INTEL® HIGH LEVEL SYNTHESIS COMPILER CONDENSED QUICK REFERENCE

The Intel® High Level Synthesis Compiler takes in untimed C++ as input and generates production-quality register transfer level (RTL) code that is optimized for Intel FPGAs. The Intel HLS Compiler is available as part of Intel Quartus® Prime Design Suite.

Use this guide to quickly find declarations and attributes that you can use with the Intel HLS Compiler. For details about these declarations and attributes, see “Intel High Level Synthesis Compiler Quick Reference” in *Intel High Level Synthesis Compiler Reference Manual*.

Some of these declarations and attributes apply only to Intel HLS Compiler Pro Edition. For details, see the *Intel High Level Synthesis Compiler Reference Manual*

Last updated for Intel HLS Compiler Pro Edition Version 19.1 (2019.05.06)

HLS Compiler i++ Command Options

For i++ command line flags, use the `--help` flag.

Header Files

`HLS/hls.h`	Common HLS attributes Explicit interfaces
`HLS/math.h`	Math functions
`HLS/extendedmath.h`	Math functions not in math.h
`HLS/ac_int.h`	Arbitrary precision integer support
`HLS/ac_fixed.h`	Arbitrary precision fixed-point support
`HLS/ac_fixed_math.h`	Arbitrary precision fixed-point math functions
`HLS/ac_complex.h`	Arbitrary precision complex number support
`HLS/stdio.h`	`printf` support for components during x86 emulation
`<iostream>`	Guard cout and cerr statements with HLS_SYNTHESIS macro

Simulation API (Testbench Only)

`ihc_hls_enqueue (`*`<ptr to storage for return type>, <function name>, <function arguments>`*`)`	Enqueue a pipelined component (with non-`void` return type) invocation
`ihc_hls_enqueue_noret (`*`<function name>,<function arguments>`*`)`	Enqueue a pipelined component (with `void` return type) invocation
`ihc_hls_component_run_all (`*`<function name>`*`)`	Simulate all enqueued invocations of the component in the HDL simulator in a pipeline-parallel fashion
`int ihc_hls_sim_reset(void)`	Send a reset signal to the component during simulation, returning 1 if reset was executed
`ihc_hls_set_component_wait_cycle (`*`<function name>, <wait cycles>`*`)`	Tell simulation to continue running for a number of cycles after a done signal for a function is observed.

Local Memory Attributes

`hls_register | hls_memory`	Implement the variable as registers | RAM blocks
`hls_singlepump | hls_doublepump`	Force a RAM block to be single | double pumped
`hls_numbanks(`*`N`*`)`	Force memory system to have *N* banks
`hls_bankwidth(`*`N`*`)`	Force memory system to have banks that are *N* bytes wide
`hls_bankbits(`b_0, b_1, … b_n`)`	Split the memory system into 2^{n+1} banks with {b_0, b_1, ..., b_n} forming the bank-select bits
`hls_numports_readonly_writeonly(`*`M, N`*`)`	Force memory to have *M* read ports and *N* write ports
`hls_simple_dual_port_memory`	Convenience attribute that is equivalent to both the hls_singlepump and the hls_numports_readonly_writeonly(1,1) macros
`hls_merge("`*`<mem_name>`*`", "depth") | hls_merge("`*`<mem_name>`*`", "width")`	Merge two or more local variables into a single memory system in a depth-wise | width-wise manner
`hls_init_on_reset | hls_init_on_powerup`	Force a static variable to be reset when the component `reset` signal is asserted | on powerup when the FPGA is programmed
`hls_memory_impl ("BLOCK_RAM|MLAB")`	Implement variable or array as block RAMs or MLABs
`hls_max_concurrency(`*`N`*`)`	Specify maximum number of private copies of a memory when allowing simultaneous loop iterations

Loop Pragmas

`#pragma ii `*`<N>`*	Set loop initiation interval to N
`#pragma ivdep safelen(`*`<N>`*`) array(`*`<array_name>`*`)`	Ignore local memory dependencies between iterations up to N iterations apart
`#pragma loop_coalesce `*`<N>`*	Convert nested loops of level N down to single loop
`#pragma unroll `*`<N>`*	Unroll the loop into N copies
`#pragma max_concurrency `*`<N>`*	Specify the number of iterations of a loop that can execute simultaneously
`#pragma speculated_iterations `*`<N>`*	Specify the number of clock cycles that a loop exit condition can take to compute

Component Attributes

`hls_max_concurrency (`*`<N>`*`)`	Specify the number of threads that can enter a component concurrently
`hls_component_ii (`*`<N>`*`)`	Force the component to have a specified II. Can adversely affect f_{MAX}
`hls_scheduler_target_fmax_mhz (`*`<target`* f_{MAX}*`>`*`)`	Specify the target clock frequency (in MHz)

Component Invocation Interface Attributes

Component invocation interface attributes apply to the whole component.

