Contents

1 Intel® High Level Synthesis Compiler User Guide.............................................................3

2 Overview of the Intel High Level Synthesis (HLS) Compiler.............................................4
   2.1 High Level Synthesis Design Flow............................................................................. 4
   2.2 The Project Directory.............................................................................................. 5

3 Creating a High-Level Synthesis Component and Testbench............................................ 7
   3.1 Compiler-Defined Preprocessor Macros...................................................................... 8

4 Verifying the Functionality of Your IP Design..................................................................9

5 Optimizing and Refining Your Component......................................................................10

6 Verifying Your IP with Simulation..................................................................................11
   6.1 Generation of the Verification Testbench Executable...................................................11
   6.2 Debugging during Verification.................................................................................12
   6.3 High-Throughput Simulation (Asynchronous Component Calls) Using Enqueue Function Calls...................................................................................................12
      6.3.1 Execution Model........................................................................................13
      6.3.2 Comparison of Explicit and Enqueued Function Calls...................................... 13

7 Synthesize your Component IP with Intel Quartus Prime..............................................15

8 Integrating your IP into a System..................................................................................16
   8.1 Adding the HLS Compiler-Generated IP into an Intel Quartus Prime Project.............16
   8.2 Adding the HLS Compiler-Generated IP into a Platform Designer System..................17

A Limitations of the Intel HLS Compiler............................................................................ 18

B Reviewing the High Level Design Report (report.html) ..............................................20
   B.1 High Level Design Report Layout...........................................................................20
   B.2 Reviewing the Report Summary..............................................................................22
   B.3 Reviewing Loop Information...................................................................................23
      B.3.1 Loop Analysis Example.............................................................................. 24
   B.4 Reviewing Your Component Area Usage...............................................................25
      B.4.1 Area Analysis Example............................................................................... 27
   B.5 Viewing Your Component Design...........................................................................29
      B.5.1 Reviewing Your Component Interfaces......................................................... 29
      B.5.2 Reviewing Memory Replication and Stallable LSU Information.......................39
   B.6 Viewing Your Component Memory System .............................................................42
   B.7 Reviewing Your Component Verification Results.......................................................44

C Document Revision History............................................................................................ 46
1 Intel® High Level Synthesis Compiler User Guide

The Intel® High Level Synthesis Compiler 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
2 Overview of the Intel High Level Synthesis (HLS) Compiler

The Intel High Level Synthesis (HLS) Compiler parses your design, compiles it to an object or RTL code, and links the objects.

The Intel HLS Compiler is command-line compatible with g++, and supports most of the g++ compiler flags. The Intel HLS Compiler 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++.

When targeting 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 HLS results, including the generated IP. It also contains reports and auxiliary information for verification purposes.

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

Running the executable file runs your testbench. When you compile your design to an FPGA architecture, the output executable runs a simulation. When you compile your design to the x86-64 architecture, the output executable runs your design on the CPU.

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 to different targets, depending on where you are in your IP development cycle.

The typical design flow when you use the Intel HLS Compiler 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 7.

2. Verify the function of your component algorithm and testbench by compiling your design to x86-64 executable.
   For details, see Verifying the Functionality of Your IP Design on page 9.

3. Optimize and refine the FPGA performance of your component.
   For details, see Optimizing and Refining Your Component on page 10.
After initial optimizations, you can see where to further refine your component by compiling it for simulation. For details, see Verifying Your IP with Simulation on page 11.

4. Synthesize your component with Intel Quartus® Prime.
   For details, see Synthesize your Component IP with Intel Quartus Prime on page 15.

   Synthesizing your component also generates accurate quality-of-results (QoR) metrics like area and performance estimates.

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 16.

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

Figure 1. Overview of Procedure for Synthesizing IP for Intel FPGA Products

Create component and test bench

Compile design with g++ or i++ -march=x86-64 for functional verification
(Note: You can debug your design using GDB, even for an i++ x86-64 output)

Compile design with the following command to generate IP and a test bench 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 the Quartus Prime software

2.2 The Project Directory

The project directory (<result>.prj) that the Intel HLS Compiler outputs has four main subdirectories.
Table 1. 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 reports with information that is useful for analyzing the hardware implementation of the synthesized components.</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.</td>
</tr>
</tbody>
</table>
3 Creating a High-Level Synthesis Component and Testbench

To create a component, write a complete C++ application where your `main()` function acts as a testbench and each function that you label as a component is synthesized to an HDL representation. The components are invoked from the testbench.

