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

The features and devices supported by the Intel HLS Compiler depend on what edition of Intel Quartus® Prime you have. The following icons indicate content in this publication that applies only to the Intel HLS Compiler provided with a certain edition of Intel Quartus Prime:

PRO Indicates that a feature or content applies only to Intel HLS Compiler Pro Edition.

STD Indicates that a feature or content applies only to Intel HLS Compiler Standard Edition.

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 and your Intel Quartus Prime edition:

**PRO**

Windows: `C:\intelFPGA_pro\19.3`

Linux: `/home/<username>/intelFPGA_pro/19.3`

**STD**

Windows: `C:\intelFPGA_standard\19.3`

Linux: `/home/<username>/intelFPGA_standard/19.3`
2. Overview of the Intel High Level Synthesis (HLS) Compiler

The Intel High Level Synthesis (HLS) Compiler parses your design, compiles it to an x86-64 object or FPGA-targeted RTL code, and creates an executable testbench.

The Intel HLS Compiler 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 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++14. The compiler does not support files conforming to newer C++ standards.
- The Intel HLS Compiler Standard Edition treats all input files as C++98. 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 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 target the compilation to an x86-64 architecture, the output executable runs your design on the CPU. 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 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 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
   compiling it for simulation. For details, see Verifying Your IP with Simulation on
   page 12.

4. Synthesize your component with Intel Quartus Prime.
   For details, see Synthesize your Component IP with Intel Quartus Prime on page
   16.
   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 17.

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 testbench executable to verify your design in simulation:

i++ -march="<FPGA_family_or_part_number>"

Optimize FPGA Performance

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

2.2. The Project Directory

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

The Intel HLS Compiler 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++14 (Pro Edition) or C++98 (Standard Edition), 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 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 an IP core 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:

```c
extern "C" int myComponent(int a, int b)
```
Important: Components are synthesized for all functions labeled with the `component` keyword and all 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 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.

PRO You can see which components were synthesized in the summary page of the high-level design report (`<name>.prj/reports/report.html`). For more information about the high-level design report, see *The Intel HLS Compiler High Level Design Report (report.html)* on page 11.

STD You can see which components were synthesized in the **Area Analysis by Source** section of the high-level design report (`<name>.prj/reports/report.html`). For more information about the high-level design report, see *The Intel HLS Compiler High Level Design Report (report.html)* 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. Compiler-Defined Preprocessor Macros

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

**Table 2. Macro Definition for __INTELFPGA_COMPILER__**

<table>
<thead>
<tr>
<th>Tool Invocation</th>
<th><strong>INTELFPGA_COMPILER</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>g++ or cl</td>
<td>Undefined</td>
</tr>
<tr>
<td>i++ -march=x86-64</td>
<td>1930</td>
</tr>
<tr>
<td>i++ -march=&quot;&lt;FPGA_family_or_part_number&gt;&quot;</td>
<td>1930</td>
</tr>
</tbody>
</table>

**Table 3. 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>
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.

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

You can automate the process by using makefile or batch script. Use the makefiles and scripts provided in the Intel HLS Compiler 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 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 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 prevents you from co-simulating your design and measure its latency and generate waveforms.

**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 view in a web browser. 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:

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

After you run a simulation flow, the area utilization estimates also show you verification statistics such as component latency and occupancy of your component I/O interfaces.

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

- Maximum clock frequency
- Accurate area usage estimate

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)](report.html) on page 25.


*Other names and brands may be claimed as the property of others.
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.

The Intel HLS Compiler 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 queuing 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 cosimulation library. Estimate the throughput of your component by dividing the component f_{MAX} by the component initiation interval (II), which indicates approximately how many times your component is invoked per second.
Table 4. Functions from Cosimulation 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 15 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 15 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

Figure 4 on page 16 illustrates the waveform of the signals for the component dut when the testbench includes dequeue function calls. Observe how the component is passed new data each clock cycle, and compare this waveform with the earlier waveform.
7. Synthesize your Component IP with Intel Quartus Prime

When you are satisfied 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 ($f_{\text{MAX}}$) estimates for your design, however your design is not expected to cleanly close timing in the Intel Quartus Prime reports.

