

Each pointer or reference argument 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 30.

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-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. The Intel HLS Compiler does not support the Avalon-ST channel or error signals.
## Streaming Input Interfaces

Table 9. Intel HLS Compiler Streaming Input Interface Template Summary

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

Table 10. Intel HLS Compiler Streaming Input Interface stream_in Function APIs

<table>
<thead>
<tr>
<th>Function API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T read()</td>
<td>Blocking read call to be used from within the component</td>
</tr>
<tr>
<td>T read(bool&amp; sop, bool&amp; eop)</td>
<td>Available only if usesPackets&lt;true&gt; is set. Blocking read with out-of-band startofpacket and endofpacket signals.</td>
</tr>
<tr>
<td>T read(bool&amp; sop, bool&amp; eop, int&amp; empty)</td>
<td>Available only if usesPackets&lt;true&gt; and usesEmpty&lt;true&gt; are set. Blocking read with out-of-band startofpacket, endofpacket, and empty signals.</td>
</tr>
<tr>
<td>T tryRead(bool &amp;success)</td>
<td>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.</td>
</tr>
<tr>
<td>T tryRead(bool&amp; success, bool&amp; sop, bool&amp; eop)</td>
<td>Available only if usesPackets&lt;true&gt; is set. Non-blocking read with out-of-band startofpacket and endofpacket signals.</td>
</tr>
<tr>
<td>T tryRead(bool&amp; success, bool&amp; sop, bool&amp; eop, int&amp; empty)</td>
<td>Available only if usesPackets&lt;true&gt; and usesEmpty&lt;true&gt; are set. Non-blocking read with out-of-band startofpacket, endofpacket, and empty signals.</td>
</tr>
<tr>
<td>void write(T data)</td>
<td>Blocking write call to be used from the testbench to populate the FIFO to be sent to the component.</td>
</tr>
<tr>
<td>void write(T data, bool sop, bool eop)</td>
<td>Available only if usesPackets&lt;true&gt; is set.</td>
</tr>
</tbody>
</table>

continued...
### Function API

<table>
<thead>
<tr>
<th>Function API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void write(T data, bool sop, bool eop, int empty)</strong></td>
<td>Blocking write call with out-of-band startofpacket and endofpacket signals. Available only if <code>usesPackets&lt;true&gt;</code> and <code>usesEmpty&lt;true&gt;</code> are set. Blocking write call with out-of-band startofpacket, endofpacket, and empty signals.</td>
</tr>
</tbody>
</table>

### Streaming Output Interfaces

**Table 11. Intel HLS Compiler Streaming Output Interface Template Summary**

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

**Table 12. Intel HLS Compiler Streaming Output Interface stream_out Function Call APIs**

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

*continued...*
### Function API

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

### 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 \texttt{ihc::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 \texttt{ihc::mm_master<}> object in your component function signature.

Each \texttt{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 13. Intel HLS Compiler Memory-Mapped Interfaces Summary

<table>
<thead>
<tr>
<th>Template Object or Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ihc::mm_master}</td>
<td>The underlying pointer type.</td>
</tr>
<tr>
<td>\texttt{ihc::dwidth}</td>
<td>The width of the memory-mapped data bus in bits.</td>
</tr>
<tr>
<td>\texttt{ihc::awidth}</td>
<td>The width of the memory-mapped address bus in bits.</td>
</tr>
<tr>
<td>\texttt{ihc::aspace}</td>
<td>The address space of the interface that associates with the master.</td>
</tr>
<tr>
<td>\texttt{ihc::latency}</td>
<td>The guaranteed latency from when a read command exits the component when the external memory returns valid read data.</td>
</tr>
<tr>
<td>\texttt{ihc::maxburst}</td>
<td>The maximum number of data transfers that can associate with a read or write transaction.</td>
</tr>
<tr>
<td>\texttt{ihc::align}</td>
<td>The alignment of the base pointer address in bytes.</td>
</tr>
</tbody>
</table>
Template Object or Argument | Description
---|---
`ihc::readwrite_mode` | The port direction of the interface.
`ihc::waitrequest` | Adds the `waitrequest` signal that is asserted by the slave when it is unable to respond to a read or write request.
`getInterfaceAtIndex` | This testbench function is used to index into an `mm_master` 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:

```cpp
dhc::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.

```cpp
#include "HLS/hs.h"

component void dut(int *ptr1, int *ptr2) {
    *ptr1 += *ptr2;
    *ptr2 += ptr1[1];
}

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

```c
#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;
}
```

4.4. Slave Interfaces

The Intel HLS Compiler provides two different types of slave interfaces that you can use with a component. In general, smaller scalar inputs should use slave registers. Large arrays should use slave memories if your intention is to copy these arrays into or out of the 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" in Avalon Interface Specifications.

<table>
<thead>
<tr>
<th>Slave Type</th>
<th>Associated Slave Interface</th>
<th>Read/Write Behavior</th>
<th>Synchronization</th>
<th>Read Latency</th>
<th>Controlling Interface Data Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>The component control and status register (CSR) slave.</td>
<td>The component cannot update these registers from the datapath, so you can read back only data that you wrote in.</td>
<td>Synchronized with the component start signal.</td>
<td>Fixed value of 1.</td>
<td>Always 64 bits</td>
</tr>
<tr>
<td>Memory (M20K/MLAB)</td>
<td>Dedicated slave interface on the component.</td>
<td>The component reads from this memory and updates it as it runs. Updates from the component datapath are visible in memory.</td>
<td>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.</td>
<td>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 (report.html) for the read latency of a specific slave memory argument.</td>
<td>The data width is a multiple of the slave data type, where the multiple is determined by coalescing the internal accesses.</td>
</tr>
</tbody>
</table>

### 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 to interact with the slave.

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:

```c
#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  | Access  |   Register Contents      | Description */
Address   |         |      (64-bits)           |
------------|---------|--------------------------|-----------------------------
0x0 |       R |         {reserved[62:0], |     Read the busy status of  
|         |               busy[0:0]} |               the component
|         |                          | 0 - the component is ready
|         |                          | to accept a new start
|         |                          | 1 - the component cannot
|         |                          |          accept a new start
|         |                          | 1 - Enable interrupt
|         |                          | 0 - Disable interrupt,
|         |                          |        1 - Enable interrupt
|         |                          | Signals component completion
|         |                          | done is read-only and
|         |                          | interrupt_status is write 1
|         |                          | 1 to clear

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], interrupt_status[0:0] | Signals component completion
      |     | 0 - Disable interrupt,
      |     |        1 - Enable interrupt

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], | Argument char_arg

Generated C header file for the component mycomp_xyz:
<table>
<thead>
<tr>
<th>char_arg[7:0]</th>
</tr>
</thead>
</table>

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)

#ifndef __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 stallable 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.

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

**STD** Unlike component memory, you cannot explicitly configure slave memory arguments (for example, banking or coalescing). You must rely on the automatic configurations generated by the compiler. You can control the structure of your slave memories only by restructuring your load and store operations.

**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.

<table>
<thead>
<tr>
<th>Argument Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hls_avalon_slave_memory_argument</td>
<td>Implement the argument, in on-chip memory blocks, which can be read from or written to over a dedicated slave interface.</td>
</tr>
</tbody>
</table>
4.5. Component Invocation Interface Arguments

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 Argument Summary

<table>
<thead>
<tr>
<th>Invocation Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hls_avalon_streaming_component</td>
<td>This is the default component invocation interface. The component uses start, busy, stall, and done signals for handshaking.</td>
</tr>
<tr>
<td>hls_avalon_slave_component</td>
<td>The start, done, and returndata (if applicable) signals are registered in the component slave memory map.</td>
</tr>
<tr>
<td>hls_always_run_component</td>
<td>The start signal is tied to 1 internally in the component. There is no done signal output.</td>
</tr>
<tr>
<td>hls_stall_free_return</td>
<td>If the downstream component never stalls, the stall signal is removed by internally setting it to 0.</td>
</tr>
</tbody>
</table>

Related Information
Control and Status Register (CSR) Slave on page 33

4.6. Unstable and Stable Component Arguments

If you do not specify the intended behavior for an argument, 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:
**4. Component Interfaces**

- 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.

<table>
<thead>
<tr>
<th>Argument Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hls_stable_argument</td>
<td>A stable argument is an argument that does not change while there is live data in the component (that is, between pipelined function invocations).</td>
</tr>
</tbody>
</table>

### 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 might result 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:

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

This synchronizer code is used in the example Intel Quartus Prime 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:

```c
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. Local Variables in Components (Memory Attributes)

The Intel High Level Synthesis (HLS) Compiler tries to provide the maximum throughput whenever possible. In certain cases, particularly when the Intel HLS Compiler optimizes component memory configurations for throughput, it might be beneficial to trade some throughput for a smaller area.

**PRO** Apply the component memory attributes to local variables, constants, and static variables 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.

**STD** Apply the component memory attributes to local variables and static variables 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 cannot apply memory attributes to constants, slave memories, or 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 Component Memory Attributes Summary

<table>
<thead>
<tr>
<th>Memory Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hls_register</strong></td>
<td>Forces a variable or array to be carried through the pipeline in registers.</td>
</tr>
<tr>
<td></td>
<td>A register variable can be implemented either exclusively in flip-flops (FFs)</td>
</tr>
<tr>
<td></td>
<td>or in a mix of FFs and RAM-based FIFOs.</td>
</tr>
<tr>
<td><strong>hls_memory</strong></td>
<td>Forces a variable or array to be implemented as embedded memory.</td>
</tr>
<tr>
<td><strong>hls_memory_impl</strong></td>
<td>Forces a variable or array to be implemented as embedded memory of a specified type.</td>
</tr>
<tr>
<td><strong>hls_singlepump</strong></td>
<td>Specifies that the memory implementing the local variable must be single pumped.</td>
</tr>
<tr>
<td><strong>hls_doublepump</strong></td>
<td>Specifies that the memory implementing the local variable must be double pumped.</td>
</tr>
<tr>
<td><strong>hls_numbanks</strong></td>
<td>Specifies that the memory implementing the local variable must have a defined number of memory banks.</td>
</tr>
<tr>
<td><strong>hls_bankwidth</strong></td>
<td>Specifies that the memory implementing the local variable must have memory banks of a defined width.</td>
</tr>
<tr>
<td><strong>hls_bankbits</strong></td>
<td>Forces the memory system to split into a defined number of memory banks and defines the bits used to select a memory bank.</td>
</tr>
<tr>
<td><strong>hls_numports_readonly_writeonly</strong></td>
<td>Specifies that the memory implementing the local variable must have a defined number of read and write ports.</td>
</tr>
<tr>
<td><strong>hls_simple_dual_port_memory</strong></td>
<td>Specifies a memory configuration that is equivalent to specifying the hls_singlepump and hls_numports_readonly_writeonly(1,1) component memory attributes.</td>
</tr>
</tbody>
</table>
## Memory Attributes

<table>
<thead>
<tr>
<th>Memory Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hls_merge (depthwise)</td>
<td>Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a depth-wise manner.</td>
</tr>
<tr>
<td>hls_merge (widthwise)</td>
<td>Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a width-wise manner.</td>
</tr>
<tr>
<td>hls_init_on_reset</td>
<td>Forces the static variables inside the component to be reset when the component reset signal is asserted.</td>
</tr>
<tr>
<td>hls_init_on_powerup</td>
<td>Sets the component memory implementing the static variable to set on power-up when the FPGA is programmed.</td>
</tr>
<tr>
<td>hls_max_concurrency</td>
<td>Specifies the memory has a defined maximum number of copies to allow simultaneous iterations of a loop at any given time.</td>
</tr>
</tbody>
</table>

### Struct Datatypes and Memory Attributes

You can apply memory attributes to the members variables in a `struct` variable 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.

