Intel® High Level Synthesis Compiler

Reference Manual

Updated for Intel® Quartus® Prime Design Suite: **17.1**

Subscribe
Send Feedback

MNL-1083 | 2017.12.22
Latest document on the web: PDF | HTML

Contents

1 Intel® HLS Compiler Reference Manual

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

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

2 Compiler

2.1 Intel HLS Compiler Command Options

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

Table 1. 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. By default, the compiler outputs an `a.out` file for Linux and an `a.exe` file for Windows. The `-o` *<result>* command option allows you 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 2. 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. Example command: `i++ -march="Arria 10" -c multiplier.c`

continued...

Option	Description
	Invoking this example command creates a `multiplier.o` file and set the name of the `<`*`result`*`>.prj` directory to `multiplier.prj`. When you later compile the `.o` file, the `-o` option affects only the name of the executable file. The name of the `<`*`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++ --component` *`<components> <input_files>`* This option is not required if you use the `component` function attribute to indicate function 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"` • `Cyclone10GX` or `"Cyclone 10 GX"` • `MAX10` or `"MAX 10"` • `StratixV` or `"Stratix V"` • `Stratix10` or `"Stratix 10"` *<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.
`--quartus-compile`	Compiles your HDL file with the Intel Quartus Prime compiler. Example command: `i++ --quartus-compile` *`<input_files>`* `-march="Arria 10"`

continued...

Option	Description
	When you specify this options, 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.
`--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`) omits verification statistics such as component reset latency. Example command: `i++ -march="<FPGA_family_or_part_number>" --simulator none multiplier.c`

Table 3. Command Options that Customize File Linking

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

Option	Description
`--clock` *<clock_spec>*	Optimizes the RTL for the specified clock frequency or period.
`--fpc`	Removes intermediate rounding and conversion whenever possible. To see an example of how to use this option, review the tutorial in `<quartus_installdir>/hls/examples/tutorials/best_practices/floating_point_ops`.
`--fp-relaxed`	Relaxes the order of arithmetic operations. To see an example of how to use this option, review the tutorial in `<quartus_installdir>/hls/examples/tutorials/best_practices/floating_point_ops`.
`-ghdl`	Enables full debug visibility and the logging of all signals when running the verification executable. After running the executable, the simulator logs waveforms to the `a.prj/verification/vsim.wlf` file.
`-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 name when linking the object file to the binary. On Windows
`--x86-only`	Creates only the testbench executable. The compiler outputs an `<result>` file for Linux and an `<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.

2.2 Compiler Interoperability

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

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

The interoperability between GCC or Microsoft Visual Studio, and the Intel HLS Compiler lets you decouple your testbench development from your component development. Decoupling your testbench development can be useful for situations where you want to iterate your testbench quickly with platform-native compilers (GCC/Microsoft Visual Studio), without having to recompile the RTL generated for your component.

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.

3 C Language and Library Support

3.1 Supported C and C++ Subset for Component Synthesis

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

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

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

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

3.2 C and C++ Libraries

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

Table 4. Intel High Level Synthesis (HLS) Compiler Header Files

Feature	Description
`#include "HLS/hls.h"`	Required for component identification and explicit interfaces.
`#include "HLS/math.h"`	Includes FPGA-specific definitions for the math functions from the `math.h` for your operating system. To learn more, review the tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/best_practices/single_vs_double_precision_math`
`#include "HLS/extendedmath.h"`	Includes additional FPGA-specific definitions of math functions not in `math.h`. To learn more, review the design: `<`*`quartus_installdir`*`>/hls/examples/QRD`
`#include "HLS/ac_int.h"`	Intel HLS Compiler version of `ac_int` header file. Provides 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`

continued...

Feature	Description
`#include "HLS/ac_fixed.h"`	Intel HLS Compiler version of the ac_fixed header file. Provides arbitrary precision fixed point support. To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_constructor`
`#include "HLS/ac_fixed_math.h"`	Intel HLS Compiler version of the `ac_fixed_math` header file. Provides arbitrary precision fixed point math functions. To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_math_library`
`#include "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.
`#include "HLS/iostream"`	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 on supported `math.h` functions, see Supported Math Functions on page 61.

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 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>" --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 noncomponent 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.

Related Links

Supported Math Functions on page 61

3.3 Arbitrary Precision Math Support

The Algorithmic C (AC) datatypes are a collection of header files that Mentor Graphics provides under the Apache license. For more information on Algorithmic C datatypes, refer to *Mentor Graphics Algorithmic C (AC) Datatypes*, which is available as `<`*`quartus_installdir`*`>/hls/include/ref/ac_datatypes_ref.pdf`.

The Intel HLS Compiler supports the following AC datatypes:

Table 5. AC Datatypes Supported by the HLS Compiler

AC Datatype	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 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 datatypes To learn more, review the tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/ac_datatypes/ac_fixed_math_library`

Intel has developed optimized versions of the AC datatypes for the Intel HLS Compiler to generate the best hardware possible on Intel FPGAs for these datataypes.

Using the `ac_int` and `ac_fixed` datatypes has the following advantages over using standard C/C++ datatypes in your components:

- You can achieve smaller datapaths and processing elements for various operations in the circuit.
- The datatypes ensure that all operations are carried out in a size guaranteed not to lose any data. However, you can still lose data if you store data into a to small data type.

The `ac_int` and `ac_fixed` datatypes have the following limitations:

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

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

Related Links

AC Datatypes Download page on the Mentor Graphics website

3.3.1 Declaring ac_int Datatypes in Your Component

The HLS compiler package includes an `ac_int.h` header file for you to include in your component to use arbitrary precision integers in your component.

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

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

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`
   — Built-in `typedefine` declaration up to 63 bits
      — `int`*`N`*` var_name; //Signed N bit integer`
      — `uint`*`N`*` var_name; //Unsigned N bit integer`

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

3.3.1.1 Important Usage Information on the ac_int Datatype

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

The `ac_int` datatype automatically increases the size of the result of the operation to guarantee that the result never overflows. The HLS compiler automatically truncates or extends the result to the size of the specified container.

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

3.3.2 Debugging Your Use of the ac_int Datatype

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

Table 6. Intel HLS Compiler ac_int Debugging Tools

Feature		Description
Macro:	`#define DEBUG_AC_INT_WARNING` If you use this macro, declare it in your code before you declare `#include HLS/ac_int.h`.	Enables runtime tracking of `ac_int` datatypes 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 emits a warning for each detected overflow. To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_int_overflow`
Compiler command line option:	`-D DEBUG_AC_INT_WARNING`	
Macro:	`#define DEBUG_AC_INT_ERROR` If you use this macro, declare it in your code before you declare `#include HLS/ac_int.h`.	Enables runtime tracking of `ac_int` datatypes 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 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`
Compiler command line option:	`-D DEBUG_AC_INT_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.