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

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.

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

- 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:

  ```sh
  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 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.

**Important prerequisite for Intel® Max® 10 FPGA users:** If you develop your component IP for Intel MAX® 10 devices and you want to integrate your component IP into a system that you are developing in Intel Quartus Prime, ensure that the Intel Quartus Prime settings file (.qsf) for your system contains one of the following lines:

- `set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE IMAGE WITH ERAM"`
- `set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE COMP IMAGE WITH ERAM"`

When you compile the component IP for an Intel MAX 10 devices with Intel HLS Compiler, the generated Intel Quartus Prime example project contains all of the required QSF settings for your component. However, the Intel Quartus Prime project for the system into which you integrate your component might not have the required QSF setting.

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>`.

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*


<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019.09.30</td>
<td>19.3</td>
<td>• <strong>PRO</strong> Added Graph Viewer (Beta) on page 41.&lt;br&gt;• Split information about viewing your component design into separate sections for Intel HLS Compiler Pro Edition and Intel HLS Compiler Standard Edition:&lt;br&gt;  — <strong>PRO</strong> Viewing Component Design on page 39&lt;br&gt;  — <strong>STD</strong> Viewing Component Design on page 70&lt;br&gt;• In Verifying the Functionality of Your Design on page 10, removed information about using MSVC to debug a design compiled with the <code>i++</code> command. You cannot use MSVC to debug a design compiled with the <code>i++</code> command. On Linux, you can use GDB to debug a design compiled with the <code>i++</code> command.</td>
</tr>
<tr>
<td>2019.07.01</td>
<td>19.2</td>
<td>• <strong>PRO</strong> Updated Reviewing the High Level Design Report (report.html) on page 25 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.</td>
</tr>
<tr>
<td>2019.04.01</td>
<td>19.1</td>
<td>• <strong>PRO</strong> Updated Reviewing the High Level Design Report (report.html) on page 25 section as follows:&lt;br&gt;  — The Function Viewer report is the new name for what previously called the Component Viewer report.&lt;br&gt;  — <strong>PRO</strong> Added Function Memory Viewer on page 64.&lt;br&gt;  — The Function Memory Viewer report replaces the Component Memory Viewer report.&lt;br&gt;  — <strong>PRO</strong> Added Reviewing System Information on page 42&lt;br&gt;  — <strong>PRO</strong> Added Reviewing Block Information on page 61&lt;br&gt;  — <strong>PRO</strong> Added Reviewing Cluster Information on page 62&lt;br&gt;• Updated to Synthesize your Component IP with Intel Quartus Prime on page 16 to indicated that compiling your component with Intel Quartus Prime is not intended to close timing for your component.&lt;br&gt;• <strong>PRO</strong> Revised the Limitations of the Intel HLS Compiler on page 23 as follows:&lt;br&gt;  — Revised C++ 14 restriction&lt;br&gt;  — Removed Overloading/Templates limitation&lt;br&gt;• <strong>PRO</strong> Added Reviewing fmax II Information on page 33</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2019.01.03       | 18.1.1                      | • Fixed typos in table headings in Compiler-Defined Preprocessor Macros on page 9.  
• Corrected typos in High-Throughput Simulation (Asynchronous Component Calls) Using Enqueue Function Calls on page 13 and Comparison of Explicit and Enqueued Function Calls on page 14:  
  — ihs_hls_component_run_all is now ihc_hls_component_run_all.  
  — ihs_hls_enqueue is now ihc_hls_enqueue. |
| 2018.09.24       | 18.1                        | • 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 13, added a reminder to run the executable compiled with the -ghdl option before viewing the waveform in ModelSim.  
• Added information to Overview of the Intel High Level Synthesis (HLS) Compiler 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 87 for details.  
• Added related links to Verifying Your IP with Simulation on page 12 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.  
• Added important prerequisite for Intel MAX 10 users to Synthesize your Component IP with Intel Quartus Prime on page 16.  
• Revised Debugging during Verification on page 13 to clarify how to view the waveform in ModelSim after simulation. |
| 2017.12.22       | 17.1.1                      | • Corrected typos in Execution Model on page 14:  
  — ihs_hls_component_run_all is now ihc_hls_component_run_all.  
  — ihs_hls_run_all_enqueued is now ihc_hls_component_run_all.  
continued... |
## 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>
| 2017.11.06       | 17.1                       | • 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 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 *Limitations of the Intel HLS Compiler on page 23* 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 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 the 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. |
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**

- **STD** The HLS compiler does not support GCC 4.7.0 or newer. The compiler requires GCC compiler and C++ Libraries version 4.4.7.
- **STD** The HLS compiler does not support GCC 5.5.0 or newer. The compiler requires GCC compiler and C++ Libraries version 5.4.0.

