1. Intel® High Level Synthesis Compiler Pro Edition User Guide

The Intel® HLS Compiler Pro Edition User Guide provides instructions on synthesizing, verifying, and simulating IP that you design for Intel FPGA products. The Intel High Level Synthesis (HLS) Compiler is sometimes referred to as the i++ compiler, reflecting the name of the compiler command.

Compared to traditional RTL development, the Intel HLS Compiler offers the following advantages:

- Fast and easy verification
- Algorithmic development in C++
- Automatic integration of RTL verification with a C++ testbench
- Powerful microarchitecture optimizations

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_pro\20.2
- Linux /home/<username>/intelFPGA_pro/20.2

About the Intel HLS Compiler Pro Edition Documentation Library

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

<table>
<thead>
<tr>
<th>Table 1. Intel High Level Synthesis Compiler Pro Edition Documentation Library</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title and Description</strong></td>
</tr>
<tr>
<td>Release Notes</td>
</tr>
<tr>
<td>Provide late-breaking information about the Intel HLS Compiler.</td>
</tr>
<tr>
<td>Getting Started Guide</td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td>User Guide</td>
</tr>
<tr>
<td>Provides instructions on synthesizing, verifying, and simulating intellectual property (IP) that you design for Intel FPGA products. Go through the entire development flow of your component from creating your component and testbench up to integrating your component IP into a larger system with the Intel Quartus Prime software.</td>
</tr>
</tbody>
</table>

*Other names and brands may be claimed as the property of others.
<table>
<thead>
<tr>
<th>Title and Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reference Manual</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Best Practices Guide</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Quick Reference</strong></td>
<td>Provides a brief summary of Intel HLS Compiler declarations and attributes on a single two-sided page.</td>
</tr>
</tbody>
</table>
2. Overview of the Intel High Level Synthesis (HLS) Compiler Pro Edition

The Intel High Level Synthesis (HLS) Compiler parses your design and compiles it to an x86-64 object or RTL code optimized for Intel FPGA device families. It also creates an executable testbench. Use the x86-64 object to quickly test and debug the function of your design.

You can use the same testbench to verify your design both when it is compiled to x86-64 instructions or to RTL.

The Intel HLS Compiler Pro Edition is command-line compatible with g++, and supports most of the g++ compiler flags. See the Intel High Level Synthesis Compiler Reference Manual for a full list of compiler flags.

The Intel HLS Compiler Pro Edition recognizes the same file name extensions as g++, namely .c, .C, .cc, .cpp, .CPP, .C++, .cp, and .cxx. The compiler treats all of these file types as C++. The compiler does not explicitly support C, other than as a subset of C++.

Important: The Intel HLS Compiler Pro Edition treats all input files as C++17. The compiler does not support files conforming to newer C++ standards.

When you target the compilation to an FPGA, the Intel HLS Compiler outputs an executable and a project directory. The default executable is a.out on Linux and a.exe on Windows. The default project directory is a.prj, and it contains the following outputs:

• Generated IP
• High-Level Design Reports (report.html)
• Verification testbench files
• Quartus project that you can use to accurately estimate area requirements and \( f_{\text{MAX}} \) for your design.

To specify the name of the compiler output, include the \(-o <\text{result}>\) option in your i++ command, where \(<\text{result}>\) is the name of the executable. This command creates a project directory called \(<\text{result}>.prj\).

Running the executable file runs your testbench. When you target the compilation to an x86-64 architecture, the output executable runs your design on the CPU like a regular C++ program. The output executable runs very quickly compared to running a simulation of your component RTL. When you target the compilation to an FPGA architecture, the output executable simulates your component RTL. This simulation can take a long time to run.
2.1. High Level Synthesis Design Flow

The Intel High Level Synthesis (HLS) Compiler helps speed your IP development by letting you compile your IP component C++ code to different targets, depending on where you are in your IP development cycle.

The typical design flow when you use the Intel HLS Compiler Pro Edition consists of the following stages:

1. Creating your component and testbench.
   You can write a complete C++ application that contains both your component code and your testbench code.
   For details, see Creating a High-Level Synthesis Component and Testbench on page 8.

2. Verify the functionality of your component algorithm and testbench.
   Verify the functionality by compiling your design to an x86-64 executable and running the executable. For details, see Verifying the Functionality of Your Design on page 10.

3. Optimize and refine the FPGA performance of your component.
   Optimize the FPGA performance of your component by compiling your design to an FPGA target and reviewing the high-level design report to see where you can optimize your component. This step generates RTL code for your component. For details, see Optimizing and Refining Your Component on page 11.
   After initial optimizations, you can see where to further refine your component by simulating it. For details, see Verifying Your IP with Simulation on page 15.

4. Synthesize your component with Intel Quartus Prime.
   For details, see Synthesize your Component IP with Intel Quartus Prime Pro Edition on page 19.
   Synthesizing your component generates accurate quality-of-results (QoR) metrics like FPGA area utilization and $f_{\text{MAX}}$.

5. Integrate your IP into a system with Intel Quartus Prime or Platform Designer (formerly Qsys).
   For details, see Integrating your IP into a System on page 20.

The following flowchart shows a coarse-grained progression through the stages of a typical Intel High Level Synthesis (HLS) Compiler design flow.
Create component and test bench

Compile design with the following command to generate IP and a testbench executable to verify your design in simulation:

```
i++ -march=<FPGA_family_or_part_number>
```

Run a Quartus Prime compilation on the project in the `<result>.prj/quartus` directory to generate QoR metrics from Quartus Prime software.

Integrate your component into a system with Quartus Prime or Platform Designer

(Nota: You can debug your design with GDB on Linux or Visual Studio debugger on Windows, even for the i++ output)

2.2. The Project Directory

The project directory (`<result>.prj`) that the Intel HLS Compiler Pro Edition outputs has four subdirectories.

Table 2. Subdirectories within the .prj Directory

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>components</td>
<td>Contains a folder for each component, and all HDL and IP files that are needed to use that component in a design.</td>
</tr>
<tr>
<td>verification</td>
<td>Contains all the files for the verification testbench.</td>
</tr>
<tr>
<td>reports</td>
<td>Contains the High-Level Design Reports. The High-Level Design Reports are a set of reports and viewers that you open in a web browser. Use the reports to analyze the synthesized hardware implementation of your components and tasks.</td>
</tr>
<tr>
<td>quartus</td>
<td>Contains an Intel Quartus Prime project that instantiates the components. You can compile this Intel Quartus Prime project to generate more detailed timing and area reports. Do not use the contents of this subdirectory to integrate your component in a design. Use the contents of the components directory.</td>
</tr>
</tbody>
</table>
3. Creating a High-Level Synthesis Component and Testbench

The Intel HLS Compiler Pro Edition converts individual functions into RTL code. The components are part of a C++ application that acts as a testbench for your component functions, and you can test your components by calling them from your main() function and verifying that the output is correct.

While the compiler supports C++17, you can often achieve better component performance by using the supported subset of C99 whenever possible. The compiler is capable of synthesizing some C++ constructs, which might be easier for you to use to create cleaner code.


The Intel HLS Compiler Pro Edition synthesizes all the code in the function or functions that you label as components, and any code that these components call, to an RTL representation.

You can identify a function in your C++ application that you want to synthesize into RTL in one of the following ways:

- Insert the component keyword in the source code before the top-level C++ function to be synthesized.
- Specify the function on the command line by using the --component <component_list> option of the i++ command.

To use this option, your component must be configured with C-linkage using the extern "C" specification. For example:

```
extern "C" int myComponent(int a, int b)
```
Important: Components are synthesized into RTL for all functions labeled with the `component` keyword and for all components listed in the `--component <component_list>` option of the `i++` command. Avoid combining these methods because you might unexpectedly synthesize unwanted components.

If you do not want components synthesized into RTL for a function, ensure that you do not have the `component attribute` specified in the function and ensure that the function is not specified in the `--component <component_list>` option of the `i++` command.

You can see which components were synthesized into RTL in the summary page of the High-Level Design Reports (<name>.prj/reports/report.html). For more information about the High-Level Design Reports, see The High-Level Design Reports on page 11.

The HLS compiler creates an executable to run on the CPU. The compiler then sends any calls to functions that you declared as components to simulation of the synthesized IP core, and the simulation results are returned.

3.1. Intel HLS Compiler Pro Edition Compiler-Defined Preprocessor Macros

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

Table 3. Macro Definition for ___INTELFPGA_COMPILER___

<table>
<thead>
<tr>
<th>Tool Invocation</th>
<th><em><strong>INTELFPGA_COMPILER</strong></em></th>
</tr>
</thead>
<tbody>
<tr>
<td>g++ or cl</td>
<td>Undefined</td>
</tr>
<tr>
<td>i++ -march=x86-64</td>
<td>2020</td>
</tr>
<tr>
<td>i++ -march=&quot;&lt;FPGA_family_or_part_number&gt;&quot;</td>
<td>2020</td>
</tr>
</tbody>
</table>

Table 4. Macro Definition for HLS_SYNTHESIS

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

3. Creating a High-Level Synthesis Component and Testbench
4. Verifying the Functionality of Your Design

Verify the functionality of your design by compiling your component and testbench to an x86-64 executable that you can debug with a native C++ debugger.

