Intel® High Level Synthesis Compiler Standard Edition

Best Practices Guide

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

Online Version
Send Feedback

UG-20267

ID: **683259**
Version: **2019.12.18**

Contents

1. Intel® HLS Compiler Standard Edition Best Practices Guide... 3

2. Best Practices for Coding and Compiling Your Component... 5

3. Interface Best Practices...6

3.1. Choose the Right Interface for Your Component..7

3.1.1. Pointer Interfaces.. 7

3.1.2. Avalon Memory Mapped Master Interfaces...10

3.1.3. Avalon Memory Mapped Slave Interfaces...13

3.1.4. Avalon Streaming Interfaces..15

3.1.5. Pass-by-Value Interface... 17

3.2. Avoid Pointer Aliasing... 19

4. Loop Best Practices..20

4.1. Reuse Hardware By Calling It In a Loop.. 21

4.2. Parallelize Loops.. 22

4.2.1. Pipeline Loops.. 22

4.2.2. Unroll Loops..24

4.2.3. Example: Loop Pipelining and Unrolling...24

4.3. Construct Well-Formed Loops.. 27

4.4. Minimize Loop-Carried Dependencies...27

4.5. Avoid Complex Loop-Exit Conditions..29

4.6. Convert Nested Loops into a Single Loop..29

4.7. Declare Variables in the Deepest Scope Possible..29

5. Memory Architecture Best Practices...30

5.1. Example: Overriding a Coalesced Memory Architecture... 31

5.2. Example: Overriding a Banked Memory Architecture...33

5.3. Merge Memories to Reduce Area.. 34

5.3.1. Example: Merging Memories Depth-Wise...34

5.3.2. Example: Merging Memories Width-Wise..37

5.4. Example: Specifying Bank-Selection Bits for Local Memory Addresses........................ 38

6. Datatype Best Practices.. 44

6.1. Avoid Implicit Data Type Conversions.. 44

6.2. Avoid Negative Bit Shifts When Using the `ac_int` Datatype.................................... 45

7. Advanced Troubleshooting...46

7.1. Component Fails Only In Cosimulation 46

7.2. Component Gets Bad Quality of Results... 48

A. Intel HLS Compiler Standard Edition Best Practices Guide Archives............................54

B. Document Revision History for Intel HLS Compiler Standard Edition Best Practices Guide.. 55

1. Intel® HLS Compiler Standard Edition Best Practices Guide

The *Intel® HLS Compiler Standard Edition Best Practices Guide* provides techniques and practices that you can apply to improve the FPGA area utilization and performance of your HLS component. Typically, you apply these best practices after you verify the functional correctness of your component.

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

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

Windows	`C:\intelFPGA_standard\22.1`
Linux	`/home/<username>/intelFPGA_standard/22.1`

About the Intel HLS Compiler Standard Edition Documentation Library

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

Table 1. Intel High Level Synthesis Compiler Standard Edition Documentation Library

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

continued...

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

2. Best Practices for Coding and Compiling Your Component

After you verify the functional correctness of your component, you might want to improve the performance and FPGA area utilization of your component. Learn about the best practices for coding and compiling your component so that you can determine which best practices can help you best optimize your component.

As you look at optimizing your component, apply the best practices techniques in the following areas, roughly in the order listed. Also, review the example designs and tutorials provided with the Intel High Level Synthesis (HLS) Compiler to see how some of these techniques can be implemented.

- Interface Best Practices on page 6

  With the Intel High Level Synthesis Compiler, your component can have a variety of interfaces: from basic wires to the Avalon Streaming and Avalon Memory-Mapped Master interfaces. Review the interface best practices to help you choose and configure the right interface for your component.

- Loop Best Practices on page 20

  The Intel High Level Synthesis Compiler pipelines your loops to enhance throughput. Review these loop best practices to learn techniques to optimize your loops to boost the performance of your component.

- Memory Architecture Best Practices on page 30

  The Intel High Level Synthesis Compiler infers efficient memory architectures (like memory width, number of banks and ports) in a component by adapting the architecture to the memory access patterns of your component. Review the memory architecture best practices to learn how you can get the best memory architecture for your component from the compiler.

- Datatype Best Practices on page 44

  The datatypes in your component and possible conversions or casting that they might undergo can significantly affect the performance and FPGA area usage of your component. Review the datatype best practices for tips and guidance how best to control datatype sizes and conversions in your component.