**Windows compiler support**

- **STD** The HLS compiler for Windows is compatible with Microsoft Visual Studio 2010 only.
- **STD** The HLS compiler for Windows is compatible with Microsoft Visual Studio 2017 only.

C++ Language Restrictions

The Intel HLS Compiler accepts C++ code.

- **STD** For the best results when you synthesize your component, code your component function with C99.
  - 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.
- **STD** C++ restrictions
  - The HLS compiler does not support certain C++ features such as initializer lists and lambda functions.
  - **STD** The HLS compiler does not support using lambda functions as components.
- **STD** 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.
- **STD** Exception handling
  - A component cannot contain exception handling.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Library calls</strong></td>
<td>The HLS compiler does not currently support calls to C++ runtime libraries on Windows, including calls from the testbench code.</td>
</tr>
<tr>
<td><strong>Library functions</strong></td>
<td>A component cannot contain standard C or C++ library functions, unless they are explicitly supported by header files provided with the Intel HLS Compiler.</td>
</tr>
<tr>
<td></td>
<td>STD A component that contains <code>printf()</code> or <code>cout</code> calls works in its x86 implementation. However, the generated RTL does not include the <code>printf()</code> or <code>cout</code> 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.</td>
</tr>
<tr>
<td><strong>Multiple inheritance</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Namespaces</strong></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>STD 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.</td>
</tr>
<tr>
<td><strong>Overloading/Templates</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
| **Parameters**       | 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 the following four main sections:

- Reports menu
- Analysis pane
- Source code pane
- Details pane
Reports Menu

You can select a report to view an analysis of different parts of your component design. All reports are interlinked. 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.
Analysis Pane

The analysis pane displays detailed information of the report you selected from the reports 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 Reports 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 Reports menu and then select Show/Hide source code.

The source code is displayed when you have not specified the \( -g0 \) compiler command option when you compiled your 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 Reports 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 the following sections: Info, Synthesized Function Name Summary, Quartus Fit Summaries, Function Summary, Estimated Resource Usage, and Compile Warnings.
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
Synthesized Function Name Mapping

For overloaded and templated functions in your hardware design, this section of the report summary 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.

Quartus Fit Summaries

After you compile your design with Intel Quartus Prime software, the sections that summarize the compilation results are populated. 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 frequency 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.

Function Summary

When you compile your design to target 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.

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 the compiler warnings generated during the compilation.

B.3. Reviewing Factors That Affect Throughput
B.3.1. Reviewing Loop Information

The Throughput Analysis section of the high-level design reports (\texttt{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:

- \texttt{#pragma unroll}
  For details about \texttt{#pragma unroll}, see "Loop Unrolling (unrollPragma)" in Intel High Level Synthesis Compiler Reference Manual.

- \texttt{#pragma loop_coalesce}
  For details about \texttt{#pragma loop_coalesce}, see "Loop Coalescing (loop_coalescePragma)" in Intel High Level Synthesis Compiler Reference Manual.

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

- \texttt{#pragma speculated_iterations}
  For details about \texttt{#pragma speculated_iterations}, see "Loop Iteration Speculation (speculated_iterationsPragma)" in Intel High Level Synthesis Compiler Reference Manual.

- \texttt{#pragma max_concurrency}
  For details about \texttt{#pragma max_concurrency}, see "Loop Concurrency (max_concurrencyPragma)" in Intel High Level Synthesis Compiler Reference Manual.

- \texttt{#pragma max_interleaving}
  For details about \texttt{#pragma max_interleaving}, see "Loop Interleaving Control (max_interleavingPragma) in Intel High Level Synthesis Compiler Reference Manual.

1. Click Throughput Analysis ➤ Loop Analysis.
2. In the analysis pane, select Show fully unrolled loops to obtain information about the loops in your design.

B.3.1.1. Loop Analysis Example

Figure 4 on page 32 shows an example High Level Design Report (report.html) file that shows the loop analysis of a component design taken from the \texttt{transpose_and_fold.cpp} file (part of the tutorial files provided in \texttt{<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;
  int in_buf[SIZE][SIZE];
  int tmp_buf[SIZE][SIZE];
  for (i = 0; i < SIZE * SIZE; i++) {
    in_buf[i / SIZE][i % SIZE] = data_in.read();
  }

#pragma ivdep safelen(SIZE)
  for (j = 0; j < SIZE * SIZE * SIZE; j++) {
#pragma unroll
    for (i = 0; i < SIZE; i++) {
      tmp_buf[j % SIZE][i] += in_buf[i][j % SIZE];
    }
  }
  for (i = 0; i < SIZE * SIZE; i++) {
    res.write(tmp_buf[i / SIZE][i % SIZE]);
  }
}
```