3.3.3 Declaring `ac_fixed` Datatypes in Your Component

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

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

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

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

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 push data into your function or return data from your function. The parameter interface for your component is based on 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 a system to interact with the hardware.

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

- A call interface that consists of `start` and `busy` signals
- A return interface that consists of done and `stall` signals
- Return data if the component function has a return type that is not `void`

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

- Scalar Parameters on page 16
- Pointer and Reference Parameters on page 16

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

- Avalon Streaming Interfaces on page 17
- Avalon Memory-Mapped Master Interfaces on page 21

4.1.1 Scalar Parameters

Each scalar argument in your component results in an input conduit that is associated with the component `start` and `busy` signals.These ports are pipelined inputs that associate with the `start` and `busy` inputs, as indicated in the Avalon Streaming (Avalon-ST) interface specifications, where `start` is valid signal and `busy` is an inverted `ready` signal.

For details about Avalon-ST interfaces in your component , see Avalon Streaming Interfaces on page 17.

4.1.2 Pointer and Reference Parameters

Each component pointer or reference arguments results in an input conduit, associated with the component `start` and `busy` signals, for the address. In addition to this input conduit, all pointers share an Avalon Memory-Mapped (MM) master interface over which the component fetches each pointer data from system memory.

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

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 21.

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 1. Block Diagram of the Interfaces and Signals for the Component dut

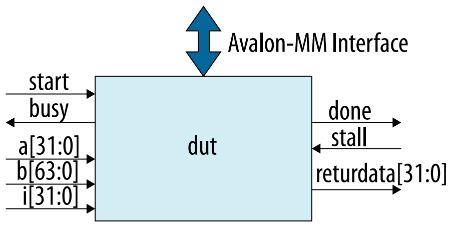
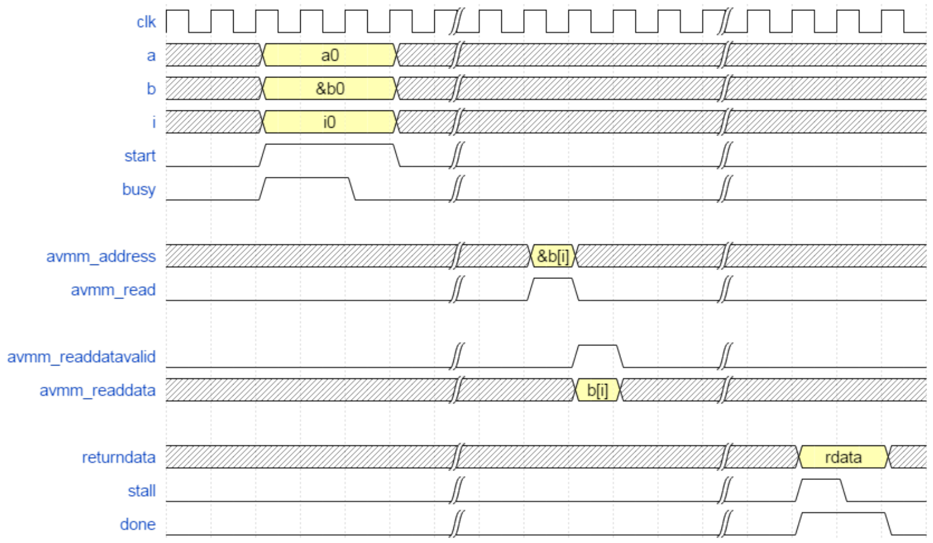


Figure 2. Waveform Diagram of the Signals for the Component dut

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.



4.2 Avalon Streaming Interfaces

All scalar parameter function arguments on a component have input ports in the generated RTL module. These ports are pipelined inputs that associate with the `start` and `busy` inputs, as indicated in the Avalon Streaming (Avalon-ST) interface specifications, where `start` is valid signal and `busy` is an inverted `ready` signal.

In addition to the scalar argument inputs, a component can have explicit 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 the provided `ihc::stream_in<>` and `ihc::stream_out<>` template classes as function arguments to the component.

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

A component can have multiple read sites for a stream. Similarly, a component can have multiple write sites for the same stream.

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*.

Streaming Input Interfaces

Table 7. Intel HLS Compiler Streaming Input Interface `stream_in` Declarations

Feature	Valid Values	Default Value	Description
`ihc::stream_in<datatype>`	Any valid C++ datatype		Streaming input interface to the component: the testbench needs to populate this buffer before sending it to the component. The width of the stream data bus is equal to a width of `sizeof(`*`type`*`)`. 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/explicit_streams_ready_latency` • `<`*`quartus_installdir`*`>/hls/examples/tutorials/interfaces/mulitple_stream_call_sites`
`ihc::buffer`	Non-negative integer value	`0`	The capacity, in words, of the FIFO buffer on the input data that associates with the stream. This parameter is available only on input streams.
`ihc::readylatency`	Non-negative integer value (between 0-8)	`0`	The number of cycles between when the `ready` signal is deasserted and when the input stream can no longer accept new inputs.
`ihc::bitsPerSymbol`	A positive integer value that evenly divides by the data type size	Datatype size	Describes how the data is broken into symbols on the data bus. Data is always broken down in little endian order.
`ihc::usesPackets`	`true` or `false`	`false`	Exposes the `startofpacket` and `endofpacket` sideband signals on the stream interface, which can be accessed by the packet based reads/writes
`ihc::usesValid`	`true` or `false`	`true`	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 having `success` always be `true`. If set to `false`, `buffer` and `readyLatency` must be 0.