Write the functions for your components in the OpenCL™-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 synthesizes all the code in the function or functions that you label as components, and any code that these components call, to an HDL representation.

1. To identify a function in your C++ application that you want to synthesize into an IP core, perform either of the following tasks.
   - 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.

   **Important:** Be careful to avoid combining these methods because you might unexpectedly synthesize unwanted components for some of your functions. If you combine these methods, components are synthesized for all functions labeled with the `component` keyword as well as all components listed in the `--component <component_list>` option of the `i++` command.

   If you do not want components synthesized 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.

   Review the "Area Analysis by Source" section of the high-level design report (`<name>.prj/reports/report.html`) to see a breakdown of functions per component.

   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 Compiler-Defined Preprocessor Macros

The Intel High Level Synthesis (HLS) Compiler has two built-in macros available. With these macros, you can tailor your code to create flow-dependent code. The macros can be useful for specifying code fragments that only need to run in execution mode or in verification mode.

The compiler-defined preprocessor macros are __INTELFPGA_COMPILER__ and __INTELFPGA_TYPE__.

Table 2. Macro Definitions for __INTELFPGA_COMPILER__ and __INTELFPGA_TYPE__

<table>
<thead>
<tr>
<th>Tool Invocation</th>
<th><strong>INTELFPGA_COMPILER</strong></th>
<th><strong>INTELFPGA_TYPE</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>g++</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>i++ -march=x86-64</td>
<td>&quot;17.1&quot;</td>
<td>&quot;NONE&quot;</td>
</tr>
<tr>
<td>i++ -march=&quot;&lt;FPGA_family_or_part_number&gt;&quot;</td>
<td>&quot;17.1&quot;</td>
<td>&quot;VERILOG&quot;</td>
</tr>
</tbody>
</table>
4 Verifying the Functionality of Your IP Design

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

Compiling your design to an x86-64 executable is faster than having to compile 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 for functional verification through any of the following methods:

- Use the `i++ -march=x86-64` command to compile your component and testbench to an x86-64 executable.
- On Linux systems, use the `g++` command to compile your component and testbench to an x86-64 executable.
- On Windows systems, use Microsoft Visual Studio to compile your component and testbench to an x86-64 executable.

Ensure that you set your compiler command to include debug information. The `i++` command generates debug information by default. You can use GDB (on Linux operating systems) or Microsoft Visual Studio (on Windows operating systems) to debug your component and testbench, even if you used the `i++` command to compile your code for functional verification.
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 report to further optimize and refine your component design. The high-level design report shows 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 report, you can see how your code changes affect your component hardware implementation without needing to run a simulation or a full Quartus synthesis.

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

```
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, but a simulation flow compile might take slightly longer. However, compiling your component with a simulation flow gives you additional information in the high-level design report.

The Intel HLS Compiler High Level Design Report (report.html)

The high-level design report is an HTML file called `report.html` that you can open in a web browser to review. You can find the high-level design report in the `<name>.prj/reports` folder created when you compile your component to RTL.

Use the high-level design report to review information about your component, including the following information:

- Component latency
- Loop information, including unroll status
- Component visualization including load-store units, component interfaces, loops, and local memory systems

After you run a simulation flow, the report also shows you verification statistics such as component reset latency.

After you synthesize your component with Quartus compilation, the following additional information is available in the report:

- Maximum clock frequency
- Area usage

For more information about the high-level design report and how to use it to optimize and refine your component, see Reviewing the High Level Design Report (report.html) on page 20.
6 Verifying Your IP with Simulation

When compiling your component to an FPGA architecture, the Intel HLS Compiler links your design C++ testbench with an RTL-compiled version of your component that runs in an RTL simulator. Use the Mentor Graphics® ModelSim software to perform the simulation. 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.

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

