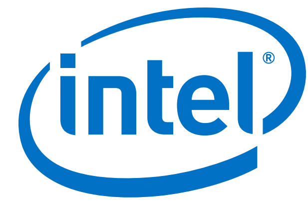
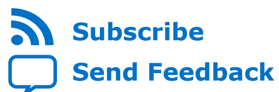


Intel® High Level Synthesis Compiler

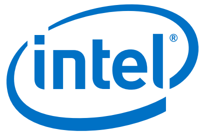
Best Practices Guide

Updated for Intel® Quartus® Prime Design Suite: **17.1**



UG-20107 | 2017.12.22

Latest document on the web: **PDF** | **HTML**



Contents

1 Best Practices for Coding and Compiling Your Component....... 3

2 Interface Best Practices....... 4

- 2.1 Choose the Right Interface for Your Component....... 5
  - 2.1.1 Pointer Interfaces....... 5
  - 2.1.2 Avalon Memory Mapped Master Interfaces....... 8
  - 2.1.3 Avalon Memory Mapped Slave Interfaces....... 10
  - 2.1.4 Avalon Streaming Interfaces....... 12
  - 2.1.5 Pass-by-Value Interface....... 14
- 2.2 Avoid Pointer Aliasing....... 16

3 Loop Best Practices....... 17

- 3.1 Parallelize Loops....... 18
  - 3.1.1 Pipeline Loops....... 18
  - 3.1.2 Unroll Loops....... 19
  - 3.1.3 Example: Loop Pipelining and Unrolling....... 20
- 3.2 Construct Well-Formed Loops....... 22
- 3.3 Minimize Loop-Carried Dependencies....... 23
- 3.4 Avoid Complex Loop-Exit Conditions....... 24
- 3.5 Convert Nested Loops into a Single Loop....... 24
- 3.6 Declare Variables in the Deepest Scope Possible....... 25

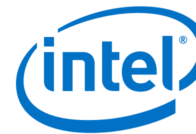
4 Memory Architecture Best Practices....... 26

- 4.1 Example: Overriding a Coalesced Memory Architecture....... 26
- 4.2 Example: Overriding a Banked Memory Architecture....... 27
- 4.3 Merge Memories to Reduce Area....... 28
  - 4.3.1 Example: Merging Memories Depth-Wise....... 29
  - 4.3.2 Example: Merging Memories Width-Wise....... 31
- 4.4 Example: Specifying Bank-Selection Bits for Local Memory Addresses....... 33

5 Datatype Best Practices....... 37

- 5.1 Avoid Implicit Data Type Conversions....... 37
- 5.2 Avoid Negative Bit Shifts When Using the `ac_int` Datatype....... 38

A Document Revision History....... 39



1 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 component so that you can apply the techniques that give you most optimized component possible.

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 Intel® High Level Synthesis (HLS) Compiler to see examples of how some of these techniques can be implemented.

- Interface Best Practices on page 4

  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 optimal interface for your component.

- Loop Best Practices on page 17

  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 26

  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 37

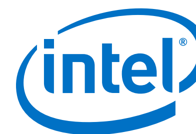
  The datatypes in your component and possible conversions that might occur to them can significantly affect the performance and FPGA area usage of your component. Review the datatype best practices for tips and guidance how best to control datatype sizes and conversions in your component.

- 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.

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

ISO 9001:2008 Registered



2 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 optimal 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 limitations in mind when determining the optimal interface for your component.

Tutorials Demonstrating Interface Best Practices

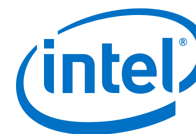
The Intel HLS Compiler comes with a number of tutorials that give you working examples to review and run so that you can see good coding practices as well as demonstrating important concepts.

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: `<quartus_installdir>/hls/examples/tutorials`	
`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`	Demonstrates the use of the `restrict` keyword on component arguments.
`interfaces/explicit_streams_buffer`	Demonstrates how to use explicit `stream_in` and `stream_out` interfaces in the component and testbench.
`interfaces/explicit_streams_packets_ready_valid`	Demonstrates how to use the `usesPackets`, `usesValid`, and `usesReady` stream template parameters.
`interfaces/mm_master_testbench_operators`	Demonstrates how to invoke a component at different indicies of an Avalon Memory Mapped (MM) Master (`mm_master` class) interface.
`interfaces/mm_slaves`	Demonstrates how to create Avalon-MM Slave interfaces (slave registers and slave memories).
`interfaces/multiple_stream_call_sites`	Demonstrates the benefits of using multiple stream call sites.
`interfaces/pointer_mm_master`	Demonstrates how to create Avalon-MM Master interfaces and control their parameters.
`interfaces/stable_arguments`	Demonstrates how to use the `stable` attribute for unchanging arguments to improve resource utilization.

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

ISO 9001:2008 Registered



Related Links

- Avalon Memory-Mapped Interface Specifications
- Avalon Streaming Interface Specifications

2.1 Choose the Right Interface for Your Component

Different component interfaces can affect the 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`.

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`.

2.1.1 Pointer Interfaces

For a typical C programmer, a first attempt at coding this algorithm might be to declare vector `a`, vector `b`, and vector `c` as pointers to get the data in and out of the component.