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

```
void foo (ihc::stream_in<int> &a) {
 int x = a.read();
 // Blocking read
}
void foo_nb (ihc::stream_in<int> &a) {
 bool success = false;
 int x = a.tryRead(&success);
 // Blocking read
 if (success) {
 // x is valid
 }
}

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

continued...

Feature	Valid Values	Default Value	Description
```
  a.write(i);
  b.write(i);
 }
 foo(&a);
 foo_nb(&b);
}
```

Table 8. Intel HLS Compiler Streaming Input Interface `stream_in` Function APIs

Feature	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. If you use `tryRead`, your x86 results for your component might not match your FPGA results because emulation does not model the hardware behavior of blocking and non-blocking reads.
`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 send 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.

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

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

Streaming Output Interfaces

Table 9. Intel HLS Compiler Streaming Output Interfaces `stream_out` Declaration

Feature	Valid Values	Default Value	Description
`ihc::stream_out<datatype>`	Any valid POD (plain old data) C++ datatype		Streaming output interface from the component. The testbench can read from this buffer once the component returns. To learn more, review the following tutorials:

continued...

Feature	Valid Values	Default Value	Description
			• `<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/explicit_streams_ready_latency` • `<quartus_installdir>/hls/examples/tutorials/interfaces/mulitple_stream_call_sites`
`ihc::readylatency`	Non-negative integer value (between 0-8)	0	The number of cycles between when the `ready` signal is deasserted and when the input stream 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`	Positive integer value that evenly divides the data type size	Datatype size	Describes how the data is broken into symbols on the data bus. Data is always broken down in little endian order.
`ihc::usesPackets`	`true` or `false`	`false`	Exposes the startofpacket and endofpacket sideband signals on the stream interface, which can be accessed by the packet based reads/writes.
`ihc::usesReady`	`true` or `false`	`true`	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 having `success` always be `true`. If set to `false`, `readyLatency` must be 0.

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

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

Table 10. Intel HLS Compiler Streaming Output Interfaces `stream_out` Function Call APIs

Feature	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 call from the component with sideband 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 call from the component. The return value represents whether the write was successful.

continued...

Feature	Description
`T read()`	Blocking read call to be used from the testbench to read back the data from the component
`T read(bool &sop, bool &eop)`	Available only if `usesPackets<true>` is set. Blocking read call to be used from the testbench to read back the data from the component with sideband signals.

The following code example illustrates both `stream_out` declarations and `stream_out` function APIs.
`void foo (ihc::stream_out<int> &a) {`
` static int count = 0;`
` a.write(count++); // Blocking write`
`}`
`void foo (stream_out<int> &a) {`
` static int count = 0;`
` bool success = a.tryWrite(count++); // Non-blocking write`
` if (success) {`
` // write was successful`
` }`
`}`	

Related Links

Avalon Interface Specifications

4.3 Avalon Memory-Mapped Master Interfaces

A component can interface with an external memory over an Avalon Memory-Mapped (MM) Master interface. You can specify the Avalon MM Master interface implicitly using a function pointer argument or reference argument, or explicitly using the `mm_master<>` class defined in the `"HLS/hls.h"` header file. An `mm_master<>` class must function as a reference parameter in the component signature.

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

Table 11. Intel HLS Compiler Memory-Mapped Interfaces

Feature	Valid Values	Default Value	Description

`ihc::mm_master <datatype, /*template arguments*/ >`	Any valid C++ datatype	Default interface for pointer arguments	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(`*`type`*`)`. The default alignment is aligned to the size of the datatype.
			Avalon Memory-Mapped (MM) Master interface argument: Multiple template arguments are supported.
			The template arguments are listed below.
			Any combination can be used as long as it 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:

continued...

Feature	Valid Values	Default Value	Description
			• `<quartus_installdir>/hls/examples/tutorials/interfaces/pointer_mm_master` • `<quartus_installdir>/hls/examples/tutorials/interfaces/mm_master_testbench_operators`
`ihc::dwidth<value>`	8, 16, 32, 64, 128, 256, 512, or 1024	64	The width of the memory-mapped data bus in bits
`ihc::awidth<value>`	Integer value in the range 1 – 64	64	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<value>`	Integer value greater than 0	1	The address space of the interface that associates with the master. 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<value>`	Non-negative integer value	1	The guaranteed latency from when a read command exits the component when the external memory returns valid read data. If this latency is variable, set it to 0.
`ihc::maxburst<value>`	Integer value in the range 1 – 1024	1	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<value>`	Integer value greater than the alignment of the data type	Alignment of the datatype	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`	`readwrite`, `readonly`, or `writeonly`	`readwrite`	Port direction of the interface. Only the relevant Avalon master signals are generated
`ihc::waitrequest`	`true` or `false`	`false`	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(int index)`			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.
			Example:
			`int main() {`
			`// …….`
			`for(int index = 0; index < N; index++) {`
			`  dut(src_mm.getInterfaceAtIndex(index));`

continued...

Feature	Valid Values	Default Value	Description

			`}`
			`// …….`
			`}`

Related Links

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, and 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.

```
#include "HLS/hls.h"
component void dut(ihc::mm_master<int, ihc::dwidth<256>,
                                     ihc::awidth<32>,
                                     ihc::aspace<1>,
                                     ihc::latency<2> > &mm1,
                   ihc::mm_master<int, ihc::dwidth<256>,
                                     ihc::awidth<32>,
                                     ihc::aspace<4>,
                                     ihc::latency<2> > &mm2) {
  *mm1 += *mm2;
  *mm2 += mm1[1];
}
int main(void) {
  int x[2] = {0, 1};
  int y = 2;

  ihc::mm_master<int, ihc::dwidth<256>, ihc::awidth<32>, ihc::aspace<1>,
                    ihc::latency<2> > mm_x(x,2*sizeof(int),false);
  ihc::mm_master<int, ihc::dwidth<256>, ihc::awidth<32>, ihc::aspace<4>,
                    ihc::latency<2> > 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 12. 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.	Updates from the component's datapath are visible in memory.	All changes to this memory are immediately visible to the component datapath. Therefore, reads and writes from outside of the component should only occur when the component is not executing.	Fixed value that is dependent on the component memory access pattern and any attributes or pragmas that you set. See the component `.qsys` file for more information.	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 may have a maximum of one CSR slave interface. The control and status registers (that is, function call and return) of an `hls_avalon_slave_component` macro are implemented in 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.

Argument Label	Description	Example

`hls_avalon_slave_register_argument`	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.
	Example:
	`component void foo(`
	`hls_avalon_slave_register_argument int b)`
	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 local memories because they can be accessed from an Avalon MM Master outside of the component. Local memories are by definition local to the component and cannot be accessed outside the component. Unlike local memory components, 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.

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(`*N*`)`	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 (that is, banking, coalescing, etc.).

Argument Label	Description

	If the compiler performs static coalescing optimizations, the slave interface's data width will be the coalesced width. This attribute applies only to a pointer argument.
	Changes to the value of this argument made by the component data path will not be reflected on this register.
	Example:
	`component void foo(`
	`   hls_avalon_slave_memory_argument(128*sizeof(int))`
	`int *a)`
	To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/interfaces/mm_slaves`

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 13. Intel HLS Compiler Component Invocation Interface Arguments

Feature	Description
`hls_avalon_streaming_component` 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: `hls_avalon_streaming_component void foo(/*component arguments*/)`
`hls_avalon_slave_component`	The `start`, `done`, and `returndata` (if applicable) signals are registered in the component slave memory map. Top-level module ports: Avalon-MM slave interface and `irq_done` signal Example: `hls_avalon_slave_component void foo(/*component arguments*/)` To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/interfaces/mm_slaves`
`hls_always_run_component`	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

continued...

Feature	Description
	Example: `hls_always_run_component void foo(/*component arguments*/)`
`hls_stall_free_return`	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. Example: `hls_stall_free_return component int dut(int a, int b) { return a * b;}`

Invocation Argument	Description	Example
`hls_stall_free_return`	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. Example: `hls_stall_free_return component int dut(int a, int b) { return a * b;}`	

Related Links

Control and Status Register (CSR) Slave on page 25

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:

- Default interface arguments
- Pointer interface arguments
- Pass-by-reference arguments
- Avalon Memory-Mapped (MM) Master interface arguments
- 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 components in flight with different stable arguments between the two component invocations.