For example, in the following code example applies memory attributes to both a declaration and instantiation:

```c
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` and not split it as the `S2` object has the `hls_memory` attribute applied.

### 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 ($N_{\text{banks}}$).
- 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 $b_0, b_1, ..., b_n$ as the bank-select bits ($N_{\text{bits}}$).
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.
- \( N_{\text{banks}} \) must be a power of 2 value.
- \( N_{\text{banks}} = 2^N_{\text{bits}} \)
  \( N_{\text{bits}} \) bank-selection bits that are required to address \( N_{\text{banks}} \) 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 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** *(default behavior)*
  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:

```c
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 also initialized when the component is reset:

```c
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:

```c
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 attempts to pipeline loops to maximize throughput of the various components that you define.

Loop Pipelining

Pipelining loops enables the Intel HLS Compiler 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 4. 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.

<table>
<thead>
<tr>
<th>Incorrect (produces a compile-time error)</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma ivdep TEST_LOOP: for(int idx = 0; idx &lt; counter; idx++) {...}</td>
<td>TEST_LOOP: #pragma ivdep for(int idx = 0; idx &lt; counter; idx++) {...}</td>
</tr>
</tbody>
</table>

**Table 17. Intel HLS Compiler Loop Pragmas Summary**

<table>
<thead>
<tr>
<th>Pragma</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ii</td>
<td>Forces a loop to have a loop initiation interval (II) of a specified value.</td>
</tr>
<tr>
<td>ivdep</td>
<td>Ignores memory dependencies between iterations of this loop.</td>
</tr>
<tr>
<td>loop_coalesce</td>
<td>Tries to fuse all loops nested within this loop into a single loop.</td>
</tr>
<tr>
<td>max_concurrency</td>
<td>Limits the number of iterations of a loop that can simultaneously execute at any time.</td>
</tr>
<tr>
<td>unroll</td>
<td>Unrolls the loop completely or by a number of times.</td>
</tr>
<tr>
<td><strong>speculated_iterations</strong></td>
<td>Specifies the number of clock cycles that a loop exit condition can take to compute.</td>
</tr>
</tbody>
</table>

### 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:

```c
#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 mater 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 a 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.

```c
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.

```c
1  // No loop-carried dependencies for A array accesses
2  #pragma ivdep array(A)
3  for(int i = 0; i < N; i++) {
4      A[i] = A[i - X[i]];
5      B[i] = B[i - Y[i]];
6  }
7
8  // No loop-carried dependencies for array A inside struct
9  #pragma ivdep array(S.A)
10  for(int i = 0; i < N; i++) {
12  }
13
14  // No loop-carried dependencies for array A inside the struct pointed by S
15  #pragma ivdep array(S->X[2][3].A)
16  for(int i = 0; i < N; i++) {
18  }
19
20  // No loop-carried dependencies for A and B because ptr aliases
21  // with both arrays
22  #pragma ivdep array(S->X[2][3].A)
23  for(int i = 0; i < N; i++) {
25  }
26
27  // No loop-carried dependencies for A and B because ptr aliases
28  // with both arrays
```
6.3. Loop Coalescing (loop_coalescePragma)

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:

```c
#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:

```c
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:

```c
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 you 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:

```c
#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 withing 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.

```c
#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;
    }
}
```
The Loop Analysis Report for these components looks like the following example:

![Loop Analysis Report Example](image)

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:

<table>
<thead>
<tr>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler failed to schedule this loop with smaller II due to data dependency on variable(s):</td>
</tr>
<tr>
<td>m (spec.cpp:7)</td>
</tr>
<tr>
<td>No critical loop feedback path during scheduling</td>
</tr>
<tr>
<td>3.00 clock cycles 32-bit Integer Multiply Operation (spec.cpp:9)</td>
</tr>
<tr>
<td>3.00 clock cycles 32-bit Integer Multiply Operation (spec.cpp:9)</td>
</tr>
</tbody>
</table>
7. Component Concurrency

The Intel HLS Compiler assumes that you want a fully pipelined datapath in your component. In the C++ implementation, think of a fully pipelined datapath as calling a function multiple times (for example, by multiple threads) before the first call has returned (see also Figure 4 on page 44 and Intel HLS Compiler Pipeline Approach on page 12). The behavior of multiple component invocations within the synthesized datapath 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:

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

hls_max_concurrency(3)
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 50.
8. 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 85.

For details about the hls_scheduler_target_fmax_mhz component attribute, see hls_scheduler_target_fmax_mhz Component Attribute on page 100.

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 fMAX determines the pipelining effort at the compilation stage. Compiling with Quartus Prime reports the achievable fMAX value for your components.
9. 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 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 18. Intel HLS Compiler System of Tasks Summary

<table>
<thead>
<tr>
<th>Template Object or Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ihc::launch</code></td>
<td>Marks a function as an Intel HLS Compiler task for hardware generation, and launches the task function asynchronously.</td>
</tr>
<tr>
<td><code>ihc::collect</code></td>
<td>Synchronizes the completion of the specified task function in the component.</td>
</tr>
<tr>
<td><code>ihc::stream</code></td>
<td>Allows streaming communication between different task functions.</td>
</tr>
</tbody>
</table>

9.1. Task Functions

The Intel HLS Compiler 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`.

Intel Corporation. All rights reserved. Agilex, 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.*
In the High Level Design Report (report.html), the ihc::launch and ihc::collect calls appear as blocking streaming write and streaming read operations.

**Pointer or Reference Arguments**

Task functions cannot have pointer or reference arguments. Therefore, the only ways to pass data into a task function are through scalar arguments (as described above), or use a global-scope streams (as described in Interaction with External Systems on page 57).

**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 an array.

**Communication Between HLS Task Functions**

For multiple task functions to communicate with each other, use 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

**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 5. **Example 1 of a Valid `ihc::launch/ihc::collect` Sequence**
Figure 6. Example 2 of a Valid `ihc::launch/ihc::collect` Sequence

```plaintext
foo {
    ihc::launch
    ihc::launch
    ihc::collect
    ihc::collect
}
```
Figure 7. Example 3 of a Valid `ihc::launch/ihc::collect` Sequence

Component Function

component baz {
    ihc::launch
    ihc::launch
}
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`

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 arguments 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`.

9.2. 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.

<table>
<thead>
<tr>
<th>(i)</th>
<th>(j)</th>
<th>(0)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
<th>(8)</th>
<th>(9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
</tr>
<tr>
<td>2</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
</tr>
<tr>
<td>3</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
</tr>
<tr>
<td>4</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
</tr>
<tr>
<td>6</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
</tr>
<tr>
<td>7</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
</tr>
<tr>
<td>8</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
</tr>
<tr>
<td>9</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
<td>R</td>
<td>W</td>
<td>RW</td>
</tr>
</tbody>
</table>

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.

### 9.3. 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 19. Intel HLS Compiler Testbench API for System of Tasks**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ihc_hls_set_component_wait_cycle</code></td>
<td>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.</td>
</tr>
</tbody>
</table>
**10. HLS Libraries**

An HLS library is a single platform-specific archive file that contains multiple 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 HLS-specific information.

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

You can include an HLS library in your Intel HLS Compiler component and call the functions in the HLS library from your component without needing to know the hardware design or the implementation details underlying the functions in the library.

You can create an HLS 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. An HLS library can contain multiple objects. You can then use the HLS library in your component and call the functions in the HLS library from your component.

Creating an HLS library is a two-step process. First, each object file is created from the RTL source and other required files with the `fpga_crossgen` command. Then, one or more object files are collected into an HLS library file with the `fpga_libtool` command.

**Figure 9. High-Level View of the HLS Library Creation Process**
To create an HLS library, you need to create the following files and components:

### Table 20. Files and Components Required for Creating an HLS Library

<table>
<thead>
<tr>
<th>File or Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RTL-based Functions</strong></td>
<td></td>
</tr>
<tr>
<td>RTL module source files</td>
<td>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), and Tcl Script File (.tcl).</td>
</tr>
<tr>
<td>Object manifest file</td>
<td>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.</td>
</tr>
<tr>
<td>RTL module function signature file</td>
<td>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.</td>
</tr>
<tr>
<td>HLS emulation model files</td>
<td>C++ files (.cpp and .h) that contain code that is functionally equivalent to the RTL component. The emulation model is used only for component emulation. Co-simulations use the RTL provided in the library.</td>
</tr>
</tbody>
</table>

**RTL Modules and the HLS Pipeline** on page 64  
**Creating an Object File from an RTL Module** on page 74  
**Packaging Object Files into an HLS Library** on page 75  
**Using HLS Libraries in Your Component** on page 75

### 10.1. 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++.

#### 10.1.1. 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 12.
Figure 10.  Example of Pipeline Architecture That Integrates an HLS Library

```c
extern "C" int myMod (int);
component int pe
    (int A, int B, int C) {
        int product1 = A * B;
        int product2 = B * C;
        int mod_output = myMod(C);
        int sum = product1 + product2;
        int result = sum + mod_output;
        return result;
    }
```

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.

10.1.2. 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 `ivalid` signals properly if your RTL module has an internal state. For more information, see Stall-Free RTL on page 72.

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

![Diagram of RTL module interfaces](image)

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.