- Alternative Algorithms

  The Intel High Level Synthesis Compiler lets you compile a component quickly to get initial insights into the performance and area utilization of your component. Take advantage of this speed to try larger algorithm changes to see how those changes affect your component performance.

3. Interface Best Practices

With the Intel High Level Synthesis Compiler, your component can have a variety of interfaces: from basic wires to the Avalon Streaming and Avalon Memory-Mapped Master interfaces. Review the interface best practices to help you choose and configure the right interface for your component.

Each interface type supported by the Intel HLS Compiler Standard Edition has different benefits. However, the system that surrounds your component might limit your choices. Keep your requirements in mind when determining the optimal interface for your component.

Demonstrating Interface Best Practices

The Intel HLS Compiler comes with tutorials that give you working examples to review and run. They demonstrate good coding practices and illustrate important concepts.

Review the following tutorials to learn about different interfaces as well as best practices that might apply to your design:

Tutorial	Description
You can find these tutorials in the following location on your Intel Quartus Prime system: `<quartus_installdir>/hls/examples/tutorials`	
`interfaces/overview`	Demonstrates the effects on quality-of-results (QoR) of choosing different component interfaces even when the component algorithm remains the same.
`best_practices/parameter_aliasing`	Demonstrates the use of the `restrict` keyword on component arguments.
`interfaces/explicit_streams_buffer`	Demonstrates how to use explicit `stream_in` and `stream_out` interfaces in the component and testbench.
`interfaces/explicit_streams_packets_ready_valid`	Demonstrates how to use the `usesPackets`, `usesValid`, and `usesReady` stream template parameters.
`interfaces/mm_master_testbench_operators`	Demonstrates how to invoke a component at different indicies of an Avalon Memory Mapped (MM) Master (`mm_master` class) interface.
`interfaces/mm_slaves`	Demonstrates how to create Avalon-MM Slave interfaces (slave registers and slave memories).
`interfaces/multiple_stream_call_sites`	Demonstrates the tradeoffs of using multiple stream call sites.
`interfaces/pointer_mm_master`	Demonstrates how to create Avalon-MM Master interfaces and control their parameters.
`interfaces/stable_arguments`	Demonstrates how to use the `stable` attribute for unchanging arguments to improve resource utilization.

Related Information

- Avalon Memory-Mapped Interface Specifications
- Avalon Streaming Interface Specifications

3.1. Choose the Right Interface for Your Component

Different component interfaces can affect the quality of results (QoR) of your component without changing your component algorithm. Consider the effects of different interfaces before choosing the interface for your component.

The best interface for your component might not be immediately apparent, so you might need to try different interfaces for your component to achieve the optimal QoR. Take advantage of the rapid component compilation time provided by the Intel HLS Compiler and the resulting High Level Design reports to determine which interface gives you the optimal QoR for your component.

This section uses a vector addition example to illustrate the impact of changing the component interface while keeping the component algorithm the same. The example has two input vectors, vector `a` and vector `b`, and stores the result to vector `c`. The vectors have a length of `N` (which could be very large).

The core algorithm is as follows:

```
#pragma unroll 8
for (int i = 0; i < N; ++i) {
  c[i] = a[i] + b[i];
}
```

The Intel HLS Compiler extracts the parallelism of this algorithm by pipelining the loops if no loop dependency exists. In addition, by unrolling the loop (by a factor of 8), more parallelism can be extracted.

Ideally, the generated component has a latency of `N`/8 cycles. In the examples in the following section, a value of 1024 is used for `N`, so the ideal latency is 128 cycles (1024/8).

The following sections present variations of this example that use different interfaces. Review these sections to learn how different interfaces affect the QoR of this component.

You can work your way through the variations of these examples by reviewing the tutorial available in `<`*`quartus_installdir`*`>/hls/examples/tutorials/interfaces/overview`.

3.1.1. Pointer Interfaces

Software developers accustomed to writing code that targets a CPU might first try to code this algorithm by declaring vectors `a`, `b`, and `c` as pointers to get the data in and out of the component. Using pointers in this way results in a single Avalon Memory-Mapped (MM) Master interface that the three input variables share.