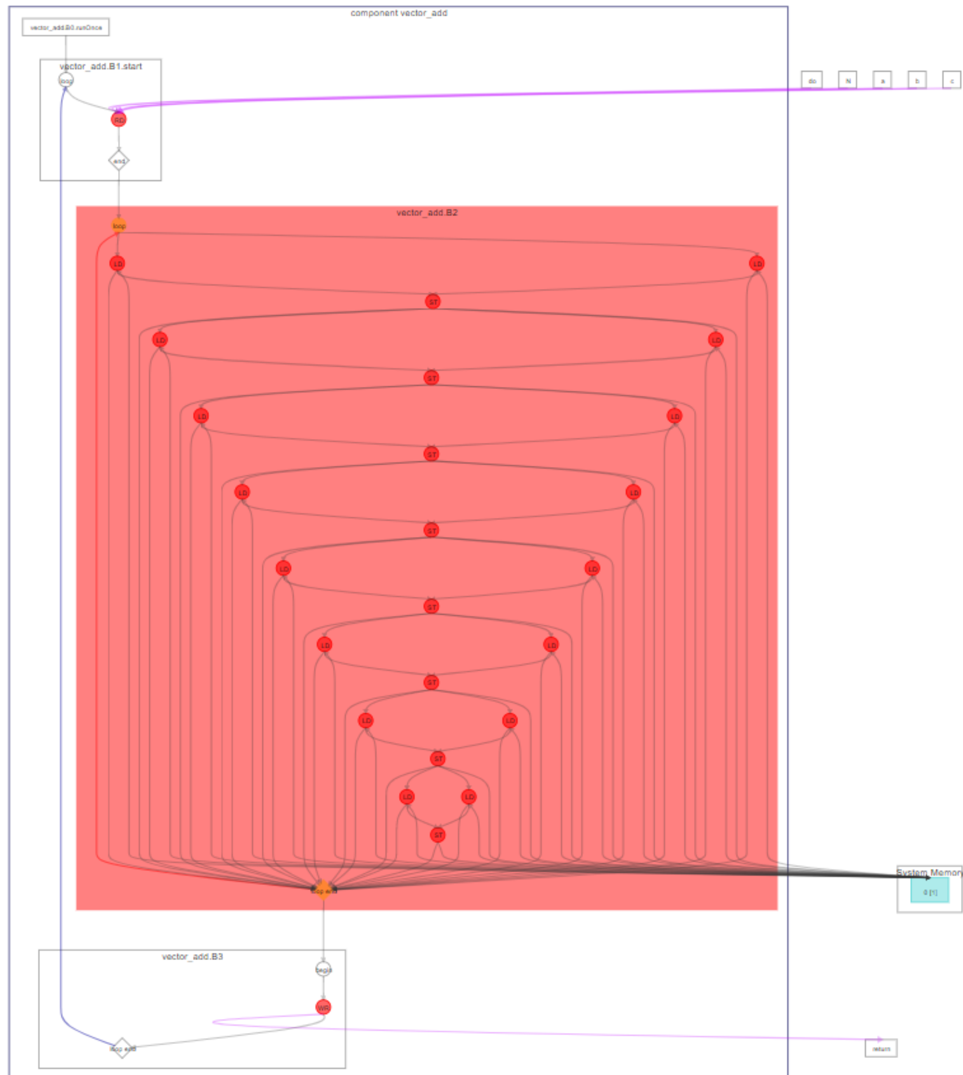
The vector addition component example with pointer interfaces can be coded as follows:

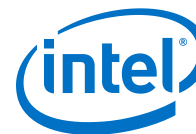
```
component void vector_add(int* a,
                          int* b,
                          int* c,
```

```
                                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 the Intel HLS Compiler cannot assume there are no data dependencies between loop iterations because pointer aliasing might exist. If data dependencies exist, the Intel HLS Compiler cannot pipeline the loop iterations effectively.

Loops analysis ☑ 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 Interface[1]

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

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

[2]The f_{max} measurement was calculated from one seed.

2.1.2 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 specializing the Avalon-MM master interfaces.

Specializing the Avalon-MM master interface to the vector addition component example can be coded 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:

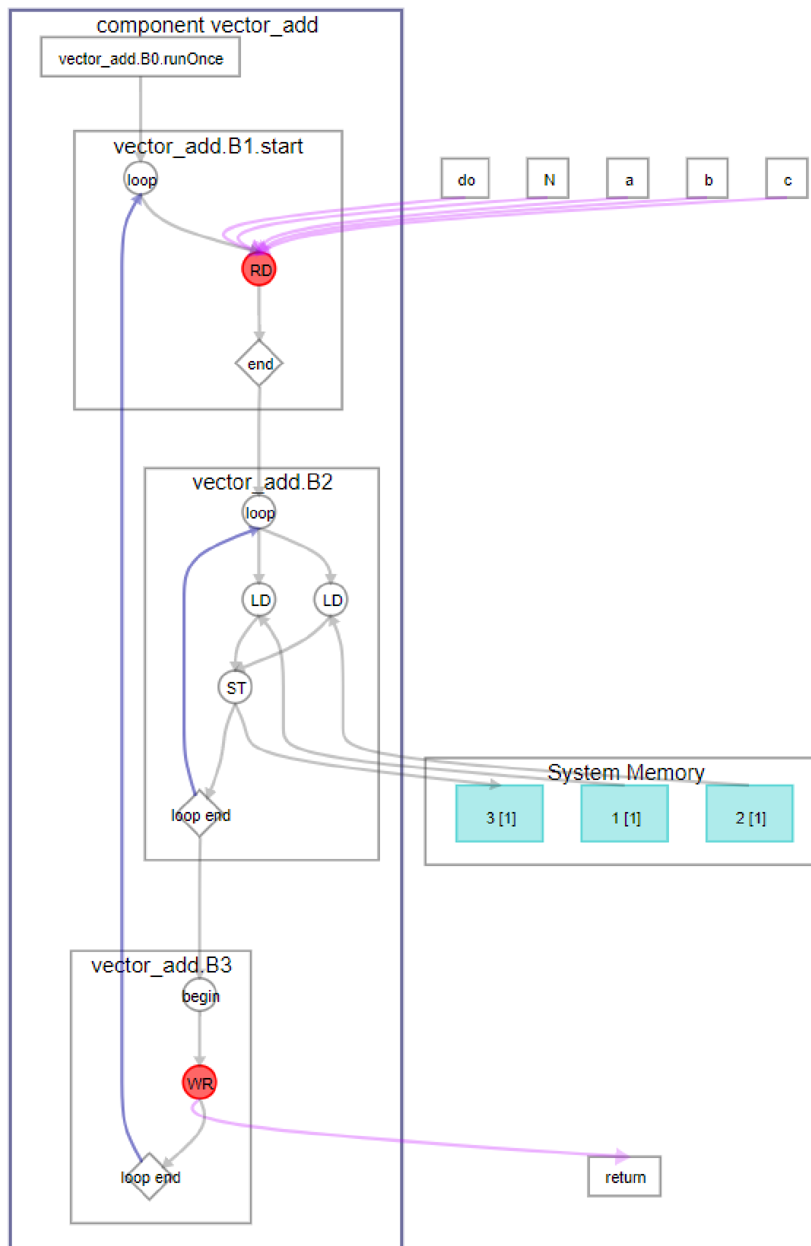
- The vectors are each assigned to different address spaces with the `ihc::aspace` attribute.

  With the vectors assigned to different address spaces, pointer aliasing is prevented and there is no memory arbitration.

- 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.

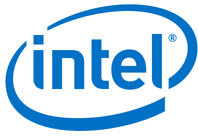
Figure 2. Component View of `vector_add` Component with Avalon-MM Master Interface



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 5 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 Interface[1]

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

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

[2]The f_{max} measurement was calculated from one 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.

2.1.3 Avalon Memory Mapped Slave Interfaces

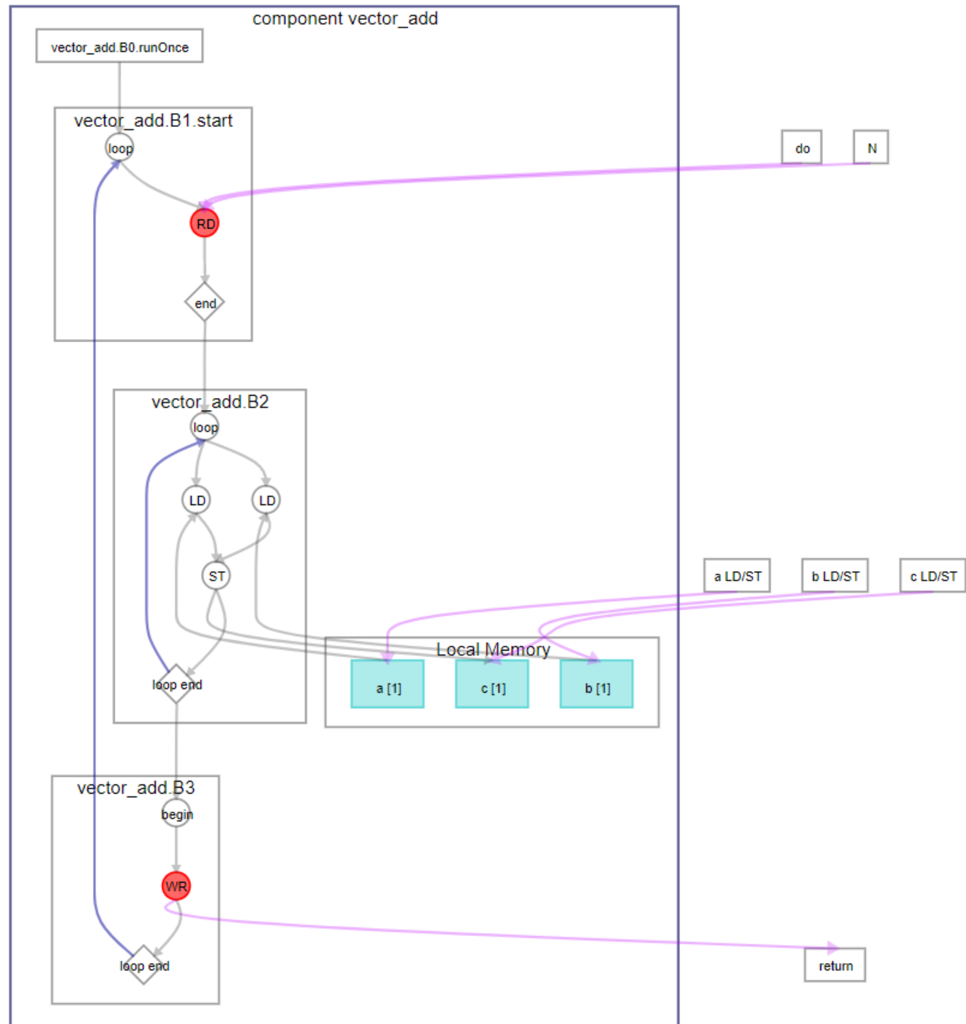
Depending on your component, you can sometimes optimize the memory structure of your component by using Avalon Memory Mapped (Avalon-MM) slave interfaces.

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.

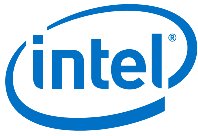
Figure 3. Component View of `vector_add` Component with Avalon-MM Slave Interface



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 Interface[1]

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



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

[2]The f_{max} measurement was calculated from one 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 did not increase. RAM usage shifted from the system to the FPGA RAM blocks.

2.1.4 Avalon Streaming Interfaces

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

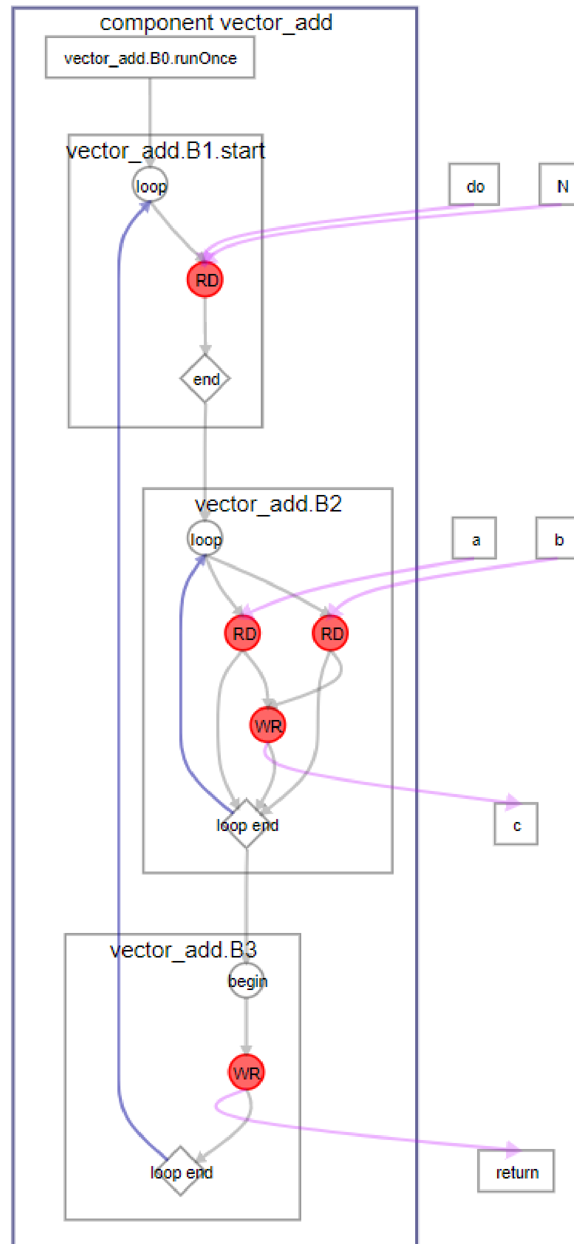
The vector addition example can be coded with an Avalon-ST interface as follows:

```
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 be parallelized 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 one (instead of one). If the component gets no bubbles (gaps in data flow) from upstream or stall signals from downstream, then the component achieves the desired II of 1.

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 Interface[1]

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

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

[2]The f_{max} measurement was calculated from one 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.

2.1.5 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.

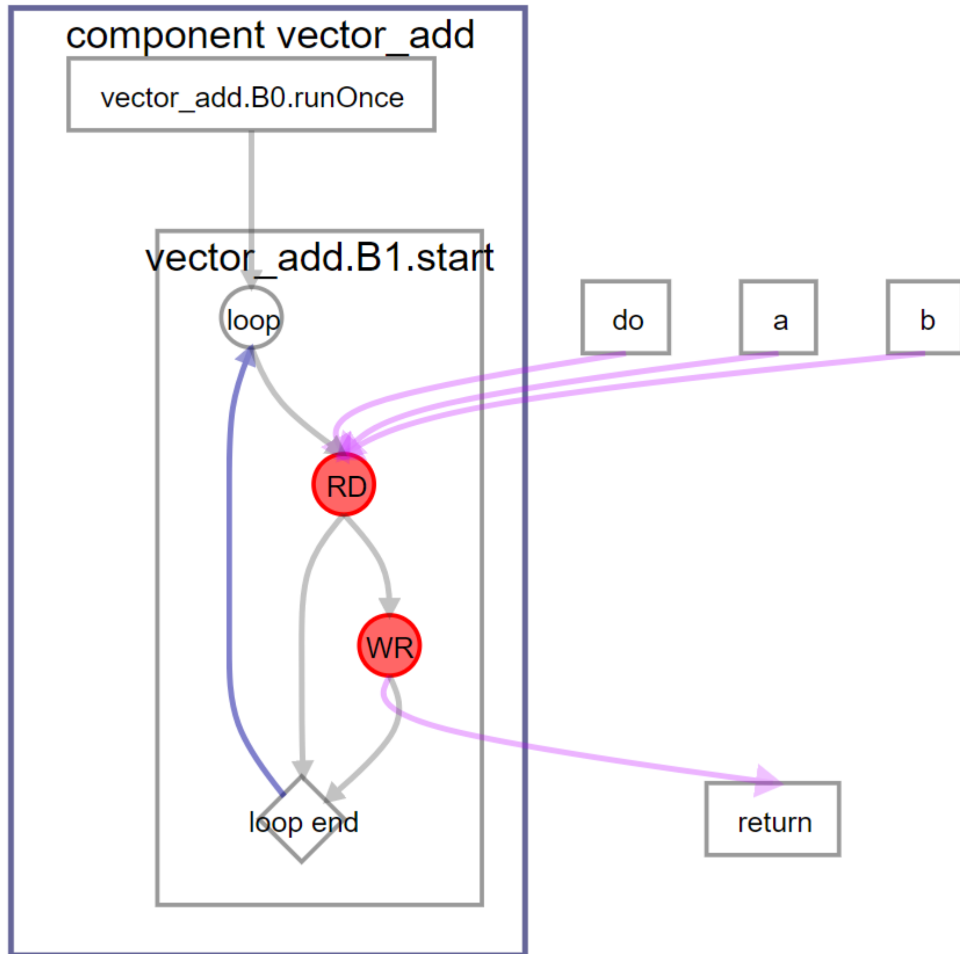
The vector addition example can be coded to pass the vector array elements by value as follows:

```
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 `b`. To compute 1024 elements for the example, the component needs to be called 128 times (1024/8). While in previous examples the component loops were pipelined, here the components calls 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.

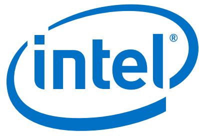
Compiling this component with an Intel Quartus Prime compilation flow targeting an Intel Arria 10 device results in the following QoR metrics:

Table 5. QoR Metrics Comparison for Pass-by-Value Interface[1]

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

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

[2]The f_{max} measurement was calculated from one seed.

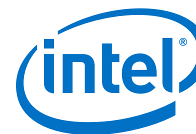


The QoR metrics for the vector_add component with a pass-by-value interface shows fewer ALM used, a high component f_{max}, and optimal values for latency and II. In this case, the II is the same as the component invocation interval. A new invocation of the component can be launched every clock cycle.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

2.2 Avoid Pointer Aliasing

Apply the 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.

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 dependencies between the two arrays, and extracts less pipeline parallelism as a result.



3 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.

Additionally, the compiler informs you of dependencies that prevent the optimization of your loops. Try to eliminate these dependencies in your code for optimal component performance.

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: `<quartus_installdir>/hls/examples/tutorials`	
`best_practices/loop_memory_dependency`	Demonstrates breaking loop carried dependencies using the `ivdep` pragma.
`best_practices/resource_sharing_filter`	Demonstrates the following versions of a 32-tap finite impulse response (FIR) filter design: • optimized-for-throughput variant • optimized-for-area variant

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

ISO 9001:2008 Registered

3.1 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 consume additional hardware resources in exchange for lower latency.

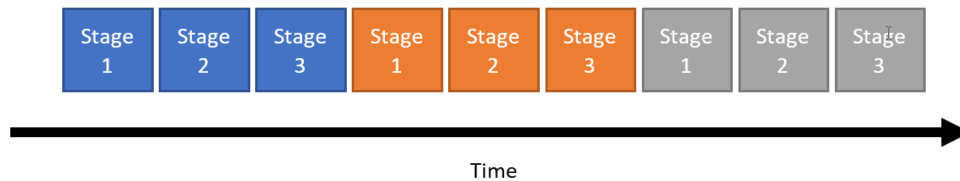
You can take advantage of the spatial compute structure to accelerate the loops by having multiple iterations of a loop executing concurrently by unrolling loops when possible and structuring your loops with independent stages so that loops can be pipelined.

3.1.1 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. Each stage represents the operations that occur in the loop within one clock cycle.

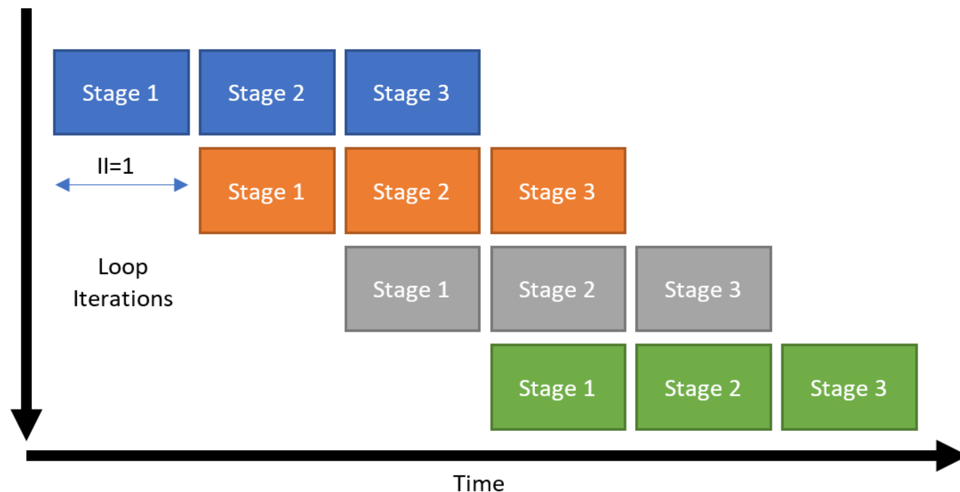
Figure 6. Basic loop with three stages and three iterations



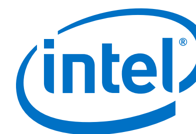
If each stage of this loop takes one clock cycle to execute, then this loop has a latency of nine cycles.

The following figure shows the pipelining of the loop from Figure 6 on page 18.

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 18, 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.