```bash
i++ -march="Arria 10" --component <component_list> [...] design.cpp
```

Related Links
- EDA Interface Information - Quartus Prime Standard Edition Software
- EDA Interface Information - 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.

When you invoke the `i++ -march="<FPGA_family_or_part_number>" <input_files>` command, 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. Compile the code for component synthesis to the FPGA. This compilation generates Verilog for the component and a Verilog-DPI testbench.
6.2 Debugging during Verification

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

To enable signal logging in the simulator, include the `-ghdl` option in your `i++` command:

- Invoke the `i++` command as follows: To enable full debug visibility and log all signals, invoke the following command:
  
  ```
  i++ -march="<FPGA_family_or_part_number>" -ghdl <input files>
  ```

When simulation is completed, open the `vsim.wlf` file inside the `a.prj/verification` directory to view the waveform.

When you view the waveform in ModelSim, you can view the component top-level signals (start, busy, stall, done, parameters, and outputs) by right-clicking on the `<component_name>_inst` block and selecting Add Wave.

Tip: When you view the simulation waveform in ModelSim, the Time axis shown when is not actual time but a simulated time at 1 GHz, therefore one nanosecond (1 ns) corresponds to one cycle. To synchronize the Time axis to show one cycle per tick mark, change the time resolution from picoseconds (ps) to nanoseconds (ns).

6.3 High-Throughput Simulation (Asynchronous Component Calls)

Using Enqueue Function Calls

Explicitly 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 queuing inputs to the component before executing the component. Queue inputs to the component with explicit interfaces by using enqueue function calls from the cosimulation library. You can estimate the throughput of your component by dividing the your component initiation interval (II) by the $f_{\text{max}}$ of your component, which indicates approximately how many times your component is invoked per second.

<p>| Table 3. Functions from Cosimulation Library for Queuing Inputs to the Component with Explicit Interfaces |</p>
<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 <code>ihc_hls_component_run_all()</code> function is invoked.</td>
</tr>
<tr>
<td>ihc_hls_enqueue_noret(void* funcptr, ...)</td>
<td>This function is similar to <code>ihc_hls_enqueue(void* retptr, void* funcptr, ...)</code>, except that it does not have an output stream 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) will not be visible in the testbench until the `ihc_hls_componen_run_all()` function explicitly triggers the execution.

6.3.2 Comparison of Explicit and Enqueued Function Calls

The enqueue function allows 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 every after II number of cycles.

Figure 2 on page 13 illustrates the waveform of the signals for the component `dut`. As shown in the following code snippet, 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 3 on page 14 illustrates the waveform of the signals for the component `dut`. As shown in the following code snippet, the testbench includes enqueue function calls. Compare how the component is passed new data each clock cycle 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;
    ihs_hls_enqueue(&x1, &dut, 1, 2);
    ihs_hls_enqueue(&x2, &dut, 3, 4);
    ihs_hls_enqueue(&x3, &dut, 5, 6);
    ihs_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

When you are happy with the predicted performance of your component, you can then perform the longer hardware synthesis compilation with Intel Quartus Prime. This compilation also generates accurate area and performance estimates for your design.

After the Intel Quartus Prime compilation completes, the high level design report file shows 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.

Compilation Time Requirements:

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 --component ...

• 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
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 Standard Edition or Intel Quartus Prime Pro Edition as well as the Platform Designer (formerly Qsys/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 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.

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

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

- For Intel Quartus Prime Standard Edition, add the `.qsys` file to the project.
- For Intel Quartus Prime Pro Edition, add the `.ip` file to the project.

The `.qsys` file or 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 project.
2. Click **Project ➤ Add/Remove Files in Project**.
3. Perform one of the following tasks:
   - For the Intel Quartus Prime Standard Edition software, in the **Settings** dialog box, browse to and select the component’s `.qsys` file.
     
     For example, `<result>.prj/components/<component_name>/<component_name>.qsys`
   - For the Intel Quartus Prime Pro Edition software, in the **Settings** dialog box, browse to and select the component’s `.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 and 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. Perform one of the following tasks:
   - For Intel Quartus Prime Standard Edition, in the **IP Catalog Search Locations** dialog box, add the path to the directory that contains the .qsys file to IP Search Paths. To find all the components, specify the path as `<result>.prj/components/**/*`.
   - For Intel Quartus Prime Pro Edition, 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>.ip`.
4. In **IP Catalog**, add your IP to the Platform Designer system by selecting it from the HLS project directory.

For more information about Platform Designer, see one of the following references, depending on your version of Intel Quartus Prime:

- "Creating a System with Platform Designer (Standard)" in *Intel Quartus Prime Standard Edition Handbook Volume 1: Design and Compilation*
A Limitations of the Intel HLS Compiler

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

Compiler support

**Linux compiler support**
The HLS compiler does not support GCC 4.7.0 or newer. The compiler requires GCC compiler and C++ Libraries version 4.4.7.

**Windows compiler support**
The HLS compiler for Windows is compatible with Microsoft Visual Studio 2010 only.

**C++ Language Restrictions**

Program your source code in C++. Where possible, adhere to the OpenCL-supported subset of C99 for code that is intended for synthesis (that is, code inside component functions).

- 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.

**C++11 restrictions**

- The HLS compiler does not support certain C++11 features such as initializer lists and lambda functions.

**Class membership**

- HLS components cannot be a C++ class member or part of a declared namespace. If you must use a component this way, create a component that is not part of a class or namespace that actually calls the implementation, then call that component.

**Exception handling**

- A component cannot contain exception handling.

**Library calls**

- The HLS compiler does not currently call to C++ runtime libraries on Windows, including calls from the testbench code.
<table>
<thead>
<tr>
<th>Limitation</th>
</tr>
</thead>
</table>
| **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.  