`hls_avalon_streaming_component` (default)	Component invocation interface (start, busy, done, stall, return) is implemented as conduits
`hls_avalon_slave_component`	Component invocation interface (start, busy, done, stall, return) is implemented in Control/Status Register (CSR) as Avalon-MM slave interface with `irq_done` signal
`hls_always_run_component`	Component invocation interface is removed
`hls_stall_free_return`	Stall signal is removed

Parameter Interface Attributes

Parameter interface attributes apply to individual function parameters.

`hls_conduit_argument` (default)	Parameter is synchronous to the component call interfaces
`hls_avalon_slave_register_argument`	Parameter is in the component CSR, and can be written to over an Avalon-MM slave interface. Synchronous to the component call interfaces
`hls_avalon_slave_memory_argument(`*`N`*`)`	Local memory that can be read from and written to over an Avalon-MM slave interface
`hls_stable_argument`	Argument does not change while there is live data in the component

Streaming Interfaces

Streaming Interface Declarations

`ihc::stream_in<`*`datatype`*`, /*template arguments*/>`	Streaming input interface to the component
`ihc::stream_out<`*`datatype`*`, /*template arguments*/>`	Streaming output interface from the component

Streaming Interface Template Arguments

`ihc::buffer`	Capacity of FIFO on input data
`ihc::readylatency`	Number of cycles between ready signal being deasserted and when the input stream can no longer accept new inputs
`ihc::bitsPerSymbol`	How data is broken into symbols
`ihc::usesPackets`	Expose `startofpacket` and `endofpacket` signals
`ihc::usesValid`	Expose valid signal
`ihc::usesReady`	Expose ready signal

Streaming Interface Function Call APIs

`T read()`	Blocking call to be used in the component
`void write(T data)`	
`T read(bool& sop, bool& eop)`	Blocking call with sideband signals to be used in the component
`void write(T data, bool sop, bool eop)`	
`T tryRead(bool &success)`	Non-blocking call to be used in the component
`bool tryWrite(T data)`	
`T tryRead(bool& success, bool& sop, bool& eop)`	Non-blocking call with sideband signals to be used in the component
`bool tryWrite(T data, bool sop, bool eop)`	

Memory-Mapped Interfaces

Memory-Mapped Interface Declarations

`ihc::mm_master<`*`datatype`*`, /*template arguments*/ >`	Avalon-MM master interface from component

Memory-Mapped Template Arguments

`ihc::dwidth`	Width of data bus in bits
`ihc::awidth`	Width of address bus in bits
`ihc::aspace`	Address space of interface
`ihc::latency`	Guaranteed latency from when a read command exits the component to when the external memory returns valid read data. Variable latency: set value to 0
`ihc::maxburst`	Maximum number of transfers in a single transaction
`ihc::align`	Byte alignment of base pointer address
`ihc::readwrite_mode`	Port direction of the interface
`ihc::waitrequest`	Expose `waitrequest` signal that the slave exerts when it is unable to respond to a read or write request

Memory-Mapped Function Call APIs

`getInterfaceAtIndex(int index)`	Testbench function to index into an `mm_master` interface object

Algorithmic C (AC) Datatypes

Arbitrary Width Integers (`ac_int`)

Declarations

`ac_int<`*`N`*`, true>` *`var_name`*	Signed *N* bit integer
`int`*`N var_name`*	
`ac_int<`*`N`*`,false>` *`var_name`*	Unsigned *N* bit integer
`uint`*`N var_name`*	

Debugging Tools

`#define DEBUG_AC_INT_WARNING`	Runtime tracking of ac_int during x86 emulation, emitting a warning when each overflow is detected
`#define DEBUG_AC_INT_ERROR`	Runtime tracking of ac_int datatypes, erroring out when the first overflow is detected

Arbitrary Precision Fixed-Point Numbers (ac_fixed)

Declarations

`ac_fixed<`*`N`*`, `*`I`*`, true, `*`Q`*`, `*`O`*`>` *`var_name`*	Signed arbitrary precision fixed-point number
`ac_fixed<`*`N`*`, `*`I`*`, false, `*`Q`*`, `*`O`*`>` *`var_name`*	Unsigned arbitrary precision fixed-point number

Where:

N	Total length in bits
I	Number of bits used to represent the integer value
Q	Quantization mode
O	Overflow mode

Complex Numbers (ac_complex)

Declaration

`ac_complex<`*`datatype`*`>` *`var_name (initial_real, initial_imaginary)`*	Complex number of type *datatype*.

System of Tasks

`ihc::launch`	Marks function as a task, and launches task function asynchronously
`ihc::collect`	Synchronizes completion of specified task function in the component
`ihc::stream`	Enables streaming communication between different task functions

`ihc::stream` Template Arguments

`ihc::buffer`	Capacity of FIFO on input data
`ihc::usesPackets`	Exposes `startofpacket` and `endofpacket` signals

The `ihc::stream` object also supports the Streaming Interface Function Call APIs.