Compiling your design to an x86-64 executable is faster than compiling your component to hardware or a hardware simulation. This faster compilation time lets you debug and refine your component algorithms quickly before you move on to see how your component is implemented in hardware.

You can compile your component and testbench to an x86-64 executable for functional verification through any of the following methods:

- Use the `i++ -march=x86-64` command.
- On Linux systems, use the `g++` command.
- On Windows systems, use Microsoft Visual Studio.

Ensure that you set your compiler command to include debug information. The `i++` command generates debug information by default.

On Linux systems, you can use GDB to debug your component and testbench, even if you used the `i++` command to compile your code for functional verification.

On Windows systems, you can use Microsoft Visual Studio to debug your component and testbench, even if you used the `i++` command to compile your code for functional verification.

Using the `g++` command or Microsoft Visual Studio might require additional configuration to compile your Intel HLS Compiler Pro Edition code. For details, see Compiler Interoperability in the Intel High Level Synthesis Compiler Pro Edition Reference Manual.

You can automate the process by using a makefile or batch script. Use the makefiles and scripts provided in the Intel HLS Compiler Pro Edition example designs and tutorials as guides for creating your own makefiles or batch scripts.
5. Optimizing and Refining Your Component

After you have verified the functionality of your component and testbench, you can compile your component to RTL and review the High-Level Design Reports to further optimize and refine your component design. The High-Level Design Reports show estimates of various aspects of how your component will be implemented in hardware.

By compiling your component to RTL and reviewing the High-Level Design Reports, you can see how your code changes affect your component hardware implementation without needing to run a simulation or a full Quartus compilation.

To compile your component to RTL without running a simulation, issue the following command:

```bash
i++ -march="<FPGA_family_or_part_number>" --simulator none
```

You can also compile your component with a ModelSim* simulation flow by omitting the `--simulator none` option. Compiling without a simulation test bench is faster, but you cannot simulate your design to measure its latency and generate waveforms.

To view the High-Level Design Reports, open the following file in a web browser:

```
<result>.prj/reports/report.html
```

For information about techniques that you can apply to optimize and refine your component, see *Intel High Level Synthesis Compiler Pro Edition Best Practices Guide*.

5.1. The High-Level Design Reports

The High-Level Design Reports are a group of reports and viewers that you can use to help optimize your design by reviewing the statistics and visualizations that the reports provide.

Access the High-Level Design Reports by using a web browser to open the `report.html` file found in the `<name>.prj/reports` folder created when you compile your component to RTL.

Use the tutorials provided with the Intel HLS Compiler to view examples of the reports and learn how to use the reports and viewers to help optimize and refine your component design.

For details about using the reports to help optimize your design, review *Reviewing the High-Level Design Reports (report.html)* on page 22.

In some cases, information in the reports and viewers are estimates and might not match the results from compiling your design with Intel Quartus Prime software. However, Compiling your component using the Intel Quartus Prime software might take several hours. In contrast, the Intel HLS Compiler can generate the High Level Design Report in minutes for most designs.
Tip: Compiling your component using the Intel Quartus Prime Pro Edition software might take several hours. In contrast, the Intel HLS Compiler Pro Edition can generate the High Level Design Report in minutes for most designs.

The reports are divided into the following categories:

- The report under **Summary** gives you a quick overview of the results of compiling your design including a summary of each component in your design and a summary of the estimated resources that each component in your design uses.
- The reports under **Throughput Analysis** help you optimize your design based results from analyzing loops and providing key performance metrics on component blocks.
- The reports under **Area Analysis** help you locate area usage inefficiency. The reports provide a detailed breakdown of the estimated FPGA area usage. It also provides feedback on key hardware features such as private memory configuration.
- The viewers under **System Viewers** provide different views into the structure, interfaces, datapaths, and computation flows in your component. These views can help identify bottlenecks in your design.

### Table 5. High-Level Design Reports

<table>
<thead>
<tr>
<th>Category</th>
<th>Report or Viewer Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>Summary</td>
<td>The Summary Report gives you a quick overview of the results of compiling your design including a summary of each component in your design and a summary of the estimated resources that each component in your design uses. The <strong>Functions</strong> section shows you the short names generated for overloaded and templated functions in your hardware design by the Intel HLS Compiler to prevent name collisions. These short names are used in other parts of the High Level Design Report. Other sections of the summary contain information only after you compile your design with Intel Quartus Prime software.</td>
</tr>
</tbody>
</table>
| Throughput Analysis     | Loop Analysis         | The Loop Analysis Report shows the loop optimization strategies either applied by the compiler or through the various loop pragmas available. The report also provides key performance metrics on all blocks including any target II, scheduled \( f_{\text{MAX}} \), block II, and maximum interleaving iterations. You can use this report to help you with the following tasks:  
  - Identify \( f_{\text{MAX}} \) bottleneck in your design
  - Set proper \( f_{\text{MAX}} \) target for your design
  - Estimate the execution latency
  - Determine where to use loop pragmas
  **Important:** The scheduled \( f_{\text{MAX}} \) that this report displays is not an accurate estimate of the \( f_{\text{MAX}} \) that your design can achieve. Synthesize your component with Intel Quartus Prime to ensure that your design meets your performance requirements. You might also find that you can lower your scheduled \( f_{\text{MAX}} \) target to save FPGA area utilization. |
|                         | \( f_{\text{MAX}} \) II Report | This report is deprecated and might be removed in a future release. Its function has been merged into the Loop Analysis Report. |
|                         | Verification Statistics | For each component that the testbench calls, the verification statistics report provides information such as the number and type of invocations, latency, initiation interval, and throughput. This report is available only after you simulate your component. |

**continued...**
<table>
<thead>
<tr>
<th>Category</th>
<th>Report or Viewer Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area Analysis</td>
<td>Area Analysis of System</td>
<td>This report provides a detailed breakdown of the estimated FPGA area usage of your design. It also provides information about key hardware features such as private memory configuration. The Quartus Fit... sections are empty or hidden until you compile your design with Intel Quartus Prime software. The estimated area usage information correlates with, but does not necessarily match, the resource usage results from the Intel Quartus Prime software. Use the estimated area usage to identify parts of the design with large area overhead. You can also use the estimates to compare area usage between different designs. <strong>Do not use the estimated area usage information for final resource utilization planning.</strong></td>
</tr>
<tr>
<td>Area Analysis of Source</td>
<td>This report is deprecated and might be removed in a future release. Use the Area Analysis of System report instead.</td>
<td></td>
</tr>
<tr>
<td>System Viewers</td>
<td>Graph Viewer</td>
<td>The Graph Viewer is an interactive view of your system that allows you to review information such as the sizes and types of loads and stores, stalls, and latencies. Clicking on different entries in the hierarchical graph list displays different views of your design.</td>
</tr>
</tbody>
</table>

![Graph List (Beta)](image)

**System view**

**Function view**

**Block views**

**Cluster views**

*continued...*
### Category | Report or Viewer Name | Description
--- | --- | ---
 | System View | The System View of the Graph Viewer provides a high-level view of components and tasks in your system.
 | Function View | The Function views of the Graph Viewer show the blocks inside your component and any tasks. A function appears in this view if it is declared as component or launched as a task function. Other called functions are automatically inlined.
 | Block View | The Block views of the Graph Viewer show stallable operations in your component and task functions, and the clusters and their connections.
 | Cluster View | The Cluster views of the Graph Viewer shows the contents of clusters in your components.
 | Function Memory Viewer | The Function Memory Viewer shows you the memory system that the Intel HLS Compiler generated for your component. Use the Function Memory Viewer to help you identify data movement bottlenecks in your component design.
 | Schedule Viewer | The Schedule Viewer shows the estimated start and ending clock cycle for functions, blocks, clusters, and individual instructions in your design.
6. Verifying Your IP with Simulation

When compiling your component to an FPGA architecture, the Intel HLS Compiler Pro Edition links your design C++ testbench with an RTL-compiled version of your component that runs in an RTL simulator.

The Intel HLS Compiler Pro Edition uses Mentor Graphics ModelSim software to perform the simulation. You must have ModelSim installed to use the Intel HLS Compiler. For a list of supported versions of the ModelSim software, refer to the EDA Interface Information section in the Intel Quartus Prime Software and Device Support Release Notes.

- To verify the functional correctness of your IP with your C++ testbench, run the executable that the compiler generates by targeting the FPGA architecture. By default, the name of the executable is `a.out` (Linux) or `a.exe` (Windows).

Example command you might invoke for a simple single-file design:

- **Linux**: `i++ -march="Arria10" [...] design.cpp && ./a.out`
- **Windows**: `i++ -march="Arria10" [...] design.cpp && a.exe`

**Related Information**

- Mentor Graphics ModelSim Software Prerequisites for the Intel HLS Compiler
- EDA Interface Information (Intel Quartus Prime Standard Edition) Software
- EDA Interface Information (Intel Quartus Prime Pro Edition) Software

6.1. Generation of the Verification Testbench Executable

When you include `-march="<FPGA_family_or_part_number>"` in your `i++` command, the HLS compiler identifies the components and performs high-level synthesis on them. It then generates an executable to run a verification testbench.

