Intel® High Level Synthesis Compiler Standard Edition

Reference Manual

Updated for Intel® Quartus® Prime Design Suite: **19.1**

Subscribe
Send Feedback

MNL-1106 | 2019.12.18
Latest document on the web: PDF | HTML

Contents

1. Intel® HLS Compiler Standard Edition Reference Manual

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

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

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

Windows	`C:\intelFPGA_standard\19.1`
Linux	`/home/<username>/intelFPGA_standard/19.1`

About the Intel HLS Compiler Standard Edition Documentation Library

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

Table 1. Intel High Level Synthesis Compiler Documentation Library

Title and Description	STD
Release Notes Provide late-breaking information about the Intel HLS Compiler.	Link
Getting Started Guide Get up and running with the Intel HLS Compiler by learning how to initialize your compiler environment and reviewing the various design examples and tutorials provided with the Intel HLS Compiler.	Link
User Guide Provides instructions on synthesizing, verifying, and simulating intellectual property (IP) that you design for Intel FPGA products. Go through the entire development flow of your component from creating your component and testbench up to integrating your component IP into a larger system with the Intel Quartus Prime software.	Link
Reference Manual Provides reference information about the features supported by the Intel HLS Compiler. Find details on Intel HLS Compiler command options, header files, pragmas, attributes, macros, declarations, arguments, and template libraries.	Link
Best Practices Guide Provides techniques and practices that you can apply to improve the FPGA area utilization and performance of your HLS component. Typically, you apply these best practices after you verify the functional correctness of your component.	Link
Quick Reference Provides a brief summary of Intel HLS Compiler declarations and attributes on a single two-sided page.	Link

2. Compiler

2.1. Intel HLS Compiler Standard Edition Command Options

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

Table 2. General Command Options

These i++ command options perform general compiler functions.

Command Option	Description
`--debug-log`	Instructs the compiler to generate a log file that contains diagnostic information. By default, the `debug.log` file is in the `a.prj` subdirectory within your current working directory. If you also include the `-o <result>` command option, the `debug.log` file will be in the `<result>.prj` subdirectory. If your compilation fails, the `debug.log` file is generated whether you set this option or not.
`-h` or `--help`	Instructs the compiler to list all the command options and their descriptions on screen.
`-o` *<result>*	Instructs the compiler to place its output into the `<result>` executable and the `<result>.prj` directory. If you do not specify the `-o` *<result>*option, the compiler outputs an `a.out` file for Linux and an `a.exe` file for Windows. Use the `-o` *<result>* 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.
`-v`	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.
`--version`	Instructs the compiler to display its version information on screen. Command: `i++ --version`

Table 3. Command Options that Customize Compilation

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

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

continued...

Option	Description
	Invoking this example command creates a `multiplier.o` file and sets the name of the `<result>.prj` directory to `multiplier.prj`. When you later link the `.o` file, the `-o` option affects only the name of the executable file. The name of the `<result>.prj` directory remains unchanged from when the directory name was set by `i++ -c` command invocation.
`--component <components>`	Allows you to specify a comma-separated list of function names that you want 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 "C"` specification. For example: `extern "C" 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.
`-D<macro>[=<val>]`	Allows you to pass a macro definition (*<macro>*) and its value (*<val>*) to the compiler. If you do not a specify a value for *<val>*, its default value will be 1.
`-g`	Generate debug information (default).
`-g0`	Do not generate debug information.
`-I<dir>`	Adds a directory (*<dir>*) to the end of the include path list.
`-march=[x86-64 | <FPGA_family> | <FPGA_part_number>`	Instructs the compiler to compile the component to the specified architecture or FPGA family. The `-march` compiler option can take one of the following values: *`x86-64`* — Instructs the compiler to compile the code for an emulator flow. *`"<FPGA_family>"`* — Instructs the compiler to compile the code for a target FPGA device family. The *<FPGA_family>* value can be any of the following device families: • `ArriaV` or `"Arria V"` • `Arria10` or `"Arria 10"` • `CycloneV` or `"Cyclone V"` • `MAX10` or `"MAX 10"`[1] • `StratixV` or `"Stratix V"` Quotation marks are required only if you specify a FPGA family name specifier that contains spaces

continued...

(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 a component for the Intel MAX 10 device family with the 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
	<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.
`--promote-integers`	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.
`--quartus-compile`	Compiles your HDL file with the Intel Quartus Prime compiler. Example command: `i++ --quartus-compile <`*`input_files`*`> -march="Arria 10"` 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 `<`*`result`*`>.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 `<`*`result`*`>.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.
`--simulator` *<simulator_name>*	Specifies the simulator you are using to perform verification. This command option can take the following values for *<simulator_name>*: • `modelsim` • `none` If you do not specify this option, `--simulator modelsim` is assumed. *Important:* 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="<`*`FPGA_family_or_part_number`*`>" --simulator none multiplier.c`
`--fpc`	Remove intermediate rounding and conversion when possible. To learn more, review the following tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/best_practices/floating_point_ops`
`--fp-relaxed`	Relax the order of floating point arithmetic operations. To learn more, review the following tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/best_practices/floating_point_ops`
`--clock` *<clock_spec>*	Optimizes the RTL for the specified clock frequency or period.

Table 4. Command Options that Customize File Linking

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

Option	Description
`-ghdl`	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 *Intel High Level Synthesis Compiler Standard Edition User Guide*.
`-L`<*dir*>	(Linux only) Adds a directory (<*dir*>) to the end of the search path for the library files.
`-l`<*library*>	(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.
`--x86-only`	Creates only the testbench executable. The compiler outputs an <*result*> file for Linux or a <*result*>`.exe` file for Windows. The <*result*>`.prj` directory and its contents are not created.
`--fpga-only`	Creates only the <*result*>`.prj` directory and its contents. The testbench executable file (<*result*>/<*result*>`.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).

2.2. Using Libraries in Your Component

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

To use the functions in a library, you must have the C-header files (`.h`) for the library available.

To include a library in your component:

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

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

2. Include the header files in your component code.

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

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

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

Related Information

Arbitrary Precision Math Support on page 44

2.3. Compiler Interoperability

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

Send Feedback

To see what versions of GCC and Microsoft Visual Studio the Intel HLS Compiler Standard Edition supports, see "Intel High Level Synthesis Compiler Prerequisites" in *Intel High Level Synthesis Compiler Standard Edition 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.

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.

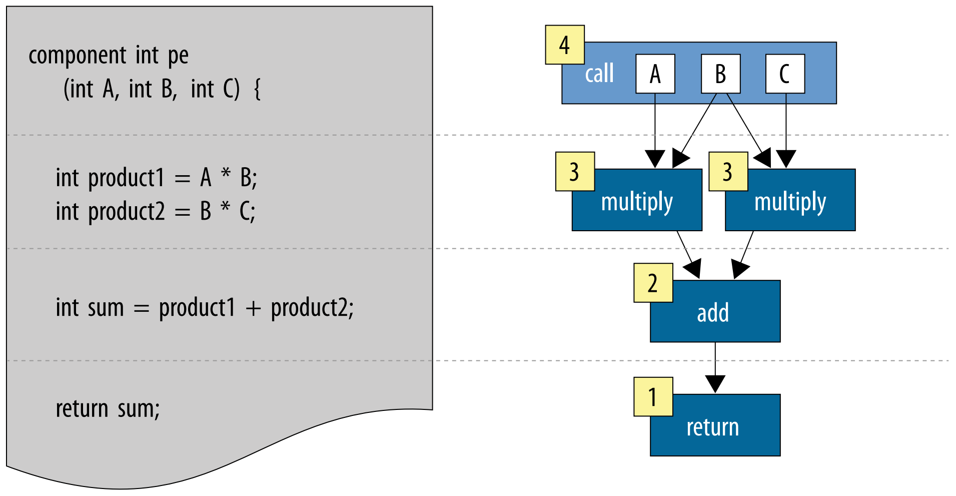
To use a native compiler (GCC or Microsoft Visual Studio) to compile your Intel HLS Compiler code, you must point the native compiler to Intel HLS Compiler resources and libraries. The Intel HLS Compiler example designs contain build scripts (`Makefile` for Linux and `build.bat` for Windows) that you 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.

2.4. Intel HLS Compiler Pipeline Approach

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

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

Figure 1. Example of Pipeline Architecture



```
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 the its next stage in the pipeline only after all of the operations of its current stage are complete.

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

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

3. C Language and Library Support

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

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

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

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

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

3.2. C and C++ Libraries

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

Table 5. Intel HLS Compiler Standard Edition Header Files Summary

HLS Header File	Description
`HLS/hls.h`	Required for component identification and component parameter interfaces.
`HLS/math.h`	Includes FPGA-specific definitions for the math functions from the `math.h` for your operating system.
`HLS/extendedmath.h`	Includes additional FPGA-specific definitions of math functions not in `math.h`.
`HLS/ac_int.h`	Provides FPGA-optimized arbitrary width integer support.
`HLS/ac_fixed.h`	Provides FPGA-optimized arbitrary precision fixed point support.
`HLS/ac_fixed_math.h`	Provides FPGA-optimized arbitrary precision fixed point math functions.
`HLS/stdio.h`	Provides `printf` support for components so that `printf` statements work in x86 emulations, but are disabled in component when compiling to an FPGA architecture.
`"HLS/iostream"`	(Linux only) Provides `cout` and `cerr` support for components so that `cout` and `cerr` statements work in x86 emulations, but are disabled in component when compiling to an FPGA architecture.

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 83.

stdio.h

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

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

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

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

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

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

iostream

Synthesized component functions do not support C++ standard library functions such as C++ stream objects (for example, `cout`).

A component can call `cout` or `cerr` by including the header file `"HLS/iostream"`. This header changes the behavior of `cout` and `cerr` depending on the compilation target:

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

If you attempt to use `cout` or `cerr` in a component function without first including the `#include "HLS/iostream"` line in your code, you will see 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.
```

Important: When you include the header file `"HLS/iostream"`, only writes to `cout` and `cerr` are affected. If you use any of the other standard input/output stream objects, you get a compile-time error. Avoid using the `"HLS/iostream"` header file if you have large sections of testbench code that use standard input/output stream objects.

