

Vitis High-Level Synthesis User Guide

UG1399 (v2022.2) December 7, 2022

Xilinx is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this [link](#) for more information.



Table of Contents

Section I: Introduction.....	11
Navigating Content by Design Process.....	11
Benefits of High-Level Synthesis.....	12
Introduction to Vitis HLS.....	14
Re-Architecting the Design Code.....	17
Tutorials and Examples.....	21
Section II: HLS Programmers Guide.....	22
 Chapter 1: Design Principles.....	23
Three Paradigms for Programming FPGAs.....	24
Combining the Three Paradigms.....	31
Conclusion - A Prescription for Performance.....	35
 Chapter 2: Abstract Parallel Programming Model for HLS.....	38
Control and Data Driven Tasks.....	40
Data-driven Task-level Parallelism.....	41
Control-driven Task-level Parallelism.....	44
Mixing Data-Driven and Control-Driven Models.....	55
Summary.....	57
 Chapter 3: Loops Primer.....	59
Pipelining Loops.....	59
Unrolling Loops.....	71
Merging Loops.....	72
Working with Nested Loops.....	74
Working with Variable Loop Bounds.....	77
 Chapter 4: Arrays Primer.....	80
Mapping Software Arrays to Hardware Memory.....	80
Array Accesses and Performance.....	81
Arrays on the Interface.....	86

Initializing and Resetting Arrays.....	90
Implementing ROMs.....	92
C Simulation with Arrays.....	93
Chapter 5: Functions Primer.....	95
Function Inlining.....	95
Function Pipelining.....	96
Function Instantiation.....	97
Chapter 6: Data Types.....	99
Standard Types.....	100
Composite Data Types.....	105
Arbitrary Precision (AP) Data Types.....	121
Global Variables.....	127
Pointers.....	128
Vector Data Types.....	139
Bit-Width Propagation.....	140
Chapter 7: Unsupported C/C++ Constructs.....	141
System Calls.....	141
Dynamic Memory Usage.....	142
Pointer Limitations.....	144
Recursive Functions.....	144
Undefined Behaviors.....	145
Virtual Functions and Pointers.....	146
Chapter 8: Interfaces of the HLS Design.....	147
Defining Interfaces.....	147
Vitis HLS Memory Layout Model.....	207
Execution Modes of HLS Designs.....	219
Controlling Initialization and Reset Behavior.....	229
Chapter 9: Best Practices for Designing with M_AXI Interfaces.	232
Chapter 10: Optimizing Techniques and Troubleshooting Tips...	235
Understanding High-Level Synthesis Scheduling and Binding.....	237
Optimizing Logic.....	244
Optimizing AXI System Performance.....	246
Managing Area and Hardware Resources	280

Unrolling Loops in C++ Classes.....	283
Limitations of Control-Driven Task-Level Parallelism.....	284
Limitations of Pipelining with Static Variables.....	289
Section III: Using Vitis HLS.....	291
Chapter 11: Launching Vitis HLS.....	292
Setting Up the Environment.....	293
Overview of the Vitis HLS IDE.....	293
Chapter 12: Creating a New Vitis HLS Project.....	298
Vitis HLS Flow Overview.....	305
Working with Sources.....	307
Setting Configuration Options.....	328
Specifying the Clock Frequency.....	331
Using the Flow Navigator.....	333
Chapter 13: Verifying Code with C Simulation.....	335
<code>hls::print</code> Function.....	338
Writing a Test Bench.....	339
Using the Debug View Layout.....	346
Output of C Simulation.....	347
Pre-Synthesis Control Flow.....	347
Chapter 14: Synthesizing the Code.....	350
Synthesis Summary.....	352
Output of C Synthesis.....	359
Improving Synthesis Runtime and Capacity.....	360
Chapter 15: Analyzing the Results of Synthesis.....	361
Schedule Viewer	361
Function Call Graph Viewer.....	365
Dataflow Viewer.....	367
Timeline Trace Viewer.....	369
Chapter 16: Optimizing the HLS Project.....	372
Creating Additional Solutions.....	372
Adding Pragmas and Directives.....	374

Chapter 17: C/RTL Co-Simulation in Vitis HLS.....	380
Output of C/RTL Co-Simulation.....	383
Automatically Verifying the RTL.....	384
Analyzing RTL Simulations.....	388
Cosim Deadlock Viewer.....	390
Debugging C/RTL Co-Simulation.....	392
Chapter 18: Exporting the RTL Design.....	396
Running Implementation.....	399
Implementation Report.....	401
Output of RTL Export.....	403
Archiving the Project.....	404
Chapter 19: Running Vitis HLS from the Command Line.....	406
Section IV: Vitis HLS Command Reference.....	408
Chapter 20: vitis_hls Command.....	409
hls_init.tcl.....	410
Chapter 21: Project Commands.....	411
add_files.....	411
cat_ini.....	413
close_project.....	414
close_solution.....	414
cosim_design.....	415
cosim_stall.....	417
create_clock.....	418
csim_design.....	419
csynth_design.....	420
delete_project.....	420
delete_solution.....	421
enable_beta_device.....	422
export_design.....	422
get_clock_period.....	424
get_clock_uncertainty.....	425
get_files.....	425
get_part.....	426

get_project.....	426
get_solution.....	427
get_top.....	428
help.....	428
list_part.....	429
open_project.....	430
open_solution.....	431
open_tcl_project.....	432
set_clock_uncertainty.....	433
set_part.....	434
set_top.....	435
Chapter 22: Configuration Commands.....	436
config_array_partition.....	436
config_compile.....	437
config_cosim.....	439
config_csim.....	441
config_dataflow.....	442
config_debug.....	444
config_export.....	444
config_interface.....	447
config_op.....	450
config_rtl.....	453
config_schedule.....	455
config_storage.....	455
config_unroll.....	456
Chapter 23: Optimization Directives.....	457
set_directive_aggregate.....	457
set_directive_alias.....	458
set_directive_allocation.....	460
set_directive_array_partition.....	461
set_directive_array_reshape.....	464
set_directive_bind_op.....	466
set_directive_bind_storage.....	470
set_directive_dataflow.....	473
set_directive_dependence.....	475
set_directive_disaggregate.....	477

set_directive_expression_balance.....	479
set_directive_function_instantiate.....	480
set_directive_inline.....	482
set_directive_interface.....	483
set_directive_latency.....	488
set_directive_loop_flatten.....	489
set_directive_loop_merge.....	491
set_directive_loop_tripcount.....	492
set_directive_occurrence.....	494
set_directive_performance.....	495
set_directive_pipeline.....	496
set_directive_protocol.....	498
set_directive_reset.....	499
set_directive_stable.....	500
set_directive_stream.....	501
set_directive_top.....	503
set_directive_unroll.....	504
Chapter 24: HLS Pragmas.....	506
pragma HLS aggregate.....	507
pragma HLS alias.....	508
pragma HLS allocation.....	510
pragma HLS array_partition.....	512
pragma HLS array_reshape.....	514
pragma HLS bind_op.....	517
pragma HLS bind_storage.....	521
pragma HLS dataflow.....	525
pragma HLS dependence.....	527
pragma HLS disaggregate.....	530
pragma HLS expression_balance.....	532
pragma HLS function_instantiate.....	533
pragma HLS inline.....	534
pragma HLS interface.....	537
pragma HLS latency.....	543
pragma HLS loop_flatten.....	545
pragma HLS loop_merge.....	547
pragma HLS loop_tripcount.....	549
pragma HLS occurrence.....	550

pragma HLS performance.....	552
pragma HLS pipeline.....	553
pragma HLS protocol.....	556
pragma HLS reset.....	557
pragma HLS stable.....	558
pragma HLS stream.....	559
pragma HLS top.....	561
pragma HLS unroll.....	562

Section V: Vitis HLS C Driver Reference.....566

Chapter 25: AXI4-Lite Slave C Driver Reference.....567

X<DUT>_Initialize.....	567
X<DUT>_CfgInitialize.....	568
X<DUT>_LookupConfig.....	568
X<DUT>_Release.....	569
X<DUT>_Start.....	569
X<DUT>_IsDone.....	569
X<DUT>_IsIdle.....	570
X<DUT>_IsReady.....	570
X<DUT>_Continue.....	570
X<DUT>_EnableAutoRestart.....	571
X<DUT>_DisableAutoRestart.....	571
X<DUT>_Set_ARG.....	571
X<DUT>_Set_ARG_vld.....	572
X<DUT>_Set_ARG_ack.....	572
X<DUT>_Get_ARG.....	573
X<DUT>_Get_ARG_vld.....	573
X<DUT>_Get_ARG_ack.....	573
X<DUT>_Get_ARG_BaseAddress.....	574
X<DUT>_Get_ARG_HighAddress.....	574
X<DUT>_Get_ARG_TotalBytes.....	575
X<DUT>_Get_ARG_BitWidth.....	575
X<DUT>_Get_ARG_Depth.....	576
X<DUT>_Write_ARG_Words.....	576
X<DUT>_Read_ARG_Words.....	577
X<DUT>_Write_ARG_Bytes.....	577
X<DUT>_Read_ARG_Bytes.....	578
X<DUT>_InterruptGlobalEnable.....	578

X<DUT>_InterruptGlobalDisable.....	579
X<DUT>_InterruptEnable.....	579
X<DUT>_InterruptDisable.....	580
X<DUT>_InterruptClear.....	580
X<DUT>_InterruptGetEnabled.....	580
X<DUT>_InterruptGetStatus.....	581
Section VI: Vitis HLS Libraries Reference.....	582
Chapter 26: C/C++ Builtin Functions.....	583
Chapter 27: Arbitrary Precision Data Types Library.....	584
Using Arbitrary Precision Data Types.....	584
C++ Arbitrary Precision Integer Types.....	587
C++ Arbitrary Precision Fixed-Point Types.....	608
Chapter 28: Vitis HLS Math Library.....	636
HLS Math Library Accuracy	636
HLS Math Library.....	638
Fixed-Point Math Functions.....	639
Verification and Math Functions.....	642
Common Synthesis Errors.....	645
Chapter 29: HLS Stream Library.....	647
C Modeling and RTL Implementation.....	648
Using HLS Streams.....	649
Chapter 30: HLS Vector Library.....	657
Chapter 31: HLS Task Library.....	660
Tasks and Channels.....	661
Tasks and Dataflow.....	663
Chapter 32: HLS Split/Merge Library.....	666
Chapter 33: HLS Stream of Blocks Library.....	671
Stream-of-Blocks Modeling Style.....	672
Checking for Full and Empty Blocks.....	675
Modeling Feedback in Dataflow Regions.....	677

Limitations.....	678
Chapter 34: HLS IP Libraries.....	680
FFT IP Library.....	680
FIR Filter IP Library.....	690
DDS IP Library.....	698
SRL IP Library.....	703
Section VII: Vitis HLS Migration Guide.....	706
Chapter 35: Migrating to Vitis HLS.....	707
Key Behavioral Differences.....	707
Chapter 36: Deprecated and Unsupported Features.....	714
Chapter 37: Unsupported Features	719
Assertions.....	719
Pragmas.....	719
HLS Video Library.....	720
C Arbitrary Precision Types	720
Appendix A: Additional Resources and Legal Notices.....	721
Xilinx Resources.....	721
Documentation Navigator and Design Hubs.....	721
References.....	721
Revision History.....	722
Please Read: Important Legal Notices.....	725

Introduction

This section contains the following chapters:

- [Navigating Content by Design Process](#)
- [Section II: HLS Programmers Guide](#)
- [Vitis HLS Flow Overview](#)

Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal® ACAP design process [Design Hubs](#) and the [Design Flow Assistant](#) materials can be found on the [Xilinx.com](#) website. This document covers the following design processes:

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the Vivado® timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
 - [Launching Vitis HLS](#)
 - [Verifying Code with C Simulation](#)
 - [Synthesizing the Code](#)
 - [Analyzing the Results of Synthesis](#)
 - [Optimizing the HLS Project](#)

Benefits of High-Level Synthesis

High-Level Synthesis is an automated design process that takes an abstract behavioral specification of a digital system and generates a register-transfer level structure that realizes the given behavior.

A typical flow using High-Level Synthesis has the following steps:

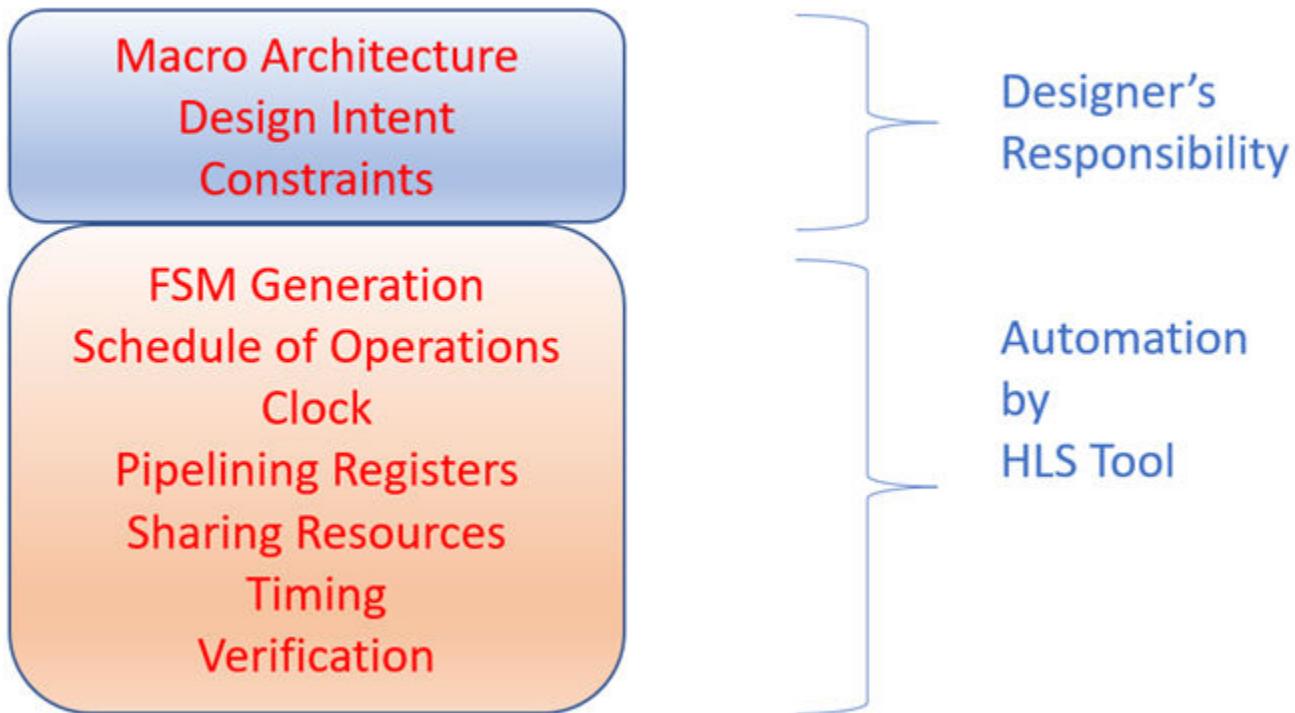
1. Write the algorithm at a high abstraction level using C/C++/SystemC with a given architecture in mind
2. Verify the functionality at the behavioral level
3. Use the HLS tool to generate the RTL for a given clock speed, input constraints
4. Verify the functionality of the generated RTL
5. Explore different architectures using the same input source code

HLS can enable the path of creating high-quality RTL, rather quickly than manually writing error-free RTL.

The designer needs to create the macro-architecture of the algorithm in C/C++ at a high level, meaning that the design intent and how this design interacts with the outside world should be carefully thought through. HLS tool also requires input constraints like clock period, performance constraints, etc.

Micro-architecture decisions like creating the state machine, datapath, register pipelines, etc are not needed at a high level. These details can be left to the HLS tool and optimized RTL can be generated by providing input constraints like clock speed, performance pragmas, target device, etc.

Figure 1: Design Processes



Using the defined macro-architecture of the C/C++ algorithm, designers can also vary constraints to generate multiple RTL solutions to explore trade-offs between performance and area. So a single algorithm can lead to multiple implementations, allowing designers to choose an implementation that best meets the needs of the overall application.

Improve Productivity

With HLS, the designer is working on a high abstraction level, meaning fewer lines of code will need to be written as input to HLS. Due to less time spent on writing the C++ code and quicker turnaround, less error-prone thus increasing overall design productivity. The designers can focus more time on creating efficient designs at a higher level than worrying about mechanical RTL implementation tasks.

HLS not only enables high design productivity but also verification productivity. With HLS, the testbench is also generated or created at a high level, meaning the original design intent can be verified very quickly. The designer can explore quick turnarounds of verified algorithms as the flow is still within the C/C++ domain. Once the algorithm is verified in C/C++, the same testbench can be used for generated RTL by the HLS tool. Nevertheless, the generated RTL can be integrated with the existing RTL verification flow for more comprehensive verification coverage.

The design and verification benefits of using high-level synthesis (HLS) are summarized here:

- Developing and validating algorithms at the C-level for the purpose of designing at an abstract level from the hardware implementation details.
- Using C-simulation to validate the design, and iterate more quickly than with traditional RTL design.
- Creating multiple design solutions from the C source code and pragmas to explore the design space, and find an optimal solution.

Enable Re-Use

The designs created for High-level synthesis are generic and unaware of implementation. These sources are not tied to any technology node or any given clock period like a given RTL. With few updates of input constraints and without any source code changes, multiple architectures can be explored. A similar practice with the RTL is not pragmatic. The designers create the RTL for a given clock period and any change for a derivative product, however small it is leads to a new complex project. Working at a higher level with HLS, designers don't need to worry about the micro-architecture and can rely on the HLS tool to regenerate new RTL automatically.

Introduction to Vitis HLS

The Vitis HLS tool synthesizes a C or C++ function into RTL code for implementation in the programmable logic (PL) region of a Versal ACAP, Zynq MPSoC, or Xilinx FPGA device. Vitis HLS is tightly integrated with both the Vivado Design Suite for synthesis, place, and route, and the Vitis core development kit for heterogeneous system-level design and application acceleration.

Vitis HLS can be used to develop and export:

- Vivado IP to be integrated into hardware designs using the Vivado Design Suite
- Vitis kernels for use in the Vitis application acceleration development flow



TIP: The Vitis kernel (.xo) is a Vivado IP with specific requirements and limitations as described in [Interfaces for Vitis Kernel Flow](#); while Vivado IP have few restrictions and offer greater design flexibility as described in [Interfaces for Vivado IP Flow](#).

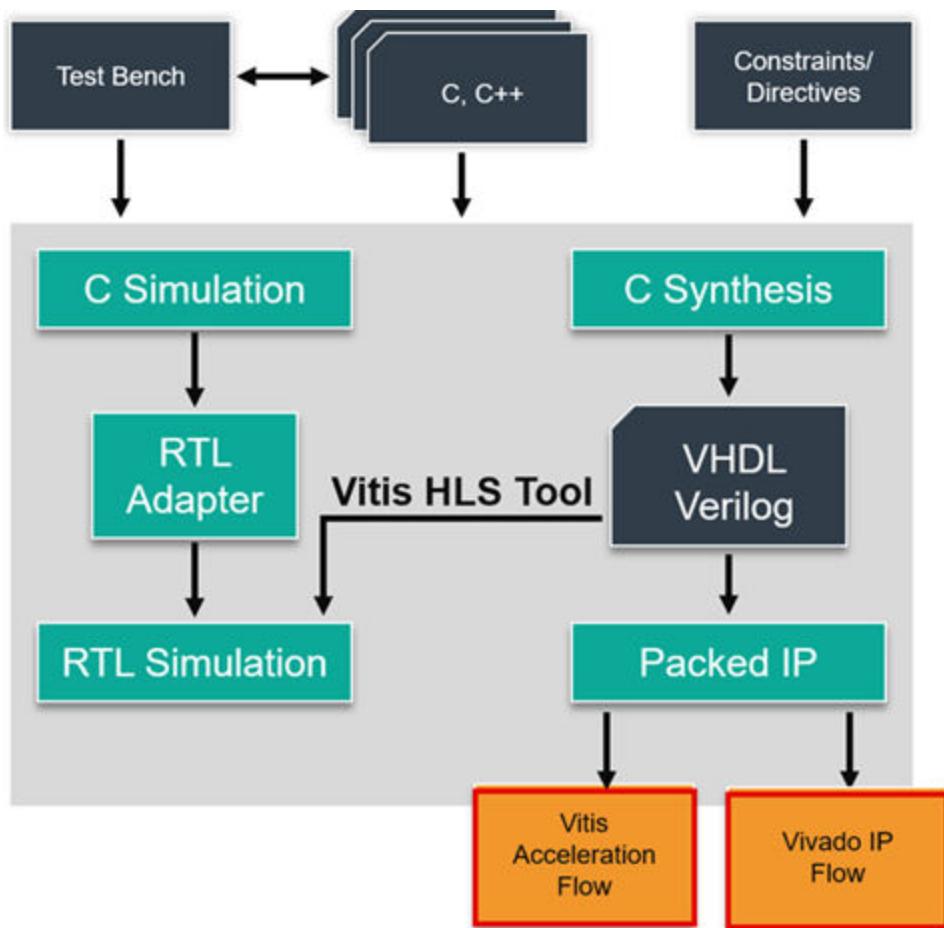
In the Vitis application acceleration flow, the Vitis HLS tool automates much of the code modifications required to implement and optimize the C/C++ code in programmable logic and to achieve low latency and high throughput. The inference of required pragmas to produce the right interface for your function arguments and to pipeline loops and functions within your code is the foundation of Vitis HLS.

In the Vivado IP flow, Vitis HLS also supports customization of your code to implement broader interface standards to achieve your design objectives. The RTL generated can be used as an IP directly within the Vivado tool or Model composer.

Here are the steps for the development of the C++ function.

1. Architect the algorithm based on the [Design Principles](#)
2. (C-Simulation) Verify the C/C++ Code with the C/C++ testbench
3. (C-Synthesis) Generate the RTL using HLS
4. (Co-Simulation) Verify the kernel generated with C++ outputs
5. (Analyze) Review the HLS synthesis reports and co-simulation reports, analyze
6. Re-run previous steps until performance goals are met.

Figure 2: Vitis HLS Development Flow



Vitis HLS implements the solution based on the target flow, default tool configuration, design constraints, and any optimization pragmas or directives you specify. You can use optimization directives to modify and control the implementation of the internal logic and I/O ports, overriding the default behaviors of the tool.

Vitis HLS generates Vivado IP or Vitis kernel based on the target flow, default tool configuration, design constraints, and any optimization pragmas or directives you specify. You can use optimization directives to modify and control the implementation of the internal logic and I/O ports, overriding the default behaviors of the tool.

Here are some key areas related to coding and synthesizing the C++ functions in your HLS design with details covered in forthcoming sections:

- **Hardware Interfaces:** The arguments of the top-level function in a Vitis HLS design are synthesized into interfaces and ports that group multiple signals to define the communication protocol between the HLS design and components external to the design. Vitis HLS defines interfaces automatically, using industry standards to specify the protocol used. The default interface protocols differ based on whether the HLS design is targeting for Vivado IP generation or the Vitis kernel. The default assignments of the interfaces can be overridden by using the INTERFACE pragma or directive.
- **Controlling the Execution of HLS Design:** The execution mode of an HLS design is specified by the block-level control protocol. The HLS design can have control signals to start/stop the execution or it can be only driven when the data is available. As a designer, you do need to be aware of how your HLS design will be executed, as described in [Execution Modes of HLS Designs](#)
- **Task-Level Parallelism:**
 - To achieve high performance on the generated hardware, the HLS tool must infer parallelism from sequential code and exploit it to achieve greater performance. The [Design Principles](#) section introduces the three main paradigms that need to be understood for writing good software for FPGA platforms. Vitis HLS tool offers multiple types of task-level parallelism (TLP), either by specifying the DATAFLOW pragma or explicitly creating parallelism using `hls::task` object as described in [Abstract Parallel Programming Model for HLS](#)
- **Memory Architecture:**
 - The memory architecture is fixed in the CPU but the developer can create their own architecture to optimize the memory accesses for running applications on FPGA
 - In C++ program, the arrays are fundamental data structures used to save or move the data around. In hardware, these arrays are implemented as memory or registers after synthesis. The memory can be implemented as local storage or global memory which is often DDR or HBM memory banks. Access to global memory has higher latency costs and can take many cycles while access to local memory is often quick and only takes one or more cycles.
 - Often the memory is allocated/deallocated dynamically in a C++ program but this can't be synthesized in hardware. So the designer needs to be aware of the exact amount of memory required for the algorithm.

- The memory accesses should be optimized to reduce the overhead of global memory accesses. The redundant accesses, which means maximizing the use of consecutive accesses so that bursting can be inferred. The burst access hides the memory access latency and improves the memory bandwidth.
- **Micro Level Optimization:**
 - In C++ programs, there is a frequent need to implement repetitive algorithms that process blocks of data — for example, signal or image processing. Typically, the C/C++ source code tends to include several loops or several nested loops. Vitis HLS can unroll, or pipeline a loop or nested loops by inserting pragmas at appropriate levels in the source code. For more information, refer to the [Loops Primer](#)
 - Once the algorithm is architected based on the design principles, inferring parallelism, you still need the right combination of micro-level HLS pragmas like PIPELINE, UNROLL, ARRAY_PARTITION, etc. These pragmas may not be intuitive to new users. Vitis HLS offers the [PERFORMANCE](#) pragma as a way to specify top-level performance goals for a given body of loop or nested loops. The tool will automatically infer the necessary lower-level pragmas to meet the goal. With PERFORMANCE pragma, fewer pragmas are needed to achieve good QoR and is an intuitive way to drive the tool.

Re-Architecting the Design Code

The following is a simple program that includes a `compute()` function written in C++ for execution on the CPU. The program is similar to any other C++ program where the main function sets up the data to be sent to `compute` function, calls the `compute` function, and checks the results against expected results. The execution of this program is sequential on the CPU. This example will need to be re-architected to achieve significant performance improvement when running on programmable logic.

```
#include <vector>
#include <iostream>
#include <ap_int.h>
#include "hls_vector.h"

#define totalNumWords 512
unsigned char data_t;

int main(int, char**) {
    // initialize input vector arrays on CPU
    for (int i = 0; i < totalNumWords; i++) {
        in[i] = i;
    }
    compute(data_t in[totalNumWords], data_t Out[totalNumWords]);
    check_results();
}

void compute (data_t in[totalNumWords ], data_t Out[totalNumWords ]) {
    data_t tmp1[totalNumWords], tmp2[totalNumWords];
    A: for (int i = 0; i < totalNumWords ; ++i) {
        tmp1[i] = in[i] * 3;
```

```
        tmp2[i] = in[i] * 3;
    }
B: for (int i = 0; i < totalNumWords ; ++i) {
    tmp1[i] = tmp1[i] + 25;
}
C: for (int i = 0; i < totalNumWords ; ++i) {
    tmp2[i] = tmp2[i] * 2;
}
D: for (int i = 0; i < totalNumWords ; ++i) {
    out[i] = tmp1[i] + tmp2[i] * 2;
}
```

This program can also run sequentially on an FPGA, producing correct results without any performance gain compared to the CPU. For the application to execute with higher performance on an FPGA, the program needs to be re-architected to enable parallelism at various levels. Examples of parallelism can include:

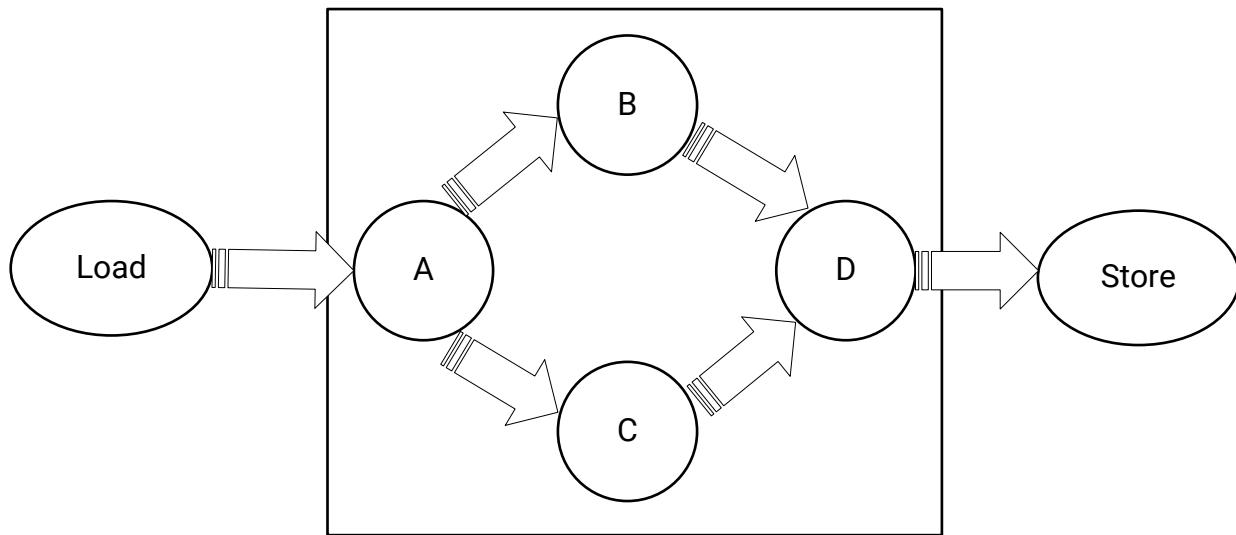
- The compute function can start before all the data is transferred to it
- Multiple compute functions can run in an overlapping fashion, for example a "for" loop can start the next iteration before the previous iteration has completed
- The operations within a "for" loop can run concurrently on multiple words and doesn't need to be executed on a per-word basis

Re-Architecting Kernel Code

From the prior example it is the `compute()` function that needs to be re-architected for FPGA-based acceleration.

The `compute()` function Loop A multiplies an input value with 3 and creates two separate paths, B and C. Loop B and C perform operations and feed the data to D. This is a simple representation of a realistic case where you have several tasks to be performed one after another and these tasks are connected to each other as a network like the one shown below.

Figure 3: Kernel Architecture



X27496-120522

The key takeaways for re-architecting the kernel code are:

- Task-level parallelism is implemented at the function level. To implement task-level parallelism loops are pushed into separate functions. The original `compute()` function is split into multiple sub-functions. As a rule of thumb, sequential functions can be made to execute concurrently, and sequential loops can be pipelined.
- Instruction-level parallelism is implemented by reading 16 32-bit words from memory (or 512-bits of data). Computations can be performed on all these words in parallel. The `hls::vector` class is a C++ template class for executing vector operations on multiple samples concurrently.
- The `compute()` function needs to be re-architected into load-compute-store sub-functions, as shown in the example below. The load and store functions encapsulate the data accesses and isolate the computations performed by the various compute functions.
- Additionally, there are compiler directives starting with `#pragma` that can transform the sequential code into parallel execution.



TIP: This is the [using_fifos](#) example found in the Vitis-HLS Introductory Examples on GitHub.

```
#include "diamond.h"
#define NUM_WORDS 16
extern "C" {

void diamond(vecOf16Words* vecIn, vecOf16Words* vecOut, int size)
{
    hls::stream<vecOf16Words> c0, c1, c2, c3, c4, c5;
    assert(size % 16 == 0);

    #pragma HLS dataflow
    load(vecIn, c0, size);
    compute_A(c0, c1, c2, size);
    compute_B(c1, c2, c3, size);
    compute_C(c2, c3, c4, size);
    compute_D(c3, c4, c5, size);
    store(vecOut, c5, size);
}
```

```
compute_B(c1, c3, size);
compute_C(c2, c4, size);
compute_D(c3, c4,c5, size);
store(c5, vecOut, size);
}

void load(vecOf16Words *in, hls::stream<vecOf16Words >& out, int size)
{
Loop0:
    for (int i = 0; i < size; i++)
    {
        #pragma HLS PERFORMANCE target_tti=32
        #pragma HLS LOOP_TRIPCOUNT max=32
        out.write(in[i]);
    }
}

void compute_A(hls::stream<vecOf16Words >& in, hls::stream<vecOf16Words >&
out1, hls::stream<vecOf16Words >& out2, int size)
{
Loop0:
    for (int i = 0; i < size; i++)
    {
        #pragma HLS PERFORMANCE target_tti=32
        #pragma HLS LOOP_TRIPCOUNT max=32
        vecOf16Words t = in.read();
        out1.write(t * 3);
        out2.write(t * 3);
    }
}
void compute_B(hls::stream<vecOf16Words >& in, hls::stream<vecOf16Words >&
out, int size)
{
Loop0:
    for (int i = 0; i < size; i++)
    {
        #pragma HLS PERFORMANCE target_tti=32
        #pragma HLS LOOP_TRIPCOUNT max=32
        out.write(in.read() + 25);
    }
}

void compute_C(hls::stream<vecOf16Words >& in, hls::stream<vecOf16Words >&
out, int size)
{
Loop0:
    for (data_t i = 0; i < size; i++)
    {
        #pragma HLS PERFORMANCE target_tti=32
        #pragma HLS LOOP_TRIPCOUNT max=32
        out.write(in.read() * 2);
    }
}
void compute_D(hls::stream<vecOf16Words >& in1, hls::stream<vecOf16Words >&
in2, hls::stream<vecOf16Words >& out, int size)
{
Loop0:
    for (data_t i = 0; i < size; i++)
    {
        #pragma HLS PERFORMANCE target_tti=32
        #pragma HLS LOOP_TRIPCOUNT max=32
    }
}
```

```
        out.write(in1.read() + in2.read());
    }

void store(hls::stream<vecOf16Words >& in, vecOf16Words *out, int size)
{
Loop0:
    for (int i = 0; i < size; i++)
    {
        #pragma HLS PERFORMANCE target_tti=32
        #pragma HLS LOOP_TRIPCOUNT max=32
        out[i] = in.read();
    }
}
```

Tutorials and Examples

To help you quickly get started with the Vitis HLS, you can find tutorials and example applications at the following locations:

- **Vitis HLS Introductory Examples** (<https://github.com/Xilinx/Vitis-HLS-Introductory-Examples>): Hosts many small code examples to demonstrate good design practices, coding guidelines, design pattern for common applications, and most importantly, optimization techniques to maximize application performance. All examples include a `README` file, and a `run_hls.tcl` script to help you use the example code.
- **Vitis Accel Examples Repository** (https://github.com/Xilinx/Vitis_Accel_Examples): Contains examples to showcase various features of the Vitis tools and platforms. This repository illustrates specific scenarios related to host code and kernel programming for the Vitis application acceleration development flow, by providing small working examples. The kernel code in these examples can be directly compiled in Vitis HLS.
- **Vitis Application Acceleration Development Flow Tutorials** (<https://github.com/Xilinx/Vitis-Tutorials>): Provides a number of tutorials that can be worked through to teach specific concepts regarding the tool flow and application development, including the use of Vitis HLS as a standalone application, and in the Vitis bottom up design flow.

HLS Programmers Guide

Introduction

This Programmers Guide is intended to provide real world design techniques, and details of hardware design which will help you get the most out of the Vitis™ HLS tool. This guide provides details on programming techniques you should apply when writing C/C++ code for high-level synthesis into RTL code, and a checklist of best practices to follow when creating IP that utilizes AXI4 interfaces. Finally, it details various optimization techniques that can improve the performance of your code, improving both the fit and function of the resulting hardware design.

This section contains the following chapters:

- [Design Principles](#)
- [Interfaces of the HLS Design](#)
- [Optimizing Techniques and Troubleshooting Tips](#)

Design Principles

Introduction

You might be working with the HLS tool to take advantage of productivity gains from writing C/C++ code to generate RTL for hardware; or you might be looking to accelerate portions of a C/C++ algorithm to run on custom hardware implemented in programmable logic. This chapter is intended to help you understand the process of synthesizing hardware from a software algorithm written in C/C++. This document introduces the fundamental concepts used to design and create good synthesizable software in such a way that it can be successfully converted to hardware using high-level synthesis (HLS) tools. The discussion in this document will be tool-agnostic and the concepts introduced are common to most HLS tools. For experienced designers, reviewing this material can provide a useful reinforcement of the importance of these concepts; help you understand how to approach HLS, and in particular how to structure HLS code to achieve high-performance designs.

Throughput and Performance

C/C++ functions implemented as custom hardware in programmable logic can run at a significantly faster rate than what is achievable on traditional CPU/GPU architectures, and achieve higher processing rates and/or performance. Let us first establish what these terms mean in the context of hardware acceleration. *Throughput* is defined as the number of specific actions executed per unit of time or results produced per unit of time. This is measured in units of whatever is being produced (cars, motorcycles, I/O samples, memory words, iterations) per unit of time. For example, the term "memory bandwidth" is sometimes used to specify the throughput of the memory systems. Similarly, *performance* is defined as not just higher throughput but higher throughput with low power consumption. Lower power consumption is as important as higher throughput in today's world.

Architecture Matters

In order to better understand how custom hardware can accelerate portions of your program, you will first need to understand how your program runs on a traditional computer. The von Neumann architecture is the basis of almost all computing done today even though it was designed more than 7 decades ago. This architecture was deemed optimal for a large class of applications and has tended to be very flexible and programmable. However, as application demands started to stress the system, CPUs began supporting the execution of multiple processes. Multithreading and/or Multiprocessing can include multiple system processes (For example: executing two or more programs at the same time), or it can consist of one process that

has multiple *threads* within it. Multi-threaded programming using a shared memory system became very popular as it allowed the software developer to design applications with parallelism in mind but with a fixed CPU architecture. But when multi-threading and the ever-increasing CPU speeds could no longer handle the data processing rates, multiple CPU cores and hyperthreading were used to improve throughput as shown in the figure on the right.

This general purpose flexibility comes at a cost in terms of power and peak throughput. In today's world of ubiquitous smart phones, gaming, and online video conferencing, the nature of the data being processed has changed. To achieve higher throughput, you must move the workload closer to memory, and/or into specialized functional units. So the new challenge is to design a new programmable architecture in such a way that you can maintain just enough programmability while achieving higher performance and lower power costs.

A field-programmable gate array (FPGA) provides for this kind of programmability and offers enough memory bandwidth to make this a high-performance and lower power cost solution. Unlike a CPU that executes a program, an FPGA can be configured into a custom hardware circuit that will respond to inputs in the same way that a dedicated piece of hardware would behave. Reconfigurable devices such as FPGAs contain computing elements of extremely flexible granularities, ranging from elementary logic gates to complete arithmetic-logic units such as digital signal processing (DSP) blocks. At higher granularities, user-specified composable units of logic called kernels can then be strategically placed on the FPGA device to perform various roles. This characteristic of reconfigurable FPGA devices allows the creation of custom macro-architectures and gives FPGAs a big advantage over traditional CPUs/GPUs in utilizing application-specific parallelism. Computation can be spatially mapped to the device, enabling much higher operational throughput than processor-centric platforms. Today's latest FPGA devices can also contain processor cores (Arm-based) and other hardened IP blocks that can be used without having to program them into the programmable fabric.

Three Paradigms for Programming FPGAs

While FPGAs can be programmed using lower-level Hardware Description Languages (HDLs) such as Verilog or VHDL, there are now several High-Level Synthesis (HLS) tools that can take an algorithmic description written in a higher-level language like C/C++ and convert it into lower-level hardware description languages such as Verilog or VHDL. This can then be processed by downstream tools to program the FPGA device.

The main benefit of this type of flow is that you can retain the advantages of the programming language like C/C++ to write efficient code that can then be translated into hardware.

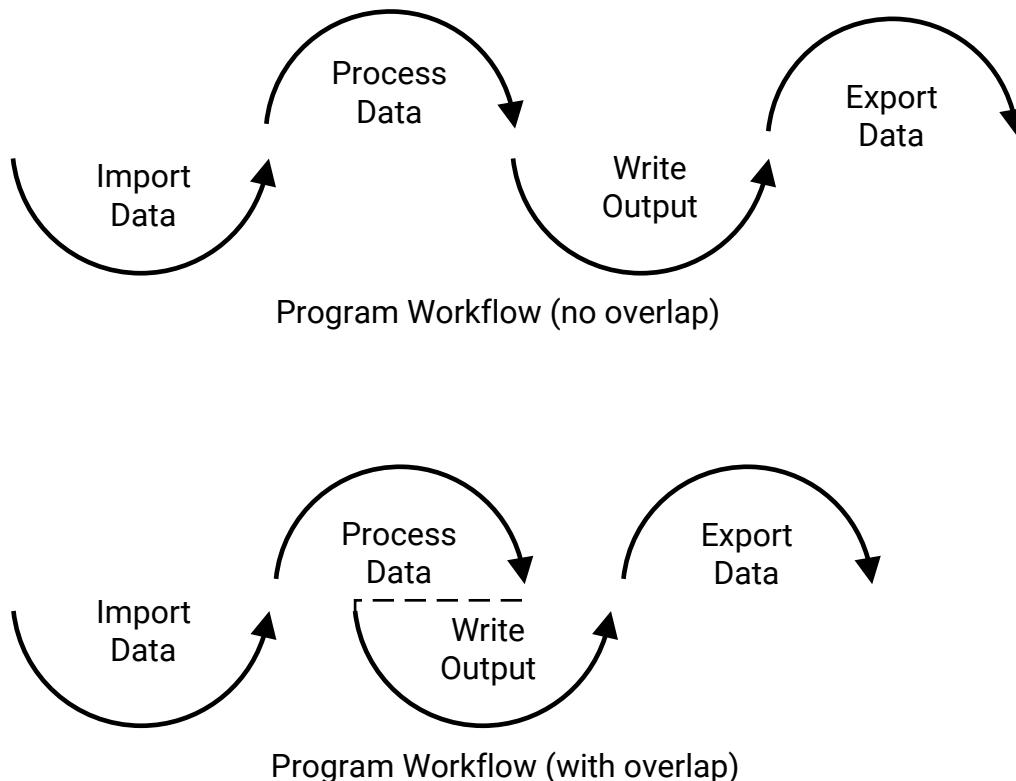
Additionally, writing good code is the software designer's forte and is easier than learning a new hardware description language. However, achieving acceptable quality of results (QoR), will require additional work such as rewriting the software to help the HLS tool achieve the desired performance goals. The next few sections will discuss how you can first identify some macro-level architectural optimizations to structure your program and then focus on some fine-grained micro-level architectural optimizations to boost your performance goals.

Producer-Consumer Paradigm

Consider how software designers write a multithreaded program - there is usually a master thread that performs some initialization steps and then forks off a number of child threads to do some parallel computation and when all the parallel computation is done, the main thread collates the results and writes to the output. The programmer has to figure out what parts can be forked off for parallel computation and what parts need to be executed sequentially. This fork/join type of parallelism applies as well to FPGAs as it does to CPUs, but a key pattern for throughput on FPGAs is the producer-consumer paradigm. You need to apply the producer-consumer paradigm to a sequential program and convert it to extract functionality that can be executed in parallel to improve performance.

You can better understand this decomposition process with the help of a simple problem statement. Assume that you have a datasheet from which you will import items into a list. You will then process each item in the list. The processing of each item takes around 2 seconds. After processing, you will write the result in another datasheet and this action will take an additional 1 second per item. So if you have a total of 100 items in the input Excel sheet then it will take a total of 300 seconds to generate output. The goal is to decompose this problem in such a way that you can identify tasks that can potentially execute in parallel and therefore increase the throughput of the system.

Figure 4: Program Workflows



X25607-073021

The first step is to understand the program workflow and identify the independent tasks or functions. The four-step workflow is something like the Program Workflow (no overlap) shown in the above diagram. In the example, the "Write Output" (step 3) task is independent of the "Process Data" (step 2) processing task. Although step 3 depends on the output of step 2, as soon as any of the items are processed in Step 2, you can immediately write that item to the output file. You don't have to wait for all the data to be processed before starting to write data to the output file. This type of interleaving/overlapping the execution of tasks is a very common principle. This is illustrated in the above diagram (For example: the program workflow with overlap). As can be seen, the work gets done faster than with no overlap. You can now recognize that step 2 is the producer, and step 3 is the consumer. The producer-consumer pattern has a limited impact on performance on a CPU. You can interleave the execution of the steps of each thread but this requires careful analysis to exploit the underlying multi-threading and L1 cache architecture and therefore a time consuming activity. On FPGAs however, due to the custom architecture, the producer and consumer threads can be executed simultaneously with little or no overhead leading to a considerable improvement in throughput.

The simplest case to first consider is the single producer and single consumer, who communicate via a finite-size buffer. If the buffer is full, the producer has a choice of either blocking/stalling or discarding the data. Once the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can stall if it finds the buffer empty. Once the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be achieved by means of inter-process communication, typically using monitors or semaphores. An inadequate solution could result in a deadlock where both processes are stalled waiting to be woken up. However, in the case of a single producer and consumer, the communication pattern strongly maps to a first-in-first-out (FIFO) or a Ping-Pong buffer (PIPO) implementation. This type of channel provides highly efficient data communication without relying on semaphores, mutexes, or monitors for data transfer. The use of such locking primitives can be expensive in terms of performance and difficult to use and debug. PIPOs and FIFOs are popular choices because they avoid the need for end-to-end atomic synchronization.

This type of macro-level architectural optimization, where the communication is encapsulated by a buffer, frees the programmer from worrying about memory models and other non-deterministic behavior (like race conditions etc). The type of network that is achieved in this type of design is purely a "dataflow network" that accepts a stream of data on the input side and essentially does some processing on this stream of data and sends it out as a stream of data. The complexities of a parallel program are abstracted away. Note that the "Import Data" (Step 1) and "Export Data" (Step 4) also have a role to play in maximizing the available parallelism. In order to allow computation to successfully overlap with I/O, it is important to encapsulate reading from inputs as the first step and writing to outputs as the last step. This will allow for a maximal overlap of I/O with computation. Reading or writing to input/output ports in the middle of the computation step will limit the available concurrency in the design. It is another thing to keep in mind while designing the workflow of your design.

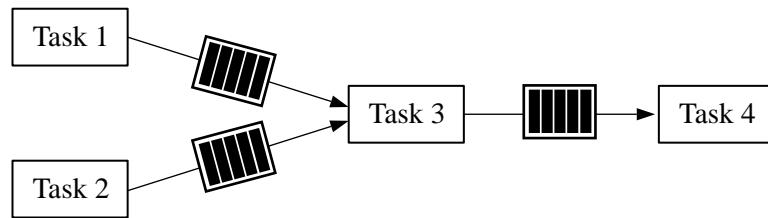
Finally, the performance of such a "dataflow network" relies on the designer being able to continually feed data to the network such that data keeps streaming through the system. Having interruptions in the dataflow can result in lower performance. A good analogy for this is video streaming applications like online gaming where the real-time high definition (HD) video is constantly streamed through the system and the frame processing rate is constantly monitored to ensure that it meets the expected quality of results. Any slowdown in the frame processing rate can be immediately seen by the gamers on their screens. Now imagine being able to support consistent frame rates for a whole bunch of gamers all the while consuming much less power than with traditional CPU or GPU architectures - this is the sweet spot for hardware acceleration. Keeping the data flowing between the producer and consumer is of paramount importance. Next, you will delve a little deeper into this *streaming* paradigm that was introduced in this section.

Streaming Data Paradigm

A *stream* is an important abstraction: it represents an unbounded, continuously updating data set, where unbounded means “of unknown or of unlimited size”. A stream can be a sequence of data (scalars or buffers) flowing unidirectionally between a source (producer) process and a destination (consumer) process. The streaming paradigm forces you to think in terms of data access patterns (or sequences). In software, random memory accesses to data are virtually free (ignoring the caching costs), but in hardware, it is really advantageous to make sequential accesses, which can be converted into streams. Decomposing your algorithm into producer-consumer relationships that communicate by streaming data through the network has several advantages. It lets the programmer define the algorithm in a sequential manner and the parallelism is extracted through other means (such as by the compiler). Complexities like synchronization between the tasks etc are abstracted away. It allows the producer and the consumer tasks to process data simultaneously, which is key for achieving higher throughput. Another benefit is cleaner and simpler code.

As was mentioned before, in the case of the producer and consumer paradigm, the data transfer pattern strongly maps to a FIFO or a PIPO buffer implementation. A FIFO buffer is simply a queue with a predetermined size/depth where the first element that gets inserted into the queue also becomes the first element that can be popped from the queue. The main advantage of using a FIFO buffer is that the consumer process can start accessing the data inside the FIFO buffer as soon as the producer inserts the data into the buffer. The only issue with using FIFO buffers is that due to varying rates of production/consumption between the producers and consumers, it is possible for improperly sized FIFO buffers to cause a deadlock. This typically happens in a design that has several producers and consumers. A Ping Pong Buffer is a double buffer that is used to speed up a process that can overlap the I/O operation with the data processing operation. One buffer is used to hold a block of data so that a consumer process will see a complete (but old) version of the data, while in the other buffer a producer process is creating a new (partial) version of data. When the new block of data is complete and valid, the consumer and the producer processes will alternate access to the two buffers. As a result, the usage of a ping-pong buffer increases the overall throughput of a device and helps to prevent eventual bottlenecks. The key advantage of PIPOs is that the tool automatically matches the rate of production vs the rate of consumption and creates a channel of communication that is both high performance and is deadlock free. It is important to note here that regardless of whether FIFOs/PIPOs are used, the key characteristic is the same: the producer sends or streams a block of data to the consumer. A block of data can be a single value or a group of N values. The bigger the block size, the more memory resources you will need.

The following is a simple sum application to illustrate the classic streaming/dataflow network. In this case, the goal of the application is to pair-wise add a stream of random numbers then print them. The first two tasks (Task 1 and 2) provide a stream of random numbers to add. These are sent over a FIFO channel to the sum task (Task 3) which reads the values from the FIFO channels. The sum task then sends the output to the print task (Task 4) to publish the result. The FIFO channels provide asynchronous buffering between these independent threads of execution.

Figure 5: Streaming/Dataflow Network

X25608-073021

The streams that connect each “task” are usually implemented as FIFO queues. The FIFO abstracts away the parallel behavior from the programmer, leaving them to reason about a “snapshot” of time when the task is active (scheduled). FIFOs make parallelization easier to implement. This largely results from the reduced variable space that programmers must contend with when implementing parallelization frameworks or fault-tolerant solutions. The FIFO between two independent kernels (see example above) exhibits classic queueing behavior. With purely streaming systems, these can be modeled using queueing or network flow models. Another big advantage of this dataflow type network and streaming optimization is that it can be applied at different levels of granularity. A programmer can design such a network inside each task as well as for a system of tasks or kernel. In fact, you can have a streaming network that instantiates and connects multiple streaming networks or tasks, hierarchically. Another optimization that allows for finer-grained parallelism is *pipelining*.

Pipelining Paradigm

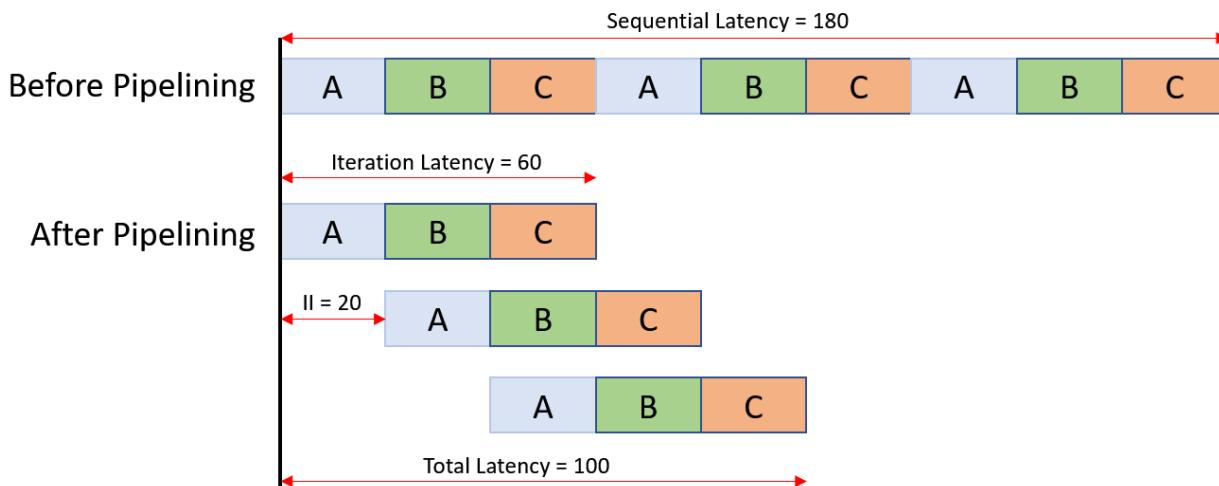
Pipelining is a commonly used concept that you will encounter in everyday life. A good example is the production line of a car factory, where each specific task such as installing the engine, installing the doors, and installing the wheels, is often done by a separate and unique workstation. The stations carry out their tasks in parallel, each on a different car. Once a car has had one task performed, it moves to the next station. Variations in the time needed to complete the tasks can be accommodated by *buffering* (holding one or more cars in a space between the stations) and/or by *stalling* (temporarily halting the upstream stations) until the next station becomes available.

Suppose that assembling one car requires three tasks A, B, and C that takes 20, 10, and 30 minutes, respectively. Then, if all three tasks were performed by a single station, the factory would output one car every 60 minutes. By using a pipeline of three stations, the factory would output the first car in 60 minutes, and then a new one every 30 minutes. As this example shows, pipelining does not decrease the *latency*, that is, the total time for one item to go through the whole system. It does however increase the system's *throughput*, that is, the rate at which new items are processed after the first one.

Since the throughput of a pipeline cannot be better than that of its slowest element, the programmer should try to divide the work and resources among the stages so that they all take the same time to complete their tasks. In the car assembly example above, if the three tasks A, B and C took 20 minutes each, instead of 20, 10, and 30 minutes, the latency would still be 60 minutes, but a new car would then be finished every 20 minutes, instead of 30. The diagram below shows a hypothetical manufacturing line tasked with the production of three cars. Assuming each of the tasks A, B and C takes 20 minutes, a sequential production line would take 180 minutes to produce three cars. A pipelined production line would take only 100 minutes to produce three cars.

The time taken to produce the first car is 60 minutes and is called the *iteration latency* of the pipeline. After the first car is produced, the next two cars only take 20 minutes each and this is known as the *initiation interval (II)* of the pipeline. The overall time taken to produce the three cars is 100 minutes and is referred to as the *total latency* of the pipeline, for example, total latency = iteration latency + II * (number of items - 1). Therefore, improving II improves total latency, but not the iteration latency. From the programmer's point of view, the pipelining paradigm can be applied to functions and loops in the design. After an initial setup cost, the ideal throughput goal will be to achieve an II of 1 - for example, after the initial setup delay, the output will be available at every cycle of the pipeline. In the example above, after an initial setup delay of 60 minutes, a car is then available every 20 minutes.

Figure 6: Pipelining



Pipelining is a classical micro-level architectural optimization that can be applied to multiple levels of abstraction. Task-level pipelining with the producer-consumer paradigm was covered earlier. This same concept applies to the instruction-level. This is in fact key to keeping the producer-consumer pipelines (and streams) filled and busy. The producer-consumer pipeline will only be efficient if each task produces/consumes data at a high rate, and hence the need for the instruction-level pipelining (ILP).

Due to the way pipelining uses the same resources to execute the same function over time, it is considered a static optimization since it requires complete knowledge about the latency of each task. Due to this, the low level instruction pipelining technique cannot be applied to dataflow type networks where the latency of the tasks can be unknown as it is a function of the input data. The next section details how to leverage the three basic paradigms that have been introduced to model different types of task parallelism.

Combining the Three Paradigms

Functions and loops are the main focus of most optimizations in the user's program. Today's optimization tools typically operate at the function/procedure level. Each function can be converted into a specific hardware component. Each such hardware component is like a class definition and many objects (or instances) of this component can be created and instantiated in the eventual hardware design. Each hardware component will in turn be composed of many smaller predefined components that typically implement basic functions such as add, sub, and multiply. Functions may call other functions although recursion is not supported. Functions that are small and called less often can be also inlined into their callers just like how software functions can be inlined. In this case, the resources needed to implement the function are subsumed into the caller function's component which can potentially allow for better sharing of common resources. Constructing your design as a set of communicating functions lends to inferring parallelism when executing these functions.

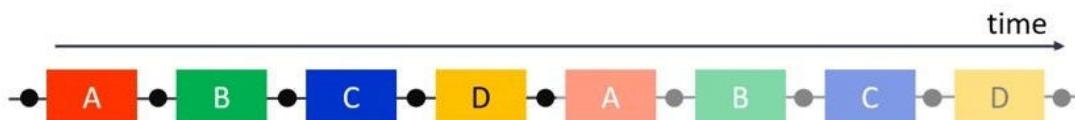
Loops are one of the most important constructs in your program. Since the body of a loop is iterated over a number of times, this property can be easily exploited to achieve better parallelism. There are several transformations (such as [pipelining](#) and [unrolling](#)) that can be made to loops and loop nests in order to achieve efficient parallel execution. These transformations enable both memory-system optimizations as well as mapping to multi-core and SIMD execution resources. Many programs in science and engineering applications are expressed as operations over large data structures. These may be simple element-wise operations on arrays or matrices or more complex loop nests with loop-carried dependencies - i.e. data dependencies across the iterations of the loop. Such data dependencies impact the parallelism achievable in the loop. In many such cases, the code must be restructured such that loop iterations can be executed efficiently and in parallel on modern parallel platforms.

The following diagrams illustrate different overlapping executions for a simple example of 4 consecutive tasks (i.e., C/C++ functions) A, B, C, and D, where A produces data for B and C, in two different arrays, and D consumes data from two different arrays produced by B and C. Let us assume that this “diamond” communication pattern is to be run twice (two invocations) and that these two runs are independent.

```
void diamond(data_t vecIn[N], data_t vecOut[N])
{
    data_t c1[N], c2[N], c3[N], c4[N];
    #pragma HLS dataflow
    A(vecIn, c1, c2);
    B(c1, c3);
    C(c2, c4);
    D(c3, c4, vecOut);
}
```

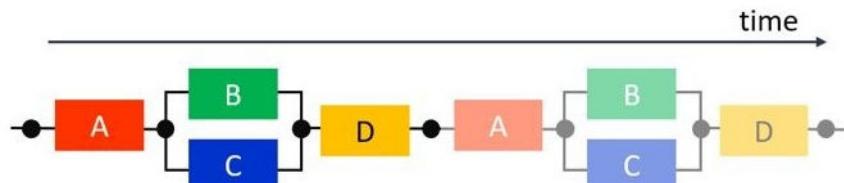
The code example above shows the C/C++ source snippet for how these functions are invoked. Note that tasks B and C have no mutual data dependencies. A fully-sequential execution corresponds to the figure below where the black circles represent some form of synchronization used to implement the serialization.

Figure 7: Sequential Execution - Two Runs



In the diamond example, B and C are fully-independent. They do not communicate nor do they access any shared memory resource, and so if no sharing of computation resource is required, they can be executed in parallel. This leads to the diagram in the figure below, with a form of fork-join parallelism within a run. B and C are executed in parallel after A ends while D waits for both B and C, but the next run is still executed in series.

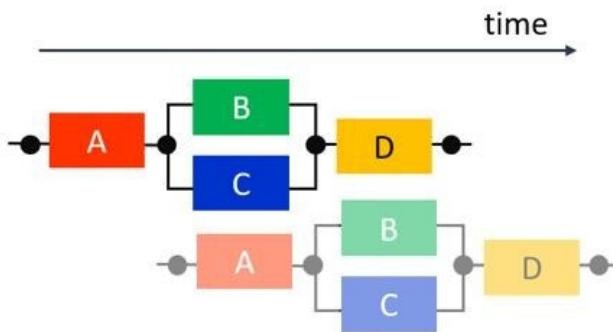
Figure 8: Task Parallelism within a Run



Such an execution can be summarized as $(A; (B \parallel C); D); (A; (B \parallel C); D)$ where “;” represents serialization and “ \parallel ” represents full parallelism. This form of nested fork-join parallelism corresponds to a subclass of dependent tasks, namely series-parallel task graphs. More generally, any DAG (directed acyclic graph) of dependent tasks can be implemented with separate fork-and-join-type synchronization. Also, it is important to note that this is exactly like how a multithreaded program would run on a CPU with multiple threads and using shared memory.

On FPGAs, you can explore what other forms of parallelism are available. The previous execution pattern exploited task-level parallelism within an invocation. What about overlapping successive runs? If they are truly independent, but if each function (i.e., A, B, C, or D) reuses the same computation hardware as for its previous run, you may still want to execute, for example, the second invocation of A in parallel with the first invocations of B and C. This is a form of task-level pipelining across invocations, leading to a diagram as depicted in the following figure. The throughput is now improved because it is limited by the maximum latency among all tasks, rather than by the sum of their latencies. The latency of each run is unchanged but the overall latency for multiple runs is reduced.

Figure 9: Task Parallelism with Pipelining

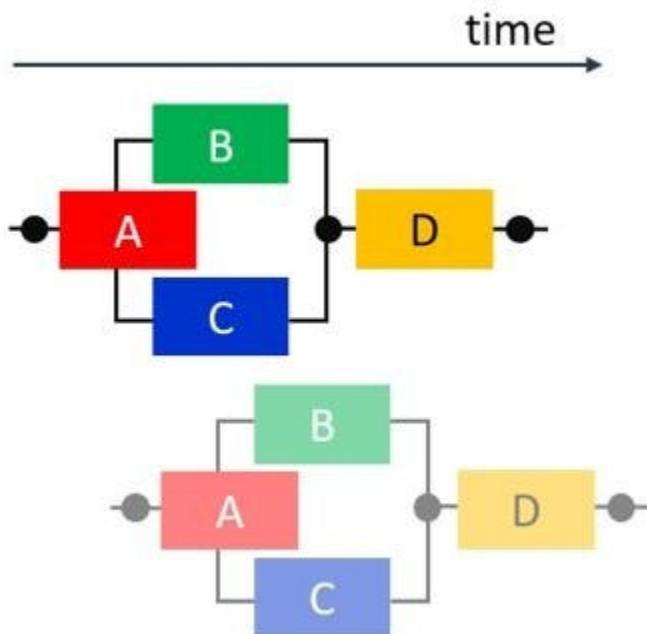


Now, however, when the first run of B reads from the memory where A placed its first result, the second run of A is possibly already writing in the same memory. To avoid overwriting the data before it is consumed, you can rely on a form of memory expansion, namely double buffering or PIPOs to allow for this interleaving. This is represented by the black circles between the tasks.

An efficient technique to improve throughput and reuse computational resources is to pipeline operators, loops, and/or functions. If each task can now overlap with itself, you can achieve simultaneously task parallelism within a run and task pipelining across runs, both of which are examples of macro-level parallelism. Pipelining within the tasks is an example of micro-level parallelism. The overall throughput of a run is further improved because it now depends on the minimum throughput among the tasks, rather than their maximum latency. Finally, depending on how the communicated data are synchronized, only after all are produced (PIPOs) or in a more element-wise manner (FIFOs), some additional overlapping within a run can be expected. For example, in the following figure, both B and C start earlier and are executed in a pipelined fashion with respect to A, while D is assumed to still have to wait for the completion of B and C. This last

type of overlap within a run can be achieved if A communicates to B and C through FIFO streaming accesses (represented as lines without circles). Similarly, D can also be overlapped with B and C, if the channels are FIFOs instead of PIPOs. However, unlike all previous execution patterns, using FIFOs can lead to deadlocks and so these streaming FIFOs need to be sized correctly.

Figure 10: Task Parallelism and Pipelining within a Run, Pipelining of Runs, and Pipelining within a Task



In summary, the three paradigms presented in the earlier section show how parallelism can be achieved in your design without needing the complexities of multi-threading and/or parallel programming languages. The producer-consumer paradigm coupled with streaming channels allows for the composition of small to large scale systems easily. As mentioned before, streaming interfaces allow for easy coupling of parallel tasks or even hierarchical dataflow networks. This is in part due to the flexibility in the programming language (C/C++) to support such specifications and the tools to implement them on the heterogeneous computing platform available on today's FPGA devices.

Conclusion - A Prescription for Performance

The design concepts presented in this document have one main central principle - a model of parallel computation that favors encapsulation of state and sequential execution within modular units or tasks to facilitate a simpler programming model for parallel programming. Tasks are then connected together with streams (for synchronization and communication). A stream can be different types of channels such as FIFOs or PIPOs. The state/logic compartmentalization makes it much easier for tools (such as a compiler and a scheduler) to figure out where to run which pieces of an application and when. The second reason why stream-based processing is becoming popular is that it breaks the traditional multi-threading based “fork/join” view on parallel execution. By enabling task-level pipelining and instruction-level pipelining, the run-time can do many more concurrent actions than what is possible today with the fork/join model. This extra parallelism is critical to taking advantage of the hardware available on today's FPGA devices. In the same vein as enabling pipeline parallelism, streaming also enables designers to build parallel applications without having to worry about locks, race conditions, etc. that make parallel programming hard in the first place.

Finally, the following checklist of high-level actions is recommended as a prescription for achieving performance on reconfigurable FPGA platforms:

- Software written for CPUs and software written for FPGAs is fundamentally different. You cannot write code that is portable between CPU and FPGA platforms without sacrificing performance. Therefore, embrace and do not resist the fact that you will have to write significantly different software for FPGAs.
- Right from the start of your project, establish a flow that can functionally verify the source code changes that are being made. Testing the software against a reference model or using golden vectors are common practices.
- Focus first on the macro-architecture of your design. Consider modeling your solution using the producer-consumer paradigm.
- Once you have identified the macro-architecture of your design, draw the desired activity timeline where the horizontal axis represents time, and show when you expect each function to execute relative to each other over multiple iterations (or invocations). This will give you a sense of the expected parallelism in the design and can then be used to compare with the final achieved results. Often the HLS GUIs can be used to visualize this achieved parallelism.
- Only start coding or refactoring your program once you have the macro-architecture and the activity timeline well established
- As a general rule, the HLS compiler will only infer task-level parallelism from function calls. Therefore, sequential code blocks (such as loops) which need to run concurrently in hardware should be put into dedicated functions.

- Decompose/partition the original algorithm into smaller components that talk to each other via streams. This will give you some ideas of how the data flows in your design.
 - Smaller modular components have the advantage that they can be replicated when needed to improve parallelism.
 - Avoid having communication channels with very wide bit-widths. Decomposing such wide channels into several smaller ones will help implementation on FPGA devices.
 - Large functions (written by hand or generated by inlining smaller functions) can have non-trivial control paths that can be hard for tools to process. Smaller functions with simpler control paths will aid implementation on FPGA devices.
 - Aim to have a single loop nest (with either fixed loop bounds that can be inferred by HLS tool, or by providing loop trip count information by hand to the HLS tool) within each function. This greatly facilitates the measurement and optimization of throughput. While this may not be applicable for all designs, it is a good approach for a large majority of cases.
- Throughput - Having an overall vision about what rates of processing will be required during each phase of your design is important. Knowing this will influence how you write your application for FPGAs.
 - Think about the critical path (i.e., critical task level paths such as ABD or ACD) in your design and study what part of this critical path is potentially a bottleneck. Look at how individual tasks are pipelined and if different branches of a path are unaligned in terms of throughput by simulating the design. HLS GUI tools and/or the simulation waveform viewer can then be used to visualize such throughput issues.
 - Stream-based communication allows consumers to start processing as soon as producers start producing which allows for overlapped execution (which in turn increases parallelism and throughput).
 - In order to keep the producer and consumer tasks running constantly without any hiccups, optimize the execution of each task to run as fast as possible using techniques such as pipelining and the appropriate sizing of streams.
- Think about the granularity (and overhead) of the streaming channels with respect to synchronization. The usage of PIPO channels allows you to overlap task execution without the fear of deadlock while explicit manual streaming FIFO channels allow you to start the overlapped execution sooner (than PIPOs) but require careful adjustment of FIFO sizes to avoid deadlocks.
- [Learn about synthesizable C/C++ coding styles.](#)
- Use the reports generated by the HLS compiler to guide the optimization process.

Keep the above checklist nearby so that you can refer to it from time to time. It summarizes the whole design activity needed to build a design that meets your performance goals.

Another important aspect of your design to consider next is the interface of your accelerated function or kernel. The interface of your kernel to the outside world is an important element of your eventual system design. Your kernel may need to plug into a bigger design, or to communicate with other kernels in a large system of kernels, or to communicate with memory or devices outside of the system. [Best Practices for Designing with M_AXI Interfaces](#) provides another checklist of items to consider when designing the external interfaces of your acceleration kernel.

Abstract Parallel Programming Model for HLS

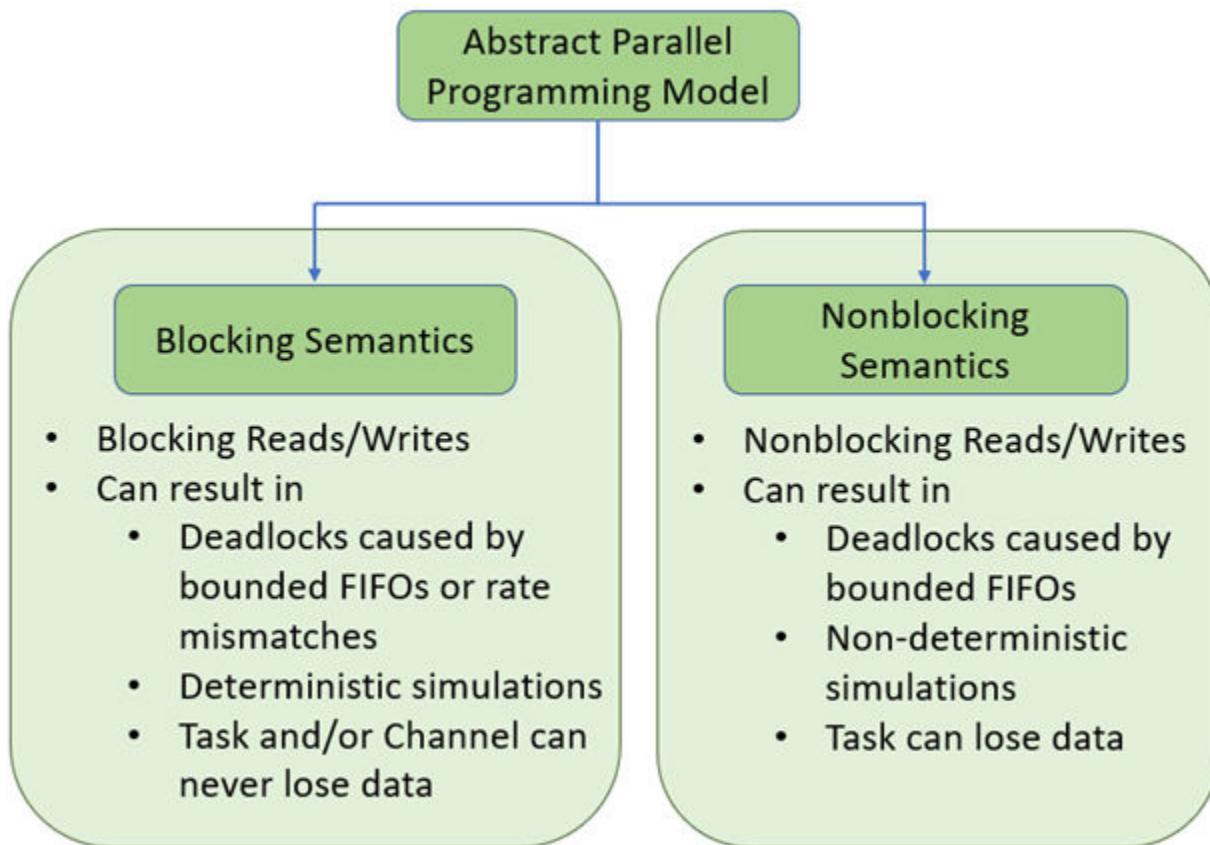
In order to achieve high performance hardware, the HLS tool must infer parallelism from sequential code and exploit it to achieve greater performance. This is not an easy problem to solve. In addition, good software design often uses well-defined rules and practices such as run-time type information ([RTTI](#)), recursion, and dynamic memory allocation. Many of these techniques have no direct equivalence in hardware and present challenges for the HLS tool. This generally means that off-the-shelf software cannot be efficiently converted into hardware. At a bare minimum, such software needs to be examined for non-synthesizable constructs and the code needs to be refactored to make it synthesizable. Even if a software program can be automatically converted (or synthesized) into hardware, to assist the tool you need to understand the best practices for writing good software for execution on the FPGA device.

The [Design Principles](#) section introduced the three main paradigms that need to be understood for writing good software for FPGA platforms: producer-consumer, streaming data, and pipelining. The underlying parallel programming model that these paradigms work on is as follows:

- The design/program needs to be constructed as a collection of tasks that communicate by sending messages to each other through communication links (aka channels)
- Tasks can be structured as control-driven, waiting for some signal to start execution, or data-driven in which the presence of data on the channel drives the execution of the task
- A task consists of an executable unit that has some local storage/memory and a collection of input/output (I/O) ports.
- The local memory contains private data, i.e., the data to which the task has exclusive access
- Access to this private memory is called local data access - like data stored in BRAM/URAM. This type of access is fast. The only way that a task can send copies of its local data to other tasks is through its output ports, and conversely, it can only receive data through its input ports
- An I/O port is an abstraction; it corresponds to a channel that the task will use for sending or receiving data and it is connected by the caller of the module, or at the system integration level if it is a top port
- Data sent or received through a channel is called non-local data access. A channel is a data queue that connects one task's output port to another task's input port

- A channel is assumed to be reliable and has the following behaviors:
 - Data written at the output of the producer are read at the input port of the consumer in the same order for FIFOs. Data can be read/written in random order for PIPOs
 - No data values are lost
- Both blocking and non-blocking read and write semantics are supported for channels, as described in [HLS Stream Library](#)

Figure 11: Blocking/Non-Blocking Semantics



When blocking semantics are used in the model, a read to an empty channel results in the blocking of the reading process. Similarly, a write to a full channel results in the blocking of the writing process. The resulting process/channel network exhibits [deterministic behavior](#) that does not depend on the timing of computation nor on [communication delays](#). These style of models have proven convenient for modeling [embedded systems](#), [high-performance computing](#) systems, [signal processing](#) systems, [stream processing](#) systems, [dataflow programming](#) languages, and other computational tasks.

The blocking style of modeling can result in deadlocks due to insufficient sizing of the channel queue (when the channels are FIFOs) and/or due to differing rates of production between producers and consumers. If non-blocking semantics are used in the model, a read to an empty channel results in the reading of uninitialized data or in the re-reading of the last data item. Similarly, a write to a full queue can result in that data being lost. To avoid such loss of data, the design must first check the status of the queue before performing the read/write. But this will cause the simulation of such models to be non-deterministic since it relies on decisions made based on the run-time status of the channel. This will make verifying the results of this model much more challenging.

Both blocking and non-blocking semantics are supported by the Vitis HLS abstract parallel programming model.

Control and Data Driven Tasks

Using this abstract model as the basis, two types of task-level parallelism (TLP) models can be used to structure and design your application. TLP can be data-driven or control-driven, or can mix control-driven and data-driven tasks in a single design. The main differences between these two models are:

- If your application is purely data-driven, does not require any interaction with external memory and the functions can execute in parallel with no data dependencies, then the data-driven TLP model is the best fit. You can design a purely data-driven model that is always running, requires no control, and reacts only to data. For additional details refer to [Data-driven Task-level Parallelism](#).
- If your application requires some interaction with external memory, and there are data dependencies between the tasks that execute in parallel, then the control-driven TLP model is the best fit. Vitis HLS will infer the parallelism between tasks and create the right channels (as defined by you) such that these functions can be overlapped during execution. The control-driven TLP model is also known as the dataflow optimization in Vitis HLS as described in [Control-driven Task-level Parallelism](#).

The next few sections describe these major modeling options that are available to use. You can use any of these models to write your source code using C++ in order to optimize the execution of the program on parallel hardware.

Data-driven Task-level Parallelism

Data-driven task-level parallelism uses a task-channel modeling style that requires you to statically instantiate and connect tasks and channels explicitly. Tasks in this modeling style only have stream type inputs and outputs. The tasks are not controlled by any function call/return semantics but rather are always running waiting for data on their input stream.

Data-driven TLP models are tasks that execute when there is data to be processed. In Vitis HLS C-simulation used to be limited to seeing only the sequential semantics and behavior. With the data-driven model it is possible during simulation to see the concurrent nature of parallel tasks and their interactions via the FIFO channels.

Implementing data-driven TLP in the Vitis HLS tool uses simple classes for modeling tasks (`hls::task`) and channels (`hls::stream/hls::stream_of_blocks`)



IMPORTANT! While Vitis HLS supports `hls::tasks` for a top-level function, you cannot use `hls::stream_of_blocks` for interfaces in top-level functions.

Consider the [simple task-channel](#) example shown below:

```
#include "test.h"

void splitter(hls::stream<int> &in, hls::stream<int> &odds_buf,
hls::stream<int> &evens_buf) {
    int data = in.read();
    if (data % 2 == 0)
        evens_buf.write(data);
    else
        odds_buf.write(data);
}

void odds(hls::stream<int> &in, hls::stream<int> &out) {
    out.write(in.read() + 1);
}

void evens(hls::stream<int> &in, hls::stream<int> &out) {
    out.write(in.read() + 2);
}

void odds_and_evens(hls::stream<int> &in, hls::stream<int> &out1,
hls::stream<int> &out2) {
    hls_thread_local hls::stream<int> s1; // channel connecting t1 and
t2
    hls_thread_local hls::stream<int> s2; // channel connecting t1 and t3

    // t1 infinitely runs function splitter, with input in and outputs s1
and s2
    hls_thread_local hls::task t1(splitter, in, s1, s2);
    // t2 infinitely runs function odds, with input s1 and output out1
    hls_thread_local hls::task t2(odds, s1, out1);
    // t3 infinitely runs function evens, with input s2 and output out2
    hls_thread_local hls::task t3(evens, s2, out2);
}
```

The special `hls::task` C++ class is:

- A new object declaration in your source code that requires a special qualifier. The `hls_thread_local` qualifier is required in order to keep the object (and the underlying thread) alive across multiple calls of the instantiating function (`odds_and_evens` in the example).

The `hls_thread_local` qualifier is only required to ensure that the C simulation of the data-driven TLP model exhibits the same behavior as the RTL simulation. In the RTL, these functions are already in always running mode once started. In order to ensure the same behavior during C Simulation, the `hls_thread_local` qualifier is required to ensure that each task is started only once and keeps the same state even when called multiple times. Without the `hls_thread_local` qualifier, each new invocation of the function would result in a new state.

- Task objects implicitly manage a thread that runs a function infinitely, passing to it a set of arguments that must be either `hls::stream` or `hls::stream_of_blocks`



TIP: No other types of arguments are supported. In particular, even constant values cannot be passed as function arguments. If constants need to be passed to the task's body, define the function as a templated function and pass the constant as a template argument to this templated function.

- The supplied function (`splitter/odds/evens` in the example above) is called the task body, and it has an implicit infinite loop wrapped around it to ensure that the task keeps running and waiting on input.
- The supplied function can contain pipelined loops but they need to be flushable pipelines (FLP) in order to prevent deadlock. The tool will automatically select the right pipeline style to use for a given pipelined loop or function.



IMPORTANT! An `hls::task` should not be treated as a function call - instead a `hls::task` needs to be thought of as a persistent instance statically bound to channels. Due to this, it will be your responsibility to ensure that multiple invocations to any function that contains `hls::tasks` be unqualified or these calls will use the same `hls::tasks` and channels.

Channels are modeled by the special templatized `hls::stream` (or `hls::stream_of_blocks`) C++ class. Such channels have the following attributes:

- In the data-driven TLP model, an `hls::stream<type, depth>` object behaves like a FIFO with a specified depth. Such streams have a default depth of 2 which can be overridden by the user.
- The streams are read from and written to sequentially. That implies that once a data item is read from an `hls::stream<>` that same data item cannot be read again.



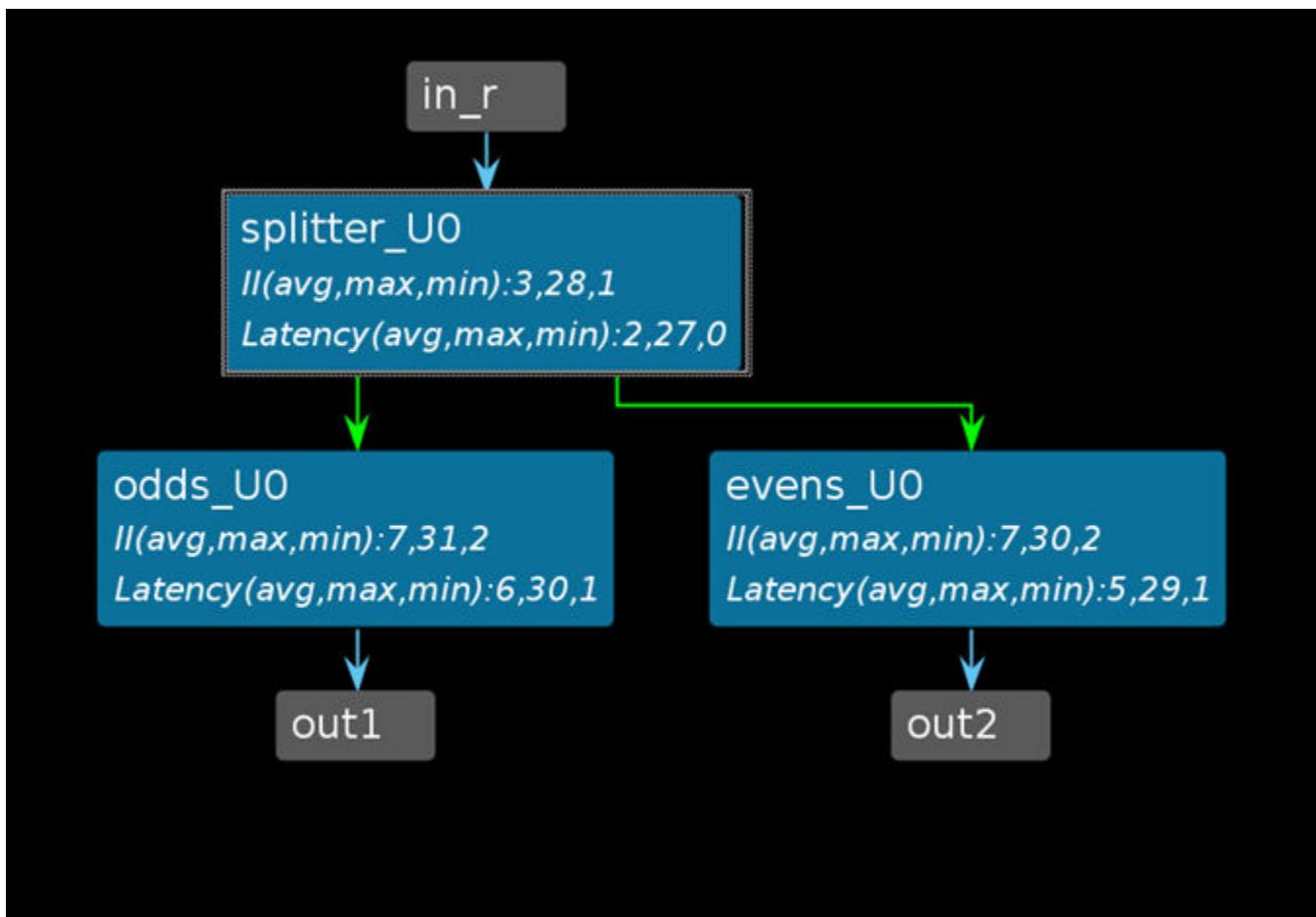
TIP: Accesses to different streams are not ordered (e.g. the order of a write to a stream and a read from a different stream can be changed by the scheduler).

- Streams may be defined either locally or globally. Streams defined in the global scope follow the same rules as any other global variables.

- The `hls_thread_local` qualifier is also required for streams (`s1` and `s2` in the example below) in order to keep the same streams alive across multiple calls of the instantiating function (`odds_and_evens` in the code example below).

The following diagram shows the graphical representation in Vitis HLS of the code example above. In this diagram, the green colored arrows are FIFO channels while the blue arrows indicate the inputs and outputs of the instantiating function (`odds_and_evens`). Tasks are shown as blue rectangular boxes.

Figure 12: Dataflow Diagram of `hls::task` Example



Due to the fact that a read of an empty stream is a blocking read, deadlocks can occur due to:

- The design itself, where the production and consumption rates by processes are unbalanced.
 - During C simulation, deadlocks can occur only due to a cycle of processes, or a chain of processes starting from a top-level input, that are attempting to read from empty channels.
 - Deadlocks can occur during both C/RTL Co-simulation and when running in hardware (HW) due to cycles of processes trying to write to full channels and/or reading from empty channels.

- The test bench, which is providing less data than those that are needed to produce all the outputs that the test bench is expecting when checking the computation results.

Due to this, a deadlock detector is automatically instantiated when the design contains an `hls::task`. The deadlock detector detects deadlocks and stops the C simulation. Further debugging is performed using a C debugger such as `gdb` and looking at where the simulated `hls::tasks` are all blocked trying to read from an empty channel. Note that this is easy to do using the Vitis HLS GUI as shown in the [handling_deadlock](#) example for debugging deadlocks.

In summary, the `hls::task` model is recommended if your design requires a completely data-driven, pure streaming type of behavior, with no sort of control. This type of model is also useful in modeling feedback and dynamic multi-rate designs. Feedback in the design occurs when there is a cyclical dependency between tasks. Dynamic multi-rate models, where the producer writes data or consumer reads data at a rate that is data dependent, can only be handled by the data-driven TLP. The [simple_data_driven](#) design on GitHub is an example of this.

Note: Static multi-rate designs, in which the producer writes data or consumer reads data at rates that are data-independent, can be managed by both data-driven and control-driven TLP. For example, the producer writes two values in a stream for each call, the consumer reads one value per call.

Control-driven Task-level Parallelism

Control-driven TLP is useful to model parallelism while relying on the sequential semantics of C++, rather than on continuously running threads. Examples include functions that can be executed in a concurrent pipelined fashion, possibly within loops, or with arguments that are not channels but C++ scalar and array variables, both referring to on-chip and to off-chip memories. For this kind of model, Vitis HLS introduces parallelism where possible while preserving the behavior obtained from the original C++ sequential execution. The control-driven TLP (or dataflow) model provides:

- A subsequent function can start before the previous finishes
- A function can be restarted before it finishes
- Two or more sequential functions can be started simultaneously

While using the dataflow model, Vitis HLS implements the sequential semantics of the C++ code by automatically inserting synchronization and communication mechanisms between the tasks.

The dataflow model takes this series of sequential functions and creates a task-level pipeline architecture of concurrent processes. The tool does this by inferring the parallel tasks and channels. The designer specifies the region to model in the dataflow style (i.e., a function body or a loop body) by specifying the **DATAFLOW** pragma or directive as shown below. The tool scans the loop/function body, extracts the parallel tasks as parallel processes, and establishes communication channels between these processes. The designer can additionally guide the tool to select the type of channels - i.e., FIFO (`hls::stream` or `#pragma HLS STREAM`) or PIPO or `hls::stream_of_blocks`. The dataflow model is a powerful method for improving design throughput and latency.

In order to understand how Vitis HLS transforms your C++ code into the dataflow model, refer to the [simple_fifos](#) example shown below. The example applies the dataflow model to the top-level diamond function using the DATAFLOW pragma as shown.

```
#include "diamond.h"

void diamond(data_t vecIn[N], data_t vecOut[N])
{
    data_t c1[N], c2[N], c3[N], c4[N];
#pragma HLS dataflow
    funcA(vecIn, c1, c2);
    funcB(c1, c3);
    funcC(c2, c4);
    funcD(c3, c4, vecOut);
}
```

In the above example, there are four functions: `funcA`, `funcB`, `funcC` and `funcD`. `funcB` and `funcC` do not have any data dependencies between them and therefore can be executed in parallel. `funcA` reads from the non-local memory (`vecIn`) and needs to be executed first. Similarly, `funcD` writes to the non-local memory (`vecOut`) and therefore has to be executed last.

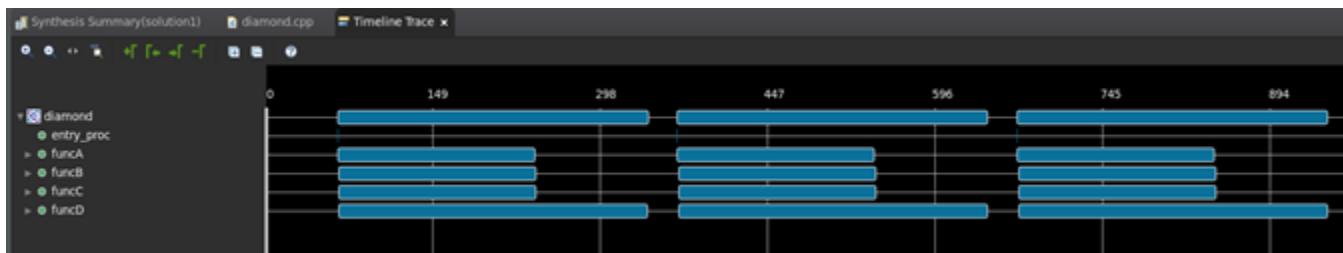
The following waveform shows the execution profile of this design without the dataflow model. There are three calls to the function `diamond` from the test bench. `funcA`, (`funcB`, `funcC`) and `funcD` are executed in sequential order. Each call to `diamond`, therefore, takes 475 cycles in total as shown in the figure below.

Figure 13: Diamond Example without Dataflow



In the following figure, when the dataflow model is applied and the designer selected to use FIFOs for channels, all the functions are started immediately by the controller and are stalled waiting on input. As soon as the input arrives, it is processed and sent out. Due to this type of overlap, each call to diamond now only takes 275 cycles in total as shown below. Refer to [Combining the Three Paradigms](#) for a more detailed discussion of the types of parallelism that can be achieved for this example.

Figure 14: Diamond Example with Dataflow



This type of parallelism cannot be achieved without incurring an increase in hardware utilization. When a particular region, such as a function body or a loop body, is identified as a region to apply the dataflow model, Vitis HLS analyzes the function or loop body and creates individual channels from C++ variables (such as scalars, arrays, or user-defined channels such as `hls::streams` or `hls::stream_of_blocks`) that model the flow of data in the dataflow region. These channels can be simple FIFOs for scalar variables, or ping-pong (PIPO) buffers for non-scalar variables like arrays (or stream of blocks when you need a combination of FIFO and PIPO behavior with explicit locking of the blocks).

Each of these channels can contain additional signals to indicate when the channel is full or empty. By having individual FIFOs and/or PIPO buffers, Vitis HLS frees each task to execute independently and the throughput is only limited by the availability of the input and output buffers. This allows for better overlapping of task execution than a normal pipelined implementation, but does so at the cost of additional FIFO or block RAM registers for the ping-pong buffer.



TIP: This overlapped execution optimization is only visible after you run the cosimulation of the design - it is not observable statically (though can be easily imagined in the [Dataflow Viewer](#) after C Synthesis).

The dataflow model is not limited to a chain of processes but can be used on any directed acyclic graph (DAG) structure, or cyclic structure when using `hls::streams`. It can produce two different forms of overlapping: within an iteration if processes are connected with FIFOs, and across different iterations when connected with PIPOs and FIFOs. This potentially improves performance over a statically pipelined solution. It replaces the strict, centrally-controlled pipeline stall philosophy with a distributed handshaking architecture using FIFOs and/or PIPOs. The replacement of the centralized control structure with a distributed one also benefits the fanout of control signals, for example register enables, which is distributed among the control structures of individual processes. Refer to the [Task Level Parallelism/Control-Driven](#) examples on Github for more examples of these concepts.

Canonical Forms

Vitis HLS transforms the region to apply the DATAFLOW optimization. For more predictability of the resulting dataflow network, AMD recommends writing the code inside this region (referred to as the canonical region) using canonical forms. There are two main canonical forms for the dataflow optimization:

1. The canonical form for a function where sub-functions are not inlined. Note that these subfunctions can themselves be dataflow in function regions or dataflow inside loop regions. Note also that variable initialization (including those performed automatically by constructors) or passing expressions by value to processes are not part of canonical form. Vitis HLS does its best to implement the resulting dataflow but, if the code is not in canonical form, you should always check the GUI dataflow viewer and the cosimulation timeline trace to ensure that the dataflow happens as expected and the achieved performance is as expected.

```
void dataflow(Input0, Input1, Output0, Output1)
{
    #pragma HLS dataflow
    UserDataType C0, C1, C2;           // UserDataType can be scalars or arrays
    func1(Input0, Input1, C0, C1);     // read Input0, read Input1, write C0,
    write C1
    func2(C0, C1, C2);               // read C0, read C1, write C2
    func3(C2, Output0, Output1);     // read C2, write Output0, write Output1
}
```

2. Dataflow inside a loop body enclosed in a function without any other code but the loop. For the `for` loop (where no function inside is inlined), the integral loop variable should have:
 - a. The initial value is declared in the loop header and set to 0.
 - b. The loop bound is a non-negative numerical constant or scalar argument of the function that encloses the loop.
 - c. Increment by 1.
 - d. Dataflow pragma needs to be inside the loop as shown below.

```
void dataflow(Input0, Input1, Output0, Output1)
{
    for (int i = 0; i < N; i++)
    {
        #pragma HLS dataflow
        UserDataType C0, C1, C2;           // UserDataType can be scalars or
        arrays
        func1(Input0, Input1, C0, C1);     // read Input0, read Input1,
        write C0, write C1
        func2(C0, C0, read C1, C2);       // read C0, read C0, read C1,
        write C2
        func3(C2, Output0, Output1);     // read C2, write Output0, write
        Output1
    }
}
```

Canonical Body

Inside the canonical region, the canonical body should follow these guidelines:

1. Use a local, non-static scalar or an array variable. A local variable is declared inside the function body (for dataflow in a function) or loop body (for dataflow inside a loop). Refer to [Limitations of Control-Driven Task-Level Parallelism](#) for additional limitations on arrays.
2. A sequence of function calls that pass data forward (with no feedback unless using `hls::stream/hls::stream_of_blocks`), from a function to one that is lexically later, under the following conditions:
 - a. Variables (except scalar) can have only one reading process and one writing process.
 - b. Use write before read (producer before consumer) if you are using local non-scalar variables, which then become channels. For scalar variables, both write before read and read before write are allowed.
 - c. Use read before write (consumer before producer) if you are using function arguments. Any intra-body anti-dependencies must be preserved by the design.
 - d. Function return type must be void.
 - e. No loop-carried dependencies are allowed among different processes via variables except when FIFOs are used. Forward loop-carried dependencies are supported for arrays transformed to streams and both forward and backward dependencies are supported for `hls::streams`.
 - Except when these dependencies exist across successive calls to the top function (i.e., `inout` argument written by one iteration and read by the following iteration).
 - f. No control whatsoever is supported inside a dataflow region, except inside function calls (that define processes).
 - For canonical dataflow, there should be no conditionals, no loops, no return or goto statements, and no C++ exceptions such as `throw`.

Dataflow Checking

Vitis HLS has a dataflow checker which, when enabled, checks the code to see if it is in the recommended canonical form. Otherwise, it will emit an error/warning message to the user. By default, this checker is set to `warning`. You can set the checker to `error` or disable it by selecting `off` in the strict mode of the `config_dataflow` TCL command:

```
config_dataflow -strict_mode (off | error | warning)
```

Configuring Dataflow Memory Channels

Vitis HLS implements channels between the tasks as either PIPO or FIFO buffers, depending on the user's choice:

- For scalars, Vitis HLS will automatically infer FIFOs as the channel type.
- If the parameter (of a producer or consumer) is an array, the user has a choice of implementing the channel as a PIPO or a FIFO based on the following considerations:
 - If the data is always accessed in sequential order, the user can choose to implement this memory channel as PIPO/FIFO. Choosing PIPOs comes with the advantage that PIPOs can never deadlock but they require more memory to use. Choosing FIFOs offers the advantage of lesser memory requirements but this comes with the risk of deadlock if the FIFO sizes are not correct.
 - If the data is accessed in an arbitrary manner, the memory channel must be implemented as a PIPO (with a default size that is twice the size of the original array).



TIP: A PIPO ensures that the channel always has the capacity to hold all samples produced in one iteration, without a loss.

Specifying the size of the FIFO channels overrides the default value that is computed by the tool to attempt to optimize the throughput. If any function in the design can produce or consume samples at a greater rate than the specified size of the FIFO, the FIFOs might become empty (or full). In this case, the design halts operation, because it is unable to read (or write). This might lead to a stalled, deadlock state.



TIP: If a deadlocked situation is created, you will only see this when executing C/RTL co-simulation or when the block is used in a complete system.

When setting the depth of the FIFOs, AMD recommends initially setting the depth as the maximum number of data values transferred (for example, the size of the array passed between tasks), confirming the design passes C/RTL co-simulation, and then reducing the size of the FIFOs and confirming C/RTL co-simulation still completes without issues. If RTL co-simulation fails, the size of the FIFO is likely too small to prevent stalling or a deadlock situation. The Vitis HLS GUI now supports an automatic way of determining the right FIFO size to use. Additionally, the Vitis HLS IDE can display a histogram of the size of each FIFO/PIPO buffer over time, after RTL co-simulation has been run. This can be useful to help determine the best depth for each buffer.

Specifying Arrays as PIPOs or FIFOs

All arrays are implemented by default as ping-pong to enable random access. These buffers can also be re-sized if needed. For example, in some circumstances, such as when a task is being bypassed, performance degradation is possible. To mitigate this effect on performance, you can give more slack to the producer and consumer by increasing the size of these buffers by using the STREAM pragma or directive as shown below.

```
void top ( ... ) {
#pragma HLS dataflow
    int A[1024];
#pragma HLS stream type=piwo variable=A depth=3

producer(A, B, ...); // producer writes A and B
middle(B, C, ...); // middle reads B and writes C
consumer(A, C, ...); // consumer reads A and C
```

In the interface, arrays are automatically specified as streaming if an array on the top-level function interface is set the following interface types: ap_fifo/axis.

Inside the design, an array must be specified as streaming using the STREAM pragma/directive if a FIFO is desired for the implementation.



TIP: When the STREAM directive is applied to an array, the resulting FIFO implemented in the hardware contains as many elements as the array. The -depth option can be used to specify the size of the FIFO.

The STREAM directive is also used to change any arrays in a DATAFLOW region from the default implementation specified by the config_dataflow configuration.

- If the config_dataflow default_channel is set as ping-pong, any array can still be implemented as a FIFO by applying the STREAM directive to the array.



TIP: To use a FIFO implementation, the array must be accessed in sequential order (and not in random access order).

- If the config_dataflow default_channel is set to FIFO, any array can still be implemented as a ping-pong implementation by applying the STREAM directive to the array with the type=piwo option.



IMPORTANT! To preserve sequential accesses, and thus the correctness of streaming, it might be necessary to prevent compiler optimizations (dead code elimination particularly) by using the volatile qualifier. In this case the HLS tool generates a warning if it cannot determine that access to the array is sequential.

When an array in a DATAFLOW region is specified as streaming and implemented as a FIFO, the FIFO is typically not required to hold the same number of elements as the original array. The tasks in a DATAFLOW region consume each data sample as soon as it becomes available. The depth of the FIFO can be specified using the config_dataflow -fifo_depth option or the STREAM pragma or directive with the -depth option. This can be used to set the size of the FIFO to ensure the flow of data never stalls.

If the `-type=piro` option is selected, the `-depth` option sets the depth (number of blocks) of the PIPO. The depth should be at least 2.

Specifying Arrays as Stream-of-Blocks

The `hls::stream_of_blocks` type provides a user-synchronized stream that supports streaming blocks of data for process-level interfaces in a dataflow context, where each block is an array or multidimensional array. The intended use of stream-of-blocks is to replace array-based communication between a pair of processes within a dataflow region. Refer to the [using_stream_of_blocks](#) example on Github.

Currently, Vitis HLS implements arrays written by a producer process and read by a consumer process in a dataflow region by mapping them to ping pong buffers (PIPOs). The buffer exchange for a PIPO buffer occurs at the return of the producer function and the calling of the consumer function in C++.

Stream-of-Blocks Modeling Style

On the other hand, for a stream-of-blocks the communication between the producer and the consumer is modeled as a stream of array-like objects, providing several advantages over array transfer through PIPO.

The use of stream-of-blocks in your code requires the following include file:

```
#include "hls_streamofblocks.h"
```

The stream-of-blocks object template is:

```
hls::stream_of_blocks<block_type, depth> v
```

Where:

- `<block_type>` specifies the datatype of the array or multidimensional array held by the stream-of-blocks
- `<depth>` is an optional argument that provides depth control just like `hls::stream` or PIPOs, and specifies the total number of blocks, including the one acquired by the producer and the one acquired by the consumer at any given time. The default value is 2
- `v` specifies the variable name for the stream-of-blocks object

Use the following steps to access a block in a stream of blocks:

1. The producer or consumer process that wants to access the stream first needs to acquire access to it, using a `hls::write_lock` or `hls::read_lock` object.
2. After the producer has acquired the lock it can start writing (or reading) the acquired block. Once the block has been fully initialized, it can be released by the producer, when the `write_lock` object goes out of scope.

Note: The producer process with a `write_lock` can also read the block as long as it only reads from already written locations, because the newly acquired buffer must be assumed to contain uninitialized data. The ability to write and read the block is unique to the producer process, and is not supported for the consumer.

3. Then the block is queued in the stream-of-blocks in a FIFO fashion, and when the consumer acquires a `read_lock` object, the block can be read by the consumer process.

The main difference between `hls::stream_of_blocks` and the PIPO mechanism seen in the prior examples is that the block becomes available to the consumer as soon as the `write_lock` goes out of scope, rather than only at the return of the producer process. Hence the size of storage required to manage the original example (without the dataflow loop) is much less with stream-of-blocks than with just PIPOs: namely $2N$ instead of $2xMxN$ in the example.

Rewriting the prior example to use `hls::stream_of_blocks` is shown in the example below. The producer acquires the block by constructing an `hls::write_lock` object called `b`, and passing it the reference to the stream-of-blocks object, called `s`. The `write_lock` object provides an overloaded array access operator, letting it be accessed as an array to access underlying storage in random order as shown in the example below.

The acquisition of the lock is performed by constructing the `write_lock/read_lock` object, and the release occurs automatically when that object is destructed as it goes out of scope. This approach uses the common *Resource Acquisition Is Initialization (RAII)* style of locking and unlocking.

```
#include "hls_streamofblocks.h"
typedef int buf[N];
void producer(hls::stream_of_blocks<buf> &s, ...) {
    for (int i = 0; i < M; i++) {
        // Allocation of hls::write_lock acquires the block for the producer
        hls::write_lock<buf> b(s);
        for (int j = 0; j < N; j++)
            b[f(j)] = ...;
        // Deallocation of hls::write_lock releases the block for the consumer
    }
}

void consumer(hls::stream_of_blocks<buf> &s, ...) {
    for (int i = 0; i < M; i++) {
        // Allocation of hls::read_lock acquires the block for the consumer
        hls::read_lock<buf> b(s);
        for (int j = 0; j < N; j++)
            ... = b[g(j)] ...;
        // Deallocation of hls::read_lock releases the block to be reused by
        the producer
    }
}

void top(...) {
    #pragma HLS dataflow
```

```
hls::stream_of_blocks<buf> s;  
  
producer(b, ...);  
consumer(b, ...);  
}
```

The key features of this approach include:

- The expected performance of the outer loop in the producer above is to achieve an overall Initiation Interval (II) of 1
- A locked block can be used as though it were private to the producer or the consumer process until it is released.
- The initial state of the array object for the producer is undefined, whereas it contains the values written by the producer for the consumer.
- The principal advantage of stream-of-blocks is to provide overlapped execution of multiple iterations of the consumer and the producer to increase throughput.

Resource Usage

The resource cost when increasing the depth beyond the default value of 2 is similar to the resource cost of PIPOs. Namely, each increment of 1 will require enough memory for a block, e.g., in the example above $N * 32\text{-bit words}$.

The stream of blocks object can be bound to a specific RAM type, by placing the `BIND_STORAGE` pragma where the stream-of-blocks is declared, for example in the top-level function. The stream of blocks uses 2-port BRAM (`type=RAM_2P`) by default.

Specifying Compiler-Created FIFO Depth

Start Propagation FIFOs

The compiler might automatically create a start FIFO to propagate the `ap_start/ap_ready` handshake to an internal process. Such FIFOs can sometimes be a bottleneck for performance, in which case you can increase the default size (which can be incorrectly estimated by the tool) with the following command:

```
config_dataflow -start_fifo_depth <value>
```

If an unbounded slack between producer and consumer is needed, and internal processes can run forever, fully and safely driven by their inputs or outputs (FIFOs or PIPOs), these start FIFOs can be removed, at user's risk, locally for a given dataflow region with the pragma:

```
#pragma HLS DATAFLOW disable_start_propagation
```

Scalar Propagation FIFOs

The compiler automatically propagates some scalars from C/C++ code through scalar FIFOs between processes. Such FIFOs can sometimes be a bottleneck for performance or cause deadlocks, in which case you can set the size (the default value is set to `-fifo_depth`) with the following command:

```
config_dataflow -scalar_fifo_depth <value>
```

Stable Arrays

The `stable` pragma can be used to mark input or output variables of a dataflow region. Its effect is to remove their corresponding task-level synchronizations, assuming that the user guarantees this removal is indeed correct.

```
void dataflow_region(int A[...], ...
#pragma HLS stable variable=A
#pragma HLS dataflow
    proc1(...);
    proc2(A, ...);
```

Without the `stable` pragma, and assuming that `A` is read by `proc2`, then `proc2` would be part of the initial synchronization for the dataflow region where it is located. This means that `proc1` would not restart until `proc2` is also ready to start again, which would prevent dataflow iterations to be overlapped and induce a possible loss of performance. The `stable` pragma indicates that this synchronization is not necessary to preserve correctness.

With the `stable` pragma, the compiler assumes that:

- If `A` is read by `proc2`, then the memory locations that are read are still accessible and will not be overwritten by any other process or calling context, while the `dataflow_region` is being executed.
- If `A` is written by `proc2`, then the memory locations written will not be read, before their definition, by any other process or calling context, while `dataflow_region` is being executed.

A typical scenario is when the caller updates or reads these variables only when the dataflow region has not started yet or has completed execution.

In summary, the Dataflow optimization is a powerful optimization that can significantly improve the throughput of your design. As there is reliance on the HLS tool to do the inference of the available parallelism in your design, it requires the designer's help to ensure that the code is written in such a way that the inference is straightforward for the HLS tool. Finally, there will be situations where the designer might see the need to deploy both the Dataflow model and the Task-Channel model in the same design. The next section describes this hybrid combination model that can lead to some interesting designs.

Mixing Data-Driven and Control-Driven Models

The following table highlights factors of the HLS design that can help you determine when to apply control-driven task-level parallelism (TLP) or data-driven TLP.

Control-Driven TLP	Data-Driven TLP
<ul style="list-style-type: none">• HLS Design requires control signals to start/stop the process• Design requires non-local memory access• Design requires interaction with an external software application• Designs with multiple processes running the same number of executions• Requires RTL simulation to model the effect of the parallelism	<ul style="list-style-type: none">• HLS Design uses a completely data-driven approach that does not require control signals to start/stop the process• Design uses pure streaming data transfers• Designs with data-dependent multi-rate behavior<ul style="list-style-type: none">◦ Producer writes data or consumer reads data at a rate that is data dependent◦ Easier to model for designs that require feedback between processes• Task-level parallelism is observable in C simulation as well as RTL simulation.

As the above table indicates, the two forms of task-level parallelism presented have different use cases and advantages. However, sometimes it is not possible to design an entire application that is purely data-driven TLP, while some portion of the design can still be constructed as a purely streaming design. In this case a mixed control-driven/data-driven model can be useful to create the application. Consider the following [mixed_control_and_data_driven](#) example from GitHub.

```
void dut(int in[N], int out[N], int n) {
#pragma HLS dataflow
    hls_thread_local hls::split::round_robin<int, NP> split1;
    hls_thread_local hls::merge::round_robin<int, NP> merge1;

    read_in(in, n, split1.in);

    // Task-Channels
    hls_thread_local hls::task t[NP];
    for (int i=0; i<NP; i++) {
#pragma HLS unroll
        t[i](worker, split1.out[i], merge1.in[i]);
    }

    write_out(merge1.out, out, n);
}
```

In the above example, there are two distinct regions - a dataflow region that has the functions `read_in/write_out` in which the sequential semantics is preserved - i.e. `read_in` will be executed before `write_out` and a task-channel region that contains the dynamic instantiation of 4 tasks (since `NP = 4` in this example) along with some special type of channels called a `split` or a `merge` channel. A split channel is one that has a single input but has multiple outputs - in this case, the split channel has 4 outputs as described in [HLS Split/Merge Library](#). Similarly, a merge channel has multiple inputs but only one output.

In addition, to the ports, these channels also support an internal job scheduler. In the above example, both the merge and the split channels have selected a round-robin scheduler that assigns the incoming data to each of the 4 tasks, one by one starting with `worker_U0`. If a load balancing scheduler had been chosen then the incoming data will have been assigned to the first available worker task (and this would lead to a non-deterministic simulation since this order might be different each time you run the simulation). Since this is a pure task-channel region, the 4 tasks are executed in parallel as soon as there is data in their incoming stream. Refer to the [merge_split](#) example on Github for more examples of these concepts.

It is important to note that, although the code above may give the impression that each task is "called" in the loop, and connected to a potentially different pair of channels every time the loop body is executed, in reality, this usage implies a static instantiation, i.e.:

- each `t[i](...)` call must be executed exactly once per execution of `dut()`.
- the loop over `i` must be fully unrolled, to infer a corresponding set of 4 instances in RTL.
- The `dut()` function must be called exactly once by the testbench.
- Each split output or merge input must be bound to exactly one `hls::task` instance.

While it is true that for `hls::task` objects the order of specification does not matter, for the control-driven dataflow network, Vitis HLS must be able to see that there is a chain of processes, such as from `read_in` to `write_out`. To define this chain of processes, Vitis HLS uses the calling order, which for `hls::tasks` is also the declaration order. This means that the model must define an explicit order from the `read_in` function to the `hls::task` region and then finally to the `write_out` function in the dataflow region as shown in the example above.

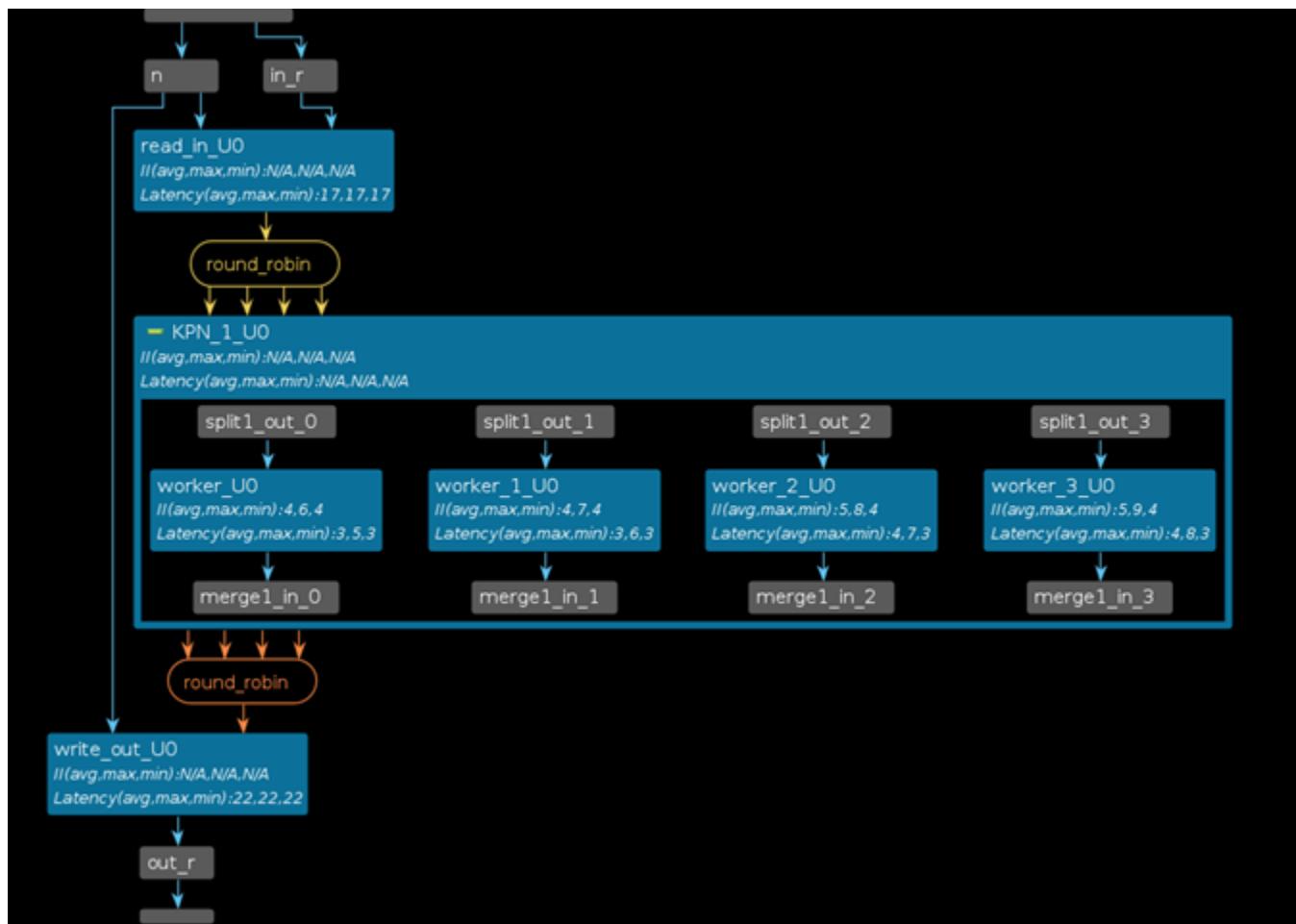
Generally:

- If a control-based process (i.e. regular dataflow) produces a stream for an `hls::task`, then it must be called before the declaration of the tasks in the code
- If a control-based process consumes a stream from an `hls::task`, then it must be called after the declaration of the tasks in the code

Violation of the above rules can cause unexpected outcomes since each of the `NP hls::task` instances is statically bound to the channels that are used in the first invocation of `t[i](...)`.

