

Intel[®] High Level Synthesis Compiler

Best Practices Guide

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



[Subscribe](#)

[Send Feedback](#)

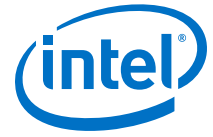
UG-20107 | 2019.04.01

Latest document on the web: [PDF](#) | [HTML](#)

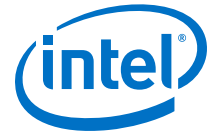


Contents

- 1. Intel® HLS Compiler Best Practices Guide..... 4**
- 2. Best Practices for Coding and Compiling Your Component..... 5**
- 3. Interface Best Practices..... 6**
 - 3.1. Choose the Right Interface for Your Component..... 7
 - 3.1.1. Pointer Interfaces..... 8
 - 3.1.2. Avalon Memory Mapped Master Interfaces..... 10
 - 3.1.3. Avalon Memory Mapped Slave Interfaces..... 13
 - 3.1.4. Avalon Streaming Interfaces..... 15
 - 3.1.5. Pass-by-Value Interface..... 17
 - 3.2. Avoid Pointer Aliasing..... 19
- 4. Loop Best Practices..... 20**
 - 4.1. Reuse Hardware By Calling It In a Loop..... 21
 - 4.2. Parallelize Loops..... 22
 - 4.2.1. Pipeline Loops..... 22
 - 4.2.2. Unroll Loops..... 24
 - 4.2.3. Example: Loop Pipelining and Unrolling..... 24
 - 4.3. Construct Well-Formed Loops..... 27
 - 4.4. Minimize Loop-Carried Dependencies..... 27
 - 4.5. Avoid Complex Loop-Exit Conditions..... 28
 - 4.6. Convert Nested Loops into a Single Loop..... 29
 - 4.7. Declare Variables in the Deepest Scope Possible..... 29
- 5. Memory Architecture Best Practices..... 31**
 - 5.1. Example: Overriding a Coalesced Memory Architecture..... 31
 - 5.2. Example: Overriding a Banked Memory Architecture..... 33
 - 5.3. Merge Memories to Reduce Area..... 34
 - 5.3.1. Example: Merging Memories Depth-Wise..... 35
 - 5.3.2. Example: Merging Memories Width-Wise..... 37
 - 5.4. Example: Specifying Bank-Selection Bits for Local Memory Addresses..... 39
- 6. PRO System of Tasks..... 45**
 - 6.1. PRO Executing Multiple Loops in Parallel..... 45
 - 6.2. PRO Sharing an Expensive Compute Block..... 46
 - 6.3. PRO Implementing a Hierarchical Design..... 46
 - 6.4. PRO Avoiding Potential Performance Pitfalls..... 46
- 7. Datatype Best Practices..... 48**
 - 7.1. Avoid Implicit Data Type Conversions..... 49
 - 7.2. Avoid Negative Bit Shifts When Using the ac_int Datatype..... 49
- 8. Advanced Troubleshooting..... 50**
 - 8.1. Component Fails Only In Cosimulation 50
 - 8.2. Component Gets Bad Quality of Results..... 51



9. Document Revision History for Intel HLS Compiler Best Practices Guide..... 58



1. Intel® HLS Compiler Best Practices Guide

The *Intel® HLS Compiler 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.

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

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

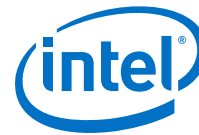
In this publication, `<quartus_installdir>` refers to the location where you installed Intel Quartus Prime Design Suite. The Intel High Level Synthesis (HLS) Compiler is installed as part of your Intel Quartus Prime Design Suite installation.

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

- PRO**

<i>Windows</i>	<code>C:\intelFPGA_pro\19.1</code>
<i>Linux</i>	<code>/home/<username>/intelFPGA_pro/19.1</code>
- STD**

<i>Windows</i>	<code>C:\intelFPGA_standard\19.1</code>
<i>Linux</i>	<code>/home/<username>/intelFPGA_standard/19.1</code>



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 examples designs and tutorials provided with the Intel High Level Synthesis (HLS) Compiler to see examples of how some of these techniques can be implemented.

- [Interface Best Practices](#) on page 6

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

- [Loop Best Practices](#) on page 20

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

- [Memory Architecture Best Practices](#) on page 31

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

- [Datatype Best Practices](#) on page 48

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

- [Alternative Algorithms](#)

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

3. Interface Best Practices

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

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

Demonstrating Interface Best Practices

The Intel HLS Compiler comes with a number of tutorials that give you working examples to review and run so that you can see good coding practices as well as demonstrating important concepts.

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

Tutorial	Description
You can find these tutorials in the following location on your Intel Quartus Prime system: <pre data-bbox="256 1073 690 1098"><quartus_installdir>/hls/examples/tutorials</pre>	
interfaces/overview	Demonstrates the effects on quality-of-results (QoR) of choosing different component interfaces even when the component algorithm remains the same.
best_practices/ parameter_aliasing	PRO Demonstrates the use of the <code>__restrict</code> keyword on component arguments STD Demonstrates the use of the <code>restrict</code> keyword on component arguments.
interfaces/ explicit_streams_buffer	Demonstrates how to use explicit <code>stream_in</code> and <code>stream_out</code> interfaces in the component and testbench.
PRO interfaces/ explicit_streams_packets_read_y_empty	Demonstrates how to use the <code>usesPackets</code> , <code>usesEmpty</code> , and <code>firstSymbolInHighOrderBits</code> stream template parameters.
interfaces/ explicit_streams_packets_read_y_valid	Demonstrates how to use the <code>usesPackets</code> , <code>usesValid</code> , and <code>usesReady</code> stream template parameters.
PRO interfaces/ explicit_streams_ready_latency	Demonstrates how to achieve a better loop initiation interval (II) with stream write using the <code>readyLatency</code> stream template parameter.
<i>continued...</i>	



Tutorial	Description
<code>interfaces/mm_master_testbench_operators</code>	Demonstrates how to invoke a component at different indices of an Avalon Memory Mapped (MM) Master (<code>mm_master</code> class) interface.
<code>interfaces/mm_slaves</code>	Demonstrates how to create Avalon-MM Slave interfaces (slave registers and slave memories).
<code>interfaces/multiple_stream_call_sites</code>	Demonstrates the benefits of using multiple stream call sites.
<code>interfaces/pointer_mm_master</code>	Demonstrates how to create Avalon-MM Master interfaces and control their parameters.
<code>interfaces/stable_arguments</code>	Demonstrates how to use the <code>stable</code> attribute for unchanging arguments to improve resource utilization.

Related Information

- [Avalon Memory-Mapped Interface Specifications](#)
- [Avalon Streaming Interface Specifications](#)

3.1. Choose the Right Interface for Your Component

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

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

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

The core algorithm is as follows:

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

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

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

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

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

3.1.1. Pointer Interfaces

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

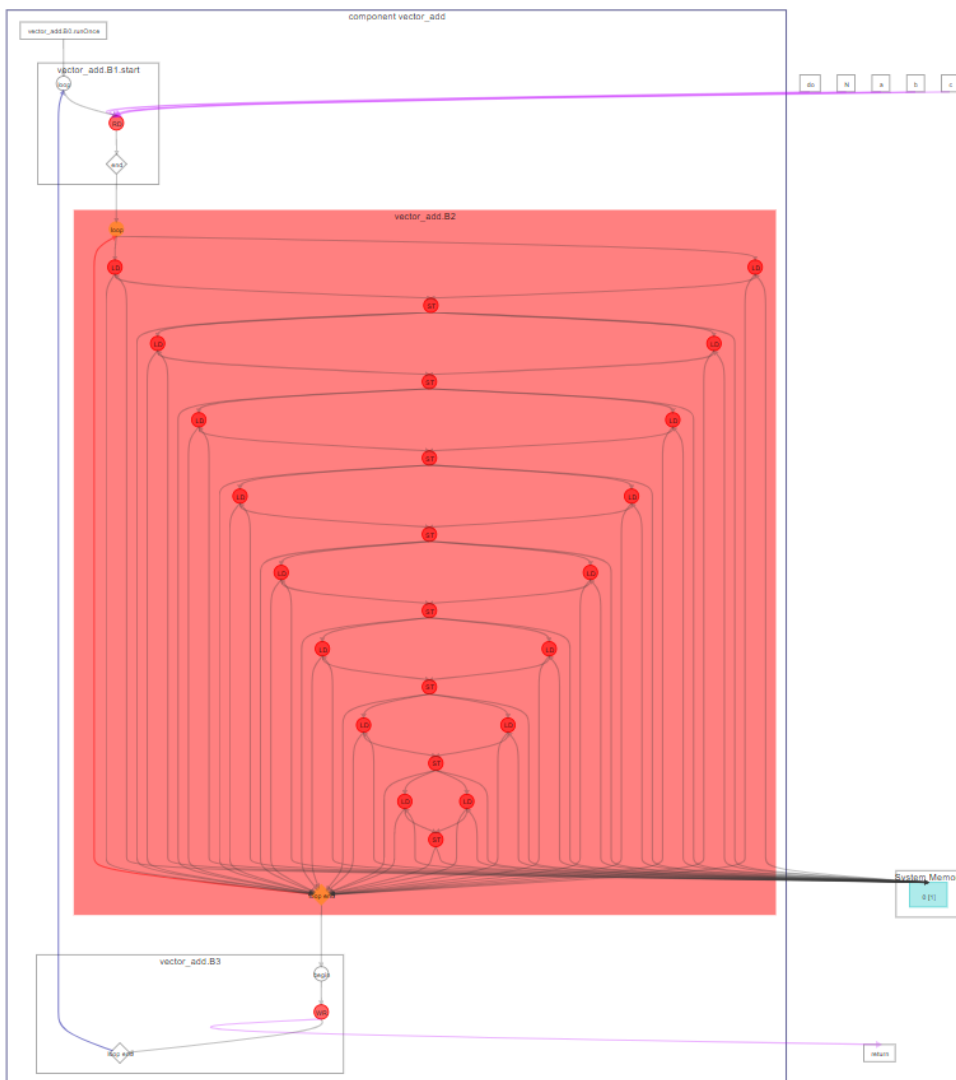
Pointers in a component are implemented as Avalon® Memory Mapped (Avalon-MM) master interfaces with default settings. For more details about pointer parameter interfaces, see [Intel HLS Compiler Default Interfaces](#) in *Intel High Level Synthesis Compiler Reference Manual*.