3.3. Intel HLS Compiler Standard Edition Compiler-Defined Preprocessor Macros

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

Table 6. Macro Definition for __INTELFPGA_COMPILER__

Tool Invocation	__INTELFPGA_COMPILER__
`g++` or `cl`	Undefined
`i++ -march=x86-64`	1910
`i++ -march="<FPGA_family_or_part_number>"`	1910

Table 7. Macro Definition for HLS_SYNTHESIS

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

4. Component Interfaces

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

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

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

4.1. Component Invocation Interface

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

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

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

See Figure 2 on page 15 for an example component showing these interfaces.

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

- Avalon Streaming Interfaces on page 16
- Pointer and Reference Parameters on page 15

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 16
- Avalon Memory-Mapped Master Interfaces on page 18

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 25.

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 15 and Figure 3 on page 16.

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 18.

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

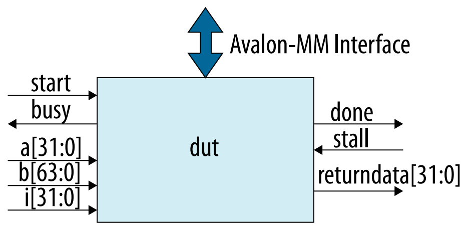
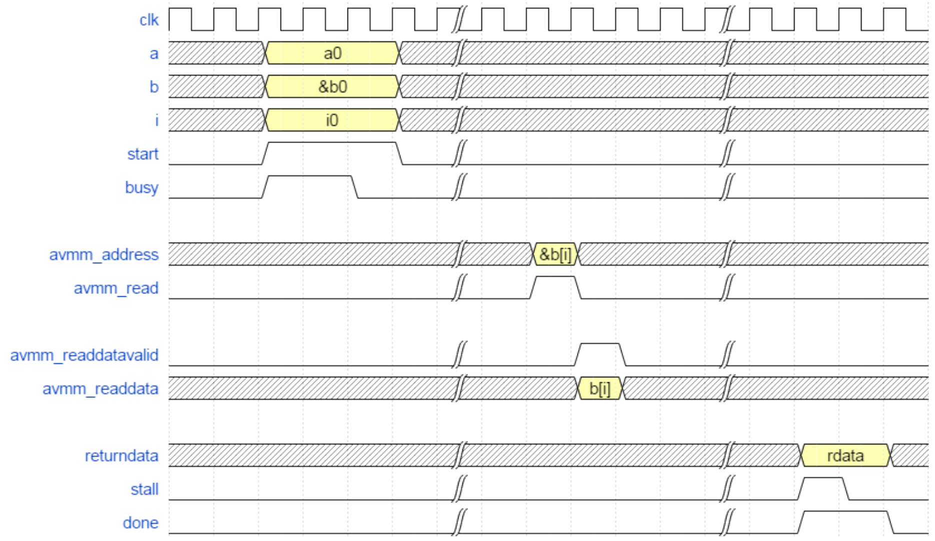


Figure 3. Waveform Diagram of the Signals for the Component dut

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 8. Intel HLS Compiler Standard Edition Streaming Input Interface Template Summary

Template Object or Argument	Description
`ihc::stream_in`	Streaming input interface to the component.
`ihc::buffer`	Specifies the capacity (in words) of the FIFO buffer on the input data that associates with the stream.
`ihc::readyLatency`	Specifies the number of cycles between when the `ready` signal is deasserted and when the input stream can no longer accept new inputs.
`ihc::bitsPerSymbol`	Describes how the data is broken into symbols on the data bus.
`ihc::usesPackets`	Exposes the `startofpacket` and `endofpacket` sideband signals on the stream interface.
`ihc::usesValid`	Controls whether a `valid` signal is present on the stream interface.

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

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

Streaming Output Interfaces

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

Template Object or Argument	Description
`ihc::stream_out`	Streaming output interface from the component.
`ihc::readylatency`	Specifies the number of cycles between when the `ready` signal is deasserted and when the input stream can no longer accept new inputs.

continued...

Template Object or Argument	Description
`ihc::bitsPerSymbol`	Describes how the data is broken into symbols on the data bus.
`ihc::usesPackets`	Exposes the `startofpacket` and `endofpacket` sideband signals on the stream interface.
`ihc::usesReady`	Controls whether a ready signal is present.

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

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

Related Information

- Avalon Interface Specifications
- Supported Math Functions on page 83

4.3. Avalon Memory-Mapped Master Interfaces

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

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

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

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

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

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

where the constructor arguments are as follows:

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

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

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

4.3.2. Implicit and Explicit Examples of Describing a Memory Interface

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

Implicit Example

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

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

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

  dut(x, &y);

  return 0;
}
```

Explicit Example

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

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

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

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

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

component void dut(Master1 &mm1,Master2 &mm2) {
  *mm1 += *mm2;
  *mm2 += mm1[1];
}
int main(void) {
  int x[2] = {0, 1};
  int y = 2;
  Master1 mm_x(x,2*sizeof(int),false);
  Master2 mm_y(&y,sizeof(int),false);

  dut(mm_x, mm_y);

  return 0;
}
```

4.4. 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*.

Table 13. Types of Slave Interfaces

Slave Type	Associated Slave Interface	Read/Write Behavior	Synchronization	Read Latency	Controlling Interface Data Width
Register	The component control and status register (CSR) slave.	The component cannot update these registers from the datapath, so you can read back only data that you wrote in.	Synchronized with the component `start` signal.	Fixed value of 1.	Always 64 bits
Memory (M20K)	Dedicated slave interface on the component.	The component reads from this memory and updates it as it runs. Updates from the component datapath are visible in memory.	Reads and writes to slave memories from outside of the component should occur **only when your component is not executing**. You might experience undefined component behavior if outside slave memory accesses occur when your component is executing. The undefined behavior can occur even if a slave memory access is to a memory address that the component does not access.	Fixed value that is dependent on the component memory access pattern and any attributes or pragmas that you set. See the Component Viewer report in the High-Level Design Report (`report.html`) for the read latency of a specific slave memory argument.	The data width is a multiple of the slave data type, where the multiple is determined by coalescing the internal accesses.

4.4.1. Control and Status Register (CSR) Slave

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

Any arguments that are labeled as `hls_avalon_slave_register_argument` are located in this memory space. The resulting memory map is described in the automatically generated header file `<`*`results`*`>.`*`prj`*`/components/<`*`component_name`*`>_csr.h`. This file also provides the C macros for a master 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:

```
#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
------------|---------|--------------------------|-----------------------------
        0x8 |       W |         {reserved[62:0], |  Write 1 to signal start to
            |         |              start[0:0]} |               the component
------------|---------|--------------------------|-----------------------------
       0x10 |     R/W |         {reserved[62:0], |      0 - Disable interrupt,
            |         |   interrupt_enable[0:0]} |        1 - Enable interrupt
------------|---------|--------------------------|-----------------------------
       0x18 |  R/Wclr |         {reserved[61:0], | Signals component completion
            |         |               done[0:0], |       done is read-only and
            |         |   interrupt_status[0:0]} | interrupt_status is write 1
            |         |                          |                    to clear
------------|---------|--------------------------|-----------------------------
       0x20 |       R |       {returndata[63:0]} |        Return data (0 of 3)
------------|---------|--------------------------|-----------------------------
       0x28 |       R |     {returndata[127:64]} |        Return data (1 of 3)
```

```
------------|---------|--------------------------|----------------------------
       0x30 |       R |    {returndata[191:128]} |        Return data (2 of 3)
------------|---------|--------------------------|----------------------------
       0x38 |     R/W |          {reserved[31:0], |                  Argument y
            |         |                  y[31:0]} |
------------|---------|--------------------------|----------------------------
       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
            |         |            char_arg[7:0]} |

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


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

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

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

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

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

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

```
#define MYCOMP_XYZ_CSR_START_OFFSET (0)

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

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


#endif /* __MYCOMP_XYZ_CSR_REGS_H__ */
```

4.4.2. Slave Memories

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

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

- The master interface has a single port. If the component has multiple load-store sites, arbitration on that port might create 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.

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.

Argument Label	Description
`hls_avalon_slave_memory_argument`	Implement the argument, in on-chip memory blocks, which can be read from or written to over a dedicated slave interface.

4.5. Component Invocation Interface 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 14. Intel HLS Compiler Standard Edition Component Invocation Interface Argument Summary

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

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:

- 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.

Argument Label	Description
`hls_stable_argument`	A stable argument is an argument that does not change while there is live data in the component (that is, between pipelined function invocations).

4.7. Global Variables

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

If you access more than one global variable, each global variable uses the same Avalon MM Master interface, which 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

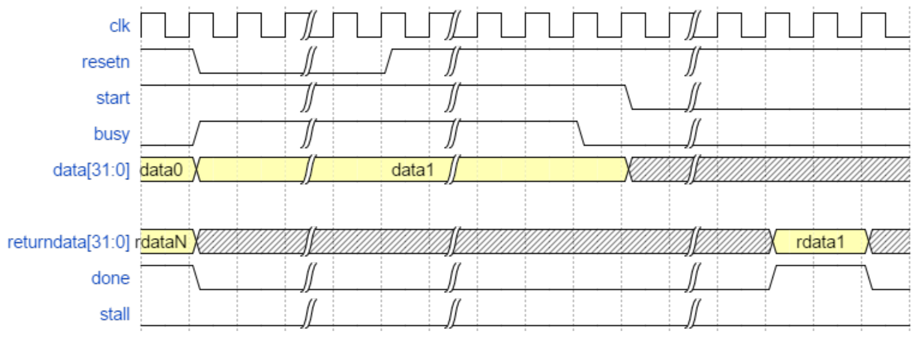
For your HLS component, the reset assertion can be asynchronous but the reset deassertion must be synchronous.

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:

```
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:

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

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

5. Component Memories (Memory Attributes)

The Intel High Level Synthesis (HLS) Compiler builds a hardware memory system using FPGA memory resources (such as block RAMs) for any local, constant, static variable or array, and slave memory declared in your code. Memory accesses are mapped to load-store units (LSUs), which transact with the hardware memory through its ports.