### 10.1.3. 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.
10.1.4. 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 shows 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:         <ISSTALL_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 21. Elements and Attributes in the Object Manifest File

<table>
<thead>
<tr>
<th>XML Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_SPEC</td>
<td>Top-level element in the object manifest file. There can only be one such top-level element in the file.</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>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.</td>
</tr>
</tbody>
</table>

*continued...*
10.1.4.1. 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>
<thead>
<tr>
<th>XML Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTRIBUTES</td>
<td>Element that contains other XML elements that describe various characteristics (for example, latency) of the RTL module. The example RTL module takes one PARAMETER setting named WIDTH, which has a value of 64. See Table 22 on page 68 for more details other ATTRIBUTES-specific elements. 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.</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Element that contains other XML elements that describe the RTL module interface. 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). 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.</td>
</tr>
<tr>
<td>REQUIREMENTS</td>
<td>Element that specifies one or more RTL resource files (that is, .v, .sv, .vhd, .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. HLS libraries do not support .qip files.</td>
</tr>
<tr>
<td>RESOURCES</td>
<td>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.</td>
</tr>
</tbody>
</table>

10.1.4.1. XML Elements for the RTL module properties file ATTRIBUTES Element

<table>
<thead>
<tr>
<th>XML Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS_STALL_FREE</td>
<td>Instructs the Intel HLS Compiler to remove all stall logic around the RTL module. Set IS_STALL_FREE to &quot;yes&quot; to indicate that the module does not generate stalls internally and it cannot properly handle incoming stalls. The module ignores the stall input. If you set IS_STALL_FREE to &quot;no&quot;, the module must properly handle all stall and valid signals. If you set IS_STALL_FREE to &quot;yes&quot;, you must also set IS_FIXED_LATENCY to &quot;yes&quot;. Also, if the RTL module has an internal state, it must properly handle ivalid=0 inputs.</td>
</tr>
<tr>
<td>IS_FIXED_LATENCY</td>
<td>Indicates whether the RTL module has a fixed latency. Set IS_FIXED_LATENCY to &quot;yes&quot; if the RTL module always takes a known number of clock cycles to compute its output. The value you assign to the EXPECTED_LATENCY element specifies the number of clock cycles. The safe value for IS_FIXED_LATENCY is &quot;no&quot;. When you set IS_FIXED_LATENCY=&quot;no&quot;, the EXPECTED_LATENCY value must be at least 1. For a given RTL module, you may set IS_FIXED_LATENCY to &quot;yes&quot; and IS_STALL_FREE to &quot;no&quot;. Such a module produces its output in a fixed number of clock cycles and handles stall signals properly.</td>
</tr>
<tr>
<td>EXPECTED_LATENCY</td>
<td>Specifies the expected latency of the RTL module.</td>
</tr>
</tbody>
</table>

continued...
If you set IS_FIXED_LATENCY to "yes", set the EXPECTED_LATENCY value to be the exact latency of the module. Otherwise, the Intel HLS Compiler generates incorrect hardware.

For a module with variable latency, the Intel HLS Compiler balances the pipeline around this module to the EXPECTED_LATENCY value that you specify. For modules that can stall and require use of signals such as iready, the EXPECTED_LATENCY value must be set to at least 1.

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.

CAPACITY

Specifies the number of multiple inputs that this module can process simultaneously. You must specify a value for CAPACITY if you also set IS_STALL_FREE="no" and IS_FIXED_LATENCY="no". Otherwise, you do not need to specify a value for CAPACITY.

If CAPACITY is strictly less than EXPECTED_LATENCY, the Intel HLS Compiler automatically inserts capacity-balancing FIFO buffers after this module when necessary.

A conservative but safe value for CAPACITY is 1.

HAS_SIDE_EFFECTS

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. Set HAS_SIDE_EFFECTS to "yes" to indicate that the module has side effects. Specifying HAS_SIDE_EFFECTS to "yes" ensures that optimization efforts do not remove calls to modules with side effects.

Stall-free modules with side effects (that is, IS_STALL_FREE="yes" and HAS_SIDE_EFFECTS="yes") must properly handle invalid=0 input cases because the module might receive invalid data occasionally.

A conservative but safe value for HAS_SIDE_EFFECTS is "yes".

ALLOW_MERGING

This attribute is reserved for future use. To prevent unexpected behavior, always set this attribute as <ALLOW_MERGING value="no"/>

PARAMETER

Specifies the value of an RTL module parameter. PARAMETER attributes:

• name
  Specifies the name of the RTL module parameter.

• value
  Specifies a decimal numeric value for the parameter.

The value for an RTL module parameter can be specified using either a value or a type attribute.

10.1.4.2. 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 23. Mandatory XML Elements for the RTL module properties file INTERFACE Element

<table>
<thead>
<tr>
<th>XML Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>Specifies the input parameter of the RTL module that receives the value of a call argument with the RTL-based function is called.</td>
</tr>
</tbody>
</table>

continued...
10.1.4.3. 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 24. XML Elements for the RTL module properties file RESOURCES Element

<table>
<thead>
<tr>
<th>XML Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUTS</td>
<td>Specifies the number of combinational adaptive look-up tables (ALUTs) that the module uses.</td>
</tr>
<tr>
<td>FFS</td>
<td>Specifies the number of dedicated logic registers that the module uses.</td>
</tr>
<tr>
<td>RAMS</td>
<td>Specifies the number of block RAMs that the module uses.</td>
</tr>
<tr>
<td>DSPS</td>
<td>Specifies the number of digital signal processing (DSP) blocks that the module uses.</td>
</tr>
<tr>
<td>MLABS</td>
<td>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.</td>
</tr>
</tbody>
</table>

10.1.5. 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.
10.1.6. HLS Emulation Models for RTL-Based Functions

For an RTL-based function, write C++ code that serves as an emulation model for that function. When your RTL-based function has an emulation model, HLS components that use that RTL-based function can be compiled and run in emulation mode to help you debug your component more quickly than co-simulating your component.

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

*Important:* If your emulation model 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 emulation is limited because all calls to the function share the same state variables. This differs from state in RTL. 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.

10.1.7. 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.
10.1.8. 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.

10.2. Creating an Object File from an RTL Module

Before an RTL module can be included in an HLS library, 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.

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 (.vhdl) 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 67 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 your 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_location> --target hls --emulation_model <emulation_model> [-o <object_file_name>]
```

Where <RTL_module_properties_file_location> is the path to the RTL module properties (.xml) file. 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.

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.

10.3. Packaging Object Files into an HLS Library

Collect object files in an HLS library file so that an HLS component can incorporate the library and call the functions that are contained in the objects in the library. Package object files into an HLS library with the fpga_libtool command.

Before you package object files into an HLS library, ensure that you have the path information for all of the object files that you want to include in the library.

Create the HLS library file with the following command:

```
fpga_libtool --create library_name[a|obj] object_file_1 [object_file_2 ...
object_file_n] --target hls
```

Where library_name is the name of the HLS library file. Specify the file extension according to the platform supported by the objects in the library: .a for Linux-platform objects and .lib for Windows-platform objects.

You can specify one or more object files to include in the HLS 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
```

10.4. Using HLS Libraries in Your Component

Use HLS libraries to reuse functions created by you or others without needing to know the function implementation details. To use the functions in an HLS library, you must have the HLS library file and the corresponding C-header files available.

To include an HLS 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, adding the library file name to the i++ command.
   
   For example, i++ -march=arria10 MyComponent.cpp libprim.a
11. HLS Source Code Libraries

The Intel HLS Compiler 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:

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
<th>Header file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random number generator</td>
<td>Generate random integers or floating point numbers that follow a uniform distribution, or random floating point numbers that follow a Gaussian distribution</td>
<td>HLS/rand_lib.h</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>Multiply two 2-D matrices.</td>
<td>HLS/matrix_mult.h</td>
</tr>
</tbody>
</table>

11.1. Random Number Generator Library

The random number generator source code library provided with the Intel HLS Compiler 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 25. Properties of Values That Can Be Generated by the Intel HLS Compiler Random Number Generator Library

<table>
<thead>
<tr>
<th>Value distribution</th>
<th>Value type</th>
<th>Value range</th>
<th>Generation method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>Integer</td>
<td>[-2³¹, 2³¹-1]</td>
<td>Tausworthe Generator</td>
</tr>
<tr>
<td></td>
<td>Floating point</td>
<td>[0, 1) (non-inclusive)</td>
<td>Tausworthe Generator</td>
</tr>
<tr>
<td>Gaussian</td>
<td>Floating point</td>
<td>[0, 1)</td>
<td>Central limit theorem (CLT) (Default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Box-Muller</td>
</tr>
</tbody>
</table>
11. HLS Source Code Libraries

11.2. Matrix Multiplication Library

The matrix multiplication source code library provided with the Intel HLS Compiler 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:

- **T**: The data type of the matrix elements (For example, int, float, long, double).
- **t_rowsA**: The number of rows in matrix A.
- **t_colsA**: The number of columns in matrix A. This value also the number of rows in matrix B.
- **t_colsB**: The number of columns in matrix B.
- **DOT_VEC_SIZE**: The number of DSP blocks to use in a single computation. This value must be a factor of **t_colsA**. 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.
- **BLOCK_SIZE**: The number of elements to read at one time from matrix A. The default value of **BLOCK_SIZE** is the value of **DOT_VEC_SIZE**. You can reduce this number if the bandwidth needed by matrix A is lower than the value of **DOT_VEC_SIZE**, but it must remain a factor of **DOT_VEC_SIZE**.
- **RUNNING_SUM_MULT_L**: This parameter can be adjusted to try and improve the fMAX of a component that uses this library. Review the header file for a detailed description of this argument and its effects.

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2019.06.04       | 19.1                       | • **PRO** In Slave Memories on page 36, clarified the use of memory attributes for slave memories.  
   • In Local Variables in Components (Memory Attributes) on page 40, clarified memory attributes support in Intel HLS Compiler Pro Edition and Intel HLS Compiler Standard Edition. |

| 2019.05.03       | 19.1                       | • **PRO** Added information about the `ihc_hls_set_component_wait_cycle` testbench API function to the following sections:  
   — System of Tasks Simulation on page 62  
   — Intel HLS Compiler Simulation API (Testbench Only) on page 89  
   • Updated diagrams in Task Functions on page 56.  
   • Updated diagram in Intel HLS Compiler Pipeline Approach on page 12.  
   • Updated diagram in HLS Libraries on page 63.  
   • Updated diagram in Integration of an RTL Module into the HLS Pipeline on page 64. |

| 2019.04.01       | 19.1                       | • **PRO** Added information about developing your system with HLS tasks in Systems of Tasks on page 56.  
   • **PRO** Added information about templated and overloaded functions in Templated and Overloaded Functions on page 16.  
   • **PRO** Added information about arbitrary precision complex number (`ac_complex`) support to Arbitrary Precision Math Support on page 18.  
   • Updated Compiler Interoperability on page 10 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 12.  
   • In Intel HLS Compiler Command Options on page 6, corrected `--gcc-toolchain` option syntax.  
   • In Intel HLS Compiler Command Options on page 6, updated the description of the `--quartus-compile` 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 `--hyper-optimized-handshaking` option of the `i++` command:  
   — Intel HLS Compiler Command Options on page 6  
   — Intel HLS Compiler `i++` Command-Line Arguments on page 84 |