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

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

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

Figure 1. Component View of vector_add Component with Pointer Interfaces



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

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

Loops analysis		<input checked="" type="checkbox"/> Show fully unrolled loops		
	Pipelined	II	Bottleneck	Details
Component: vector_add (part_1_pointers.cpp:8)				Task function
vector_add.B1.start (Component invocation)	No	n/a	n/a	Out-of-order inner loop
8X Partially unrolled vector_add.B2 (part_1_pointers.cpp:10)	Yes	~508	II	Memory dependency

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_{MAX} . All of which are undesirable properties in a component.

Table 1. QoR Metrics for a Component with a Pointer Interface¹

QoR Metric	Value
ALMs	15593.5
DSPs	0
RAMs	30
f_{MAX} (MHz) ²	298.6
Latency (cycles)	24071
Initiation Interval (II) (cycles)	~508

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

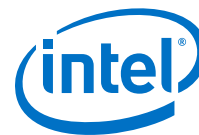
²The f_{MAX} measurement was calculated from a single seed.

3.1.2. 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)>,
    ihc::align<8*sizeof(int)> >& a,
```



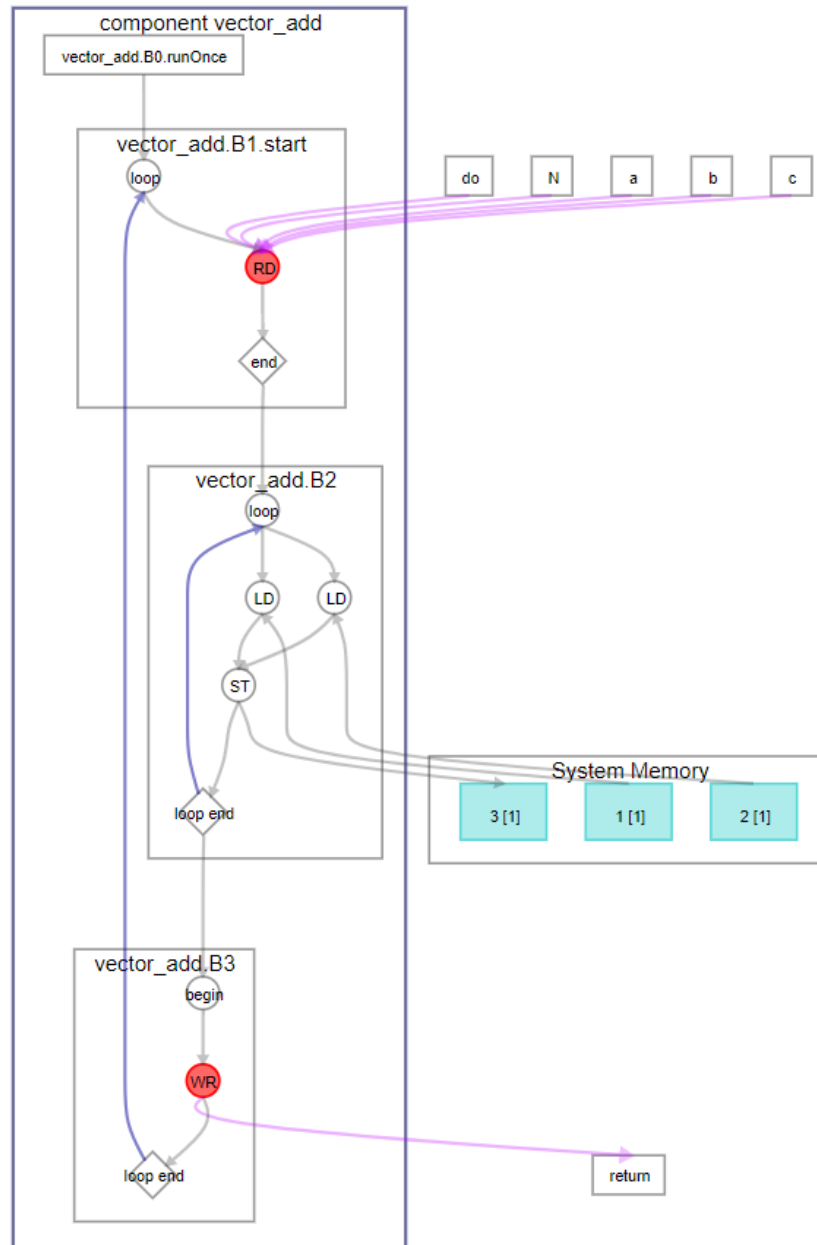
```
ihc::mm_master<int, ihc::aspace<2>, ihc::dwidth<8*8*sizeof(int)>,  
ihc::align<8*sizeof(int)> >& b,  
  
ihc::mm_master<int, ihc::aspace<3>, ihc::dwidth<8*8*sizeof(int)>,  
ihc::align<8*sizeof(int)> >& c,  
  
        int N) {  
#pragma unroll 8  
for (int i = 0; i < N; ++i) {  
    c[i] = a[i] + 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 Component Viewer report generated when you compile this example.

Figure 2. Component View of `vector_add` Component with Avalon-MM Master Interface



The diagram shows that `vector_add.B2` has two loads and one store. The default Avalon-MM Master settings used by the code example in [Pointer Interfaces](#) on page 8 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 `a` and vector `b`, and one wide store for vector `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 2. QoR Metrics Comparison for Avalon-MM Master Interface¹

QoR Metric	Pointer	Avalon-MM Master
ALMs	15593.5	643
DSPs	0	0
RAMs	30	0
f_{MAX} (MHz) ²	298.6	472.37
Latency (cycles)	24071	142
Initiation Interval (II) (cycles)	~508	1

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

²The f_{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.

3.1.3. 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.

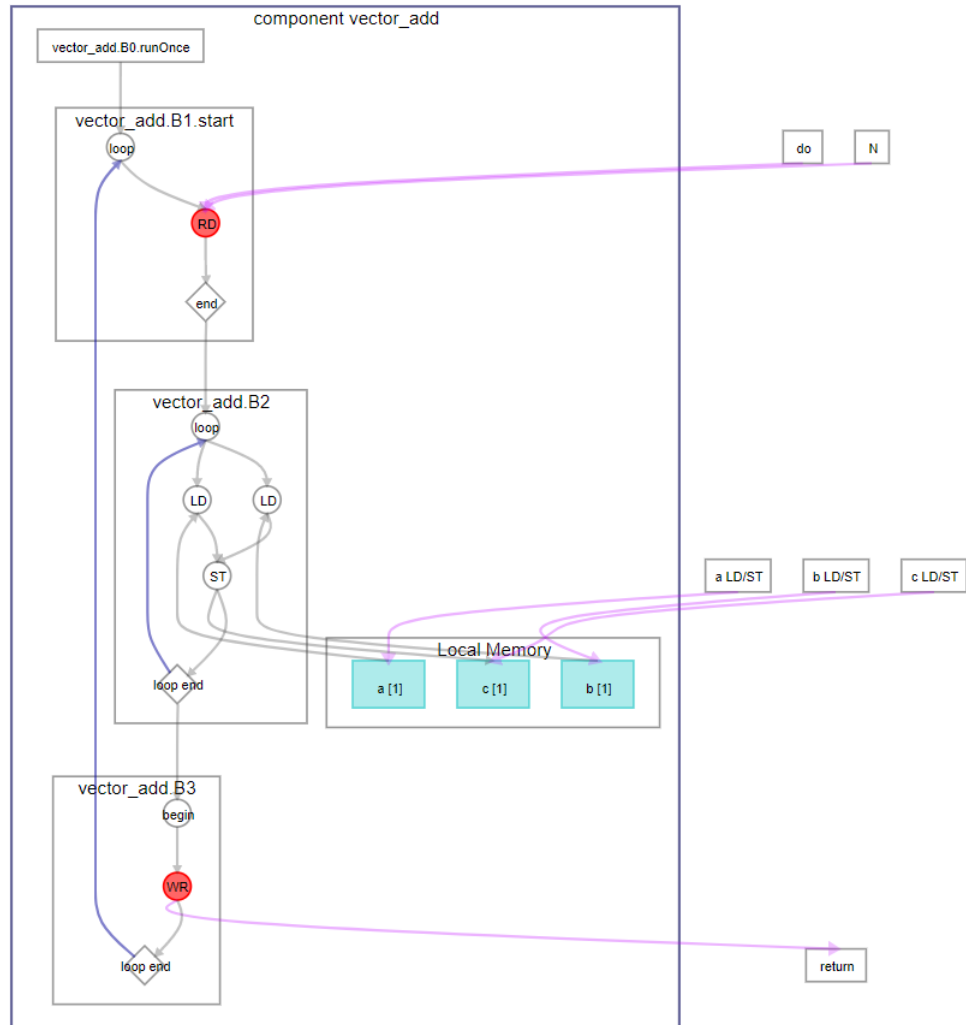
When you allocate a slave memory, you must define its size. Defining the size puts a limit on how large a value of N that the component can process. In this example, the RAM size is 1024 words. This RAM size means that N can have a maximal size of 1024 words.