The following diagram shows the graph of this mixed task-channel and dataflow example:

Figure 15: Mixed Task-Channel and Dataflow



Summary

HLS designs that are purely data-driven and that do not require any interaction with the software application can be modeled using data-driven TLP models. Examples of such designs are:

- Simple rule-based “firewall” with “rules” compiled into the kernel
- Fast-Fourier Transforms with configuration data compiled into the kernel
- FIR filters with coefficients compiled into the kernel

If the design requires data transfer to/from external memory, then the control-driven TLP model can be used. Examples of such designs include:

- Network router where the routing table must be updated entirely for kernel execution

- Load-balancer that uses a hash map to send data to a server, that must update server list, server map, and corresponding IP addresses simultaneously

However, most designs will be a mixed control-driven and data-driven model, requiring some access to external memory, and enabling streaming between parallel and pipelined tasks within the HLS design.

In summary, this chapter described some modeling choices to consider when designing your application written in C/C++. So far, this discussion talked about structuring your algorithm at a high level to make use of these special models such as the task-channel or dataflow optimizations. Another key aspect to achieving good throughput is to also consider instruction-level parallelism. Instruction-level parallelism in HLS refers to the ability to efficiently parallelize the operations inside loops, functions, and even arrays. The next few sections will walk you through these lower-level optimizations that work hand-in-hand with the macro-level optimizations.

Loops Primer

When writing code intended for high-level synthesis (HLS), there is a frequent need to implement repetitive algorithms that process blocks of data – for example, signal or image processing. Typically, the C/C++ source code tends to include several loops or several nested loops.

When it comes to optimizing performance, loops are one of the best places to start exploring optimization. Each iteration of the loop takes at least one clock cycle to execute in hardware. Thinking from the hardware perspective, there is an implicit *wait until clock* for the loop body. The next iteration of a loop only starts when the previous iteration is finished. To improve performance loops can generally be either *pipelined* or *unrolled* to take advantage of the highly distributed and parallel FPGA architecture, as explained in the following sections.

Pipelining Loops

Pipelining loops permits starting the next iteration of a loop before the previous iteration finishes, enabling portions of the loop to overlap in execution. By default, every iteration of a loop only starts when the previous iteration has finished. In the loop example below, a single iteration of the loop adds two variables and stores the result in a third variable. Assume that in hardware this loop takes three cycles to finish one iteration. Also, assume that the loop variable `len` is 20, that is, the `vadd` loop runs for 20 iterations in the kernel. Therefore, it requires a total of 60 clock cycles (20 iterations * 3 cycles) to complete all the operations of this loop.

```
vadd: for(int i = 0; i < len; i++) {  
    c[i] = a[i] + b[i];  
}
```



TIP: It is good practice to always label a loop as shown in the example above (`vadd : ...`). This practice helps with debugging the design in Vitis HLS. Sometimes the unused labels generate warnings during compilation, which can be safely ignored.

Pipelining the loop allows subsequent iterations of the loop to overlap and run concurrently. Pipelining a loop can be enabled by adding the pragma HLS pipeline inside the body of the loop as shown below:

```
vadd: for(int i = 0; i < len; i++) {  
#pragma HLS PIPELINE  
    c[i] = a[i] + b[i];  
}
```

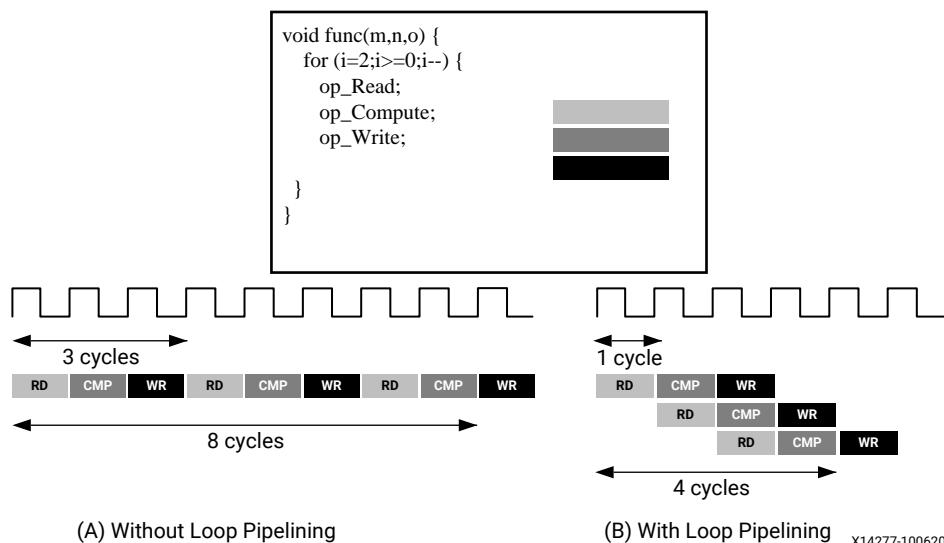


TIP: Vitis HLS automatically pipelines loops with 64 iterations or more. This feature can be changed or disabled using the `config_compile -pipeline_loops` command.

The number of cycles it takes to start the next iteration of a loop is called the Initiation Interval (II) of the pipelined loop. So II = 2 means the next iteration of a loop starts two cycles after the current iteration. An II = 1 is the ideal case, where each iteration of the loop starts in the very next cycle. When you use **pragma HLS pipeline**, you can specify the II for the compiler to achieve. If a target II is not specified, the compiler will try to achieve II=1 by default.

The following figure illustrates the difference in execution between pipelined and non-pipelined loops. In this figure, (A) shows the default sequential operation where there are three clock cycles between each input read (II = 3), and it requires eight clock cycles before the last output write is performed.

Figure 16: Loop Pipelining



In the pipelined version of the loop shown in (B), a new input sample is read every cycle (II = 1) and the final output is written after only four clock cycles: substantially improving both the II and latency while using the same hardware resources.



IMPORTANT! Pipelining a loop causes any loops nested inside the pipelined loop to get automatically unrolled.

If there are data dependencies inside a loop, it might not be possible to achieve $II = 1$, and a larger initiation interval might be the result. Loop dependencies are data dependencies that may constrain the optimization of loops, typically pipelining. They can be within a single iteration of a loop and or between different iterations of a loop. The easiest way to understand loop dependencies is to examine an extreme example. In the following example, the result of the loop is used as the loop continuation or exit condition. Each iteration of the loop must finish before the next can start.

```
Minim_Loop: while (a != b) {  
    if (a > b)a -= b;  
    else b -= a;  
}
```

The `Minim_Loop` loop in the example above cannot be pipelined because the next iteration of the loop cannot begin until the previous iteration ends. Not all loop dependencies are as extreme as this, but the example highlights that some operations cannot begin until some other operation has been completed. The solution is to try to ensure that the initial operation is performed as early as possible.

Loop dependencies can occur with any and all types of data. They are particularly common when using arrays.

Automatic Loop Pipelining

The `config_compile -pipeline_loops` command enables loops to be pipelined automatically based on the iteration count. All loops with an iteration count below the specified limit are automatically pipelined. The default is 64.

Given the following example code:

```
for (y = 0; y < 480; y++) {  
    for (x = 0; x < 640; x++) {  
        for (i = 0; i < 5; i++) {  
            // do something 5 times ...  
        }  
    }  
}
```

If the `pipeline_loops` option is set to 6, the innermost `for` loop in the above code snippet will be automatically pipelined. This is equivalent to the following code snippet:

```
for (y = 0; y < 480; y++) {  
    for (x = 0; x < 640; x++) {  
        for (i = 0; i < 5; i++) {  
            #pragma HLS PIPELINE II=1  
            // do something 5 times ...  
        }  
    }  
}
```

If there are loops in the design for which you do not want to use automatic pipelining, apply the PIPELINE directive with the `off` option to that loop. The `off` option prevents automatic loop pipelining.



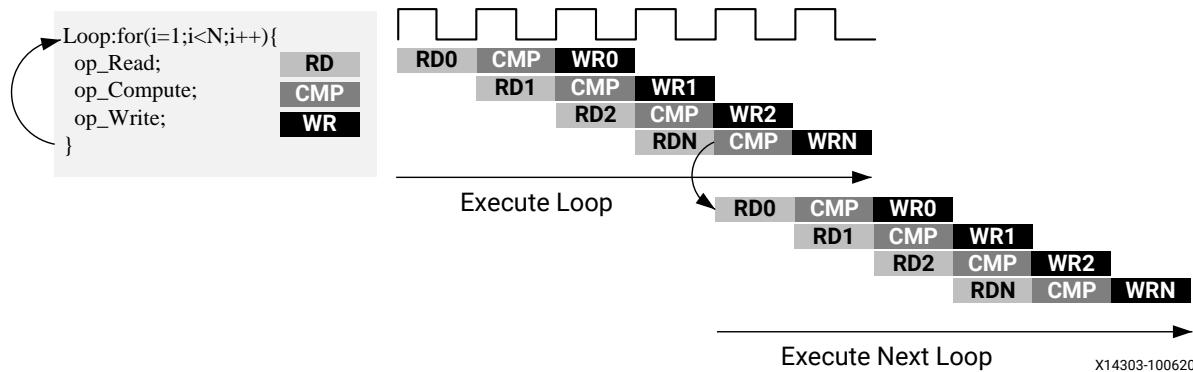
IMPORTANT! Vitis HLS applies the `config_compile -pipeline_loops` command after performing all user-specified directives. For example, if Vitis HLS applies a user-specified UNROLL directive to a loop, the loop is first unrolled, and automatic loop pipelining cannot be applied.

Rewinding Pipelined Loops for Performance

The PIPELINE pragma has an option called `rewind`. This option enables overlap of the execution of successive calls to the pipelined loop when this loop is the outermost construct of the top function, or of a dataflow region and the dataflow region is executed multiple times.

The following figure shows the operation when the `rewind` option is used when pipelining a loop. At the end of the loop iteration count, the loop starts to execute again. While it generally re-executes immediately, a delay is possible and is shown and described in the GUI.

Figure 17: Loop Pipelining with Rewind Option



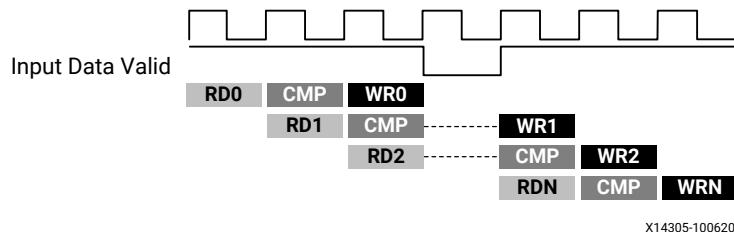
Note: If a loop is used around a DATAFLOW region, Vitis HLS automatically implements it to allow successive executions to overlap.

Flushing Pipelines and Pipeline Types

Flushing Pipelines

Pipelines continue to execute as long as data is available at the input of the pipeline. If there is no data available to process, the pipeline will stall. This is shown in the following figure, where the `Input Data Valid` signal goes low to indicate there is no more valid input data. Once the signal goes high, indicating there is new data available to process, the pipeline will continue operation.

Figure 18: Loop Pipelining with Stall



In some cases, it is desirable to have a pipeline that can be “emptied” or “flushed.” The `flush` option is provided to perform this. When a pipeline is “flushed” the pipeline stops reading new inputs when none are available (as determined by a `data valid` signal at the start of the pipeline) but continues processing, shutting down each successive pipeline stage, until the final input has been processed through to the output of the pipeline.

As described below, Vitis HLS will automatically select the right pipeline style to use for a given pipelined loop or function. However, you can override this default behavior by using the `config_compile -pipeline_style` command to specify the default pipeline style. You can also specify stalling pipelines (`sdp`), or a flushable pipeline (`f1p`) with the `PIPELINE` pragma or directive, using the `enable_flush` option. This option applies to the specific scope of the pragma or directive and does not change the global default assigned by `config_compile`.

Both `sdp` and `f1p` types of pipelines use the standard pipeline logic where the hardware pipeline created use various kinds of blocking signals to stall the pipeline. These blocking signals often become the driver of a high-fanout net, especially on pipelines that are deep in the number of physical stages and work on significant data sizes. Such high fanout nets, when they are created, are the prime cause of timing closure issues which cannot be fixed in RTL/Logic Synthesis or during Place-and-Route. To solve this issue, a new type of pipeline implementation called *free-running pipeline* (or `frp`) was created. The free-running pipeline is the most efficient architecture for handling a pipeline that operates with blocking signals. This is because

- It completely eliminates the blocking signal connections to the register enables
- It is a fully flushable pipeline, which allows bubbling invalid transactions
- Unlike the previous architectures which distribute fanouts (across flops), this reduces the fanouts
- It does not rely on the synthesis and/or place and route optimizations such as flop cloning
- This helps PnR by creating a structure where the wire length is reduced along with the reduction of the high fanouts

But there is a cost associated with this fanout reduction:

- the size of the FIFO buffers required for the blocking output ports causes additional resource usage.
- the mux delay at those blocking output ports

- the potential performance hit due to early validation of forward-pressure triggers



IMPORTANT! The free-running pipeline (`frp`) can only be called from within a DATAFLOW region. The `frp` style cannot be applied to a loop that is called in a sequential or a pipelined region.

Pipeline Types

The three types of pipelines available in the tool are summarized in the following table. Vitis HLS will automatically select the right pipeline style to use for a given pipelined loop or function. If the pipeline is used with `hls::tasks`, the flushing pipeline (FLP) style is automatically selected to avoid deadlocks. If the pipeline control requires high fanout, and meets other free-running requirements, the tool will select the free-running pipeline (FRP) style to limit the high fanout. Finally, if neither of the above cases apply, then the standard pipeline (STP) style is selected.

Table 1: Pipelining Types

Name	Stalled Pipeline	Free-Running/ Flushable Pipeline	Flushable Pipeline
Use cases	<ul style="list-style-type: none"> When there is no timing issue due to high fanout on pipeline control When flushable is not required (such as no performance or deadlock issue due to stall) 	<ul style="list-style-type: none"> When you need better timing due to fanout to register enables from pipeline control When flushable is required for better performance or avoiding deadlock Can only be called from a dataflow region. 	<ul style="list-style-type: none"> When flushable is required for better performance or avoiding deadlock
Pragma/Directive	<code>#pragma HLS pipeline style=stp</code>	<code>#pragma HLS pipeline style=frp</code>	<code>#pragma HLS pipeline style=fhp</code>
Global Setting	<code>config_compile - pipeline_style stp (default)</code>	<code>config_compile - pipeline_style frp</code>	<code>config_compile - pipeline_style fhp</code>
Disadvantages	<ul style="list-style-type: none"> Not flushable, hence it can: <ul style="list-style-type: none"> Cause more deadlocks in dataflow Prevent already computed outputs from being delivered, if the inputs to the next iterations are missing Timing issues due to high fanout on pipeline controls 	<ul style="list-style-type: none"> Moderate resource increase due to FIFOs added on outputs Requires at least one blocking I/O (stream or ap_hs). Not all pipelining scenarios and I/O types are supported. 	<ul style="list-style-type: none"> Can have larger II Greater resource usage due to less sharing when II>1

Table 1: Pipelining Types (cont'd)

Name	Stalled Pipeline	Free-Running/ Flushable Pipeline	Flushable Pipeline
Advantages	<ul style="list-style-type: none">Default pipeline. No usage constraints.Typically the lowest overall resource usage.	<ul style="list-style-type: none">Better timing due to<ul style="list-style-type: none">Less fanoutSimpler pipeline control logicFlushable	<ul style="list-style-type: none">Flushable

Managing Pipeline Dependencies

Vitis HLS constructs a hardware datapath that corresponds to the C/C++ source code. When there is no pipeline directive, the execution is always sequential and so there are no dependencies that the tool needs to take into account. But when features of the design has been pipelined, the tool needs to ensure that any possible dependencies are respected in the hardware that Vitis HLS generates.

Typical cases of *data dependencies* or *memory dependencies* are when a read or a write occurs after a previous read or write.

- A read-after-write (RAW), also called a true dependency, is when an instruction (and data it reads/uses) depends on the result of a previous operation.
 - I1: $t = a * b;$
 - I2: $c = t + 1;$The read in statement I2 depends on the write of t in statement I1. If the instructions are reordered, it uses the previous value of t .

- A write-after-read (WAR), also called an anti-dependence, is when an instruction cannot update a register or memory (by a write) before a previous instruction has read the data.
 - I1: $b = t + a;$
 - I2: $t = 3;$The write in statement I2 cannot execute before statement I1, otherwise the result of b is invalid.

- A write-after-write (WAW) is a dependence when a register or memory must be written in specific order otherwise other instructions might be corrupted.
 - I1: $t = a * b;$
 - I2: $c = t + 1;$
 - I3: $t = 1;$

The write in statement I3 must happen after the write in statement I1. Otherwise, the statement I2 result is incorrect.

- A read-after-read has no dependency as instructions can be freely reordered if the variable is not declared as volatile. If it is, then the order of instructions has to be maintained.

For example, when a pipeline is generated, the tool needs to take care that a register or memory location read at a later stage has not been modified by a previous write. This is a true dependency or read-after-write (RAW) dependency. A specific example is:

```
int top(int a, int b) {  
    int t,c;  
    I1: t = a * b;  
    I2: c = t + 1;  
    return c;  
}
```

Statement I2 cannot be evaluated before statement I1 completes because there is a dependency on variable t. In hardware, if the multiplication takes 3 clock cycles, then I2 is delayed for that amount of time. If the above function is pipelined, then Vitis HLS detects this as a true dependency and schedules the operations accordingly. It uses data forwarding optimization to remove the RAW dependency, so that the function can operate at II =1.

Memory dependencies arise when the example applies to an array and not just variables.

```
int top(int a) {  
    int r=1,rnext,m,i,out;  
    static int mem[256];  
    L1: for(i=0;i<=254;i++) {  
        #pragma HLS PIPELINE II=1  
        I1: m = r * a; mem[i+1] = m; // line 7  
        I2: rnext = mem[i]; r = rnext; // line 8  
    }  
    return r;  
}
```

In the above example, scheduling of loop L1 leads to a scheduling warning message:

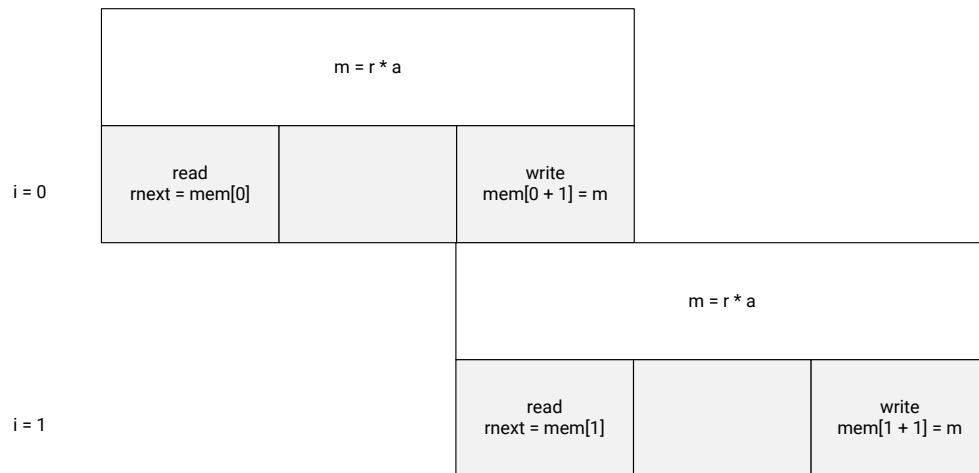
```
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint  
(II = 1, distance = 1) between 'store' operation (top.cpp:7) of variable  
'm', top.cpp:7  
on array 'mem' and 'load' operation ('rnext', top.cpp:8) on array 'mem'.  
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

There are no issues within the same iteration of the loop as you write an index and read another one. The two instructions could execute at the same time, concurrently. However, observe the read and writes over a few iterations:

```
// Iteration for i=0
I1:     m = r * a; mem[1] = m;          // line 7
I2:     rnext = mem[0]; r = rnext; // line 8
// Iteration for i=1
I1:     m = r * a; mem[2] = m;          // line 7
I2:     rnext = mem[1]; r = rnext; // line 8
// Iteration for i=2
I1:     m = r * a; mem[3] = m;          // line 7
I2:     rnext = mem[2]; r = rnext; // line 8
```

When considering two successive iterations, the multiplication result `m` (with a latency = 2) from statement `I1` is written to a location that is read by statement `I2` of the next iteration of the loop into `rnext`. In this situation, there is a RAW dependence as the next loop iteration cannot start reading `mem[i]` before the previous computation's write completes.

Figure 19: Dependency Example



X24685-100620

Note that if the clock frequency is increased, then the multiplier needs more pipeline stages and increased latency. This will force `II` to increase as well.

Consider the following code, where the operations have been swapped, changing the functionality:

```
int top(int a) {
    int r,m,i;
    static int mem[256];
    L1: for(i=0;i<=254;i++) {
        #pragma HLS PIPELINE II=1
```

```
I1:     r = mem[i];           // line 7
I2:     m = r * a, mem[i+1]=m; // line 8
}
return r;
}
```

The scheduling warning is:

```
INFO: [SCHED 204-61] Pipelining loop 'L1'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint
(II = 1,
distance = 1)
between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array
'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint
(II = 2,
distance = 1)
between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array
'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint
(II = 3,
distance = 1)
between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array
'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 4, Depth: 4.
```

Observe the continued read and writes over a few iterations:

```
Iteration with i=0
I1:     r = mem[0];           // line 7
I2:     m = r * a, mem[1]=m; // line 8
Iteration with i=1
I1:     r = mem[1];           // line 7
I2:     m = r * a, mem[2]=m; // line 8
Iteration with i=2
I1:     r = mem[2];           // line 7
I2:     m = r * a, mem[3]=m; // line 8
```

A longer II is needed because the RAW dependence is via reading `r` from `mem[i]`, performing multiplication, and writing to `mem[i+1]`.

Removing False Dependencies to Improve Loop Pipelining

False dependencies are dependencies that arise when the compiler is too conservative. These dependencies do not exist in the real code, but cannot be determined by the compiler. These dependencies can prevent loop pipelining.

The following example illustrates false dependencies. In this example, the read and write accesses are to two different addresses in the same loop iteration. Both of these addresses are dependent on the input data, and can point to any individual element of the `hist` array. Because of this, Vitis HLS assumes that both of these accesses can access the same location. As a result, it schedules the read and write operations to the array in alternating cycles, resulting in a loop II of 2. However, the code shows that `hist[old]` and `hist[val]` can never access the same location because they are in the `else` branch of the conditional `if(old == val)`.

```
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE]) f
    int acc = 0;
    int i, val;
    int old = in[0];
    for(i = 0; i < INPUT SIZE; i++)
    {
        #pragma HLS PIPELINE II=1
        val = in[i];
        if(old == val)
        {
            acc = acc + 1;
        }
        else
        {
            hist[old] = acc;
            acc = hist[val] + 1;
        }

        old = val;
    }

    hist[old] = acc;
```

To overcome this deficiency, you can use the `DEPENDENCE` directive to provide Vitis HLS with additional information about the dependencies.

```
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE]) {
    int acc = 0;
    int i, val;
    int old = in[0];
    #pragma HLS DEPENDENCE variable=hist type=intra direction=RAW
dependent=false
    for(i = 0; i < INPUT SIZE; i++)
    {
        #pragma HLS PIPELINE II=1
        val = in[i];
        if(old == val)
        {
            acc = acc + 1;
        }
        else
        {
            hist[old] = acc;
            acc = hist[val] + 1;
        }

        old = val;
    }

    hist[old] = acc;
```



IMPORTANT! Specifying a FALSE dependency, when in fact the dependency is not FALSE, can result in incorrect hardware. Be sure dependencies are correct (TRUE or FALSE) before specifying them.

When specifying dependencies there are two main types:

- Inter - Specifies the dependency is between different iterations of the same loop. If this is specified as FALSE it allows Vitis HLS to perform operations in parallel if the pipelined or loop is unrolled or partially unrolled and prevents such concurrent operation when specified as TRUE.
- Intra - Specifies dependence within the same iteration of a loop, for example an array being accessed at the start and end of the same iteration. When intra dependencies are specified as FALSE, Vitis HLS may move operations freely within the loop, increasing their mobility and potentially improving performance or area. When the dependency is specified as TRUE, the operations must be performed in the order specified.

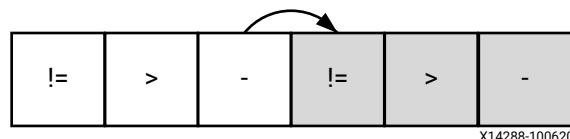
Scalar Dependencies

Some scalar dependencies are much harder to resolve and often require changes to the source code. A scalar data dependency could look like the following:

```
while (a != b) {  
    if (a > b) a -= b;  
    else b -= a;  
}
```

The next iteration of this loop cannot start until the current iteration has calculated the updated the values of `a` and `b`, as shown in the following figure.

Figure 20: Scalar Dependency



If the result of the previous loop iteration must be available before the current iteration can begin, loop pipelining is not possible. If Vitis HLS cannot pipeline with the specified initiation interval, it increases the initiation internal. If it cannot pipeline at all, as shown by the above example, it halts pipelining and proceeds to output a non-pipelined design.

Unrolling Loops

A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations might also be impacted by logic inside the loop body (for example, `break` conditions or modifications to a loop exit variable). You can unroll loops to create multiple copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel. Using the `UNROLL` pragma you can unroll loops to increase data access and throughput.

By default, HLS loops are kept rolled. This means that each iteration of the loop uses the same hardware. Unrolling the loop means that each iteration of the loop has its own hardware to perform the loop function. This means that the performance for unrolled loops can be significantly better than for rolled loops. However, the added performance comes at the expense of added area and resource utilization.

Consider the [basic_loops_primer](#) example from GitHub, as shown below:

```
#include "test.h"

dout_t test(din_t A[N]) {
    dout_t out_accum=0;
    dsel_t x;

    LOOP_1:for (x=0; x<N; x++) {
        out_accum += A[x];
    }
    return out_accum;
}
```

With no optimization, the Synthesis Summary report in the figure below shows that the implementation is sequential. This can be confirmed by looking at the trip count for `LOOP_1`, which reports the number of iterations as 10 and the Latency as 200. The latency is the time before the loop can accept new input values.

Figure 21: Performance & Resource Estimates

The screenshot shows a software interface for performance analysis. At the top, there's a toolbar with various icons. Below it is a header bar with tabs like 'Performance & Resource Estimates' and dropdown menus for 'Modules' and 'Loops'. The main area is a table titled 'Modules & Loops' with several columns: Issue Type, Violation Type, Distance, Slack, Latency(cycles), Latency(ns), Iteration Latency, Interval, Trip Count, Pipelined, BRAM, DSP, FF, LUT, and URAM. There are two rows in the table: one for the 'test' module and one for 'LOOP_1'. The 'test' row has a green circle icon and shows a trip count of 10. The 'LOOP_1' row has a blue square icon and shows a trip count of 10. The 'Latency(cycles)' column for 'test' is 21, and for 'LOOP_1' it is 20. The 'Latency(ns)' column for 'test' is 210.000, and for 'LOOP_1' it is 200.000. The 'Iteration Latency' column for 'test' is 22, and for 'LOOP_1' it is 2. The 'Interval' column for 'test' is '-' and for 'LOOP_1' it is '-'. The 'Trip Count' column for both is 10. The 'Pipelined' column for both is 'no'. The resource usage columns (BRAM, DSP, FF, LUT, URAM) show values of 0, 0, 19, 78, and 0 respectively for both rows.

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
test			-	-	21	210.000	-	22	-	no	0	0	19	78	0
LOOP_1			-	-	20	200.000	2	-	10	no	-	-	-	-	-

To get optimal throughput, the latency needs to be as short as possible. To increase performance, assuming the loop bounds are static, the loop can be fully unrolled using the [UNROLL](#) pragma to create parallel implementations of the loop body. After the LOOP_1 is fully unrolled a significant reduction in the latency (50ns) is shown in the figure below. Unrolling loops implies a trade-off by achieving higher performance but at the cost of using extra resources (as seen below in the increase of FFs and LUTs). Fully unrolling the loop will also cause the loop itself to disappear and be replaced by the parallel implementations of the loop body which will use up the extra resources as shown below.

Figure 22: Performance & Resource Estimates

Performance & Resource Estimates															
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
test					5	50.000		6	-	no	0	0	51	261	0

Of course, there will be cases where it is not possible to unroll the loop completely due to the increase in resources and the available resources of the platform. In this situation, partially unrolled loops can be the preferred solution offering some improvement of performance while not requiring as many resources. To partially unroll a loop you will define an unroll factor for the pragma or directive. Unrolling the same loop with a factor of 2 (which implies that the loop body is duplicated and the trip count is reduced by half to 5) can be an acceptable solution for this constrained case as shown below.

Figure 23: Performance & Resource Estimates

Performance & Resource Estimates															
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
test					11	110.000		12	-	no	0	0	19	98	0
	LOOP_1				10	100.000	2	-	5	no	-	-	-	-	-

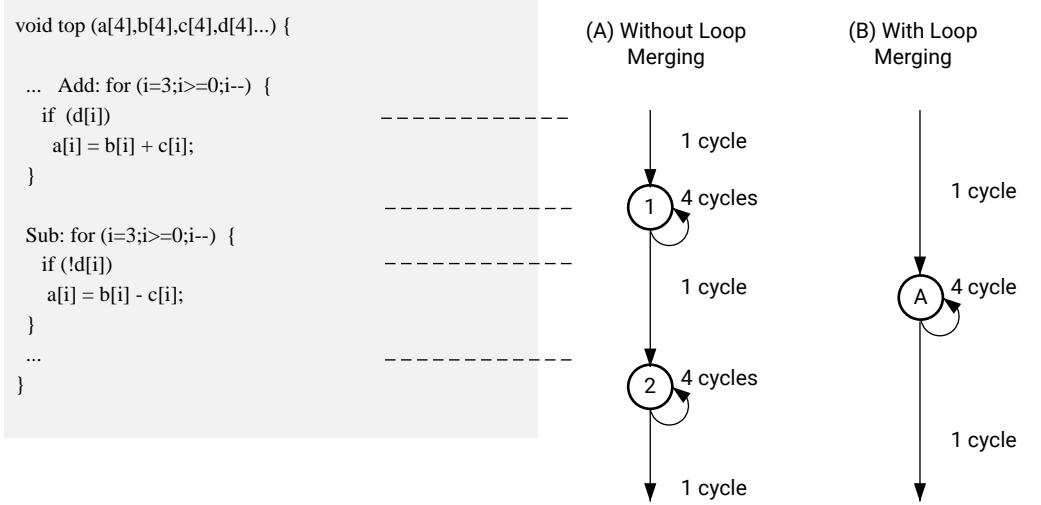
Additionally, when you partially unroll the loop, the HLS tool will implement an exit check in the loop in case the trip count is not perfectly divisible by the unroll factor. The exit check is skipped if the trip count is perfectly divisible by the unroll factor.

Merging Loops

All rolled loops imply and create at least one state in the design FSM. When there are multiple sequential loops it can create additional unnecessary clock cycles and prevent further optimizations.

The following figure shows a simple example where a seemingly intuitive coding style has a negative impact on the performance of the RTL design.

Figure 24: Loop Directives



X14276-100620

In the preceding figure, (A) shows how, by default, each rolled loop in the design creates at least one state in the FSM. Moving between those states costs clock cycles: assuming each loop iteration requires one clock cycle, it takes a total of 11 cycles to execute both loops:

- 1 clock cycle to enter the ADD loop.
- 4 clock cycles to execute the add loop.
- 1 clock cycle to exit ADD and enter SUB.
- 4 clock cycles to execute the SUB loop.
- 1 clock cycle to exit the SUB loop.
- For a total of 11 clock cycles.

In this simple example, it is obvious that an else branch in the ADD loop would also solve the issue but in a more complex example it may be less obvious and the more intuitive coding style may have greater advantages.

The LOOP_MERGE optimization directive is used to automatically merge loops. The loop merge optimization directive will seek to merge all loops within the scope it is placed. In the above example, merging the loops creates a control structure similar to that shown in (B) in the preceding figure, which requires only 6 clocks to complete.

Merging loops allows the logic within the loops to be optimized together. The loop merging transformation has limitations and may not always succeed. However, it is still possible to manually merge the loops by refactoring the code. In the example above, using a dual-port block RAM allows the add and subtraction operations to be performed in parallel.

Working with Nested Loops

To get the best performance (lowest latency) when working with nested loops, it becomes crucial to create perfectly nested loops. In a perfect nested loop, the loop bounds are constant and only the innermost loop contains any functionality (as shown below)

```
Perfect_nested_loop_1: for (int i = 0; i < N; ++i) {  
    Perfect_nested_loop_2: for (int j = 0; j < M; ++j) {  
        // Perfect Nested Loop Code goes here and no where else  
    }  
}  
  
Imperfect_nested_loop_1: for (int i = 0; i < N; ++i) {  
    // Imperfect Nested Loop Code contains code here  
    Imperfect_nested_loop_2: for (int j = 0; j < M; ++j) {  
        // Imperfect Nested Loop Code goes here  
    }  
    // Imperfect Nested Loop Code may contain code here as well  
}
```

- **Perfect loop nest:** Only the innermost loop has loop body content, there is no logic specified between the loop statements and all the loop bounds are constant
- **Semi-perfect loop nest:** Only the innermost loop has loop body content, there is no logic specified between the loop statements but the outermost loop bound can be a variable.
- **Imperfect loop nest:** The inner loop has variable bounds or the loop body is not exclusively inside the inner loop. In this case designers should try to restructure the code or unroll the loops in the loop body to create a perfect loop nest.

It also requires additional clock cycles to move between rolled nested loops. It requires one clock cycle to move from an outer loop to an inner loop or from an inner loop to an outer loop. In the small example shown here, this implies 200 extra clock cycles to execute the loop Outer.

```
void foo_top { a, b, c, d} {  
    ...  
    Outer: while(j<100)  
        Inner: while(i<6) // 1 cycle to enter inner  
        ...  
        LOOP_BODY  
        ...  
    } // 1 cycle to exit inner  
}  
...
```

The LOOP_FLATTEN pragma or directive is used to allow labeled perfect and semi-perfect nested loops to be flattened, removing the need to re-code for optimal hardware performance and reducing the number of cycles it takes to perform the operations in the loop. When the LOOP_FLATTEN optimization is applied to a set of nested loops, it should be applied to the innermost loop that contains the loop body. Loop flattening can also be performed either by applying it to individual loops or applying it to all loops in a function by applying the directive at the function level.

When pipelining nested loops, the optimal balance between area and performance is typically found by pipelining the innermost loop. This also results in the fastest runtime. The following code example, [pipelined_loop](#) available on GitHub, demonstrates the trade-offs when pipelining loops and functions.

```
#include "loop_pipeline.h"

dout_t loop_pipeline(din_t A[N]) {
    int i, j;
    static dout_t acc;

    LOOP_I:for(i=0; i < 20; i++){
        LOOP_J: for(j=0; j < 20; j++){
            acc += A[j] * i;
        }
    }
    return acc;
}
```

In the above example, if the innermost (LOOP_J) is pipelined, there is one copy of LOOP_J in hardware (a single multiplier). Vitis HLS automatically flattens the loops when possible, as in this case, and effectively creates a new single loop (now called LOOP_I_LOOP_J) with 20×20 iterations. Only one multiplier operation and one array access need to be scheduled, then the loop iterations can be scheduled as a single loop-body entity (20x20 loop iterations).



TIP: When a loop or function is pipelined, any loop in the hierarchy below the loop or function being pipelined must be unrolled.

If the outer loop (LOOP_I) is pipelined, inner-loop (LOOP_J) is unrolled creating 20 copies of the loop body: 20 multipliers and 1 array accesses must now be scheduled. Then each iteration of LOOP_I can be scheduled as a single entity.

If the top-level function is pipelined, both loops must be unrolled: 400 multipliers and 20 array accesses must now be scheduled. It is very unlikely that Vitis HLS will produce a design with 400 multiplications because, in most designs, data dependencies often prevent maximal parallelism, for example, even if a dual-port RAM is used for A, the design can only access two values of A in any clock cycle. Otherwise, the array must be partitioned into 400 registers, which then can all be read in one clock cycle, with a very significant HW cost.

The concept to appreciate when selecting at which level of the hierarchy to pipeline is to understand that pipelining the innermost loop gives the smallest hardware with generally acceptable throughput for most applications. Pipelining the upper levels of the hierarchy unrolls all sub-loops and can create many more operations to schedule (which could impact compile time and memory capacity), but typically gives the highest performance design in terms of throughput and latency. The data access bandwidth must be matched to the requirements of the operations that are expected to be executed in parallel. This implies that we might need to partition array A in order to make this work.

To summarize the above options:

- Pipeline `LOOP_J`: Latency is approximately 400 cycles (20×20) and requires less than 250 LUTs and registers (the I/O control and FSM are always present).

Figure 25: Performance & Resource Estimates

Performance & Resource Estimates															
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
loop_pipeline			-	-	405	1.013E4	-	406	-	no	0	1	68	171	0
LOOP_I_LOOP_J			-	-	403	1.008E4	5	1	400	yes	-	-	-	-	-

- Pipeline `LOOP_I`: Latency is 13 cycles but requires a few hundred LUTs and registers. About twice the logic as the first option, minus any logic optimizations that can be made.

Figure 26: Performance & Resource Estimates

Performance & Resource Estimates															
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
loop_pipeline			-	-	13	325.000	-	14	-	no	0	1	93	444	0

- Pipeline function `loop_pipeline`: Latency is now only 3 cycles (due to 20 parallel register accesses) but requires almost twice the logic as the second option (and about 4 times the logic of the first option), minus any optimizations that can be made.

Figure 27: Performance & Resource Estimates

Performance & Resource Estimates															
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
loop_pipeline			-	-	3	75.000	-	1	-	yes	0	1	240	649	0

Vitis HLS cannot flatten imperfect loop nests. This will result in additional clock cycles to enter and exit the loops. When the design contains nested loops, analyze the results to ensure that as many nested loops as possible have been flattened: review the log file or look in the synthesis report for cases (as shown above) where the loop labels have been merged (`LOOP_I` and `LOOP_J` are now reported as `LOOP_I_LOOP_J`).

Working with Variable Loop Bounds

Some of the optimizations that Vitis HLS can apply are prevented when the loop has variable bounds. In the following code example, [variable_bound_loops](#) on GitHub, the loop bounds are determined by the variable `width`, which is driven from a top-level input. In this case, the loop is considered to have a variable bound, because Vitis HLS cannot know when the loop will complete.

```
#include "ap_int.h"
#define N 32

typedef ap_int<8> din_t;
typedef ap_int<13> dout_t;
typedef ap_uint<5> dsel_t;

dout_t code028(din_t A[N], dsel_t width) {

    dout_t out_accum=0;
    dsel_t x;

    LOOP_X:for (x=0;x<width; x++) {
        out_accum += A[x];
    }

    return out_accum;
}
```

Attempting to optimize the design in the example above reveals the issues created by variable loop bounds. The first issue with variable loop bounds is that they prevent Vitis HLS from determining the latency of the loop. Vitis HLS can determine the latency to complete one iteration of the loop, but because it cannot statically determine the exact variable `width`, it does not know how many iterations are performed and thus cannot report the loop latency (the number of cycles to completely execute all iterations of the loop).

When variable loop bounds are present, Vitis HLS reports the latency as a question mark (?) instead of using exact values. The following shows the result after the synthesis of the previous example:

```
+ Summary of overall latency (clock cycles):
* Best-case latency:      ?
* Worst-case latency:     ?
+ Summary of loop latency (clock cycles):
+ LOOP_X:
* Trip count:    ?
* Latency:       ?
```

The way to overcome this issue is to use the LOOP_TRIPCOUNT pragma or directive to specify a minimum and/or maximum iteration count for the loop. The tripcount is the number of loop iterations. If a maximum tripcount of 32 is applied to LOOP_X in the first example, the report is updated to the following:

```
+ Summary of overall latency (clock cycles):
* Best-case latency:      2
* Worst-case latency:     34
+ Summary of loop latency (clock cycles):
+ LOOP_X:
* Trip count: 0 ~ 32
* Latency:     0 ~ 32
```

The user-provided values for the LOOP_TRIPCOUNT directive are used only for reporting, or to support the PERFORMANCE pragma or directive. The specified tripcount value allows Vitis HLS to determine latency values in the report, allowing values from different solutions to be compared. To have this same loop-bound information used for synthesis, the C/C++ code must be updated by using asserts, which impact synthesis (however, they must be used carefully since the assert condition is assumed to be true).

The next steps in optimizing the first example for a lower initiation interval are:

- Unroll the loop and allow the accumulations to occur in parallel.
- Partition the array input, or the parallel accumulations are limited by a single memory port.

If these code transformations are applied, the output from Vitis HLS highlights the most significant issue with variable bound loops:

```
WARNING: [HLS 200-936] Cannot unroll loop 'LOOP_X' (loop_var.cpp:22) in
function 'loop_var': cannot completely unroll a loop with a variable trip
count.
```

Because variable bounds loops cannot be fully unrolled, they not only prevent the unroll directive from being applied, they also prevent pipelining the levels above the loop.



IMPORTANT! When a loop or function is pipelined, Vitis HLS unrolls all loops in the hierarchy below the function or loop. If there is a loop with variable bounds in this hierarchy, it prevents pipelining.

The solution to loops with variable bounds is to make the number of loop iteration a fixed value with conditional executions inside the loop. The code from the variable loop bounds example can be rewritten as shown in the following code example. Here, the loop bounds are explicitly set to the maximum value of variable `width` and the loop body is conditionally executed:

```
#include "ap_int.h"
#define N 32

typedef ap_int<8> din_t;
typedef ap_int<13> dout_t;
typedef ap_uint<5> dsel_t;

dout_t loop_max_bounds(din_t A[N], dsel_t width) {

    dout_t out_accum=0;
    dsel_t x;

    LOOP_X:for (x=0; x<N; x++) {
        if (x<width) {
            out_accum += A[x];
        }
    }

    return out_accum;
}
```

The `for`-loop (`LOOP_X`) in the example above can be fully unrolled. Because the loop has fixed upper bounds, Vitis HLS knows how much hardware to create. There are `N(32)` copies of the loop body in the RTL design. Each copy of the loop body has conditional logic associated with it and is executed depending on the value of variable `width`. Refer to [Vitis-HLS-Introductory-Examples/Modeling/variable_bound_loops](#) for an example.

Arrays Primer

Mapping Software Arrays to Hardware Memory

Arrays are a fundamental data structure in any C++ software program. Software programmers view arrays as simply a container and allocate/deallocate arrays on demand - often dynamically. This type of dynamic memory allocation for arrays is not supported when the same program needs to be synthesized for hardware. For synthesizing arrays to hardware, knowing the exact amount of memory (statically) required for your algorithm becomes necessary. In addition, the memory architecture on FPGAs (also called "local memory") has very different trade-offs when compared to global memory which is often the DDR or HBM memory banks. Access to global memory has high latency costs and can take many cycles while access to local memory is often quick and only takes one or more cycles.

When an HLS design has been suitably pipelined and/or unrolled, the memory access pattern becomes established. Vitis HLS allows users to map arrays to various types of resources - where the array elements are available in parallel with or without handshaking signals. Both internal arrays and arrays in the top-level function's interface can be mapped to registers or memories. If the array is in the top-level interface, Vitis HLS automatically creates the address, data, and control signals required to interface to external memory. If the array is internal to the design, Vitis HLS not only creates the necessary address, data, and control signals to access the memory but also instantiates the memory model (which is then inferred as memory by the downstream RTL synthesis tool).

Arrays are typically implemented as memory (RAM, ROM, or shift registers) after synthesis. Arrays can also be fully partitioned into individual registers to create a fully parallel implementation provided the platform has enough registers to support this step. The [initialization_and_reset](#) example available on GitHub demonstrates different implementations of memory.

Arrays on the top-level function interface are synthesized as RTL ports that access external memory. Internal to the design, arrays sized less than 1024 will be synthesized as a shift register. Arrays sized greater than 1024 will be synthesized into block RAM (BRAM), LUTRAM, or UltraRAM (URAM) depending on the optimization settings (see *BIND_STORAGE directive/pragma*).

Consider the following example in which Vitis HLS infers a shift register when encountering the following code:

```
int A[N]; // This will be replaced by a shift register

for(...) {
    // The loop below is the shift operation
    for (int i = 0; i < N-1; ++i)
        A[i] = A[i+1];
    A[N] = ...;

    // This is an access to the shift register
    ... A[x] ...
}
```

Shift registers can perform a one-shift operation per cycle, and also allows random read access per cycle anywhere in the shift register, and thus is more flexible than a FIFO.

Cases in which arrays can create issues in the RTL include:

- When implemented as a memory (BRAM/LUTRAM/URAM), the number of memory ports can limit access to the data leading to II violations in pipelined loops
- Mutually exclusive accesses may not be correctly inferred by Vitis HLS
- Some care must be taken to ensure arrays that only require read accesses are implemented as ROMs in the RTL.

Vitis HLS supports arrays of pointers. Each pointer can point only to a scalar or an array of scalars.



TIP: *Arrays must be sized. This is required even for function arguments (the size is ignored by the C++ compiler, but it is used by Vitis HLS), for example: `Array[10]:`. However, unsized arrays are not supported, for example: `Array[]:`.*

Array Accesses and Performance

In a previous section, we introduced optimization concepts such as loop unrolling and pipelining as a means for exploring parallelism. However, this was done without considering how array access patterns may prevent such optimizations when the arrays are mapped to memories instead of registers. Arrays mapped to memories can become the bottleneck in a design's performance. Vitis HLS provides a number of optimizations, such as array reshaping and array partitioning, that can remove these memory bottlenecks. Whenever possible, these automatic memory optimizations should be used, minimizing the number of code modifications. However, there may be situations where explicitly coding the memory architecture is either required to meet performance or may allow designers to achieve an even better quality of results. In these cases, it is essential that array accesses are coded in such a way as to not limit performance. This means analyzing array access patterns and organizing the memories in a design so that the

desired throughput and area can be achieved. The following code example shows a case in which access to an array can limit performance in the final RTL design. In this example, there are three accesses to the array `mem[N]` to create a summed result. Refer to [Vitis-HLS-Introductory-Examples/Interface/Memory/memory_bottleneck](#) for the full version of this example.

```
#include "array_mem_bottleneck.h"

dout_t array_mem_bottleneck(din_t mem[N]) {
    dout_t sum=0;
    int i;

    SUM_LOOP:for(i=2;i<N;++i)
        sum += mem[i] + mem[i-1] + mem[i-2];

    return sum;
}
```

During synthesis, the array is implemented as a RAM. If the RAM is specified as a single-port RAM it is impossible to pipeline loop `SUM_LOOP` to process a new loop iteration every clock cycle.

Trying to pipeline `SUM_LOOP` with an initiation interval of 1 results in the following message (after failing to achieve a throughput of 1, Vitis HLS relaxes the constraint):

```
INFO: [SCHED 61] Pipelining loop 'SUM_LOOP'.
WARNING: [SCHED 69] Unable to schedule 'load' operation ('mem_load_2',
bottleneck.c:62) on array 'mem' due to limited memory ports.
INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

The issue here is that the single-port RAM has only a single data port: only one read (or one write) can be performed in each clock cycle.

- `SUM_LOOP` Cycle1: read `mem[i]`;
- `SUM_LOOP` Cycle2: read `mem[i-1]`, sum values;
- `SUM_LOOP` Cycle3: read `mem[i-2]`, sum values;

A dual-port RAM could be used, but this allows only two accesses per clock cycle. Three reads are required to calculate the value of sum, and so three accesses per clock cycle are required to pipeline the loop with a new iteration every clock cycle.

The code in the example above can be rewritten as shown in the following code example to allow the code to be pipelined with a throughput of 1. In the following code example, by performing pre-reads and manually pipelining the data accesses, there is only one array read specified in each iteration of the loop. This ensures that only a single-port RAM is required to achieve the performance.

```
#include "array_mem_perform.h"

dout_t array_mem_perform(din_t mem[N]) {
    din_t tmp0, tmp1, tmp2;
    dout_t sum=0;
    int i;

    tmp0 = mem[0];
    tmp1 = mem[1];
    SUM_LOOP:for (i = 2; i < N; i++) {
        tmp2 = mem[i];
        sum += tmp2 + tmp1 + tmp0;
        tmp0 = tmp1;
        tmp1 = tmp2;
    }

    return sum;
}
```

Such changes to the source code as shown above are not always required. The more typical case is to use optimization directives/pragmas to achieve the same result. Vitis HLS includes optimization directives for changing how arrays are implemented and accessed. There are two main classes of optimization:

- Array Partition splits apart the original array into smaller arrays or into individual registers.
- Array Reshape reorganizes the array into a different memory arrangement to increase parallelism but without splitting apart the original array.

Array Partitioning

Arrays can be partitioned into blocks or into their individual elements. In some cases, Vitis HLS partitions arrays into individual elements. This is controllable using the configuration settings for auto-partitioning. When an array is partitioned into multiple blocks, the single array is implemented as multiple RTL RAM blocks. When partitioned into elements, each element is implemented as a register in the RTL. In both cases, partitioning allows more elements to be accessed in parallel and can help with performance; the design trade-off is between performance and the number of RAMs or registers required to achieve it.

A common issue when pipelining functions is the following message:

```
INFO: [SCHED 204-61] Pipelining loop 'SUM_LOOP'.
WARNING: [SCHED 204-69] Unable to schedule 'load' operation ('mem_load_2',
bottleneck.c:62) on array 'mem' due to limited memory ports.
WARNING: [SCHED 204-69] The resource limit of core:RAM:mem:p0 is 1, current
assignments:
```

```

WARNING: [SCHED 204-69]      'load' operation ('mem_load', bottleneck.c:62)
on array
'mem',
WARNING: [SCHED 204-69] The resource limit of core:RAM:mem:p1 is 1, current
assignments:
WARNING: [SCHED 204-69]      'load' operation ('mem_load_1',
bottleneck.c:62) on array
'mem',
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.

```

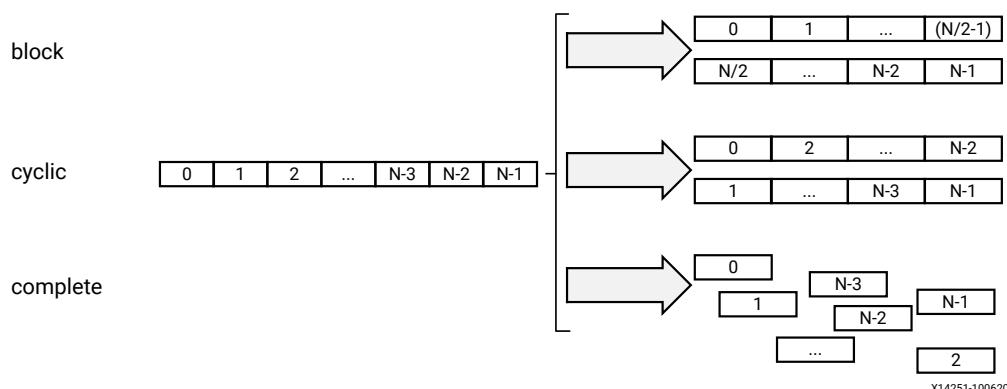
In this example, Vitis HLS states it cannot reach the specified initiation interval (II) of 1 because it cannot schedule a `load` (read) operation (`mem_load_2`) onto the memory because of limited memory ports. The above message notes that the resource limit for "core:RAM:mem:p0 is 1" which is used by the operation `mem_load` on line 62. The second port of the block RAM also only has 1 resource, which is also used by operation `mem_load_1`. Due to this memory port contention, Vitis HLS reports a final II of 2 instead of the desired 1.

This issue is typically caused by arrays. Arrays that are not interfaces to the top-level function are implemented as block RAM which has a maximum of two data ports. This can limit the throughput of a read/write (or load/store) intensive algorithm. The bandwidth can be improved by splitting the array (a single block RAM resource) into multiple smaller arrays (multiple block RAMs), effectively increasing the number of ports.

Arrays are partitioned using the `ARRAY_PARTITION` directive. Vitis HLS provides three types of array partitioning, as shown in the following figure. The three styles of partitioning are:

- **block**: The original array is split into equally sized blocks of consecutive elements of the original array.
- **cyclic**: The original array is split into equally sized blocks interleaving the elements of the original array.
- **complete**: The default operation is to split the array into its individual elements. This corresponds to resolving a memory into registers.

Figure 28: Array Partitioning



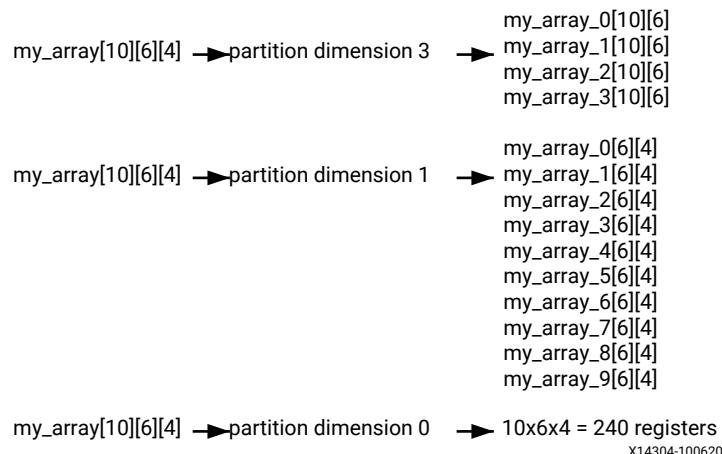
For block and cyclic partitioning the `factor` option specifies the number of arrays that are created. In the preceding figure, a factor of 2 is used, that is, the array is divided into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the final array has fewer elements.

When partitioning multi-dimensional arrays, the `dimension` option is used to specify which dimension is partitioned. The following figure shows how the `dimension` option is used to partition the following example code:

```
void foo (...) {  
    int my_array[10][6][4];  
    ...  
}
```

The examples in the figure demonstrate how partitioning `dimension 3` results in 4 separate arrays and partitioning `dimension 1` results in 10 separate arrays. If zero is specified as the `dimension`, all dimensions are partitioned.

Figure 29: Partitioning Array Dimensions



Automatic Array Partitioning

The `config_array_partition` command determines how arrays are automatically partitioned based on the number of elements.

Array Reshaping

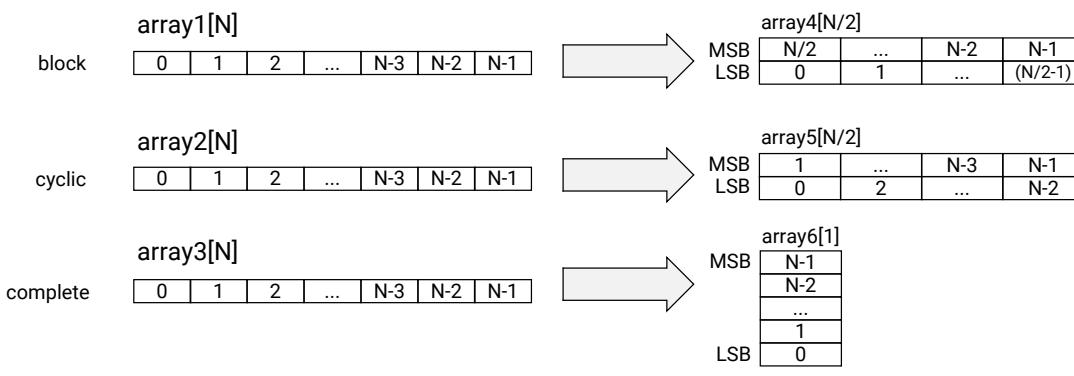
The `ARRAY_RESHAPE` directive reforms the array with a vertical mode of remapping, and is used to reduce the number of block RAM consumed while providing parallel access to the data.

Given the following example code:

```
void foo (...) {
    int array1[N];
    int array2[N];
    int array3[N];
    #pragma HLS ARRAY_RESHAPE variable=array1 type=block factor=2 dim=1
    #pragma HLS ARRAY_RESHAPE variable=array2 type=cycle factor=2 dim=1
    #pragma HLS ARRAY_RESHAPE variable=array3 type=complete dim=1
    ...
}
```

The ARRAY_RESHAPE directive transforms the arrays into the form shown in the following figure.

Figure 30: Array Reshaping



X14307-100620

The ARRAY_RESHAPE directive allows more data to be accessed in a single clock cycle. In cases where more data can be accessed in a single clock cycle, Vitis HLS might automatically unroll any loops consuming this data, if doing so will improve the throughput. The loop can be fully or partially unrolled to create enough hardware to consume the additional data in a single clock cycle. This feature is controlled using the `config_unroll` command and the option `tripcount_threshold`. In the following example, any loops with a tripcount of less than 16 will be automatically unrolled if doing so improves the throughput.

```
config_unroll -tripcount_threshold 16
```

Arrays on the Interface

When you use an array as an argument to the top-level function, Vitis HLS assumes one of the following:

- Memory is off-chip and Vitis HLS synthesizes M_AXI ports on the interface to access the memory as the default for the Vitis Kernel flow.

- Memory is standard block RAM with a latency of 1 as the default behavior in the Vivado IP flow. The data is ready one clock cycle after the address is supplied.

To configure how Vitis HLS creates these ports:

- Specify the interface as a M_AXI, BRAM, or FIFO interface using the INTERFACE pragma or directive.
- Specify the RAM as a single or dual-port RAM using the `storage_type` option of the INTERFACE pragma or directive.
- Specify the RAM latency using the `latency` option of the INTERFACE pragma or directive.
- Use array optimization directives, ARRAY_PARTITION, or ARRAY_RESHAPE, to reconfigure the structure of the array and therefore, the number of I/O ports.



TIP: Because access to the data is limited through a memory (RAM or FIFO) port, arrays on the interface can create a performance bottleneck. Typically, you can overcome these bottlenecks using optimization directives.

Arrays must be sized when used in synthesizable code. If, for example, the declaration `d_i[4]` in [Array Interfaces](#) is changed to `d_i[]`, Vitis HLS issues a message that the design cannot be synthesized:

```
@E [SYNCHK-61] array_RAM.c:52: unsupported memory access on variable 'd_i'  
which is (or contains) an array with unknown size at compile time.
```

Array Interfaces

The INTERFACE pragma or directive lets you explicitly define which type of RAM or ROM is used with the `storage_type=<value>` option. This defines which ports are created (single-port or dual-port). If no `storage_type` is specified, Vitis HLS uses:

- A single-port RAM by default.
- A dual-port RAM if it reduces the initiation interval or reduces latency.

The ARRAY_PARTITION and ARRAY_RESHAPE pragmas can re-configure arrays on the interface. Arrays can be partitioned into multiple smaller arrays, each implemented with its own interface. This includes the ability to completely partition the array into a set of scalars. On the function interface, this results in a unique port for every element in the array. This provides maximum parallel access, but creates many more ports and might introduce routing issues during hardware implementation.

By default, the array arguments in the function shown in the following code example are synthesized into a single-port RAM interface.

```
#include "array_RAM.h"

void array_RAM (dout_t d_o[4], din_t d_i[4], didx_t idx[4]) {
    int i;

    For_Loop: for (i=0;i<4;i++) {
        d_o[i] = d_i[idx[i]];
    }
}
```

A single-port RAM interface is used because the `for-loop` ensures that only one element can be read and written in each clock cycle. There is no advantage in using a dual-port RAM interface. If the `for-loop` is unrolled, Vitis HLS uses a dual-port RAM. Doing so allows multiple elements to be read at the same time and improves the initiation interval. The type of RAM interface can be explicitly set by applying the INTERFACE pragma or directive, and setting the `storage_type`.

Issues related to arrays on the interface are typically related to throughput. For example, if the arrays in the example above are partitioned into individual elements, and the `for-loop` is unrolled, all four elements in each array are accessed simultaneously.

You can also use the INTERFACE pragma or directive to specify the latency of the RAM, using the `latency=<value>` option. This lets Vitis HLS model external SRAMs with a latency greater than 1 at the interface.

FIFO Interfaces

Vitis HLS allows array arguments to be implemented as FIFO ports in the RTL. If a FIFO ports is to be used, you must ensure that the accesses to and from the array are sequential.

Note: If the accesses at the interface are not sequential, there is an RTL simulation mismatch.

The following code example shows a case in which the tool cannot determine whether the accesses are sequential. In this example, both `d_i` and `d_o` are specified to be implemented with a FIFO interface during synthesis. In this case, you must ensure the access is sequential or you will be introducing errors into your system.

```
#include "array_FIFO.h"

void array_FIFO (dout_t d_o[4], din_t d_i[4], didx_t idx[4]) {
    int i;
#pragma HLS INTERFACE mode=ap_fifo port=d_i
#pragma HLS INTERFACE mode=ap_fifo port=d_o
    For_Loop: for (i=0;i<4;i++) {
        d_o[i] = d_i[idx[i]];
    }
}
```

In this case, the values of variable `idx` would determine whether or not a FIFO interface can be successfully created for argument `d_i[]`.

- If the values of the elements of `idx` are sequential, then a FIFO interface could be created
- If random values are used for `idx`, a FIFO interface fails in Co-simulation when implemented in RTL, and may also fail during runtime

However, because these conditions cannot be validated at compile time, Vitis HLS issues a message during synthesis and creates a FIFO interface:

```
@W [XF0RM-124] Array 'd_i': may have improper streaming access(es).
```

In addition, the `idx` array is never read, because its elements are assumed to have sequential values starting from 0, due to the presence of the `ap_fifo` INTERFACE pragma.

Note: FIFO ports cannot be synthesized for arrays that are both read from and written to in the same loop or function. Separate input and output arrays (as in the example above) must be created.

The following general rules apply to arrays that are implemented with a FIFO interface:

- The array must be only read or written in the loop or function. This can be transformed into a point-to-point connection that matches the characteristics of FIFO links.
- The array reads must be in the same order as the array writes. Because random access is not supported for FIFO channels, the array must be used in the program following first in, first out semantics.

The following conditions apply when the data type of an array is a struct, and the array is sequentially accessed (i.e. the array is specified with the `axis` or `ap_fifo` interface or is marked with STREAM pragma or directive):

- You cannot access struct members directly from I/O arguments that use array-to-stream, or are as streaming interfaces. You can make a local copy of the struct in order to read/write member elements.
- You must ensure sequential order access, as shown below

```
struct A {
    short foo;
    int bar;
};

void dut(A in[N], A out[out], bool flag) {
    #pragma HLS interface ap_fifo port=in,out
    for (unsigned i=0; i<N; i++) {
        A tmp = in[i];
        if (flag)
            tmp.bar += 5;
        out[i] = tmp;
    }
}

Bad example 1:
```

```
void dut(A in[N], A out[out], bool flag) {
    #pragma HLS interface ap_fifo port=in,out
    for (unsigned i=0; i<N; i++) {
        out[i] = in[i];
        if (flag)
            out[i].bar += 5;
    }
}

Bad example 2:
void dut(A in[N], A out[out], bool flag) {
    #pragma HLS interface ap_fifo port=in,out
    for (unsigned i=0; i<N; i++) {
        out[i].foo = in[i].foo;
        if (flag)
            out[i].bar = in[i].bar + 5;
        else
            out[i].bar = in[i].bar;
    }
}
```

Memory Mapped Interfaces

As mentioned earlier, Vitis HLS allows the user to specify M_AXI interfaces for arrays in the interface. Since this memory is off-chip and not local, access to these memories can be expensive in terms of cycles. To optimize how these accesses are made, Vitis HLS performs an automatic *burst* optimization, to efficiently read/write to these external memories. Bursting is an optimization that tries to intelligently aggregate the memory accesses to the DDR to maximize the throughput bandwidth and/or minimize the latency. Bursting is one of many possible optimizations to the kernel. Bursting typically gives you a 4-5x improvement. Bursting is useful when you have contention on the DDR ports from multiple competing kernels.

For more information on this bursting optimization and details on how to write code to infer more bursts, please review the [Optimizing AXI System Performance](#) section.

Initializing and Resetting Arrays



RECOMMENDED: Although not a requirement, Xilinx recommends specifying arrays that are to be implemented as memories with the *static* qualifier. This not only ensures that Vitis HLS implements the array with a memory in the RTL; it also allows the default initialization behavior of the static types to be used.

In the following code, an array is initialized with a set of values. Each time the function is executed, array `coeff` is assigned these values. After synthesis, each time the design executes the RAM that implements `coeff` is loaded with these values. For a single-port RAM this would take eight clock cycles. For an array of 1024, it would of course take 1024 clock cycles, during which time no operations depending on `coeff` could occur.

```
int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};
```

The following code uses the `static` qualifier to define array `coeff`. The array is initialized with the specified values at start of execution. Each time the function is executed, array `coeff` remembers its values from the previous execution. A static array behaves in C/C++ code as a memory does in RTL.

```
static int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};
```

In addition, if the variable has the `static` qualifier, Vitis HLS initializes the variable in the RTL design and in the FPGA bitstream. This removes the need for multiple clock cycles to initialize the memory and ensures that initializing large memories is not an operational overhead. Refer to the [initialization_and_reset](#) example available on GitHub for examples.

The RTL configuration command `config_rtl -reset` can specify if static variables return to their initial state after a reset is applied. This is not the default. When `reset state` or `all` are used, it forces all arrays implemented as block RAM to be returned to their initialized state after reset. This can result in two very undesirable conditions in the RTL design:

- Unlike a power-up initialization (or power-on reset), an explicit reset requires the RTL design to iterate through each address in the block RAM to set the value: this can take many clock cycles if N is large, and requires more area resources to implement the reset.
- A reset is added to every array in the design.

To prevent adding reset logic onto every such block RAM, and incurring the cycle overhead to reset all elements in the RAM, specify the default `control` reset mode and use the `RESET` pragma or directive to identify individual static or global variables to be reset.

Alternatively, you can use the `state` reset mode, and use the `RESET` directive `off` option to select which individual static or global variables to not reset.

Finally, depending on the hardware device or platform of your choice (UltraScale+ or Versal, etc), there can be differences in how BRAMs and URAMs are initialized and/or reset. In general, Vitis HLS supports two types of reset: one is when the device is powered on (and also termed as power-up initialization or power-on reset), and the second is when a hardware `RESET` signal is asserted during device execution. The following shows the differences in behavior for the different memory resources:

- Initialization Behavior: Applies to all BRAMs on all platforms and only to Versal URAMs. This is the behavior during power-on initialization (or power-on reset).
- Maintaining an “initial value array” and “run time array” if the array is read/written. This applies to both BRAMs and URAMs and this corresponds to the hardware “RESET” signal during device execution.

Implementing ROMs

The `const` qualifier is recommended when arrays are only read, because Vitis HLS cannot always infer that a ROM should be used by analysis of the design. The general rule for the automatic inference of a ROM is that a local (non-global), `static` array is fully written to before being read, and never written again. The following practices in the code can help infer a ROM:

- Initialize the array as early as possible in the function that uses it.
- Group writes together.
- Do not interleave `array(ROM)` initialization writes with non-initialization code.
- Do not store different values to the same array element (group all writes together in the code).
- Element value computation must not depend on any non-constant (at compile-time) design variables, other than the initialization loop counter variable.

If complex assignments are used to initialize a ROM (for example, functions from the `math.h` library), placing the array initialization into a separate function allows a ROM to be inferred. In the following example, array `sin_table[256]` is inferred as a memory and implemented as a ROM after RTL synthesis.

```
#include "array_ROM_math_init.h"
#include <math.h>

void init_sin_table(din1_t sin_table[256])
{
    int i;
    for (i = 0; i < 256; i++) {
        dint_t real_val = sin(M_PI * (dint_t)(i - 128) / 256.0);
        sin_table[i] = (din1_t)(32768.0 * real_val);
    }
}

dout_t array_ROM_math_init(din1_t inval, din2_t idx)
{
    short sin_table[256];
    init_sin_table(sin_table);
    return (int)inval * (int)sin_table[idx];
}
```



TIP: Because the `sin()` function results in constant values, no core is required in the RTL design to implement the `sin()` function.

C Simulation with Arrays

Arrays can introduce issues during C/C++ simulation, even before the synthesis step is performed. If you specify a very large array, it might cause the C/C++ simulation to run out of memory and fail, as shown in the following example:

```
#include "ap_int.h"

int i, acc;
// Use an arbitrary precision type
ap_int<32> la0[10000000], la1[10000000];

for (i=0 ; i < 10000000; i++) {
    acc = acc + la0[i] + la1[i];
}
```

The simulation might fail by running out of memory, because the array is placed on the stack that exists in memory rather than the heap that is managed by the OS and can use local disk space to grow. Certain issues might make this issue more likely:

- On PCs, the available memory is often less than large Linux boxes and there might be less memory available.
- Using arbitrary precision types as shown in the example above could make this issue worse as they require more memory to model than standard C/C++ types.
- Using the more complex fixed-point arbitrary precision types found in C++ might make the issue of designs running out of memory even more likely as types require even more memory.

The standard way to improve memory resources in C/C++ code development is to increase the size of the stack using the linker options such as the following option which explicitly sets the stack size `-z stack-size=10485760`. This can be applied in Vitis HLS GUI by using the **Project Settings → Simulation → Linker** command, or using the following Tcl commands:

```
csim_design -ldfflags {-z stack-size=10485760}
cosim_design -ldfflags {-z stack-size=10485760}
```

In some cases, the machine may not have enough available memory, and increasing the stack size will not help. In this case a solution is to use dynamic memory allocation for simulation but a fixed-sized array for synthesis, as shown in the next example. This means that the memory required for this is allocated on the heap, managed by the OS, and can use local disk space to grow.

```
#include "ap_int.h"

int i, acc;
#ifndef __SYNTHESIS__
    // Use an arbitrary precision type & array for synthesis
    ap_int<32> la0[10000000], la1[10000000];
#else
    // Use an arbitrary precision type & dynamic memory for simulation
```

```
ap_int<int32> *la0 = malloc(10000000 * sizeof(ap_int<32>));  
ap_int<int32> *la1 = malloc(10000000 * sizeof(ap_int<32>));  
#endif  
for (i=0 ; i < 10000000; i++) {  
    acc = acc + la0[i] + la1[i];  
}
```

However, this is not an ideal solution because the simulated code and the synthesized code are not the same. But this might be the only way to complete simulation. If you take this approach be sure that the C/C++ test bench covers all aspects of accessing the array. The RTL simulation performed by `cosim_design` will verify that the memory accesses are correct in the synthesized code.

Note: Only use the `__SYNTHESIS__` macro on the code to be synthesized. Do not use this macro in the test bench, because it has no significance in the C/C++ simulation or C/C++ RTL co-simulation. Refer to [Vitis-HLS-Introductory-Examples/Pipelining/Functions/hier_func](#) for the full version of this example.

Functions Primer

The top-level function becomes the top-level module of the RTL design after synthesis. All sub-functions that are not in-lined are synthesized into separate modules in the RTL design. Arguments of the top-level function are implemented as interface ports in the hardware as described in [Interfaces of the HLS Design](#). Global variables used by the kernel cannot be accessed from the outside. Any variable that is accessed by both the test bench (or other compiled kernels or host) and the kernel itself should be defined as an argument of the top-level function.



IMPORTANT! *The top-level function cannot be a static function.*

After synthesis, each function in the design has its own synthesis report and HDL file (Verilog and VHDL).

Function Inlining

Function inlining removes the function hierarchy. A function is inlined using the `INLINE` directive. Inlining a function may improve the area by allowing the components within the function to be a better HLS STREAM or optimized with the logic in the calling function. This type of function inlining is also performed automatically by Vitis HLS for small functions.

Inlining allows function sharing to be better controlled. For functions to be shared they must be used within the same level of hierarchy. In this code example, function `foo_top` calls `foo` twice and function `foo_sub`.

```
foo_sub (p, q) {  
    int q1 = q + 10;  
    foo(p1,q); // foo_3  
    ...  
}  
void foo_top { a, b, c, d} {  
    ...  
    foo(a,b); //foo_1  
    foo(a,c); //foo_2  
    foo_sub(a,d);  
    ...  
}
```

Inlining function `foo_sub` and using the `ALLOCATION` directive to specify that only one instance of the function `foo` is used, results in a design that only has one instance of function `foo`: one-third the area of the example above. Using this strategy allows for more fine-grained control of what user-defined resources can be shared and is a useful feature to when area utilization is a consideration.

```
foo_sub (p, q) {  
#pragma HLS INLINE  
    int q1 = q + 10;  
    foo(p1,q); // foo_3  
    ...  
}  
void foo_top { a, b, c, d} {  
#pragma HLS ALLOCATION instances=foo limit=1 function  
    ...  
    foo(a,b); //foo_1  
    foo(a,c); //foo_2  
    foo_sub(a,d);  
    ...  
}
```

The `INLINE` directive optionally allows all functions below the specified function to be recursively inlined by using the `recursive` option. If the `recursive` option is used on the top-level function, all function hierarchy in the design is removed.

The `INLINE off` option can optionally be applied to functions to prevent them from being inlined. This option may be used to prevent Vitis HLS from automatically inlining a function.

The `INLINE` directive is a powerful way to substantially modify the structure of the code without actually performing any modifications to the source code and provides a very powerful method for architectural exploration.

Function Pipelining

Function pipelining is handled similarly to loop pipelining as described in [Pipelining Loops](#). Vitis HLS treats the function body as if it were the same as a loop body being called multiple times - except in this case, it is the function that is called multiple times and the tools pipelines the execution of these calls. So similar to loops, when a function is pipelined, all the loops in the function body and in the hierarchy below are automatically unrolled. This is a requirement for pipelining to proceed. If a loop has variable bounds and it cannot be unrolled then this will prevent the function from being pipelined.

Function Instantiation

Function instantiation is an optimization technique that has the area benefits of maintaining the function hierarchy but provides an additional powerful option: performing targeted local optimizations on specific instances of a function. This can simplify the control logic around the function call and potentially improve latency and throughput.

The `FUNCTION_INSTANTIATE` pragma or directive exploits the fact that some inputs to a function may be a constant value when the function is called and uses this to both simplify the surrounding control structures and produce smaller more optimized function blocks. This is best explained by example as shown in the following code.

```
char func(char inval, char incr) {
    #pragma HLS INLINE OFF
    #pragma HLS FUNCTION_INSTANTIATE variable=incr
    return inval + incr;
}

void top(char inval1, char inval2, char inval3,
        char *outval1, char *outval2, char *outval3)
{
    *outval1 = func(inval1, 0);
    *outval2 = func(inval2, 1);
    *outval3 = func(inval3, 100);
}
```



TIP: The Vitis HLS tool automatically decomposes (or inlines) small functions into higher-level calling functions. Using the `INLINE` pragma with the `OFF` option can be used to prevent this automatic inlining.

It is clear that function `func` has been written to perform three exclusive operations (depending on the value of `incr`). Each instance of function `func` is implemented in an identical manner. While this is great for function reuse and area optimization, it also means that the control logic inside the function must be more complex to account for the two exclusive operations. Refer to [Vitis-HLS-Introductory-Examples/Pipelining/Functions/function_instantiate](#) for the full version of this example.

The `FUNCTION_INSTANTIATE` optimization allows each instance to be independently optimized, reducing the functionality and area. After `FUNCTION_INSTANTIATE` optimization, the code above can effectively be transformed to have two separate functions, each optimized for different possible values of mode, as shown:

```
void func1() {
    // code segment 1
}

void func2() {
    // code segment 2
}
```

If the function is used at different levels of hierarchy such that function sharing is difficult without extensive inlining or code modifications, function instantiation can provide the best means of improving area: many small locally optimized copies are better than many large copies that cannot be shared.

Data Types

The data types used in a C/C++ function compiled into an executable impact the accuracy of the result and the memory requirements, and can impact the performance.

- A 32-bit integer `int` data type can hold more data and therefore provide more precision than an 8-bit `char` type, but it requires more storage.
- If 64-bit `long long` types are used on a 32-bit system, the runtime is impacted because it typically requires multiple accesses to read and write those values.

Similarly, when the C/C++ function is to be synthesized to an RTL implementation, the types impact the precision, the area, and the performance of the RTL design. The data types used for variables determine the size of the operators required and therefore the area and performance of the RTL.

Vitis HLS supports the synthesis of all standard C/C++ types, including exact-width integer types.

- `(unsigned) char`, `(unsigned) short`, `(unsigned) int`
- `(unsigned) long`, `(unsigned) long long`
- `(unsigned) intN_t` (where N is 8, 16, 32, and 64, as defined in `stdint.h`)
- `float`, `double`

Exact-width integers types are useful for ensuring designs are portable across all types of system.

The C/C++ standard dictates that type `(unsigned) long` is implemented as 64 bits on 64-bit operating systems and as 32 bits on 32-bit operating systems. Synthesis matches this behavior and produces different sized operators, and therefore different RTL designs, depending on the type of operating system on which Vitis HLS is run. On Windows OS, Microsoft defines type `long` as 32-bit, regardless of the OS.

- Use data type `(unsigned)int` or `(unsigned)int32_t` instead of type `(unsigned)long` for 32-bit.
- Use data type `(unsigned)long long` or `(unsigned)int64_t` instead of type `(unsigned)long` for 64-bit.

Note: The C/C++ compile option `-m32` may be used to specify that the code is compiled for C/C++ simulation and synthesized to the specification of a 32-bit architecture. This ensures the long data type is implemented as a 32-bit value. This option is applied using the `-CFLAGS` option to the `add_files` command.

Xilinx highly recommends defining the data types for all variables in a common header file, which can be included in all source files.

- During the course of a typical Vitis HLS project, some of the data types might be refined, for example to reduce their size and allow a more efficient hardware implementation.
- One of the benefits of working at a higher level of abstraction is the ability to quickly create new design implementations. The same files typically are used in later projects but might use different (smaller or larger or more accurate) data types.

Both of these tasks are more easily achieved when the data types can be changed in a single location: the alternative is to edit multiple files.



IMPORTANT! When using macros in header files, always use unique names. For example, if a macro named `_TYPES_H` is defined in your header file, it is likely that such a common name might be defined in other system files, and it might enable or disable some other code causing unforeseen side effects.



TIP: The `std::complex<long double>` data type is not supported in Vitis HLS and should not be used.

Standard Types

The following code example shows some basic arithmetic operations being performed.

```
#include "types_standard.h"

void types_standard(din_A inA, din_B inB, din_C inC, din_D inD,
    dout_1 *out1, dout_2 *out2, dout_3 *out3, dout_4 *out4)
{
    // Basic arithmetic operations
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;
}
```

The data types in the example above are defined in the header file `types_standard.h` shown in the following code example. They show how the following types can be used:

- Standard signed types
- Unsigned types

- Exact-width integer types (with the inclusion of header file `stdint.h`)

```
#include <stdio.h>
#include <stdint.h>

#define N 9

typedef char din_A;
typedef short din_B;
typedef int din_C;
typedef long long din_D;

typedef int dout_1;
typedef unsigned char dout_2;
typedef int32_t dout_3;
typedef int64_t dout_4;

void types_standard(din_A inA,din_B inB,din_C inC,din_D inD,dout_1
*out1,dout_2 *out2,dout_3 *out3,dout_4 *out4);
```

These different types result in the following operator and port sizes after synthesis:

- The multiplier used to calculate result `out1` is a 24-bit multiplier. An 8-bit `char` type multiplied by a 16-bit `short` type requires a 24-bit multiplier. The result is sign-extended to 32-bit to match the output port width.
- The adder used for `out2` is 8-bit. Because the output is an 8-bit `unsigned char` type, only the bottom 8-bits of `inB` (a 16-bit `short`) are added to 8-bit `char` type `inA`.
- For output `out3` (32-bit exact width type), 8-bit `char` type `inA` is sign-extended to 32-bit value and a 32-bit division operation is performed with the 32-bit (`int` type) `inC` input.
- A 64-bit modulus operation is performed using the 64-bit `long long` type `inD` and 8-bit `char` type `inA` sign-extended to 64-bit, to create a 64-bit output result `out4`.

As the result of `out1` indicates, Vitis HLS uses the smallest operator it can and extends the result to match the required output bit-width. For result `out2`, even though one of the inputs is 16-bit, an 8-bit adder can be used because only an 8-bit output is required. As the results for `out3` and `out4` show, if all bits are required, a full sized operator is synthesized.

Floats and Doubles

Vitis HLS supports `float` and `double` types for synthesis. Both data types are synthesized with IEEE-754 standard partial compliance (see *Floating-Point Operator LogiCORE IP Product Guide (PG060)*).

- Single-precision 32-bit
 - 24-bit fraction
 - 8-bit exponent

- Double-precision 64-bit
 - 53-bit fraction
 - 11-bit exponent



RECOMMENDED: When using floating-point data types, Xilinx highly recommends that you review Floating-Point Design with Vivado HLS ([XAPP599](#)). Also refer to [Vitis-HLS-Introductory-Examples/Modeling/using_float_and_double](#) on Github for an example of using floating and double data types.

In addition to using floats and doubles for standard arithmetic operations (such as +, -, *) floats and doubles are commonly used with the `math.h` (and `cmath.h` for C++). This section discusses support for standard operators.

The following code example shows the header file used with [Standard Types](#) updated to define the data types to be `double` and `float` types.

```
#include <stdio.h>
#include <stdint.h>
#include <math.h>

#define N 9

typedef double din_A;
typedef double din_B;
typedef double din_C;
typedef float din_D;

typedef double dout_1;
typedef double dout_2;
typedef double dout_3;
typedef float dout_4;

void types_float_double(din_A inA,din_B inB,din_C inC,din_D inD,dout_1
*out1,dout_2 *out2,dout_3 *out3,dout_4 *out4);
```

This updated header file is used with the following code example where a `sqrtf()` function is used.

```
#include "types_float_double.h"

void types_float_double(
    din_A inA,
    din_B inB,
    din_C inC,
    din_D inD,
    dout_1 *out1,
    dout_2 *out2,
    dout_3 *out3,
    dout_4 *out4
) {

    // Basic arithmetic & math.h sqrtf()
    *out1 = inA * inB;
```

```
*out2 = inB + inA;  
*out3 = inC / inA;  
*out4 = sqrtf(inD);  
}
```

When the example above is synthesized, it results in 64-bit double-precision multiplier, adder, and divider operators. These operators are implemented by the appropriate floating-point Xilinx IP catalog cores.

The square-root function used `sqrtf()` is implemented using a 32-bit single-precision floating-point core.

If the double-precision square-root function `sqrt()` was used, it would result in additional logic to cast to and from the 32-bit single-precision float types used for `inD` and `out4`: `sqrt()` is a double-precision (`double`) function, while `sqrtf()` is a single precision (`float`) function.

In C functions, be careful when mixing float and double types as float-to-double and double-to-float conversion units are inferred in the hardware.

```
float foo_f      = 3.1459;  
float var_f = sqrt(foo_f);
```

The above code results in the following hardware:

```
wire(foo_t)  
-> Float-to-Double Converter unit  
-> Double-Precision Square Root unit  
-> Double-to-Float Converter unit  
-> wire (var_f)
```

Using a `sqrtf()` function:

- Removes the need for the type converters in hardware
- Saves area
- Improves timing

When synthesizing float and double types, Vitis HLS maintains the order of operations performed in the C code to ensure that the results are the same as the C simulation. Due to saturation and truncation, the following are not guaranteed to be the same in single and double precision operations:

```
A=B*C; A=B*F;  
D=E*F; D=E*C;  
O1=A*D O2=A*D;
```

With `float` and `double` types, `O1` and `O2` are not guaranteed to be the same.



TIP: In some cases (design dependent), optimizations such as unrolling or partial unrolling of loops, might not be able to take full advantage of parallel computations as Vitis HLS maintains the strict order of the operations when synthesizing float and double types. This restriction can be overridden using `config_compile -unsafe_math_optimizations`.

For C++ designs, Vitis HLS provides a bit-approximate implementation of the most commonly used math functions.

Floating-Point Accumulator and MAC

Floating point accumulators (`facc`), multiply and accumulate (`fmacc`), and multiply and add (`fmadd`) can be enabled using the `config_op` command shown below:

```
config_op <facc|fmacc|fmadd> -impl <none|auto> -precision <low|standard|high>
```

Vitis HLS supports different levels of precision for these operators that tradeoff between performance, area, and precision on both Versal and non-Versal devices.

- Low-precision accumulation is suitable for high-throughput low-precision floating point accumulation and multiply-accumulation, this mode is only available in non-Versal devices.
 - It uses an integer accumulator with a pre-scaler and a post-scaler (to convert input and output to single-precision or double-precision floating point).
 - It uses a 60 bit and 100 bit accumulator for single and double precision inputs respectively.
 - It can cause cosim mismatches due to insufficient precision with respect to C++ simulation
 - It can always be pipelined with an `II=1` without source code changes
 - It uses approximately 3X the resources of standard-precision floating point accumulation, which achieves an `II` that is typically between 3 and 5, depending on clock frequency and target device.

Using low-precision, accumulation for floats and doubles is supported with an initiation interval (`II`) of 1 on all devices. This means that the following code can be pipelined with an `II` of 1 without any additional coding:

```
float foo(float A[10], float B[10]) {
    float sum = 0.0;
    for (int i = 0; i < 10; i++) {
        sum += A[i] * B[i];
    }
    return sum;
}
```

- Standard-precision accumulation and multiply-add is suitable for most uses of floating-point, and is available on Versal and non-Versal devices.
 - It always uses a true floating-point accumulator

- It can be pipelined with an $II=1$ on Versal devices.
- It can be pipelined with an II that is typically between 3 and 5 (depending on clock frequency and target device) on non-Versal devices. The standard precision mode is more efficient on Versal devices than on non-Versal devices.
- High-precision fused multiply-add is suitable for high-precision applications and is available on Versal devices.
 - It uses one extra bit of precision
 - It always uses a single fused multiply-add, with a single rounding at the end, although it uses more resources than the unfused multiply-add
 - It can cause cosim mismatches due to the extra precision with respect to C++ simulation

Composite Data Types

HLS supports composite data types for synthesis:

- [Structs](#)
- [Enumerated Types](#)
- [Unions](#)

Structs

Structs in the code, for instance internal and global variables, are disaggregated by default. They are decomposed into separate objects for each of their member elements. The number and type of elements created are determined by the contents of the struct itself. Arrays of structs are implemented as multiple arrays, with a separate array for each member of the struct.



IMPORTANT! Structs used as arguments to the top-level function are aggregated by default as described in [Structs in the Interface](#).

Alternatively, you can use the [AGGREGATE](#) pragma or directive to collect all the elements of a struct into a single wide vector. This allows all members of the struct to be read and written to simultaneously. The aggregated struct will be padded as needed to align the elements on a 4-byte boundary, as discussed in [Struct Padding and Alignment](#). The member elements of the struct are placed into the vector in the order they appear in the C/C++ code: the first element of the struct is aligned on the LSB of the vector and the final element of the struct is aligned with the MSB of the vector. Any arrays in the struct are partitioned into individual array elements and placed in the vector from lowest to highest, in order.



TIP: You should take care when using the AGGREGATE pragma on structs with large arrays. If an array has 4096 elements of type `int`, this will result in a vector (and port) of width $4096 * 32 = 131072$ bits. While Vitis HLS can create this RTL design, it is unlikely that the Vivado tool will be able to route this during implementation.

The single wide-vector created by using the AGGREGATE directive allows more data to be accessed in a single clock cycle. When data can be accessed in a single clock cycle, Vitis HLS automatically unrolls any loops consuming this data, if doing so improves the throughput. The loop can be fully or partially unrolled to create enough hardware to consume the additional data in a single clock cycle. This feature is controlled using the `config_unroll` command and the option `tripcount_threshold`. In the following example, any loops with a tripcount of less than 16 will be automatically unrolled if doing so improves the throughput.

```
config_unroll -tripcount_threshold 16
```

If a struct contains arrays, the AGGREGATE directive performs a similar operation as `ARRAY_RESHAPE` and combines the reshaped array with the other elements in the struct. However, a struct cannot be optimized with AGGREGATE and then partitioned or reshaped. The AGGREGATE, `ARRAY_PARTITION`, and `ARRAY_RESHAPE` directives are mutually exclusive.

Structs in the Interface

Structs in the interface are kept aggregated by Vitis HLS by default; combining all of the elements of a struct into a single wide vector. This allows all members of the struct to be read and written-to simultaneously. You can disaggregate structs in the interface by using the [DISAGGREGATE](#) pragma or directive. When a struct contains one or more `hls::stream` objects Vitis HLS will automatically disaggregate the struct as described below in [Structs in the Interface with hls::stream Elements](#).



IMPORTANT! Disaggregating a struct in the interface is not supported in the Vitis kernel flow because the Vitis tool cannot map a single C-argument to multiple RTL ports. When disaggregating a struct in the interface, either manually or automatically, Vitis HLS will build and export the Vitis kernel output (`.xo`), but that output will result in an error when used with the `v++` command. To support the Vitis Kernel flow you must manually break the struct into its constituent elements, and define any `hls::stream` objects as using an [AXIS interface](#).

As part of aggregation, the elements of the struct are also aligned on a 4 byte alignment for the Vitis kernel flow, and on 1 byte alignment for the Vivado IP flow. This alignment might require the addition of bit padding to keep or make things aligned, as discussed in [Struct Padding and Alignment](#). By default the aggregated struct is padded rather than packed, but in the Vivado IP flow you can pack it using the `compact=bit` option of the [AGGREGATE](#) pragma or directive. However, any port that gets defined as an AXI4 interface (`m_axi`, `s_axilite`, or `axis`) cannot use `compact=bit`.

The member elements of the struct are placed into the vector in the order they appear in the C/C++ code: the first element of the struct is aligned on the LSB of the vector and the final element of the struct is aligned with the MSB of the vector. This allows more data to be accessed in a single clock cycle. Any arrays in the struct are partitioned into individual array elements and placed in the vector from lowest to highest, in order.

In the following example, `struct data_t` is defined in the header file shown. The struct has two data members:

- An unsigned vector `varA` of type `short` (16-bit).
- An array `varB` of four `unsigned char` types (8-bit).

```
typedef struct {  
    unsigned short varA;  
    unsigned char varB[4];  
} data_t;  
  
data_t struct_port(data_t i_val, data_t *i_pt, data_t *o_pt);
```

Aggregating the struct on the interface results in a single 48-bit port containing 16 bits of `varA`, and 4x8 bits of `varB`.



TIP: The maximum bit-width of any port or bus created by data packing is 8192 bits, or 4096 bits for axis streaming interfaces.

There are no limitations in the size or complexity of structs that can be synthesized by Vitis HLS. There can be as many array dimensions and as many members in a struct as required. The only limitation with the implementation of structs occurs when arrays are to be implemented as streaming (such as a FIFO interface). In this case, follow the same general rules that apply to arrays on the interface (FIFO Interfaces).

Structs on the Interface with `hls::stream` Elements

User-defined structs on the interface containing `hls::stream` elements are automatically disaggregated by Vitis HLS. This disaggregated struct is supported in the Vivado IP flow, and the exported IP will work as expected. However, this disaggregated struct is not supported for the Vitis Kernel flow, and the exported kernel (`.xo`) will cause an error when used with the `v++ --link` command. To support the Vitis Kernel flow you must manually break the struct into its constituent elements, and define the `hls::stream` object as using an AXIS interface.

If you have a struct that is disaggregated automatically, Vitis HLS applies any INTERFACE pragmas to the individual elements of the disaggregated struct. If there is only one INTERFACE pragma specified for the struct, it is applied to each element of the struct. If you provide an INTERFACE pragma for each element of the disaggregated struct, it is applied as expected.

Struct Padding and Alignment

Structs in Vitis HLS can have different types of padding and alignment depending on the use of `--attribute__` or `#pragmas`. These features are described below.

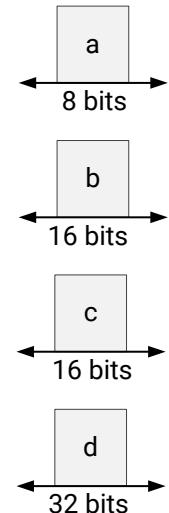
- **Disaggregate:** By default, structs in the code as internal variables are disaggregated into individual elements. The number and type of elements created are determined by the contents of the struct itself. Vitis HLS will decide whether a struct will be disaggregated or not based on certain optimization criteria.



TIP: You can use the [AGGREGATE](#) pragma or directive to prevent the default disaggregation of structs in the code.

Figure 31: Disaggregated Struct

```
struct example {  
    ap_int<5> a;  
    unsigned short int b;  
    unsigned short int c;  
    int d;  
};  
void foo()  
{  
    example s0;  
    #pragma HLS disaggregate variable=s0  
}
```



X24681-100520

- **Aggregate:** Aggregating structs on the interface is the default behavior of the tool, as discussed in [Structs in the Interface](#). Vitis HLS joins the elements of the struct, aggregating the struct into a single data unit. This is done in accordance with the [AGGREGATE](#) pragma or directive, although you do not need to specify the pragma as this is the default for structs on the interface. The aggregate process may also involve bit padding for elements of the struct, to align the byte structures on a default 4-byte alignment, or specified alignment.



TIP: The tool can issue a warning when bits are added to pad the struct, by specifying `-Wpadded` as a compiler flag.

- **Aligned:** By default, Vitis HLS will align struct on a 4-byte alignment, padding elements of the struct to align it to a 32-bit width. However, you can use the `--attribute__(aligned(X))` to add padding between elements of the struct, to align it on "X" byte boundaries.



IMPORTANT! Note that "X" can only be defined as a power of 2.

The `_attribute_((aligned))` does not change the sizes of variables it is applied to, but may change the memory layout of structures by inserting padding between elements of the struct. As a result the size of the structure will change.

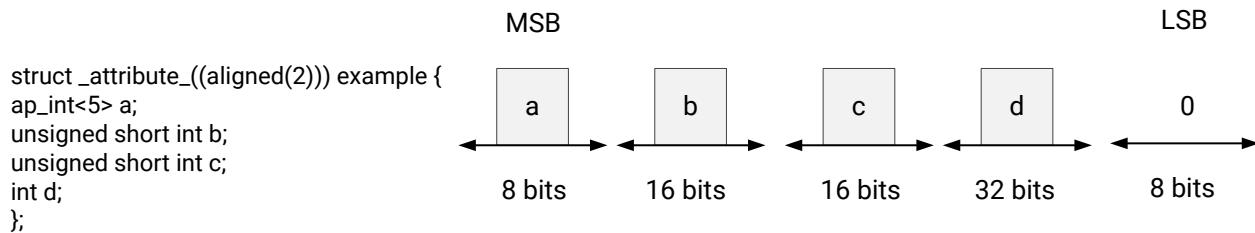
Data types in struct with custom data widths, such as `ap_int`, are allocated with sizes which are powers of 2. Vitis HLS adds padding bits for aligning the size of the data type to a power of 2.

Vitis HLS will also pad the `bool` data type to align it to 8 bits.

In the following example, the size of `varA` in the struct will be padded to 8 bits instead of 5.

```
struct example {
    ap_int<5> varA;
    unsigned short int varB;
    unsigned short int varC;
    int d;
};
```

Figure 32: Aligned Struct Implementation



X24682-102220

The padding used depends on the order and size of elements of your struct. In the following code example, the struct alignment is 4 bytes, and Vitis HLS will add 2 bytes of padding after the first element, `varA`, and another 2 bytes of padding after the third element, `varC`. The total size of the struct will be 96-bits.

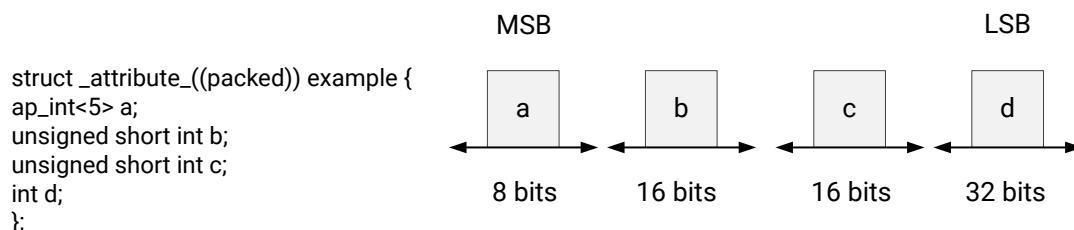
```
struct data_t {
    short varA;
    int varB;
    short varC;
};
```

However, if you rewrite the struct as follows, there will be no need for padding, and the total size of the struct will be 64-bits.

```
struct data_t {  
    short varA;  
    short varC;  
    int varB;  
};
```

- **Packed:** Specified with `_attribute_(packed(X))`, Vitis HLS packs the elements of the struct so that the size of the struct is based on the actual size of each element of the struct. In the following example, this means the size of the struct is 72 bits:

Figure 33: Packed Struct Implementation



X24680-102220



TIP: This can also be achieved using the `compact=bit` option of the [AGGREGATE](#) pragma or directive.

C++ Classes and Templates

C++ classes are fully supported for synthesis with Vitis HLS. The top-level for synthesis must be a function. A class cannot be the top-level for synthesis. To synthesize a class member function, instantiate the class itself into function. Do not simply instantiate the top-level class into the test bench. The following code example shows how class `CFir` (defined in the header file discussed next) is instantiated in the top-level function `CPP_FIR` and used to implement an `FIR` filter.

```
#include "cpp_FIR.h"

// Top-level function with class instantiated
data_t cpp_FIR(data_t x)
{
    static CFir<coef_t, data_t, acc_t> fir1;

    cout << fir1;

    return fir1(x);
}
```



IMPORTANT! Classes and class member functions cannot be the top-level for synthesis. Instantiate the class in a top-level function.

Before examining the class used to implement the design in the C++ FIR Filter example above, it is worth noting Vitis HLS ignores the standard output stream `cout` during synthesis. When synthesized, Vitis HLS issues the following warnings:

```
INFO [SYNCHK-101] Discarding unsynthesizable system call:
'std::ostream::operator<<' (cpp_FIR.h:108)
INFO [SYNCHK-101] Discarding unsynthesizable system call:
'std::ostream::operator<<' (cpp_FIR.h:108)
INFO [SYNCHK-101] Discarding unsynthesizable system call: 'std::operator<<
<std::char_traits<char> >' (cpp_FIR.h:110)
```

The following code example shows the header file `cpp_FIR.h`, including the definition of class `CFir` and its associated member functions. In this example the operator member functions () and << are overloaded operators, which are respectively used to execute the main algorithm and used with `cout` to format the data for display during C/C++ simulation.

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

#define N 85

typedef int coef_t;
typedef int data_t;
typedef int acc_t;

// Class CFir definition
template<class coef_T, class data_T, class acc_T>
class CFir {
protected:
    static const coef_T c[N];
    data_T shift_reg[N-1];
private:
public:
    data_T operator()(data_T x);
    template<class coef_TT, class data_TT, class acc_TT>
    friend ostream&
    operator<<(ostream& o, const CFir<coef_TT, data_TT, acc_TT> &f);
};

// Load FIR coefficients
template<class coef_T, class data_T, class acc_T>
const coef_T CFir<coef_T, data_T, acc_T>::c[N] = {
    #include "cpp_FIR.h"
};

// FIR main algorithm
template<class coef_T, class data_T, class acc_T>
data_T CFir<coef_T, data_T, acc_T>::operator()(data_T x) {
    int i;
    acc_t acc = 0;
    data_t m;

    loop: for (i = N-1; i >= 0; i--) {
        if (i == 0) {
            m = x;
            shift_reg[0] = x;
```

```

    } else {
        m = shift_reg[i-1];
        if (i != (N-1))
            shift_reg[i] = shift_reg[i - 1];
    }
    acc += m * c[i];
}
return acc;
}

// Operator for displaying results
template<class coef_T, class data_T, class acc_T>
ostream& operator<<(ostream& o, const CFir<coef_T, data_T, acc_T> &f) {
    for (int i = 0; i < (sizeof(f.shift_reg)/sizeof(data_T)); i++) {
        o << shift_reg[ << i << ] = << f.shift_reg[i] << endl;
    }
    o << ----- << endl;
    return o;
}

data_t cpp_FIR(data_t x);

```

The test bench in the C++ FIR Filter example is shown in the following code example and demonstrates how top-level function `cpp_FIR` is called and validated. This example highlights some of the important attributes of a good test bench for Vitis HLS synthesis:

- The output results are checked against known good values.
- The test bench returns 0 if the results are confirmed to be correct.

```

#include "cpp_FIR.h"

int main() {
    ofstream result;
    data_t output;
    int retval=0;

    // Open a file to saves the results
    result.open(result.dat);

    // Apply stimuli, call the top-level function and saves the results
    for (int i = 0; i <= 250; i++)
    {
        output = cpp_FIR(i);

        result << setw(10) << i;
        result << setw(20) << output;
        result << endl;
    }
    result.close();

    // Compare the results file with the golden results
    retval = system(diff --brief -w result.dat result.golden.dat);
    if (retval != 0) {
        printf(Test failed !!!\n);
        retval=1;
    } else {
        printf(Test passed !\n);
    }
}

```

```

    }

    // Return 0 if the test
    return retval;
}

```

C++ Test Bench for `cpp_FIR`

To apply directives to objects defined in a class:

1. Open the file where the class is defined (typically a header file).
2. Apply the directive using the **Directives** tab.

As with functions, all instances of a class have the same optimizations applied to them.

Global Variables and Classes

Xilinx does not recommend using global variables in classes. They can prevent some optimizations from occurring. In the following code example, a class is used to create the component for a filter (class `polyd_cell` is used as a component that performs shift, multiply and accumulate operations).

```

typedef long long acc_t;
typedef int mult_t;
typedef char data_t;
typedef char coef_t;

#define TAPS 3
#define PHASES 4
#define DATA_SAMPLES 256
#define CELL_SAMPLES 12

// Use k on line 73 static int k;

template <typename T0, typename T1, typename T2, typename T3, int N>
class polyd_cell {
private:
public:
    T0 areg;
    T0 breg;
    T2 mreg;
    T1 preg;
    T0 shift[N];
    int k;      //line 73
    T0 shift_output;
    void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
    {
        Function_label0:

        if (col==0) {
            SHIFT:for (k = N-1; k >= 0; --k) {
                if (k > 0)
                    shift[k] = shift[k-1];
                else
                    shift[k] = data;
            }
            *dataOut = shift_output;
        }
    }
}

```

```

        shift_output = shift[N-1];
    }
    *pcout = (shift[4*col]* coeff) + pcin;
}

};

// Top-level function with class instantiated
void cpp_class_data (
    acc_t *dataOut,
    coef_t coeff1[PHASES][TAPS],
    coef_t coeff2[PHASES][TAPS],
    data_t dataIn[DATA_SAMPLES],
    int row
) {

    acc_t pcin0 = 0;
    acc_t pcout0, pcout1;
    data_t dout0, dout1;
    int col;
    static acc_t accum=0;
    static int sample_count = 0;
    static polyd_cell<data_t, acc_t, mult_t, coef_t, CELL_SAMPLES>
polyd_cell0;
    static polyd_cell<data_t, acc_t, mult_t, coef_t, CELL_SAMPLES>
polyd_cell1;

    COL:for (col = 0; col <= TAPS-1; ++col) {

        polyd_cell0.exec(&pcout0,&dout0,pcin0,coeff1[row]
[col],dataIn[sample_count],
col);

        polyd_cell1.exec(&pcout1,&dout1,pcout0,coeff2[row][col],dout0,col);

        if ((row==0) && (col==2)) {
            *dataOut = accum;
            accum = pcout1;
        } else {
            accum = pcout1 + accum;
        }
        sample_count++;
    }
}

```

Within class `polyd_cell` there is a loop `SHIFT` used to shift data. If the loop index `k` used in loop `SHIFT` was removed and replaced with the global index for `k` (shown earlier in the example, but commented `static int k`), Vitis HLS is unable to pipeline any loop or function in which class `polyd_cell` was used. Vitis HLS would issue the following message:

```
@W [XFORM-503] Cannot unroll loop 'SHIFT' in function 'polyd_cell<char,
long long,
int, char, 12>::exec' completely: variable loop bound.
```

Using local non-global variables for loop indexing ensures that Vitis HLS can perform all optimizations.

Templates

Vitis HLS supports the use of templates in C++ for synthesis. Vitis HLS does not support templates for the top-level function. Refer to [Vitis-HLS-Introductory-Examples/Modeling/using_C++_templates](#) on Github for an example of these concepts.



IMPORTANT! *The top-level function cannot be a template.*

Using Templates to Create Unique Instances

A static variable in a template function is duplicated for each different value of the template arguments.

Different C++ template values passed to a function creates unique instances of the function for each template value. Vitis HLS synthesizes these copies independently within their own context. This can be beneficial as the tool can provide specific optimizations for each unique instance, producing a straightforward implementation of the function.

```
template<int NC, int K>
void startK(int* dout) {
    static int acc=0;
    acc += K;
    *dout = acc;
}

void foo(int* dout) {
    startK<0,1> (dout);
}

void goo(int* dout) {
    startK<1,1> (dout);
}

int main() {
    int dout0,dout1;
    for (int i=0;i<10;i++) {
        foo(&dout0);
        goo(&dout1);
        cout << "dout0/1 = " << dout0 << " / " << dout1 << endl;
    }
    return 0;
}
```

Using Templates for Recursion

Templates can also be used to implement a form of recursion that is not supported in standard C synthesis (Recursive Functions).

The following code example shows a case in which a templated `struct` is used to implement a tail-recursion Fibonacci algorithm. The key to performing synthesis is that a termination class is used to implement the final call in the recursion, where a template size of one is used.

```
//Tail recursive call
template<data_t N> struct fibon_s {
    template<typename T>
    static T fibon_f(T a, T b) {
        return fibon_s<N-1>::fibon_f(b, (a+b));
    }
};

// Termination condition
template<> struct fibon_s<1> {
    template<typename T>
    static T fibon_f(T a, T b) {
        return b;
    }
};

void cpp_template(data_t a, data_t b, data_t &dout){
    dout = fibon_s<FIB_N>::fibon_f(a,b);
}
```

Enumerated Types

The header file in the following code example defines some `enum` types and uses them in a `struct`. The `struct` is used in turn in another `struct`. This allows an intuitive description of a complex type to be captured.

The following code example shows how a complex `define (MAD_NSBSAMPLES)` statement can be specified and synthesized.

```
#include <stdio.h>

enum mad_layer {
    MAD_LAYER_I    = 1,
    MAD_LAYER_II   = 2,
    MAD_LAYER_III  = 3
};

enum mad_mode {
    MAD_MODE_SINGLE_CHANNEL = 0,
    MAD_MODE_DUAL_CHANNEL   = 1,
    MAD_MODE_JOINT_STEREO   = 2,
    MAD_MODE_STEREO         = 3
};

enum mad_emphasis {
    MAD_EMPHASIS_NONE      = 0,
    MAD_EMPHASIS_50_15_US    = 1,
    MAD_EMPHASIS_CCITT_J_17  = 3
};

typedef     signed int mad_fixed_t;

typedef struct mad_header {
```

```

enum mad_layer layer;
    enum mad_mode mode;
int mode_extension;
enum mad_emphasis emphasis;

unsigned long long bitrate;
unsigned int samplerate;

unsigned short crc_check;
unsigned short crc_target;

int flags;
int private_bits;

} header_t;

typedef struct mad_frame {
header_t header;
int options;
mad_fixed_t sbsample[2][36][32];
} frame_t;

#define MAD_NSBSAMPLES(header) \
((header)->layer == MAD_LAYER_I ? 12 : \
(((header)->layer == MAD_LAYER_III && \
((header)->flags & 17)) ? 18 : 36))

void types_composite(frame_t *frame);

```

The `struct` and `enum` types defined in the previous example are used in the following example. If the `enum` is used in an argument to the top-level function, it is synthesized as a 32-bit value to comply with the standard C/C++ compilation behavior. If the enum types are internal to the design, Vitis HLS optimizes them down to the only the required number of bits.

The following code example shows how `printf` statements are ignored during synthesis.

```

#include "types_composite.h"

void types_composite(frame_t *frame)
{
if (frame->header.mode != MAD_MODE_SINGLE_CHANNEL) {
    unsigned int ns, s, sb;
    mad_fixed_t left, right;

    ns = MAD_NSBSAMPLES(&frame->header);
    printf("Samples from header %d \n", ns);

    for (s = 0; s < ns; ++s) {
        for (sb = 0; sb < 32; ++sb) {
            left = frame->sbsample[0][s][sb];
            right = frame->sbsample[1][s][sb];
            frame->sbsample[0][s][sb] = (left + right) / 2;
        }
    }
    frame->header.mode = MAD_MODE_SINGLE_CHANNEL;
}
}

```

Unions

In the following code example, a union is created with a `double` and a `struct`. Unlike C/C++ compilation, synthesis does not guarantee using the same memory (in the case of synthesis, registers) for all fields in the `union`. Vitis HLS perform the optimization that provides the most optimal hardware.

```
#include "types_union.h"

dout_t types_union(din_t N, dinfp_t F)
{
    union {
        struct {int a; int b; } intval;
        double fpval;
    } intfp;
    unsigned long long one, exp;

    // Set a floating-point value in union intfp
    intfp.fpval = F;

    // Slice out lower bits and add to shifted input
    one = intfp.intval.a;
    exp = (N & 0x7FF);

    return ((exp << 52) + one) & (0x7fffffffffffffLL);
}
```

Vitis HLS does *not* support the following:

- Unions on the top-level function interface.
- Pointer reinterpretation for synthesis. Therefore, a union cannot hold pointers to different types or to arrays of different types.
- Access to a union through another variable. Using the same union as the previous example, the following is not supported:

```
for (int i = 0; i < 6; ++i)
if (i<3)
    A[i] = intfp.intval.a + B[i];
else
    A[i] = intfp.intval.b + B[i];
```

- However, it can be explicitly re-coded as:

```
A[0] = intfp.intval.a + B[0];
A[1] = intfp.intval.a + B[1];
A[2] = intfp.intval.a + B[2];
A[3] = intfp.intval.b + B[3];
A[4] = intfp.intval.b + B[4];
A[5] = intfp.intval.b + B[5];
```

The synthesis of unions does not support casting between native C/C++ types and user-defined types.

Often with Vitis HLS designs, unions are used to convert the raw bits from one data type to another data type. Generally, this raw bit conversion is needed when using floating point values at the top-level port interface. For one example, see below:

```
typedef float T;
unsigned int value; // the "input" of the conversion
T myhalfvalue; // the "output" of the conversion
union
{
    unsigned int as_uint32;
    T as_floatingpoint;
} my_converter;
my_converter.as_uint32 = value;
myhalfvalue = my_converter. as_floatingpoint;
```

This type of code is fine for float C/C++ data types and with modification, it is also fine for double data types. Changing the `typedef` and the `int` to `short` will not work for half data types, however, because half is a class and cannot be used in a union. Instead, the following code can be used:

```
typedef half T;
short value;
T myhalfvalue = static_cast<T>(value);
```

Similarly, the conversion the other way around uses `value=static_cast<ap_uint<16>>(myhalfvalue)` or `static_cast< unsigned short >(myhalfvalue)`.