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

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2019.01.03       | 18.1.1                     | • Updated Loop-Carried Dependencies (ivdep Pragma) on page 46 to indicate that arrays specified by the ivdep loop pragma can now be a reference to an mm_master object.  
• Revised and reorganized Intel High Level Synthesis Compiler Quick Reference on page 84.  
• In Declaring ac_int Datatypes in Your Component on page 19, revised the advice for initializing an ac_int variable to a value larger than 64 bits. To initialize this size of ac_int variable, use the bit_fill or bit_fill_hex utility functions. |
• Removed information about the "HLS/iostream" header file. The function provided by this header file is replaced by using the standard C++ iostream header and the HLS_SYNTHESIS macro.  
• Added description of the HLS_SYNTHESIS macro to C and C++ Libraries on page 14.  
• Updated Slave Interfaces on page 32 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 30. |
| 2018.09.24       | 18.1                       | • The Intel HLS Compiler has a new front end. For a summary of the changes introduced by this new front end, see Improved Intel HLS Compiler Front End in the Intel High Level Synthesis Compiler Version 18.1 Release Notes.  
• The --promote-integers flag and the best_practices/integer_promotion 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 hls_avalon_slave_component 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 hls_avalon_slave_component argument in Component Invocation Interface Arguments on page 37 and Quick Reference now reflects that requirement.  
• In Loops in Components on page 44, clarified the pragma statements that apply to loops must immediately precede the loop that the pragma applies to.  
• In Declaring ac_int Datatypes in Your Component on page 19, added initialization requirement for ac_int variables larger than 64 bits. You must use ac::init_array constructors to initialize ac_int variables larger than 64 bits.  
• In Static Variables on page 42, 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                       | • In Static Variables on page 42, 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 33, corrected a typo. The sentence "You do not need to use the hls_avalon_slave_component attribute to use the hls_avalon_slave_component attribute" was corrected to say "You do not need to use the hls_avalon_slave_component attribute to use the hls_avalon_slave_register_argument attribute." |

continued...
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018.05.07</td>
<td>18.0</td>
<td>- 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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>PRO</strong> Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Pro Edition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>STD</strong> Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Standard Edition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>PRO</strong> Corrected the code example in Intel HLS Compiler Streaming Input Interfaces Code Example. The corrected line is <code>int x = a.tryRead(success);</code> (was <code>int x = a.tryRead(&amp;success);</code>).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>PRO</strong> Added <code>&lt;quartus_installdir&gt;/hls/examples/tutorials/interfaces/explicit_streams_packets_empty</code> to list of tutorials in Table 40 on page 108 and Quick Reference.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>PRO</strong> Added <code>ihc::firstSymbolInHighOrderBits</code> and <code>ihc::usesEmpty</code> to the list of stream interface declarations in Table 40 on page 108 and Quick Reference. Also, revised the description of the <code>ihc::firstSymbolInHighOrderBits</code> declaration.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>STD</strong> Added a footnote to the <code>-march MAX10</code> option in Table 2 on page 6 about a prerequisite required before you synthesize your component IP for Intel MAX 10 devices.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>STD</strong> Updated <code>hls_avalon_slave_memory_argument(N)</code> description in Slave Memories on page 36 to include the description that the parameter value <code>N</code> is the size of the memory in bytes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>STD</strong> Updated Table 8 on page 22 and Table 46 on page 122 to indicate that the <code>ac_int</code> debug macros have the following restrictions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- You must declare the macros in your code before you declare <code>#include HLS/ac_int.h</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The <code>ac_int</code> debugging tools work only for x86 emulation of your component.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>STD</strong> Updated <code>-march </code>&lt;FPGA_family&gt;` options in Intel HLS Compiler Command Options on page 6 to include FPGA family options without a space.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>STD</strong> Revised the description of the <code>ihc::align</code> argument in <code>ihc::align Template Argument</code> on page 120 in Quick Reference. The same information also appears in Avalon Memory-Mapped Master Interfaces on page 30.</td>
</tr>
<tr>
<td>2017.12.22</td>
<td>17.1.1</td>
<td><strong>PRO</strong> Updated <code>hls_avalon_slave_memory_argument(N)</code> description in Slave Memories on page 36 to include the description that the parameter value <code>N</code> is the size of the memory in bytes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>PRO</strong> Updated Table 8 on page 22 and Table 46 on page 122 to indicate that the <code>ac_int</code> debug macros have the following restrictions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- You must declare the macros in your code before you declare <code>#include HLS/ac_int.h</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The <code>ac_int</code> debugging tools work only for x86 emulation of your component.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>PRO</strong> Updated <code>-march </code>&lt;FPGA_family&gt;` options in Intel HLS Compiler Command Options on page 6 to include FPGA family options without a space.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>PRO</strong> Revised the description of the <code>ihc::align</code> argument in <code>ihc::align Template Argument</code> on page 120 in Quick Reference. The same information also appears in Avalon Memory-Mapped Master Interfaces on page 30.</td>
</tr>
<tr>
<td>Document Version</td>
<td>Intel Quartus Prime Version</td>
<td>Changes</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| 2017.11.06       | 17.1                       | • Updated Intel HLS Compiler Command Options on page 6 as follows:  
|                  |                            |   — Revised description of -c i++ command option.  
|                  |                            |   — Added descriptions of the --x86-only and --fpga-only i++ command options.  
|                  |                            | • Updated Supported Math Functions on page 124 as follows:  
|                  |                            |   — Noted that the HLS/extendedmath.h header file is supported only by the Intel HLS Compiler, not by the GCC or MSVC compilers.  
|                  |                            |   — Added popcount to the list functions supported by the HLS/extendedmath.h header file.  
|                  |                            |   — Expanded list of functions provided by HLS/extendedmath.h to explicitly list double-precision and single-precision floating point versions of the functions.  
|                  |                            |   — Added a list of popcount function variations available for different data types.  
|                  |                            | • Updated Arbitrary Precision Math Support on page 18 to include restriction that the Intel arbitrary precision header files cannot be compiled with GCC.  
|                  |                            | • Added the ihc::readwrite_mode Avalon-MM interface to Avalon Memory-Mapped Master Interfaces on page 30 and Quick Reference.  
|                  |                            | • Added the ihc::waitrequest Avalon-MM interface to Avalon Memory-Mapped Master Interfaces on page 30 and Quick Reference.  
|                  |                            | • Added the hls_stall_free_return macro and stall_free_return attribute to Unstable and Stable Component Arguments on page 37 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 Intel High Level Synthesis Compiler Best Practices Guide:  
|                  |                            |   — Moved “Avoid Pointer Aliasing” section to “Avoid Pointer Aliasing”. |
| 2017.06.23       | —                          | • Updated Static Variables on page 42 to add information about static variable initialization and how to control it.  
|                  |                            | • Minor changes and corrections. |
| 2017.06.09       | —                          | • Revised Declaring ac_int Datatypes in Your Component on page 19 for changes in how to include ac_int.h.  
|                  |                            | • Revised Arbitrary Precision Math Support on page 18 to clarify support for Algorithmic C datatypes.  
|                  |                            | • Removed all mentions of --device compiler option. This option has been replaced by the changed function of the -march compiler option. See Table 2 on page 6 for details about the changed function of the -march compiler option.  
|                  |                            | • Updated the generated C header file for the component mycomp_xyz in Control and Status Register (CSR) Slave on page 33.  
|                  |                            | • Added information about structs in component interfaces to Component Interfaces on page 25.  
|                  |                            | • Revised C and C++ Libraries on page 14 with updates to iostream behavior.  
|                  |                            | • Added information about math functions supported by extendedmath.h header file to Supported Math Functions on page 124. |
| 2017.02.03       | —                          | • In Scalar Parameters and Avalon Streaming Interfaces, updated information in the Available Scalar Parameters for Avalon-ST Interfaces table. |

continued...
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2016.11.30        |                             | • In HLS Compiler Command Options, modified the table Command Options that Customize Compilation in the following manner:  
  — Removed the --rtl-only command option and its description because it is no longer in use.  
  — Added the --simulator <name> command option and its description.  
  — Remove the --g 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 Pointer Parameters, Reference Parameters, and Avalon Memory-Mapped Master Interfaces, added information on the `altera::align<value>` template argument in the table.  
• Added the topics Memory-Mapped Test Bench Constructor and Implicit and Explicit Examples of Creating a Memory-Mapped Master Test Bench.  
• In HLS Compiler Command Options, modified the table Command Options that Customize Compilation in the following manner:  
  — Removed the --rtl-only command option and its description because it is no longer in use.  
  — Added the --simulator <name> command option and its description.  
  — Remove the --g 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 Pointer Parameters, Reference Parameters, and Avalon Memory-Mapped Master Interfaces, added information on the `altera::align<value>` template argument in the table.  
• Added the topics Memory-Mapped Test Bench Constructor and Implicit and Explicit Examples of Creating a Memory-Mapped Master Test Bench.  
• In Pointer Parameters, Reference Parameters, and Avalon Memory-Mapped Master Interfaces, added information on the `altera::align<value>` template argument in the table.  
• Added new information to Global Variables about area usage and optimizing for global constants, pointers, and variables. |
| 2016.09.12        |                             | Initial release. |
A. Intel High Level Synthesis Compiler Quick Reference

A.1. Intel HLS Compiler i++ Command-Line Arguments

Use the i++ command-line arguments to affect how your component is compiled and linked.

General i++ Command Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--debug-log</td>
<td>Generate the compiler diagnostics log.</td>
</tr>
<tr>
<td>-h, --help</td>
<td>List compiler command options along with brief descriptions.</td>
</tr>
<tr>
<td>-o result</td>
<td>Place compiler output into the &lt;result&gt; executable and the &lt;result&gt;.prj directory.</td>
</tr>
<tr>
<td>-v</td>
<td>Display messages describing the progress of the compilation.</td>
</tr>
<tr>
<td>--version</td>
<td>Display compiler version information.</td>
</tr>
</tbody>
</table>

Command Options Affecting Compiling

<table>
<thead>
<tr>
<th>Option</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td></td>
<td>Preprocess, parse, and generate object files.</td>
</tr>
<tr>
<td>--component component name</td>
<td></td>
<td>Comma-separated list of function names to synthesize to RTL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To use this option, your component must be configured with C-linkage using the extern &quot;C&quot; specification. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>extern &quot;C&quot; int myComponent(int a, int b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Using the component function attribute is preferred over using the -- component command option to indicate functions that you want the compiler to synthesize.</td>
</tr>
<tr>
<td>-Dmacro[=val]</td>
<td></td>
<td>Define a &lt;macro&gt; with &lt;val&gt; as its value.</td>
</tr>
<tr>
<td>-g</td>
<td></td>
<td>Generate debug information (default option).</td>
</tr>
<tr>
<td>-g0</td>
<td></td>
<td>Do not generate debug information.</td>
</tr>
<tr>
<td>--gcc-toolchain=GCC_dir</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>--hyper-optimized-handshaking</td>
<td>auto</td>
<td>This option applies to Intel Stratix 10 devices only. Use this option to modify the handshaking protocol used in certain areas of your design.</td>
</tr>
</tbody>
</table>
## Command Options Affecting Linking