A component that contains printf() or cout calls works in its x86 implementation. However, the generated RTL does not include the printf() or cout function calls if you include the HLS/stdio.h library or the HLS/iostream standard C library functions provided with the Intel HLS Compiler. If you try to generate RTL with the regular stdio.h or iostream headers you will likely experience compiler errors. |
| **Multiple inheritance** | The HLS compiler does not support classes with multiple inheritance used as parameters. You may use classes as parameters as long as each class inherits from, at most, one class directly. |
| **Namespaces** | HLS components cannot be a C++ class member or part of a declared namespace. If you must use a component this way, create a component that is not part of a class or namespace that actually calls the implementation, then call that component. |
| **Overloading/Templates** | Components cannot be templated functions or overloaded functions. If you must use a component this way, create a component that is not part of a templated function or overloaded function, then call that component. |
| **Parameters** | The HLS compiler does not support classes with multiple inheritance used as parameters. You may use classes as parameters as long as 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. |
B Reviewing the High Level Design Report (report.html)

After compiling your component, the Intel HLS Compiler generates an HTML report that helps you analyze various component aspects, such as area, loop structure, memory usage, and component pipeline. To launch the high level design report, open the following file in a web browser: `<result>.prj/reports/report.html`.

B.1 High Level Design Report Layout

The High Level Design Report (report.html) is divided into four main sections: report menu, analysis pane, source code pane, and details pane.

Report Menu

From the View reports pull-down menu, you can select a report to see an analysis of different parts of your component design.
Analysis Pane

The analysis pane displays detailed information of the report that you selected from the View reports pull-down menu.

Source Code Pane

The source code pane displays the code for all the source files in your component.

To select between different source files in your component, click the pull-down menu at the top of the source code pane. To collapse the source code pane, do one of the following actions:

- Click the X icon beside the source code pane pull-down menu.
- Click the vertical ellipsis icon on the right-hand side of the report menu and then select Show/Hide source code.

If you previously collapsed the source code pane and want to expand it, click the vertical ellipsis icon on the right-hand side of the report menu and then select Show/Hide source code.

Details Pane

For each line that appears in a loop analysis or area report, the details pane shows additional information, if available, that elaborates on the comment in the Details column report. To collapse the details pane, do one of the following actions:

- Click the X icon on the right-hand side of the details pane.
- Click the vertical ellipsis icon on the right-hand side of the report menu and then select Show/Hide details.
B.2 Reviewing the Report Summary

The report 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 report summary is divided into four sections: Info, Quartus Fit Summary, Estimated Resource Usage, and Compile Warnings.

### Info

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Name</td>
<td>a</td>
</tr>
<tr>
<td>Target Family Device</td>
<td>Arria 10, 10AX115U1F4615G</td>
</tr>
<tr>
<td>ACDS Version</td>
<td>17.1.0 Build 106</td>
</tr>
<tr>
<td>I++ Version</td>
<td>0.02 Build 17.1.0.112</td>
</tr>
<tr>
<td>Command</td>
<td>i++ --v --march=Arria10 --simulator name vector_add.cpp</td>
</tr>
<tr>
<td>Reports Generated At</td>
<td>Wed Jun 7 18:05:18 2017</td>
</tr>
</tbody>
</table>

### Quartus Fit Summary

Run Quartus compile to populate this section. See details for more information.

### Estimated Resource Usage

<table>
<thead>
<tr>
<th>Component Name</th>
<th>ALUTs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector_add</td>
<td>5384</td>
<td>5476</td>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>5384 (1%)</td>
<td>5476 (0%)</td>
<td>46 (2%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Available</td>
<td>854,600</td>
<td>170,980,000</td>
<td>2713</td>
<td>1510</td>
</tr>
</tbody>
</table>

### Compile Warnings

None

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

- Name of the project
- Target FPGA family and device
- Intel Quartus Prime version
- HLS compiler version
- The command that was used to compile the design
- The date and time at which the reports were generated
Quartus Fit Summary

The Quartus Fit Summary section of the report.html Summary page is populated after compiling your design with Intel Quartus Prime software. After compilation, the following sections appear on the Summary page:

• Quartus Fit Clock Summary
• Quartus Fit Resource Utilization Summary

The Quartus Fit Clock Summary section shows the maximum clock frequencies that can be achieved for the design.

The Quartus Fit 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.

Estimated Resource Usage

The Estimated Resource Usage 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.

Compile Warnings

The Compile Warnings section shows some of the compiler warnings generated during the compilation.

B.3 Reviewing Loop Information

The High Level Design Report (<result>.prj/reports/report.html) file contains information about all the loops in your design and their unroll statuses. This loop analysis report helps you examine whether the Intel HLS Compiler is able to maximize the throughput of your component.

You can use the loop analysis report to help determine where to deploy one or more of the following pragmas on your loops:

• #pragma unroll
  For details about #pragma unroll, see "Loop Unrolling (unrollPragma)" in Intel High Level Synthesis Compiler Reference Manual.

• #pragma loop_coalesce
  For details about #pragma loop_coalesce, see "Loop Coalescing (loop_coalescePragma)" in Intel High Level Synthesis Compiler Reference Manual.

• #pragma ii
  For details about #pragma ii, see "Loop Initiation Interval (iiPragma)" in Intel High Level Synthesis Compiler Reference Manual.

1. Click View reports ➤ Loop Analysis.
2. In the analysis pane, select Show fully unrolled loops to obtain information about the loops in your design.
3. Consult the flowchart below to identify actions you can take to improve the throughput of your design.
B.3.1 Loop Analysis Example

An example High Level Design Report (report.html) file that shows the loop analysis of a component design taken from the transpose_and_fold.cpp file (part of the tutorial files provided in <quartus_installdir>/hls/examples/tutorials/best_practices/loop_memory_dependency).

Consider the following example code snippet for transpose_and_fold.cpp:

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

#define SIZE 32

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

component void transpose_and_fold(my_operand &data_in, my_result &res)
{
  int i;
  int j;
}  
```
```c
14:   int in_buf[SIZE][SIZE];
15:   int tmp_buf[SIZE][SIZE];
16:   for (i = 0; i < SIZE * SIZE; i++) {
17:     in_buf[i / SIZE][i % SIZE] = data_in.read();
18:     tmp_buf[i / SIZE][i % SIZE] = 0;
19:   }
20:
21:   #ifdef USE_IVDEP
22:   #pragma ivdep safelen(SIZE)
23:   #endif
24:   for (j = 0; j < SIZE * SIZE * SIZE; j++) {
25:     #pragma unroll
26:     for (i = 0; i < SIZE; i++) {
27:       tmp_buf[j % SIZE][i] += in_buf[i][j % SIZE];
28:     }
29:   }
30:   for (i = 0; i < SIZE * SIZE; i++) {
31:     res.write(tmp_buf[i / SIZE][i % SIZE]);
32:   }
33: }
```