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

```
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 Component Viewer report generated when you compile this example.

Figure 3. Component 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 3. QoR Metrics Comparison for Avalon-MM Slave Interface¹

QoR Metric	Pointer	Avalon-MM Master	Avalon-MM Slave
ALMs	15593.5	643	490.5
DSPs	0	0	0
RAMs	30	0	48
f _{MAX} (MHz) ²	298.6	472.37	498.26
Latency (cycles)	24071	142	139
Initiation Interval (II) (cycles)	~508	1	1



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

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

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_{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.

3.1.4. 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.

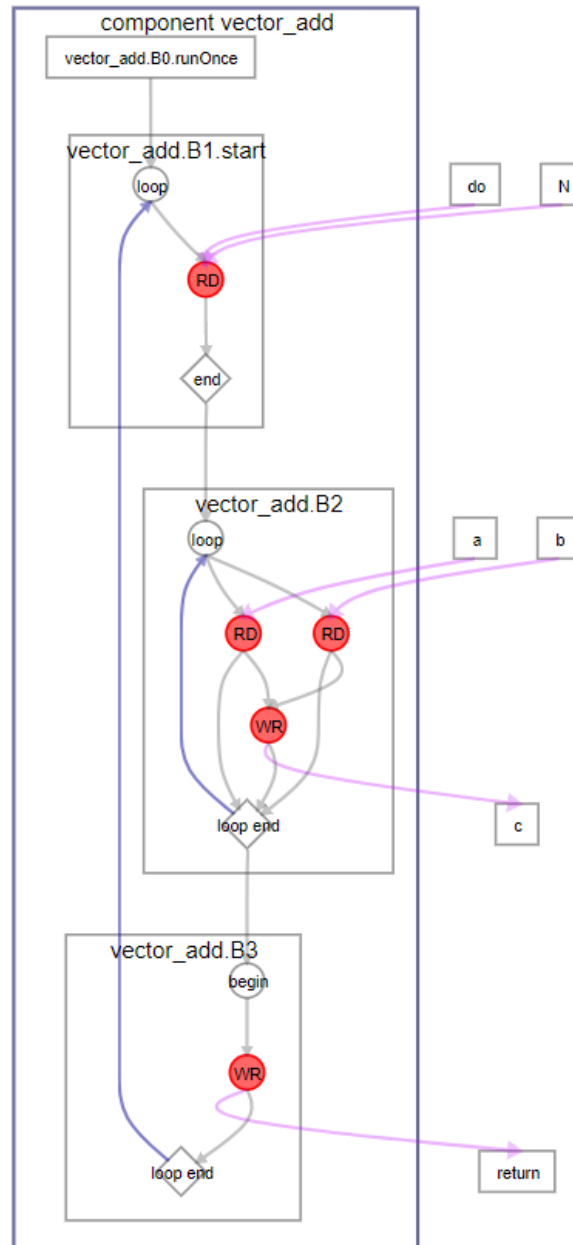
The vector addition example can be coded with an Avalon-ST interface as follows:

```
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 Component Viewer report generated when you compile this example.

Figure 4. Component View of `vector_add` Component with Avalon-ST Interface



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 4. QoR Metrics Comparison for Avalon-ST Interface¹

QoR Metric	Pointer	Avalon-MM Master	Avalon-MM Slave	Avalon-ST
ALMs	15593.5	643	490.5	314.5
DSPs	0	0	0	0
RAMs	30	0	48	0
f_{MAX} (MHz) ²	298.6	472.37	498.26	389.71
Latency (cycles)	24071	142	139	134
Initiation Interval (II) (cycles)	~508	1	1	1

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

²The f_{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.

3.1.5. 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 targeting an FPGA, passing array elements by value can result in smaller and simpler hardware on the FPGA.

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

```

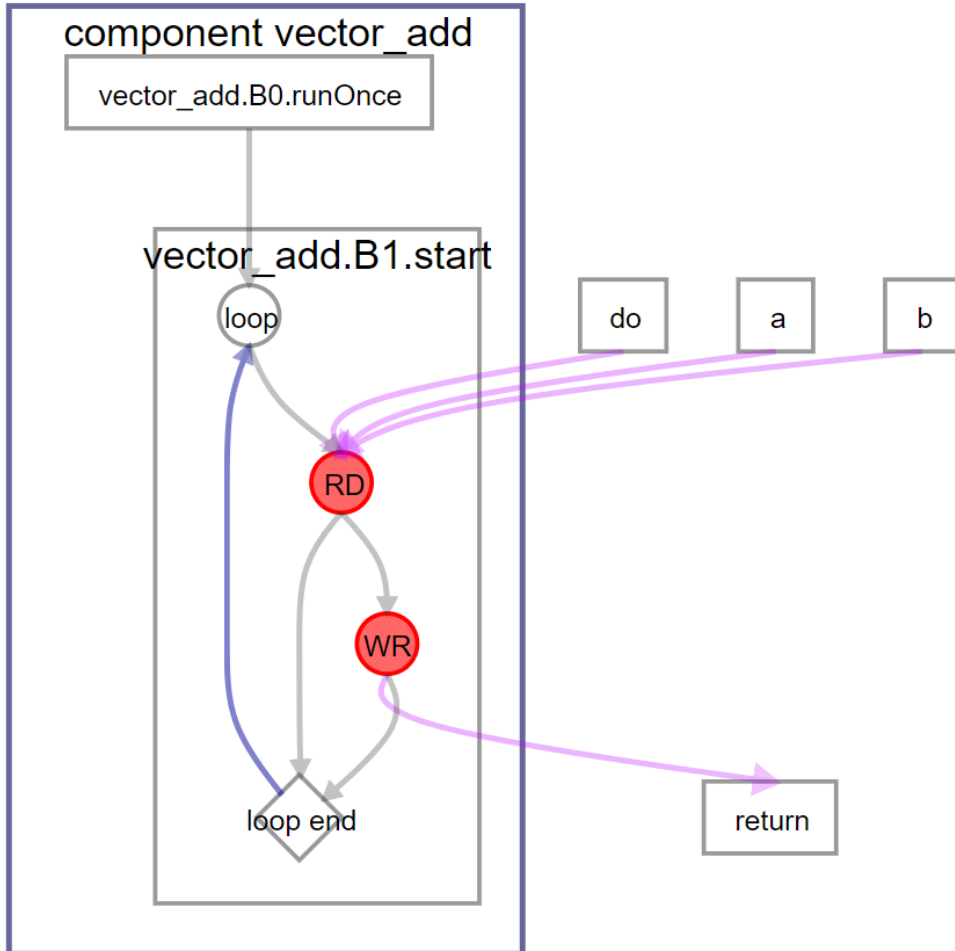
struct int_v8 {
    int data[8];
};
component int_v8 vector_add(
    int_v8 a,
    int_v8 b) {
    int_v8 c;
    #pragma unroll 8
    for (int i = 0; i < 8; ++i) {
        c.data[i] = a.data[i]
                   + b.data[i];
    }
    return c;
}

```

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 Component Viewer report generated when you compile this example.

Figure 5. Component View of vector_add Component with Pass-By-Value Interface

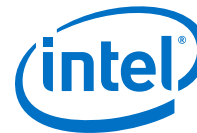


The latency of this component is one, and it has a loop initiation interval (II) of one.

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

Table 5. QoR Metrics Comparison for Pass-by-Value Interface¹

QoR Metric	Pointer	Avalon-MM Master	Avalon-MM Slave	Avalon-ST	Pass-by-Value
ALMs	15593.5	643	490.5	314.5	130
DSPs	0	0	0	0	0
RAMs	30	0	48	0	0
f_{MAX} (MHz) ²	298.6	472.37	498.26	389.71	581.06
Latency (cycles)	24071	142	139	134	128
Initiation Interval (II) (cycles)	~508	1	1	1	1



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

²The f_{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_{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 a initiation interval of 1, 128 component calls are processed in 128 cycles so the overall latency is 128.

3.2. Avoid Pointer Aliasing

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

The restrict type qualifier to use depends on your edition of Intel Quartus Prime:

- **PRO** The restrict type qualifier is `__restrict`.
- **STD** 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 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.

For more details, review the parameter aliasing tutorial in the following location:

```
<quartus_installdir>/hls/examples/tutorials/best_practices/parameter_aliasing
```

4. Loop Best Practices

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

The Intel HLS Compiler 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.

Tutorials Demonstrating Loop Best Practices

The Intel HLS Compiler comes with a number of tutorials that give you working examples to review and run so that you can see good coding practices as well as demonstrating important concepts.