<table>
<thead>
<tr>
<th>Option</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--clock clock target</td>
<td>240 MHz</td>
<td>Optimize the RTL for the specified clock frequency or period. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i++ -march=&quot;Arria 10&quot; test.cpp --clock 100MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i++ -march=&quot;Arria 10&quot; test.cpp --clock 10ns</td>
</tr>
<tr>
<td>--fp-relaxed</td>
<td></td>
<td>Relax the order of floating point arithmetic operations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To learn more, review the tutorial: &lt;quartus_installdir&gt;/hls/examples/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tutorials/best_practices/floating_point_ops</td>
</tr>
<tr>
<td>--fpc</td>
<td></td>
<td>Remove intermediate rounding and conversion when possible.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To learn more, review the tutorial: &lt;quartus_installdir&gt;/hls/examples/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tutorials/best_practices/floating_point_ops</td>
</tr>
<tr>
<td>-ghdl</td>
<td></td>
<td>Enable full debug visibility and logging of all HDL signals in simulation.</td>
</tr>
<tr>
<td>-l dir dir</td>
<td></td>
<td>(Linux only) Add directory &lt;dir&gt; to the list of directories to be searched</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for library files specified with the -l option.</td>
</tr>
<tr>
<td>-l library library</td>
<td></td>
<td>(Linux only) Search the library name &lt;library&gt; when linking.</td>
</tr>
<tr>
<td>--x86-only</td>
<td></td>
<td>Create only the testbench executable (&lt;result&gt;.out/&lt;result&gt;.exe).</td>
</tr>
<tr>
<td>--fpga-only</td>
<td></td>
<td>Create only the &lt;result&gt;.prj directory and its contents.</td>
</tr>
</tbody>
</table>
A.2. Intel HLS Compiler 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>
<thead>
<tr>
<th>HLS Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLS/hls.h</td>
<td>Required for component identification and component parameter interfaces.</td>
</tr>
<tr>
<td>HLS/math.h</td>
<td>Includes FPGA-specific definitions for the math functions from the <code>math.h</code> for your operating system.</td>
</tr>
<tr>
<td>HLS/extendedmath.h</td>
<td>Includes additional FPGA-specific definitions of math functions not in <code>math.h</code>.</td>
</tr>
<tr>
<td>HLS/ac_int.h</td>
<td>Provides FPGA-optimized arbitrary width integer support.</td>
</tr>
<tr>
<td>HLS/ac_fixed.h</td>
<td>Provides FPGA-optimized arbitrary precision fixed point support.</td>
</tr>
<tr>
<td>HLS/ac_fixed_math.h</td>
<td>Provides FPGA-optimized arbitrary precision fixed point math functions.</td>
</tr>
<tr>
<td>HLS/stdio.h</td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;iostream&gt;</code></td>
<td>To use <code>cout</code> and <code>cerr</code> in your component, guard the statements with the HLS_SYNTHESIS macro.</td>
</tr>
</tbody>
</table>

**hls.h Header File**

Syntax

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

Description

Required for component identification and component parameter interfaces.

**math.h Header File**

Syntax

```c
#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 tutorial: `<quartus_installdir>/hls/examples/tutorials/best_practices/single_vs_double_precision_math`.

**extendedmath.h Header File**

Syntax

```c
#include "HLS/extendedmath.h"
```

Description

Includes additional FPGA-specific definitions of math functions not in `math.h`. 
To learn more, review the design: `<quartus_installdir>/hls/examples/QRD`.

**ac_int.h Header File**

**Syntax**

```c
#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**

```c
#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 tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_constructor`

**ac_fixed_math.h Header File**

**Syntax**

```c
#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 tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_math_library`

**stdio.h Header File**

**Syntax**

```c
#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.

A.3. Compiler-Defined Preprocessor Macros

The has a built-in macro that you can use to customize your code to create flow-dependent behaviors.

Table 27. Macro Definition for `__INTELFPGA_COMPILER__`

<table>
<thead>
<tr>
<th>Tool Invocation</th>
<th><code>__INTELFPGA_COMPILER__</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>g++ or cl</td>
<td>Undefined</td>
</tr>
<tr>
<td>-march=x86-64</td>
<td></td>
</tr>
<tr>
<td>-march=&quot;&lt;FPGA_family_or_part_number&gt;*&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Table 28. Macro Definition for HLS_SYNTHESIS

<table>
<thead>
<tr>
<th>Tool Invocation</th>
<th>HLS_SYNTHESIS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Testbench Code</td>
</tr>
<tr>
<td>g++ or cl</td>
<td>Undefined</td>
</tr>
<tr>
<td>-march=x86-64</td>
<td>Undefined</td>
</tr>
<tr>
<td>-march=&quot;&lt;FPGA_family_or_part_number&gt;*&quot;</td>
<td>Undefined Defined</td>
</tr>
</tbody>
</table>

A.4. Intel HLS Compiler Keywords

Table 29. Intel HLS Compiler Keywords

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
</table>
| component  | Indicates that a function is a component. Example:  
```
component void foo()
``` |
### A.5. Intel HLS Compiler Simulation API (Testbench Only)

Table 30. Intel HLS Compiler Simulation API (Testbench only) Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ihc_hls_enqueue</td>
<td>This function enqueues one invocation of an HLS component.</td>
</tr>
<tr>
<td>ihc_hls_enqueue_noret</td>
<td>This function enqueues one invocation of an HLS component. This function</td>
</tr>
<tr>
<td></td>
<td>should be used when the return type of the HLS component is void.</td>
</tr>
<tr>
<td>ihc_hls_component_run_all</td>
<td>This function pushes all enqueued invocations of a component into the</td>
</tr>
<tr>
<td></td>
<td>component in the HDL simulator as quickly as the component can accept</td>
</tr>
<tr>
<td></td>
<td>new invocations.</td>
</tr>
<tr>
<td>ihc_hls_sim_reset</td>
<td>This function sends a reset signal to the component during automated</td>
</tr>
<tr>
<td></td>
<td>simulation.</td>
</tr>
<tr>
<td>ihc_hls_set_component_wait_cycle</td>
<td>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.</td>
</tr>
</tbody>
</table>

#### ihc_hls_enqueue Function

**Syntax**

```c
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**

```c
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**

```c
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**
```c
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**
```c
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**

```c
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);
}
```
A.6. Intel HLS Compiler Component Memory Attributes

Use the component memory attributes to control the on-chip component memory architecture of your component.

Table 31. Intel HLS Compiler Component Memory Attributes Summary

<table>
<thead>
<tr>
<th>Memory Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hls_register</td>
<td>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.</td>
</tr>
<tr>
<td>hls_memory</td>
<td>Forces a variable or array to be implemented as embedded memory.</td>
</tr>
<tr>
<td>hls_memory_impl</td>
<td>Forces a variable or array to be implemented as embedded memory of a specified type.</td>
</tr>
<tr>
<td>hls_singlepump</td>
<td>Specifies that the memory implementing the local variable must be single pumped.</td>
</tr>
<tr>
<td>hls_doublepump</td>
<td>Specifies that the memory implementing the local variable must be double pumped.</td>
</tr>
<tr>
<td>hls_numbanks</td>
<td>Specifies that the memory implementing the local variable must have a defined number of memory banks.</td>
</tr>
<tr>
<td>hls_bankwidth</td>
<td>Specifies that the memory implementing the local variable must have memory banks of a defined width.</td>
</tr>
<tr>
<td>hls_bankbits</td>
<td>Forces the memory system to split into a defined number of memory banks and defines the bits used to select a memory bank.</td>
</tr>
<tr>
<td>hls_numportsreadonlywriteonly</td>
<td>Specifies that the memory implementing the local variable must have a defined number of read and write ports.</td>
</tr>
<tr>
<td>hls_simple_dual_port_memory</td>
<td>Specifies a memory configuration that is equivalent to specifying the hls_singlepump and hls_numportsreadonlywriteonly(1,1) component memory attributes.</td>
</tr>
<tr>
<td>hls_merge (depthwise)</td>
<td>Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a depth-wise manner.</td>
</tr>
<tr>
<td>hls_merge (widthwise)</td>
<td>Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a width-wise manner.</td>
</tr>
<tr>
<td>hls_init_on_reset</td>
<td>Forces the static variables inside the component to be reset when the component reset signal is asserted.</td>
</tr>
<tr>
<td>hls_init_on_powerup</td>
<td>Sets the component memory implementing the static variable to set on power-up when the FPGA is programmed.</td>
</tr>
<tr>
<td>hls_max_concurrency</td>
<td>Specifies the memory has a defined maximum number of copies to allow simultaneous iterations of a loop at any given time.</td>
</tr>
</tbody>
</table>

hls_register Memory Attribute

Syntax       hls_register
Constraints   N/A
Default Value Based on the memory access pattern inferred by the compiler.
Description   Forces a variable or array to be implemented as registers.
To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/best_practices/swap_vs_copy`.

**hls_memory Memory Attribute**

**Syntax**

```
hls_memory
```

**Constraints**
N/A

**Default Value**
Based on the memory access pattern inferred by the compiler.

**Description**
Forces a variable or array to be implemented as embedded memory.

To learn more, review the design: `<quartus_installdir>/hls/examples/QRD`.

**hls_memory_impl Memory Attribute**

**Syntax**

```
hls_memory_impl("type")
```

**Constraints**
N/A

**Default Value**
Based on the memory size and memory access pattern inferred by the compiler.

**Description**
Forces a variable or array to be implemented as embedded memory of the specified type.

The `type` 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).

**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 local variable must be single pumped.

That is, the memory is clocked at the same operating frequency as the operating frequency of your component.
To learn more, review the design: `<quartus_install_dir>/hls/examples/QRD`.

**hls_doublepump Memory Attribute**

**Syntax**

```plaintext
hls_doublepump
```

**Constraints**

N/A

**Default Value**

Based on the memory access pattern inferred by the compiler.

**Description**

Specifies that the memory implementing the local variable must be double pumped.

That is, the memory is clocked at twice the operating frequency of your component.

**hls_numbanks Memory Attribute**

**Syntax**

```plaintext
hls_numbanks(N)
```

**Constraints**

This attribute is subject to constraints outlined in Constraints on Attributes for Memory Banks on page 41.

**Default Value**

Based on the memory access pattern inferred by the compiler.

**Description**

Specifies that the memory implementing the local variable must have N banks, where N is a power-of-two constant number.

**hls_bankwidth Memory Attribute**

**Syntax**

```plaintext
hls_bankwidth(N)
```

**Constraints**

This attribute is subject to constraints outlined in Constraints on Attributes for Memory Banks on page 41.

**Default Value**

Based on the memory access pattern inferred by the compiler.

**Description**

Specifies that the memory implementing the local variable must have banks that are N bytes wide, where N is a power-of-two constant number.

To learn more, review the design: `<quartus_install_dir>/hls/tutorials/component_memories/bank_bits`.

**hls_bankbits Memory Attribute**

**Syntax**

```plaintext
hls_bankbits(b_0, b_1, ..., b_n)
```
**Constraints**

This attribute is subject to constraints outlined in Constraints on Attributes for Memory Banks on page 41.

**Default Value**

Lowest bits of the address based on number of banks.