**Figure 4. Loop Analysis Report of the transpose_and_fold Component**

The `transpose_and_fold` component has four loops. The loop analysis report shows that the compiler performed different kinds of loop optimizations:

- The loop on line 24 is fully unrolled, as defined by `#pragma unroll`.
- The loops on lines 16 and 30 are pipelined with an II value of 1.

The `Block1.start` loop in the loop analysis report is not present in the code. It is an implicit infinite loop that the compiler adds to allow the component to run continuously, instead of only once.

### B.4 Reviewing Your Component Area Usage

The High Level Design Report ([report.html](report.html)) provides a detailed breakdown of the estimated FPGA area usage. It also provides feedback on key hardware features such as private memory configuration.

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. Do not use the estimated area usage information for final resource utilization planning.

The Quartus Fit Summary section of the High Level Design Report Summary page is populated after compiling your design with Intel Quartus Prime software. After that compilation, the following sections appear on the Summary page:

- Quartus Fit Clock Summary
- Quartus Fit Resource Utilization Summary
The Quartus Fit Clock Summary section shows the maximum clock frequencies that can be achieved for the design.

The Quartus Fit 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.

**Tip:**

Compiling your component using the Intel Quartus Prime software might take several hours. In contrast, the Intel HLS Compiler compiler can generate the High Level Design Report in minutes for most designs.