```
ap_fixed<16,4> afix = 1.5;
ap_fixed<20,6> bfix = 1.25;
half ahlf = afix.to_half();
half bhlf = bfix.to_half();
```

Another method is to use the helper class `fp_struct<half>` to make conversions using the methods `data()` or `to_int()`. Use the header file `hls/utils/x_hls_utils.h`.

Type Qualifiers

The type qualifiers can directly impact the hardware created by high-level synthesis. In general, the qualifiers influence the synthesis results in a predictable manner, as discussed below. Vitis HLS is limited only by the interpretation of the qualifier as it affects functional behavior and can perform optimizations to create a more optimal hardware design. Examples of this are shown after an overview of each qualifier.

Volatile

The `volatile` qualifier impacts how many reads or writes are performed in the RTL when pointers are accessed multiple times on function interfaces. Although the `volatile` qualifier impacts this behavior in all functions in the hierarchy, the impact of the `volatile` qualifier is primarily discussed in the section on [top-level interfaces](#).

Note: Accesses to/from volatile variables is preserved. This means:

- no burst access
- no port widening
- no dead code elimination

Tip: Arbitrary precision types do not support the volatile qualifier for arithmetic operations. Any arbitrary precision data types using the volatile qualifier must be assigned to a non-volatile data type before being used in arithmetic expression.

Statics

Static types in a function hold their value between function calls. The equivalent behavior in a hardware design is a registered variable (a flip-flop or memory). If a variable is required to be a static type for the C/C++ function to execute correctly, it will certainly be a register in the final RTL design. The value must be maintained across invocations of the function and design.

It is *not* true that `only static` types result in a register after synthesis. Vitis HLS determines which variables are required to be implemented as registers in the RTL design. For example, if a variable assignment must be held over multiple cycles, Vitis HLS creates a register to hold the value, even if the original variable in the C/C++ function was *not* a static type.

Vitis HLS obeys the initialization behavior of statics and assigns the value to zero (or any explicitly initialized value) to the register during initialization. This means that the `static` variable is initialized in the RTL code and in the FPGA bitstream. It does not mean that the variable is re-initialized each time the reset signal is.

See the RTL configuration (`config_rtl` command) to determine how static initialization values are implemented with regard to the system reset.

Const

A `const` type specifies that the value of the variable is never updated. The variable is read but never written to and therefore must be initialized. For most `const` variables, this typically means that they are reduced to constants in the RTL design. Vitis HLS performs constant propagation and removes any unnecessary hardware).

In the case of arrays, the `const` variable is implemented as a ROM in the final RTL design (in the absence of any auto-partitioning performed by Vitis HLS on small arrays). Arrays specified with the `const` qualifier are (like statics) initialized in the RTL and in the FPGA bitstream. There is no need to reset them, because they are never written to.

ROM Optimization

The following shows a code example in which Vitis HLS implements a ROM even though the array is not specified with a `static` or `const` qualifier. This demonstrates how Vitis HLS analyzes the design, and determines the most optimal implementation. The qualifiers guide the tool, but do not dictate the final RTL.

```
#include "array_ROM.h"

dout_t array_ROM(din1_t inval, din2_t idx)
{
    din1_t lookup_table[256];
    dint_t i;

    for (i = 0; i < 256; i++) {
        lookup_table[i] = 256 * (i - 128);
    }

    return (dout_t)inval * (dout_t)lookup_table[idx];
}
```

In this example, the tool is able to determine that the implementation is best served by having the variable `lookup_table` as a memory element in the final RTL.

Arbitrary Precision (AP) Data Types

C/C++-based native data types are based-on on 8-bit boundaries (8, 16, 32, 64 bits). However, RTL buses (corresponding to hardware) support arbitrary data lengths. Using the standard C/C++ data types can result in inefficient hardware implementation. For example, the basic multiplication unit in a Xilinx device is the DSP library cell. Multiplying "ints" (32-bit) would require more than one DSP cell while using arbitrary precision types could use only one cell per multiplication.

Arbitrary precision (AP) data types allow your code to use variables with smaller bit-widths, and for the C/C++ simulation to validate the functionality remains identical or acceptable. The smaller bit-widths result in hardware operators which are in turn smaller and run faster. This allows more logic to be placed in the FPGA, and for the logic to execute at higher clock frequencies.

AP data types are provided for C++ and allow you to model data types of any width from 1 to 1024-bit. You must specify the use of AP libraries by including them in your C++ source code as explained in [Arbitrary Precision Data Types Library](#).



TIP: Arbitrary precision types are only required on the function boundaries, because Vitis HLS optimizes the internal logic and removes data bits and logic that do not fanout to the output ports.

AP Example

For example, a design with a filter function for a communications protocol requires 10-bit input data and 18-bit output data to satisfy the data transmission requirements. Using standard C/C++ data types, the input data must be at least 16-bits and the output data must be at least 32-bits. In the final hardware, this creates a datapath between the input and output that is wider than necessary, uses more resources, has longer delays (for example, a 32-bit by 32-bit multiplication takes longer than an 18-bit by 18-bit multiplication), and requires more clock cycles to complete.

Using arbitrary precision data types in this design, you can specify the exact bit-sizes needed in your code prior to synthesis, simulate the updated code, and verify the results prior to synthesis. Refer to [Vitis-HLS-Introductory-Examples/Modeling](#) on Github for examples of using arbitrary precision and fixed point ap data types.

Advantages of AP Data Types



IMPORTANT! One disadvantage of AP data types is that arrays are not automatically initialized with a value of 0. You must manually initialize the array if desired.

The following code performs some basic arithmetic operations:

```
#include "types.h"

void apint_arith(dinA_t  inA, dinB_t  inB, dinC_t  inC, dinD_t  inD,
                  dout1_t *out1, dout2_t *out2, dout3_t *out3, dout4_t *out4
) {

    // Basic arithmetic operations
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;
}
```

The data types `dinA_t`, `dinB_t`, etc. are defined in the header file `types.h`. It is highly recommended to use a project wide header file such as `types.h` as this allows for the easy migration from standard C/C++ types to arbitrary precision types and helps in refining the arbitrary precision types to the optimal size.

If the data types in the above example are defined as:

```
typedef char dinA_t;
typedef short dinB_t;
typedef int dinC_t;
typedef long long dinD_t;
typedef int dout1_t;
typedef unsigned int dout2_t;
typedef int32_t dout3_t;
typedef int64_t dout4_t;
```

The design gives the following results after synthesis:

```
+ Timing (ns):
  * Summary:
  +-----+-----+-----+-----+
  | Clock | Target| Estimated| Uncertainty|
  +-----+-----+-----+-----+
  | default | 4.00 | 3.85 | 0.50 |
  +-----+-----+-----+-----+

+ Latency (clock cycles):
  * Summary:
  +-----+-----+-----+-----+
  | Latency | Interval | Pipeline|
  | min | max | min | max | Type |
  +-----+-----+-----+-----+
  | 66 | 66 | 67 | 67 | none |
  +-----+-----+-----+-----+

* Summary:
+-----+-----+-----+-----+-----+
| Name | BRAM_18K | DSP48E1 | FF | LUT |
+-----+-----+-----+-----+-----+
| Expression | - | - | 0 | 17 |
| FIFO | - | - | - | - |
| Instance | - | 1 | 17920 | 17152 |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | - |
| Register | - | - | 7 | - |
+-----+-----+-----+-----+
| Total | 0 | 1 | 17927 | 17169 |
+-----+-----+-----+-----+
| Available | 650 | 600 | 202800 | 101400 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | ~0 | 8 | 16 |
+-----+-----+-----+-----+
```

However, if the width of the data is not required to be implemented using standard C/C++ types but in some width which is smaller, but still greater than the next smallest standard C/C++ type, such as the following:

```
typedef int6 dinA_t;
typedef int12 dinB_t;
typedef int22 dinC_t;
typedef int33 dinD_t;
typedef int18 dout1_t;
typedef uint13 dout2_t;
typedef int22 dout3_t;
typedef int6 dout4_t;
```

The synthesis results show an improvement to the maximum clock frequency, the latency and a significant reduction in area of 75%.

```
+ Timing (ns):
  * Summary:
  +-----+-----+-----+-----+
  | Clock | Target| Estimated| Uncertainty|
  +-----+-----+-----+-----+
  | default | 4.00 | 3.49 | 0.50 |
  +-----+-----+-----+-----+
```

```

+ Latency (clock cycles):
* Summary:
+-----+-----+-----+-----+
| Latency | Interval | Pipeline |
| min | max | min | max | Type   |
+-----+-----+-----+-----+
| 35 | 35 | 36 | 36 | none   |
+-----+-----+-----+-----+
* Summary:
+-----+-----+-----+-----+-----+
| Name      | BRAM_18K | DSP48E | FF     | LUT     |
+-----+-----+-----+-----+-----+
| Expression | - | - | 0 | 13 |
| FIFO      | - | - | - | - |
| Instance   | - | 1 | 4764 | 4560 |
| Memory     | - | - | - | - |
| Multiplexer | - | - | - | - |
| Register   | - | - | 6 | - |
+-----+-----+-----+-----+
| Total      | 0 | 1 | 4770 | 4573 |
+-----+-----+-----+-----+
| Available   | 650 | 600 | 202800 | 101400 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | ~0 | 2 | 4 |
+-----+-----+-----+-----+

```

The large difference in latency between both design is due to the division and remainder operations which take multiple cycles to complete. Using AP data types, rather than force fitting the design into standard C/C++ data types, results in a higher quality hardware implementation: the same accuracy with better performance with fewer resources.

Overview of Arbitrary Precision Integer Data Types

Vitis HLS provides integer and fixed-point arbitrary precision data types for C++.

Table 2: Arbitrary Precision Data Types

Language	Integer Data Type	Required Header
C++	ap_[u]int<W> (1024 bits) Can be extended to 4K bits wide as described below.	#include "ap_int.h"
C++	ap_[u]fixed<W,I,Q,O,N>	#include "ap_fixed.h"

For the C++ language `ap_[u]int` data types the header file `ap_int.h` defines the arbitrary precision integer data type. To use arbitrary precision integer data types in a C++ function:

- Add header file `ap_int.h` to the source code.
- Change the bit types to `ap_int<N>` or `ap_uint<N>`, where N is a bit-size from 1 to 1024.

The following example shows how the header file is added and two variables implemented to use 9-bit integer and 10-bit unsigned integer types:

```
#include "ap_int.h"

void foo_top () {
    ap_int<9> var1;           // 9-bit
    ap_uint<10> var2;         // 10-bit unsigned
```

The default maximum width allowed for `ap_[u]int` data types is 1024 bits. This default may be overridden by defining the macro `AP_INT_MAX_W` with a positive integer value less than or equal to 4096 before inclusion of the `ap_int.h` header file.



IMPORTANT! Setting the value of `AP_INT_MAX_W` too high can cause slow software compile and runtimes.

The following is an example of overriding `AP_INT_MAX_W`:

```
#define AP_INT_MAX_W 4096 // Must be defined before next line
#include "ap_int.h"

ap_int<4096> very_wide_var;
```

Overview of Arbitrary Precision Fixed-Point Data Types

Fixed-point data types model the data as an integer and fraction bits with the format `ap_fixed<W, I, [Q, O, N]>` as explained in the table below. In the following example, the Vitis HLS `ap_fixed` type is used to define an 18-bit variable with 6 bits specified as representing the numbers above the binary point, and 12 bits implied to represent the value after the decimal point. The variable is specified as signed and the quantization mode is set to round to plus infinity. Because the overflow mode is not specified, the default wrap-around mode is used for overflow.

```
#include <ap_fixed.h>
...
ap_fixed<18, 6, AP_RND> my_type;
...
```

When performing calculations where the variables have different number of bits or different precision, the binary point is automatically aligned. For example, when performing division with fixed-point type variables of different sizes, the fraction of the quotient is no greater than that of the dividend. To preserve the fractional part of the quotient you can cast the result to the new variable width before assignment.

The behavior of the C++ simulations performed using fixed-point matches the resulting hardware. This allows you to analyze the bit-accurate, quantization, and overflow behaviors using fast C-level simulation.

Fixed-point types are a useful replacement for floating point types which require many clock cycle to complete. Unless the entire range of the floating-point type is required, the same accuracy can often be implemented with a fixed-point type resulting in the same accuracy with smaller and faster hardware.

A summary of the `ap_fixed` type identifiers is provided in the following table.

Table 3: Fixed-Point Identifier Summary

Identifier	Description																	
W	Word length in bits																	
I	The number of bits used to represent the integer value, that is, the number of integer bits to the <i>left</i> of the binary point. When this value is negative, it represents the number of <i>implicit</i> sign bits (for signed representation), or the number of <i>implicit</i> zero bits (for unsigned representation) to the <i>right</i> of the binary point. For example, <pre>ap_fixed<2, 0> a = -0.5; // a can be -0.5, ap_ufixed<1, 0> x = 0.5; // 1-bit representation. x can be 0 or 0.5 ap_ufixed<1, -1> y = 0.25; // 1-bit representation. y can be 0 or 0.25 const ap_fixed<1, -7> z = 1.0/256; // 1-bit representation for z = 2^-8</pre>																	
Q	Quantization mode: This dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result. <table border="1"> <thead> <tr> <th>ap_fixed Types</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>AP_RND</td> <td>Round to plus infinity</td> </tr> <tr> <td>AP_RND_ZERO</td> <td>Round to zero</td> </tr> <tr> <td>AP_RND_MIN_INF</td> <td>Round to minus infinity</td> </tr> <tr> <td>AP_RND_INF</td> <td>Round to infinity</td> </tr> <tr> <td>AP_RND_CONV</td> <td>Convergent rounding</td> </tr> <tr> <td>AP_TRN</td> <td>Truncation to minus infinity (default)</td> </tr> <tr> <td>AP_TRN_ZERO</td> <td>Truncation to zero</td> </tr> </tbody> </table>		ap_fixed Types	Description	AP_RND	Round to plus infinity	AP_RND_ZERO	Round to zero	AP_RND_MIN_INF	Round to minus infinity	AP_RND_INF	Round to infinity	AP_RND_CONV	Convergent rounding	AP_TRN	Truncation to minus infinity (default)	AP_TRN_ZERO	Truncation to zero
ap_fixed Types	Description																	
AP_RND	Round to plus infinity																	
AP_RND_ZERO	Round to zero																	
AP_RND_MIN_INF	Round to minus infinity																	
AP_RND_INF	Round to infinity																	
AP_RND_CONV	Convergent rounding																	
AP_TRN	Truncation to minus infinity (default)																	
AP_TRN_ZERO	Truncation to zero																	
O	Overflow mode: This dictates the behavior when the result of an operation exceeds the maximum (or minimum in the case of negative numbers) possible value that can be stored in the variable used to store the result. <table border="1"> <thead> <tr> <th>ap_fixed Types</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>AP_SAT¹</td> <td>Saturation</td> </tr> <tr> <td>AP_SAT_ZERO¹</td> <td>Saturation to zero</td> </tr> <tr> <td>AP_SAT_SYM¹</td> <td>Symmetrical saturation</td> </tr> <tr> <td>AP_WRAP</td> <td>Wrap around (default)</td> </tr> <tr> <td>AP_WRAP_SM</td> <td>Sign magnitude wrap around</td> </tr> </tbody> </table>		ap_fixed Types	Description	AP_SAT ¹	Saturation	AP_SAT_ZERO ¹	Saturation to zero	AP_SAT_SYM ¹	Symmetrical saturation	AP_WRAP	Wrap around (default)	AP_WRAP_SM	Sign magnitude wrap around				
ap_fixed Types	Description																	
AP_SAT ¹	Saturation																	
AP_SAT_ZERO ¹	Saturation to zero																	
AP_SAT_SYM ¹	Symmetrical saturation																	
AP_WRAP	Wrap around (default)																	
AP_WRAP_SM	Sign magnitude wrap around																	
N	This defines the number of saturation bits in overflow wrap modes.																	

Notes:

- Using the AP_SAT* modes can result in higher resource usage as extra logic will be needed to perform saturation and this extra cost can be as high as 20% additional LUT usage.
- Fixed-point math functions from the `hls_math` library do not support the `ap_[u]fixed` template parameters Q,O, and N, for quantization mode, overflow mode, and the number of saturation bits, respectively. The quantization and overflow modes are only effective when an `ap_[u]fixed` variable is on the left hand of assignment or being initialized, but not during the calculation.

The default maximum width allowed for `ap_[u]fixed` data types is 1024 bits. This default may be overridden by defining the macro `AP_INT_MAX_W` with a positive integer value less than or equal to 4096 before inclusion of the `ap_int.h` header file.



IMPORTANT! ROM Synthesis can take a long time when using `ap_[u]fixed`. Changing it to `int` results in a quicker synthesis. For example:

```
static ap_fixed<32,0> a[32][depth] =
```

Can be changed to:

```
static int a[32][depth] =
```

Global Variables

Global variables can be freely used in the code and are fully synthesizable. However, global variables can not be inferred as arguments to the top-level function, but must instead be explicitly specified as arguments for ports in the RTL design.

The following code example shows the default synthesis behavior of global variables. It uses three global variables. Although this example uses arrays, Vitis HLS supports all types of global variables.

- Values are read from array `Ain`.
- Array `Aint` is used to transform and pass values from `Ain` to `Aout`.
- The outputs are written to array `Aout`.



IMPORTANT! Access to the global variables `Ain` and `Aout` must be explicitly listed in the argument list.

```
#include "top.h"

void top(const int idx, const int Ain[N], int Aout[Nhalf]) {
    int Aint[N];
    // Move elements in the input array
    ILOOP: for (int i = 0; i < N; i++) {
        int iadj = (i + idx) % N;
        Aint[i] = Ain[i] + Ain[iadj];
    } // end ILOOP
    // sum the 1st and 2nd halves
    OLOOP: for (int i = 0; i < Nhalf; i++) {
```

```
Aout[i] = (Aint[i] + Aint[Nhalf + i]);  
} // end OLOOP  
} // end top()
```

Pointers

Pointers are used extensively in C/C++ code and are supported for synthesis, but it is generally recommended to avoid the use of pointers in your code. This is especially true when using pointers in the following cases:

- When pointers are accessed (read or written) multiple times in the same function.
- When using arrays of pointers, each pointer must point to a scalar or a scalar array (not another pointer).
- Pointer casting is supported only when casting between standard C/C++ types, as shown.

Note: Pointer to pointer is not supported.

The following code example shows synthesis support for pointers that point to multiple objects.

```
#include "pointer_multi.h"  
  
dout_t pointer_multi (sel_t sel, din_t pos) {  
    static const dout_t a[8] = {1, 2, 3, 4, 5, 6, 7, 8};  
    static const dout_t b[8] = {8, 7, 6, 5, 4, 3, 2, 1};  
  
    dout_t* ptr;  
    if (sel)  
        ptr = a;  
    else  
        ptr = b;  
  
    return ptr[pos];  
}
```

Vitis HLS supports pointers to pointers for synthesis but does not support them on the top-level interface, that is, as argument to the top-level function. If you use a pointer to pointer in multiple functions, Vitis HLS inlines all functions that use the pointer to pointer. Inlining multiple functions can increase runtime.

```
#include "pointer_double.h"  
  
data_t sub(data_t ptr[10], data_t size, data_t**flagPtr)  
{  
    data_t x, i;  
  
    x = 0;  
    // Sum x if AND of local index and pointer to pointer index is true  
    for(i=0; i<size; ++i)  
        if (**flagPtr & i)
```

```

        x += *(ptr+i);
    return x;
}

data_t pointer_double(data_t pos, data_t x, data_t* flag)
{
    data_t array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    data_t* ptrFlag;
    data_t i;

    ptrFlag = flag;

    // Write x into index position pos
    if (pos >=0 & pos < 10)
        *(array+pos) = x;

    // Pass same index (as pos) as pointer to another function
    return sub(array, 10, &ptrFlag);
}

```

Arrays of pointers can also be synthesized. See the following code example in which an array of pointers is used to store the start location of the second dimension of a global array. The pointers in an array of pointers can point only to a scalar or to an array of scalars. They cannot point to other pointers.

```

#include "pointer_array.h"

data_t A[N][10];

data_t pointer_array(data_t B[N*10]) {
    data_t i,j;
    data_t sum1;

    // Array of pointers
    data_t* PtrA[N];

    // Store global array locations in temp pointer array
    for (i=0; i<N; ++i)
        PtrA[i] = &(A[i][0]);

    // Copy input array using pointers
    for(i=0; i<N; ++i)
        for(j=0; j<10; ++j)
            *(PtrA[i]+j) = B[i*10 + j];

    // Sum input array
    sum1 = 0;
    for(i=0; i<N; ++i)
        for(j=0; j<10; ++j)
            sum1 += *(PtrA[i] + j);

    return sum1;
}

```

Pointer casting is supported for synthesis if native C/C++ types are used. In the following code example, type `int` is cast to type `char`.

```
#define N 1024

typedef int data_t;
typedef char dint_t;

data_t pointer_cast_native (data_t index, data_t A[N]) {
    dint_t* ptr;
    data_t i = 0, result = 0;
    ptr = (dint_t*)(&A[index]);

    // Sum from the indexed value as a different type
    for (i = 0; i < 4*(N/10); ++i) {
        result += *ptr;
        ptr+=1;
    }
    return result;
}
```

Vitis HLS does not support pointer casting between general types. For example, if a `struct` composite type of signed values is created, the pointer cannot be cast to assign unsigned values.

```
struct {
    short first;
    short second;
} pair;

// Not supported for synthesis
*(unsigned*)(&pair) = -1U;
```

In such cases, the values must be assigned using the native types.

```
struct {
    short first;
    short second;
} pair;

// Assigned value
pair.first = -1U;
pair.second = -1U;
```

Pointers on the Interface

Pointers can be used as arguments to the top-level function. It is important to understand how pointers are implemented during synthesis, because they can sometimes cause issues in achieving the desired RTL interface and design after synthesis. Refer to [Vitis-HLS-Introductory-Examples/Modeling/Pointers](#) on Github for examples of some of the following concepts.

Basic Pointers

A function with basic pointers on the top-level interface, such as shown in the following code example, produces no issues for Vitis HLS. The pointer can be synthesized to either a simple wire interface or an interface protocol using handshakes.



TIP: To be synthesized as a FIFO interface, a pointer must be read-only or write-only.

```
#include "pointer_basic.h"

void pointer_basic (dio_t *d) {
    static dio_t acc = 0;

    acc += *d;
    *d = acc;
}
```

The pointer on the interface is read or written only once per function call. The test bench is shown in the following code example.

```
#include "pointer_basic.h"

int main () {
    dio_t d;
    int i, retval=0;
    FILE *fp;

    // Save the results to a file
    fp=fopen(result.dat,w);
    printf( Din Dout\n, i, d);

    // Create input data
    // Call the function to operate on the data
    for (i=0;i<4;i++) {
        d = i;
        pointer_basic(&d);
        fprintf(fp, %d \n, d);
        printf( %d %d\n, i, d);
    }
    fclose(fp);

    // Compare the results file with the golden results
    retval = system(diff --brief -w result.dat result.golden.dat);
    if (retval != 0) {
        printf(Test failed!!!\n);
        retval=1;
    } else {
        printf(Test passed!\n);
    }

    // Return 0 if the test
    return retval;
}
```

C and RTL simulation verify the correct operation (although not all possible cases) with this simple data set:

```
Din Dout
 0   0
 1   1
 2   3
 3   6
Test passed!
```

Pointer Arithmetic

Introducing pointer arithmetic limits the possible interfaces that can be synthesized in RTL. The following code example shows the same code, but in this instance simple pointer arithmetic is used to accumulate the data values (starting from the second value).

```
#include "pointer_arith.h"

void pointer_arith (dio_t *d) {
    static int acc = 0;
    int i;

    for (i=0;i<4;i++) {
        acc += *(d+i+1);
        *(d+i) = acc;
    }
}
```

The following code example shows the test bench that supports this example. Because the loop to perform the accumulations is now inside function `pointer_arith`, the test bench populates the address space specified by array `d[5]` with the appropriate values.

```
#include "pointer_arith.h"

int main () {
    dio_t d[5], ref[5];
    int i, retval=0;
    FILE           *fp;

    // Create input data
    for (i=0;i<5;i++) {
        d[i] = i;
        ref[i] = i;
    }

    // Call the function to operate on the data
    pointer_arith(d);

    // Save the results to a file
    fp=fopen(result.dat,w);
    printf( Din Dout\n, i, d);
    for (i=0;i<4;i++) {
        fprintf(fp, %d \n, d[i]);
        printf( %d    %d\n, ref[i], d[i]);
    }
    fclose(fp);
```

```
// Compare the results file with the golden results
retval = system(diff --brief -w result.dat result.golden.dat);
if (retval != 0) {
    printf(Test failed!!!\n);
    retval=1;
} else {
    printf(Test passed!\n);
}

// Return 0 if the test
return retval;
}
```

When simulated, this results in the following output:

```
Din Dout
 0   1
 1   3
 2   6
 3  10
Test passed!
```

The pointer arithmetic can access the pointer data out of sequence. On the other hand, wire, handshake, or FIFO interfaces can only access data in order:

- A wire interface reads data when the design is ready to consume the data or write the data when the data is ready.
- Handshake and FIFO interfaces read and write when the control signals permit the operation to proceed.

In both cases, the data must arrive (and is written) in order, starting from element zero. In the Interface with Pointer Arithmetic example, the code starts reading from index 1 (*i* starts at 0, $0+1=1$). This is the second element from array *d* [5] in the test bench.

When this is implemented in hardware, some form of data indexing is required. Vitis HLS does not support this with wire, handshake, or FIFO interfaces.

Alternatively, the code must be modified with an array on the interface instead of a pointer, as in the following example. This can be implemented in synthesis with a RAM (*ap_memory*) interface. This interface can index the data with an address and can perform out-of-order, or non-sequential, accesses.

Wire, handshake, or FIFO interfaces can be used only on streaming data. It cannot be used with pointer arithmetic (unless it indexes the data starting at zero and then proceeds sequentially).

```
#include "array_arith.h"

void array_arith (dio_t d[5]) {
    static int acc = 0;
    int i;
```

```
for ( i=0; i<4; i++ ) {  
    acc += d[i+1];  
    d[i] = acc;  
}  
}
```

Multi-Access Pointers on the Interface



IMPORTANT! Although multi-access pointers are supported on the interface, it is strongly recommended that you implement the required behavior using the `hls::stream` class instead of multi-access pointers to avoid some of the difficulties discussed below. Details on the `hls::stream` class can be found in [HLS Stream Library](#).

Designs that use pointers in the argument list of the top-level function (on the interface) need special consideration when multiple accesses are performed using pointers. Multiple accesses occur when a pointer is *read from* or *written to* multiple times in the same function.

Using pointers which are accessed multiple times can introduce unexpected behavior after synthesis. In the following "bad" example pointer `d_i` is read four times and pointer `d_o` is written to twice: the pointers perform multiple accesses.

```
#include "pointer_stream_bad.h"  
  
void pointer_stream_bad ( dout_t *d_o, din_t *d_i) {  
    din_t acc = 0;  
  
    acc += *d_i;  
    acc += *d_i;  
    *d_o = acc;  
    acc += *d_i;  
    acc += *d_i;  
    *d_o = acc;  
}
```

After synthesis this code will result in an RTL design which reads the input port once and writes to the output port once. As with any standard C/C++ compiler, Vitis HLS will optimize away the redundant pointer accesses. The test bench to verify this design is shown in the following code example:

```
#include "pointer_stream_bad.h"  
int main () {  
    din_t d_i;  
    dout_t d_o;  
    int retval=0;  
    FILE *fp;  
  
    // Open a file for the output results  
    fp=fopen(result.dat,w);  
  
    // Call the function to operate on the data  
    for (d_i=0;d_i<4;d_i++) {  
        pointer_stream_bad(&d_o,&d_i);  
        fprintf(fp, %d %d\n, d_i, d_o);  
    }  
    fclose(fp);
```

```
// Compare the results file with the golden results
retval = system(diff --brief -w result.dat result.golden.dat);
if (retval != 0) {
    printf(Test failed !!!\n);
    retval=1;
} else {
    printf(Test passed !\n);
}

// Return 0 if the test
return retval;
```

To implement the code as written, with the “anticipated” 4 reads on `d_i` and 2 writes to the `d_o`, the pointers must be specified as `volatile` as shown in the "pointer_stream_better" example.

```
#include "pointer_stream_better.h"

void pointer_stream_better ( volatile dout_t *d_o,  volatile din_t *d_i) {
din_t acc = 0;

acc += *d_i;
acc += *d_i;
*d_o = acc;
acc += *d_i;
acc += *d_i;
*d_o = acc;
}
```

To support multi-access pointers on the interface you should take the following steps:

- Validate the C/C++ before synthesis to confirm the intent and that the C/C++ model is correct.
- The pointer argument must have the number of accesses on the port interface specified when verifying the RTL using co-simulation within Vitis HLS.

Understanding Volatile Data

The code in [Multi-Access Pointers on the Interface](#) is written with *intent* that input pointer `d_i` and output pointer `d_o` are implemented in RTL as FIFO (or handshake) interfaces to ensure that:

- Upstream producer modules supply new data each time a read is performed on RTL port `d_i`.
- Downstream consumer modules accept new data each time there is a write to RTL port `d_o`.

When this code is compiled by standard C/C++ compilers, the multiple accesses to each pointer is reduced to a single access. As far as the compiler is concerned, there is no indication that the data on `d_i` changes during the execution of the function and only the final write to `d_o` is relevant. The other writes are overwritten by the time the function completes.

Vitis HLS matches the behavior of the `gcc` compiler and optimizes these reads and writes into a single read operation and a single write operation. When the RTL is examined, there is only a single read and write operation on each port.

The fundamental issue with this design is that the test bench and design do not adequately model how you expect the RTL ports to be implemented:

- You expect RTL ports that read and write multiple times during a transaction (and can stream the data in and out).
- The test bench supplies only a single input value and returns only a single output value. A C/C++ simulation of [Multi-Access Pointers on the Interface](#) shows the following results, which demonstrates that each input is being accumulated four times. The same value is being read once and accumulated each time. It is not four separate reads.

```
Din Dout
0    0
1    4
2    8
3   12
```

To make this design read and write to the RTL ports multiple times, use a `volatile` qualifier as shown in [Multi-Access Pointers on the Interface](#). The `volatile` qualifier tells the C/C++ compiler and Vitis HLS to make no assumptions about the pointer accesses, and to not optimize them away. That is, the data is volatile and might change.

The `volatile` qualifier:

- Prevents pointer access optimizations.
- Results in an RTL design that performs the expected four reads on input port `d_i` and two writes to output port `d_o`.

Even if the `volatile` keyword is used, the coding style of accessing a pointer multiple times still has an issue in that the function and test bench do not adequately model multiple distinct reads and writes. In this case, four reads are performed, but the same data is read four times. There are two separate writes, each with the correct data, but the test bench captures data only for the final write.



TIP: In order to see the intermediate accesses, use `cosim_design -trace_level` to create a trace file during RTL simulation and view the trace file in the appropriate viewer.

The Multi-Access volatile pointer interface can be implemented with wire interfaces. If a FIFO interface is specified, Vitis HLS creates an RTL test bench to stream new data on each read. Because no new data is available from the test bench, the RTL fails to verify. The test bench does not correctly model the reads and writes.

Modeling Streaming Data Interfaces

Unlike software, the concurrent nature of hardware systems allows them to take advantage of streaming data. Data is continuously supplied to the design and the design continuously outputs data. An RTL design can accept new data before the design has finished processing the existing data.

As [Understanding Volatile Data](#) shows, modeling streaming data in software is non-trivial, especially when writing software to model an existing hardware implementation (where the concurrent/streaming nature already exists and needs to be modeled).

There are several possible approaches:

- Add the `volatile` qualifier as shown in the Multi-Access Volatile Pointer Interface example. The test bench does not model unique reads and writes, and RTL simulation using the original C/C++ test bench might fail, but viewing the trace file waveforms shows that the correct reads and writes are being performed.
- Modify the code to model explicit unique reads and writes. See the following example.
- Modify the code to using a streaming data type. A streaming data type allows hardware using streaming data to be accurately modeled.

The following code example has been updated to ensure that it reads four unique values from the test bench and write two unique values. Because the pointer accesses are sequential and start at location zero, a streaming interface type can be used during synthesis.

```
#include "pointer_stream_good.h"

void pointer_stream_good ( volatile dout_t *d_o,  volatile din_t *d_i) {
    din_t acc = 0;

    acc += *d_i;
    acc += *(d_i+1);
    *d_o = acc;
    acc += *(d_i+2);
    acc += *(d_i+3);
    *(d_o+1) = acc;
}
```

The test bench is updated to model the fact that the function reads four unique values in each transaction. This new test bench models only a single transaction. To model multiple transactions, the input data set must be increased and the function called multiple times.

```
#include "pointer_stream_good.h"

int main () {
    din_t d_i[4];
    dout_t d_o[4];
    int i, retval=0;
    FILE          *fp;

    // Create input data
    for (i=0;i<4;i++) {
        d_i[i] = i;
    }

    // Call the function to operate on the data
    pointer_stream_good(d_o,d_i);

    // Save the results to a file
    fp=fopen(result.dat,w);
    for (i=0;i<4;i++) {
        if (i<2)
```

```
fprintf(fp, %d %d\n, d_i[i], d_o[i]);
    else
fprintf(fp, %d \n, d_i[i]);
}
fclose(fp);

// Compare the results file with the golden results
retval = system(diff --brief -w result.dat result.golden.dat);
if (retval != 0) {
    printf(Test failed !!!\n);
    retval=1;
} else {
    printf(Test passed !\n);
}

// Return 0 if the test
return retval;
}
```

The test bench validates the algorithm with the following results, showing that:

- There are two outputs from a single transaction.
- The outputs are an accumulation of the first two input reads, plus an accumulation of the next two input reads and the previous accumulation.

Din	Dout
0	1
1	6
2	
3	

- The final issue to be aware of when pointers are accessed multiple time at the function interface is RTL simulation modeling.

Multi-Access Pointers and RTL Simulation

When pointers on the interface are accessed multiple times, to read or write, Vitis HLS cannot determine from the function interface how many reads or writes are performed. Neither of the arguments in the function interface informs Vitis HLS how many values are read or written.

```
void pointer_stream_good (volatile dout_t *d_o, volatile din_t *d_i)
```

Unless the code informs Vitis HLS how many values are required (for example, the maximum size of an array), the tool assumes a single value and models C/RTL co-simulation for only a single input and a single output. If the RTL ports are actually reading or writing multiple values, the RTL co-simulation stalls. RTL co-simulation models the external producer and consumer blocks that are connected to the RTL design through the port interface. If it requires more than a single value, the RTL design stalls when trying to read or write more than one value because there is currently no value to read, or no space to write.

When multi-access pointers are used at the interface, Vitis HLS must be informed of the required number of reads or writes on the interface. Manually specify the INTERFACE pragma or directive for the pointer interface, and set the `depth` option to the required depth.

For example, argument `d_i` in the code sample above requires a FIFO depth of four. This ensures RTL co-simulation provides enough values to correctly verify the RTL.

Vector Data Types

The vector data type is provided to easily model and synthesize single instruction multiple data (SIMD) type operations. Many operators are overloaded to provide SIMD behavior for vector types. The Vitis™ HLS library provides the reference implementation for the `hls::vector<T, N>` type which represent a single-instruction multiple-data (SIMD) vector, as defined below.

- `T`: The type of elements that the vector holds, can be a user-defined type which must provide common arithmetic operations.
- `N`: The number of elements that the vector holds, must be a positive integer.
- The best performance is achieved when both the bit-width of `T` and `N` are integer powers of 2.

Vitis HLS provides a template type `hls::vector` that can be used to define SIMD operands. All the operation performed using this type are mapped to hardware during synthesis that will execute these operations in parallel. These operations can be carried out in a loop which can be pipelined with `II=1`. The following example shows how an eight element vector of integers is defined and used:

```
typedef hls::vector<int, 8> t_int8Vec;
t_int8Vec intVectorA, intVectorB;
.

.

void processVecStream(hls::stream<t_int8Vec>
&inVecStream1,hls::stream<t_int8Vec> &inVecStream2, hls::stream<int8Vec>
&outVecStream)
{
    for(int i=0;i<32;i++)
    {
        #pragma HLS pipeline II=1
        t_int8Vec aVec = inVecStream1.read();
        t_int8Vec bVec = inVecStream2.read();
        //performs a vector operation on 8 integers in parallel
        t_int8Vec cVec = aVec * bVec;
        outVecStream.write(cVec);
    }
}
```

Refer to [HLS Vector Library](#) for additional information. Refer to [Vitis-HLS-Introductory-Examples/Modeling/using_vectors](#) on Github for an example.

Bit-Width Propagation

The primary impact of a coding style on functions is on the function arguments and interface. If the arguments to a function are sized accurately, Vitis HLS can propagate this information through the design. There is no need to create arbitrary precision types for every variable. In the following example, two integers are multiplied, but only the lower 24 bits are used for the result.

```
#include "ap_int.h"

ap_int<24> foo(int x, int y) {
    int tmp;

    tmp = (x * y);
    return tmp
}
```

When this code is synthesized, the result is a 32-bit multiplier with the output truncated to 24-bit.

If the inputs are correctly sized to 12-bit types (int12) as shown in the following code example, the final RTL uses a 24-bit multiplier.

```
#include "ap_int.h"
typedef ap_int<12> din_t;
typedef ap_int<24> dout_t;

dout_t func_sized(din_t x, din_t y) {
    int tmp;

    tmp = (x * y);
    return tmp
}
```

Using arbitrary precision types for the two function inputs is enough to ensure Vitis HLS creates a design using a 24-bit multiplier. The 12-bit types are propagated through the design. Xilinx recommends that you correctly size the arguments of all functions in the hierarchy so that there is no need to size local variables.

In general, when variables are driven directly from the function interface, especially from the top-level function interface, variables can prevent some optimizations from taking place. A typical case of this is when an input is used as the upper limit for a loop index.

Unsupported C/C++ Constructs

While Vitis HLS supports a wide range of the C/C++ languages, some constructs are not synthesizable, or can result in errors further down the design flow. This section discusses areas in which coding changes must be made for the function to be synthesized and implemented in a device.

To be synthesized:

- The function and its calls must contain the entire functionality of the design.
- None of the functionality can be performed by system calls to the operating system.
- The C/C++ constructs must be of a fixed or bounded size.
- The implementation of those constructs must be unambiguous.

System Calls

System calls cannot be synthesized because they are actions that relate to performing some task upon the operating system in which the C/C++ program is running.

Vitis HLS ignores commonly-used system calls that display only data and that have no impact on the execution of the algorithm, such as `printf()` and `fprintf(stdout,)`. In general, calls to the system cannot be synthesized and should be removed from the function before synthesis. Other examples of such calls are `getc()`, `time()`, `sleep()`, all of which make calls to the operating system.

Vitis HLS defines the macro `__SYNTHESIS__` when synthesis is performed. This allows the `__SYNTHESIS__` macro to exclude non-synthesizable code from the design.

Note: Only use the `__SYNTHESIS__` macro in the code to be synthesized. Do not use this macro in the test bench, because it is not obeyed by C/C++ simulation or C/C++ RTL co-simulation.



CAUTION! You must not define or undefine the `__SYNTHESIS__` macro in code or with compiler options, otherwise compilation might fail.

In the following code example, the intermediate results from a sub-function are saved to a file on the hard drive. The macro `__SYNTHESIS__` is used to ensure the non-synthesizable files writes are ignored during synthesis.

```
#include "hier_func4.h"

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
    *outA = *in1 >> 1;
    *outB = *in2 >> 2;
}

void hier_func4(dint_t A, dint_t B, dout_t *C, dout_t *D)
{
    dint_t apb, amb;

    sumsub_func(&A,&B,&apb,&amb);
#ifndef __SYNTHESIS__
    FILE *fp1; // The following code is ignored for synthesis
    char filename[255];
    sprintf(filename,Out_apb_%03d.dat,apb);
    fp1=fopen(filename,w);
    fprintf(fp1, %d \n, apb);
    fclose(fp1);
#endif
    shift_func(&apb,&amb,C,D);
}
```

The `__SYNTHESIS__` macro is a convenient way to exclude non-synthesizable code without removing the code itself from the function. Using such a macro does mean that the code for simulation and the code for synthesis are now different.



CAUTION! If the `__SYNTHESIS__` macro is used to change the functionality of the C/C++ code, it can result in different results between C/C++ simulation and C/C++ synthesis. Errors in such code are inherently difficult to debug. Do not use the `__SYNTHESIS__` macro to change functionality.

Dynamic Memory Usage

Any system calls that manage memory allocation within the system, for example, `malloc()`, `alloc()`, and `free()`, are using resources that exist in the memory of the operating system and are created and released during runtime. To be able to synthesize a hardware implementation the design must be fully self-contained, specifying all required resources.

Memory allocation system calls must be removed from the design code before synthesis. Because dynamic memory operations are used to define the functionality of the design, they must be transformed into equivalent bounded representations. The following code example shows how a design using `malloc()` can be transformed into a synthesizable version and highlights two useful coding style techniques:

- The design does not use the `__SYNTHESIS__` macro.

The user-defined macro `NO_SYNTH` is used to select between the synthesizable and non-synthesizable versions. This ensures that the same code is simulated in C/C++ and synthesized in Vitis HLS.

- The pointers in the original design using `malloc()` do not need to be rewritten to work with fixed sized elements.

Fixed sized resources can be created and the existing pointer can simply be made to point to the fixed sized resource. This technique can prevent manual recoding of the existing design.

```
#include "malloc_removed.h"
#include <stdlib.h>
//#define NO_SYNTH

dout_t malloc_removed(din_t din[N], dsel_t width) {

#ifndef NO_SYNTH
    long long *out_accum = malloc (sizeof(long long));
    int* array_local = malloc (64 * sizeof(int));
#else
    long long _out_accum;
    long long *out_accum = &_out_accum;
    int _array_local[64];
    int* array_local = &_array_local[0];
#endif
    int i, j;

    LOOP_SHIFT:for (i=0;i<N-1; i++) {
        if (i<width)
            *(array_local+i)=din[i];
        else
            *(array_local[i])=din[i]>>2;
    }

    *out_accum=0;
    LOOP_ACCUM:for (j=0;j<N-1; j++) {
        *out_accum += *(array_local+j);
    }

    return *out_accum;
}
```

Because the coding changes here impact the functionality of the design, Xilinx does not recommend using the `_SYNTHESIS_` macro. Xilinx recommends performing the following steps:

1. Add the user-defined macro `NO_SYNTH` to the code and modify the code.
2. Enable macro `NO_SYNTH`, execute the C/C++ simulation, and save the results.
3. Disable the macro `NO_SYNTH`, and execute the C/C++ simulation to verify that the results are identical.
4. Perform synthesis with the user-defined macro disabled.

This methodology ensures that the updated code is validated with C/C++ simulation and that the identical code is then synthesized. As with restrictions on dynamic memory usage in C/C++, Vitis HLS does not support (for synthesis) C/C++ objects that are dynamically created or destroyed.

Pointer Limitations

General Pointer Casting

Vitis HLS does not support general pointer casting, but supports pointer casting between native C/C++ types.

Pointer Arrays

Vitis HLS supports pointer arrays for synthesis, provided that each pointer points to a scalar or an array of scalars. Arrays of pointers cannot point to additional pointers.

Function Pointers

Function pointers are not supported.

Note: Pointer to pointer is not supported.

Recursive Functions

Recursive functions cannot be synthesized. This applies to functions that can form multiple recursions:

```
unsigned foo (unsigned n)
{
    if (n == 0 || n == 1) return 1;
    return (foo(n-2) + foo(n-1));
}
```

Vitis HLS also does not support tail recursion, in which there is a finite number of function calls.

```
unsigned foo (unsigned m, unsigned n)
{
    if (m == 0) return n;
    if (n == 0) return m;
    return foo(n, m%n);
}
```

In C++, templates can implement tail recursion and can then be used for synthesizable tail-recursive designs.



CAUTION! Virtual Functions are not supported.

Standard Template Libraries

Many of the C++ Standard Template Libraries (STLs) contain function recursion and use dynamic memory allocation. For this reason, the STLs cannot be synthesized by Vitis HLS. The solution for STLs is to create a local function with identical functionality that does not feature recursion, dynamic memory allocation, or the dynamic creation and destruction of objects.

Note: Standard data types, such as `std::complex`, are supported for synthesis. However, the `std::complex<long double>` data type is not supported in Vitis HLS and should not be used.

Undefined Behaviors

The C/C++ undefined behaviors may lead to a different behavior in simulation and synthesis. An example of this behavior is shown below:

```
for (int i=0; i<N; i++) {
    int val; // uninitialized value
    if (i==0) val=0;
    else if (cond) val=1;
    // val may have indeterminate value here
    A[i] = val; // undefined behavior
    val++; // dead code
}
```

In the above example you should not expect that `A[i]` gets the value of `val` from the previous loop iteration if neither `i==0`, nor `(cond)` are true. You should even not expect that the increment (`val++`) will happen. The same is true for scalars values obtained after complete partition.

For such C/C++ undefined behavior situations, the behavior between GCC and Vitis HLS when compiling code is likely to be different, which will lead to a mismatch during RTL/Co-simulation. This is because in GCC, compiled for CPU, `val` is often left in the same register or in the same stack location across loop iterations, and therefore the behavior is that the value of `val` is retained between loop iterations.

The solution is either to initialize `val` at each iteration (if this is the expected behavior) or to move the declaration of `val` above the loop, as high as necessary, so that its lifetime scope matches the intent reuse. You should not expect that the compiler will infer a specific defined RTL behavior from an undefined C/C++ behavior.

Virtual Functions and Pointers

Not supported.

Interfaces of the HLS Design

After the C++ code is complete, and synthesis is converting it to an RTL design, there are elements of the hardware implementation that are also in your control. Specifically, you need to consider the inputs to and the outputs from the HLS design, the layout of memory and managing data alignment, and the execution models of the HLS design. This section discusses the following topics:

- [Defining Interfaces](#)
 - [Vitis HLS Memory Layout Model](#)
 - [Execution Modes of HLS Designs](#)
 - [Controlling Initialization and Reset Behavior](#)
-

Defining Interfaces

Introduction to Interface Synthesis

The arguments of the top-level function in a Vitis™ HLS design are synthesized into interfaces and ports that group multiple signals to define the communication protocol between the HLS design and components external to the design. Vitis HLS defines interfaces automatically, using industry standards to specify the protocol used. The type of interfaces that Vitis HLS creates depends on the data type and direction of the parameters of the top-level function, the target flow for the active solution, the default interface configuration settings as specified by `config_interface`, and any specified [INTERFACE](#) pragmas or directives.



TIP: Interfaces can be manually assigned using the [INTERFACE](#) pragma or directive. Refer to [Adding Pragmas and Directives](#) for more information.

The target flows supported by Vitis HLS as described in [Vitis HLS Flow Overview](#) include:

- The Vivado® IP flow which is the default flow for the tool
- The Vitis Kernel flow, which is the bottom-up design flow for the Vitis Application Acceleration Development flow

You can specify the target flow when creating a project solution, as described in [Creating a New Vitis HLS Project](#), or by using the following command:

```
open_solution -flow_target [vitis | vivado]
```

The interface defines three elements of the kernel:

1. The interface defines channels for data to flow into or out of the HLS design. Data can flow from a variety of sources external to the kernel or IP, such as a host application, an external camera or sensor, or from another kernel or IP implemented on the Xilinx device. The default channels for Vitis kernels are AXI adapters as described in [Interfaces for Vitis Kernel Flow](#).
2. The interface defines the port protocol that is used to control the flow of data through the data channel, defining when the data is valid and can be read or can be written, as defined in [Port-Level Protocols for Vivado IP Flow](#).



TIP: These port protocols can be customized in the Vivado IP flow, but are set and cannot be changed in the Vitis kernel flow, in most cases.

3. The interface also defines the execution control scheme for the HLS design, specifying the operation of the kernel or IP as pipelined or sequential, as defined in [Block-Level Control Protocols](#).

As described in [Best Practices for Designing with M_AXI Interfaces](#) the choice and configuration of interfaces is a key to the success of your design. However, Vitis HLS tries to simplify the process by selecting default interfaces for the target flows. For more information on the defaults used refer to [Interfaces for Vivado IP Flow](#) or [Interfaces for Vitis Kernel Flow](#) as appropriate to your design.

After synthesis completes you can review the mapping of the software arguments of your C/C++ code to hardware ports or interfaces in the *SW I/O Information* section of the [Synthesis Summary](#) report.

Interfaces for Vitis Kernel Flow

The Vitis kernel flow provides support for compiled kernel objects (.xo) for software control from a host application and by the Xilinx Run Time (XRT). As described in [Kernel Properties](#) in the *Vitis Unified Software Platform Documentation*, this flow has very specific interface requirements that Vitis HLS must meet.

Vitis HLS supports memory, stream, and register interface paradigms where each paradigm follows a certain interface protocol and uses the adapter to communicate with the external world.

- Memory Paradigm (`m_axi`): the data is accessed by the kernel through memory such as DDR, HBM, PLRAM/BRAM/URAM

- Stream Paradigm (`axis`): the data is streamed into the kernel from another streaming source, such as video processor or another kernel, and can also be streamed out of the kernel.
- Register Paradigm (`s_axilite`): The data is accessed by the kernel through register interfaces and accessed by software as register reads/writes.

The Vitis kernel flow implements the following interfaces by default:

C-argument type	Paradigm	Interface protocol (I/O/Inout)
Scalar(pass by value)	Register	AXI4-Lite (<code>s_axilite</code>)
Array	Memory	AXI4 Memory Mapped (<code>m_axi</code>)
Pointer to array	Memory	<code>m_axi</code>
Pointer to scalar	Register	<code>s_axilite</code>
Reference	Register	<code>s_axilite</code>
<code>hls::stream</code>	Stream	AXI4-Stream (<code>axis</code>)

As you can see from the table above, a pointer to an array is implemented as an `m_axi` interface for data transfer, while a pointer to a scalar is implemented using the `s_axilite` interface. A scalar value passed as a constant does not need read access, while a pointer to a scalar value needs both read/write access. The `s_axilite` interface implements an additional internal protocol depending upon the C argument type. This internal implementation can be controlled using [Port-Level Protocols for Vivado IP Flow](#). However, you should not modify the default port protocols in the Vitis kernel flow unless necessary.

Note: Vitis HLS will not automatically infer the default interfaces for the member elements of a struct/class when the elements require different interface types. For example, when one element of a struct requires a stream interface while another member element requires an `s_axilite` interface. You must explicitly define an INTERFACE pragma for each element of the struct instead of relying on the default interface assignment. If no INTERFACE pragma or directive is defined Vitis HLS will issue the following error message:

```
ERROR: [HLS 214-312] Vitis mode requires explicit INTERFACE pragmas for structs in the interface. Please add one INTERFACE pragma for each struct member field for argument 'd' of function 'dut(A&)' (example.cpp:19:0)
```

The default execution mode for Vitis kernel flow is pipelined execution, which enables overlapping execution of a kernel to improve throughput. This is specified by the `ap_ctrl_chain` block control protocol on the `s_axilite` interface.



TIP: The Vitis environment supports kernels with all of the supported block control protocols as described in [Block-Level Control Protocols](#).

The `vadd` function in the following code provides an example of interface synthesis.

```
#define VDATA_SIZE 16

typedef struct v_datatype { unsigned int data[VDATA_SIZE]; } v_dt;

extern "C" {
void vadd(const v_dt* in1, // Read-Only Vector 1
          const v_dt* in2, // Read-Only Vector 2
          v_dt* out_r, // Output Result for Addition
          const unsigned int size // Size in integer
) {

    unsigned int vSize = ((size - 1) / VDATA_SIZE) + 1;

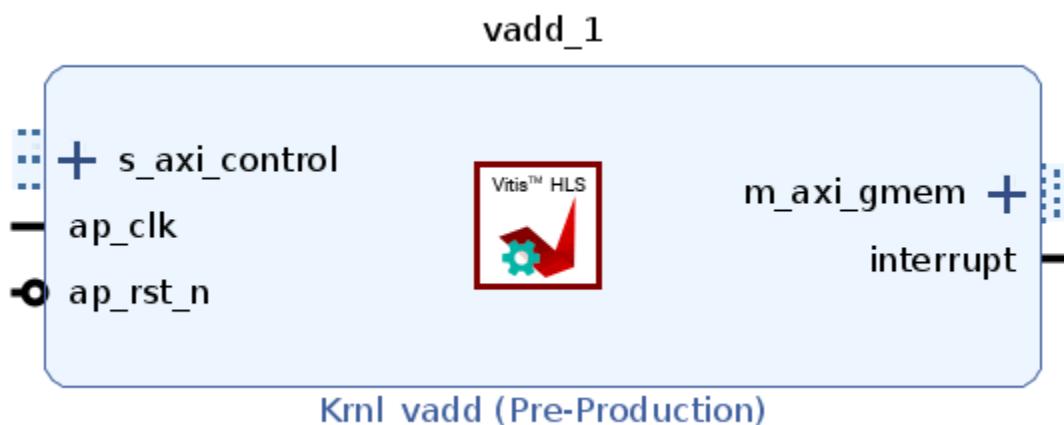
    // Auto-pipeline is going to apply pipeline to this loop
    vadd1:
    for (int i = 0; i < vSize; i++) {
        vadd2:
        for (int k = 0; k < VDATA_SIZE; k++) {
            out_r[i].data[k] = in1[i].data[k] + in2[i].data[k];
        }
    }
}
}
```

The `vadd` function includes:

- Two pointer inputs: `in1` and `in2`
- A pointer output: `out_r` that the results are written to
- A scalar value `size`

With the default interface synthesis settings used by Vitis HLS for the Vitis kernel flow, the design is synthesized into an RTL block with the ports and interfaces shown in the following figure.

Figure 34: RTL Ports After Default Interface Synthesis



The tool creates three types of interface ports on the RTL design to handle the flow of both data and control.

- Clock, Reset, and Interrupt ports: `ap_clk` and `ap_rst_n` and `interrupt` are added to the kernel.
- AXI4-Lite interface: `s_axi_control` interface which contains the scalar arguments like `size`, and manages address offsets for the `m_axi` interface, and defines the block control protocol.
- AXI4 memory mapped interface: `m_axi_gmem` interface which contains the pointer arguments: `in1`, `in2`, and `out_r`.

Details of M_AXI Interfaces for Vitis

AXI4 memory-mapped (`m_axi`) interfaces allow kernels to read and write data in global memory (DDR, HBM, PLRAM). Memory-mapped interfaces are a convenient way of sharing data across different elements of the accelerated application, such as between the host and kernel, or between kernels on the accelerator card. The main advantages for `m_axi` interfaces are listed below:

- The interface has independent read and write channels
- It supports burst-based accesses
- It provides a queue for outstanding transactions
- **Understanding Burst Access:** AXI4 memory-mapped interfaces support high throughput bursts of up to 4K bytes with just a single address phase. With burst mode transfers, Vitis HLS reads or writes data using a single base address followed by multiple sequential data samples, which makes this mode capable of higher data throughput. Burst mode of operation is possible when you use the C `memcpy` function or a pipelined `for` loop. Refer to [Controlling AXI4 Burst Behavior](#) or [AXI Burst Transfers](#) for more information.
- **Automatic Port Widening and Port Width Alignment:**

As discussed in [Automatic Port Width Resizing](#), Vitis HLS has the ability to automatically widen a port width to facilitate data transfers and improve burst access, if a burst access can be seen by the tool. Therefore all the preconditions needed for bursting, as described in [AXI Burst Transfers](#), are also needed for port resizing.

In the Vitis Kernel flow automatic port width resizing is enabled by default with the following configuration commands (notice that one command is specified as bits and the other is specified as bytes):

```
config_interface -m_axi_max_widen_bitwidth 512
config_interface -m_axi_alignment_byte_size 64
```

- **Rules for Offset:**



IMPORTANT! In the Vitis kernel flow the default mode of operation is offset=direct and default_slave_interface=s_axilite and should not be changed.

The correct specification of the offset will let the HLS kernel correctly integrate into the Vitis system. Refer to [Offset and Modes of Operation](#) for more information.

- **Bundle Interfaces - Performance vs. Resource Utilization:**

By default, Vitis HLS groups function arguments with compatible options into a single `m_axi` interface adapter as described in [M_AXI Bundles](#). Bundling ports into a single interface helps save device resources by eliminating AXI4 logic, which can be necessary when working in congested designs.

However, a single interface bundle can limit the performance of the kernel because all the memory transfers have to go through a single interface. The `m_axi` interface has independent READ and WRITE channels, so a single interface can read and write simultaneously, though only at one location. Using multiple bundles lets you increase the bandwidth and throughput of the kernel by creating multiple interfaces to connect to memory banks.

Details of S_AXILITE Interfaces for Vitis

In C++, a function starts to process data when the function is called from a parent function. The function call is pushed onto the stack when called, and removed from the stack when processing is complete to return control to the calling function. This process ensures the parent knows the status of the child.

Since the host and kernel occupy two separate compute spaces in the Vitis kernel flow, the "stack" is managed by the Xilinx Run Time (XRT), and communication is managed through the `s_axilite` interface. The kernel is software controlled through XRT by reading and writing the control registers of an `s_axilite` interface as described in [S_AXILITE Control Register Map](#). The interface provides the following features:

- **Control Protocols:** The block control protocol defines control registers in the `s_axilite` interface that let you set control signals to manage execution and operation of the kernel.
- **Scalar Arguments:** Scalar inputs on a kernel are typical, and can be thought of as programming constants or parameters. The host application transfers these values through the `s_axilite` interface.
- **Pointers to Scalar Arguments:** Vitis HLS lets you read to or write from a pointer to a scalar value when assigned to an `s_axilite` interface. Pointers are assigned by default to `m_axi` interfaces, so this requires you to manually assign the pointer to the `s_axilite` using the INTERFACE pragma or directive:

```
int top(int *a, int *b) {  
#pragma HLS interface s_axilite port=a
```

- **Rules for Offset:**

Note: The Vitis kernel flow determines the required offsets. Do not specify the `offset` option in that flow.

- **Rules for Bundle:**

The Vitis kernel flow supports only a single `s_axilite` interface, which means that all `s_axilite` interfaces must be bundled together.

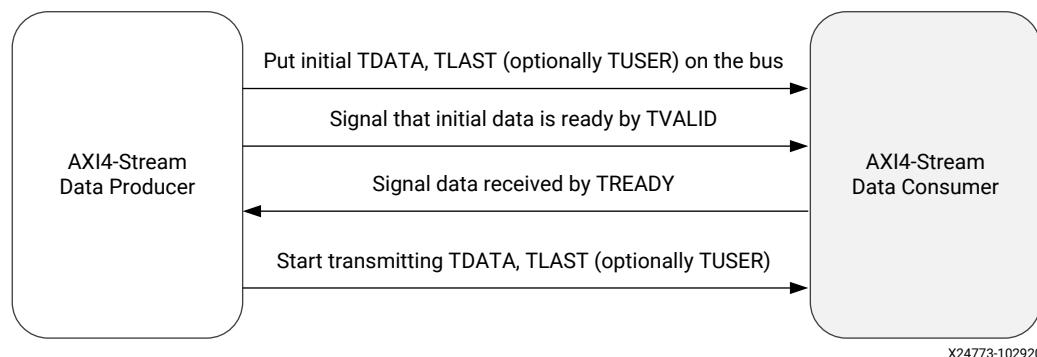
- When no bundle is specified the tool automatically creates a default bundle named `Control`.
- If for some reason you want to manually specify the bundle name, you must apply the same bundle to all `s_axilite` interfaces to create a single bundle.

Details of AXIS Interfaces for Vitis

The AXI4-Stream protocol (AXIS) defines a single uni-directional channel for streaming data in a sequential manner. The AXI4-Stream interfaces can burst an unlimited amount of data, which significantly improves performance. Unlike the AXI4 memory-mapped interface which needs an address to read/write the memory, the AXIS interface simply passes data to another AXIS interface without needing an address, and so uses fewer device resources. Combined, these features make the streaming interface a light-weight high performance interface.

The AXI4-Stream works on an industry-standard `ready/valid` handshake between a producer and consumer, as shown in the figure below. The data transfer is started once the producer sends the `TVALID` signal, and the consumer responds by sending the `TREADY` signal. This handshake of data and control should continue until either `TREADY` or `TVALID` are set low, or the producer asserts the `TLAST` signal indicating it is the last data packet of the transfer.

Figure 35: AXI4-Stream Handshake



IMPORTANT! The AXIS interface can only be assigned to the top-level arguments (ports) of a kernel or IP, and cannot be assigned to the arguments of functions internal to the design. Streaming channels used inside the HLS design should use `hls::stream` and not an AXIS interface.

You should define the streaming data type using `hls::stream<T_data_type>`, and use the `ap_axis` struct type to implement the AXIS interface. As explained in [AXI4-Stream Interfaces](#) the `ap_axis` struct lets you choose the implementation of the interface as with or without side-channels:

- [AXI4-Stream Interfaces without Side-Channels](#) implements the AXIS interface as a very light-weight interface using fewer resources
- [AXI4-Stream Interfaces with Side-Channels](#) implements a full featured interface providing greater control



TIP: You should not define your own struct for modeling the AXIS signals (side channels, TLAST, TVALID). Instead you can overload the TDATA signal for implementing your data type .

Interfaces for Vivado IP Flow

The Vivado IP flow supports a wide variety of I/O protocols and handshakes due to the requirement of supporting FPGA design for a wide variety of applications. This flow supports a traditional system design flow where multiple IP are integrated into a system. IP can be generated through Vitis HLS. In this IP flow there are two modes of control for execution of the system:

- Software Control: The system is controlled through a software application running on an embedded Arm processor or external x86 processor, using drivers to access elements of the hardware design, and reading and writing registers in the hardware to control the execution of IP in the system.
- Self Synchronous: In this mode the IP exposes signals which are used for starting and stopping the kernel. These signals are driven by other IP or other elements of the system design that handles the execution of the IP.

The Vivado IP flow supports memory, stream, and register interface paradigms where each paradigm supports different interface protocols to communicate with the external world, as shown in the following table. Note that while the Vitis kernel flow supports only the AXI4 interface adapters, this flow supports a number of different interface types.

Table 4: Interface Types

Paradigm	Description	Interface Types
Memory	Data is accessed by the kernel through memory such as DDR, HBM, PLRAM/BRAM/URAM	<code>ap_memory</code> , <code>bram</code> , AXI4 Memory Mapped (<code>m_axi</code>)
Stream	Data is streamed into the kernel from another streaming source, such as video processor or another kernel, and can also be streamed out of the kernel.	<code>ap_fifo</code> , AXI4-Stream (<code>axis</code>)
Register	Data is accessed by the kernel through register interfaces performed by register reads and writes.	<code>ap_none</code> , <code>ap_hs</code> , <code>ap_ack</code> , <code>ap_ovld</code> , <code>ap_vld</code> , and AXI4-Lite adapter (<code>s_axilite</code>).

The default interfaces are defined by the C-argument type in the top-level function, and the default paradigm, as shown in the following table.

Table 5: Default Interfaces

C-Argument Type	Supported Paradigms	Default Paradigm	Default Interface Protocol		
			Input	Output	Inout
Scalar variable (pass by value)	Register	Register	ap_none	N/A	N/A
Array	Memory, Stream	Memory	ap_memory	ap_memory	ap_memory
Pointer	Memory, Stream, Register	Register	ap_none	ap_vld	ap_ovld
Reference	Register	Register	ap_none	ap_vld	ap_vld
<code>hls::stream</code>	Stream	Stream	ap_fifo	ap_fifo	N/A

The default execution mode for Vivado IP flow is sequential execution, which requires the HLS IP to complete one iteration before starting the next. This is specified by the `ap_ctrl_hs` block control protocol. The control protocol can be changed as specified in [Block-Level Control Protocols](#).

The `vadd` function in the following code provides an example of interface synthesis in the Vivado IP flow.

```
#define VDATA_SIZE 16

typedef struct v_datatype { unsigned int data[VDATA_SIZE]; } v_dt;

extern "C" {
void vadd(const v_dt* in1, // Read-Only Vector 1
          const v_dt* in2, // Read-Only Vector 2
          v_dt* out_r, // Output Result for Addition
          const unsigned int size // Size in integer
) {

    unsigned int vSize = ((size - 1) / VDATA_SIZE) + 1;

    // Auto-pipeline is going to apply pipeline to this loop
    vadd1:
    for (int i = 0; i < vSize; i++) {
        vadd2:
        for (int k = 0; k < VDATA_SIZE; k++) {
            out_r[i].data[k] = in1[i].data[k] + in2[i].data[k];
        }
    }
}
}
```

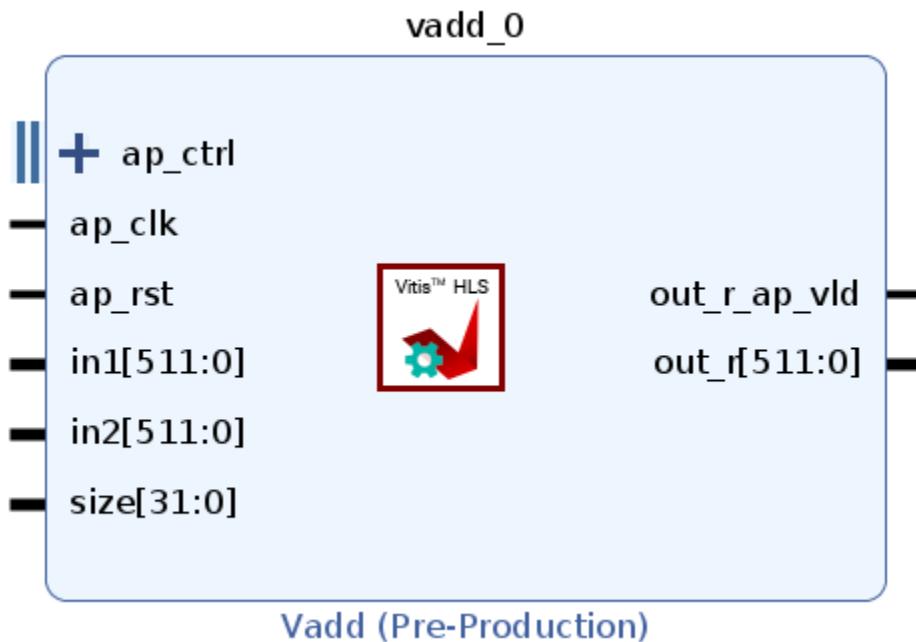
The `vadd` function includes:

- Two pointer inputs: `in1` and `in2`
- A pointer: `out_r` that the results are written to

- A scalar value `size`

With the default interface synthesis settings used for the Vivado IP flow, the design is synthesized into an RTL block with the ports and interfaces shown in the following figure.

Figure 36: RTL Ports After Default Interface Synthesis



In the default Vivado IP flow the tool creates three types of interface ports on the RTL design to handle the flow of both data and control.

- Clock and Reset ports: `ap_clk` and `ap_rst` are added to the kernel.
- Block-level control protocol: The `ap_ctrl` interface is implemented as an `s_axilite` interface.
- Port-level interface protocols: These are created for each argument in the top-level function and the function return (if the function returns a value). As explained in the table above most of the arguments use a port protocol of `ap_none`, and so have no control signals. In the `vadd` example above these ports include: `in1`, `in2`, and `size`. However, the `out_r_o` output port uses the `ap_vld` protocol and so is associated with the `out_r_o_ap_vld` signal.

AP_Memory in the Vivado IP Flow

The `ap_memory` is the default interface for the memory paradigm described in the tables above. In the Vivado IP flow it is used for communicating with memory resources such as BRAM and URAM. The `ap_memory` protocol also follows the address and data phase. The protocol initially requests to read/write the resource and waits until it receives an acknowledgment of the resource availability. It then initiates the data transfer phase of read/write.

An important consideration for `ap_memory` is that it can only perform a single beat data transfer to a single address, which is different from `m_axi` which can do burst accesses. This makes the `ap_memory` a lightweight protocol, compared to the others.

- Memory Resources: By default Vitis HLS implements a protocol to communicate with a single-port RAM resource. You can control the implementation of the protocol by specifying the `storage_type` as part of the INTERFACE pragma or directive. The `storage_type` lets you explicitly define which type of RAM is used, and which RAM ports are created (single-port or dual-port). If no `storage_type` is specified Vitis HLS uses:
 - A single-port RAM by default.
 - A dual-port RAM if it reduces the initiation interval or latency.

M_AXI Interfaces in the Vivado IP Flow

AXI4 memory-mapped (`m_axi`) interfaces allow an IP to read and write data in global memory (DDR, HBM, PLRAM). Memory-mapped interfaces are a convenient way of sharing data across multiple IP. The main advantages for `m_axi` interfaces are listed below:

- The interface has independent read and write channels
- It supports burst-based accesses
- It provides a queue for outstanding transactions
- **Understanding Burst Access:** AXI4 memory-mapped interfaces support high throughput bursts of up to 4K bytes with just a single address phase. With burst mode transfers, Vitis HLS reads or writes data using a single base address followed by multiple sequential data samples, which makes this mode capable of higher data throughput. Burst mode of operation is possible when you use the C `memcpy` function or a pipelined `for` loop. Refer to [Controlling AXI4 Burst Behavior](#) or [AXI Burst Transfers](#) for more information.
- **Automatic Port Widening and Port Width Alignment:**

As discussed in [Automatic Port Width Resizing](#), Vitis HLS has the ability to automatically widen a port width to facilitate data transfers and improve burst access when all the preconditions needed for bursting are present. In the Vivado IP flow the following configuration settings disable automatic port width resizing by default. To enable this feature you must change these configuration options (notice that one command is specified as bits and the other is specified as bytes):

```
config_interface -m_axi_max_widen_bitwidth 0
config_interface -m_axi_alignment_byte_size 0
```

- **Specifying Alignment for Vivado IP mode:**

The alignment for an `m_axi` port allows the port to read and write memory according to the specified alignment. Choosing the correct alignment is important as it will impact performance in the best case, and can impact functionality in the worst case.

Aligned memory access means that the pointer (or the start address of the data) is a multiple of a type-specific value called the alignment. The alignment is the natural address multiple where the type must be or should be stored (e.g. for performance reasons) on a Memory. For example, Intel 32-bit architecture stores words of 32 bits, each of 4 bytes in the memory. The data is aligned to one-word or 4-byte boundary.

The alignment should be consistent in the system. The alignment is determined when the IP is operating in AXI4 master mode and should be specified, like the Intel 32-bit architecture with 4-byte alignment. When the IP is operating in slave mode the alignment should match the alignment of the master.

- **Rules for Offset:**

The default for `m_axi` offset is `offset=direct` and `default_slave_interface=s_axilite`. However, in the Vivado IP flow you can change it as described in [Offset and Modes of Operation](#).

- **Bundle Interfaces - Performance vs. Resource Utilization:**

By default, Vitis HLS groups function arguments with compatible options into a single `m_axi` interface adapter as described in [M_AXI Bundles](#). Bundling ports into a single interface helps save device resources by eliminating AXI4 logic, which can be necessary when working in congested designs.

However, a single interface bundle can limit the performance of the IP because all the memory transfers have to go through a single interface. The `m_axi` interface has independent READ and WRITE channels, so a single interface can read and write simultaneously, though only at one location. Using multiple bundles lets you increase performance by creating multiple interfaces to connect to memory banks.

S_AXILITE in the Vivado IP Flow

In the Vivado IP flow, the default execution control is managed by register reads and writes through an `s_axilite` interface using the default `ap_ctrl_hs` control protocol. The IP is software controlled by reading and writing the control registers of an `s_axilite` interface as described in [S_AXILITE Control Register Map](#).

The `s_axilite` interface provides the following features:

- **Control Protocols:** The block control protocol as specified in [Block-Level Control Protocols](#).
- **Scalar Arguments:** Scalar arguments from the top-level function can be mapped to an `s_axilite` interface which creates a register for the value as described in [S_AXILITE Control Register Map](#). The software can perform reads/writes to this register space.
- **Rules for Offset:** The Vivado IP flow defines the size, or range of addresses assigned to a port based on the data type of the associated C-argument in the top-level function. However, the tool also lets you manually define the offset size as described in [S_AXILITE Offset Option](#).

- **Rules for Bundle:** In the Vivado IP flow you can specify multiple bundles using the `s_axilite` interface, and this will create a separate interface adapter for each bundle you have defined. However, there are some rules related to using multiple bundles that you should be familiar with as explained in [S_AXILITE Bundle Rules](#).

AP_FIFO in the Vivado IP Flow

In the Vivado IP flow, the `ap_fifo` interface protocol is the default interface for the streaming paradigm on the interface for communication with a memory resource FIFO, and can also be used as a communication channel between different functions inside the IP. This protocol should only be used if the data is accessed sequentially, and Xilinx *strongly recommends* using the `hls::stream<data_type>` which implements a FIFO.



TIP: The `<data_type>` should not be the same as the `T_data_type`, which should only be used on the interface.

AXIS Interfaces in the Vivado IP Flow

The AXI4-Stream protocol (`axis`) is an alternative for streaming interfaces, and defines a single uni-directional channel for streaming data in a sequential manner. Unlike the `m_axi` protocol, the AXI4-Stream interfaces can burst an unlimited amount of data, which significantly improves performance. Unlike the AXI4 memory-mapped interface which needs an address to read/write the memory, the `axis` interface simply passes data to another `axis` interface without needing an address, and so uses fewer device resources. Combined, these features make the streaming interface a light-weight high performance interface as described in [AXI4-Stream Interfaces](#).

AXI Adapter Interface Protocols



IMPORTANT! As discussed in [Interfaces for Vitis Kernel Flow](#), the AXI4 adapter interfaces are the default interfaces used by Vitis HLS for the Vitis Application Acceleration Development flow, though they are also supported in the Vivado IP flow. The AXI4-Stream Accelerator Adapter is a soft Xilinx® LogiCORE™ Intellectual Property (IP) core used as a infrastructure block for connecting hardware accelerators to embedded CPUs.

The AXI4 interfaces supported by Vitis HLS include the AXI4-Stream interface (`axis`), AXI4-Lite (`s_axilite`), and AXI4 master (`m_axi`) interfaces. For a complete description of the AXI4 interfaces, including timing and ports, see the [Vivado Design Suite: AXI Reference Guide \(UG1037\)](#). As described in the following sections, the AXI4 interfaces implement an adapter to manage communication according to the protocol. None of the other available Vitis HLS interfaces implement such an adapter.

- **`m_axi`:** Specify on arrays and pointers (and references in C++) only. The `m_axi` mode specifies an [AXI4 Memory Mapped interface](#).



TIP: You can group bundle arguments into a single `m_axi` interface.

- **s_axilite**: Specify this protocol on any type of argument except streams. The `s_axilite` mode specifies an [AXI4-Lite slave interface](#).



TIP: You can bundle multiple arguments into a single `s_axilite` interface.

- **axis**: Specify this protocol on input arguments or output arguments only, not on input/output arguments. The `axis` mode specifies an [AXI4-Stream interface](#).

AXI4 Master Interface

AXI4 memory-mapped (`m_axi`) interfaces allow kernels to read and write data in global memory (DDR, HBM, PLRAM). Memory-mapped interfaces are a convenient way of sharing data across different elements of the accelerated application, such as between the host and kernel, or between kernels on the accelerator card. Refer to [Vitis-HLS-Introductory-Examples/Interface/Memory](#) on Github for examples of some of these concepts.

The main advantages for `m_axi` interfaces are listed below:

- The interface has a separate and independent read and write channels
- It supports burst-based accesses with potential performance of ~17 GB/s
- It provides support for outstanding transactions

In the Vitis Kernel flow the `m_axi` interface is assigned by default to pointer and array arguments. In this flow it supports the following default features:

- Pointer and array arguments are automatically mapped to the `m_axi` interface
- The default mode of operation is `offset=slave` in the Vitis flow and should not be changed
- All pointer and array arguments are mapped to a single interface bundle to conserve device resources, and ports share read and write access across the time it is active
- The default alignment in the Vitis flow is set to 64 bytes
- The maximum read/write burst length is set to 16 by default

While not used by default in the Vivado IP flow, when the `m_axi` interface is specified it has the following default features:

- The default operation mode is `offset=off` but you can change it as described in [Offset and Modes of Operation](#)
- Assigned pointer and array arguments are mapped to a single interface bundle to conserve device resources, and share the interface across the time it is active
- The default alignment in Vivado IP flow is set to 1 byte
- The maximum read/write burst length is set to 16 by default

In both the Vivado IP flow and Vitis kernel flow, the INTERFACE pragma or directive can be used to modify default values as needed. Some customization can help improve design performance as described in [Optimizing AXI System Performance](#).

You can use an AXI4 master interface on array or pointer/reference arguments, which Vitis HLS implements in one of the following modes:

- Individual data transfers
- Burst mode data transfers

With individual data transfers, Vitis HLS reads or writes a single element of data for each address. The following example shows a single read and single write operation. In this example, Vitis HLS generates an address on the AXI interface to read a single data value and an address to write a single data value. The interface transfers one data value per address.

```
void bus (int *d) {  
    static int acc = 0;  
  
    acc += *d;  
    *d = acc;  
}
```

With burst mode transfers, Vitis HLS reads or writes data using a single base address followed by multiple sequential data samples, which makes this mode capable of higher data throughput. Burst mode of operation is possible when you use the C `memcpy` function or a pipelined `for` loop. Refer to [AXI Burst Transfers](#) for more information.



IMPORTANT! The C `memcpy` function is only supported for synthesis when used to transfer data to or from a top-level function argument specified with an AXI4 master interface.

When this example is synthesized, it results in the interface shown in the following figure.

Note: In this figure, the AXI4 interfaces are collapsed.

Figure 37: AXI4 Interface



When using a `for` loop to implement burst reads or writes, follow these requirements:

- Pipeline the loop
- Access addresses in increasing order
- Do not place accesses inside a conditional statement

- For nested loops, do not flatten loops, because this inhibits the burst operation

Note: Only one read and one write is allowed in a `for` loop unless the ports are bundled in different AXI ports.

Offset and Modes of Operation



IMPORTANT! In the Vitis kernel flow the default mode of operation is `offset=slave` and should not be changed.

The AXI4 Master interface has a read/write address channel that can be used to read/write specific addresses. By default the `m_axi` interface starts all read and write operations from the address `0x00000000`. For example, given the following code, the design reads data from addresses `0x00000000` to `0x000000C7` (50 32-bit words, gives 200 bytes), which represents 50 address values. The design then writes data back to the same addresses.

```
#include <stdio.h>
#include <string.h>

void example(int *a){

#pragma HLS INTERFACE mode=m_axi port=a depth=50

    int i;
    int buff[50];

    //memcpy creates a burst access to memory
    //multiple calls of memcpy cannot be pipelined and will be scheduled
    sequentially
    //memcpy requires a local buffer to store the results of the memory
    transaction
    memcpy(buff,(const int*)a,50*sizeof(int));

    for(i=0; i < 50; i++){
        buff[i] = buff[i] + 100;
    }

    memcpy((int *)a,buff,50*sizeof(int));
}
```

The tool provides the capability to let the base address be configured statically in the Vivado IP for instance, or dynamically by the application or another IP during run time.

The `m_axi` interface can be both a master initiating transactions, and also a slave interface that receives the data and sends acknowledgment. Depending on the mode specified with the `offset` option of the `INTERFACE` pragma, an HLS IP can use multiple approaches to set the base address.



TIP: The `config_interface -m_axi_offset` command provides a global setting for the offset, that can be overridden for specific `m_axi` interfaces using the `INTERFACE` pragma `offset` option.

- **Master Mode:** When acting as a master interface with different `offset` options, the `m_axi` interface start address can be either hard-coded or set at run time.

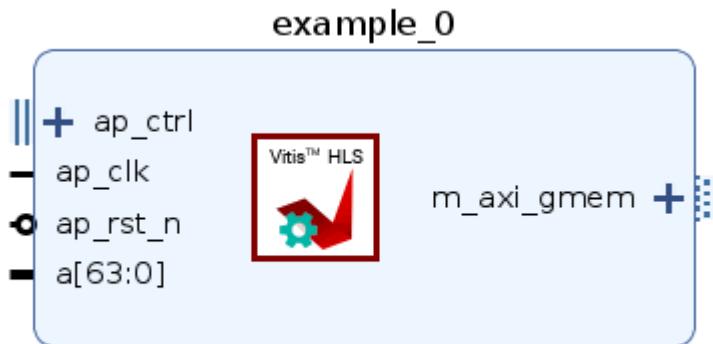
- `offset=off`: Vitis HLS sets a base address for the `m_axi` interface when the IP is used in the Vivado IP integrator tool. One disadvantage with this approach is that you cannot change the base address during run time. See [Customizing AXI4 Master Interfaces in IP Integrator](#) for setting the base address.

The following example is synthesized with `offset=off`, the default for the Vivado IP flow.

```
void example(int *a){  
#pragma HLS INTERFACE m_axi depth=50 port=a offset=off  
  
    int i;  
    int buff[50];  
  
    //memcpy creates a burst access to memory  
    //multiple calls of memcpy cannot be pipelined and will be scheduled  
    sequentially  
    //memcpy requires a local buffer to store the results of the memory  
    transaction  
    memcpy(buff,(const int*)a,50*sizeof(int));  
  
    for(i=0; i < 50; i++){  
        buff[i] = buff[i] + 100;  
    }  
  
    memcpy((int *)a,buff,50*sizeof(int));  
}
```

- `offset=direct`: Vitis HLS generates a port on the IP for setting the address. Note the addition of the `a` port as shown in the figure below. This lets you update the address at run time, so you can have one `m_axi` interface reading and writing different locations. For example, an HLS module that reads data from an ADC into RAM, and an HLS module that processes that data. Since you can change the address on the module, while one HLS module is processing the initial dataset the other module can be reading more data into different address.

```
void example(int *a){  
#pragma HLS INTERFACE m_axi depth=50 port=a offset=direct  
...  
}
```

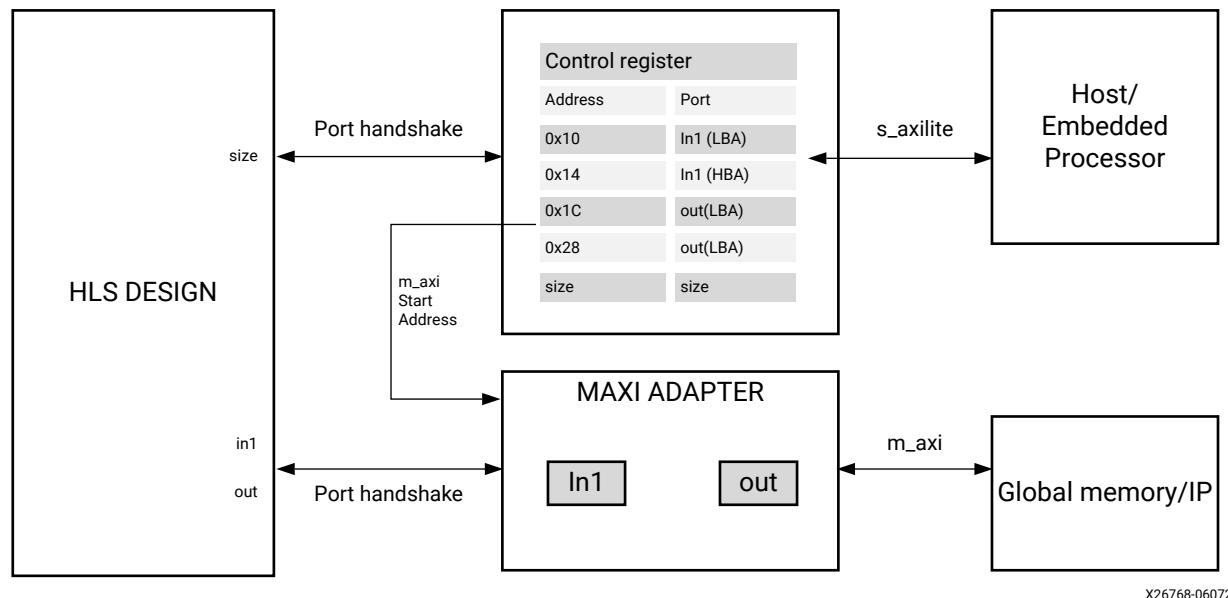
Figure 38: **offset=direct**

- **Slave Mode:** The slave mode for an interface is set with `offset=slave`. In this mode the IP will be controlled by the host application, or the micro-controller through the `s_axilite` interface. This is the default for the Vitis kernel flow, and can also be used in the Vivado IP flow. Here is the flow of operation:

1. initially, the Host/CPU will start the IP or kernel using the block-level control protocol which is mapped to the `s_axilite` adapter.
2. The host will send the scalars and address offsets for the `m_axi` interfaces through the `s_axilite` adapter.
3. The `m_axi` adapter will read the start address from the `s_axilite` adapter and store it in a queue.
4. The HLS design starts to read the data from the global memory.

As shown in the figure below, the HLS design will have both the `s_axilite` adapter for the base address, and the `m_axi` to perform read and write transfer to the global memory.

Figure 39: AXI Adapters in Slave Mode



X26768-060722

Offset Rules

The following are rules associated with the `offset` option:

- Fully Specified Offset: When the user explicitly sets the offset value the tool uses the specified settings. The user can also set different offset values for different `m_axi` interfaces in the design, and the tool will use the specified offsets.

```
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=out offset=direct
#pragma HLS INTERFACE mode=m_axi bundle=BUS_B port=in1 offset=slave
#pragma HLS INTERFACE mode=m_axi bundle=BUS_C port=in2 offset=off
```

- No Offset Specified: If there are no offsets specified in the INTERFACE pragma, the tool will defer to the setting specified by `config_interface -m_axi_offset`.

Note: If the global `m_axi_offset` setting is specified, and the design has an `s_axilite` interface, the global setting is ignored and `offset=slave` is assumed.

```
void top(int *a) {
#pragma HLS interface mode=m_axi port=a
#pragma HLS interface mode=s_axilite port=a
}
```

Controlling the Address Offset in an AXI4 Interface

By default, the AXI4 master interface starts all read and write operations from address 0x00000000. For example, given the following code, the design reads data from addresses 0x00000000 to 0x000000C7 (50 32-bit words, gives 200 bytes), which represents 50 address values. The design then writes data back to the same addresses.

```
void example(int *a){  
  
#pragma HLS INTERFACE mode=m_axi depth=50 port=a  
#pragma HLS INTERFACE mode=s_axilite port=return bundle=AXILiteS  
  
int i;  
int buff[50];  
  
memcpy(buff,(const int*)a,50*sizeof(int));  
  
for(i=0; i < 50; i++){  
    buff[i] = buff[i] + 100;  
}  
memcpy((int *)a,buff,50*sizeof(int));  
}
```

To apply an address offset, use the `-offset` option with the INTERFACE directive, and specify one of the following options:

- `off`: Does not apply an offset address. This is the default.
- `direct`: Adds a 32-bit port to the design for applying an address offset.
- `slave`: Adds a 32-bit register inside the AXI4-Lite interface for applying an address offset.

In the final RTL, Vitis HLS applies the address offset directly to any read or write address generated by the AXI4 master interface. This allows the design to access any address location in the system.

If you use the `slave` option in an AXI interface, you must use an AXI4-Lite port on the design interface. Xilinx recommends that you implement the AXI4-Lite interface using the following pragma:

```
#pragma HLS INTERFACE mode=s_axilite port=return
```

In addition, if you use the `slave` option and you used several AXI4-Lite interfaces, you must ensure that the AXI master port offset register is bundled into the correct AXI4-Lite interface. In the following example, port `a` is implemented as an AXI master interface with an offset and AXI4-Lite interfaces called `AXI_Lite_1` and `AXI_Lite_2`:

```
#pragma HLS INTERFACE mode=m_axi port=a depth=50 offset=slave  
#pragma HLS INTERFACE mode=s_axilite port=return bundle=AXI_Lite_1  
#pragma HLS INTERFACE mode=s_axilite port=b bundle=AXI_Lite_2
```

The following INTERFACE directive is required to ensure that the offset register for port `a` is bundled into the AXI4-Lite interface called `AXI_Lite_1`:

```
#pragma HLS INTERFACE mode=s_axilite port=a bundle=AXI_Lite_1
```

M_AXI Bundles

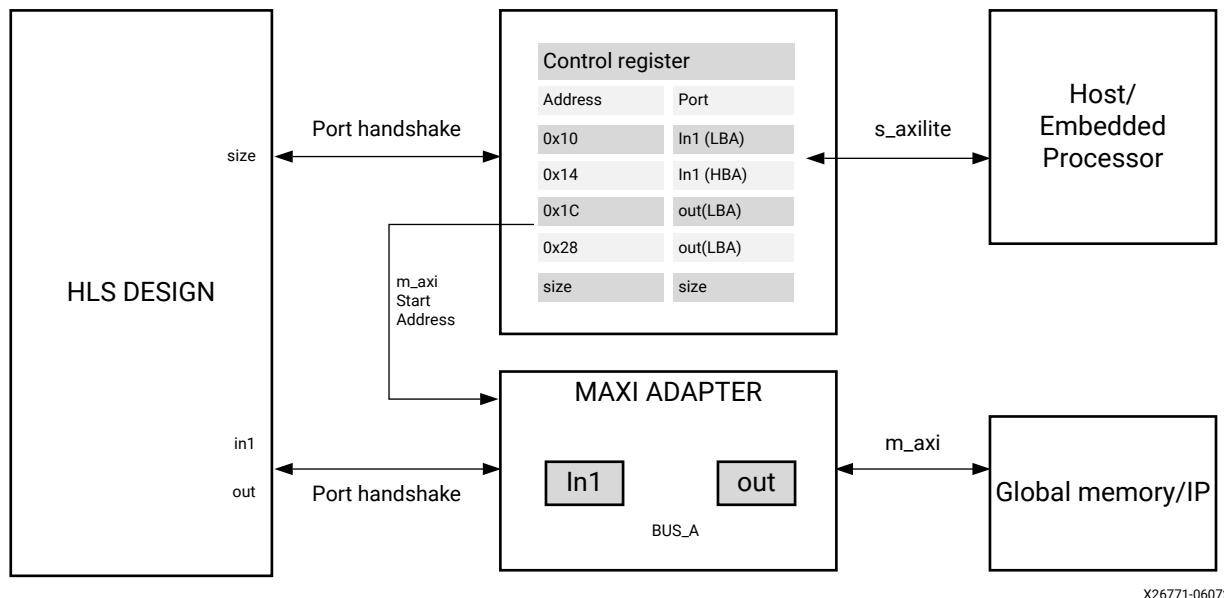
Vitis HLS groups function arguments with compatible options into a single `m_axi` interface adapter. Bundling ports into a single interface helps save FPGA resources by eliminating AXI logic, but it can limit the performance of the kernel because all the memory transfers have to go through a single interface. The `m_axi` interface has independent READ and WRITE channels, so a single interface can read and write simultaneously, though only at one location. Using multiple bundles the bandwidth and throughput of the kernel can be increased by creating multiple interfaces to connect to multiple memory banks.

In the following example all the pointer arguments are grouped into a single `m_axi` adapter using the interface option `bundle=BUS_A`, and adds a single `s_axilite` adapter for the `m_axi` offsets, the scalar argument `size`, and the function return.

```
extern "C" {
void vadd(const unsigned int *in1, // Read-Only Vector 1
          const unsigned int *in2, // Read-Only Vector 2
          unsigned int *out,      // Output Result
          int size               // Size in integer
) {

#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=out
#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=in1
#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=in2
#pragma HLS INTERFACE mode=s_axilite port=in1
#pragma HLS INTERFACE mode=s_axilite port=in2
#pragma HLS INTERFACE mode=s_axilite port=out
#pragma HLS INTERFACE mode=s_axilite port=size
#pragma HLS INTERFACE mode=s_axilite port=return
```

Figure 40: MAXI and S_AXILITE



X26771-060722

You can also choose to bundle function arguments into separate interface adapters as shown in the following code. Here the argument `in2` is grouped into a separate interface adapter with `bundle=BUS_B`. This creates a new `m_axi` interface adapter for port `in2`.

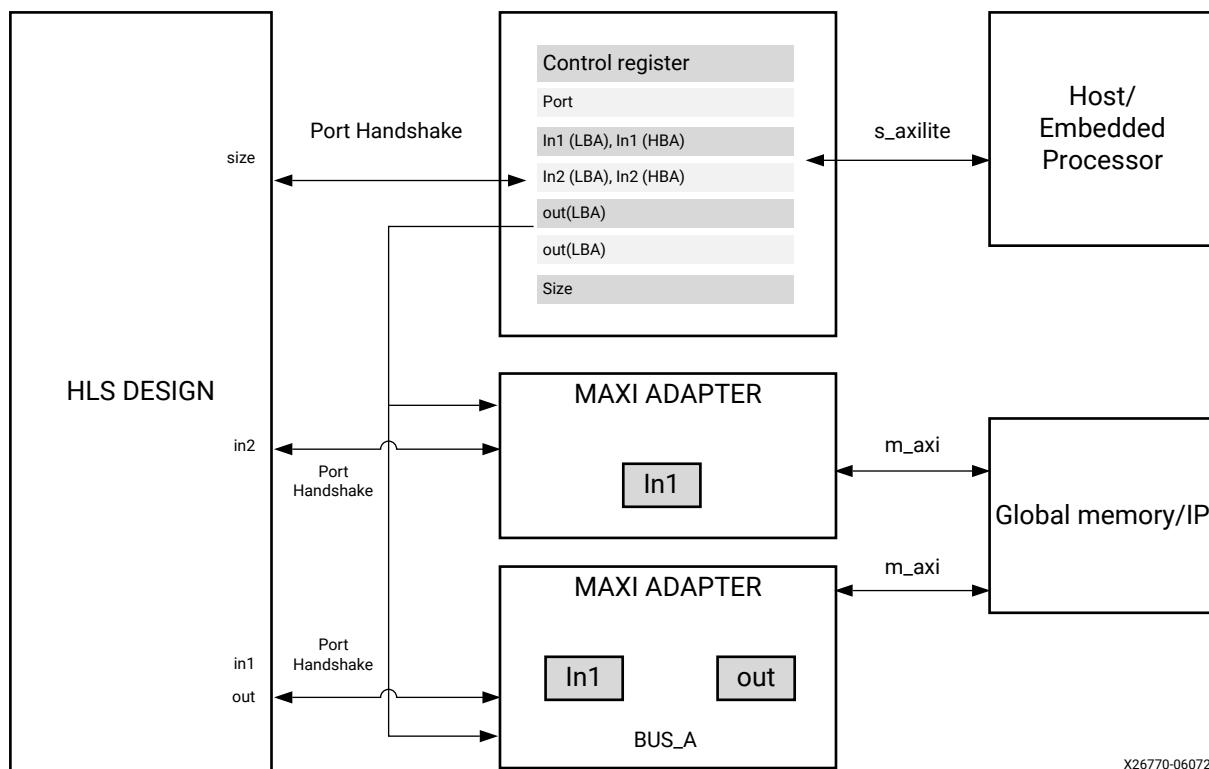
```

extern "C" {
void vadd(const unsigned int *in1, // Read-Only Vector 1
          const unsigned int *in2, // Read-Only Vector 2
          unsigned int *out,      // Output Result
          int size                // Size in integer
) {

#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=out
#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=in1
#pragma HLS INTERFACE mode=m_axi bundle=BUS_B port=in2
#pragma HLS INTERFACE mode=s_axilite port=in1
#pragma HLS INTERFACE mode=s_axilite port=in2
#pragma HLS INTERFACE mode=s_axilite port=out
#pragma HLS INTERFACE mode=s_axilite port=size
#pragma HLS INTERFACE mode=s_axilite port=return
}

```

Figure 41: 2 MAXI Bundles



Bundle Rules

The global configuration command `config_interface -m_axi_auto_max_ports false` will limit the number of interface bundles to the minimum required. It will allow the tool to group compatible ports into a single `m_axi` interface. The default setting for this command is disabled (false), but you can enable it to maximize bandwidth by creating a separate `m_axi` adapter for each port.

With `m_axi_auto_max_ports` disabled, the following are some rules for how the tool handles bundles under different circumstances:

- Default Bundle Name:** The tool groups all interface ports with no bundle name into a single `m_axi` interface port using the tool default name `bundle=<default>`, and names the RTL port `m_axi_<default>`. The following pragmas:

```
#pragma HLS INTERFACE mode=m_axi port=a depth=50
#pragma HLS INTERFACE mode=m_axi port=a depth=50
#pragma HLS INTERFACE mode=m_axi port=a depth=50
```

Result in the following messages:

```
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem' to
'm_axi'.
```

2. User-Specified Bundle Names: The tool groups all interface ports with the same user-specified `bundle=<string>` into the same `m_axi` interface port, and names the RTL port the value specified by `m_axi_<string>`. Ports without bundle assignments are grouped into the default bundle as described above. The following pragmas:

```
#pragma HLS INTERFACE mode=m_axi port=a depth=50 bundle=BUS_A
#pragma HLS INTERFACE mode=m_axi port=b depth=50
#pragma HLS INTERFACE mode=m_axi port=c depth=50
```

Result in the following messages:

```
INFO: [RTGEN 206-500] Setting interface mode on port 'example/BUS_A' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem' to
'm_axi'.
```



IMPORTANT! If you bundle incompatible interfaces Vitis HLS issues a message and ignores the bundle assignment.

M_AXI Resources

The AXI Master Adapter converts the customized AXI commands from the HLS scheduler to standard AXI AMBA protocol and sends them to the external memory. The MAXI adapter uses resources such as FIFO to store the requests/Data and ack. Here is the summary of the modules and the resource they consume:

- **Write Module:** The bus write modules performs the write operations.
 - `FIFO_wreq`: This FIFO module stores the future write requests. When the AW channel is available a new write request to global memory will be popped out of this FIFO.
 - `buff_wdata`: This FIFO stores the future write data that needs to be sent to the global memory. When the W channel is available and AXI protocol conditions are met, the write data of size= `burst_length` will be popped out of this FIFO and sent to the global memory.
 - `FIFO_resp`: This module is responsible for controlling the number of pipelined outstanding requests sent to the global memory.
- **Read Module:** These modules perform the read operations. It uses the following resources
 - `FIFO_rreq`: This FIFO module stores the future write requests. When the AR channel is free a read request to global memory will be popped out of this FIFO.
 - `buff_rdata`: This FIFO stores the read data that are received from the global memory.

The device resource consumption of the M_AXI adapter is a sum of all the write modules (size of the FIFO_wreq module, buff_wdata, and size of FIFO_resp) and the sum of all read modules. In general, the size of the FIFO is calculated as = Width * Depth. When you refer to a 1KB FIFO storage it can be one of the configurations such as 32*32, 8*64 etc, which are selected according to the design specification. Similarly, the adapter FIFO storage can be globally configured for the design using the following options of the [config_interface](#) command:

- -m_axi_latency
- -m_axi_max_read/write_burst_length
- -m_axi_num_read/write_outstanding
- -m_axi_addr64



TIP: You can also use similar options on the [INTERFACE](#) pragma or directive to configure specific m_axi interfaces.

These configuration options control the width and depth of the FIFO as shown below.

- Size of the FIFO_wreq/rreq module = (width(config_interface -m_axi_addr64 [=true|false])) * Depth(config_interface -m_axi_latency)). This FIFO will be implemented as a shift register by the Vivado tool.
 - Size of the buff_wdata module = (width (port width after HLS synthesis) * Depth (config_interface -m_axi_num_read/write_outstanding * config_interface -m_axi_max_read/write_burst_length)).
-
- TIP:** This FIFO by default will be implemented as BRAM, but it can be implemented in LUTRAM or URAM as determined by config_interface -maxi_buffer_impl.
- Size of the FIFO_resp module = width(2) * depth (config_interface -m_axi_num_read/write_outstanding-1).

Controlling AXI4 Burst Behavior

An optimal AXI4 interface is one in which the design never stalls while waiting to access the bus, and after bus access is granted, the bus never stalls while waiting for the design to read/write. To create the optimal AXI4 interface, the following options are provided in the INTERFACE pragma or directive to specify the behavior of the bursts and optimize the efficiency of the AXI4 interface. Refer to [AXI Burst Transfers](#) for more information on burst transfers.



TIP: The volatile qualifier prevents burst access to or from the variable.

Some of these options use internal storage to buffer data and may have an impact on area and resources:

- **latency:** Specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request a number of cycles (latency) before the read or write is expected. If this figure is too low, the design will be ready too soon and may stall waiting for the bus. If this figure is too high, bus access may be granted but the bus may stall waiting on the design to start the access.
- **max_read_burst_length:** Specifies the maximum number of data values read during a burst transfer.
- **num_read_outstanding:** Specifies how many read requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size: $\text{num_read_outstanding} * \text{max_read_burst_length} * \text{word_size}$.
- **max_write_burst_length:** Specifies the maximum number of data values written during a burst transfer.
- **num_write_outstanding:** Specifies how many write requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size: $\text{num_read_outstanding} * \text{max_read_burst_length} * \text{word_size}$

The following example can be used to help explain these options:

```
#pragma HLS interface mode=m_axi port=input offset=slave bundle=gmem0
depth=1024*1024*16/(512/8)
latency=100
num_read_outstanding=32
num_write_outstanding=32
max_read_burst_length=16
max_write_burst_length=16
```

The interface is specified as having a latency of 100. Vitis HLS seeks to schedule the request for burst access 100 clock cycles before the design is ready to access the AXI4 bus. To further improve bus efficiency, the options `num_write_outstanding` and `num_read_outstanding` ensure the design contains enough buffering to store up to 32 read and write accesses. This allows the design to continue processing until the bus requests are serviced. Finally, the options `max_read_burst_length` and `max_write_burst_length` ensure the maximum burst size is 16 and that the AXI4 interface does not hold the bus for longer than this.

These options allow the behavior of the AXI4 interface to be optimized for the system in which it will operate. The efficiency of the operation does depend on these values being set accurately.

Automatic Port Width Resizing

In the Vitis tool flow Vitis HLS provides the ability to automatically re-size `m_axi` interface ports to 512-bits to improve burst access. However, automatic port width resizing only supports standard C data types and does not support aggregate types such as `ap_int`, `ap_uint`, `struct`, or `array`.



IMPORTANT! Structs on the interface prevent automatic widening of the port. You must break the struct into individual elements to enable this feature.

Vitis HLS controls automatic port width resizing using the following two commands:

- `config_interface -m_axi_max_widen_bitwidth <N>`: Directs the tool to automatically widen bursts on M-AXI interfaces up to the specified bitwidth. The value of `<N>` must be a power-of-two between 0 and 1024.
- `config_interface -m_axi_alignment_byte_size <N>`: Note that burst widening also requires strong alignment properties. Assume pointers that are mapped to `m_axi` interfaces are at least aligned to the provided width in bytes (power of two). This can help automatic burst widening.

In the Vitis Kernel flow automatic port width resizing is enabled by default with the following:

```
config_interface -m_axi_max_widen_bitwidth 512
config_interface -m_axi_alignment_byte_size 64
```

In the Vivado IP flow this feature is disabled by default:

```
config_interface -m_axi_max_widen_bitwidth 0
config_interface -m_axi_alignment_byte_size 0
```

Automatic port width resizing will only re-size the port if a burst access can be seen by the tool. Therefore all the preconditions needed for bursting, as described in [AXI Burst Transfers](#), are also needed for port resizing. These conditions include:

- Must be a monotonically increasing order of access (both in terms of the memory location being accessed as well as in time). You cannot access a memory location that is in between two previously accessed memory locations- aka no overlap.
- The access pattern from the global memory should be in sequential order, and with the following additional requirements:
 - The sequential accesses need to be on a non-vector type
 - The start of the sequential accesses needs to be aligned to the widen word size
 - The length of the sequential accesses needs to be divisible by the widen factor

The following code example is used in the calculations that follow:

```
vadd_pipeline:
    for (int i = 0; i < iterations; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_len/c_n max = c_len/c_n

        // Pipelining loops that access only one variable is the ideal way to
        // increase the global memory bandwidth.
        read_a:
            for (int x = 0; x < N; ++x) {
#pragma HLS LOOP_TRIPCOUNT min = c_n max = c_n
#pragma HLS PIPELINE II = 1
                result[x] = a[i * N + x];
            }

        read_b:
            for (int x = 0; x < N; ++x) {
```

```
#pragma HLS LOOP_TRIPCOUNT min = c_n max = c_n
#pragma HLS PIPELINE II = 1
    result[x] += b[i * N + x];
}

write_c:
for (int x = 0; x < N; ++x) {
#pragma HLS LOOP_TRIPCOUNT min = c_n max = c_n
#pragma HLS PIPELINE II = 1
    c[i * N + x] = result[x];
}
}
```

The width of the automatic optimization for the code above is performed in three steps:

1. The tool checks for the number of access patterns in the `read_a` loop. There is one access during one loop iteration, so the optimization determines the interface bit-width as $32 = 32 * 1$ (bitwidth of the int variable * accesses).
2. The tool tries to reach the default max specified by the `config_interface -m_axi_max_widen_bitwidth 512`, using the following expression terms:

```
length = (ceil((loop-bound of index inner loops) *
(loop-bound of index - outer loops)) * #(of access-patterns))
```

- In the above code, the outer loop is an imperfect loop so there will not be burst transfers on the outer-loop. Therefore the length will only include the inner-loop. Therefore the formula will be shortened to:

```
length = (ceil((loop-bound of index inner loops)) * #(of access-
patterns))
```

or: $\text{length} = \text{ceil}(128) * 32 = 4096$

3. Is the calculated length a power of 2? If Yes, then the length will be capped to the width specified by `-m_axi_max_widen_bitwidth`.

There are some pros and cons to using the automatic port width resizing which you should consider when using this feature. This feature improves the read latency from the DDR as the tool is reading a big vector, instead of the data type size. It also adds more resources as it needs to buffer the huge vector and shift the data accordingly to the data path size.

Creating an AXI4 Interface with 32-bit Address

By default, Vitis HLS implements the AXI4 port with a 64-bit address bus. However, some devices such as the Zynq-7000 have a 32 bit address bus. In this case you can implement the AXI4 interface with a 32-bit address bus by disabling the `m_axi_addr64` interface configuration option as follows:

1. Select **Solution** → **Solution Settings**.
2. In the Solution Settings dialog box, click the **General** category, and **Edit** the existing `config_interface` command, or click **Add** to add one.

3. In the Edit or Add dialog box, select **config_interface**, and disable **m_axi_addr64**.

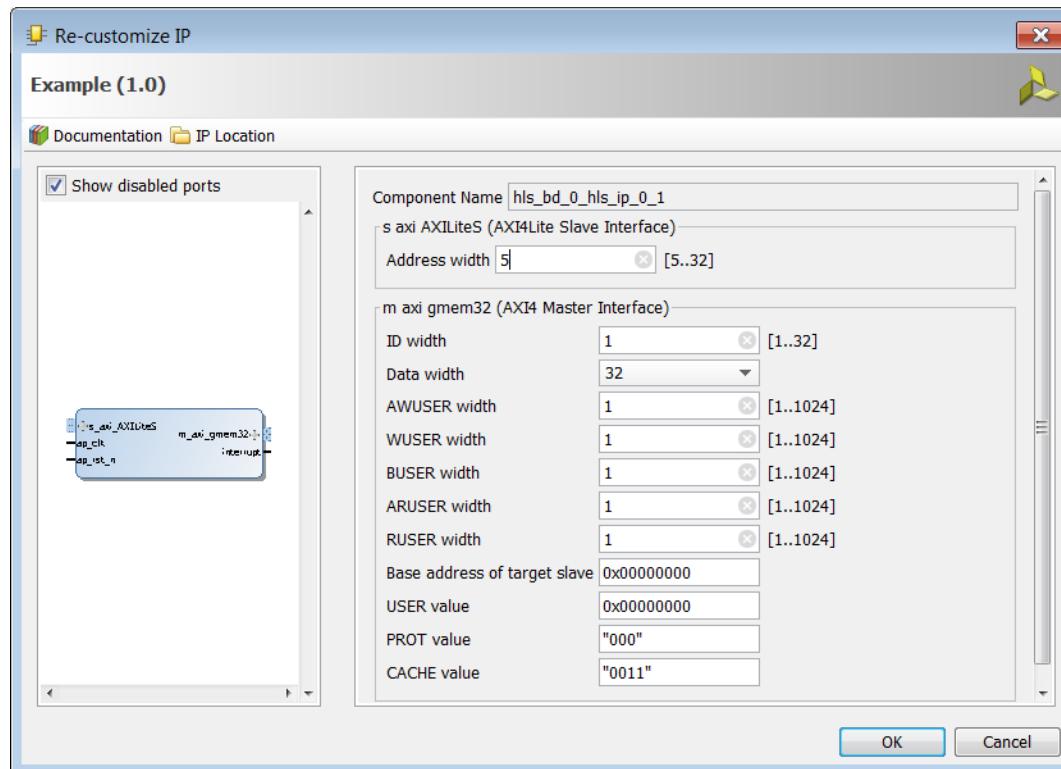
IMPORTANT! When you disable the **m_axi_addr64** option, Vitis HLS implements all AXI4 interfaces in the design with a 32-bit address bus.

Customizing AXI4 Master Interfaces in IP Integrator

When you incorporate an HLS RTL design that uses an AXI4 master interface into a design in the Vivado IP integrator, you can customize the block. From the block diagram in IP integrator, select the HLS block, right-click, and select **Customize Block** to customize any of the settings provided. A complete description of the AXI4 parameters is provided in this [link](#) in the Vivado Design Suite: AXI Reference Guide ([UG1037](#)).

The following figure shows the Re-Customize IP dialog box for the design shown below. This design includes an AXI4-Lite port.

Figure 42: Customizing AXI4 Master Interfaces in IP Integrator



AXI4-Lite Interface

Overview

An HLS IP or kernel can be controlled by a host application, or embedded processor using the Slave AXI4-Lite interface (`s_axilite`) which acts as a system bus for communication between the processor and the kernel. Using the `s_axilite` interface the host or an embedded processor can start and stop the kernel, and read or write data to it. When Vitis HLS synthesizes the design the `s_axilite` interface is implemented as an adapter that captures the data that was communicated from the host in registers on the adapter. Refer to [Vitis-HLS-Introductory-Examples/Interface/Register](#) on Github for examples of some of these concepts.

The AXI4-Lite interface performs several functions within a Vivado IP or Vitis kernel:

- It maps a block-level control mechanism which can be used to start and stop the kernel.
- It provides a channel for passing scalar arguments, pointers to scalar values, function return values, and address offsets for `m_axi` interfaces from the host to the IP or kernel
- For the Vitis Kernel flow:
 - The tool will automatically infer the `s_axilite` interface pragma to provide offsets to pointer arguments assigned to `m_axi` interfaces, scalar values, and function return type.
 - Vitis HLS lets you read to or write from a pointer to a scalar value when assigned to an `s_axilite` interface. Pointers are assigned by default to `m_axi` interfaces, so this requires you to manually assign the pointer to the `s_axilite` using the INTERFACE pragma or directive:

```
int top(int *a, int *b) {  
#pragma HLS interface s_axilite port=a
```

- **Bundle:** Do not specify the `bundle` option for the `s_axilite` adapter in the Vitis Kernel flow. The tool will create a single `s_axilite` interface that will serve for the whole design.



IMPORTANT! HLS will return an error if multiple bundles are specified for the Vitis Kernel flow.

-
- **Offset:** The tool will automatically choose the offsets for the interface. Do not specify any offsets in this flow.
 - For the Vivado IP flow:
 - This flow will not use the `s_axilite` interface by default.
 - To use the `s_axilite` as a communication channel for scalar arguments, pointers to scalar values, offset to `m_axi` pointer address, and function return type, you must manually specify the INTERFACE pragma or directive.
 - **Bundle:** This flow supports multiple `s_axilite` interfaces, specified by bundle. Refer to [S_AXILITE Bundle Rules](#) for more information.

- Offset: By default the tool will place the arguments in a sequential order starting from 0x10 in the control register map. Refer to [S_AXILITE Offset Option](#) for additional details.

S_AXILITE Example

The following example shows how Vitis HLS implements multiple arguments, including the function return, as an `s_axilite` interface. Because each pragma uses the same name for the bundle option, each of the ports is grouped into a single interface.

