Intel® High Level Synthesis Compiler Pro Edition

Best Practices Guide

Updated for Intel® Quartus® Prime Design Suite: 20.3
# Intel® HLS Compiler Pro Edition Best Practices Guide


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

3. **3. FPGA Concepts** ........................................................................................................................................ 7
   3.1. FPGA Architecture Overview .............................................................................................................. 7
   3.1.1. Adaptive Logic Module (ALM) ........................................................................................................ 8
   3.1.2. Digital Signal Processing (DSP) Block ........................................................................................... 10
   3.1.3. Random-Access Memory (RAM) Blocks ...................................................................................... 10
   3.2. Concepts of FPGA Hardware Design .................................................................................................. 10
   3.2.1. Maximum Frequency ($f_{MAX}$) .................................................................................................... 10
   3.2.2. Latency ........................................................................................................................................... 11
   3.2.3. Pipelining ...................................................................................................................................... 11
   3.2.4. Throughput ................................................................................................................................... 12
   3.2.5. Datapath ....................................................................................................................................... 12
   3.2.6. Control Path ................................................................................................................................. 12
   3.2.7. Occupancy .................................................................................................................................... 13
   3.3. Methods of Hardware Design ........................................................................................................... 14
   3.3.1. How Source Code Becomes a Custom Hardware Datapath ............................................................ 14
   3.3.2. Scheduling ................................................................................................................................... 17
   3.3.3. Mapping Parallelism Models to FPGA Hardware ........................................................................... 23
   3.3.4. Memory Types .............................................................................................................................. 32

4. **4. Interface Best Practices** ................................................................................................................... 38
   4.1. Choose the Right Interface for Your Component ................................................................................ 39
   4.1.1. Pointer Interfaces ........................................................................................................................... 40
   4.1.2. Avalon Memory Mapped Master Interfaces .................................................................................. 42
   4.1.3. Avalon Memory Mapped Slave Interfaces .................................................................................... 45
   4.1.4. Avalon Streaming Interfaces ......................................................................................................... 47
   4.1.5. Pass-by-Value Interface ................................................................................................................ 49
   4.2. Control LSUs For Your Variable-Latency MM Master Interfaces .................................................... 52
   4.3. Avoid Pointer Aliasing ....................................................................................................................... 53

5. **5. Loop Best Practices** ........................................................................................................................ 55
   5.1. Reuse Hardware By Calling It In a Loop ............................................................................................ 56
   5.2. Parallelize Loops .............................................................................................................................. 57
   5.2.1. Pipeline Loops ............................................................................................................................... 58
   5.2.2. Unroll Loops .................................................................................................................................. 59
   5.2.3. Example: Loop Pipelining and Unrolling .................................................................................... 60
   5.3. Construct Well-Formed Loops .......................................................................................................... 62
   5.4. Minimize Loop-Carried Dependencies ............................................................................................. 63
   5.5. Avoid Complex Loop-Exit Conditions .............................................................................................. 64
   5.6. Convert Nested Loops into a Single Loop .......................................................................................... 65
   5.7. Declare Variables in the Deepest Scope Possible ............................................................................. 66
   5.8. Raise Loop II to Increase $f_{MAX}$ ...................................................................................................... 66
   5.9. Control Loop Interleaving ............................................................................................................... 67

6. **6. $f_{MAX}$ Bottleneck Best Practices** .................................................................................................. 68

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

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

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

Windows  
C:\intelFPGA_pro\20.3

Linux  
/home/<username>/intelFPGA_pro/20.3

About the Intel HLS Compiler Pro Edition Documentation Library

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

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

<table>
<thead>
<tr>
<th>Title and Description</th>
<th>PRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release Notes</td>
<td>Link</td>
</tr>
<tr>
<td>Provide late-breaking information about the Intel HLS Compiler.</td>
<td></td>
</tr>
<tr>
<td>Getting Started Guide</td>
<td>Link</td>
</tr>
<tr>
<td>Get up and running with the Intel HLS Compiler by learning how to initialize your compiler environment and reviewing the various design examples and tutorials provided with the Intel HLS Compiler.</td>
<td></td>
</tr>
<tr>
<td>User Guide</td>
<td>Link</td>
</tr>
<tr>
<td>Provides instructions on synthesizing, verifying, and simulating intellectual property (IP) that you design for Intel FPGA products. Go through the entire development flow of your component from creating your component and testbench up to integrating your component IP into a larger system with the Intel Quartus Prime software.</td>
<td></td>
</tr>
<tr>
<td>Reference Manual</td>
<td>Link</td>
</tr>
<tr>
<td>Provides reference information about the features supported by the Intel HLS Compiler. Find details on Intel HLS Compiler command options, header files, pragmas, attributes, macros, declarations, arguments, and template libraries.</td>
<td></td>
</tr>
<tr>
<td>Best Practices Guide</td>
<td>Link</td>
</tr>
<tr>
<td>Provides techniques and practices that you can apply to improve the FPGA area utilization and performance of your HLS component. Typically, you apply these best practices after you verify the functional correctness of your component.</td>
<td></td>
</tr>
<tr>
<td>Quick Reference</td>
<td>Link</td>
</tr>
<tr>
<td>Provides a brief summary of Intel HLS Compiler declarations and attributes on a single two-sided page.</td>
<td></td>
</tr>
</tbody>
</table>

Intel Corporation. All rights reserved. Agigx, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel’s standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.
2. Best Practices for Coding and Compiling Your Component

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

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

- **Understand FPGA Concepts** on page 7
  A key best practice to help you get the most out of the Intel HLS Compiler is to understand important concepts about FPGAs. With an understanding of FPGA architecture, and some FPGA hardware design concepts and methods, you can create better designs that take advantage of your target FPGA devices.

- **Interface Best Practices** on page 38
  With the Intel High Level Synthesis Compiler, your component can have a variety of interfaces: from basic wires to the Avalon Streaming and Avalon Memory-Mapped Master interfaces. Review the interface best practices to help you choose and configure the right interface for your component.

- **Loop Best Practices** on page 55
  The Intel High Level Synthesis Compiler pipelines your loops to enhance throughput. Review these loop best practices to learn techniques to optimize your loops to boost the performance of your component.

- **Memory Architecture Best Practices** on page 69
  The Intel High Level Synthesis Compiler infers efficient memory architectures (like memory width, number of banks and ports) in a component by adapting the architecture to the memory access patterns of your component. Review the memory architecture best practices to learn how you can get the best memory architecture for your component from the compiler.

- **System of Tasks Best Practices** on page 84
  Using a system of HLS tasks in your component enables a variety of design structures that you can implement including executing multiple loops in parallel and sharing an expensive compute block.

- **Datatype Best Practices** on page 89
  The datatypes in your component and possible conversions or casting that they might undergo can significantly affect the performance and FPGA area usage of your component. Review the datatype best practices for tips and guidance how best to control datatype sizes and conversions in your component.
• Alternative Algorithms

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

A key best practice to help you get the most out of the Intel HLS Compiler is to understand important concepts about FPGAs. With an understanding of FPGA architecture, and some FPGA hardware design concepts and methods, you can create better designs that take advantage of your target FPGA devices.

3.1. FPGA Architecture Overview

A field-programmable gate array (FPGA) is a reconfigurable semiconductor integrated circuit (IC).

FPGAs occupy a unique computational niche relative to other compute devices, such as central and graphics processing units (CPUs and GPUs), and custom accelerators, such as application-specific integrated circuits (ASICs). CPUs and GPUs have a fixed hardware structure to which a program maps, while ASICs and FPGAs can build custom hardware to implement a program.

While a custom ASIC generally outperforms an FPGA on a specific task, ASICs take significant time and money to develop. FPGAs are a cheaper off-the-shelf alternative that you can reprogram for each new application.

An FPGA is made up of a grid of configurable logic, known as adaptive logic modules (ALMs), and specialized blocks, such as digital signal processing (DSP) blocks and random-access memory (RAM) blocks. These programmable blocks are combined using configurable routing interconnects to implement complete digital circuits.

The total number of ALMs, DSP blocks, and RAM blocks used by a design is often referred to as the FPGA area or area that the design uses.
3.1.1. Adaptive Logic Module (ALM)

The basic building block in an FPGA is an adaptive logic module (ALM).

A simplified ALM consists of a lookup table (LUT) and an output register from which the compiler can build any arbitrary Boolean logic circuit.
The following figure illustrates a simplified ALM:

![ALM Diagram]

### 3.1.1.1. Lookup Table (LUT)

A *lookup table (LUT)* that implements an arbitrary Boolean function of N inputs is often referred to as an *N-LUT*.

### 3.1.1.2. Register

A *register* is the most basic storage element in an FPGA. It has an input (in), an output (out), and a clock signal (clk). It is synchronous, that is, it synchronizes output changes to a clock. In an ALM, a register may store the output of the LUT.

The following figure illustrates a register:

![Register Diagram]

*Note:* The clock signal is implied and not shown in some figures.

The following figure illustrates the waveform of register signals:

![Waveform Diagram]
The input data propagates to the output on every clock cycle. The output remains unchanged between clock cycles.

### 3.1.2. Digital Signal Processing (DSP) Block

A digital signal processing (DSP) block implements specific support for common fixed-point and floating-point arithmetic, which reduces the need to build equivalent logic from general-purpose ALMs.

The following figure illustrates a simplified three-input DSP block consisting of a multiplier (×) and an adder (+):

![Simplified DSP Block Diagram]

### 3.1.3. Random-Access Memory (RAM) Blocks

A random-access memory (RAM) block implements memory by using a high density of memory cells.

For more information, refer to Memory Types.

### 3.2. Concepts of FPGA Hardware Design

#### 3.2.1. Maximum Frequency (f<sub>MAX</sub>)

The maximum clock frequency at which a digital circuit can operate is called its f<sub>MAX</sub>. The f<sub>MAX</sub> is the maximum rate at which the outputs of registers are updated.

The physical propagation delay of the signal across Boolean logic between two consecutive register stages limits the clock speed. This propagation delay is a function of the complexity of the combinational logic in the path.

The path with the most combinational logic elements (and the highest delay) limits the speed of the entire circuit. This speed limiting path is often referred to as the critical path.

The f<sub>MAX</sub> is calculated as the inverse of the critical path delay. You may want to have high f<sub>MAX</sub> since it results in high performance in the absence of other bottlenecks.
3.2.2. Latency

Latency is the measure of how long it takes to complete one or more operations in a digital circuit. You can measure latency at different granularities. For example, you can measure the latency of a single operation or the latency of the entire circuit.

You can measure latency in time (for example, microseconds) or in clock cycles. Typically, clock cycles are the preferred way to express latency because measuring latency in clock cycles disconnects latency from your circuit clock frequency. By expressing latency independent of circuit clock frequency, it is easier to discern the true impact of circuit changes to the performance of the circuit.

You may want to have low latency, but lowering latency might result in decreased \( f_{\text{MAX}} \).

For more information and an example, refer to Pipelining.

3.2.3. Pipelining

Pipelining is a design technique used in synchronous digital circuits to increase \( f_{\text{MAX}} \). Pipelining involves adding registers to the critical path, which decreases the amount of logic between each register. Less logic takes less time to execute, which enables an increase in \( f_{\text{MAX}} \).

The critical path in a circuit is the path between any two consecutive registers with the highest latency. That is, the path between two consecutive registers where the operations take the longest to complete.

Pipelining is especially useful when processing a stream of data. A pipelined circuit can have different stages of the pipeline operating on different input stream data in the same clock cycle, which leads to better data processing throughput.

Example

Consider a simple circuit with operations A and B on the critical path. If operation A takes 5 ns to complete and operation B takes 15 ns to complete, then the time delay on the critical path is 20 ns. This results in an \( f_{\text{MAX}} \) of 50 MHz (\( 1/\text{max\_delay} \)).

Figure 1. Unpipelined Logic Block with the \( f_{\text{MAX}} \) of 50 MHz and Latency of Two Clock Cycles

If a pipeline register is added between A and B, the critical path changes. The delay on the critical path is now 15 ns. Pipelining this block results in an \( f_{\text{MAX}} \) of 66.67 MHz, and the maximum delay between two consecutive registers is 15 ns.
While pipelining generally results in a higher $f_{\text{MAX}}$, it increases latency. In the previous example, the latency of the block containing A and B increases from two to three clock cycles after pipelining.

**Related Information**

Pipeline Loops on page 58

### 3.2.4. Throughput

Throughput of a digital circuit is the rate at which data is processed.

In the absence of other bottlenecks, higher $f_{\text{MAX}}$ results in higher throughput (for example, samples/second).

Throughput is a good measure of the performance of a circuit, and throughput and performance are often used interchangeably when discussing a circuit.

### 3.2.5. Datapath

A **datapath** is a chain of registers and Boolean logic in a digital circuit that performs computations.

For example, the datapath in Pipelining on page 11 consists of all of the elements shown, from the input register to the last output register.

In contrast, memory blocks are outside the datapath and reads and writes to memory are also considered to be outside of the datapath.

### 3.2.6. Control Path

While the datapath is the path on which computations occur, the control path is the path of signals that control the datapath circuitry.

The control path is the logic added by the compiler to manage the flow of data through your design. Control paths include controls such as the following:

- Handshaking flow control

  Handshaking ensures that one part of your design is ready and able to accept data from another part of your design.
Loop control
Loop controls control the flow of data through the hardware generated for loops in your code, including any loop carried dependencies.

Branch control
Branch controls implement conditional statements in your code. Branch control can include parallelizing parts of conditional statements to improve performance.

The control path also consumes FPGA area, and the compiler uses techniques like clustering the datapath to help reduce the control path and save area. To learn about clustering, refer to Clustering the Datapath on page 18.

3.2.7. Occupancy

The occupancy of a datapath at a point in time refers to the proportion of the datapath that contains valid data.

The occupancy of a circuit over the execution of a program is the average occupancy over time from the moment the program starts to run until it has completed.

Unoccupied portions of the datapath are often referred to as bubbles. Bubbles are analogous to a "no operation" (no-op) instructions for a CPU that have no effect on the final output.

Decreasing bubbles increases occupancy. In the absence of other bottlenecks, maximizing occupancy of the datapath results in higher throughput.
3.3. Methods of Hardware Design

Traditionally, you program an FPGA using a hardware description language (HDL) such as Verilog or VHDL. However, a recent trend is to use higher-level languages. Higher levels of abstraction can reduce the design time and increase the portability of your design.

The sections that follow discuss how Intel HLS Compiler maps high-level languages to a hardware datapath.

3.3.1. How Source Code Becomes a Custom Hardware Datapath

Based on your source code and the following principles, the Intel HLS Compiler builds a custom hardware datapath in the FPGA logic:

- The datapath is functionally equivalent to the C++ program described by the source. Once the datapath is constructed, the compiler orchestrates work items and loop iterations such that the hardware is effectively occupied.
- The compiler builds the custom hardware datapath while minimizing area of the FPGA resources (like ALMs and DSPs) used by the design.
3.3.1.1. Mapping Source Code Instructions to Hardware

For fixed architectures, such as CPUs and GPUs, a compiler compiles code into a set of instructions that run on functional units that have a fixed functionality. For these fixed architectures to be useful in a broad range of applications, some of their available functional units are not useful to every program. Unused functional units mean that your program does not fully occupy the fixed architecture hardware.