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

Tutorial	Description
You can find these tutorials in the following location on your Intel Quartus Prime system: <pre data-bbox="256 1352 1386 1388"><quartus_installdir>/hls/examples/tutorials</pre>	
 <code>best_practices/ loop_coalesce</code>	Demonstrates the performance and resource utilization improvements of using <code>loop_coalesce</code> pragma on nested loops.
<code>best_practices/ loop_memory_dependency</code>	Demonstrates breaking loop-carried dependencies using the <code>ivdep</code> pragma.
<code>best_practices/ resource_sharing_filter</code>	Demonstrates the following versions of a 32-tap finite impulse response (FIR) filter design: <ul style="list-style-type: none"> • optimized-for-throughput variant • optimized-for-area variant



4.1. 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:

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

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);
    }
}
```

```

}
} ...
}
H

```

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`.

4.2. 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.

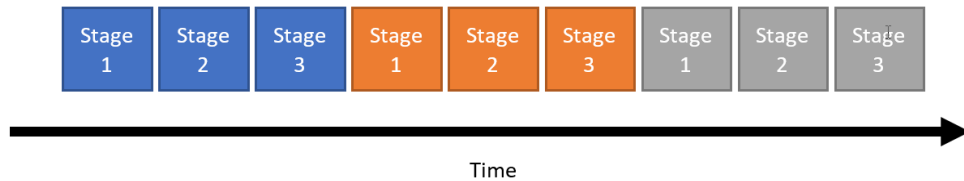
You can take advantage of the spatial compute structure to accelerate the loops by having multiple iterations of a loop executing concurrently. To have multiple iterations of a loop execute concurrently, unroll loops when possible and structure your loops so that dependencies between loop iterations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

4.2.1. 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 6. Basic loop with three stages and three iterations

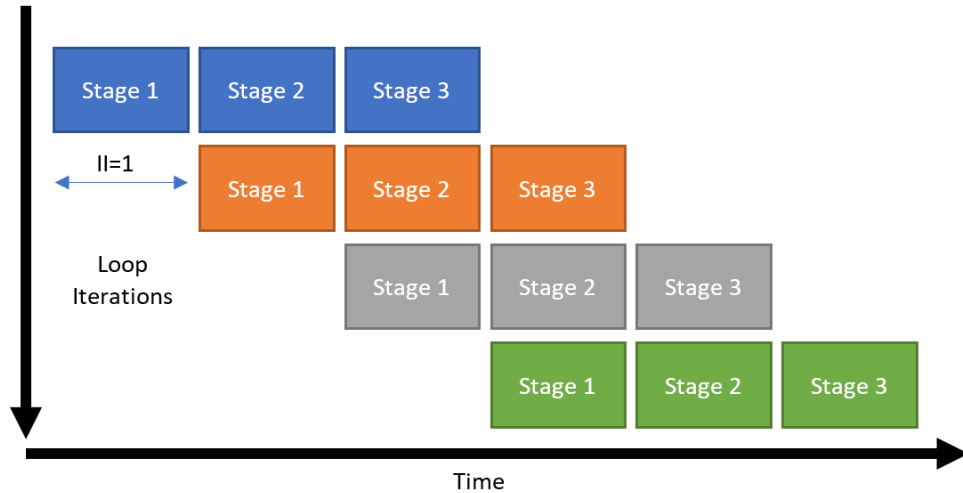


If each stage of this loop takes one clock cycle to execute, then this loop has a latency of nine cycles.

The following figure shows the pipelining of the loop from [Figure 6](#) on page 22.



Figure 7. 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 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 7](#) on page 23, 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 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:

$$\text{latency}_{\text{loop}} = (\text{iterations} - 1) * II + \text{latency}_{\text{body}}$$

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.

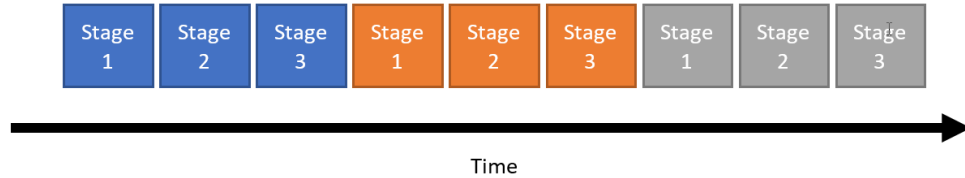
The Intel HLS Compiler supports pipelining nested loops without unrolling inner loops. When calculating the latency of nested loops, apply this formula recursively. This recursion means that having $II > 1$ is more problematic for inner loops than for outer loops. Therefore, algorithms that do most of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

4.2.2. 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.

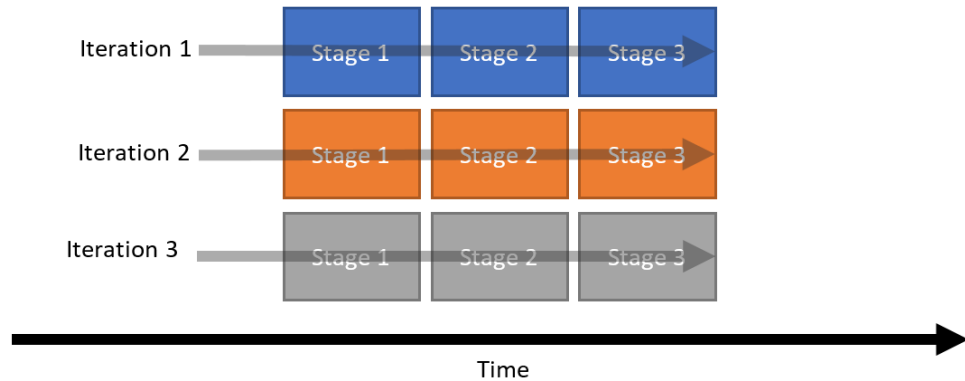
Figure 8. Basic loop with three stages and three iterations



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 8](#) on page 24 unrolled three times.

Figure 9. Unrolled loop with three stages and three iterations



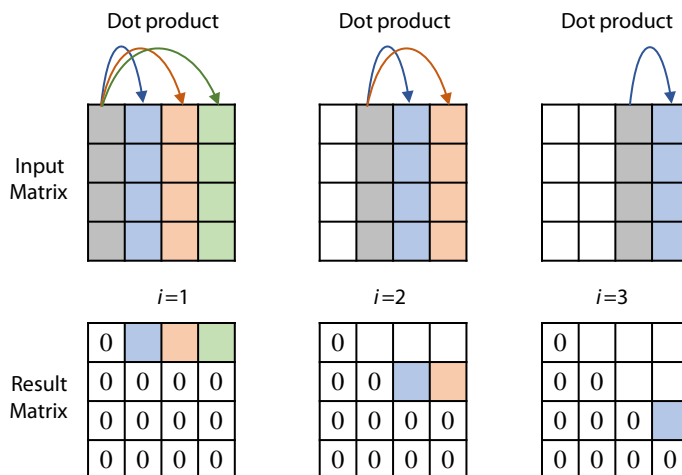
Three iterations of the loop can now be completed in only three clock cycles, but three times as many hardware resources are required.

You can control how the compiler unrolls a loop with the `#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 `#pragma unroll` directive, see the [best_practices/resource_sharing_filter](#) tutorial.

4.2.3. 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:

```

1.  #define ROWS 4
2.  #define COLS 4
3.
4.  component void dut(...) {
5.      float a_matrix[COLS][ROWS]; // store in column-major format
6.      float r_matrix[ROWS][COLS]; // store in row-major format
7.
8.      // setup...
9.
10.     for (int i = 0; i < COLS; i++) {
11.         for (int j = i + 1; j < COLS; j++) {
12.
13.             float dotProduct = 0;
14.             for (int mRow = 0; mRow < ROWS; mRow++) {
15.                 dotProduct += a_matrix[i][mRow] * a_matrix[j][mRow];
16.             }
17.             r_matrix[i][j] = dotProduct;
18.         }
19.     }
20.
21.     // continue...
22.
23. }

```

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 `--fp-relaxed` 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.

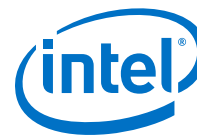
```
01: #define ROWS 4
02: #define COLS 4
03:
04: component void dut(...) {
05:     float a_matrix[COLS][ROWS]; // store in column-major format
06:     float r_matrix[ROWS][COLS]; // store in row-major format
07:
08:     // setup...
09:
10:     for (int i = 0; i < COLS; i++) {
11:
12:         #pragma unroll
13:         for (int j = 0; j < COLS; j++) {
14:             float dotProduct = 0;
15:
16:             #pragma unroll
17:             for (int mRow = 0; mRow < ROWS; mRow++) {
18:                 dotProduct += a_matrix[i][mRow] * a_matrix[j][mRow];
19:             }
20:
21:             r_matrix[i][j] = (j > i) ? dotProduct : 0; // predication
22:         }
23:     }
24: }
25:
26: // continue...
27:
28: }
```

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

4.3. 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 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:

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

4.4. 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 below 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 of pipeline parallelism that the Intel HLS Compiler can achieve, which reduces component performance.

```
for(int i = 1; i < N; i++)  
{  
    A[i] = A[i - 1] + i;  
}
```

The Intel HLS Compiler performs a static memory dependency analysis on loops to determine the extent of parallelism that it can achieve. If the Intel HLS Compiler 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:

```
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.

4.5. 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.

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

Related Information

[Loop Iteration Speculation \(speculated_iterations Pragma\)](#)



4.6. 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:

Nested Loop	Converted Single Loop
<pre>for (i = 0; i < N; i++) { //statements for (j = 0; j < M; j++) { //statements } //statements }</pre>	<pre>for (i = 0; i < N*M; i++) { //statements }</pre>

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:

```
#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:

```
int i = 0;
int j = 0;
while(i < N){

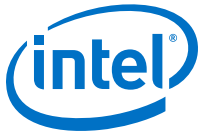
    sum[i][j] += i+j;
    j++;

    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 Reference Manual*.

4.7. 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 does not need to preserve the variable data across loops that do not use the variables.