![Figure 4. Loop Analysis Report of the transpose_and_fold Component](image)

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 26 is fully unrolled, as defined by `#pragma unroll`.
- The loops on lines 16 and 30 are pipelined with an II value of ~1. The value is ~1 because both loops contain access to streams that could stall. If these access stall, then the loop II becomes greater than 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. In hardware, the component runs continuously and checks its inputs to see if it should start executing.

### B.3.2. Reviewing f\(_{\text{max}}\) II Information

The f\(_{\text{max}}\) II report provides key performance metrics on all blocks including any target II, scheduled f\(_{\text{max}}\), block II, and maximum interleaving iterations.

Use the f\(_{\text{max}}\) II 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

Access the f\(_{\text{max}}\) II report, by selecting **Throughput Analysis → f\(_{\text{max}}\) II Report** on the high-level design reports page (report.html).

For example, if you have the following component in a file called fmax_ii.cpp, your f\(_{\text{max}}\) II Report would appear as shown in the later image.

```c
component int lowered_fmax (int N) {
  int res = N;
  #pragma unroll 9
  for (int i = 0; i < N; i++) {
    res += 1;
    res ^= i;
  }
  return res;
}
```
The report shows that the design $f_{\text{MAX}}$ is limited by the loop feedback path of block B2, and the information in the Details panel shows the loop feedback path that is limiting $f_{\text{MAX}}$. In this case, block B2 is your likely performance bottleneck and a good candidate to focus on in your component optimization process.

**B.4. Reviewing Component Area Usage**

The High Level Design Report (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.

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 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><strong>Info</strong></td>
</tr>
<tr>
<td>Project Name</td>
</tr>
<tr>
<td>Target Family, Device</td>
</tr>
<tr>
<td>I+ Version</td>
</tr>
<tr>
<td>Quartus Version</td>
</tr>
<tr>
<td>Command</td>
</tr>
<tr>
<td>Reports Generated At</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synthesized Function Name Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-defined Function Name</td>
</tr>
<tr>
<td>hw_fft256(hcstrl_stream_in, chcstrl_stream_out)</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>Function Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Name</td>
</tr>
<tr>
<td>hw_fft256</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Estimated Resource Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Name</td>
</tr>
<tr>
<td>hw_fft256</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>Compile warnings not specified above (for example, global trimmings, initialization)</td>
</tr>
</tbody>
</table>

After compiling your design with Intel Quartus Prime software, the High Level Design Report looks like the following example. The Quartus Fit Summary section is now populated.
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 of System**

Area analysis of system shows an area breakdown that is closest to the actual hardware implemented in the FPGA.
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. In this example, there is only one component (that is, **transpose_and_fold**).