Before compiling your design with Intel Quartus Prime software, the High Level Design Report looks like the following example:

```
<table>
<thead>
<tr>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Info</td>
</tr>
<tr>
<td>Project Name</td>
</tr>
<tr>
<td>Target Family, Device</td>
</tr>
<tr>
<td>ACDS Version</td>
</tr>
<tr>
<td>I++ Version</td>
</tr>
<tr>
<td>Command</td>
</tr>
<tr>
<td>Reports Generated At</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quartus Fit Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Quartus compile to populate this section. See details for more information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Estimated Resource Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Name</td>
</tr>
<tr>
<td>vector_add</td>
</tr>
<tr>
<td>Total</td>
</tr>
<tr>
<td>Available</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compile Warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
</tbody>
</table>
```
After compiling your design with Intel Quartus Prime software, the High Level Design Report looks like the following example:

### Summary

<table>
<thead>
<tr>
<th>Info</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Name</td>
<td>qii</td>
</tr>
<tr>
<td>Target Family, Device</td>
<td>Arria 10, 10AX115U1F484I5G</td>
</tr>
<tr>
<td>ACDS Version</td>
<td>17.1.0 Internal Build 95</td>
</tr>
<tr>
<td>C++ Version</td>
<td>0.92 Build 17.1.0.93</td>
</tr>
<tr>
<td>Command</td>
<td>i++ -march=Arria 10 --simulator none mem.cpp -o qii -v --quartus-compile</td>
</tr>
<tr>
<td>Reports Generated At</td>
<td>Wed May 24 10:30:05 2017</td>
</tr>
</tbody>
</table>

#### Quartus Fit Clock Summary

<table>
<thead>
<tr>
<th></th>
<th>1x clock fmax</th>
<th>2x clock fmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>438.02</td>
<td>645.16</td>
</tr>
</tbody>
</table>

#### Quartus Fit Resource Utilization Summary

<table>
<thead>
<tr>
<th></th>
<th>ALMs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full design (all components)</td>
<td>241.5</td>
<td>523</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>simple_mem</td>
<td>101</td>
<td>167</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>simple_mem2x</td>
<td>140.5</td>
<td>236</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**B.4.1 Area Analysis Example**

You have the option to review the area analysis of your design based on source line or system.

**Area Analysis by Source**

Area analysis by source shows an approximation of how each line of the source code affects area. In the area analysis by source view, the report shows the area hierarchically.
The **System** entry in the area report refers to all the components in the design. Expanding the **System** entry allows you to view all the components in the design. In this example, there is only one component (that is, **transpose_and_fold**).

Each line in the report contains state and corresponding information. In the figure below, the example area report shows that on line 17, where a stream of data is stored to `in_buf`, the consumed area is used for computing the pointer value and then storing it. On line 14, area consumption is a result of `in_buf` using 16 RAM blocks and some logic.

**Figure 5. Breakdown of Area Usage by Source Line**

![Figure 5](image.png)

**Area Analysis by System**

Area analysis of system shows an area breakdown that is closest to the actual hardware implemented in the FPGA.
B.5 Viewing Your Component Design

The Component Viewer in the High Level Design Report (report.html) shows an abstracted netlist of your component design. You can visualize component interfaces, load-store units (LSUs), loops, and local memory systems.

B.5.1 Reviewing Your Component Interfaces

The Component Viewer report shows a visual representation of the interfaces in your component. You can view details about the following interface arguments: default, pointer, pass-by-reference, Avalon Memory-Mapped (MM), and Avalon Streaming.

Some interface arguments in your component can be marked as being stable. A stable interface argument is an argument that does not change while your component executes, but the argument might change between component executions.

In the Component Viewer report, a stable node does not have any edge connection.
The Component Viewer report displays the different interfaces as outlined in the following sections:

- Default Interface Arguments on page 30
- Pointer, Pass-By-Reference, and Avalon MM Master Interface Arguments on page 32
- Avalon MM Slave Register Interface Arguments on page 34
- Avalon MM Slave Memory Interface Arguments on page 36
- Avalon Streaming Interface Arguments on page 37

**Default Interface Arguments**

Default interface arguments are any scalars or simple structs. The Component Viewer report connects the default argument nodes to the corresponding channel read (RD) node.

```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

For reviewing the High Level Design Report (report.html)

UG-20037 | 2017.12.22
Pointer, Pass-By-Reference, and Avalon MM Master Interface Arguments

Pointer interfaces, pass-by-reference interfaces, Avalon MM master interfaces, and global variables all correspond to addresses to memory outside of your component. Similarly to the default interface arguments, these nodes connect to the corresponding channel 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;
}
```