Figure 4. A Basic Memory Configuration Inferred by the Intel HLS Compiler

The following diagram shows a basic memory configuration:

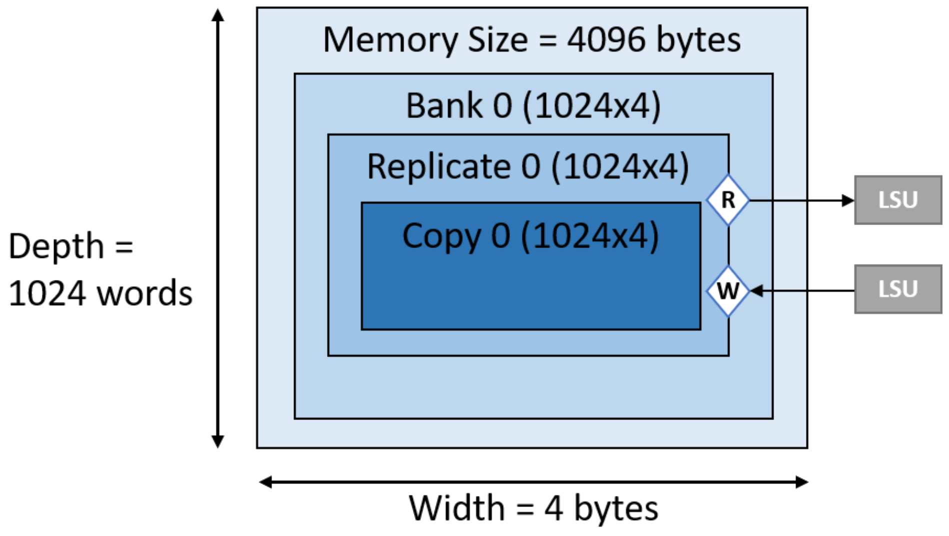


Figure 5. A Memory System With Two Memory Banks

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

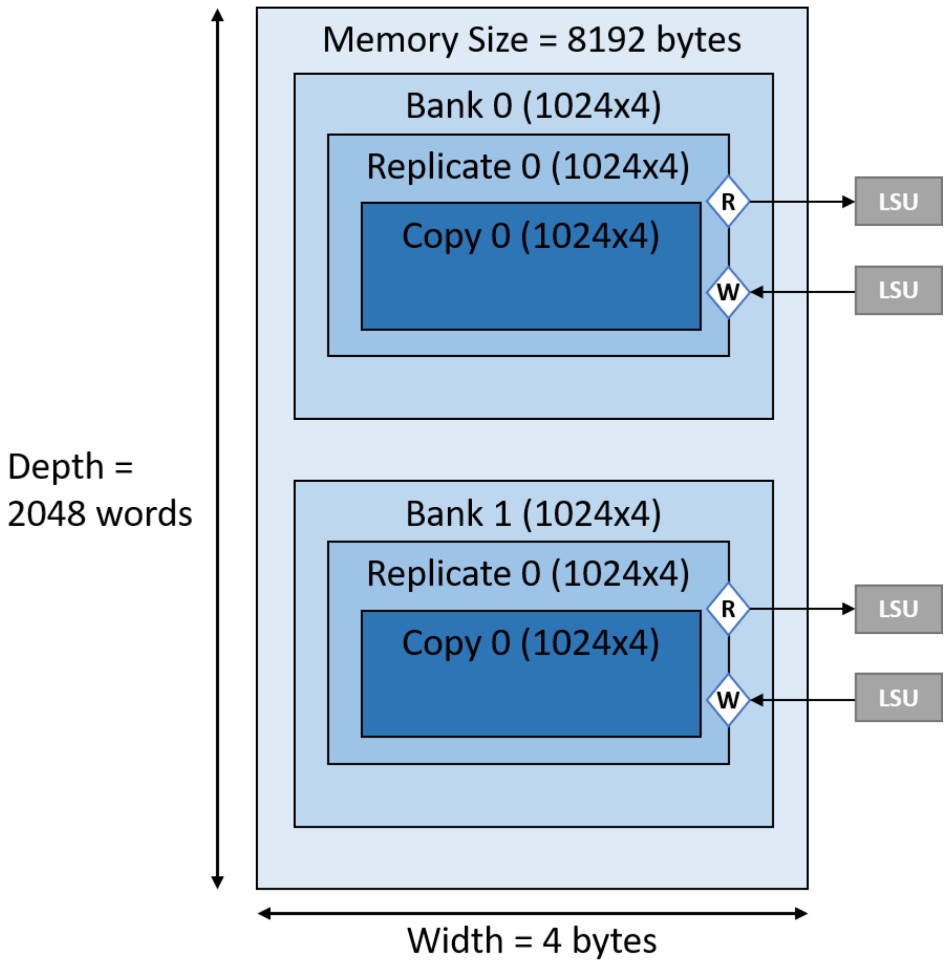
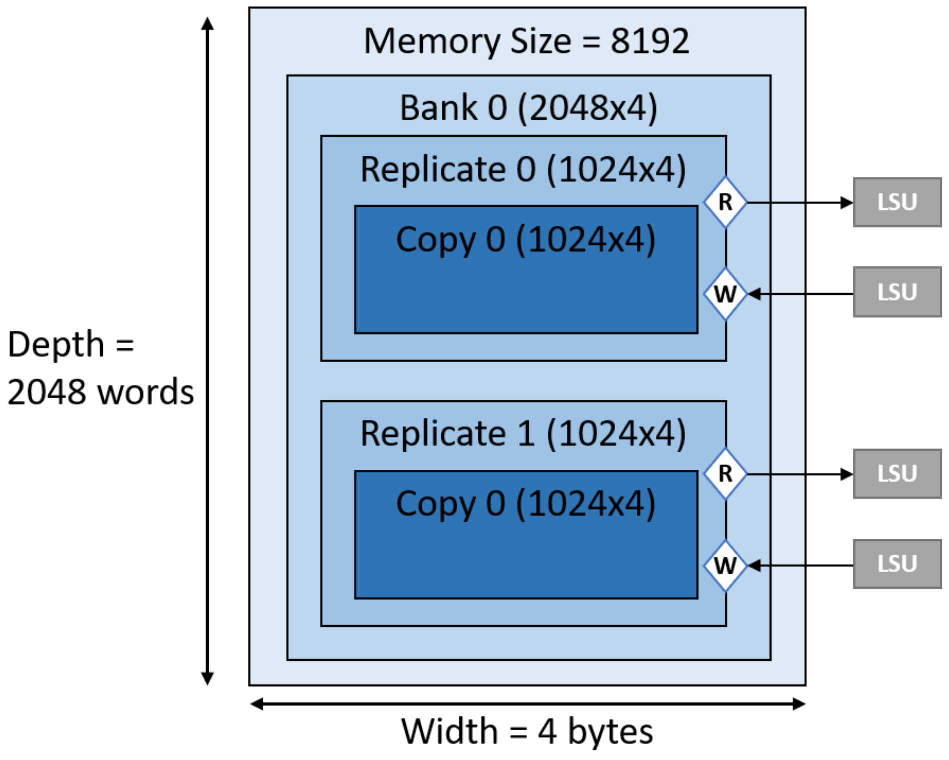


Figure 6. A Memory System With Two Replicates

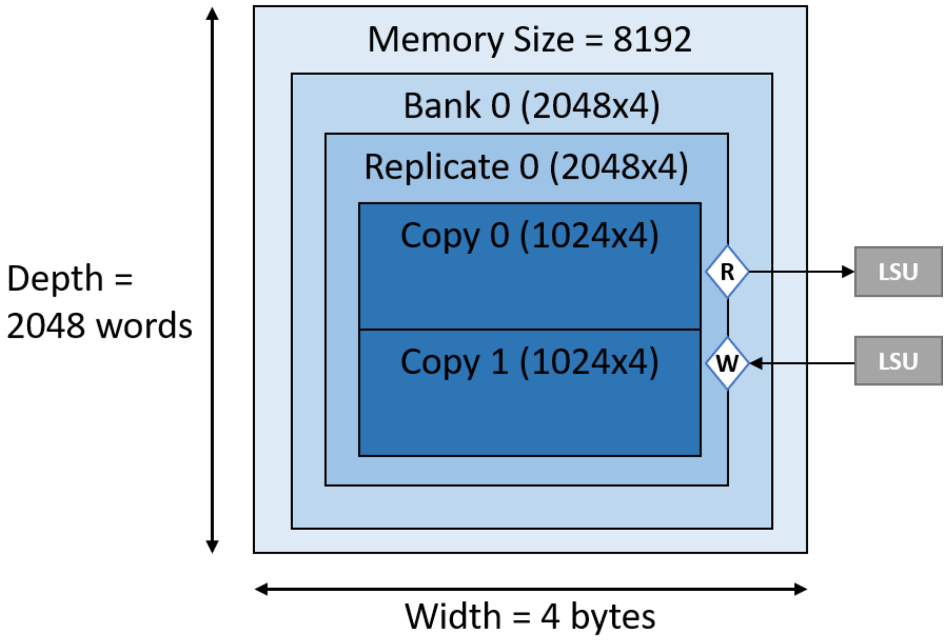
A memory bank can contain one or more memory replicates. The compiler might create memory replicates to create more read ports. Having more read ports allows faster access to your memory system if you have many read operations.

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



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

Figure 7. A Memory System With Two Private Copies



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

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

Component Memory Attributes

Apply the component memory attributes to local 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 15. Intel HLS Compiler Standard Edition Component Memory Attributes Summary

Memory Attribute	Description
`hls_register`	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.
`hls_memory`	Forces a variable or array to be implemented as embedded memory.

continued...

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

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 ($N_{\mathbf{banks}}$).
- 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 ($N_{\mathbf{bits}}$).

These attributes are subject to the following constraints:

- $N_{\textbf{banks}} \times W = S$

  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.
- $N_{\textbf{banks}}$ must be a power of 2 value.
- $N_{\textbf{banks}} = 2^{N_{\textbf{bits}}}$

  $N_{\textbf{bits}}$ bank-selection bits that are required to address $N_{\textbf{banks}}$ 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 Intel HLS Compiler Standard Edition supports function-scope static variables with the same semantics as in C and C++.

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

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

Static Variable Initialization

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

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

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

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

`hls_init_on_reset`

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

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

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

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

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

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

`hls_init_on_powerup`

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

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

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

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

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

For information about resetting your component, see Reset Behavior on page 27.