Argument Label	Description	Example

`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).
	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.
	Example:
	`component int dut(`
	`   hls_stable_argument int a,`
	`   hls_stable_argument int b) {`
	` return a * b;}`
	To learn more, review the tutorial: <*quartus_installdir*>`/hls/examples/tutorials/interfaces/stable_arguments`	

4.7 Global Variables

Components can use and update global variables. Global variables are modeled as Avalon Memory-Mapped (MM) Master interfaces.

Each global variable that the component uses has a conduit argument, named @<*global variable name*>, which must be supplied with the address of that particular global variable in system memory. As a result, using global variables in a component increases the area required for your design as compared to declaring the global variable as a constant.

If the global variable is a constant, then declaring it as `const` results in no extra area usage for the 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

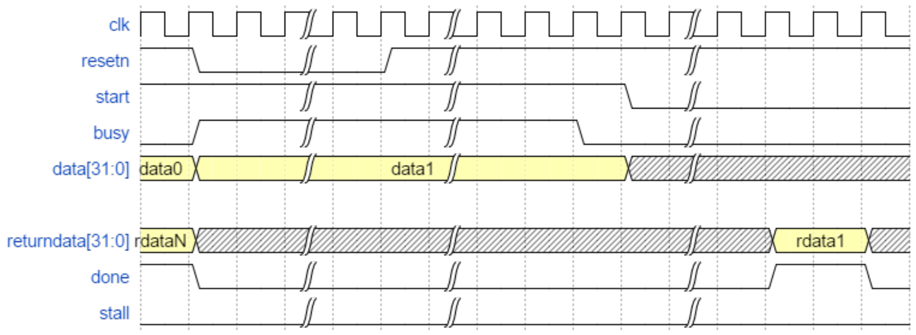
For your HLS component, the reset assertion can be asynchronous but the reset deassertion must be synchronous.

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 `busy` signal is held high and its `done` signal is held low. After the reset is deasserted, the component `busy` signal is held 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 Local Variables in Components (Memory Attributes)

The Intel High Level Synthesis (HLS) Compiler tries to provide the maximum throughput whenever possible. In certain cases, particularly when the Intel HLS Compiler optimizes local memory configurations for throughput, it might be beneficial to trade some throughput for a smaller area. Apply the component memory attributes to local variables in your component to customize the on-chip memory architecture of the local memory system and lower the FPGA area utilization of your component. These component memory attributes are defined in the "`HLS/hls.h`" header file, which you can include in your code.

Restriction: You may apply these attributes to primitives and objects, but not to class members.

Table 14. Intel HLS Compiler Component Memory Attributes

Attribute	Default Value	Description
`hls_register`	Based on the memory access pattern inferred by the compiler.	Forces a variable or array inside component to be implemented as registers. To learn more, review the tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/best_practices/swap_vs_copy`
`hls_memory`	Based on the memory access pattern inferred by the compiler.	Forces a variable or array inside component to be implemented as embedded memory. To learn more, review the design: `<`*`quartus_installdir`*`>/hls/examples/QRD`
`hls_singlepump`	Based on the memory access pattern inferred by the compiler.	Specifies that the memory implementing the local variable must be single pumped. That is, the memory is clocked at the same operating frequency as the operating frequency of your component. To learn more, review the design: `<`*`quartus_installdir`*`>/hls/examples/QRD`
`hls_doublepump`	Based on the memory access pattern inferred by the compiler.	Specifies that the memory implementing the local variable must be double pumped. That is, the memory is clocked at twice the operating frequency of your component.
`hls_numbanks(`*`N`*`)`(*)	Based on the memory access pattern inferred by the compiler.	Specifies that the memory implementing the local variable must have *N* banks, where *N* is a power-of-two constant number.
`hls_bankwidth(`*`N`*`)`(*)	Based on the memory access pattern inferred by the compiler.	Specifies that the memory implementing the local variable must have banks that are *N* bytes wide, where *N* is a power-of-two constant number. To learn more, review the design: `<`*`quartus_installdir`*`>/hls/tutorials/component_memories/bank_bits`

continued...

(*) This attribute is subject to constraints outlined in Constraints on Attributes for Memory Banks on page 35.

Attribute	Default Value	Description
`hls_bankbits(`b_0`,` b_1`, ...,` b_n`)`(*)	Lowest bits of the address based on number of banks.	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. 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. To learn more, review the design: `<`*`quartus_installdir`*`>/hls/tutorials/component_memories/bank_bits`
`hls_numports_readonly_writeonly(`*M*`,` *N*`)`	Based on the memory access pattern inferred by the compiler.	Specifies that the memory implementing the local variable must have *M* read ports and *N* write ports, where *M* and *N* are constant numbers greater than zero.
`hls_simple_dual_port_memory`		Specifies the configuration that is defined by the presence of both the `hls_singlepump` and the `hls_numports_readonly_writeonly(1,1)` macros.
`hls_merge("<`*`mem_name`*`>", "depth")`		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 tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/best_practices/depth_wise_merge`
`hls_merge("<`*`mem_name`*`>", "width")`		Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a width-wise manner. All variables with same <*mem_name*> label specified in their `hls_merge` attribute are merged into the same memory system. To learn more, review the tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/best_practices/width_wise_merge`
`hls_init_on_reset`	Default behavior for static variables.	Forces the static variables inside the component to be reset when the component `reset` signal is asserted. This requires the an additional write port to the component memory implemented and can increase the power-up latency when the component is reset. To learn more, review the tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/component_memories/static_var_init`
`hls_init_on_powerup`		Sets the component memory implementing the static variable to set on power-up when the FPGA is programmed. When the component is reset, the component memory is not reset back to the initialized value of the static. To learn more, review the tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/component_memories/ static_var_init`

Constraints on Attributes for Memory Banks

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

The relationship between the following properties is constrained:

- The size (in bytes) of the local variable (S)
- The number of memory banks specified by `hls_numbanks` attribute (N1)
- The width (in bytes) of the memory banks specified by `hls_bankwidth` attribute (N2)
- 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 (N3)

These attributes are subject to the following constraints:

- `N1 * N2 == S`

  The size of a local variable is equal to the number of memory banks it uses and the width of the memory banks.
- N1 must be a power of 2 value.
- N3 = $\log_2$(N1)

  N3 bank-selection bits that are required to address N1 number of memory banks.

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

5.1 Static Variables

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

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

The HLS compiler supports file-scope static variables that are accessed only by a single component. The HLS compiler compiles these file-scope static variables as if they are function-scope static variables for that component.