The Component Viewer report shows the following details for these interface arguments:

- **Stable**: Describes whether the interface argument is stable.
- **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.
- **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.
Avalon MM Slave Register Interface Arguments

When you label an interface argument 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 Component Viewer report 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 
) 
{
    return slave_scalar_f + *slave_pointer_g;
}
```

Reviewing the High Level Design Report (report.html)

UG-20037 | 2017.12.22
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 editor. Clicking on the CSR container node in the Component Viewer report also opens up the header file:

```c
/* This header file describes the CSR Slave for the slave_reg_component */

#ifndef __SLAVE_REG_CSR_REGS_H__
#define __SLAVE_REG_CSR_REGS_H__

/* Memory Map Summary */

Register | Access | Register Contents (64-bits) | Description
---------|--------|-----------------------------|-------------
	| R/W | slave_scalar_f[31:0] | Argument slave_scalar_f
	| (reserved[31:8], | slave_scalar_f[31:8]) | |
	| (reserved[8] | slave_scalar_f[8]) | |

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

/* Register Address Macros */

#define SLAVE_REG_CSR_ARG_SLAVE_SCALAR_F_REG (0x0)
#define SLAVE_REG_CSR_ARG_SLAVE_POINTER_0_REG (0x8)

/* Argument Sizes (byte) */

#define SLAVE_REG_CSR_ARG_SLAVE_SCALAR_F_SIZE (4)
#define SLAVE_REG_CSR_ARG_SLAVE_POINTER_0_SIZE (8)

/* Argument Masks */

#define SLAVE_REG_CSR_ARG_SLAVE_SCALAR_F_MASK (0xffffffff)
#define SLAVE_REG_CSR_ARG_SLAVE_POINTER_0_MASK (0xffffffffffffffff)