Consider the following example:

```
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. 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 gives you some control over the memory architecture.

Tutorials Demonstrating Memory Architecture Best Practices

The Intel HLS Compiler comes with a number of tutorials that give you working examples to review and run so that you can see good coding practices as well as demonstrating important concepts.

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

Tutorial	Description
You can find these tutorials in the following location on your Intel Quartus Prime system: <pre data-bbox="256 1073 690 1098"><quartus_installdir>/hls/examples/tutorials</pre>	
component_memories/bank_bits	Demonstrates how to control component internal memory architecture for parallel memory access by enforcing which address bits are used for banking.
component_memories/depth_wise_merge	Demonstrates how to improve resource utilization by implementing two logical memories as a single physical memory with a depth equal to the sum of the depths of the two original memories.
component_memories/width_wise_merge	Demonstrates how to improve resource utilization by implementing two logical memories as a single physical memory with a width equal to the sum of the widths of the two original memories.

5.1. 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 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:

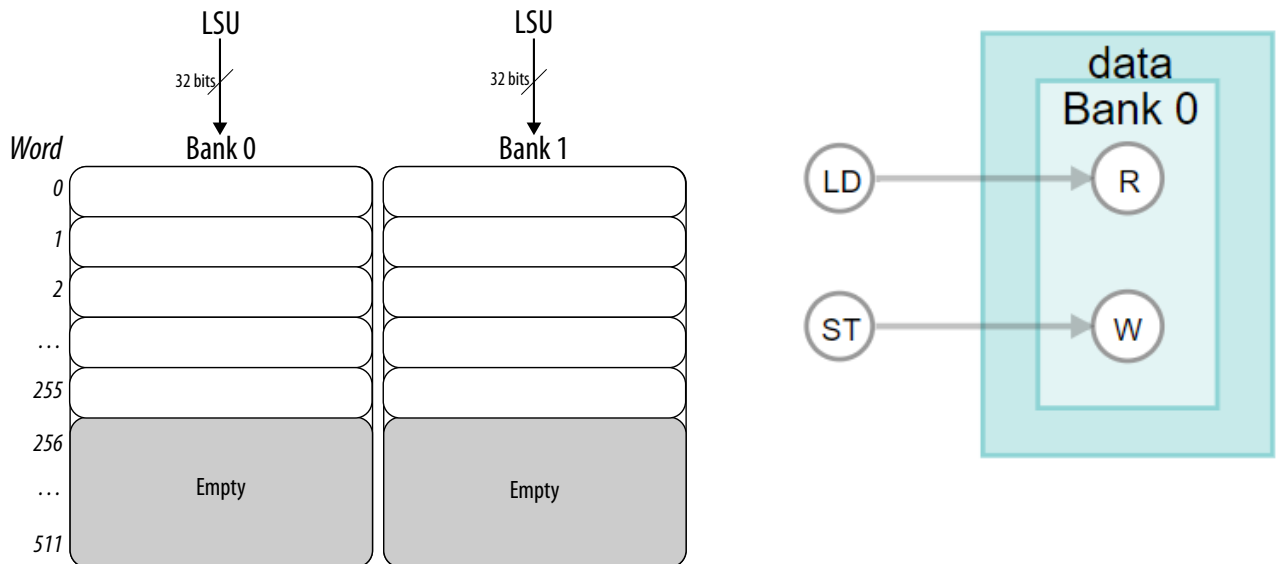
- `hls_bankwidth(n)`
- `hls_numbanks(n)`
- `hls_singlepump`
- `hls_numports_readonly_writeonly(m,n)`

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

```
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 10. Memory Structure Generated for mem_coalesce_default



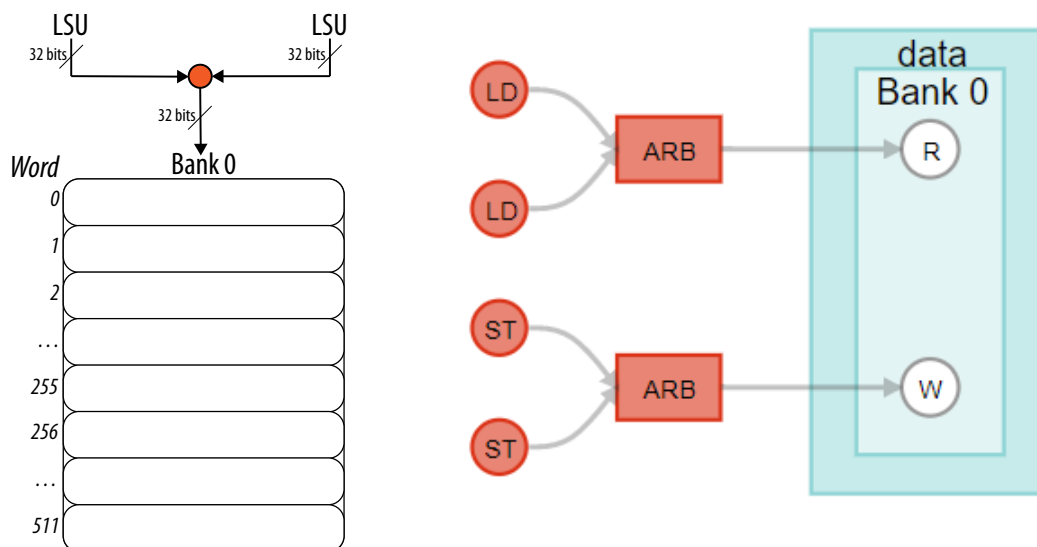
The modified code implements a simple dual-port on-chip memory block that is 512 locations deep by 32 bits wide (512x32 bits) with stallable arbitration:

```
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 simple dual port
    hls_singlepump hls_numports_readonly_writeonly(1,1)
    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;
}
```

Tip: Instead of specifying the `hls_singlepump` and `hls_numports_readonly_writeonly(1,1)` attributes, you can specify this configuration with the `hls_simple_dual_port_memory` attribute.

The following images show how the 512x32 bit memory with storable 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 11. 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 storable arbitration causes the amount of hardware needed to implement the component to increase. In the following table, you can compare the number ALMs and FFs required by the components.

Quartus Fit Resource Utilization Summary					
	ALMs	FFs	RAMs	DSPs	MLABs
Full design (all components)	1164.5	1922	3	0	25
mem_coalesce_override	1074.5	1654	1	0	23
mem_coalesce_default	90	268	2	0	2

5.2. 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 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:

```
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.

```
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.

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

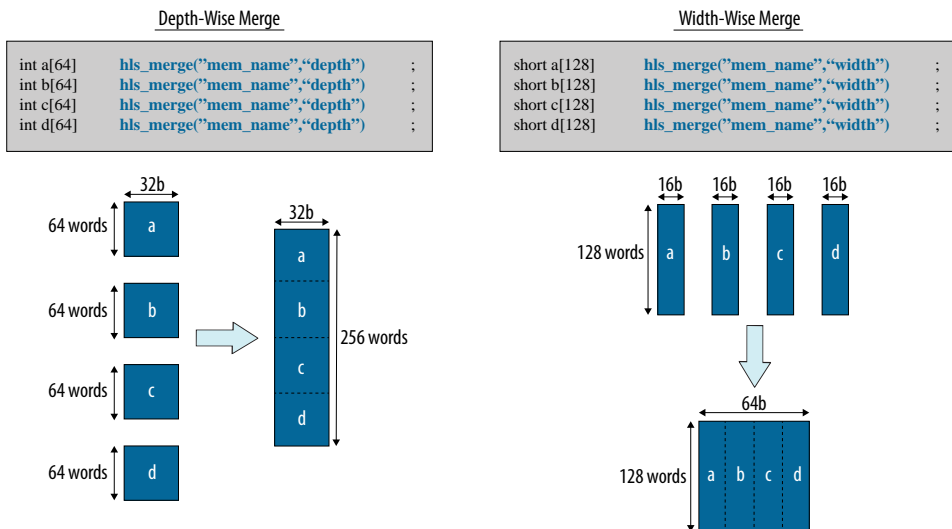
5.3. Merge Memories to Reduce Area

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 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.

Figure 12. Overview of width-wise merge and depth-wise merge

The following diagram shows how four memories can be merged width-wise and depth-wise.



5.3.1. Example: Merging Memories Depth-Wise

Use the `hls_merge("mem_name", "depth")` attribute to force the Intel HLS Compiler 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:

```

component int depth_manual(bool use_a, int raddr, int waddr, int wdata) {
  int a[128];
  int b[128];

  int rdata;

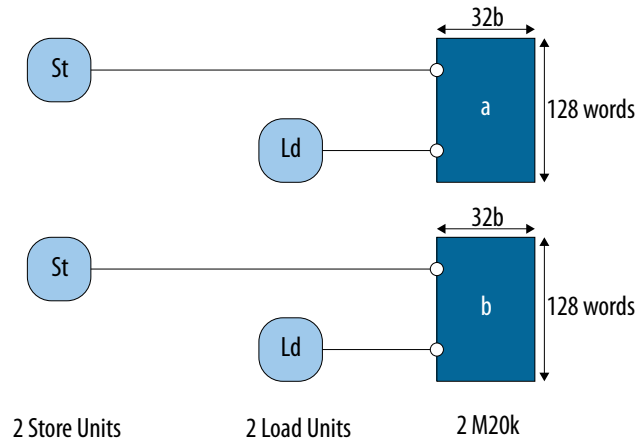
  // mutually exclusive write
  if (use_a) {
    a[waddr] = wdata;
  } else {
    b[waddr] = wdata;
  }

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

  return rdata;
}
          
```

The code instructs the Intel HLS Compiler to implement local memories a and b as two on-chip memory blocks, each with its own load and store instructions.