FPGAs are not subject to these restrictions of fixed functional units. On an FPGA, you can synthesize a specialized hardware datapath that can be fully occupied for an arbitrary set of instructions, which means you can be more efficient with the silicon area of your chip.

By implementing your algorithm in hardware, you can fill your chip with custom hardware that is always (or almost always) working on your problem instead of having idle functional units.

The Intel HLS Compiler maps statements from the source code to individual specialized hardware operations, as shown in the example in the following image:

```
c = a + b;
```

In general, each instruction maps to its own unique instance of a hardware operation. However, a single statement can map to more than one hardware operation, or multiple statements can combine into a single hardware operation when the compiler finds that it can generate hardware that is more efficient.

The latency of hardware operations is dependent on the complexity of the operation and the target $f_{\text{MAX}}$.

The compiler takes these hardware operations and connects them into a graph based on their dependencies. When operations are independent, the compiler automatically infers parallelism by executing those operations simultaneously in time.

The following figure shows a dependency graph created for the hardware datapath. The dependency graph shows how the instruction is mapped to hardware operations and how the hardware operations are connected based on their dependencies. The loads in this example instruction are independent of each other and can therefore run simultaneously.
3.3.1.2. Mapping Arrays and Their Accesses to Hardware

Similar to the mapping of statements to specialized hardware operations, the compiler maps arrays to hardware memories.

The datapath interacts with this memory through load/store units (LSUs), which are inferred from array accesses in the source code.

The following figure illustrates a simple example of mapping arrays and their accesses to hardware:

A RAM can have a limited number of read ports and write ports, but a datapath can have many LSUs. When the number of LSUs does not match the available number of read and write ports, the compiler uses techniques like replication, double-pumping, sharing, and arbitration. For more details about configuring memories, refer the following topics:

- Memory Architecture Best Practices on page 69

FPGAs provide specialized hardware block RAMs that you can configure and combine to match the size of your arrays. Customizing your memory configuration for your design can provide terabytes-per-second of on-chip memory bandwidth because each of these memories can interact with the datapath simultaneously.
Arrays might also be implemented in your component datapath. In this case, the array contents are stored as registers in the datapath when your algorithm is pipelined (as discussed in Pipelining on page 25). Storing array contents as registers in the datapath can improve performance in some cases, but it is a design decision whether to implement an array as registers or as memories.

When you access an array that is implemented as registers, LSUs are not used. The compiler might choose to use a select or a barrel shifter instead.

```c
hls_register int a[256];

a[i] = writeVal;  // store to array
readVal = a[j];   // load from array
```

3.3.2. Scheduling

*Scheduling* refers to the process of determining the clock cycles at which each operation in the datapath executes.

Pipelining is the outcome of scheduling.

**Related Information**

Pipelining on page 11

3.3.2.1. Dynamic Scheduling

The Intel HLS Compiler generates pipelined datapaths that are dynamically scheduled.

A *dynamically scheduled* portion of the datapath does not pass data to its successor until its successor signals that it is ready to receive it.

This signaling is accomplished using handshaking control logic. For example, a variable latency load from memory may refuse to accept its predecessors' data until the load is complete.

Handshaking helps remove bubbles in the pipeline, which increases occupancy. For more information about bubbles, refer to Occupancy.
The following figure illustrates four regions of dynamically scheduled logic:

**Figure 4. Dynamically Scheduled Logic**

Black arrows represent data and valid signals and red arrows represent signals to stall incoming valid data flow.

3.3.2.2. Clustering the Datapath

Dynamically scheduling all operations adds overhead in the form of additional FPGA area needed to implement the required handshaking control logic.

To reduce this overhead, the compiler groups fixed latency operations into *clusters*. A cluster of fixed latency operations, such as arithmetic operations, needs fewer handshaking interfaces, thereby reducing the area overhead.
If A, B, and C from Figure 1 do not contain variable latency operations, the compiler can cluster them together, as illustrated in Figure 1.

Clustering the logic reduces area by removing the need for signals to stall data flow in addition to other handshaking logic within the cluster.
Cluster Types

The Intel HLS Compiler can create the following types of clusters:

- **Stall-Enable Cluster (SEC)**: This cluster type passes the handshaking logic to every pipeline stage in the cluster in parallel. If the cluster is stalled by logic from further down in the datapath, all logic in the SEC stalls at the same time. If there were no cluster, the stall propagates back from register to register, instead of all registers in the cluster stalling in parallel.

  ![Stall-Enable Cluster](image)

- **Stall-Free Cluster (SFC)**: This cluster type adds a first in, first out (FIFO) buffer to the end of the cluster that can accommodate at least the entire latency of the pipeline in the cluster. This FIFO is often called a *capacity FIFO* because the FIFO can accommodate the capacity of the cluster. The *capacity* of a cluster is the minimum number of valid data pieces that a cluster can operate on simultaneously. Capacity is always less than or equal to the latency of the datapath.

  Because of this FIFO, the pipeline stages in the cluster do not require any handshaking logic. The stages can run freely and drain into the capacity FIFO, even if the cluster is stalled from logic further down in the datapath.


Cluster Characteristics

Different cluster architectures result in different characteristics for each cluster type:

- **Bubble Handling**: SECs remove only leading bubbles. A leading bubble is a bubble that arrives before the first piece of valid data arrives in the cluster. SECs do not remove any arriving afterwards.

  SFCs can use the capacity FIFO to remove all bubbles from the pipeline.

- **Capacity**: For SECs, capacity can vary where the best-case capacity is equal to the number of register stages and the worst-case capacity is 1 because all registers in an SEC share the same stall signal. A capacity of 1 means that the cluster may be able to hold only a single piece of data, regardless of the number of pipeline stages in the cluster.

  This worst-case scenario corresponds to a single valid data at the end of the cluster pipeline and bubbles in the rest of the pipeline.

  SFCs have a capacity equal to the depth of the capacity FIFO.

- **Handshaking**: The capacity FIFO inside SFCs allow them to take advantage of hyper-optimized handshaking between clusters. For more information, refer to Hyper Optimized Handshaking.

  SECs do not support this capability.
3.3.2.3. Handshaking Between Clusters

By default, the handshaking protocol between clusters is a simple stall/valid protocol. Data from the upstream cluster is consumed when the stall signal is low and the valid signal is high.

**Figure 8. Handshaking Between Clusters**

![Handshaking Between Clusters Diagram]

**Hyper-Optimized Handshaking**

If the distance across the FPGA between these two clusters is large, handshaking may become the critical path that affects peak $f_{MAX}$ in the design.

To improve these cases, the Intel HLS Compiler can add pipelining registers to the stall/valid protocol to ease the critical path and improve $f_{MAX}$. This enhanced handshaking protocol is called *hyper-optimized handshaking*.

**Figure 9. Hyper-Optimized Handshaking Data Flow**

![Hyper-Optimized Handshaking Data Flow Diagram]

Valid data will not stop flowing until 4 cycles after Cluster 2 asserts stall.
The following timing diagram illustrates an example of upstream cluster 1 and downstream cluster 2 with two pipelining registers inserted in-between:

**Figure 10. Hyper-Optimized Handshaking**

<table>
<thead>
<tr>
<th>Clk</th>
<th>oValid_1</th>
<th>iStall_1</th>
<th>iData_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Restriction:** Hyper-optimized handshaking is currently available only for the Intel Agilex™ and Intel Stratix® 10 device families.

### 3.3.3. Mapping Parallelism Models to FPGA Hardware

This section describes how to map parallelism models to FPGA hardware:

#### 3.3.3.1. Data Parallelism

Traditional instruction-set-architecture-based (ISA-based) accelerators, such as GPUs, derive data parallelism from vectorized instructions and by executing the same operation on multiple processing units.

In comparison, FPGAs derive their performance by taking advantage of their spatial architecture. FPGA compilers do not require you to vectorize your code. The compiler vectorizes your code automatically whenever it can.

The generated hardware implements data parallelism in the following ways:

- Executing Independent Operations Simultaneously on page 23
- Pipelining on page 25

#### 3.3.3.1.1. Executing Independent Operations Simultaneously

As described in Mapping Source Code Instructions to Hardware, the compiler can automatically identify independent operations and execute them simultaneously in hardware.

This simultaneous execution of independent operations combined with pipelining is how performance through data parallelism is achieved on an FPGA.
The following image illustrates an example of an adder and a multiplier, which are scheduled to execute simultaneously while operating on separate inputs:

**Figure 11. Automatic Vectorization in the Generated Hardware Datapath**

This automatic vectorization is analogous to how a superscalar processor takes advantage of instruction-level parallelism, but this vectorization happens statically at compile time instead of dynamically, at runtime.

Because determining instruction-level parallelism occurs at compile time, there is no hardware or runtime cost of dependency checking for the generated hardware datapath. Additionally, the flexible logic and routing of an FPGA means that only the available resources (like ALMs and DSPs) of the FPGA restrict the number of independent operations that can occur simultaneously.

**Unrolling Loops**

You can unroll loops in the design by using loop attributes. Loop unrolling decreases the number of iterations executed at the expense of increasing hardware resource consumption corresponding to executing multiple iterations of the loop simultaneously.

Once unrolled, the hardware resources are scheduled as described in Scheduling.

The Intel HLS Compiler never attempts to unroll any loops in your source code automatically. You must always control loop unrolling by using the corresponding pragma. For details, refer to *Loop Unrolling (unroll Pragma)* in the Intel High Level Synthesis Compiler Reference Manual.

**Conditional Statements**

The Intel HLS Compiler attempts to eliminate conditional or branch statements as much as possible.

Conditionally executed code becomes predicated in the hardware. That is, branched instructions are replaced with conditionally-executed instructions. Predication increases the possibilities for executing operations simultaneously and achieving better performance. Additionally, removing branches allows the compiler to apply other optimizations to the design.
### Related Information

Unroll Loops on page 59

### 3.3.3.1.2. Pipelining

Similar to the implementation of a CPU with multiple pipeline stages, the compiler generates a deeply-pipelined hardware datapath. For more information, refer to Concepts of FPGA Hardware Design and How Source Code Becomes a Custom Hardware Datapath.

Pipelining allows for many data items to be processed concurrently (in the same clock cycle) while making efficient use of the hardware in the datapath by keeping it occupied.
Example 1. Example of Vectorization of the Datapath vs. Pipelining the Datapath

Consider the following example of code mapping to hardware:

Figure 13. Example Code Mapping to Hardware

\[
\text{Mem}[100] += 42 \times \text{Mem}[101];
\]
A naïve approach to parallelizing this code on FPGAs would be to vectorize it by replicating the datapath or by explicitly writing instructions that operate on vector types, as shown in the following figure:

**Figure 14. Vectorizing the Datapath Resulting in High Throughput but Low Occupancy**

![Diagram showing vectorized operations]

In the above diagram, $t_0$, $t_1$, $t_2$ and so on, represent different data items flowing through the datapath. However, this vectorization results in an inefficient use of the hardware since many of the operations are sitting idle while other parts of the datapath are operating on the data.
An alternative approach is to increase occupancy of the existing hardware by sending a new data item into the datapath on every clock cycle, as shown in the following figure:

Figure 15. Pipelining the Datapath Results in High Throughput and High Occupancy

In this example, both approaches achieve the same throughput of one data item per cycle. However, the pipelined implementation uses one third of the hardware and keeps that hardware utilized more effectively.

Understanding where the data you need to pipeline is coming from is key to achieving high performance designs on the FPGA. You can use the following sources of data to take advantage of pipelining:

- Components
- Loop iterations

**Pipelining Loops Within a Component**

Within a component, loops are the primary source of pipeline parallelism.

When the Intel HLS Compiler pipelines a loop, it attempts to schedule the loop execution such that the next iteration of the loop enters the pipeline before the previous iteration has completed. This pipelining of loop iterations can lead to higher throughput.

The number of clock cycles between iterations of the loop is called the *Initiation Interval* (II).

For the highest performance, a loop iteration would start every clock cycle, which corresponds to an II of 1.

Data dependencies that are carried from one loop iteration to another can affect the ability to achieve II of 1. These dependencies are called *loop-carried dependencies*.

The II of a loop must be high enough to accommodate all loop carried dependencies.
Tip: The II required to satisfy this constraint is a function of the \( f_{\text{MAX}} \) of the design. If the \( f_{\text{MAX}} \) is lower, the II might also be lower. Conversely, if the \( f_{\text{MAX}} \) is higher, a higher II might be required.

The Intel HLS Compiler automatically identifies these dependencies and builds hardware to resolve them while minimizing the II, subject to the target \( f_{\text{MAX}} \).

Refer to Figure 16 on page 29 where the source code describes a dependency between loop iterations (that is, accesses to array \( c \)) that can be satisfied within one clock cycle. This allows the loop to achieve an II of 1 and maximize performance.

**Figure 16. Pipelining a Datapath with Loop Iteration**

```c
for (int i=1; i < n; i++) {
    c[i] = c[i-1] + b[i];
}
```

The dependency on the value stored to \( c \) in the previous iteration is resolved in a single clock cycle, so an II of 1 is achieved for the loop even though the iterations are not independent.

For additional information about pipelining loops, refer to Pipeline Loops on page 58.
When the Intel HLS Compiler cannot initially achieve II of 1, you have several optimization strategies to choose from:

- **Interleaving**: When a loop nest with an inner loop II that is greater than 1, the Intel HLS Compiler can attempt to interleave iterations of the outer loop into iterations of the inner loop to better utilize the hardware resources and achieve higher throughput.

![Interleaving](image)

For additional information about controlling interleaving in your component, refer to *Loop Interleaving Control (max_interleaving Pragma)* in the *Intel High Level Synthesis Compiler Reference Manual*. 
• **Speculative Execution**: When the critical path that affects II is the computation of the exit condition and not a loop-carried dependency, the Intel HLS Compiler can attempt to relax this scheduling constraint by speculatively continuing to execute iterations of the loop while the exit condition is being computed.

If it is determined that the exit condition is satisfied, the effects of these extra iterations are suppressed.

This speculative execution can achieve lower II and higher throughput, but it will incur additional overhead between loop invocations (equivalent to the number of speculated iterations). A larger loop trip count helps to minimize this overhead.

**Terminology**

**Reminder**

A loop invocation is what starts a series of loop iterations.

One loop iteration is one execution of the body of a loop.

**Figure 18. Speculative Execution**

This diagram does not show extra non-operative iterations incurred between loop invocations.

For additional information about speculation, refer to *Loop Iteration Speculation (speculated_iterationsPragma)* in the Intel High Level Synthesis Compiler Reference Manual.

These optimizations are applied automatically by the Intel HLS Compiler, and additionally can be controlled through pragma statements in the design.