/* Status/Control Masks */

#define SLAVE_REG_CSR_REGS_H__ */
```
If you use the `hls_avalon_slave_component` macro, then the “do” and “return” streams (control and status registers) are implemented in the CSR interface:

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

hls_avalon_slave_component
c
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 Interface Arguments**

When you declare a pointer argument as a slave memory, the Component Viewer report 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
c
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 Interface Arguments

A streaming interface is shown in the Component Viewer report 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 Component Viewer report shows the following details for streaming interface arguments:

**stream stream_in_c Info**
- Width: 32 bits
- Depth: 0
- Bits per symbol: 32 bits
- Uses Packets: No
- Uses Valid: Yes

**stream stream_out_e Info**
- Width: 32 bits
- Depth: 0
- Bits per symbol: 32 bits
- Uses Packets: No
- Uses Ready: Yes
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>The width of the data bus in bits.</td>
</tr>
<tr>
<td>Depth</td>
<td>The depth of the stream.</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</td>
<td>(stream_in) 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 Reader</td>
<td>(stream_in) 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>

### B.5.2 Reviewing Memory Replication and Stallable LSU Information

Consider the following code excerpt from the `transpose_and_fold` component (part of the tutorial files provided in `<QPDS_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 on line 23 is highlighted in red to prompt you to review the loop. Because loop analysis of Block3 shows that it is a pipelined loop with an II value of 2, the loop pipeline might affect the throughput of your design. The Component Viewer shows that the II value is caused by a memory dependency on loads to the b_buf variable.

**Figure 7. System View of the transpose_and_fold Component**

By hovering your mouse 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 of the load is 6, and the LSU is stall-free.
The Component Viewer allows you to select the type of connections you want to view. Selecting **Control** instructs the system viewer to display the connections between blocks and loops. Selecting **Memory** instructs the Component Viewer to display the connections to and from global and local memories. Selecting **Streams** instructs the system viewer to display the connections reading from and writing to streams.
B.6 Viewing Your Component Memory System

Data movement is often a bottleneck in many algorithms. The component memory viewer in the High Level Design Report (report.html) shows you how the Intel High Level Synthesis (HLS) Compiler interprets the data connections across the memory system of your component. Use the Component Memory Viewer to help you identify data movement bottlenecks in your component design.
Also, 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 Component Memory Viewer to find where you might have unwanted arbitration in the LSUs.

The Component Memory Viewer has the following panes:

Memory List  The Memory List pane shows you a hierarchy of components, memories in that component, and the corresponding memory banks.

Clicking a memory name in the list displays a graphical representation of the memory in the Component memory viewer pane. Also, the line in your code where you declared the memory is highlighted in the Source Code pane.

Clearing a check box for a memory bank collapses that bank in the Component Memory Viewer pane, which can help you to focus on specific memory banks when you view a complex memory design. By default, all banks in component memory are selected and shown in the Component Memory Viewer pane.
The Component Memory Viewer pane shows you connections between loads and stores to specific logical ports on the banks in a memory system. The following types of nodes might be shown in the Component Memory Viewer pane, depending on the component memory system:

- **Memory node**: The component memory.
- **Bank node**: A bank in the memory. Only banks selected in the Memory List pane are shown. Select banks in the Memory List pane to help you focus on specific on memory banks when you view a complex memory design.
- **Port node**: The logical port for a bank. 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.
- **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 store to the port node.

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

### Details

The Details pane shows the attributes of the node selected in the Component 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, as well as any user-defined HLS attributes that you specified in your source code.

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

### B.7 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.
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>Verification Statistics</th>
<th>Invocations</th>
<th>Latency (in usec)</th>
<th># Invocations</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit component invocations (Unmonitored)</td>
<td>1</td>
<td>4.44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enqueued component invocations (Unmonitored)</td>
<td>100</td>
<td>4.44</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 invocations</td>
</tr>
<tr>
<td>Unmonitored throughput (in usec)</td>
</tr>
<tr>
<td>4.44</td>
</tr>
</tbody>
</table>

Streams with data rates (in usec) |
| 4.44 |

- Measurement of the data rate starts on the first valid word seen on the interface and ends when the last valid word is transmitted on the interface. The data rate is then calculated as (location of words seen on the interface) / (location of cycle) in which the data word is received.
- A word is considered valid when the first non-zero value is seen on the interface.
# Document Revision History

## Table 4. Document Revision History of the Intel High Level Synthesis Compiler User Guide

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| December 2017 | 2017.12.22 | • Corrected typos in *Execution Model* on page 13:  
  — `ihs_hls_component_run_all` is now `ihc_hls_component_run_all`.  
  — `ihs_hls_run_all_enqueued` is now `ihc_hls_component_run_all`. |
| November 2017 | 2017.11.06 | • Moved the following content to *Intel High Level Synthesis Compiler Best Practices Guide*:  
  — Moved compiler best practice content from "Creating a High-Level Synthesis Component and Testbench on page 7" 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 10 to provide a brief introduction to the high-level design report (report.html).  
  • Added new chapter "Verifying the Functionality of Your IP Design" on page 9 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. |
| June 2017     | 2019.06.23 | • Minor changes and corrections. |
| June 2017     | 2017.06.09 | • Updated *Limitations of the Intel HLS Compiler* on page 18 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 Reference Manual*.  
  • Added recommendation to compile components with `-Wconversion` to *Creating a High-Level Synthesis Component and Testbench* on page 7.  
  • Added information about HLS component reset behavior in *Reset Behavior*. |

*continued...*
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>February 2017</td>
<td>2017.02.03</td>
<td>• Added note about what functions have components synthesized for them when you run the <code>i++</code> command.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Under Reviewing Your Component's report.html File, added Component memory viewer section to introduce the Component memory viewer report.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Under Reviewing Your Component's report.html File, updated examples and screen captures to reflect examples and tutorials provided with the Intel HLS Compiler.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated the values for the <code>__ALTERA_COMPILER__</code> HLS compiler-defined preprocessor macro.</td>
</tr>
<tr>
<td>November 2016</td>
<td>2016.11.30</td>
<td>• Under Reviewing Your Component's report.html File, added the Information on Component Verification Results section to introduce the Verification Statistics report.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed the Latency Measurement during Verification section because the APIs described within have been removed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• In Adding the Compiler-Generated IP into a Intel Quartus Prime Project and Adding the Compiler-Generated IP into a Qsys System, specified that the for the Intel Quartus Prime Standard Edition software, the file in question is the <code>.qsys</code> file. For the Intel Quartus Prime Pro Edition software, the file in question is the <code>.ip</code> file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated the Limitations of the HLS Compiler section:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— Removed the limitation on ModelSim software version support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— Added the limitation that C++ library calls are not supported on Windows.</td>
</tr>
<tr>
<td>September 2016</td>
<td>2016.09.12</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>