**Description**

Forces the memory system to split into \(2^{n+1}\) banks, with \(\{b_0, b_1, \ldots, b_n\}\) forming the bank-select bits.

**Important:** \(b_0, b_1, \ldots, b_n\) 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 \(b_0, b_1, \ldots, b_n\) are mapped to array index bits 0 to \(n-1\) in the memory bank implementation.

To learn more, review the design: `<quartus_installdir>/hls/tutorials/component_memories/bank_bits`.

---

**hls_numports_readonly_writeonly Memory Attribute**

**Syntax**

\[ \text{hls\_numports\_readonly\_writeonly}(M, N) \]

**Constraints**

N/A

**Default Value**

Based on the memory access pattern inferred by the compiler.

**Description**

Specifies that the memory implementing the local variable must have \(M\) read ports and \(N\) write ports, where \(M\) and \(N\) are constant numbers greater than zero.

---

**hls_simple_dual_port_memory Memory Attribute**

**Syntax**

\[ \text{hls\_simple\_dual\_port\_memory} \]

**Constraints**

N/A

**Default Value**

N/A

**Description**

Specifies a memory configuration that is equivalent to specifying the following component memory attributes:

- `hls_singlepump`
- `hls_numports_readonly_writeonly(1,1)`

---

**hls_merge (depthwise) Memory Attribute**

**Syntax**

\[ \text{hls\_merge("mem\_name", "depth")} \]

**Constraints**

N/A
**hls_merge (widthwise) Memory Attribute**

**Syntax**

```plaintext
hls_merge("mem_name", "width")
```

**Constraints**

N/A

**Default Value**

N/A

**Description**

Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a width-wise manner.

All variables with same `<mem_name>` label specified in their hls_merge attribute are merged into the same memory system.

To learn more, review the tutorial: `<quartus_install_dir>/hls/examples/tutorials/best_practices/width_wise_merge`.

**hls_init_on_reset Memory Attribute**

**Syntax**

```plaintext
hls_init_on_reset
```

**Constraints**

N/A

**Default Value**

Default behavior for static variables.

**Description**

Forces the static variables inside the component to be reset when the component reset 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.

To learn more, review the tutorial: `<quartus_install_dir>/hls/examples/tutorials/component_memories/static_var_init`.

**hls_init_on_powerup Memory Attribute**

**Syntax**

```plaintext
hls_init_on_powerup
```
Constraints  N/A
Default Value  N/A
Description  Sets the component memory implementing the static variable to set 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.

To learn more, review the 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 $N$ simultaneous 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.

### A.7. Intel HLS Compiler Loop Pragmas

Use the Intel HLS Compiler loop pragmas to control how the compiler pipelines the loops in your component.

**Table 32. Intel HLS Compiler Loop Pragmas Summary**

<table>
<thead>
<tr>
<th>Pragma</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ii</td>
<td>Forces a loop to have a loop initiation interval (II) of a specified value.</td>
</tr>
<tr>
<td>ivdep</td>
<td>Ignores memory dependencies between iterations of this loop.</td>
</tr>
<tr>
<td>loop_coalesce</td>
<td>Tries to fuse all loops nested within this loop into a single loop.</td>
</tr>
<tr>
<td>max_concurrency</td>
<td>Limits the number of iterations of a loop that can simultaneously execute at any time.</td>
</tr>
<tr>
<td>unroll</td>
<td>Unrolls the loop completely or by a number of times.</td>
</tr>
<tr>
<td>speculated_iterations</td>
<td>Specifies the number of clock cycles that a loop exit condition can take to compute.</td>
</tr>
</tbody>
</table>
### ii Loop Pragma

**Syntax**

```plaintext
#pragma ii N
```

**Description**

Forces the loop that this is applied on to have a loop initiation interval (II) of `<N>`, where `<N>` is a positive integer value.

This can have an adverse effect on the `fMAX` 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 `fMAX` value.

**Example:**

```c
#pragma ii 2
for (int i = 0; i < 8; i++) {
    // Loop body
}
```

### ivdep Loop Pragma

**Syntax**

```plaintext
#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:**

```c
#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**

```plaintext
#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
}
```

### unroll Loop Pragma

**Syntax**: `#pragma unroll N`

**Description**: This pragma unrolls the loop completely or by $<N>$ times, where $<N>$ 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`. 
**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:**

```cpp
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;
}
```

### A.8. Intel HLS Compiler Component Attributes

**Table 33. Intel HLS Compiler Component Attributes Summary**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hls_component_ii</td>
<td>Force the component that you apply this attribute to to have a specified component initiation interval (II).</td>
</tr>
<tr>
<td>hls_max_concurrency</td>
<td>Request more copies of the component memory so that the component can run multiple invocations in parallel.</td>
</tr>
<tr>
<td>hls_scheduler_target_fmax_mhz</td>
<td>Specify the target clock frequency of your component.</td>
</tr>
</tbody>
</table>

**hls_component_ii Component Attribute**

**Syntax**

hls_component_ii(\(<N>\) )

**Description**

Forces the component that you apply this attribute to to have a component initiation interval (II) of \(<N>\), where \(<N>\) is a positive integer value.

This can have an adverse effect on the \( f_{\text{MAX}} \) 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 \( f_{\text{MAX}} \) value.
**hls_max_concurrency Component Attribute**

**Syntax**

\[ \text{hls\_max\_concurrency}(N) \]

**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 \text{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:

\(<\text{quartus\_installdir}/hls/examples/inter\_decim\_filter>\).

**Example**

```cpp
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**

\[ \text{hls\_scheduler\_target\_fmax\_mhz}(N) \]

**Description**

Apply the \text{hls\_scheduler\_target\_fmax\_mhz} component attribute to have the compiler target a specific \( f_{\text{MAX}} \) value. Specify the target \( f_{\text{MAX}} \) 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.
A.9. Intel HLS Compiler Component Default Interfaces

Table 34. Intel HLS Compiler Default Interfaces

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component invocation interface (component call and return)</td>
<td>The component call is implemented as an interface consisting of the component start and busy conduits. The component return is also implemented as an interface that includes the component done and stall signals.</td>
</tr>
<tr>
<td>Scalar parameter interface (passed by value)</td>
<td>Scalar parameters are implemented as input conduits that are synchronized with the component invocation interface.</td>
</tr>
<tr>
<td>Pointer parameter interface (passed by reference)</td>
<td>Pointer parameters are implemented as an implicit Avalon Memory-Mapped Master (mm_master) 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.</td>
</tr>
</tbody>
</table>

A.10. Intel HLS Compiler Component Invocation Interface Arguments

Table 35. Intel HLS Compiler Component Invocation Interface Argument Summary

<table>
<thead>
<tr>
<th>Invocation Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hls_avalon_streaming_component</td>
<td>This is the default component invocation interface. The component uses start, busy, stall, and done signals for handshaking.</td>
</tr>
<tr>
<td>hls_avalon_slave_component</td>
<td>The start, done, and returndata (if applicable) signals are registered in the component slave memory map.</td>
</tr>
<tr>
<td>hls_always_run_component</td>
<td>The start signal is tied to 1 internally in the component. There is no done signal output.</td>
</tr>
<tr>
<td>hls_stall_free_return</td>
<td>If the downstream component never stalls, the stall signal is removed by internally setting it to 0.</td>
</tr>
</tbody>
</table>

**hls_avalon_streaming_component Argument**

**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

```c
component hls_avalon_streaming_component void foo(/*component arguments*/)
```

**hls_avalon_slave_component Argument**

**Description**

The start, done, and return data (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

```c
component hls_avalon_slave_component void foo(/*component arguments*/)
```

**hls_always_run_component Argument**

**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

```c
component hls_always_run_component void foo(/*component arguments*/)
```
**hls_stall_free_return Argument**

*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;}
```

### A.11. Intel HLS Compiler Component Macros

#### Table 36. Intel HLS Compiler Component Macros Summary

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hls_conduit_argument</td>
<td>Implement the argument as an input conduit that is synchronous to the component call (start and busy).</td>
</tr>
<tr>
<td>hls_avalon_slave_register_argumen</td>
<td>Implement the argument as a register that can be read from and written to over an Avalon-MM slave interface.</td>
</tr>
<tr>
<td>hls_avalon_slave_memory_argumen</td>
<td>Implement the argument, in on-chip memory blocks, which can be read from or written to over a dedicated slave interface.</td>
</tr>
<tr>
<td>hls_stable_argument</td>
<td>A stable argument is an argument that does not change while there is live data in the component (that is, between pipelined function invocations).</td>
</tr>
</tbody>
</table>

#### 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**
```c
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**
```c
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

```c
component int dut(
    hls_stable_argument int a,
    hls_stable_argument int b) {
    return a * b;
}
```

A.12. **Pro** Systems of Tasks API

Systems of tasks are available only for Intel HLS Compiler Pro Edition.

Table 37. Intel HLS Compiler System of Tasks Summary

<table>
<thead>
<tr>
<th>Template Object or Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ihc::launch</td>
<td>Marks a function as an Intel HLS Compiler task for hardware generation, and launches the task function asynchronously.</td>
</tr>
<tr>
<td>ihc::collect</td>
<td>Synchronizes the completion of the specified task function in the component.</td>
</tr>
<tr>
<td>ihc::stream</td>
<td>Allows streaming communication between different task functions.</td>
</tr>
</tbody>
</table>

**ihc::launch Function**

**Syntax**

ihc::launch(<function>,<fuction_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:
  ```c
  ihc::launch((foo<int>)), arg_a, arg_b);
  ```

- **<fuction_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.