6. Loops in Components

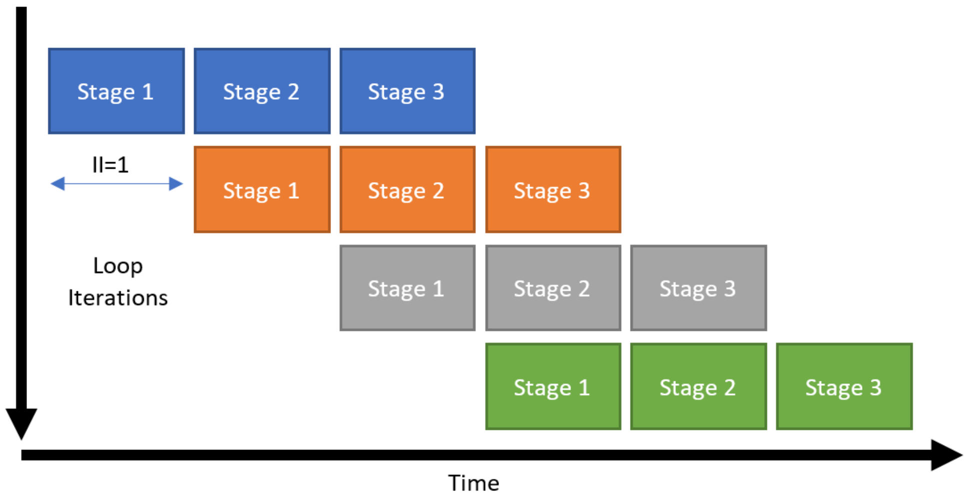
The Intel HLS Compiler attempts to pipeline loops to maximize throughput of the various components that you define.

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 8. Pipelined loop with three stages and four iterations

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



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

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

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

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

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

Compiler Pragmas Controlling Loop Pipelining

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

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

Incorrect (produces a compile-time error)	Correct

`#pragma ivdep`
`TEST_LOOP: for(int idx = 0; idx < counter; idx++) {...}`
	`TEST_LOOP:`
	`#pragma ivdep`
	`for(int idx = 0; idx < counter; idx++) {...}`

Table 16. Intel HLS Compiler Standard Edition Loop Pragmas Summary

Pragma	Description
`ii`	Forces a loop to have a loop initiation interval (II) of a specified value.
`ivdep`	Ignores memory dependencies between iterations of this loop.
`loop_coalesce`	Tries to fuse all loops nested within this loop into a single loop.
`max_concurrency`	Limits the number of iterations of a loop that can simultaneously execute at any time.
`unroll`	Unrolls the loop completely or by a number of times.

6.1. Loop Initiation Interval (ii Pragma)

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

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

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

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

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

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

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

Example

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

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

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

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

6.2. Loop-Carried Dependencies (ivdep Pragma)

When compiling your components, the HLS compiler generates hardware to avoid any data hazards between load and store instructions to component memories, slave memories, and external memories (through Avalon MM 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 an `mm_master` object

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

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

Use Case 1:

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

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

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.

```
// No loop-carried dependencies for A array accesses
// Compiler inserts hardware that reinforces dependency constraints for B
#pragma ivdep array(A)
for(int i = 0; i < N; i++) {
    A[i] = A[i - X[i]];
    B[i] = B[i - Y[i]];
}

// No loop-carried dependencies for array A inside struct
#pragma ivdep array(S.A)
for(int i = 0; i < N; i++) {
    S.A[i] = S.A[i - X[i]];
}

// No loop-carried dependencies for array A inside the struct pointed by S
#pragma ivdep array(S->X[2][3].A)
for(int i = 0; i < N; i++) {
    S->X[2][3].A[i] = S.A[i - X[i]];
}

// No loop-carried dependencies for A and B because ptr aliases
// with both arrays
```

```
int *ptr = select ? A : B;
#pragma ivdep array(ptr)
for(int i = 0; i < N; i++) {
    A[i] = A[i - X[i]];
    B[i] = B[i - Y[i]];
}

// No loop-carried dependencies for A because ptr only aliases with A
int *ptr = &A[10];
#pragma ivdep array(ptr)
for(int i = 0; i < N; i++) {
    A[i] = A[i - X[i]];
    B[i] = B[i - Y[i]];
}
```

6.3. Loop Coalescing (`loop_coalesce` Pragma)

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

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

To coalesce nested loops, specify the pragma as follows:

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

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

For example, consider the following set of nested loops:

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

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

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

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

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

Example

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

Consider a simple nested loop written as follows:

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

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

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

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

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

6.4. Loop 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 Standard Edition 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 Component Memory Viewer, where it graphically shows the number of private copies.

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

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

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

You can control the number of private copies created for a component memory accessed 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).

6.5. Loop Unrolling (`unroll` Pragma)

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

Example code:

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

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`).

7. Component Concurrency

The Intel HLS Compiler assumes that you want a fully pipelined data path in your component. In the C++ implementation, think of a fully pipelined data path as calling a function multiple times before the first call has returned (see also Figure 8 on page 35 and Intel HLS Compiler Pipeline Approach on page 9). The behavior of multiple component invocations within the synthesized data path is subject to the concurrency model, so the Intel HLS Compiler might not be able to deliver a component with a component initiation interval (II) of 1, or even any pipelining.

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

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

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

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

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

7.2. Concurrency Control (`hls_max_concurrency` Attribute)

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

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

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

hls_max_concurrency(3)
```

```
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 40.

8. Arbitrary Precision Math Support

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

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

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

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

Table 17. Arbitrary Precision Data Types Supported by the Intel HLS Compiler Standard Edition

Data Type	Intel Header File	Description
`ac_int`	`HLS/ac_int.h`	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`	`HLS/ac_fixed.h`	Arbitrary-precision fixed-point number support To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_constructor`
	`HLS/ac_fixed_math.h`	Support for some nonstandard math functions for arbitrary-precision fixed-point data types To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_math_library`

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

- `ac_fixed` data type

  Include the `HLS/ac_fixed_math.h` header file

Advantages of Arbitrary Precision Data Types

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

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

Limitations of AC Data Types

The AC data types have the following limitations:

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

  To compile AC data types with GCC or MSVC, use the reference AC data types headers also provided with he Intel HLS Compiler. For details, see AC Data Types and Native Compilers on page 49.

Related Information

AC Datatypes at HLSLibs

8.1. Declaring ac_int Data Types

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

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

```
#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:

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

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

8.1.1. Important Usage Information on the ac_int Data Type

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

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

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

8.2. Integer Promotion and ac_int Data Types

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

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

Send Feedback

- Both operands are standard integer types (int, short, long, unsigned char, or signed char):

  If both operands are of standard integer type (for example char or short) operations, integers are promoted following the C/C++ standard. That is, the operation is carried out in the data type and size of the largest operand, but at least 32 bits. The expression returns the result in the larger data type.
- Both operands are ac_int data types:

  If both operands are ac_int data types, operations are carried out in the smallest ac_int data type needed to contain all values. For example, the multiplication of two 8-bit ac_int values is carried out as an 16-bit operation. The expression returns the result in that type.
- One operand is a standard integer type and one operand is an ac_int type:

  If the expression has one standard data type and one ac_int type, the rules for ac_int data type promotion apply. The resulting expression type is always an ac_int data type. For example, if you add a short data type and an ap_int<16> data type, the resulting data type is ac_int<17>.

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

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

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

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

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

8.3. Debugging Your Use of the ac_int Data Type

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

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

Table 18. Intel HLS Compiler Standard Edition ac_int Debugging Tools Summary

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

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

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

8.4. Declaring ac_fixed Data Types

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

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

```
#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.
Q	The quantization mode that determines how to handle values where the generated precision (number of decimal places) exceeds the number of bits available in the variable to represent the fractional part of the number. For a list of quantization modes and their descriptions, see "2.1. Quantization and Overflow" in *Mentor Graphics Algorithmic C (AC) Datatypes*, which is available in the following file: `<`*`quartus_installdir`*`>/hls/include/ref/ac_datatypes_ref.pdf`.

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

For a list of overflow modes and their descriptions, , see "2.1. Quantization and Overflow" in *Mentor Graphics Algorithmic C (AC) Datatypes*, which is available in the following file: `<`*`quartus_installdir`*`>/hls/include/ref/ac_datatypes_ref.pdf`.

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

8.5. AC Data Types and Native Compilers

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

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

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

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

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

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

9. Component Target Frequency

You can specify component target frequency either in the `i++` command by specifying the `--clock` option.

For details about the `--clock` option, see Command Options Affecting Compiling on page 51.

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

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

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

10. Intel High Level Synthesis Compiler Standard Edition Compiler Reference Summary

10.1. Intel HLS Compiler Standard Edition i++ Command-Line Arguments

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

General i++ Command Options

Option	Description
`--debug-log`	Generate the compiler diagnostics log.
`-h`, `--help`	List compiler command options along with brief descriptions.
`-o` *result*	Place compiler output into the <*result*> executable and the <*result*>`.prj` directory.
`-v`	Display messages describing the progress of the compilation.
`--version`	Display compiler version information.

Command Options Affecting Compiling

Option	Default Value	Description
`-c`		Preprocess, parse, and generate object files.
`--component` *component name*		Comma-separated list of function names to synthesize to RTL. To use this option, your component must be configured with C-linkage using the `extern "C"` specification. For example: `extern "C" 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.
`-D`*macro*`[=`*val*`]`		Define a <*macro*> with <*val*> as its value.
`-g`		Generate debug information (default option).
`-g0`		Do not generate debug information.
`-I`*dir*		Add directory <*dir*> to the end of the main include path.
`-march=[x86-64` | *FPGA_family* | *FPGA_part_number*`]`	`x86-64`	Generate code for an emulator flow (`x86-64`) or for the specified FPGA family or FPGA part number.

continued...

Option	Default Value	Description
`--promote-integers`		Use extra FPGA resources to mimic g++ integer promotion. In Pro Edition, the compiler always promotes integers for standard types. Use the `ac_int` datatypes if you want smaller (or larger) datatypes. To learn more, review the tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/best_practices/integer_promotion`
`--quartus-compile`		Run the HDL generated through Intel Quartus Prime to generate accurate f_{MAX} and area estimates. Your component is not expected to cleanly close timing.
`--simulator` *simulator_name*	`modelsim`	Specifies the simulator you are using to perform verification. This command option can take the following values for <*simulator_name*>: *`modelsim`* Use ModelSim for component verification. *none* Disable verification. That is, generate RTL for components without the test bench. If you do not specify this option, `--simulator modelsim` is assumed.
`--fpc`		Remove intermediate rounding and conversion when possible. To learn more, review the following tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/best_practices/floating_point_ops`
`--fp-relaxed`		Relax the order of floating point arithmetic operations. To learn more, review the following tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/best_practices/floating_point_ops`

`--clock` *clock target*	`240 MHz`	Optimize the RTL for the specified clock frequency or period.
		For example:
		`i++ -march="Arria 10" test.cpp --clock 100MHz`
		`i++ -march="Arria 10" test.cpp --clock 10ns`

Command Options Affecting Linking

Option	Default Value	Description
`-ghdl`		Enable full debug visibility and logging of all HDL signals in simulation.
`-L`*dir*		(Linux only) Add directory <*dir*> to the list of directories to be searched for library files specified with the -l option.
`-l`*library*		(Linux only) Use the library name <*library*> when linking.
`--x86-only`		Create only the testbench executable (`<`*`result`*`>.out/<`*`result`*`>.exe`).
`--fpga-only`		Create only the `<`*`result`*`>.prj` directory and its contents.