Static Variable Initialization

Unlike a typical program, you can control when the static variables in your component are initialized. A static variable can be initialized either when a component is powered up or 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`
(default behavior)

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

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

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

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

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

```
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. 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 32.

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.

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.

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.

Table 15. Intel HLS Compiler Loop Pragmas

Pragma	Description
`#pragma ii <N>`	Forces the loop that this is applied on to have a loop initiation interval (II) of <*N*>, where <*N*> is a positive integer value. This can have an adverse effect on the f_{max} of your component because this pragma combines pipeline stages together and creates logic with a long propagation delay.

continued...

Pragma	Description

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

`#pragma ivdep safelen(<N>) array(<array_name>)`	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 between your load/stores in the loop. The compiler uses this information to properly annotate the dependencies. The only case when it is safe to not include `safelen` is 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`

`#pragma loop_coalesce <N>`	The compiler tries to fuse all loops nested within this loop into a single loop. This pragma accepts an optional value *<N>* which indicates the number of loops to coalesce together.
	`#pragma loop_coalesce`
	`for (int i = 0; i < 8; i++) {`
	` for (int j = 0; j < 8; j++) {`
	` // Loop body`
	` }`
	`}`

`#pragma unroll <N>`	This pragma unrolls the loop completely or by *<N>* times, where *<N>* is optional and is a positive integer value.
	Example:
	`#pragma unroll`
	`for (int i = 0; i < 8; i++) {`
	` // Loop body`
	`}`
	To learn more, review the tutorial: `<quartus_installdir>/hls/examples/best_practices/resource_sharing_filter`

`#pragma max_concurrency <N>`	This pragma limits the number of iterations of a loop that can simultaneously execute at any time.
	This pragma is useful mainly when component memory is duplicated to improve the throughput of the loop. This is mentioned in the details pane for the loop in the Loop Analysis pane 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 copies`
	`// of the array "arr"`
	`#pragma max_concurrency 1`
	`for (int i = 0; i < 8; i++) {`
	` int arr[1024];`
	` // Loop body`
	`}`

6.1 Loop Initiation Interval (ii Pragma)

Use the `ii` pragma to direct the Intel High Level Synthesis (HLS) Compiler to attempt to set the loop initiation interval (II) for the loop that follows the pragma declaration. If the compiler cannot achieve the specified II for the loop, then the compile errors out.

The initiation interval, or II, is the number of clock cycles between launching successive loop iterations. The higher the II value, the longer the wait before the next loop iteration.

For some loops in your component, specifying a higher II with the `ii` pragma than the compiler might choose can increase the maximum operating frequency (f_{max}) of your component without losing throughput.

A loop is a good candidate to have the `ii` pragma applied to it if the loop meets 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.

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 successive loop iterations.

Example

Consider a case where your component has two distinct, pipelineable loops: a short-running initialization loop that has a loop-carried dependence and a long-running loop that does the bulk of your processing. 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 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. 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 dependencies across loop iterations in your component by adding the `ivdep` pragma in your code.

The `ivdep` pragma tells the compiler that a 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 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 a local or private memory array, or a pointer variable that points to a global, local, or private memory storage. 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 High Level Synthesis (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 the loops.

Coalescing nested loops also reduces the latency of the component, which could further reduce you 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 a set of nested loops like this:

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

If you put the pragma before loop (A), the loop nesting level for these loops is as follows:

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

Example

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

Consider a simple nested loop written as follows:

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

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

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

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

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

6.4 Loop Unrolling (`unroll` Pragma)

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

Example code:

```
#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`).

6.5 Loop Concurrency (`max_concurrency` Pragma)

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

To achieve maximum concurrency in loops, sometimes local memory has to be duplicated to break dependencies on the underlying hardware that prevents the loop from being fully pipelined. You can see this in the Details pane Loop analysis report in your component HLD report (`report.html`) as a message that says that the maximum number of simultaneous executions has been limited to N. Duplicating local 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 local memory savings, apply `#pragma max_concurrency` *<N>* to the loop. When you apply this pragma, the duplication factor changes and controls the number of threads 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 also control the concurrency of your component by using the `hls_max_concurrency(N)` component attribute. For more information about the `hls_max_concurrency(N)` component attribute, see Concurrency Control (hls_max_concurrency Attribute) on page 45.

7 Component Concurrency

The Intel HLS Compiler assumes that you want a fully pipelined datapath in your component. In the C++ implementation, you may think of a fully pipelined datapath as calling a function multiple times (for example, by multiple threads) before the first call has returned. The behavior of threads within the synthesized datapath is subject to the concurrency model, so the Intel HLS Compiler might not be able to deliver a component with a fully-pipelined datapath.

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, every call to the component, that is, every cycle where the `start` signal is asserted and the `busy` signal is deasserted, on the function protocol interface behaves as though the previous function call has been fully executed.

When visualizing a single shared memory space, think of multiple function calls as executing sequentially, one after another. This way, when the done signal is asserted, 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 I/O instructions is not guaranteed.

7.2 Concurrency Control (`hls_max_concurrency` Attribute)

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

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

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

hls_max_concurrency(3)
```

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

The Intel HLS Compiler sets the component concurrency to one in the following cases:

- The Intel HLS compiler does not automatically duplicate local memory to increase the throughput at the component level. If your component invocation uses a (non-static) local memory system that is used by a component invocation, the next invocation cannot start until the previous invocation has finished all of its accesses to and from that local memory. This limitation is shown in the Loop analysis report as load-store dependencies on the array. Adding the `hls_max_concurrency(`*`N`*`)` attribute on the component duplicates the local memory so that you can have multiple invocations of your component in progress at the same time.
- 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 uses 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 with the `max_concurrency(`*`N`*`)` pragma. For more information about the `max_concurrency(`*`N`*`)` pragma, see Loop Concurrency (max_concurrency Pragma) on page 44.

A Intel High Level Synthesis Compiler Quick Reference

Table 16. i++ Command Line Arguments

Feature	Default Value	Description
General `i++` command options		
`--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 compilation		
`-c`		Preprocess, parse, and generate object files.
`--component` *<component name>*		Comma-separated list of function names to synthesize to RTL.
`-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.
`--promote-integers`		Use extra FPGA resources to mimic g++ integer promotion. 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.
`-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.
Command options affecting linking		

continued...

Feature	Default Value	Description

`--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`

`--fp-relaxed`		Relax the order of floating point arithmetic operations. To learn more, review the tutorial: *<quartus_installdir>*`/hls/examples/tutorials/best_practices/floating_point_ops`
`--fpc`		Remove intermediate rounding and conversion when possible. To learn more, review the tutorial: *<quartus_installdir>*`/hls/examples/tutorials/best_practices/floating_point_ops`
`-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 -I.
`-l`*<library>*		(Linux only) Search the library name *<library>* when linking.
`--x86-only`		Create only the testbench executable (*<result>*`.out`/*<result>*`.exe`).
`--fpga-only`		Create only the *<result>*`.prj` directory and its contents.

Table 17. Intel High Level Synthesis (HLS) Compiler Header Files

Feature	Description
`#include "HLS/hls.h"`	Required for component identification and explicit interfaces.
`#include "HLS/math.h"`	Includes FPGA-specific definitions for the math functions from the `math.h` for your operating system. To learn more, review the tutorial: *<quartus_installdir>*`/hls/examples/tutorials/best_practices/single_vs_double_precision_math`
`#include "HLS/extendedmath.h"`	Includes additional FPGA-specific definitions of math functions not in `math.h`. To learn more, review the design: *<quartus_installdir>*`/hls/examples/QRD`
`#include "HLS/ac_int.h"`	Intel HLS Compiler version of `ac_int` header file. Provides 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`
`#include "HLS/ac_fixed.h"`	Intel HLS Compiler version of the `ac_fixed` header file. Provides arbitrary precision fixed point support. To learn more, review the tutorial: *<quartus_installdir>*`/hls/examples/tutorials/ac_datatypes/ac_fixed_constructor`
`#include "HLS/ac_fixed_math.h"`	Intel HLS Compiler version of the `ac_fixed_math` header file. Provides arbitrary precision fixed point math functions.

continued...

Feature	Description
	To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_math_library`
`#include "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.
`#include "HLS/iostream"`	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.

Table 18. Intel HLS Compiler Keywords

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

Table 19. Intel HLS Compiler Simulation API (Testbench only)

Function	Description
`ihc_hls_enqueue(/*ptr to return type*/, /*function arguments*/)`	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 arguments*/)`	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 name*/)`	This function accepts the name of the HLS component. 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`
`int ihc_hls_sim_reset(void)`	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`