Figure 13. Implementation of Local Memory for Component `depth_manual`



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.

```

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;

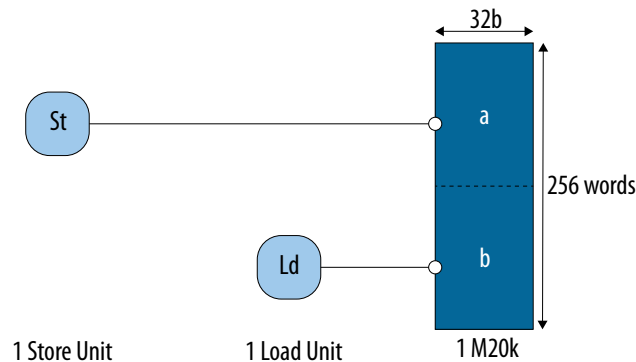
  // mutually exclusive write
  if (use_a) {
    a[waddr] = wdata;
  } else {
    b[waddr] = wdata;
  }

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

  return rdata;
}

```

Figure 14. 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 should not merge the accesses to local memories a and b because the load and store instructions to each memory are not mutually exclusive.

```
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;

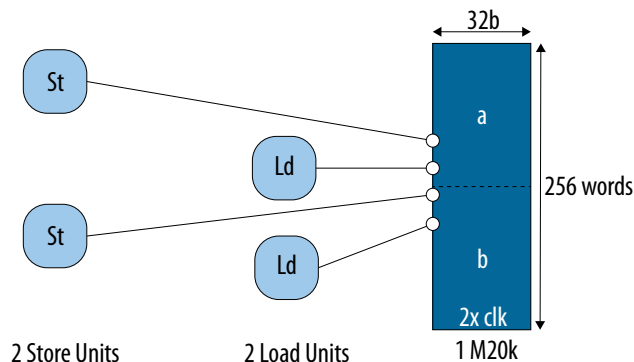
  // NOT mutually exclusive write
  a[waddr] = wdata;
  b[waddr] = wdata;

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

  return rdata;
}
```

In this case, the Intel HLS Compiler 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 15. Local Memories for Component `depth_manual` with Non-Mutually Exclusive Accesses



5.3.2. Example: Merging Memories Width-Wise

Use the `hls_merge("<mem_name>", "width")` attribute to force the Intel HLS Compiler 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:

```

component short width_manual (int raddr, int waddr, short wdata) {

    short a[256];
    short b[256];

    short rdata = 0;

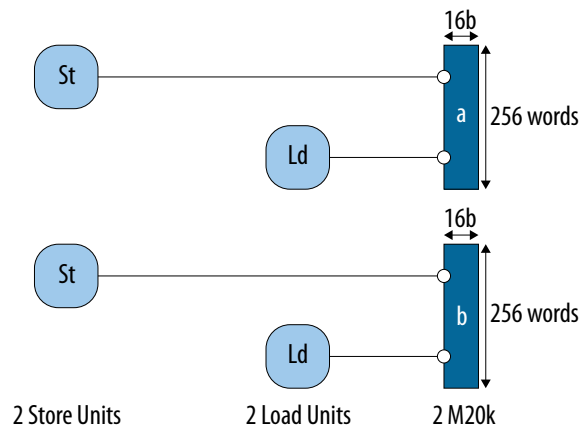
    // Lock step write
    a[waddr] = wdata;
    b[waddr] = wdata;

    // Lock step read
    rdata += a[raddr];
    rdata += b[raddr];

    return rdata;
}

```

Figure 16. Implementation of Local Memory for Component width_manual



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

```

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;

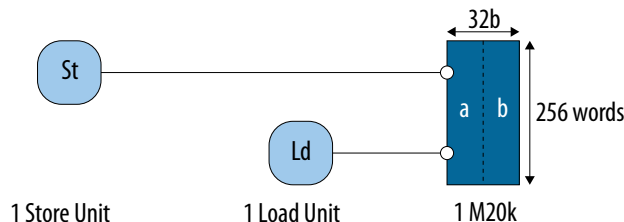
    // Lock step write
    a[waddr] = wdata;
    b[waddr] = wdata;

    // Lock step read
    rdata += a[raddr];
    rdata += b[raddr];

    return rdata;
}

```

Figure 17. Width-Wise Merge of Local Memories for Component width_manual



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

You have the option to tell the Intel HLS Compiler 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(b0, b1, ..., bn)` attribute.

The (b_0, b_1, \dots, b_n) arguments refer to the local memory address bit positions that the Intel HLS Compiler should use for the bank-selection bits. Specifying the `hls_bankbits(b0, b1, ..., bn)` attribute implies that the number of banks equals $2^{(\text{number of bank bits})}$.

Table 6. 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.

	Bank 0	Bank 1	Bank 2	Bank 3
Word 0	0000	01000	10000	11000
Word 1	00001	01001	10001	11001
Word 2	00010	01010	10010	11010
Word 3	00011	01011	10011	11011
Word 4	00100	01100	10100	11100
Word 5	00101	01101	10101	11101
Word 6	00110	01110	10110	11110
Word 7	00111	01111	10111	11111

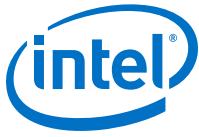
Restriction: Currently, the `hls_bankbits(b0, b1, ..., bn)` attribute supports only consecutive bank bits.

Example of Implementing the `hls_bankbits` Attribute

Consider the following example component code:

```
component int bank_arb_consecutive_multidim (int raddr,
                                             int waddr,
                                             int wdata,
                                             int upperdim) {

    int a[2][4][128] hls_numbanks(1);
```



```
#pragma unroll
for (int i = 0; i < 4; i++) {
    a[upperdim][i][(waddr & 0x7f)] = wdata + i;
}

int rdata = 0;

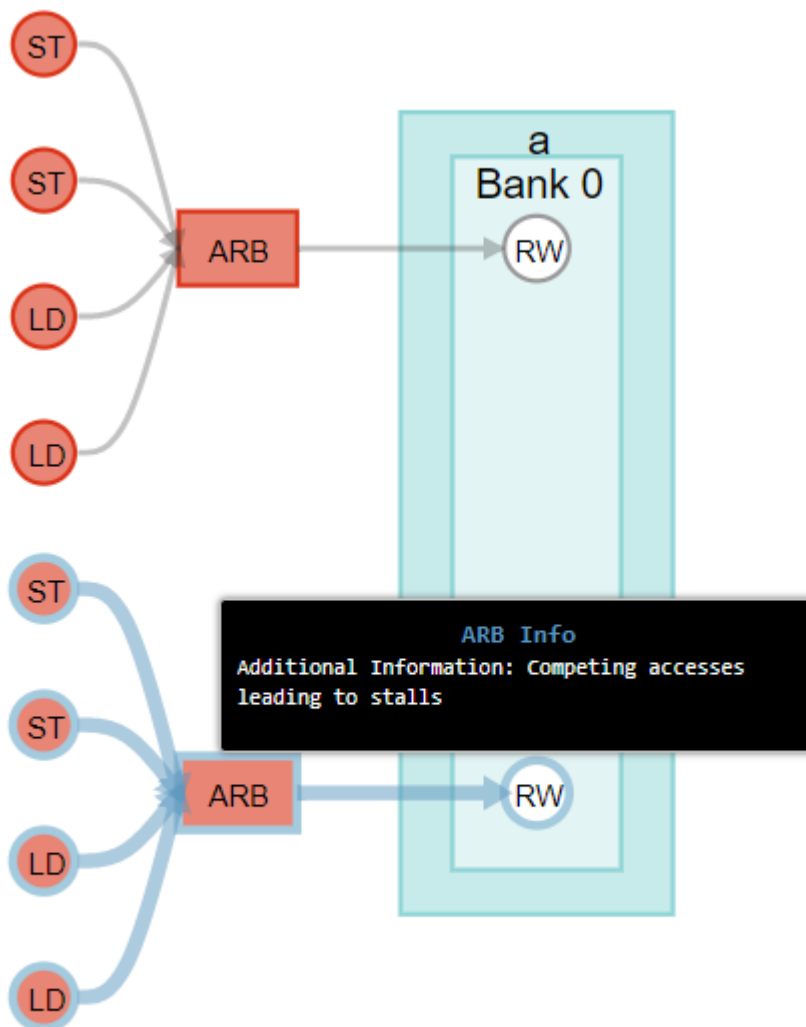
#pragma unroll
for (int i = 0; i < 4; i++) {
    rdata += a[upperdim][i][(raddr & 0x7f)];
}

return rdata;
}
```

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 blocks whenever possible because they consume a lot of FPGA area and can hurt the performance of your component.

Figure 18. Accesses to Local Memory a for Component bank_arb_consecutive_multidim

(1)



By default, the Intel HLS Compiler splits the memory into banks if it determines that the split is beneficial to the performance of your component. When the compiler generates a memory system, it uses the lower-order memory address bits to access the different memory banks. This behavior means that if you define your component memory structure so that the lowest order addresses are accessed in parallel, the compiler automatically infers the bank-selection bits for you.

(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.

This access pattern prevents storable arbitration on the memory. In this case, preventing storable arbitration reduced the II value to 1. In practice, this might mean that you store a matrix in column-major format instead of row-major format, if you intend to access multiple matrix rows concurrently.

Swapping the 128-element and 4-element dimension in the code example that follows results in no storable memory arbitration.

```

component int bank_arb_consecutive_multidim (int raddr,
int waddr,
int wdata,
int upperdim) {
    int a[2][128][4];

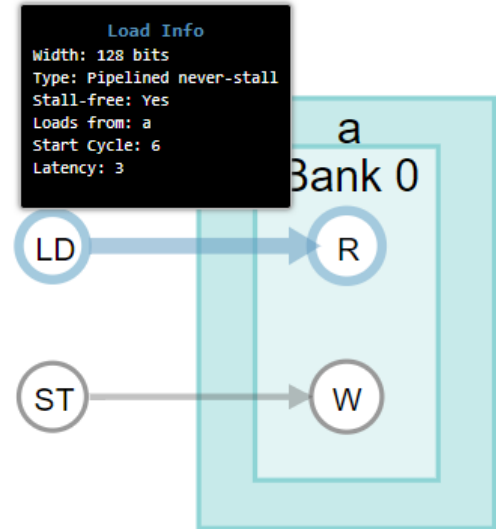
    #pragma unroll
    for (int i = 0; i < 4; i++) {
        a[upperdim][(waddr & 0x7f)][i] = wdata + i;
    }

    int rdata = 0;

    #pragma unroll
    for (int i = 0; i < 4; i++) {
        rdata += a[upperdim][(raddr & 0x7f)][i];
    }

    return rdata;
}

```



The dimension that is accessed in parallel is moved to be the lowest-order dimension in the memory array. The load has a width of 128 bits, which is the same as four 32-bit loads.