10.2. Intel HLS Compiler Standard Edition Header Files

Coding your component to be compiled by the Intel HLS Compiler requires you to include the `hls.h` header file. Other header files provided with the Intel HLS Compiler provide FPGA-optimized implementations of certain C and C++ functions.

Table 19. Intel HLS Compiler Standard Edition Header Files Summary

HLS Header File	Description
`HLS/hls.h`	Required for component identification and component parameter interfaces.
`HLS/math.h`	Includes FPGA-specific definitions for the math functions from the `math.h` for your operating system.
`HLS/extendedmath.h`	Includes additional FPGA-specific definitions of math functions not in `math.h`.
`HLS/ac_int.h`	Provides FPGA-optimized arbitrary width integer support.
`HLS/ac_fixed.h`	Provides FPGA-optimized arbitrary precision fixed point support.
`HLS/ac_fixed_math.h`	Provides FPGA-optimized arbitrary precision fixed point math functions.
`HLS/stdio.h`	Provides `printf` support for components so that `printf` statements work in x86 emulations, but are disabled in component when compiling to an FPGA architecture.
`"HLS/iostream"`	(Linux only) Provides `cout` and `cerr` support for components so that `cout` and `cerr` statements work in x86 emulations, but are disabled in component when compiling to an FPGA architecture.

`hls.h` Header File

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

Description Required for component identification and component parameter interfaces.

`math.h` Header File

Syntax `#include "HLS/math.h"`

Description Includes FPGA-specific definitions for the math functions from the `math.h` for your operating system.

To learn more, review the following tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/best_practices/single_vs_double_precision_math`.

`extendedmath.h` Header File

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

Description Includes additional FPGA-specific definitions of math functions not in `math.h`.

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

`ac_int.h` Header File

Syntax `#include "HLS/ac_int.h"`

Description Intel HLS Compiler version of `ac_int` header file.

Provides FPGA-optimized arbitrary width integer support.

To learn more, review the following tutorials:

- `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_int_basic_ops`
- `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_int_overflow`
- `<quartus_installdir>/hls/examples/tutorials/best_practices/struct_interfaces`

`ac_fixed.h` Header File

Syntax `#include "HLS/ac_fixed.h"`

Description Intel HLS Compiler version of the `ac_fixed` header file.

Provides FPGA-optimized arbitrary precision fixed point support.

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

`ac_fixed_math.h` Header File

Syntax `#include "HLS/ac_fixed_math.h"`

Description Intel HLS Compiler version of the `ac_fixed_math` header file.

Provides FPGA-optimized arbitrary precision fixed point math functions.

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

`stdio.h` Header File

Syntax `#include "HLS/stdio.h"`

Description	Provides `printf` support for components so that `printf` statements work in x86 emulations, but are disabled in component when compiling to an FPGA architecture.

`iostream` Header File (Linux only)

Syntax	`#include HLS/iostream`
Description	Provides `cout` and `cerr` support for components so that `cout` and `cerr` statements work in x86 emulations, but are disabled in component when compiling to an FPGA architecture.

.

10.3. Standard Edition Compiler-Defined Preprocessor Macros

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

Table 20. Macro Definition for __INTELFPGA_COMPILER__

Tool Invocation	__INTELFPGA_COMPILER__
`g++` or `cl`	Undefined
`-march=x86-64`	
`-march="<FPGA_family_or_part_number>"`	

Table 21. Macro Definition for HLS_SYNTHESIS

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

10.4. Intel HLS Compiler Standard Edition Keywords

Table 22. Intel HLS Compiler Keywords

Feature	Description
`component`	Indicates that a function is a component. Example: `component void foo()`

10.5. Intel HLS Compiler Standard Edition Simulation API (Testbench Only)

Table 23. Intel HLS Compiler Standard Edition Simulation API (Testbench only) Summary

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

`ihc_hls_enqueue` Function

Syntax `ihc_hls_enqueue(void* retptr, void* funcptr, /*function arguments*/)`

Description This function enqueues one invocation of an HLS component. The return value is stored in the first argument which should be a pointer to the return type. The component is not run until the `ihc_hls_component_run_all()` is invoked.

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/usability/enqueue_call`.

`ihc_hls_enqueue_noret` Function

Syntax `ihc_hls_enqueue_noret(void* funcptr, /*function arguments*/)`

Description This function enqueues one invocation of an HLS component. This function should be used when the return type of the HLS component is void. The component is not run until the `ihc_hls_component_run_all()` is invoked.

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/usability/enqueue_call`.

`ihc_hls_component_run_all` Function

Syntax `ihc_hls_component_run_all (void* funcptr)`

Description This function accepts a pointer to the HLS component function. When run, all enqueued invocations of the component will be pushed into the component in the HDL simulator as quickly as the component can accept new invocations.

To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/usability/enqueue_call`.

`ihc_hls_sim_reset` Function

Syntax `int ihc_hls_sim_reset(void)`

Description This function sends a reset signal to the component during automated simulation. It returns 1 if the reset was exercised or 0 otherwise.

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

Simulation API Code Example

```
component int foo(int val) {
  // function definition
}

component void bar (int val) {
  // function definition
}
int main() {
  // …….
  int input = 0;
  int res[5];
  ihc_hls_enqueue(&res, &foo, input);
  ihc_hls_enqueue_noret(&bar, input);
  input = 1;
  ihc_hls_enqueue(&res, &foo, input);
  ihc_hls_enqueue_noret(&bar, input);
  ihc_hls_component_run_all(&foo);
  ihc_hls_component_run_all(&bar);
}
```

10.6. Intel HLS Compiler Standard Edition Component Memory Attributes

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

Table 24. Intel HLS Compiler Standard Edition Component Memory Attributes Summary

Memory Attribute	Description
`hls_register`	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.
`hls_memory`	Forces a variable or array to be implemented as embedded memory.

continued...

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

`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 following 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.

`hls_singlepump` Memory Attribute

Syntax	`hls_singlepump`
Constraints	N/A
Default Value	Based on the memory access pattern inferred by the compiler.
Description	Specifies that the memory implementing the variable or array must be clocked at the same rate as the component accessing the memory. To learn more, review the following tutorial: `<quartus_installdir>/hls/examples/QRD`.

`hls_doublepump` Memory Attribute

Syntax	`hls_doublepump`
Constraints	N/A
Default Value	Based on the memory access pattern inferred by the compiler.
Description	Specifies that the memory implementing the variable or array must be clocked at twice the rate of the component accessing the memory.

`hls_numbanks` Memory Attribute

Syntax	`hls_numbanks(N)`
Constraints	This attribute is subject to constraints outlined in Constraints on Attributes for Memory Banks on page 32.
Default Value	Based on the memory access pattern inferred by the compiler.
Description	Specifies that the memory implementing the variable or array must have *N* banks, where *N* is a power-of-two constant number.

`hls_bankwidth` Memory Attribute

Syntax	`hls_bankwidth(N)`
Constraints	This attribute is subject to constraints outlined in Constraints on Attributes for Memory Banks on page 32.
Default Value	Based on the memory access pattern inferred by the compiler.

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

`hls_bankbits` Memory Attribute

Syntax	`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 32.
Default Value	Based on the memory access pattern inferred by the compiler.
Description	Forces the memory system to split into 2^{n+1} banks, with {b_0, b_1, ..., b_n} forming the bank-select bits.

Important: b_0, b_1, ..., 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, ..., b_n are mapped to array index bits `0` to `n-1` in the memory bank implementation.

`hls_numports_readonly_writeonly` Memory Attribute

Syntax	`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 variable or array 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	`hls_simple_dual_port_memory`
Constraints	N/A
Default Value	N/A
Description	Specifies that the memory implementing the variable or array should have no port that services both reads and writes.

`hls_merge` (*depthwise*) Memory Attribute

Syntax	`hls_merge("`*`mem_name`*`", "depth")`
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 depth-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 following tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/component_memories/depth_wise_merge`.

`hls_merge` (*widthwise*) Memory Attribute

Syntax	`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 following tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/component_memories/width_wise_merge`.

`hls_init_on_reset` Memory Attribute

Syntax	`hls_init_on_reset`
Constraints	N/A
Default Value	Default behavior for static variables.
Description	Forces the static variable inside the component to be initialized 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 following tutorial: <*quartus_installdir*>`/hls/examples/tutorials/component_memories/static_var_init`.

`hls_init_on_powerup` Memory Attribute

Syntax	`hls_init_on_powerup`
Constraints	N/A
Default Value	N/A
Description	Sets the component memory implementing the static variable to initialize on power-up when the FPGA is programmed. When the component is reset, the component memory is not reset back to the initialized value of the static. To learn more, review the following tutorial: <*quartus_installdir*>`/hls/examples/tutorials/component_memories/static_var_init`.

`hls_max_concurrency` Memory Attribute

Syntax	`hls_max_concurrency(`*`N`*`)`
Constraints	N/A
Default Value	N/A
Description	Specifies that the memory can have a maximum *N* private copies to allow *N* concurrent iterations of a loop at any given time, where *N* is rounded up to the nearest power of 2. Apply this attribute only when the scope of a variable (through its declaration or access pattern) is limited to a loop. If the loop has the `max_concurrency` pragma applied to it, the number of private copies created is the lesser of the `hls_max_concurrency` memory attribute value and the `max_concurrency` pragma value.

10.7. Intel HLS Compiler Standard Edition Loop Pragmas

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

Table 25. Intel HLS Compiler Standard Edition Loop Pragmas Summary

Pragma	Description
`ii`	Forces a loop to have a loop initiation interval (II) of a specified value.
`ivdep`	Ignores memory dependencies between iterations of this loop.

continued...