Loops with loop-carried dependencies cannot be pipelined with an II of 1.

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:

$$latency_{loop} = (iterations - 1) * II + latency_{body}$$

where $latency_{loop}$ is the overall latency of the loop and $latency_{body}$ is the latency of the code within the loop.

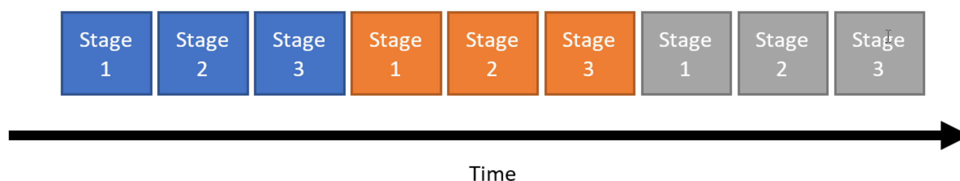
For nested loops, you must apply this formula recursively. This recursion means that having II>1 is more problematic for inner loops than for outer loops. Therefore, algorithms that do most of their work on an inner loop with II=1 still perform well, even if their outer loops have II>1.

3.1.2 Unroll Loops

When a loop is unrolled, each iteration of the loop is replicated in hardware and executes simultaneously as long as the iterations are independent. Unrolling loops trades an increase in FPGA area utilization 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.

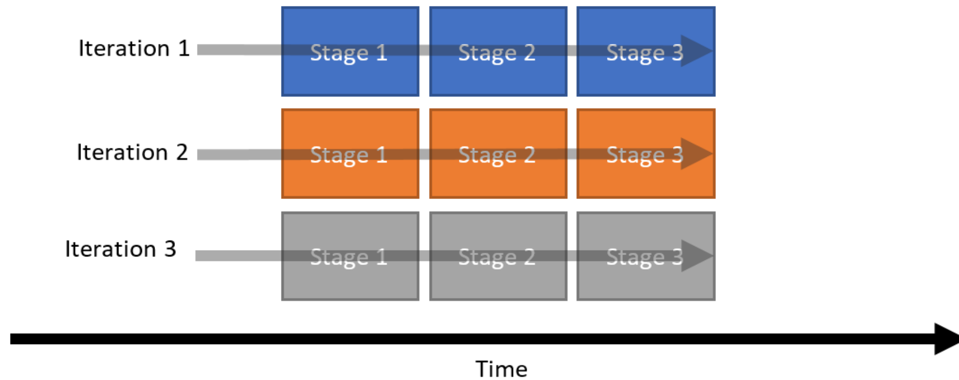
Figure 8. Basic loop with three stages and three iterations



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 19 unrolled three times.

Figure 9. Unrolled loop with three stages and three iterations

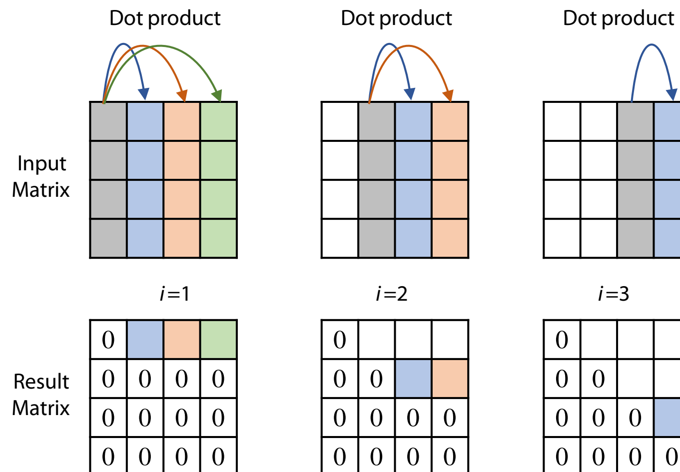


Three iterations of the loop can now be completed in only three clock cycles, but three times as many hardware resources are required.

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.

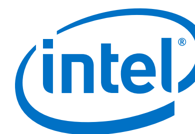
3.1.3 Example: Loop Pipelining and Unrolling

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. If you use the `--fp-relaxed` compiler flag to relax the ordering of floating-point operations, then all of the multiplications in this loop occur in parallel, and the accumulation is implemented as an adder tree. 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 will be 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 default, the compiler will not unroll this loop, even if it does have a constant number of iterations. 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

3.2 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.

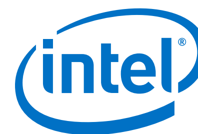
The following example is a well-formed loop:

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



3.3 Minimize Loop-Carried Dependencies

Loop-carried dependencies occur when code in a loop iteration depends on output of previous loop iterations. Loop-carried dependencies in your component decrease the extent of pipeline parallelism that the Intel HLS Compiler can achieve, 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 decreases the extent of pipeline parallelism that the Intel HLS Compiler can achieve, which reduces kernel 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.

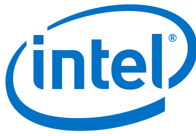
- Multiple index variables in the same subscript location.

  For example, `A[i + 2 × 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 constant or a nonconstant variable.



Use Loops with Constant Bounds Whenever Possible

Range analysis can be performed effectively when loops have constant bounds.

3.4 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.

3.5 Convert Nested Loops into a Single Loop

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

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

<table>
<tr><th>Nested Loop</th><th>Converted Single Loop</th></tr>
<tr><td>

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

</td><td>

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

</td></tr>
</table>

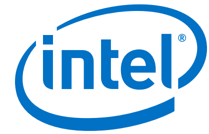
You can also specify the `loop_coalesce` pragma to coalesce nested loops into a single loop without affecting the loop functionality. The following simple example shows how the compiler coalesces two loops into a single loop when you specify the `loop_coalesce` pragma.

Consider a simple nested loop written as follows:

```
#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*.

3.6 Declare Variables in the Deepest Scope Possible

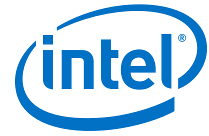
To reduce the FPGA hardware resources necessary for implementing a variable, declare the variable just before you use it in a loop. Declaring variables in the deepest scope possible minimizes data dependencies and FPGA hardware usage because the Intel HLS Compiler does not need to preserve the variable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



4 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 manual controls 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: `<quartus_installdir>/hls/examples/tutorials`	
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.

4.1 Example: Overriding a Coalesced Memory Architecture

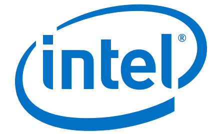
Using memory attributes in various combinations in your code allows you to override the memory architecture that the Intel HLS Compiler infers for your component.

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)`

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

ISO 9001:2008 Registered



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(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 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(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.

4.2 Example: Overriding a Banked Memory Architecture

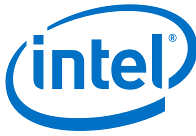
Using memory attributes in various combinations in your code allows you to override the memory architecture that the Intel HLS Compiler infers for your component.

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 have the option to implement one bank of double-pumped 32-bit wide on-chip memory block instead 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 with stallable arbitration by adding the following attributes before the declaration of `data[1024]`:

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

4.3 Merge Memories to Reduce Area

In some case, you can save FPGA memory blocks by merging your component memories so that they consume fewer M20K 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.

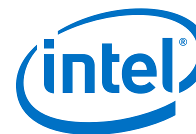
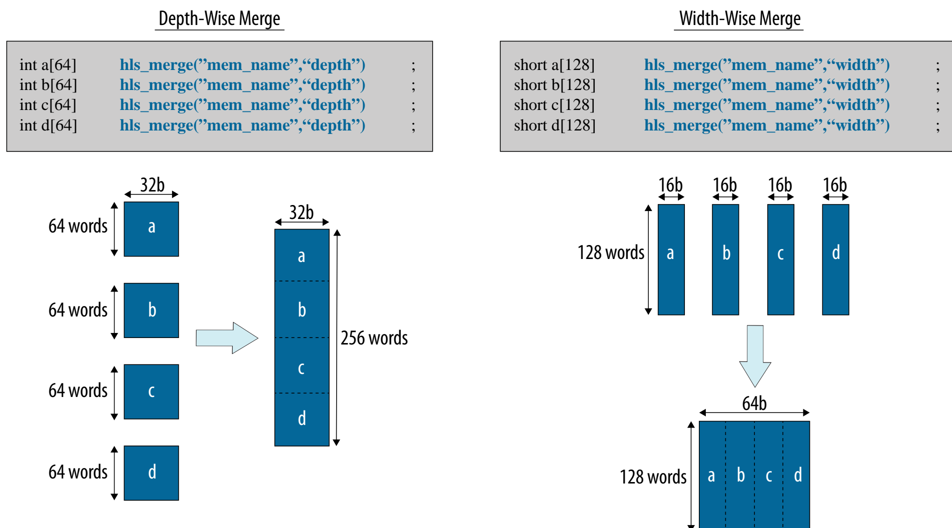


Figure 10. Overview of width-wise merge and depth-wise merge

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



4.3.1 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.