**Pipelining Across Component Invocations**

The pipelining of work across component invocations is similar to how loops are pipelined.
The Intel HLS Compiler attempts to schedule the execution of component invocations such that the next invocation of a component enters the pipeline before the previous invocation has completed.

### 3.3.3.2. Task Parallelism

The compiler achieves concurrency by scheduling independent individual operations to execute simultaneously, but it does not achieve concurrency at coarser granularities (for example, across loops).

For larger code structures to execute in parallel with each other, you must write them as separate components or tasks that launch simultaneously. These components or tasks then run asynchronously with respect to each other and you can achieve synchronization and communication using pipes, as shown in the following figure:

**Figure 19. Multiple Task Functions Running Asynchronously**

```cpp
for(i=0..N) {
    ...
    mypipe::write(x);
    ...
}
```

```cpp
for(i=0..N) {
    ...
    y = mypipe::read();
    ...
}
```

This is similar to how a program running on a CPU can leverage threads running on separate cores to achieve simultaneous asynchronous behavior.

For details, see *Systems of Tasks* in the *Intel High Level Synthesis Compiler Pro Edition Reference Manual*.

### 3.3.4. Memory Types

The compiler maps user-defined arrays in source code to hardware memories. You can classify these hardware memories into the following categories:

- **Component Memory** on page 32
  
  Component memory is memory allocated from memory resources (such as RAM blocks) available on the FPGA.

- **External Memory** on page 37
  
  External memory is memory resources that are outside of the FPGA.

#### 3.3.4.1. Component Memory

If you declare an array inside your component, the Intel HLS Compiler creates component memory in hardware. Component memory is sometimes referred to as *local memory* or *on-chip memory* because it is created from memory resources (such as RAM blocks) available on the FPGA.
The following source code snippet results in the creation of a component memory system, an interface to an external memory system, and access to these memory systems:

```c
#include <HLS/hls.h>
constexpr int SIZE = 128;
constexpr int N = SIZE - 1;

using MasterInterface = ihc::mm_master<int, ihc::waitrequest<true>,
                                          ihc::latency<0>>;

component void memoryComponent(MasterInterface &masterA)
{
    hls_memory int T[SIZE]; // declaring an array as a component memory
    for (unsigned i = 0; i < SIZE; i++)
    {
        T[i] = i; // writing to component memory
    }
    for (int i = 0; i < N; i += 2)
    {
        // reading from a component memory and writing to a external
        // Avalon memory-mapped slave component through an Avalon
        // memory-mapped master interface
        masterA[i] = T[i] + T[i + 1];
    }
}
```

The compiler performs the following tasks to build a memory system:

- Build a component memory from FPGA memory resources (such as block RAMs) and presents it to the datapath as a single memory.
- Map an array access to a load-store unit (LSU) in the datapath that transacts with the component memory through its ports.
- Build the component memory and LSUs and retains complete control over their structure.
- Automatically optimizes the component memory geometry to maximize the bandwidth available to loads and stores in the datapath.
- Attempts to guarantee that component memory accesses never stall.

To learn more about controlling memory system architectures, review the following topics:

- Memory Architecture Best Practices on page 69

**Stallable and Stall-Free Memory Systems**

Accesses to a memory (read or write) can be stall-free or stallable:

**Stall-free memory access**
A memory access is stall-free if it has contention-free access to a memory port. A memory system is stall-free if each of its memory operations has contention-free access to a memory port.

**Stallable memory access**
A memory access is stallable when two memory accesses want to access one memory port in the same clock cycle, one of the memory accesses is delayed (or stalled) until the memory port in contention is available.
As much as possible, the Intel HLS Compiler tries to create stall-free memory systems for your component.

**Figure 20. Examples of Stall-free and Stallable Memory Systems**

This figure shows the following example memory systems:

- **A**: A stall-free memory system
  
  This memory system is stall-free because, even though the reads are scheduled in the same cycle, they are mapped to different ports. There is no contention for accessing the memory ports.

- **B**: A stall-free memory system
  
  This memory system is stall-free because the two reads are statically-scheduled to occur in different clock cycles. The two reads can share a memory port without any contention for the read access.

- **C**: A stallable memory system
  
  This memory system is stallable because two reads are mapped to the same port in the same cycle. The two reads happen at the same time. There reads require collision arbitration to manage their port access requests, and arbitration can affect throughput.

A component memory system consists of the following parts:
Port

A memory *port* corresponds to either a read-operation or a write-operation in the datapath. A port is connected to one or more load-store unit (LSU). An LSU can be connected to multiple ports.

Bank

A memory *bank* is a division of the component memory system that contains a subset of the data stored. That is, all the data stored for a component is split across banks, with each bank containing a unique piece of the stored data.

A memory system always has at least one bank.

Replicate

A memory bank *replicate* is a copy of the data in the memory bank with its own ports. All replicates in a bank contain the same data. Each replicate can be accessed independent of the others.

A memory bank always has at least one replicate.

Private Copy

A *private copy* is a copy of the data in a replicate that is created for nested loops to enable concurrent iterations of the outer loop.

A replicate can comprise multiple private copies, with each iteration of an outer loop having its own private copy. Because each outer loop iteration has its own private copy, private copies are not expected to all contain the same data.
The following figure illustrates the relationship between banks, replicates, ports, and private copies:

**Figure 21. Schematic Representation of Component Memories Showing the Relationship between Banks, Replicates, Ports, and Private Copies**

- **A) Simplest memory configuration**
  - Memory Size = 4096 bytes
  - Bank 0 (1024x4)
  - Replicate 0 (1024x4)
  - Copy 0 (1024x4)
  - Depth = 1024 words
  - Width = 4 bytes

- **B) Memory with two banks**
  - Memory Size = 8192 bytes
  - Bank 0 (1024x4)
  - Replicate 0 (1024x4)
  - Copy 0 (1024x4)
  - Bank 1 (1024x4)
  - Replicate 0 (1024x4)
  - Copy 0 (1024x4)
  - Depth = 2048 words
  - Width = 4 bytes

- **C) Memory with two replicates**
  - Memory Size = 8192 bytes
  - Bank 0 (2048x4)
  - Replicate 0 (1024x4)
  - Copy 0 (1024x4)
  - Replicate 1 (1024x4)
  - Copy 0 (1024x4)
  - Depth = 2048 words
  - Width = 4 bytes

- **D) Memory with two private copies**
  - Memory Size = 8192 bytes
  - Bank 0 (2048x4)
  - Replicate 0 (2048x4)
  - Copy 0 (1024x4)
  - Copy1 (1024x4)
  - Depth = 2048 words
  - Width = 4 bytes
Strategies that Enable Concurrent Stall-Free Memory Accesses

The compiler uses a variety of strategies to ensure that concurrent accesses are stall-free including:

- Adjusting the number of ports the memory system has. This can be done either by replicating the memory to enable more read ports or by using double pumping to enable four ports instead of two per replicate.
  
  All of a replicate’s physical access ports can be accessed concurrently.

- Partitioning memory content into one or more banks, such that each bank contains a subset of the data contained in the original memory (corresponds to the top-right box of Schematic Representation of Local Memories Showing the Relationship between Banks, Replicates, Ports, and Private Copies).
  
  The banks of a component memory can be accessed concurrently by the datapath.

- Replicating a bank to create multiple coherent replicates (corresponds to the bottom-left box of Schematic Representation of Local Memories Showing the Relationship between Banks, Replicates, Ports, and Private Copies). Each replicate in a bank contains identical data.
  
  The replicates are loaded concurrently.

- Creating private copies to increase the throughput of a loop nest if the inner loop has II > 1 (corresponds to the bottom-right box of Schematic Representation of Local Memories Showing the Relationship between Banks, Replicates, Ports, and Private Copies).
  
  Each iteration of the outer loop accesses its own private copy. Therefore, private copies might not contain the same data.

Despite the compiler’s best efforts, the component memory system can still be stallable. This might happen due to resource constraints or memory attributes defined in your source code. In that case, the compiler tries to minimize the hardware resources consumed by the arbitrated memory system.

3.3.4.2. External Memory

If the component accesses memory outside of the component, the compiler creates a hardware interface through which the datapath accesses this external memory. The interface is described using a pointer or Avalon® memory-mapped master interface as a function argument to the component. One interface is created for every pointer or memory-mapped master interface component argument.

The code snippet in Component Memory on page 32 shows an external memory described with an Avalon memory-mapped master interface and its accesses within the component.

Unlike component memory, the compiler does not define the structure of the external memory. The compiler instantiates a specialized LSU for each access site based on the type of interface and the memory access patterns.

The compiler also tries various strategies to maximize the efficient use of the available memory interface bandwidth such as eliminating unnecessary accesses and statically coalescing contiguous accesses.
4. Interface Best Practices

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

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

Demonstrating Interface Best Practices

The Intel HLS Compiler Pro Edition comes with a number of tutorials that illustrate important Intel HLS Compiler concepts and demonstrate good coding practices.

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

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;quartus_installdir&gt;/hls/examples/tutorials</td>
<td>You can find these tutorials in the following location on your Intel Quartus Prime system:</td>
</tr>
<tr>
<td>interfaces/overview</td>
<td>Demonstrates the effects on quality-of-results (QoR) of choosing different component interfaces even when the component algorithm remains the same.</td>
</tr>
<tr>
<td>best_practices/const_global</td>
<td>Demonstrates the performance and resource utilization improvements of using const qualified global variables. Also demonstrates the type of interface created when you access global variables.</td>
</tr>
<tr>
<td>best_practices/parameter_aliasing</td>
<td>Demonstrates the use of the __restrict keyword on component arguments.</td>
</tr>
<tr>
<td>best_practices/lsu_control</td>
<td>Demonstrates the effects of controlling the type of LSUs instantiated for variable-latency Avalon Memory Mapped Master interfaces</td>
</tr>
<tr>
<td>interfaces/explicit_streams_buffer</td>
<td>Demonstrates how to use explicit stream_in and stream_out interfaces in the component and testbench.</td>
</tr>
<tr>
<td>interfaces/explicit_streams_packets_empty</td>
<td>Demonstrates how to use the usesPackets, usesEmpty, and firstSymbolInHighOrderBits stream template parameters.</td>
</tr>
<tr>
<td>interfaces/explicit_streams_packets_ready_valid</td>
<td>Demonstrates how to use the usesPackets, usesValid, and usesReady stream template parameters.</td>
</tr>
<tr>
<td>interfaces/mm_master_testbench_operators</td>
<td>Demonstrates how to invoke a component at different indices of an Avalon Memory Mapped (MM) Master (mm_master class) interface.</td>
</tr>
</tbody>
</table>

*Other names and brands may be claimed as the property of others.
4. Interface Best Practices

4.1. Choose the Right Interface for Your Component

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

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

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

The core algorithm is as follows:

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

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

Ideally, the generated component has a latency of \( N/8 \) cycles. In the examples in the following section, a value of 1024 is used for \( N \), so the ideal latency is 128 cycles (1024/8).
The following sections present variations of this example that use different interfaces. Review these sections to learn how different interfaces affect the QoR of this component.

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

### 4.1.1. Pointer Interfaces

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

Pointers in a component are implemented as Avalon Memory Mapped (Avalon-MM) master interfaces with default settings. For more details about pointer parameter interfaces, see [Intel HLS Compiler Default Interfaces in Intel High Level Synthesis Compiler Pro Edition Reference Manual](https://www.intel.com/data/literature/high-level-synthesis/)

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

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

The following diagram shows the Function View in the Graph Viewer that is generated when you compile this example. Because the loop is unrolled by a factor of 8, the diagram shows that `vector_add.B2` has 8 loads for vector `a`, 8 loads for vector `b`, and 8 stores for vector `c`. In addition, all of the loads and stores are arbitrated on the same memory, resulting in inefficient memory accesses.
The following Loop Analysis report shows that the component has an undesirably high loop initiation interval (II). The II is high because vectors $a$, $b$, and $c$ are all accessed through the same Avalon-MM Master interface. The Intel HLS Compiler Pro Edition uses stallable arbitration logic to schedule these accesses, which results in poor performance and high FPGA area use.

In addition, the compiler cannot assume there are no data dependencies between loop iterations because pointer aliasing might exist. The compiler cannot determine that vectors $a$, $b$, and $c$ do not overlap. If data dependencies exist, the Intel HLS Compiler cannot pipeline the loop iterations effectively.
Compiling the component with an Intel Quartus Prime compilation flow targeting an Intel Arria® 10 device results in the following QoR metrics, including high ALM usage, high latency, high II, and low $f_{\text{MAX}}$. All of which are undesirable properties in a component.

Table 2. QoR Metrics for a Component with a Pointer Interface

<table>
<thead>
<tr>
<th>QoR Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALMs</td>
<td>15593.5</td>
</tr>
<tr>
<td>DSPs</td>
<td>0</td>
</tr>
<tr>
<td>RAMs</td>
<td>30</td>
</tr>
<tr>
<td>$f_{\text{MAX}}$ (MHz)$^2$</td>
<td>298.6</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>24071</td>
</tr>
<tr>
<td>Initiation Interval (II) (cycles)</td>
<td>~508</td>
</tr>
</tbody>
</table>

$^1$The compilation flow used to calculate the QoR metrics used Intel Quartus Prime Pro Edition Version 17.1.

$^2$The $f_{\text{MAX}}$ measurement was calculated from a single seed.

4.1.2. Avalon Memory Mapped Master Interfaces

By default, pointers in a component are implemented as Avalon Memory Mapped (Avalon MM) master interfaces with default settings. You can mitigate poor performance from the default settings by configuring the Avalon MM master interfaces.

You can configure the Avalon MM master interface for the vector addition component example using the ihc::mm_master class as follows:
component void vector_add(
    ihc::mm_master<int, ihc::aspace<1>, ihc::dwidth<8*8*sizeof(int)>> & a,
    ihc::mm_master<int, ihc::aspace<2>, ihc::dwidth<8*8*sizeof(int)>> & b,
    ihc::mm_master<int, ihc::aspace<3>, ihc::dwidth<8*8*sizeof(int)>> & c,
    int N) {
    #pragma unroll 8
    for (int i = 0; i < N; ++i) {
        c[i] = a[i] + b[i];
    }
}

The memory interfaces for vector a, vector b, and vector c have the following attributes specified:

- The vectors are each assigned to different address spaces with the ihc::aspace attribute, and each vector receives a separate Avalon MM master interface. With the vectors assigned to different physical interfaces, the vectors can be accessed concurrently without interfering with each other, so memory arbitration is not needed.
- The width of the interfaces for the vectors is adjusted with the ihc::dwidth attribute.
- The alignment of the interfaces for the vectors is adjusted with the ihc::align attribute.

The following diagram shows the Function View in the Graph Viewer that is generated when you compile this example.
The diagram shows that \texttt{vector\_add.B2} has two loads and one store. The default Avalon MM Master settings used by the code example in \textit{Pointer Interfaces} on page 40 had 16 loads and 8 stores.