Send Feedback

Pragma	Description
`loop_coalesce`	Tries to fuse all loops nested within this loop into a single loop.
`max_concurrency`	Limits the number of iterations of a loop that can simultaneously execute at any time.
`unroll`	Unrolls the loop completely or by a number of times.

ii Loop Pragma

Syntax `#pragma ii N`

Description Forces the loop to which you apply this pragma to have a loop initiation interval (II) of <*N*>, where <*N*> is a positive integer value.

Forcing a loop II value can have an adverse effect on the f_{MAX} of your component because using this pragma to get a lower loop II combines pipeline stages together and creates logic with a long propagation delay.

Using this pragma with a larger loop II inserts more pipeline stages and can give you a better component f_{MAX} value.

Example:

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

ivdep Loop Pragma

Syntax `#pragma ivdep safelen(N) array(array_name)`

Description Tells the compiler to ignore memory dependencies between iterations of this loop.

It can accept an optional argument that specifies the name of the array. If `array` is not specified, all component memory dependencies are ignored. If there are loop-carried dependencies, your generated RTL produces incorrect results.

The `safelen` parameter specifies the dependency distance. The dependency distance is the number of iterations between successive load/stores that depend on each other. It is safe to not include `safelen` is only when the dependence distance is infinite (that is, there are no real dependencies).

Example:

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

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

`loop_coalesce` Loop Pragma

Syntax `#pragma loop_coalesce N`

Description Tells the compiler to try to fuse all loops nested within this loop into a single loop. This pragma accepts an optional value *N* which indicates the number of levels of loops to coalesce together.

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

`max_concurrency` Loop Pragma

Syntax `#pragma max_concurrency N`

Description This pragma limits the number of iterations of a loop that can simultaneously execute at any time.

This pragma is useful mainly when private copies of are created to improve the throughput of the loop. This is mentioned in the details pane for the loop in the Loop Analysis pane and the Bank view of the Component 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.

10.8. Intel HLS Compiler Standard Edition Component Attributes

Table 26. Intel HLS Compiler Standard Edition Component Attributes Summary

Feature	Description
`hls_max_concurrency`	Request more copies of the component memory so that the component can run multiple invocations in parallel.

hls_max_concurrency Component Attribute

Syntax `hls_max_concurrency(<N>)`

Description In some cases, the concurrency of a component is limited to 1. This limit occurs when the generated hardware cannot be shared across component invocations. For example, when using component memories for a non-static variable.

You can use this attribute to request more copies of the component memory so that the component can run multiple invocations in parallel.

This attribute can accept any non-negative whole number, including `0`.

Value greater than 0	A value greater than 0 indicates how many copies of the component memory to instantiate as well as how many component invocations can be in flight at once.
Value equal to 0	Setting `hls_max_concurrency` to a value of `0` is useful in cases when there is no component memory but the component still has a poor dynamic loop initiation interval (II) even if you believe your component II should be 1. You can review the II for loops in your component in the high level design report.

To learn more, review the design example: <*quartus_installdir*>/hls/examples/inter_decim_filter.

Example

```
hls_max_concurrency(2)
component void foo(ihc::stream_in<int> &data_in,
      ihc::stream_out<int> &data_out) {
 int arr[N];
 for (int i = 0; i < N; i++) {
   arr[i] = data_in.read();
 }
```

```
  // Operate on the data and modify in place
  for (int i = 0; i < N; i++) {
    data_out.write(arr[i]);
  }
}
```

10.9. Intel HLS Compiler Standard Edition Component Default Interfaces

Table 27. Intel HLS Compiler Standard Edition Default Interfaces

Feature	Description
Component invocation interface (component call and return)	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.
Scalar parameter interface (passed by value)	Scalar parameters are implemented as input conduits that are synchronized with the component invocation interface.
Pointer parameter interface (passed by reference)	Pointer parameters are implemented as an implicit Avalon Memory-Mapped Master (`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.

10.10. Intel HLS Compiler Standard Edition Component Invocation Interface Arguments

Table 28. Intel HLS Compiler Standard Edition Component Invocation Interface Argument Summary

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

`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

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

`hls_avalon_slave_component` Argument

Description

The `start`, done, and `returndata` (if applicable) signals are registered in the component slave memory map.

These component must take either slave, stream, or stable arguments. If you do not specify these types of arguments, the compiler generates an error message when you compile this component.

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

Top-level module ports

- Avalon-MM slave interface
- `irq_done` signal

Example

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

`hls_always_run_component` 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	`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;}
```

Related Information

Control and Status Register (CSR) Slave on page 21

10.11. Intel HLS Compiler Standard Edition Component Macros

Table 29. Intel HLS Compiler Standard Edition Component Macros Summary

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

`hls_conduit_argument` Component Macro

Syntax `hls_conduit_argument`

Description

This is the default interface for scalar arguments.

The compiler implements the argument as an input conduit that is synchronous to the component's call (start and busy).

Example

```
component void foo(
  hls_conduit_argument int b)
```

`hls_avalon_slave_register_argument` Component Macro

Syntax `hls_avalon_slave_register_argument`

Description The compiler implements the argument as a register that can be read from and written to over an Avalon-MM slave interface. The argument will be read into the component's pipeline, similar to the conduit implementation. The implementation is synchronous to the start and busy interface.

Changes to the value of this argument made by the component data path will not be reflected on this register.

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

Example

```
component void foo(
  hls_avalon_slave_register_argument int b)
```

`hls_avalon_slave_memory_argument` Component Macro

Syntax `hls_avalon_slave_memory_argument(`*`N`*`)`

Description The compiler implements the argument, where *N* specifies the size of the memory in bytes, in on-chip memory blocks, which can be read from or written to over a dedicated slave interface. The generated memory has the same architectural optimizations as all other internal component memories (such as banking or coalescing).

If the compiler performs static coalescing optimizations, the slave interface data width is the coalesced width. This attribute applies only to a pointer argument.

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

Example

```
component void foo(
   hls_avalon_slave_memory_argument(128*sizeof(int)) int *a)
```

`hls_stable_argument` Component Macro

Syntax `hls_stable_argument`

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

Changing a stable argument during component execution results in undefined behavior; each use of the stable argument might be the old value or the new value, but with no guarantee of consistency. The same variable in the same invocation can appear with multiple values. Using stable arguments, where appropriate, might save a significant number of registers in a design.

Stable arguments can be used with conduits, mm_master interfaces, and slave_registers.

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

Example

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

10.12. Intel HLS Compiler Standard Edition Streaming Input Interfaces

Use the `stream_in` object and template arguments to explicitly declare Avalon Streaming (ST) input interfaces. You can also use the `stream_in` Function APIs.

Table 30. Intel HLS Compiler Standard Edition Streaming Input Interface Template Summary

Template Object or Argument	Description
`ihc::stream_in`	Streaming input interface to the component.
`ihc::buffer`	Specifies the capacity (in words) of the FIFO buffer on the input data that associates with the stream.
`ihc::readyLatency`	Specifies the number of cycles between when the `ready` signal is deasserted and when the input stream can no longer accept new inputs.
`ihc::bitsPerSymbol`	Describes how the data is broken into symbols on the data bus.
`ihc::usesPackets`	Exposes the `startofpacket` and `endofpacket` sideband signals on the stream interface.
`ihc::usesValid`	Controls whether a `valid` signal is present on the stream interface.

`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_packet_ready_valid`
- `<quartus_installdir>/hls/examples/tutorials/interfaces/mulitple_stream_call_sites`

`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. 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). This parameter is available only on input streams.

`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 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. Data is always broken down in little 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::usesValid` Template Argument

Syntax	`ihc::usesValid<`*`value`*`>`
Valid Values	`true` or `false`
Default Value	`true`
Description	Controls whether a `valid` signal is present on the stream interface. If `false`, the upstream source must provide valid data on every cycle that `ready` is asserted. This is equivalent to changing the stream read calls to `tryRead` and assuming that `success` is always `true`. If set to `false`, `buffer` and `readyLatency` must be 0.

Intel HLS Compiler Standard Edition Streaming Input Interface `stream_in` Function APIs

Table 31. Intel HLS Compiler Standard Edition Streaming Input Interface `stream_in` Function APIs

Function API	Description
`T read()`	Blocking read call to be used from within the component
`T read(bool& sop, bool& eop)`	Available only if `usesPackets<true>` is set. Blocking read with out-of-band `startofpacket` and `endofpacket` signals.

continued...

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

Intel HLS Compiler Streaming Input Interfaces Code Example

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

```
 // Blocking read
void foo (ihc::stream_in<int> &a) {
 int x = a.read();

}
 // Non-blocking read
void foo_nb (ihc::stream_in<int> &a) {
 bool success = false;
 int x = a.tryRead(success);

 if (success) {
 // x is valid
 }
}

int main() {
 ihc::stream_in<int> a;
 ihc::stream_in<int> b;
 for (int i = 0; i < 10; i++) {
  a.write(i);
  b.write(i);
 }
 foo(a);
 foo_nb(b);
}
```

10.13. Intel HLS Compiler Standard Edition Streaming Output Interfaces

Use the `stream_out` object and template arguments to explicitly declare Avalon Streaming (ST) output interfaces. You can also use the `stream_out` Function APIs.

Table 32. Intel HLS Compiler Standard Edition Streaming Output Interface Template Summary

Template Object or Argument	Description
`ihc::stream_out`	Streaming output interface from the component.
`ihc::readylatency`	Specifies the number of cycles between when the `ready` signal is deasserted and when the input stream can no longer accept new inputs.
`ihc::bitsPerSymbol`	Describes how the data is broken into symbols on the data bus.
`ihc::usesPackets`	Exposes the `startofpacket` and `endofpacket` sideband signals on the stream interface.
`ihc::usesReady`	Controls whether a ready signal is present.

`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_packet_ready_valid`
- *`<quartus_installdir>`*`/hls/examples/tutorials/interfaces/ mulitple_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. Data is always broken down in little 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::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 Standard Edition Streaming Output Interface `stream_out` Function APIs

Table 33. Intel HLS Compiler Standard Edition Streaming Output Interface `stream_out` Function Call APIs

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

Intel HLS Compiler Streaming Output Interfaces Code Example

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

```
// Blocking write
void foo (ihc::stream_out<int> &a) {
  static int count = 0;
  for(int idx = 0; idx < 5; idx ++){
    a.write(count++); // Blocking write
  }
}

// Non-blocking write
void foo_nb (ihc::stream_out<int> &a) {
  static int count = 0;
  for(int idx = 0; idx < 5; idx ++){
    bool success = a.tryWrite(count++); // Non-blocking write
    if (success) {
      // write was successful
    }
  }
}

int main() {
  ihc::stream_out<int> a;
  foo(a); // or foo_nb(a);

  // copy output to an array
  int outputData[5];
  for (int i = 0; i < 5; i++) {
    outputData[idx] = a.read();
  }
}
```

10.14. Intel HLS Compiler Standard Edition Memory-Mapped Interfaces

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

Table 34. Intel HLS Compiler Standard Edition Memory-Mapped Interfaces Summary

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

`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:

```
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 indicies of the array. This function is supported only in the testbench.