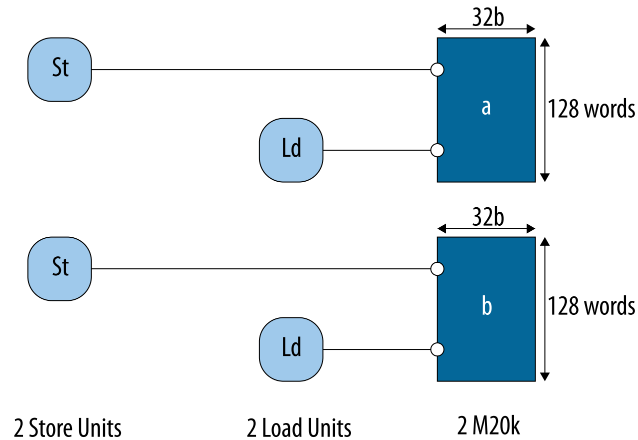
All variables with the same *<mem_name>* label set in their `hls_merge` attributes are merged.

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.

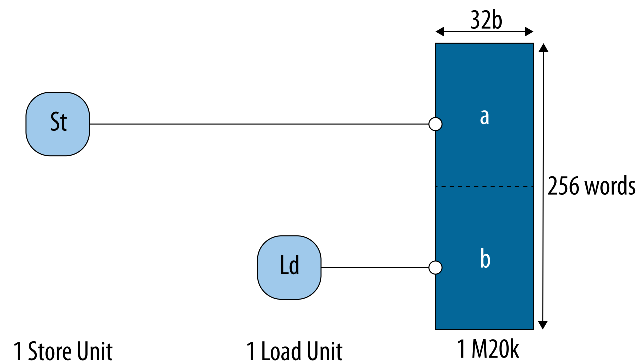
Figure 11. Implementation of Local Memory for Component `depth_manual`

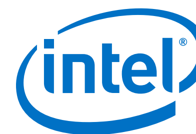


Because the load and store instructions for local memories `a` and `b` are mutually exclusive, you can merge the accesses, as shown in the example code below, which reduces the number of load and store instructions, and the number of on-chip memory blocks, by half.

```
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 12. Depth-Wise Merge of Local Memories for Component `depth_manual`



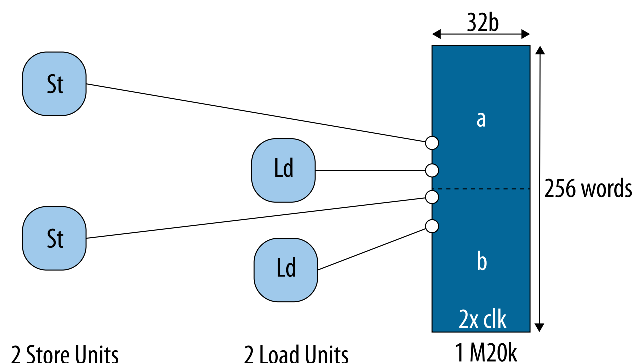


There are cases where merging local memories with respect to depth might degrade memory access efficiency. Prior to deciding whether to merge the local memories with respect to depth, refer to the HLD report (`<result>.prj/reports/report.html`) to ensure that they have produced the expected memory configuration 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 13. Local Memories for Component `depth_manual` with Non-Mutually Exclusive Accesses



4.3.2 Example: Merging Memories Width-Wise

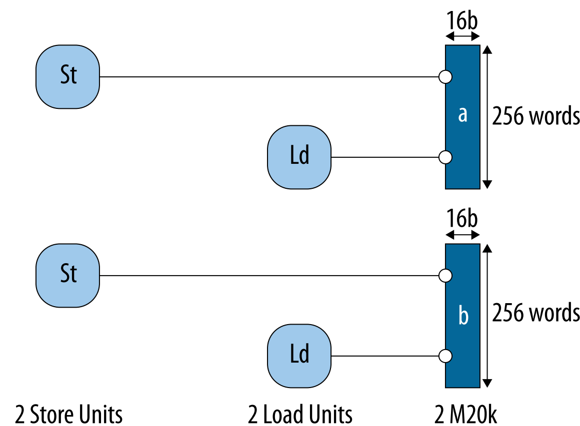
Use the `merger("<mem_name>","width")` attribute to force the Intel HLS Compiler to implement variables in the same memory system, merging their memories by depth.

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

Consider the following component code:

```
component 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 14. Implementation of Local Memory for Component `width_manual`