### Intel HLS Compiler System of Tasks Code Example

The following code example illustrates how you can use the systems of tasks API.

```cpp
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;
}
```
A.12.1. `ihc::stream` Class

Table 38. Intel HLS Compiler Systems of Tasks Streaming Interface Template Summary

<table>
<thead>
<tr>
<th>Template Object or Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ihc::stream</code></td>
<td>Streaming interface to the component or task function.</td>
</tr>
<tr>
<td><code>ihc::buffer</code></td>
<td>Specifies the capacity (in words) of the FIFO buffer on the input data that associates with the stream.</td>
</tr>
<tr>
<td><code>ihc::usesPackets</code></td>
<td>Exposes the <code>startofpacket</code> and <code>endofpacket</code> sideband signals on the stream interface.</td>
</tr>
</tbody>
</table>

**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 Argument**

*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 Argument**

*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.
Table 39. Intel HLS Compiler Streaming Input Interface stream Function APIs

<table>
<thead>
<tr>
<th>Function API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T read()</td>
<td>Blocking read call to be used from within the component or task.</td>
</tr>
<tr>
<td>T read(bool&amp; sop, bool&amp; eop)</td>
<td>Available only if usesPackets&lt;true&gt; is set. Blocking read with out-of-band startofpacket and endofpacket signals.</td>
</tr>
<tr>
<td>T tryRead(bool &amp;success)</td>
<td>Non-blocking read call to be used from within the component or task. The success bool is set to true if the read was valid.</td>
</tr>
<tr>
<td>T tryRead(bool success, bool sop, bool eop)</td>
<td>Available only if usesPackets&lt;true&gt; is set. Non-blocking read with out-of-band startofpacket and endofpacket signals.</td>
</tr>
<tr>
<td>void write(T data)</td>
<td>Blocking write call from the component or task.</td>
</tr>
<tr>
<td>void write(T data, bool sop, bool eop)</td>
<td>Available only if usesPackets&lt;true&gt; is set. Blocking write with out-of-band startofpacket and endofpacket signals.</td>
</tr>
<tr>
<td>bool tryWrite(T data)</td>
<td>Non-blocking write call from the component or task. The return value represents whether the write was successful.</td>
</tr>
<tr>
<td>bool tryWrite(T data, bool sop, bool eop)</td>
<td>Available only if usesPackets&lt;true&gt; is set. Non-blocking write with out-of-band startofpacket and endofpacket signals. The return value represents whether the write was successful.</td>
</tr>
</tbody>
</table>

A.13. Intel HLS Compiler 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 40. Intel HLS Compiler Streaming Input Interface Template Summary

<table>
<thead>
<tr>
<th>Template Object or Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ihc::stream_in</td>
<td>Streaming input interface to the component.</td>
</tr>
<tr>
<td>ihc::buffer</td>
<td>Specifies the capacity (in words) of the FIFO buffer on the input data that associates with the stream.</td>
</tr>
<tr>
<td>ihc::readylatency</td>
<td>Specifies the number of cycles between when the ready signal is deasserted and when the input stream can no longer accept new inputs.</td>
</tr>
<tr>
<td>ihc::bitsPerSymbol</td>
<td>Describes how the data is broken into symbols on the data bus.</td>
</tr>
<tr>
<td>ihc::firstSymbolInHighOrderBits</td>
<td>Specifies whether the data symbols in the stream are in big endian order.</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Template Object or Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ihc::usesPackets</td>
<td>Exposes the startofpacket and endofpacket sideband signals on the stream interface.</td>
</tr>
<tr>
<td>ihc::usesEmpty</td>
<td>Exposes the empty out-of-band signal on the stream interface.</td>
</tr>
<tr>
<td>ihc::usesValid</td>
<td>Controls whether a valid signal is present on the stream interface.</td>
</tr>
</tbody>
</table>

**ihc::stream_in Template Object**

**Syntax**

ihc::stream_in<datatype, template arguments>

**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/multiplex_stream_call_sites`

**ihc::buffer Template Argument**

**Syntax**

ihc::buffer<value>

**Valid Values**

Non-negative integer value.

**Default Value**

0
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 report in 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::readlatency` Template Argument

**Syntax**

```
ihc::readlatency<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 Argument

**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.

**PRO** Data is broken down according to how you set the `ihc::firstSymbolInHighOrderBits` declaration. By default, data is broken down in little endian order.

**STD** Data is always broken down in little endian order.

### `ihc::firstSymbolInHighOrderBits` Template Argument

**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.

```
0A0B0C0D
```

Big Endian  
Little Endian

```
0A 0B 0C 0D  
0D 0C 0B 0A
```

**ihc::usesPackets Template Argument**

**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 Argument**

**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 Argument

Syntax  

\[ \text{ihc::usesValid}\langle\text{value}\rangle \]

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 \text{tryRead} and assuming that success is always true.

If set to false, buffer and readyLatency must be 0.

**Intel HLS Compiler Streaming Input Interface stream\_in Function APIs**

Table 41. Intel HLS Compiler Streaming Input Interface stream\_in Function APIs

<table>
<thead>
<tr>
<th>Function API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T read()</td>
<td>Blocking read call to be used from within the component</td>
</tr>
<tr>
<td>T read(bool &amp;sop, bool &amp;eop)</td>
<td>Available only if \text{usesPackets&lt;true&gt;} is set. Blocking read with out-of-band startofpacket and endofpacket signals.</td>
</tr>
<tr>
<td>T \textcolor{red}{\text{PRO}} read(bool &amp;sop, bool &amp;eop, int &amp;empty)</td>
<td>Available only if \text{usesPackets&lt;true&gt;} and \text{usesEmpty&lt;true&gt;} are set. Blocking read with out-of-band startofpacket, endofpacket, and empty signals.</td>
</tr>
<tr>
<td>T \text{tryRead}(bool &amp;success)</td>
<td>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 \text{tryRead()} is not cycle-accurate, so the behavior of \text{tryRead()} might differ between emulation and co-simulation.</td>
</tr>
<tr>
<td>T \text{tryRead}(bool &amp;success, bool &amp;sop, bool &amp;eop)</td>
<td>Available only if \text{usesPackets&lt;true&gt;} is set. Non-blocking read with out-of-band startofpacket and endofpacket signals.</td>
</tr>
<tr>
<td>\textcolor{red}{\text{PRO}} T \text{tryRead}(bool &amp;success, bool &amp;sop, bool &amp;eop, int &amp;empty)</td>
<td>Available only if \text{usesPackets&lt;true&gt;} and \text{usesEmpty&lt;true&gt;} are set. Non-blocking read with out-of-band startofpacket, endofpacket, and empty signals.</td>
</tr>
<tr>
<td>void write(T data)</td>
<td>Blocking write call to be used from the testbench to populate the FIFO to be sent to the component.</td>
</tr>
<tr>
<td>void write(T data, bool sop, bool eop)</td>
<td>Available only if \text{usesPackets&lt;true&gt;} is set.</td>
</tr>
</tbody>
</table>

\textit{continued...}
**Intel HLS Compiler Streaming Input Interfaces Code Example**

The following code example illustrates both `stream_in` declarations and `stream_in` function APIs.

```c++
// 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);
}
```

**A.14. Intel HLS Compiler 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 42. Intel HLS Compiler Streaming Output Interface Template Summary**

<table>
<thead>
<tr>
<th>Template Object or Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ihc::stream_out</code></td>
<td>Streaming output interface from the component.</td>
</tr>
<tr>
<td><code>ihc::readylatency</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>ihc::bitsPerSymbol</code></td>
<td>Describes how the data is broken into symbols on the data bus.</td>
</tr>
<tr>
<td><code>ihc::firstSymbolInHighOrderBits</code></td>
<td>Specifies whether the data symbols in the stream are in big endian order.</td>
</tr>
<tr>
<td>Template Object or Argument</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ihc::usesPackets</td>
<td>Exposes the startofpacket and endofpacket sideband signals on the stream interface.</td>
</tr>
<tr>
<td>ihc::usesEmpty</td>
<td>Exposes the empty out-of-band signal on the stream interface.</td>
</tr>
<tr>
<td>ihc::usesReady</td>
<td>Controls whether a ready signal is present.</td>
</tr>
</tbody>
</table>

**ihc::stream_out Template Object**

**Syntax**

```
ihc::stream_out<datatype, template arguments>
```

**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/multiple_stream_call_sites`

**ihc::readylatency Template Argument**

**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 Argument

**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.

- **PRO**: Data is broken down according to how you set the `ihc::firstSymbolInHighOrderBits` declaration. By default, data is broken down in little endian order.
- **STD**: Data is always broken down in little endian order.

**ihc::firstSymbolInHighOrderBits** Template Argument

**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 Argument

**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 Argument**

**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 Argument**

**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 Streaming Output Interface stream_in Function APIs

**Table 43.** Intel HLS Compiler Streaming Output Interface `stream_out` Function Call APIs

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

### Intel HLS Compiler Streaming Output Interfaces Code Example

The following code example illustrates both `stream_out` declarations and `stream_out` function APIs.

```c
// 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();
    }
}
```
A.15. Intel HLS Compiler Memory-Mapped Interfaces

Use the `mm_master` object and template arguments to explicitly declare Avalon Memory-Mapped (MM) Master interfaces for your component.

### Table 44. Intel HLS Compiler Memory-Mapped Interfaces Summary

<table>
<thead>
<tr>
<th>Template Object or Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ihc::mm_master</td>
<td>The underlying pointer type.</td>
</tr>
<tr>
<td>ihc::dwidth</td>
<td>The width of the memory-mapped data bus in bits.</td>
</tr>
<tr>
<td>ihc::awidth</td>
<td>The width of the memory-mapped address bus in bits.</td>
</tr>
<tr>
<td>ihc::aspace</td>
<td>The address space of the interface that associates with the master.</td>
</tr>
<tr>
<td>ihc::latency</td>
<td>The guaranteed latency from when a read command exits the component when the external memory returns valid read data.</td>
</tr>
<tr>
<td>ihc::maxburst</td>
<td>The maximum number of data transfers that can associate with a read or write transaction.</td>
</tr>
<tr>
<td>ihc::align</td>
<td>The alignment of the base pointer address in bytes.</td>
</tr>
<tr>
<td>ihc::readwrite_mode</td>
<td>The port direction of the interface.</td>
</tr>
<tr>
<td>ihc::waitrequest</td>
<td>Adds the <code>waitrequest</code> signal that is asserted by the slave when it is unable to respond to a read or write request.</td>
</tr>
<tr>
<td>getInterfaceAtIndex</td>
<td>This testbench function is used to index into an mm_master object.</td>
</tr>
</tbody>
</table>

**ihc::mm_master Template Object**

**Syntax**

`ihc::mm_master<datatype, template arguments>`

**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:**

```c
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 Argument**

**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 Argument**

**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 Argument**

**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 Argument**

**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 Argument**

**Syntax**  ihc::maxburst\(<value>\)

**Valid Values**  Integer value in the range 1 – 1024

**Default value**  1

**Description**  The maximum number of data transfers that can associate with a read or write transaction. This value controls the width of the burstcount signal.

For fixed latency interfaces, this value must be set to 1.

For more details, review information about burst signals and the burstcount signal role in "Avalon Memory-Mapped Interface Signal Roles" in Avalon Interface Specifications.

**ihc::align Template Argument**

**Syntax**  ihc::align\(<value>\)

**Valid Values**  Integer value greater than the alignment of the datatype

**Default value**  Alignment of the datatype

**Description**  The alignment of the base pointer address in bytes.

The Intel HLS Compiler uses this information to determine how many simultaneous loads and stores this pointer can permit.

For example, if you have a bus with 4 32-bit integers on it, you should use ihc::dwidth<128> (bits) and ihc::align<16> (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.

**Important:** The caller is responsible for aligning the data to the set value for the align argument; otherwise, functional failures might occur.
**ihc::readwrite_mode Template Argument**

**Syntax**

ihc::readwrite_mode<value>

**Valid Values**

readwrite, readonly, or writeonly

**Default value**

readwrite

**Description**

The port direction of the interface. Only the relevant Avalon master signals are generated.

**ihc::waitrequest Template Argument**

**Syntax**

ihc::waitrequest<value>

**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 indices of the array. This function is supported only in the testbench.