Code Example

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

10.15. Intel HLS Compiler Standard Edition Arbitrary Precision Data Types

Table 35. Arbitrary Precision Data Types Supported by the Intel HLS Compiler Standard Edition

Data Type	Intel Header File	Description
`ac_int`	`HLS/ac_int.h`	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`	`HLS/ac_fixed.h`	Arbitrary-precision fixed-point number support To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_constructor`
	`HLS/ac_fixed_math.h`	Support for some nonstandard math functions for arbitrary-precision fixed-point data types To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_math_library`

Table 36. Intel HLS Compiler Standard Edition ac_int Debugging Tools Summary

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

DEBUG_AC_INT_WARNING ac_int Debugging Tool

Macro Syntax

`#define DEBUG_AC_INT_WARNING`

If you use this macro, declare it in your code before you declare `#include HLS/ac_int.h`.

i++ Command Option Syntax

`-D DEBUG_AC_INT_WARNING`

Description

Enables runtime tracking of `ac_int` data types during x86 emulation (the `-march=x86-64` option, which the default option, of the `i++` command).

This tool uses additional resources for tracking the overflow and empty constructors, and emits a warning for each detected overflow.

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

DEBUG_AC_INT_ERROR ac_int Debugging Tool

Macro Syntax

`#define DEBUG_AC_INT_ERROR`

	If you use this macro, declare it in your code before you declare `#include HLS/ac_int.h`.
i++ Command Option Syntax	`-D DEBUG_AC_INT_ERROR`
Description	Enables runtime tracking of `ac_int` data types during x86 emulation of your component (the `-march=x86-64` option, which the default option, of the `i++` command). This tool uses additional resources to track the overflow and empty constructors, and emits a message for the first overflow that is detected and then exits the component with an error. To learn more, review the tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/ac_datatypes/ac_int_overflow`

A. Supported Math Functions

The Intel HLS Compiler has built-in support for generating efficient IP out of standard math functions present in the `math.h` C header file. The compiler also has support for some math functions that are not supported by the `math.h` header file, and these functions are provided in `extendedmath.h` C header file.

To use the Intel implementation of `math.h` for Intel FPGAs, include `HLS/math.h` in your function by adding the following line:

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

To use the nonstandard math functions that are optimized for Intel FPGAs, include `HLS/extendedmath.h` in your function by adding the following line:

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

The `extendedmath.h` header is compatible only with Intel HLS Compiler. It is not compatible with GCC or Microsoft Visual Studio.

If your component uses arbitrary precision fixed-point datatypes provided in the `ac_fixed.h` header, you use some of the datatypes with some math functions by including the following line:

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

To see examples of how to use the math functions provided by these header files, review the following tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/best_practices/single_vs_double_precision_math`.

A.1. Math Functions Provided by the `math.h` Header File

The Intel HLS Compiler Standard Edition supports a subset of functions that are present in your native compiler through the `HLS/math.h` header file.

For each `math.h` function listed below, "•" indicates that the HLS compiler supports the function; "X" indicates that the function is not supported.

The math functions supported on Linux operating systems might differ from the math functions supported on Windows operating systems. Review the comments in the `HLS/math.h` header file to see which math functions are supported on the different operating systems.

Table 37. Trigonometric Functions

Trigonometric Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
cos	cosf	•
sin	sinf	•
tan	tanf	•
acos	acosf	•
asin	asinf	•
atan	atanf	•
atan2	atan2f	•

Table 38. Hyperbolic Functions

Hyperbolic Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
cosh	coshf	•
sinh	sinhf	•
tanh	tanhf	•
acosh	acoshf	X
asinh	asinhf	X
atanh	atanhf	X

Table 39. Exponential and Logarithmic Functions

Exponential or Logarithmic Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
exp	expf	•
frexp	frexpf	•
ldexp	ldexpf	•
log	logf	•
log10	log10f	•
modf	modff	•
exp2	exp2f	•
exp10 (Linux only)	exp10f (Linux only)(*)	X
expm1	expm1f	•
ilogb	ilogbf	X
log1p	log1pf	•
log2	log2f	•

continued...

(*) For Windows, support for this function is in the `extendedmath.h` header file

Exponential or Logarithmic Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
logb	logbf	X
scalbn	scalbnf	X
scalbln	scalblnf	●

Table 40. Power Functions

Power Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
pow	powf	●
sqrt	sqrtf	●
cbrt	cbrtf	X
hypot	hypotf	X

Table 41. Error and Gamma Functions

Error or Gamma Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
erf	erff	X
erfc	erfcf	X
tgamma	tgammaf	X
lgamma	lgammaf	X
lgamma_r (Linux only)(*)	lgamma_rf (Linux only)(*)	X

Table 42. Rounding and Remainder Functions

Rounding or Remainder Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
ceil	ceilf	●
floor	floorf	●
fmod	fmodf	●
trunc	truncf	●
round	roundf	●
lround	lroundf	X
llround	llroundf	X
rint	rintf	●
lrint	lrintf	X
llrint	llrintf	X
nearbyint	nearbyintf	X
remainder	remainderf	X
remquo	remquof	X

Table 43. Floating-Point Manipulation Functions

Floating-Point Manipulation Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
copysign	copysignf	X
nan	nanf	X
nextafter	nextafterf	X
nexttoward	nexttowardf	X

Table 44. Minimum, Maximum, and Difference Functions

Minimum, Maximum, or Difference Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
fdim	fdim	●
fmax	fmax	●
fmin	fmin	●

Table 45. Other Functions

Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
fabs	fabsf	●
fma	fmaf	X

Table 46. Classification Macros

Classification Macro		Supported ?
Double-precision floating point function	Single-precision floating point function	
fpclassify (Linux only)	fpclassifyf (Linux only)	X
isfinite	isfinitef	●
isinf	isinff	●
isnan	isnanf	●
isnormal (Linux only)	isnormalf (Linux only)	X
signbit (Linux only)	signbitf (Linux only)	X

Table 47. Comparison Macros

Comparison Macro		Supported ?
Double-precision floating point function	Single-precision floating point function	
isgreater	isgreaterf	X
isgreaterequal	isgreaterequalf	X
isless	islessf	X

continued...

Comparison Macro		Supported ?
Double-precision floating point function	**Single-precision floating point function**	
islessequal	islessequalf	X
islessgreater	islessgreaterf	X
isunordered (Linux only)	isunorderedf (Linux only)	X

A.2. Math Functions Provided by the `extendedmath.h` Header File

The Intel HLS Compiler Standard Edition supports an additional subset of math functions through the `HLS/extendedmath.h` header file.

For each `extendedmath.h` function listed below, "●" indicates that the Intel HLS Compiler Standard Edition supports the function; "X" indicates that the function is not supported.

The math functions supported on Linux operating systems might differ from the math functions supported on Windows operating systems. Review the comments in the `HLS/extendedmath.h` header file to see which math functions are supported on the different operating systems.

Table 48. Extended Math Functions

Extended Math Functions		Supported ?
Double-precision floating point function	**Single-precision floating point function**	
sincos	sincosf	●
acospi	acospif	●
asinpi	asinpif	●
atanpi	atanpif	●
cospi	cospif	●
sinpi	sinpif	●
tanpi	tanpif	●
pown	pownf	●
powr	powrf	●
rsqrt	rsqrtf	●

Table 49. Exponential and Logarithmic Functions

Exponential or Logarithmic Function		Supported ?
Double-precision floating point function	**Single-precision floating point function**	
exp10 (Windows only)	exp10f (Windows only) (*)	X

(*) For Linux, support for this function is in the `math.h` header file

Table 50. Error and Gamma Functions

Error or Gamma Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
lgamma_r (Windows only)[*]	lgamma_rf (Windows only)[*]	X

Table 51. Minimum, Maximum, and Difference Functions

Minimum, Maximum, or Difference Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
maxmag	maxmagf	X
minmag	minmagf	X

Table 52. Other Functions

Function		Supported ?
Double-precision floating point function	Single-precision floating point function	
fract	fractf	X
mad	madf	X
oclnan	oclnanf	X
rootn	rootnf	X

Table 53. Classification Macros

Classification Macro		Supported ?
Double-precision floating point function	Single-precision floating point function	
isordered	isorderedf	X

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

Table 54. Popcount function

Data type	Function
Unsigned char	popcountc
Unsigned short	popcounts
Unsigned int	popcount
Unsigned long	popcountl
Unsigned long long	popcountll

To see an example of how to use the math functions provided by the `extendedmath.h` header file and how to override a math function in the header file so that you can compile your design with GCC or Microsoft Visual Studio, review the following example design: `<`*`quartus_installdir`*`>/hls/examples/QRD`.

Send Feedback

A.3. Math Functions Provided by the `ac_fixed_math.h` Header File

Adding the `ac_fixed_math.h` header file adds support for the following arbitrary precision fixed-point (`ac_fixed`) datatype functions:

- sqrt_fixed
- reciprocal_fixed
- reciprocal_sqrt_fixed
- sin_fixed
- cos_fixed
- sincos_fixed
- sinpi_fixed
- cospi_fixed
- sincospi_fixed
- log_fixed
- exp_fixed

For details about inputs type restrictions, input value limits, and output type propagation rules, review the comments in the `ac_fixed_math.h` header file.

B. Intel HLS Compiler Standard Edition Reference Manual Archives

Intel HLS Compiler Version	Title
19.1	Intel HLS Compiler Standard Edition Reference Manual
18.1.1	Intel HLS Compiler Reference Manual
18.1	Intel HLS Compiler Reference Manual
18.0	Intel HLS Compiler Reference Manual
17.1.1	Intel HLS Compiler Reference Manual
17.1	Intel HLS Compiler Reference Manual

C. Document Revision History of the Intel HLS Compiler Standard Edition Reference Manual

Document Version	Intel HLS Compiler Standard Edition Version	Changes
2019.12.18	19.1	• Removed information about Intel HLS Compiler Pro Edition. For reference information for the Intel HLS Compiler Pro Edition, see Intel HLS Compiler Pro Edition Reference Manual.