By expanding the width and alignment of the vector interfaces, the original pointer interface loads and stores were coalesced into one wide load each for vector \texttt{a} and vector \texttt{b}, and one wide store for vector \texttt{c}.
Also, the memories are stall-free because the loads and stores in this example access separate memories.

Compiling this component with an Intel Quartus Prime compilation flow targeting an Intel Arria 10 device results in the following QoR metrics:

**Table 3. QoR Metrics Comparison for Avalon MM Master Interface**

<table>
<thead>
<tr>
<th>QoR Metric</th>
<th>Pointer</th>
<th>Avalon MM Master</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALMs</td>
<td>15593.5</td>
<td>643</td>
</tr>
<tr>
<td>DSPs</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RAMs</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>$f_{\text{MAX}}$ (MHz)</td>
<td>298.6</td>
<td>472.37</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>24071</td>
<td>142</td>
</tr>
<tr>
<td>Initiation Interval (II) (cycles)</td>
<td>~508</td>
<td>1</td>
</tr>
</tbody>
</table>

1 The compilation flow used to calculate the QoR metrics used Intel Quartus Prime Pro Edition Version 17.1.

2 The $f_{\text{MAX}}$ measurement was calculated from a single seed.

All QoR metrics improved by changing the component interface to a specialized Avalon MM Master interface from a pointer interface. The latency is close to the ideal latency value of 128, and the loop initiation interval (II) is 1.

**Important:** This change to a specialized Avalon MM Master interface from a pointer interface requires the system to have three separate memories with the expected width. The initial pointer implementation requires only one system memory with a 64-bit wide data bus. If the system cannot provide the required memories, you cannot use this optimization.

### 4.1.3. Avalon Memory Mapped Slave Interfaces

Depending on your component, you can sometimes optimize the memory structure of your component by using Avalon Memory Mapped (Avalon MM) slave interfaces.

When you allocate a slave memory, you must define its size. Defining the size puts a limit on how large a value of $N$ that the component can process. In this example, the RAM size is 1024 words. This RAM size means that $N$ can have a maximal size of 1024 words.

The vector addition component example can be coded with an Avalon MM slave interface as follows:

```c
component void vector_add(  
    hls_avalon_slave_memory_argument(1024*sizeof(int)) int* a,  
    hls_avalon_slave_memory_argument(1024*sizeof(int)) int* b,  
    hls_avalon_slave_memory_argument(1024*sizeof(int)) int* c,  
    int N) {  
    #pragma unroll 8  
    for (int i = 0; i < N; ++i) {  
        c[i] = a[i] + b[i];  
    }  
}
```
The following diagram shows the Function View in the Graph Viewer that is generated when you compile this example.

**Figure 24.** Graph Viewer Function View of `vector_add` Component with Avalon MM Slave Interface

Compiling this component with an Intel Quartus Prime compilation flow targeting an Intel Arria 10 device results in the following QoR metrics:

**Table 4.** QoR Metrics Comparison for Avalon MM Slave Interface

<table>
<thead>
<tr>
<th>QoR Metric</th>
<th>Pointer</th>
<th>Avalon MM Master</th>
<th>Avalon MM Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALMs</td>
<td>15593.5</td>
<td>643</td>
<td>490.5</td>
</tr>
<tr>
<td>DSPs</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RAMs</td>
<td>30</td>
<td>0</td>
<td>48</td>
</tr>
</tbody>
</table>

*continued...*
The QoR metrics show by changing the ownership of the memory from the system to the component, the number of ALMs used by the component are reduced, as is the component latency. The \( f_{\text{MAX}} \) of the component is increased as well. The number of RAM blocks used by the component is greater because the memory is implemented in the component and not the system. The total system RAM usage (not shown) should not increase because RAM usage shifted from the system to the FPGA RAM blocks.

### 4.1.4. Avalon Streaming Interfaces

Avalon Streaming (Avalon ST) interfaces support a unidirectional flow of data, and are typically used for components that drive high-bandwidth and low-latency data.

The vector addition example can be coded with an Avalon ST interface as follows:

```c
struct int_v8 {
    int data[8];
};

component void vector_add(
    ihc::stream_in<int_v8>& a,
    ihc::stream_in<int_v8>& b,
    ihc::stream_out<int_v8>& c,
    int N) {
    for (int j = 0; j < (N/8); ++j) {
        int_v8 av = a.read();
        int_v8 bv = b.read();
        int_v8 cv;
        #pragma unroll 8
        for (int i = 0; i < 8; ++i) {
            cv.data[i] = av.data[i] + bv.data[i];
        }
        c.write(cv);
    }
}
```

An Avalon ST interface has a data bus, and ready and busy signals for handshaking. The `struct` is created to pack eight integers so that eight operations at a time can occur in parallel to provide a comparison with the examples for other interfaces. Similarly, the loop count is divided by eight.

The following diagram shows the Function View in the Graph Viewer that is generated when you compile this example.
The main difference from other versions of the example component is the absence of memory.

The streaming interfaces are stallable from the upstream sources and the downstream output. Because the interfaces are stallable, the loop initiation interval (II) is approximately 1 (instead of exactly 1). If the component does not receive any bubbles (gaps in data flow) from upstream or stall signals from downstream, then the component achieves the desired II of 1.
If you know that the stream interfaces will never stall, you can further optimize this component by taking advantage of the `usesReady` and `usesValid` stream parameters.

Compiling this component with an Intel Quartus Prime compilation flow targeting an Intel Arria 10 device results in the following QoR metrics:

<table>
<thead>
<tr>
<th>QoR Metric</th>
<th>Pointer</th>
<th>Avalon MM Master</th>
<th>Avalon MM Slave</th>
<th>Avalon ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALMs</td>
<td>15593.5</td>
<td>643</td>
<td>490.5</td>
<td>314.5</td>
</tr>
<tr>
<td>DSPs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RAMs</td>
<td>30</td>
<td>0</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>$f_{\text{MAX}}$ (MHz)(^2)</td>
<td>298.6</td>
<td>472.37</td>
<td>498.26</td>
<td>389.71</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>24071</td>
<td>142</td>
<td>139</td>
<td>134</td>
</tr>
<tr>
<td>Initiation Interval (II) (cycles)</td>
<td>~508</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\(^1\)The compilation flow used to calculate the QoR metrics used Intel Quartus Prime Pro Edition Version 17.1.

\(^2\)The $f_{\text{MAX}}$ measurement was calculated from a single seed.

Moving the `vector_add` component to an Avalon ST interface, further improved ALM usage, RAM usage, and component latency. The component II is optimal if there are no stalls from the interfaces.

### 4.1.5. Pass-by-Value Interface

For software developers accustomed to writing code that targets a CPU, passing each element in an array by value might be unintuitive because it typically results in many function calls or large parameters. However, for code that targets an FPGA device, passing array elements by value can result in smaller and simpler hardware on the FPGA device.

The vector addition example can be coded to pass the vector array elements by value as follows. A `struct` is used to pass the entire array (of 8 data elements) by value.

When you use a `struct` to pass the array, you must also do the following things:

- Define element-wise copy constructors.
- Define element-wise copy assignment operators.
- Add the `hls_register` memory attribute to all `struct` members in the definition.

```cpp
struct int_v8 {
    hls_register int data[8];
    //copy assignment operator
    int_v8 operator=(const int_v8& org) {
        #pragma unroll
        for (int i=0; i< 8; i++) {
            data[i] = org.data[i];
        }
        return *this;
    }
};
```
This component takes and processes only eight elements of vector \( a \) and vector \( b \), and returns eight elements of vector \( c \). To compute 1024 elements for the example, the component needs to be called 128 times (1024/8). While in previous examples the component contained loops that were pipelined, here the component is invoked many times, and each of the invocations are pipelined.

The following diagram shows the Function View in the Graph Viewer that is generated when you compile this example.
The latency of this component is one, and it has a loop initiation interval (II) of one.

Compiling this component with an Intel Quartus Prime compilation flow targeting an Intel Arria 10 device results in the following QoR metrics:

**Table 6. QoR Metrics Comparison for Pass-by-Value Interface**

<table>
<thead>
<tr>
<th>QoR Metric</th>
<th>Pointer</th>
<th>Avalon MM Master</th>
<th>Avalon MM Slave</th>
<th>Avalon ST</th>
<th>Pass-by-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALMs</td>
<td>15593.5</td>
<td>643</td>
<td>490.5</td>
<td>314.5</td>
<td>130</td>
</tr>
<tr>
<td>DSPs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RAMs</td>
<td>30</td>
<td>0</td>
<td>48</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_{\text{MAX}}$ (MHz)$^2$</td>
<td>298.6</td>
<td>472.37</td>
<td>498.26</td>
<td>389.71</td>
<td>581.06</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>24071</td>
<td>142</td>
<td>139</td>
<td>134</td>
<td>128</td>
</tr>
<tr>
<td>Initiation Interval (II) (cycles)</td>
<td>~508</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The compilation flow used to calculate the QoR metrics used Intel Quartus Prime Pro Edition Version 17.1.

The f$_{\text{MAX}}$ measurement was calculated from a single seed.

The QoR metrics for the vector_add component with a pass-by-value interface shows fewer ALM used, a high component f$_{\text{MAX}}$, and optimal values for latency and II. In this case, the II is the same as the component invocation interval. A new invocation of the component can be launched every clock cycle. With an initiation interval of 1, 128 component calls are processed in 128 cycles so the overall latency is 128.

### 4.2. Control LSUs For Your Variable-Latency MM Master Interfaces

Controlling the type of load-store units (LSUs) that the Intel HLS Compiler Pro Edition uses to interact with variable-latency Memory Mapped (MM) Master interfaces can help save area in your design. You might also encounter situations where disabling static coalescing of a load/store with other load/store operations benefits the performance of your design.

Review the following tutorial to learn about controlling LSUs:

<quartus_installdir>/hls/examples/tutorials/best_practices/lsu_control.

To see if you need to use LSU controls, review the High-Level Design Reports for your component, especially the Function Memory Viewer, to see if the memory access pattern (and its associated LSUs) inferred by the Intel HLS Compiler Pro Edition match your expected memory access pattern. If they do not match, consider controlling the LSU type, LSU coalescing, or both.

#### Control the Type of LSU Created

The Intel HLS Compiler Pro Edition creates either burst-coalesced LSUs or pipelined LSUs.

In general, use burst-coalesced LSUs when an LSU is expected to process many load/store requests to memory words that are consecutive. The burst-coalesced LSU attempts to "dynamically coalesce" the requests into larger bursts in order to utilize memory bandwidth more efficiently.

The pipelined LSU consumes significantly less FPGA area, but processes load/store requests individually without any coalescing. This processing is useful when your design is tight on area or when the accesses to the variable-latency MM Master interface are not necessarily consecutive.

The following code example shows both types of LSU being implemented for a variable-latency MM Master interface:

```cpp
component void
dut(mm_master<int, dwidth<128>, awidth<32>, aspace<4>, latency<0>> &Buff1,
    mm_master<int, dwidth<32>, awidth<32>, aspace<5>, latency<0>> &Buff2) {
    int Temp[SIZE];

    using pipelined = lsu<>&PIPELINED>; // Burst-Coalesced LSU
    using burst_coalesced = lsu<>&BURST_COALESCED>; // Burst-Coalesced LSU

    for (int i = 0; i<SIZE; i++) {
        Temp[i] = burst_coalesced::load(&Buff1[i]);
    }
}
```
for (int i = 0; i<SIZE; i++) {
    pipelined::store(&Buff2[i], 2*Temp[i]); // Pipelined LSU
}

Disable Static Coalescing

Static coalescing is typically beneficial because it reduces the total number of LSUs in your design by statically combining multiple load/store operations into wider load/store operations.

However, there are cases where static coalescing leads to unaligned accesses, which you might not want to occur. There are also cases where multiple loads/stores get coalesced even though you intended for only a subset of them to be operational at a time. In these cases, consider disable static coalescing for the load/store operations that you did not want to be coalesced.

For the following code example, the Intel HLS Compiler does not statically coalesce the two load operations into one wide load operation:

```c
component int
dut(mm_master<int, dwidth<256>, awidth<32>, aspace<1>, latency<0>> &Buff1, int i, bool Cond1, bool Cond2) {
    using no_coalescing = lsu<style<PIPELINED>, static_coalescing<false>>;
    int Val = 0;
    if (Cond1) {
        Val = no_coalescing::load(&Buff1[i]);
    }
    if (Cond2) {
        Val = no_coalescing::load(&Buff1[i + 1]);
    }
    return Val;
}
```

If the two load operations were coalesced, an unaligned LSU would be created, which would hurt the throughput of your component.

Related Information
Avalon Memory-Mapped Master Interfaces and Load-Store Units

4.3. Avoid Pointer Aliasing

Add a restrict type-qualifier to pointer types whenever possible. By having restrict-qualified pointers, you prevent the Intel HLS Compiler Pro Edition from creating unnecessary memory dependencies between nonconflicting read and write operations.

The restrict type-qualifier is __restrict.

Consider a loop where each iteration reads data from one array, and then it writes data to another array in the same physical memory. Without adding the restrict type-qualifier to these pointer arguments, the compiler must assume that the two arrays might overlap. Therefore, the compiler must keep the original order of memory accesses to both arrays, resulting in poor loop optimization or even failure to pipeline the loop that contains the memory accesses.

You can also use the restrict type-qualifier with Avalon memory-mapped (MM) master interfaces.
For more details, review the parameter aliasing tutorial in the following location:

<quartus_installdir>/hls/examples/tutorials/best_practices/parameter_aliasing
5. Loop Best Practices

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

The Intel HLS Compiler Pro Edition lets you know if there are any dependencies that prevent it from optimizing your loops. Try to eliminate these dependencies in your code for optimal component performance. You can also provide additional guidance to the compiler by using the available loop pragmas.

As a start, try the following techniques:

• Manually fuse adjacent loop bodies when the instructions in those loop bodies can be performed in parallel. These fused loops can be pipelined instead of being executed sequentially. Pipelining reduces the latency of your component and can reduce the FPGA area your component uses.

• Use the `#pragma loop_coalesce` directive to have the compiler attempt to collapse nested loops. Coalescing loops reduces the latency of your component and can reduce the FPGA area overhead needed for nested loops.

• If you have two loops that can execute in parallel, consider using a system of tasks. For details, see System of Tasks Best Practices on page 84.

Tutorials Demonstrating Loop Best Practices

The Intel HLS Compiler Pro Edition comes with a number of tutorials that illustrate important Intel HLS Compiler concepts and demonstrate good coding practices.