```
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);
 }
```

Table 20. Intel HLS Compiler Component Memory Attributes

Attribute	Default Value	Description
`hls_register`	Based on the memory access pattern inferred by the compiler.	Forces a variable or array inside component to be implemented as registers. To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/best_practices/swap_vs_copy`
`hls_memory`	Based on the memory access pattern inferred by the compiler.	Forces a variable or array inside component to be implemented as embedded memory. To learn more, review the design: `<quartus_installdir>/hls/examples/QRD`
`hls_singlepump`	Based on the memory access pattern inferred by the compiler.	Specifies that the memory implementing the local variable must be single pumped. That is, the memory is clocked at the same operating frequency as the operating frequency of your component. To learn more, review the design: `<quartus_installdir>/hls/examples/QRD`
`hls_doublepump`	Based on the memory access pattern inferred by the compiler.	Specifies that the memory implementing the local variable must be double pumped. That is, the memory is clocked at twice the operating frequency of your component.
`hls_numbanks(N)`(*)	Based on the memory access pattern inferred by the compiler.	Specifies that the memory implementing the local variable must have *N* banks, where *N* is a power-of-two constant number.
`hls_bankwidth(N)`(*)	Based on the memory access pattern inferred by the compiler.	Specifies that the memory implementing the local variable must have banks that are *N* bytes wide, where *N* is a power-of-two constant number. To learn more, review the design: `<quartus_installdir>/hls/tutorials/component_memories/bank_bits`
`hls_bankbits(`b_0`,` b_1`, ...,` b_n`)`(*)	Lowest bits of the address based on number of banks.	Forces the memory system to split into 2^{n+1} banks, with $\{b_0, b_1, \ldots, b_n\}$ forming the bank-select bits. *Important:* $b_0, b_1, \ldots, b_n$ must be consecutive, positive integers. If you do not specify the `hls_bankwidth(N)` attribute along with this attribute, then $b_0, b_1, \ldots, b_n$ are mapped to array index bits `0` to `n-1` in the memory bank implementation. To learn more, review the design: `<quartus_installdir>/hls/tutorials/component_memories/bank_bits`
`hls_numports_readonly_writeonly(M, N)`	Based on the memory access pattern inferred by the compiler.	Specifies that the memory implementing the local variable must have *M* read ports and *N* write ports, where *M* and *N* are constant numbers greater than zero.
`hls_simple_dual_port_memory`		Specifies the configuration that is defined by the presence of both the `hls_singlepump` and the `hls_numports_readonly_writeonly(1,1)` macros.
`hls_merge("<mem_name>", "depth")`		Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a depth-wise manner.

continued...

(*) This attribute is subject to constraints outlined in Constraints on Attributes for Memory Banks on page 35.

Attribute	Default Value	Description
		All variables with same <*mem_name*> label specified in their `hls_merge` attribute are merged into the same memory system. To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/best_practices/depth_wise_merge`
`hls_merge("<mem_name>", "width")`		Allows merging two or more local variables to be implemented in component memory as a single merged memory system in a width-wise manner. All variables with same <*mem_name*> label specified in their `hls_merge` attribute are merged into the same memory system. To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/best_practices/width_wise_merge`
`hls_init_on_reset`	Default behavior for static variables.	Forces the static variables inside the component to be reset when the component `reset` signal is asserted. This requires the an additional write port to the component memory implemented and can increase the power-up latency when the component is reset. To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/component_memories/static_var_init`
`hls_init_on_powerup`		Sets the component memory implementing the static variable to set on power-up when the FPGA is programmed. When the component is reset, the component memory is not reset back to the initialized value of the static. To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/component_memories/ static_var_init`

Table 21. Intel HLS Compiler Loop Pragmas

Pragma	Description
`#pragma ii <N>`	Forces the loop that this is applied on to have a loop initiation interval (II) of <*N*>, where <*N*> is a positive integer value. This can have an adverse effect on the f_{max} of your component because this pragma combines pipeline stages together and creates logic with a long propagation delay. Example: `#pragma ii 2` `for (int i = 0; i < 8; i++) {` `// Loop body` `}`
`#pragma ivdep safelen(<N>) array(<array_name>)`	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 between your load/stores in the loop. The compiler uses this information to properly annotate the dependencies. The only case when it is safe to not include `safelen` is when the dependence distance is infinite (that is, there are no real dependencies).

continued...

Pragma	Description

	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`

`#pragma loop_coalesce <N>`	The compiler tries to fuse all loops nested within this loop into a single loop. This pragma accepts an optional value *<N>* which indicates the number of loops to coalesce together.
	`#pragma loop_coalesce`
	`for (int i = 0; i < 8; i++) {`
	` for (int j = 0; j < 8; j++) {`
	` // Loop body`
	` }`
	`}`

`#pragma unroll <N>`	This pragma unrolls the loop completely or by *<N>* times, where *<N>* is optional and is a positive integer value.
	Example:
	`#pragma unroll`
	`for (int i = 0; i < 8; i++) {`
	` // Loop body`
	`}`
	To learn more, review the tutorial: `<quartus_installdir>/hls/examples/best_practices/resource_sharing_filter`

`#pragma max_concurrency <N>`	This pragma limits the number of iterations of a loop that can simultaneously execute at any time.
	This pragma is useful mainly when component memory is duplicated to improve the throughput of the loop. This is mentioned in the details pane for the loop in the Loop Analysis pane 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 copies`
	`// of the array "arr"`
	`#pragma max_concurrency 1`
	`for (int i = 0; i < 8; i++) {`
	` int arr[1024];`
	` // Loop body`
	`}`

Table 22. Intel HLS Compiler Component Attributes

Feature	Description
`hls_max_concurrency (<N>)`	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 local memories for a non-static variable. You can use this attribute to request more copies of the local 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 local memory to instantiate as well as how many component invocations can be in flight at once.