The HLS compiler performs the following tasks to generate the verification executable:

1. Parses your design, and extracts the functions and symbols necessary for component synthesis to the FPGA. The HLS compiler also extracts the functions and symbols necessary for compiling the C++ testbench.
2. Compiles the testbench code to generate an x86-64 executable that also runs the simulator.
3. Compiles the code for component synthesis to the FPGA. This compilation generates RTL for the component and an interface to the x86-64 executable testbench.
6.2. Debugging during Verification

By default, the HLS compiler instructs the simulator not to log any signals because logging signals slows the simulation, and the waveforms files can be very large. However, you can configure the compiler to save these waveforms for debugging purposes.

To enable signal logging in the simulator, invoke the `i++` command with the `--ghdl` option in your `i++` command, as follows:

```
i++ -march="<FPGA_family_or_part_number>" --ghdl <input files>
```

**Remember:** After you compile your component and testbench with the `--ghdl` option, run the resulting executable to run the simulation and generate the waveform. By default, the name of the executable is `a.out` (Linux) or `a.exe` (Windows).

When the simulation finishes, open the `vsim.wlf` file inside the `<result>.prj/verification` directory to view the waveform.

To view the waveform after the simulation finishes:
1. In ModelSim, open the `vsim.wlf` file inside the `<result>.prj/verification` directory.
2. Right-click the `<component_name>_inst` block and select **Add Wave**.

You can now view the component top-level signals: `start`, `busy`, `stall`, `done`, parameters, and outputs. Use the waveform to see how the component interacts with its interfaces.

**Tip:** When you view the simulation waveform in ModelSim, the simulation clock period is set to a default value of 1000 picoseconds (ps). To synchronize the Time axis to show one cycle per tick mark, change the time resolution from picoseconds (ps) to nanoseconds (ns):
   a. Right-click the timeline and select **Grid, Timeline & Cursor Control**.
   b. Under **Timeline Configuration**, set the **Time units** to **ns**.

6.3. High-Throughput Simulation (Asynchronous Component Calls) Using Enqueue Function Calls

An explicit call to a component in simulation is a blocking call. To be consistent with C++ language conventions, the testbench waits for a return value from the component before continuing execution. This blocking call results in serial execution of the component. You can test how well successive invocations of your component can be pipelined by queueing inputs to the component before executing the component. You can queue inputs to a component that has explicit interfaces by using enqueue function calls from the simulation library. Estimate the throughput of your component by dividing the component $f_{\text{MAX}}$ by the component initiation interval (II), which indicates approximately how many times your component is invoked per second.
Table 6. Functions from Simulation Library for Queuing Inputs to the Component with Explicit Interfaces

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ihc_hls_enqueue(void* retptr, void* funcptr, ...)</td>
<td>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 does not execute until the ihc_hls_component_run_all() function is invoked.</td>
</tr>
<tr>
<td>ihc_hls_enqueue_noret(void* funcptr, ...)</td>
<td>This function is similar to ihc_hls_enqueue(void* retptr, void* funcptr, ...), except that it does not have an output pointer to capture return values.</td>
</tr>
<tr>
<td>ihc_hls_component_run_all (void* funcptr)</td>
<td>This function executes all enqueued calls to the specified component in a pipelined fashion.</td>
</tr>
</tbody>
</table>

6.3.1. Execution Model

Execution of enqueued component calls only occurs when the ihc_hls_component_run_all(void* funcptr) function is called. All externally visible side effects of the execution (for example, return data, pointers, or masters) are not visible in the testbench until the ihc_hls_component_run_all() function explicitly triggers the execution.

6.3.2. Comparison of Explicit and Enqueued Function Calls

The ihc_hls_enqueue and ihc_hls_enqueue_noret functions allow a new invocation of a component to start every cycle if the component can be pipelined with a component initiation interval (II) of one. If the component II is greater than one, then the component invocation starts after II number of cycles.

Figure 2 on page 18 illustrates the waveform of the signals for the component dut. The testbench does not include any enqueue function calls.

```c
#include "HLS/hls.h"
#include <stdio.h>

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

int main (void) {
    int x1, x2, x3;
    x1 = dut(1, 2);
    x2 = dut(3, 4);
    x3 = dut(5, 6);
    printf("x1 = %d, x2 = %d, x3 = %d\n", x1, x2, x3);
    return 0;
}
```
Figure 2. **Waveform Diagram of the Signals for Component dut Without Enqueue Function Calls**

Figure 3 on page 18 illustrates the waveform of the signals for the component dut when the testbench includes enqueue function calls. Observe how the component is passed new data each clock cycle, and compare this waveform with the earlier waveform.

```c
#include "HLS/hls.h"
#include <stdio.h>

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

int main (void) {
    int x1, x2, x3;
    ihc_hls_enqueue(&x1, &dut, 1, 2);
    ihc_hls_enqueue(&x2, &dut, 3, 4);
    ihc_hls_enqueue(&x3, &dut, 5, 6);
    ihc_hls_component_run_all(&dut);
    printf("x1 = %d, x2 = %d, x3 = %d\n", x1, x2, x3);
    return 0;
}
```

Figure 3. **Waveform Diagram of the Signals for Component dut With Enqueue Function Calls**
7. Synthesize your Component IP with Intel Quartus Prime Pro Edition

When you are satisfied with the predicted performance of your component, use Intel Quartus Prime Pro Edition to synthesize your component. Synthesis also generates accurate area and performance ($f_{\text{MAX}}$) estimates for your design, however your design is not expected to cleanly close timing in the Intel Quartus Prime reports.

You can expect to see timing closure warnings in the Intel Quartus Prime logs because the generated project in the Intel HLS Compiler quartus folder targets a clock speed of 1000 MHz to achieve the best possible placement for your design. The $f_{\text{MAX}}$ value presented in the High-Level Design Reports is an estimate of the maximum clock rate that your component can cleanly close timing for.

After the Intel Quartus Prime compilation completes, the High-Level Design Reports show the area and performance data for your components. These estimates are more accurate than estimates generated when you compile your component with the Intel HLS Compiler Pro Edition.

Typical Intel Quartus Prime compilation times can take minutes to hours depending on the size and complexity of your components.

To synthesize your component IP and generate quality of results (QoR) data, do one of the following actions:

- Instruct the HLS compiler to run the Intel Quartus Prime compilation flow automatically after synthesizing the components. Include the --quartus-compile option in your i++ command.

  ```
  i++ -march="<FPGA_family_or_part_number>" --quartus-compile ...
  ```

  If you are trying to get an accurate estimate of the $f_{\text{MAX}}$ range of your component, use the --quartus-seed option to automate seed sweeps:

  ```
  i++ -march="<FPGA_family_or_part_number>" --quartus-compile --quartus-seed <num_of_sweeps> ...
  ```

- If you already have the RTL for your component synthesized, you can navigate to the quartus directory and compile the Intel Quartus Prime project by invoking the following command:

  ```
  quartus_sh --flow compile quartus_compile
  ```

*Tip:* Add the path to quartus_sh (Linux) or quartus_sh.exe (Windows) to your PATH environment variable.
8. Integrating your IP into a System

To integrate your HLS compiler-generated IP into a system with Intel Quartus Prime, you must be familiar with Intel Quartus Prime Pro Edition as well as the Platform Designer (formerly Qsys Pro) system integration tool included with Intel Quartus Prime.

The `<result>.prj/components` directory contains all the files you need to include your IP in an Intel Quartus Prime Pro Edition project.

The IP that the HLS compiler generates for each component is self contained. You can move the folders in the `components` directory to a different location or machine if desired.

For overloaded and templated functions in your hardware design, the Intel HLS Compiler sometimes generates short names to prevent name collisions. Use these short names when integrating the generated RTL into your design.

Review the Summary Report in the High-Level Design Reports (`report.html`) to see any short names generated by the compiler.

8.1. Adding the HLS Compiler-Generated IP into an Intel Quartus Prime Pro Edition Project

To use the IP generated by the Intel HLS Compiler in an Intel Quartus Prime Pro Edition project, you must first add the `.ip` file to the project.

The `.ip` file contains information to add to all of the necessary HDL files for the component. It also applies to any component-specific Intel Quartus Prime Settings File (QSF) settings that are necessary for IP synthesis.

1. Create an Intel Quartus Prime Pro Edition project.
2. Click `Project ➤ Add/Remove Files in Project`.
3. In the `Settings` dialog box, browse to and select the component `.ip` file:
   
   For example, `<result>.prj/components/<component_name>/<component_name>.ip`

4. Instantiate the component top-level module in the Intel Quartus Prime project. For an example on how to instantiate the component’s top-level module, refer to the `<result>.prj/components/<component_name>/<component_name>_inst.v` file.

8.2. Adding the HLS Compiler-Generated IP into a Platform Designer System

To use the HLS compiler-generated IP in a Platform Designer (formerly Qsys Pro) System, you must first add the directory to the IP search path or the IP Catalog.
In Platform Designer, if your HLS compiler-generated IP does not appear in the IP Catalog, perform the following tasks:

1. In Intel Quartus Prime, click **Tools ➤ Options**.
2. In the **Options** dialog box, under Category, expand **IP Settings** and click **IP Catalog Search Locations**.
3. In the **IP Catalog Search Locations** dialog box, add the path to the directory that contains the `.ip` file to IP Search Paths as `<result>.prj/components/<component_name>/<component_name>`.
4. In **IP Catalog**, add your IP to the Platform Designer system by selecting it from the HLS project directory.