Review the following tutorials to learn about loop best practices that might apply to your design:

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>You can find these tutorials in the following location on your Intel Quartus Prime system:</td>
<td></td>
</tr>
<tr>
<td><code>&lt;quartus_installdir&gt;/hls/examples/tutorials</code></td>
<td></td>
</tr>
<tr>
<td><code>best_practices/divergent_loops</code></td>
<td>Demonstrates a source-level optimization for designs with divergent loops</td>
</tr>
<tr>
<td><code>best_practices/loop_coalesce</code></td>
<td>Demonstrates the performance and resource utilization improvements of using</td>
</tr>
<tr>
<td></td>
<td><code>loop_coalesce</code> pragma on nested loops.</td>
</tr>
<tr>
<td><code>best_practices/loop_fusion</code></td>
<td>Demonstrates the latency and resource utilization improvements of loop fusion.</td>
</tr>
<tr>
<td><code>best_practices/loop_memory_dependency</code></td>
<td>Demonstrates breaking loop-carried dependencies using the <code>ivdep</code> pragma.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop_controls/max_interleaving</td>
<td>Demonstrates a method to reduce the area utilization of a loop that meets the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The loop has an II &gt; 1</td>
</tr>
<tr>
<td></td>
<td>• The loop is contained in a pipelined loop</td>
</tr>
<tr>
<td></td>
<td>• The loop execution is serialized across the invocations of the pipelined loop</td>
</tr>
<tr>
<td>best_practices/optimize_ii_using_hls_register</td>
<td>Demonstrates how to use the hls_register attribute to reduce loop II and how to use hls_max_concurrency to improve component throughput.</td>
</tr>
<tr>
<td>best_practices/parallelize_array_operation</td>
<td>Demonstrates how to improve fMax by correcting a bottleneck that arises when performing operations on an array in a loop.</td>
</tr>
<tr>
<td>best_practices/relax_reduction_dependency</td>
<td>Demonstrates a method to reduce the II of a loop that includes a floating point accumulator, or other reduction operation that cannot be computed at high speed in a single clock cycle.</td>
</tr>
<tr>
<td>best_practices/remove_loop_carried_dependency</td>
<td>Demonstrates how to improve loop performance by removing accesses to the same variable across nested loops.</td>
</tr>
<tr>
<td>best_practices/resource_sharing_filter</td>
<td>Demonstrates the following versions of a 32-tap finite impulse response (FIR) filter design:</td>
</tr>
<tr>
<td></td>
<td>• optimized-for-throughput variant</td>
</tr>
<tr>
<td></td>
<td>• optimized-for-area variant</td>
</tr>
<tr>
<td>best_practices/triangular_loop</td>
<td>Demonstrates a method for describing triangular loop patterns with dependencies.</td>
</tr>
</tbody>
</table>

### 5.1. Reuse Hardware By Calling It In a Loop

Loops are a useful way to reuse hardware. If your component function calls another function, the called function will be the top-level component. Calling a function multiple times results in hardware duplication.

For example, the following code example results in multiple hardware copies of the function `foo` in the component `myComponent` because the function `foo` is inlined:

```c
int foo(int a)
{
    return 4 + sqrt(a) /
}
```

```c
component
void myComponent()
{
    ...
    int x =
    x += foo(0);
    x += foo(1);
    x += foo(2);
    ...
}
```

If you place the function `foo` in a loop, the hardware for `foo` can be reused for each invocation. The function is still inlined, but it is inlined only once.
component
void myComponent()
{
  ...
  int x = 0;
  #pragma unroll 1
  for (int i = 0; i < 3; i++)
  {
    x += foo(i);
  }
  ...
}

You could also use a switch/case block if you want to pass your reusable function different values that are not related to the loop induction variable i:

component
void myComponent()
{
  ...
  int x = 0;
  #pragma unroll 1
  for (int i = 0; i < 3; i++)
  {
    int val = 0;
    switch(i)
    {
    case 0:
      val = 3;
      break;
    case 1:
      val = 6;
      break;
    case 2:
      val = 1;
      break;
    }
    x += foo(val);
  }
  ...
}

You can learn more about reusing hardware and minimizing inlining by reviewing the resource sharing tutorial available in `<quartus_installdir>/hls/examples/tutorials/best_practices/resource_sharing_filter`.

5.2. Parallelize Loops

One of the main benefits of using an FPGA instead of a microprocessor is that FPGAs use a spatial compute structure. A design can use additional hardware resources in exchange for lower latency.

You can take advantage of the spatial compute structure to accelerate the loops by having multiple iterations of a loop executing concurrently. To have multiple iterations of a loop execute concurrently, unroll loops when possible and structure your loops so that dependencies between loop iterations are minimized and can be resolved within one clock cycle.

These practices show how to parallelize different iterations of the same loop. If you have two different loops that you want to parallelize, consider using a system of tasks. For details, see System of Tasks Best Practices on page 84.
5.2.1. Pipeline Loops

Pipelining is a form of parallelization where multiple iterations of a loop execute concurrently, like an assembly line.

Consider the following basic loop with three stages and three iterations. A loop stage is defined as the operations that occur in the loop within one clock cycle.

Figure 27. Basic loop with three stages and three iterations

If each stage of this loop takes one clock cycle to execute, then this loop has a latency of nine cycles.

The following figure shows the pipelining of the loop from Figure 27 on page 58.

Figure 28. Pipelined loop with three stages and four iterations

The pipelined loop has a latency of five clock cycles for three iterations (and six cycles for four iterations), but there is no area tradeoff. During the second clock cycle, Stage 1 of the pipeline loop is processing iteration 2, Stage 2 is processing iteration 1, and Stage 3 is inactive.

This loop is pipelined with a loop initiation interval (II) of 1. An II of 1 means that there is a delay of 1 clock cycle between starting each successive loop iteration.

The Intel HLS Compiler Pro Edition attempts to pipeline loops by default, and loop pipelining is not subject to the same constant iteration count constraint that loop unrolling is.

Not all loops can be pipelined as well as the loop shown in Figure 28 on page 58, particularly loops where each iteration depends on a value computed in a previous iteration.
For example, consider if Stage 1 of the loop depended on a value computed during Stage 3 of the previous loop iteration. In that case, the second (orange) iteration could not start executing until the first (blue) iteration had reached Stage 3. This type of dependency is called a \textit{loop-carried dependency}.

In this example, the loop would be pipelined with II=3. Because the II is the same as the latency of a loop iteration, the loop would not actually be pipelined at all. You can estimate the overall latency of a loop with the following equation:

\[
\text{latency}_{\text{loop}} = (\text{iterations} - 1) \times \text{II} + \text{latency}_{\text{body}}
\]

where \text{latency}_{\text{loop}} is the number of cycles the loop takes to execute and \text{latency}_{\text{body}} is the number of cycles a single loop iteration takes to execute.

The Intel HLS Compiler Pro Edition supports pipelining nested loops without unrolling inner loops. When calculating the latency of nested loops, apply this formula recursively. This recursion means that having II>1 is more problematic for inner loops than for outer loops. Therefore, algorithms that do most of their work on an inner loop with II=1 still perform well, even if their outer loops have II>1.

### 5.2.2. Unroll Loops

When a loop is unrolled, each iteration of the loop is replicated in hardware and executes simultaneously if the iterations are independent. Unrolling loops trades an increase in FPGA area use for a reduction in the latency of your component.

Consider the following basic loop with three stages and three iterations. Each stage represents the operations that occur in the loop within one clock cycle.

**Figure 29. Basic loop with three stages and three iterations**

If each stage of this loop takes one clock cycle to execute, then this loop has a latency of nine cycles.

The following figure shows the loop from Figure 29 on page 59 unrolled three times.
Three iterations of the loop can now be completed in only three clock cycles, but three times as many hardware resources are required.

You can control how the compiler unrolls a loop with the \texttt{#pragma unroll} directive, but this directive works only if the compiler knows the trip count for the loop in advance or if you specify the unroll factor. In addition to replicating the hardware, the compiler also reschedules the circuit such that each operation runs as soon as the inputs for the operation are ready.

For an example of using the \texttt{#pragma unroll} directive, see the \texttt{best_practices/resource_sharing_filter} tutorial.

### 5.2.3. Example: Loop Pipelining and Unrolling

Consider a design where you want to perform a dot-product of every column of a matrix with each other column of a matrix, and store the six results in a different upper-triangular matrix. The rest of the elements of the matrix should be set to zero.
The code might look like the following code example:

```c
#define ROWS 4
#define COLS 4

component void dut(...) {
    float a_matrix[COLS][ROWS]; // store in column-major format
    float r_matrix[ROWS][COLS]; // store in row-major format

    // setup...

    for (int i = 0; i < COLS; i++) {
        for (int j = i + 1; j < COLS; j++) {
            float dotProduct = 0;
            for (int mRow = 0; mRow < ROWS; mRow++) {
                dotProduct += a_matrix[i][mRow] * a_matrix[j][mRow];
            }
            r_matrix[i][j] = dotProduct;
        }
    }

    // continue...
}
```

You can improve the performance of this component by unrolling the loops that iterate across each entry of a particular column. If the loop operations are independent, then the compiler executes them in parallel.

Floating-point operations typically must be carried out in the same order that they are expressed in your source code to preserve numerical precision. However, you can use the `-ffp-contract=fast` compiler flag to relax the ordering of floating-point operations. With the order of floating-point operations relaxed, all of the multiplications in this loop can occur in parallel. To learn more, review the tutorial: `<quartus_installdir>/hls/examples/tutorials/best_practices/floating_point_ops`

The compiler tries to unroll loops on its own when it thinks unrolling improves performance. For example, the loop at line 14 is automatically unrolled because the loop has a constant number of iterations, and does not consume much hardware (ROWS is a constant defined at compile-time, ensuring that this loop has a fixed number of iterations).

You can improve the throughput by unrolling the j-loop at line 11, but to allow the compiler to unroll the loop, you must ensure that it has constant bounds. You can ensure constant bounds by starting the j-loop at j = 0 instead of j = i + 1. You must also add a predication statement to prevent `r_matrix` from being assigned with invalid data during iterations 0, 1, 2, ..., i of the j-loop.

```c
#define ROWS 4
#define COLS 4

component void dut(...) {
    float a_matrix[COLS][ROWS]; // store in column-major format
    float r_matrix[ROWS][COLS]; // store in row-major format

    // setup...

    #pragma unroll
    for (int j = 0; j < COLS; j++) {
        ...
    }
    #pragma unroll
```
Now the j-loop is fully unrolled. Because they do not have any dependencies, all four iterations run at the same time.

Refer to the resource_sharing_filter tutorial located at <quartus_installdir>/hls/examples/tutorials/best_practices/resource_sharing_filter for more details.

You could continue and also unroll the loop at line 10, but unrolling this loop would result in the area increasing again. By allowing the compiler to pipeline this loop instead of unrolling it, you can avoid increasing the area and pay about only four more clock cycles assuming that the i-loop only has an II of 1. If the II is not 1, the Details pane of the Loops Analysis page in the high-level design report (report.html) gives you tips on how to improve it.

The following factors are factors that can typically affect loop II:

- loop-carried dependencies
  See the tutorial at <quartus_installdir>/hls/examples/tutorials/best_practices/loop_memory_dependency
- long critical loop path
- inner loops with a loop II > 1

5.3. Construct Well-Formed Loops

A well-formed loop has an exit condition that compares against an integer bound and has a simple induction increment of one per iteration. The Intel HLS Compiler Pro Edition can analyze well-formed loops efficiently, which can help improve the performance of your component.

The following example is a well-formed loop:

```
for(int i=0; i < N; i++)
{
    //statements
}
```

Well-formed nested loops can also help maximize the performance of your component.
The following example is a well-formed nested loop structure:

```cpp
for(int i=0; i < N; i++)
{
    //statements
    for(int j=0; j < M; j++)
    {
        //statements
    }
}
```

### 5.4. Minimize Loop-Carried Dependencies

Loop-carried dependencies occur when the code in a loop iteration depends on the output of previous loop iterations. Loop-carried dependencies in your component increase loop initiation interval (II), which reduces the performance of your component.

The loop structure that follows has a loop-carried dependency because each loop iteration reads data written by the previous iteration. As a result, each read operation cannot proceed until the write operation from the previous iteration completes. The presence of loop-carried dependencies reduces the pipeline parallelism that the Intel HLS Compiler Pro Edition can achieve, which reduces component performance.

```cpp
for(int i = 1; i < N; i++)
{
}
```

The Intel HLS Compiler Pro Edition performs a static memory dependency analysis on loops to determine the extent of parallelism that it can achieve. If the Intel HLS Compiler Pro Edition cannot determine that there are no loop-carried dependencies, it assumes that loop-dependencies exist. The ability of the compiler to test for loop-carried dependencies is impeded by unknown variables at compilation time or if array accesses in your code involve complex addressing.

To avoid unnecessary loop-carried dependencies and help the compiler to better analyze your loops, follow these guidelines:

**Avoid Pointer Arithmetic**

Compiler output is suboptimal when your component accesses arrays by dereferencing pointer values derived from arithmetic operations. For example, avoid accessing an array as follows:

```cpp
for(int i = 0; i < N; i++)
{
    int t = *(A++);
    *A = t;
}
```

**Introduce Simple Array Indexes**

Some types of complex array indexes cannot be analyzed effectively, which might lead to suboptimal compiler output. Avoid the following constructs as much as possible:
- Nonconstants in array indexes.
  For example, \(A[K + i]\), where \(i\) is the loop index variable and \(K\) is an unknown variable.

- Multiple index variables in the same subscript location.
  For example, \(A[i + 2 \times j]\), where \(i\) and \(j\) are loop index variables for a double nested loop.
  The array index \(A[i][j]\) can be analyzed effectively because the index variables are in different subscripts.

- Nonlinear indexing.
  For example, \(A[i \& C]\), where \(i\) is a loop index variable and \(C\) is a nonconstant variable.

**Use Loops with Constant Bounds Whenever Possible**

The compiler can perform range analysis effectively when loops have constant bounds.

You can place an `if`-statement inside your loop to control in which iterations the loop body executes.

**Ignore Memory Dependencies**

If there are no implicit memory dependencies across loop iterations, you can use the `ivdep` pragma to tell the Intel HLS Compiler Pro Edition to ignore possible memory dependencies.

For details about how to use the `ivdep` pragma, see Loop-Carried Dependencies (ivdepPragma) in the Intel High Level Synthesis Compiler Pro Edition Reference Manual.

5.5. Avoid Complex Loop-Exit Conditions

If a loop in your component has complex exit conditions, memory accesses or complex operations might be required to evaluate the condition. Subsequent iterations of the loop cannot launch in the loop pipeline until the evaluation completes, which can decrease the overall performance of the loop.

Use the `speculated_iterations` pragma to specify how many cycles the loop exit condition can take to compute.

**Related Information**

Loop Iteration Speculation (speculated_iterationsPragma)
5.6. Convert Nested Loops into a Single Loop

To maximize performance, combine nested loops into a single loop whenever possible. The control flow for a loop adds overhead both in logic required and FPGA hardware footprint. Combining nested loops into a single loop reduces these aspects, improving the performance of your component.