Document Revision History for Intel HLS Compiler Reference Manual

Previous versions of the *Intel HLS Compiler Reference Manual* contained information for both Intel HLS Compiler Standard Edition and Intel HLS Compiler Pro Edition.

Document Version	Intel Quartus Prime Version	Changes
2019.09.30	19.3	• PRO Expanded and reorganized information about HLS libraries into a new chapter that starts with Object Libraries. • PRO Added information about arbitrary precision floating point number support to Declaring hls_float Data Types. • PRO Added Scope Pragmas. • PRO For variable-latency Avalon Memory-Mapped (MM) Master interfaces, added information about load-store unit control to Load-Store Unit Control. • PRO Added the `--ffp-reassoc` and `--ffp-contract=fast` options to i++ Comand-Line Arguments • Revised Component Memories (Memory Attributes) (formerly *Local Variables in Components*). • PRO Added information about the `hls_max_replicates` memory attribute to the following sections: — Component Memories (Memory Attributes) — Component Memory Attributes • PRO Deprecated the `hls_numports_readonly_writeonly` memory attribute throughout this document. Use `hls_max_replicates` instead. • PRO Added information about the `max_interleaving` loop pragma to the following sections: — Loop Interleaving Control (`max_interleaving` Pragma) — Loop Pragmas • PRO Added information about the `hls_fpga_reg` function to Advanced Hardware Synthesis Controls.

continued...

Document Version	Intel Quartus Prime Version	Changes
		• PRO Removed the restriction that task functions cannot have pointer or reference arguments from Task Functions. • In Intel HLS Compiler Standard Edition Component Memory Attributes on page 57, revised the description of the default value of the `hls_bankbits` memory attribute. • In Intel HLS Compiler Standard Edition Component Memory Attributes on page 57, removed references to the `bank_bits` tutorial. This tutorial has been removed.
2019.09.10	19.2	• Corrected typo in the description of the `-c` option in Intel HLS Compiler Standard Edition Command Options on page 5. The sentence that began, "When you later compile the`.o` file..." has been corrected to say, "When you later link the `.o` file".
2019.07.01	19.2	• PRO Added information about datapath pipelining control to the following sections: — Loop Pipelining Control (`disable_loop_pipelinng` Pragma) — Loop Pragmas — Component Pipelining Control (`hls_disable_component_pipelining` Attribute) — Component Attributes • Revised and update the following topics about supported math functions: — Math Functions Provided by the math.h Header File on page 83 — Math Functions Provided by the extendedmath.h Header File on page 87
2019.06.04	19.1	• PRO In Slave Memories, clarified the use of memory attributes for slave memories. • In Component Memories (Memory Attributes) on page 28, 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 — Simulation API (Testbench Only) • PRO Updated diagrams in Task Functions. • Updated diagram in Intel HLS Compiler Pipeline Approach on page 9. • PRO Updated diagram in Creating Objects From RTL Code. • PRO Updated diagram in Integration of an RTL Module into the HLS Pipeline.
2019.04.01	19.1	• PRO Added information about developing your system with HLS tasks in Systems of Tasks. • PRO Added information about templated and overloaded functions in Templated and Overloaded Functions. • PRO Added information about arbitrary precision complex number (`ac_complex`) support to Arbitrary Precision Math Support. • Updated Compiler Interoperability on page 8 with details about how to use GCC and Microsoft Visual Studio to compile your component.

continued...

Document Version	Intel Quartus Prime Version	Changes
		• Added information about the compiler pipeline approach in Intel HLS Compiler Pipeline Approach on page 9. • In Intel HLS Compiler Standard Edition Command Options on page 5, corrected `--gcc-toolchain` option syntax. • In Intel HLS Compiler Standard Edition Command Options on page 5, 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 Standard Edition Command Options on page 5 — Intel HLS Compiler Standard Edition i++ Command-Line Arguments on page 51 • Updated Loop-Carried Dependencies (ivdep Pragma) on page 37 to indicate that arrays specified by the `ivdep` loop pragma can now be a reference a reference to an `mm_master` object. • Revised and reorganized Intel High Level Synthesis Compiler Standard Edition Compiler Reference Summary on page 51. • In Declaring ac_int Data Types on page 45, 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.
2019.01.03	18.1.1	• Fixed typos in table headings in Intel HLS Compiler Standard Edition Compiler-Defined Preprocessor Macros on page 13.
2018.12.24	18.1.1	• 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 11.
2018.12.24	18.1	• Updated Slave Interfaces on page 21 and Slave Memories on page 24 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 18.
2018.09.24	18.1	• PRO 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 HLS Compiler Version 18.1 Release Notes. • PRO 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 25 and Quick Reference now reflects that requirement.

continued...

Document Version	Intel Quartus Prime Version	Changes
		• In Loops in Components on page 35, clarified the pragma statements that apply to loops must immediately precede the loop that the pragma applies to. • In Declaring ac_int Data Types on page 45, 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 33, 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 33, 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 21, 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".
2018.05.07	18.0	• 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: PRO Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Pro Edition. STD Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Standard Edition. • PRO Corrected the code example in Intel HLS Compiler Streaming Input Interfaces Code Example. The corrected line is `int x = a.tryRead(success);` (was `int x = a.tryRead(&success);`). • PRO Added `<`*`quartus_installdir`*`>/hls/examples/tutorials/interfaces/ explicit_streams_packets_empty` to list of tutorials in Streaming Input Interfaces and Quick Reference. • PRO Added `ihc::firstSymbolInHighOrderBits` and `ihc::usesEmpty` to the list of stream interface declarations in Streaming Input Interfaces and Quick Reference. Also, revised the description of the `ihc::bitsPerSymbol` declaration to include the effect of the `ihc::firstSymbolInHighOrderBits` declaration. • STD Added a footnote to the `-march MAX10` option in Table 3 on page 5 about a prerequisite required before you synthesize your component IP for Intel MAX 10 devices. • Added new topic AC Data Types and Native Compilers on page 49 describing use of reference AC datatype headers with the Intel HLS Compiler. • Advanced Math Source Code Libraries added to document Intel HLS Compiler libraries. The following Intel HLS Compiler libraries were added: — PRO Random Number Generator Library — PRO Matrix Multiplication Library

continued...

Document Version	Intel Quartus Prime Version	Changes
2017.12.22	17.1.1	• Updated `hls_avalon_slave_memory_argument(N)` description in Slave Memories on page 24 to include the description that the parameter value *N* is the size of the memory in bytes. • Updated Table 18 on page 48 and Table 36 on page 81 to indicate that the `ac_int` debug macros have the following restrictions: — You must declare the macros in your code before you declare `#include HLS/ac_int.h`. — The `ac_int` debugging tools work only for x86 emulation of your component. • Updated `-march "<FPGA_family>"` options in Intel HLS Compiler Standard Edition Command Options on page 5 to include FPGA family options without a space. • Revised the description of the `ihc::align` argument in ihc::align Template Argument on page 79 in Quick Reference. The same information also appears in Intel HLS Compiler Standard Edition Memory-Mapped Interfaces on page 77.
2017.11.06	17.1	• Updated Intel HLS Compiler Standard Edition Command Options on page 5 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 83 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 44 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 18 and Quick Reference. • Added the `ihc::waitrequest` Avalon-MM interface to Avalon Memory-Mapped Master Interfaces on page 18 and Quick Reference. • Added the `hls_stall_free_return` macro and `stall_free_return` attribute to Unstable and Stable Component Arguments on page 25 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 33 to add information about static variable initialization and how to control it. • Minor changes and corrections.
2017.06.09	—	• Revised Declaring ac_int Data Types on page 45 for changes in how to include `ac_int.h`. • Revised Arbitrary Precision Math Support on page 44 to clarify support for Algorithmic C datatypes.

continued...

Document Version	Intel Quartus Prime Version	Changes
		• Removed all mentions of `--device` compiler option. This option has been replaced by the changed function of the `-march` compiler option. See Table 3 on page 5 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 21. • Added information about structs in component interfaces to Component Interfaces on page 14. • Revised C and C++ Libraries on page 11 with updates to iostream behavior. • Added information about math functions supported by `extendedmath.h` header file to Supported Math Functions on page 83.
2017.02.03	—	• In *Scalar Parameters and Avalon Streaming Interfaces*, updated information in the *Available Scalar Parameters for Avalon-ST Interfaces* table. • In *Pointer Parameters, Reference Parameters, and Avalon Memory-Mapped Master Interfaces*, updated information in the *Available Template Arguments for Configuration of the Avalon-MM Interface* table. • Added new information to *Global Variables* about area usage and optimizing for global constants, pointers, and variables.
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 *Usage Examples of Component Invocation Protocol Macros*, replaced component invocation protocol attributes in the code examples with their corresponding macros. • Added the line `#include "HLS/hls.h"` to the code snippets in the following sections: — *Usage Examples of Interface Synthesis Macros* — *Usage Examples of Component Invocation Protocol Macros* • Added the topic *Arbitrary Precision Integer Support* to introduce the `ac_int` datatype and the Intel-provided `ac_int.h` header file. Included the following subtopics: — *Defining the ac_int Datatype in Your Component for Arbitrary Precision Integer Support* — *Important Usage Information on the ac_int Datatype*

continued...

Document Version	Intel Quartus Prime Version	Changes
		• Updated the content in *Area Minimization and Control of On-Chip Memory Architecture*: — Replaced the `numreadports(n)` and `numwriteports(n)` entries the *Attributes for Controlling On-Chip Memory Architecture* table with a single `numports_readonly_writeonly(m,n)` entry. — Added information on the `hls_simple_dual_port_memory` macro. — Added information on the `hls_merge ("label","direction")` and the `hls_bankbits(b0, b1, ..., bn)` attributes. • Added example use cases for the `hls_merge("label","direction")` and the `hls_bankbits(b0, b1, ..., bn)` attributes. • Added the topic *Relationship between hls_bankbits Specifications and Memory Address Bits* to explain the derivation of a memory address in the presence of the `hls_bankbits` and `hls_bankwidth` attributes.
2016.09.12	—	Initial release.