For an example of adding HLS compiler-generated IP to a Platform Designer systems, review the following tutorial: `<quartus_installdir>/hls/examples/tutorials/usability/platform_designer_stitching`. 
A. Reviewing the High-Level Design Reports (report.html)

After compiling your component, the Intel HLS Compiler generates a group of reports and viewers that helps you analyze various component aspects, such as area, loop structure, memory usage, and component pipeline. To launch the High-Level Design Reports, open the following file in a web browser: `<result>.prj/reports/report.html`.

A.1. Reviewing the Summary Report

The Summary Report gives you a quick overview of the results of compiling your design including a summary of each component in your design and a summary of the estimated resources that each component in your design uses.

The Summary report is divided into the following sections based on the order of compilation:

- Compile Info
- Functions
- Clock Frequency Summary
- Quartus Estimated Resource Utilization Summary
- HLS Estimated Resource Utilization Summary
- Warnings Summary
A. Reviewing the High-Level Design Reports (report.html)

Compile Info

The Compile Info section shows general information about the compile including the following items:

- Name of the project
- Target FPGA family and device
- HLS compiler version
- Intel Quartus Prime version
- The command that was used to compile the design
- The date and time at which the reports were generated
Functions
For overloaded and templated functions in your hardware design, this section of the Summary Report shows you the short names generated by the Intel HLS Compiler for the functions to prevent name collisions.

These short names are used in other parts of the High Level Design Report.

The Synthesized Function Name Mapping section does not show testbench functions. Only functions that are synthesizable to hardware are shown.

Clock Frequency Summary
After you compile your design with Intel Quartus Prime software, the Clock Frequency Summary shows the following:

- Quartus Fitter Clock Frequency
- Compiler Target Frequency (MHz)
- Compiler estimated frequency (MHz)

The Quartus Fitter clock frequency is the maximum clock frequency that can be achieved for the design. When the compiler estimates a lower frequency than the targeted frequency, the frequency value is highlighted in red.

Both the Functions section and Clock Frequency Summary display the target clock frequency applied at the source on the component. When the values of the source is different than the compilation flag you applied, the Clock Frequency Summary Compiler Target Frequency shows “Various” instead of reporting a number.

Quartus Estimated Resource Utilization Summary
After you compile your design with Intel Quartus Prime software, the Quartus Estimated Resource Utilization Summary section shows the total area utilization both for the entire design, and for each component individually. There is no breakdown of area information by source line.

Function Summary
When you compile your design to target Intel Agilex™ or Intel Stratix® 10 devices, this section of the report indicates whether a function in your design uses a modified handshaking protocol.

You can control the use of the modified handshaking protocol with the --hyper-optimized-handshaking option of the i++ command.

HLS Estimated Resource Utilization Summary
The HLS Estimated Resource Utilization Summary section shows a summary of the estimated resources used by each component in your design, as well as the total resources used for all components.

Warnings Summary
The Warnings Summary section shows the compiler warnings generated during the compilation.
A.2. Reviewing Factors That Affect Throughput

A.2.1. Reviewing Loop Information

The Loops Analysis report in the High-Level Design Reports (\texttt{<result>.prj/reports/report.html}) contains information about all the loops (coalesced, unrolled, and fused loops) in your design and their unroll statuses. This report helps you examine whether the Intel HLS Compiler Pro Edition can maximize the throughput of your component.

\textit{Note:} The f\textsubscript{MAX} II report is now deprecated and its information is merged with the Loop Analysis report.

To access the report, click Throughput Analysis $\rightarrow$ Loop Analysis. The left-hand Loops List pane displays the following types of loops:

- Fused loops
- Fused subloops
- Coalesced loops
- Fully unrolled loops
- Partial unrolled loops
- Regular loops

The Loops Analysis report captures the following key performance metrics on all blocks:

- \textbf{Source Location}: Indicates the loop location in the source code.
- \textbf{Pipelined}: Indicates whether the body of a loop is pipelined. Pipelining allows for many data items to be processed concurrently (in the same clock cycle) while making efficient use of the hardware in the datapath by keeping it occupied.
- \textbf{II}: Shows the sustainable initiation interval (II) of the loop. Processing data in loops is an additional source of pipeline parallelism. When you pipeline a loop, the next iteration of the loop begins before previous iterations complete.

You can determine the number of clock cycles between iterations by the number of clock cycles you require to resolve any dependencies between iterations. You can refer to this number as the initiation interval (II) of the loop.

The Intel HLS Compiler automatically identifies these dependencies and builds hardware to resolve these dependencies while minimizing the II.
Scheduled $f_{\text{MAX}}$: Shows an early estimate of the maximum clock frequency ($f_{\text{MAX}}$) at which the loop operates. This estimate does not account for the effects of placing and routing.

The $f_{\text{MAX}}$ is the maximum rate at which the outputs of registers are updated. The physical propagation delay of the signal between two consecutive registers limits the clock speed. This propagation delay is a function of the complexity of the Boolean logic in the path. The path with the most logic (and the highest delay) limits the speed of the entire circuit, and you can refer to this path as the critical path.

The $f_{\text{MAX}}$ is calculated as the inverse of the critical path delay. High $f_{\text{MAX}}$ is desirable because it correlates directly with high performance in the absence of other bottlenecks. The compiler attempts to optimize the component for different objectives for the scheduled $f_{\text{MAX}}$ depending on whether the $f_{\text{MAX}}$ target is set and whether the #pragma II is set for each of the loops. The $f_{\text{MAX}}$ target is a strong suggestion and the compiler does not error out if it is not able to achieve this $f_{\text{MAX}}$, whereas the #pragma II triggers an error if the compiler cannot achieve the requested II. The $f_{\text{MAX}}$ achieved for each block of code is shown in the Loops report.

The following table outlines the behavior of the scheduler in the Intel HLS Compiler:

<table>
<thead>
<tr>
<th>Explicitly Specify $f_{\text{MAX}}$?</th>
<th>Explicitly Specify II?</th>
<th>Compiler Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Use heuristic to achieve best $f_{\text{MAX}}$/II trade-off.</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Best effort to achieve the II for the corresponding loop (may not achieve the best possible $f_{\text{MAX}}$).</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Best effort to achieve $f_{\text{MAX}}$ specified (may not achieve the best possible II).</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Best effort to achieve the $f_{\text{MAX}}$ specified at the given II. The compiler errors out if it cannot achieve the requested II.</td>
</tr>
</tbody>
</table>

Note: If you are using an $f_{\text{MAX}}$ target in the command line or for a , use #pragma II = &lt;N&gt; for performance-critical loops in your design.

- **Latency**: Shows the number of clock cycles a loop takes to complete one or more instructions. Typically, you want to have low latency. However, lowering latency often results in decreased $f_{\text{MAX}}$.

- **Speculated Iterations**: Shows the loop speculation. Loop speculation is an optimization technique that enables more efficient loop pipelining by allowing future iterations to be initiated before determining whether the loop was exited already.

For more information, refer to Loop Iteration Speculation (speculated_iterations Pragma) in the Intel HLS Compiler Reference Manual.

- **Max Interleaving Iterations**: Indicates the number of interleaved invocations of an inner loop that can be executed simultaneously.

For more information, refer to Loop Interleaving Control (max_interleaving Pragma) in the Intel HLS Compiler Reference Manual.
A.2.2. Reviewing Your Component Verification Results

For each component that the testbench calls, the verification statistics report provides information such as the number and type of invocations, latency, initiation interval, and throughput.

The verification statistics report becomes available after you simulate your component.

Important:
- The data presented in the verification statistics report might be dependent on the input values to the component from the test bench.
- The verification statistics report only reports the component loop initiation interval (II) values and throughput for enqueued invocations.

The following example verification statistics report is for a component `dut` that has been run once as a simple function call and 100 times as an enqueued invocation:

```
<table>
<thead>
<tr>
<th>Component Invocation (Location)</th>
<th>Invocations</th>
<th>Latency (ns avg)</th>
<th># Invocations</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit component invocation</td>
<td>1</td>
<td>4.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enqueued component invocation</td>
<td>100</td>
<td>4.4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

For components that use explicit streams, such as `ihc::stream_in<>` or `ihc::stream_out<>`, the verification statistics report also provides the throughput for each individual stream, as shown in the details pane:

```
<table>
<thead>
<tr>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit component invocation: Location: (ns avg)</td>
</tr>
<tr>
<td>Number of valid words read on the interface: 1</td>
</tr>
<tr>
<td>Number of valid words sent on the interface: 1</td>
</tr>
<tr>
<td>Throughput (bits/s): 1</td>
</tr>
</tbody>
</table>
```

View the simulation waveform by following the instructions in Debugging during Verification on page 16.

A.3. Reviewing Component Area Usage

The Area Analysis of System report (Area Analysis ➤ Area Analysis of System) in the High-Level Design Reports (report.html) provides a detailed breakdown of the estimated FPGA area usage. It also provides information about key hardware features such as private memory configuration.

Note: The Area Analysis of Source report is deprecated. You can find the same information in this report in the Area Analysis of System report.

The estimated area usage information correlates with, but does not necessarily match, the resource usage results from the Intel Quartus Prime Pro Edition software. Use the estimated area usage to identify parts of the design with large area overhead. You can also use the estimates to compare area usage between different designs. Do not use the estimated area usage information for final resource utilization planning.
The System entry in the Area Analysis of System report refers to all the components in the design. Expanding the System entry allows you to view all the components in the design.

A.4. Reviewing Component Architecture

The High Level Design Report (report.html) contains reports that show different views into the structure, interfaces, datapaths, and computation flows in your component.

The Intel HLS Compiler Pro Edition provides the following reports about your component design:

- **Graph Viewer** (beta)
  The Graph Viewer is an interactive view of your system as a hierarchy of block diagrams of how the compiler constructed your component. These diagrams can help you understand how data flows through your component. Clicking on different parts of a diagram shows you information such as the sizes and types of loads and stores, stalls, and latencies. The information is presented at various levels of granularity: system, function (component and task), block, and cluster.

- **Function Memory Viewer**
  The Function Memory Viewer report shows the data connections across the memory system of your component.

- **Schedule Viewer** (alpha)
  The Schedule Viewer report displays a Gantt-chart-like format that shows when each instruction is active relative to the other instructions.

A.4.1. Graph Viewer (Beta)

The Graph Viewer is an interactive view of your system as a hierarchy of block diagrams of how the compiler constructed your component. These diagrams can help you understand how data flows through your component.

Access system-, function-, block-, and cluster-level views of your component by selecting **System Viewers ➤ Graph Viewer** in the High Level Design Reports menu.

The Graph Viewer represents your component and tasks as a system of functions. Each function is divided into a set of blocks. Inside each block is a set of non-branching instructions that covers your code and the compiler loop orchestration optimization. The connections between blocks show the execution flow of your component.

There is an initialization block, called the runOnce block. When a function contains loops, cycles of blocks form, depending on the loop structure. Loops often impose initiation interval (II) bottlenecks and are a main optimization focus when you optimize your component.

A component block has three main parts:

- An input or loop input node
- A set of instructions
- A branch node
The input node and the branch node might not be present depending on if there is branching in or out of the block. The input or loop input node determines the initial value for variables depending on where the branch into this block is from. The rest of the block should ideally be filled with non-stallable instructions and have a minimal amount of stallable instructions, like I/O instructions or memory access instructions.

To save the amount of control handshaking infrastructure needed when synthesizing your design, the Intel HLS Compiler Pro Edition groups instructions within a block into groups called clusters.

The Intel HLS Compiler Pro Edition can create two types of clusters:

- **Stall-enable clusters**
  A stall-enable cluster contains stallable instructions. A stall-enable cluster has a minimum capacity of 1, meaning that in the worst case scenario, a stall-enable cluster can accept 1 thread under stall.

- **Stall-free clusters**
  A stall-free cluster contains nonstallable instructions. The cluster has a FIFO to store data that needs to be passed to logic outside of the cluster. The FIFO information is shown in the cluster exit node when you examine the cluster in the Cluster View of the Graph Viewer.

A branch node indicates the next block to go to and the condition required to go to that block.

The Graph Viewer is divided into two main panes: Graph List and Graph View. Clicking on different entries in the Graph List shows you different views of your component.
The different views available are as follows:

- **System View**
  The System View of the Graph Viewer provides a high-level view of components and tasks in your system.

- **Function View**
  The Function views of the Graph Viewer show the blocks inside your component and any tasks. A function appears in this view if it is declared as component or launched as a task function. Other called functions are automatically inlined. Click a function name in the Graph List of the Graph Viewer to see the function view.

- **Block View**
  The Block views of the Graph Viewer show stallable operations in your component and task functions, and the clusters and their connections.

- **Cluster View**
  The Cluster views of the Graph Viewer shows the contents of clusters in your components.

Hovering over elements in the Graph View pane typically displays a tooltip with details about the element and highlights connections to and from the element.

Clicking on an element displays details about the element in the Details pane below Graph List and Graph Viewer panes and highlights connections to and from the element.
You can control the zoom of the diagram in the Graph Viewer pane using your mouse scroll wheel or equivalent touchpad gesture.

**A.4.1.1. Reviewing System Information**

Use the System View in the Graph Viewer report to view of the various components in your system as well as the tasks. Additionally, this view displays the connectivity between a component and its tasks.

Click the system in the Graph List pane to see the system in the Graph View pane.

The System View of the Graph Viewer shows `ihc::launch` and `ihc::collect` calls that are the synchronization points from component/task to the task function. They are represented as write nodes (labeled WR) and read nodes (labeled RD) that connect to the `call-task_name` and `return-task_name` nodes of the corresponding task.

This report also shows explicit streaming interfaces between tasks with the name of the stream displayed as a rectangular node in the connection between the tasks connected by the streaming interface.
A.4.1.2. Reviewing Function Information

The Function Views in the Graph Viewer show the block connections that depend on the loop structures in your design and highlights the block that is an II bottleneck in red. They also show the interfaces and all stream read/write operation and memory load/store operation points of the component or task and highlight those that are stallable in red.

Click a function in the Graph List pane to see the function in the Graph View pane.
A. Reviewing the High-Level Design Reports (report.html)

Use the Function View for the following tasks:

- **Reviewing Component Interfaces** on page 33
- **Reviewing Loops and Blocks in Your Component or Task** on page 45

When viewing a function, the Graph View pane shows connections between nodes in a graph:

- **Control**
  - Control connections are connections between blocks and loops.
- **Memory**
  - Memory connections are connections between global or local memories.
- **Streams**
  - Stream connections are connections to and from read or write streams

By default, all connections are displayed. If you find your view is too cluttered, you can hide connections by clearing the checkbox for the type of connection you want to hide.

### A.4.1.2.1. Reviewing Component Interfaces

The Function View of the Graph Viewer shows a visual representation of the interfaces in your component.
Some parameters in your component can be marked as being stable. A parameter can be marked as stable if its argument does not change while your component executes, but the argument might change between component executions. In the Function View, a stable argument does not have any edge connection.

The Function View displays the different interfaces as outlined in the following sections:

- Default (Conduit) Interfaces on page 34
- Avalon® MM Master Interfaces on page 36
- Avalon MM Slave Register Interfaces on page 38
- Avalon MM Slave Memory Interfaces on page 41
- Avalon Streaming Interfaces on page 42

**Default (Conduit) Interfaces**

Conduit interfaces are implemented for any function parameter whose arguments are passed by value. The Function View of the Graph Viewer report connects the conduit interface to the corresponding stream read (RD) node. The read is synchronize with the start/busy signals on the component invocation interface.

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

struct coordinate_t { 
    int x; 
    int y; 
};

component int default_comp(int b, coordinate_t p) { 
    return b + p.x; 
}
```
For each default interface argument node, you can view details about the node when you hover over the node:

- **b Info**
  - Stable: No
  - Width: 32 bits
- **p Info**
  - Stable: No
  - Width: 64 bits
- **k Info**
  - Stable: Yes
  - Width: 32 bits
Avalon® MM Master Interfaces

Pointer arguments, pass-by-reference arguments, `ihc::mm_master<>` argument, and global variables all correspond to addresses to memory outside of your component. They result in at least one Avalon® MM Master interface in your component. Similarly to conduit interface arguments, these nodes connect to the corresponding stream read (RD) node for your component.

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

component int master_comp(
    int *pointer_d,
    ihc::mm_master<int, ihc::aspace<3>, ihc::awidth<4>, ihc::dwidth<32>,ihc::latency<1>, ihc::align<4> &master_i,
    int &result
) { 
    result = *pointer_d + *master_i;
    return result;
}
```

A. Reviewing the High-Level Design Reports (report.html)
The Function View of the Graph Viewer shows the following details for these interface arguments:

Stable  
Describes whether the interface argument is stable. That is, whether the hls_stable_argument attribute was applied.

Data width  
The width of the memory-mapped data bus in bits.

Address width  
The width of the memory-mapped address bus in bits.

Latency  
The guaranteed latency from when the read command exits the component to when the external memory returns valid read data. The latency is measured in clock cycles.

Maximum burst  
The maximum number of data transfers that can associate with a read or write transaction. For fixed latency interfaces, this value is set to 1.

Alignment  
The byte alignment of the base pointer address. The Intel HLS Compiler uses this information to determine the amount of coalescing that is possible for loads and stores to this pointer.
The Function View of the Graph Viewer shows the following details for Avalon MM Master interfaces:

<table>
<thead>
<tr>
<th>Memory address space number</th>
<th>The memory address space number for Avalon MM Master interface.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of banks</td>
<td>The number of memory banks contained in the memory.</td>
</tr>
<tr>
<td>Argument Name:</td>
<td>The names of arguments that access the Avalon MM Master interface.</td>
</tr>
</tbody>
</table>

Avalon MM Slave Register Interfaces

When you label a function parameter as an Avalon MM slave register (hls_avalon_slave_register_argument), then the interface argument is implemented in the control and status register (CSR) slave interface. The Function View of the Graph Viewer puts the slave register arguments inside a CSR container.

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

component int slavereg_comp{
    int hls_avalon_slave_register_argument slave_scalar_f,
    int* hls_avalon_slave_register_argument slave_pointer_g
```
The resulting memory map is described in the automatically generated header file `<component_name>_csr.h`. This header file is available in the menu in the source code pane. Clicking on the CSR container node in the Function View of the Graph Viewer also opens up the header file:
If you use the `hls_avalon_slave_component` macro, then the call and return signals from the component invocation interface are implemented in the control-and-status register (CSR) interface:

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

hls_avalon_slave_component
component int slavereg_comp(
    int hls_avalon_slave_register_argument slave_scalar_f,
    int* hls_avalon_slave_register_argument slave_pointer_g
) {  
    return slave_scalar_f + *slave_pointer_g;
}
```
Avalon MM Slave Memory Interfaces

When you declare a pointer argument as a slave memory, the Function View of the Graph Viewer shows the slave memory interface with a `<slave memory name>` LD/ST node that is connected to the Local Memory node in the component.

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

hls_avalon_slave_component component int slavemem_comp(    
    hls_avalon_slave_memory_argument(4096) int* slave_mem_h,    
    int index,    
    int hls_avalon_slave_register_argument slave_scalar_f
) {    
    return slave_mem_h[index] * slave_scalar_f;
}
```
If you look at the same Avalon MM slave memory interface in the Component Memory Viewer report, the same *slave memory name* LD/ST node is shown to be connected to an external RW port.