The following code examples illustrate the conversion of a nested loop into a single loop:

<table>
<thead>
<tr>
<th>Nested Loop</th>
<th>Converted Single Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for (i = 0; i &lt; N; i++)</code>&lt;br&gt;<code>{</code>&lt;br&gt;<code>//statements</code>&lt;br&gt;<code>for (j = 0; j &lt; M; j++)</code>&lt;br&gt;<code>{</code>&lt;br&gt;<code>//statements</code>&lt;br&gt;<code>}</code>&lt;br&gt;<code>//statements</code>&lt;br&gt;<code>}</code></td>
<td><code>for (i = 0; i &lt; N*M; i++)</code>&lt;br&gt;<code>{</code>&lt;br&gt;<code>//statements</code>&lt;br&gt;<code>}</code></td>
</tr>
</tbody>
</table>

You can also specify the `loop_coalesce` pragma to coalesce nested loops into a single loop without affecting the loop functionality. The following simple example shows how the compiler coalesces two loops into a single loop when you specify the `loop_coalesce` pragma.

Consider a simple nested loop written as follows:

```c
#pragma loop_coalesce
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    sum[i][j] += i+j;
```

The compiler coalesces the two loops together so that they run as if they were a single loop written as follows:

```c
int i = 0;
int j = 0;
while(i < N){
    sum[i][j] += i+j;
    j++;
    if (j == M){
        j = 0;
        i++;
    }
}
```

For more information about the `loop_coalesce` pragma, see "Loop Coalescing (loop_coalesce Pragma)" in Intel High Level Synthesis Compiler Pro Edition Reference Manual.

You can also review the following tutorial: `<quartus_install_dir>/hls/examples/tutorials/best_practices/loop_coalesce`
5.7. Declare Variables in the Deepest Scope Possible

To reduce the FPGA hardware resources necessary for implementing a variable, declare the variable just before you use it in a loop. Declaring variables in the deepest scope possible minimizes data dependencies and FPGA hardware usage because the Intel HLS Compiler Pro Edition does not need to preserve the variable data across loops that do not use the variables.

Consider the following example:

```c
int a[N];
for (int i = 0; i < m; ++i)
{
    int b[N];
    for (int j = 0; j < n; ++j)
    {
        // statements
    }
}
```

The array `a` requires more resources to implement than the array `b`. To reduce hardware usage, declare array `a` outside the inner loop unless it is necessary to maintain the data through iterations of the outer loop.

**Tip:** Overwriting all values of a variable in the deepest scope possible also reduces the resources necessary to represent the variable.

5.8. Raise Loop II to Increase $f_{\text{MAX}}$

If you have a loop that does not affect the throughput of your component, you can raise the initiation interval (II) of the loop with the `ii` pragma to try and increase the $f_{\text{MAX}}$ of your design.

**Example**

Consider a case where your component has two distinct sequential pipelineable loops: an initialization loop with a low trip count and a processing loop with a high trip count and no loop-carried memory dependencies. In this case, the compiler does not know that the initialization loop has a much smaller impact on the overall throughput of your design. If possible, the compiler attempts to pipeline both loops with an II of 1.

Because the initialization loop has a loop-carried dependence, it will have a feedback path in the generated hardware. To achieve an II with such a feedback path, some clock frequency might be sacrificed. Depending on the feedback path in the main loop, the rest of your design could have run at a higher operating frequency.

If you specify `#pragma ii 2` on the initialization loop, you tell the compiler that it can be less aggressive in optimizing II for this loop. Less aggressive optimization allows the compiler to pipeline the path limiting the $f_{\text{max}}$ and could allow your overall component design to achieve a higher $f_{\text{max}}$.

The initialization loop takes longer to run with its new II. However, the decrease in the running time of the long-running loop due to higher $f_{\text{max}}$ compensates for the increased length in running time of the initialization loop.
5.9. Control Loop Interleaving

The initiation interval (II) of a loop is the statically determined number of cycles between successive iteration launches of a given loop invocation. However, the statically scheduled II may differ from the realized dynamic II when considering interleaving.

With loop interleaving, the dynamic II of a loop can be approximated by the static II of the loop divided by the degree of interleaving, that is, by the number of concurrent invocations of the loop that are in flight.

Interleaving allows the iterations of more than one invocation of a loop to execute in parallel, provided that the static II of that loop is greater than 1. By default, the maximum amount of interleaving for a loop is equal to the static II of that loop.

In the presence of interleaving, the dynamic II of a loop can be approximated by the static II of the loop divided by the degree of interleaving, that is, by the number of concurrent invocations of the loop that are in flight.

Review the following tutorial to learn more about loop interleaving and how to control it:

<quartus_installdir>/hls/examples/tutorials/loop_controls/max_interleaving.
6. fMAX Bottleneck Best Practices

The fMAX of your component can be limited for various reasons. Review these best practices to understand some common fMAX bottlenecks and how to mitigate them.

Tutorials Demonstrating fMAX Bottleneck Best Practices

The Intel HLS Compiler Pro Edition comes with a number of tutorials that illustrate important Intel HLS Compiler concepts and demonstrate good coding practices.

Review the following tutorials to learn about fMAX bottleneck best practices that might apply to your design:

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>best_practices/fpga_reg</td>
<td>Demonstrates how manually adding pipeline registers can increase fMAX</td>
</tr>
<tr>
<td>best_practices/overview</td>
<td>Demonstrates how fMAX can depend on the interface used in your component.</td>
</tr>
<tr>
<td>best_practices/parallelize_array_operation</td>
<td>Demonstrates how to improve fMAX by correcting a bottleneck that arises when performing operations on an array in a loop.</td>
</tr>
<tr>
<td>best_practices/reduce_exit_fifo_width</td>
<td>Demonstrates how to improve fMAX by reducing the width of the FIFO belonging to the exit node of a stall-free cluster</td>
</tr>
<tr>
<td>best_practices/relax_reduction_dependency</td>
<td>Demonstrates how fMAX can depend on the loop-carried feedback path.</td>
</tr>
</tbody>
</table>

You can find these tutorials in the following location on your Intel Quartus Prime system:

<quartus_installdir>/hls/examples/tutorials
7. Memory Architecture Best Practices

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

In most cases, you can optimize the memory architecture by modifying the access pattern. However, the Intel HLS Compiler Pro Edition gives you some control over the memory architecture.

Tutorials Demonstrating Memory Architecture Best Practices

The Intel HLS Compiler Pro Edition comes with a number of tutorials that illustrate important Intel HLS Compiler concepts and demonstrate good coding practices.

Review the following tutorials to learn about memory architecture best practices that might apply to your design:

Table 7. Tutorials Provided with Intel HLS Compiler Pro Edition

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes_on_mm_slave_arg</td>
<td>Demonstrates how to apply memory attributes to Avalon Memory Mapped (MM) slave arguments.</td>
</tr>
<tr>
<td>exceptions</td>
<td>Demonstrates how to use memory attributes on constants and struct members.</td>
</tr>
</tbody>
</table>
| memory_bank_configuration       | Demonstrates how to control the number of load/store ports of each memory bank and optimize your component area usage, throughput, or both by using one or more of the following memory attributes:  
  • hls_max_replicates  
  • hls_singlepump   
  • hls_doublepump   
  • hls_simple_dual_port_memory |
| memory_geometry                 | Demonstrates how to control the number of load/store ports of each memory bank and optimize your component area usage, throughput, or both by using one or more of the following memory attributes:  
  • hls_bankwidth    
  • hls_numbanks     
  • hls_bankbits     |

continued...
### 7.1. Example: Overriding a Coalesced Memory Architecture

Using memory attributes in various combinations in your code allows you to override the memory architecture that the Intel HLS Compiler Pro Edition infers for your component.

The following code examples demonstrate how you can use the following memory attributes to override coalesced memory to conserve memory blocks on your FPGA:

- `hls_bankwidth(N)`
- `hls_numbanks(N)`
- `hls_singlepump`
- `hls_max_replicates(N)`

The original code coalesces two memory accesses, resulting in a memory system that is 256 locations deep by 64 bits wide (256x64 bits) (two on-chip memory blocks):

```c
component unsigned int mem_coalesce_default(unsigned int raddr,
                                          unsigned int waddr,
                                          unsigned int wdata){
    unsigned int data[512];
    data[2*waddr] = wdata;
    data[2*waddr + 1] = wdata + 1;
    unsigned int rdata = data[2*raddr] + data[2*raddr + 1];
    return rdata;
}
```

The following images show how the 256x64 bit memory for this code sample is structured, as well how the component memory structure is shown in the high-level design report (`report.html`).

**Figure 31. Memory Structure Generated for `mem_coalesce_default`**
The modified code implements a single on-chip memory block that is 512 words deep by 32 bits wide with stallable arbitration:

```c
component unsigned int mem_coalesce_override(unsigned int raddr,
                                              unsigned int waddr,
                                              unsigned int wdata){
  //Attributes that stop memory coalescing
  hls_bankwidth(4) hls_numbanks(1)
  //Attributes that specify a single-pumped single-replicate memory
  hls_singlepump hls_max_replicates(1)
  unsigned int data[512];
  data[2*waddr] = wdata;
  data[2*waddr + 1] = wdata + 1;
  unsigned int rdata = data[2*raddr] + data[2*raddr + 1];
  return rdata;
}
```

The following images show how the 512x32 bit memory with stallable arbitration for this code sample is structured, as well how the component memory structure is shown in the high-level design report (report.html).

**Figure 32.** Memory Structure Generated for `mem_coalesce_override`
While it might appear that you save hardware area by reducing the number of RAM blocks needed for the component, the introduction of stallable arbitration increases the amount of hardware needed to implement the component. In the following table, you can compare the number ALMs and FFs required by the components.

<table>
<thead>
<tr>
<th>Quartus II Resource Utilization Summary</th>
<th>ALMs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
<th>MLABs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full design (all components)</td>
<td>11645</td>
<td>1022</td>
<td>3</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>mem_coalesce_override</td>
<td>10748</td>
<td>1054</td>
<td>1</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>mem_coalesce_default</td>
<td>90</td>
<td>208</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

7.2. Example: Overriding a Banked Memory Architecture

Using memory attributes in various combinations in your code allows you to override the memory architecture that the Intel HLS Compiler Pro Edition infers for your component.

The following code examples demonstrate how you can use the following memory attributes to override banked memory to conserve memory blocks on your FPGA:

- `hls_bankwidth(N)`
- `hls_numbanks(N)`
- `hls_singlepump`
- `hls_doublepump`
The original code creates two banks of single-pumped on-chip memory blocks that are 16 bits wide:

```c
component unsigned short mem_banked(unsigned short raddr,
unsigned short waddr,
unsigned short wdata){

    unsigned short data[1024];
    data[2*waddr] = wdata;
    data[2*waddr + 9] = wdata +1;
    unsigned short rdata = data[2*raddr] + data[2*raddr + 9];
    return rdata;
}
```

To save banked memory, you can implement one bank of double-pumped 32-bit wide on-chip memory block by adding the following attributes before the declaration of `data[1024]`. These attributes fold the two half-used memory banks into one fully-used memory bank that is double-pumped, so that it can be accessed as quickly as the two half-used memory banks.

```c
hls_bankwidth(2) hls_numbanks(1)
hls_doublepump
unsigned short data[1024];
```

Alternatively, you can avoid the double-clock requirement of the double-pumped memory by implementing one bank of single-pumped on-chip memory block by adding the following attributes before the declaration of `data[1024]`. However, in this example, these attributes add stallable arbitration to your component memories, which hurts your component performance.

```c
hls_bankwidth(2) hls_numbanks(1)
hls_singlepump
unsigned short data[1024];
```

### 7.3. Merge Memories to Reduce Area

In some cases, you can save FPGA memory blocks by merging your component memories so that they consume fewer memory blocks, reducing the FPGA area your component uses. Use the `hls_merge` attribute to force the Intel HLS Compiler Pro Edition to implement different variables in the same memory system.

When you merge memories, multiple component variables share the same memory block. You can merge memories by width (width-wise merge) or depth (depth-wise merge). You can merge memories where the data in the memories have different datatypes.
7.3.1. Example: Merging Memories Depth-Wise

Use the `hls_merge("<mem_name","depth")` attribute to force the Intel HLS Compiler Pro Edition to implement variables in the same memory system, merging their memories by depth.

All variables with the same `<mem_name>` label set in their `hls_merge` attributes are merged.

Consider the following component code:

```c
component int depth_manual(bool use_a, int raddr, int waddr, int wdata) {
    int a[128];
    int b[128];
    int rdata;
    // mutually exclusive write
    if (use_a) {
        a[waddr] = wdata;
    } else {
        b[waddr] = wdata;
    }
    // mutually exclusive read
    if (use_a) {
        rdata = a[raddr];
    } else {
        rdata = b[raddr];
    }
    return rdata;
}
```

The code instructs the Intel HLS Compiler Pro Edition to implement local memories `a` and `b` as two on-chip memory blocks, each with its own load and store instructions.
Because the load and store instructions for local memories \(a\) and \(b\) are mutually exclusive, you can merge the accesses, as shown in the example code below. Merging the memory accesses reduces the number of load and store instructions, and the number of on-chip memory blocks, by half.

```c
component int depth_manual(bool use_a, int raddr, int waddr, int wdata) {
    int a[128] hls_merge("mem","depth");
    int b[128] hls_merge("mem","depth");

    int rdata;
    // mutually exclusive write
    if (use_a) {
        a[waddr] = wdata;
    } else {
        b[waddr] = wdata;
    }

    // mutually exclusive read
    if (use_a) {
        rdata = a[raddr];
    } else {
        rdata = b[raddr];
    }

    return rdata;
}
```

Figure 35.  Depth-Wise Merge of Local Memories for Component depth_manual
There are cases where merging local memories with respect to depth might degrade memory access efficiency. Before you decide whether to merge the local memories with respect to depth, refer to the HLD report (<result>.prj/reports/report.html) to ensure that they have produced the expected memory configuration with the expected number of loads and stores instructions. In the example below, the Intel HLS Compiler Pro Edition should not merge the accesses to local memories \( a \) and \( b \) because the load and store instructions to each memory are not mutually exclusive.

```c
component int depth_manual(bool use_a, int raddr, int waddr, int wdata) {
    int a[128] hls_merge("mem","depth");
    int b[128] hls_merge("mem","depth");
    int rdata;

    // NOT mutually exclusive write
    a[waddr] = wdata;
    b[waddr] = wdata;

    // NOT mutually exclusive read
    rdata = a[raddr];
    rdata += b[raddr];
    return rdata;
}
```

In this case, the Intel HLS Compiler Pro Edition might double pump the memory system to provide enough ports for all the accesses. Otherwise, the accesses must share ports, which prevent stall-free accesses.

**Figure 36.** Local Memories for Component `depth_manual` with Non-Mutually Exclusive Accesses

7.3.2. Example: Merging Memories Width-Wise

Use the `hls_merge("<mem_name>","width")` attribute to force the Intel HLS Compiler Pro Edition to implement variables in the same memory system, merging their memories by width.