```
void example(char *a, char *b, char *c)
{
#pragma HLS INTERFACE mode=s_axilite port=return bundle=BUS_A
#pragma HLS INTERFACE mode=s_axilite port=a bundle=BUS_A
#pragma HLS INTERFACE mode=s_axilite port=b bundle=BUS_A
#pragma HLS INTERFACE mode=s_axilite port=c bundle=BUS_A
#pragma HLS INTERFACE mode=ap_vld port=b

    *c += *a + *b;
}
```

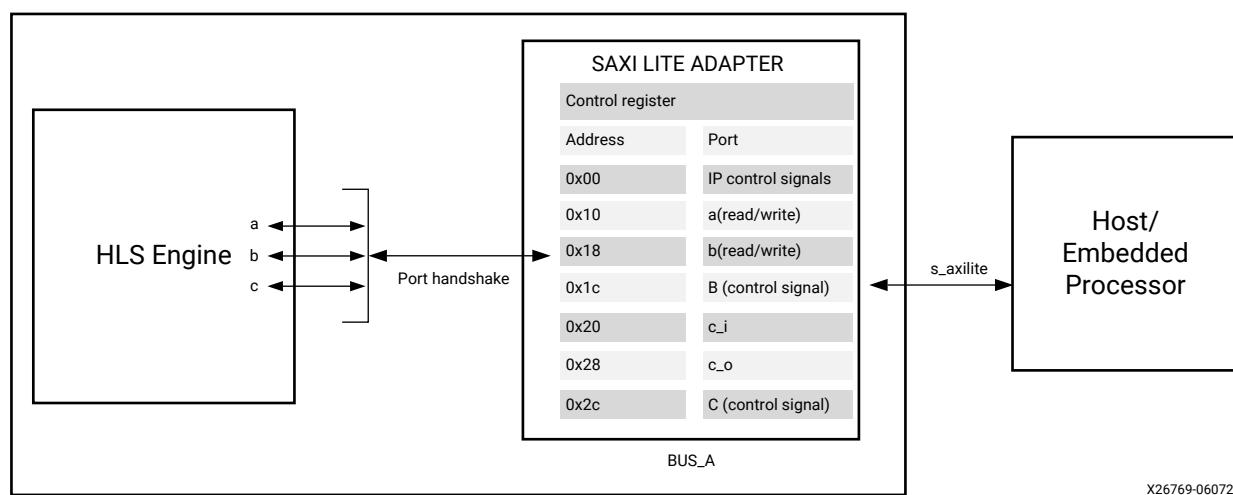


TIP: If you do not specify the `bundle` option, Vitis HLS groups all arguments into a single `s_axilite` bundle and automatically names the port.

The synthesized example will be part of a system that has three important elements as shown in the figure below:

1. Host application running on an x86 or embedded processor interacting with the IP or kernel
2. AXI Lite Adapter: The INTERFACE pragma implements an `s_axilite` adapter. The adapter has two primary functions: implementing the interface protocol to communicate with the host, and providing a Control Register Map to the IP or kernel.
3. The HLS engine or function that implements the design logic

Figure 43: S_AXILITE Adapter



By default, Vitis HLS automatically assigns the address for each port that is grouped into an `s_axilite` interface. The size, or range of addresses assigned to a port is dependent on the argument data type and the port protocol used, as described below. You can also explicitly define the address using the `offset` option as discussed in [S_AXILITE Offset Option](#).

- Port a: By default, is implemented as `ap_none`. 1-word for the data signal is assigned and only 3 bits are used as the argument data type is `char`. Remaining bits are unused.
- Port b: is implemented as `ap_vld` defined by the INTERFACE pragma in the example. The corresponding control register is of size 2 bytes (16-bits) and is divided into two sections as follows:
 - (0x1c) Control signal : 1-word for the control signal is assigned.
 - (0x18) Data signal: 1-word for the data signal is assigned and only 3 bits are used as the argument data type is `char`. Remaining bits are unused.
- Port c: By default, is implemented as `ap_ovld` as an output. The corresponding control register is of size 4 bytes (32 bits) and is divided into three sections:
 - (0x20) Data signal of `c_i`: 1-word for the input data signal is assigned, and only 3 bits are used as the argument data type is `char`, the rest are not used.
 - (0x24) Reserved Space
 - (0x28) Data signal of `c_o`: 1-word for the output data signal is assigned.
 - (0x2c) Control signal of `c_o` : 1-word for control signal `ap_ovld` is assigned and only 3 bits are used as the argument data type is `char`. Remaining bits are unused.

In operation the host application will initially start the kernel by writing into the Control address space (0x00). The host/CPU completes the initial setup by writing into the other address spaces which are associated with the various function arguments as defined in the example.

The control signal for port b is asserted and only then can the kernel read ports a and b (port a is `ap_none` and does not have a control signal). Until that time the design is stalled and waiting for the `valid` register to be set for port b. Each time port b is read by the HLS engine the input `valid` register is cleared and the register resets to logic 0.

After the HLS engine finishes its computation, the output value on port C is stored in the control register and the corresponding `valid` bit is set for the host to read. After the host reads the data, the HLS engine will write the `ap_done` bit in the Control register (0x00) to mark the end of the IP computation.

Vitis HLS reports the assigned addresses in the [S_AXILITE Control Register Map](#), and also provides them in [C Driver Files](#) to aid in your software development. Using the `s_axilite` interface, you can exploit the C driver files for use with code running on an embedded or x86 processor using provided C application program interface (API) functions, to let you control the hardware from your software.

S_AXILITE Control Register Map

Vitis HLS automatically generates a Control Register Map for controlling the Vivado IP or Vitis kernel, and the ports grouped into `s_axilite` interface. The register map, which is added to the generated RTL files, can be divided into two sections:

1. Block-level control signals
2. Function arguments mapped into the `s_axilite` interface

In the Vitis kernel flow, the block protocol is associated with the `s_axilite` interface by default. To change the default block protocol, specify the interface pragma as follows:

```
#pragma HLS INTERFACE mode=ap_ctrl_hs port=return
```

In the Vivado IP flow though, the block control protocol is assigned to its own interface, `ap_ctrl`, as seen in [Interfaces for Vivado IP Flow](#). However, if you are using an `s_axilite` interface in your IP, you can also assign the block control protocol to that interface using the following INTERFACE pragmas, as an example:

```
#pragma HLS INTERFACE mode=s_axilite port=return bundle=BUS_A
#pragma HLS INTERFACE mode=ap_ctrl_hs port=return bundle=BUS_A
```

In the Control Register Map, Vitis HLS reserves addresses 0x00 through 0x18 for the block-level protocol, interrupt, mailbox and auto-restart controls. The latter are present only when counted auto-restart and the mailbox are enabled, as shown below:

Table 6: Addresses

Address	Description
0x00	Control signals
0x04	Global Interrupt Enable Register
0x08	IP Interrupt Enable Register (Read/Write)
0x0c	IP Interrupt Status Register (Read/TOW)
0x10	Auto-restart counter (Write; present only with counted autorestart)
0x14	Input mailbox write (Read/Write; present only when the input mailbox is enabled)
0x18	Output mailbox read (Read/Write; present only when the output mailbox is enabled)

The Control signals (0X00) contains `ap_start`, `ap_done`, `ap_ready`, and `ap_idle`; and in the case of `ap_ctrl_chain` the block protocol also contains `ap_continue`. These are the *block-level interface* signals which are accessed through the `s_axilite` adapter.

To start the block operation the `ap_start` bit in the Control register must be set to 1. The HLS engine will then proceed and read any inputs grouped into the AXI4-Lite slave interface from the register in the interface.

When the block completes the operation, the `ap_done`, `ap_idle` and `ap_ready` registers will be set by the hardware output ports and the results for any output ports grouped into the `s_axilite` interface read from the appropriate register.

For function arguments, Vitis HLS automatically assigns the address for each argument or port that is assigned to the `s_axilite` interface. The tool will assign each port an offset starting from `0x10`, the lower addresses being reserved for control signals. The size, or range of addresses assigned to a port is dependent on the argument data type and the port protocol used.

Because the variables grouped into an AXI4-Lite interface are function arguments which do not have a default value in the C code, none of the argument registers in the `s_axilite` interface can be assigned a default value. The registers can be implemented with a reset using the `config_rtl` command, but they cannot be assigned any other default value.

The Control Register Map generated by Vitis HLS is provided below:

```
-----Address Info-----
// 0x00 : Control signals
//     bit 0 - ap_start (Read/Write/COH)
//     bit 1 - ap_done (Read)
//     bit 2 - ap_idle (Read) can be disabled with config_rtl -no_idle
//
//     bit 3 - ap_ready (Read/COR)
//     bit 4 - ap_continue (Read/Write/SC) for ap_ctrl_chain protocol
//     bit 7 - auto_restart (Read/Write) enabled by config_interface -
//             s_axilite_auto_restart_counter
//     bit 9 - interrupt (Read) Present when there is at least one
//             enabled interrupt
//     others - reserved
// 0x04 : Global Interrupt Enable Register
//     bit 0 - Global Interrupt Enable (Read/Write)
//     others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//     bit 0 - enable ap_done interrupt (Read/Write)
//     bit 1 - enable ap_ready interrupt (Read/Write)
//     others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//     bit 0 - ap_done (COR/TOW)
//     bit 1 - ap_ready (COR/TOW)
//     others - reserved
// 0x10 : Data signal of a
//     bit 7~0 - a[7:0] (Read/Write)
//     others - reserved
// 0x14 : reserved
// 0x18 : Data signal of b
//     bit 7~0 - b[7:0] (Read/Write)
//     others - reserved
//     : Control signal of b
//     bit 0 - b_ap_vld (Read/Write/SC)
//     others - reserved
// 0x20 : Data signal of c_i
//     bit 7~0 - c_i[7:0] (Read/Write)
//     others - reserved
// 0x24 : reserved
// 0x28 : Data signal of c_o
//     bit 7~0 - c_o[7:0] (Read)
//     others - reserved
```

```
// 0x2c : Control signal of c_o
//      bit 0 - c_o_ap_vld (Read/COR)
//      others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH =
// Clear on Handshake)
```

S_AXILITE and Port-Level Protocols

Port-level I/O protocols sequence data into and out of the HLS engine from the `s_axilite` adapter as seen in [S_AXILITE Example](#). In the Vivado IP flow, you can assign port-level I/O protocols to the individual ports and signals bundled into an `s_axilite` interface. In the Vitis kernel flow, changing the default port-level I/O protocols is not recommended unless necessary. The tool assigns a default port protocol to a port depending on the type and direction of the argument associated with it. The port can contain one or more of the following:

- Data signal for the argument
- Valid signal (`ap_vld/ap_ovld`) to indicate when the data can be read
- Acknowledge signal (`ap_ack`) to indicate when the data has been read

The default port protocol assignments for various argument types are as follows:

Table 7: Supported Argument Types

Argument Type	Default	Supported
scalar	<code>ap_none</code>	<code>ap_ack</code> and <code>ap_vld</code> can also be used
Pointers/References		
Inputs	<code>ap_none</code>	<code>ap_ack</code> and <code>ap_vld</code>
Outputs	<code>ap_vld</code>	<code>ap_none</code> , <code>ap_ack</code> , and <code>ap_ovld</code> can also be used
Inouts	<code>ap_ovld</code>	<code>ap_none</code> , <code>ap_ack</code> , and <code>ap_vld</code> are also supported



IMPORTANT! Arrays default to `ap_memory`. The `bram` port protocol is not supported for arrays in an `s_axilite` interface.

The [S_AXILITE Example](#) groups port `b` into the `s_axilite` interface and specifies port `b` as using the `ap_vld` protocol with INTERFACE pragmas. As a result, the `s_axilite` adapter contains a register for the port `b` data, and a register for the port `b` input valid signal.

If the input valid register is not set to logic 1, the data in the `b` data register is not considered valid, and the design stalls and waits for the valid register to be set. Each time port `b` is read, Vitis HLS automatically clears the input valid register and resets the register to logic 0.



RECOMMENDED: To simplify the operation of your design, Xilinx recommends that you use the default port protocols associated with the `s_axilite` interface.

S_AXILITE Bundle Rules

In the [S_AXILITE Example](#) all the function arguments are grouped into a single `s_axilite` interface adapter specified by the `bundle=BUS_A` option in the INTERFACE pragma. The `bundle` option simply lets you group ports together into one interface.

In the Vitis kernel flow there should only be a single interface bundle, commonly named `s_axi_control` by the tool. So you should not specify the `bundle` option in that flow, or you will probably encounter an error during synthesis. However, in the Vivado IP flow you can specify multiple bundles using the `s_axilite` interface, and this will create a separate interface adapter for each bundle you have defined. The following example shows this:

```
void example(char *a, char *b, char *c)
{
#pragma HLS INTERFACE mode=s_axilite port=a bundle=BUS_A
#pragma HLS INTERFACE mode=s_axilite port=b bundle=BUS_A
#pragma HLS INTERFACE mode=s_axilite port=c bundle=OUT
#pragma HLS INTERFACE mode=s_axilite port=return bundle=BUS_A
#pragma HLS INTERFACE mode=ap_vld port=b
    *c += *a + *b;
}
```

After synthesis completes, the Synthesis Summary report provides feedback regarding the number of `s_axilite` adapters generated. The SW-to-HW Mapping section of the report contains the HW info showing the control register offset and the address range for each port.

However, there are some rules related to using bundles with the `s_axilite` interface.

1. Default Bundle Names: This rule explicitly groups all interface ports with no bundle name into the same AXI4-Lite interface port, uses the tool default bundle name, and names the RTL port `s_axi_<default>`, typically `s_axi_control`.

In this example all ports are mapped to the default bundle:

```
void top(char *a, char *b, char *c)
{
#pragma HLS INTERFACE mode=s_axilite port=a
#pragma HLS INTERFACE mode=s_axilite port=b
#pragma HLS INTERFACE mode=s_axilite port=c
    *c += *a + *b;
}
```

2. User-Specified Bundle Names: This rule explicitly groups all interface ports with the same `bundle` name into the same AXI4-Lite interface port, and names the RTL port the value specified by `s_axi_<string>`.

The following example results in interfaces named `s_axi_BUS_A`, `s_axi_BUS_B`, and `s_axi_OUT`:

```
void example(char *a, char *b, char *c)
{
#pragma HLS INTERFACE mode=s_axilite port=a bundle=BUS_A
#pragma HLS INTERFACE mode=s_axilite port=b bundle=BUS_B
#pragma HLS INTERFACE mode=s_axilite port=c bundle=OUT
#pragma HLS INTERFACE mode=s_axilite port=return bundle=OUT
#pragma HLS INTERFACE mode=ap_vld port=b
    *c += *a + *b;
}
```

3. Partially Specified Bundle Names: If you specify `bundle` names for some arguments, but leave other arguments unassigned, then the tool will bundle the arguments as follows:

- Group all ports into the specified bundles as indicated by the `INTERFACE` pragmas.
- Group any ports without bundle assignments into a default named bundle. The default name can either be the standard tool default, or an alternative default name if the tool default has already been specified by the user.

In the following example the user has specified `bundle=control`, which is the tool default name. In this case, port c will be assigned to `s_axi_control` as specified by the user, and the remaining ports will be bundled under `s_axi_control_r`, which is an alternative default name used by the tool.

```
void top(char *a, char *b, char *c) {
#pragma HLS INTERFACE mode=s_axilite port=a
#pragma HLS INTERFACE mode=s_axilite port=b
#pragma HLS INTERFACE mode=s_axilite port=c bundle=control
}
```

S_AXILITE Offset Option

Note: The Vitis kernel flow determines the required offsets. Do not specify the `offset` option in that flow.

In the Vivado IP flow, Vitis HLS defines the size, or range of addresses assigned to a port in the [S_AXILITE Control Register Map](#) depending on the argument data type and the port protocol used. However, the `INTERFACE` pragma also contains an `offset` option that lets you specify the address offset in the AXI4-Lite interface.

When specifying the offset for your argument, you must consider the size of your data and reserve some extra for the port control protocol. The range of addresses you reserve should be based on a 32-bit word. You should reserve enough 32-bit words to fit your argument data type, and add one additional word for the control protocol, even for `ap_none`.



TIP: In the case of the `ap_memory` protocol for arrays, you do not need to reserve the extra word for the control protocol. In this case, simply reserve enough 32-bit words to fit your argument data type.

For example, to reserve enough space for a double you need to reserve two 32-bit words for the 64-bit data type, and then reserve an additional 32-bit word for the control protocol. So you need to reserve a total of three 32-bit words, or 96 bits. If your argument offset starts at 0x020, then the next available offset would begin at 0x02c, in order to reserve the required address range for your argument.

If you make a mistake in setting the offset of your arguments, by not reserving enough address range to fit your data type and the control protocol, Vitis HLS will recognize the error, will warn you of the issue, and will recover by moving your misplaced argument register to the end of the Control Register Map. This will allow your build to proceed, but may not work with your host application or driver if they were written to your specified offset.

C Driver Files

When an AXI4-Lite slave interface is implemented, a set of C driver files are automatically created. These C driver files provide a set of APIs that can be integrated into any software running on a CPU and used to communicate with the device via the AXI4-Lite slave interface.

The C driver files are created when the design is packaged as IP in the IP catalog.

Driver files are created for standalone and Linux modes. In standalone mode the drivers are used in the same way as any other Xilinx standalone drivers. In Linux mode, copy all the C files (.c) and header files (.h) files into the software project.

The driver files and API functions derive their name from the top-level function for synthesis. In the above example, the top-level function is called “example”. If the top-level function was named “DUT” the name “example” would be replaced by “DUT” in the following description. The driver files are created in the packaged IP (located in the `impl` directory inside the `solution`).

Table 8: C Driver Files for a Design Named Example

File Path	Usage Mode	Description
<code>data/example.mdd</code>	Standalone	Driver definition file.
<code>data/example.tcl</code>	Standalone	Used by SDK to integrate the software into an SDK project.
<code>src/xexample_hw.h</code>	Both	Defines address offsets for all internal registers.
<code>src/xexample.h</code>	Both	API definitions
<code>src/xexample.c</code>	Both	Standard API implementations
<code>src/xexample_sinit.c</code>	Standalone	Initialization API implementations
<code>src/xexample_linux.c</code>	Linux	Initialization API implementations
<code>src/Makefile</code>	Standalone	Makefile

In file `xexample.h`, two structs are defined.

- **XExample_Config:** This is used to hold the configuration information (base address of each AXI4-Lite slave interface) of the IP instance.

- **XExample:** This is used to hold the IP instance pointer. Most APIs take this instance pointer as the first argument.

The standard API implementations are provided in files `xexample.c`, `xexample_sinit.c`, `xexample_linux.c`, and provide functions to perform the following operations.

- Initialize the device
- Control the device and query its status
- Read/write to the registers
- Set up, monitor, and control the interrupts

Refer to [Section V: Vitis HLS C Driver Reference](#) for a description of the API functions provided in the C driver files.



IMPORTANT! The C driver APIs always use an unsigned 32-bit type (U32). You might be required to cast the data in the C code into the expected type.

C Driver Files and Float Types

C driver files always use a data 32-bit unsigned integer (U32) for data transfers. In the following example, the function uses float type arguments `a` and `r1`. It sets the value of `a` and returns the value of `r1`:

```
float calculate(float a, float *r1)
{
#pragma HLS INTERFACE mode=ap_vld register port=r1
#pragma HLS INTERFACE mode=s_axilite port=a
#pragma HLS INTERFACE mode=s_axilite port=r1
#pragma HLS INTERFACE mode=s_axilite port=return

    *r1 = 0.5f*a;
    return (a>0);
}
```

After synthesis, Vitis HLS groups all ports into the default AXI4-Lite interface and creates C driver files. However, as shown in the following example, the driver files use type U32:

```
// API to set the value of A
void XCalculate_SetA(XCalculate *InstancePtr, u32 Data) {
    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
    XCalculate_WriteReg(InstancePtr->Hls_periph_bus_BaseAddress,
XCALCULATE_HLS_PERIPH_BUS_ADDR_A_DATA, Data);
}

// API to get the value of R1
u32 XCalculate_GetR1(XCalculate *InstancePtr) {
    u32 Data;

    Xil_AssertNonvoid(InstancePtr != NULL);
    Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
```

```
    Data = XCalculate_ReadReg(InstancePtr->Hls_periph_bus_BaseAddress,  
    XCALCULATE_HLS_PERIPH_BUS_ADDR_R1_DATA);  
    return Data;  
}
```

If these functions work directly with float types, the write and read values are not consistent with expected float type. When using these functions in software, you can use the following casts in the code:

```
float a=3.0f,r1;  
u32 ua,url1;  
  
// cast float "a" to type U32  
XCalculate_SetA(&calculate,*((u32*)&a));  
url1=XCalculate_GetR1(&calculate);  
  
// cast return type U32 to float type for "r1"  
r1=*((float*)&url1);
```

Controlling Hardware



TIP: The example provided below demonstrates the `ap_ctrl_hs` block control protocol, which is the default for the Vivado IP flow. Refer to [Block-Level Control Protocols](#) for more information and a description of the `ap_ctrl_chain` protocol which is the default for the Vitis kernel flow.

In this example, the hardware header file `xexample_hw.h` provides a complete list of the memory mapped locations for the ports grouped into the AXI4-Lite slave interface, as described in [S_AXILITE Control Register Map](#).

```
// 0x00 : Control signals  
//         bit 0 - ap_start (Read/Write/SC)  
//         bit 1 - ap_done (Read/COR)  
//         bit 2 - ap_idle (Read)  
//         bit 3 - ap_ready (Read)  
//         bit 7 - auto_restart (Read/Write)  
//         others - reserved  
// 0x04 : Global Interrupt Enable Register  
//         bit 0 - Global Interrupt Enable (Read/Write)  
//         others - reserved  
// 0x08 : IP Interrupt Enable Register (Read/Write)  
//         bit 0 - Channel 0 (ap_done)  
//         bit 1 - Channel 1 (ap_ready)  
// 0x0c : IP Interrupt Status Register (Read/TOW)  
//         bit 0 - Channel 0 (ap_done)  
//         others - reserved  
// 0x10 : Data signal of a  
//         bit 7~0 - a[7:0] (Read/Write)  
//         others - reserved  
// 0x14 : reserved  
// 0x18 : Data signal of b  
//         bit 7~0 - b[7:0] (Read/Write)  
//         others - reserved  
// 0x1c : reserved  
// 0x20 : Data signal of c_i  
//         bit 7~0 - c_i[7:0] (Read/Write)  
//         others - reserved  
// 0x24 : reserved
```

```

// 0x28 : Data signal of c_o
//          bit 7~0 - c_o[7:0] (Read)
//          others - reserved
// 0x2c : Control signal of c_o
//          bit 0 - c_o_ap_vld (Read/COR)
//          others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH =
Clear on
Handshake)

```

To correctly program the registers in the `s_axilite` interface, you must understand how the hardware ports operate with the default port protocols, or the custom protocols as described in [S_AXILITE and Port-Level Protocols](#).

For example, to start the block operation the `ap_start` register must be set to 1. The device will then proceed and read any inputs grouped into the AXI4-Lite slave interface from the register in the interface. When the block completes operation, the `ap_done`, `ap_idle` and `ap_ready` registers will be set by the hardware output ports and the results for any output ports grouped into the AXI4-Lite slave interface read from the appropriate register.

The implementation of function argument `c` in the example highlights the importance of some understanding how the hardware ports operate. Function argument `c` is both read and written to, and is therefore implemented as separate input and output ports `c_i` and `c_o`, as explained in [S_AXILITE Example](#).

The first recommended flow for programming the `s_axilite` interface is for a one-time execution of the function:

- Use the interrupt function standard API implementations provided in the [C Driver Files](#) to determine how you want the interrupt to operate.
- Load the register values for the block input ports. In the above example this is performed using API functions `XExample_Set_a`, `XExample_Set_b`, and `XExample_Set_c_i`.
- Set the `ap_start` bit to 1 using `XExample_Start` to start executing the function. This register is self-clearing as noted in the header file above. After one transaction, the block will suspend operation.
- Allow the function to execute. Address any interrupts which are generated.
- Read the output registers. In the above example this is performed using API functions `XExample_Get_c_o_vld`, to confirm the data is valid, and `XExample_Get_c_o`.

Note: The registers in the `s_axilite` interface obey the same I/O protocol as the ports. In this case, the output valid is set to logic 1 to indicate if the data is valid.

- Repeat for the next transaction.

The second recommended flow is for continuous execution of the block. In this mode, which is described in much more detail in the next section, the input ports included in the AXI4-Lite interface should only be ports which perform configuration. The block will typically run much faster than a CPU. If the block must wait for inputs, the block will spend most of its time waiting:

- Use the interrupt function to determine how you wish the interrupt to operate.
- Load the register values for the block input ports. In the above example this is performed using API functions `XExample_Set_a`, `XExample_Set_a` and `XExample_Set_c_i`.
- Set the auto-start function using API `XExample_EnableAutoRestart`.
- Allow the function to execute. The individual port I/O protocols will synchronize the data being processed through the block.
- Address any interrupts which are generated. The output registers could be accessed during this operation but the data may change often.
- Use the API function `XExample_DisableAutoRestart` to prevent any more executions.
- Read the output registers. In the above example this is performed using API functions `XExample_Get_c_o` and `XExample_Set_c_o_vld`.

Controlling Software

The API functions can be used in the software running on the CPU to control the hardware block. An overview of the process is:

- Create an instance of the hardware
- Look Up the device configuration
- Initialize the device
- Set the input parameters of the HLS block
- Start the device and read the results

An example application is shown below.

```
#include "xexample.h"      // Device driver for HLS HW block
#include "xparameters.h"

// HLS HW instance
XExample HlsExample;
XExample_Config *ExamplePtr

int main() {
    int res_hw;

    // Look Up the device configuration
    ExamplePtr = XExample_LookupConfig(XPAR_XEXAMPLE_0_DEVICE_ID);
    if (!ExamplePtr) {
        print("ERROR: Lookup of accelerator configuration failed.\n\r");
        return XST_FAILURE;
    }

    // Initialize the Device
    status = XExample_CfgInitialize(&HlsExample, ExamplePtr);
    if (status != XST_SUCCESS) {
        print("ERROR: Could not initialize accelerator.\n\r");
        exit(-1);
    }
}
```

```

//Set the input parameters of the HLS block
XExample_Set_a(&HlsExample, 42);
XExample_Set_b(&HlsExample, 12);
XExample_Set_c_i(&HlsExample, 1);

// Start the device and read the results
XExample_Start(&HlsExample);
do {
    res_hw = XExample_Get_c_o(&HlsExample);
} while (XExample_Get_c_o(&HlsExample) == 0); // wait for valid data output
print("Detected HLS peripheral complete. Result received.\n\r");
}

```

Control Clock and Reset in AXI4-Lite Interfaces

Note: If you instantiate the slave AXI4-Lite register file in a bus fabric that uses a different clock frequency, Vivado IP integrator will automatically generate a clock domain crossing (CDC) slice that performs the same function as the control clock described below, making use of the option unnecessary.

By default, Vitis HLS uses the same clock for the AXI4-Lite interface and the synthesized design. Vitis HLS connects all registers in the AXI4-Lite interface to the clock used for the synthesized logic (`ap_clk`).

Optionally, you can use the INTERFACE directive `clock` option to specify a separate clock for each AXI4-Lite port. When connecting the clock to the AXI4-Lite interface, you must use the following protocols:

- AXI4-Lite interface clock must be synchronous to the clock used for the synthesized logic (`ap_clk`). That is, both clocks must be derived from the same master generator clock.
- AXI4-Lite interface clock frequency must be equal to or less than the frequency of the clock used for the synthesized logic (`ap_clk`).

If you use the `clock` option with the INTERFACE directive, you only need to specify the `clock` option on one function argument in each bundle. Vitis HLS implements all other function arguments in the bundle with the same clock and reset. Vitis HLS names the generated reset signal with the prefix `ap_rst_` followed by the clock name. The generated reset signal is active-Low independent of the `config_rtl` command.

The following example shows how Vitis HLS groups function arguments `a` and `b` into an AXI4-Lite port with a clock named `AXI_clk1` and an associated reset port.

```

// Default AXI-Lite interface implemented with independent clock called
AXI_clk1
#pragma HLS interface mode=s_axilite port=a clock=AXI_clk1
#pragma HLS interface mode=s_axilite port=b

```

In the following example, Vitis HLS groups function arguments `c` and `d` into AXI4-Lite port `CTRL1` with a separate clock called `AXI_clk2` and an associated reset port.

```

// CTRL1 AXI-Lite bundle implemented with a separate clock (called AXI_clk2)
#pragma HLS interface mode=s_axilite port=c bundle=CTRL1 clock=AXI_clk2
#pragma HLS interface mode=s_axilite port=d bundle=CTRL1

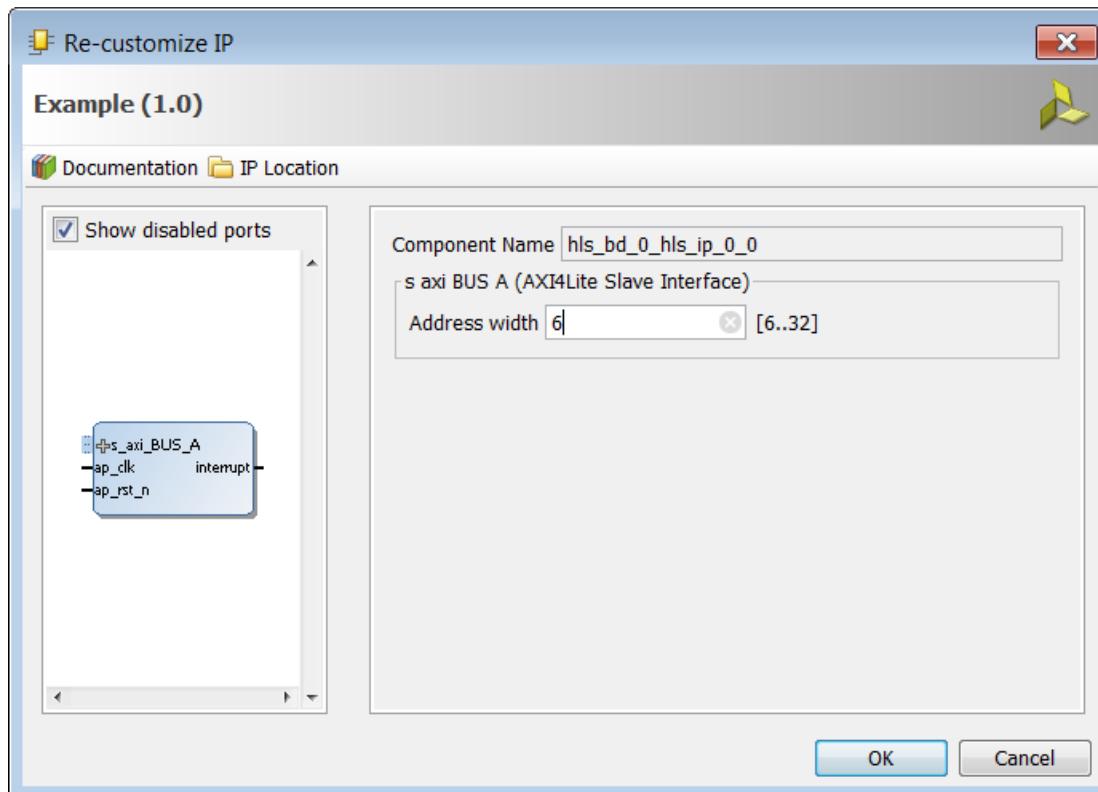
```

Customizing AXI4-Lite Slave Interfaces in IP Integrator

When an HLS RTL design using an AXI4-Lite slave interface is incorporated into a design in Vivado IP integrator, you can customize the block. From the block diagram in IP integrator, select the HLS block, right-click with the mouse button and select **Customize Block**.

The address width is by default configured to the minimum required size. Modify this to connect to blocks with address sizes less than 32-bit.

Figure 44: Customizing AXI4-Lite Slave Interfaces in IP Integrator



AXI4-Stream Interfaces



IMPORTANT! `hls::axis` (and `ap_axiu/ap_axis`) cannot be used on internal functions or variables as the AXI4-Stream protocol is only supported on the interfaces of top-level functions. For internal functions or variables you must use `hls::stream` objects as described in [HLS Stream Library](#).

An AXI4-Stream interface can be applied to any input argument and any array or pointer output argument. Because an AXI4-Stream interface transfers data in a sequential streaming manner, it cannot be used with arguments that are both read and written. In terms of data layout, the data type of the AXI4-Stream is aligned to the next byte. For example, if the size of the data type is 12 bits, it will be extended to 16 bits. Depending on whether a signed/unsigned interface is selected, the extended bits are either sign-extended or zero-extended.

If the stream data type is an user-defined struct, the default procedure is to keep the struct aggregated and align the struct to the size of the largest data element to the nearest byte. The only exception to this rule is if the struct contains a `hls::stream` object. In this special case, the struct will be disaggregated and an axi stream will be created for each member element of the struct.



TIP: The maximum supported port width is 4096 bits, even for aggregated structs or reshaped arrays.

The following code examples show how the packed alignment depends on your struct type. If the struct contains only char type, as shown in the following example, then it will be packed with alignment of one byte. Total size of the struct will be two bytes:

```
struct A {  
    char foo;  
    char bar;  
};
```

However, if the struct has elements with different data types, as shown below, then it will be packed and aligned to the size of the largest data element, or four bytes in this example. Element `bar` will be padded with three bytes resulting in a total size of eight bytes for the struct:

```
struct A {  
    int foo;  
    char bar;  
};
```

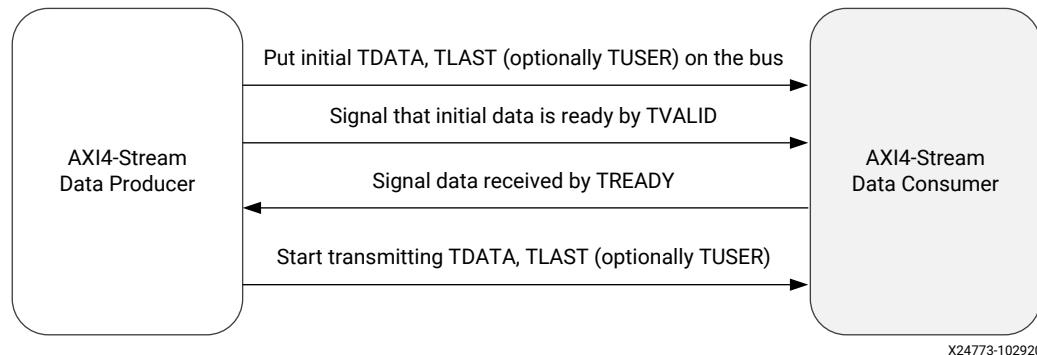


IMPORTANT! Structs contained in AXI4-Stream interfaces (`axis`) are aggregated by default, and the stream itself cannot be disaggregated. If separate streams for member elements of the struct are desired then this must be manually coded as separate elements, resulting in a separate axis interface for each element. Refer to [Vitis-HLS-Introductory-Examples/Interface/Aggregation_Disaggregation/disaggregation_of_axis_port](#) on Github for an example.

How AXI4-Stream Works

AXI4-Stream is a protocol designed for transporting arbitrary unidirectional data. In an AXI4-Stream, TDATA width of bits is transferred per clock cycle. The transfer is started once the producer sends the TVALID signal and the consumer responds by sending the TREADY signal (once it has consumed the initial TDATA). At this point, the producer will start sending TDATA and TLAST (TUSER if needed to carry additional user-defined sideband data). TLAST signals the last byte of the stream. So the consumer keeps consuming the incoming TDATA until TLAST is asserted.

Figure 45: AXI4-Stream Handshake



AXI4-Stream has additional optional features like sending positional data with TKEEP and TSTRB ports which makes it possible to multiplex both the data position and data itself on the TDATA signal. Using the TID and TDIST signals, you can route streams as these fields roughly corresponds to stream identifier and stream destination identifier. Refer to *Vivado Design Suite: AXI Reference Guide (UG1037)* or the *AMBA AXI4-Stream Protocol Specification (ARM IHI 0051A)* for more information.

How AXI4-Stream is Implemented

If your design requires a streaming interface begin by defining and using a streaming data structure like `hls::stream` in Vitis HLS. This simple object encapsulates the requirements of streaming and its streaming interface is by default implemented in the RTL as a FIFO interface (`ap_fifo`) but can be optionally, implemented as a handshake interface (`ap_hs`) or an AXI4-Stream interface (`axis`). Refer to [Vitis-HLS-Introductory-Examples/Interface/Streaming](#) on Github for different examples of streaming interfaces.

If a AXI4-Stream interface (`axis`) is specified via the interface pragma mode option, the interface implementation will mimic the style of an AXIS interface by defining the TDATA, TVALID and TREADY signals.

If a more formal AXIS implementation is desired, then Vitis HLS requires the usage of a special data type (`hls::axis` defined in `ap_axi_sdata.h`) to encapsulate the requirements of the AXI4-Stream protocol and implement the special RTL signals needed for this interface.

The AXI4-Stream interface is implemented as a struct type in Vitis HLS and has the following signature (defined in `ap_axi_sdata.h`):

```
template <typename T, size_t WUser, size_t WId, size_t WDest> struct axis
{ .. };
```

Where:

- `T`: The data type to be streamed.



TIP: This can support any data type, including `ap_fixed`.

- `WUser`: Width of the TUSER signal
- `WId`: Width of the TID signal
- `WDest`: Width of the TDest signal

When the stream data type (`T`) are simple integer types, there are two predefined types of AXI4-Stream implementations available:

- A signed implementation of the AXI4-Stream class (or more simply `ap_axis<Wdata, WUser, WId, WDest>`)
`hls::axis<ap_int<WData>, WUser, WId, WDest>`
- An unsigned implementation of the AXI4-Stream class (or more simply `ap_axiu<WData, WUser, WId, WDest>`)
`hls::axis<ap_uint<WData>, WUser, WId, WDest>`

The value specified for the `WUser`, `WId`, and `WDest` template parameters controls the usage of side-channel signals in the AXI4-Stream interface.

When the `hls::axis` class is used, the generated RTL will typically contain the actual data signal `TDATA`, and the following additional signals: `TVALID`, `TREADY`, `TKEEP`, `TSTRB`, `TLAST`, `TUSER`, `TID`, and `TDEST`.

`TVALID`, `TREADY`, and `TLAST` are necessary control signals for the AXI4-Stream protocol. `TKEEP`, `TSTRB`, `TUSER`, `TID`, and `TDEST` signals are optional special signals that can be used to pass around additional bookkeeping data.



TIP: If `WUser`, `WId`, and `WDest` are set to 0, the generated RTL will not include the optional `TUSER`, `TID`, and `TDEST` signals in the interface.

Registered AXI4-Stream Interfaces

As a default, AXI4-Stream interfaces are always implemented as registered interfaces to ensure that no combinational feedback paths are created when multiple HLS IP blocks with AXI4-Stream interfaces are integrated into a larger design. For AXI4-Stream interfaces, four types of register modes are provided to control how the interface registers are implemented:

- **Forward:** Only the `TDATA` and `TVALID` signals are registered.
- **Reverse:** Only the `TREADY` signal is registered.
- **Both:** All signals (`TDATA`, `TREADY`, and `TVALID`) are registered. This is the default.
- **Off:** None of the port signals are registered.

The AXI4-Stream side-channel signals are considered to be data signals and are registered whenever TDATA is registered.



RECOMMENDED: When connecting HLS generated IP blocks with AXI4-Stream interfaces at least one interface should be implemented as a registered interface or the blocks should be connected via an AXI4-Stream Register Slice.

There are two basic methods to use an AXI4-Stream in your design:

- Use an AXI4-Stream without side-channels.
- Use an AXI4-Stream with side-channels.

This second use model provides additional functionality, allowing the optional side-channels which are part of the AXI4-Stream standard, to be used directly in your C/C++ code.

AXI4-Stream Interfaces without Side-Channels

An AXI4-Stream is used without side-channels when the function argument, ap_axis or ap_axiu data type, does not contain any AXI4 side-channel elements (that is, when the WUser, WIId, and WDest parameters are set to 0). In the following example, both interfaces are implemented using an AXI4-Stream:

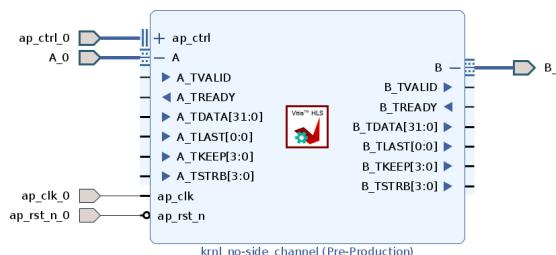
```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axiu<32, 0, 0, 0> trans_pkt;

void example(hls::stream< trans_pkt > &A, hls::stream< trans_pkt > &B)
{
#pragma HLS INTERFACE mode=axis port=A
#pragma HLS INTERFACE mode=axis port=B
    trans_pkt tmp;
    A.read(tmp);
    tmp.data += 5;
    B.write(tmp);
}
```

After synthesis, both arguments are implemented with a data port (TDATA) and the standard AXI4-Stream protocol ports, TVALID, TREADY, TKEEP, TLAST, and TSTRB, as shown in the following figure.

Figure 46: AXI4-Stream Interfaces without Side-Channels





TIP: If you specify an `hls::stream` object with a data type other than `ap_axis` or `ap_axiu`, the tool will infer an AXI4-Stream interface without the `TLAST` signal, or any of the side-channel signals. This implementation of the AXI4-Stream interface consumes fewer device resources, but offers no visibility into when the stream is ending.

Multiple variables can be combined into the same AXI4-Stream interface by using a struct, which is aggregated by Vitis HLS by default. Aggregating the elements of a struct into a single wide-vector, allows all elements of the struct to be implemented in the same AXI4-Stream interface.

AXI4-Stream Interfaces with Side-Channels

The following example shows how the side-channels can be used directly in the C/C++ code and implemented on the interface. The code uses `#include "ap_axi_sdata.h"` to provide an API to handle the side-channels of the AXI4-Stream interface. In the following example a signed 32-bit data type is used:

```
#include "ap_axi_sdata.h"
#include "ap_int.h"
#include "hls_stream.h"

#define DWIDTH 32

typedef ap_axiu<DWIDTH, 1, 1, 1> trans_pkt;

extern "C"{
    void krnl_stream_vmult(hls::stream<trans_pkt> &A,
                           hls::stream<trans_pkt> &B) {
#pragma HLS INTERFACE mode=axis port=A
#pragma HLS INTERFACE mode=axis port=B
#pragma HLS INTERFACE mode=s_axilite port=return bundle=control
    bool eos = false;

    vmult: do {
#pragma HLS PIPELINE II=1
        trans_pkt t2 = A.read();

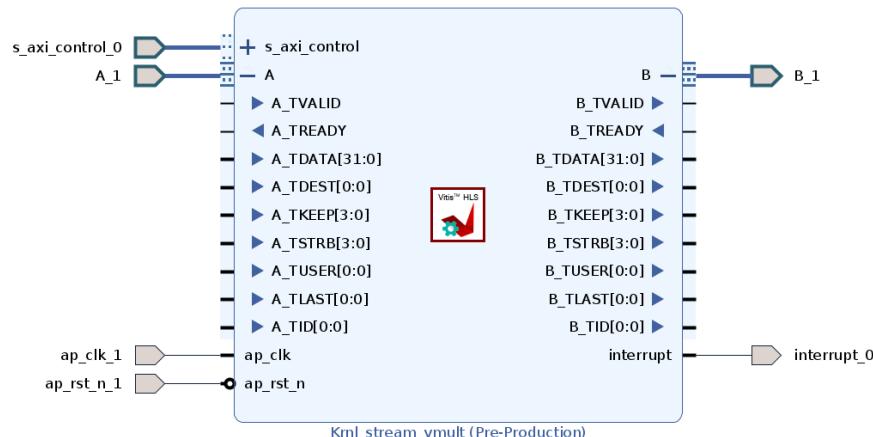
        // Packet for Output
        trans_pkt t_out;

        // Reading data from input packet
        ap_uint<DWIDTH> in2 = t2.data;
        ap_uint<DWIDTH> tmpOut = in2 * 5;

        // Setting data and configuration to output packet
        t_out.data = tmpOut;
        t_out.last = t2.last;
        t_out.keep = -1; //Enabling all bytes
        // Writing packet to output stream
        B.write(t_out);
        if (t2.last) {
            eos = true;
        }
    } while (eos == false);
}
}
```

After synthesis, both the A and B arguments are implemented with data ports, the standard AXI4-Stream protocol ports, TVALID and TREADY and all of the optional ports described in the struct.

Figure 47: AXI4-Stream Interfaces with Side-Channels



Coding Style for Array to Stream

While arrays can be converted to streams, it can often lead to coding and synthesis issues as arrays can be accessed in random order while a stream requires a sequential access pattern where every element is read in order. To avoid such issues, any time a streaming interface is required, it is highly recommended to use the `hls::stream` object as described in [Using HLS Streams](#). Usage of this construct will enforce streaming semantics in the source code.

However, to convert an array to a stream you should perform all the operations on temp variables. Read the input stream, process the temp variable, and write the output stream, as shown in the example below. This approach lets you preserve the sequential reading and writing of the stream of data, rather than attempting multiple or random reads or writes.

```

struct A {
    short varA;
    int varB;
};

void dut(A in[N], A out[N], bool flag) {
#pragma HLS interface mode=axis port=in,out
    for (unsigned i=0; i<N; i++) {
        A tmp = in[i];
        if (flag)
            tmp.varB = tmp.varA + 5;
        out[i] = tmp;
    }
}

```

If this coding style is not adhered to, it will lead to functional failures of the stream processing.

The recommended method is to define the arguments as `hls::stream` objects as shown below:

```
void dut(hls::stream<A> &in, hls::stream<A> &out, bool flag) {  
    #pragma HLS interface mode=axis port=in,out  
  
    for (unsigned i=0; i<N; i++) {  
        A tmp = in.read();  
        if (flag)  
            tmp.varB = tmp.varA + 5;  
        out.write(tmp);  
    }  
}
```

Port-Level Protocols for Vivado IP Flow



IMPORTANT! The port-level protocols described here can be used in the Vivado IP flow as explained in [Interfaces for Vivado IP Flow](#). These protocols are not supported in the Vitis kernel flow.

By default input pointers and pass-by-value arguments are implemented as simple wire ports with no associated handshaking signal. For example, in the `vadd` function discussed in [Interfaces for Vivado IP Flow](#), the input ports are implemented without an I/O protocol, only a data port. If the port has no I/O protocol, (by default or by design) the input data must be held stable until it is read.

By default output pointers are implemented with an associated output valid signal to indicate when the output data is valid. In the `vadd` function example, the output port is implemented with an associated output valid port (`out_r_o_ap_vld`) which indicates when the data on the port is valid and can be read. If there is no I/O protocol associated with the output port, it is difficult to know when to read the data.



TIP: It is always a good idea to use an I/O protocol on an output.

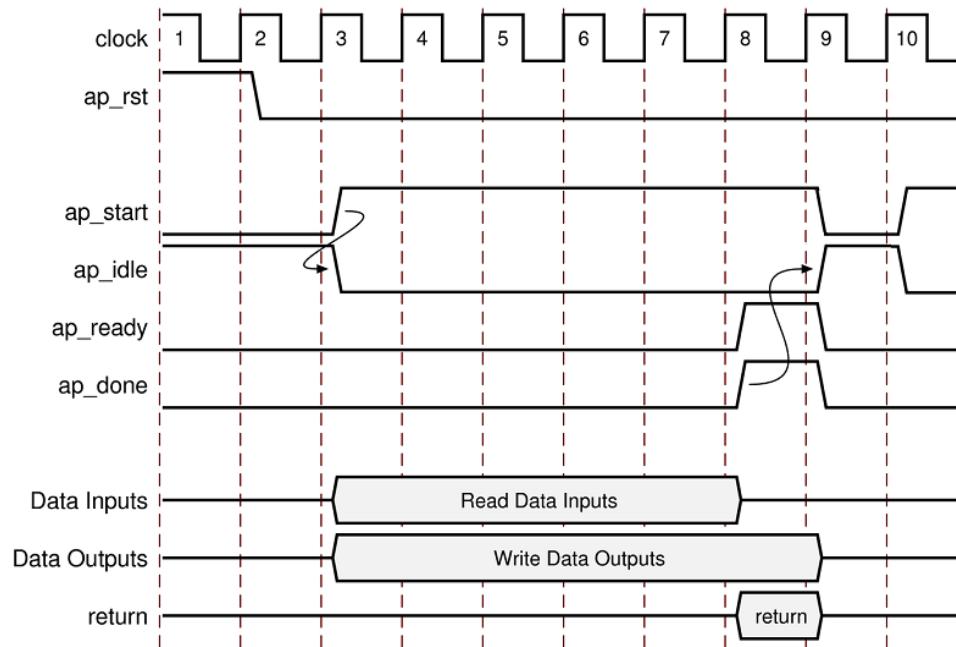
Function arguments which are both read from and written to are split into separate input and output ports. In the `vadd` function example, the `out_r` argument is implemented as both an input port `out_r_i`, and an output port `out_r_o` with associated I/O protocol port `out_r_o_ap_vld`.

If the function has a return value, an output port `ap_return` is implemented to provide the return value. When the RTL design completes one transaction, this is equivalent to one execution of the C/C++ function, the block-level protocols indicate the function is complete with the `ap_done` signal. This also indicates the data on port `ap_return` is valid and can be read.

Note: The return value of the top-level function cannot be a pointer.

For the example code shown the timing behavior is shown in the following figure (assuming that the target technology and clock frequency allow a single addition per clock cycle).

Figure 48: RTL Port Timing with Default Synthesis



- The design starts when `ap_start` is asserted High.
- The `ap_idle` signal is asserted Low to indicate the design is operating.
- The input data is read at any clock after the first cycle. Vitis HLS schedules when the reads occur. The `ap_ready` signal is asserted High when all inputs have been read.
- When output `sum` is calculated, the associated output handshake (`sum_o_ap_vld`) indicates that the data is valid.
- When the function completes, `ap_done` is asserted. This also indicates that the data on `ap_return` is valid.
- Port `ap_idle` is asserted High to indicate that the design is waiting start again.

Port-Level I/O: No Protocol

The `ap_none` specifies that no I/O protocol be added to the port. When this is specified the argument is implemented as a data port with no other associated signals. The `ap_none` mode is the default for scalar inputs.

ap_none

The `ap_none` port-level I/O protocol is the simplest interface type and has no other signals associated with it. Neither the input nor output data signals have associated control ports that indicate when data is read or written. The only ports in the RTL design are those specified in the source code.

An `ap_none` interface does not require additional hardware overhead. However, the `ap_none` interface does requires the following:

- Producer blocks to do one of the following:
 - Provide data to the input port at the correct time, typically before the design starts.
 - Hold data for the length of a transaction until the design raises the `ap_ready` signal.
- Consumer blocks to read output ports when the design is done, and before it is started again.

Note: The `ap_none` interface cannot be used with array arguments.

Port-Level I/O: Wire Handshakes

Interface mode `ap_hs` includes a two-way handshake signal with the data port. The handshake is an industry standard valid and acknowledge handshake. Mode `ap_vld` is the same but only has a valid port and `ap_ack` only has a acknowledge port.

Mode `ap_ovld` is for use with in-out arguments. When the in-out is split into separate input and output ports, mode `ap_none` is applied to the input port and `ap_vld` applied to the output port. This is the default for pointer arguments that are both read and written.

The `ap_hs` mode can be applied to arrays that are read or written in sequential order. If Vitis HLS can determine the read or write accesses are not sequential, it will halt synthesis with an error. If the access order cannot be determined, Vitis HLS will issue a warning.

ap_hs (ap_ack, ap_vld, and ap_ovld)

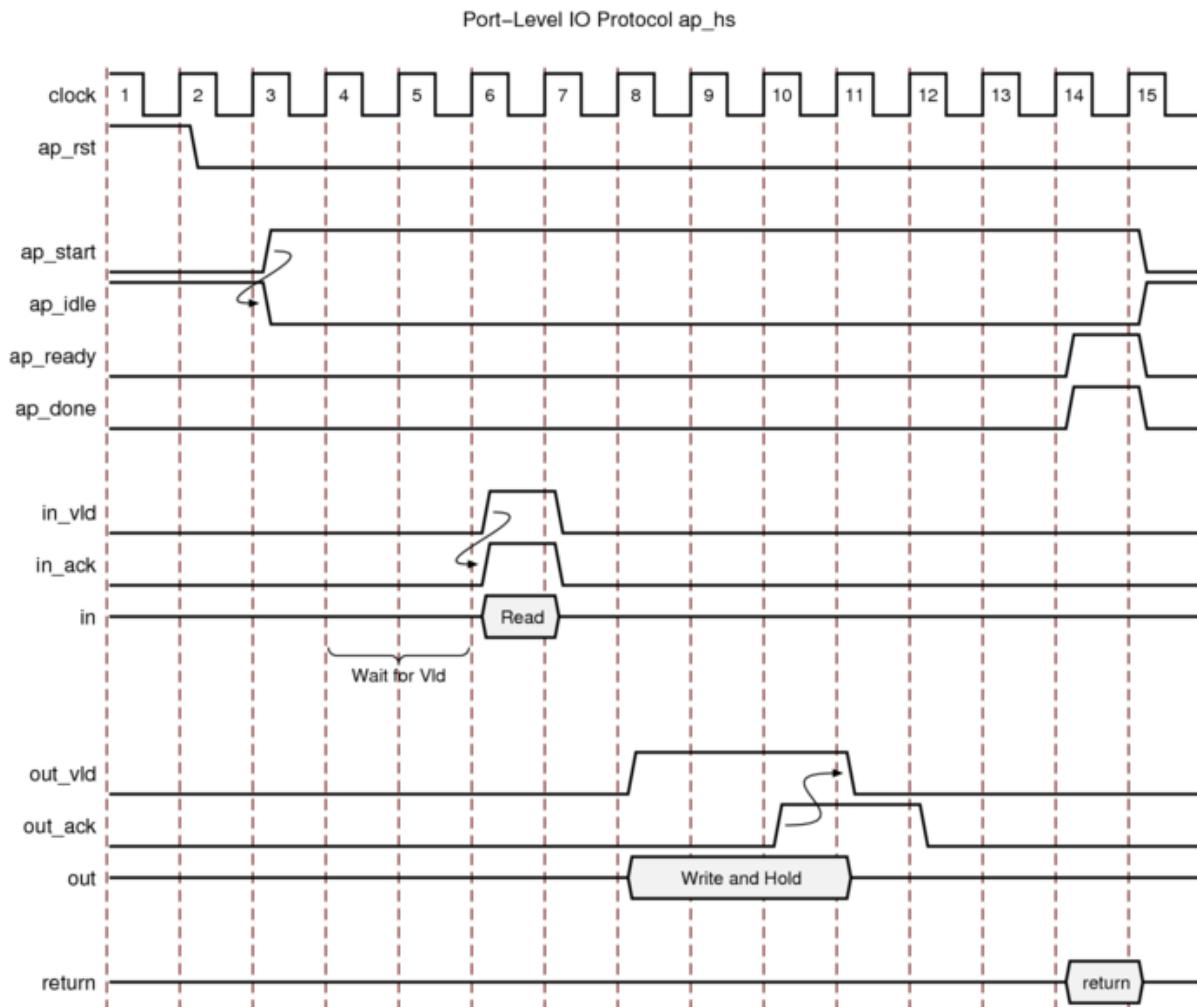
The `ap_hs` port-level I/O protocol provides the greatest flexibility in the development process, allowing both bottom-up and top-down design flows. Two-way handshakes safely perform all intra-block communication, and manual intervention or assumptions are not required for correct operation. The `ap_hs` port-level I/O protocol provides the following signals:

- Data port
- Valid signal to indicate when the data signal is valid and can be read
- Acknowledge signal to indicate when the data has been read

The following figure shows how an `ap_hs` interface behaves for both an input and output port. In this example, the input port is named `in`, and the output port is named `out`.

Note: The control signals names are based on the original port name. For example, the `valid` port for data input `in` is named `in_vld`.

Figure 49: Behavior of ap_hs Interface



For inputs, the following occurs:

- After start is applied, the block begins normal operation.
- If the design is ready for input data but the input `valid` is Low, the design stalls and waits for the input `valid` to be asserted to indicate a new input value is present.

Note: The preceding figure shows this behavior. In this example, the design is ready to read data input `in` on clock cycle 4 and stalls waiting for the input `valid` before reading the data.

- When the input `valid` is asserted High, an output acknowledge is asserted High to indicate the data was read.

For outputs, the following occurs:

- After start is applied, the block begins normal operation.

- When an output port is written to, its associated output `valid` signal is simultaneously asserted to indicate valid data is present on the port.
- If the associated input acknowledge is Low, the design stalls and waits for the input acknowledge to be asserted.
- When the input acknowledge is asserted, indicating the data has been read, the output `valid` is deasserted on the next clock edge.

ap_ack

The `ap_ack` port-level I/O protocol is a subset of the `ap_hs` interface type. The `ap_ack` port-level I/O protocol provides the following signals:

- Data port
- Acknowledge signal to indicate when data is consumed
 - For input arguments, the design generates an output acknowledge that is active-High in the cycle the input is read.
 - For output arguments, Vitis HLS implements an input acknowledge port to confirm the output was read.

Note: After a write operation, the design stalls and waits until the input acknowledge is asserted High, which indicates the output was read by a consumer block. However, there is no associated output port to indicate when the data can be consumed.



CAUTION! You cannot use C/RTL co-simulation to verify designs that use `ap_ack` on an output port.

ap_vld

The `ap_vld` is a subset of the `ap_hs` interface type. The `ap_vld` port-level I/O protocol provides the following signals:

- Data port
- Valid signal to indicate when the data signal is valid and can be read
 - For input arguments, the design reads the data port as soon as the `valid` is active. Even if the design is not ready to read new data, the design samples the data port and holds the data internally until needed.
 - For output arguments, Vitis HLS implements an output `valid` port to indicate when the data on the output port is valid.

ap_ovld

The `ap_ovld` is a subset of the `ap_hs` interface type. The `ap_ovld` port-level I/O protocol provides the following signals:

- Data port

- Valid signal to indicate when the data signal is valid and can be read
 - For input arguments and the input half of inout arguments, the design defaults to type ap_none.
 - For output arguments and the output half of inout arguments, the design implements type ap_vld.

Port-Level I/O: Memory Interface Protocol

Array arguments are implemented by default as an ap_memory interface. This is a standard block RAM interface with data, address, chip-enable, and write-enable ports.

An ap_memory interface can be implemented as a single-port or dual-port interface. If Vitis HLS can determine that using a dual-port interface will reduce the initial interval, it will automatically implement a dual-port interface. The BIND_STORAGE pragma or directive is used to specify the memory resource and if this directive is specified on the array with a single-port block RAM, a single-port interface will be implemented. Conversely, if a dual-port interface is specified using the BIND_STORAGE pragma and Vitis HLS determines this interface provides no benefit it will automatically implement a single-port interface.

If the array is accessed in a sequential manner an ap_fifo interface can be used. As with the ap_hs interface, Vitis HLS will halt if it determines the data access is not sequential, report a warning if it cannot determine if the access is sequential or issue no message if it determines the access is sequential. The ap_fifo interface can only be used for reading or writing, not both.

ap_memory, bram

The ap_memory and bram interface port-level I/O protocols are used to implement array arguments. This type of port-level I/O protocol can communicate with memory elements (for example, RAMs and ROMs) when the implementation requires random accesses to the memory address locations.

Note: If you only need sequential access to the memory element, use the ap_fifo interface instead. The ap_fifo interface reduces the hardware overhead, because address generation is not performed.

The ap_memory and bram interface port-level I/O protocols are identical. The only difference is the way Vivado IP integrator shows the blocks:

- The ap_memory interface appears as discrete ports.
- The bram interface appears as a single, grouped port. In IP integrator, you can use a single connection to create connections to all ports.

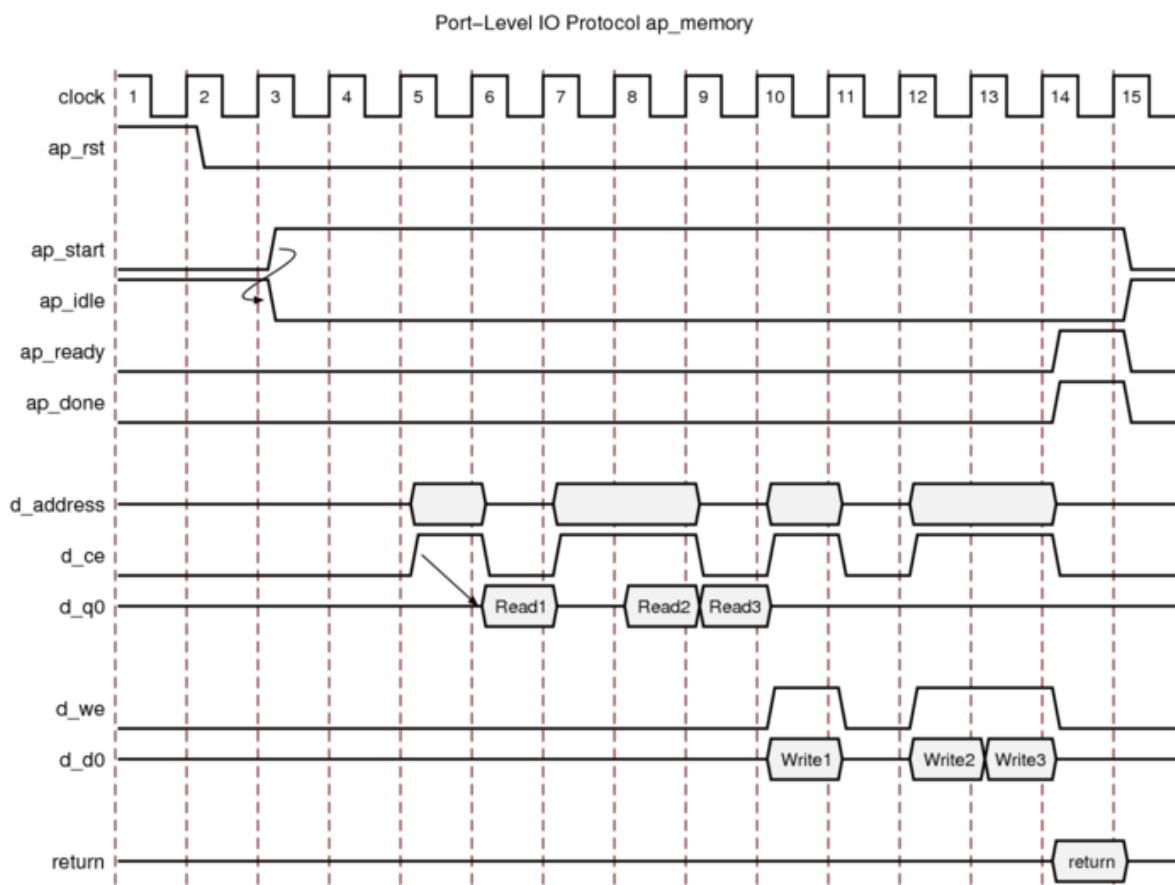
When using an ap_memory interface, specify the array targets using the BIND_STORAGE pragma. If no target is specified for the arrays, Vitis HLS determines whether to use a single or dual-port RAM interface.



TIP: Before running synthesis, ensure array arguments are targeted to the correct memory type using the `BIND_STORAGE` pragma. Re-synthesizing with corrected memories can result in a different schedule and RTL.

The following figure shows an array named `d` specified as a single-port block RAM. The port names are based on the C/C++ function argument. For example, if the C/C++ argument is `d`, the chip-enable is `d_ce`, and the input data is `d_d0` based on the `output/q` port of the BRAM.

Figure 50: Behavior of ap_memory Interface



After reset, the following occurs:

- After start is applied, the block begins normal operation.
- Reads are performed by applying an address on the output address ports while asserting the output signal `d_ce`.

Note: For a default block RAM, the design expects the input data `d_q0` to be available in the next clock cycle. You can use the `BIND_STORAGE` pragma to indicate the RAM has a longer read latency.

- Write operations are performed by asserting output ports `d_ce` and `d_we` while simultaneously applying the address and output data `d_d0`.

ap_fifo

When an output port is written to, its associated output `valid` signal interface is the most hardware-efficient approach when the design requires access to a memory element and the access is always performed in a sequential manner, that is, no random access is required. The `ap_fifo` port-level I/O protocol supports the following:

- Allows the port to be connected to a FIFO
- Enables complete, two-way `empty`-`full` communication
- Works for arrays, pointers, and pass-by-reference argument types

Note: Functions that can use an `ap_fifo` interface often use pointers and might access the same variable multiple times. To understand the importance of the `volatile` qualifier when using this coding style, see [Multi-Access Pointers on the Interface](#).

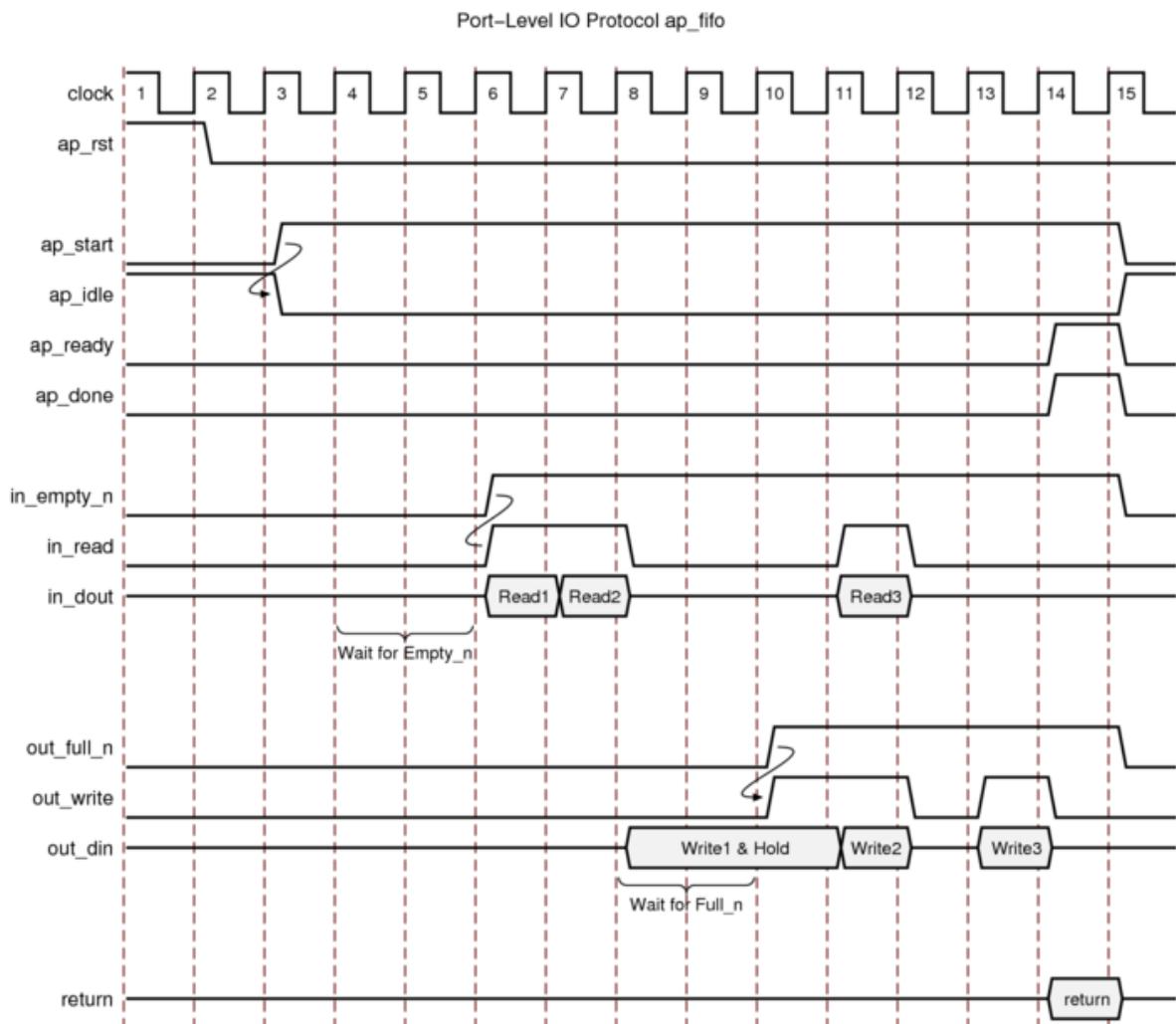
In the following example, `in1` is a pointer that accesses the current address, then two addresses above the current address, and finally one address below.

```
void foo(int* in1, ...) {
    int data1, data2, data3;
    ...
    data1= *in1;
    data2= *(in1+2);
    data3= *(in1-1);
    ...
}
```

If `in1` is specified as an `ap_fifo` interface, Vitis HLS checks the accesses, determines the accesses are not in sequential order, issues an error, and halts. To read from non-sequential address locations, use an `ap_memory` or `bram` interface.

You cannot specify an `ap_fifo` interface on an argument that is both read from and written to. You can only specify an `ap_fifo` interface on an input or an output argument. A design with input argument `in` and output argument `out` specified as `ap_fifo` interfaces behaves as shown in the following figure.

Figure 51: Behavior of ap_fifo Interface



For inputs, the following occurs:

- After ap_start is applied, the block begins normal operation.
- If the input port is ready to be read but the FIFO is empty as indicated by input port `in_empty_n` Low, the design stalls and waits for data to become available.
- When the FIFO contains data as indicated by input port `in_empty_n` High, an output acknowledge `in_read` is asserted High to indicate the data was read in this cycle.

For outputs, the following occurs:

- After start is applied, the block begins normal operation.
- If an output port is ready to be written to but the FIFO is full as indicated by `out_full_n` Low, the data is placed on the output port but the design stalls and waits for the space to become available in the FIFO.

- When space becomes available in the FIFO as indicated by `out_full_n` High, the output acknowledge signal `out_write` is asserted to indicate the output data is valid.
- If the top-level function or the top-level loop is pipelined using the `-rewind` option, Vitis HLS creates an additional output port with the suffix `_lwr`. When the last write to the FIFO interface completes, the `_lwr` port goes active-High.

Programming Model for Multi-Port Access in HBM

HBM provides high bandwidth if arrays are split in different banks/pseudo-channels in the design. This is a common practice in partitioning an array into different memory regions in high-performance computing. The host allocates a single buffer, which will be spread across the pseudo-channels.

Vitis HLS would consider different pointers to be independent channels, and removes any dependency analysis. But the host allocates a single buffer for both pointers, and this lets the tool maintain the dependency analysis through `pragma HLS ALIAS`. The `ALIAS` pragma informs data dependence analysis about the pointer distance. Refer to the `ALIAS` pragma for more information.

The kernel `arg0` is allocated in bank0 and kernel `arg1` is allocated in bank1. The pointer distance should be specified in the `distance` option of the `ALIAS` pragma as shown below:

```
//Assume that the host code looks like this:  
int *buf = clCreateBuffer(ctx, CL_MEM_READ_ONLY, 2*bank_size, ...);  
clSetKernelArg(kernel, 0, 0x20000000, buf); // bank0  
clSetKernelArg(kernel, 1, 0x20000000, buf+bank_size); // bank1  
  
//The ALIAS pragma informs data dependence analysis about the pointer  
//distance  
void kernel(int *bank0, int *bank1, ...)  
{  
#pragma HLS alias ports=bank0, bank1 distance=bank_size
```

The `ALIAS` pragma can be specified using one of the following forms:

- Constant distance:

```
#pragma HLS alias ports=arr0,arr1,arr2,arr3 distance=1024
```

- Variable distance:

```
#pragma HLS alias ports=arr0,arr1,arr2,arr3 offset=0,512,1024,2048
```

Constraints:

- The depths of all the ports in the interface pragma must be the same
- All ports must be assigned to different bundles, bound to different HBM controllers
- The number of ports specified in the second form must be the same as the number of offsets specified, one offset per port. `#pragma HLS interface offset=off` is not supported

- Each port can only be used in one ALIAS pragma

Managing Interfaces with SSI Technology Devices

Certain Xilinx devices use stacked silicon interconnect (SSI) technology. In these devices, the total available resources are divided over multiple super logic regions (SLRs). The connections between SLRs use super long line (SSL) routes. SSL routes incur delays costs that are typically greater than standard FPGA routing. To ensure designs operate at maximum performance, use the following guidelines:

- Register all signals that cross between SLRs at both the SLR output and SLR input.
- You do not need to register a signal if it enters or exits an SLR via an I/O buffer.
- Ensure that the logic created by Vitis HLS fits within a single SLR.

Note: When you select an SSI technology device as the target technology, the utilization report includes details on both the SLR usage and the total device usage.

If the logic is contained within a single SLR device, Vitis HLS provides a `-register_all_io` option to the `config_rtl` command. If the option is enabled, all inputs and outputs are registered. If disabled, none of the inputs or outputs are registered.

Vitis HLS Memory Layout Model

The Vitis application acceleration development flow provides a framework for developing and delivering FPGA accelerated applications using standard programming languages for both software and hardware components. The software component, or host program, is developed using C/C++ to run on x86 or embedded processors, with OpenCL/ Native XRT API calls to manage run time interactions with the accelerator. The hardware component, or kernel (that runs on the actual FPGA card/platform), can be developed using C/C++, OpenCL C, or RTL. The Vitis software platform promotes concurrent development and test of the hardware and software elements of a heterogeneous application. Due to this, the software program that runs on the host computer needs to communicate with the acceleration kernel that runs on the FPGA hardware model using well-defined interfaces and protocols. As a result, it becomes important to define the exact memory model that is used so that the data that is being read/written can be correctly processed. The memory model defines the way data is arranged and accessed in computer memory. It consists of two separate but related issues: data alignment and data structure padding. In addition, the Vitis HLS compiler supports the specification of special attributes (and pragmas) to change the default data alignment and data structure padding rules.

Data Alignment

Software programmers are conditioned to think of memory as a simple array of bytes and the basic data types are composed of one or more blocks of memory. However, the computer's processor does not read from and write to memory in single byte-sized chunks. Instead, today's modern CPUs access memory in 2, 4, 8, 16, or even 32-byte chunks at a time - although 32 bit and 64 bit instruction set architecture (ISA) architectures are the most common. Due to how the memory is organized in your system, the addresses of these chunks should be multiples of their sizes. If an address satisfies this requirement, then it is said to be *aligned*. The difference between how high-level programmers think of memory and how modern processors actually work with memory is pretty important in terms of application correctness and performance. For example, if you don't understand the address alignment issues in your software, the following situations are all possible:

- your software will run slower
- your application will lock up/hang
- your operating system can crash
- your software will silently fail, yielding incorrect results

The C++ language provides a set of fundamental types of various sizes. To make manipulating variables of these types fast, the generated object code will try to use CPU instructions that read/write the whole data type at once. This in turn means that the variables of these types should be placed in memory in a way that makes their addresses suitably aligned. As a result, besides size, each fundamental type has another property: its alignment requirement. It may seem that the fundamental type's alignment is the same as its size. This is not generally the case since the most suitable CPU instruction for a particular type may only be able to access a part of its data at a time. For example, a 32-bit x86 GNU/Linux machine may only be able to read at most 4 bytes at a time so a 64-bit `long long` type will have a size of 8 and an alignment of 4. The following table shows the size and alignment (in bytes) for the basic native data types in C/C++ for both 32-bit and 64-bit x86-64 GNU/Linux machines.

Table 9: Data Types

Type	32-bit x86 GNU/Linux		64-bit x86 GNU/Linux	
	Size	Alignment	Size	Alignment
bool	1	1	1	1
char	1	1	1	1
short int	2	2	2	2
int	4	4	4	4
long int	4	4	8	8
long long int	8	4	8	8
float	4	4	4	4
double	8	4	8	8

Table 9: Data Types (cont'd)

Type	32-bit x86 GNU/Linux		64-bit x86 GNU/Linux	
	Size	Alignment	Size	Alignment
long double	12	4	16	16
void*	4	4	8	8

Given the above arrangement, why does a programmer need to change the alignment? There are several reasons but the main reason will be to trade-off between memory requirements and performance. When you are sending data back and forth from the host computer and the accelerator, every byte that is transmitted has a cost. Fortunately, the GCC C/C++ compiler provides the language extension `__attribute__((aligned(X)))` in order to change the default alignment for the variable, structures/classes, or a structure field, measured in bytes. For example, the following declaration causes the compiler to allocate the global variable `x` on a 16-byte boundary.

```
int x __attribute__((aligned(16))) = 0;
```

The `__attribute__((aligned(X)))` does not change the sizes of variables it is applied to, but may change the memory layout of structures by inserting padding between elements of the struct. As a result, the size of the structure will change. If you don't specify the alignment factor in an aligned attribute, the compiler automatically sets the alignment for the declared variable or field to the largest alignment used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way. The `aligned` attribute can only increase the alignment and can never decrease it. The C++ function `offsetof` can be used to determine the alignment of each member element in a structure.

Data Structure Padding

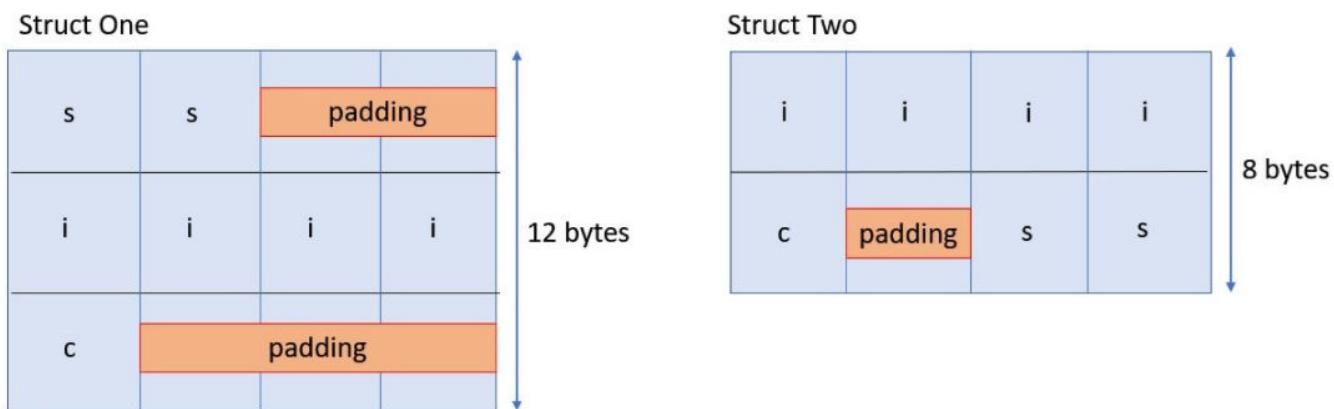
As shown in the table in [Data Alignment](#), the native data types have a well-defined alignment structure but what about user-defined data types? The C++ compiler also needs to make sure that all the member variables in a struct or class are properly aligned. For this, the compiler may insert *padding* bytes between member variables. In addition, to make sure that each element in an array of a user-defined type is aligned, the compiler may add some extra padding after the last data member. Consider the following example:

<pre>struct One { short int s; int i; char c; }</pre>	<pre>struct Two { int i; char c; short int s; }</pre>
---	---

The GCC compiler always assumes that an instance of `struct One` will start at an address aligned to the most strict alignment requirement of all of the struct's members, which is `int` in this case. This is actually how the alignment requirements of user-defined types are calculated. Assuming the memory are on x86-64 alignment with `short int` having the alignment of 2 and `int` having an alignment of 4, to make the `i` data member of `struct One` suitably aligned, the compiler needs to insert two extra bytes of padding between `s` and `i` to create alignment, as shown in the figure below. Similarly, to align data member `c`, the compiler needs to insert three bytes after `c`.

In the case of `struct One`, the compiler will infer a total size of 12 bytes based on the arrangement of the elements of the struct. However, if the elements of the struct are reordered (as shown in `struct Two`), the compiler is now able to infer the smaller size of 8 bytes.

Figure 52: Padding of Structs



By default, the C/C++ compiler will lay out members of a struct in the order in which they are declared, with possible padding bytes inserted between members, or after the last member, to ensure that each member is aligned properly. However, the GCC C/C++ compiler provides a language extension, `__attribute__((packed))` which tells the compiler not to insert padding but rather allow the struct members to be misaligned. For example, if the system normally requires all `int` objects to have 4-byte alignment, the usage of `__attribute__((packed))` can cause `int` struct members to be allocated at odd offsets.

Usage of `__attribute__((packed))` must be carefully considered because accessing unaligned memory can cause the compiler to insert code to read the memory byte by byte instead of reading multiple chunks of memory at one time.

Vitis HLS Alignment Rules and Semantics

Given the behavior of the GCC compiler described previously, this section will detail how Vitis HLS uses aligned and packed attributes to create efficient hardware. First, you need to understand the [Aggregate](#) and [Disaggregate](#) features in Vitis HLS. Structures or class objects in the code, for instance internal and global variables, are disaggregated by default. Disaggregation implies that the structure/class is decomposed into separate objects, one for each struct/class member. The number and type of elements created are determined by the contents of the struct itself. Arrays of structs are implemented as multiple arrays, with a separate array for each member of the struct.

However, structs used as arguments to the top-level function are kept aggregated by default. Aggregation implies that all the elements of a struct are collected into a single wide vector. This allows all members of the struct to be read and written simultaneously. The member elements of the struct are placed into the vector in the order in which they appear in the C/C++ code: the first element of the struct is aligned on the LSB of the vector and the final element of the struct is aligned with the MSB of the vector. Any arrays in the struct are partitioned into individual array elements and placed in the vector from lowest to the highest order.

Table 11: Interface Arguments and Internal Variables

	Behavior without AGGREGATE pragma		Behavior with AGGREGATE pragma (compact=auto or not specified)	
	Interface Argument	Internal Variable	Interface Argument	Internal variable
AXI protocol interface (<code>m_axi/s_axilite/axis</code>)	aggregate compact= none	N/A	compact= none	N/A
Struct/Class containing <code>hls::stream</code> object	Automatically disaggregate the struct/class	Automatically disaggregate the struct/class	N/A	N/A
other interface protocols	aggregate compact= bit	Automatically disaggregated	compact= bit	compact= bit

The goal of the default aggregation behavior in Vitis HLS is to use an x86_64-gnu-linux memory layout at the top level hardware interface while optimizing the internal hardware for better quality of results (QoR). The above table shows the default behavior of Vitis HLS. Two modes are shown in the table: the default mode where the AGGREGATE pragma is not specified by the user, and the case where the AGGREGATE pragma is specified by the user.

In the case of AXI4 interfaces (`m_axi/s_axilite/axis`), a structure is padded by default according to the order of elements of the struct as explained in [Data Structure Padding](#). This aggregates the structure to a size that is the closest power of 2, and so some padding may be applied in this case. This in effect infers the `compact=none` option on the AGGREGATE pragma.

In the case of other interface protocols, the struct is packed at the bit-level, so the aggregated vector is only the size of the various elements of the struct. This in effect infers the `compact=bit` option on the `AGGREGATE` pragma.

The only exception to the above rules is when using `hls::stream` in the interface indirectly (i.e. the `hls::stream` object is specified inside a struct/class that is then used as the type of an interface port). The struct containing the `hls::stream` object is always disaggregated into its individual member elements.

Examples of Aggregation

Aggregate Memory Mapped Interface

This is an example of the `AGGREGATE` pragma or directive for an `m_axi` interface. The following is the [aggregation_of_m_axi_ports](#) example available on GitHub.

```
struct A {
    char foo;      // 1 byte
    short bar;     // 2 bytes
};

int dut(A* arr) {
#pragma HLS interface m_axi port=arr depth=10
#pragma HLS aggregate variable=arr compact=auto
    int sum = 0;
    for (unsigned i=0; i<10; i++) {
        auto tmp = arr[i];
        sum += tmp.foo + tmp.bar;
    }
    return sum;
}
```

For the above example, the size of the `m_axi` interface port `arr` is 3 bytes (or 24 bits) but due to the `AGGREGATE compact=auto` pragma, the size of the port will be aligned to 4 bytes (or 32 bits) as this is the closest power of 2. Vitis HLS will issue the following message in the log file:

```
INFO: [HLS 214-241] Aggregating maxi variable 'arr' with compact=none mode
in 32-bits (example.cpp:19:0)
```



TIP: The message above is only issued if the `AGGREGATE` pragma is specified. But even without the pragma, the tool will automatically aggregate and pad the interface port `arr` to 4 bytes as the default behavior for an AXI interface port.

Aggregate Structs on the Interface

This is an example of the AGGREGATE pragma or directive for an `ap_fifo` interface. The following is the [aggregation_of_struct](#) example available on GitHub.

```
struct A {
    int myArr[3];          // 4 bytes per element (12 bytes total)
    ap_int<23> length;    // 23 bits
};

int dut(A arr[N]) {
#pragma HLS interface ap_fifo port=arr
#pragma HLS aggregate variable=arr compact=auto
    int sum = 0;
    for (unsigned i=0; i<10; i++) {
        auto tmp = arr[i];
        sum += tmp.myArr[0] + tmp.myArr[1] + tmp.myArr[2] + tmp.length;
    }
    return sum;
}
```

For `ap_fifo` interface, the struct will packed at the bit-level with or without aggregate pragma.

In the above example, the AGGREGATE pragma will create a port of size 119 bits for port `arr`. The array `myArr` will take 12 bytes (or 96 bits) and the element `length` will take 23 bits for a total of 119 bits. Vitis HLS will issue the following message in the log file:

```
INFO: [HLS 214-241] Aggregating fifo (array-to-stream) variable 'arr' with
compact=bit mode
    in 119-bits (example.cpp:19:0)
```

Aggregate Nested Struct Port

This is an example of the AGGREGATE pragma or directive in the Vivado IP flow. The following is the [aggregation_of_nested_structs](#) example available on GitHub.

```
#define N 8

struct T {
    int m;      // 4 bytes
    int n;      // 4 bytes
    bool o;     // 1 byte
};

struct S {
    int p;      // 4 bytes
    T q;       // 9 bytes
};
void top(S a[N], S b[N], S c[N]) {
#pragma HLS interface bram port=c
#pragma HLS interface ap_memory port=a
#pragma HLS aggregate variable=a compact=byte
#pragma HLS aggregate variable=b compact=bit
#pragma HLS aggregate variable=c compact=byte
    for (int i=0; i<N; i++) {
        c[i].q.m = a[i].q.m + b[i].q.m;
```

```
    c[i].q.n = a[i].q.n - b[i].q.n;
    c[i].q.o = a[i].q.o || b[i].q.o;
    c[i].p = a[i].q.n;
}
}
```

In the above example, the aggregation algorithm will create a port of size 104 bits for ports `a`, and `c` as the `compact=byte` option was specified in the aggregate pragma but the `compact=bit` default option is used for port `b` and its packed size will be 97 bits. The nested structures `S` and `T` are aggregated to encompass three 32 bit member variables (`p`, `m`, and `n`) and one bit/byte member variable (`o`).



TIP: This example uses the Vivado IP flow to illustrate the aggregation behavior. In the Vitis kernel flow, port `b` will be automatically inferred as an `m_axi` port and will not allow the `compact=bit` setting.

Vitis HLS will issue the following messages in the log file:

```
INFO: [HLS 214-241] Aggregating bram variable 'b' with compact=bit mode in
97-bits (example.cpp:19:0)
INFO: [HLS 214-241] Aggregating bram variable 'a' with compact=byte mode in
104-bits (example.cpp:19:0)
INFO: [HLS 214-241] Aggregating bram variable 'c' with compact=byte mode in
104-bits (example.cpp:19:0)
```

Examples of Disaggregation

Disaggregate AXIS Interface

This is an example of the `DISAGGREGATE` pragma or directive for an `axis` interface. The following is the [disaggregation_of_axis_port](#) example available on GitHub.

Table 12: Disaggregated Struct on AXIS Interface

HLS Source Code	Synthesized IP Module
<pre>#define N 10 struct A { char c; int i; }; void dut(A in[N], A out[N]) { #pragma HLS interface axis port=in #pragma HLS interface axis port=out #pragma HLS disaggregate variable=in #pragma HLS disaggregate variable=out int sum = 0; for (unsigned i=0; i<N; i++) { out[i].c = in[i].c; out[i].i = in[i].i; } }</pre>	<pre>module dut (ap_local_block, ap_local_deadlock, ap_clk, ap_rst_n, ap_start, ap_done, ap_idle, ap_ready, in_c_TVALID, in_i_TVALID, out_c_TREADY, out_i_TREADY, in_c_TDATA, in_c_TREADY, in_i_TDATA, in_i_TREADY, out_c_TDATA, out_c_TVALID, out_i_TDATA, out_i_TVALID);</pre>

In the above disaggregation example, the struct arguments `in` and `out` are mapped to AXIS interfaces, and then disaggregated. This results in Vitis HLS creating two AXI streams for each argument: `in_c`, `in_i`, `out_c` and `out_i`. Each member of the struct `A` becomes a separate stream.

The RTL interface of the generated module is shown on the right above where the member elements `c` and `i` are individual AXI stream ports, each with its own TVALID, TREADY and TDATA signals.

Vitis HLS will issue the following messages in the log file:

```
INFO: [HLS 214-210] Disaggregating variable 'in' (example.cpp:19:0)
INFO: [HLS 214-210] Disaggregating variable 'out' (example.cpp:19:0)
```

Disaggregate HLS::STREAM

This is an example of the DISAGGREGATE pragma or directive when used with the `hls::stream` type.

Table 13: Disaggregated Struct of HLS::STREAM

HLS Source Code	Synthesized IP Module
<pre>#define N 1024 struct A { hls::stream<int> s_in; long arr[N]; }; long dut(struct A &d) { long sum = 0; while(!d.s_in.empty()) sum += d.s_in.read(); for (unsigned i=0; i<N; i++) sum += d.arr[i]; return sum; }</pre>	<pre>module dut (ap_local_block, ap_local_deadlock, ap_clk, ap_rst, ap_start, ap_done, ap_idle, ap_ready, d_s_in_dout, d_s_in_empty_n, d_s_in_read, d_arr_ce0, d_arr_q0, ap_return);</pre>

Using an `hls::stream` object inside a structure that is used in the interface will cause the struct port to be automatically disaggregated by the Vitis HLS compiler. As shown in the above example, the generated RTL interface will contain separate RTL ports for the `hls::stream` object `s_in` (named `d_s_in_*`) and separate RTL ports for the array `arr` (named `d_arr_*`).

Vitis HLS will issue the following messages in the log file:

```
INFO: [HLS 214-210] Disaggregating variable 'd'
INFO: [HLS 214-241] Aggregating fifo (hls::stream) variable 'd_s_in' with
compact-bit mode in 32-bits
```

Impact of Struct Size on Pipelining

The size of a struct used in a function interface can adversely impact pipelining of loops in that function that have access to the interface in the loop body. Consider the following code example which has two M_AXI interfaces:

```
struct A { /* Total size = 192 bits (32 x 6) or 24 bytes */
    int s_1;
    int s_2;
    int s_3;
    int s_4;
    int s_5;
    int s_6;
};

void read(A *a_in, A buf_out[NUM]) {
READ:
    for (int i = 0; i < NUM; i++)
    {
        buf_out[i] = a_in[i];
    }
}

void compute(A buf_in[NUM], A buf_out[NUM], int size) {
COMPUTE:
    for (int j = 0; j < NUM; j++)
    {
        buf_out[j].s_1 = buf_in[j].s_1 + size;
        buf_out[j].s_2 = buf_in[j].s_2;
        buf_out[j].s_3 = buf_in[j].s_3;
        buf_out[j].s_4 = buf_in[j].s_4;
        buf_out[j].s_5 = buf_in[j].s_5;
        buf_out[j].s_6 = buf_in[j].s_6 % 2;
    }
}

void write(A buf_in[NUM], A *a_out) {
WRITE:
    for (int k = 0; k < NUM; k++)
    {
        a_out[k] = buf_in[k];
    }
}

void dut(A *a_in, A *a_out, int size)
{
#pragma HLS INTERFACE m_axi port=a_in bundle=gmem0
#pragma HLS INTERFACE m_axi port=a_out bundle=gmem1
    A buffer_in[NUM];
    A buffer_out[NUM];

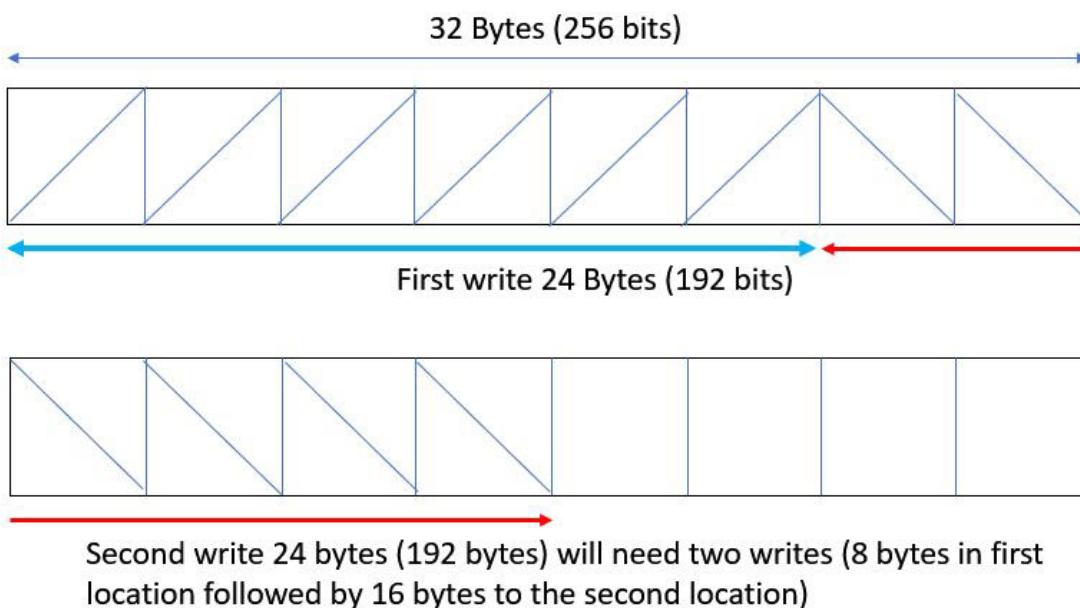
#pragma HLS dataflow
    read(a_in, buffer_in);
    compute(buffer_in, buffer_out, size);
    write(buffer_out, a_out);
}
```

In the above example, the size of struct A is 192 bits, which is not a power of 2. As stated earlier in the document, all AXI4 interfaces are by default sized to a power of 2. Vitis HLS will automatically size the two M_AXI interfaces (`a_in` and `a_out`) to be of size 256 - the closest power of 2 to the size of 192 bits (and report in the log file as shown below).

```
INFO: [HLS 214-241] Aggregating maxi variable 'a_out' with compact=none mode in 256-bits (example.cpp:49:0)
INFO: [HLS 214-241] Aggregating maxi variable 'a_in' with compact=none mode in 256-bits (example.cpp:49:0)
```

This will imply that when writing the struct data out, the first write will write 24 bytes to the first buffer in one cycle but the second write will have to write 8 bytes to the remaining 8 bytes in the first buffer and then write 16 bytes into a second buffer resulting in two writes - as shown in the figure below.

Figure 53: Misaligned Write Cycles



This will cause the II of the WRITE loop in function `write()` to have an II violation since it needs II=2 instead of II=1. Similar behavior will happen when reading and therefore the `read()` function will also have an II violation since it needs II=2. Vitis HLS will issue the following warning for the II violation in function `read()` and `write()`:

```
WARNING: [HLS 200-880] The II Violation in module 'read_r' (loop 'READ'): Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 1) between bus read operation ('gmem0_addr_read_1', example.cpp:23) on port 'gmem0' (example.cpp:23)
```

```
and bus read operation ('gmem0_addr_read', example.cpp:23) on port 'gmem0'
(example.cpp:23).
```

```
WARNING: [HLS 200-880] The II Violation in module 'write_Pipeline_WRITE'
(loop 'WRITE'):
Unable to enforce a carried dependence constraint (II = 1, distance = 1,
offset = 1)
between bus write operation ('gmem1_addr_write_ln44', example.cpp:44) on
port 'gmem1'
(example.cpp:44) and bus write operation ('gmem1_addr_write_ln44',
example.cpp:44) on
port 'gmem1' (example.cpp:44).
```

The way to fix such II issues is to pad struct A with 8 additional bytes such that you are always writing 256 bits (32 bytes) at a time or by using the other alternatives shown in the table below. This will allow the scheduler to schedule the reads/writes in the READ/WRITE loop with II=1.

Table 14: Struct Alignment

Code Block	Description
<pre>struct A { int s_1; int s_2; int s_3; int s_4; int s_5; int s_6; int pad_1; int pad_2; };</pre>	Defines the total size of the struct as 256 bits (32 x 8) or 32 bytes, by adding required padding elements.
<pre>struct A { int s_1; int s_2; int s_3; int s_4; int s_5; int s_6; } __attribute__((aligned(32)));</pre>	Uses the standard <code>__aligned__</code> attribute.
<pre>struct alignas(32) A { int s_1; int s_2; int s_3; int s_4; int s_5; int s_6; }</pre>	Uses the C++ standard <code>alignas</code> type specifier to specify custom alignment of variables and user defined types.

Execution Modes of HLS Designs

The execution mode of the HLS design refers to the way the design works as a block (or module) both with regard to itself and the functions within the block, or in relationship with other blocks modules, or outside software that addresses the block. These modes are determined by block control protocols assigned to the HLS design as described in [Block-Level Control Protocols](#), and by the internal structure of the HLS design as described in [Abstract Parallel Programming Model for HLS](#).

Execution modes of kernels include:

- **Overlap:** Lets the next execution of a new transaction begin before the current transaction is complete. Pipelined execution allows overlapping block runs to begin processing additional data as soon as the design is ready.
- **Sequential:** Requires the current transaction to complete before a new transaction can be started.
- **Auto-Restarting:** Auto-restart mode lets the HLS design automatically restart at the end of each execution. Auto-restarting is the approach for data-driven TLP designs, but can also be implemented in control-driven TLP designs as described in [Auto-Restarting Mode](#).

Block-Level Control Protocols

The execution mode of a Vitis kernel or Vivado IP is defined by the block-level control protocol and the structure of sub-functions within the HLS design. For control-driven TLP the `ap_ctrl_chain` and `ap_ctrl_hs` protocols support both sequential or pipelined execution. For data-driven TLP `ap_ctrl_none` is the required control protocol.

The `ap_ctrl_chain` control protocol is the default for the Vitis kernel flow as explained in [Interfaces for Vitis Kernel Flow](#). The `ap_ctrl_hs` block-level control protocol is the default for the Vivado IP flow as described in [Interfaces for Vivado IP Flow](#). However, you should use `ap_ctrl_chain` when chaining HLS designs together to better support pipelined execution.

You can specify the block-level control protocol on the function return using the [INTERFACE](#) pragma or directive. If the C/C++ code does not return a value, you can still specify the control protocol on the function return. If the C/C++ code uses a function return, Vitis HLS creates an output port `ap_return` for the return value.

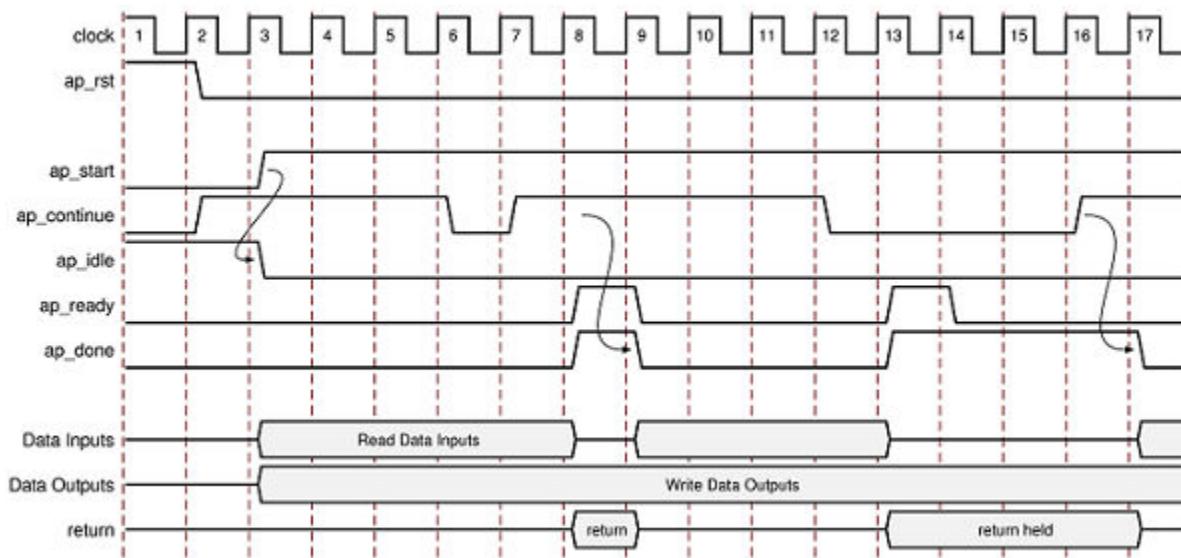


TIP: When the function return is specified as an AXI4-Lite interface (`s_axilite`) all the ports in the control protocol are bundled into the `s_axilite` interface. This is a common practice for software-controllable kernels or IP when an application or software driver is used to configure and control when the block starts and stops operation. This is a requirement of XRT and the Vitis kernel flow.

ap_ctrl_chain

The following figure shows the behavior of the block-level handshake signals created by the ap_ctrl_chain control protocol. In the following figure, the first transaction of the HLS design completes, and the second transaction starts immediately because ap_continue is High when ap_done is High. However, the design halts at the end of the second transaction until ap_continue is asserted High.

Figure 54: Behavior of ap_ctrl_chain Interface



The timing diagram displays the following behavior after reset occurs:

1. The block waits for ap_start to go High before it begins operation.
2. Output ap_idle goes Low immediately to indicate the design is no longer idle.
3. The ap_start signal must remain High until ap_ready goes High. Once ap_ready goes High:
 - If ap_start remains High the design will start the next transaction.
 - If ap_start is taken Low, the design will complete the current transaction and halt operation.
4. Data can be read on the input ports.
5. Data can be written to the output ports.

Note: The input and output ports can also specify a port-level I/O protocol that is independent of the control protocol. For details, see [Port-Level Protocols for Vivado IP Flow](#).

6. Output ap_done goes High when the block completes operation.

Note: If there is an `ap_return` port, the data on this port is valid when `ap_done` is High. Therefore, the `ap_done` signal also indicates when the data on output `ap_return` is valid.

7. The `ap_ctrl_chain` control protocol provides an active-High `ap_continue` signal that indicates when the downstream block that consumes the output data is ready for new data inputs. This allows the downstream block to provide back-pressure to prevent the flow of data.
 - If the `ap_continue` signal is High when `ap_done` is High, the design continues operating.
 - If the downstream block is not able to consume new data inputs, the `ap_continue` signal is Low. If the `ap_continue` signal is Low when `ap_done` is High, the design stops operating, the `ap_done` signal remains High waiting for `ap_continue` to go High.
8. When the design is ready to accept new inputs, the `ap_ready` signal goes High. The `ap_ready` port of a downstream block can directly drive the `ap_continue` port. Following is additional information about the `ap_ready` signal:
 - The `ap_ready` signal is inactive until the design starts operation.
 - In non-pipelined designs, the `ap_ready` signal is asserted at the same time as `ap_done`.
 - In pipelined designs, the `ap_ready` signal might go High at any cycle after `ap_start` is sampled High. This depends on how the design is pipelined.
 - If the `ap_start` signal is Low when `ap_ready` is High, the design executes until `ap_done` is High and then stops operation.
 - If the `ap_start` signal is High when `ap_ready` is High, the next transaction starts immediately, and the design continues to operate.
9. The `ap_idle` signal indicates when the design is idle and not operating. Following is additional information about the `ap_idle` signal:
 - If the `ap_start` signal is Low when `ap_ready` is High, the design stops operation, and the `ap_idle` signal goes High one cycle after `ap_done`.
 - If the `ap_start` signal is High when `ap_ready` is High, the design continues to operate, and the `ap_idle` signal remains Low.

ap_ctrl_hs

The `ap_ctrl_hs` control protocol has the same signals as `ap_ctrl_chain`, but sets the `ap_continue` signal to 1 so it remains high. This control protocol supports sequential and pipelined execution modes, but does not offer back-pressure from downstream design modules to control the flow of data.

ap_ctrl_none

`ap_ctrl_none` also has the same signals as `ap_ctrl_chain`, but the handshake signal ports (`ap_start`, `ap_idle`, `ap_ready`, and `ap_done`) are set high and optimized away.



IMPORTANT! If you use the `ap_ctrl_none` control protocol in your design, you must meet at least one of the conditions for C/RTL co-simulation as described in [Interface Synthesis Requirements](#) to verify the RTL design. If at least one of these conditions is not met, C/RTL co-simulation halts with the following message:

```
@E [SIM-345] Cosim only supports the following 'ap_ctrl_none' designs:  
(1) combinational designs; (2) pipelined design with task interval of 1;  
(3) designs with array streaming or hls_stream ports.  
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

Auto-Restarting Mode

Vitis HLS designs can be control-driven modules or data-driven modules, as described in [Abstract Parallel Programming Model for HLS](#). This means that the execution of the HLS design is managed through control signals, and managed by an external software application, or by software drivers, or is driven by the presence of data at the inputs.

Data-driven modules are auto-restarting designs by the mere fact of additional data to be processed. The HLS design finishes execution of one transaction and begins execution of the next transaction when data is available. Execution stalls when no data is present, and restarts when data is present. This section is not concerned with these data-driven designs.

For control-driven modules though, the managing software application or a connected HLS design must trigger the `ap_start` signal of the module to begin processing, as described in [Block-Level Control Protocols](#). While these designs are not auto-restarting by default, they can be implemented using auto-restarting mechanisms as described here. This auto-restarting mode is useful for control-driven TLP when the HLS design uses streaming I/O for data input or output, but also has control signals allowing it to be managed by a software application. In this case the software application starts the kernel one time and the kernel runs for a specified number of transactions, or restarts continuously until it is reset or explicitly stopped by the software application.

This control-driven auto-restarting design is useful for designs with streaming data coming from and going to the I/O pins (Ethernet, SerDes) of the Xilinx device, or data streamed from or to connected HLS design modules. In other words, the HLS design is primarily data-driven, but not exclusively.

Working with Auto-Restarting Designs

Auto-restarting designs can run continuously after being started once by the software application. They can run continuously until reset and restarted, or they can be programmed to run for a predetermined number of iterations without the software application explicitly calling them multiple times in a mode called counted auto-restart. This functionality is similar to a `while(running)` loop in software code, where the `running` variable is controlled by the software application.

The control of the design is managed in hardware so that once the block is started by the software application it is automatically restarted until the iteration count is exceeded, or until explicitly stopped by the host code. In addition, the application can query the status of the design to check specific register states or to provide new parameters to be used at the next opportunity.

Auto-restarting designs make use of several unique features:

1. The auto-restart bit in the control register is used to continuously restart the design, or restart it for a specified number of iterations, without explicit software calls for each execution.
2. A mailbox feature of the Xilinx run time (XRT) to enable the software application to occasionally synchronize with the design to set new operating parameters, or check the status of the current run, as described in [Using the Mailbox](#).
3. The `software_reset` feature letting the software application reset the design to stop execution.

Supported Interfaces

Auto-restarting HLS design require the `ap_ctrl_chain` or `ap_ctrl_hs` protocol specified by Vitis HLS, where `ap_ctrl_chain` is the recommended protocol.

- The design should specify the `mode=ap_ctrl_chain` on the INTERFACE pragma or directive.
- Auto-restarting designs support streaming interfaces (`axis`) and both scalar arguments (`s_axilite`) and memory mapped (`m_axi`) arguments which can be both read and written.

Auto-restart designs can be used in a couple of different scenarios:

1. Auto-restart kernel with streaming interfaces (`axis`) only, that does not require the kernel to interact with the host application at all. Examples of this would be a Fast-Fourier Transform (FFT) with configuration data compiled into the kernel, or FIR filters with coefficients compiled into the kernel.
2. Auto-restart kernel using scalars and memory mapped (`m_axi`) arguments, either with `axis` interfaces or without. The scalar and memory mapped arguments require the mailbox to update the kernel parameters when needed. Examples of this would include a simple rule-based firewall with rules written by the host application at reset, with a counter of dropped packets that can be read by the host code but where all values must come from a single kernel execution. Or, a load balancer that uses a hash map to send data to a server, must update the server list, server map, and corresponding IP addresses simultaneously.

Enabling Auto-Restart

To enable a continuously restarting HLS design specify the following:

```
config_interface -s_axilite_auto_restart_counter 1
```

To enable auto-restart with the mailbox features in the HLS design use the following Vitis HLS commands:

```
config_interface -s_axilite_mailbox both
config_interface -s_axilite_auto_restart_counter 1
```

If the HLS design uses a streaming interface (axis), you will also need to set the following:

```
config_interface -s_axilite_sw_reset
```

Using the Mailbox

The main advantage of auto-restarting HLS designs is that they run semi-autonomously operating as data-driven without the need for frequent interaction with the software application or for software control. But auto-restarting kernels also offer semi-synchronization through a feature known as the mailbox, which provides the ability to exchange data with the software application in an asynchronous, non-blocking, and safe way.

Once started by the software application the HLS design is automatically restarted until explicitly stopped. The software application can also query the status of the design to determine when it actually has finished executing after being instructed to do so. The application and auto-restarting design use the following communications protocol:

- For passing argument values from the software application to the design, the mailbox implements a set of double-buffered `s_axilite` mapped registers to ensure non-blocking communication and consistent passing of inputs by the software application and passing of outputs by the design.
- Whenever the software application writes to an input argument, it changes the software-side copy. The HLS design running in hardware does not see that change. After the software application requests a mailbox write, the next time the design automatically restarts the copy of the registers that are seen by the design is updated with the latest inputs. Hence the software application can write any number of arguments in any order, and the HLS design does not see these updates until the software application requests a mailbox write, and the design restarts.



TIP: If some arguments are arrays mapped to `s_axilite` register files, then the entire array must be written between successive mailbox writes because it is implemented as a ping-pong buffer.

- The same process occurs on the output side when the HLS design writes to the design-side register and requests a mailbox read. The next time the design is done, the values of the `s_axilite` mapped output arguments are updated, and the software application can read them as needed.

Hence the software application has the following criteria:

- Not in charge of providing input data at every execution of the HLS design and collecting output data at its finish, as it would be in the case of control-driven designs.

- Involved in occasionally setting and updating some design input parameters (for example, routing tables, etc.) and checking the status of the design execution. This semi-synchronization operation is typically done without a fixed communication rate between the software application and the HLS design.
- When the software application has a new set of parameters to send to the HLS design, it does so without caring where the design is in its execution. When the software application needs to check the status of the HLS design, it does so without caring where design is in its execution. It is satisfied only that the parameter update and status check is performed consistently for the hardware.

Mailbox Semantics

The mailbox features can be used for both input and output, and would need to be specified for all `s_axilite` I/Os of an HLS design. It is enabled with a global option for the interfaces using the `Vitis HLS config_interface` command:

```
config_interface -s_axilite_mailbox both
config_interface -s_axilite_auto_restart_counter 1
```

After setting up the `config_interface` option, the mailbox implements a pair of registers called `HW copy` and `SW copy`. The input mailbox and output mailbox has an independent pair of registers as shown in the below figure. Communication from the software application with the HLS design includes:

- **Input Mailbox:**

- The application writes some or all elements to the `SW copy` register of the mailbox.
- The application notifies the mailbox that `SW copy` is updated.
- When the HLS design restarts, the `SW copy` register is copied to the `HW copy` register.
- The application is notified that the `HW copy` has been updated, and can change the `SW copy` register again as needed.



TIP: Multiple reads by the HLS design can occur without an update from the software application.

- **Output Mailbox:**

- The application notifies the mailbox that it wants to read an updated copy of the mailbox. The HLS design writes some or all the elements to the `HW copy` register of the mailbox at the end of the current execution.
- When the design is done, `HW copy` is copied to the `SW copy` register.
- The application is notified that `SW copy` is updated and can read it at any time.



TIP: Multiple writes by the hardware can occur without any software request to update.

Examples of Auto-Restarting Designs

The following sections describe the auto-restarting HLS design examples.

Using Auto-Restart with the Mailbox

The mailbox feature provides the ability to have semi-synchronization with a software application. The mailbox is a non-blocking mechanism that updates the HLS design parameters. Any updates provided through the mailbox will be picked up the next time the design starts.

This example design uses scalar values which will be programmed from the software application, and the design will pick them at the next software call. The scalars, `adder1` and `adder2`, will be asynchronously updated from the software application. Set the HLS design in auto-restarting mode and enable the mailbox feature using the following Vitis HLS commands:

```
config_interface -s_axilite_mailbox both
config_interface -s_axilite_auto_restart_counter 1
config_interface -s_axilite_sw_reset
```

The example HLS design code follows:

```
#define DWIDTH 32
11
12 typedef ap_axiu<DWIDTH, 0, 0, 0> pkt;
13
14 extern "C" {
15 void krnl_stream_vdatamover(hls::stream<pkt> &in,
16 | | | | | hls::stream<pkt> &out,
17 | | | | | int adder1,
18 | | | | | int adder2
19 | | | | ) {
20
21 #pragma HLS interface ap_ctrl_chain port=return
22 #pragma HLS INTERFACE s_axilite port=adder2
23 #pragma HLS port=adder1 stable
24 #pragma HLS port=adder2 stable
25 bool eos = false;
26 vdatamover:
27     do {
28         // Reading a and b streaming into packets
29         pkt t1 = in.read();
30
31         // Packet for output
32         pkt t_out;
33
34         // Reading data from input packet
35         ap_uint<DWIDTH> in1 = t1.data;
36
37         // Vadd operation
38         ap_uint<DWIDTH> tmpOut = in1+adder1+adder2;
39
40         // Setting data and configuration to output packet
41         t_out.data = tmpOut;
42         t_out.last = t1.last;
43         t_out.keep = -1; // Enabling all bytes
44
45         // Writing packet to output stream
46         out.write(t_out);
47 }
```

```

49      if (t1.last) {
50          | eos = true;
51      }
52  }
53 } while (eos == false);
54

```

Create a mailbox to update the scalars values `adder1` and `adder2`.