---

**Avalon Streaming Interfaces**

A streaming interface is shown in the Function View of the Graph Viewer by a *stream name* node connected to the corresponding RD node (for `stream_in<`) or WR node (for `stream_out<`).

```c
#include "HLS/hls.h"
#include "stdio.h"
component int stream_comp{
  ihc::stream_in<int> &stream_in_c,
  ihc::stream_out<int> &stream_out_e,
  int scalar_b
} {
  stream_out_e.write(scalar_b + 1);
  return stream_in_c.read() + scalar_b * 2;
}
```
The Function View of the Graph Viewer shows the following details for streaming interface arguments:
<table>
<thead>
<tr>
<th>Width</th>
<th>The width of the data signal in bits.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth</td>
<td>The depth of the stream in words</td>
</tr>
<tr>
<td></td>
<td>The word size of the stream is the size of the stream datatype.</td>
</tr>
<tr>
<td>Bits per symbol</td>
<td>Describes how the data is broken into symbols on the data bus.</td>
</tr>
<tr>
<td>Uses Packets</td>
<td>Indicates whether the interface exposes the startofpacket and endofpacket sideband signals on the stream interfaces. The signals can be access by the packet-based reads and writes.</td>
</tr>
<tr>
<td>Uses Valid (stream_in)</td>
<td>Indicates whether a valid signal is present on the stream interface. When Yes, the upstream source must provide valid data on every cycle that ready is asserted.</td>
</tr>
<tr>
<td>Uses Ready (stream_in)</td>
<td>Indicates whether a ready signal is present on the stream interface. When Yes, the downstream sink must be able to accept data on every cycle that valid is asserted.</td>
</tr>
</tbody>
</table>
A.4.1.2.2. Reviewing Loops and Blocks in Your Component or Task

The Function Views of the Graph Viewer in the High Level Design Report (report.html) shows an abstracted netlist of your component design. In the Function Views, you can visualize loops in your component and your component interactions with its internal RAM block and external interfaces.

Consider the following code excerpt from the transpose_and_fold component (part of the tutorial files provided in <quartus_installdir>/hls/examples/tutorials/loop_memory_dependency):

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

#define SIZE 32

typedef altera::stream_in<int> my_operand;
typedef altera::stream_out<int> my_result;