**Figure 5. Breakdown of Area Usage by System**
Area Analysis by Source (Deprecated)

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 6. Breakdown of Area Usage by Source Line**

B.5. Viewing Component Design

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 reports represent 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 groups instructions within a block into groups called clusters.

The Intel HLS Compiler 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 views of your components are split into different reports. The reports available to you depend on your edition of the Intel HLS Compiler.

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 that allows you to review information about your system 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.

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

• **Component Viewer**
  The Component Viewer report shows branching in your component.

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

### B.5.1. **PRO** Graph Viewer (Beta)

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. 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 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 view of the Graph Viewer shows branching in your component. Click the function name in the **Graph List** of the Graph Viewer to see the function view.

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

- **Cluster View**
  The cluster view 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 the 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.

**B.5.1.1. Reviewing System Information**

Use the system view of 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.
B.5.1.2. Reviewing Function Information

The function view of the Graph Viewer shows the block connections that depend on the loop structures in your design and highlights the block that is an II bottleneck in red. It also shows 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.
B. Reviewing the High Level Design Report (report.html)

Use the function view for the following tasks:

- Reviewing Component Interfaces on page 71
- Reviewing Loops and Blocks in Your Component or Task on page 56

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.

### B.5.1.2.1. Reviewing Component Interfaces

The function view of the Graph Viewer shows a visual representation of the interfaces in your component. You can view details about the interface arguments such as 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 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 Interface Arguments** on page 46
- **Pointer, Pass-By-Reference, and Avalon MM Master Interface Arguments** on page 48
- **Avalon MM Slave Register Interface Arguments** on page 50
- **Avalon MM Slave Memory Interface Arguments** on page 52
- **Avalon Streaming Interface Arguments** on page 54

### Default Interface Arguments

Default interface arguments are any scalars or simple structs. The function view of the Graph 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
**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.

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

class 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 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 global memories:

- **Memory address space number**: The memory address space number for global memory.
- **Number of banks**: The number of memory banks contained in the memory.
- **Argument Name**: The names of arguments that access the global memory.

### 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 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” streams (control and status registers) are implemented in the 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 Interface Arguments**

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"
```
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 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:

**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
Width
The width of the data bus in bits.

Depth
The depth of the stream in words
The word size of the stream is the size of the stream datatype.

Bits per symbol
Describes how the data is broken into symbols on the data bus.

Uses Packets
Indicates whether the interface exposes the `startofpacket` and `endofpacket` sideband signals on the stream interfaces. The signals can be accessed by the packet-based reads and writes.

Uses Valid
`(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.

Uses Reader
`(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.

B.5.1.2.2. Reviewing Loops and Blocks in Your Component or Task

The function view of the Graph Viewer in the High Level Design Report (report.html) shows an abstracted netlist of your kernel design. In the function view, you can visualize load-store units (LSUs) between a component and different memories, streams, and loops.
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
01 #include "HLS/hls.h"
02 #include "stdio.h"
03 #include "stdlib.h"
04
05 #define SIZE 32
06
07 typedef altera::stream_in<int> my_operand;
08 typedef altera::stream_out<int> my_result;
09
10 void transpose_and_fold(my_operand &a, my_operand &b, my_result &c)
11  {
12    int i;
13    int j;
14    int a_buf[SIZE][SIZE];
15    int b_buf[SIZE][SIZE];
16    for (i = 0; i < SIZE * SIZE; i++) {
17      a_buf[i / SIZE][i % SIZE] = a.read();
18      b_buf[i / SIZE][i % SIZE] = b.read();
19    }
20 #ifdef USE_IVDEP
21      #pragma ivdep
22 #endif
23    for (j = 0; j < SIZE * SIZE * SIZE; j++) {
24      #pragma unroll
25      for (i = 0; i < SIZE; i++) {
26        b_buf[j % SIZE][i] += a_buf[i][j % SIZE];
27      }
28    }
29    for (i = 0; i < SIZE * SIZE; i++) {
30      c.write(b_buf[i / SIZE][i % SIZE]);
31    }
32 }
```