All variables with the same `<mem_name>` label set in their `hls_merge` attributes are merged.
Consider the following component code:

```c
component short width_manual (int raddr, int waddr, short wdata) {
    short a[256];
    short b[256];
    short rdata = 0;
    // Lock step write
    a[waddr] = wdata;
    b[waddr] = wdata;
    // Lock step read
    rdata += a[raddr];
    rdata += b[raddr];
    return rdata;
}
```

Figure 37. Implementation of Local Memory for Component width_manual

In this case, the Intel HLS Compiler Pro Edition can coalesce the load and store instructions to local memories `a` and `b` because their accesses are to the same address, as shown below.

```c
component short width_manual (int raddr, int waddr, short wdata) {
    short a[256] hls_merge("mem","width");
    short b[256] hls_merge("mem","width");
    short rdata = 0;
    // Lock step write
    a[waddr] = wdata;
    b[waddr] = wdata;
    // Lock step read
    rdata += a[raddr];
    rdata += b[raddr];
    return rdata;
}
```
Figure 38. Width-Wise Merge of Local Memories for Component width_manual

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

You have the option to tell the Intel HLS Compiler Pro Edition which bits in a local memory address select a memory bank and which bits select a word in that bank. You can specify the bank-selection bits with the `hls_bankbits(b_0, b_1, \ldots, b_n)` attribute.

The `(b_0, b_1, \ldots, b_n)` arguments refer to the local memory address bit positions that the Intel HLS Compiler Pro Edition should use for the bank-selection bits. Specifying the `hls_bankbits(b_0, b_1, \ldots, b_n)` attribute implies that the number of banks equals $2^{number \ of \ bank \ bits}$.

Table 8. Example of Local Memory Addresses Showing Word and Bank Selection Bits

This table of local memory addresses shows an example of how a local memory might be addressed. The memory attribute is set as `hls_bankbits(3,4)`. The memory bank selection bits (bits 3, 4) in the table bits are in bold text and the word selection bits (bits 0-2) are in italic text.

<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word 0</td>
<td>00000</td>
<td>01000</td>
<td>10000</td>
</tr>
<tr>
<td>Word 1</td>
<td>00001</td>
<td>01001</td>
<td>10001</td>
</tr>
<tr>
<td>Word 2</td>
<td>00010</td>
<td>01010</td>
<td>10010</td>
</tr>
<tr>
<td>Word 3</td>
<td>00011</td>
<td>01011</td>
<td>10011</td>
</tr>
<tr>
<td>Word 4</td>
<td>00100</td>
<td>01100</td>
<td>10100</td>
</tr>
<tr>
<td>Word 5</td>
<td>00101</td>
<td>01101</td>
<td>10101</td>
</tr>
<tr>
<td>Word 6</td>
<td>00110</td>
<td>01110</td>
<td>10110</td>
</tr>
<tr>
<td>Word 7</td>
<td>00111</td>
<td>01111</td>
<td>10111</td>
</tr>
</tbody>
</table>

Restriction: Currently, the `hls_bankbits(b_0, b_1, \ldots, b_n)` attribute supports only consecutive bank bits.

Example of Implementing the `hls_bankbits` Attribute

Consider the following example component code:
component int bank_arbitration (int raddr,
   int waddr,
   int wdata) {

  #define DIM_SIZE 4

  // Adjust memory geometry by preventing coalescing
  hls_numbanks(1)
  hls_bankwidth(sizeof(int) * DIM_SIZE)

  // Force each memory bank to have 2 ports for read/write
  hls_singlepump
  hls_max_replicates(1)

  int a[DIM_SIZE][DIM_SIZE][DIM_SIZE];
  // initialize array a...
  int result = 0;

  #pragma unroll
  for (int dim1 = 0; dim1 < DIM_SIZE; dim1++)
    #pragma unroll
    for (int dim3 = 0; dim3 < DIM_SIZE; dim3++)
      a[dim1][waddr & (DIM_SIZE-1)][dim3] = wdata;

  #pragma unroll
  for (int dim1 = 0; dim1 < DIM_SIZE; dim1++)
    #pragma unroll
    for (int dim3 = 0; dim3 < DIM_SIZE; dim3++)
      result += a[dim1][raddr & (DIM_SIZE-1)][dim3];

  return result;
}

As illustrated in the following figure, this code example generates multiple load and store instructions, and therefore multiple load/store units (LSUs) in the hardware. If the memory system is not split into multiple banks, there are fewer ports than memory access instructions, leading to arbitrated accesses. This arbitration results in a high loop initiation interval (II) value. Avoid arbitration whenever possible because it increases the FPGA area utilization of your component and impairs the performance of your component.

(1) For this example, the initial component was generated with the hls_numbanks attribute set to 1 (hls_numbanks(1)) to prevent the compiler from automatically splitting the memory into banks.
By default, the Intel HLS Compiler Pro Edition splits the memory into banks if it determines that the split is beneficial to the performance of your component. The compiler checks if any bits remain constant between accesses, and automatically infers bank-selection bits.

Now, consider the following component code:

```c
component int bank_no_arbitration (int raddr,
                               int waddr,
                               int wdata) {
    #define DIM_SIZE 4

    // Component code
}
```
// Adjust memory geometry by preventing coalescing and splitting memory
hls_bankbits(4, 5)
hls_bankwidth(sizeof(int)*DIM_SIZE)

// Force each memory bank to have 2 ports for read/write
hls_singlepump
hls_max_replicates(1)

int a[DIM_SIZE][DIM_SIZE][DIM_SIZE];

// initialize array a...
int result = 0;

#pragma unroll
for (int dim1 = 0; dim1 < DIM_SIZE; dim1++)
#pragma unroll
for (int dim3 = 0; dim3 < DIM_SIZE; dim3++)
a[dim1][waddr&(DIM_SIZE-1)][dim3] = wdata;

#pragma unroll
for (int dim1 = 0; dim1 < DIM_SIZE; dim1++)
#pragma unroll
for (int dim3 = 0; dim3 < DIM_SIZE; dim3++)
result += a[dim1][raddr&(DIM_SIZE-1)][dim3];

return result;
}

The following diagram shows that this example code creates a memory configuration with four banks. Using bits 4 and 5 as bank selection bits ensures that each load/store access is directed to its own memory bank.
In this code example, setting `hls_numbanks(4)` instead of `hls_bankbits(4,5)` results in the same memory configuration because the Intel HLS Compiler Pro Edition automatically infers the optimal bank selection bits.

In the Function Memory Viewer (inf the High-Level Design Reports), the **Address bit information** shows the bank selection bits as b6 and b7, instead of b4 and b5:

<table>
<thead>
<tr>
<th>Address bit information</th>
<th>Byte address</th>
<th>Sub-word bits</th>
<th>Word address bits</th>
<th>Bank bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b7  b6  b5  b4  b3  b2  b1  b0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>b7  b6  b5  b4</td>
<td>b7  b6</td>
</tr>
</tbody>
</table>
This difference occurs because the address bits reported in the Function Memory Viewer are based on byte addresses and not element addresses. Because every element in array \( a \) is four bytes in size, bits \( b_4 \) and \( b_5 \) in element address bits correspond to bits \( b_6 \) and \( b_7 \) in byte addressing.
8. System of Tasks Best Practices

Using a system of HLS tasks in your component enables a variety of design structures that you can implement.

Common uses for a system of tasks include the following cases:

- Executing multiple loops in parallel
- Sharing an expensive compute block
- Designing your HLS system hierarchically and testing it in the Intel HLS Compiler Pro Edition simulation environment.

After you implement systems of tasks, you might want to balance the capacity of your task functions. For details, review the advice in Balancing Capacity in a System of Tasks on page 85.

8.1. Executing Multiple Loops in Parallel

By using HLS tasks, you can run sequential loops in a pipelined manner within the context of the loop nest.

For example, in the following code sample, the first and second loops can be executing different invocations of the component `foo()` if the invocations can be pipelined by the Intel HLS Compiler Pro Edition:

```c
component void foo() {
    // first loop
    for (int i = 0; i < n; i++) {
        // Do something
    }
    // second loop
    for (int i = 0; i < m; i++) {
        // Do something else
    }
}
```

However, the same invocation of the component `foo()` cannot execute the two loops in parallel. System of tasks provides a way to achieve this by moving the loops into asynchronous tasks. With the first loop in an asynchronous task, the second loop can run concurrently with the first loop.

```c
void first_loop() {
    for (int i = 0; i < n; i++) {
        // Do something
    }
}

void second_loop() {
    for (int i = 0; i < m; i++) {
        // Do something else
    }
}
```
component void foo() {
    ihc::launch<first_loop>();
    ihc::launch<second_loop>();
    ihc::collect<first_loop>();
    ihc::collect<second_loop>();
}

Review the tutorial `<quartus_installdir>/hls/examples/tutorials/system_of_tasks/parallel_loop` to learn more about how to run multiple loops in parallel.

### 8.2. Sharing an Expensive Compute Block

With a system of tasks, you can share hardware resources at a function level. A component or another HLS task can invoke an HLS task multiple times. All `ihc::launch` and `ihc::collect` calls to the same function share the same hardware.

To allow for calls from multiple places to a task, the Intel HLS Compiler Pro Edition generates arbitration logic to the called task function. This arbitration logic can increase the area utilization of the component. However, if the shared logic is large, the trade-off can help you save FPGA resources. The savings can be especially noticed when your component has a large compute block that is not always active.

Review the tutorial `<quartus_installdir>/hls/examples/tutorials/system_of_tasks/resource_sharing` to see a simple example of how to share a compute block in component.

### 8.3. Implementing a Hierarchical Design

When you use a system of tasks, you can implement your design hierarchically, which allows for bottom-up design.

If you do not use a system of tasks, function calls in your HLS component are in-lined and optimized together with the calling code, which can be detrimental in some situations. Use a system of tasks to prevent smaller blocks of your design from being affected by the rest of the system.

The hierarchical design pattern implemented by using a system of tasks can give you the following benefits:

- Modularity similar to what a hardware description language (HDL) might provide
- Unpipelineable or poorly pipelined loops can be isolated so that they do not affect an entire loop nest.

### 8.4. Balancing Capacity in a System of Tasks

If your component contains parallel task paths with different latencies, you might experience poor performance, and in some cases, deadlock.
Typically, these performance issues are caused by a lack of capacity in the datapath of the functions calling task function using the \texttt{ihc::launch} and \texttt{ihc::collect} calls. You can improve system throughput in these cases by adding a buffer to the explicit streams to account for the latency of the task functions.

Review the following tutorials to learn more about avoiding potential performance issues in a component that uses a system of tasks:

- `<quartus_installdir>/hls/examples/tutorials/system_of_tasks/balancing_pipeline_latency`
- `<quartus_installdir>/hls/examples/tutorials/system_of_tasks/balancing_loop_delay`

The Intel HLS Compiler Pro Edition emulator models the size of the buffer attached to a stream. However, the emulator does not fully account for hardware latencies, and it might exhibit different behavior between simulation and emulation in these cases.

In addition to the techniques outlined in the tutorials, follow the practices that follow to try to maximize the data throughput of your design.

### 8.4.1. Enable the Intel HLS Compiler to Infer Data Path Buffer Capacity Requirements

In many situations, the Intel HLS Compiler can add buffer capacity automatically to the data path in a system of tasks design to achieve maximum throughput for your design. Follow a few best practices to help the Intel HLS Compiler effectively add data path buffer capacity to your design when needed.

As an example, consider the following design that runs two independent tasks. This kind of structure can be generated by code like the following example:

```cpp
class foo {
  // Parse/compute data for tasks
  ihc::launch<task1>(data1);
  ihc::launch<task2>(data2);
  auto r1 = ihc::collect<task1>();
  auto r2 = ihc::collect<task2>();
  // Usage of r1, r2
}
```

The following diagram shows the state of the system of tasks at the start of the third invocation of the component, and the location of data in the overall pipeline from previous invocations.
Figure 41. **Data Flow of Multiple Component Invocations Through a System of Tasks**

The circles represent pipelined stages of the component, while the number indicate the location of data from different invocations of component `foo`. This digram shows three invocations of the component underway.

In this diagram, **Entry** represents the two independent launch calls, and the **Exit** represents the two independent collect calls.

**Entry** provides work to both tasks only if both tasks can take in data (that is, both task have available buffer capacity). Similarly, **Exit** consumes the results only when both results are available.

If **Task1** and **Task2** have the same number of pipeline stages, then the data path performs at full throughput. Some data path buffer capacity is needed in the caller function to ensure that the caller can continue issuing launch calls while the collect calls wait for the task functions to complete. The compiler adds this data path buffer capacity automatically.

If the two tasks have different pipeline depths, then the design encounters a bottleneck because the task with the smaller pipeline depth lacks the buffer capacity to store finished results while waiting for the other task to finish. In this case, you can add buffer capacity to either launch or the collect call of the task with the smaller pipeline depth. For details about adding launch/collect buffer capacity, see *Explicitly Add Buffer Capacity to Your Design When Needed* on page 88.

The Intel HLS Compiler tries to balance data path buffer capacity automatically, but it can only add data path capacity automatically when your design follows certain practices.

Use the following best practices to obtain the maximum throughput for your system of tasks design:
A component or task function should do one of the following things:

— Do all of the work by itself without launching other tasks.
— Act as an orchestrator for issuing `ihc::launch` or `ihc::collect` calls and do none of the work.

If throughput is a priority for your design, avoid using multiple `ihc::launch` or `ihc::collect` calls to the same task function unless you are reusing the calls to the function by iterating in a loop.

Keep `ihc::launch` and `ihc::collect` calls to the same task function within the same block.

Review the block structure of your design with the Graph Viewer in the High-Level Design Reports to confirm that your calls are in the same block.

Avoid guarding your `ihc::launch` and `ihc::collect` calls with an `if`-condition.

If you are guarding your `ihc::launch` and `ihc::collect` calls with an `if`-condition, use the same `if`-condition for both the `ihc::launch` and `ihc::collect` calls.

### 8.4.2. Explicitly Add Buffer Capacity to Your Design When Needed

When the Intel HLS Compiler cannot infer the optimal capacity requirements, you can explicitly add buffer capacity to your design by specifying a value for the `capacity` parameter of the `ihc::launch` and `ihc::collect` functions.

**Adding Capacity When Launching Task Functions**

Consider specifying the `capacity` parameter of the `ihc::launch` call if you see stall patterns in your simulation waveforms that indicate an imbalance between the following things:

— Any back-pressure introduced by the task function
— How often the caller launches the task function

Figure 41 on page 87 show the block diagram of such a design.

Setting the `capacity` parameter in an `ihc::launch` call inserts a buffer that allows the caller to push work onto the task function which is then free to pull work off that queue when it can.

**Adding Capacity When Collecting Task Functions**

Consider specifying the `capacity` parameter of the `ihc::collect` call if you see stall patterns in your design waveforms that indicate a difference in the following things:

— The cadence of data production in the task function
— The cadence reading that data by the caller function

Setting the `capacity` parameter in an `ihc::collect` call inserts a buffer that can hold the return values as they are computed by the task function. The caller function is free to pull the return values from the buffer at a convenient later time without causing backpressure on the task function.
9. Datatype Best Practices

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

After you optimize the algorithm bottlenecks of your design, you can fine-tune some datatypes in your component by using arbitrary precision datatypes to shrink data widths, which reduces FPGA area utilization. The Intel HLS Compiler Pro Edition provides debug functionality so that you can easily detect overflows in arbitrary precision datatypes.