Update the design parameters from the software application using the `set_arg` and `write` methods as shown below. The auto-restarting HLS design will not stop itself because there is no start and stop for a streaming interface. It requires to be explicitly stopped or reset. The application code can explicitly stop the design from running using the `abort()` method.

```

// add(in1, in2, nullptr, data_size)
xrt::kernel add(device, uuid, "krnl_stream_vadd");
xrt::bo in1(device, data_size_bytes, add.group_id(0));
auto in1_data = in1.map<int*>();
xrt::bo in2(device, data_size_bytes, add.group_id(1));
auto in2_data = in2.map<int*>();

// mult(in3, nullptr, out, data_size)
xrt::kernel mult(device, uuid, "krnl_stream_vmult");
xrt::bo in3(device, data_size_bytes, mult.group_id(0));
auto in3_data = in3.map<int*>();
xrt::bo out(device, data_size_bytes, mult.group_id(2));
auto out_data = out.map<int*>();

xrt::kernel incr(device, uuid, "krnl_stream_vdatamover");
int adder1 = 20; // arbitrarily chosen to be different from 0
int adder2 = 10; // arbitrarily chosen to be different from 0

// create run objects for re-use in loop
xrt::run add_run(add);
xrt::run mult_run(mult);
std::cout << "performing never-ending mode with infinite auto
restart" << std::endl;
auto incr_run = incr(xrt::autostart{0}, nullptr, nullptr, adder1, adder2);

// create mailbox to programmatically update the incr scalar adder
xrt::mailbox incr_mbox(incr_run);

// computed expected result
std::vector<int> sw_out_data(data_size);

std::cout << " for loop started" << std::endl;
bool error = false; // indicates error in any of the iterations
for (unsigned int cnt = 0; cnt < iter; ++cnt) {

    // Create the test data and software result
    for(size_t i = 0; i < data_size; ++i) {
        in1_data[i] = static_cast<int>(i);
        in2_data[i] = 2 * static_cast<int>(i);
        in3_data[i] = static_cast<int>(i);
        out_data[i] = 0;
        sw_out_data[i] = (in1_data[i] + in2_data[i] + adder1 + adder2) *
in3_data[i];
    }
}

```

```
// sync test data to kernel
in1.sync(XCL_BO_SYNC_BO_TO_DEVICE);
in2.sync(XCL_BO_SYNC_BO_TO_DEVICE);
in3.sync(XCL_BO_SYNC_BO_TO_DEVICE);

// start the pipeline
add_run(in1, in2, nullptr, data_size);
mult_run(in3, nullptr, out, data_size);

// wait for the pipeline to finish
add_run.wait();
mult_run.wait();

// prepare for next iteration, update the mailbox with the next
// value of 'adder'.
incr_mbox.set_arg(2, ++adder1); // update the mailbox
incr_mbox.set_arg(3, --adder2); // update the mailbox

// write the mailbox content to hw, the write will not be picked
// up until the next iteration of the pipeline (incr).
incr_mbox.write(); // requests sync of mailbox to hw

// sync result from device to host
out.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

// compare with expected scalar adders
for (size_t i = 0 ; i < data_size; i++) {
    if (out_data[i] != sw_out_data[i]) {
        std::cout << "error in iteration = " << cnt
            << " expected output = " << sw_out_data[i]
            << " observed output = " << out_data[i]
            << " adder1 = " << adder1 - 1
            << " adder2 = " << adder2 + 1 << '\n';
        throw std::runtime_error("result mismatch");
    }
}
incr_run.abort();
}
```

Using Counted Auto-Restart

This example uses the same code as the prior example, but in this case the HLS design is configured to run and restart for three iterations and then stop. The only required change is to specify the number of iterations from the software application as shown in the code example below.



TIP: In the counted auto-restart case, there is no need to use the `abort()`, because the software application will know when to stop.

```
143     xrt::kernel incr(device, uuid, "increment");
144     int adder1 = 20; // arbitrarily chosen to be different from 0
145     int adder2 = 10; // arbitrarily chosen to be different from 0
151     // start the incr kernel in auto restart mode with default adders
152     // since it is a streaming kernel it will be stalled waiting for
153     // input
154     auto incr_run = incr(xrt::autostart{3}, nullptr, nullptr, adder1);
```

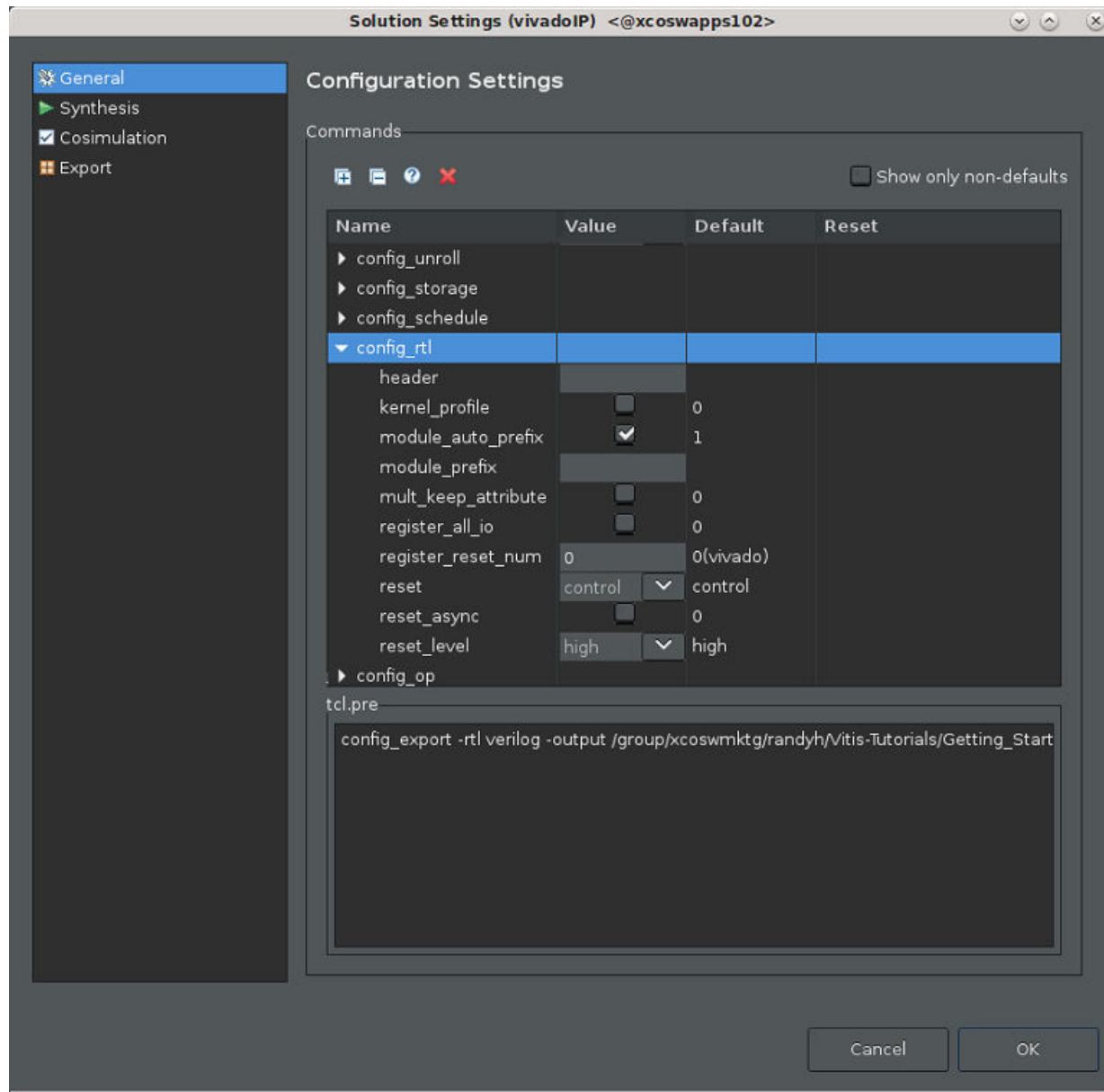
Controlling Initialization and Reset Behavior

The reset port is used in an FPGA to return the registers and block RAM connected to the reset port to an initial value any time the reset signal is applied. Typically the most important aspect of RTL configuration is selecting the reset behavior.

Note: When discussing reset behavior it is important to understand the difference between initialization and reset. Refer to [Initialization Behavior](#) for more information.

The presence and behavior of the RTL reset port is controlled using the `config_rtl` command, as shown in the following figure. You can access this command by selecting the **Solution → Solution Settings** menu command.

Figure 55: RTL Configurations



The reset settings include the ability to set the polarity of the reset and whether the reset is synchronous or asynchronous but more importantly it controls, through the **reset** option, which registers are reset when the reset signal is applied.



IMPORTANT! When AXI4 interfaces are used on a design the reset polarity is automatically changed to active-Low irrespective of the setting in the `config_rtl` configuration. This is required by the AXI4 standard.

The **reset** option has four settings:

- **none:** No reset is added to the design.

- **control:** This is the default and ensures all control registers are reset. Control registers are those used in state machines and to generate I/O protocol signals. This setting ensures the design can immediately start its operation state.
- **state:** This option adds a reset to control registers (as in the **control** setting) plus any registers or memories derived from static and global variables in the C/C++ code. This setting ensures static and global variable initialized in the C/C++ code are reset to their initialized value after the reset is applied.
- **all:** This adds a reset to all registers and memories in the design.

Finer grain control over reset is provided through the RESET pragma or directive. Static and global variables can have a reset added through the RESET directive. Variables can also be removed from those being reset by using the RESET directive's `off` option.



IMPORTANT! It is important when using the `reset state` or `all` options to consider the effect on resetting arrays.

Initialization Behavior

In C/C++, variables defined with the static qualifier and those defined in the global scope are initialized to zero, by default. These variables may optionally be assigned a specific initial value. For these initialized variables, the value in the C/C++ code is assigned at compile time (at time zero) and never again. In both cases, the initial value is implemented in the RTL.

- During RTL simulation the variables are initialized with the same values as the C/C++ code.
- The variables are also initialized in the bitstream used to program the FPGA. When the device powers up, the variables will start in their initialized state.

In the RTL, although the variables start with the same initial value as the C/C++ code, there is no way to force the variable to return to this initial state. To restore the initial state, variables must be implemented with a reset signal.



IMPORTANT! Top-level function arguments can be implemented in an AXI4-Lite interface. Because there is no way to provide an initial value in C/C++ for function arguments, these variables cannot be initialized in the RTL as doing so would create an RTL design with different functional behavior from the C/C++ code which would fail to verify during C/RTL co-simulation.

Best Practices for Designing with M_AXI Interfaces

For designers implementing a Vitis™ kernel there are various trade-offs available when working with the device memory (PLRAM, HBM and DDR) available on FPGA devices. The following is a checklist of best practices to use when designing AXI4 memory mapped interfaces for your application.

With throughput as the chief optimization goal, it is clear that accelerating the compute part of your application using the macro and micro-architecture optimizations is the first step but the time taken for transferring data to/from the kernel can also influence the application architecture with respect to throughput goals. Due to the high overhead for data transfer, it becomes important to think about overlapping the computation with the communication (data movement) that is present in your application.

For your given application:

- Decompose the kernel algorithm by building a pipeline of producer-consumer tasks, modeled using a Load, Compute, Store (LCS) coding pattern
 - All external I/O accesses must be in the Load and Store tasks.
 - There should be multiple Load or Store tasks if the kernel needs to read or write from different ports in parallel.
 - The Compute task(s) should only have scalars, array, streams or stream of blocks arguments.
 - Ensure that all these tasks (specified as functions) can be executed in overlapped fashion (enables task-level parallelism by the compiler).
 - Compute tasks can be further split up into smaller compute tasks which may contain further optimizations such as pipelining. The same rules as LCS apply for these smaller compute functions as well.
 - Always use local memory to pass data to/from the Compute tasks.
- Load and Store blocks are responsible for moving data between global memory and the Compute blocks as efficiently as possible.
 - On one end, they must read or write data through the streaming interface according to the (temporal) sequential order mandated by the Compute task inside the kernel

- On the other end, they must read or write data through the memory-mapped interface according to the (spatial) arrangement order set by the software application
- Changing your mindset about data accesses is key to building a proper HW design with HLS
 - In SW, it is common to think about how the data is “accessed” (the algorithm *pulls* the data it needs).
 - In HW, it is more efficient in think of how data “flows” through the algorithm (the data is *pushed* to the algorithm)
 - In SW, you reason about array indices and “where” data is accessed
 - In HW, you reason about streams and “when” data is accessed
- Global memories have long access times (DRAM, HBM) and their bandwidth is limited (DRAM). To reduce the overhead of accessing global memory, the interface function needs to
 - Access sufficiently large contiguous blocks of data (to benefit from **bursting**)
 - Accessing data sequentially leads to larger bursts (and higher data throughput efficiency) as compared to accessing random and/or out-of-order data (where burst analysis will fail)
 - Avoid redundant accesses (to preserve bandwidth)
- In many cases, the sequential order of data in and out of the Compute tasks is different from the arrangement order of data in global memory.
 - In this situation, optimizing the interface functions requires creating internal *caching* structures that gather enough data and organize it appropriately to minimize the overhead of global memory accesses while being able to satisfy the sequential order expected by the streaming interface
 - Example: [2D Convolution](#)
 - In order to simplify the data movement logic, the developer can also consider different ways of storing the data in memory. For instance, accessing data in DRAM in a column-major fashion can be very inefficient. Rather than implementing a dedicated data-mover in the kernel, it may be better to transpose the data in SW and store in row-major order instead which will greatly simplify HW access patterns.
- Maximize the port width of the interface, i.e., the bit-width of each AXI port by setting it to 512 bits (64 bytes).
 - Use [`hls::vector`](#) or [`ap_\(u\)int<512>`](#) as the data type of the port to infer maximal burst lengths. Usage of structs in the interface may result in poor burst performance.
 - Accessing the global memory is expensive and so accessing larger word sizes is more efficient.
 - Imagine the interface ports to be like pipes feeding data to your kernel. The wider the pipe, the more data that can be accessed and processed, and sent back.

- Transfer large blocks of data from the global device memory. One large transfer is more efficient than several smaller transfers. The bandwidth is limited by the PCIe performance. Run the [DMA test](#) to measure PCIe® transfer effective max throughput. It is usually in the range of 10-17 GB/sec for reading and writing respectively.
 - Memory resources include PLRAM (small size but fast access with the lowest latency), HBM (moderate size and access speed with some latency), and DRAM (large size but slow access with high latency).
 - Given the asynchronous nature of reads, distributed RAMs are ideal for fast buffers. You can use the read value immediately, rather than waiting for the next clock cycle. You can also use distributed RAM to create small ROMs. However, distributed ram is not suited for large memories, and you'll get better performance (and lower power consumption) for memories larger than about 128 bits using block RAM or UltraRAM.
- Decide on the optimal number of concurrent ports, i.e., the number of concurrent AXI (memory-mapped) ports
 - If the Load task needs to get multiple input data sets to feed to the Compute task, it can choose to use multiple interface ports to access this data in parallel.
 - However, the data needs to be stored in [different memory banks](#) or the accesses will be sequentialized. There is a maximum of 4 DDR banks on FPGAs while there are [32 HBM channels](#).
 - When multiple processes are accessing the same memory port or memory bank, an [arbiter](#) will sequentialize these concurrent accesses to the same memory port or bank.
- Setting the right burst length i.e., the maximum burst access length (in terms of the number of elements) for each AXI port.
 - Set the burst length equivalent to the maximum 4k bytes transfer. For example, using AXI data width of 512-bit (64 bytes), the burst length should be set to 64.
 - Transferring data in bursts hides the memory access latency and improves bandwidth usage and efficiency of the memory controller
 - Write application code in such a way to infer the maximal length bursts for both reads and writes to/from global memory
- Setting the number of outstanding memory requests that an AXI port can sustain before stalling
 - Setting a reasonable number of [outstanding requests](#) allows the system to submit multiple memory requests before stalling - this pipelining of requests allows the system to hide some of the memory latency at the cost of additional BRAM/URAM resources.

Optimizing Techniques and Troubleshooting Tips

This section outlines the various optimization techniques you can use to direct Vitis™ HLS to produce a micro-architecture that satisfies the desired performance and area goals. Using Vitis HLS, you can apply different optimization directives to the design, including:

- Pipelining tasks, allowing the next execution of the task to begin before the current execution is complete.
- Specifying a target latency for the completion of functions, loops, and regions.
- Specifying a limit on the number of resources used.
- Overriding the inherent or implied dependencies in the code to permit specific operations. For example, if it is acceptable to discard or ignore the initial data values, such as in a video stream, allow a memory read before write if it results in better performance.
- Specifying the I/O protocol to ensure function arguments can be connected to other hardware blocks with the same I/O protocol.

Note: Vitis HLS automatically determines the I/O protocol used by any sub-functions. You *cannot* control these ports except to specify whether the port is registered.

It helps to understand the process used to synthesize RTL hardware description from C/C++ source code. The [Understanding High-Level Synthesis Scheduling and Binding](#) describes some of the important details of this process to help you better understand how you can optimize for it.

You can add optimization directives directly into the source code as compiler pragmas using various HLS pragmas, or you can use `Tcl set_directive` commands to apply optimization directives in a Tcl script to be used by a solution during compilation as discussed in [Adding Pragmas and Directives](#). The following table lists the optimization directives provided by Vitis HLS as either pragma or Tcl directive.

Table 15: Vitis HLS Optimization Directives

Directive	Description
AGGREGATE	The AGGREGATE pragma is used for grouping all the elements of a struct into a single wide vector to allow all members of the struct to be read and written to simultaneously.
ALIAS	The ALIAS pragma enables data dependence analysis in Vitis HLS by defining the distance between multiple pointers accessing the same DRAM buffer.

Table 15: Vitis HLS Optimization Directives (cont'd)

Directive	Description
ALLOCATION	Specify a limit for the number of operations, implementations, or functions used. This can force the sharing or hardware resources and may increase latency.
ARRAY PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.
ARRAY_RESHAPE	Reshape an array from one with many elements to one with greater word-width. Useful for improving block RAM accesses without using more block RAM.
BIND_OP	Define a specific implementation for an operation in the RTL.
BIND_STORAGE	Define a specific implementation for a storage element, or memory, in the RTL.
DATAFLOW	Enables task level pipelining, allowing functions and loops to execute concurrently. Used to optimize throughput and/or latency.
DEPENDENCE	Used to provide additional information that can overcome loop-carried dependencies and allow loops to be pipelined (or pipelined with lower intervals).
DISAGGREGATE	Break a struct down into its individual elements.
EXPRESSION_BALANCE	Allows automatic expression balancing to be turned off.
INLINE	Inlines a function, removing function hierarchy at this level. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.
INTERFACE	Specifies how RTL ports are created from the function description.
LATENCY	Allows a minimum and maximum latency constraint to be specified.
LOOP_FLATTEN	Allows nested loops to be collapsed into a single loop with improved latency.
LOOP_MERGE	Merge consecutive loops to reduce overall latency, increase sharing and improve logic optimization.
LOOP_TRIPCOUNT	Used for loops which have variables bounds. Provides an estimate for the loop iteration count. This has no impact on synthesis, only on reporting.
OCCURRENCE	Used when pipelining functions or loops, to specify that the code in a location is executed at a lesser rate than the code in the enclosing function or loop.
PERFORMANCE	Specify the desired transaction interval for a loop and let the tool to determine the best way to achieve the result.
PIPELINE	Reduces the initiation interval by allowing the overlapped execution of operations within a loop or function.
PROTOCOL	This command specifies a region of code, a protocol region, in which no clock operations will be inserted by Vitis HLS unless explicitly specified in the code.
RESET	This directive is used to add or remove reset on a specific state variable (global or static).
STABLE	Indicates that a variable input or output of a dataflow region can be ignored when generating the synchronizations at entry and exit of the dataflow region.
STREAM	Specifies that a specific array is to be implemented as a FIFO or RAM memory channel during dataflow optimization. When using <code>hls::stream</code> , the STREAM optimization directive is used to override the configuration of the <code>hls::stream</code> .
TOP	The top-level function for synthesis is specified in the project settings. This directive may be used to specify any function as the top-level for synthesis. This then allows different solutions within the same project to be specified as the top-level function for synthesis without needing to create a new project.
UNROLL	Unroll for-loops to create multiple instances of the loop body and its instructions that can then be scheduled independently.

In addition to the optimization directives, Vitis HLS provides a number of configuration commands that can influence the performance of synthesis results. Details on using configurations commands can be found in [Setting Configuration Options](#). The following table reflects some of these commands.

Table 16: Vitis HLS Configurations

GUI Directive	Description
Config Array Partition	Determines how arrays are partitioned, including global arrays and if the partitioning impacts array ports.
Config Compile	Controls synthesis specific optimizations such as the automatic loop pipelining and floating point math optimizations.
Config Dataflow	Specifies the default memory channel and FIFO depth in dataflow optimization.
Config Interface	Controls I/O ports not associated with the top-level function arguments and allows unused ports to be eliminated from the final RTL.
Config Op	Configures the default latency and implementation of specified operations.
Config RTL	Provides control over the output RTL including file and module naming, and reset controls.
Config Schedule	Determines the effort level to use during the synthesis scheduling phase and the verbosity of the output messages
Config Storage	Configures the default latency and implementation of specified storage types.
Config Unroll	Configures the default tripcount threshold for unrolling loops.

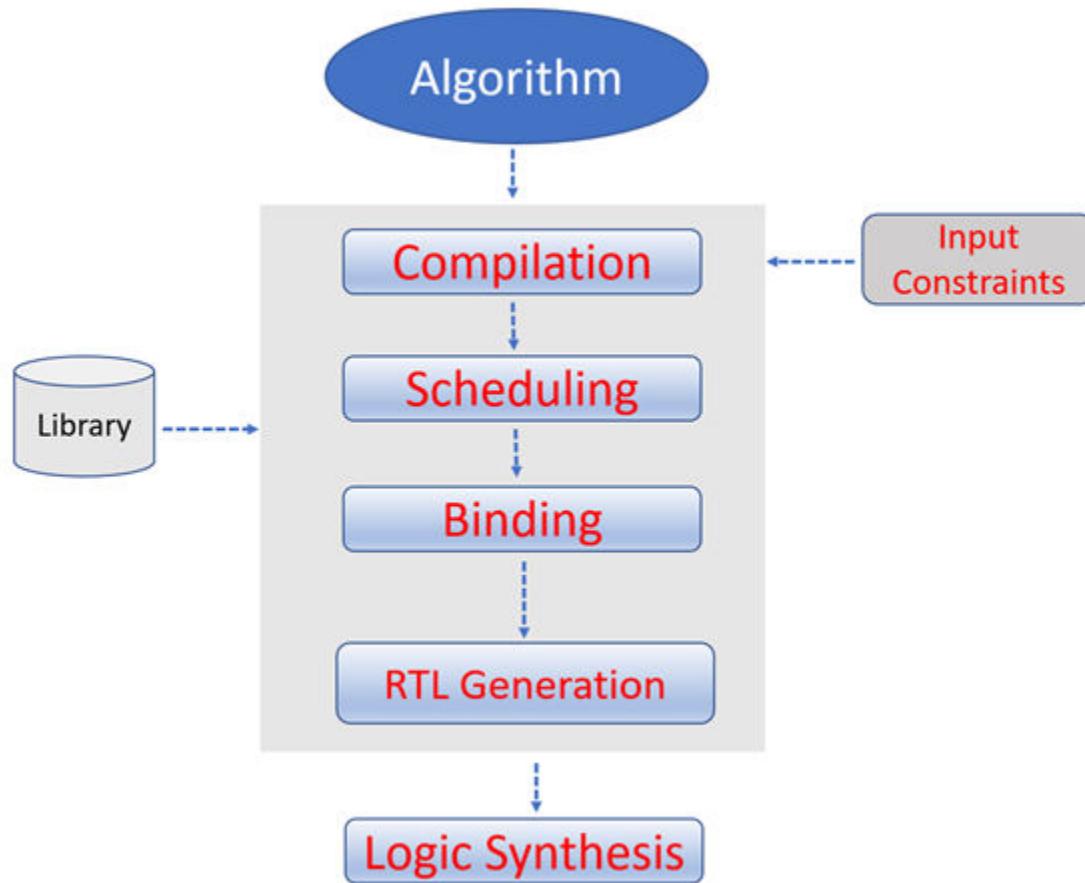
Understanding High-Level Synthesis Scheduling and Binding

High-Level Synthesis tools transform an untimed high-level specification into a fully timed implementation. During this transformation, a custom architecture is implemented to meet the specification requirements. The architecture generated contains the data path, control logic, memory interfaces, and how the RTL communicates with the external world. A data path consists of a set of storage elements such as (registers, register files, or memories), a set of functional units (such as ALUs, multipliers, shifters, and other custom functions), and interconnect elements (such as tristate drivers, multiplexers, and buses). Each component can take one or more clock cycles to execute, can be pipelined, and can have input or output registers. In addition, the entire data path and controller can be pipelined in several stages.

The designers should invest the early part of the project in redefining the architecture of the algorithm to meet the performance while keeping the algorithm at a higher level. For any specific HLS tool, there are design principles and best practices that are required to be followed to generate the optimized RTL that meets the expected performance.

The HLS Tool executes the following tasks as shown in the diagram below.

Figure 56: HLS Tasks



1. Compile the algorithm written to meet specifications: This step includes several code optimizations such as dead-code elimination, constant folding, reporting unsupported constructs, etc.
2. Schedule the operations for given clock cycles:
 - a. The "Schedule" phase determines which operations occur during each clock cycle based on:
 - When an operation's dependencies have been satisfied or are available.
 - The length of the clock cycle or clock frequency.
 - The time it takes for the operation to complete, as defined by the target device. More operations can be completed in a single clock cycle for longer clock periods. Some operations might need to be implemented as multi-cycle resources. HLS automatically schedules operations over more clock cycles
 - The available resources.
 - Incorporation of any user-specified optimization directives.

- b. During the "Schedule" phase, the tool determines what operator will execute in a given cycle and how many of these components are needed. The next step determines what operation binds to what resource.
3. Bind the operations to the functional components and variables to the storage elements
 - a. The binding task assigns hardware resources to implement each scheduled operation and maps operators (such as addition, multiplication, and shift) to specific RTL implementations. For example, a mult operation can be implemented in RTL as a combinational or pipelined multiplier.
 - b. The binding task assigns memories, registers, or combinations of these to the array variables inside the function to meet the desired performance.
 - c. If multiple operations use the same resource, this step can perform the resource sharing if not used in the same cycle.
4. Control logic extraction creates a finite state machine (FSM) that sequences the operations in the RTL design according to the defined schedule.
5. Creates the logic to communicate with the external world: The RTL generated will be communicating with the external world like streaming data from the external port or start/stop logic or accessing external memory.
6. Finally, generate the RTL architecture

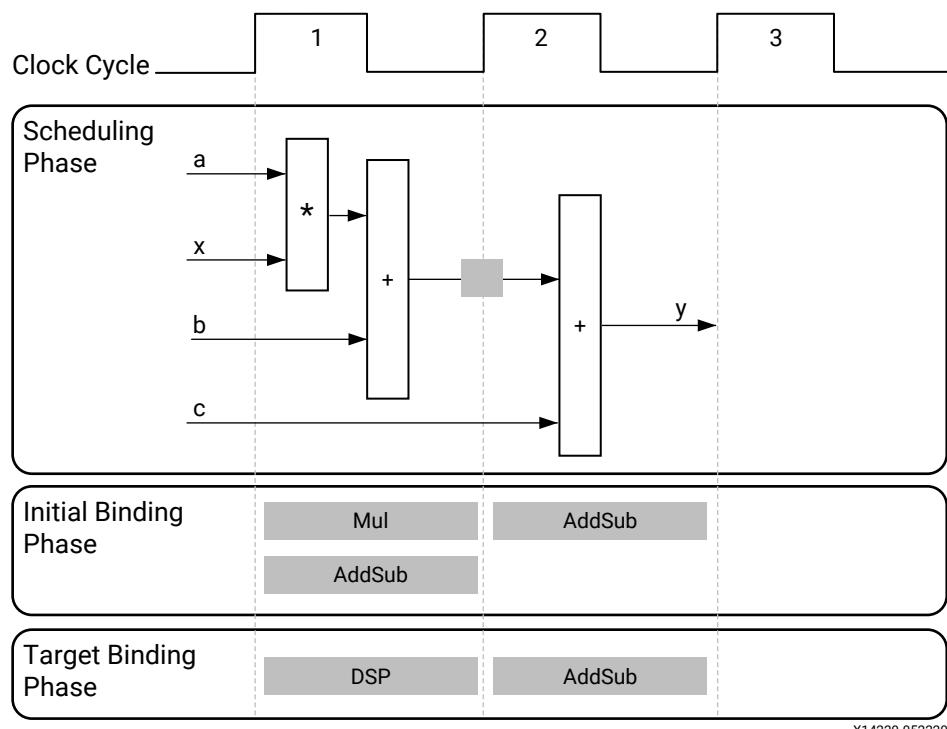
The next section walks through conceptually, how an HLS tool in general schedules the operators based on input constraints like a clock cycle and binds them to available hardware resources.

Scheduling and Binding Example

The following figure shows an example of the scheduling and binding phases for this code example:

```
int foo(char x, char a, char b, char c) {
    char y;
    y = x*a+b+c;
    return y;
}
```

Figure 57: Scheduling and Binding Example



X14220-052220

In the scheduling phase of this example, high-level synthesis schedules the following operations to occur during each clock cycle:

- First clock cycle: Multiplication and the first addition
- Second clock cycle: Second addition, if the result of the first addition is available in the second clock cycle, and output generation

Note: In the preceding figure, the square between the first and second clock cycles indicates when an internal register stores a variable. In this example, high-level synthesis only requires that the output of the addition is registered across a clock cycle. The first cycle reads x , a , and b data ports. The second cycle reads data port c and generates output y .

In the final hardware implementation, high-level synthesis implements the arguments to the top-level function as input and output (I/O) ports. In this example, the arguments are simple data ports. Because each input variable is a `char` type, the input data ports are all 8-bits wide. The function `return` is a 32-bit `int` data type, and the output data port is 32-bits wide.



IMPORTANT! The advantage of implementing the C code in the hardware is that all operations finish in a shorter number of clock cycles. In this example, the operations complete in only two clock cycles. In a central processing unit (CPU), even this simple code example takes more clock cycles to complete.

In the initial binding phase of this example, high-level synthesis implements the multiplier operation using a combinational multiplier (Mul) and implements both add operations using a combinational adder/subtractor (AddSub).

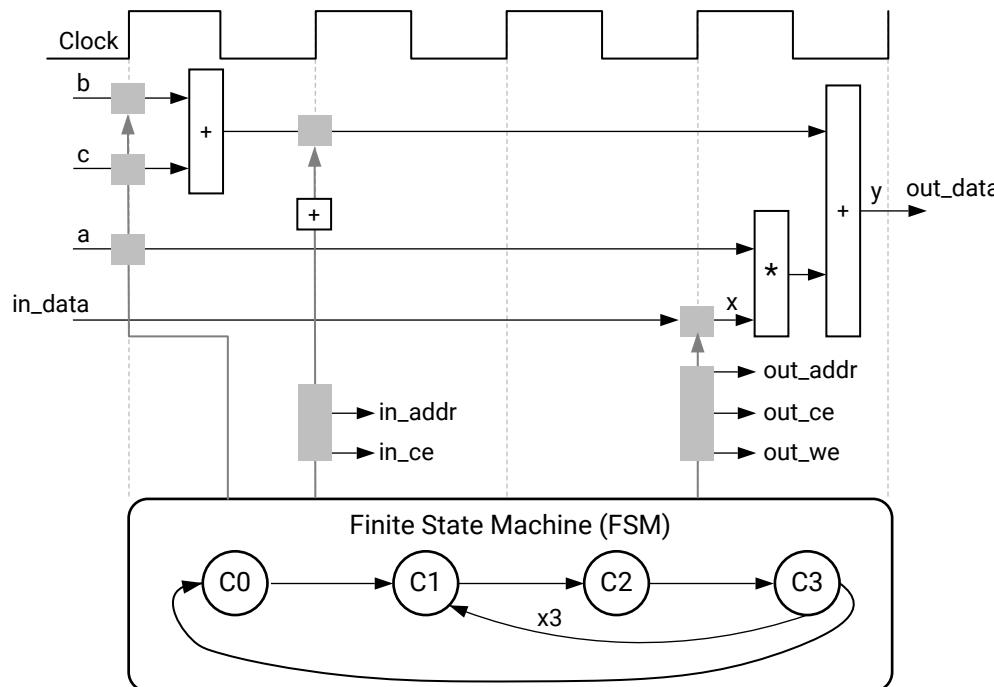
In the target binding phase, high-level synthesis implements both the multiplier and one of the addition operations using a DSP module resource. Some applications use many binary multipliers and accumulators that are best implemented in dedicated DSP resources. The DSP module is a computational block available in the FPGA architecture that provides the ideal balance of high-performance and efficient implementation.

Extracting Control Logic and Implementing I/O Ports Example

The following figure shows the extraction of control logic and implementation of I/O ports for this code example:

```
void foo(int in[3], char a, char b, char c, int out[3]) {  
    int x,y;  
    for(int i = 0; i < 3; i++) {  
        x = in[i];  
        y = a*x + b + c;  
        out[i] = y;  
    }  
}
```

Figure 58: Control Logic Extraction and I/O Port Implementation Example



X14218-100520

This code example performs the same operations as the previous example. However, it performs the operations inside a for-loop, and two of the function arguments are arrays. The resulting design executes the logic inside the for-loop three times when the code is scheduled. High-level synthesis automatically extracts the control logic from the C code and creates an FSM in the RTL design to sequence these operations. Top-level function arguments become ports in the final RTL design. The scalar variable of type `char` maps into a standard 8-bit data bus port. Array arguments, such as `in` and `out`, contain an entire collection of data.

In high-level synthesis, arrays are synthesized into block RAM by default, but other options are possible, such as FIFOs, distributed RAM, and individual registers. When using arrays as arguments in the top-level function, high-level synthesis assumes that the block RAM is outside the top-level function and automatically creates ports to access a block RAM outside the design, such as data ports, address ports, and any required chip-enable or write-enable signals.

The FSM controls when the registers store data and controls the state of any I/O control signals. The FSM starts in the state `C0`. On the next clock, it enters state `C1`, then state `C2`, and then state `C3`. It returns to state `C1` (and `C2`, `C3`) a total of three times before returning to state `C0`.

Note: This closely resembles the control structure in the C code for-loop. The full sequence of states are: `C0,{C1, C2, C3},{C1, C2, C3},{C1, C2, C3}`, and return to `C0`.

The design requires the addition of `b` and `c` only one time. High-level synthesis moves the operation outside the for-loop and into state `C0`. Each time the design enters state `C3`, it reuses the result of the addition.

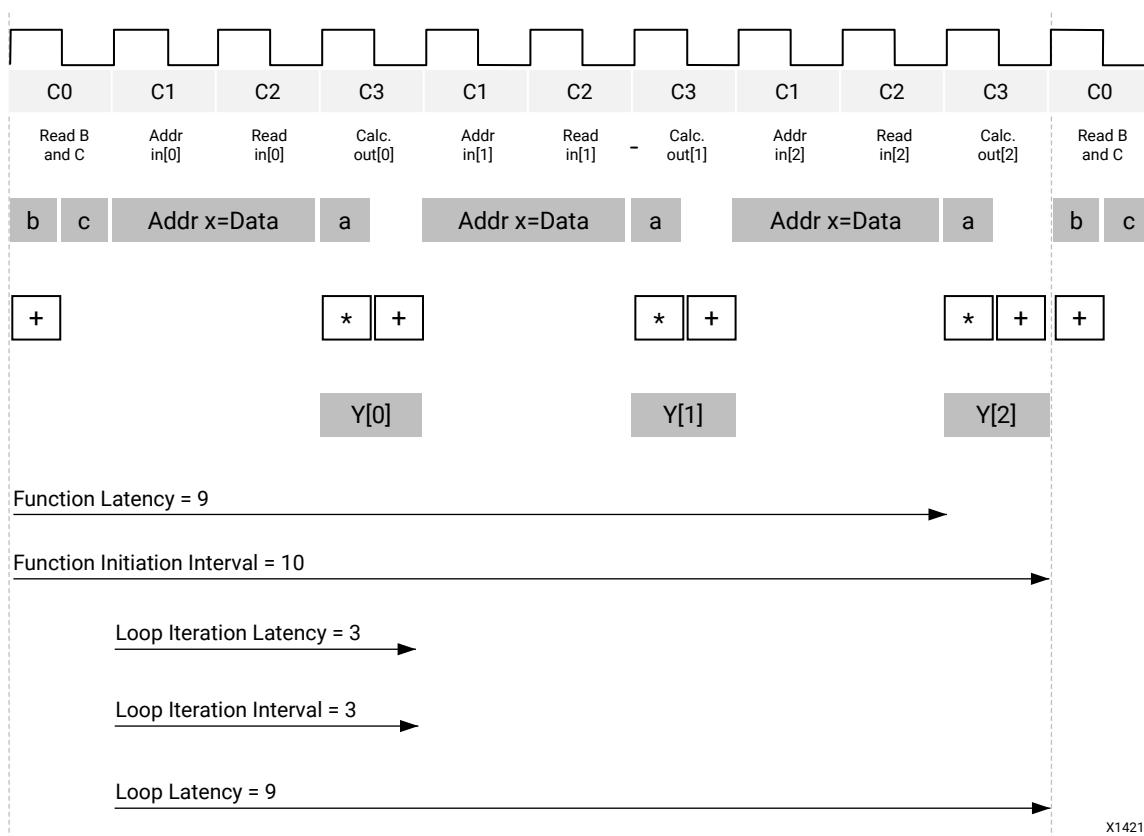
The design reads the data from `in` and stores the data in `x`. The FSM generates the address for the first element in state `C1`. In addition, in state `C1`, an adder increments to keep track of how many times the design must iterate around states `C1`, `C2`, and `C3`. In state `C2`, the block RAM returns the data for `in` and stores it as variable `x`.

High-level synthesis reads the data from port `a` with other values to perform the calculation and generates the first `y` output. The FSM ensures that the correct address and control signals are generated to store this value outside the block. The design then returns to state `C1` to read the next value from the array/block RAM `in`. This process continues until all outputs are written. The design then returns to state `C0` to read the next values of `b` and `c` to start the process again.

Performance Metrics Example

The following figure shows the complete cycle-by-cycle execution for the code in the previous example, including the states for each clock cycle, read operations, computation operations, and write operations.

Figure 59: Latency and Initiation Interval Example



The following are performance metrics for this example:

- **Latency:** It takes the function 9 clock cycles to output all values.
Note: When the output is an array, the latency is measured to the last array value output.
- **Initiation Interval (II):** The II is 10, which means it takes 10 clock cycles before the function can initiate a new set of input reads and start to process the next set of input data.
Note: The time to perform one complete execution of a function is referred to as one *transaction*. In this example, it takes 11 clock cycles before the function can accept data for the next transaction.
- **Loop iteration latency:** The latency of each loop iteration is 3 clock cycles.
- **Loop II:** The interval is 3.
- **Loop latency:** The latency is 9 clock cycles.

Optimizing Logic

Inferring Shift Registers

Vitis HLS will now infer a shift register when encountering the following code:

```
int A[N]; // This will be replaced by a shift register

for(...) {
    // The loop below is the shift operation
    for (int i = 0; i < N-1; ++i)
        A[i] = A[i+1];
    A[N] = ...;

    // This is an access to the shift register
    ... A[x] ...
}
```

Shift registers can perform a one shift operation per cycle, and also allows a random read access per cycle anywhere in the shift register, thus it is more flexible than a FIFO.

Optimizing Logic Expressions

During synthesis several optimizations, such as strength reduction and bit-width minimization are performed. Included in the list of automatic optimizations is expression balancing.

Expression balancing rearranges operators to construct a balanced tree and reduce latency.

- For integer operations expression balancing is on by default but may be disabled using the [EXPRESSION_BALANCE](#) pragma or directive.
- For floating-point operations, expression balancing is off by default but may be enabled using using the `config_compile -unsafe_math_optimizations` command, as discussed below.

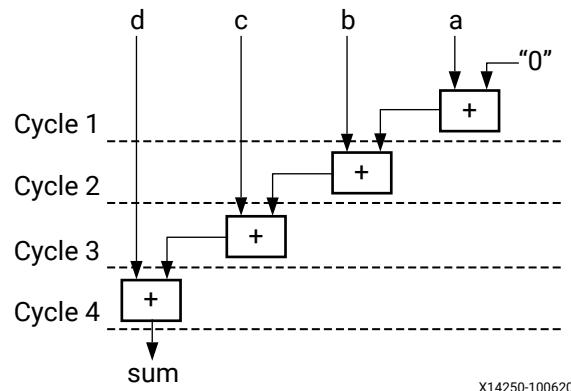
Given the highly sequential code using assignment operators such as `+=` and `*=` in the following example (or resulting from loop unrolling):

```
data_t foo_top (data_t a, data_t b, data_t c, data_t d)
{
    data_t sum;

    sum = 0;
    sum += a;
    sum += b;
    sum += c;
    sum += d;
    return sum;
}
```

Without expression balancing, and assuming each addition requires one clock cycle, the complete computation for `sum` requires four clock cycles shown in the following figure.

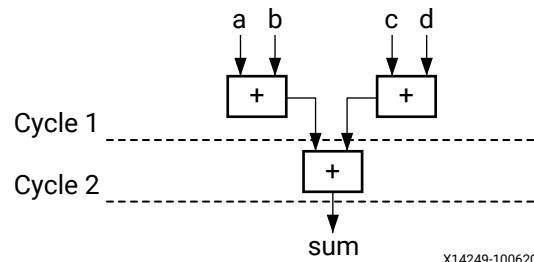
Figure 60: Adder Tree



X14250-100620

However additions $a+b$ and $c+d$ can be executed in parallel allowing the latency to be reduced. After balancing the computation completes in two clock cycles as shown in the following figure. Expression balancing prohibits sharing and results in increased area.

Figure 61: Adder Tree After Balancing



X14249-100620

For integers, you can disable expression balancing using the `EXPRESSION_BALANCE` optimization directive with the `off` option. By default, Vitis HLS does not perform the `EXPRESSION_BALANCE` optimization for operations of type `float` or `double`. When synthesizing `float` and `double` types, Vitis HLS maintains the order of operations performed in the C/C++ code to ensure that the results are the same as the C/C++ simulation. For example, in the following code example, all variables are of type `float` or `double`. The values of `O1` and `O2` are not the same even though they appear to perform the same basic calculation.

```
A=B*C; A=B*F;  
D=E*F; D=E*C;  
O1=A*D O2=A*D;
```

This behavior is a function of the saturation and rounding in the C/C++ standard when performing operation with types `float` or `double`. Therefore, Vitis HLS always maintains the exact order of operations when variables of type `float` or `double` are present and does not perform expression balancing by default.

You can enable expression balancing for specific operations, or you can configure the tool to enable expression balancing with `float` and `double` types using the `config_compile -unsafe_math_optimizations` command as follows:

1. In the Vitis HLS IDE, select **Solution** → **Solution Settings**.
2. In the Solution Settings dialog box, click the **General** category, select **config_compile**, and enable **unsafe_math_optimizations**.

With this setting enabled, Vitis HLS might change the order of operations to produce a more optimal design. However, the results of C/RTL co-simulation might differ from the C/C++ simulation.

The `unsafe_math_optimizations` feature also enables the `no_signed_zeros` optimization. The `no_signed_zeros` optimization ensures that the following expressions used with `float` and `double` types are identical:

```
x - 0.0 = x;  
x + 0.0 = x;  
0.0 - x = -x;  
x - x = 0.0;  
x*0.0 = 0.0;
```

Without the `no_signed_zeros` optimization the expressions above would not be equivalent due to rounding. The optimization may be optionally used without expression balancing by selecting only this option in the `config_compile` command.



TIP: When the `unsafe_math_optimizations` and `no_signed_zero` optimizations are used, the RTL implementation will have different results than the C/C++ simulation. The test bench should be capable of ignoring minor differences in the result: check for a range, do not perform an exact comparison.

Optimizing AXI System Performance

Introduction

A Vitis accelerated system includes a global memory subsystem that is used to share data between the kernels and the host application. Global memory available on the host system, outside of the Xilinx device, provides very large amounts of storage space but at the cost of longer access time compared to local memory on the Xilinx device. One of the measurements of the performance of a system/application is throughput, which is defined as the number of bytes transferred in a given time frame. Therefore, inefficient data transfers from/to the global memory will have a long memory access time which can adversely affect system performance and kernel execution time.

Development of accelerated applications in Vitis HLS should include two phases: kernel development, and improving system performance. [Design Principles](#) suggested a kernel development approach implementing a cache-like Load-Compute-Store structure where the load-store functions read/write data to the global memory. Improving system performance involves implementing an efficient load and store design that can improve the kernel execution time. This chapter describes the features and metrics that can impact and improve the throughput of the load-store (LS) functions. Refer to [Vitis-HLS-Introductory-Examples/Interface/Memory](#) on Github for examples of some of the following concepts.

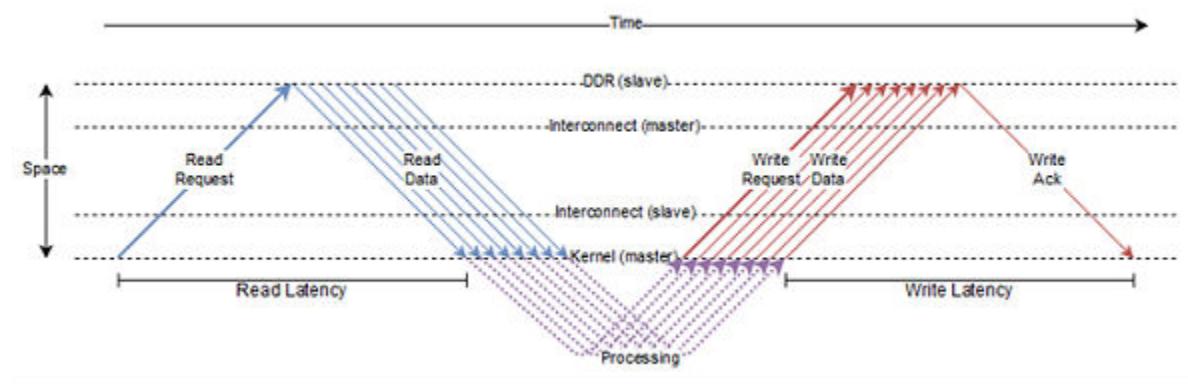
AXI Burst Transfers

Overview of Burst Transfers

Bursting is an optimization that tries to intelligently aggregate your memory accesses to the DDR to maximize the throughput bandwidth and/or minimize the latency. Bursting is one of many possible optimizations to the kernel. Bursting typically gives you a 4-5x improvement while other optimizations, like access widening or ensuring there are no dependencies through the DDR, can provide even bigger performance improvements. Typically, bursting is useful when you have contention on the DDR ports from multiple competing kernels.

The burst feature of the AXI4 protocol improves the throughput of the load-store functions by reading/writing chunks of data to or from the global memory in a single request. The larger the size of the data, the higher the throughput. This metric is calculated as follows ($(\# \text{of bytes transferred})^*$ (kernel frequency)/(Time)). The maximum kernel interface bitwidth is 512 bits, and if the kernel is compiled to run at 300 MHz then it can theoretically achieve $(512^* 300 \text{ Mhz})/1 \text{ sec} = \sim 17 \text{ GB/s}$ for a DDR.

Figure 62: AXI Protocol



The figure above shows how the AXI protocol works. The HLS kernel sends out a read request for a burst of length 8 and then sends a write request burst of length 8. The read latency is defined as the time taken between the sending of the read request burst to when the data from the first read request in the burst is received by the kernel. Similarly, the write latency is defined as the time taken between when data for the last write in the write burst is sent and the time the write acknowledgment is received by the kernel. Read requests are usually sent at the first available opportunity while write requests get queued until the data for each write in the burst becomes available.

To understand the underlying semantics of burst transfers consider the following code snippet:

```
for(size_t i = 0; i < size; i++) {  
    out[f(i)] = in[f(i)];  
}
```

Vitis HLS performs automatic burst optimization, which intelligently aggregates the memory accesses inside the loops/functions from the user code and performs read/write to the global memory of a particular size. These read/writes are converted into a read request, write request, and write response to the global memory. Depending on the memory access pattern Vitis HLS automatically inserts these read and write requests either outside the loop bound or inside the loop body. Depending on the placement of these requests, Vitis HLS defines two types of burst requests: sequential burst and pipelined burst.

Burst Semantics

For a given kernel, the HLS compiler implements the burst analysis optimization as a multi-pass optimization, but on a per function basis. Bursting is only done for a function and bursting across functions is not supported. The burst optimizations are reported in the [Synthesis Summary](#) report, and missed burst opportunities are also reported to help you improve burst optimization.

At first, the HLS compiler looks for memory accesses in the basic blocks of the function, such as memory accesses in a sequential set of statements inside the function. Assuming the preconditions of bursting are met, each burst inferred in these basic blocks is referred to as *sequential burst*. The compiler will automatically scan the basic block to build the longest sequence of accesses into a single sequential burst.

The compiler then looks at loops and tries to infer what are known as *pipeline* bursts. A pipeline burst is the sequence of reads/writes across the iterations of a loop. The compiler tries to infer the length of the burst by analyzing the loop induction variable and the trip count of the loop. If the analysis is successful, the compiler can chain the sequences of reads/writes in each iteration of the loop into one long pipeline burst. The compiler today automatically infers a pipeline or a sequential burst, but there is no way to request a specific type of burst. The code needs to be written so as to cause the tool to infer the pipeline or sequential burst.

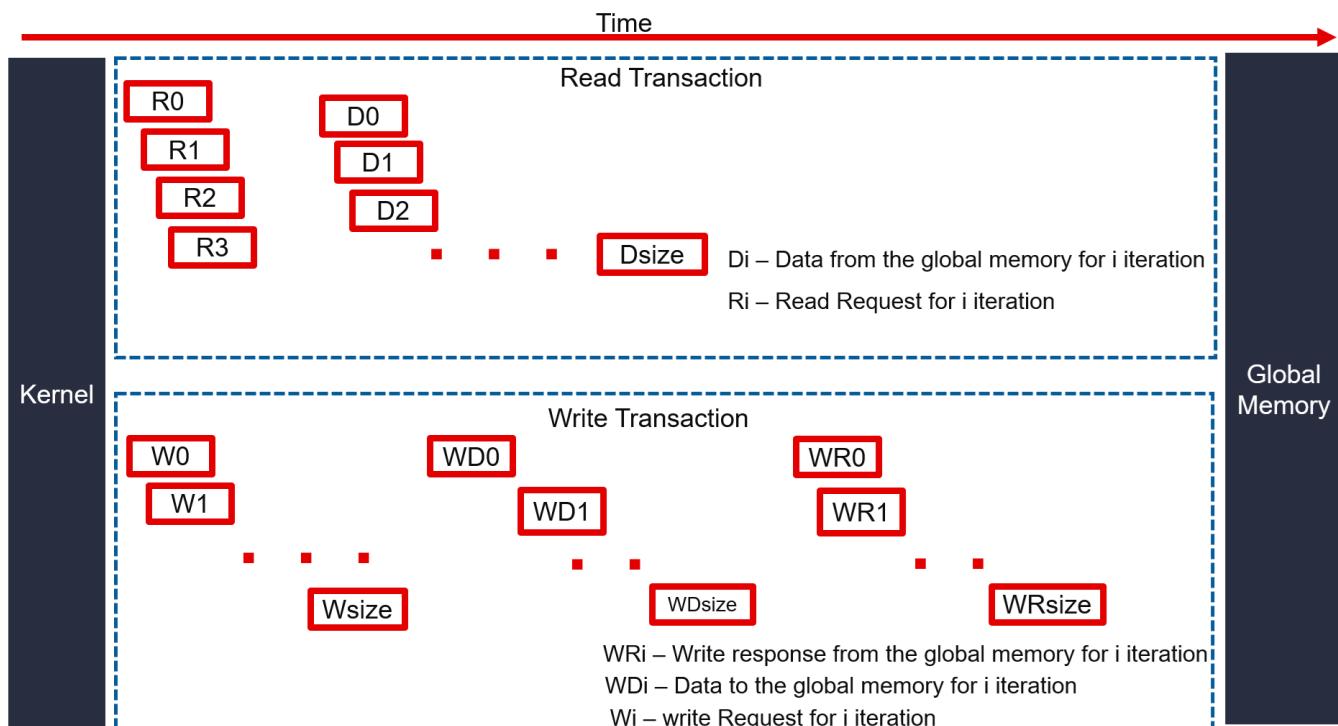
Pipeline Burst

Pipeline burst improves the throughput of the functions by reading or writing large amounts, or the maximum amount of data in a single request. The advantage of the pipeline burst is that the future requests ($i+1$) do not have to wait for the current request (i) to finish because the read request, write request, and write response are outside the loop body and performs the requests as soon as possible, as shown in the code example below. This significantly improves the throughput of the functions as it takes less time to read/write the whole loop bound.

```

rb = ReadReq(i, size);
wb = WriteReq(i, size);
for(size_t i = 0; i < size; i++) {
    Write(wb, i) = f(Read(rb, i));
}
WriteResp(wb);
    
```

Figure 63: Pipeline Burst



If the compiler can successfully deduce the burst length from the induction variable (`size`) and the trip count of the loop, it will infer one big pipeline burst and will move the `ReadReq`, `WriteReq` and `WriteResp` calls outside the loop, as shown in the Pipeline Burst code example. So, the read requests for all loop iterations are combined into one read request and all the write requests are combined into one write request. Note that all read requests are typically sent out immediately while write requests are only sent out after the data becomes available.

However, if any of the preconditions of bursting are not met, as described in [Preconditions and Limitations of Burst Transfer](#), the compiler may not infer a pipeline burst but will instead try and infer a sequential burst where the `ReadReq`, `WriteReg` and `WriteResp` are alongside the read/write accesses being burst optimized, as shown in the Sequential Burst code example. In this case, the read and write requests for each loop iteration are combined into one read or write request.

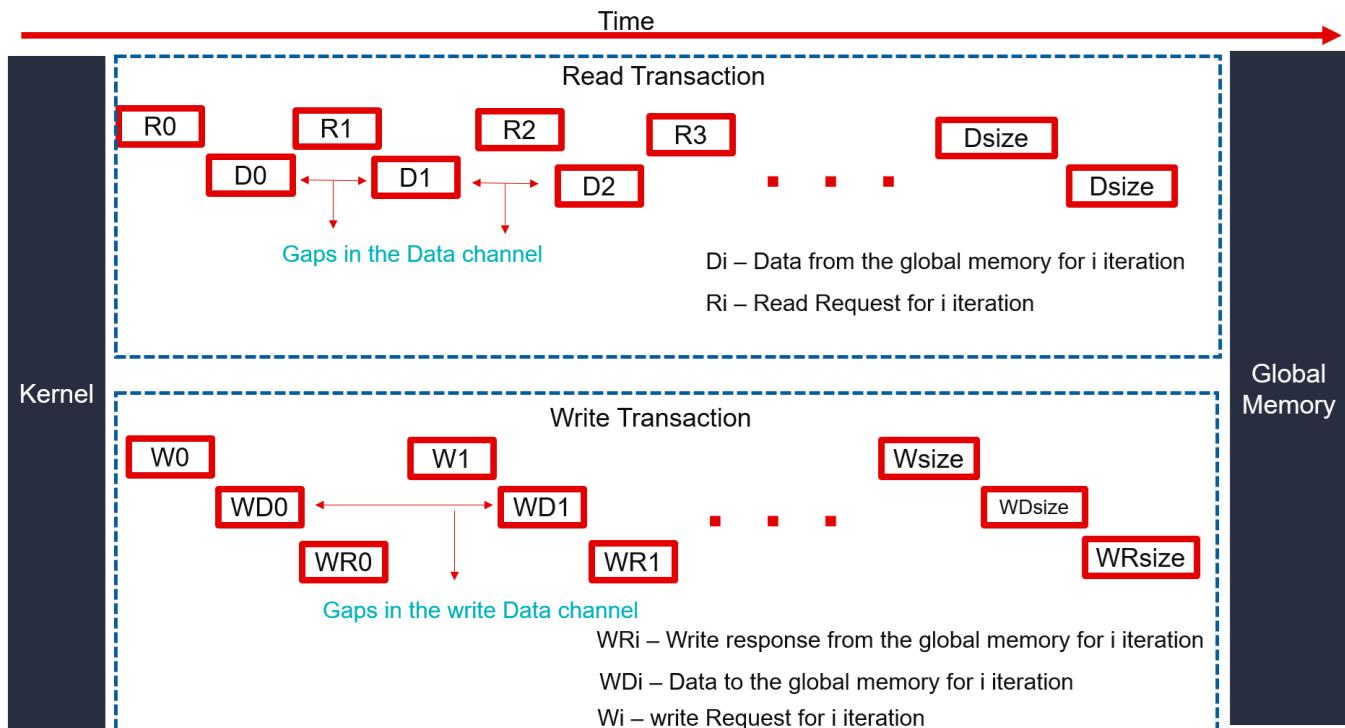
Sequential Burst

A sequential burst consists of smaller data sizes where the read requests, write requests, and write responses are inside a loop body as shown in the following code example.

```
for(size_t i = 0; i < size; i++) {  
    rb = ReadReq(i, 1);  
    wb = WriteReq(i, 1);  
    Write(wb, i) = f(Read(rb, i));  
    WriteResp(wb);  
}
```

The drawback of sequential burst is that a future request ($i+1$) depends on the current request (i) finishing because it is waiting for the read request, write request, and write response to complete. This will create gaps between requests as shown in the figure below.

Figure 64: Sequential Burst



A sequential burst is not as effective as pipeline burst because it is reading or writing a small data size multiple times to compensate for the loop bounds. Although this will have a significant impact on the throughput, sequential burst is still better than no burst. Vitis HLS uses this burst technique if your code does not adhere to the [Preconditions and Limitations of Burst Transfer](#).



TIP: The size of burst requests can be further partitioned into multiple requests of user-specified size, which is controlled using the `max_read_burst_length` and `max_write_burst_length` of the INTERFACE pragma or directive, as discussed in [Options for Controlling AXI4 Burst Behavior](#).

Preconditions and Limitations of Burst Transfer

Bursting Preconditions

Bursting is about aggregating successive memory access requests. Here are the set of preconditions that these successive accesses must meet for the bursting optimization to launch successfully:

- Must be all reads, or all writes – bursting reads and writes is not possible.
- Must be a monotonically increasing order of access (both in terms of the memory location being accessed as well as in time). You cannot access a memory location that is in between two previously accessed memory locations.
- Must be consecutive in memory – one next to another with no gaps or overlap and in forward order.
- The number of read/write accesses (or burst length) must be determinable before the request is sent out. This means that even if the burst length is computed at runtime, it must be computed before the read/write request is sent out.
- If bundling two arrays to the same M-AXI port, bursting will be done only for one array, at most, in each direction at any given time.
- There must be no dependency issues from the time a burst request is initiated and finished.



TIP: The `volatile` qualifier prevents burst access to or from the variable.

Outer Loop Burst Failure Due to Overlapping Memory Accesses

Outer loop burst inference will fail in the following example because both iteration 0 and iteration 1 of the loop L1 access the same element in arrays a and b. Burst inference is an all or nothing type of optimization - the tool will not infer a partial burst. It is a greedy algorithm that tries to maximize the length of the burst. The auto-burst inference will try to infer a burst in a bottom up fashion - from the inner loop to the outer loop, and will stop when one of the preconditions is not met. In the example below the burst inference will stop when it sees that element 8 is being read again, and so an inner loop burst of length 9 will be inferred in this case.

```
L1: for (int i = 0; i < 8; ++i)
    L2: for (int j = 0; j < 9; ++j)
        b[i*8 + j] = a[i*8 + j];

itr 0: 10 1 2 3 4 5 6 7 8 |
itr 1: | 8 9 10 11 12 13 14 15 16 |
```

Usage of ap_int/ap_uint Types as Loop Induction Variables

Because the burst inference depends on the loop induction variable and the trip count, using non-native types can hinder the optimization from firing. It is recommended to always use unsigned integer type for the loop induction variable.

Must Enter Loop at Least Once

In some cases, the compiler can fail to infer that the max value of the loop induction variable can never be zero – that is, if it cannot prove that the loop will always be entered. In such cases, an assert statement will help the compiler infer this.

```
assert (N > 0);
L1: for(int a = 0; a < N; ++a) { ... }
```

Inter or Intra Loop Dependencies on Arrays

If you write to an array location and then read from it in the same iteration or the next, this type of array dependency can be hard for the optimization to decipher. Basically, the optimization will fail for these cases because it cannot guarantee that the write will happen before the read.

Conditional Access to Memory

If the memory accesses are being made conditionally, it can cause the burst inferencing algorithm to fail as it cannot reason through the conditional statements. In some cases, the compiler will simplify the conditional and even remove it but it is generally recommended to not use conditional statements around the memory accesses.

M-AXI Accesses Made from Inside a Function Called from a Loop

Cross-functional array access analysis is not a strong suit for compiler transformations such as burst inferencing. In such cases, users can inline the function using the `INLINE` pragma or directive to avoid burst failures.

```
void my_function(hls::stream<T> &out_pkt, int *din, int input_idx) {
    T v;
    v.data = din[input_idx];
    out_pkt.write(v);
}

void my_kernel(hls::stream<T> &out_pkt,
               int             *din,
               int             num_512_bytes,
               int             num_times) {
#pragma HLS INTERFACE mode=m_axi port = din offset=slave bundle=gmem0
#pragma HLS INTERFACE mode=axis port=out_pkt
#pragma HLS INTERFACE mode=s_axilite port=din bundle=control
#pragma HLS INTERFACE mode=s_axilite port=num_512_bytes bundle=control
#pragma HLS INTERFACE mode=s_axilite port=num_times bundle=control
#pragma HLS INTERFACE mode=s_axilite port=return bundle=control

unsigned int idx = 0;
L0: for (int i = 0; i < ntimes; ++i) {
    L1: for (int j = 0; j < num_512_bytes; ++j) {
#pragma HLS PIPELINE
        my_function(out_pkt, din, idx++);
    }
}
```

Burst inferencing will fail because the memory accesses are being made from a called function. For the burst inferencing to work, it is recommended that users inline any such functions that are making accesses to the M-AXI memory.

An additional reason the burst inferencing will fail in this example is that the memory accessed through `din` in `my_function`, is defined by a variable (`idx`) which is not a function of the loop induction variables `i` and `j`, and therefore may not be sequential or monotonic. Instead of passing `idx`, use `(i*num_512_bytes+j)`.

Pipelined Burst Inference on a Dataflow Loop

Burst inference is not supported on a loop that has the `DATAFLOW` pragma or directive. However, each process/task inside the dataflow loop can have bursts. Also, sharing of M-AXI ports is not supported inside a dataflow region because the tasks can execute in parallel.

Options for Controlling AXI4 Burst Behavior

An optimal AXI4 interface is one in which the design never stalls while waiting to access the bus, and after bus access is granted, the bus never stalls while waiting for the design to read/write. There are many elements of the design that affect the system performance and burst transfer, such as the following:

- Latency
- Port Width
- Multiple Ports
- Specified Burst Length
- Number of Outstanding Reads/Writes

Latency

The read latency is defined as the time taken between sending the burst read request to when the kernel receives the data from the first read request in the burst. Similarly, the write latency is defined as the time it takes between when data for the last write in the burst is sent and the time the write response is received by the kernel. These latencies can be non-deterministic since they depend on system characteristics such as congestion on the DDR access. Because of this Vitis HLS can not accurately determine the memory read/write latency during synthesis, and so uses a default latency of 64 kernel cycles to schedule the requests and operations as below.

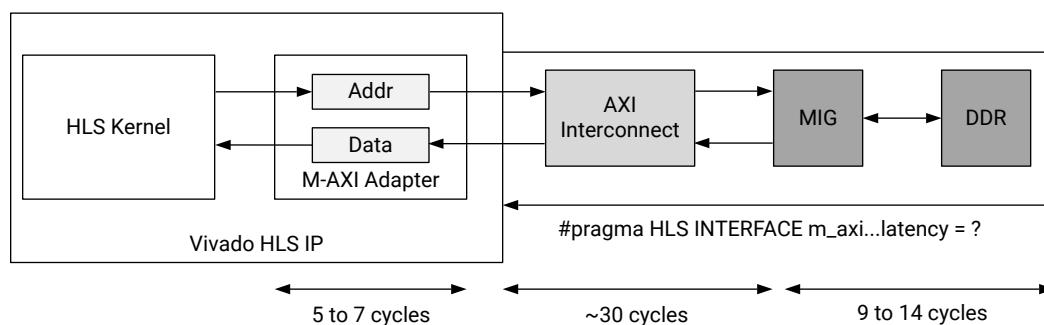
- It schedules the read/write requests and waits for the data, in parallel perform memory-independent operations, such as working on streams or compute
- Wait to schedule new read/write requests



TIP: The default tool latency can be changed using the `LATENCY` pragma or directive.

To help you understand the various latencies that are possible in the system, the following figure shows what happens when an HLS kernel sends a burst to the DDR.

Figure 65: Burst Transaction Diagram



X24687-100620

When your design makes a read/write request, the request is sent to the DDR through several specialized helper modules. First, the M-AXI adapter serves as a buffer for the requests created by the HLS kernel. The adapter contains logic to cut large bursts into smaller ones (which it needs to do to prevent hogging the channel or if the request crosses the 4 KB boundary, see *Vivado Design Suite: AXI Reference Guide (UG1037)*), and can also stall the sending of burst requests (depending on the maximum outstanding requests parameter) so that it can safely buffer the entirety of the data for each kernel. This can slightly increase write latency but can resolve deadlock due to concurrent requests (read or write) on the memory subsystem. You can configure the M-AXI interface to hold the write request until all data is available using `config_interface -m_axi_conservative_mode`.

Getting through the adapter will cost a few cycles of latency, typically 5 to 7 cycles. Then, the request goes to the AXI interconnect that routes the kernel's request to the MIG and then eventually to the DDR. Getting through the interconnect is expensive in latency and can take around 30 cycles. Finally, getting to the DDR and back can cost anywhere from 9 to 14 cycles. These are not precise measurements of latency but rather estimates provided to show the relative latency cost of these specialized modules. For more precise measurements, you can test and observe these latencies using the Application Timeline report for your specific system, as described in [AXI Performance Case Study](#).



TIP: For information about the Application Timeline report, see [Application Timeline in the Vitis Unified Software Platform Documentation](#).

Another way to view the latencies in the system is as follows: the interconnect has an average II of 2 while the DDR controller has an average II of 4-5 cycles on requests (while on the data they are both II=1). The interconnect arbitration strategy is based on the size of read/write requests, and so data requested with longer burst lengths get prioritized over requests with shorter bursts (thus leading to a bigger channel bandwidth being allocated to longer bursts in case of contention). Of course, a large burst request has the side-effect of preventing anyone else from accessing the DDR, and therefore there must be a compromise between burst length and reducing DDR port contention. Fortunately, the large latencies help prevent some of this port contention, and effective pipelining of the requests can significantly improve the bandwidth throughput available in the system.

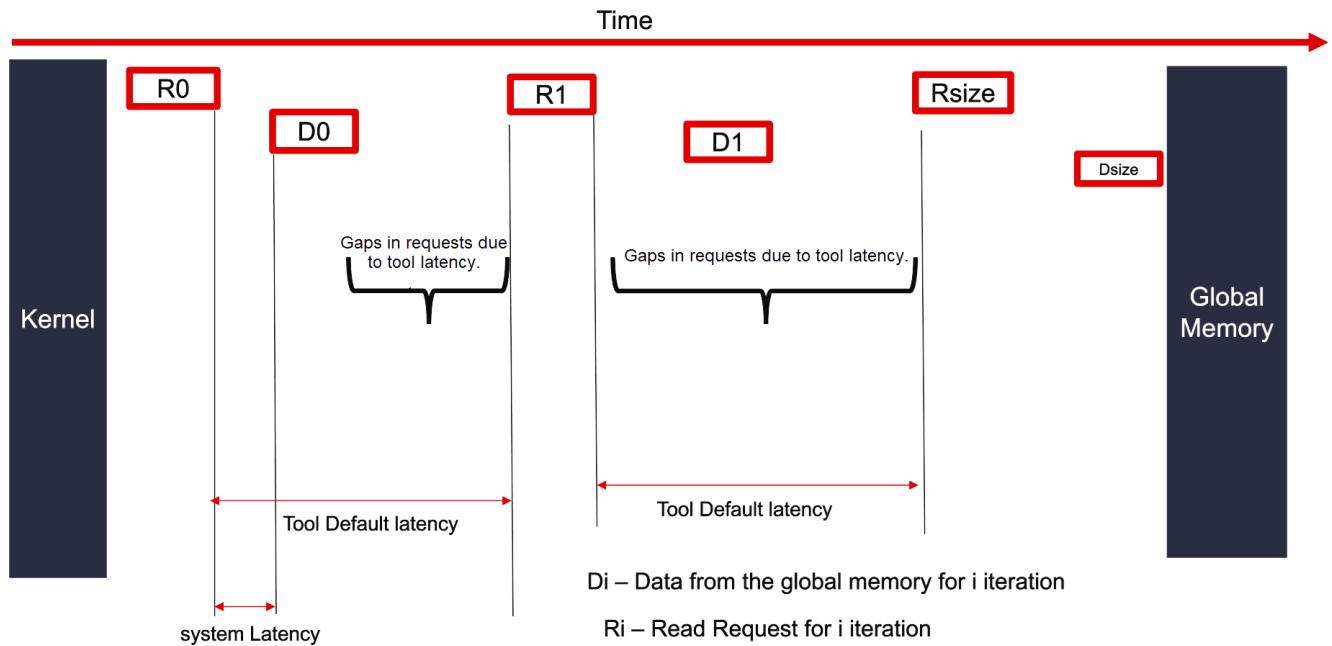
Latency does not affect loops/functions with pipelined bursts since the burst requests the maximum size in a single request.

Latency effects loops/functions with sequential burst in two possible ways:

- If the system read/write latency is larger than the default tool latency, Vitis HLS has to wait for the data. Changing the LATENCY pragma or directive will not improve the performance of the system.

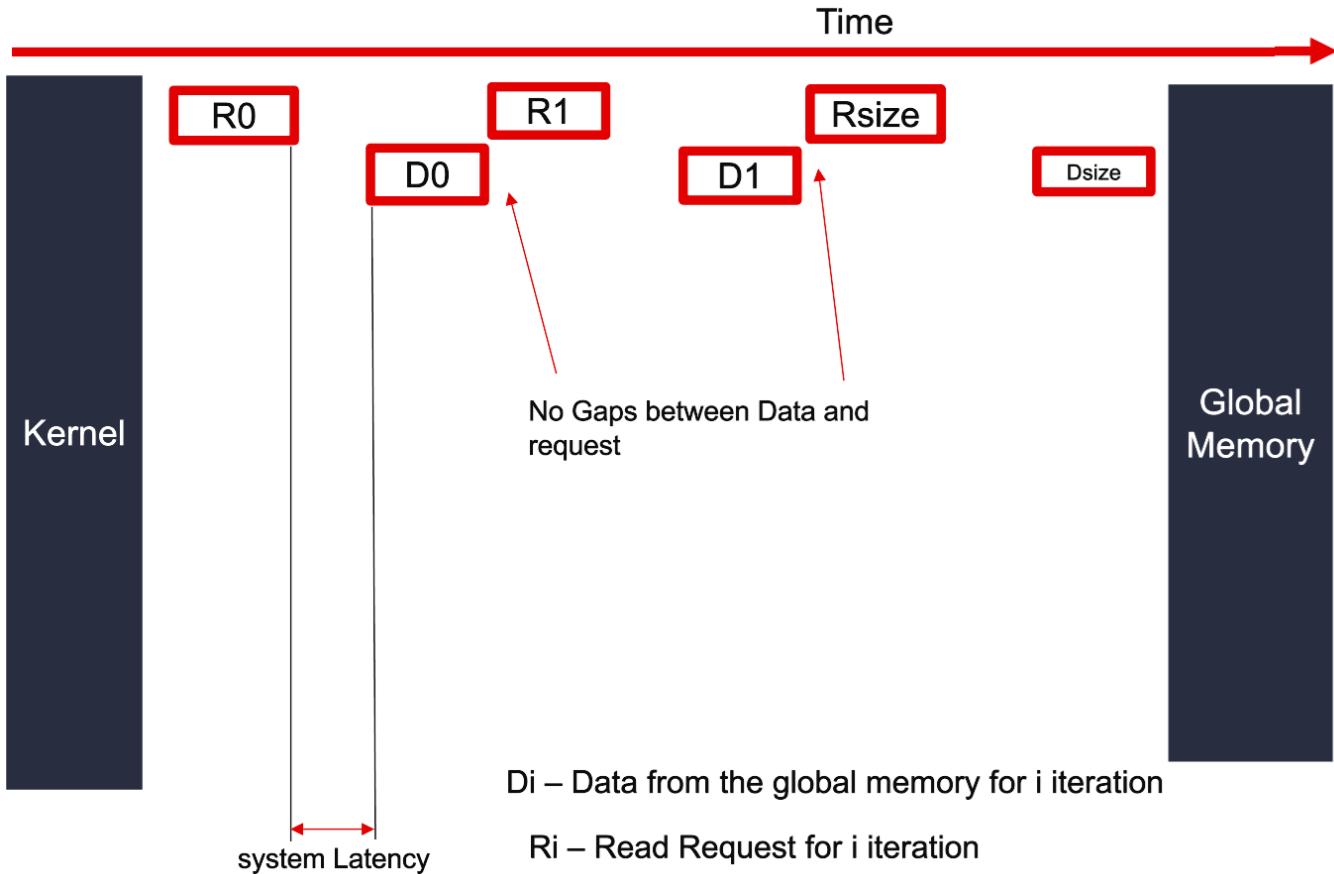
- If the read/write latency is less than the tool default, then Vitis HLS sits in an idle state and wastes the remaining kernel cycles. This can impact the performance of the design because during this idle state it does not perform tasks. As you can see from the figure below the difference between the system latency and the default latency parameter will cause the sequential requests to be delayed further in time. This causes a significant loss of throughput.

Figure 66: Default Tool Latency



However, when you reduce the tool latency using the LATENCY pragma or directive, Vitis HLS will tightly pack the requests for a sequential burst, as shown in the following figure.

Figure 67: Adjusted Tool Latency



RECOMMENDED: *Latency has a significant impact on sequential burst. Decreasing the default tool latency can improve the system performance.*

Port Width

The throughput of load-store functions can be further improved by maximizing the number of bytes transferred. Vitis HLS supports kernel ports up to 512 bits wide, which means that a kernel can read or write up to 64 bytes per clock cycle per port.

RECOMMENDED: *You should maximize the port width of the interface, i.e., the bit-width of each AXI port, by setting it to 512 bits (64 bytes).*

Vitis HLS also supports automatic port width optimization by analyzing the memory access pattern of the source code. If the code satisfies the preconditions and limitations for burst access, it will automatically resize the port to 512 bit width in the Vitis kernel flow.

IMPORTANT! *If the size and number of iterations are variable at compile time, then the tool will not automatically widen port widths.*

If the tool cannot automatically widen the port, you can manually change the port width by using [Vector Data Types](#) or [Arbitrary Precision \(AP\) Data Types](#) as the data type of the port.

Multiple Ports

The throughput of load-store functions can be further improved by maximizing concurrent read/writes. In Vitis HLS, the function arguments by default are bundled/mapped/grouped to a single port. Bundling ports into a single port helps save resources. However, a single port can limit the performance of the kernel because all the memory transfers have to go through a single port. The `m_axi` interface has independent READ and WRITE channels, so a single port can read and write simultaneously.

Using multiple ports lets you increase the bandwidth and throughput of the kernel by creating multiple interfaces to connect to different memory banks, as shown in the [Multi-DDR tutorial](#), or the accesses will be sequential. When multiple arguments are accessing the same memory port or memory bank, an arbiter will sequence the concurrent accesses to the same memory port or bank. Having multiple ports connected to different memory banks increases the throughput of the load and store functions, and as a result, the compute block should also be equally scaled to meet the throughput demand from the load and store functions otherwise it will put back-pressure or stalls on the load-store functions.



RECOMMENDED: Analyze the concurrent memory reads/writes and have a dedicated/independent port for concurrent accesses.

Number of Outstanding Reads/Writes

The throughput of load-store functions can be further improved by allowing the system to hide some of the memory latency. The `m_axi_num_read_outstanding` and `m_axi_num_write_outstanding` options of the `config_interface` command, or of the INTERFACE pragma or directive, lets the Kernel control the number of pipelined memory requests sent to the global memory without waiting for the previous request to complete.

Increasing the number of pipelined requests increases the pipeline depth of the read/write requests, which will cost additional BRAM/URAM resources.

Note: In most cases where burst length ≥ 16 , the default number of outstanding reads/writes should be sufficient. For a burst of size less than 16, Xilinx recommends doubling the size of the number of outstanding from the default of 16.

Defining Burst Attributes with the INTERFACE Pragma

To create the optimal AXI4 interface, the following command options are provided in the INTERFACE directive to specify the behavior of the bursts and optimize the efficiency of the AXI4 interface.

Note that some of these options can use internal storage to buffer data and this may have an impact on area and resources:

- **latency:** Specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request several cycles (*latency*) before the read or write is expected. If this figure is too low, the design will be ready too soon and may stall waiting for the bus. If this figure is too high, bus access may be granted but the bus may stall waiting on the design to start the access. Default latency in Vitis HLS is 64.
- **max_read_burst_length:** Specifies the maximum number of data values read during a burst transfer. Default value is 16.
- **num_read_outstanding:** Specifies how many read requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design: a FIFO of size $\text{num_read_outstanding} * \text{max_read_burst_length} * \text{word_size}$. Default value is 16.
- **max_write_burst_length:** Specifies the maximum number of data values written during a burst transfer. Default value is 16.
- **num_write_outstanding:** Specifies how many write requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design: a FIFO of size $\text{num_read_outstanding} * \text{max_read_burst_length} * \text{word_size}$. Default value is 16.

The following INTERFACE pragma example can be used to help explain these options:

```
#pragma HLS interface mode=m_axi port=input offset=slave
bundle=gmem0
depth=1024*1024*16/(512/8) latency=100 num_read_outstanding=32
num_write_outstanding=32
max_read_burst_length=16 max_write_burst_length=16
```

- The interface is specified as having a latency of 100. The HLS compiler seeks to schedule the request for burst access 100 clock cycles before the design is ready to access the AXI4 bus.
- To further improve bus efficiency, the options `num_write_outstanding` and `num_read_outstanding` ensure the design contains enough buffering to store up to 32 read and/or write accesses. Each request will require its own buffer. This allows the design to continue processing until the bus requests are serviced.
- Finally, the options `max_read_burst_length` and `max_write_burst_length` ensure the maximum burst size is 16 and that the AXI4 interface does not hold the bus for longer than this. The HLS tool will partition longer bursts according to the specified burst length, and report this condition with a message like the following:

```
Multiple burst reads of length 192 and bit width 128 in loop
'VITIS_LOOP_2'./src/filter.cpp:247:21)has been inferred on port
'mm_read'.
These burst requests might be further partitioned into multiple requests
during RTL generation based on the max_read_burst_length settings.
```

Commands to Configure the Burst

These commands configure global settings for the tool to optimize the AXI4 interface for the system in which it will operate. The efficiency of the operation depends on these values being set accurately. The provided default values are conservative, and may require changing depending on the memory access profile of your design.

Table 17: Vitis HLS Controls

Vitis HLS Command	Value	Description
config_rtl - m_axi_conservative_mode	bool default=true	Delay M-AXI each write request until the associated write data are entirely available (typically, buffered into the adapter or already emitted). This can slightly increase write latency but can resolve deadlock due to concurrent requests (read or write) on the memory subsystem.
config_interface - m_axi_latency	uint 0 is auto default=0 (for Vivado IP flow) default=64 (for Vitis Kernel flow)	Provide the scheduler with an expected latency for M-AXI accesses. Latency is the delay between a read request and the first read data, or between the last write data and the write response. Note that this number need not be exact, underestimation makes for a lower-latency schedule, but with longer dynamic stalls. The scheduler will account for the additional adapter latency and add a few cycles.
config_interface - m_axi_min_bitwidth	uint default=8	Minimum bitwidth for M-AXI interfaces data channels. Must be a power-of-two between 8 and 1024. Note that this does not necessarily increase throughput if the actual accesses are smaller than the required interface.
config_interface - m_axi_max_bitwidth	uint default=1024	Minimum bitwidth for M-AXI interfaces data channels. Must be a power-of-two between 8 and 1024. Note that this does decrease throughput if the actual accesses are bigger than the required interface as they will be split into a multi-cycle burst of accesses.
config_interface - m_axi_max_widen_bitwidth	uint default=0 (for Vivado IP flow) default=512 (for Vitis Kernel flow)	Allow the tool to automatically widen bursts on M-AXI interfaces up to the chosen bitwidth. Must be a power-of-two between 8 and 1024. Note that burst widening requires strong alignment properties (in addition to burst).
config_interface - m_axi_auto_max_ports	bool default=false	If the option is false, all the M-AXI interfaces that are not explicitly bundled will be bundled into a single common interface, thus minimizing resource usage (single adapter). If the option is true, all the M-AXI interfaces that are not explicitly bundled will be mapped into individual interfaces, thus increasing the resource usage (multiple adapters).

Table 17: Vitis HLS Controls (cont'd)

Vitis HLS Command	Value	Description
config_interface - m_axi_alignment_byte_size	uint default=1 (for Vivado IP flow) default=64 (for Vitis Kernel flow)	Assume top function pointers that are mapped to M-AXI interfaces are at least aligned to the provided width in byte (power of two). This can help automatic burst widening. Warning: behavior will be incorrect if the pointers are not actually aligned at runtime.
config_interface - m_axi_num_read_outstanding	uint default=16	Default value for M-AXI num_read_outstanding interface parameter.
config_interface - m_axi_num_write_outstanding	uint default=16	Default value for M-AXI num_write_outstanding interface parameter.
config_interface - m_axi_max_read_burst_length	uint default=16	Default value for M-AXI max_read_burst_length interface parameter.
config_interface - m_axi_max_write_burst_length	uint default=16	Default value for M-AXI max_write_burst_length interface parameter.

Examples of Recommended Coding Styles

As described in [Synthesis Summary](#), Vitis HLS issues a report summarizing burst activities and also identifying burst failures. If bursts of variable lengths are done, then the report will mention that bursts of variable lengths were inferred. The compiler also provides burst messages that can be found in the compiler log, `vitis_hls.log`. These messages are issued before the scheduling step.

Simple Read/Write Burst Inference

The following example is the standard way of reading and writing to the DDR and inferring a read and write burst. The Vitis HLS compiler will report the following burst inferences for the example below:

```
INFO: [HLS 214-115] Burst read of variable length and bit width 32 has been
inferred on port 'gmem'
INFO: [HLS 214-115] Burst write of variable length and bit width 32 has
been inferred on port 'gmem' (./src/vadd.cpp:75:9).
```

The code for this example follows:

```
***** BEGIN EXAMPLE *****

#define DATA_SIZE 2048
// Define internal buffer max size
#define BURSTBUFSIZE 256

//TRIPCOUNT identifiers
const unsigned int c_min = 1;
const unsigned int c_max = BURSTBUFSIZE;
const unsigned int c_chunk_sz = DATA_SIZE;
```

```

extern "C" {
void vadd(int *a, int size, int inc_value) {
    // Map pointer a to AXI4-master interface for global memory access
#pragma HLS INTERFACE mode=m_axi port=a offset=slave bundle=gmem
max_read_burst_length=256 max_write_burst_length=256
    // We also need to map a and return to a bundled axilite slave interface
#pragma HLS INTERFACE mode=s_axilite port=a bundle=control
#pragma HLS INTERFACE mode=s_axilite port=size bundle=control
#pragma HLS INTERFACE mode=s_axilite port=inc_value bundle=control
#pragma HLS INTERFACE mode=s_axilite port=return bundle=control

    int burstbuffer[BURSTBUFFERSIZE];

    // Per iteration of this loop perform BURSTBUFFERSIZE vector addition
    for (int i = 0; i < size; i += BURSTBUFFERSIZE) {
#pragma HLS LOOP_TRIPCOUNT min=c_min*c_min max=c_chunk_sz*c_chunk_sz/
(c_max*c_max)
        int chunk_size = BURSTBUFFERSIZE;
        //boundary checks
        if ((i + BURSTBUFFERSIZE) > size)
            chunk_size = size - i;

        // memcpy creates a burst access to memory
        // multiple calls of memcpy cannot be pipelined and will be
        scheduled sequentially
        // memcpy requires a local buffer to store the results of the
        memory transaction
        memcpy(burstbuffer, &a[i], chunk_size * sizeof(int));

        // Calculate and write results to global memory, the sequential write
        in a for loop can be
        // inferred as a memory burst access
        calc_write:
        for (int j = 0; j < chunk_size; j++) {
            #pragma HLS LOOP_TRIPCOUNT min=c_size_max max=c_chunk_sz
            #pragma HLS PIPELINE II=1
            burstbuffer[j] = burstbuffer[j] + inc_value;
            a[i + j] = burstbuffer[j];
        }
    }
}
}

```

Pipelining Between Bursts

The following example will infer bursts of length N:

```

for(int x=0; x < k; ++x) {
    int off = f(x);
    for(int i = 0; i < N; ++i) {
        #pragma HLS PIPELINE II=1
        ... = gmem[off + i];
    }
}

```

But notice that the outer loop is not pipelined. This means that while there is pipelining inside bursts, there won't be any pipelining between bursts.

To remedy this you can unroll the inner loop and pipeline the outer loop to get pipelining between bursts as well. The following example will still infer bursts of length N, but now there will also be pipelining between bursts leading to higher throughput:

```
for(int x=0; x < k; ++x) {  
    #pragma HLS PIPELINE II=N  
    int off = f(x);  
    for(int i = 0; i < N; ++i) {  
        #pragma HLS UNROLL  
        ... = gmem[off + i];  
    }  
}
```

Accessing Row Data from a Two-Dimensional Array

The following is an example of reading/writing to/from a two dimensional array. Vitis HLS infers read and write bursts and issues the following messages:

```
INFO: [HLS 214-115] Burst read of length 256 and bit width 512 has been  
inferred on port 'gmem' (./src/row_array_2d.cpp:43:5)  
INFO: [HLS 214-115] Burst write of length 256 and bit width 512 has been  
inferred on port 'gmem' (./src/row_array_2d.cpp:56:5)
```

Notice that a bit width of 512 is achieved in this example. This is more efficient than the 32 bit width achieved in the simple example above. Bursting wider bit widths is another way bursts can be optimized as discussed in [Automatic Port Width Resizing](#).

The code for this example follows:

```
***** BEGIN EXAMPLE *****/  
// Parameters Description:  
//      NUM_ROWS:          matrix height  
//      WORD_PER_ROW:       number of words in a row  
//      BLOCK_SIZE:         number of words in an array  
#define NUM_ROWS   64  
#define WORD_PER_ROW 64  
#define BLOCK_SIZE (WORD_PER_ROW*NUM_ROWS)  
  
// Default datatype is integer  
typedef int DTTYPE;  
typedef hls::stream<DTTYPE> my_data_fifo;  
  
// Read data function: reads data from global memory  
void read_data(DTTYPE *inx, my_data_fifo &inFifo) {  
    read_loop_i:  
        for (int i = 0; i < NUM_ROWS; ++i) {  
            read_loop_jj:  
                for (int jj = 0; jj < WORD_PER_ROW; ++jj) {  
                    #pragma HLS PIPELINE II=1  
                    inFifo << inx[WORD_PER_ROW * i + jj];  
                }  
        }  
    }  
  
    // Write data function - writes results to global memory  
    void write_data(DTTYPE *outx, my_data_fifo &outFifo) {
```

```

write_loop_i:
    for (int i = 0; i < NUM_ROWS; ++i) {
        write_loop_jj:
            for (int jj = 0; jj < WORD_PER_ROW; ++jj) {
                #pragma HLS PIPELINE II=1
                outFifo >> outx[WORD_PER_ROW * i + jj];
            }
    }
}

// Compute function is pretty simple because this example is focused on
// efficient
// memory access pattern.
void compute(my_data_fifo &inFifo, my_data_fifo &outFifo, int alpha) {
compute_loop_i:
    for (int i = 0; i < NUM_ROWS; ++i) {
        compute_loop_jj:
            for (int jj = 0; jj < WORD_PER_ROW; ++jj) {
                #pragma HLS PIPELINE II=1
                DTTYPE inTmp;
                inFifo >> inTmp;
                DTTYPE outTmp = inTmp * alpha;
                outFifo << outTmp;
            }
    }
}

extern "C" {
    void row_array_2d(DTTYPE *inx, DTTYPE *outx, int alpha) {
        // AXI master interface
        #pragma HLS INTERFACE mode=m_axi port = inx offset = slave bundle = gmem
        #pragma HLS INTERFACE mode=m_axi port = outx offset = slave bundle =
gmem
        // AXI slave interface
        #pragma HLS INTERFACE mode=s_axilite port = inx bundle = control
        #pragma HLS INTERFACE mode=s_axilite port = outx bundle = control
        #pragma HLS INTERFACE mode=s_axilite port = alpha bundle = control
        #pragma HLS INTERFACE mode=s_axilite port = return bundle = control

        my_data_fifo inFifo;
        // By default the FIFO depth is 2, user can change the depth by
using
        // #pragma HLS stream variable=inFifo depth=256
        my_data_fifo outFifo;

        // Dataflow enables task level pipelining, allowing functions and
loops to execute
        // concurrently. For more details please refer to UG902.
        #pragma HLS DATAFLOW
        // Read data from each row of 2D array
        read_data(inx, inFifo);
        // Do computation with the acquired data
        compute(inFifo, outFifo, alpha);
        // Write data to each row of 2D array
        write_data(outx, outFifo);
        return;
    }
}

```

Summary

Write code in such a way that bursting can be inferred. Ensure that none of the preconditions are violated.

Bursting does not mean that you will get all your data in one shot – it is about merging the requests together into one request, but the data will arrive sequentially, one after another.

Burst length of 16 is ideal, but even burst lengths of 8 are enough. Bigger bursts have more latency while shorter bursts can be pipelined. Do not confuse bursting with pipelining, but note that bursts can be pipelined with other bursts.

If your bursts are of fixed length, you can unroll the inner loop where bursts are inferred and pipeline the outer loop. This will achieve the same burst length, but also pipelining between the bursts to enable higher throughput.

For greater throughput, focus on widening the interface up to 512 bits rather than simply achieving longer bursts.

Bigger bursts have higher priority with the AXI interconnect. No dynamic arbitration is done inside the kernel.

You can have two `m_axi` ports connected to same DDR to model mutually exclusive access inside kernel, but the AXI interconnect outside the kernel will arbitrate competing requests.

One way to get around the out-of-order access restriction is to create your own buffer in BRAM, store the bursts in this buffer and then use this buffer to do out of order accesses. This is typically called a line buffer and is a common optimization used in video processing.

Review the Burst Optimization section of the [Synthesis Summary](#) report to learn more about burst optimizations in the design, and missed burst opportunities. If automatic burst is not occurring in your design, you may want to use the `hls::burst_maxi` data type for manual burst, as described in [Using Manual Burst](#).

Using Manual Burst

Burst transfers improve the throughput of the I/O of the kernel by reading or writing large chunks of data to the global memory. The larger the size of the burst, the higher the throughput, this metric is calculated as follows ((# of bytes transferred) * (kernel frequency)/(Time)). The maximum kernel interface bitwidth is 512 bits and if the kernel is compiled at 300 MHz, then it can theoretically achieve = (80-95% efficiency of the DDR)*(512* 300 Mhz)/1 sec = ~17-19 GB/s for a DDR. As explained, Vitis HLS performs automatic burst optimizations which intelligently aggregates the memory accesses of the loops/functions from the user code and performs read/write of a particular size in a single burst request. However, burst transfer also has requirements that can sometimes be overburdening or difficult to meet, as discussed in

Preconditions and Limitations of Burst Transfer. In such cases, if you are familiar with the AXI4 `m_axi` protocol and understand hardware transaction modeling you can implement manual burst transfers using the `hls::burst_maxi` class as described below. Refer to [Vitis-HLS-Introductory-Examples/Interface/Memory/manual_burst](#) on Github for examples of these concepts.

hls::burst_maxi Class

The `hls::burst_maxi` class provides a mechanism to perform read/write access to the DDR. These methods will translate the class methods usage behavior into respective AXI4 protocol and send and receive requests on the AXI4 bus signals - AW, AR, WDATA, BVALID, RDATA. These methods control the burst behavior of the HLS scheduler. The adapter, which receives the commands from the scheduler, is responsible for sending the data to the DDR. These requests will adhere to the user specified INTERFACE pragma options, such as `max_read_burst_length` and `max_write_burst_length`. The class methods should only be used in the kernel code, and not in the test bench (except for the class constructor as described below).

- Constructors:

- `burst_maxi(const burst_maxi &B) : Ptr(B.Ptr) {}`
- `burst_maxi(T *p) : Ptr(p) {}`



IMPORTANT! The HLS design and test bench must be in different files, because the constructor `burst_maxi(T *p)` is only available in C-simulation model.

- Read Methods:

- `void read_request(size_t offset, size_t len);`

This method is used to perform a read request to the `m_axi` adapter. The function returns immediately if the read request queue inside `m_axi` adapter is not full, otherwise it waits until space becomes available.

- `offset`: Specify the memory offset from which to read the data
- `len`: Specify the scheduler burst length. This burst length is sent to the adapter, which can then convert it to the standard AXI AMBA protocol

- `T read();`

This method is used to transfer the data from the `m_axi` adapter to the scheduler FIFO. If the data is not available, `read()` will be blocking. The `read()` method should be called `len` number of times, as specified in the `read_request()`.

- Write Methods:

- `void write_request(size_t offset, size_t len);`

This method is used to perform a write request to the `m_axi` adapter. The function returns immediately if the write request queue inside `m_axi` adapter is not full.

- `offset`: Specify the memory offset into which the data should be written
- `len`: Specify the scheduler burst length. This burst length is sent to the adapter, which can then convert it to the standard AXI AMBA protocol
- `void write(const T &val, ap_int<sizeof(T)> byteenable_mask = -1);`

This method is used to transfer data from the internal buffer of the scheduler to the `m_axi` adapter. It blocks if the internal write buffer is full. The `byteenable_mask` is used to enable the bytes in the WDATA. By default it will enable all the bytes of the transfer. The `write()` method should be called `len` number of times, as specified in the `write_request()`.

- `void write_response();`

This method blocks until all write responses are back from the global memory. This method should be called the same number of times as `write_request()`.

Using Manual Burst in HLS Design

In the HLS design, when you find that automatic burst transfers are not working as desired, and you cannot optimize the design as needed, you can implement the read and write transactions using the `hls::burst_maxi` object. In this case, you will need to modify your code to replace the original pointer argument with `burst_maxi` as a function argument. These arguments must be accessed by the explicit `read` and `write` methods of the `burst_maxi` class, as shown in the following examples.

The following shows an original code sample, which uses a pointer argument to read data from global memory.

```
void dut(int *A) {
    for (int i = 0; i < 64; i++) {
        #pragma pipeline II=1
        ... = A[i]
    }
}
```

In the modified code below, the pointer is replaced with the `hls::burst_maxi<>` class objects and methods. In the example, the HLS scheduler puts 4 requests of `len 16` from port A to the `m_axi` adapter. The Adapter stores them inside a FIFO and whenever the AW/AR bus is available it will send the request to the global memory. In the 64 loop iterations, the `read()` command issues a blocking call that will wait for the data to come back from the global memory. After the data becomes available the HLS scheduler will read it from the `m_axi` adapter FIFO.

```
#include "hls_burst_maxi.h"
void dut(hls::burst_maxi<int> A) {
    // Issue 4 burst requests
    A.read_request(0, 16); // request 16 elements, starting from A[0]
    A.read_request(128, 16); // request 16 elements, starting from A[128]
    A.read_request(256, 16); // request 16 elements, starting from A[256]
    A.read_request(384, 16); // request 16 elements, starting from A[384]
    for (int i = 0; i < 64; i++) {
        #pragma pipeline II=1
        ... = A.read(); // Read the requested data
    }
}
```

In example 2 below, the HLS scheduler/kernel puts 2 requests from port A to the adapter, the first request of `len 2`, and the second request of `len 1`, for a total of 2 write requests. It then issues corresponding, since the total burst length is 3 write commands. The Adapter stores these requests inside a FIFO and whenever the AW, W bus is available it will send the request and data to the global memory. Finally, two `write_response` commands are used, to await response for the two `write_requests`.

```
void trf(hls::burst_maxi<int> A) {
    A.write_request(0, 2);
    A.write(x); // write A[0]
    A.write_request(10, 1);
    A.write(x, 2); // write A[1] with byte enable 0010
    A.write(x); // write A[10]
    A.write_response(); // response of write_request(0, 2)
    A.write_response(); // response of write_request(10, 1)
}
```

Using Manual Burst in C-Simulation

You can pass a regular array to the top function, and the array will be transformed to `hls::burst_maxi` automatically by the constructor.



IMPORTANT! The HLS design and test bench must be in different files, because the `burst_maxi(T *p)` constructor is only valid for use in C simulation model.

```
#include "hls_burst_maxi.h"
void dut(hls::burst_maxi<int> A);

int main() {
    int Array[1000];
    dut(Array);
    .....
}
```

Using Manual Burst to Optimize Performance

Vitis HLS characterizes two types of burst behaviors: pipeline burst, and sequential burst.

- **Pipeline Burst:** Pipeline Burst improves throughput by reading or writing the maximum amount of data in a single request. The compiler infers pipeline burst if the `read_request`, `write_request` and `write_response` calls are outside the loop, as shown in the following code example. In the below example the size is a variable that is sent from the testbench.

```
9 int buf[8192];
10 in.read_request(0, size);
11 for (int i = 0; i < size; i++) {
12 #pragma HLS PIPELINE II=1
13     buf[i] = in.read();
14     out.write_request(0, size*NT);
17     for (int i = 0; i < NT; i++) {
19         for (int j = 0; j < size; j++) {
20             #pragma HLS PIPELINE II=1
21                 int a = buf[j];
22                 out.write(a);
23 }
24 }
25 out.write_response();
```

Figure 68: Synthesis Results

M_AXI Burst Information				
Manual Burst (burst_maxi)				
HW Interface	Direction	Length	Width	Location
m_axi_gmem	read	variable	32	example.cpp:10:12
m_axi_gmem	write	variable	32	example.cpp:16:7

As you can see from the figure above, the tool has inferred the burst from the user code and length is mentioned as variable at compile time.

Figure 69: Performance Benefits



During the run time the HLS compiler sends a burst request of `length = size` and the adapter will partition them into the user-specified `burst_length` pragma option. In this case the default burst length is set to 16, which is used in the `ARlen` and `AWlen` channels. The read/write channel achieved maximum throughput since there are no bubbles during the transfer.

Figure 70: Co-sim Results

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
transfer_kernel	5735	5735	5735			
transfer_kernel_Pipeline_VITIS_LOOP_11_1	517	517	517			
transfer_kernel_Pipeline_VITIS_LOOP_18_2_VITIS_LOOP_19_3	5122	5122	5122			

- Sequential Burst:

This burst is a sequential burst of smaller data sizes, where the read requests, write requests and write responses are inside the loop body as shown in the below snippet. The drawback of the sequential burst is that the future request ($i+1$) depends on the previous request (i) to finish since it is waiting for the read request, write request and write response to complete, this will cause gaps between requests. Sequential burst is not as effective as pipeline burst because it is reading or writing a small data size multiple times to compensate for the loop bounds. Although this will limit the improvement to throughput, sequential burst is still better than no burst.

```

void transfer_kernel(hls::burst_maxi<int> in, hls::burst_maxi<int> out,
const int size )
{
    #pragma HLS INTERFACE m_axi port=in depth=512 latency=32 offset=slave
    #pragma HLS INTERFACE m_axi port=out depth=5120 offset=slave latency=32
}
    
```

```
int buf[8192];

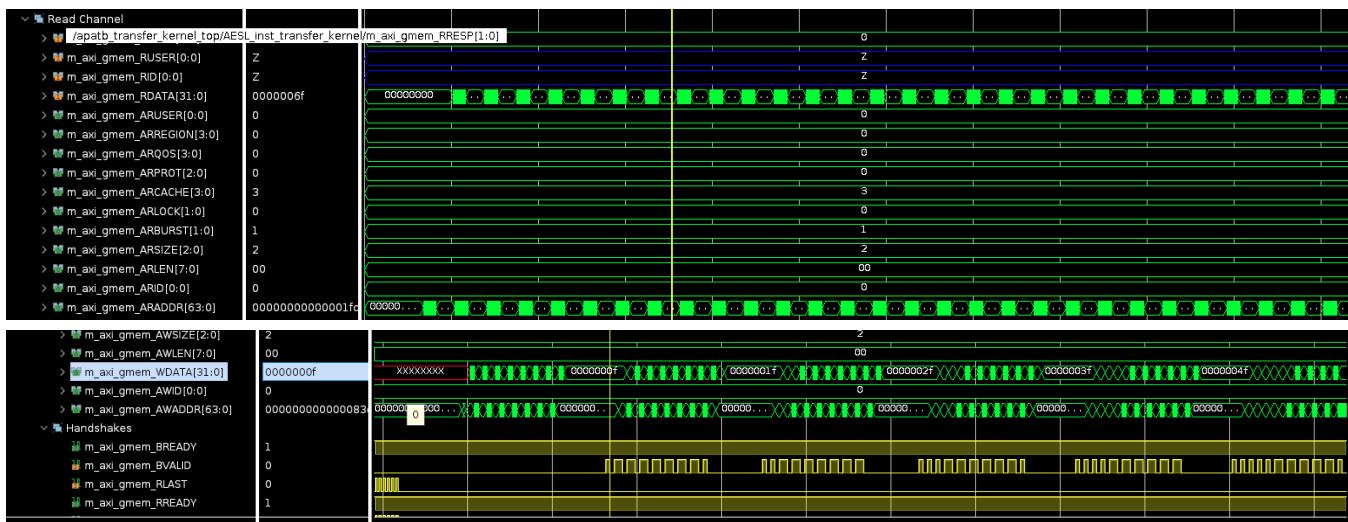
for (int i = 0; i < size; i++) {
    in.read_request(i, 1);
#pragma HLS PIPELINE II=1
    buf[i] = in.read();
}

for (int i = 0; i < NT; i++) {
    for (int j = 0; j < size; j++) {
        out.write_request(j, 1);
#pragma HLS PIPELINE II=1
        int a = buf[j];
        out.write(a);
        out.write_response();
    }
}
}
```

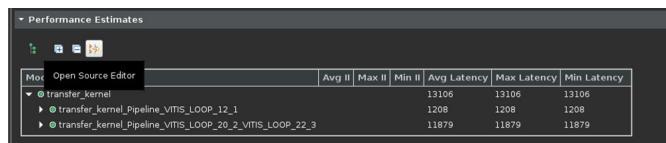
Figure 71: Synthesis Results

M_AXI Burst Information				
Manual Burst (burst_maxi)				
HW Interface	Direction	Length	Width	Location
m_axi_gmem	read	1	32	example.cpp:13:8
m_axi_gmem	write	1	32	example.cpp:23:15

As you can see from the report sample above, the tool achieved a burst of length =1.

Figure 72: Performance Impacts


The read/write loop R/WDATA channel has gaps equal to read/write latency, as discussed in [AXI4 Master Interface](#). For the read channel, the loop waits for all the read data to come back from the global memory. For the write channel, the innermost loop waits for the response (BVALID) to come back from the global memory. This results in performance degradation. The co-sim results also show that a 2x degradation in performance for this burst semantics.

Figure 73: Performance Estimates


Features and Limitations

- If the `m_axi` element is a struct:
 - The struct will be packed into a wide int. Disaggregation of the struct is not allowed.
 - The size of struct must be a power-of-2, and should not exceed 1024 bits or the max width specified by the `config_interface -m_axi_max_bitwidth` command.
- ARRAY_PARTITION and ARRAY_RESHAPE of burst_maxi ports is not allowed.
- You can apply the INTERFACE pragma or directive to `hls::burst_maxi`, defining an `m_axi` interface. If the `burst_maxi` port is bundled with other ports, all ports in this bundle must be `hls::burst_maxi` and must have the same element type.

```
void dut(hls::burst_maxi<int> A, hls::burst_maxi<int> B, int *C,
hls::burst_maxi<short> D) {
    #pragma HLS interface m_axi port=A offset=slave bundle=gmem // OK
    #pragma HLS interface m_axi port=B offset=slave bundle=gmem // OK
    #pragma HLS interface m_axi port=C offset=slave bundle=gmem // Bad. C
```

```
must also be hls::burst_maxi type, because it shares the same bundle
'gmem' with A and B
#pragma HLS interface m_axi port=D offset=slave bundle=gmem // Bad. D
should have 'int' element type, because it shares the same bundle
'gmem' with A and B
}
```

4. You can use the INTERFACE pragma or directive to specify the num_read_outstanding and num_write_outstanding, and the max_read_burst_length and max_write_burst_length to define the size of the internal buffer of the m_axi adapter.

```
void dut(hls::burst_maxi<int> A) {
    #pragma HLS interface m_axi port=A num_read_outstanding=32
    num_write_outstanding=32 max_read_burst_length=16
    max_write_burst_length=16
}
```

5. The INTERFACE pragma or directive max_widen_bitwidth is not supported, because HLS will not change the bit width of hls::burst_maxi ports.
6. You must make a read_request before read, or write_request before write:

```
void dut(hls::burst_maxi<int> A) {
    ... = A.read(); // Bad because read() before read_request(). You can
    catch this error in C-sim.
    A.read_request(0, 1);
}
```

7. If the address and life time of the read group (read_request() > read()) and write group (write_request() > write() > write_response()) overlap, the tool cannot guarantee the access order. C-simulation will report an error.

```
void dut(hls::burst_maxi<int> A) {
    A.write_request(0, 1);
    A.write(x);
    A.read_request(0, 1);
    ... = A.read(); // What value is read? It is undefined. It could be
    original A[0] or updated A[0].
    A.write_response();
}

void dut(hls::burst_maxi<int> A) {
    A.write_request(0, 1);
    A.write(x);
    A.write_response();
    A.read_request(0, 1);
    ... = A.read(); // this will read the updated A[0].
}
```

8. If multiple hls::burst_maxi ports are bundled to same m_axi adapter and their transaction lifetimes overlap, the behavior is unexpected.

```
void dut(hls::burst_maxi<int> A, hls::burst_maxi<int> B) {
    #pragma HLS INTERFACE m_axi port=A bundle=gmem depth = 10
    #pragma HLS INTERFACE m_axi port=B bundle=gmem depth = 10
    A.read_request(0, 10);
    B.read_request(0, 10);

    for (int i = 0; i < 10; i++) {
```

```
#pragma HLS pipeline II=1
.... = A.read(); // get value of A[0], A[2], A[4] ...
.... = B.read(); // get value of A[1], A[3], A[5] ...
}
}
```

9. Read or write requests and read or writes in different dataflow process are not supported. Dataflow checker will report an error: multiple writes in different dataflow processes are not allowed.

For example:

```
void transfer(hls::burst_maxi<int> A) {
#pragma HLS dataflow
    IssueRequests(A); // issue multiple write_request() of A
    Write(A); // multiple writes to A
    GetResponse(A); // write_response() of A
}
```

Potential Pitfalls

The following are some concerns you must be aware of when implementing manual burst techniques:

- Deadlock: Improper use of manual burst can lead to deadlocks.

Too many `read_request` before `read()` commands will cause deadlock because the `read_request` loop will push the request into the read requests FIFO, and this FIFO will only be emptied after the read from the global memory is completed. The job of the `read()` command is to read the data from the adapter FIFO and mark the request done, after which the `read_request` will be popped from the FIFO and a new request can be pushed onto it.

```
//reads/writes. will deadlock if N is larger
for (i = 0; i < N; i++)
{   A.read_request(i * 128, 16);}
for (i = 0; i < 16 *N; i++) { ... = A.read();}

for (int i = 0; i < N; i++) {
    p.write_request(i * 128, 16);
}

for (int i = 0; i < N * 16; i++) {
    p.write(i);
}

for (int i = 0; i < N; i++) {
    p.write_response();
}
```

In the example above, if N is large then the `read_request` and `read` FIFO will be full as it tends to $N/2$. The `read request` loop would not finish, and the `read` command loop wouldn't start, which results in deadlock.

Note: This is case also true for `write_request()` and `write()` commands.

- AXI protocol violation: There should be an equal number of write requests and write responses. An unequal number of requests and responses would lead to AXI protocol violation

AXI Performance Case Study

Introduction

The objective of the case study is to show a step-by-step optimization to improve the throughput of the read/write loops/functions using HLS metrics. These optimizations will improve the kernel time and throughput of the system by performing efficient data transfers from global memory to the kernel. The `transfer_kernel` example below performs a DDR simple read/write (of variable size and NUM_ITERATIONS).



TIP: The host code, which is not shown, only transfers the data and enqueues the kernel in an in-order queue.

```
1 #include "config.h"
2 #include "assert.h"
3 extern "C" {
4     void transfer_kernel(wd* in,wd* out, const int size, const int iter )
{
5 ...
6     wd buf[256];
7     int off = (size/16);
8
9     read_loop: for (int i = 0; i < off; i++)
10    {
11        buf[i] = in[i];
12    }
13
14    write_loop: L1: for (int i = 0; i < iter; i++) {
15        L2: for (int j = 0; j < off; j++) {
16            #pragma HLS PIPELINE II=1
17            out[j+off*i] = buf[j];
18        }
19    }
20 ...
21 }
22 }
```

This case study is divided into 4 steps:

1. Baseline kernel run time with port width set to 512-bit width
2. Improve performance by changing latency parameter
3. Improve the auto burst inference of the write loop.
4. No further improvements using multiple ports and number write outstanding

Step 1: Baseline the Kernel with 512-bit Port Width

Baseline the kernel time using the default settings. During this run, the auto burst inferences the following for the read and write loops:

- The Read loop achieves the pipeline burst since the tool can predict the consecutive memory access pattern. So the pipelined requests to read from the DDR of variable size.
- The Write outer loop, L1, gets sequential burst because the compiler iterates over all the combinations and identifies that since the size is unknown at compile-time, it inserts an if condition in the L1 loop before the start of the L2 loop. At the same time, the inner-most loop - L2 achieves pipeline burst. The L2 loop requests a write request of variable size, while L1 waits for all the data of L2 Loop to come back from the DDR to start the next iteration of L1.

After building and running the application, the performance can be evaluated using the Vitis Analyzer tool to view the reports generated by the build process or the run summary. Review the Burst Summary available in the Synthesis Report from Vitis HLS. It confirms the success and failures of the burst for the Read loop and Write loops.

Figure 74: Synthesis Report - Burst Summary

BURST SUMMARY			
HW Interface	Loop	Message	
m_axi_gmem	VITIS_LOOP_11_1	Multiple burst reads of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the memory controller's burst width.	
m_axi_gmem	VITIS_LOOP_17_3	Multiple burst writes of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the memory controller's burst width.	
BURSTS AND WIDENING MISSED			
HW Interface	Variable	Loop	Problem
m_axi_gmem	out	VITIS_LOOP_16_2	Access call is in the conditional branch
m_axi_gmem	out		Could not widen since the size of type 'i512' is greater than or equal to the max_widen_bitwidth threshold of '64'.
m_axi_gmem	in		Could not widen since the size of type 'i512' is greater than or equal to the max_widen_bitwidth threshold of '64'.

In Vitis Analyzer, the Profile Summary and Timeline Trace reports are also useful tools to analyze the performance of the FPGA-accelerated application. In the Profile Summary the **Kernels & Compute Unit: Kernel Execution** reports the total time required by the `transfer_kernel` in the baseline build.

Figure 75: Profile Summary - Kernel Execution

Kernels & Compute Units					
Kernel Execution					
Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
transfer_kernel	300	1683.150	3.766	5.610	7.947

Step 2: Improve Performance Latency

Vitis HLS uses the default latency of 64 kernel cycles, which in some cases may be too high. The latency depends on the system characteristics. For this example, the latency is reduced from the default to 21 kernel cycles. The code can be changed to specify the latency using the **INTERFACE** pragma or directive as shown in the following example:

```

1 #include "config.h"
2 #include "assert.h"
3 extern "C" {
4     void transfer_kernel(wd* in,wd* out, const int size, const int iter )
{
5     #pragma HLS INTERFACE m_axi port=in0_index offset=slave latency=21
6     #pragma HLS INTERFACE m_axi port=out offset=slave latency=21
7 ...