The figure below shows that Block3 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 function view of the Graph Viewer shows that the II value is caused by a memory dependency on loads to the `b_buf` variable.
Figure 7. Function View of the transpose_and_fold Component

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. For more accurate latency values, run simulation on your component and obtain latency values from the simulation.
The function view of the Graph Viewer allows you to select the type of connections you want to view. Selecting **Control** instructs the function view to display the connections between blocks and loops. Selecting **Memory** instructs the function view of the Graph Viewer to display the connections to and from global and local memories. Selecting **Streams** instructs the function view to display the connections reading from and writing to streams.
B. Reviewing the High Level Design Report (report.html)
B.5.1.3. **Reviewing Block Information**

Use the block view of 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.

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 more accurate latency values, run simulation on your component.

For cluster nodes, you can find the type of the cluster and other cluster attributes.
B.5.1.4. Reviewing Cluster Information

Use the cluster viewer of the Graph Viewer report to examine the data path of computations within a cluster. The Intel HLS Compiler 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 Orch nodes and variable nodes are shown along with their Feedback nodes.

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

A variable node corresponds to a variable that has 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 means the variable is passed to the next iteration after the new value is evaluated.

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.

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.
B.5.2. Function Memory Viewer

Data movement is often a bottleneck in many algorithms. The Function Memory Viewer in the High Level Design Report (report.html) shows you how the Intel HLS Compiler Pro Edition interprets the data connections and synthesizes memory 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.
<table>
<thead>
<tr>
<th>Icon or Label</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Component or task name" /></td>
<td><strong>Component or task name</strong></td>
<td>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 <strong>(Other)</strong>.</td>
</tr>
<tr>
<td><img src="image" alt="RAM" /></td>
<td><strong>RAM</strong></td>
<td>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.</td>
</tr>
<tr>
<td><img src="image" alt="ROM" /></td>
<td><strong>ROM</strong></td>
<td>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.</td>
</tr>
<tr>
<td><img src="image" alt="Bank #num" /></td>
<td><strong>Bank #num</strong></td>
<td>A memory bank is always associated with a RAM or a ROM. Each bank is named as <strong>Bank #num</strong>, where #num is the ID of the memory bank starting from 0.</td>
</tr>
</tbody>
</table>

---

**Component name(s)**

1. **RAM**
2. **ROM**
3. **Bank (for RAMs and ROMs)**
4. **Register**
5. **Optimized away**
6. **Filter**
### Icon or Label Name Description

<table>
<thead>
<tr>
<th>Icon or Label</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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 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>
<td></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 text label</td>
<td>Optimized Away</td>
<td>A component variable might be optimized away because it is unused in your design or compiler optimizations have transformed all uses of the variable such that it is unnecessary. The name of the optimized away variable is the same as its name in your design.</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 connect to one 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 simultaneous 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
**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.

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

**B.5.3. Schedule Viewer (Alpha)**

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 on an item in the Schedule List to focus the Schedule View at the selected level. You cannot on 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, its start cycle, and its 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.
B.6. Viewing Component Design

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 reports represent your component and tasks as a system of functions. Each function is represented by 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 component 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 groups instructions within a block into groups called clusters. A cluster has an entry and exit node to handle the control handshaking. There is no further control handshaking within a cluster.

The Intel HLS Compiler can create two types of clusters:

• Stall-free clusters
  A stall-free cluster contains nonstallable instructions, and 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 Viewer.

• Stall-enable clusters
  A stall-enable cluster contains stallable instructions.

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

The views of your components are split into different reports. The reports available to you depend on your edition of the Intel HLS Compiler.

The Intel HLS Compiler Standard Edition provides the following reports about your component design:
B.6.1. Reviewing Component Interfaces

The Component Viewer 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, a stable node does not have any edge connection.