**Code Example**

```c
int main() {
    // ....
    for(int idx = 0; idx < N; idx++) {
        dut(src_mm.getInterfaceAtIndex(idx));
    }
    // ....
}
```

**A.16. Intel HLS Compiler AC Datatypes**

**Table 45. AC Datatypes Supported by the HLS Compiler**

<table>
<thead>
<tr>
<th>AC Datatype</th>
<th>Intel Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac_int</td>
<td>HLS/ac_int.h</td>
<td>Arbitrary width integer support</td>
</tr>
</tbody>
</table>

To learn more, review the following tutorials:

**continued...**
### AC Datatypes

- **ac_fixed**
  - **Intel Header File**: HLS/ac_fixed.h
  - **Description**: Arbitrary precision fixed-point support
  - To learn more, review the tutorial: <quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_constructor

- **ac_complex**
  - **Intel Header File**: HLS/ac_complex.h
  - **Description**: Arbitrary precision complex number support

### Intel HLS Compiler ac_int Debugging Tools Summary

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG_AC_INT_WARNING</td>
<td>Emits a warning for each detected overflow.</td>
</tr>
<tr>
<td>DEBUG_AC_INT_ERROR</td>
<td>Emits a message for the first overflow that is detected and then exits the component with an error.</td>
</tr>
</tbody>
</table>

#### DEBUG_AC_INT_WARNING ac_int Debugging Tool

**Macro Syntax**

```c
#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**

```c
-D DEBUG_AC_INT_WARNING
```

**Description**

Enables runtime tracking of `ac_int` datatypes during x86 emulation (the `-march=x86-64` option, which is 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**

```c
#define DEBUG_AC_INT_ERROR
```

Table 46. Intel HLS Compiler ac_int Debugging Tools Summary
If you use this macro, declare it in your code before you declare
#include HLS/ac_int.h.

<table>
<thead>
<tr>
<th>Command</th>
<th>Option Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i++</code> Command</td>
<td><code>-D DEBUG_AC_INT_ERROR</code></td>
<td>Enables runtime tracking of <code>ac_int</code> datatypes during x86 emulation of your component (the <code>-march=x86-64</code> option, which the default option, of the <code>i++</code> 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.</td>
</tr>
</tbody>
</table>

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

```c
#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:

```c
#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:

```c
#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`.

B.1. Math Functions Provided by the math.h Header File

The Intel HLS Compiler 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 47. Trigonometric Functions

<table>
<thead>
<tr>
<th>Trigonometric Function</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>cos</td>
<td>●</td>
</tr>
<tr>
<td>sin</td>
<td>●</td>
</tr>
<tr>
<td>tan</td>
<td>●</td>
</tr>
<tr>
<td>acos</td>
<td>●</td>
</tr>
<tr>
<td>asin</td>
<td>●</td>
</tr>
<tr>
<td>atan</td>
<td>●</td>
</tr>
<tr>
<td>atan2</td>
<td>●</td>
</tr>
</tbody>
</table>

#### Double-precision floating point functions

| cosf                   | ●                               |
| sinc                   | ●                               |
| tanf                   | ●                               |
| acosf                  | ●                               |
| asinf                  | ●                               |
| atanf                  | ●                               |
| atan2f                 | ●                               |

#### Single-precision floating point functions

### Table 48. Hyperbolic Functions

<table>
<thead>
<tr>
<th>Hyperbolic Function</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>cosh</td>
<td>●</td>
</tr>
<tr>
<td>sinh</td>
<td>●</td>
</tr>
<tr>
<td>tanh</td>
<td>●</td>
</tr>
<tr>
<td>acosh</td>
<td>X</td>
</tr>
<tr>
<td>asinh</td>
<td>X</td>
</tr>
<tr>
<td>atanh</td>
<td>X</td>
</tr>
</tbody>
</table>

### Table 49. Exponential and Logarithmic Functions

<table>
<thead>
<tr>
<th>Exponential or Logarithmic Function</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>●</td>
</tr>
<tr>
<td>frexp</td>
<td>●</td>
</tr>
<tr>
<td>ldexp</td>
<td>●</td>
</tr>
<tr>
<td>log</td>
<td>●</td>
</tr>
<tr>
<td>log10</td>
<td>●</td>
</tr>
<tr>
<td>modf</td>
<td>●</td>
</tr>
<tr>
<td>exp2</td>
<td>●</td>
</tr>
<tr>
<td>exp10 (Linux only)</td>
<td>●</td>
</tr>
</tbody>
</table>

(•) For Windows, support for this function is in the `extendedmath.h` header file
### Exponential or Logarithmic Function

<table>
<thead>
<tr>
<th>Exponential or Logarithmic Function</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>expm1</td>
<td>●</td>
</tr>
<tr>
<td>ilogb</td>
<td>●</td>
</tr>
<tr>
<td>log1p</td>
<td>● (double only)</td>
</tr>
<tr>
<td>log2</td>
<td>●</td>
</tr>
<tr>
<td>logb</td>
<td>X</td>
</tr>
<tr>
<td>scalbn</td>
<td>X</td>
</tr>
<tr>
<td>scalbln</td>
<td>X</td>
</tr>
</tbody>
</table>

**Table 50. Power Functions**

<table>
<thead>
<tr>
<th>Power Function</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>pow</td>
<td>●</td>
</tr>
<tr>
<td>sqrt</td>
<td>●</td>
</tr>
<tr>
<td>cbrt</td>
<td>●</td>
</tr>
<tr>
<td>hypot</td>
<td>●</td>
</tr>
</tbody>
</table>

**Table 51. Error and Gamma Functions**

<table>
<thead>
<tr>
<th>Error or Gamma Function</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>erf</td>
<td>X</td>
</tr>
<tr>
<td>erfc</td>
<td>X</td>
</tr>
<tr>
<td>tgamma</td>
<td>X</td>
</tr>
<tr>
<td>lgamma</td>
<td>X</td>
</tr>
</tbody>
</table>

**Table 52. Rounding and Remainder Functions**

<table>
<thead>
<tr>
<th>Rounding or Remainder Function</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceil</td>
<td>●</td>
</tr>
<tr>
<td>floor</td>
<td>●</td>
</tr>
<tr>
<td>fmod</td>
<td>●</td>
</tr>
<tr>
<td>trunc</td>
<td>●</td>
</tr>
<tr>
<td>round</td>
<td>●</td>
</tr>
<tr>
<td>iround</td>
<td>X</td>
</tr>
<tr>
<td>ilround</td>
<td>X</td>
</tr>
<tr>
<td>rint</td>
<td>●</td>
</tr>
<tr>
<td>lrint</td>
<td>X</td>
</tr>
<tr>
<td>llrint</td>
<td>X</td>
</tr>
<tr>
<td>nearbyint</td>
<td>X</td>
</tr>
<tr>
<td>remainder</td>
<td>X</td>
</tr>
<tr>
<td>remainder</td>
<td>X</td>
</tr>
<tr>
<td>remquo</td>
<td>X</td>
</tr>
</tbody>
</table>
### Table 53. Floating-Point Manipulation Functions

<table>
<thead>
<tr>
<th>Floating-Point Manipulation Function</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>copysign</td>
<td>X</td>
</tr>
<tr>
<td>nan</td>
<td>X</td>
</tr>
<tr>
<td>nextafter</td>
<td>X</td>
</tr>
<tr>
<td>nexttoward</td>
<td>X</td>
</tr>
</tbody>
</table>

### Table 54. Minimum, Maximum, and Difference Functions

<table>
<thead>
<tr>
<th>Minimum, Maximum, or Difference Function</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdim</td>
<td>●</td>
</tr>
<tr>
<td>fmax</td>
<td>●</td>
</tr>
<tr>
<td>fmin</td>
<td>●</td>
</tr>
</tbody>
</table>

### Table 55. Other Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>fabs</td>
<td>●</td>
</tr>
<tr>
<td>abs</td>
<td>X</td>
</tr>
<tr>
<td>fma</td>
<td>X</td>
</tr>
</tbody>
</table>

### Table 56. Classification Macros

<table>
<thead>
<tr>
<th>Classification Macro</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpclassify</td>
<td>X</td>
</tr>
<tr>
<td>isnormal</td>
<td>●</td>
</tr>
<tr>
<td>isnan</td>
<td>●</td>
</tr>
<tr>
<td>isnan (Linux only)*</td>
<td>●</td>
</tr>
<tr>
<td>isnfinite (Linux only)*</td>
<td>●</td>
</tr>
<tr>
<td>signbit</td>
<td>X</td>
</tr>
</tbody>
</table>

### Table 57. Comparison Macros

<table>
<thead>
<tr>
<th>Comparison Macro</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>isgreater</td>
<td>X</td>
</tr>
<tr>
<td>isgreaterequal</td>
<td>X</td>
</tr>
<tr>
<td>isless</td>
<td>X</td>
</tr>
<tr>
<td>islessequal</td>
<td>X</td>
</tr>
<tr>
<td>islessgreater</td>
<td>X</td>
</tr>
<tr>
<td>isunordered</td>
<td>X</td>
</tr>
</tbody>
</table>
B.2. Math Functions Provided by the extendedmath.h Header File

Adding the HLS/extendedmath.h header file adds support for the following functions:

Table 58. Extended math functions

<table>
<thead>
<tr>
<th>Data type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double-precision floating point</td>
<td>sincos, acospi, asinpi, atanpi, cospi, sinpi, tanpi, pown, power, rsqrt</td>
</tr>
<tr>
<td>Single-precision floating point</td>
<td>sincosf, acospif, asinpif, atanpif, cospif, sinpif, tanpif, pownf, powrf, rsqrtf</td>
</tr>
</tbody>
</table>

Table 59. Exponential and Logarithmic Functions

<table>
<thead>
<tr>
<th>Exponential or Logarithmic Function</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp10 (Windows only)</td>
<td>●</td>
</tr>
</tbody>
</table>

Table 60. Classification Macros

<table>
<thead>
<tr>
<th>Classification Macro</th>
<th>Supported by the HLS Compiler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>isnfinite (Windows only)(*)</td>
<td>●</td>
</tr>
<tr>
<td>isninf (Windows only)(*)</td>
<td>●</td>
</tr>
</tbody>
</table>

In addition, the HLS/extendedmath.h header file supports the following versions of the popcount function:

Table 61. Popcount function

<table>
<thead>
<tr>
<th>Data type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned char</td>
<td>popcountc</td>
</tr>
<tr>
<td>Unsigned short</td>
<td>popcounts</td>
</tr>
</tbody>
</table>

(*) For Linux, support for this function is in the math.h header file
### B. Supported Math Functions

#### 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.

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`.

---

<table>
<thead>
<tr>
<th>Data type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned int</td>
<td>popcount</td>
</tr>
<tr>
<td>Unsigned long</td>
<td>popcountl</td>
</tr>
<tr>
<td>Unsigned long long</td>
<td>popcountll</td>
</tr>
</tbody>
</table>