```

Build and run the application and use Vitis Analyzer to review the reports generated by the build process or the run summary. Review the Synthesis Report from Vitis HLS, and examine the **HW Interface** table to see the specified latency has been applied.

Figure 76: Synthesis Report - HW Interface

M_AXI										
Interface	Data Width (SW->HW)	Address Width	Latency	Offset	Register	Max Widen Bitwidth	Max Read Burst Length	Max Write Burst Length	Num Read Outstanding	
m_axi_gmem	512 -> 512	64	21	slave	0	512	16	16	16	

Review the **Burst Summary** to examine the success or failures of that process.

Figure 77: Synthesis Report - Burst Summary 2

BURST SUMMARY					
HW Interface	Variable	Loop	Message	Resolution	Location
m_axi_gmem		VITIS_LOOP_11_1	Multiple burst reads of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the width of the memory interface.		
m_axi_gmem		VITIS_LOOP_17_3	Multiple burst writes of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the width of the memory interface.		
BURSTS AND WIDENING MISSED					
HW Interface	Variable	Loop	Problem	Resolution	Location
m_axi_gmem	out	VITIS_LOOP_16_2	Access call is in the conditional branch	214-232	krnl_tf.cpp:16:22
m_axi_gmem	out		Could not widen since the size of type '512' is greater than or equal to the max_widen_bitwidth threshold of '64'.		krnl_tf.cpp:17:27
m_axi_gmem	in		Could not widen since the size of type '512' is greater than or equal to the max_widen_bitwidth threshold of '64'.		krnl_tf.cpp:11:26

Examine the Kernel Execution in the Profile Summary report, and notice the performance improvement due to setting the latency for the interface.

Figure 78: Profile Summary - Kernel Execution 2

Kernels & Compute Units						
Kernel Execution						
Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)	
transfer_kernel	300	1656.180	3.445	5.521	8.002	

Step 3: Improve the Automatic Burst Inference of the Write Loop

The compiler is pessimistic in auto burst inference because size and loop trip counts are unknown at compile time. You can modify the code to help the compiler infer pipelined burst, as shown below.

```
1 #include "config.h"
2 #include "assert.h"
3 extern "C" {
4     void transfer_kernel(wd* in,wd* out, const int size, const int
iter ) {
5         #pragma HLS INTERFACE m_axi port=in offset=slave latency=21
6         #pragma HLS INTERFACE m_axi port=out offset=slave latency=21
7
8         int k=0;
9         wd buf[256];
10        int off = (size/16);
11
12        read_loop: for (int i = 0; i <off; i++)
13        {
14            buf[i] = in[i];
15        }
16
17        write_loop: for (int j = 0; j <off*iter; j++) {
18            #pragma HLS PIPELINE II=1
19            out[k++] = buf[j%off];
20        }
21    }
22 }
```

Build and run the application and use Vitis Analyzer to review the reports generated by the build process or the run summary. The Synthesis Report confirms that the burst hints to the compiler fixed the sequential burst of the write loop. The **Burst and Widening Missed** messages are related to widening ports to 512 bits. Since this example already has a 512 port width, it can be ignored. If the width isn't 512-bit in your code, you might need to focus on resolving these messages.

Figure 79: Synthesis Report - Burst Summary 3

BURST SUMMARY			
HW Interface	Loop	Message	
m_axi_gmem	VITIS_LOOP_12_1	Multiple burst reads of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the memory access pattern.	
m_axi_gmem	VITIS_LOOP_18_2	Multiple burst writes of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the memory access pattern.	
BURSTS AND WIDENING MISSED			
HW Interface	Variable	Problem	Location
m_axi_gmem	out	Could not widen since the size of type 'i512' is greater than or equal to the max_widen_bitwidth threshold of '64'.	krnl_tf.cpp:18:27
m_axi_gmem	in	Could not widen since the size of type 'i512' is greater than or equal to the max_widen_bitwidth threshold of '64'.	krnl_tf.cpp:12:26

Examine the Kernel Execution in the Profile Summary report, and notice the performance improvement due to the latency change from Step 2, and the pipeline burst for the write loop in the current step.

Figure 80: Profile Summary - Kernel Execution 3

Kernels & Compute Units						
Kernel Execution						
Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)	
⚡ transfer_kernel	300	1206.800	1.748	4.023	6.720	

Summary

There are no further improvements that can be made from the Vitis HLS interface metrics. The case study example does not have concurrent read or write, so targeting multiple ports will not help in this case. In this example the tool has achieved pipeline burst for the maximum throughput, so the number of outstanding reads and writes can also be ignored. No further improvements can be confirmed from the kernel time.

As seen in the case study, implementing efficient load-store functions is dependent on the HLS interface metrics of port width, burst access, latency, multiple ports, and the number of outstanding reads and writes. Xilinx recommends the following guidelines for improving your system performance:

- Port width: Maximize the port width of the interface, i.e., the bit-width of each AXI port, by using `hls::vector` or `ap_(u)int<512>` as the data type of the port.
- Multiple ports: Analyze the concurrent memory reads/writes and have a dedicated/independent port for concurrent accesses.
- Pipeline burst: The AXI latency parameter does not have an impact on pipelined burst, the user is advised to write code to achieve the pipelined burst which can significantly improve the performance.
- Sequential burst: The AXI latency parameter has a significant impact on sequential burst, decreasing the latency number from the default latency of the tool will improve the performance.
- Num outstanding: In most of the cases of burst length ≥ 16 , the default num outstanding should be sufficient. For a burst of size less than 16, Xilinx recommends doubling the size of the num outstanding from the default(=16).
- Data Re-ordering: Achieving pipelined burst is always recommended, but at times because of the memory access pattern compiler can achieve only a sequential burst. In order to improve the performance, the developer can also consider different ways of storing the data in memory. For instance, accessing data in DRAM in a column-major fashion can be very inefficient. Rather than implementing a dedicated data-mover in the kernel, it may be better to transpose the data in SW and store in row-major order instead which will greatly simplify HW access patterns.

Managing Area and Hardware Resources

During synthesis, Vitis HLS performs the following basic tasks:

- Elaborates the C, C++ source code into an internal database containing the operators in the C code, such as additions, multiplications, array reads, and writes.
- Maps the operators onto implementations in the hardware.

Implementations are the specific hardware components used to create the design (such as adders, multipliers, pipelined multipliers, and block RAM).

Commands, pragmas and directives provide control over each of these steps, allowing you to control the hardware implementation at a fine level of granularity.

Limiting the Number of Operators

Explicitly limiting the number of operators to reduce area may be required in some cases: the default operation of Vitis HLS is to first maximize performance. Limiting the number of operators in a design is a useful technique to reduce the area of the design: it helps reduce area by forcing the sharing of operations. However, this might cause a decline in performance.

The ALLOCATION directive allows you to limit how many operators are used in a design. For example, if a design called `foo` has 317 multiplications but the FPGA only has 256 multiplier resources (DSP macrocells). The ALLOCATION pragma shown below directs Vitis HLS to create a design with a maximum of 256 multiplication (`mul`) operators:

```
dout_t array_arith (dio_t d[317]) {
    static int acc;
    int i;
#pragma HLS ALLOCATION instances=fmul limit=256 operation

    for (i=0;i<317;i++) {
#pragma HLS UNROLL
        acc += acc * d[i];
    }
    rerun acc;
}
```

Note: If you specify an ALLOCATION limit that is greater than needed, Vitis HLS attempts to use the number of resources specified by the limit, or the maximum necessary, which reduces the amount of sharing.

You can use the `type` option to specify if the ALLOCATION directives limits operations, implementations, or functions. The following table lists all the operations that can be controlled using the ALLOCATION directive.

Note: The operations listed below are supported by the ALLOCATION pragma or directive. The BIND_OP pragma or directive supports a subset of operators as described in the command syntax.

Table 18: Vitis HLS Operators

Operator	Description
add	Integer Addition
ashr	Arithmetic Shift-Right
dadd	Double-precision floating-point addition
dcmp	Double-precision floating-point comparison
ddiv	Double-precision floating-point division
dmul	Double-precision floating-point multiplication
drecip	Double-precision floating-point reciprocal
drem	Double-precision floating-point remainder
drsqrt	Double-precision floating-point reciprocal square root
dsub	Double-precision floating-point subtraction
dsqrt	Double-precision floating-point square root
fadd	Single-precision floating-point addition
fcmp	Single-precision floating-point comparison
fdiv	Single-precision floating-point division
fmul	Single-precision floating-point multiplication
frexp	Single-precision floating-point reciprocal
frem	Single-precision floating point remainder
frsqrt	Single-precision floating-point reciprocal square root
fsub	Single-precision floating-point subtraction
fsqrt	Single-precision floating-point square root
icmp	Integer Compare
lshr	Logical Shift-Right
mul	Multiplication
sdiv	Signed Divider
shl	Shift-Left
srem	Signed Remainder
sub	Subtraction
udiv	Unsigned Division
urem	Unsigned Remainder

Controlling Hardware Implementation

When synthesis is performed, Vitis HLS uses the timing constraints specified by the clock, the delays specified by the target device together with any directives specified by you, to determine which hardware implementations to use for various operators in the code. For example, to implement a multiplier operation, Vitis HLS could use the combinational multiplier or use a pipeline multiplier.

The implementations which are mapped to operators during synthesis can be limited by specifying the ALLOCATION pragma or directive, in the same manner as the operators. Instead of limiting the total number of multiplication operations, you can choose to limit the number of combinational multipliers, forcing any remaining multiplications to be performed using pipelined multipliers (or vice versa).

The BIND_OP or BIND_STORAGE pragmas or directives are used to explicitly specify which implementations to use for specific operations or storage types. The following command informs Vitis HLS to use a two-stage pipelined multiplier using fabric logic for variable `c`. It is left to Vitis HLS which implementation to use for variable `d`.

```
int foo (int a, int b) {
    int c, d;
    #pragma HLS BIND_OP variable=c op=mul impl=fabric latency=2
    c = a*b;
    d = a*c;

    return d;
}
```

In the following example, the BIND_OP pragma specifies that the add operation for variable `temp` is implemented using the `dsp` implementation. This ensures that the operation is implemented using a DSP module primitive in the final design. By default, add operations are implemented using LUTs.

```
void apint_arith(dinA_t  inA, dinB_t  inB,
                  dout1_t *out1
                 ) {

    dout2_t temp;
    #pragma HLS BIND_OP variable=temp op=add impl=dsp

    temp = inB + inA;
    *out1 = temp;

}
```

Refer to the BIND_OP or BIND_STORAGE pragmas or directives to obtain details on the implementations available for assignment to operations or storage types.

In the following example, the BIND_OP pragma specifies the multiplication for `out1` is implemented with a 3-stage pipelined multiplier.

```
void foo(...){
    #pragma HLS BIND_OP variable=out1 op=mul latency=3

    // Basic arithmetic operations
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;

}
```

If the assignment specifies multiple identical operators, the code must be modified to ensure there is a single variable for each operator to be controlled. For example, in the following code, if only the first multiplication (`inA * inB`) is to be implemented with a pipelined multiplier:

```
*out1 = inA * inB * inC;
```

The code should be changed to the following with the pragma specified on the `Result_tmp` variable:

```
#pragma HLS BIND_OP variable=Result_tmp op=mul latency=3
Result_tmp = inA * inB;
*out1 = Result_tmp * inC;
```

Controlling Operator Pipelining

Vitis HLS automatically determines the level of pipelining to use for internal operations. You can use the `BIND_OP` or `BIND_STORAGE` pragmas with the `-latency` option to explicitly specify the number of pipeline stages and override the number determined by Vitis HLS.

RTL synthesis might use the additional pipeline registers to help improve timing issues that might result after place and route. Registers added to the output of the operation typically help improve timing in the output datapath. Registers added to the input of the operation typically help improve timing in both the input datapath and the control logic from the FSM.

You can use the `config_op` command to pipeline all instances of a specific operation used in the design that have the same pipeline depth. Refer to [config_op](#) for more information.

Unrolling Loops in C++ Classes



IMPORTANT! When loops are used in C++ classes, care should be taken to ensure that the loop induction variable is not a data member of the class as this prevents the loop from being unrolled.

In this example, loop induction variable `k` is a member of class `loop_class`.

```
template <typename T0, typename T1, typename T2, typename T3, int N>
class loop_class {
private:
    pe_mac<T0, T1, T2> mac;
public:
    T0 areg;
    T0 breg;
    T2 mreg;
    T1 preg;
    T0 shift[N];
    int k; // Class Member
    T0 shift_output;
    void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
    {
        Function_label0::
```

```
#pragma HLS inline off
SRL:for (k = N-1; k >= 0; --k) {
#pragma HLS unroll // Loop will fail UNROLL
if (k > 0)
shift[k] = shift[k-1];
else
shift[k] = data;
}

*dataOut = shift_output;
shift_output = shift[N-1];
}

*pcout = mac.exec1(shift[4*col], coeff, pcin);
};
```

For Vitis HLS to be able to unroll the loop as specified by the UNROLL pragma directive, the code should be rewritten to remove `k` as a class member and make it local to the `exec` function.

Limitations of Control-Driven Task-Level Parallelism



TIP: Control-driven TLP requires the `DATAFLOW` pragma or directive to be specified in the appropriate location of the code.

The control-driven TLP model optimizes the flow of data between tasks (functions and loops), and ideally pipelined functions and loops for maximum performance. It does not require these tasks to be chained, one after the other, however there are some limitations in how the data is transferred. The following behaviors can prevent or limit the overlapping that Vitis HLS can perform in the dataflow model:

- Reading from function inputs or writing to function outputs in the middle of the dataflow region
- Single-producer-consumer violations
- Conditional execution of tasks
- Loops with multiple exit conditions



IMPORTANT! If any of these coding styles are present, Vitis HLS issues a message describing the situation.

Reading from Inputs/Writing to Outputs

Reading of inputs of the function should be done at the start of the dataflow region, and writing to outputs should be done at the end of the dataflow region. Reading/writing to the ports of the function can cause the processes to be executed in sequence rather than in an overlapped fashion, adversely impacting performance.

Single-producer-consumer Violations

For Vitis HLS to use the dataflow model, all elements passed between tasks must follow a single-producer-consumer model. Each variable must be driven from a single task and only be consumed by a single task. In the following code example, `temp1` fans out and is consumed by both `Loop2` and `Loop3`. This violates the single-producer-consumer model.

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {  
    int temp1[N];  
  
    Loop1: for(int i = 0; i < N; i++) {  
        temp1[i] = data_in[i] * scale;  
    }  
    Loop2: for(int j = 0; j < N; j++) {  
        data_out1[j] = temp1[j] * 123;  
    }  
    Loop3: for(int k = 0; k < N; k++) {  
        data_out2[k] = temp1[k] * 456;  
    }  
}
```

A modified version of this code uses function `Split` to create a single-producer-consumer design. The following code block example shows how the data flows with the function `Split`. The data now flows between all four tasks, and Vitis HLS can use the dataflow model.

```
void Split (in[N], out1[N], out2[N]) {  
// Duplicated data  
L1:for(int i=1;i<N;i++) {  
out1[i] = in[i];  
out2[i] = in[i];  
}  
}  
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {  
  
int temp1[N], temp2[N]. temp3[N];  
Loop1: for(int i = 0; i < N; i++) {  
temp1[i] = data_in[i] * scale;  
}  
Split(temp1, temp2, temp3);  
Loop2: for(int j = 0; j < N; j++) {  
data_out1[j] = temp2[j] * 123;  
}  
Loop3: for(int k = 0; k < N; k++) {  
data_out2[k] = temp3[k] * 456;  
}  
}
```

Bypassing Tasks and Channel Sizing

In addition, data should generally flow from one task to another. If you bypass tasks, this can reduce the performance of the dataflow model. In the following example, `Loop1` generates the values for `temp1` and `temp2`. However, the next task, `Loop2`, only uses the value of `temp1`. The value of `temp2` is not consumed until after `Loop2`. Therefore, `temp2` bypasses the next task in the sequence, which can limit the performance of the dataflow model.

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {  
    int temp1[N], temp2[N], temp3[N];  
    Loop1: for(int i = 0; i < N; i++) {  
        temp1[i] = data_in[i] * scale;  
        temp2[i] = data_in[i] >> scale;  
    }  
    Loop2: for(int j = 0; j < N; j++) {  
        temp3[j] = temp1[j] + 123;  
    }  
    Loop3: for(int k = 0; k < N; k++) {  
        data_out[k] = temp2[k] + temp3[k];  
    }  
}
```

In this case, you should increase the depth of the PIPO buffer used to store `temp2` to be 3, instead of the default depth of 2. This lets the buffer store the value intended for `Loop3`, while `Loop2` is being executed. Similarly, a PIPO that bypasses two processes should have a depth of 4. Set the depth of the buffer with the `STREAM` pragma or directive:

```
#pragma HLS STREAM type=piwo variable=temp2 depth=3
```



IMPORTANT! Channel sizing can also similarly affect performance. Having mismatched FIFO/PIPO depths can inadvertently cause synchronization points inside the dataflow region because of back pressure from the FIFO/PIPO.

Feedback between Tasks

Feedback occurs when the output from a task is consumed by a previous task in the dataflow region. Feedback between tasks is not recommended in a dataflow region. When Vitis HLS detects feedback, it issues a warning, depending on the situation, and might not use the dataflow model.

However, dataflow can support feedback when used with `hls::streams`. The following example demonstrates this exception.

```
#include "ap_axi_sdata.h"  
#include "hls_stream.h"  
  
void firstProc(hls::stream<int> &forwardOUT, hls::stream<int> &backwardIN) {  
    static bool first = true;  
    int fromSecond;  
  
    //Initialize stream  
    if (first)  
        fromSecond = 10; // Initial stream value
```

```
else
    //Read from stream
    fromSecond = backwardIN.read(); //Feedback value
first = false;

//Write to stream
forwardOUT.write(fromSecond*2);
}

void secondProc(hls::stream<int> &forwardIN, hls::stream<int> &backwardOUT)
{
    backwardOUT.write(forwardIN.read() + 1);
}

void top(...){
#pragma HLS dataflow
    hls::stream<int> forward, backward;
    firstProc(forward, backward);
    secondProc(forward, backward);
}
```

In this simple design, when `firstProc` is executed, it uses 10 as an initial value for input. Because `hls::streams` do not support an initial value, this technique can be used to provide one without violating the single-producer-consumer rule. In subsequent iterations `firstProc` reads from the `hls::stream` through the `backwardIN` interface.

`firstProc` processes the value and sends it to `secondProc`, via a stream that goes *forward* in terms of the original C++ function execution order. `secondProc` reads the value on `forwardIN`, adds 1 to it, and sends it back to `firstProc` via the feedback stream that goes *backwards* in the execution order.

From the second execution, `firstProc` uses the value read from the stream to do its computation, and the two processes can keep going forever, with both forward and feedback communication, using an initial value for the first execution.

Conditional Execution of Tasks

The dataflow model does not optimize tasks that are conditionally executed. The following example highlights this limitation. In this example, the conditional execution of `Loop1` and `Loop2` prevents Vitis HLS from optimizing the data flow between these loops, because the data does not flow from one loop into the next.

```
void foo(int data_in1[N], int data_out[N], int sel) {
    int temp1[N], temp2[N];

    if (sel) {
        Loop1: for(int i = 0; i < N; i++) {
            temp1[i] = data_in[i] * 123;
            temp2[i] = data_in[i];
        }
    } else {
        Loop2: for(int j = 0; j < N; j++) {
            temp1[j] = data_in[j] * 321;
            temp2[j] = data_in[j];
        }
    }
}
```

```
    }
}
Loop3: for(int k = 0; k < N; k++) {
    data_out[k] = temp1[k] * temp2[k];
}
}
```

To ensure each loop is executed in all cases, you must transform the code as shown in the following example. In this example, the conditional statement is moved into the first loop. Both loops are always executed, and data always flows from one loop to the next.

```
void foo(int data_in[N], int data_out[N], int sel) {

    int temp1[N], temp2[N];

    Loop1: for(int i = 0; i < N; i++) {
        if (sel) {
            temp1[i] = data_in[i] * 123;
        } else {
            temp1[i] = data_in[i] * 321;
        }
    }
    Loop2: for(int j = 0; j < N; j++) {
        temp2[j] = data_in[j];
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out[k] = temp1[k] * temp2[k];
    }
}
```

Loops with Multiple Exit Conditions

Loops with multiple exit points cannot be used in a dataflow region. In the following example, Loop2 has three exit conditions:

- An exit defined by the value of `N`; the loop will exit when `k>=N`.
- An exit defined by the `break` statement.
- An exit defined by the `continue` statement.

```
#include "ap_int.h"
#define N 16

typedef ap_int<8> din_t;
typedef ap_int<15> dout_t;
typedef ap_uint<8> dsc_t;
typedef ap_uint<1> dsel_t;

void multi_exit(din_t data_in[N], dsc_t scale, dsel_t select, dout_t
data_out[N]) {
    dout_t temp1[N], temp2[N];
    int i,k;

    Loop1: for(i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
        temp2[i] = data_in[i] >> scale;
    }
}
```

```
Loop2: for(k = 0; k < N; k++) {  
    switch(select) {  
        case 0: data_out[k] = temp1[k] + temp2[k];  
        case 1: continue;  
        default: break;  
    }  
}
```

Because a loop's exit condition is always defined by the loop bounds, the use of `break` or `continue` statements will prohibit the loop being used in a DATAFLOW region.

Finally, the dataflow model has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from dataflow, you must apply the dataflow model to the loop, the sub-function, or inline the sub-function.

You can also use `std::complex` inside the dataflow region. However, they should be used with an `__attribute__((no_ctor))` as shown in the following example:

```
void proc_1(std::complex<float> (&buffer)[50], const std::complex<float> *in);  
void proc_2(hls::Stream<std::complex<float>> &fifo, const std::complex<float> (&buffer)[50], std::complex<float> &acc);  
void proc_3(std::complex<float> *out, hls::Stream<std::complex<float>> &fifo, const std::complex<float> acc);  
  
void top(std::complex<float> *out, const std::complex<float> *in) {  
#pragma HLS DATAFLOW  
  
    std::complex<float> acc __attribute__((no_ctor)); // Here  
    std::complex<float> buffer[50] __attribute__((no_ctor)); // Here  
    hls::Stream<std::complex<float>, 5> fifo; // Not here  
  
    proc_1(buffer, in);  
    proc_2(fifo, buffer, acc);  
    proc_3(out, fifo, acc);  
}
```

Limitations of Pipelining with Static Variables

Static variables are used to keep data between loop iterations, often resulting in registers in the final implementation. If this is encountered in pipelined functions, Vitis HLS might not be able to optimize the design sufficiently, which would result in initiation intervals longer than required.

The following is a typical example of this situation:

```
function_foo( )
{
    static bool change = 0
    if (condition_xyz){
        change = x; // store
    }
    y = change; // load
}
```

If Vitis HLS cannot optimize this code, the stored operation requires a cycle and the load operation requires an additional cycle. If this function is part of a pipeline, the pipeline has to be implemented with a minimum initiation interval of 2 as the static change variable creates a loop-carried dependency.

One way the user can avoid this is to rewrite the code, as shown in the following example. It ensures that only a read or a write operation is present in each iteration of the loop, which enables the design to be scheduled with $II=1$.

```
function_readstream( )
{
    static bool change = 0
    bool change_temp = 0;
    if (condition_xyz)
    {
        change = x; // store
        change_temp = x;
    }
    else
    {
        change_temp = change; // load
    }
    y = change_temp;
}
```

Using Vitis HLS

This section contains the following chapters:

- [Navigating Content by Design Process](#)
- [Design Principles](#)
- [Vitis HLS Flow Overview](#)
- [Launching Vitis HLS](#)
- [Creating a New Vitis HLS Project](#)
- [Verifying Code with C Simulation](#)
- [Synthesizing the Code](#)
- [Analyzing the Results of Synthesis](#)
- [Optimizing the HLS Project](#)
- [C/RTL Co-Simulation in Vitis HLS](#)
- [Exporting the RTL Design](#)
- [Running Vitis HLS from the Command Line](#)

Launching Vitis HLS

To launch Vitis™ HLS, you must first configure the environment to run the tool as described in [Setting Up the Environment](#). This requires setting the environment variables and paths needed for the tool.

To launch Vitis HLS on a Linux platform, or from the command prompt on Windows, execute the following:

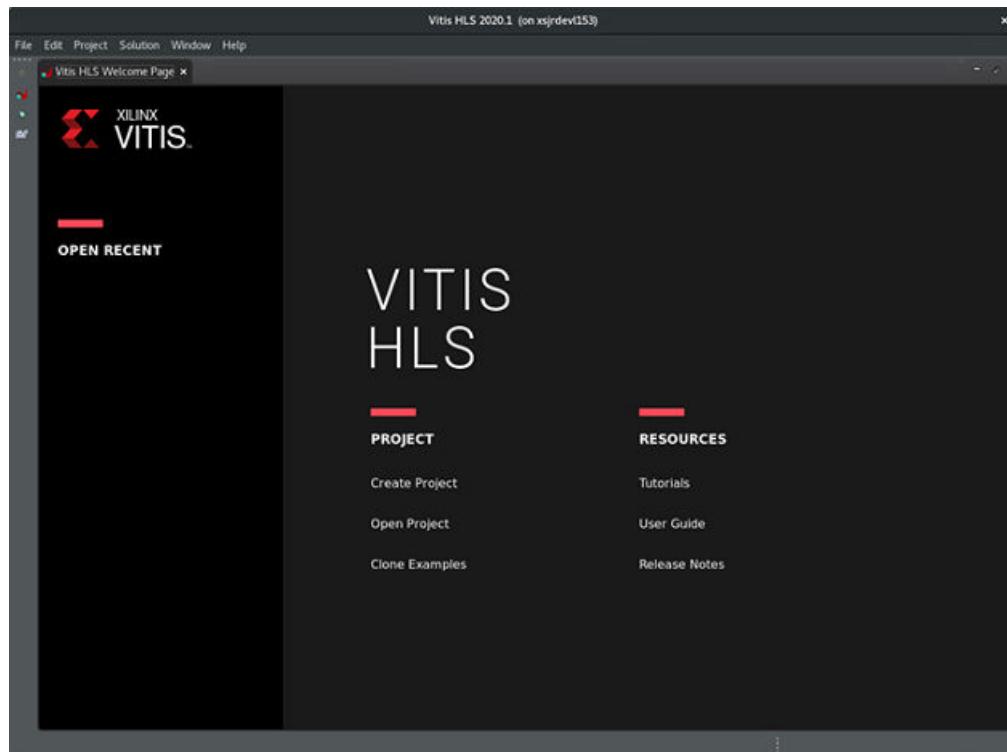
```
$ vitis_hls
```



TIP: You can also launch Vitis HLS by double-clicking the application from the Windows desktop.

The Vitis HLS GUI opens as shown in the following figure.

Figure 81: Vitis HLS GUI Welcome Page



Under Project, you have the following options.

- **Create Project:** Launch the project setup wizard to create a new project. Refer to [Creating a New Vitis HLS Project](#) for more information.
- **Open Project:** Navigate to an existing project.
- **Clone Examples:** Clone Example projects from GitHub repository to create a local copy for your use. See [Tutorials and Examples](#).

Under Resources, you will find documentation and tutorials to help you work with the tool.

If you have previously launched Vitis HLS to create a project, you can also select from a list of recent projects under Open Recent.

Setting Up the Environment

Vitis HLS is delivered as part of the Vitis unified software platform. For instructions on installing the tool, refer to [Installation](#) in *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)*.



TIP: For information on the Vitis HLS release, and known limitations of the release refer to [AR# 75342](#).

After you have installed the elements of the Vitis software platform, you need to setup the operating environment to run Vitis HLS in a specific command shell by running the `settings64.sh` bash script, or `settings64.csh` script:

```
#setup XILINX_VITIS and XILINX_VIVADO variables
source <Vitis_install_path>/settings64.sh
```



TIP: While the Vitis unified software platform also requires the installation and setup of the Xilinx runtime (XRT) and hardware platforms, these elements are not required for the use of Vitis HLS.

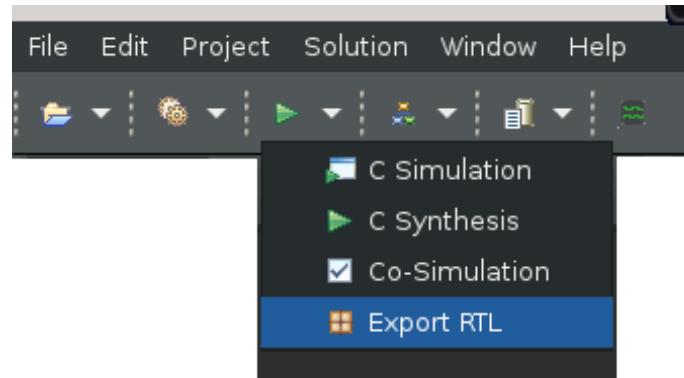
Overview of the Vitis HLS IDE

The toolbar menu shown below provides access to the primary commands for using Vitis HLS. The main menu provides access to all available commands for creating and managing designs. Each of the buttons on the toolbar menu has an equivalent command in the main menu.



TIP: Project control ensures that only commands that can be currently executed are highlighted. For example, synthesis must be performed before C/RTL co-simulation can be executed.

Figure 82: Vitis HLS Toolbar and Main Menus



In the toolbar menu, the buttons are (from left to right):

- **Open Project:** Opens a file browser to let you locate and open an HLS project. The drop-down menu also provides access to the New File command, which lets you create a new file to open in the text editor.
- **Solution Settings:** Opens the Solution Settings dialog box to modify the settings of the active solution. The drop-down menu also provides access to:
 - Project Settings to let you configure the settings of the open project.
 - New Solution to let you define a new solution for the open project.
- **C Synthesis:** Starts C source code to RTL synthesis in Vitis HLS as described in [Synthesizing the Code](#). The drop-down menu provides a process overview of Vitis HLS, including:
 - C Simulation to let you launch C simulation of the open project as described in [Verifying Code with C Simulation](#).
 - Co-Simulation to let you launch [C/RTL Co-Simulation in Vitis HLS](#).
 - Export RTL to let you export the open project as explained in [Exporting the RTL Design](#).
- **Open Analysis Viewer:** Displays various analysis reports when they have been generated during simulation, synthesis, or C/RTL co-simulation. The drop-down menu also provides access to:
 - Open Pre-Synthesis Control Flow to display the [Pre-Synthesis Control Flow](#) report when it has been generated during simulation.
 - Open Dataflow View to display the [Dataflow Viewer](#) report when it has been generated during C/RTL co-simulation.
 - Open Schedule Viewer when the [Schedule Viewer](#) has been generated during C synthesis.
- **Open Report:** Displays the report generated during C synthesis. The drop-down menu also provides access to:
 - Synthesis to display the report generated during C synthesis.

- Co-Simulation to display the report generated during C/RTL co-simulation.
- Export RTL to display the report generated while exporting the RTL.
- **Open Wave Viewer:** Displays the Waveform Viewer when the C/RTL Co-simulation includes the waveform from the Vivado simulator.

In addition, Vitis HLS IDE provides three perspectives. When you select a perspective, the windows automatically adjust to a more suitable layout for the selected task.

- The Debug perspective opens the C debugger.
- The Synthesis perspective is the default perspective and arranges the windows for performing synthesis.
- The Analysis perspective is used after synthesis completes to analyze the design in detail.

Customizing the Vitis HLS IDE Behavior

The behavior of the Vitis HLS IDE can be customized using settings available from the **Windows → Preferences** menu, and user-defined preferences saved.

Reviewing the different sub-menus in the **Preferences** dialog box allows most elements of the Vitis HLS environment to be customized.

Customizing the Console View

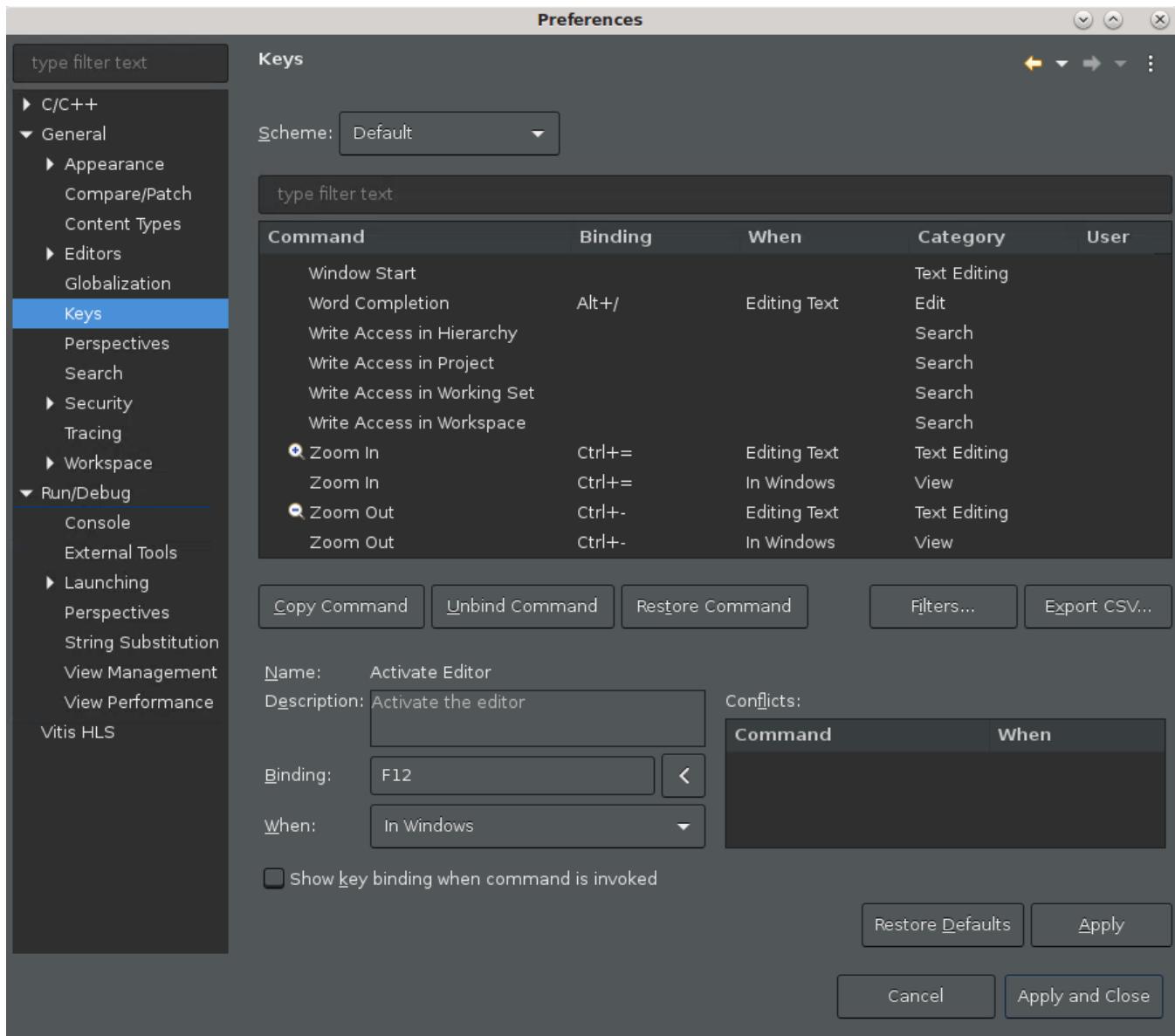
The **Console** view displays the messages issued during tool operations such as synthesize and verification. The default buffer size for this windows is 80,000 characters and can be changed, or the limit can be removed, to ensure all messages can be reviewed.

Change the Console settings using **Window → Preferences → Run/Debug → Console**. You can change the **Console buffer size** in characters, or disable the **Limit console output** checkbox to remove the limit. There are additional settings that can be modified as well.

Customizing Keyboard Shortcuts

The Vitis HLS tool comes with default keyboard shortcuts for the various editors and windows. These can be viewed and modified from the **Window → Preferences → General → Keys** menu.

Figure 83: Keyboard Shortcuts



For instance, as shown in the figure above, to change the size of the font in the text editor window you can use the keyboard shortcut **Ctrl + =** to zoom in and make the text larger, or use **Ctrl + -** to zoom out, and make the text smaller.

You can use the **Binding** field as shown above to change the keyboard shortcut for specific commands or activities. If you define a keyboard shortcut that conflicts with another command it will be reported in the **Conflicts** window. You can save any custom keyboard shortcuts by using the **Apply** button. You can restore the tool defaults by using the **Restore Defaults** button.

The window has a search bar that displays the phrase **type filter text** when not in use, as shown above. You can type a phrase or keyword to locate a specific keyboard shortcut.

Unbind Command will remove the keyboard shortcut for a specific command. **Restore Command** will restore the original binding.

For example, the key combination **Ctrl + Tab** toggles between the source code and the header file in the text editor. You can change this keyboard shortcut to make each tab active using the following steps:

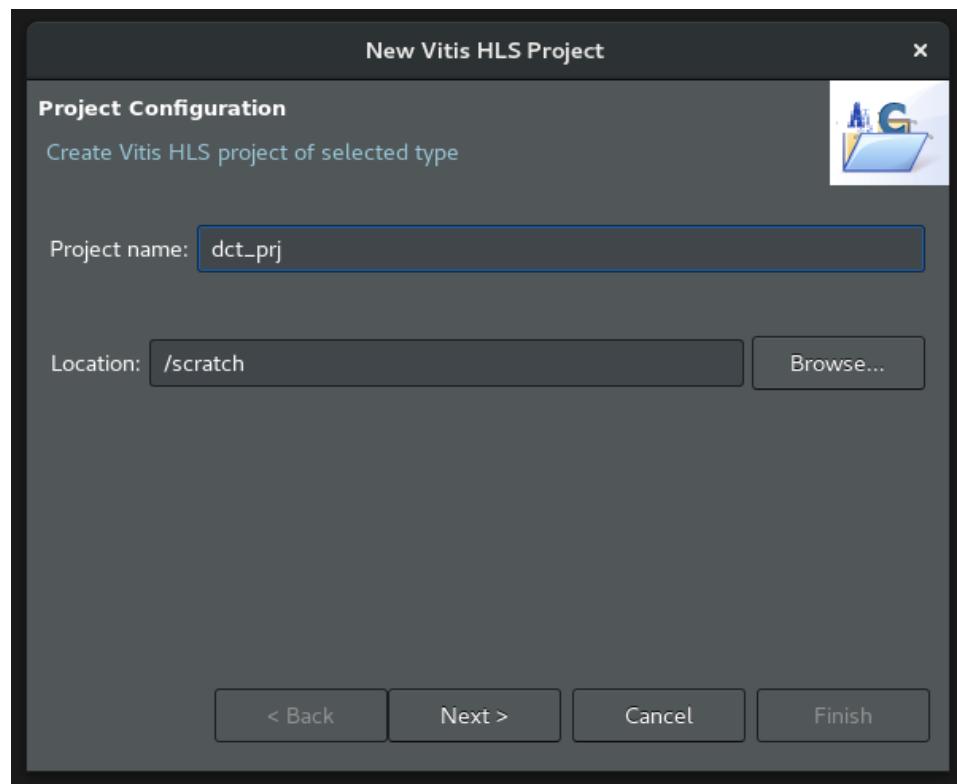
1. In **Window → Preferences → General → Keys** search for and select **Toggle Source/Header** and remove the binding by using the **Unbind Command** button.
2. Search for and select **Next Tab**, place the cursor in the **Binding** field and press backspace to clear the current binding, and then press the **Ctrl** and **Tab** keys together to define the new keyboard binding for the command.
3. Click **Apply**, or **Apply and Close**.

You can change the key-binding scheme from the tool default to make it more like a familiar tool. The two supported schemes are **Microsoft Visual Studio** and **Emacs**. Changing the scheme will change the keyboard shortcuts accordingly.

Creating a New Vitis HLS Project

To create a new project, click the **Create New Project** link on the Welcome page, or select the **File→New Project** menu command. This opens the New Vitis HLS Project wizard, as shown in the following figure.

Figure 84: New Vitis HLS Project Wizard



Create a new Vitis™ HLS project using the following steps:

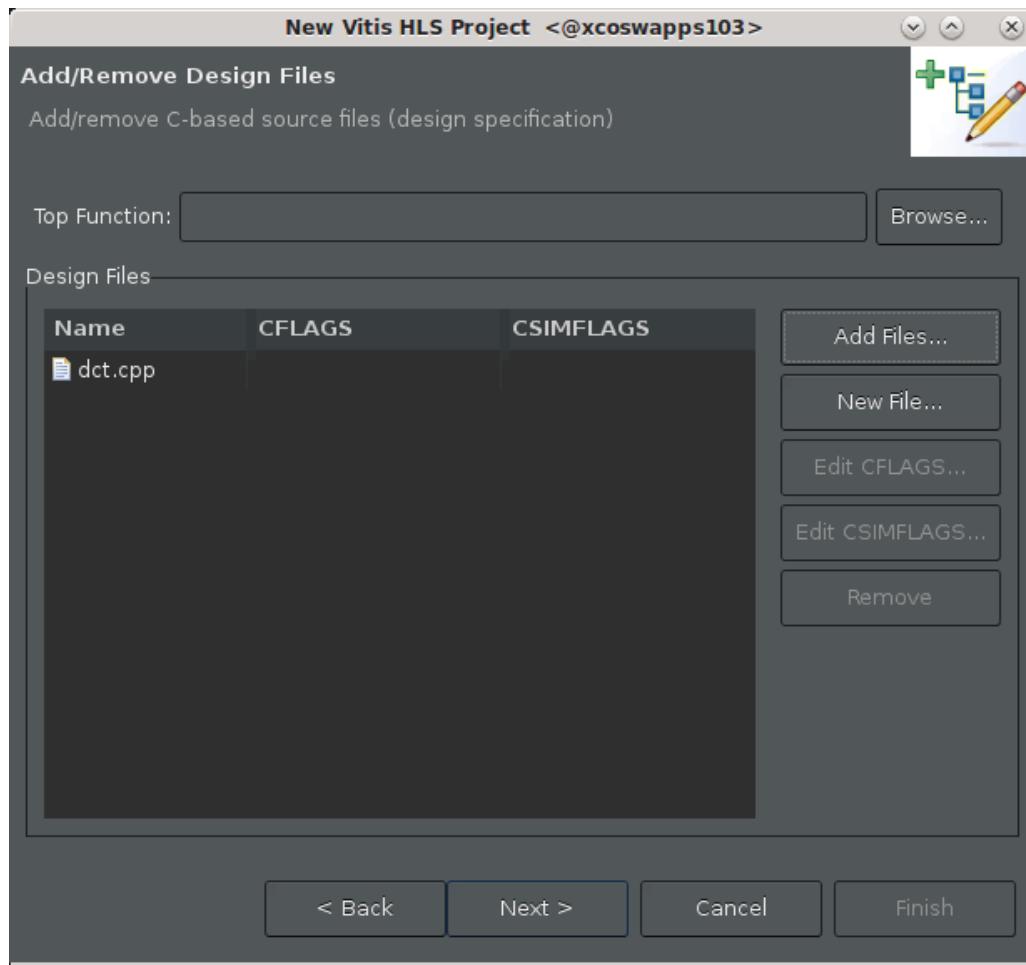
1. Specify the project name, which is also the name of the directory in which the project files and folders are written.
2. Specify the location where the project is written.



IMPORTANT! The Windows operating system has a 255-character limit for path lengths, which can affect the Vitis tools. To avoid this issue, use the shortest possible names and directory locations when creating projects, or adding new files.

3. Click **Next** to proceed to the Add/Remove Design Files page.

The Add/Remove Design Files page lets you add C/C++ source files to your project, as shown in the following figure:



4. Click **Add Files**, and navigate to the location of the source code files to add to your project.

Do not add header files (with the `.h` suffix) to the project using the Add Files button, or the `add_files` Tcl command. Vitis HLS automatically adds the following directories to the compilation search path:

- Working directory, which contains the Vitis HLS project directory.
- Any directory that contains C/C++ files that have been added to the project.

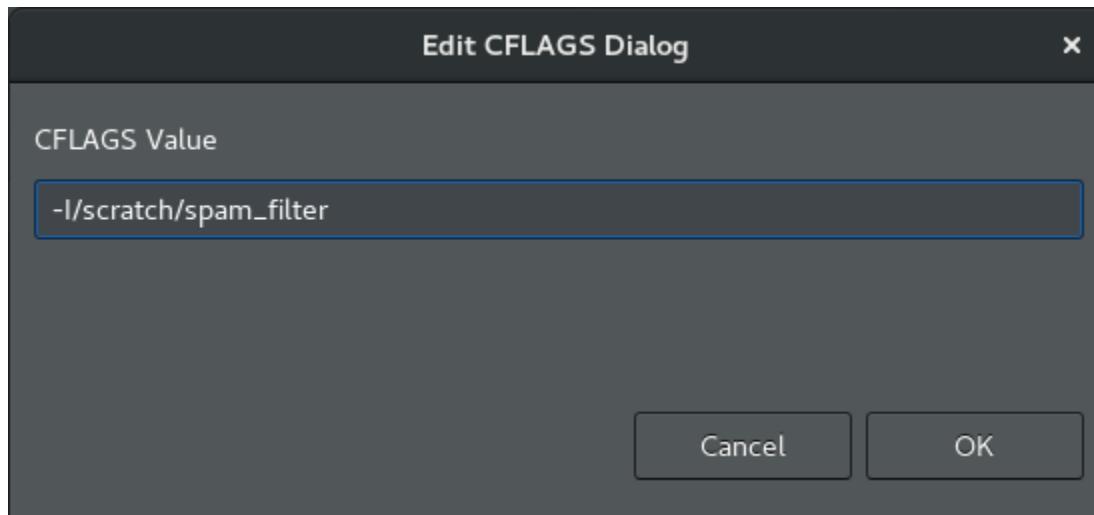
Header files that reside in these directories are automatically included in the project during compilation. However, you can specify other include paths using the Edit CFLAGS function.

5. Optionally, click **New File** to create a new source file to add to your project. The File Browser dialog box opens to let you specify the file name and location to store the new file.

 **TIP:** If you want to write the new file to the directory that will be created for your new project, you must wait to create the new file until after the project has been created.

6. You can select a file, and click **Edit CFLAGS** or **Edit CSIMFLAGS** to open a dialog box letting you add one or more compiler or simulation flags for the selected file.

The following figure shows example CFLAGS:



Compiler flags are standard compiler options for `gcc` or `g++`. For a complete list of options, refer to <http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html> on the GNU Compiler Collection (GCC) website. The following are some example CFLAGS:

- **-I/source/header_files:** Provides the search path to associated header files. You can specify absolute or relative paths to files.



IMPORTANT! You must specify relative paths in relation to the working directory, not the project directory.

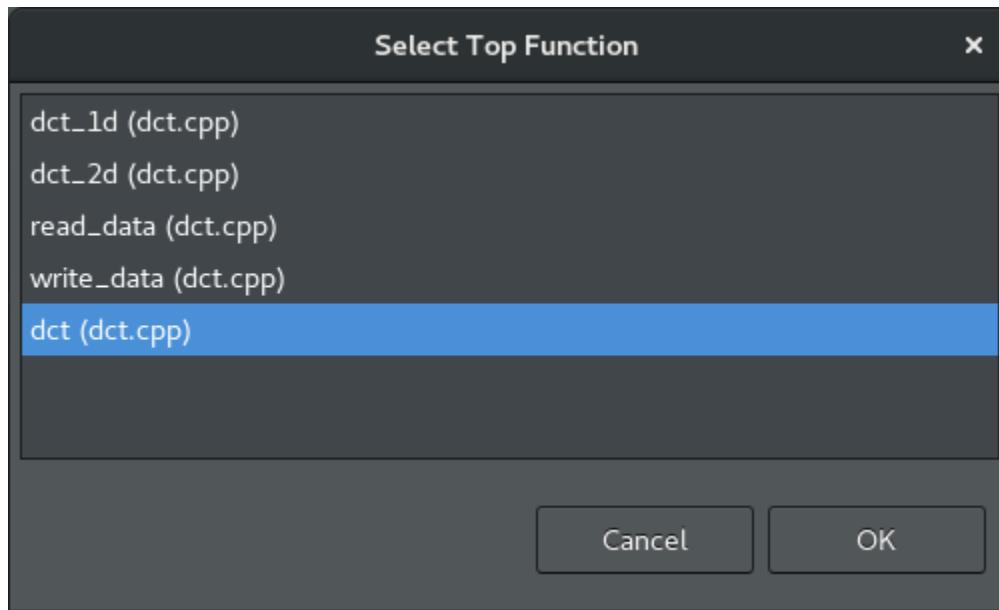
- **-DMACRO_1:** Defines macro MACRO_1 during compilation.
- **-fnested-functions:** Defines directives required for any design that contains nested functions.



TIP: You can use `$::env(MY_ENV_VAR)` to specify environment variables in CFLAGS. For example, to include the directory `$MY_ENV_VAR/include` for compilation, you can specify the CFLAG as `-I$::env(MY_ENV_VAR)/include`.

7. Click **Remove** to delete any files from your project that are not needed or were added by mistake.
8. Next to the Top Function field, click **Browse** to list the functions and sub-functions found in the added files.

The Select Top Function dialog box is opened as shown below. This dialog box lists the functions found in the added files, and lets you specify which of these is the top function for the purposes of HLS.



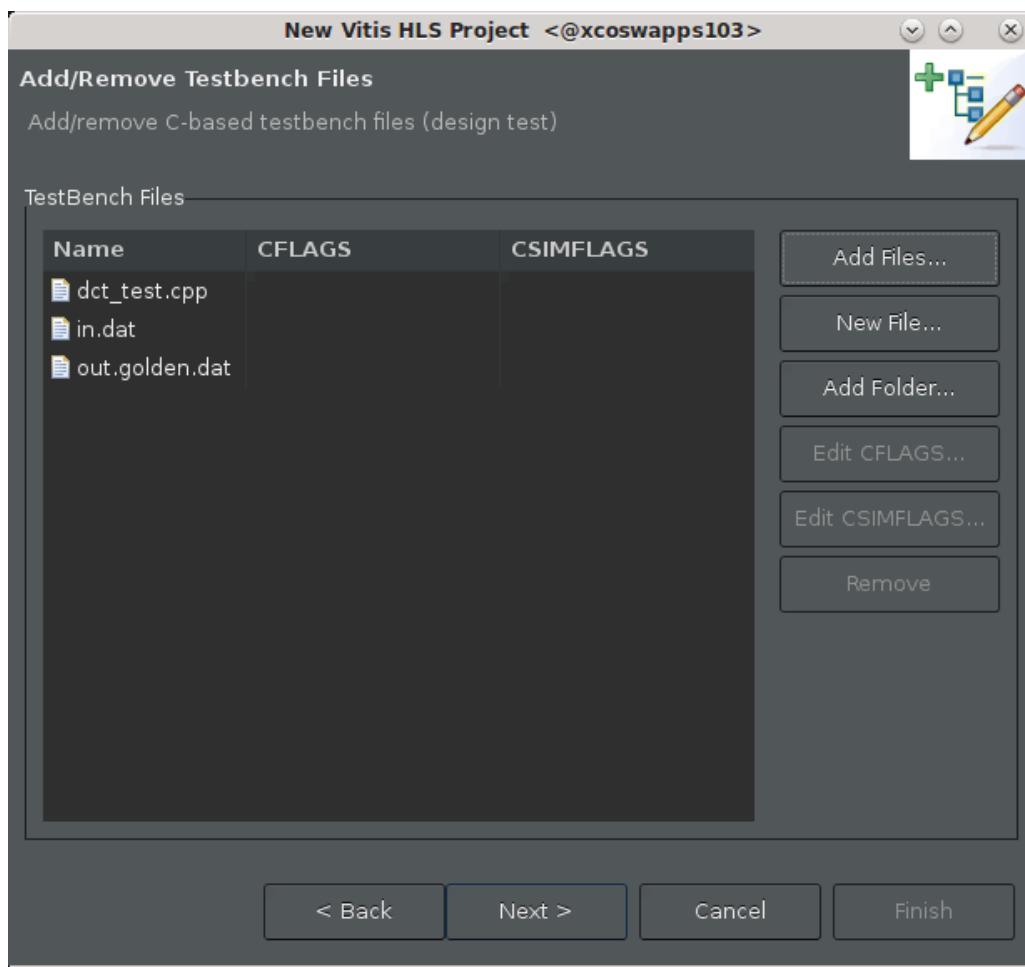
TIP: You can simply type the name of top-level function in the available field. However, after source files have been added to the project, the tool lists the available functions for you to choose from.

9. In the Add/Remove Design Files page, with files added and the top function specified, click **Next** to proceed.

In the Add/Remove Testbench Files dialog box, you can add test bench files and other required files to your project, as shown in the following figure.



TIP: There is no requirement to add a test bench to the project. You can simply click **Next** to skip this step if you prefer.

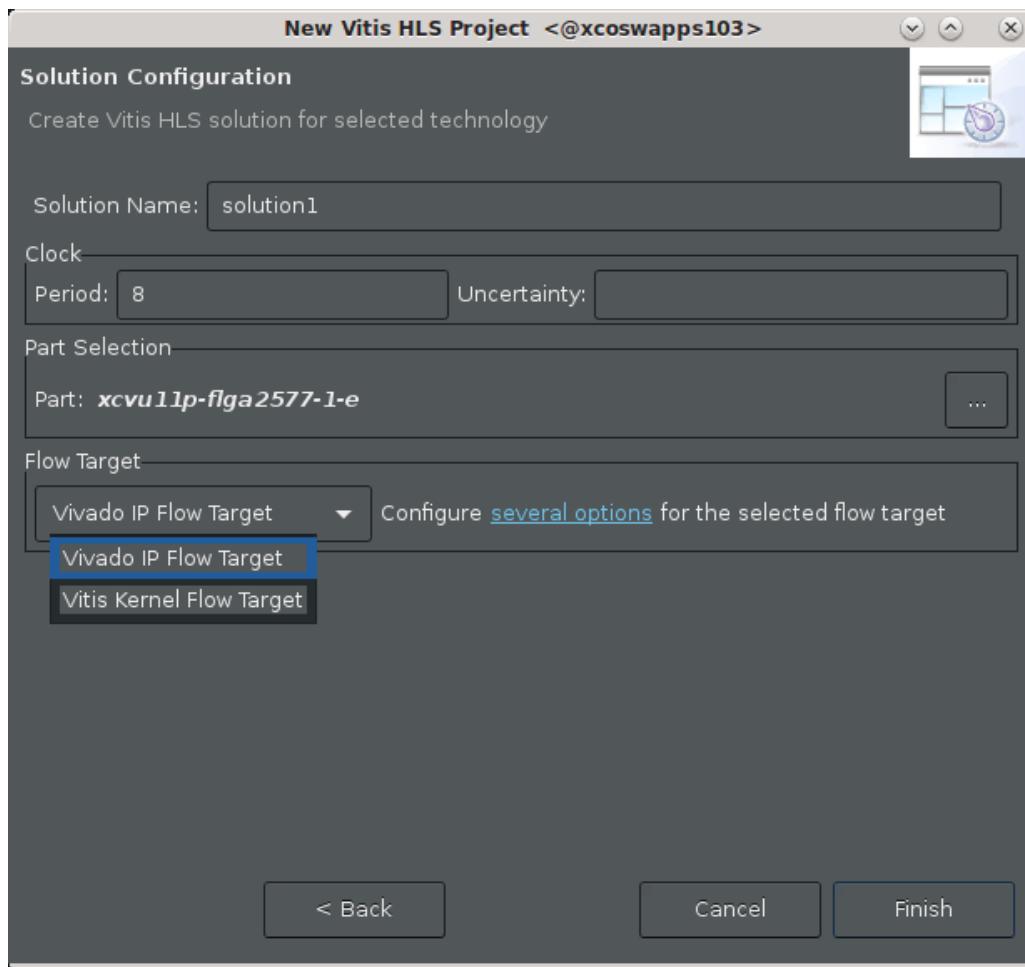


10. As with the C source files, click **Add Files** to add the test bench. Click **Edit CFLAGS** or **Edit CSIMFLAGS** to include any compiler options.
11. In addition to the C source files, all files read by the test bench must be added to the project. In the example shown in the figure above, the test bench opens file `in.dat` to supply input stimuli to the design, and reads `out.golden.dat` to read the expected results. Because the test bench accesses these files, both files must be included in the project.

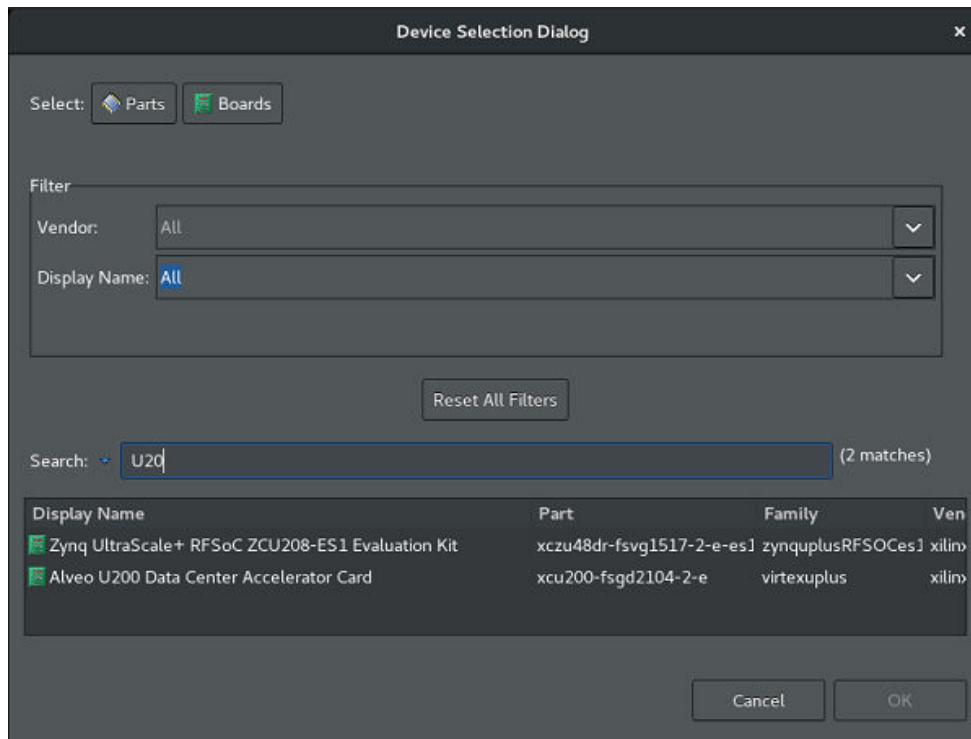


TIP: If the test bench files exist in a directory, you can add the entire directory to the project, rather than the individual files, by clicking **Add Folder**.

12. Click **Next** to proceed and the Solution Configuration dialog box is displayed, letting you configure the initial solution for your project.

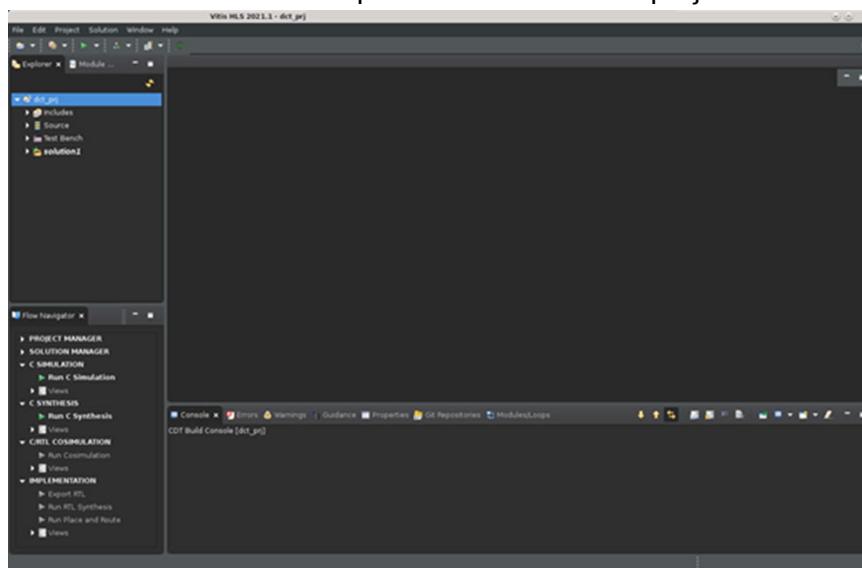


13. Specify a **Solution Name** to collect the directives, the results, and the reports for a specific configuration of the project. Multiple solutions let you create different project configurations to quickly find the best solution.
14. Under Clock, specify the **Period** in units of ns, or as a frequency value specified with the MHz suffix (for example, 150 MHz). Refer to [Specifying the Clock Frequency](#) for more information.
15. Specify the **Uncertainty** used for synthesis as the clock period minus the clock uncertainty. Vitis HLS uses internal models to estimate the delay of the operations for each device. The clock uncertainty value provides a controllable margin to account for any increases in net delays due to RTL logic synthesis, place, and route. Specify as a value in nanoseconds (ns), or as a percentage of the clock period. The default clock uncertainty is 27% of the clock period.
16. Complete Part Selection for your project by clicking the browse button (...) to display the Device Selection Dialog box, as shown below.



The Device Selection Dialog box lets you select the device for your project as a part, or as a board, such as an Alveo™ Data Center accelerator card. You can click the **Search** filter to reduce the number of devices in the device list.

17. Vitis HLS supports two primary output flows: the Vivado IP flow, and the Vitis Kernel flow. The **Flow Target** drop-down menu lets you enable one of these flows as described in [Vitis HLS Flow Overview](#).
18. Click **Finish** to create and open the new Vitis HLS project as shown in the following figure.



By default the Vitis HLS IDE initially displays four panes:

- In the upper left-hand side, the Explorer view lets you navigate through the project hierarchy. A similar hierarchy exists in the project directory on the disk.
- In the center, the Information area displays report summaries and open files. Files can be opened by double-clicking them in the Explorer view.
- At the bottom, the Console view displays the output when Vitis HLS is running synthesis or simulation.
- In the lower left-hand side, the Flow Navigator view which provides access to commands and processes as described in [Using the Flow Navigator](#) to take your source code through simulation, synthesis, and exported output.
- Though not displayed by default, when source code is opened in the Information area the Outline and Directive views are displayed on the right-side, and show information related to the hierarchy of the code.

In addition to the views displayed by default, there are additional views that are opened by launching specific processes such as C/RTL co-simulation, or opening source files or reports. Additional views can be opened using the **Window→Show View** command from the main menu.

Vitis HLS Flow Overview

Vitis HLS is project based and can contain multiple variations of a project called "solutions" to drive synthesis and simulation. Each solution can target either the Vivado IP flow, or the Vitis Kernel flow.

The Vivado IP flow produces an RTL IP files for use in the Vivado Design Suite, for inclusion in the IP catalog, and for use in block designs of the IP integrator tool. The IP can be used for hardware design in the IP integrator feature of Vivado, or for RTL design.

The Vitis Kernel flow is a structured hardware development environment that lets you quickly expand custom hardware platforms using PL kernels developed in Vitis HLS. Vitis kernels can be used in application acceleration for Data Center applications, or in embedded system design for heterogeneous compute systems.

Vitis HLS implements the solution based on the target flow, default tool configuration, design constraints, and any optimization pragmas or directives you specify. You can use optimization directives to modify and control the implementation of the internal logic and I/O ports, overriding the default behaviors of the tool.

Enabling the Vivado IP Flow

When you select the **Vivado IP Flow Target** on the Solution Settings dialog box you are configuring Vitis HLS to generate RTL IP files for use in the Vivado Design Suite.



TIP: The flow target can also be enabled using the `open_solution -flow_target vivado` Tcl command.

The exported Vivado IP can be included in the IP catalog, and used in block designs of the IP integrator tool or in RTL-based design. HLS synthesis transforms your C or C++ code into register transfer level (RTL) code that you can synthesize and implement into the programmable logic region of a Xilinx device. The Vivado IP flow lets you develop and export IP as part of a larger hardware design, and provides hardware drivers to let you perform traditional embedded software design as described in *Vitis Unified Software Platform Documentation: Embedded Software Development* ([UG1400](#)). The Vivado IP flow provides greater flexibility in your design choices, however it leaves the integration and management of the IP to you as well.

The Vivado IP flow can support a wide variety of interface specifications and data transfer protocols, but has default interfaces assigned to function arguments as described in [Interfaces for Vivado IP Flow](#). You can also override the default settings by manually assigning the interface specification for your function argument, using the `INTERFACE` pragma or `set_directive_interface` command, to meet the needs of your Vivado design.

Enabling the Vitis Kernel Flow

When you select the **Vitis Kernel Flow Target** on the Solution Settings dialog box, as discussed in [Creating a New Vitis HLS Project](#), you are configuring Vitis HLS to generate the compiled kernel object (.xo) for the Vitis application acceleration flow, or heterogeneous compute flow.



TIP: The flow target can also be enabled using the `open_solution -flow_target vitis` Tcl command.

The Vitis Kernel flow is more restrictive than the Vivado IP flow, and the kernels produced by the HLS tool must meet the specific requirements of the platforms and Xilinx runtime (XRT), as described in [Kernel Properties](#) in the *Vitis Unified Software Platform Documentation*.

When specifying `open_solution -flow_target vitis`, or enabling the **Vitis Kernel Flow** in the IDE, Vitis HLS implements interface ports using the AXI standard as described in [Interfaces for Vitis Kernel Flow](#). If there are no existing `INTERFACE` pragmas or directives in the code, then the following interface protocols will be applied by default.

- AXI4-Lite interfaces (`s_axilite`) are assigned to scalar arguments, control signals for arrays, and the return value of the software function.
- AXI4 Master interfaces (`m_axi`) are assigned to pointer and array arguments of the C/C++ function.
- Vitis HLS automatically tries to infer BURST transactions whenever possible to aggregate memory accesses to maximize the throughput bandwidth and/or minimize the latency.
- Defining a software function argument using an `hls::stream` data type implies an AXI4-Stream (`axis`) port.

Default Settings of Vivado/Vitis Flows

The `open_solution` target will configure the compiler for either the Vivado IP flow or the Vitis Kernel flow. This will change the default behavior of the tool according to the flow specified. The following table shows the default settings of both flows so that you can quickly determine the differences in the default configuration.



TIP: Beyond the default configuration, there are additional features of the Vitis HLS tool that support one flow, but not the other, or are configured differently between the two flows. Those differences are highlighted throughout this document.

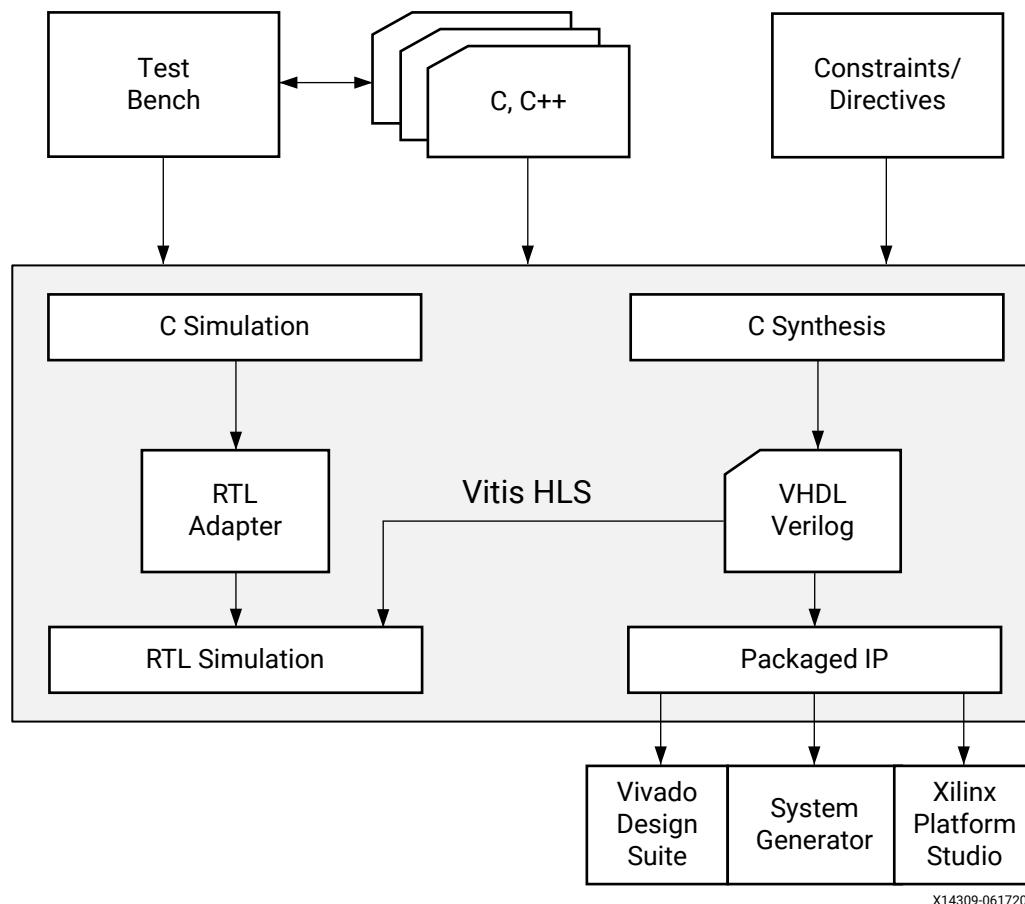
Table 19: Default Configuration

Configuration	Vivado	Vitis
<code>set_clock_uncertainty</code>	27%	27%
<code>config_compile -pipeline_loops</code>	64	64
<code>config_compile -name_max_length</code>	255	255
<code>config_export -vivado_optimization_level</code>	0	0
<code>config_export -vivado_phys_opt</code>	none	none
<code>config_rtl -module_auto_prefix</code>	true	true
<code>config_rtl -register_reset_num</code>	0	3
<code>config_schedule -enable_dsp_full_reg</code>	true	true
INTERFACE pragma defaults	IP mode	Kernel mode
<code>config_interface -m_axi_addr64</code>	true	true
<code>config_interface -m_axi_latency</code>	0	64
<code>config_interface -m_axi_alignment_byte_size</code>	1	64
<code>config_interface -m_axi_max_widen_bitwidth</code>	0	512
<code>config_interface -default_slave_interface</code>	<code>s_axilite</code>	<code>s_axilite</code>
<code>config_interface -m_axi_offset</code>	slave	slave

Working with Sources

The following figure illustrates the Vitis HLS design flow, showing the inputs and output files.

Figure 85: Vitis HLS Design Flow



X14309-061720

Vitis HLS inputs include:

- C functions written in C and C++11/C++14. This is the primary input to Vitis HLS. The function can contain a hierarchy of sub-functions.
- C functions with RTL blackbox content as described in [Adding RTL Blackbox Functions](#).
- Design Constraints that specify the clock period, clock uncertainty, and the device target.
- Directives are optional and direct the synthesis process to implement a specific behavior or optimization.
- C test bench and any associated files needed to simulate the C function prior to synthesis, and to verify the RTL output using C/RTL Co-simulation.

You can add the C input files, directives, and constraints to a project using the Vitis HLS graphical user interface (GUI), or using Tcl commands from the command prompt, as described in [Running Vitis HLS from the Command Line](#). You can also create a Tcl script, and execute the commands in batch mode.

The following are Vitis HLS outputs:

- Compiled object files (`.xo`).

This output lets you create compiled hardware functions for use in the Vitis application acceleration development flow. Vitis HLS produces this output when called as part of the compilation process from the Vitis tool flow, or when invoked as a stand-alone tool in the bottom up flow.

- RTL implementation files in hardware description language (HDL) formats.

This is a primary output from Vitis HLS. This flow lets you use C/C++ code as a source for hardware design in the Vitis tool flow. RTL IP produced by Vitis HLS is available in both Verilog (IEEE 1364-2001), and VHDL (IEEE 1076-2000) standards, and can be synthesized and implemented into Xilinx devices using the Vivado Design Suite.

- Report files.

Reports generated as a result of simulation, synthesis, C/RTL co-simulation, and generating output.

Coding C/C++ Functions

Coding Style

In any C program, the top-level function is called `main()`. In the Vitis HLS design flow, you can specify any sub-function below `main()` as the top-level function for synthesis. You *cannot* synthesize the top-level function `main()`. Following are additional rules:

- Only one function is allowed as the top-level function for synthesis.
- Any sub-functions in the hierarchy under the top-level function for synthesis are also synthesized.
- If you want to synthesize functions that are not in the hierarchy under the top-level function for synthesis, you must merge the functions into a single top-level function for synthesis.

C/C++ Language Support

Vitis HLS supports the C/C++ 11/14 for compilation/simulation. Vitis HLS supports many C and C++ language constructs, and all native data types for each language, including float and double types. However, synthesis is *not* supported for some constructs, including:

- Dynamic memory allocation: An FPGA has a fixed set of resources, and the dynamic creation and freeing of memory resources is not supported.
- Operating system (OS) operations: All data to and from the FPGA must be read from the input ports or written to output ports. OS operations, such as file read/write or OS queries like time and date, are not supported. Instead, the host application or test bench can perform these operations and pass the data into the function as function arguments.

Accessing Source Files in Git Repositories

When adding source files to your project, Vitis HLS offers an integrated view of GitHub repositories integrated into the tool. You can use this feature to work with your own repositories for managing source code for the project, or for linking to external repositories to download files for your design.

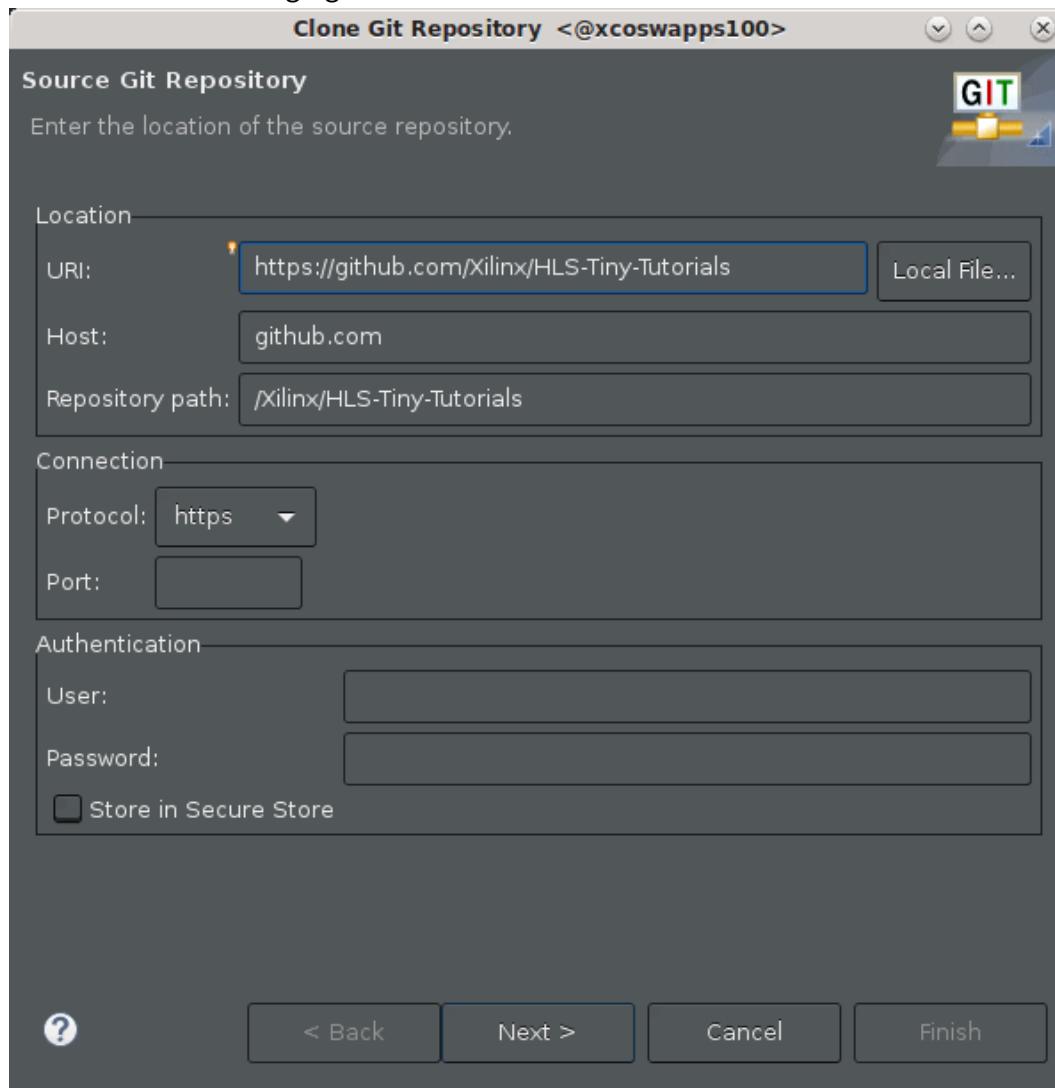
At the bottom of the Vitis HLS GUI, where the Console view is located, you will see the Git Repositories view.



TIP: If this view is not open, you can enable it using the **Window→Show View→Git Repository** menu command.

Clone a repository using the following steps.

1. Select the **Clone a Git Repository** command. This opens the Clone Git Repository wizard as shown in the following figure.



2. In the Source Git Repository page of the wizards, enter the following in for URL: <https://github.com/Xilinx/HLS-Tiny-Tutorials/tree/master>
This sets up the *Tiny Tutorials* repository as described in [Tutorials and Examples](#). Click **Next** to proceed.
3. In the Branch Selection page, select the **master** branch of the repository, or another branch as appropriate. Click **Next** to proceed.
4. In the Local Destination page, specify the **Destination Directory** where the repository will be cloned. Click **Next** to proceed.

At this time you should see the list of examples from the *Tiny Tutorials* repository. You can now use these files as source files for your own projects. You can also add an existing local repository to the Vitis HLS GUI, or create a new repository to help you manage projects.

Using Libraries in Vitis HLS

Vitis HLS Libraries

Vitis HLS provides foundational C libraries allowing common hardware design constructs and functions to be easily modeled in C and synthesized to RTL. The following C libraries are provided with Vitis HLS:

- [Arbitrary Precision Data Types Library](#): Arbitrary precision data types let your C code use variables with smaller bit-widths than standard C or C++ data types, to enable improved performance and reduced area in hardware.
- [Vitis HLS Math Library](#): Used to specify standard math operations for synthesis into RTL and implementation on Xilinx devices.
- [HLS Stream Library](#): For modeling and compiling streaming data structures.

You can use each of the C libraries in your design by including the library header file in your code. These header files are located in the `include` directory in the Vitis HLS installation area.



IMPORTANT! The header files for the Vitis HLS C libraries do not have to be in the `include` path if the design is used in Vitis HLS. The paths to the library header files are automatically added.

Vitis Libraries

In addition, the Vitis accelerated libraries are available for use with Vitis HLS, including common functions of math, statistics, Linear algebra and DSP; and also supporting domain specific applications, like vision and image processing, quantitative finance, database, data analytics, and data compression. Documentation for the libraries can be found at https://xilinx.github.io/Vitis_Libraries/. The libraries can be downloaded from https://github.com/Xilinx/Vitis_Libraries.

The Vitis™ libraries contain functions and constructs that are optimized for implementation on Xilinx devices. Using these libraries helps to ensure high quality of results (QoR); that the results of synthesis are a high-performance design that optimizes resource usage. Because the libraries are provided in C and C++, you can incorporate the libraries into your top-level function and simulate them to verify the functional correctness before synthesis.



TIP: The Vitis application acceleration libraries are not available for use on the Windows operating system.

Resolving References and Viewing #include Files

By default, the Vitis HLS GUI continually parses all header files to resolve coding references. Valid references allow the code to compile correctly of course, but also let you right-click on an `#include` statement to use the **Open Declaration** command to open the included file. You can also select a function name, variable, or data type and use the **Open Declaration** command to view its definition.

The GUI highlights unresolved references, as shown in the following figure:

Figure 86: Unresolved References

The screenshot shows a code editor window for a file named "sgd.cpp". The code is written in C++ and contains several pragmas for HLS (High-Level Synthesis). The left sidebar highlights specific lines of code with red markers, indicating unresolved references. The right sidebar lists these references, which are also highlighted in red in the code. The code itself is a dot product function and a look-up table function.

```
#include "src/host/typedefs.h"
#include "src/ocl/lut.h"

// Function to compute the dot product of data (feature) vector and parameter
FeatureType dotProduct(FeatureType param[NUM FEATURES],
                        DataType      feature[NUM FEATURES])
{
    //#pragma HLS INLINE

    FeatureType result = 0;
    DOT: for (int i = 0; i < NUM FEATURES / PAR_FACTOR; i++)
    {
        #pragma HLS PIPELINE II=1
        DOT_INNER: for(int j = 0; j < PAR_FACTOR; j++)
        {
            FeatureType term = param[i*PAR_FACTOR+j] * feature[i*PAR_FACTOR+j];
            result += term;
        }
    }
    return result;
}

// values of sigmoid function stored in a look-up table
FeatureType useLUT(FeatureType in)
{
    //#pragma HLS INLINE
    IdxFixed index;
    if (in < 0)
    {
        in = -in;
        index = (IdxFixed)LUT_SIZE - (IdxFixed)((TmpFixed)in) << (LUTIN_TWIDHT -
    }
    else
        index = ((TmpFixed)in) << (LUTIN_TWIDHT - LUTIN_IWIDHT);
    return lut[index];
}
```

- Left sidebar: Highlights unresolved references at the line number of the source code.
- Right sidebar: Displays unresolved references relative to the whole file.

Unresolved references occur when code defined in a header file (.h or .hpp extension) cannot be resolved. The primary causes of unresolved references are:

- The code was recently added to the file.

If the code is new, ensure the header file is saved. After saving the header file, Vitis HLS automatically indexes the header files and updates the code references.

- The header file is not in the search path.

Ensure the header file is included in the C code using an `#include` statement, and the header file is found in the same directory as the source C file, or the location to the header file is in the search path.



TIP: To explicitly add the search path for a source file, select **Project**→**Project Settings**, click **Synthesis** or **Simulation**, and use the **Edit CFLAGS** or **Edit CXXFLAGS** commands for the source file as discussed in [Creating a New Vitis HLS Project](#).

- Automatic indexing has been disabled.

Ensure that Vitis HLS is parsing all header files automatically. Select **Project**→**Project Settings**, click **General**, and make sure **Disable Parsing All Header Files** is deselected.



TIP: To manually force Vitis HLS to index all C files, select the **Project**→**Index C Source** command from the main menu. This enables the tool to open the declaration of a selected function, variable, or data type if it occurs in an included file.

Resolving Comments in the Source Code

In some localizations, non-English comments in the source file appears as strange characters. This can be corrected using the following steps:

- Right-click the project in the **Explorer** view and select the **Properties** menu command.
- Select the **Resource** section in the left side of the dialog box.
- Under **Text file encoding**, select the **Other** radio button, and choose appropriate encoding from the drop-down menu.
- Select **Apply and Close** to accept the change.

Adding RTL Blackbox Functions

The RTL blackbox enables the use of existing Verilog RTL IP in an HLS project. This lets you add RTL code to your C/C++ code for synthesis of the project by Vitis HLS. The RTL IP can be used in a sequential, pipeline, or dataflow region. Refer to [Vitis-HLS-Introductory-Examples/Misc/rtl_as_blackbox](#) on Github for examples of this technique.



TIP: Adding an RTL blackbox to your design will restrict the tool from outputting VHDL code, Because the RTL blackbox must be Verilog, the output will be Verilog only.

Integrating RTL IP into a Vitis HLS project requires the following files:

- C function signature for the RTL code. This can be placed into a header (.h) file.
- Blackbox JSON description file as discussed in [JSON File for RTL Blackbox](#).
- RTL IP files.

To use the RTL blackbox in an HLS project, use the following steps.

- Call the C function signature from within your top-level function, or a sub-function in the Vitis HLS project.

2. Add the blackbox JSON description file to your HLS project using the **Add Files** command from the Vitis HLS IDE as discussed in [Creating a New Vitis HLS Project](#), or using the `add_files` command:

```
add_files -blackbox my_file.json
```



TIP: As explained in the next section, the new RTL Blackbox wizard can help you generate the JSON file and add the RTL IP to your project.

3. Run the Vitis HLS design flow for simulation, synthesis, and co-simulation as usual.

Requirements and Limitations

RTL IP used in the RTL blackbox feature have the following requirements:

- Should be Verilog (.v) code.
- Must have a unique clock signal, and a unique active-High reset signal.
- Must have a CE signal that is used to enable or stall the RTL IP.
- Must use the `ap_ctrl_chain` protocol as described in [Block-Level Control Protocols](#).

Within Vitis HLS, the RTL blackbox feature:

- Supports only C++.
- Cannot connect to top-level interface I/O signals.
- Cannot directly serve as the design-under-test (DUT).
- Does not support `struct` or `class` type interfaces.
- Supports the following interface protocols as described in [JSON File for RTL Blackbox](#):
 - **hls::stream:** The RTL blackbox IP supports the `hls::stream` interface. When this data type is used in the C function, use a FIFO RTL port protocol for this argument in the RTL blackbox IP.
 - **Arrays:** The RTL blackbox IP supports RAM interface for arrays. For array arguments in the C function, use one of the following RTL port protocols for the corresponding argument in the RTL blackbox IP:
 - Single port RAM – RAM_1P
 - Dual port RAM – RAM_T2P
 - **Scalars and Input Pointers:** The RTL Blackbox IP supports C scalars and input pointers only in sequential and pipeline regions. They are not supported in a dataflow region. When these constructs are used in the C function, use `wire` port protocol in the RTL IP.
 - **Inout and Output Pointers:** The RTL blackbox IP supports inout and output pointers only in sequential and pipeline regions. They are not supported in a dataflow region. When these constructs are used in the C function, the RTL IP should use `ap_vld` for output pointers, and `ap_ovld` for inout pointers.

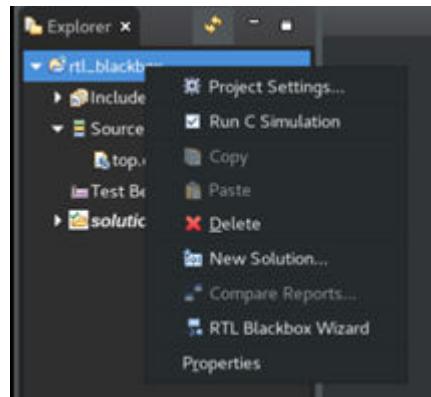


TIP: All other Vitis HLS design restrictions also apply when using RTL blackbox in your project.

Using the RTL Blackbox Wizard

Navigate to the project, right-click to open the **RTL Blackbox Wizard** as shown in the following figure:

Figure 87: Opening RTL Blackbox Wizard

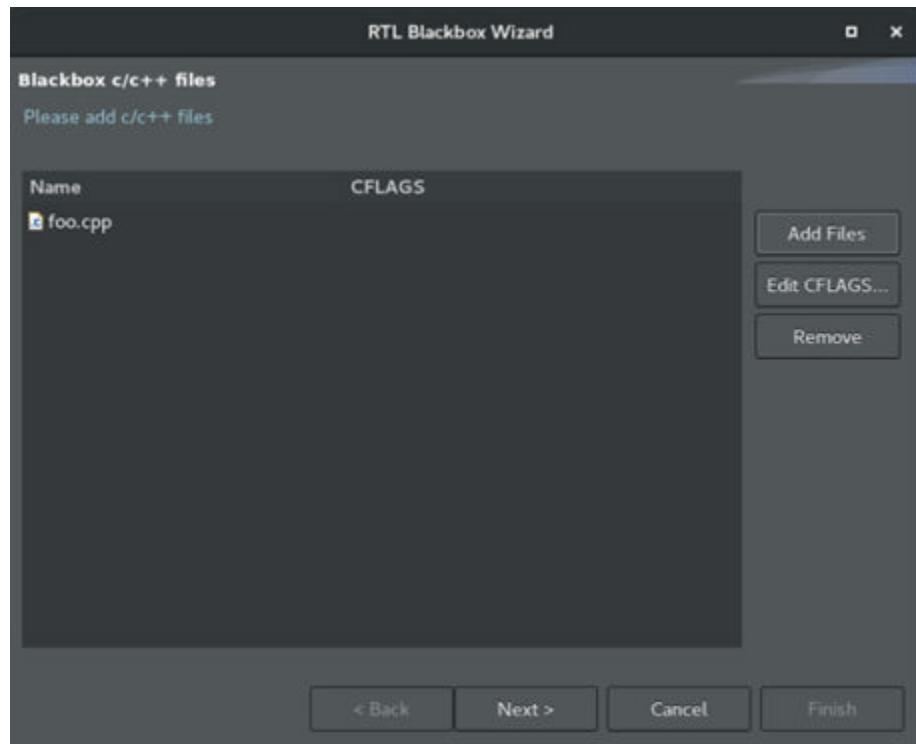


The Wizard is organized into pages that break down the process for creating a JSON file. To navigate between pages, click **Next** and select **Back**. Once the options are finalized, you can generate a JSON by clicking **OK**. Each of the following section describes each page and its input options.

C++ Model and Header Files

In the Blackbox C/C++ files page, you provide the C++ files which form the functional model of the RTL IP. This C++ model is only used during C++ simulation and C++/RTL co-simulation. The RTL IP is combined with Vitis HLS results to form the output of synthesis.

Figure 88: Blackbox C/C++ files Page



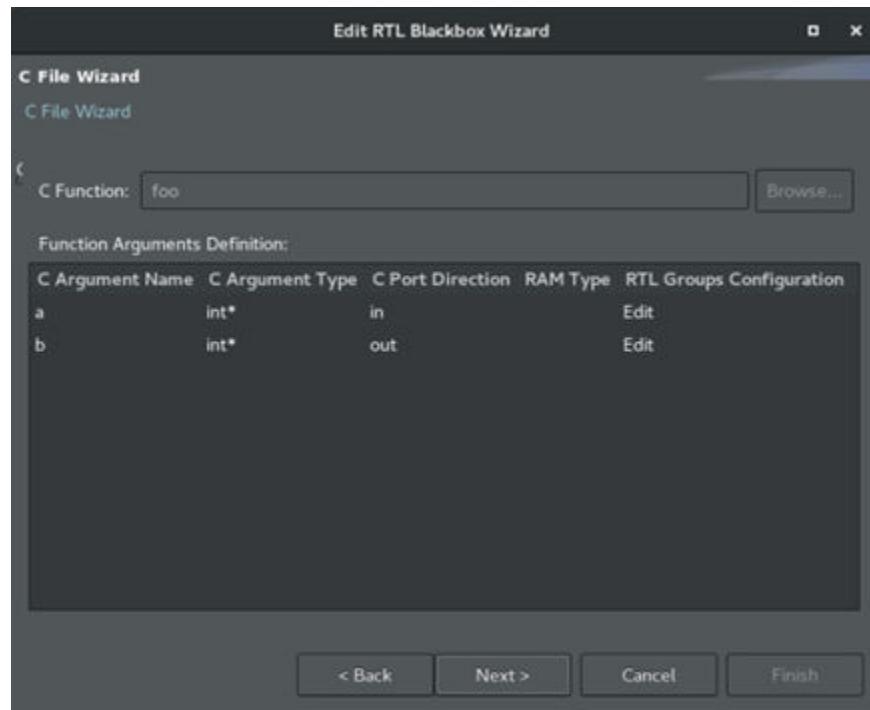
In this page, you can perform the following:

- Click **Add Files** to add files.
- Click **Edit CFLAGS** to provide a linker flag to the functional C model.
- Click **Next** to proceed.

The C File Wizard page lets you specify the values used for the C functional model of the RTL IP. The fields include:

- **C Function:** Specify the C function name of the RTL IP.
- **C Argument Name:** Specify the name(s) of the function arguments. These should relate to the ports on the IP.
- **C Argument Type:** Specify the data type used for each argument.
- **C Port Direction:** Specify the port direction of the argument, corresponding to the port in the IP.
- **RAM Type:** Specify the RAM type used at the interface.
- **RTL Group Configuration:** Specifies the corresponding RTL signal name.

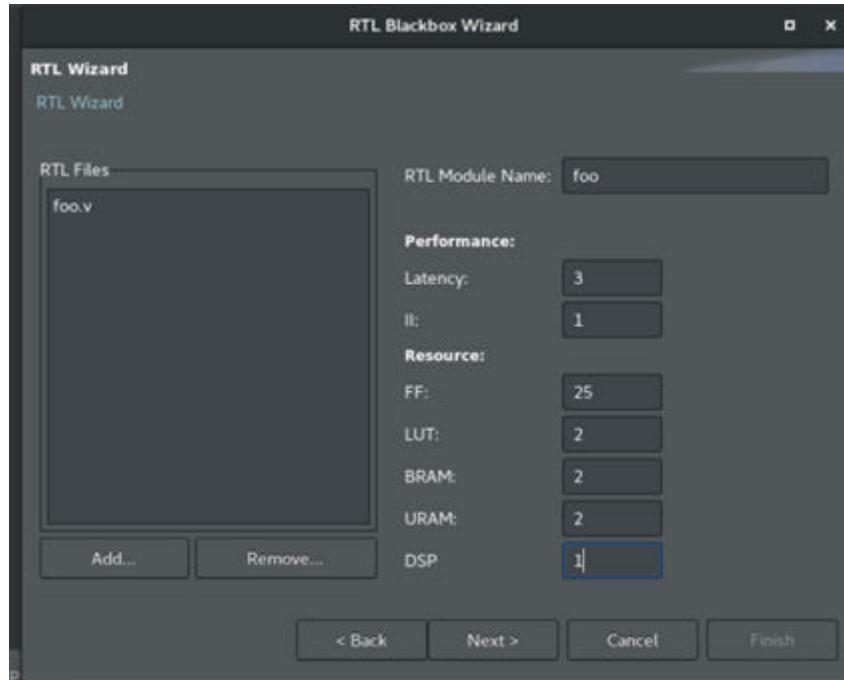
Figure 89: C File Wizard Page



Click **Next** to proceed.

RTL IP Definition

Figure 90: RTL Blackbox Wizard

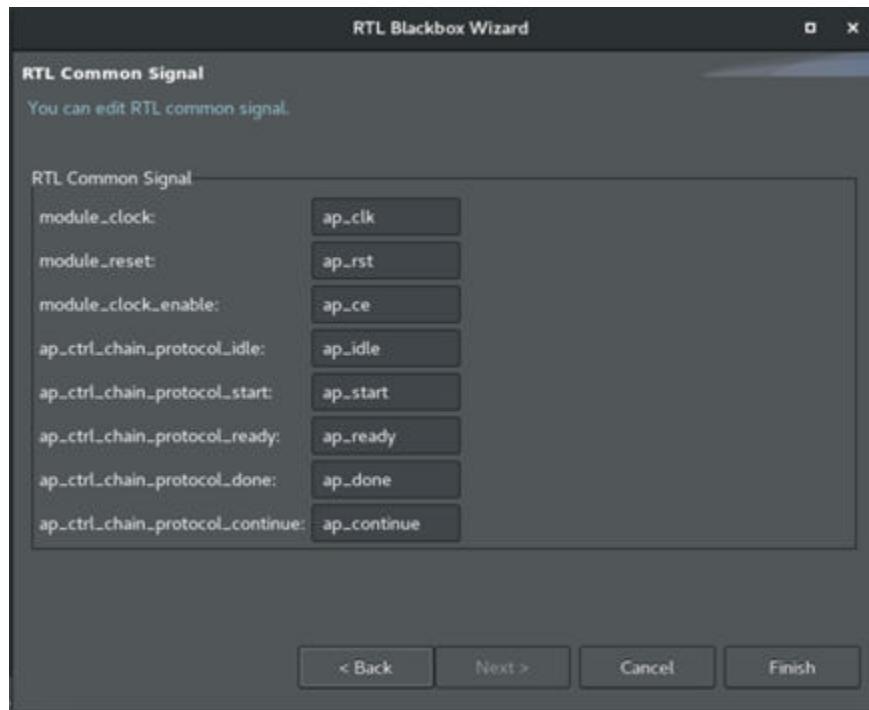


The RTL Wizard page lets you define the RTL source for the IP. The fields to define include:

- RTL Files:** This option is used to add or remove the pre existing RTL IP files.
- RTL Module Name:** Specify the top level RTL IP module name in this field.
- Performance:** Specify performance targets for the IP.
 - Latency:** Latency is the time required for the design to complete. Specify the Latency information in this field.
 - II:** Define the target II (Initiation Interval). This is the number of clocks cycles before new input can be applied.
- Resource:** Specify the device resource utilization for the RTL IP. The resource information provided here will be combined with utilization from synthesis to report the overall design resource utilization. You should be able to extract this information from the Vivado Design Suite

Click **Next** to proceed to the RTL Common Signal page, as shown below.

Figure 91: RTL Common Signals



- **module_clock:** Specify the name of the clock used in the RTL IP.
- **module_reset:** Specify the name of the reset signal used in the IP.
- **module_clock_enable:** Specify the name of the clock enable signal in the IP.
- **ap_ctrl_chain_protocol_start:** Specify the name of the block control start signal used in the IP.
- **ap_ctrl_chain_protocol_ready:** Specify the name of the block control ready signal used in the IP.
- **ap_ctrl_chain_protocol_done:** Specify the name of the block control done signal used in the IP.
- **ap_ctrl_chain_protocol_continue:** Specify the name of the block control continue signal used in the RTL IP.

Click **Finish** to automatically generate a JSON file for the specified IP. This can be confirmed through the log message as shown below.

Log Message:

```
" [2019-08-29 16:51:10] RTL Blackbox Wizard Information: the "foo.json" file has been created in the rtl_blackbox/Source folder."
```

The JSON file can be accessed through the Source file folder, and will be generated as described in the next section.

JSON File for RTL Blackbox

JSON File Format

The following table describes the JSON file format:

Table 20: JSON File Format

Item	Attribute	Description
c_function_name		The C++ function name for the blackbox. The <code>c_function_name</code> must be consistent with the C function simulation model.
rtl_top_module_name		The RTL function name for the blackbox. The <code>rtl_top_module_name</code> must be consistent with the <code>c_function_name</code> .
c_files	c_file	Specifies the C file used for the blackbox module.
	cflag	Provides any compile option necessary for the corresponding C file.
rtl_files		Specifies the RTL files for the blackbox module.

Table 20: JSON File Format (cont'd)

Item	Attribute	Description
c_parameters	c_name	<p>Specifies the name of the argument used for the black box C++ function.</p> <p>Unused <code>c_parameters</code> should be deleted from the template.</p>
	c_port_direction	<p>The access direction for the corresponding C argument.</p> <ul style="list-style-type: none"> <code>in</code>: Read only by blackbox C++ function. <code>out</code>: Write only by blackbox C++ function. <code>inout</code>: Will both read and write by blackbox C++ function.
	RAM_type	<p>Specifies the RAM type to use if the corresponding C argument uses the RTL RAM protocol. Two type of RAM are used:</p> <ul style="list-style-type: none"> RAM_1P: For 1 port RAM module RAM_T2P: For 2 port RAM module <p>Omit this attribute when the corresponding C argument is not using RTL 'RAM' protocol.</p>
	rtl_ports	<p>Specifies the RTL port protocol signals for the corresponding C argument (<code>c_name</code>). Every <code>c_parameter</code> should be associated with an <code>rtl_port</code>. Five type of RTL port protocols are used. Refer to the <i>RTL Port Protocols</i> table for additional details.</p> <ul style="list-style-type: none"> <code>wire</code>: An argument can be mapped to <code>wire</code> if it is a scalar or pointer with 'in' direction. <code>ap_vld</code>: An argument can be mapped to <code>ap_vld</code> if it uses pointer with 'out' direction. <code>ap_ovld</code>: An argument can be mapped to <code>ap_ovld</code> if it use a pointer with an inout direction. <code>FIFO</code>: An argument can be mapped to <code>FIFO</code> if it uses the <code>hls::stream</code> data type. <code>RAM</code>: An argument can be mapped to <code>RAM</code> if it uses an array type. The array type supports inout directions. <p>The specified RTL port protocols have associated control signals, which also need to be specified in the JSON file.</p>
c_return	c_port_direction	It must be <code>out</code> .
	rtl_ports	Specifies the corresponding RTL port name used in the RTL blackbox IP.

Table 20: JSON File Format (cont'd)

Item	Attribute	Description
rtl_common_signal	module_clock	The unique clock signal for RTL blackbox module.
	module_reset	Specifies the reset signal for RTL blackbox module. The reset signal must be active-High or positive valid.
	module_clock_enable	Specifies the clock enable signal for the RTL blackbox module. The enable signal must be active-High or positive valid.
	ap_ctrl_chain_protocol_idle	The <code>ap_idle</code> signal in the <code>ap_ctrl_chain</code> protocol for the RTL blackbox module.
	ap_ctrl_chain_protocol_start	The <code>ap_start</code> signal in the <code>ap_ctrl_chain</code> protocol for the RTL blackbox module.
	ap_ctrl_chain_protocol_ready	The <code>ap_ready</code> signal in the <code>ap_ctrl_chain</code> protocol for the RTL blackbox IP.
	ap_ctrl_chain_protocol_done	The <code>ap_done</code> signal in the <code>ap_ctrl_chain</code> protocol for blackbox RTL module.
rtl_performance	latency	Specifies the Latency of the RTL blackbox module. It must be a non-negative integer value. For Combinatorial RTL IP specify 0, otherwise specify the exact latency of the RTL module.
	II	Number of clock cycles before the function can accept new input data. It must be non-negative integer value. 0 means the blackbox can not be pipelined. Otherwise, it means the blackbox module is pipelined.
rtl_resource_usage	FF	Specifies the register utilization for the RTL blackbox module.
	LUT	Specifies the LUT utilization for the RTL blackbox module.
	BRAM	Specifies the block RAM utilization for the RTL blackbox module.
	URAM	Specifies the URAM utilization for the RTL blackbox module.
	DSP	Specifies the DSP utilization for the RTL blackbox module.

Table 21: RTL Port Protocols

RTL Port Protocol	RAM Type	C Port Direction	Attribute	User-Defined Name	Notes
wire		in	data_read_in		
ap_vld		out	data_write_out	Specifies a user defined name used in the RTL blackbox IP. As an example for wire, if the RTL port name is "flag" then the JSON FILE format is "data_read_in" : "flag".	
			data_write_valid		
ap_ovld		inout	data_read_in		Must be negative valid.
			data_write_out		
			data_write_valid		
FIFO		in	FIFO_empty_flag	Specifies a user defined name used in the RTL blackbox IP. As an example for FIFO, if the RTL port name is "flag" then the JSON FILE format is "data_read_in" : "flag".	Must be negative valid.
			FIFO_read_enable		
			FIFO_data_read_in		
		out	FIFO_full_flag		Must be negative valid.
			FIFO_write_enable		
			FIFO_data_write_out		
RAM	RAM_1P	in	RAM_address	Specifies a user defined name used in the RTL blackbox IP. As an example for RAM, if the RTL port name is "flag" then the JSON FILE format is "data_read_in" : "flag".	
			RAM_clock_enable		
			RAM_data_read_in		
		out	RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
		inout	RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
			RAM_data_read_in		

Table 21: RTL Port Protocols (cont'd)

RTL Port Protocol	RAM Type	C Port Direction	Attribute	User-Defined Name	Notes
RAM	RAM_T2P	in	RAM_address RAM_clock_enable RAM_data_read_in RAM_address_snd RAM_clock_enable_snd RAM_data_read_in_snd	Specifies a user defined name used in the RTL blackbox IP. As an example for wire, if the RTL port name is "flag" then the JSON FILE format is "data_read_in" : "flag".	Signals with _snd belong to the second port of the RAM. Signals without _snd belong to the first port.
		out	RAM_address RAM_clock_enable RAM_write_enable RAM_data_write_out RAM_address_snd RAM_clock_enable_snd RAM_write_enable_snd RAM_data_write_out_snd		
		inout	RAM_address RAM_clock_enable RAM_write_enable RAM_data_write_out RAM_data_read_in RAM_address_snd RAM_clock_enable_snd RAM_write_enable_snd RAM_data_write_out_snd RAM_data_read_in_snd		

Note: The behavioral C-function model for the RTL blackbox must also adhere to the recommended HLS coding styles.

JSON File Example

This section provides details on manually writing the JSON file required for the RTL blackbox. The following is an example of a JSON file:

```
{
  "c_function_name" : "foo",
  "rtl_top_module_name" : "foo",
  "c_files" :
  [
    {
      "c_file" : "../../a/top.cpp",
      "cflag" : ""
    }
  ]
}
```

```
        "c_file" : "xx.cpp",
        "cflag" : "-D KF"
    ],
],
"rtl_files" : [
    "../foo.v",
    "xx.v"
],
"c_parameters" : [ {
    "c_name" : "a",
    "c_port_direction" : "in",
    "rtl_ports" : {
        "data_read_in" : "a"
    }
},
{
    "c_name" : "b",
    "c_port_direction" : "in",
    "rtl_ports" : {
        "data_read_in" : "b"
    }
},
{
    "c_name" : "c",
    "c_port_direction" : "out",
    "rtl_ports" : {
        "data_write_out" : "c",
        "data_write_valid" : "c_ap_vld"
    }
},
{
    "c_name" : "d",
    "c_port_direction" : "inout",
    "rtl_ports" : {
        "data_read_in" : "d_i",
        "data_write_out" : "d_o",
        "data_write_valid" : "d_o_ap_vld"
    }
},
{
    "c_name" : "e",
    "c_port_direction" : "in",
    "rtl_ports" : {
        "FIFO_empty_flag" : "e_empty_n",
        "FIFO_read_enable" : "e_read",
        "FIFO_data_read_in" : "e"
    }
},
{
    "c_name" : "f",
    "c_port_direction" : "out",
    "rtl_ports" : {
        "FIFO_full_flag" : "f_full_n",
        "FIFO_write_enable" : "f_write",
        "FIFO_data_write_out" : "f"
    }
},
{
    "c_name" : "g",
    "c_port_direction" : "in",
    "RAM_type" : "RAM_1P",
    "rtl_ports" : {
        "RAM_address" : "g_address0",
        "RAM_data_out" : "g_data0"
    }
}]]
```

```
        "RAM_clock_enable" : "g_ce0",
        "RAM_data_read_in" : "g_q0"
    }
},
{
    "c_name" : "h",
    "c_port_direction" : "out",
    "RAM_type" : "RAM_1P",
    "rtl_ports" : [
        "RAM_address" : "h_address0",
        "RAM_clock_enable" : "h_ce0",
        "RAM_write_enable" : "h_we0",
        "RAM_data_write_out" : "h_d0"
    ]
},
{
    "c_name" : "i",
    "c_port_direction" : "inout",
    "RAM_type" : "RAM_1P",
    "rtl_ports" : [
        "RAM_address" : "i_address0",
        "RAM_clock_enable" : "i_ce0",
        "RAM_write_enable" : "i_we0",
        "RAM_data_write_out" : "i_d0",
        "RAM_data_read_in" : "i_q0"
    ]
},
{
    "c_name" : "j",
    "c_port_direction" : "in",
    "RAM_type" : "RAM_T2P",
    "rtl_ports" : [
        "RAM_address" : "j_address0",
        "RAM_clock_enable" : "j_ce0",
        "RAM_data_read_in" : "j_q0",
        "RAM_address_snd" : "j_address1",
        "RAM_clock_enable_snd" : "j_ce1",
        "RAM_data_read_in_snd" : "j_q1"
    ]
},
{
    "c_name" : "k",
    "c_port_direction" : "out",
    "RAM_type" : "RAM_T2P",
    "rtl_ports" : [
        "RAM_address" : "k_address0",
        "RAM_clock_enable" : "k_ce0",
        "RAM_write_enable" : "k_we0",
        "RAM_data_write_out" : "k_d0",
        "RAM_address_snd" : "k_address1",
        "RAM_clock_enable_snd" : "k_ce1",
        "RAM_write_enable_snd" : "k_we1",
        "RAM_data_write_out_snd" : "k_d1"
    ]
},
{
    "c_name" : "l",
    "c_port_direction" : "inout",
    "RAM_type" : "RAM_T2P",
    "rtl_ports" : [
        "RAM_address" : "l_address0",
        "RAM_clock_enable" : "l_ce0",
        "RAM_write_enable" : "l_we0",

```

```
        "RAM_data_write_out" : "l_d0",
        "RAM_data_read_in" : "l_q0",
        "RAM_address_snd" : "l_address1",
        "RAM_clock_enable_snd" : "l_ce1",
        "RAM_write_enable_snd" : "l_we1",
        "RAM_data_write_out_snd" : "l_d1",
        "RAM_data_read_in_snd" : "l_q1"
    }
},
"c_return" : {
    "c_port_direction" : "out",
    "rtl_ports" : {
        "data_write_out" : "ap_return"
    }
},
"rtl_common_signal" : {
    "module_clock" : "ap_clk",
    "module_reset" : "ap_rst",
    "module_clock_enable" : "ap_ce",
    "ap_ctrl_chain_protocol_idle" : "ap_idle",
    "ap_ctrl_chain_protocol_start" : "ap_start",
    "ap_ctrl_chain_protocol_ready" : "ap_ready",
    "ap_ctrl_chain_protocol_done" : "ap_done",
    "ap_ctrl_chain_protocol_continue" : "ap_continue"
},
"rtl_performance" : {
    "latency" : "6",
    "II" : "2"
},
"rtl_resource_usage" : {
    "FF" : "0",
    "LUT" : "0",
    "BRAM" : "0",
    "URAM" : "0",
    "DSP" : "0"
}
}
```

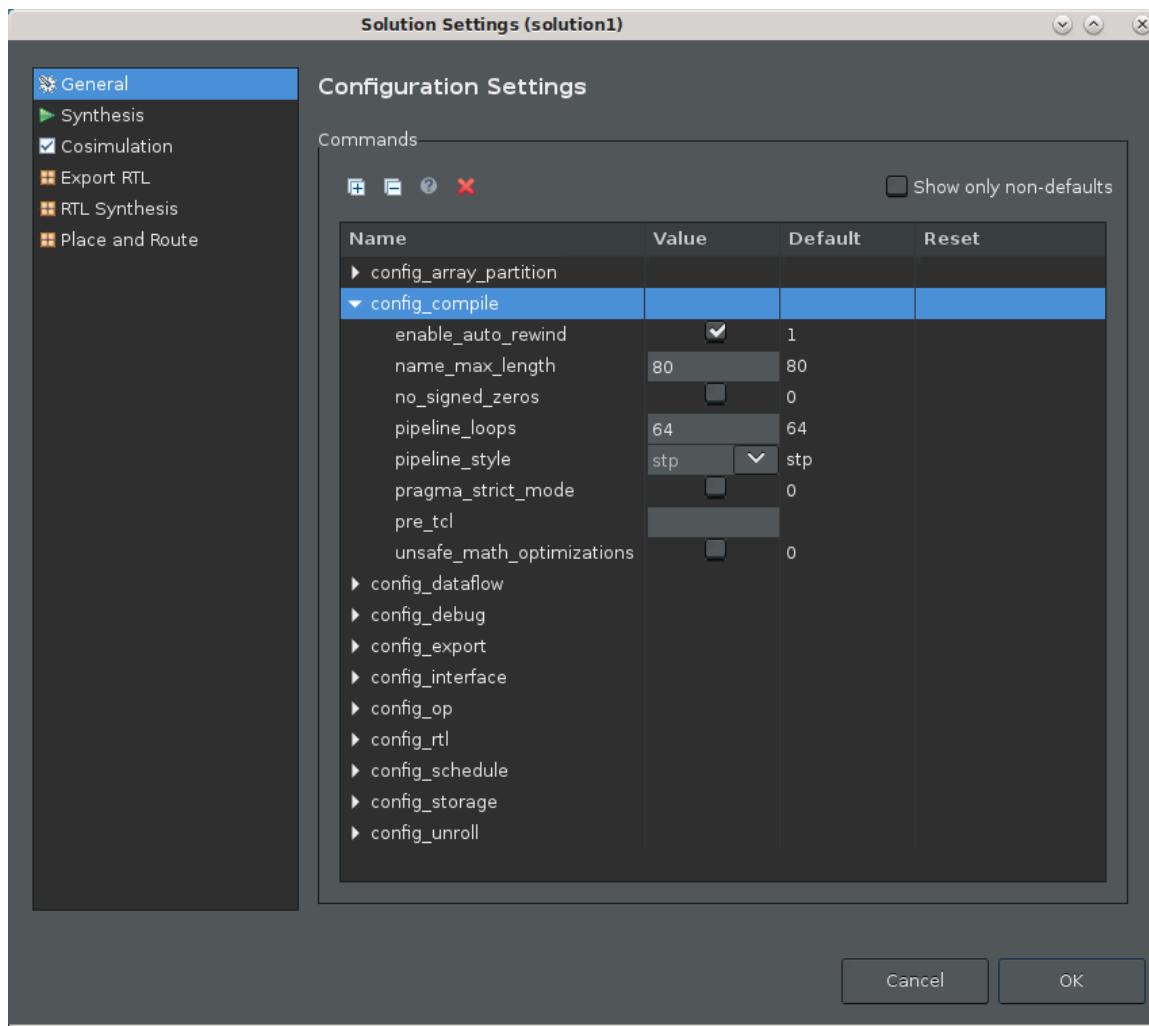
Setting Configuration Options

After the project and solution have been created, you can configure default settings of the Vitis HLS tool using the **Solution** → **Solution Settings** menu command. This command opens the Solution Settings dialog box for the currently active solution.