The Component Viewer displays the different interfaces as outlined in the following sections:

- Default Interface Arguments on page 71
- Pointer, Pass-By-Reference, and Avalon MM Master Interface Arguments on page 73
- Avalon MM Slave Register Interface Arguments on page 75
- Avalon MM Slave Memory Interface Arguments on page 77
- Avalon Streaming Interface Arguments on page 79

Default Interface Arguments

Default interface arguments are any scalars or simple structs. The Component Viewer 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:

```plaintext
b Info
Stable: No
Width: 32 bits
```

```plaintext
p Info
Stable: No
Width: 64 bits
```

```plaintext
k Info
Stable: Yes
Width: 32 bits
```
**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;
}
```

![Diagram of component master_comp](image)
The Component Viewer shows the following details for these interface arguments:

<table>
<thead>
<tr>
<th>Stable</th>
<th>Describes whether the interface argument is stable. That is, whether the <code>hls_stable_argument</code> attribute was applied.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data width</td>
<td>The width of the memory-mapped data bus in bits.</td>
</tr>
<tr>
<td>Address width</td>
<td>The width of the memory-mapped address bus in bits.</td>
</tr>
<tr>
<td>Latency</td>
<td>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.</td>
</tr>
<tr>
<td>Maximum burst</td>
<td>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.</td>
</tr>
<tr>
<td>Alignment</td>
<td>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.</td>
</tr>
</tbody>
</table>

![Diagram showing interface details](image_url)
The Component Viewer shows the following details for global memories:

- **Memory address space number**: The memory address space number for global memory.
- **Number of banks**: The number of memory banks contained in the memory.
- **Argument Name**: The names of arguments that access the global memory.

### 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 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 Component Viewer also opens up the header file.
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:

```
#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 Interface Arguments

When you declare a pointer argument as a slave memory, the Component 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.

```
#include "HLS/hls.h"
#include "stdio.h"
```
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 by a `<stream name>` node connected to the corresponding RD node (for `stream_in<>`) or WR node (for `stream_out<>`).

```
#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 shows the following details for streaming interface arguments:

**Width**
The width of the data bus in bits.
Depth  The depth of the stream in words
The word size of the stream is the size of the stream datatype.

Bits per symbol  Describes how the data is broken into symbols on the data bus.

Uses Packets  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.

Uses Valid  (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.

Uses Reader  (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.

B.6.2. Reviewing Memory Replication and Stallable LSU Information

The Component Viewer report in the High Level Design Report (report.html) shows an abstracted netlist of your kernel design. In the Component Viewer, you can visualize load-store units (LSUs) between a component and different memories, streams, and loops.

Consider the following code excerpt from the transpose_and_fold component (part of the tutorial files provided in <QPDS_install_dir>/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();
  }
  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++) {
    ...
  }
```

B. Reviewing the High Level Design Report (report.html)

UG-20037 | 2019.09.30
The figure below shows that Block3 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 10. Component View of the transpose_and_fold Component**
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.

**Figure 11. Detailed View of Node and Tooltip**

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 function view of the Graph 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.6.3. Viewing Your Component Memory System

Data movement is a bottleneck in many algorithms. The Component Memory Viewer in the High Level Design Report (report.html) shows you how the Intel HLS Compiler Standard Edition 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.
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.
Component Memory Viewer

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 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 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.
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>Verification Statistics</th>
<th>Invocations</th>
<th>Latency (not measured)</th>
<th># (expected)</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>dut</td>
<td>1</td>
<td>4.4μs</td>
<td>1,1</td>
<td>Link for details</td>
</tr>
<tr>
<td>dut (Enqueue location)</td>
<td>100</td>
<td>4.4μs</td>
<td>1,1</td>
<td>Link for details</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:

### B.8. Accessing HLD FPGA Reports in JSON Format

The high level design report data for the Intel HLS Compiler 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>
<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>lmv.json</td>
<td>Component Viewer</td>
</tr>
<tr>
<td><strong>STD</strong></td>
<td>Component Memory Viewer</td>
</tr>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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": 
"Data": [Yes, No, <#|n/a>, <II|n/a>] # The data field corresponds to "Pipelined", "II", "Bottleneck"
```