If you cannot change your memory structure, you can use the `hls_bankbits` attribute to explicitly control how load and store instructions access local memory. As shown in the following modified code example and figure, when you choose constant bank-select bits for each access to the local memory `a`, each pair of load and store instructions needs to connect to only one memory bank. In this example, there are four 32-bit loads, which results in a memory system similar to the earlier example.



Figure 19. Accesses to Local Memory a for Component bank_arb_consecutive_multidim with the hls_bankbits Attribute

```

component int bank_arb_consecutive_multidim (int raddr,
      int waddr,
      int wdata,
      int upperdim) {

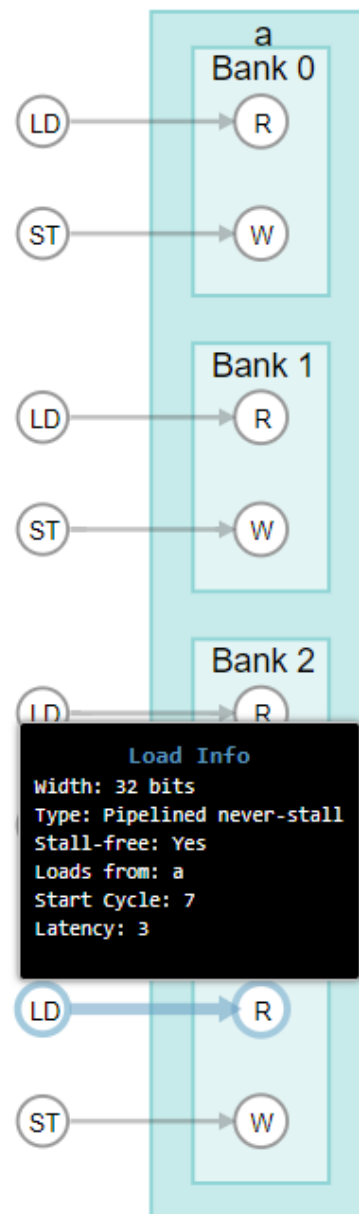
  int a[2][4][128] hls_bankbits(8,7);

  #pragma unroll
  for (int i = 0; i < 4; i++) {
    a[upperdim][i][(waddr & 0x7f)] = wdata + i;
  }

  int rdata = 0;

  #pragma unroll
  for (int i = 0; i < 4; i++) {
    rdata += a[upperdim][i][(raddr & 0x7f)];
  }

  return rdata;
}
  
```



When specifying the word-address bits for the `hls_bankbits` attribute, ensure that the resulting bank-select bits are constant for each access to local memory. As shown in the following example, the local memory access pattern does not guarantee that the chosen bank-select bits are constant for each access. As a result, each pair of load and store instructions must connect to all the local memory banks, leading to storable accesses.

Figure 20. Stallable Accesses to Local Memory a for Component bank_arb_consecutive_multidim with the hls_bankbits Attribute

```

component int bank_arb_consecutive_multidim (int raddr,
                                             int waddr,
                                             int wdata,
                                             int upperdim){

    int a[2][4][128] hls_bankbits(5,4);

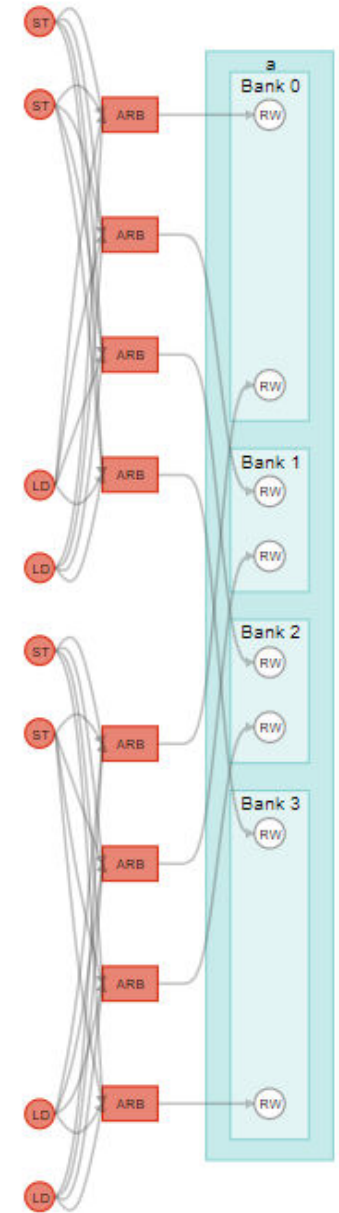
    #pragma unroll
    for (int i = 0; i < 4; i++) {
        a[upperdim][i][(waddr & 0x7F)] = wdata + i;
    }

    int rdata = 0;

    #pragma unroll
    for (int i = 0; i < 4; i++) {
        rdata += a[upperdim][i][(raddr & 0x7F)];
    }

    return rdata;
}

```



In this case, the II is estimated to be approximately 64.

6. **PRO** System of Tasks

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 test it in the Intel HLS Compiler cosimulation environment.

6.1. **PRO** 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:

```
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 one of the loops into an asynchronous task. With the first loop in an asynchronous task, the second loop can run concurrently with the first loop.

```
void offloaded_work() {
    // first loop
    for (int i = 0; i < n; i++) {
        // Do something
    }
}

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

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

6.2. **PRO** 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.

To allow for calls from multiple places to a task, the Intel HLS Compiler 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.

6.3. **PRO** 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.

6.4. **PRO** Avoiding Potential Performance Pitfalls

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 `ihc::launch` and `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_dealy`



The Intel HLS Compiler 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.

7. 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 provides debug functionality so that you can easily detect overflows in arbitrary precision datatypes.

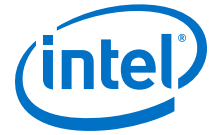
PRO Because C++ automatically promotes smaller datatypes such as `short` or `char` to 32 bits for operations such as addition or bit-shifting, you must use the arbitrary precision datatypes if you want to create narrow datapaths in your component.

Tutorials Demonstrating Datatype Best Practices

The Intel HLS Compiler comes with a number of tutorials that give you working examples to review and run so that you can see good coding practices as well as demonstrating important concepts.

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

Tutorial	Description
You can find these tutorials in the following location on your Intel Quartus Prime system:	
<code><quartus_installdir>/hls/examples/tutorials</code>	
<code>best_practices/ac_datatypes</code>	Demonstrates the effect of using <code>ac_int</code> datatype instead of <code>int</code> datatype.
<code>ac_datatypes/ac_fixed_constructor</code>	Demonstrates the use of the <code>ac_fixed</code> constructor where you can get a better QoR by using minor variations in coding style.
<code>ac_datatypes/ac_int_basic_ops</code>	Demonstrates the operators available for the <code>ac_int</code> class.
<code>ac_datatypes/ac_int_overflow</code>	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.
<code>best_practices/single_vs_double_precision_math</code>	Demonstrates the effect of using single precision literals and functions instead of double precision literals and functions.
STD <code>best_practices/integer_promotion</code>	Demonstrates how integer promotion rules can influence the behavior of a C or C++ program.



7.1. 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.

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

7.2. 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.

```
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:

```
int14 efficient_left_only_shift(int14 a, uint14 b) {  
    return (a << b);  
}
```

8. Advanced Troubleshooting

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

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

8.1. Component Fails Only In Cosimulation

Discrepancies between the results of compiling your component in emulation (`-march=x86-64`) mode or cosimulation (`-march=FPGA_name_or_part_no`) mode are typically caused by bugs in your component or testbench. However, there are some common cases where the discrepancies are caused by something other than a bug.

Comparing Floating Point Results

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

Using `#pragma ivdep` to Ignore Memory Dependencies

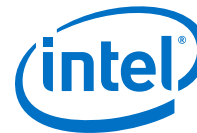
The `#pragma ivdep` compiler pragma can cause functional incorrectness in your component if your component has a memory dependency that you attempted to ignore with the pragma. You can try to use the `safelen` modifier to control how many memory accesses that you can permit before a memory dependency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

See [Loop-Carried Dependencies \(`ivdep` Pragma\)](#) in *Intel High Level Synthesis Compiler Reference Manual* for a description of this pragma.

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

STD Unintentional Integer Promotion

The Intel HLS Compiler provided with Intel Quartus Prime Standard Edition does not automatically promote small data types (such as `unsigned char` or `short`) to 32-bit widths during cosimulation. Other compilers (like `g++`) might promote integers when compiling your component for emulation. This difference in integer promotion behavior means that a value in your component has preserved overflow bits during emulation but is truncated during cosimulation.



The Intel HLS Compiler provided with Intel Quartus Prime Pro Edition automatically promotes small data types during cosimulation, so the behavior in emulation and cosimulation should be consistent. To prevent promotion of small datatypes in the Intel HLS Compiler provided with Intel Quartus Prime Pro Edition, use the `ac_int` datatype.

For example, the operands in the following code example are of type `unsigned char`, so by default the Intel HLS Compiler compiles the code to perform the operation with 8-bit arithmetic. Compilers that apply integer promotion rules promote the operation to use 32-bit arithmetic.