TIP: If you have created multiple solutions for your project, as described in [Creating Additional Solutions](#), you can make a solution active by right clicking on a solution in the Explorer view and using the **Set Active Solution** command. You can also open the Solution Settings dialog box for a specific solution by right-clicking the solution and using the **Solution Settings** command.

Figure 92: Solution Settings Dialog Box



The Solutions Setting dialog box provides access to the following settings:

- **General:** Displays the Configuration Settings page for the current solution, listing settings that generally apply to the Vitis HLS tool overall.
- **Synthesis:** Synthesis settings are initially defined when the project is created as described in [Creating a New Vitis HLS Project](#).
- **Cosimulation:** These settings control the C/RTL Co-simulation feature as described in [C/RTL Co-Simulation in Vitis HLS](#).
- **Export:** These settings affect the output generated by Vitis HLS as described in [Exporting the RTL Design](#).
- **RTL Synthesis:** These settings affect the results and reports generated by Vivado synthesis as described in [Exporting the RTL Design](#).

- **Place and Route:** These settings affect the results and reports generated by Vivado implementation as described in [Exporting the RTL Design](#).

Configuration Settings

On the Configuration Settings page, as displayed in the figure above, you have access to the various configuration commands like `config_compile` and `config_interface`. These commands are described in detail in [Configuration Commands](#).

Select one of the listed configuration commands, and click the **Expand All (+)** command to expand the selected configuration command to view the available options. You can edit the options for the selected command, or use the **Reset all (X)** command to restore the selected configuration to its default setting.

Use the **Collapse All (-)** command to collapse any selected configuration command.

Use the **Help (?)** command to open a window that provides a text description of the selected configuration command and all its options.

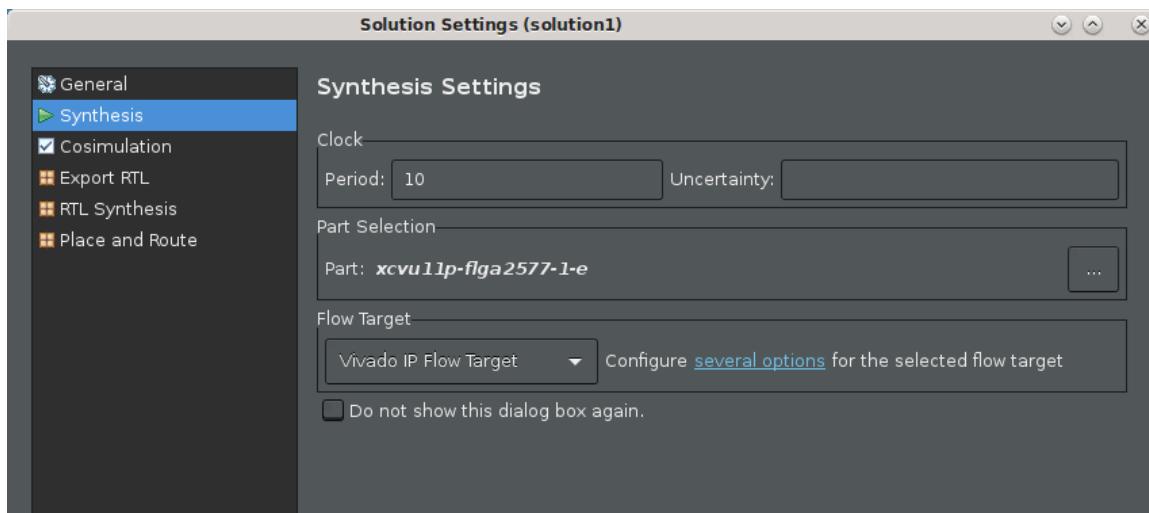
Enable the **Show only non-defaults** check box to only display the configuration commands that have been modified from their default values.

Click **OK** to confirm the settings of the various configuration commands and close the Solution Settings dialog box. Click **Cancel** to cancel any changes and close the dialog box.

Synthesis Settings

On the Synthesis Settings page, as shown in the following figure, you have access to the various settings to drive the synthesis process, such as the target Xilinx device, the clock period and uncertainty, and the target flow for the solution.

Figure 93: Synthesis Settings Page



- Specify the clock period in units of nanoseconds (ns), or as a frequency value specified with the MHz suffix (for example, 150 MHz). Refer to [Specifying the Clock Frequency](#) for more information.
- Specify the clock uncertainty used for synthesis as the clock period minus the clock uncertainty. Vitis HLS uses internal models to estimate the delay of the operations for each device. The clock uncertainty value provides a controllable margin to account for any increases in net delays due to RTL logic synthesis, place, and route. Specify as a value in ns, or as a percentage of the clock period. The default clock uncertainty is 27% of the clock period.
- Specify the target device (Part) for your project by clicking the Browse button (...) to open the Device Selection Dialog box to select a device or board for the solution. You can click the **Search** filter to reduce the number of devices listed.
- Select the Flow Target as explained in [Vitis HLS Flow Overview](#).

Specifying the Clock Frequency

For C and C++ designs only a single clock is supported. The same clock is applied to all functions in the design.

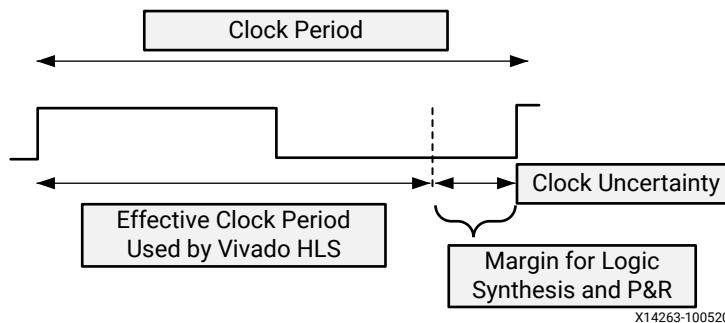
The clock period, in ns, is set in the **Solutions → Solutions Setting**. The default clock period is 10 ns. Vitis HLS uses the concept of a clock uncertainty to provide a user defined timing margin. You can define the clock uncertainty for your design using the Solutions Setting dialog box as well. The default clock uncertainty, when it is not specified, is 27% of the clock period.



TIP: You can also set the clock period using the [create_clock](#) Tcl command, and the clock uncertainty using the [set_clock_uncertainty](#) Tcl command.

Using the clock frequency and device target information Vitis HLS estimates the timing of operations in the design but it cannot know the final component placement and net routing: these operations are performed by logic synthesis of the output RTL. As such, Vitis HLS cannot know the exact delays.

To calculate the clock period used for synthesis, Vitis HLS subtracts the clock uncertainty from the clock period, as shown in the following figure.

Figure 94: Clock Period and Margin

This provides a user specified margin to ensure downstream processes, such as logic synthesis and place & route, have enough timing margin to complete their operations. If the FPGA is mostly used the placement of cells and routing of nets to connect the cells might not be ideal and might result in a design with larger than expected timing delays. For a situation such as this, an increased timing margin ensures Vitis HLS does not create a design with too much logic packed into each clock cycle and allows RTL synthesis to satisfy timing in cases with less than ideal placement and routing options.

Vitis HLS aims to satisfy all constraints: timing, throughput, latency. However, if a constraints cannot be satisfied, Vitis HLS always outputs an RTL design.

If the timing constraints inferred by the clock period cannot be met Vitis HLS issues message SCHED-644, as shown below, and creates a design with the best achievable performance.

```
@W [SCHED-644] Max operation delay (<operation_name> 2.39ns) exceeds the
effective
cycle time
```

Even if Vitis HLS cannot satisfy the timing requirements for a particular path, it still achieves timing on all other paths. This behavior allows you to evaluate if higher optimization levels or special handling of those failing paths by downstream logic syntheses can pull-in and ultimately satisfy the timing.

IMPORTANT! It is important to review the constraint report after synthesis to determine if all constraints is met: the fact that Vitis HLS produces an output design does not guarantee the design meets all performance constraints. Review the Performance Estimates section of the design report.

A design report is generated for each function in the hierarchy when synthesis completes and can be viewed in the solution reports folder. The worse case timing for the entire design is reported as the worst case in each function report. There is no need to review every report in the hierarchy.

If the timing violations are too severe to be further optimized and corrected by downstream processes, review the techniques for specifying an exact latency and specifying exact implementation cores before considering a faster target technology.

Clock and Reset Ports

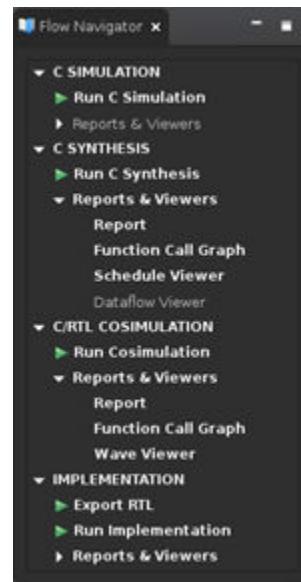
If the design takes more than 1 cycle to complete operation, a clock-enable port (`ap_ce`) can optionally be added to the entire block using the `config_interface` command, or in the Vitis HLS GUI using the **Solution → Solution Settings → General** command.

The operation of the reset is described in [Controlling Initialization and Reset Behavior](#), and can be modified using the `config_rtl` command, also available in the Solutions Settings dialog box.

Using the Flow Navigator

The Flow Navigator is a process flow representation of the Vitis HLS design flow. Each step in the process is represented by actions that you can launch to work through the flow. All viewers and reports are also available through the Flow Navigator as each step is completed.

Figure 95: Flow Navigator



The different steps represented in the Flow Navigator include:

- **C SIMULATION**: opens the C Simulation dialog box, and lists the available reports after simulation has been run, as described in [Verifying Code with C Simulation](#).
- **C SYNTHESIS**: opens the C Synthesis dialog box, and lists the available reports after synthesis has been run, as discussed in [Synthesizing the Code](#).
- **C/RTL COSIMULATION**: opens the C/RTL Cosimulation dialog box, and lists the available reports after simulation has been run, as described in [C/RTL Co-Simulation in Vitis HLS](#).

- **IMPLEMENTATION:** lets you specify the format and location of the exported RTL file from Vitis HLS as discussed in [Exporting the RTL Design](#), and also run Vivado synthesis and implementation to generate more detailed utilization and timing reports.



TIP: You can cancel Simulation, Synthesis, C/RTL Cosimulation, or Implementation using the **Stop** command from the Flow Navigator.

Verifying Code with C Simulation

Verification in the Vitis HLS flow can be separated into two distinct processes.

- Pre-synthesis validation that the C program correctly implements the required functionality.
- Post-synthesis verification that the generated RTL code performs as expected.

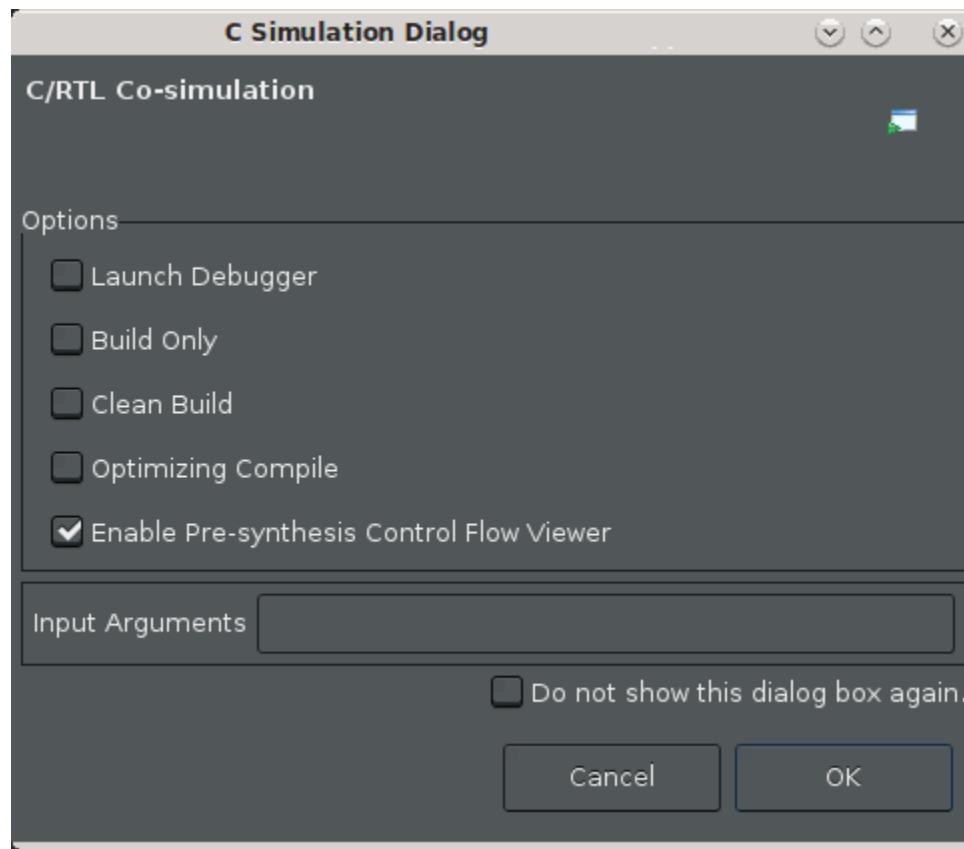
Both processes are referred to as simulation: C simulation and C/RTL co-simulation.

Before synthesis, the function to be synthesized should be validated with a test bench using C simulation. A C test bench includes a `main()` top-level function, that calls the function to be synthesized by the Vitis HLS project. The test bench can also include other functions. An ideal test bench has the following features:

- The test bench is self-checking, and validates that the results from the function to be synthesized are correct.
- If the results are correct the test bench returns a value of 0 to `main()`. Otherwise, the test bench should return any non-zero value.

In the Vitis HLS GUI, clicking the **Run C Simulation** toolbar button  opens the C Simulation Dialog box, as shown in the following figure:

Figure 96: C Simulation Dialog Box



The options for the C Simulation Dialog box include the following:

- **Launch Debugger:** This compiles the C code and automatically opens the Debug perspective. From within the Debug perspective, the Synthesis perspective button (top left) can be used to return the windows to the Synthesis perspective.
- **Build Only:** Compiles the source code and test bench, but does not run simulation. This option can be used to test the compilation process and resolve any issues with the build prior to running simulation. It generates a `csim.exe` file that can be used to launch simulation from a command shell.
- **Clean Build:** Remove any existing executable and object files from the project before compiling the code.
- **Optimizing Compile:** By default the design is compiled with debug information enabled, allowing the compilation to be analyzed and debugged. The Optimizing Compile option uses a higher level of optimization effort when compiling the design, but does not add information required by the debugger. This increases the compile time but should reduce the simulation runtime.



TIP: The Launch Debugger and Optimizing Compile options are mutually exclusive. Selecting one in the C Simulation Dialog box disables the other.

- Enable Pre-Synthesis Control Flow Viewer:** Generates the Pre-synthesis Control Flow report as described in [Pre-Synthesis Control Flow](#).
- Input Arguments:** Specify any inputs required by your test bench `main()` function.
- Do not show this dialog box again:** Lets you disable the display of the C Simulation Dialog box.



TIP: You can re-enable the display of the C Simulation Dialog box by selecting **Project → Project Settings** and selecting the **Simulation** settings.

After clicking **OK** in the dialog box, the C code is compiled and the C simulation is run. As the simulation runs, the console displays any `printf` statements from the test bench, or `hls::print` statements from the kernel or IP. When the simulation completes successfully, the following message is also returned to the console:

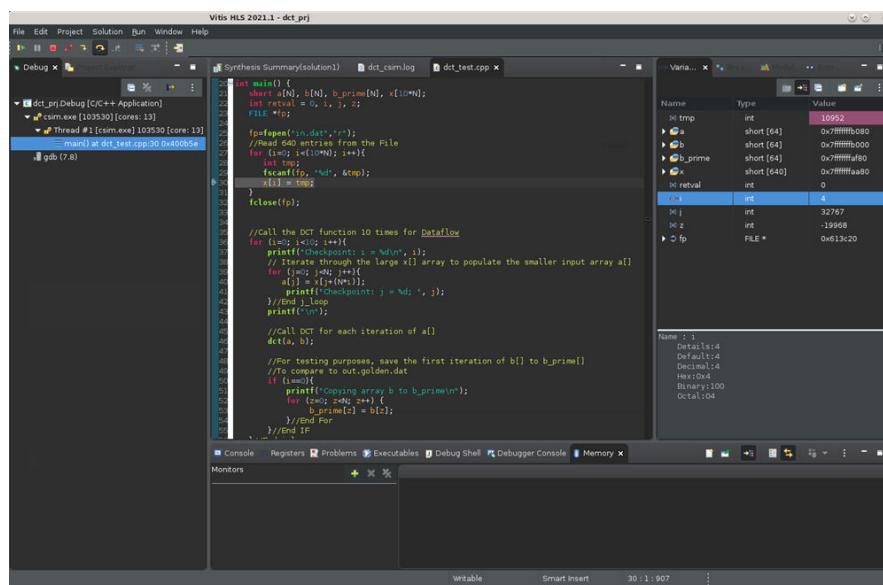
```
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] **** CSIM finish ****
Finished C simulation.
```

When the simulation fails, an error is returned:

```
@E Simulation failed: Function 'main' returns nonzero value '1'.
ERROR: [SIM 211-100] 'csim_design' failed: nonzero return value.
INFO: [SIM 211-3] **** CSIM finish ****
```

If you select the **Launch Debugger** option, the tool automatically switches to the Debug layout view as shown in the following figure. The simulation is started, but lets you step through the code to observe and debug the function. This is a full featured debug environment: you can step into and over code, specify breakpoints, and observe and set the value of variables in the code.

Figure 97: C Debug Environment





TIP: You can return to the *Synthesis* layout view by selecting the **Window**→**Synthesis**.

hls::print Function

The `hls::print` function is similar to `printf` in C, because it prints a format string and a single optional int or double argument to standard output, and to the simulation log in C simulation, RTL co-simulation and HW emulation in Vitis™.

However, it is limited to printing at most one argument, with a restricted set of datatypes, as mentioned below. It may also change the initiation interval and latency of a pipeline, so it must be used very sparingly.

`hls::print` Function Uses:

- Trace the values of some selected variables.
- Trace the order in which code blocks are executed across complex control and concurrent execution (for example in dataflow). It cannot be used to trace the order in which individual statements are scheduled within a basic block of code, because the scheduler may significantly change that order.

When used in this simple example:

```
#include "hls_print.h"
...
    for (int i=0; i<N; i++) {
#pragma HLS pipeline ii=1
        hls::print("loop %d\n", i);
    }
```

It prints the value of "i" at each iteration of the loop in both C simulation, SW emulation, RTL co-simulation, and HW emulation (it is currently ignored when the target is a HW implementation).

Note the following:

- For now the functionality is supported only in Verilog RTL.
- The only supported format specifiers are:
 - `%d` for integer or unsigned
 - `%f` for float or double
- Values of type long and long long, and the unsigned variants, must be cast to int or unsigned int (due to the argument promotion rules of C++).
- By adding an "observation" point, insertion of `hls::print` may alter the optimizations performed by HLS. Thus it can change the behavior of the RTL (just like a `printf` in SW can alter the behavior of the binary, but much more dramatically due to the nature of HLS).

- Only a single int or double value can be passed, in order to minimize the above mentioned impact.
- The order of execution of different `hls::print` functions within a code block may change due to optimizations and scheduling.
- In RTL the current simulation time is printed out as well, in order to ease debugging.
- The amount of data that it may produce may be huge, thus it should not be used to dump large arrays.

Writing a Test Bench

When using the Vitis HLS design flow, it is time consuming to synthesize an improperly coded C function and then analyze the implementation details to determine why the function does not perform as expected. Therefore, the first step in high-level synthesis should be to validate that the C function is correct, before generating RTL code, by performing simulation using a well written test bench. Writing a good test bench can greatly increase your productivity, as C functions execute in orders of magnitude faster than RTL simulations. Using C to develop and validate the algorithm before synthesis is much faster than developing and debugging RTL code.

Vitis HLS uses the test bench to compile and execute the C simulation. During the compilation process, you can select the **Launch Debugger** option to open a full C-debug environment, which enables you to more closely analyze the C simulation. Vitis HLS also uses the test bench to verify the RTL output of synthesis as described in [C/RTL Co-Simulation in Vitis HLS](#).

The test bench includes the `main()` function, as well as any needed sub-functions that are not in the hierarchy of the top-level function designated for synthesis by Vitis HLS. The main function verifies that the top-level function for synthesis is correct by providing stimuli and calling the function for synthesis, and by consuming and validating its output.



IMPORTANT! The test bench can accept input arguments that can be provided when C simulation is launched, as described in [Verifying Code with C Simulation](#). However, the test bench must not require interactive user inputs during execution. The Vitis HLS GUI does not have a command console, and therefore cannot accept user inputs while the test bench executes.

The following code shows the important features of a self-checking test bench, as an example:

```
int main () {
    //Establish an initial return value. 0 = success
    int ret=0;

    // Call any preliminary functions required to prepare input for the test.

    // Call the top-level function multiple times, passing input stimuli as
    // needed.
    for(i=0; i<NUM_TRANS; i++){
        top_func(input, output);
    }
```

```
// Capture the output results of the function, write to a file
// Compare the results of the function against expected results
ret = system("diff --brief -w output.dat output.golden.dat");

if (ret != 0) {
    printf("Test failed !!!\n");
    ret=1;
} else {
    printf("Test passed !\n");
}

return ret;
}
```

The test bench should execute the top-level function for multiple transactions, allowing many different data values to be applied and verified. The test bench is only as good as the variety of tests it performs. In addition, your test bench must provide multiple transactions if you want to calculate II during RTL simulation as described in [C/RTL Co-Simulation in Vitis HLS](#).

This self-checking test bench compares the results of the function, `output.dat`, against known good results in `output.golden.dat`. This is just one example of a self-checking test bench. There are many ways to validate your top-level function, and you must code your test bench as appropriate to your code.

In the Vitis HLS design flow, the return value of function `main()` indicates the following:

- Zero: Results are correct.
- Non-zero value: Results are incorrect.

The test bench can return any non-zero value. A complex test bench can return different values depending on the type of failure. If the test bench returns a non-zero value after C simulation or C/RTL co-simulation, Vitis HLS reports an error and simulation fails.



TIP: Because the system environment (for example, Linux, Windows, or Tcl) interprets the return value of the `main()` function, it is recommended that you constrain the return value to an 8-bit range for portability and safety.

Of course, the results of simulation are only as good as the test bench you provide. You are responsible for ensuring that the test bench returns the correct result. If the test bench returns zero, Vitis HLS indicates that the simulation has passed, regardless of what occurred during simulation.

Example Test Bench

Xilinx recommends that you separate the top-level function for synthesis from the test bench, and that you use header files. The following code example shows a design in which the top-level function for the HLS project, `hier_func`, calls two sub-functions:

- `sumsub_func` performs addition and subtraction.
- `shift_func` performs shift.

The data types are defined in the header file (`hier_func.h`). The code for the function follows:

```
#include "hier_func.h"

int sumsub_func(dint_t *in1, dint_t *in2, dint_t *outSum, dint_t *outSub)
{
    *outSum = *in1 + *in2;
    *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
    *outA = *in1 >> 1;
    *outB = *in2 >> 2;
}

void hier_func(dint_t A, dint_t B, dout_t *C, dout_t *D)
{
    dint_t apb, amb;

    sumsub_func(&A, &B, &apb, &amb);
    shift_func(&apb, &amb, C, D);
}
```

As shown, the top-level function can contain multiple sub-functions. There can only be one top-level function for synthesis. To synthesize multiple functions, group them as sub-functions of a single top-level function.

The header file (`hier_func.h`), shown below, demonstrates how to use macros and how `typedef` statements can make the code more portable and readable.



TIP: [Arbitrary Precision \(AP\) Data Types](#) discusses arbitrary precision data types, and how the `typedef` statement allows the types and therefore the bit-widths of the variables to be refined for both area and performance improvements in the final FPGA implementation.

```
#ifndef _HIER_FUNC_H_
#define _HIER_FUNC_H_

#include <stdio.h>

#define NUM_TRANS 40

typedef int dint_t;
typedef int dint_t;
typedef int dout_t;

void hier_func(dint_t A, dint_t B, dout_t *C, dout_t *D);

#endif
```

The header file above includes some `#define` statements, such as `NUM_TRANS`, that are not required by the `hier_func` function, but are provided for the test bench, which also includes the same header file.

The following code defines a test bench for the `hier_func` design:

```
#include "hier_func.h"

int main() {
    // Data storage
    int a[NUM_TRANS], b[NUM_TRANS];
    int c_expected[NUM_TRANS], d_expected[NUM_TRANS];
    int c[NUM_TRANS], d[NUM_TRANS];

    //Function data (to/from function)
    int a_actual, b_actual;
    int c_actual, d_actual;

    // Misc
    int retval=0, i, i_trans, tmp;
    FILE *fp;

    // Load input data from files
    fp=fopen(tb_data/inA.dat,r);
    for (i=0; i<NUM_TRANS; i++){
        fscanf(fp, %d, &tmp);
        a[i] = tmp;
    }
    fclose(fp);

    fp=fopen(tb_data/inB.dat,r);
    for (i=0; i<NUM_TRANS; i++){
        fscanf(fp, %d, &tmp);
        b[i] = tmp;
    }
    fclose(fp);

    // Execute the function multiple times (multiple transactions)
    for(i_trans=0; i_trans<NUM_TRANS-1; i_trans++){

        //Apply next data values
        a_actual = a[i_trans];
        b_actual = b[i_trans];

        hier_func(a_actual, b_actual, &c_actual, &d_actual);

        //Store outputs
        c[i_trans] = c_actual;
        d[i_trans] = d_actual;
    }

    // Load expected output data from files
    fp=fopen(tb_data/outC.golden.dat,r);
    for (i=0; i<NUM_TRANS; i++){
        fscanf(fp, %d, &tmp);
        c_expected[i] = tmp;
    }
    fclose(fp);

    fp=fopen(tb_data/outD.golden.dat,r);
    for (i=0; i<NUM_TRANS; i++){
        fscanf(fp, %d, &tmp);
        d_expected[i] = tmp;
    }
    fclose(fp);

    // Check outputs against expected
}
```

```
for (i = 0; i < NUM_TRANS-1; ++i) {  
    if(c[i] != c_expected[i]) {  
        retval = 1;  
    }  
    if(d[i] != d_expected[i]) {  
        retval = 1;  
    }  
}  
  
// Print Results  
if(retval == 0){  
    printf("**** *** *** ***\n");  
    printf("Results are good\n");  
    printf("**** *** *** ***\n");  
} else {  
    printf("**** *** *** ***\n");  
    printf("Mismatch: retval=%d\n", retval);  
    printf("**** *** *** ***\n");  
}  
  
// Return 0 if outputs are corre  
return retval;  
}
```

Design Files and Test Bench Files

Because Vitis HLS reuses the C test bench for RTL verification, it requires that the test bench and any associated files be denoted as test bench files when they are added to the Vitis HLS project. Files associated with the test bench are any files that are:

- Accessed by the test bench.
- Required for the test bench to operate correctly.

Examples of such files include the data files `inA.dat` and `inB.dat` in the example test bench. You must add these to the Vitis HLS project as test bench files.

The requirement for identifying test bench files in a Vitis HLS project does not require that the design and test bench be in separate files (although separate files are recommended). To demonstrate this, a new example is defined from the same code used in [Example Test Bench](#), except a new top-level function is defined. In this example the function `sumsub_func` is defined as the top-level function in the Vitis HLS project.



TIP: You can change the top-level function by selecting the [Project Settings](#) command from the Flow Navigator, selecting the [Synthesis](#) settings, and specifying a new [Top Function](#).

With the `sumsub_func` function defined as the top-level function, the higher-level function, `hier_func` becomes part of the test bench, as it is the calling function for `sumsub_func`. The peer-level `shift_func` function is also now part of the test bench, as it is a required part of the test. Even though these functions are in the same code file as the top-level `sumsub_func` function, they are part of the test bench.

Single File Test Bench and Design

You can also include the design and test bench into a single design file. The following example has the same `hier_func` function as discussed [Example Test Bench](#), except that everything is coded in a single file: top-level function, sub functions, and main function for the test bench.

 **IMPORTANT!** Having both the test bench and design in a single file requires you to add that file to the Vitis HLS project as both a design file, and a test bench file.

```
#include <stdio.h>

#define NUM_TRANS 40

typedef int din_t;
typedef int dint_t;
typedef int dout_t;

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
    *outSum = *in1 + *in2;
    *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
    *outA = *in1 >> 1;
    *outB = *in2 >> 2;
}

void hier_func(din_t A, din_t B, dout_t *C, dout_t *D)
{
    dint_t apb, amb;

    sumsub_func(&A, &B, &apb, &amb);
    shift_func(&apb, &amb, C, D);
}

int main() {
    // Data storage
    int a[NUM_TRANS], b[NUM_TRANS];
    int c_expected[NUM_TRANS], d_expected[NUM_TRANS];
    int c[NUM_TRANS], d[NUM_TRANS];

    //Function data (to/from function)
    int a_actual, b_actual;
    int c_actual, d_actual;

    // Misc
    int retval=0, i, i_trans, tmp;
    FILE *fp;
    // Load input data from files
    fp=fopen(tb_data/inA.dat,r);
    for (i=0; i<NUM_TRANS; i++){
        fscanf(fp, %d, &tmp);
        a[i] = tmp;
    }
    fclose(fp);

    fp=fopen(tb_data/inB.dat,r);
    for (i=0; i<NUM_TRANS; i++) {
```

```

fscanf(fp, %d, &tmp);
b[i] = tmp;
}
fclose(fp);

// Execute the function multiple times (multiple transactions)
for(i_trans=0; i_trans<NUM_TRANS-1; i_trans++){

    //Apply next data values
    a_actual = a[i_trans];
    b_actual = b[i_trans];

    hier_func(a_actual, b_actual, &c_actual, &d_actual);

    //Store outputs
    c[i_trans] = c_actual;
    d[i_trans] = d_actual;
}

// Load expected output data from files
fp=fopen(tb_data/outC.golden.dat,r);
for (i=0; i<NUM_TRANS; i++){
fscanf(fp, %d, &tmp);
c_expected[i] = tmp;
}
fclose(fp);

fp=fopen(tb_data/outD.golden.dat,r);
for (i=0; i<NUM_TRANS; i++){
fscanf(fp, %d, &tmp);
d_expected[i] = tmp;
}
fclose(fp);

// Check outputs against expected
for (i = 0; i < NUM_TRANS-1; ++i) {
if(c[i] != c_expected[i]){
retval = 1;
}
if(d[i] != d_expected[i]){
retval = 1;
}
}

// Print Results
if(retval == 0){
printf(    *** *** *** *** \n);
printf(        Results are good \n);
printf(    *** *** *** *** \n);
} else {
printf(    *** *** *** *** \n);
printf(        Mismatch: retval=%d \n, retval);
printf(    *** *** *** *** \n);
}

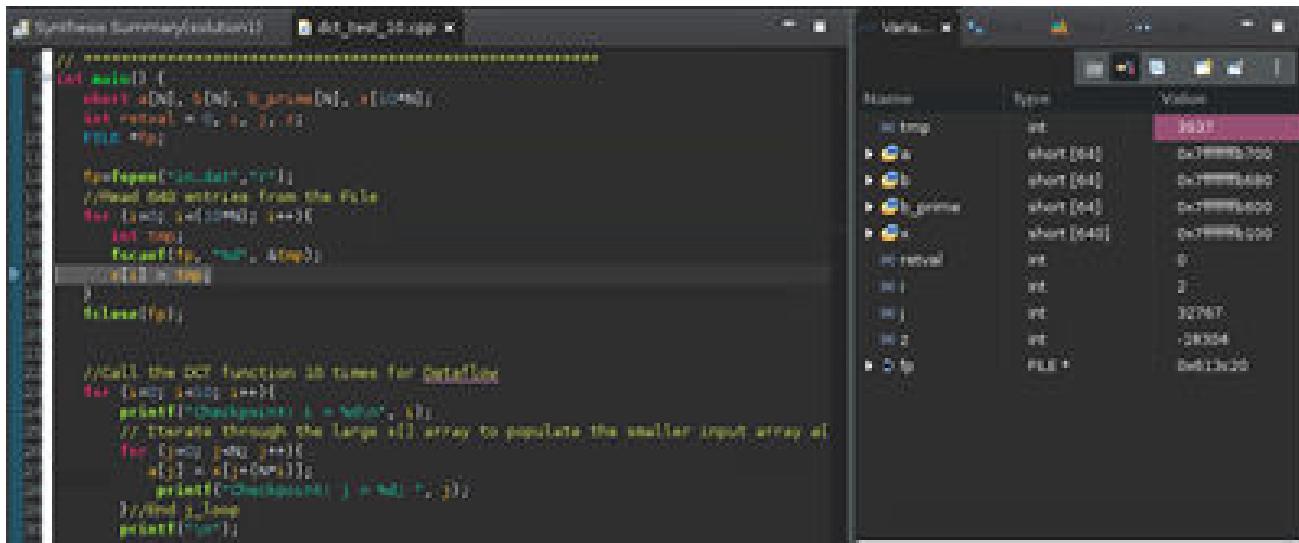
// Return 0 if outputs are correct
return retval;
}

```

Using the Debug View Layout

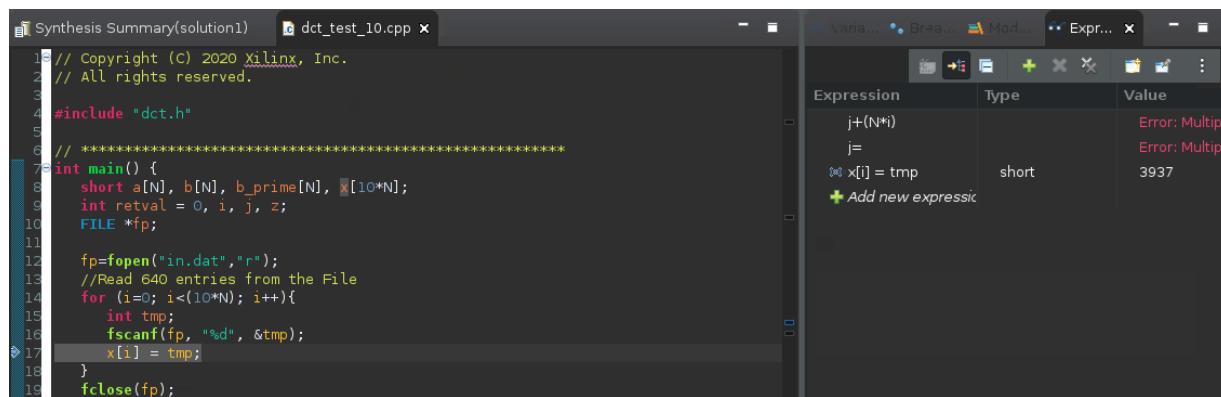
You can view the values of variables and expressions directly in the Debug view layout. The following figure shows how you can monitor the value of individual variables. In the Variables view, you can edit the values of variables to force the variable to a specific state for instance.

Figure 98: Monitoring Variables



You can monitor the value of expressions using the Expressions tab.

Figure 99: Monitoring Expressions



Output of C Simulation

When C simulation completes, a `csim` folder is created inside the solution folder. This folder contains the following elements:

- `csim/build`: The primary location for all files related to the C simulation
 - Any files read by the test bench are copied to this folder.
 - The C executable file `csim.exe` is created and run in this folder.
 - Any files written by the test bench are created in this folder.
 - `csim/obj`: Contains object files (`.o`) for the compiled source code, and make dependency files (`.d`) for the source code build.
- `csim/report`: Contains a log file of the C simulation build and run.

Pre-Synthesis Control Flow

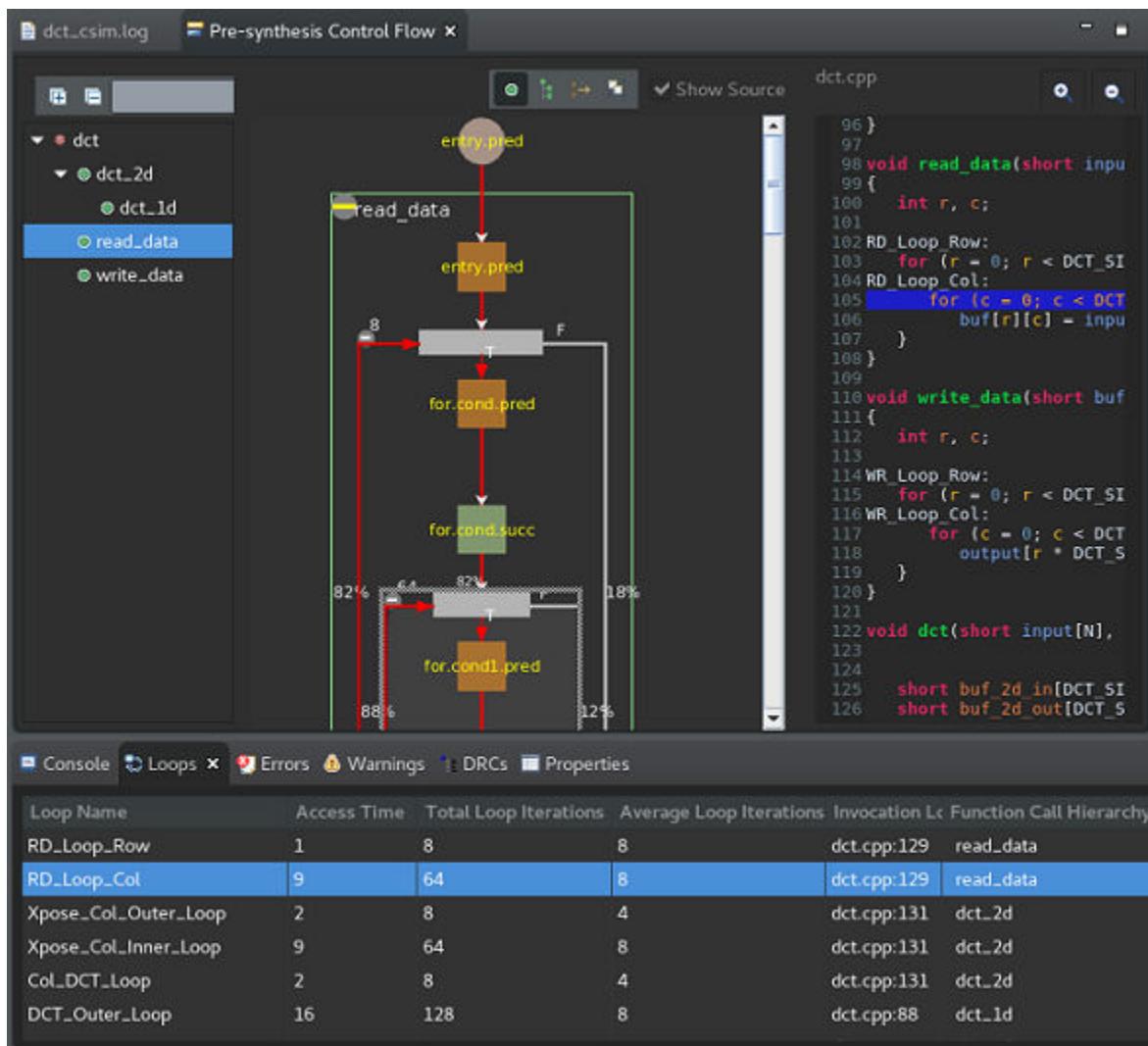


IMPORTANT! This feature is only available on Linux platforms, and is not supported on Windows systems.

You can generate the Pre-Synthesis Control Flow Graph (CFG) as an option from the Run C Simulation dialog box. Select the **Enable Pre-Synthesis Control Flow Viewer** check box on the dialog box to generate the report. After generating the report you can open it by selecting it from the **C Simulation → Reports & Viewers** section of the Flow Navigator.

The Pre-Synthesis Control Flow viewer helps you to identify the hot spots in your function, the compute-intensive control structures, and to apply pragmas or directives to improve or optimize the results. The CFG shows the control flow through your C code, as shown in the following figure, to help you visualize the top-level function. The CFG also provides static profiling, such as the trip-count of loops, and dynamic profiling information to analyze the design.

Figure 100: Pre-Synthesis Control Flow Viewer



As shown in the figure above, the Pre-Synthesis Control Flow viewer has multiple elements:

- Function Call Tree on the upper left.
- Control Flow Graph (CFG) in the middle.
- Source Code viewer on the upper right.
- Loops view in the lower Console area that is associated with, and provides cross-probing with the CFG viewer.

Selecting a sub-function or loop in one view, also selects it in other views. This lets you quickly navigate the hierarchy of your code. Every function call can be further expanded to see the control structure of the loops and condition statements. Click on the control structure to view the source code.

Double-clicking on a function in the Function Tree opens that level of the code hierarchy in the CFG, single clicking in the Function Tree simply selects the function. The CFG can be expanded to show the hierarchy of the design using the **Expand function call** command display the function levels of hierarchy specified.

You can also type in the Search field of the Function Call Tree to highlight the first matching occurrence of the typed text. You can use this to quickly navigate through your code.

The CFG can help you analyze the dynamic behavior of your function, reporting the number of times the function or sub-function executed on different control paths. Loops are often a source of computing intensity, and the Loops window provides statistics such as access time, total and average loop iterations (tripcount). This information regarding the respective loops can be found in the Loops view, which has cross-linking capabilities. Clicking on a loop will highlight both the source code and the control structure.

Memory operations can also be annotated in the CFG viewer, identifying another area for possible performance optimization.

Synthesizing the Code

To synthesize the active solution of the project, select the **Run C Synthesis** command in the Flow Navigator, or select the  command on the toolbar menu.

Note: When your project has multiple solutions as described in [Creating Additional Solutions](#), you can Run C Synthesis on the active solution, all solutions, or selected solutions using the **Solution**→**Run C Synthesis** from the main menu.

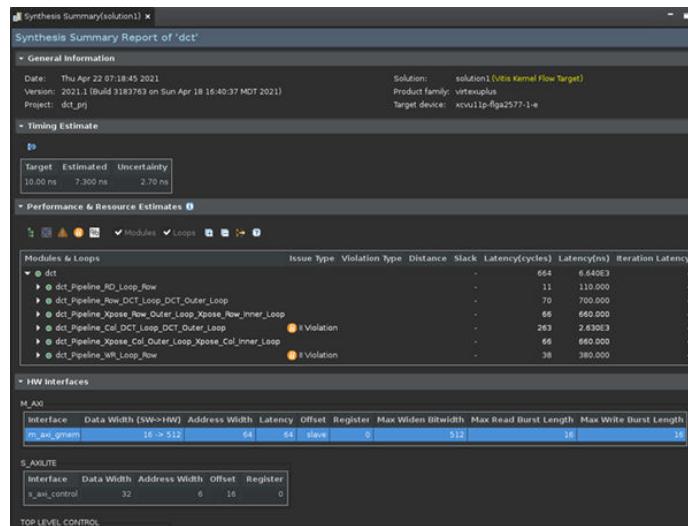
The C/C++ source code is synthesized into an RTL implementation. During the synthesis process messages are transcribed to the console window, and to the `vitis_hls.log` file.

```
INFO: [HLS 200-1470] Pipelining result : Target II = 1, Final II = 4, Depth = 6.
INFO: [SCHED 204-11] Finished scheduling.
INFO: [HLS 200-111] Elapsed time: 19.38 seconds; current allocated memory: 397.747 MB.
INFO: [BIND 205-100] Starting micro-architecture generation ...
INFO: [BIND 205-101] Performing variable lifetime analysis.
INFO: [BIND 205-101] Exploring resource sharing.
INFO: [BIND 205-101] Binding ...
INFO: [BIND 205-100] Finished micro-architecture generation.
INFO: [HLS 200-111] Elapsed time: 0.57 seconds; current allocated memory: 400.218 MB.
INFO: [HLS 200-10]
-----
INFO: [HLS 200-10] -- Generating RTL for module 'dct'
```

Within the Vitis™ HLS IDE, some messages contain links to additional information. The links are highlighted in blue underlined text, and open help messages, source code files, or documents with additional information in some cases. Clicking the messages provides more details on why the message was issued and possible resolutions.

When synthesis completes, the Simplified Synthesis report for the top-level function opens automatically in the information pane as shown in the following figure.

Figure 101: Synthesis Summary Report



You can quickly review the performance metrics displayed in the Simplified Synthesis report to determine if the design meets your requirements. The synthesis report contains information on the following performance metrics:

- **Issue Type:** Shows any issues with the results.
- **Latency:** Number of clock cycles required for the function to compute all output values.
- **Initiation interval (II):** Number of clock cycles before the function can accept new input data.
- **Loop iteration latency:** Number of clock cycles it takes to complete one iteration of the loop.
- **Loop iteration interval:** Number of clock cycles before the next iteration of the loop starts to process data.
- **Loop latency:** Number of cycles to execute all iterations of the loop.
- **Resource Utilization:** Amount of hardware resources required to implement the design based on the resources available in the FPGA, including look-up tables (LUT), registers, block RAMs, and DSP blocks.

If you specified the Run C Synthesis command on multiple solutions, the Console view reports the synthesis transcript for each of the solutions as they are synthesized. After synthesis has completed, instead of the Simplified Synthesis report, Vitis HLS displays a Report Comparison to compare the synthesis results for all of the synthesized solutions. A portion of this report is shown below.

Figure 102: Report Comparison

The screenshot shows the 'Vitis HLS Report Comparison' interface. It displays performance estimates for three solutions: solution1, solution2, and solution3. The interface includes sections for All Compared Solutions, Performance Estimates (Timing and Latency), and Utilization Estimates.

All Compared Solutions:

- solution1: xcu200-fsgd2104-2-e
- solution2: xcu200-fsgd2104-2-e
- solution3: xcu200-fsgd2104-2-e

Performance Estimates:

Timing

Clock		solution1	solution2	solution3
ap_clk	Target	8.00 ns	5.00 ns	10.00 ns
	Estimated	5.840 ns	3.650 ns	7.300 ns

Latency

		solution1	solution2	solution3
Latency (cycles)	min	648	648	648
	max	648	648	648
Latency (absolute)	min	5.184 us	3.240 us	6.480 us
	max	5.184 us	3.240 us	6.480 us
Interval (cycles)	min	649	649	649
	max	649	649	649

Utilization Estimates:

	solution1	solution2	solution3
BRAM_18K	34	34	34
DSP	16	16	16

Synthesis Summary

When synthesis completes, Vitis HLS generates a Synthesis Summary report for the top-level function that opens automatically in the information pane.

The specific sections of the Synthesis Summary are detailed below.

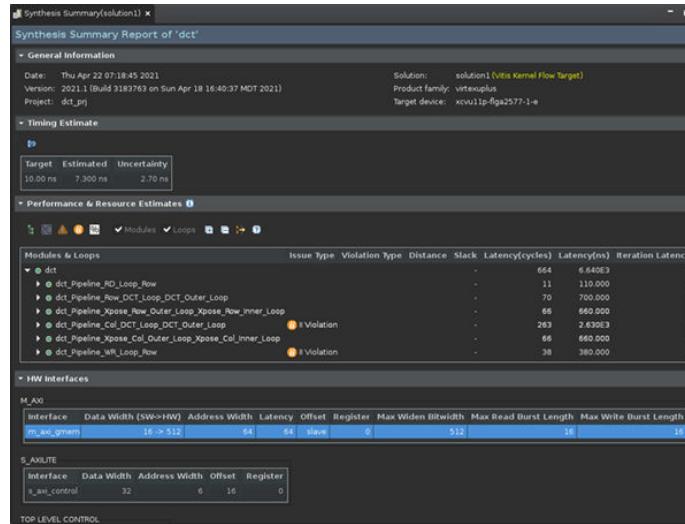


TIP: Clicking the header line for any of the sections causes the branch to collapse or expand in the report window.

General Information

Provides information on when the report was generated, the version of the software used, the project name, the solution name and target flow, and the technology details.

Figure 103: Synthesis Summary Report



Timing Estimate

Displays a quick estimate of the timing specified by the solution, as explained in [Specifying the Clock Frequency](#). This includes the Target clock period specified, and the period of Uncertainty. The clock period minus the uncertainty results in the Estimated clock period.



TIP: These values are only estimates provided by the user in the solution settings. More accurate estimates can be reported by selecting the **Run RTL Synthesis** command or **Run RTL Place and Route** from the Flow Navigator, as explained in [Exporting the RTL Design](#).

Performance & Resource Estimates

The Performance Estimate columns report the latency and initiation interval for the top-level function and any sub-blocks instantiated in the top-level. Each sub-function called at this level in the C/C++ source is an instance in the generated RTL block, unless the sub-function was in-lined into the top-level function using the `INLINE` pragma or directive, or automatically in-lined.

The Slack column displays any timing issues in the implementation.

The Latency column displays the number of cycles it takes to produce the output, and is also displayed in time (ns). The Initiation Interval is the number of clock cycles before new inputs can be applied. In the absence of any PIPELINE directives, the latency is one cycle less than the initiation interval (the next input is read after the final output is written).



TIP: When latency is displayed as a "?" it means that Vitis HLS cannot determine the number of loop iterations. If the latency or throughput of the design is dependent on a loop with a variable index, Vitis HLS reports the latency of the loop as being unknown. In this case, use the `LOOP_TRIPCOUNT` pragma or directive to manually specify the number of loop iterations. The `LOOP_TRIPCOUNT` value is only used to ensure the generated reports show meaningful ranges for latency and interval and does not impact the results of synthesis.

The Iteration Latency is the latency of a single iteration for a loop. The Trip Count column displays the number of iterations a specific loop makes in the implemented hardware. This reflects any unrolling of the loop in hardware.

The Resource Estimate columns of the report indicates the estimated resources needed to implement the software function in the RTL code. Estimates of the BRAM, DSP, FFs, and LUTs are provided.

HW Interfaces

The HW Interfaces section of the synthesis report provides tables for the different hardware interfaces generated during synthesis. The type of hardware interfaces generated by the tool depends on the flow target specified by the solution, as well as any INTERFACE pragmas or directives applied to the code. In the following image, the solution targets the Vitis Kernel flow, and therefore generates AXI interfaces as required.

Figure 104: HW Interfaces

HW Interfaces									
M_AXI									
Interface	Data Width (SW->HW)	Address Width	Latency	Offset	Offset Interface	Register	Bundle	Max Widen Bitwidth	Max Rea
m_axi_gmem	16 -> 512	64	64	slave	s_axi_control	0		512	
S_AXILITE									
Interface	Offset	Register	Bundle						
s_axi_control	16	0							
TOP LEVEL CONTROL									
Interface	Type	Ports							
ap_clk	clock	ap_clk							
ap_rst_n	reset	ap_rst_n							
interrupt	interrupt	interrupt							

The following should be observed when reviewing these tables:

- Separate tables are provided for the different interfaces.
- Columns are provided to display different properties of the interface. For the M_AXI interface, these include the Data Width and Max Widen Bitwidth columns which indicate whether **Automatic Port Width Resizing** has occurred, and to what extent. In the example above, you can see that the port was widened to 512 bits from the 16 bits specified in the software.
- The Latency column displays the latency of the interface:

- In an `ap_memory` interface, the column displays the read latency of the RAM resource driving the interface.
- For an `m_axi` interface, the column displays the expected latency of the AXI4 interface, allowing the design to initiate a bus request a number of cycles (latency) before the read or write is expected.
- The Bundle column displays any specified bundle names from the INTERFACE pragma or directive.
- Additional columns display burst and read and write properties of the M_AXI interface as described in [set_directive_interface](#).
- The Bit Fields column displays the bits used by an the registers in an `s_axilite` interface.

SW I/O Information

Highlights how the function arguments from the C/C++ source is associated with the port names in the generated RTL code. Additional details of the software and hardware ports are provided as shown below. Notice that the SW argument is expanded into multiple HW interfaces. For example, the `input` argument is related to three HW interfaces, the `m_axi` for data, and the `s_axilite` for required control signals.

Figure 105: SW I/O Information

SW I/O Information				
Top Function Arguments				
Argument	Direction	Datatype	Size	
input	inout	short*	16	
output	inout	short*	16	

SW-to-HW Mapping					
Argument	HW Name	HW Type	HW Usage	HW Info	
input	<code>m_axi_gmem</code> interface (out)				
input	<code>s_axi_control input_r_1</code>	register (in)	offset	offset=0x10 range=32	
input	<code>s_axi_control input_r_2</code>	register (in)	offset	offset=0x14 range=32	
output	<code>m_axi_gmem</code> interface (out)				
output	<code>s_axi_control output_r_1</code>	register (in)	offset	offset=0x1c range=32	
output	<code>s_axi_control output_r_2</code>	register (in)	offset	offset=0x20 range=32	

M_AXI Burst Information

In the M_AXI Burst Information section the Burst Summary table reports the successful burst transfers, with a link to the associated source code. The reported burst length refers to either `max_read_burst_length` or `max_write_burst_length` and represents the number of data values read/written during a burst transfer. For example, in a case where the input type is integer (32 bits), and HLS auto-widens the interface to 512 bits, each burst transfers 1024 integers. Because the widened interface can carry 16 integers at a time, the result is 64 beat bursts. The Burst Missed table reports why a particular burst transfer was missed with a link to Guidance messages related to the burst failures to help with resolution.

Figure 106: M_AXI Burst Information

M_AXI Burst Information			
Burst Summary			
HW Interface	Message	Resolution	Location
m_axi_gmem0	Multiple burst writes of variable length and bit width 32 in loop 'VITIS_LOOP_17_2'	214-232	hls.cpp:17:20
m_axi_gmem0	Multiple burst writes of variable length and bit width 32 in loop 'VITIS_LOOP_24_3'	214-232	hls.cpp:24:19

Burst Missed			
HW Interface	Variable	Problem	Resolution
m_axi_gmem0	out	Access call is in the conditional branch	214-232
m_axi_gmem0	out	Could not widen since Type i32 size is greater than or equal to alignment 0(bytes)	hls.cpp:24:19
m_axi_gmem0	out	Could not widen since Type i32 size is greater than or equal to alignment 0(bytes)	hls.cpp:17:20

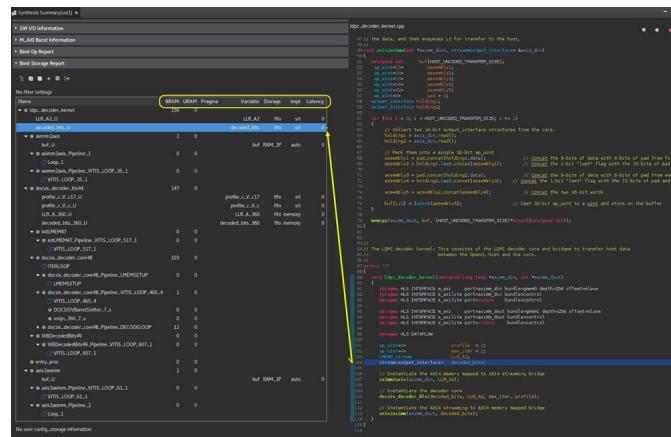
Bind Op and Bind Storage Reports

The Bind Op and Bind Storage reports are added to the Synthesis Summary report. Both reports can help you understand choices made by Vitis HLS when it maps operations to resources. The tool will map operations to the right resources with the right latency. You can influence this process by using the `BIND_OP` pragma or directive, and requesting a particular resource mapping and latency. The Bind Op report will show which of the mappings were automatically done versus those enforced by the use of a pragma. Similarly, the Bind Storage report shows the mappings of arrays to memory resources on the platform like BRAM/LUTRAM/URAM.

The Bind Op Report displays the implementation details of the kernel or IP. The hierarchy of the top-level function is displayed and variables are listed with any HLS pragmas or directives applied, the operation defined, the implementation used by the HLS tool, and any applied latency.

This report is useful for examining the programmable logic implementation details specified by the RTL design.

Figure 107: Synthesis Summary



As shown above, the Bind OP report highlights certain important characteristics in your design. Currently, it calls out the number of DSPs used in the design and shows in a hierarchy where these DSPs are used in the design. The table also highlights whether the particular resource allocation was done because of a user-specified pragma and if so, a "yes" entry will be present in the Pragma column. If no entry exists in the Pragma column, it means that the resource was auto inferred by the tool. The table also shows the RTL names of the resources allocated for each module in the user's design and you can hierarchy descend down the hierarchy to see the various resources.

It does not show all the inferred resources but instead shows resources of interest such as arithmetic, floating-point, and DSPs. The particular implementation choice of fabric (implemented using LUTs) or DSP is also shown. Finally, the latency of the resource is also shown. This is helpful in understand and increasing the latency of resources if needed to add pipeline stages to the design. This is extremely useful when attempting to break a long combinational path when trying to solve timing issues during implementation.

Each resource allocation is correlated to the source code line where the corresponding op was inferred from and the user can right-click on the resource and select the "Goto Source" option to see this correlation. Finally, the second table below the Bind Op report illustrates any global config settings that can also alter the resource allocation algorithm used by the tool. In the above example, the implementation choice for a `dadd` (double precision floating point addition) operation has been fixed to a `fulldsp` implementation. Similarly, the latency of a `ddiv` operation has been fixed to 2.

Similar to the BIND_OP pragma, the BIND_STORAGE pragma can be used to select a particular memory type (such as single port or dual port) and/or a particular memory implementation (such as BRAM/LUTRAM/URAM/SRL, etc.) and a latency value. The Bind Storage report highlights the storage mappings used in the design. Currently, it calls out the number of BRAMs and URAMs used in the design. The table also highlights whether the particular storage resource allocation was done because of a user-specified pragma and if so, a "yes" entry will be present in the Pragma column. If no entry exists in the Pragma column, then this means that the storage resource was auto inferred by the tool. The particular storage type, as well as the implementation choice, are also shown along with the variable name and latency.

Using this information, you can review the storage resource allocation in the design and make design choices by altering the eventual storage implementation depending upon availability. Finally, a second table below the Bind Storage report will be shown if there are any global config settings that can also alter the storage resource allocation algorithm used by the tool.

User Pragma Report

Displays the ignored and incorrect Pragmas in the design. This report is intended to summarize issues that can otherwise be found in the Vitis HLS log files. It lets you quickly identify issues with the pragmas used in your design, to see which ones may not have been used as expected. In addition, valid pragmas are separately reported so you can see all pragmas in use in the design.



TIP: A link to the source code where the pragma is applied is provided in the report.

Figure 108: Pragma Report

User Pragma Report				
Ignored Pragmas				
Type	Options	Location	Function	Messages
resource	variable=src core=AXI4M	convolution.cpp:211	filter11x11_orig	'Resource pragma' is deprecated, use 'bind_op/bind_storage pragma' instead '#pragma HLS RESOURCE core=axi' is obsolete and replaced by '#pragma HLS INTERFACE axi'
resource	variable=dst core=AXI4M	convolution.cpp:213	filter11x11_orig	'Resource pragma' is deprecated, use 'bind_op/bind_storage pragma' instead '#pragma HLS RESOURCE core=axi' is obsolete and replaced by '#pragma HLS INTERFACE axi'
resource	variable=return core=AXI4LiteS metadata="-bus_bundle hls_ctrl"	convolution.cpp:214	filter11x11_orig	'Resource pragma' is deprecated, use 'bind_op/bind_storage pragma' instead '#pragma HLS RESOURCE core=axi' is obsolete and replaced by '#pragma HLS INTERFACE axi'
resource	variable=width core=AXI4LiteS metadata="-bus_bundle hls_ctrl"	convolution.cpp:215	filter11x11_orig	'Resource pragma' is deprecated, use 'bind_op/bind_storage pragma' instead '#pragma HLS RESOURCE core=axi' is obsolete and replaced by '#pragma HLS INTERFACE axi'
resource	variable=height core=AXI4LiteS metadata="-bus_bundle hls_ctrl"	convolution.cpp:216	filter11x11_orig	'Resource pragma' is deprecated, use 'bind_op/bind_storage pragma' instead '#pragma HLS RESOURCE core=axi' is obsolete and replaced by '#pragma HLS INTERFACE axi'
Pragmas With Warnings				
Type	Options	Location	Function	Messages
inline	region	convolution.cpp:218	filter11x11_orig	'region' in '#pragma HLS Inline' is deprecated, use 'InlinePragma' instead
interface	axis port=&src	convolution.cpp:234	filter11x11 strm	extra token before variable expression is ignored
interface	axis port=&dst	convolution.cpp:235	filter11x11 strm	extra token before variable expression is ignored
inline	region	convolution.cpp:238	filter11x11 strm	'region' in '#pragma HLS Inline' is deprecated, use 'InlinePragma' instead
Valid Pragma Syntax				
Type	Options	Location	Function	
array_partition	variable=linebuf dim=1 complete	convolution.cpp:130	convolution strm	
inline		convolution.cpp:131	convolution strm	
pipeline		convolution.cpp:141	convolution strm	
dependence	variable=linebuf inter false	convolution.cpp:158	convolution strm	
pipeline		convolution.cpp:159	convolution strm	
pipeline		convolution.cpp:178	convolution strm	
interface	port=src m_axi depth=32400	convolution.cpp:210	filter11x11_orig	
interface	port=dst m_axi depth=32400	convolution.cpp:212	filter11x11_orig	
dataflow		convolution.cpp:219	filter11x11_orig	
dataflow		convolution.cpp:237	filter11x11 strm	

Output of C Synthesis

When synthesis completes, the `syn` folder is created inside the solution folder. This folder contains the following elements:

- The `verilog` and `vhdl` folders contain the output RTL files.
 - The top-level file has the same name as the top-level function for synthesis.
 - There is one RTL file created for each sub-function that has not been inlined into a higher level function.
 - There could be additional RTL files to implement sub-blocks of the RTL hierarchy, such as block RAM, and pipelined multipliers.
- The `report` folder contains a report file for the top-level function and one for every sub-function that has not been in-lined into a higher level function by Vitis HLS. The report for the top-level function provides details on the entire design.



IMPORTANT! You should not use the RTL files generated in the `syn/verilog` or `syn/vhdl` folder for synthesis in the Vivado tool. You should instead use the packaged output files for use with the Vitis application acceleration development flow, or the Vivado Design Suite as described in [Exporting the RTL Design](#). In cases where Vitis HLS uses Xilinx IP in the generated RTL code, such as with floating point designs, the `verilog` and `vhdl` folders contain a script to create that IP during RTL synthesis by the Xilinx tools. If you use the files in the `syn/verilog` or `syn/vhdl` folder directly for RTL synthesis, you must also correctly use any script files present in those folders. If the packaged output is used, this process is performed automatically by the Xilinx tools.

Improving Synthesis Runtime and Capacity

Vitis HLS schedules operations hierarchically. The operations within a loop are scheduled, then the loop, the sub-functions and operations with a function are scheduled. Runtime for Vitis HLS increases when:

- There are more objects to schedule.
- There is more freedom and more possibilities to explore.

Vitis HLS schedules objects. Whether the object is a floating-point multiply operation or a single register, it is still an object to be scheduled. The floating-point multiply may take multiple cycles to complete and use many resources to implement but at the level of scheduling it is still one object.

Unrolling loops and partitioning arrays creates more objects to schedule and potentially increases the runtime. Inlining functions creates more objects to schedule at this level of hierarchy and also increases runtime. These optimizations may be required to meet performance but be very careful about simply partitioning all arrays, unrolling all loops and inlining all functions: you can expect a runtime increase. Use the optimization strategies provided earlier and judiciously apply these optimizations.

If the loops must be unrolled, or if the use of the PIPELINE directive in the hierarchy above has automatically unrolled the loops, consider capturing the loop body as a separate function. This will capture all the logic into one function instead of creating multiple copies of the logic when the loop is unrolled: one set of objects in a defined hierarchy will be scheduled faster. Remember to pipeline this function if the unrolled loop is used in pipelined region.

The degrees of freedom in the code can also impact runtime. Consider Vitis HLS to be an expert designer who by default is given the task of finding the design with the highest throughput, lowest latency and minimum area. The more constrained Vitis HLS is, the fewer options it has to explore and the faster it will run. Consider using latency constraints over scopes within the code: loops, functions or regions. Setting a LATENCY directive with the same minimum and maximum values reduces the possible optimization searches within that scope.

Analyzing the Results of Synthesis

After synthesis completes, Vitis HLS automatically creates synthesis reports to help you understand and analyze the performance of the implementation. Examples of these reports include the [Synthesis Summary](#) report, Schedule Viewer, Function Call Graph, and Dataflow Viewer. You can view these reports from the Flow Navigator in the Vitis HLS IDE.

- **Schedule Viewer:** Shows each operation and control step of the function, and the clock cycle that it executes in.
- **Dataflow Viewer:** Shows the dataflow structure inferred by the tool, inspect the channels (FIFO/PIPO), to let you examine the effect of channel depth on performance
- **Function Call Graph Viewer:** Displays your full design after C Synthesis or C/RTL Co-simulation to show the throughput of the design in terms of latency and II.

In addition to the various graphs and viewers described above, the Vitis HLS tool provides additional views to expand on the information available for analysis of your design.

- **Module Hierarchy:** Shows the resources and latency contribution for each block in the RTL hierarchy. It also indicates any II or timing violations. In case of timing violations, the hierarchy window will also show the total negative slack observed in a specific module.
- **Performance Profile:** Shows details on the performance of the block currently selected in the Module Hierarchy view. Performance is measured in terms of latency and the initiation interval, and includes details on whether the block was pipelined or not.
- **Resource Profile:** Shows the resources used at the selected level of hierarchy, and shows the control state of the operations used.
- **Properties view:** Shows the properties of the currently selected control step or operation in the Schedule Viewer.

Schedule Viewer

The Schedule Viewer provides a detailed view of the synthesized RTL, showing each operation and control step of the function, and the clock cycle that it executes in. It helps you to identify any loop dependencies that are preventing parallelism, timing violations, and data dependencies.

The Schedule Viewer is displayed by default in the Analysis perspective. You can open it from the Module Hierarchy window by right-clicking a module and selecting **Open Schedule Viewer** from the menu.

In the Schedule Viewer,

- The left vertical axis shows the names of operations and loops that will get implemented as logic in the RTL hierarchy. The name in the brackets indicate the operation name. Operations are in topological order, implying that an operation on line n can only be driven by operations from a previous line, and will only drive an operation in a later line. Depending upon the type of violations found the Schedule Viewer shows additional information for each operation:
 - Resource limitation: displays the type of operation(read/write), type of memory used(RAM_1p or RAM_2p). In the image below the `vecIn` is a memory which is a dual port ram and trying to perform 3 reads in a single iteration. This causes an IL violation because of a resource limitation and the tool is highlighting the operation which is scheduled in the next cycle of the load operation.

Figure 109: Sum Loop Iteration



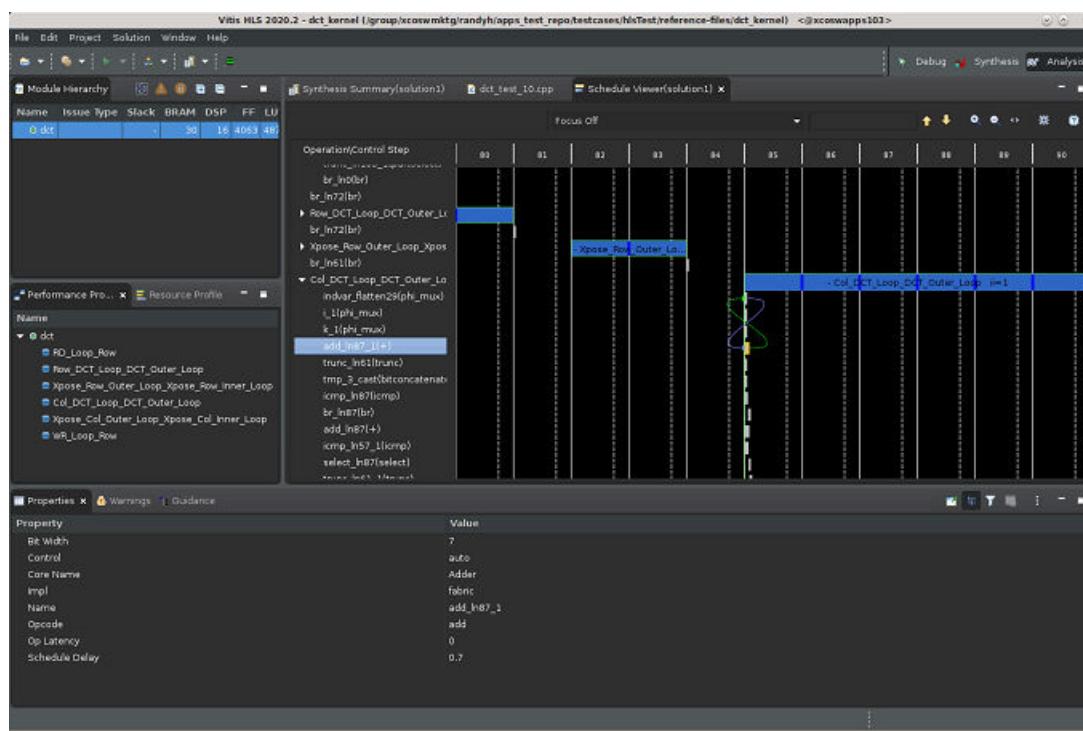
- Dependency: displays information related to iterations which have a loop carried dependency. For example, a read transaction could have a dependency on a prior write value.
- The top horizontal axis shows the clock cycles in consecutive order.
- The vertical dashed line in each clock cycle shows the reserved portion of the clock period due to clock uncertainty. This time is left by the tool for the Vivado back-end processes, like place and route.
- Each operation is shown as a gray box in the table. The box is horizontally sized according to the delay of the operation as percentage of the total clock cycle. In case of function calls, the provided cycle information is equivalent to the operation latency.
- Multi-cycle operations are shown as gray boxes with a horizontal line through the center of the box.
- The Schedule Viewer also displays general operator data dependencies as solid blue lines. As shown in the figure below, when selecting an operation you can see solid blue arrows highlighting the specific operator dependencies. This gives you the ability to perform detailed analysis of data dependencies. The green dotted line indicates an inter-iteration data dependency.
- Memory dependencies are displayed using golden lines.

- In addition, lines of source code are associated with each operation in the Schedule Viewer report. Right-click the operation to use the **Goto Source** command to open the input source code associated with the operation.

In the figure below, the loop called `RD_Loop_Row` is selected. This is a pipelined loop and the initiation interval (II) is explicitly stated in the loop bar. Any pipelined loop is visualized unfolded, meaning one full iteration is shown in the schedule viewer. Overlap, as defined by II, is marked by a thick clock boundary on the loop marker.

The total latency of a single iteration is equivalent to the number of cycles covered by the loop marker. In this case, it is three cycles.

Figure 110: Schedule Viewer

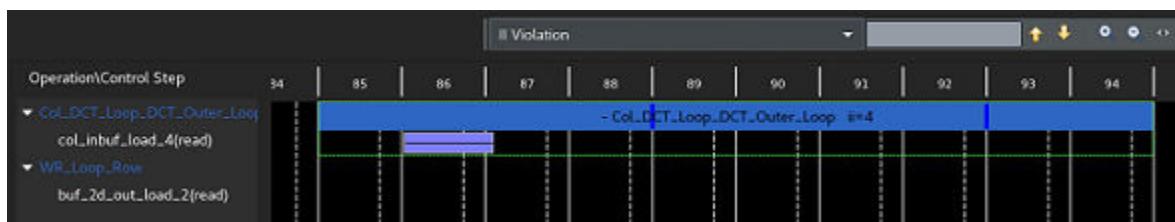


The Schedule Viewer displays a menu bar at the top right of the report that includes the following features:

- A drop-down menu, initially labeled Focus Off, that lets you specify operations or events in the report to select.
- A text search field to search for specific operations or steps () , and commands to **Scroll Up** or **Scroll Down** through the list of objects that match your search text
- Zoom In**, **Zoom Out**, and **Zoom Fit** commands ().

- The **Filter** command () lets you dynamically filter the operations that are displayed in the viewer. You can filter operations by type, or by clustered operations.
 - Filtering by type allows you to limit what operations get presented based on their functionality. For example, visualizing only adders, multipliers, and function calls will remove all of the small operations such as “and” and “or”s.
 - Filtering by clusters exploits the fact that the scheduler is able to group basic operations and then schedule them as one component. The cluster filter setting can be enabled to color the clusters or even collapse them into one large operation in the viewer. This allows a more concise view of the schedule.

Figure 111: Operation Causing Violation



You can quickly locate II violations using the drop-down menu in the Schedule Viewer, as shown in the figure above. You can also select it through the context menu in the **Module Hierarchy** view.

To locate the operations causing the violation in the source code, right-click the operation and use the **Goto Source** command, or double-click the operation and the source viewer will appear and identify the root of the object in the source.

Timing violations can also be quickly found from the **Module Hierarchy** view context menu, or by using the drop-down menu in the Schedule Viewer menu. A timing violation is a path of operations requiring more time than the available clock cycle. To visualize this, the problematic operation is represented in the Schedule Viewer in a red box.

By default all dependencies (blue lines) are shown between each operation in the critical timing path.

Properties View

At the bottom of the Schedule Viewer, as shown in the top figure, is the Properties view that displays the properties of a currently selected object in the Schedule Viewer. This lets you see details of the specific function, loop, or operation that is selected in the Schedule Viewer. The types of elements that can be selected, and the properties displayed include:

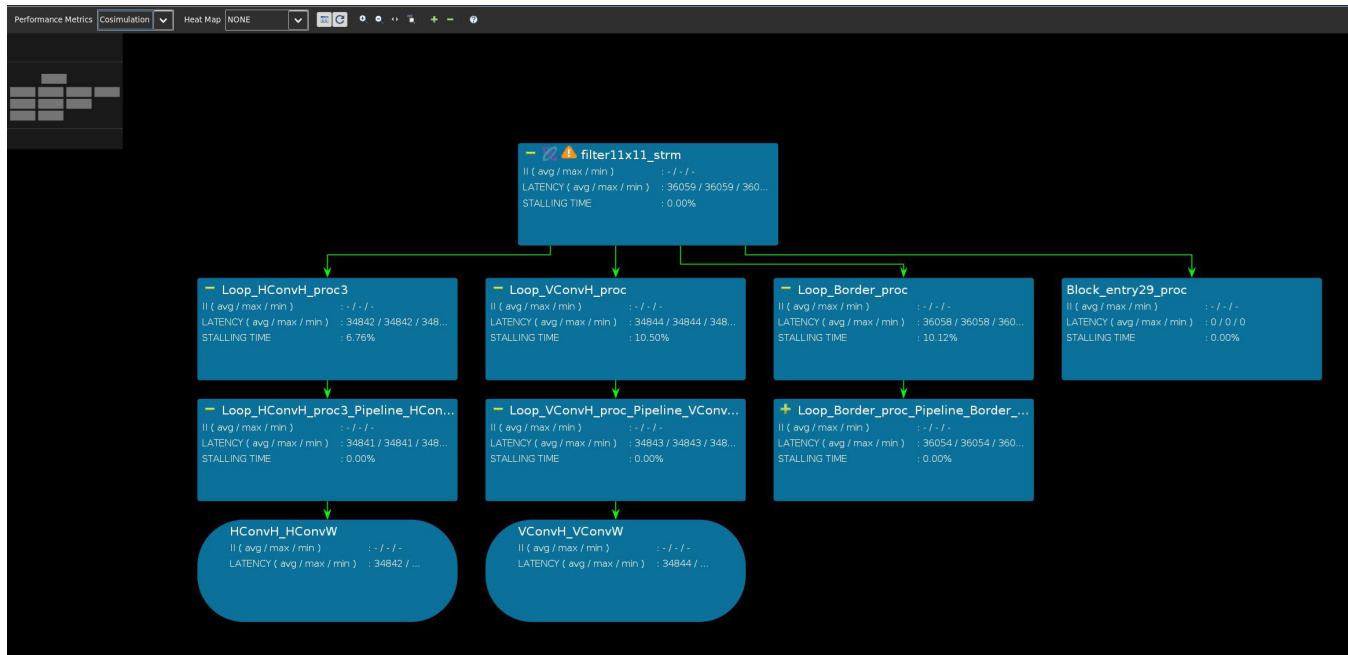
- Functions or Loops
 - **Initiation Interval (II):** The number of clock cycles before the function or loop can accept new input data.

- **Loop Iteration Latency:** The number of clock cycles it takes to complete one iteration of the loop.
- **Latency:** The number of clock cycles required for the function to compute all output values, or for the loop to complete all iterations.
- **Pipelined:** Indicates that the function or loop are pipelined in the RTL design.
- **Slack:** The timing slack for the function or loop.
- **Tripcount:** The number of iterations a loop completes.
- **Resource Utilization:** Displays the number of BRAM, DSP, LUT, or FF used to implement the function or loop.
- Operation and Storage Mapping
 - **Name:** Location which contains the code.
 - **Op Code:** Operation which has been scheduled, for example, add, sub, and mult. For more information, refer to the [BIND_OP](#) or [BIND_STORAGE](#) pragmas or directives.
 - **Op Latency:** Displays the default or specified latency for the binding of the operation or storage.
 - **Bitwidth:** Bitwidth of the Operation.
 - **Impl:** Defines the implementation used for the specified operation or storage.

Function Call Graph Viewer

The new Function Call Graph Viewer, which can be opened from the Flow Navigator, illustrates your full design after C Synthesis or C/RTL Co-simulation. The goal of this viewer is to show the throughput of the design in terms of latency and II. It helps identify the critical path in your design and helps you identify bottlenecks in the design to focus on to improve throughput. It can also show the paths through the design where throughput may be imbalanced leading to FIFO stalls and/or deadlock.

Figure 112: Performance Metrics Synthesis



In some cases, the displayed hierarchy of the design might not be the same as your source code as a result of HLS optimizations that convert loops into function pipelines, etc. Functions that are in-lined will no longer be visible in the call graph, as they are no longer separate functions in the synthesized code. If multiple instances of a function are created, each unique instance of the function is shown in the call graph. This lets you see what functions contribute to a calling function's latency and II.

The graph as shown above displays functions as rectangular boxes, and loops as oval boxes, each with II, latency, and resource or timing data depending on the specific view. Before C/RTL co-simulation is completed the performance and resource metrics that are shown in the graph are from the C Synthesis phase, and are therefore estimates from the HLS tool.

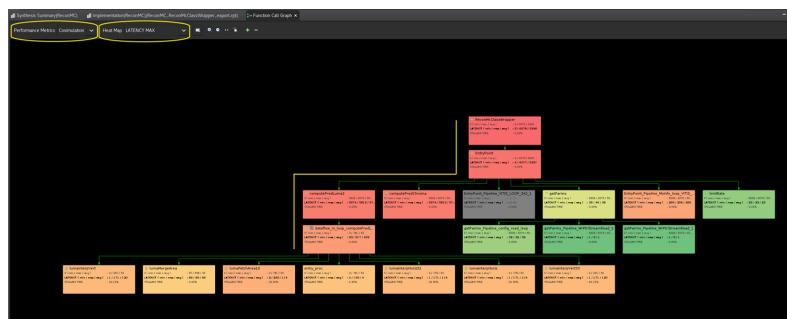
Note: For more accurate resource and timing estimates, logic synthesis or implementation can be performed as part of [Exporting the RTL Design](#).

After co-simulation, actual II and latency numbers are reported along with stalling percentages, and this information is back annotated from data collected during co-simulation. You can toggle between the synthesis performance metrics and co-simulation metrics using the drop-down menu at the upper-left of the Function Call Graph viewer.

You can also use the **Heat Map** feature to highlight several metrics of interest:

- II (min, max, avg)
- Latency (min, max, avg)
- Stalling Time Percentage

Figure 113: Performance Metrics



The heat map uses color coding to highlight problematic modules. Using a color scale of red to green where red indicates the high value of the metric (i.e. highest II or highest latency) while green indicates a low value of the metric in question. The colors that are neither red nor green represent the range of values that are in between the highest and lowest values. As shown above, this helps in quickly identifying the modules that need attention. In the example shown above, a heat map for LATENCY MAX is shown and the path of red modules indicates where the high latency values are observed.

As mentioned before, the Function Call Graph illustrates at a high level, the throughput numbers of your design. The user can view the Function Call Graph as a cockpit from which further investigations can be carried out. Right-click on any of the displayed modules to display a menu of options that you can use to display additional information. This lets you see the overall design and then jump into specific parts of the design which need extra attention. Additional reports include the Schedule Viewer, Synthesis Summary report, Dataflow Viewer, and source files. The Function Call Graph is the one viewer in Vitis HLS where you can see the full picture of your design and have the latency and II information of each module available for analysis - this includes the dataflow modules for whom the performance information can only be obtained after co-simulation.



TIP: Additional performance and resource metrics are displayed for each function/loop in the **Modules/Loops** table under the report.

Dataflow Viewer

The DATAFLOW optimization is a dynamic optimization which can only be fully understood after the RTL co-simulation is complete. Due to this fact, the Dataflow viewer lets you see the dataflow structure inferred by the tool, inspect the channels (FIFO/PIPO), and examine the effect of channel depth on performance. Performance data is back-annotated to the Dataflow viewer from the co-simulation results.



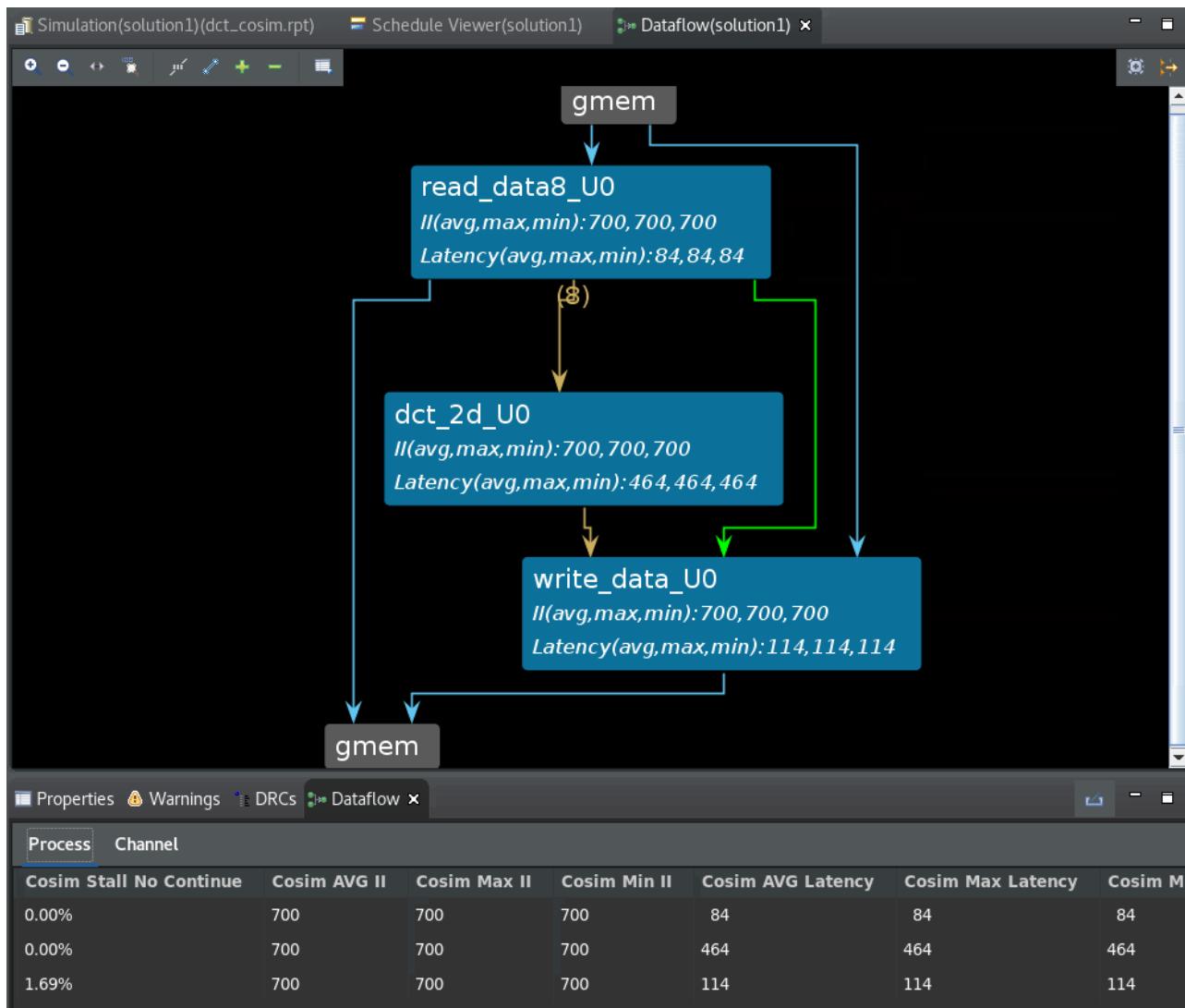
IMPORTANT! You can open the Dataflow viewer without running RTL co-simulation, but your view will not contain important performance information such as read/write block times, co-sim depth, and stall times.

You must apply the DATAFLOW pragma or directive to your design for the Dataflow viewer to be populated. You can apply dataflow to the top-level function, or specify regions of a function, or loops. The Dataflow viewer displays a representation of the dataflow graph structure, showing the different processes and the underlying producer-consumer connections.



In the Module Hierarchy view, the icon beside the function indicates that a Dataflow Viewer report is available. When you see this icon, you can right-click the function and use the **Open Dataflow Viewer** command.

Figure 114: Dataflow Viewer



Features of the Dataflow viewer include the following:

- Source Code browser.
- Automatic cross-probing from process/channel to source code.
- Filtering of ports and channel types.
- Process and Channel table details the characteristics of the design:
 - Channel Profiling (FIFO sizes etc), enabled from **Solution Settings** dialog box.
 - Process Read Blocking/Write Blocking/Stalling Time reported after RTL co-simulation.



IMPORTANT! You must use `cosim_design -enable_dataflow_profiling` to capture data for the Dataflow viewer, and your test bench must run at least two iterations of the top-level function.

- Process Latency and II displayed.
- Channel type and widths are displayed in the Channel table.
- Automatic cross-probing from Process and Channel table to the Graph and Source browser.
- Hover over channel or process to display tooltips with design information.

The Dataflow viewer can help with performance debugging your designs. When your design deadlocks during RTL co-simulation, the GUI will open the Dataflow viewer and highlight the channels and processes involved in the deadlock so you can determine if the cause is insufficient FIFO depth, for instance.

When your design does not perform as expected, the Process and Channels table can help you understand why. A process can stall waiting to read input, or can stall because it cannot write output. The channel table provides you with stalling percentages, as well as identifying if the process is "read blocked" or "write blocked."



TIP: If you use a Tcl script to create the Vitis HLS project, you can still open it in the GUI to analyze the design.

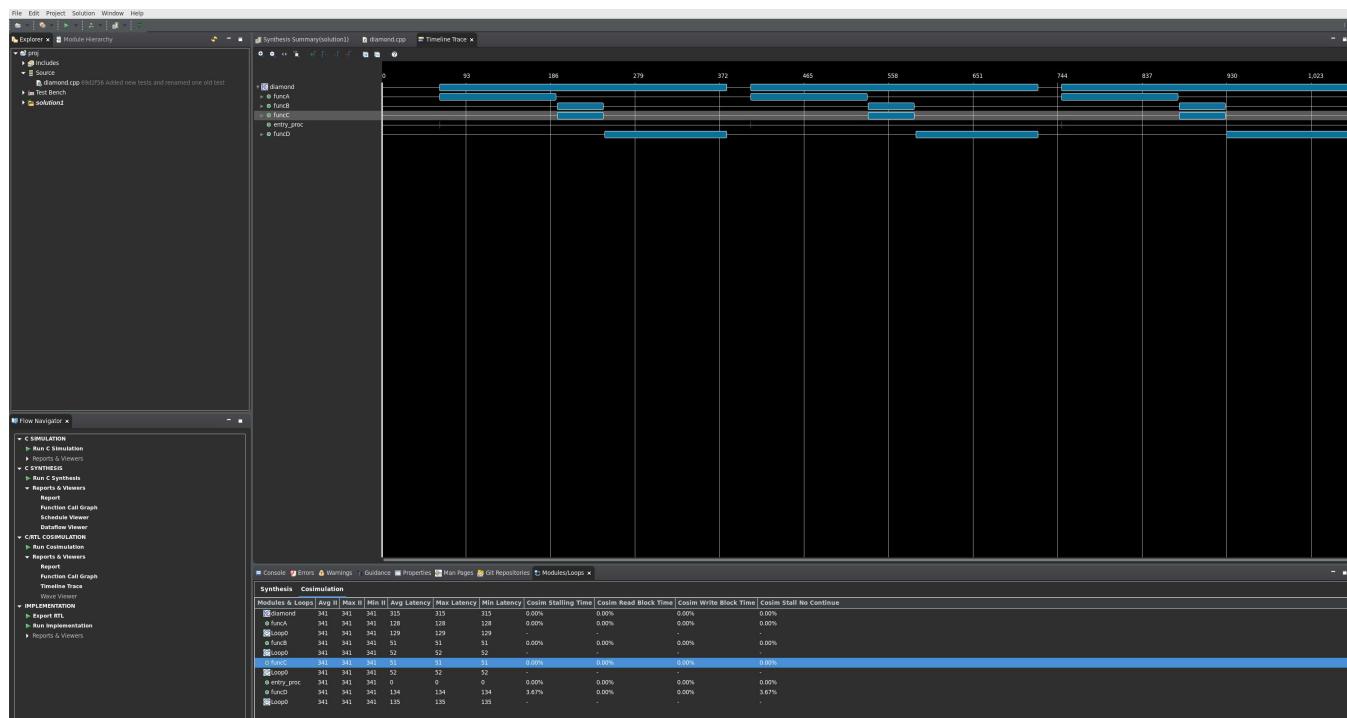
Timeline Trace Viewer

Timeline Trace viewer displays the run time profile of the functions of your design. It is especially useful to see the behavior of dataflow regions after Co-simulation, as there is no need to launch the Vivado logic simulator to view the timeline.

Timeline Trace viewer displays multiple iterations through the various sub-functions of a dataflow region, shows where the functions are starting and ending, and displays the Co-simulation data in tables below the timeline.

The viewer provides basic tools to use while viewing the timeline, such as adding markers, stepping from one marker to the next and measuring the time between markers.

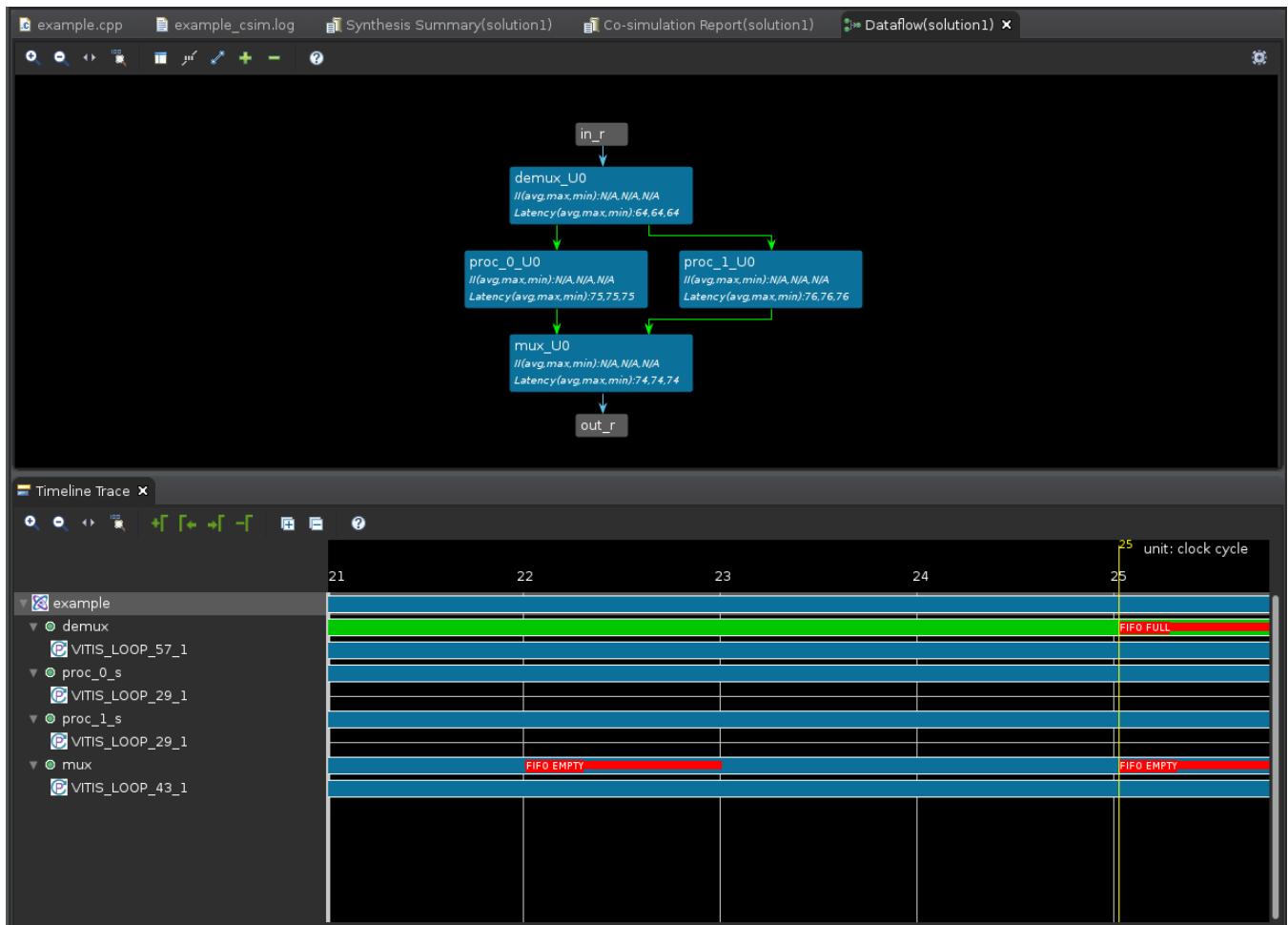
Figure 115: Timeline Trace Viewer



You can generate the Timeline Trace view from RTL Co-simulation. You should enable Dump Trace All, and Enable Channel Profiling options from the Co-Simulation dialog box, or from the Solutions Settings dialog box, and the Co-Sim window.

The Timeline Trace view also shows FIFO channel stall/starve states with Full and Empty markers. In the following figure, you can see the `demux` FIFO is full, resulting in a stall as highlighted in the timeline. In addition, the `mux` FIFO is empty and also stalled. The report also shows the loop internal II and latency, and a table at the bottom of the display to show dataflow path status, including performance, total time, stalling time and percentage.

Figure 116: Timeline Full/Empty



Optimizing the HLS Project

After analysis, you will most likely need or want to optimize the performance of your function. Even if it is performing well there may be opportunities for improvement. This section discusses the mechanisms for applying optimizations to your project. Refer to [Optimizing Techniques and Troubleshooting Tips](#) for a discussion of the various types of optimizations you can perform.

You can add optimization directives directly into the source code as compiler pragmas, using various HLS PRAGMAS, or you can use `Tcl set_directive` commands to apply optimization directives in a Tcl script to be used by a solution.

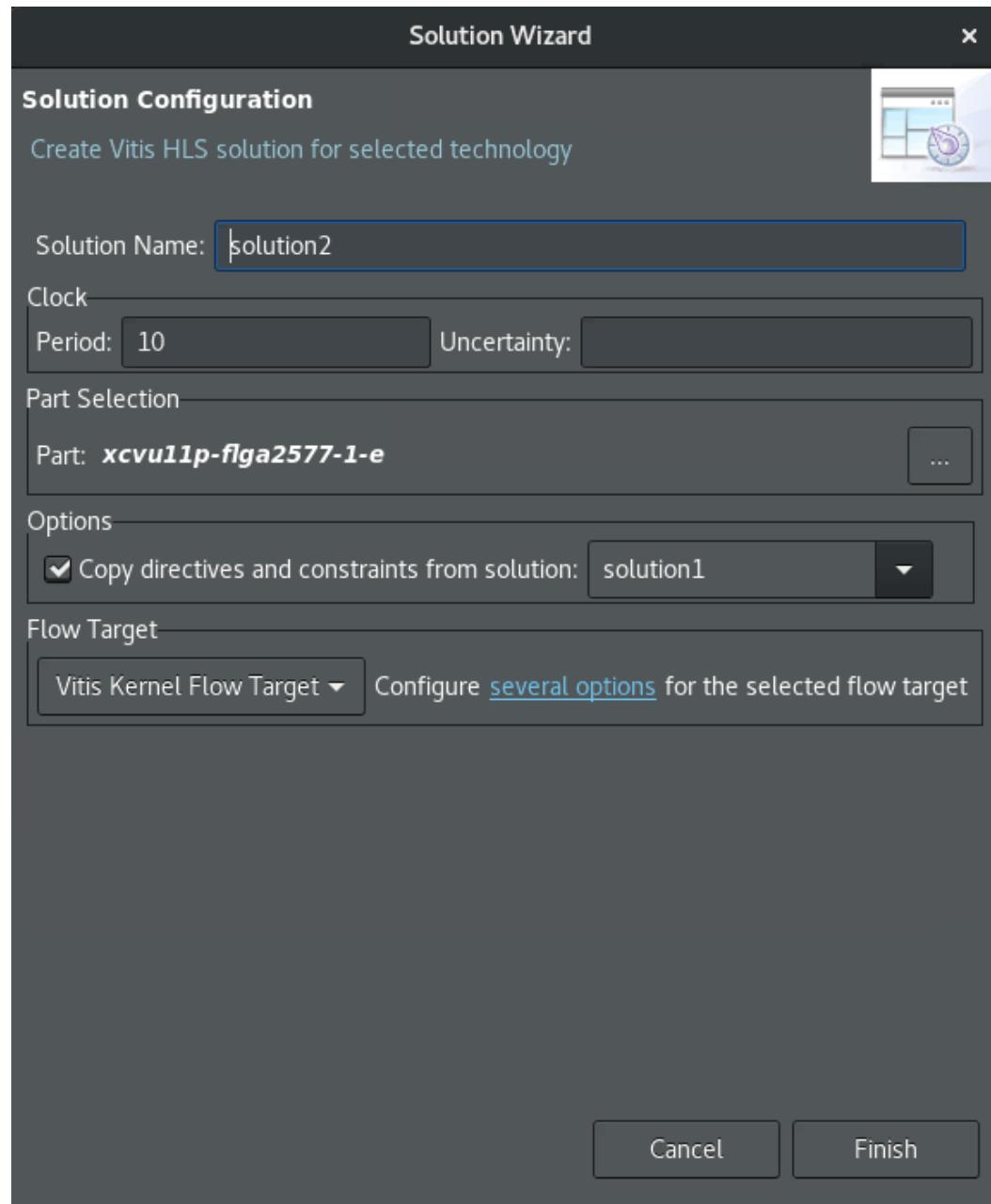
In addition to optimization pragmas and directives, Vitis™ HLS provides a number of configuration settings to let you manage the default results of simulation and synthesis. These configuration settings are accessed using the **Solution → Solution Settings...** menu command, and clicking the **Add** command to add configuration settings. Refer to [Configuration Commands](#) for more information on applying specific configuration settings.

Creating Additional Solutions

The most typical use of Vitis HLS is to create an initial design, analyze the results, and then perform optimizations to meet the desired area and performance goals. This is often an iterative process, requiring multiple steps and multiple optimizations to achieve the desired results. Solutions offer a convenient way to configure the tool, add directives to your function to improve the results, and preserve those results to compare with other solutions.

To create an additional solution for your , use the **Project → New Solution** menu command, or the **New Solution** toolbar button . This opens the Solution Wizard as shown in the following figure.

Figure 117: Solution Wizard



The Solution Wizard has the same options as described in [Creating a New Vitis HLS Project](#), with an additional option to let you **Copy directives and constraints from solution**. In the case where there are already multiple solutions, you can specify which solution to copy from. After the new solution has been created, optimization directives can be added (or modified if they were copied from the previous solution).

When your project has multiple solutions, the commands are generally directed at the current active solution. You can specify the active solution by right-clicking on a solution in the Explorer view, and use the **Set Active Solution** command. By default, synthesis and simulation commands build the active solution, directives are applied to the active solution, and reports are opened for the active solution. You want to ensure you are working in the correct solution when your project has multiple solutions.



TIP: The Explorer view shows which solution is active by applying a **bold-italic** font to the solution name.

Adding Pragmas and Directives

Vitis HLS pragmas and directives let you configure the synthesis results for your code.

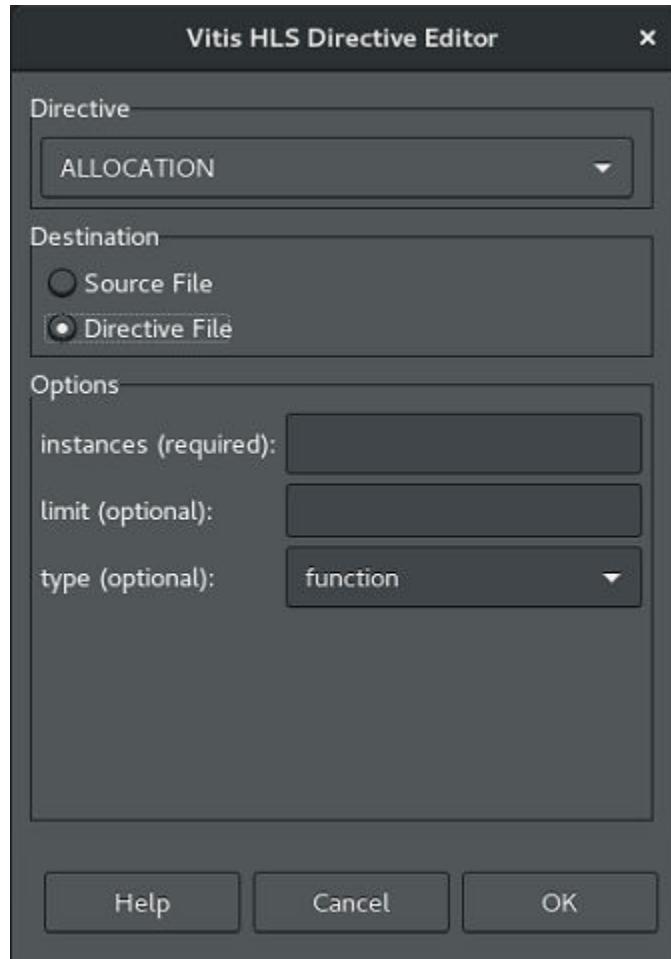
- **HLS Pragmas** are added to the source code to enable the optimization or change in the original source code. Every time the code is synthesized, it is implemented according to the specified pragmas.
- **Optimization Directives**, or the `set_directive` commands, can be specified as Tcl commands that are associated with a specific solution, or set of solutions. Allowing you to customize the synthesis results for the same code base across different solutions.



IMPORTANT! In some cases where pragmas or directives conflict with other pragmas or directives, the synthesis process returns an error until you resolve the conflict. However, in some cases the first pragma or directive takes precedence over the second pragma or directive, and the second is ignored. This information should be reported in the log file or console window.

To add pragmas or directives to your project:

1. In the Explorer view of the Vitis HLS IDE, double-click the code file under the Source folder to open the Code Editor dialog box, the Outline view, and the Directive view.
2. Use the Directive view to add pragmas to your source code. This view helps you add and manage pragmas and directives for your project, and it ensures that the pragmas are correct and applied in the proper location. To use this view:
 - a. With your source code open, select the **Directive** view tab to locate the function, loop, or feature of the code to add a pragma or directive to.
Vitis HLS applies directives to the appropriate scope for the object currently selected in the Directive view.
 - b. Right-click an object in the Directive view to use the **Insert Directive** command. The Vitis HLS Directive Editor opens, as shown in the following figure:



- c. Review the Vitis HLS Directive Editor dialog box. It includes the following sections:
- **Directive:** Specifies the directive or pragma to apply. This is a drop-down menu that lets you choose from the list of available directives.
 - **Destination:** Specifies that a pragma should be added to the source file, or that a `set_directive` command should be added to a Tcl script, the directive file, associated with the active solution.



TIP: If your project only has one solution then it is always active. However, if you have multiple solutions you will need to ensure the desired solution is active in the project. Right-click the solution in the Explorer view of the project and click the **Set Active Solution** command. Refer to [Creating Additional Solutions](#) for details on adding solutions.

- **Options:** Lists various configurable options associated with the currently selected directive.

- d. Click **OK** to apply the pragma or directive.

Note: To view information related to a selected directive, click **Help**.

Using Directives in Scripts vs. Pragmas in Code

In the Vitis HLS Directive Editor dialog box, you can specify either of the following Destination settings:

- **Directive File:** Vitis HLS inserts the directive as a Tcl command into the file `directives.tcl` in the solution directory.
- **Source File:** Vitis HLS inserts the directive directly into the C source file as a pragma.

The following table describes the advantages and disadvantages of both approaches.

Table 22: Tcl Directives Versus Pragmas

Directive Format	Advantages	Disadvantages
Directives file (Tcl Script)	Each solution has independent directives. This approach is ideal for design exploration. If any solution is re-synthesized, only the directives specified in that solution are applied.	If the C source files are transferred to a third-party or archived, the <code>directives.tcl</code> file must be included. The <code>directives.tcl</code> file is required if the results are to be re-created.
Source Code (Pragma)	The optimization directives are embedded into the C source code. Ideal when the C sources files are shipped to a third-party as C IP. No other files are required to recreate the same results. Useful approach for directives that are unlikely to change, such as TRIPCOUNT and INTERFACE.	If the optimization directives are embedded in the code, they are automatically applied to every solution when re-synthesized.



TIP: When using the Vitis core development kit to define hardware acceleration of your C/C++ code, you should use pragmas in your source code, rather than trying to work with directives in a Tcl file. In the Vitis HLS bottom-up flow (or the Vitis kernel flow) you can use directives to develop different solutions, but should convert your final directives to pragmas in the finished project.

When specifying values for pragma arguments, you can use literal values (for example, 1, 55, 3.14), or pass a macro using #define. The following example shows a pragma with literal values:

```
#pragma HLS ARRAY_PARTITION variable=k_matrix_val type=cyclic factor=5
```

This example uses defined macros:

```
#define E 5
#pragma HLS ARRAY_PARTITION variable=k_matrix_val type=cyclic factor=E
```

Applying Directives to the Proper Scope

Although the Vitis HLS GUI lets you apply directives to specific code objects, the directives are added to the scope that contains the object. For example, you can apply the `INTERFACE` pragma to an interface object in the Vitis HLS GUI, but the directive is applied to the top-level function (scope). The interface port (object) is identified in the directive.

You can apply optimization directives to the following objects and scopes:

- **Functions:** When you apply directives to functions, Vitis HLS applies the directive to all objects within the scope of that function. The effect of any directive stops at the next level of the function hierarchy, and does not apply to sub-functions.



TIP: Directives that include a recursive option, such as the `PIPELINE` directive, can be applied recursively through the hierarchy.

- **Interfaces:** Vitis HLS applies the directive to the top-level function, which is the scope that contains the interface.
- **Loops:** Directives apply to all objects within the scope of the loop.

For example, if you apply the `LOOP_MERGE` directive to a loop, Vitis HLS applies the directive to any sub-loops within the loop, but not to the loop itself. The loop to which the directive is applied is not merged with siblings at the same level of hierarchy.

- **Arrays:** Directives are applied to the scope that contains the array.

Applying Optimization Directives to Global Variables

Directives can only be applied to scopes, or to objects within a scope. As such, they cannot be directly applied to global variables which are declared outside the scope of any function. Therefore, to apply a directive to a global variable you must manually assign it using the following process:

1. With the code open in the Code Editor, select the scope (function, loop or region) where the global variable is used in the Directive view.
2. Right-click and use the **Insert Directive** command to open the Vitis HLS Directives Editor.
3. Select and configure the required directive, and click **OK** to add it.
4. Locate the added directive in the Directive view, and manually edit the `variable` name to assign it to the global variable.

Applying Optimization Directives to Class Objects

Optimization directives can be also applied to objects or scopes defined in a class. The difference is typically that classes are defined in a header file. Use one of the following actions to open the header file:

- From the **Explorer** view in the Vitis HLS GUI, open the `Includes` folder, double-click the header file to open it in the Code Editor.
- From within an open source code file, place the cursor over the `#include` statement for the header file, hold down the **Ctrl** key, and click the header file to open it in the Code Editor.

The **Directives** tab is populated with the objects in the header file, and directives can be applied.



CAUTION! Care should be taken when applying directives as pragmas to a header file. The file might be used by other people or used in other projects. Any directives added as a pragma are applied each time the header file is included in a design.

Applying Optimization Directives to Templates

To apply optimization directives manually on templates when using Tcl commands, specify the template arguments and class when referring to class methods. For example, given the following C++ code:

```
template <uint32 SIZE, uint32 RATE>
void DES10<SIZE,RATE>::calcRUN() {}
```

The following Tcl command is used to specify the `INLINE` directive on the function:

```
set_directive_inline DES10<SIZE,RATE>::calcRUN
```

Using Constants with Pragmas

You can use a constant such as `const int`, or `constexpr` with pragmas or directives. For example:

```
const int MY_DEPTH=1024;
#pragma HLS stream variable=my_var depth=MY_DEPTH
```

You can also use macros in the C code to implement this functionality. The key to using macros is to use a level of hierarchy in the macro. This allows the expansion to be correctly performed. The code can be made to compile as follows:

```
#include <hls_stream.h>
using namespace hls;

#define PRAGMA_SUB(x) _Pragma (#x)
#define PRAGMA_HLS(x) PRAGMA_SUB(x)
#define STREAM_IN_DEPTH 8

void foo (stream<int> &InStream, stream<int> &OutStream) {

    // Legal pragmas
    PRAGMA_HLS(HLS stream depth=STREAM_IN_DEPTH variable=InStream)
    #pragma HLS stream depth=8 variable=OutStream
}
```

Failure to Satisfy Optimization Directives

When optimization directives are applied, Vitis HLS outputs information to the console (and log file) detailing the progress. In the following example the PIPELINE directive was applied to the C function with an II=1 (iteration interval of 1) but synthesis failed to satisfy this objective.

```
INFO: [SCHED 11] Starting scheduling ...
INFO: [SCHED 61] Pipelining function 'array_RAM'.
WARNING: [SCHED 63] Unable to schedule the whole 2 cycles 'load' operation
('d_i_load', array_RAM.c:98) on array 'd_i' within the first cycle (II = 1).
WARNING: [SCHED 63] Please consider increasing the target initiation
interval of the
pipeline.
WARNING: [SCHED 69] Unable to schedule 'load' operation ('idx_load_2',
array_RAM.c:98) on array 'idx' due to limited memory ports.
INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 4, Depth: 6.
INFO: [SCHED 11] Finished scheduling.
```



IMPORTANT! If Vitis HLS fails to satisfy an optimization directive, it automatically relaxes the optimization target and seeks to create a design with a lower performance target. If it cannot relax the target, it will halt with an error.

By seeking to create a design which satisfies a lower optimization target, Vitis HLS is able to provide three important types of information:

- What target performance can be achieved with the current C code and optimization directives.
- A list of the reasons why it was unable to satisfy the higher performance target.
- A design which can be analyzed to provide more insight and help understand the reason for the failure.

In message SCHED - 69, the reason given for failing to reach the target II is due to limited ports. The design must access a block RAM, and a block RAM only has a maximum of two ports.