In this case, the Intel HLS Compiler can merge the load and store instructions to local memories `a` and `b` because their accesses are to the same address, meaning that the load and store instructions can be coalesced, 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;
}
```

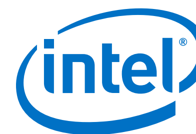
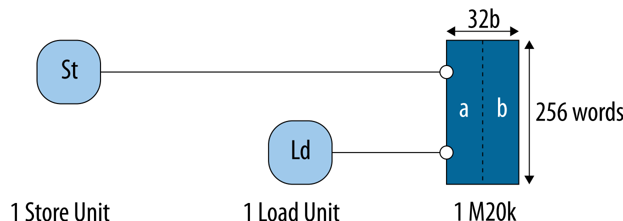


Figure 15. Width-Wise Merge of Local Memories for Component `width_manual`



4.4 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 {b0, b1, ... ,bn} arguments refer to the word-address bits that the Intel HLS Compiler should use for the bank-select bits. Specifying the `hls_bankbits(`*b0*`,` *b1*`, ...,` *bn*`)` attribute implies that the number of banks equals 2^(*number of bank bits*).

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

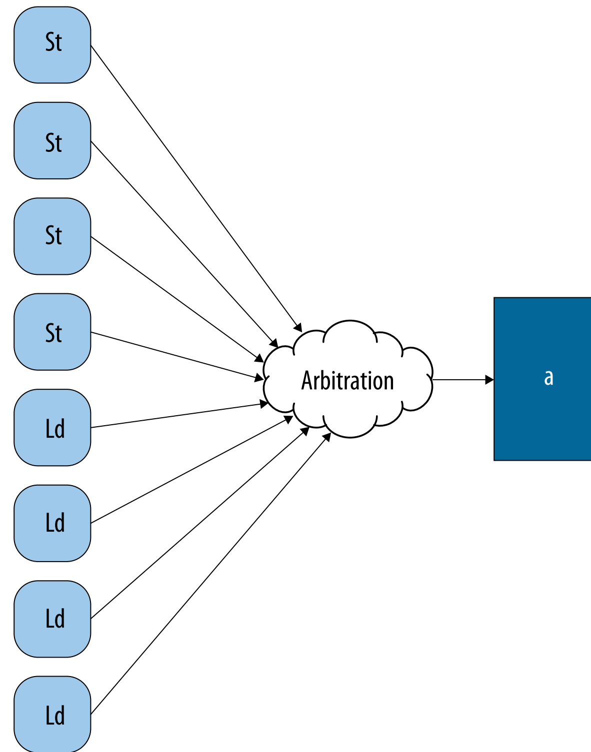
Example of Implementing the `hls_bankbits` Attribute

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];

  #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 figure below, this code example generates multiple load and store instructions. However, there are insufficient number of ports for the number of instructions, leading to stallable accesses and an II value of 66.

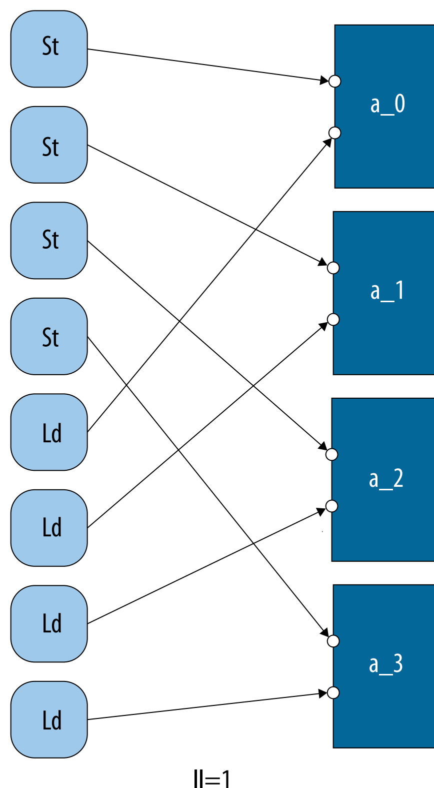
Figure 16. Accesses to Local Memory a for Component bank_arb_consecutive_multidim



By specifying the hls_bankbits attribute, you have control in the manner in which load and store instructions access local memory. As shown in the modified code example and figure below, when you choose constant bank-select bits for each access to the local memory a, each pair of load and store instructions only needs to connect to one memory bank. This access pattern drastically reduces the II value from 66 to 1.

```
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;
}
```

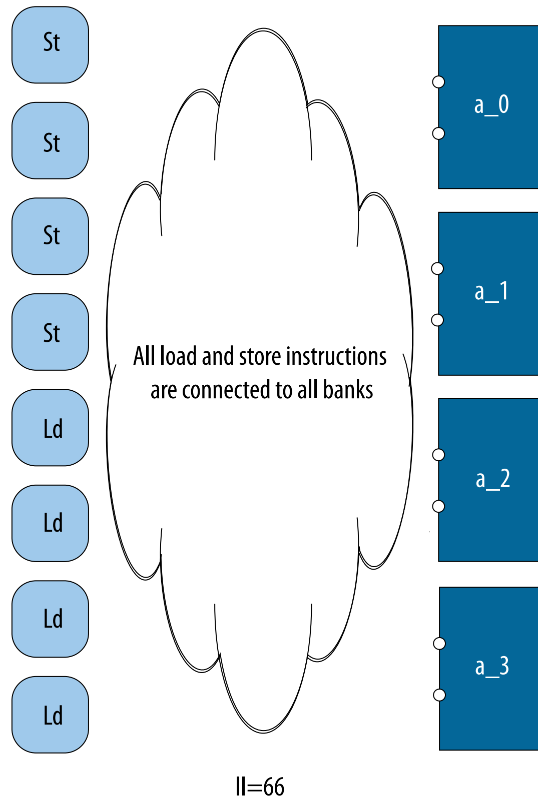
Figure 17. Accesses to Local Memory a for Component `bank_arb_consecutive_multidim` with the `hls_bankbits` Attribute

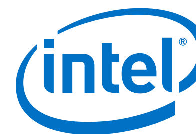


When specifying the word-address bits for the `hls_bankbits` attribute, ensure that the resulting bank-select bits will be constant for each access to local memory. As shown in the example below, the local memory access pattern is such that there is no guarantee that the chosen bank-select bits will be constant for each access. As a result, each pair of load and store instructions must connect to all the local memory banks, leading to stallable accesses.

```
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;
}
```

Figure 18. Stallable Accesses to Local Memory a for Component `bank_arb_consecutive_multidim` with the `hls_bankbits` Attribute





5 Datatype Best Practices

The datatypes in your component and possible conversions that might occur to them 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.

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.

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 math 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: `<quartus_installdir>/hls/examples/tutorials`	
`best_practices/single_vs_double_precision_math`	Demonstrates the effect of using single precision literals and functions instead of double precision literals and functions.

5.1 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-precisions variables are not needed. In FPGAs, using double-precision variables can negatively affect the data transfer rate, the latency, and FPGA resource utilization of your component.

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

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

ISO 9001:2008 Registered

5.2 Avoid Negative Bit Shifts When Using the `ac_int` Datatype

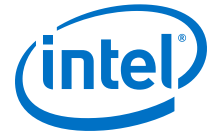
The `ac_int` datatype differs from other languages, including C and Verilog, in bit shifting. By default, if the shift amount is of a signed datatype `ac_int` allows negative shifts.

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);
}
```



A Document Revision History

Table 6. Document Revision History of the Intel High Level Synthesis Compiler Best Practices Guide

Date	Version	Changes
December 2017	2017.12.22	• Added Choose the Right Interface for Your Component on page 5 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 4.
November 2017	2017.11.06	Initial release. 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.

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

ISO 9001:2008 Registered