Because C++ automatically promotes smaller datatypes such as `short` or `char` to 32 bits for operations such as addition or bit-shifting, you must use the arbitrary precision datatypes if you want to create narrow datapaths in your component.

Tutorials Demonstrating Datatype Best Practices

The Intel HLS Compiler Pro Edition comes with a number of tutorials that illustrate important Intel HLS Compiler concepts and demonstrate good coding practices.

Review the following tutorials to learn about datatype best practices that might apply to your design:

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;quartus_installdir&gt;/hls/examples/tutorials</code></td>
<td>You can find these tutorials in the following location on your Intel Quartus Prime system:</td>
</tr>
<tr>
<td><code>best_practices/ac_datatypes</code></td>
<td>Demonstrates the effect of using <code>ac_int</code> datatype instead of <code>int</code> datatype.</td>
</tr>
<tr>
<td><code>ac_datatypes/ac_fixed_constructor</code></td>
<td>Demonstrates the use of the <code>ac_fixed</code> constructor where you can get a better QoR by using minor variations in coding style.</td>
</tr>
<tr>
<td><code>ac_datatypes/ac_int_basic_ops</code></td>
<td>Demonstrates the operators available for the <code>ac_int</code> class.</td>
</tr>
<tr>
<td><code>ac_datatypes/ac_int_overflow</code></td>
<td>Demonstrates the usage of the <code>DEBUG_AC_INT_WARNING</code> and <code>DEBUG_AC_INT_ERROR</code> keywords to help detect overflow during emulation runtime.</td>
</tr>
<tr>
<td><code>best_practices/single_vs_double_precision_mat</code></td>
<td>Demonstrates the effect of using single precision literals and functions instead of double precision literals and functions.</td>
</tr>
<tr>
<td><code>ac_datatypes/ac_fixed_constructor</code></td>
<td>Demonstrates the use of the <code>ac_fixed</code> math library functions.</td>
</tr>
</tbody>
</table>

continued...
9.1. Avoid Implicit Data Type Conversions

Compile your component code with the -Wconversion compiler option, especially if your component uses floating point variables.

Using this option helps you avoid inadvertently having conversions between double-precision and single-precision values when double-precision variables are not needed. In FPGAs, using double-precision variables can negatively affect the data transfer rate, the latency, and resource utilization of your component.

Additionally, constants are treated as signed int or signed double. If you want efficient operations with narrower constants, cast constants to other, narrower data types like ac_int<> or float.

If you use the Algorithmic C (AC) arbitrary precision datatypes, pay attention to the type propagation rules.

9.2. Avoid Negative Bit Shifts When Using the ac_int Datatype

The ac_int datatype differs from other languages, including C and Verilog, in bit shifting. By default, if the shift amount is of a signed datatype ac_int allows negative shifts.

In hardware, this negative shift results in the implementation of both a left shifter and a right shifter. The following code example shows a shift amount that is a signed datatype.

```c
int14 shift_left(int14 a, int14 b) {
    return (a << b);
}
```

If you know that the shift is always in one direction, to implement an efficient shift operator, declare the shift amount as an unsigned datatype as follows:

```c
int14 efficient_left_only_shift(int14 a, uint14 b) {
    return (a << b);
}
```
10. Advanced Troubleshooting

As you develop components with the Intel HLS Compiler Pro Edition, you might encounter issues whose solution is unclear. The issues typically fall into the following categories:

- Your component behaves differently in simulation and emulation.
- Your component has unexpectedly poor performance, resource utilization, or both.

10.1. Component Fails Only In Simulation

Discrepancies between the results of compiling your component in emulation (-march=x86-64) mode or simulation (-march=FPGA_name_or_part_no) mode are typically caused by relying on undefined or implementation-defined behavior in your component or testbench. However, there are some common cases where the discrepancies are caused by something else.

Comparing Floating Point Results

Use an epsilon when comparing floating point value results in the testbench. Floating points results from the RTL hardware are different from the x86 emulation flow.

Using #pragma ivdep to Ignore Memory Dependencies

The #pragma ivdep compiler pragma can cause functional incorrectness in your component if your component has a memory dependency that you attempted to ignore with the pragma. You can try to use the safelen modifier to control how many memory accesses that you can permit before a memory dependency occurs.


To see an example of using the ivdep pragma, review the tutorial in <quartus_installdir>/hls/examples/tutorials/best_practices/loop_memory_dependency.

Check for Uninitialized Variables

Many coding practices can result in behavior that is undefined by the C++ specification. Sometimes this undefined behavior works one way in emulation and a different way in simulation.

A common example of this situation occurs when your design reads from uninitialized variables, especially uninitialized struct variables.
Check your code for uninitialized values with the -Wuninitialized compiler flag, or debug your emulation testbench with the valgrind debugging tool. The -Wuninitialized compiler flag does not show uninitialized struct variables.

You can also check for misbehaving variables by using one or more stream interfaces as debug streams. You can add one or more ihc::stream_out interfaces to your component to have the component write out its internal state variables as it executes. By comparing the output of the emulation flow and the simulation flow, you can see where the RTL behavior diverges from the emulator behavior.

Non-blocking Stream Accesses

The emulation model of tryRead() is not cycle-accurate, so the behavior of tryRead() might differ between emulation and simulation.

If you have a non-blocking stream access (for example, tryRead()) from a stream with a FIFO (that is, the ihc::depth<> template parameter), then the first few iterations of tryRead() might return false in simulation, but return true in emulation.

In this case, invoke your component a few extra times from the testbench to guarantee that it consumes all data in the stream. These extra invocations should not cause functional problems because tryRead() returns false.

10.2. Component Gets Poor Quality of Results

While there are many reasons why your design achieves a poor quality of results (QoR), bad memory configurations are often an important factor. Review the Function Memory Viewer report in the High Level Design Reports, and look for stallable arbitration nodes and unexpected RAM utilization.

The information in this section describes some common sources of stallable arbitration nodes or excess RAM utilization.

Component Uses More FPGA Resource Than Expected

By default, the Intel HLS Compiler Pro Edition tries to optimize your component for the best throughput by trying to maximize the maximum operating frequency (f_{\text{MAX}}).

A way to reduce area consumption is to relax the f_{\text{MAX}} requirements by setting a target f_{\text{MAX}} value with the --clock i++ command option or the hls_scheduler_target_fmax_mhz component attribute. The HLS compiler can often achieve a higher f_{\text{MAX}} than you specify, so when you set a target f_{\text{MAX}} to a lower value than you need, your design might still achieve an acceptable f_{\text{MAX}} value, and a design that consumes less area.

To learn more about the behavior of f_{\text{MAX}} target value control see the following tutorial: <quartus_install_dir>/hls/examples/tutorials/best_practices/set_component_target_fmax

Incorrect Bank Bits

If you access parts of an array in parallel (either a single- or multidimensional array), you might need to configure the memory bank selection bits.
See Memory Architecture Best Practices on page 69 for details about how to configure efficient memory systems.

**Conditional Operator Accessing Two Different Arrays of struct Variables**

In some cases, if you try to access different arrays of struct variables with a conditional operator, the Intel HLS Compiler Pro Edition merges the arrays into the same RAM block. You might see stallable arbitration in the Function Memory Viewer because there are not enough Load/Store site on the memory system.

For example, the following code examples show an array of struct variables, a conditional operator that results in stallable arbitration, and a workaround that avoids stallable arbitration.

```c
struct MyStruct {
    float a;
    float b;
};
MyStruct array1[64];
MyStruct array2[64];

The following conditional operator that uses these arrays of struct variables causes stallable arbitration:

MyStruct value = (shouldChooseArray1) ? array1[idx] : array2[idx];

You can avoid the stallable arbitration that the conditional operator causes here by removing the operator and using an explicit if statement instead.

```c
MyStruct value;
if (shouldChooseArray1) {
    value = array1[idx];
} else {
    value = array2[idx];
}
```

**Cluster Logic**

Your design might consume more RAM blocks than you expect, especially if you store many array variables in large registers.

You can use the `hls_use_stall_enable_clusters` component attribute to prevent the compiler from inserting stall-free cluster exit FIFOs.

The Area Analysis of System report in the high-level design report (report.html) can help find this issue.
The three matrices are stored intentionally in RAM blocks, but the RAM blocks for the matrices account for less than half of the RAM blocks consumed by the component.

If you look further down the report, you might see that many RAM blocks are consumed by Cluster logic or State variable. You might also see that some of your array values that you intended to be stored in registers were instead stored in large numbers of RAM blocks.

Notice the number of RAM blocks that are consumed by Cluster Logic and State.
In some cases, you can reduce this RAM block usage by with the following techniques:

- Pipeline loops instead of unrolling them.
- Storing local variables in local RAM blocks (`hls_memory` memory attribute) instead of large registers (`hls_register` memory attribute).

## Archives

<table>
<thead>
<tr>
<th>Intel HLS Compiler Version</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.3</td>
<td>Intel HLS Compiler Best Practices Guide</td>
</tr>
<tr>
<td>19.2</td>
<td>Intel HLS Compiler Best Practices Guide</td>
</tr>
<tr>
<td>19.1</td>
<td>Intel HLS Compiler Best Practices Guide</td>
</tr>
<tr>
<td>18.1.1</td>
<td>Intel HLS Compiler Best Practices Guide</td>
</tr>
<tr>
<td>18.1</td>
<td>Intel HLS Compiler Best Practices Guide</td>
</tr>
<tr>
<td>18.0</td>
<td>Intel HLS Compiler Best Practices Guide</td>
</tr>
<tr>
<td>17.1.1</td>
<td>Intel HLS Compiler Best Practices Guide</td>
</tr>
<tr>
<td>17.1</td>
<td>Intel HLS Compiler Best Practices Guide</td>
</tr>
</tbody>
</table>

|------------------|---------------------------------------|---------|
| 2020.09.28       | 20.3                                  | • Added FPGA Concepts on page 7.  
• Revised Pass-by-Value Interface on page 49. This revision includes added requirements you must meet when using a `struct` to pass an array to your component and an updated code example showing how to implement these requirements.  
• Added Control Loop Interleaving on page 67. |
| 2020.06.22       | 20.2                                  | • Added information about capacity balancing in a system of tasks:  
— Balancing Capacity in a System of Tasks on page 85 (formerly Avoiding Potential Performance Pitfalls)  
— Enable the Intel HLS Compiler to Infer Data Path Buffer Capacity Requirements on page 86  
— Explicitly Add Buffer Capacity to Your Design When Needed on page 88  
• Updated Loop Best Practices on page 55 to add `remove_loop_carried_dependency` tutorial to list of loop best practices tutorials.  
• Added Raise Loop II to Increase fMAX on page 66. This information was available previously in the Intel HLS Compiler Reference Manual in the Loop Initiation Interval (`ii` Pragma) section. |
| 2020.04.13       | 20.1                                  | • Added new tutorials to Loop Best Practices on page 55.  
• Added new tutorials to Interface Best Practices on page 38.  
• Updated Executing Multiple Loops in Parallel on page 84 to reflect new syntax of `ihc::launch` and `ihc::collect` functions. |
| 2020.01.27       | 19.4                                  | • Corrected the spelling of the `-ffp-contract=fast` command option in Example: Loop Pipelining and Unrolling on page 60. |
• Added information to Example: Specifying Bank-Selection Bits for Local Memory Addresses on page 78 to explain the difference between the element-address bank-selection bits selected with the `hls_bankbits` attribute and the byte-address bank-selection bits reported in the Function Memory Viewer in the High-Level Design Reports.  
• References to the Component Viewer have been replaced with references to the Function View of the Graph Viewer.  
• Reference to the Component Memory Viewer have been replaced with references to the Function Memory Viewer. |

## Document Revision History for Intel HLS Compiler Best Practices Guide

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2019.09.30       | 19.3                        | • Added Control LSUs For Your Variable-Latency MM Master Interfaces on page 52.  
                      • Updated Memory Architecture Best Practices on page 69 to list updated and improved tutorials and new memory attributes.  
                      • Split memory architecture examples for overriding coalesced memory architectures and overriding banked memory architectures into the following sections:  
                        - Example: Overriding a Coalesced Memory Architecture on page 70  
                        - Example: Overriding a Coalesced Memory Architecture  
                        - Example: Specifying Bank-Selection Bits for Local Memory Addresses on page 78  
                        - Example: Specifying Bank-Selection Bits for Local Memory Addresses |
| 2019.07.01       | 19.2                        | • Maintenance release. |
| 2019.04.01       | 19.1                        | • Added new chapter to cover best practices when using HLS tasks in System of Tasks Best Practices on page 84.  
                      • Moved some content from Loop Best Practices on page 55 into a new section called Reuse Hardware By Calling It In a Loop on page 56.  
                      • Revised Component Uses More FPGA Resource Than Expected on page 92 to include information about the hls_scheduler_target_fmax_mhz component attribute. |
| 2018.12.24       | 18.1                        | • Updated to Loop Best Practices on page 55 to include information about function inlining in components and using loops to minimize the resulting hardware duplication. |
| 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.  
                      • The --promote-integers flag and the best_practices/integer_promotion tutorial are no longer supported in Pro Edition because integer promotion is now done by default. References to these items were adjusted to indicate that they apply to Standard Edition only in the following topics:  
                        — Component Fails Only In Simulation on page 91  
                        — Datatype Best Practices on page 89 |
| 2018.07.02       | 18.0                        | • Added a new chapter, Advanced Troubleshooting on page 91, to help you troubleshoot when your component behaves differently in simulation and emulation, and when your component has unexpectedly poor performance, resource utilization, or both.  

*continued...*
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 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](image) Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Pro Edition. |
|                   |                            | ![STD](image) Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Standard Edition. |
|                   |                            | • Added `best_practices/loop_coalesce` to the list of tutorials in Loop Best Practices on page 55. |
|                   |                            | • Added `interfaces/explicit_streams_packets_ready_empty` to list of tutorials in Interface Best Practices on page 38. |
|                   |                            | • Revised Example: Specifying Bank-Selection Bits for Local Memory Addresses on page 78 with improved descriptions and new graphics that reflect what you would see in the high-level design reports (report.html) for the example component. |
|                   |                            | • Updated Example: Overriding a Coalesced Memory Architecture on page 70 with new images to show the memory structures as well as how the FPGA resource usage differs between the two components. |
| 2017.12.22        | 17.1.1                     | • Added Choose the Right Interface for Your Component on page 39 section to show how changing your component interface affects your component QoR even when the algorithm stays the same. |
|                   |                            | • Added interface overview tutorial to the list of tutorials in Interface Best Practices on page 38. |
| 2017.11.06        | 17.1                       | Initial release. |