Feature	Description

	Value equal to 0 Setting `hls_max_concurrency` to a value of `0` is useful in cases when there is no local 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.
	Example:
	`hls_max_concurrency(2)`
	`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]);`
	` }`
	`}`
	To learn more, review the design example: <*quartus_installdir*>/hls/examples/inter_decim_filter

Table 23. Intel HLS Compiler 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.

Table 24. Intel HLS Compiler Component Invocation Interface Arguments

Feature	Description
`hls_avalon_streaming_component` 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: `hls_avalon_streaming_component void foo(/*component arguments*/)`
`hls_avalon_slave_component`	The `start`, `done`, and `returndata` (if applicable) signals are registered in the component slave memory map. Top-level module ports: Avalon-MM slave interface and `irq_done` signal

continued...

Feature	Description
	Example: `hls_avalon_slave_component void foo(/*component arguments*/)` To learn more, review the tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/interfaces/mm_slaves`
`hls_always_run_component`	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: `hls_always_run_component void foo(/*component arguments*/)`

`hls_stall_free_return`	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.
	Example:
	`hls_stall_free_return component int dut(int a, int b)`
	` { return a * b;}`

Table 25. Intel HLS Compiler Component Argument Macros

Feature	Description

`hls_conduit_argument`
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`	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.
	Example:
	`component void foo(`
	`  hls_avalon_slave_register_argument int b)`
	To learn more, review the tutorial: `<`*`quartus_installdir`*`>/hls/examples/tutorials/interfaces/mm_slaves`

`hls_avalon_slave_memory_argument(`*`N`*`)`	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 (that is, banking, coalescing, etc.). If the compiler performs static coalescing optimizations, the slave interface's data width will be the coalesced width. This attribute applies only to a pointer argument. Changes to the value of this argument made by the component data path will not be reflected on this register.

continued...

Feature	Description

	Example:
	`component void foo(`
	`   hls_avalon_slave_memory_argument(128*sizeof(int)) int *a)`
	To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/interfaces/mm_slaves`

`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).
	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.
	Example:
	`component int dut(`
	`   hls_stable_argument int a,`
	`   hls_stable_argument int b) {`
	` return a * b;}`
	To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/interfaces/stable_arguments`

Table 26. Intel HLS Compiler Streaming Input Interface `stream_in` Declarations

Feature	Valid Values	Default Value	Description
`ihc::stream_in<datatype>`	Any valid C++ datatype		Streaming input interface to the component: the testbench needs to populate this buffer before sending it to the component. The width of the stream data bus is equal to a width of `sizeof(type)`. 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/explicit_streams_ready_latency` • `<quartus_installdir>/hls/examples/tutorials/interfaces/mulitple_stream_call_sites`
`ihc::buffer`	Non-negative integer value	`0`	The capacity, in words, of the FIFO buffer on the input data that associates with the stream. This parameter is available only on input streams.
`ihc::readylatency`	Non-negative integer value (between 0-8)	`0`	The number of cycles between when the `ready` signal is deasserted and when the input stream can no longer accept new inputs.
`ihc::bitsPerSymbol`	A positive integer value that evenly divides by the data type size	Datatype size	Describes how the data is broken into symbols on the data bus. Data is always broken down in little endian order.

continued...

Feature	Valid Values	Default Value	Description
`ihc::usesPackets`	`true` or `false`	`false`	Exposes the `startofpacket` and `endofpacket` sideband signals on the stream interface, which can be accessed by the packet based reads/ writes
`ihc::usesValid`	`true` or `false`	`true`	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 having `success` always be `true`. If set to `false`, `buffer` and `readyLatency` must be 0.

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

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

Table 27. Intel HLS Compiler Streaming Input Interface `stream_in` Function APIs

Feature	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. If you use `tryRead`, your x86 results for your component might not match your FPGA results because emulation does not model the hardware behavior of blocking and non-blocking reads.
`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 send 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.

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

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

continued...

Feature	Description

`int x = a.tryRead(&success);`
`// Blocking read`
`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);`
`}`	

Table 28. Intel HLS Compiler Streaming Output Interfaces `stream_out` Declaration

Feature	Valid Values	Default Value	Description
`ihc::stream_out<datatype>`	Any valid POD (plain old data) C ++ datatype		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/explicit_streams_ready_latency • *<quartus_installdir>*/hls/examples/tutorials/interfaces/mulitple_stream_call_sites
`ihc::readylatency`	Non-negative integer value (between 0-8)	0	The number of cycles between when the `ready` signal is deasserted and when the input stream 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`	Positive integer value that evenly divides the data type size	Datatype size	Describes how the data is broken into symbols on the data bus. Data is always broken down in little endian order.
`ihc::usesPackets`	`true` or `false`	`false`	Exposes the startofpacket and endofpacket sideband signals on the stream interface, which can be accessed by the packet based reads/writes.
`ihc::usesReady`	`true` or `false`	`true`	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 having `success` always be `true`. If set to `false`, `readyLatency` must be 0.

The following code example illustrates both `stream_out` declarations and `stream_out` function APIs.
 	
`void foo (ihc::stream_out<int> &a) {`
`static int count = 0;`
`a.write(count++); // Blocking write`
`}`			

continued...

Feature	Valid Values	Default Value	Description