```
component
int add_width(unsigned char a, unsigned char b) {
    int sum = a + b;
    return sum;
}
```

This code example generates different overflow behavior in emulation and cosimulation. The Intel HLS Compiler truncates the integers at 8 bit, while other C++ compilers preserve the overflow bits.

You can mimic the integer promotion behavior by using the `--promote-integers` compiler option. See [Compiler Options](#) in *Intel High Level Synthesis Compiler Reference Manual* for a description of this compiler option.

To see an example of using the `--promote-integers` compiler option, review the tutorial in `<quartus_installdir>/hls/examples/tutorials/best_practices/integer_promotion`.

Check for Uninitialized Variables

Many coding practices can result in behavior that is undefined by the C++ specification. Sometimes this undefined behavior works as expected in emulation, but not in cosimulation.

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 cosimulation flow, you can see where the RTL behavior diverges from the emulator behavior.

8.2. Component Gets Bad 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 Component Memory Viewer report in the High Level Design Reports, and look for storable arbitration nodes and unexpected RAM utilization.

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

Component Uses More FPGA Resource Than Expected

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

A way to reduce area consumption is to relax the f_{MAX} requirements by setting a target f_{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_{MAX} than you specify, so when you set a target f_{MAX} to a lower value than you need, your design might still achieve an acceptable f_{MAX} value, and a design that consumes less area.

PRO The `hls_scheduler_target_fmax_mhz` component attribute is available only in Intel HLS Compiler Pro Edition.

PRO To learn more about the behavior of f_{MAX} target value control see the following tutorial: `<quartus_installdir>/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 31 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 merges the arrays into the same RAM block. You might see storable arbitration in the Component Memory Viewer because there are not enough Load/Store sites on the memory system.

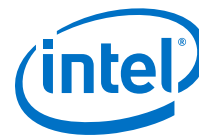
For example, the following code examples show an array of `struct` variables, a conditional operator that results in storable arbitration, and a workaround that avoids storable arbitration.

```
struct MyStruct {
    float a;
    float b;
}

MyStruct array1[64];
MyStruct array2[64];
```

The following conditional operator that uses these arrays of `struct` variables causes storable 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.

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

STD File-Scoped Static Variables

The Intel HLS Compiler provided with Intel Quartus Prime Standard Edition supports file-scoped static variables, but any memory attributes that you apply to static arrays work only if the static array is declared within the component function. Memory attributes applied to file-scope static variables are ignored. Memory attributes applied to a variable are also ignored if you attempt to apply attributes to a array members in a struct or class definition.

If you want to override the default memory settings for an array variable, ensure that the array variable is declared in the scope of the component function where the array variable is used. You can pass pointers to the static array to any subroutines that might access the static array.

This code change is shown in the following example. The code samples and high-level design report views that follow compare two implementations of a component that reads data from a stream into a local memory, then processes the data that is in that local memory.

In the first code example, the local memory is a file-scoped static variable. In the second code example, the local memory is a function-scoped static variable.

The second code example gets better QoR because you can apply memory optimization attributes to the static variable declaration. In this second example, the `hls_memory` and `hls_numbanks(1)` attributes force the static array into a single bank of on-chip RAM blocks.

Figure 21. Example 1: File-scoped Static Variable

```

hls_memory hls_numbanks(1) static int myStaticArray[64];

void loadData(ihc::stream_in<int> &intStreamIn)
{
    for(int idx = 0; idx < 64; idx++)
    {
        myStaticArray[idx] = intStreamIn.read();
    }
}

int findMax()
{
    int maxVal = 0;
    for(int idx = 0; idx < 64; idx++)
    {
        int val = myStaticArray[idx];
        if (val > maxVal)
        {
            maxVal = val;
        }
    }

    return maxVal;
}

component
int dut(ihc::stream_in<int> &intStreamIn)
{
    loadData(intStreamIn);
    return findMax();
}

```

The screenshot shows an IDE interface with three main panels:

- Memory list:** A tree view showing the component 'dut' and its memory elements, including 'myStaticArray' and 16 'Bank' entries (Bank 0 to Bank 15).
- Component memory viewer:** A diagram showing the internal structure of the component, with memory banks connected to a central processing unit.
- fileScopeStatic.cpp:** A code editor showing the implementation of the C++ code, including headers, the 'loadData' function, the 'findMax' function, and the 'dut' component definition.

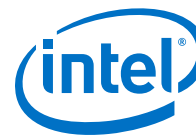


Figure 22. Example 2: Function-scoped Static Variable

```
void loadData(ihc::stream_in<int> &intStreamIn, int myStaticArray[64])
{
    for(int idx = 0; idx < 64; idx++)
    {
        myStaticArray[idx] = intStreamIn.read();
    }
}

int findMax(int myStaticArray[64])
{
    int maxVal = 0;
    for(int idx = 0; idx < 64; idx++)
    {
        int val = myStaticArray[idx];
        if (val > maxVal)
        {
            maxVal = val;
        }
    }
    return maxVal;
}

component
int dut(ihc::stream_in<int> &intStreamIn)
{
    hls_memory hls_numbanks(1) static int myStaticArray[64];

    loadData(intStreamIn, myStaticArray);
    return findMax(myStaticArray);
}
```

The screenshot shows an IDE with three main panels. On the left is a 'Memory list' showing a tree view with 'dut', 'myStaticArray', and 'Bank 0'. The central panel is the 'Component memory viewer', which displays a diagram of memory access. It shows two 'ST' (Store) operations on the left, a 'SHARE' block in the middle, and a 'W' (Write) operation on the right. Below the 'SHARE' block is an 'LD' (Load) operation, and below that is an 'R' (Read) operation. A large cyan box labeled 'myStaticArray Bank 0' encompasses the 'W' and 'R' operations. On the right is a code editor window titled 'funcScopeStatic.cpp' showing the implementation of the functions from the code block above, with line numbers 24 through 57 visible.

Cluster Logic

Your design might consume more RAM blocks than you expect, especially if you store many array variables in large registers. The Area Report view of the high-level design report (`report.html`) can help find this issue.

Area analysis of system
(area utilization values are estimated)
Notation *file:X > file:Y* indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs	RAMs	DSPs	Details
▼ System	11203 (1%)	13117 (1%)	44 (2%)	29 (2%)	
▼ qrd	11203 (1%)	13117 (1%)	44 (2%)	29 (2%)	• Number of ...
Component call	0	0	0	0	• Stream imp...
Component return	0	0	0	0	• Stream imp...
MGS.cpp:15 (q_matrix)	0	0	8	0	• Memory sys... • Requested ... • Implemente...
MGS.cpp:18 (r_matrix)	0	0	1	0	• Memory sys... • Requested ... • Implemente...
MGS.cpp:21 (t_matrix)	264	2048	16	0	• Memory sys... • Requested ... • Implemente...
Stream 'matrixData'	0	0	0	0	• Stream imp...
Stream 'qMatrixStream'	0	0	0	0	• Stream imp...

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.

▼ qrd.B4	3835 (0%)	3638 (0%)	19 (1%)	21 (1%)	
Cluster logic	70	178	11	0	• Logic requ...
▶ Computation	1785	1836	6	21	
▶ Feedback	55	31	0	0	• Resources ...
▼ State	1925	1593	2	0	• Resources ...
No Source Line	1925	1593	2	0	
▶ qrd.B5	4700 (1%)	4910 (0%)	0 (0%)	8 (1%)	

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).

9. Document Revision History for Intel HLS Compiler Best Practices Guide

Document Version	Intel Quartus Prime Version	Changes
2019.04.01	19.1	<ul style="list-style-type: none"> PRO Added new chapter to cover best practices when using HLS tasks in System of Tasks on page 45. Moved some content from Loop Best Practices on page 20 into a new section called Reuse Hardware By Calling It In a Loop on page 21. PRO Revised Component Uses More FPGA Resource Than Expected on page 52 to include information about the <code>hls_scheduler_target_fmax_mhz</code> component attribute.
2018.12.24	18.1	<ul style="list-style-type: none"> Updated to Loop Best Practices on page 20 to include information about function inlining in components and using loops to minimize the resulting hardware duplication.
2018.09.24	18.1	<ul style="list-style-type: none"> PRO The Intel HLS Compiler has a new front end. For a summary of the changes introduced by this new front end, see Improved Intel HLS Compiler Front End in the Intel High Level Synthesis Compiler Version 18.1 Release Notes. PRO The <code>--promote-integers</code> flag and the <code>best_practices/integer_promotion</code> 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: <ul style="list-style-type: none"> — Component Fails Only In Cosimulation on page 50 — Datatype Best Practices on page 48
2018.07.02	18.0	<ul style="list-style-type: none"> Added a new chapter, Advanced Troubleshooting on page 50, to help you troubleshoot when your component behaves differently in cosimulation and emulation, and when your component has unexpectedly poor performance, resource utilization, or both.
<i>continued...</i>		



Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	<ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> PRO Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Pro Edition. STD Indicates that a feature or content applies only to the Intel HLS Compiler provided with Intel Quartus Prime Standard Edition. Added <code>best_practices/loop_coalesce</code> to the list of tutorials in Loop Best Practices on page 20. Added <code>interfaces/explicit_streams_packets_ready_empty</code> to list of tutorials in Interface Best Practices on page 6. Revised Example: Specifying Bank-Selection Bits for Local Memory Addresses on page 39 with improved descriptions and new graphics that reflect what you would see in the high-level design reports (<code>report.html</code>) for the example component. Updated Example: Overriding a Coalesced Memory Architecture on page 31 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	<ul style="list-style-type: none"> Added Choose the Right Interface for Your Component on page 7 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 6.
2017.11.06	17.1	<p>Initial release.</p> <p>Parts of this book consist of content previously found in the Intel High Level Synthesis Compiler User Guide and the Intel High Level Synthesis Compiler Reference Manual.</p>