void transpose_and_fold(my_operand &a, my_operand &b, my_result &c)
{
  int i;
  int j;
  int a_buf[SIZE][SIZE];
  int b_buf[SIZE][SIZE];
  for (i = 0; i < SIZE * SIZE; i++) {
    a_buf[i / SIZE][i % SIZE] = a.read();
    b_buf[i / SIZE][i % SIZE] = b.read();
  }
  #ifdef USE_IVDEP
  #pragma ivdep
  #endif
  for (j = 0; j < SIZE * SIZE * SIZE; j++) {
    #pragma unroll
    for (i = 0; i < SIZE; i++) {
      b_buf[j % SIZE][i] += a_buf[i][j % SIZE];
    }
    for (i = 0; i < SIZE * SIZE; i++) {
      c.write(b_buf[i / SIZE][i % SIZE]);
    }
  }
}
```

The figure below shows that Block3 is highlighted in red to prompt you to review the loop in the Loop Analysis Report. The report shows that Block3 is a pipelined loop with an II value of 2. The loop pipeline with this II value might affect the throughput of your design.

The Loop Analysis Report shows that the II value is caused by a memory dependency on loads to the b_buf variable.

Confirm the memory dependency by looking at the memory arbitration in the Function Memory Viewer.
By hovering your mouse pointer over a node, you can view the tooltip and details that provide more information on the LSU. In the figure below, the tooltip shows information like the latency and that the LSU is stall-free.

For stallable nodes, latency values provided are estimates.
When viewing a function, the Graph View pane shows connections between nodes in a graph:

- **Control**
  Control connections are connections between blocks and loops.

- **Memory**
  Memory connections are connections between local memories, slave memories, or Avalon MM Master interfaces.

- **Streams**
  Stream connections are connections to and from read or write streams

By default, all connections are displayed. If you find your view is too cluttered, you can hide connections by clearing the checkbox for the type of connection you want to hide.
A.4.1.3. Reviewing Block Information

Use the Block Views in the Graph Viewer report to inspect the datapath of your design. The report shows the datapath within and between input nodes, clusters, and instructions that cannot be grouped into clusters.

If your design has loops, the compiler encapsulates the loop control logics into loop orchestration nodes (labeled as Loop Orch) and the initial conditions of the loops to loop input nodes.
Click a block in the Graph List pane to see the block in the Graph View pane.

![Graph List (Beta)](image)

Within a block, the report shows connections between instruction nodes and cluster nodes.

Click the different nodes and look at the Details panel to see the information about the node.

For instruction nodes, you can find the type of instruction with specific details. For example, on a stream RD or WR node, you can see the width, depth, name, scheduling info, stall-free attributes of a stream in the Details panel. For stallable nodes, the latency value provided is an estimate.

For cluster nodes, you can find the type of the cluster and other cluster attributes.
A.4.1.4. Reviewing Cluster Information

Use the Cluster Views in the Graph Viewer report to examine the data path of computations within a cluster. The Intel HLS Compiler Pro Edition groups instructions into clusters to reduce the amount of handshaking logic required when synthesizing your component.

A cluster starts with an entry node and ends with an exit node. These nodes perform handshaking with logic outside of the cluster.
For a stall-free cluster, the exit node of the cluster has a FIFO with a depth greater than or equal to the latency of the cluster. This FIFO stores any in-flight data that needs to be passed outside of the cluster. To see the size of the cluster exit FIFO, click the exit node and see the information in the Details pane.

If your design contains loops, you see loop orchestration nodes (labeled as **Loop Orch**) and variable nodes as well as computation nodes. The loop orchestration nodes and variable nodes are shown along with their Feedback nodes.

The compiler generates the loop orchestration nodes to pipeline your loop to increase the performance.

A variable node corresponds to a variable that has a loop-carried dependency in your design. A variable node goes through various computation logic and finally feeds to a feedback node that connects back to the variable node. This connection shows that the new value of the variable is passed to the next iteration.

Look for loop-carried variables that have a long latency to the feedback nodes as they can be the initiation interval (II) bottlenecks. See the Loop Analysis report to reflect the II bottleneck.

The feedback node has a FIFO to store in-flight values of the variable in different iterations of the loop and has a size equal to the dependency distance multiplied by the II. The dependency distance is the number of iterations between successive load/store operations that depend on each other.

In a Cluster View, you can find the size of the cluster exit FIFO by clicking the exit node and looking at the Details pane. You can also find the size of the cluster exit FIFO in a Block View by clicking the exit node and looking at the Details pane.

Click a cluster in the Graph List pane to see that cluster in the Graph View pane.
Click nodes in the Cluster View pane to see details about that node as well as highlight the connections for the node inputs and outputs. For stallable nodes, the latency value provided in the details is an estimate. For more accurate latency values, run simulation on your component.
A.4.2. Function Memory Viewer

Data movement is often a bottleneck in many algorithms. The Function Memory Viewer in the High Level Design Reports (report.html) shows you the memory system that the Intel HLS Compiler Pro Edition generated for your component. Use the Function Memory Viewer to help you identify data movement bottlenecks in your component design.

Some patterns in memory accesses can cause undesired arbitration in the load-store units (LSUs), which can affect the throughput performance of your component. Use the Function Memory Viewer to find where you might have unwanted arbitration in the LSUs.
Access the Function Memory Viewer by clicking **System Viewers ➤ Function Memory Viewer**.

The following figure shows the layout of the Function Memory Viewer:

1. **Function Memory List** pane
   - The **Function Memory List** pane lists all of the memories in your design. Click a memory name to see its graphical representation in the **Function Memory Viewer** pane.

2. **Function Memory Viewer** pane
   - The **Function Memory Viewer** pane shows a graphical representation of the memory system or memory bank selected in the **Function Memory List** pane.

3. Code view pane
   - The code view pane shows the source code files for which the reports are generated.

4. **Details** pane
   - The **Details** pane shows the details of the memory system or memory bank selected in the **Function Memory List** pane.

---

**Function Memory List**

The Memory List pane shows you a hierarchy of component and task functions, with memories that are synthesized (RAMs, ROMs, and registers) and are optimized away in that component or task.
### Icon or Label | Name | Description
---|---|---
1 | ![Component or task name icon] | Component or task name
   | | The list of memories in your component or task can be expanded or collapsed. Memories that do not belong to any component or task are shown under *(Other)*.

2 | ![RAM icon] | RAM
   | | A RAM is a memory that has at least one write to it. The name of the RAM memory is the same as its name in your design. Clicking the memory name displays a logical representation of the RAM in the Function Memory Viewer pane. By default, only the first bank of the memory system is displayed. To select which banks to display, expand the memory name. Clear the memory name check box to collapse all memory banks in the view. Select the memory name check box to show all memory banks in the view.

3 | ![ROM icon] | ROM
   | | A ROM is a memory that is only read from. The name of the ROM memory is the same as its name in your design. Clicking the memory name displays a logical representation of the ROM in the Function Memory Viewer pane. By default, only the first bank of the memory system is displayed. To select which banks to display, expand the memory name. Clear the memory name check box to collapse all memory banks in the view. Select the memory name check box to show all memory banks in the view.

---

*continued...*
<table>
<thead>
<tr>
<th>Icon or Label</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Bank #num</td>
<td>A memory bank is always associated with a RAM or a ROM. Each bank is named as Bank #num, where #num is the ID of the memory bank starting from 0. Clicking the bank name shows the bank view in the Function Memory Viewer pane: a graphical representation of the bank, with all of its replicates and private copies. This view can help you to focus on specific memory banks when you view a complex memory design. Clear the memory bank name check box to collapse the bank in the logical representation of the memory. Select the memory bank name check box to show the bank in the logical representation of the memory.</td>
</tr>
<tr>
<td>5</td>
<td>Register</td>
<td>A register is a component variable that is carried through the pipeline in registers rather than being stored in a RAM or ROM. The name of the register is the same as its name in your design. A register variable can be implemented either exclusively in flip-flops (FFs), or in a mix of FFs and RAM-based FIFOs.</td>
</tr>
<tr>
<td>6</td>
<td>text label</td>
<td>Optimized Away</td>
</tr>
<tr>
<td>7</td>
<td>Filter</td>
<td>Use the Function Memory List filter to selectively view the list of RAMs, ROMs, registers and optimized away variables in your design. Clearing the check box associated with a item in the filter hides all occurrences of that kind of item in the Function Memory List. Filter your Function Memory List to help you focus on a specific type of memory in your design.</td>
</tr>
</tbody>
</table>

**Function Memory Viewer**

The Function Memory Viewer pane shows you connections between loads and stores to specific logical ports on the banks in a memory system. It also shows you the number of replicates and private copies created per bank for your memory system. The following types of nodes might be shown in the Function Memory Viewer pane, depending on the component memory system and what you have selected in the Function Memory List pane:

- **Memory node**: The memory system for a given variable in your design.
- **Bank node**: A bank in the memory system. A memory system contains at least one memory bank. A memory bank can have one or more port nodes. Only banks selected in the Function Memory List pane are shown.
- **Replication node**: A replication node shows memory bank replicates that are created to efficiently support multiple accesses to a local memory. A bank contains at least one replicate. You can view replicate nodes only when you view a memory bank by clicking its name in the Function Memory List pane.
- **Private-copy node**: A private-copy node shows private copies within a replicate that are created to allow concurrent execution of multiple loop iterations. A replicate contains at least one private copy. You can view private-copy nodes only when you view a memory bank by clicking its name in the Function Memory List pane.
- **Port node**: Each read or write access to a local memory is mapped to a port. There are three types of port:
  - **R**: A read-only port
  - **W**: A write-only port
  - **RW**: A read and write port
• LSU node: A store (ST) or load (LD) node connected to the memory through port nodes.
• Arbitration node: An arbitration (ARB) node shows that LSUs compete for access to a shared port node, which can lead to stalls.
• Port-sharing node: A port-sharing node (SHARE) shows that LSUs have mutually exclusive access to a shared port node, so the load-store units are free from stalls.

Hover over any node to view the attributes of that node.

Hover over an LSU node to highlight the path from the LSU node to all of the ports that the LSU connects to.

Hover over a port node to highlight the path from the port node to all of the LSUs that read or write to the port node.

Click a node to select it and have the node attributes displayed in the Details pane.

The following figures show examples of what you can see in the Function Memory Viewer:

Logical representation of a memory in the Function Memory Viewer pane
Bank view of a memory bank in the Function Memory Viewer pane

**Code View**

The code view pane shows your source code. Clicking on a memory or a bank in the Function Memory Viewer pane highlights the line of your code (in the code view pane) where you declared the memory.

**Details**

The Details pane shows the attributes of the node selected in the Function Memory Viewer pane. For example, when you select a memory in a component, the Details pane shows information such as the width and depths of the memory banks, the memory layout information, the address bit mapping and any user-defined HLS attributes that you specified in your source code.

*Important:* The bit information in the **Address bit information** section of the Details pane is based on byte address and not element addresses. This difference means that bits you might specify in a memory attribute might be shown as different bits in the **Address bit information** section.

For an example of this difference, see *Example: Specifying Bank-Selection Bits for Local Memory Addresses* in the *Intel HLS Compiler Pro Edition Best Practices Guide*.

The content of the Details pane persists until you select a different node in the Component/Function Memory Viewer pane.


A.4.3. Schedule Viewer (Beta)

Use the Schedule Viewer to identify latency bottlenecks in your design. The Schedule Viewer shows the estimated start and ending clock cycle for functions, blocks, clusters, and individual instructions in your design.

Click an item in the Schedule List to focus the Schedule View at the selected level. You cannot click the system level of the component hierarchy in the Schedule List.

Click the schedule bar for an item in the Schedule View updates the Details pane with the information available about that item. The information can include a description of the item, start cycle, and latency.

Click an item in the Schedule View to show a popup menu for the item. For instructions, the popup menu includes a link that takes you to your C++ code that resulted in that instruction.

A.5. Accessing HLD FPGA Reports in JSON Format

The high level design report data for the Intel HLS Compiler Pro Edition is also available as JSON-formatted data.
The JSON files containing the data are available in the `<result>.prj/reports/lib/json` directory. The directory provides the following `.json` files:

### Table 7. JSON Files in the `<result>.prj/reports/lib/json` Directory

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>area.json</td>
<td>Area Analysis of System</td>
</tr>
<tr>
<td>area_src.json</td>
<td>Area Analysis of Source</td>
</tr>
<tr>
<td>block.json</td>
<td>Block View of Graph Viewer</td>
</tr>
<tr>
<td>info.json</td>
<td>Summary</td>
</tr>
<tr>
<td>loops.json</td>
<td>Loop Analysis</td>
</tr>
<tr>
<td>mav.json</td>
<td>Function View of Graph Viewer</td>
</tr>
<tr>
<td>new_lmv.json</td>
<td>Function Memory Viewer</td>
</tr>
<tr>
<td>pipeline.json</td>
<td>Cluster View of Graph Viewer</td>
</tr>
<tr>
<td>quartus.json</td>
<td>Summary</td>
</tr>
<tr>
<td>summary.json</td>
<td>Summary</td>
</tr>
<tr>
<td>system.json</td>
<td>System View of Graph Viewer</td>
</tr>
<tr>
<td>warnings.json</td>
<td>Summary</td>
</tr>
</tbody>
</table>

You can read the following `.json` files without a special parser:

- area.json
- area_src.json
- loops.json
- quartus.json
- summary.json

For example, if you want to identify all of the values and bottlenecks for the initiation interval (II) of a loop, you can find the information in the `children` section in the `loops.json` file, as shown below:

```json
"name":"<block name|Component: component name>  # Find the loops which does not begin with "Component:"
"data":[<Yes|No>, <#|n/a>, <II|n/a>]      # The data field corresponds to "Pipelined", "II", "Bottleneck"
```
B. Intel HLS Compiler Pro Edition Restrictions

When creating your IP using the HLS compiler, be aware of the current set of software and programming restrictions.

**C++ Language Restrictions**

The Intel HLS Compiler accepts C++ code.

- A component cannot include virtual functions, function pointers, or bit fields.
- Function-scoped static variables that are a part of the component cannot use function arguments for initialization.
- A component or task function cannot contain an irreducible loop. That is, loops in component and task functions must have only one entry point into the loop.

**C++ restrictions**

- The HLS compiler does not support using lambda functions as components.

**Class membership**

- HLS component functions cannot be a C++ class member. However, you can declare your component function as a wrapper function. This wrapper function can call a member function of a class or a part of a namespace.

**Exception handling**

- A component cannot contain exception handling.

**Library calls**

- The HLS compiler does not currently support calls to C++ runtime libraries on Windows, including calls from the testbench code.

**Library functions**

- A component cannot contain standard C or C++ library functions, unless they are explicitly supported by header files provided with the Intel HLS Compiler.

**Multiple inheritance**

- The HLS compiler does not support classes with multiple inheritance used as parameters. You may use classes as parameters provided that each class inherits from, at most, one class directly.

**Namespaces**

- HLS component functions cannot be a part of a declared namespace. However, you can declare your component function as a global wrapper function. This wrapper function can call a member function of a class or a part of a namespace.
Parameters

- The HLS compiler does not support classes with multiple inheritance used as parameters. You may use classes as parameters if each class inherits from, at most, one class directly.

Recursion

- The HLS compiler does not support the synthesis of components that use recursion; however, tail recursion is supported.

If a component has an algorithm that uses recursion, and it is identified for FPGA acceleration, modify the algorithm to use tail recursion, if possible.
# C. Intel HLS Compiler Pro Edition User Guide Archives

<table>
<thead>
<tr>
<th>Intel HLS Compiler Version</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1</td>
<td>Intel HLS Compiler Pro Edition User Guide</td>
</tr>
<tr>
<td>19.3</td>
<td>Intel HLS Compiler User Guide</td>
</tr>
<tr>
<td>19.2</td>
<td>Intel HLS Compiler User Guide</td>
</tr>
<tr>
<td>19.1</td>
<td>Intel HLS Compiler User Guide</td>
</tr>
<tr>
<td>18.1.1</td>
<td>Intel HLS Compiler User Guide</td>
</tr>
<tr>
<td>18.1</td>
<td>Intel HLS Compiler User Guide</td>
</tr>
<tr>
<td>18.0</td>
<td>Intel HLS Compiler User Guide</td>
</tr>
<tr>
<td>17.1.1</td>
<td>Intel HLS Compiler User Guide</td>
</tr>
<tr>
<td>17.1</td>
<td>Intel HLS Compiler User Guide</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2020.06.22</td>
<td>20.2</td>
<td>• Added information about compiler-generated short names to Integrating your IP into a System on page 20.</td>
</tr>
</tbody>
</table>
• Added a restriction on loops to Intel HLS Compiler Pro Edition Restrictions on page 61.  
• Removed Reviewing f_{MAX} II Information. This information is now part of Reviewing Loop Information on page 25. |
• Renamed Viewing Component Design to Reviewing Component Architecture on page 28.  
• Moved description of component and task representation in the High-Level Design reports from Reviewing Component Architecture on page 28 to Graph Viewer (Beta) on page 28. |

Document Revision History for Intel HLS Compiler User Guide

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2019.09.30       | 19.3                        | • **Added** Graph Viewer (Beta) on page 28.  
• Split information about viewing your component design into separate sections for Intel HLS Compiler Pro Edition and Intel HLS Compiler Standard Edition:  
  — **PRO** Reviewing Component Architecture on page 28  
  — **STD** Viewing Component Design  
• In Verifying the Functionality of Your Design on page 10, removed information about using MSVC to debug a design compiled with the i++ command. You cannot use MSVC to debug a design compiled with the i++ command. On Linux, you can use GDB to debug a design compiled with the i++ command. |
| 2019.07.01       | 19.2                        | • **PRO** Updated Reviewing the High-Level Design Reports (report.html) on page 22 to reflect the merging of various viewers into the Graph Viewer (beta). Some images in the section do not reflect the new reporting interface. The images will be updated in a future release of this document. |
| 2019.04.01       | 19.1                        | • **PRO** Updated Reviewing the High-Level Design Reports (report.html) on page 22 section as follows:  
  — The Function Viewer report is the new name for what previously called the Component Viewer report.  
  — Added Function Memory Viewer on page 53.  
  The Function Memory Viewer report replaces the Component Memory Viewer report.  
  — Added Reviewing System Information on page 31  
  — Added Reviewing Block Information on page 48  
  — Added Reviewing Cluster Information on page 50  
• Updated to Synthesize your Component IP with Intel Quartus Prime Pro Edition on page 19 to indicated that compiling your component with Intel Quartus Prime is not intended to close timing for your component.  
• **PRO** Revised the Intel HLS Compiler Pro Edition Restrictions on page 61 as follows:  
  — Revised C++ 14 restriction  
  — Removed Overloading/Templates limitation  
• **PRO** Added Reviewing fMAX II Information. |
| 2019.01.03       | 18.1.1                      | • Fixed typos in table headings in Intel HLS Compiler Pro Edition Compiler-Defined Preprocessor Macros on page 9.  
• Corrected typos in High-Throughput Simulation (Asynchronous Component Calls) Using Enqueue Function Calls on page 16 and Comparison of Explicit and Enqueued Function Calls on page 17:  
  — ihs_hls_component_run_all is now ihc_hls_component_run_all.  
  — ihs_hls_enqueue is now ihc_hls_enqueue. |

*continued...*
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2018.09.24       | 18.1                        | • **PRO** The Intel HLS Compiler has a new front end. For a summary of the changes introduced by this new front end, see *Improved Intel HLS Compiler Front End* in the *Intel High Level Synthesis Compiler Version 18.1 Release Notes*.
  • In *Debugging during Verification* on page 16, added a reminder to run the executable compiled with the `-ghdl` option before viewing the waveform in ModelSim.
  • **PRO** Added information to *Overview of the Intel High Level Synthesis (HLS) Compiler Pro Edition* on page 5 topic to indicate that the Intel HLS Compiler treats all input file as C++14-compliant code. While you can compile code compliant with other standards by using the `-std` compile option, not all Intel HLS Compiler features are supported for other C++ standards. |
| 2018.07.02       | 18.0                        | • Added information about viewing the high level design report data in JSON files. See *Accessing HLD FPGA Reports in JSON Format* on page 59 for details.
  • Added related links to *Verifying Your IP with Simulation* on page 15 for Mentor Graphics ModelSim prerequisites. |
| 2018.05.07       | 18.0                        | • Starting with Intel Quartus Prime Version 18.0, the features and devices supported by the Intel HLS Compiler depend on what edition of Intel Quartus Prime you have. Intel HLS Compiler publications now use icons to indicate content and features that apply only to a specific edition as follows:
  • **PRO** Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Pro Edition.
  • **STD** Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Standard Edition.
  • **STD** Added important prerequisite for Intel MAX® 10 users to *Synthesize your Component IP with Intel Quartus Prime Standard Edition*.
  • Revised *Debugging during Verification* on page 16 to clarify how to view the waveform in ModelSim after simulation. |
| 2017.12.22       | 17.1.1                      | • Corrected typos in *Execution Model* on page 17:
  • `ihs_hls_component_run_all` is now `ihs_hls_component_run_all`
  • `ihs_hls_component_run_all` is now `ihs_hls_component_run_all`.  |

*continued...*
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2017.11.06       | 17.1                        | • Moved the following content to *Intel High Level Synthesis Compiler Pro Edition Best Practices Guide*:
|                  |                             |   — Moved compiler best practice content from "Creating a High-Level Synthesis Component and Testbench on page 8" to "Best Practices for Coding and Compiling Your Component". |
|                  |                             |   • Moved the following content to *Intel High Level Synthesis Compiler Reference Manual*:
|                  |                             |   — Moved "High Level Synthesis Component Interface Definition" to Component Interface Definition. |
|                  |                             |   — Moved Reset Behavior section to "Reset Behavior". |
|                  |                             | Added new chapter "Optimizing and Refining Your Component" on page 11 to provide a brief introduction to the high-level design report (report.html). |
|                  |                             | Added new chapter "Verifying the Functionality of Your Design" on page 10 to provide some details about how to perform functional verification on your HLS component. |
|                  |                             | Rearranged the order of sections to better reflect the user flow of using the compiler. |
| 2017.06.23       |                             | • Minor changes and corrections. |
| 2017.06.09       |                             | • Updated *Intel HLS Compiler Pro Edition Restrictions* on page 61 to add, remove, and change compiler limitations found in this release. |
|                  |                             | • Rebranding __ALTERA_COMPILER__ and __ALTERA_TYPE__ to __INTELFPGA_COMPILER__ and __INTELFPGA_TYPE__ |
|                  |                             | • Changed references for the compiler option -march=fpga to -march="<FPGA_family_or_part_number>". For details about changes to the -march compiler option, see *Command Options that Customize Compilation* in the *Intel HLS Compiler Pro Edition Reference Manual*. |
|                  |                             | • Added recommendation to compile components with -Wconversion to *Creating a High-Level Synthesis Component and Testbench* on page 8. |
|                  |                             | • Added information about HLS component reset behavior in *Reset Behavior*. |

*continued...*
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2017.02.03       | —                           | • Added note about what functions have components synthesized for them when you run the i++ command.  
• Under Reviewing Your Component’s report.html File, added Component memory viewer section to introduce the Component memory viewer report.  
• Under Reviewing Your Component’s report.html File, updated examples and screen captures to reflect examples and tutorials provided with the Intel HLS Compiler.  
• Updated the values for the __ALTERA_COMPILER__ HLS compiler-defined preprocessor macro. |
| 2016.11.30       | —                           | • Under Reviewing Your Component’s report.html File, added the Information on Component Verification Results section to introduce the Verification Statistics report.  
• In Verifying Your HLS IP, noted that information on the supported versions of the ModelSim software is available in the Intel Quartus Prime Software and Device Support Release Notes.  
• Removed the Latency Measurement during Verification section because the APIs described within have been removed.  
• In Adding the Compiler-Generated IP into a Intel Quartus Prime Project and Adding the Compiler-Generated IP into a Qsys System, specified that for the Intel Quartus Prime Standard Edition software, the file in question is the .qsys file. For the Intel Quartus Prime Pro Edition software, the file in question is the .ip file.  
• Updated the Limitations of the HLS Compiler section:  
  — Removed the limitation on ModelSim software version support.  
  — Added the limitation that C++ library calls are not supported on Windows. |
| 2016.09.12       | —                           | • Initial release. |