```
void foo (stream_out<int> &a) {
 static int count = 0;
 bool success = a.tryWrite(count++); // Non-blocking write
 if (success) {
 // write was successful
 }
}
```

Table 29. Intel HLS Compiler Streaming Output Interfaces `stream_out` Function Call APIs

Feature	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 call from the component with sideband 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 call from the component. The return value represents whether the write was successful.
`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 sideband signals.

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

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

Table 30. Intel HLS Compiler Memory-Mapped Interfaces

Feature	Valid Values	Default Value	Description

`ihc::mm_master <datatype, /*template arguments*/ >`	Any valid C++ datatype	Default interface for pointer arguments	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(`*type*`)`. The default alignment is aligned to the size of the datatype.
			Avalon Memory-Mapped (MM) Master interface argument: Multiple template arguments are supported.
			The template arguments are listed below.
			Any combination can be used as long as it 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:

continued...

Feature	Valid Values	Default Value	Description
			• `<quartus_installdir>/hls/examples/tutorials/interfaces/pointer_mm_master` • `<quartus_installdir>/hls/examples/tutorials/interfaces/mm_master_testbench_operators`
`ihc::dwidth<value>`	8, 16, 32, 64, 128, 256, 512, or 1024	64	The width of the memory-mapped data bus in bits
`ihc::awidth<value>`	Integer value in the range 1 – 64	64	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<value>`	Integer value greater than 0	1	The address space of the interface that associates with the master. 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<value>`	Non-negative integer value	1	The guaranteed latency from when a read command exits the component when the external memory returns valid read data. If this latency is variable, set it to 0.
`ihc::maxburst<value>`	Integer value in the range 1 – 1024	1	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<value>`	Integer value greater than the alignment of the data type	Alignment of the datatype	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`	`readwrite`, `readonly`, or `writeonly`	`readwrite`	Port direction of the interface. Only the relevant Avalon master signals are generated
`ihc::waitrequest`	`true` or `false`	`false`	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(int index)`			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.
			Example:
			`int main() {`
			`// …….`
			`for(int index = 0; index < N; index++) {`
			`  dut(src_mm.getInterfaceAtIndex(index));`

continued...

Feature	Valid Values	Default Value	Description
			`}` `// …….` `}`

Table 31. AC Datatypes Supported by the HLS Compiler

AC Datatype	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 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 datatypes To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_fixed_math_library`

Table 32. Intel HLS Compiler ac_int Debugging Tools

Feature	Description
Macro: `#define DEBUG_AC_INT_WARNING` If you use this macro, declare it in your code before you declare `#include HLS/ac_int.h`. *Compiler command line option:* `-D DEBUG_AC_INT_WARNING`	Enables runtime tracking of `ac_int` datatypes 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 emits a warning for each detected overflow. To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/ac_datatypes/ac_int_overflow`
Macro: `#define DEBUG_AC_INT_ERROR` If you use this macro, declare it in your code before you declare `#include HLS/ac_int.h`. *Compiler command line option:* `-D DEBUG_AC_INT_ERROR`	Enables runtime tracking of `ac_int` datatypes 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 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`

B Supported Math Functions

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

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

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

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

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

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

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

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

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

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

The Intel HLS Compiler supports a subset of functions that are present in your native compiler `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 33. Trigonometric Functions

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

Table 34. Hyperbolic Functions

Hyperbolic Function	Supported by the HLS Compiler?
cosh	•
sinh	•
tanh	•
acosh	X
asinh	X
atanh	X

Table 35. Exponential and Logarithmic Functions

Exponential or Logarithmic Function	Supported by the HLS Compiler?
exp	•
frexp	•
ldexp	•
log	•
log10	•
modf	•
exp2	•
expm1	X
ilogb	•
log1p	X

continued...

Exponential or Logarithmic Function	Supported by the HLS Compiler?
log2	●
logb	●
scalbn	X
scalbln	X

Table 36. Power Functions

Power Function	Supported by the HLS Compiler?
pow	●
sqrt	●
cbrt	X
hypot	X

Table 37. Error and Gamma Functions

Error or Gamma Function	Supported by the HLS Compiler?
erf	X
erfc	X
tgamma	X
lgamma	X

Table 38. Rounding and Remainder Functions

Rounding or Remainder Function	Supported by the HLS Compiler?
ceil	●
floor	●
fmod	●
trunc	●
round	●
lround	X
llround	X
rint	●
lrint	X
llrint	X
nearbyint	X
remainder	X
remquo	X

Table 39. Floating-Point Manipulation Functions

Floating-Point Manipulation Function	Supported by the HLS Compiler?
copysign	X
nan	X
nextafter	X
nexttoward	X

Table 40. Minimum, Maximum, and Difference Functions

Minimum, Maximum, or Difference Function	Supported by the HLS Compiler?
fdim	•
fmax	•
fmin	•

Table 41. Other Functions

Function	Supported by the HLS Compiler?
fabs	•
abs	X
fma	X

Table 42. Classification Macros

Classification Macro	Supported by the HLS Compiler?
fpclassify	X
isfinite	•
isinf	•
isnan	•
isnormal	X
signbit	X

Table 43. Comparison Macros

Comparison Macro	Supported by the HLS Compiler?
isgreater	X
isgreaterequal	X
isless	X
islessequal	X
islessgreater	X
isunordered	X

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

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

Table 44. Extended math functions

Data type	Function
Double-precision floating point	• sincos • acospi • asinpi • atanpi • cospi • sinpi • tanpi • pown • powr • rsqrt
Single-precision floating point	• sincosf • acospif • asinpif • atanpif • cospif • sinpif • tanpif • pownf • powrf • rsqrtf

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

Table 45. Popcount function

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

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

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

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

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

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

C Document Revision History

Table 46. Document Revision History of the Intel HLS Compiler Reference Manual

Date	Version	Changes
December 2017	2017.12.22	• Updated `hls_avalon_slave_memory_argument(`*N*`)` description in Slave Memories on page 28 to include the description that the parameter value *N* is the size of the memory in bytes. • Updated Table 6 on page 13 and Table 32 on page 60 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 Command Options on page 5 to include FPGA family options without a space. • Revised the description of the `ihc::allign` argument in Table 30 on page 58 in Intel High Level Synthesis Compiler Quick Reference on page 47. The same information also appears in Avalon Memory-Mapped Master Interfaces on page 21.
November 2017	2017.11.06	• Updated Intel HLS Compiler 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 61 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 11 to include restriction that the Intel arbitrary precision header files cannot be compiled with GCC.

continued...

Date	Version	Changes
		• Added the `ihc::readwrite_mode` Avalon-MM interface to Avalon Memory-Mapped Master Interfaces on page 21 and Intel High Level Synthesis Compiler Quick Reference on page 47. • Added the `ihc::waitrequest` Avalon-MM interface to Avalon Memory-Mapped Master Interfaces on page 21 and Intel High Level Synthesis Compiler Quick Reference on page 47. • Added the `hls_stall_free_return` macro and `stall_free_return` attribute to Unstable and Stable Component Arguments on page 30 and Intel High Level Synthesis Compiler Quick Reference on page 47. • 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".
June 2017	2017.06.23	• Updated Static Variables on page 36 to add information about static variable initialization and how to control it. • Minor changes and corrections.
June 2017	2017.06.09	• Revised Declaring ac_int Datatypes in Your Component on page 12 for changes in how to include `ac_int.h`. • Revised Arbitrary Precision Math Support on page 11 to clarify support for Algorithmic C datatypes. • Removed all mentions of `--device` compiler option. This option has been replaced by the changed function of the `-march` compiler option. See #unique_3/unique_3_Connect_42_section_N10130_N1001B_N10001 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 25. • Added information about structs in component interfaces to Component Interfaces on page 15. • Revised C and C++ Libraries on page 9 with updates to iostream behavior. • Added information about math functions supported by `extendedmath.h` header file to Supported Math Functions on page 61.

continued...

Date	Version	Changes
February 2017	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.
November 2016	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* • 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.
September 2016	2016.09.12	Initial release.