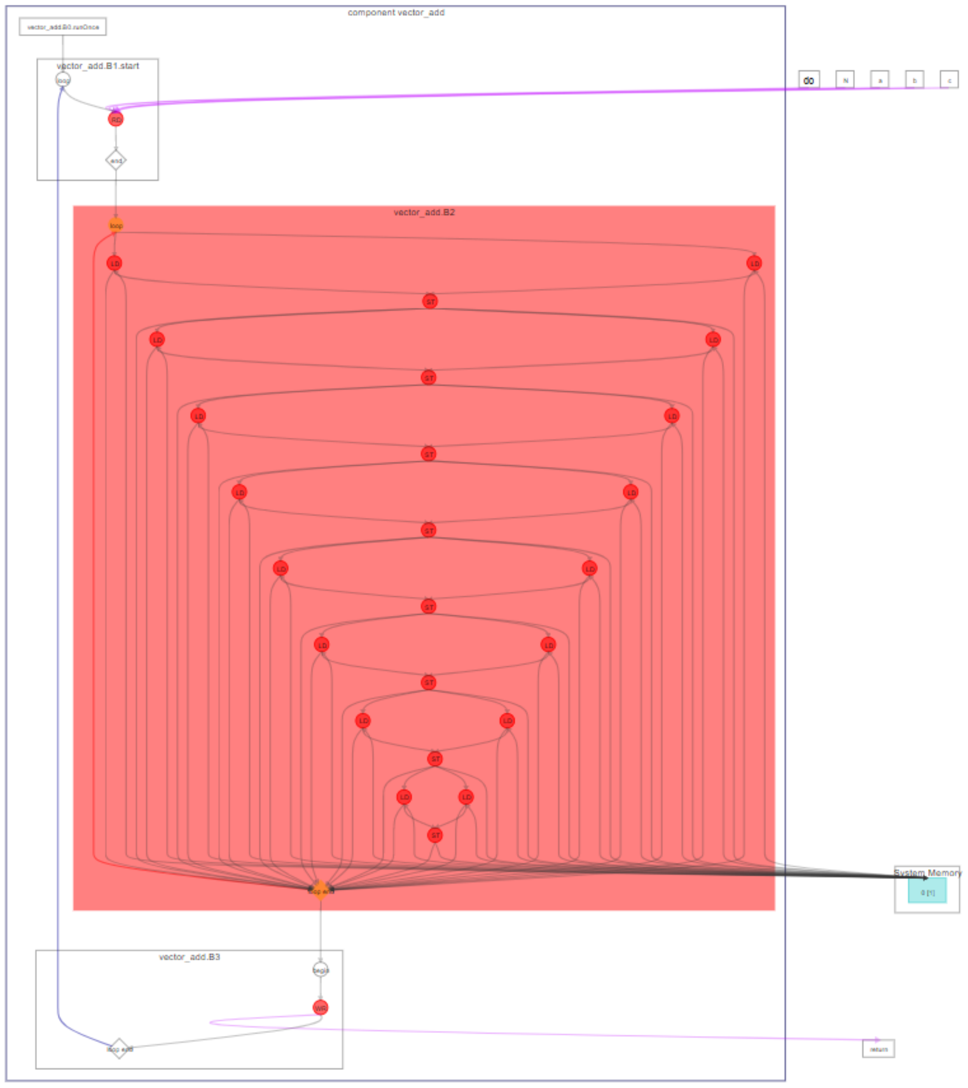
Pointers in a component are implemented as Avalon® Memory Mapped (Avalon MM) master interfaces with default settings. For more details about pointer parameter interfaces, see Intel HLS Compiler Default Interfaces in *Intel High Level Synthesis Compiler Standard Edition Reference Manual*.

The vector addition component example with pointer interfaces can be coded as follows:

```
component void vector_add(int* a,
                          int* b,
                          int* c,
                          int N) {
  #pragma unroll 8
  for (int i = 0; i < N; ++i) {
    c[i] = a[i] + b[i];
  }
}
```

The following diagram shows the Component Viewer report generated when you compile this example. Because the loop is unrolled by a factor of 8, the diagram shows that `vector_add.B2` has 8 loads for vector `a`, 8 loads for vector `b`, and 8 stores for vector `c`. In addition, all of the loads and stores are arbitrated on the same memory, resulting in inefficient memory accesses.

Figure 1. Component View of `vector_add` Component with Pointer Interfaces



The following Loop Analysis report shows that the component has an undesirably high loop initiation interval (II). The II is high because vectors a, b, and c are all accessed through the same Avalon MM Master interface. The Intel HLS Compiler uses stallable arbitration logic to schedule these accesses, which results in poor performance and high FPGA area use.

In addition, the compiler cannot assume there are no data dependencies between loop iterations because pointer aliasing might exist. The compiler cannot determine that vectors a, b, and c do not overlap. If data dependencies exist, the Intel HLS Compiler cannot pipeline the loop iterations effectively.