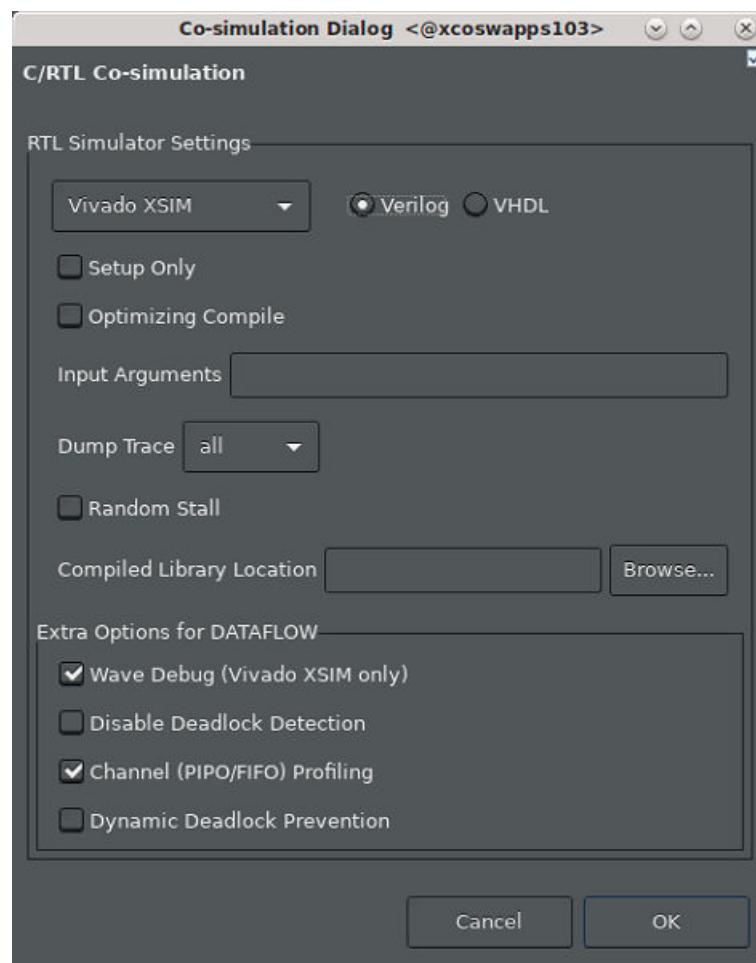
The next step after a failure such as this is to analyze what the issue is. In this example, analyze line 52 of the code and/or use the **Analysis** perspective to determine the bottleneck and if the requirement for more than two ports can be reduced or determine how the number of ports can be increased.

After the design is optimized and the desired performance achieved, the RTL can be verified and the results of synthesis packaged as IP.

C/RTL Co-Simulation in Vitis HLS

If you added a C test bench to the project for simulation purposes, you can also use it for C/RTL co-simulation to verify that the RTL is functionally identical to the C source code. Select the **Run Cosimulation** command from the Flow Navigator to verify the RTL results of synthesis. The Co-simulation Dialog box is opened as shown in the following figure lets you select which type of RTL output to use for verification (Verilog or VHDL) and which HDL simulator to use for the simulation.

Figure 118: Co-Simulation Dialog Box



The dialog box features the following settings:

- **Simulator:** Choose from one of the supported HDL simulators in the Vivado® Design Suite. Vivado simulator is the default simulator.
- **Language:** Specify the use of Verilog or VHDL as the output language for simulation.
- **Setup Only:** Create the required simulation files, but do not run the simulation. The simulation executable can be run from a command shell at a later time.
- **Optimizing Compile:** Enable optimization to improve the runtime performance, if possible, at the expense of compilation time.
- **Input Arguments:** Specify any command-line arguments to the C test bench.
- **Dump Trace:** Specifies the level of trace file output written to the `sim/Verilog` or `sim/VHDL` directory of the current solution when the simulation executes. Options include:
 - **all:** Output all port and signal waveform data being saved to the trace file.
 - **port:** Output waveform trace data for the top-level ports only.
 - **none:** Do not output trace data.
- **Random Stall:** Applies a randomized stall for each data transmission.
- **Compiled Library Location:** Specifies the directory for the compiled simulation library to use with third-party simulators.
- **Extra Options for DATAFLOW:**
 - **Wave Debug:** Enables waveform visualization of all processes in the RTL simulation. This option is only supported when using Vivado logic simulator. Enabling this will launch the Simulator GUI to let you examine dataflow activity in the waveforms generated by simulation. Refer to the *Vivado Design Suite User Guide: Logic Simulation (UG900)* for more information on that tool.
 - **Disable Deadlock Detection:** Disables deadlock detection, and opening the [Cosim Deadlock Viewer](#) in co-simulation.
 - **Channel (PIPO/FIFO) Profiling:** Enables capturing profile data for display in the [Dataflow Viewer](#).
 - **Dynamic Deadlock Prevention:** Prevent deadlocks by enabling automatic FIFO channel size tuning for dataflow profiling during co-simulation.



TIP: You can pre-configure C/RTL Co-Simulation by right-clicking a solution in the Explorer view and selecting the **Solutions Settings** command to open the Solution Settings dialog box, and editing the Co-simulation settings. The settings are the same as described above, but can be configured prior to running the simulation.

After the C/RTL co-simulation completes, the console displays the following messages to confirm the verification was successful:

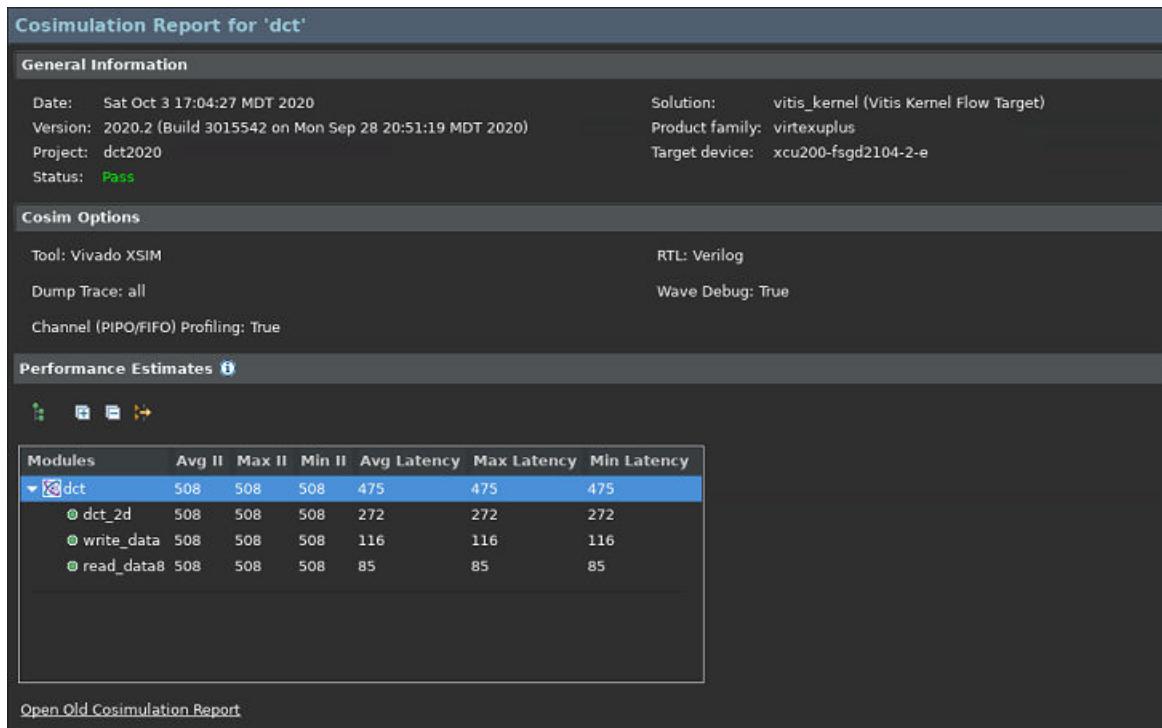
```
INFO: [Common 17-206] Exiting xsim ...
INFO: [COSIM 212-316] Starting C post checking ...
...
Test passed !
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
Finished C/RTL cosimulation.
```

Any `printf` commands in the C test bench, or `hls::print` statements in the kernel or IP are also echoed to the console during simulation.

As described in [Writing a Test Bench](#), the test bench verifies output from the top-level function for synthesis, and returns zero to the `main()` function of the test bench if the output is correct. Vitis HLS uses the same return value for both C simulation and C/RTL co-simulation to determine if the results are correct. If the C test bench returns a non-zero value, Vitis HLS reports that the simulation failed.

The Vitis HLS GUI automatically switches to the Analysis perspective after simulation and opens the Cosimulation Report showing the pass or fail status and the measured statistics on latency and II. Any additional reports that are generated, such as the Dataflow report, are also opened in the Analysis perspective.

Figure 119: Cosimulation Report



The Cosimulation Report displays the full design hierarchy, and if **Channel (PIPO/FIFO) Profiling** is enabled, you will be able to see details of the dataflow regions as well.



IMPORTANT! *II is marked as NA in the Cosimulation Report unless the transaction number in the RTL simulation is greater than 1. If you want to calculate II, you must ensure there are at least two transactions in the RTL simulation as described in [Writing a Test Bench](#).*

Output of C/RTL Co-Simulation

When C/RTL Cosimulation completes, the `sim` folder is created inside the solution folder. This folder contains the following elements:

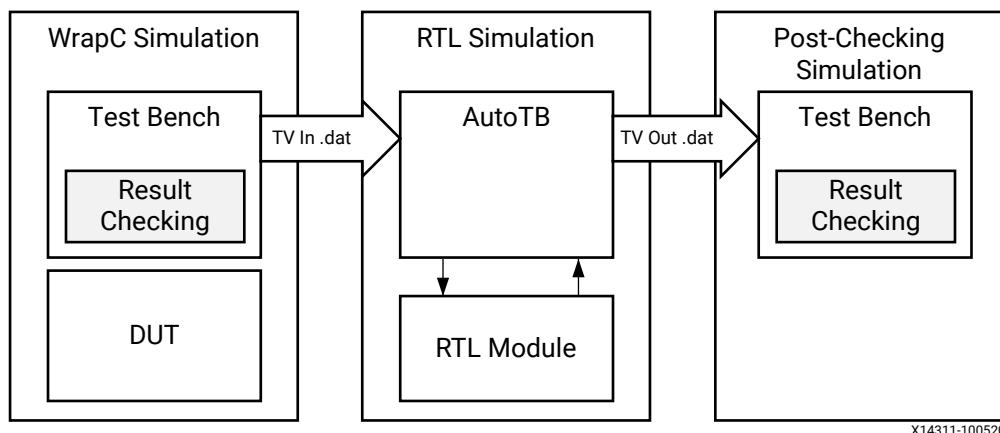
- The `sim/report` folder contains the report and log file for each type of RTL simulated.
- A verification folder named `sim/verilog` or `vhdl` is created for each RTL language that is verified.
 - The RTL files used for simulation are stored in the `verilog` or `vhdl` folder.
 - The RTL simulation is executed in the verification folder.
 - Any outputs, such as trace files and waveform files, are written to the `verilog` or `vhdl` folder.
- Additional folders `sim/autowrap`, `tv`, `wrap` and `wrap_pc` are work folders used by Vitis HLS. There are no user files in these folders.



TIP: *If the **Setup Only** option was selected in the C/RTL Co-Simulation dialog box, an executable is created in the verification folder but the simulation is not run. The simulation can be manually run by executing the simulation .exe at the command prompt.*

Automatically Verifying the RTL

Figure 120: C/RTL Verification Flow



C/RTL co-simulation uses a C test bench, running the `main()` function, to automatically verify the RTL design running in behavioral simulation. The C/RTL verification process consists of three phases:

1. The C simulation is executed and the inputs to the top-level function, or the Design-Under-Test (DUT), are saved as “input vectors.”
2. The “input vectors” are used in an RTL simulation using the RTL created by Vitis HLS in Vivado simulator, or a supported third-party HDL simulator. The outputs from the RTL, or results of simulation, are saved as “output vectors.”
3. The “output vectors” from the RTL simulation are returned to the `main()` function of the C test bench to verify the results are correct. The C test bench performs verification of the results, in some cases by comparing to known good results.

The following messages are output by Vitis™ HLS as verification progresses:

While running C simulation:

```
INFO: [COSIM 212-14] Instrumenting C test bench ...
Build using ".../bin/g++"
Compiling dct_test.cpp-pre.cpp.tb.cpp
Compiling dct_inline.cpp-pre.cpp.tb.cpp
Compiling apatb_dct.cpp
Generating cosim.tv.exe
INFO: [COSIM 212-302] Starting C TB testing ...
Test passed !
```

At this stage, because the C simulation was executed, any messages written by the C test bench will be output to the Console window and log file.

While running RTL simulation:

```
INFO: [COSIM 212-333] Generating C post check test bench ...
INFO: [COSIM 212-12] Generating RTL test bench ...
INFO: [COSIM 212-1] *** C/RTL co-simulation file generation completed. ***
INFO: [COSIM 212-323] Starting verilog/vhdl simulation.
INFO: [COSIM 212-15] Starting XSIM ...
```

At this stage, any messages from the RTL simulation are output in console window or log file.

While checking results back in the C test bench:

```
INFO: [COSIM 212-316] Starting C post checking ...
Test passed !
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
```

The following are requirements of C/RTL co-simulation:

- The test bench must be self-checking as described in [Writing a Test Bench](#), and return a value of 0 if the test passes or returns a non-zero value if the test fails.
- Any third-party simulators must be available in the search path to be launched by Vitis HLS.
- [Interface Synthesis Requirements](#) must be met.
- Any `arrays` or `structs` on the design interface cannot use the optimization directives listed in [Unsupported Optimizations for Co-Simulation](#).
- IP simulation libraries must be compiled for use with third-party simulators as described in [Simulating IP Cores](#).

Interface Synthesis Requirements

To use the C/RTL co-simulation feature to verify the RTL design, at least one of the following conditions must be true:

- Top-level function must be synthesized using an `ap_ctrl_chain` or `ap_ctrl_hs` block-level protocol
- Design must be purely combinational
- Top-level function must have an initiation interval of 1
- Interfaces must be all arrays that are streaming and implemented with `axis` or `ap_hs` interface modes

Note: The `hls::stream` variables are automatically implemented as `ap_fifo` interfaces.

If at least one of these conditions is not met, C/RTL co-simulation halts with the following message:

```
@E [SIM-345] Cosim only supports the following 'ap_ctrl_none' designs: (1)
combinational designs; (2) pipelined design with task interval of 1; (3)
designs with
array streaming or hls_stream ports.
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```



IMPORTANT! If the design is specified to use the block-level IO protocol `ap_ctrl_none` and the design contains any `hls::stream` variables which employ non-blocking behavior, C/RTL co-simulation is not guaranteed to complete.

If any top-level function argument is specified as an AXI4-Lite interface, the function return must also be specified as an AXI4-Lite interface.

Verification of DATAFLOW and DEPENDENCE

C/RTL co-simulation automatically verifies aspects of the DATAFLOW and DEPENDENCE directives.

If the DATAFLOW directive is used to pipeline tasks, it inserts channels between the tasks to facilitate the flow of data between them. It is typical for the channels to be implemented with FIFOs and the FIFO depth specified using the STREAM directive, or the `config_dataflow` command. If a FIFO depth is too small, the RTL simulation can stall. For example, if a FIFO is specified with a depth of 2 but the producer task writes three values before any data values are read by the consumer task, the FIFO blocks the producer. In some conditions this can cause the entire design to stall as described in [Cosim Deadlock Viewer](#).

In this case, C/RTL co-simulation issues a message as shown below, indicating the channel in the DATAFLOW region is causing the RTL simulation to stall.

```
///////////
/
// ERROR!!! DEADLOCK DETECTED at 1292000 ns! SIMULATION WILL BE STOPPED! //
///////////
/
///////////
// Dependence cycle 1:
// (1): Process: hls_fft_1kxburst.fft_rank_rad2_nr_man_9_U0
//       Channel: hls_fft_1kxburst.stage_chan_in1_0_V_s_U, FULL
//       Channel: hls_fft_1kxburst.stage_chan_in1_1_V_s_U, FULL
//       Channel: hls_fft_1kxburst.stage_chan_in1_0_V_1_U, FULL
//       Channel: hls_fft_1kxburst.stage_chan_in1_1_V_1_U, FULL
// (2): Process: hls_fft_1kxburst.fft_rank_rad2_nr_man_6_U0
//       Channel: hls_fft_1kxburst.stage_chan_in1_2_V_s_U, EMPTY
//       Channel: hls_fft_1kxburst.stage_chan_in1_2_V_1_U, EMPTY
///////////
// Total 1 cycles detected!
///////////
```

If co-simulation is attempted from the Vitis HLS IDE and the simulation results in a deadlock, the Vitis HLS IDE will automatically launch the Dataflow Viewer and show the processes involved in the deadlock (displayed in red). It will also show which channels are full (in red) versus empty (in white). In this case, review the implementation of the channels between the tasks and ensure any FIFOs are large enough to hold the data being generated.

In a similar manner, the RTL test bench is also configured to automatically check the validity of false dependencies specified using the DEPENDENCE directive. A warning message during co-simulation indicates the dependency is not false, and the corresponding directive must be removed to achieve a functionally valid design.



TIP: The `-disable_deadlock_detection` option of the `cosim_design` command disables these checks.

Unsupported Optimizations for Co-Simulation

For Vivado IP mode, automatic RTL verification does not support cases where multiple transformations are performed on arrays on the interface, or arrays within structs.



IMPORTANT! This feature is not supported for the Vitis kernel flow.

In order for automatic verification to be performed, arrays on the function interface, or array inside structs on the function interface, can use any of the following optimizations, but not two or more:

- Vertical mapping on arrays of the same size
- Reshape
- Partition, for dimension 1 of the array

Automatic RTL verification does not support any of the following optimizations used on a top-level function interface:

- Horizontal mapping.
- Vertical mapping of arrays of different sizes.
- Conditional access on the AXI4-Stream with register slice enabled.
- Mapping arrays to streams.

Simulating IP Cores

When the design is implemented with floating-point cores, bit-accurate models of the floating-point cores must be made available to the RTL simulator. This is automatically accomplished if the RTL simulation is performed using the Vivado logic simulator. However, for supported third-party HDL simulators, the Xilinx floating-point library must be pre-compiled and added to the simulator libraries.

For example, to compile the Xilinx floating-point library in Verilog for use with the VCS simulator, open the Vivado IDE and enter the following command in the Tcl Console window:

```
compile_simlib -simulator vcs_mx -family all -language verilog
```

This creates the floating-point library in the current directory for VCS. See the Vivado Tcl Console window for the directory name. In this example, it is `./rev3_1`.

You must refer to this library from within the Vitis HLS IDE by specifying the Compiled Library Location field in the Co-simulation dialog box as described in [C/RTL Co-Simulation in Vitis HLS](#), or by running C/RTL co-simulation using the following command:

```
cosim_design -tool vcs -compiled_library_dir <path_to_library>/rev3_1
```

Analyzing RTL Simulations

When the C/RTL co-simulation completes, the simulation report opens and shows the measured latency and II. These results may differ from values reported after HLS synthesis, which are based on the absolute shortest and longest paths through the design. The results provided after C/RTL co-simulation show the actual values of latency and II for the given simulation data set (and may change if different input stimuli is used).

In non-pipelined designs, C/RTL co-simulation measures latency between `ap_start` and `ap_done` signals. The II is 1 more than the latency, because the design reads new inputs 1 cycle after all operations are complete. The design only starts the next transaction after the current transaction is complete.

In pipelined designs, the design might read new inputs before the first transaction completes, and there might be multiple `ap_start` and `ap_ready` signals before a transaction completes. In this case, C/RTL co-simulation measures the latency as the number of cycles between data input values and data output values. The II is the number of cycles between `ap_ready` signals, which the design uses to requests new inputs.

Note: For pipelined designs, the II value for C/RTL co-simulation is only determined if the design is simulated for multiple transactions.

Viewing Simulation Waveforms

To view waveform data during RTL co-simulation, you must enable the following in the Co-simulation Dialog box:

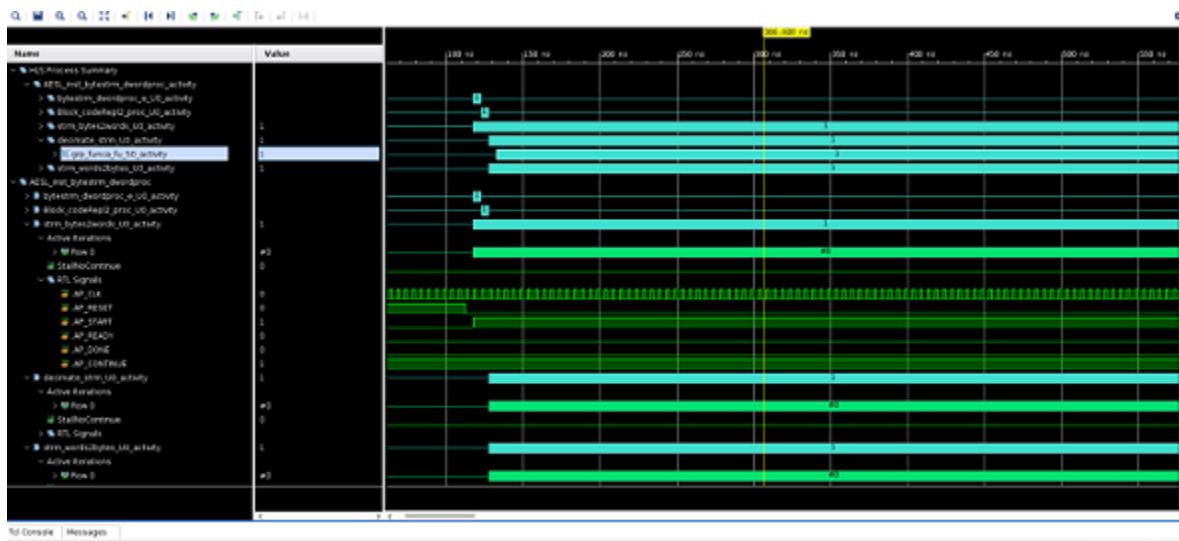
- Select **Vivado XSIM** as the RTL simulator.
- Enable Dump Trace with either the **port** or **all** options.

Vivado simulator GUI opens and displays all the processes in the RTL design. Visualizing the active processes within the HLS design allows detailed profiling of process activity and duration within each activation of the top module. The visualization helps you to analyze individual process performance, as well as the overall concurrent execution of independent processes. Processes dominating the overall execution have the highest potential to improve performance, provided process execution time can be reduced.

This visualization is divided into two sections:

- HLS process summary contains a hierarchical representation of the activity report for all processes.
 - **DUT name:** <name>
 - **Function:** <function name>
- Dataflow analysis provides detailed activity information about the tasks inside the dataflow region.
 - **DUT name:** <name>
 - **Function:** <function name>
 - **Dataflow/Pipeline Activity:** Shows the number of parallel executions of the function when implemented as a dataflow process.
 - **Active Iterations:** Shows the currently active iterations of the dataflow. The number of rows is dynamically incremented to accommodate for the visualization of any concurrent execution.
 - **StallNoContinue:** A stall signal that tells if there were any output stalls experienced by the dataflow processes (the function is done, but it has not received a continue from the adjacent dataflow process).
 - **RTL Signals:** The underlying RTL control signals that interpret the transaction view of the dataflow process.

Figure 121: Waveform Viewer



After C/RTL co-simulation completes, you can reopen the RTL waveforms in the Vivado IDE by clicking the **Open Wave Viewer** toolbar button, or selecting **Solution** → **Open Wave Viewer**.



IMPORTANT! When you open the Vivado IDE using this method, you can only use the waveform analysis features, such as zoom, pan, and waveform radix.

Cosim Deadlock Viewer

A deadlock is a situation in which processes inside a DATAFLOW region share the same channels, effectively preventing each other from writing or reading from it, resulting in both processes getting stuck. This scenario is common when there are either FIFO's or a mix of PIPOs and FIFOs as channels inside the DATAFLOW.

The deadlock viewer visualizes this deadlock scenario on the static dataflow viewer. It highlights the problematic processes and channels. The viewer also provides a cross-probing capability to link between the problematic dataflow channels and the associated source code. The user can use the information in solving the issue with less time and effort. The viewer automatically opens only after the co-simulation detects the deadlock situation and the co-sim run has finished.

A small example is shown below. The dataflow region consists of two processes which are communicating through PIPO and FIFO. The first loop in `proc_1` writes 10 data items in `data_channel1`, before writing anything in `data_array`. Because of the insufficient FIFO depth the `data_channel1` loop does not complete which blocks the rest of the process. Then `proc_2` blocks because it cannot read the data from `data_channel2` (because it is empty), and cannot remove data from `data_channel1`. This creates a deadlock that requires increasing the size of `data_channel1` to at least 10.

```

void example(hls::stream<data_t>& A, hls::stream<data_t>& B){
#pragma HLS dataflow
.

.

hls::stream<int> data_channel;
int data_array[10];
#pragma HLS STREAM variable=data_channel depth=8 dim=1
    proc_1(A, data_channel, data_array);
    proc_2(B, data_channel, data_array);
}

void proc_1(hls::stream<data_t>& A, hls::stream<int>& data_channel, int
data_array[10]){
    .
    for(i = 0; i < 10; i++){
        tmp = A.read();
        tmp.data = tmp.data.to_int();
        data_channel.write(tmp.data);
    }
    for(i = 0; i < 10; i++){
        data_array[i] = i + tmp.data.to_int();
    }
}

void proc_2(hls::stream<data_t>& B, hls::stream<int>& data_channel, int
data_array[10]){
    int i;
    .
    for(i = 0; i < 10; i++){
        if (i == 0){
            tmp.data = data_channel.read() + data_array[5];
        }
        else {
            tmp.data = data_channel.read();
        }
        B.write(tmp);
    }
}

```

Co-sim Log:

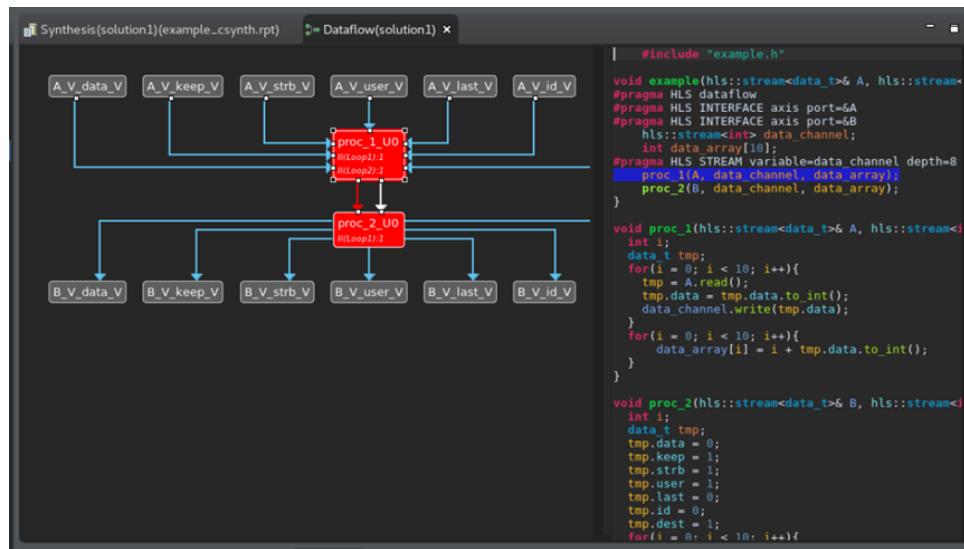
```

///////////
/////
// Inter-Transaction Progress: Completed Transaction / Total Transaction
// Intra-Transaction Progress: Measured Latency / Latency Estimation * 100%
//
// RTL Simulation : "Inter-Transaction Progress" ["Intra-Transaction
Progress"] @ "Simulation Time"
///////////
// RTL Simulation : 0 / 1 [0.00%] @ "105000"

```

```
//////////  
/  
// ERROR!!! DEADLOCK DETECTED at 132000 ns! SIMULATION WILL BE STOPPED! //  
//////////  
/  
//////////  
// Dependence cycle 1:  
// (1): Process: example_example.proc_1_U0  
//       Channel: example_example.data_channel1_U, FULL  
// (2): Process: example_example.proc_2_U0  
//       Channel: example_example.data_array_U, EMPTY  
//////////  
// Totally 1 cycles detected!  
//////////
```

Figure 122: Deadlock Viewer



Debugging C/RTL Co-Simulation

When C/RTL co-simulation completes, Vitis HLS typically indicates that the simulations passed and the functionality of the RTL design matches the initial C code. When the C/RTL co-simulation fails, Vitis HLS issues the following message:

```
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

Following are the primary reasons for a C/RTL co-simulation failure:

- Incorrect environment setup
- Unsupported or incorrectly applied optimization directives
- Issues with the C test bench or the C source code

To debug a C/RTL co-simulation failure, run the checks described in the following sections. If you are unable to resolve the C/RTL co-simulation failure, see [Xilinx Support](#) for support resources, such as answers, documentation, downloads, and forums.

Setting Up the Environment

Check the environment setup as shown in the following table.

Table 23: Debugging Environment Setup

Questions	Actions to Take
Are you using a third-party simulator?	Ensure the path to the simulator executable is specified in the system search path. When using the Vivado simulator, you do not need to specify a search path. Ensure that you have compiled the simulation libraries as discussed in Simulating IP Cores .
Are you running Linux?	Ensure that your setup files (for example <code>.cshrc</code> or <code>.bashrc</code>) do not have a change directory command. When C/RTL co-simulation starts, it spawns a new shell process. If there is a <code>cd</code> command in your setup files, it causes the shell to run in a different location and eventually C/RTL co-simulation fails.

Optimization Directives

Check the optimization directives as shown in the following table.

Table 24: Debugging Optimization Directives

Questions	Actions to Take
Are you using the DEPENDENCE directive?	Remove the DEPENDENCE directives from the design to see if C/RTL co-simulation passes. If co-simulation passes, it likely indicates that the TRUE or FALSE setting for the DEPENDENCE directive is incorrect as discussed in Verification of DATAFLOW and DEPENDENCE .
Does the design use <code>volatile</code> pointers on the top-level interface?	Ensure the DEPTH option is specified on the INTERFACE directive. When <code>volatile</code> pointers are used on the interface, you must specify the number of reads/writes performed on the port in each transaction or each execution of the C function.
Are you using FIFOs with the DATAFLOW optimization?	Check to see if C/RTL co-simulation passes with the standard ping-pong buffers. Check to see if C/RTL co-simulation passes without specifying the size for the FIFO channels. This ensures that the channel defaults to the size of the array in the C code. Reduce the size of the FIFO channels until C/RTL co-simulation stalls. Stalling indicates a channel size that is too small. Review your design to determine the optimal size for the FIFOs. You can use the STREAM directive to specify the size of individual FIFOs.

Table 24: Debugging Optimization Directives (cont'd)

Questions	Actions to Take
Are you using supported interfaces?	Ensure you are using supported interface modes. For details, see Interface Synthesis Requirements .
Are you applying multiple optimization directives to arrays on the interface?	Ensure you are using optimizations that are designed to work together. For details, see Unsupported Optimizations for Co-Simulation .
Are you using arrays on the interface that are mapped to streams?	To use interface-level streaming (the top-level function of the DUT), use <code>hls::stream</code> .

C Test Bench and C Source Code

Check the C test bench and C source code as shown in the following table.

Table 25: Debugging the C Test Bench and C Source Code

Questions	Actions to Take
Does the C test bench check the results and return the value 0 (zero) if the results are correct?	Ensure the C test bench returns the value 0 for C/RTL co-simulation. Even if the results are correct, the C/RTL co-simulation feature reports a failure if the C test bench fails to return the value 0.
Is the C test bench creating input data based on a random number?	Change the test bench to use a fixed seed for any random number generation. If the seed for random number generation is based on a variable, such as a time-based seed, the data used for simulation is different each time the test bench is executed, and the results can vary.
Are you using pointers on the top-level interface that are accessed multiple times?	Use a <code>volatile</code> pointer for any pointer that is accessed multiple times within a single transaction (one execution of the C function). If you do not use a <code>volatile</code> pointer, everything except the first read and last write is optimized out to adhere to the C standard.
Does the C code contain undefined values or perform out-of-bounds array accesses?	Confirm all arrays are correctly sized to match all accesses. Loop bounds that exceed the size of the array are a common source of issues (for example, N accesses for an array sized at N-1). Confirm that the results of the C simulation are as expected and that output values were not assigned random data values. Consider using the industry-standard Valgrind application outside of the HLS design environment to confirm that the C code does not have undefined or out-of-bounds issues. It is possible for a C function to execute and complete even if some variables are undefined or are out-of-bounds. In the C simulation, undefined values are assigned a random number. In the RTL simulation, undefined values are assigned an unknown or X value.
Are you using floating-point math operations in the design?	Check that the C test bench results are within an acceptable error range instead of performing an exact comparison. For some of the floating point math operations, the RTL implementation is not identical to the C. For details, see Verification and Math Functions . Ensure that the RTL simulation models for the floating-point cores are provided to the third-party simulator. For details, see Simulating IP Cores .

Table 25: Debugging the C Test Bench and C Source Code (cont'd)

Questions	Actions to Take
Are you using Xilinx IP blocks and a third-party simulator?	Ensure that the path to the Xilinx IP simulation models is provided to the third-party simulator.
Are you using the <code>hls::stream</code> construct in the design that changes the data rate (for example, decimation or interpolation)?	<p>Analyze the design and use the STREAM directive to increase the size of the FIFOs used to implement the <code>hls::stream</code>.</p> <p>By default, an <code>hls::stream</code> is implemented as a FIFO with a depth of 2. If the design results in an increase in the data rate (for example, an interpolation operation), a default FIFO size of 2 might be too small and cause the C/RTL co-simulation to stall.</p>
Are you using very large data sets in the simulation?	<p>Use the <code>reduce_diskspace</code> option when executing C/RTL co-simulation. In this mode, HLS only executes 1 transaction at a time. The simulation might run marginally slower, but this limits storage and system capacity issues.</p> <p>The C/RTL co-simulation feature verifies all transaction at one time. If the top-level function is called multiple times (for example, to simulate multiple frames of video), the data for the entire simulation input and output is stored on disk. Depending on the machine setup and OS, this might cause performance or execution issues.</p>

Exporting the RTL Design

The final step in the Vitis™ HLS flow is to export the RTL design in a form that can be used by other tools in the Xilinx® design flow. Click the **Export RTL** command in the Flow Navigator to open the Export RTL dialog box shown in the following figure.



TIP: When Vitis HLS reports the results of the high-level synthesis, it only provides an estimate of the results with projected clock frequencies and resource utilization (LUTs, DSPs, BRAMs, etc.). These results are only estimates because Vitis HLS cannot know what optimizations or routing delays will be in the final synthesized or implemented design. Therefore use the **Run Implementation** command from Flow Navigator to return reports from Vivado synthesis or place and route.

Figure 123: Export RTL Dialog Box

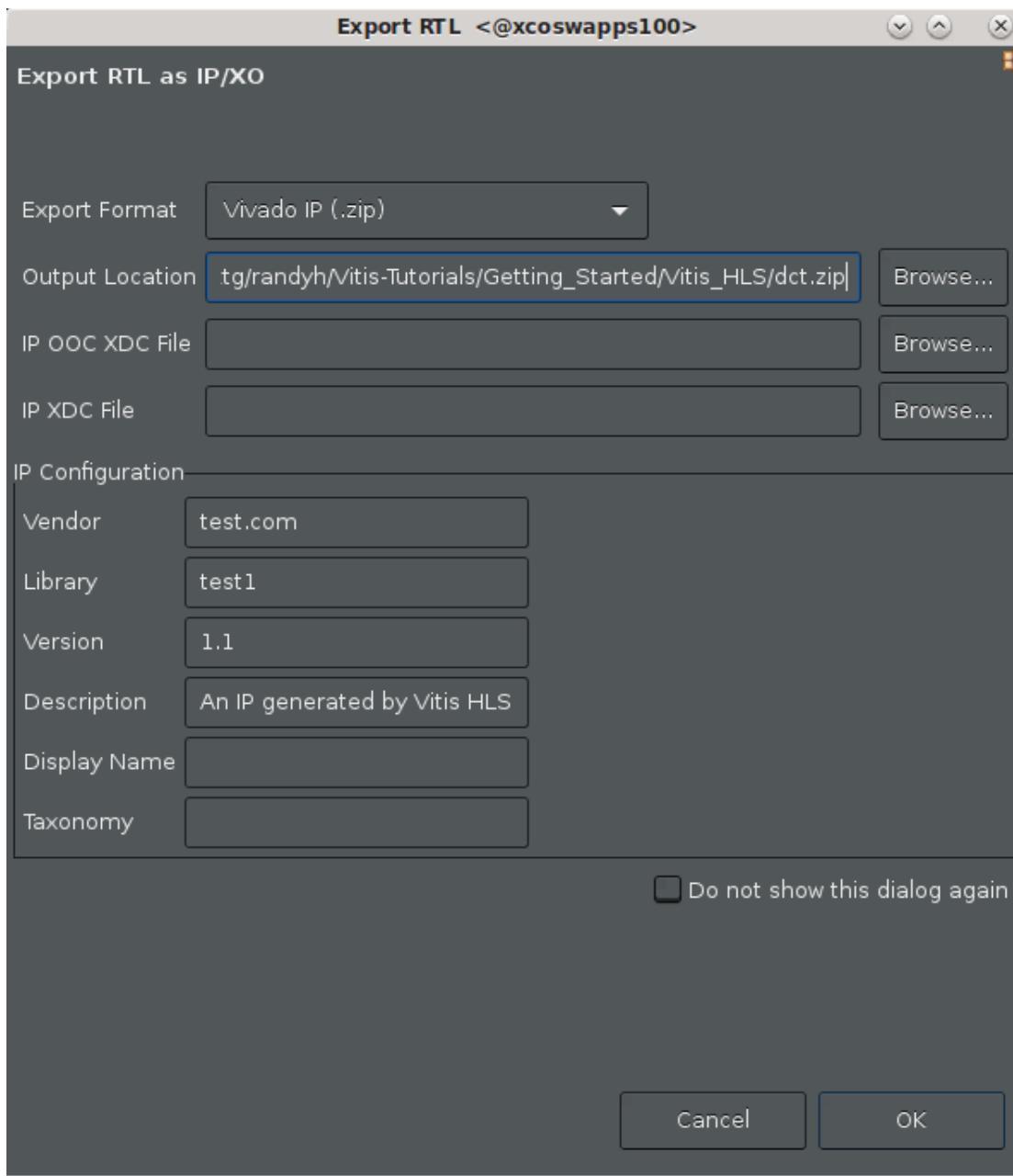


Table 26: RTL Export Selections

Export Format	Default Location	Comments
Vivado IP (.zip)	solution/impl/export.zip	The IP is exported as a ZIP file that can be added to the Vivado IP catalog. The <code>impl/ip</code> folder also contains the contents of the unzipped IP.

Table 26: RTL Export Selections (cont'd)

Export Format	Default Location	Comments
Vitis Kernel (.xo)	solution/impl/export.xo	The XO file output can be used for linking by the Vitis compiler in the application acceleration development flow. You can link the Vitis kernel with other kernels, and the target accelerator card, to build the <code>xclbin</code> file for your accelerated application.
Vivado IP for System Generator	solution/impl/ip	This option creates IP for use with the Vivado edition of System Generator for DSP.

- Output Location:** Lets you specify the path and file name for the exported RTL design.
- IP OOC XDC File:** Specifies an XDC file to be used for the RTL IP for out-of-context (OOC) synthesis.
- IP XDC File:** Lets you specify an XDC file for use during Vivado place and route.

IP Configuration

When you select the **Vivado IP** format on the Export RTL dialog box, you also have the option of configuring specific fields, such as the Vendor, Library, Name, and Version (VNV) of the IP.

The Configuration information is used to differentiate between multiple instances of the same IP when it is loaded into the Vivado IP catalog. For example, if an implementation is packaged for the IP catalog, and then a new solution is created and packaged as IP, the new solution by default has the same name and configuration information. If the new solution is also added to the IP catalog, the IP catalog will identify it as an updated version of the same IP and the last version added to the IP catalog will be used.

The Configuration options, and their default values are listed below:

- Vendor:** xilinx.com
- Library:** hls
- Version:** 1.0
- Description:** An IP generated by Vitis HLS
- Display Name:** This field is left blank by default
- Taxonomy:** This field is left blank by default

After the IP packaging process is complete, the ZIP file archive written to the specified **Output Location**, or written in the `solution/impl` folder, can be imported into the Vivado IP catalog and used in any design.

Software Driver Files

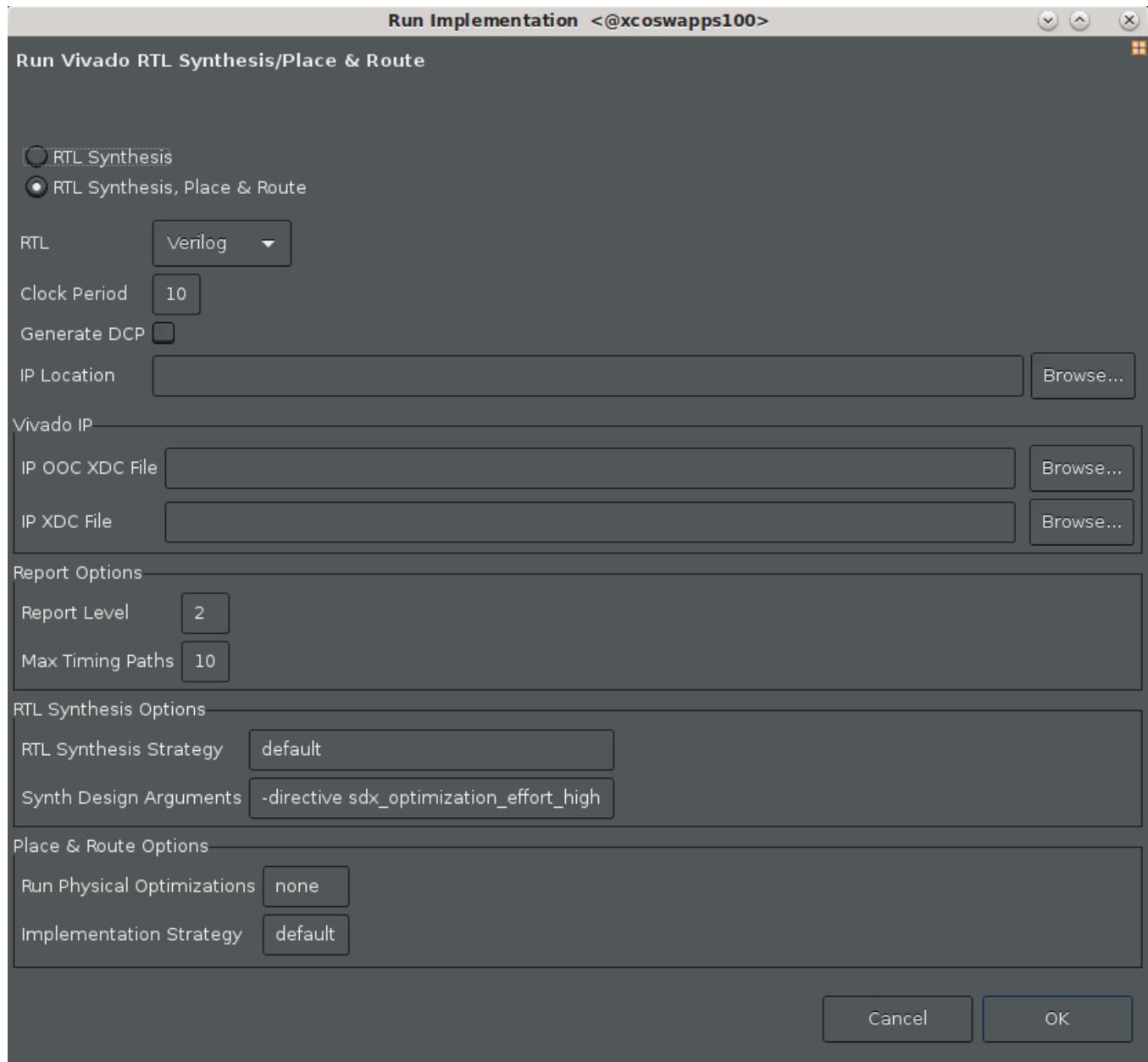
For designs that include AXI4-Lite slave interfaces, a set of software driver files is created during the export process. These C driver files can be included in a Vitis embedded software development project, and used to access the AXI4-Lite slave port.

The software driver files are written to directory `solution/impl/ip/drivers` and are included in the packaged IP `export.zip`. Refer to [AXI4-Lite Interface](#) for details on the C driver files.

Running Implementation

The Vitis HLS tool is limited in terms of the estimations it can provide about the RTL design that it generates. It can project resource utilization and timing of the end result, but these are just projections. To get a better view of the RTL design, you can actually run Vivado synthesis and place and route on the generated RTL design, and review actual results of timing and resource utilization. Select the **Run Implementation** command from the Flow Navigator to open the dialog box as shown below.

Figure 124: Run Implementation



The dialog presents the choice of running **RTL Synthesis** or **RTL Synthesis, Place & Route**. The dialog box is largely unchanged in either selection, with the exception of the **Place & Route Options** that appear at the bottom.

- **RTL:** Generate RTL in Verilog or VHDL form.
- **Clock Period:** Specify the clock period, which is defined by the active solution by default.
- **Generate DCP:** Check box to generate a DCP file for the synthesized or implemented design.

- **IP Location:** Specify the location to write the generated IP file.
- **IP OOC XDC File:** Specifies an XDC file to be used for the RTL IP for out-of-context (OOC) synthesis.
- **IP XDC File:** Lets you specify an XDC file for use during Vivado place and route.
- **Report Level:** Defines the report-level generated during synthesis or implementation.
- **Max Timing Paths:** Specify the number of timing paths to extract from the Timing Summary report. The worst case paths are returned as defined by the specified value.
- **RTL Synthesis Strategy:** Specify the strategy to employ in the synthesis run.
- **Synth Design Arguments:** Specify options for the `synth_design` command.
- **Run Physical Optimizations:** Specify the physical optimization to run. Choices include: `none`, `place`, `route`, and `all`
- **Implementation Strategy:** Specify the strategy to employ in the implementation run.



TIP: You can cancel the Implementation run using the **Stop Implementation** command from the Flow Navigator.

Implementation Report

The Implementation Report contains the results of Synthesis and Place and Route if it was run. The sections of the report include the following:

- **General Information:** Provides general information related to the design and implementation.
- **Run Constraints and Options:** Reports the constraints and options that were set for the RTL Synthesis run and/or the Place & Route run. This shows you what constraints were set and/or modified for the run.
- **Resource Usage/Final Timing:** The Resource Usage and the Final Timing sections show a quick summary of the resources and timing achieved by either the RTL Synthesis run or the Place & Route run. These sections give a very high-level overview of the resource utilization and status on whether timing goals were met or not. The information in the succeeding sections provide details useful in debugging timing issues.
- **Resources:** A detailed per-module split up of resources is shown in this table. In addition, the tables can also show the original variable and source location information from the source code. If a particular resource was the result of a user-specified pragma, then this can also be shown in the table. This allows you to relate your C code with the synthesized RTL implementation. Inspecting this report is very beneficial because this is after Vivado has synthesized the design and therefore, functional blocks like DSPs and other logic units have all now been instantiated in the circuit.

- **Fail Fast:** The fail fast reports that Vivado provides can guide your investigation into specific issues encountered by the tool. In the fail fast report, you should look into anything with the Status of **REVIEW** to improve the implementation and timing closure. Different sections of the fail fast report include:
 - **Design Characteristics:** The default utilization guidelines are based on SSI technology devices and can be relaxed for non-SSI technology devices. Designs with one or more REVIEW checks are feasible but are difficult to implement.
 - **Clocking Checks:** These checks are critical and must be addressed.
 - **LUT and Net Budgeting:** Use a conservative method to better predict which logic paths are unlikely to meet timing after placement with high device utilization.

Figure 125: Design Characteristics

Fail Fast Synth

Created on Thu May 06 16:45:38 PDT 2021 with report_failfast (2020.12.07)

Design Summary

design_1

xvc1902-viva1596-1LP-e-S-es1

Criteria	Guideline	Actual	Status
LUT	70%	7.45%	OK
FD	50%	3.99%	OK
LUTRAM+SRL	25%	1.49%	OK
LOOKAHEAD8	25%	0.07%	OK
DSP	80%	0.20%	OK
RAMB	80%	6.51%	OK
URAM	80%	0.00%	OK
DSP+RAMB+URAM (Avg)	70%	3.35%	OK
BUFGCE* + BUFGCTRL	24	0	OK
DONT_TOUCH (cells/net)	0	0	OK
MARK_DEBUG (nets)	0	0	OK
Control Sets	16872	407	OK
Average Fanout for modules > 100k cells	4	3.56	OK
Non-FD high fanout nets > 10k loads	0	2	REVIEW
TIMING-6 (No common primary clock between related clocks)	0	0	OK
TIMING-7 (No common node between related clocks)	0	0	OK
TIMING-8 (No common period between related clocks)	0	0	OK
TIMING-14 (LUT on the clock tree)	0	0	OK
TIMING-35 (No common node in paths with the same clock)	0	0	OK
Number of paths above max LUT budgeting (0.449ns)	0	100	REVIEW
Number of paths above max Net budgeting (0.302ns)	0	100	REVIEW

Fail Fast Routed

Created on Thu May 06 17:33:58 PDT 2021 with report_failfast (2020.12.07)

Design Summary

- **Timing Paths:** The Timing Paths reports show the timing critical paths that result in the worst slack for the design. By default, the tool will show the top 10 worst negative slack paths. Each path in the table has detailed information that shows the combination path between one flip-flop to another. Breaking these long combinational paths will be required to address the timing issues. So you need to analyze these paths and reason where they are coming from and map these paths back to the user's C code. Using both these paths and the resources table presented earlier can help in determining and correlating the path back to your source code.

In the figure below, you can see that the top 10 negative slack paths in the **Place & Route** report actually have higher logic levels (9) as compared to after **RTL Synthesis** (5), and the max fanout also got worse ($64 \rightarrow 9366$). This clearly shows how congestion in the design is causing high logic levels and higher fanouts which in turn causes issues for meeting timing. Using such clues, you can modify your design to remove some of this congestion either by rewriting the C code or making some different design decisions with respect to BRAM/LUTRAM/URAM resource choices.

Figure 126: RTL Synthesis Timing Paths

RTL Synthesis Timing Paths	
Worst Negative Slack:	-1.151
Total Negative Slack:	-13681.002
Max levels:	5
Max fanout:	64
Full Timing Report:	verilog/report/lidpc_decoder_kernel_timing_synth.rpt
Name	Value
▶ Path 1	slack=-1.151 levels=5 fanout=64
▶ Path 2	slack=-1.151 levels=5 fanout=64
▶ Path 3	slack=-1.151 levels=5 fanout=64
▶ Path 4	slack=-1.151 levels=5 fanout=64
▶ Path 5	slack=-1.151 levels=5 fanout=64
▶ Path 6	slack=-1.151 levels=5 fanout=64
▶ Path 7	slack=-1.151 levels=5 fanout=64
▶ Path 8	slack=-1.151 levels=5 fanout=64
▶ Path 9	slack=-1.151 levels=5 fanout=64
▶ Path 10	slack=-1.151 levels=5 fanout=64

Place & Route Timing Paths	
Worst Negative Slack:	-1.936
Total Negative Slack:	-110607.625
Max levels:	9
Max fanout:	9366
Full Timing Report:	verilog/report/lidpc_decoder_kernel_timing_routed.rpt
Name	Value
▶ Path 1	slack=-1.936 levels=6 fanout=176
▶ Path 2	slack=-1.936 levels=2 fanout=1690
▶ Path 3	slack=-1.936 levels=1 fanout=1689
▶ Path 4	slack=-1.936 levels=9 fanout=12
▶ Path 5	slack=-1.936 levels=9 fanout=12
▶ Path 6	slack=-1.936 levels=6 fanout=176
▶ Path 7	slack=-1.936 levels=9 fanout=10
▶ Path 8	slack=-1.936 levels=1 fanout=1690
▶ Path 9	slack=-1.936 levels=1 fanout=9366
▶ Path 10	slack=-1.936 levels=1 fanout=9366

Output of RTL Export

Vitis HLS writes to the `impl` folder of the active solution folder when you run the **Export RTL** command.

The output files and folders include the following:

- `component.xml`: The IP component file that defines the interfaces and architecture.
- `export.zip`: The zip archive of the IP and its contents. The zip file can be directly added to the Vivado IP catalog.
- `export.xo`: The compiled kernel object for use in the Vitis application acceleration development flow.
- `impl/ip`: The IP contents unzipped.
- `impl/ip/example`: A folder with a Tcl script used to generate the packaged IP, and a shell script to export the IP.
- `impl/report`: The report for the synthesized, or placed and routed IP is written to this folder.
- `impl/verilog`: Contains the Verilog format RTL output files.
- `impl/vhdl`: Contains the VHDL format RTL output files.



TIP: If the Vivado synthesis or Vivado synthesis, place, and route options are selected, Vivado synthesis and implementation are performed in the Verilog or VHDL folders. In this case the folder includes a `project.xpr` file that can be opened in the Vivado Design Suite.



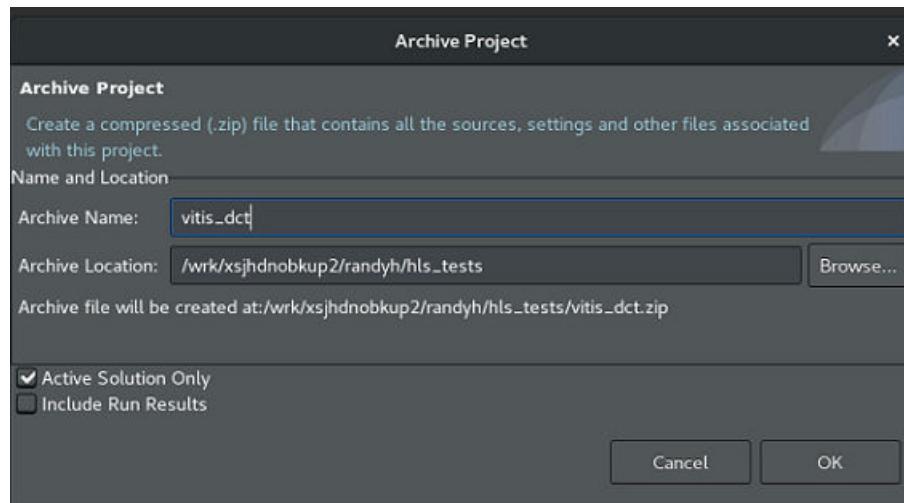
IMPORTANT! Xilinx does not recommend directly using the files in the `verilog` or `vhdl` folders for your own RTL synthesis project. Instead, Xilinx recommends using the packaged IP output files. Please carefully read the text that immediately follows this note.

In cases where Vitis HLS uses Xilinx IP in the design, such as with floating point designs, the RTL directory includes a script to create the IP during RTL synthesis. If the files in the `verilog` or `vhdl` folders are copied out and used for RTL synthesis, it is your responsibility to correctly use any script files present in those folders. If the package IP is used, this process is performed automatically by the design Xilinx tools. If C/RTL co-simulation has been executed in Vitis HLS, the Vivado project also contains an RTL test bench, and the design can be simulated.

Archiving the Project

After the project has been completed, and the RTL exported, you can archive the Vitis HLS project to an industry-standard Zip file. Select the **File→Archive Project** menu command to open the Archive Project dialog box as shown below.

Figure 127: Archive Project Dialog Box



The Archive Project dialog box features the following settings:

- **Archive Name:** Specifies the name of the archive file to create.
- **Active Solution Only:** This is selected by default. Disable this option to include all solutions from the current project.
- **Include Run Results:** By default only the source files and constraints will be included in the archive file. Enable this option to also include the results of simulation and synthesis in the archive file.

Running Vitis HLS from the Command Line

Vitis™ HLS can be run from the GUI, as previously discussed, interactively from the command line, or in batch mode from a Tcl script. This section discusses running the tool interactively, or in batch mode.

Running Vitis HLS Interactively

You can launch Vitis HLS using the `-i` option to open the tool in interactive mode.

```
$ vitis_hls -i
```

When running interactively, the tool displays a command line prompt for you to enter commands:

```
vitis_hls>
```

You can use the `help` command to get a list of commands that you can use in this mode, as described in [Section IV: Vitis HLS Command Reference](#).

```
vitis_hls> help
```

Help for any individual command is provided by using the command name as an option to the `help` command. For example, help for the `add_files` command can be returned with:

```
vitis_hls> help add_files
```

Vitis HLS also supports an auto-complete feature by pressing the tab key at any point when entering commands. The tool displays the possible matches based on typed characters to complete the command, or command option. Entering more characters improves the filtering of the possible matches.

Type the `exit` or `quit` command to quit Vitis HLS.



TIP: On the Windows OS, the Vitis HLS command prompt is implemented using the Minimalist GNU for Windows (minGW) environment, that supports both standard Windows DOS commands, and a subset of Linux commands. For example, both the Linux `ls` command and the DOS `dir` command is used to list the contents of a directory. Linux paths in a Makefile expand into minGW paths. Therefore, in all Makefile files you must put the path name in quotes to prevent any path substitutions, for example `FOO := " :/"`.

Running Vitis HLS in Batch Mode

Vitis™ HLS can also be run in batch mode, by specifying a Tcl script for the tool to run when launching as follows:

```
vitis_hls -f tcl_script.tcl
```

Commands embedded in the specified Tcl script are executed in the specified sequence. If the Tcl script includes the `exit` or `quit` command, then the tool exits at that point, completing the batch process. If the Tcl script does not end with the `exit` command, Vitis HLS returns to the command prompt, letting you continue in interactive mode.

All of the Tcl commands used when creating a project in the GUI are written to the `solution/script.tcl` file within the project. You can use this script as a starting point for developing your own batch scripts. An example script is provided below:

```
open_project dct
set_top dct
add_files ..../dct_src/dct.cpp
add_files -tb ..../dct_src/out.golden.dat -cflags "-Wno-unknown-pragmas" -
csimflags "-Wno-unknown-pragmas"
add_files -tb ..../dct_src/in.dat -cflags "-Wno-unknown-pragmas" -csimflags "-
Wno-unknown-pragmas"
add_files -tb ..../dct_src/dct_test.cpp -cflags "-Wno-unknown-pragmas" -
csimflags "-Wno-unknown-pragmas"
open_solution "solution1" -flow_target vitis
set_part {xcvu11p-flga2577-1-e}
create_clock -period 10 -name default
source "./dct/solution1/directives.tcl"
csim_design
csynth_design
cosim_design
export_design -format ip_catalog
```

When opening a legacy Vitis™ HLS project in Vitis HLS, you must specify the `-upgrade` or `-reset` option.

- `-upgrade` will perform conversion of a Vivado HLS project to a Vitis HLS project.
- `-reset` will restore the project to its initial state.



TIP: The `open_project` command will return an error when opening a Vitis HLS project unless the `-upgrade` or `-reset` option is used.

Vitis HLS Command Reference

This section contains the following chapters:

- [vitis_hls Command](#)
- [Project Commands](#)
- [Configuration Commands](#)
- [Optimization Directives](#)
- [HLS Pragmas](#)

vitis_hls Command

The `vitis_hls` command opens in the Vitis™ HLS integrated design environment (IDE) mode by default. However, you can also run `vitis_hls` interactively, specifying commands from the command line, or specifying Tcl scripts for the tool to run in batch mode.

To see what options are available for use with `vitis_hls` you can use the `-help` option:

```
vitis_hls -help
```

The `vitis_hls` command supports the following options:

- `-f <string>`: Start Vitis HLS by running a specified Tcl script. After the Tcl script is ended the tool remains open in interactive mode as described below, unless `quit` or `exit` has been called from the script.



TIP: When running the tool in interactive mode, you can type the `help` command to display a list of available Vitis HLS commands:

```
vitis_hls> help
```

- `-i`: This option invokes the tool in interactive mode with a command prompt, ready to receive any Vitis HLS command, as documented in [Project Commands](#), [Configuration Commands](#), or [Optimization Directives](#).
- `-l <string>`: Defines the name and location of the Vitis HLS log file. By default the tool creates a log file called `vitis_hls.log` in the directory from which Vitis HLS was launched.
- `-n | -nosplash`: Do not show the splash screen when starting the GUI.
- `-p`: Open an existing project in IDE mode. Specify a project folder or Tcl file to open the project when you are launching the tool. If a Tcl file is specified it will be automatically opened in the IDE through `open_tcl_project`.
- `-terse`: Filter `stdout` commands to only show status INFO and WARNING messages. The log file will contain all messages.
- `-version`: Return the version of Vitis HLS being used.

The following example launches Vitis HLS in command-line interactive (CLI) mode:

```
vitis_hls -i
```

hls_init.tcl

When you start Vitis™ HLS the tool looks for a Tcl initialization script in the following location:

1. User Specific: In a local user directory, for all versions of the tool:

- On Windows: %APPDATA%/Xilinx/HLS_init.tcl
- On Linux: \$HOME/.Xilinx/HLS_init.tcl

The `HLS_init.tcl` lets you use Vitis™ HLS commands to initialize the tool prior to opening a project.



TIP: There is no `HLS_init.tcl` script in the software installation. You must create one if needed.

Project Commands

Project commands are Vitis™ HLS commands that let you create and manage projects and solutions. The commands must be run in the interactive mode, `vitis_hls -i`, or can be run as a script using the `-f` option as described in [vitis_hls Command](#).

The features of these commands are also available through the Vitis HLS GUI when performing specific functions as described in sections like [Creating a New Vitis HLS Project](#) and [C/RTL Co-Simulation in Vitis HLS](#).



TIP: When running the commands through the GUI, the Tcl commands are added to a script of your project written to `solution/constraints/script.tcl`.

add_files

Description

Adds design source files to the current project.

The tool searches the current directory for any header files included in the design source. To use header files stored in other directories, use the `-cflags` option to include those directories to the search path.

Syntax

```
add_files [OPTIONS] <src_files>
```

- `<src_files>` lists one or more supported source files.

Options

- `-blackbox <file_name.json>`: Specify the JSON file to be used for RTL blackbox. The information in this file is used by the HLS compiler during synthesizing and running C/C++ and co-simulation. See [Adding RTL Blackbox Functions](#) for more information.
- `-cflags <string>`: A string with any GCC compilation options.

- **-csimflags <string>**: A string with any desired simulation compilation options. Flags specified with this option are only applied to simulation compilation, which includes C/C++ simulation and RTL co-simulation, not synthesis compilation. This option does not impact the **-cflags** option.
- **-tb**: Specifies any files used as part of the design test bench. These files are not synthesized. They are used when simulation is run by the `csim_design` or `cosim_design` commands.

Do not use the **-tb** option when adding source files for the design. Use separate `add_files` commands to add design files and simulation files.

Examples

Add three design files to the project.

```
add_files a.cpp
add_files b.cpp
add_files c.cpp
```

Add multiple files with a single command line.

```
add_files "a.cpp b.cpp c.cpp"
```

Use the **-tb** option to add test bench files to the project. This example adds multiple files with a single command, including:

- The test bench `a_test.cpp`
- All data files read by the test bench:
 - `input_stimuli.dat`
 - `out.gold.dat`

```
add_files -tb "a_test.cpp input_stimuli.dat out.gold.dat"
```

If the test bench data files in the previous example are stored in a separate directory (for example `test_data`), the directory can be added to the project in place of the individual data files.

```
add_files -tb a_test.cpp
add_files -tb test_data
```

cat_ini

Description

Concatenate one or more INI files into a single file and output the results to stdout or to a specified file. The resulting INI can be read into the Vitis HLS tool using the `apply_ini` command.

Syntax

```
cat_ini { <ini_file1> <ini_file2> ... } [OPTIONS]
```

- { <ini_file1> <ini_file2> ... } specifies one or more INI files to read into the Vitis™ HLS tool and concatenate into a single file. The braces "{}" are required to group multiple inputs.

Options

- `-exclude <string>`: Specify INI section glob patterns to exclude when writing the file.
- `-include <string>`:

Specify glob patterns found in the INI files to include in the output file. Anything not included will be excluded.



TIP: `-exclude` and `-include` are mutually exclusive, and cannot be used together.

- `-out <file name>`: Specifies the INI file name to write as output. When not specified, the output is written to stdout.
- `-quiet <true | false>`: Suppress all warning and information messages when writing the INI file. Note that errors will still be returned. The default value is false.
- `-show <true | false>`: Show verbose messages when writing the INI file. The default value is false.

Examples

This example joins the specified INI files and returns them to stdout:

```
cat_ini {./run.ini ./test.ini}
```

close_project

Description

Closes the current project. The project is no longer active in the Vitis HLS session.

The `close_project` command:

- Prevents you from entering any project-specific or solution-specific commands.
- Is not required. Opening or creating a new project closes the current project.

Syntax

```
close_project
```

Options

This command has no options.

Examples

```
close_project
```

- Closes the current project.
 - Saves all results.
-

close_solution

Description

Closes the current solution. The current solution is no longer active in the Vitis HLS session.

The `close_solution` command does the following:

- Prevents you from entering any solution-specific commands.
- Is not required. Opening or creating a new solution closes the current solution.

Syntax

```
close_solution
```

Options

This command has no options.

Examples

```
close_solution
```

- Closes the current solution.
- Saves all results.

cosim_design

Description

Executes post-synthesis co-simulation of the synthesized RTL with the original C/C++-based test bench.



TIP: To specify the files for the test bench run the following command:

```
add_files -tb
```

The simulation results are written to the `sim/Verilog` or `sim/VHDL` folder of the active solution, depending on the setting of the `-rtl` option.

Syntax

```
cosim_design [OPTIONS]
```

Options

- `-O`: Enables optimized compilation of the C/C++ test bench and RTL wrapper. This increases compilation time, but results in better runtime performance.
- `-argv <string>`: The `<string>` is passed onto the main C/C++ function.

Specifies an argument list for the behavioral test bench.

- `-compiled_library_dir <string>`: Specifies the compiled library directory during simulation with third-party simulators. The `<string>` is the path name to the compiled library directory. The library must be compiled ahead of time using the `compile_simlib` command as explained in the *Vivado Design Suite Tcl Command Reference Guide* ([UG835](#)).
- `-coverage`: Enables the coverage feature during simulation with the VCS simulator.

- **-disable_deadlock_detection:** Disables the deadlock detection feature in co-simulation.
- **-disable_dependency_check:** Disables dependency checks when running co-simulation.
- **-enable_dataflow_profiling:** This option turns on the dataflow channel profiling to track channel sizes during co-simulation.
- **-enable_fifo_sizing:** This option turns on automatic FIFO channel size tuning for dataflow profiling during co-simulation.
- **-hwemu_trace_dir <kernel_name>/<instance_name>:** Specifies the location of test vectors generated during hardware emulation to be used during co-simulation. The test vectors are generated by the `config_export -cosim_trace_generation` command. The argument lets you specify the kernel and instance name of the Vitis kernel in the hardware emulation simulation results to locate the test vectors.
- **-ldflags <string>:** Specifies the options passed to the linker for co-simulation.

This option is typically used to pass include path information or library information for the C/C++ test bench.

- **-mflags <string>:** Specifies options required for simulation.
- **-random_stall:** Enable random stalling of top-level interfaces during co-simulation.
- **-rtl [verilog | vhdl]:** Specifies which RTL language to use for C/RTL co-simulation. The default is Verilog.
- **-setup:** Creates all simulation files created in the `sim/<HDL>` directory of the active solution. The simulation is not executed, but can be run later from a command shell.
- **-stable_axilite_update:** Enable `s_axilite` to configure registers which are stable compared with the prior transaction.
- **-tool [auto | vcs | modelsim | riviera | isim | xsim | ncsim | xceilum]:** Specifies the simulator to use to co-simulate the RTL with the C/C++ test bench. The Vivado® simulator (`xsim`) is the default, unless otherwise specified.
- **-trace_level [*none* | all | port | port_hier]:** Determines the level of waveform trace data to save during C/RTL co-simulation.
 - `none` does not save trace data. This is the default.
 - `all` results in all port and signal waveforms being saved to the trace file.
 - `port` only saves waveform traces for the top-level ports.
 - `port_hier` save the trace information for all ports in the design hierarchy.

The trace file is saved in the `sim/Verilog` or `sim/VHDL` folder of the current solution when the simulation executes, depending on the selection used with the `-rtl` option.

- **-user_stall <string>**: Specifies the JSON stall file to be used during co-simulation. The stall file can be generated using the `cosim_stall` command.
- **-wave_debug**: Opens the Vivado simulator GUI to view waveforms and simulation results. Enables waveform viewing of all processes in the generated RTL, as in the dataflow and sequential processes. This option is only supported when using Vivado simulator for co-simulation by setting `-tool xsim`. See [Viewing Simulation Waveforms](#) for more information.

Examples

Performs verification using the Vivado simulator:

```
cosim_design
```

Uses the VCS simulator to verify the Verilog RTL and enable saving of the waveform trace file:

```
cosim_design -tool VCS -rtl verilog -coverage -trace_level all
```

Verifies the VHDL RTL using ModelSim. Values 5 and 1 are passed to the test bench function and used in the RTL verification:

```
cosim_design -tool modelsim -rtl vhdl -argv "5 1"
```

cosim_stall

Description

Command for development of the co-simulation stall file in JSON format.

Syntax

```
cosim_stall [OPTIONS]
```

Options

- **-check <string>**: Specify the JSON format stall file to use when running co-simulation.
- **-generate <string>**: Generate a JSON stall file to be used during co-simulation.
- **-list**: List all ports which can apply stall during co-simulation. This option returns a list of ports based on the current design.

Examples

The following example generates the specified stall file, and then specifies the file for use during co-simulation:

```
cosim_stall -generate my_cosim_stall.json  
cosim_stall -check my_cosim_stall.json
```

create_clock

Description

Creates a virtual clock for the current solution.

The command can be executed only in the context of an active solution. The clock period is a constraint that drives optimization (chaining as many operations as feasible in the given clock period).

C and C++ designs support only a single clock.

Syntax

```
create_clock -period <number> [OPTIONS]
```

Options

- **-name <string>**: Specifies the clock name. If no name is given, a default name is used.
- **-period <number>**: Specifies the clock period in ns or MHz.
 - If no units are specified, ns is assumed.
 - If no period is specified, a default period of 10 ns is used.

Examples

Specifies a clock period of 50 ns:

```
create_clock -period 50
```

Uses the default name and period of 10 ns to specify the clock:

```
create_clock
```

Specifies clock frequency in MHz:

```
create_clock -period 100MHz
```

csim_design

Description

Compiles and runs pre-synthesis C/C++ simulation using the provided C/C++ test bench.



TIP: To specify the files for the test bench run the following command:

```
add_files -tb
```

The simulation results are written to the `csim` folder inside the active solution.

Syntax

```
csim_design [OPTIONS]
```

Options

- `-O`: Enables optimized compilation of the C/C++ test bench. This increases compilation time, but results in better runtime performance.
- `-argv <string>`: Specifies the argument list for the behavioral test bench. The `<string>` is passed onto the `main()` C/C++ function of the test bench.
- `-clean`: Enables a clean build. Without this option, `csim_design` compiles incrementally.
- `-ldflags <string>`: Specifies the options passed to the linker for simulation. This option is typically used to pass include path information or library information for the C/C++ test bench.
- `-mflags <string>`: Specifies options required for simulation.
- `-profile`: Enable the creation of the [Pre-Synthesis Control Flow](#).
- `-setup`: When this option is specified, the simulation binary will be created in the `csim` directory of the active solution, but simulation will not be executed. Simulation can be launched later from the compiled executable.

Examples

Compiles and runs C/C++ simulation:

```
csim_design
```

Compiles source design and test bench to generate the simulation binary. Does not execute the simulation binary:

```
csim_design -O -setup
```



TIP: To run the simulation, execute `run.sh` in a command terminal, from the `csim/build` directory of the active solution.

csynth_design

Description

Synthesizes the Vitis HLS project for the active solution.

The command can be executed only in the context of an active solution. The elaborated design in the database is scheduled and mapped onto RTL, based on any constraints that are set.

Syntax

```
csynth_design [OPTIONS]
```

Options

- `-dump_cfg`: Write a pre-synthesis control flow graph (CFG).
- `-dump_post_cfg`: Write a post-synthesis control flow graph (CFG).
- `-synthesis_check`: Runs a pre-synthesis design rule check, but does not generate RTL.

Examples

Runs Vitis HLS synthesis on the top-level design.

```
csynth_design
```

delete_project

Description

Deletes the directory associated with the project.

The `delete_project` command checks the corresponding project directory `<project>` to ensure that it is a valid Vitis HLS project before deleting it. If the specified project directory does not exist in the current work directory, the command has no effect.

Syntax

```
delete_project <project>
```

- <project> is the project name.

Options

This command has no options.

Examples

Deletes the Project_1 by removing the directory and all its contents.

```
delete_project Project_1
```

delete_solution

Description

Removes a solution from an active project, and deletes the <solution> sub-directory from the project directory.

If the solution does not exist in the project directory, the command has no effect.

Syntax

```
delete_solution <solution>
```

- <solution> is the solution to be deleted.

Options

This command has no options.

Examples

Deletes solution solution1 from the active project by deleting the sub-directory.

```
delete_solution solution1
```

enable_beta_device

Description

Enables specified beta access devices in the Vitis HLS tool set.

Syntax

```
enable_beta_device <pattern>
```

- **<pattern>** Specifies a pattern matching the beta devices to enable.

Options

This command has no options.

Examples

The following example enables all beta devices in the release:

```
enable_beta_device *
```

export_design

Description

Exports and packages the generated RTL code as a packaged IP for use in the Vivado Design Suite, or as a compiled Vitis kernel object (`.xo`) for the Vitis application acceleration development flow.

Supported formats include:

- Vivado IP for inclusion in the IP catalog.
- Vitis application acceleration kernel (`.xo`).
- Synthesized or implemented design checkpoint (DCP) format.
- Vivado IP and ZIP archive for use in the System Generator for DSP tool.

The packaged project is written to the `solution/impl` folder of the active solution.

Syntax

```
export_design [OPTIONS]
```

Options

- **-description <string>**: Provides a description for the catalog entry for the generated IP, used when packaging the IP.
- **-display_name <string>**: Provides a display name for the catalog entry for the generated IP, used when packaging the IP.
- **-flow (syn | impl)**: Obtains more accurate timing and resource usage data for the generated RTL using Vivado synthesis and implementation. The option `syn` performs RTL synthesis. The option `impl` performs both RTL synthesis and implementation, including a detailed place and route of the RTL netlist. In the Vitis HLS IDE, these options appear as check boxes labeled **Vivado Synthesis** and **Vivado Synthesis, place and route stage**.
- **-format (ip_catalog | xo | syn_dcp | sysgen)**: Specifies the format to package the IP. The supported formats are:
 - **ip_catalog**: A format suitable for adding to the Xilinx IP catalog.
 - **ip_catalog**: A format suitable for adding to the Xilinx IP catalog.
 - **xo**: A format accepted by the `v++` compiler for linking in the Vitis application acceleration flow.
 - **syn_dcp**: Synthesized checkpoint file for Vivado Design Suite. If this option is used, RTL synthesis is automatically executed. Vivado implementation can be optionally added.
 - **sysgen**: Generate a Vivado IP and .zip archive for use in System Generator.
- **-ipname <string>**: Provides the name component of the Vendor:Library:Name:Version (VLSI) identifier for generated IP.
- **-library <string>**: Provides the library component of the Vendor:Library:Name:Version (VLSI) identifier for generated IP.
- **-output <string>**: Specifies the output location of the generated IP, XO, or DCP files. The file is written to the `solution/impl` folder of the current project if no output path is specified.
- **-rtl (verilog | VHDL)**: Specifies which HDL is used when the `-flow` option is executed. If not specified, Verilog is the default language for the Vivado synthesized netlist.
- **-taxonomy <string>**: Specifies the taxonomy for the catalog entry for the generated IP, used when packaging the IP.
- **-vendor <string>**: Provides the vendor component of the Vendor:Library:Name:Version (VLSI) identifier for generated IP.
- **-version <string>**: Provides the version component of the Vendor:Library:Name:Version (VLSI) identifier for generated IP.

Examples

Exports RTL for the Vitis application acceleration flow:

```
export_design -format xo
```

Exports the RTL as VHDL code in the Vivado IP catalog format. The VHDL is synthesized in Vivado synthesis tool to obtain better timing and usage data:

```
export_design -rtl vhdl -format ip_catalog -flow syn
```

get_clock_period

Description

This command returns the clock period for specified clock objects, or returns the default clock period for the active solution.

Syntax

```
get_clock_period [OPTIONS]
```

Options

- **-default**: Return the default period if the clock period is not specified.
- **-name <string>**: Get the clock period for the specified clock.
- **-ns**: Return the clock period in nanoseconds (ns). By default Vitis HLS returns the clock period in the same units as it was specified.

Examples

The following example creates a clock, ap_clk, and specifies the clock period in MHz. Then it gets the clock period for the clock as ns:

```
create_clock -name ap_clk -period 200MHz
get_clock_period -name ap_clk -ns
```

get_clock_uncertainty

Description

This command returns the clock uncertainty for specified clock, or returns the default clock uncertainty for the active solution.

Syntax

```
get_clock_uncertainty [clock_name]
```

- <clock_name> indicates the clock to get the uncertainty for.

Options

- **-default (true | false):**

Return the default uncertainty value if it has not been set by user. If `true` then the default uncertainty is returned, unless set by the user. If `false`, then the default is not returned.

Examples

The following example gets the clock uncertainty for the specified clock:

```
get_clock_uncertainty clk1
```

get_files

Description

This command gets the files that have been added to the active solution.

Syntax

```
get_files [OPTIONS]
```

Options

- **-cflags:** Return any compiler flags specified with the files.
- **-csimflags:** Return any C simulation flags specified with the files.

- **-fullpath**: Return the full path of the files.
- **-tb**: Return only the files that were added as part of the test bench (added with the **-tb** option).

Examples

The following example gets the added test bench files from the current solution, and returns the full path for the files:

```
get_files -tb -fullpath
```

get_part

Description

This command returns the Xilinx device used in the active solution.

Syntax

```
get_part
```

Options

There are no options for this command.

Examples

The following example returns the part used in the active solution:

```
get_part
```

get_project

Description

This command gets information for the currently opened project.

Syntax

```
get_project [OPTIONS]
```

Options

- **-directory:** Return the full path to the project directory.
- **-name:** Return the project name.
- **-solutions:** Return a list of all the solution names in the project.

Examples

The following example gets the full path for the current project:

```
get_project -directory
```

get_solution

Description

This command returns information related to the active solution.

Syntax

```
get_solution [OPTIONS]
```

Options

- **-directory:** Returns the full path to the active solution.
- **-flow_target:** Returns the flow target for the active solution.
- **-json:** Return the absolute path to the solution meta-data JSON file
- **-name:** Returns the solution name.

Examples

The following example returns the flow target for the active solution:

```
get_solution -flow_target
```

get_top

Description

This command returns the name of the top-level function for the open Vitis HLS project.

Syntax

```
get_top
```

Options

There are no options for this command.

Examples

The following example returns the top-level function for the open project:

```
get_top
```

help

Description

- When specified without a command name, the `help` command lists all Vitis HLS Tcl commands.
- When used with a Vitis HLS Tcl command as an argument, the `help` command returns details of the specified command.



TIP: For recognized Vitis HLS commands, auto-completion using the tab key is available when entering the command name.

Syntax

```
help <cmd>
```

- `<cmd>` specifies a command name to return the help for. If no command is specified, a list of all Vitis HLS commands will be returned.

Options

This command has no options.

Examples

Displays help for all commands and directives:

```
help
```

Displays help for the `add_files` command:

```
help add_files
```

list_part

Description

This command returns names of supported device families, parts, or boards.

If no argument is provided, the command will return all supported part families. To return specific parts of a family, provide the family name as an argument.

Syntax

```
list_part [name] [OPTIONS]
```

- `<family>` specifies a device family to return the specific devices of. If no `<family>` is specified, the `list_part` command returns a list of available device families.

Options

- `-name <string>`: Family, part, board name, or glob pattern.
- `-board[=false|true]`: Returns list of board names instead of part names.
- `-clock_regions`: Return a list of clock regions for the specified part. Must be specified with the `-name` command.
- `-slr_pblocks`: Return the SLR pblock dictionary for the specified part. Must be specified with the `-name` command.

Examples

Returns all supported device families.

```
list_part
```

Returns the clock regions of the current part in the active solution.

```
list_part -name [get_part] -clock_regions
```

open_project

Description

Opens an existing project, or creates a new one if the specified project does not exist.



IMPORTANT! In Vitis HLS, the `open_project` command returns an error when opening a Vivado HLS project, unless the `-upgrade` or `-reset` option is used.

There can only be one active project in a Vitis HLS session. To close a project:

- Use the `close_project` command, or
- Open or create another project with the `open_project` or `open_tcl_project` commands.

Use the `delete_project` command to completely delete the project directory (removing it from the disk) and any solutions associated with it.

Syntax

```
open_project [OPTIONS] <name>
```

- `<name>` specifies the project name.

Options

- **-reset:**
 - Resets the project by removing any data that already exists in the project.
 - Removes any previous project information on design source files, header file search paths, and the top-level function. The associated solution directories and files are kept, but might now have invalid results.
- **RECOMMENDED:** Use the `-reset` option when executing Vitis HLS with Tcl scripts. Otherwise, each new `add_files` command adds additional files to the existing data.
- **-upgrade:** Upgrade a Vivado HLS project to Vitis HLS.

Examples

Opens an existing project named `Project_1`, or creates a new one if it does not exist:

```
open_project Project_1
```

Opens a project and removes any existing data:

```
open_project -reset Project_2
```

open_solution

Description

Opens an existing solution or creates a new one in the currently active project. There can only be one active solution at any given time in a Vitis HLS session. As described in [Vitis HLS Flow Overview](#), the solution targets either the Vivado IP flow, or the Vitis Kernel flow. The default flow is the Vivado IP flow if no flow target is specified.



IMPORTANT! Attempting to open or create a solution when there is no open project results in an error.

Each solution is managed in a sub-directory of the current project. A new solution is created if the specified solution does not exist in the open project. To close a solution:

- Run the `close_solution` command, or
- Open another solution with the `open_solution` command.

Use the `delete_solution` command to remove a solution from the project and delete the corresponding sub-directory.

Syntax

```
open_solution [OPTIONS] <name>
```

- `<name>` specifies the solution name.



TIP: You can specify both the project name and the solution name in order to use `open_solution` to open the project and solution in a single command: `open_solution dctProj/solution1`

Options

- `-flow_target [vitis | vivado]`:

- **vivado:** Configures the solution to run in support of the Vivado IP generation flow, requiring strict use of pragmas and directives, and exporting the results as Vivado IP. This is the default flow when `-flow_target` is not specified.
- **vitis:** Configures the solution for use in the Vitis application acceleration development flow. This configures the Vitis HLS tool to properly infer interfaces for the function arguments without the need to specify the INTERFACE pragma or directive, and to output the synthesized RTL code as a Vitis kernel object file (`.xo`).
- **-reset:**
 - Resets the solution data if the solution already exists. Any previous solution information on libraries, constraints, and directives is removed.
 - Also removes synthesis, verification, and implementation results.

Examples

Opens an existing solution named `Solution_1` in the open project, or creates a new solution if one with the specified name does not exist. The solution is configured for creating kernel objects (`.xo`) for use in the Vitis tool flow.

```
open_solution -flow_target vitis Solution_1
```

Opens and resets the specified solution in the open project. Removes any existing data from the solution.

```
open_solution -reset Solution_2
```

open_tcl_project

Description

Create a project by sourcing a Tcl file, but skipping all design commands in the Tcl script: `cosim_design`, `csynth_design`, and `csim_design`. This command only creates and configures the project from a Tcl script. This lets you create a project using Tcl scripts from existing projects without running simulation or synthesis.

There can only be one active project in a Vitis HLS session. To close a project:

- Use the `close_project` command, or
- Open or create another project with the `open_tcl_project` or `open_project` commands.

Use the `delete_project` command to completely delete the project directory (removing it from the disk) and any solutions associated it.

Syntax

```
open_tcl_project <tclfile>
```

- <tclfile> specifies the path and name of a Tcl script to use when creating a project.

Options

This command has no options.

Examples

Creates and opens a project from the specified Tcl script:

```
open_tcl_project run_hls.tcl
```

set_clock_uncertainty

Description

Sets a margin on the clock period defined by `create_clock`.

The margin of uncertainty is subtracted from the clock period to create an effective clock period. The clock uncertainty is defined in ns, or as a percentage of the clock period. The clock uncertainty defaults to 27% of the clock period.

Vitis HLS optimizes the design based on the effective clock period, providing a margin for downstream tools to account for logic synthesis and routing. The command can be executed only in the context of an active solution. Vitis HLS still uses the specified clock period in all output files for verification and implementation.

Syntax

```
set_clock_uncertainty <uncertainty> <clock_list>
```

- <uncertainty>: A value, specified in ns, representing how much of the clock period is used as a margin. The uncertainty can also be specified as a percentage of the clock period. The default uncertainty is 27% of the clock period.
- <clock_list>: A list of clocks to which the uncertainty is applied. If none is provided, it is applied to all clocks.

Options

This command has no options.

Examples

Specifies an uncertainty or margin of 0.5 ns on the clock. This effectively reduces the clock period that Vitis HLS can use by 0.5 ns.

```
set_clock_uncertainty 0.5
```

set_part

Description

Sets a target device, device family, or board for the current solution. The command can be executed only in the context of an active solution.



TIP: *Each solution in a project can target a separate device or device family.*

Syntax

```
set_part <device_specification>
```

- `<device_specification>` is a device specification that sets the target device for Vitis HLS synthesis and implementation.
- The device specification includes `<device>`, `<package>`, and `<speed_grade>` information.
- Specifying the `<device_family>` uses the default device for the device family.

Options

- `-board`: Specify the part as defined on a board.

Examples

The FPGA libraries provided with Vitis HLS can be added to the current solution by providing the device family name as shown below. In this case, the default device, package, and speed grade specified in the Vitis HLS FPGA library for the Virtex-7 device family are used.

```
set_part virtex7
```

The FPGA libraries provided with Vitis HLS can optionally specify the specific device with package and speed grade information.

```
set_part xc6vlx240tff1156-1
```

Specifies the part through the definition of a board.

```
set_part -board u200
```

set_top

Description

Defines the top-level function to be synthesized.



IMPORTANT! Any functions called from the top-level function will also become part of the HLS design.

Syntax

```
set_top <name>
```

- <name> is the function to be synthesized by HLS.

Options

This command has no options.

Examples

Sets the top-level function as foo.

```
set_top foo
```

Configuration Commands

The configuration commands let you configure the Vitis™ HLS tool to control the results of synthesis and simulation, specify defaults for pragmas and directives, and specify the outputs generated by default. The commands must be run in the interactive mode, `vitis_hls -i`, or can be run as a script using the `-f` option as described in [vitis_hls Command](#).

These configuration commands can also be set in the Vitis HLS IDE using the **Solution Settings** dialog box as described in [Setting Configuration Options](#).

config_array_partition

Description

Specifies the default behavior for array partitioning.

Syntax

```
config_array_partition [OPTIONS]
```

Options

- `-throughput_driven <off | auto>:`

Enable automatic partial and/or complete array partitioning.

- `auto` : Enable automatic array partitioning with smart trade-offs between area and throughput. This is the default value.
- `off` : Disable automatic array partitioning.

- `-complete_threshold <uint:4>:`

Sets the threshold for completely partitioning arrays. Arrays with fewer elements than the specified threshold will be completely partitioned into individual elements.

Examples

Partitions all arrays in the design, except global arrays, with less than 12 elements into individual elements.

```
config_array_partition -complete_threshold 12
```

config_compile

Description

Configures the default behavior of front-end compiling.

Syntax

```
config_compile [OPTIONS]
```

Options

- **-enable_auto_rewind[=true|false]**: When TRUE uses alternative HLS implementation of pipelined loops which enables automatic loop rewind. This accepts values of TRUE or FALSE. The default value is TRUE.
- **-ignore_long_run_time[=true|false]**: Do not report the "long run time" warning. This accepts values of TRUE or FALSE. The default value is FALSE.
- **-name_max_length <value>**: Specifies the maximum length of function names. If the length of the name is longer than the threshold, the last part of the name is truncated, and digits are added to make the name unique when required. The default is 256.
- **-no_signed_zeros[=true|false]**: Ignores the signedness of floating-point zero so that the compiler can perform aggressive optimizations on floating-point operations. This accepts values of TRUE or FALSE. The default value is FALSE.



IMPORTANT! Using this option might change the result of any floating point calculations and result in a mismatch in C/RTL co-simulation. Please ensure your test bench is tolerant of differences and checks for a margin of difference, not exact values.

- **-pipeline_flush_in_task <always | never | ii1>**: Specifies that pipelines will be flushing by default in `hls::tasks` to reduce the probability of deadlocks in C/RTL Co-simulation. This option is limited to pipelines that achieve an `II=1` with the default option of `ii1`. This default can be overridden using `always` to always enable flushing pipelines in either `hls::tasks` or dataflow, or can be completely disabled using `never`. For more information refer to [Flushing Pipelines and Pipeline Types](#).



IMPORTANT! Flushing pipelines are not compatible with the `rewind` option specified in the `PIPELINE` pragma or directive.

- `always`: Always make pipelines flushable in `hls::tasks` or dataflow regardless of II.
- `never`: Never make pipeline flushable unless specifically overridden by other directives or pragmas.
- `i11`: Make pipelines that achieve II=1 flushable in `hls::tasks`. This is the default setting.
- `-pipeline_loops <threshold>`: Specifies the lower limit used when automatically pipelining loops. The default is 64, causing Vitis HLS to automatically pipeline loops with a tripcount of 64, or greater.

If the option is applied, the innermost loop with a tripcount higher than the threshold is pipelined, or if the tripcount of the innermost loop is less than or equal to the threshold, the innermost loop is unrolled. This analysis is then repeated for the parent loop. If the innermost loop has no parent loop, the innermost loop is pipelined regardless of its tripcount.

- `-pipeline_style <stp | f1p | frp>`: Specifies the default type of pipeline used by Vitis HLS for the `PIPELINE` pragma or directive, or for loop pipelining due to the `-pipeline_loops` threshold specified above. For more information on pipeline styles, refer to [Flushing Pipelines and Pipeline Types](#).



IMPORTANT! This is a hint not a hard constraint. The tool checks design conditions for enabling pipelining. Some loops might not conform to a particular style and the tool reverts to the default style (stp) if necessary.

- `stp`: Stall pipeline. Runs only when input data is available otherwise it stalls. This is the default setting, and is the type of pipeline used by Vitis HLS for both loop and function pipelining. Use this when a flushable pipeline is not required. For example, when there are no performance or deadlock issue due to stalls.
- `f1p`: Flushable pipeline architecture: flushes when input data is not available then stalls waiting for new data.
- `frp`: Free-running, flushable pipeline. Runs even when input data is not available. Use this when you need better timing due to reduced pipeline control signal fanout, or when you need improved performance to avoid deadlocks. However, this pipeline style may consume more power, as the pipeline registers are clocked even if there is no data.
- `-pragma_strict_mode[=true|false]`: Enable error messages for misplaced or misused pragmas.
- `-pre_tcl <arg>`: Specify a TCL script to run prior to starting the `csynth_design` command.

- **-unsafe_math_optimizations [=true|false]**: Ignores the signedness of floating-point zero and enables associative floating-point operations so that compiler can perform aggressive optimizations on floating-point operations. This accepts values of TRUE or FALSE. The default value is FALSE.

Note: Using this option might change the result of any floating point calculations and result in a mismatch in C/RTL co-simulation. Please ensure your test bench is tolerant of differences and checks for a margin of difference, not exact values.

Examples

Pipeline the innermost loop with a tripcount higher than 30, or pipeline the parent loop of the innermost loop when its tripcount is less than or equal 30:

```
config_compile -pipeline_loops 30
```

Ignore the signedness of floating-point zero:

```
config_compile -no_signed_zeros
```

Ignore the signedness of floating-point zero and enable the associative floating-point operations:

```
config_compile -unsafe_math_optimizations
```

config_cosim

Description

Lets you configure the settings of the C/RTL Co-simulation command (`cosim_design`).

Syntax

```
config_cosim [OPTIONS]
```

Options

- **-O**: Enables optimized compilation of the C/C++ test bench and RTL wrapper. This increases compilation time, but results in better runtime performance.
- **-argv <string>**: The `<string>` is passed onto the main C/C++ function.

Specifies an argument list for the behavioral test bench.

- **-compiled_library_dir <string>**: Specifies the compiled library directory during simulation with third-party simulators. The <string> is the path name to the compiled library directory. The library must be compiled ahead of time using the `compile_simlib` command as explained in the *Vivado Design Suite Tcl Command Reference Guide* ([UG835](#)).
- **-coverage**: Enables the coverage feature during simulation with the VCS simulator.
- **-disable_deadlock_detection**: Disables the deadlock detection feature in co-simulation.
- **-disable_dependency_check**: Disables dependency checks when running co-simulation.
- **-enable_dataflow_profiling**: This option turns on the dataflow channel profiling to track channel sizes during co-simulation.
- **-enable_fifo_sizing**: This option turns on automatic FIFO channel size tuning for dataflow profiling during co-simulation.
- **-hwemu_trace_dir <kernel_name>/<instance_name>**: Specifies the location of test vectors generated during hardware emulation to be used during co-simulation. The test vectors are generated by the `config_export -cosim_trace_generation` command. The argument lets you specify the kernel and instance name of the Vitis kernel in the hardware emulation simulation results to locate the test vectors.
- **-ldflags <string>**: Specifies the options passed to the linker for co-simulation.

This option is typically used to pass include path information or library information for the C/C++ test bench.

- **-mflags <string>**: Specifies options required for simulation.
- **-random_stall**: Enable random stalling of top-level interfaces during co-simulation.
- **-rtl [verilog | vhdl]**: Specifies which RTL language to use for C/RTL co-simulation. The default is Verilog.
- **-setup**: Creates all simulation files created in the `sim/<HDL>` directory of the active solution. The simulation is not executed, but can be run later from a command shell.
- **-stable_axilite_update**: Enable `s_axilite` to configure registers which are stable compared with the prior transaction.
- **-tool [auto | vcs | modelsim | riviera | isim | xsim | ncsim | xceilum]**: Specifies the simulator to use to co-simulate the RTL with the C/C++ test bench. The Vivado® simulator (xsim) is the default, unless otherwise specified.
- **-trace_level [*none* | all | port | port_hier]**: Determines the level of waveform trace data to save during C/RTL co-simulation.
 - `none` does not save trace data. This is the default.
 - `all` results in all port and signal waveforms being saved to the trace file.

- `port` only saves waveform traces for the top-level ports.
- `port_hier` save the trace information for all ports in the design hierarchy.

The trace file is saved in the `sim/Verilog` or `sim/VHDL` folder of the current solution when the simulation executes, depending on the selection used with the `-rtl` option.

- `-user_stall <string>`: Specifies the JSON stall file to be used during co-simulation. The stall file can be generated using the `cosim_stall` command.
- `-wave_debug`: Opens the Vivado simulator GUI to view waveforms and simulation results. Enables waveform viewing of all processes in the generated RTL, as in the dataflow and sequential processes. This option is only supported when using Vivado simulator for co-simulation by setting `-tool xsim`. See [Viewing Simulation Waveforms](#) for more information.

config_csim

Description

Lets you configure the settings of the C simulation command (`csim_design`).

Syntax

```
config_csim [OPTIONS]
```

Options

- `-O`: Enables optimized compilation of the C/C++ test bench. This increases compilation time, but results in better runtime performance.
- `-argv <string>`: Specifies the argument list for the behavioral test bench. The `<string>` is passed onto the `main()` C/C++ function of the test bench.
- `-clean`: Enables a clean build. Without this option, `csim_design` compiles incrementally.
- `-ldflags <string>`: Specifies the options passed to the linker for simulation. This option is typically used to pass include path information or library information for the C/C++ test bench.
- `-mflags <string>`: Specifies options required for simulation.
- `-profile`: Enable the creation of the [Pre-Synthesis Control Flow](#).
- `-setup`: When this option is specified, the simulation binary will be created in the `csim` directory of the active solution, but simulation will not be executed. Simulation can be launched later from the compiled executable.

config_dataflow

Description

- Specifies the default behavior of dataflow pipelining (implemented by the `set_directive_dataflow` command).
- Allows you to specify the default channel memory type and depth.

Syntax

```
config_dataflow [OPTIONS]
```

Options

- default_channel [fifo | pingpong]:** By default, a RAM memory, configured in pingpong fashion, is used to buffer the data between functions or loops when dataflow pipelining is used. When streaming data is used (that is, the data is always read and written in consecutive order), a FIFO memory is more efficient and can be selected as the default memory type.



TIP: Set arrays to streaming using the `set_directive_stream` command to perform FIFO accesses.

- disable_fifo_sizing_opt:** Disable FIFO sizing optimizations that increase resource usage and may improve performance and reduce deadlocks.
- fifo_depth <integer>:** Specifies the default depth of the FIFOs. The default depth is 2.

This option has no effect when ping-pong memories are used. If not specified, the default depth is 2, or if this is an array converted into a FIFO, the default size is the size of the original array. In some cases, this might be too conservative and introduce FIFOs that are larger than necessary. Use this option when you know that the FIFOs are larger than required.



CAUTION! Be careful when using this option. Insufficient FIFO depth might lead to deadlock situations.

- override_user_fifo_depth <value>:**

Use the specified depth for every `hls::stream`, overriding any user settings.

Note: This is useful for checking if a deadlock is due to insufficient FIFO depths in the design. By setting it to a very large value (for example, the maximum depth printed by co-simulation at the end of simulation), if there is no deadlock, then you can use the FIFO depth profiling options of co-simulation and the GUI to find the minimum depth that ensures performance and avoids deadlocks.

- scalar_fifo_depth <integer>:** Specifies the minimum for scalar propagation FIFO.

Through scalar propagation, the compiler converts the scalar from C/C++ code into FIFOs. The minimal sizes of these FIFOs can be set with `-start_fifo_depth`. If this option is not provided, then the value of `-fifo_depth` is used.

- `-start_fifo_depth <integer>`: Specifies the minimum depth of start propagation FIFOs.

This option is only valid when the channel between the producer and consumer is a FIFO. This option uses the same default value as the `-fifo_depth` option, which is 2. Such FIFOs can sometimes cause deadlocks, in which case you can use this option to increase the depth of the FIFO.

- `-strict_mode [off | warning | error]`: Set the severity for messages related to dataflow canonical form.
- `-strict_stable_sync[=true|false]`: Force synchronization of stable ports with `ap_done`.
- `-task_level_fifo_depth <integer>`: Specifies the depth of the task level FIFO.

A FIFO is synchronized by `ap_ctrl_chain`. The write is the `ap_done` of the producer, the read is the `ap_ready` of the consumer. Like a PIPO in terms of synchronization, and like a FIFO in terms of access.

Examples

Changes the default channel from ping-pong memories to FIFOs:

```
config_dataflow -default_channel fifo
```

Changes the default channel from ping-pong memories to FIFOs with a depth of 6:

```
config_dataflow -default_channel fifo -fifo_depth 6
```



CAUTION! If the design implementation requires a FIFO with greater than six elements, this setting results in a design that fails RTL verification. Be careful when using this option, because it is a user override.

To find the cause of deadlocks, try to increase all the FIFO depths significantly, especially those that are reported by C/RTL co-simulation. If the deadlock disappears with a large "N", then it is due to insufficient FIFO depths. You can test this as follows:

```
config_dataflow -fifo_depth N -start_fifo_depth N -scalar_fifo_depth N -task_level_fifo_depth N
```

config_debug

Generate HLS debug files used in the Vitis application acceleration development flow.

Description

Configures the default behavior of front-end compiling.

Syntax

```
config_debug [OPTIONS]
```

Options

- **-directory <path>**: Specifies the location of HLS debugging output. If not specified, location is set to solution/.debug.
- **-enable[=true|false]**: Enable generation of HLS debugging files used in Vitis debug flow.

Examples

The following example enables the debug files:

```
config_debug -enable true
```

config_export

Description

Configures options for [export_design](#) which can either run downstream tools or package a Vivado IP or Vitis compiled kernel object (.xo).

Syntax

```
config_export [OPTIONS]
```

Options

- **-cosim_trace_generation=<true | false>**: Generate test vectors during hardware emulation in the Vitis tool flow when the kernel is synthesized as a Vitis kernel, to be used during C/RTL Co-simulation in future iterations.

- **-description <string>**: Provides a description for the catalog entry for the generated IP, used when packaging the IP.
- **-display_name <string>**: Provides a display name for the catalog entry for the generated IP, used when packaging the IP.
- **-flow (none | syn | impl)**: Lets you obtain more accurate timing and resource usage data for the generated RTL using Vivado synthesis and implementation. The option `syn` performs RTL synthesis. The option `impl` performs both RTL synthesis and implementation, including a detailed place and route of the RTL netlist. The default option is `none` which does not run either synthesis or implementation.



TIP: In the Vitis HLS IDE, these options appear as check boxes labeled **Vivado Synthesis** and **Vivado Synthesis, place and route stage**.

- **-format (ip_catalog | xo | syn_dcp | sysgen)**: Specifies the format to package the IP. The supported formats are:
 - **ip_catalog**: A format suitable for adding to the Xilinx IP catalog.
 - **xo**: A format accepted by the v++ compiler for linking in the Vitis application acceleration flow.
 - **syn_dcp**: Synthesized checkpoint file for Vivado Design Suite. If this option is used, RTL synthesis is automatically executed. Vivado implementation can be optionally added.
 - **sysgen**: Generates a Vivado IP and .zip archive for use in System Generator.
- **-ip_xdc_file <arg>**: Specify an XDC file whose contents will be included in the packaged IP for use during implementation in the Vivado tool.
- **-ip_xdc_ooc_file <arg>**: Specify an out-of-context (OOC) XDC file whose contents will be included in packaged IP and used during out-of-context Vivado synthesis for the exported IP.
- **-ipname <string>**: Provides the name component of the <Vendor>:<Library>:<Name>:<Version> (VLSI Name Value) identifier for generated IP.
- **-library <string>**: Provides the library component of the <Vendor>:<Library>:<Name>:<Version> (VLSI Name Value) identifier for generated IP.
- **-output <string>**: Specifies the output location of the generated IP, .xo, or DCP files. The file is written to the `solution/impl` folder of the current project if no output path is specified.
- **-rtl (verilog | VHDL)**: Specifies which HDL is used when the `-flow` option is executed. If not specified, Verilog is the default language for the Vivado synthesized netlist.
- **-taxonomy <string>**: Specifies the taxonomy for the catalog entry for the generated IP, used when packaging the IP.

- **-vendor <string>**: Provides the vendor component of the <Vendor>:<Library>:<Name>:<Version> (VNV) identifier for generated IP.
- **-version <string>**: Provides the version component of the <Vendor>:<Library>:<Name>:<Version> (VNV) identifier for generated IP.
- **-vivado_clock <arg>**: Override the specified HLS clock constraint used in Vivado OOC run. This is only used for reporting purposes and will not apply to the exported IP.
- **-vivado_impl_strategy <string>**: Specifies Vivado implementation strategy name. The default name is 'default'.
- **-vivado_max_timing_paths <uint : 10>**: Specify the max number of timing paths to report when the timing is not met in the Vivado synthesis or implementation.
- **-vivado_optimization_level (0 | 1 | 2 | 3)**: Vivado optimization level. This option sets other `vivado_*` options. This only applies for report generation and will not apply to the exported IP. The default setting is 0.
- **-vivado_pblock <arg>**: Specify a PBLOCK range to use during implementation for reporting purposes. This will not apply to the exported IP.
- **-vivado_phys_opt (none | place | route | all)**: Specifies whether Vivado physical optimization should be run during Vivado implementation. Valid values are:
 - **none**: Do not run (default).
 - **place**: Run post-place.
 - **route**: Run post-route.
 - **all**: Run post-place and post-route.
- **-vivado_report_level (0 | 1 | 2)**: Specifies how many Vivado reports are generated, and does not apply to the exported IP. The valid values and the associated reports are:
 - 0: Post-synthesis utilization. Post-implementation utilization and timing.
 - 1: Post-synthesis utilization, timing, and analysis. Post-implementation utilization, timing, and analysis.
 - 2: Post-synthesis utilization, timing, analysis, and failfast. Post-implementation utilization, timing, and failfast. This is the default setting.
- **-vivado_synth_design_args <string>**: Specifies extra arguments to pass to the Vivado `synth_design` command. The default is `-directive sdx_optimization_effort_high`.
- **-vivado_synth_strategy <string>**: Specifies Vivado synth strategy name. The default strategy is "default".

Examples

The following example exports the Vitis .xo to the specified file:

```
export_design -description "Kernel Export" -display_name kernel_export \
-flow impl -format xo -output "tmp/hls_tests/kernel.xo"
```

config_interface

Description

Specifies the default interface options used to implement the RTL ports of each function during interface synthesis.

Syntax

```
config_interface [OPTIONS]
```

Options

- **-clock_enable[=true|false]**: Adds a clock-enable port (`ap_ce`) to the design. The default is false.
The clock enable prevents all clock operations when it is active-Low. It disables all sequential operations
- **-default_slave_interface [none | s_axilite]**: Enables the default for the slave interface as either `none`, which is the default for the Vivado IP flow, or as `s_axilite` which is the default for the Vitis Kernel flow, as described in [Vitis HLS Flow Overview](#).
- **-m_axi_addr64[=true|false]**: Globally enables 64-bit addressing for all `m_axi` ports in the design. By default, this is enabled for the Vitis flow, and otherwise disabled.
- **-m_axi_alignment_byte_size <size>**: Specifies the memory alignment boundary for `m_axi` interfaces provided as bitwidth. The `<size>` value must be a valid power of 2. A value of 0 is an invalid value. The default value is 64 when `open_solution -flow_target vitis`, and 1 when the `-flow_target=vivado`, aligning to a single byte.



IMPORTANT! *Burst behavior will be incorrect if pointers are not aligned at runtime.*

- **-m_axi_auto_max_ports[=true|false]**: If the option is `true`, all the `m_axi` interfaces that are not explicitly bundled, with INTERFACE pragmas or directives, will be mapped into individual interfaces, thus increasing the resource utilization (multiple adapters). The default is `false` and `m_axi` ports are bundled into a single interface.

- **-m_axi_buffer_impl [auto | lutram | uram | bram]**: Select the implementation for all `m_axi` internal buffers.
 - **auto**: Let the tool choose the implementation.
 - **lutram**: Specifies distributed RAM for the buffers.
 - **bram**: Use the Block RAM. This is the default setting.
 - **uram**: Use the UltraRAM.
- **-m_axi_conservative_mode=<true|false>**: This mode tells the `m_axi` not to issue a write request until the associated write data is entirely available (typically, buffered into the adapter or already emitted). It uses a buffer inside the MAXI adapter to store all the data for a burst (both in case of reading and in case of writing). This is enabled (`true`) by default, and may slightly increase write latency but can resolve deadlock due to concurrent requests (read or write) on the memory subsystem. This feature can be disabled by setting it to `false`.



TIP: *This mode can be safely set to `false` to save this internal buffering if the design implements buffering that is larger than the max write burst length using an alternative approach.*

- **-m_axi_flush_mode**: Configure the `m_axi` adapter to be flushable, writing or reading garbage data if a burst is interrupted due to pipeline blocking, missing data inputs when not in conservative mode, or missing output space. The default is `false`. This is enabled when the option is specified.
- **-m_axi_latency <latency>**: Globally specifies the expected latency of the `m_axi` interface, allowing the design to initiate a bus request a number of cycles (`latency`) before the read or write is expected. The default value is 64 when `open_solution -flow_target vitis`, and 0 when `-flow_target vivado`.
- **-m_axi_max_bitwidth <size>**: Specifies the maximum bitwidth for the `m_axi` interfaces data channel. The default is 1024 bits. The specified value must be a power-of-two, between 8 and 1024. Note that this decreases throughput if the actual accesses are bigger than the required interface, as they will be split into a multi-cycle burst of accesses.
- **-m_axi_max_read_burst_length <size>**: Specifies a global maximum number of data values read during a burst transfer for all `m_axi` interfaces. The default is 16.
- **-m_axi_max_widen_bitwidth <size>**: Automatic port width resizing to widen bursts for the `m_axi` interface, up to the chosen bitwidth. The specified value must be a power of 2 between 8 and 1024, and must align with the `-m_axi_alignment_size`. The default value is 512 when `open_solution -flow_target vitis`, and 0 when the `-flow_target vivado`.
- **-m_axi_max_write_burst_length <size>**: Specifies a global maximum number of data values written during a burst transfer for all `m_axi` interfaces. The default is 16.

- **-m_axi_min_bitwidth <size>**: Specifies the minimum bitwidth for the `m_axi` interfaces data channel. The default is 8 bits. The value must be a power of 2, between 8 and 1024. Note that this does not necessarily increase throughput if the actual accesses are smaller than the required interface.
- **-m_axi_num_read_outstanding <size>**: Specifies how many read requests can be made to the `m_axi` interface without a response, before the design stalls. The default value is 16. This implies internal storage in the design, and a FIFO of size:

```
num_read_outstanding*max_read_burst_length*word_size
```

- **-m_axi_num_write_outstanding <size>**: Specifies how many write requests can be made to the `m_axi` interface without a response, before the design stalls. The default value is 16. This implies internal storage in the design, and a FIFO of size:

```
num_write_outstanding*max_write_burst_length*word_size
```

- **-m_axi_offset [off | direct | slave]**: Globally controls the offset ports for all `m_axi` interfaces in the design.
 - **off**: No offset port is generated. This is the default value in the Vivado IP flow.
 - **direct**: Generates a scalar input offset port for directly passing the address offset into the IP through the offset port.
 - **slave**: Generates an offset port and automatically maps it to an AXI4-Lite slave. This is the default value.
- **-register_io [off | scalar_in | scalar_out | scalar_all]**: Globally enables registers for all inputs, all outputs, or all ports on the top function. The default is `off`.
- **-s_axilite_auto_restart_counter [0 | 1]**: Enables the auto-restart behavior for kernels. Use 0 to disable the auto-restart feature, which is the default, or use 1 to enable the feature. When enabled, the tool establishes the auto-restart bit in the `ap_ctrl_chain` control protocol for the `s_axilite` interface. For more information refer to *Continuously Running Kernels in Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#)).
- **-s_axilite_data64[=true|false]**: Set the data width for the `s_axilite` interface to 64 bits.
- **-s_axilite_interrupt_mode[=cor|tow]**: Specify the interrupt mode for `s_axilite` interface to be Clear on Read (`cor`) or Toggle on Write (`tow`). Clear on Read interrupt can be completed in a single transaction, while `tow` requires two. `Tow` is the default interrupt mode.
- **-s_axilite_mailbox [both | in | out]**: Enables the creation of a mailboxes for non-stream non-stable `s_axilite` arguments. The mailbox feature is used in the setting and management of never-ending kernels as described in *Continuously Running Kernels in Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#)). The argument values specify:

- **both** : Enable mailbox for input and output arguments
 - **in** : Enable mailbox for only input arguments
 - **out** : Enable mailbox for only output arguments
 - **none** : No mailbox created (default)
- **-s_axilite_status_regs [ecc | off]:**
Enables exposure of ECC error bits in the `s_axilite` register map via two clear-on-read (COR) counters per BRAM or URAM with ECC enabled.
 - **off**: No status registers generated. This is the default setting.
 - **ecc**: Enable counters for ECC errors for BRAMs and URAMs
 - **-s_axilite_sw_reset[=false|true]**: Enable the software reset of a kernel in the `s_axilite` adapter.

Examples

The following example adds a clock enable port to the IP:

```
config_interface -clock_enable
```

config_op

Description

Sets the default options for micro-architecture binding of an operator (add, mul, sub...) to an FPGA implementation resource, and specify its latency.

Binding is the process in which operators (such as addition, multiplication, and shift) are mapped to specific RTL implementations. For example, a `mult` operation implemented as a combinational or pipelined RTL multiplier.

This command can be used multiple times to configure the default binding of different operation types to different implementation resources, or specify the default latency for that operation.

The default configuration defined by `config_op` can be overridden by specifying the `BIND_OP` pragma or directive for a specific design element.

Syntax

```
config_op [OPTIONS] <op>
```

- **<op>**: Specifies the type of operation for the specified variable. Supported values include:

- mul: integer multiplication operation
- add: integer add operation
- sub: integer subtraction operation
- fadd: single precision floating-point add operation
- fsub: single precision floating-point subtraction operation
- fdiv: single precision floating-point divide operation
- fexp: single precision floating-point exponential operation
- flog: single precision floating-point logarithmic operation
- fmul: single precision floating-point multiplication operation
- frsqrt: single precision floating-point reciprocal square root operation
- frecip: single precision floating-point reciprocal operation
- fsqrt: single precision floating-point square root operation
- dadd: double precision floating-point add operation
- dsub: double precision floating-point subtraction operation
- ddiv: double precision floating-point divide operation
- dexp: double precision floating-point exponential operation
- dlog: double precision floating-point logarithmic operation
- dmul: double precision floating-point multiplication operation
- drsqrt: double precision floating-point reciprocal square root operation
- drecip: double precision floating-point reciprocal operation
- dsqrt: double precision floating-point square root operation
- hadd: half precision floating-point add operation
- hsub: half precision floating-point subtraction operation
- hdiv: half precision floating-point divide operation
- hmul: half precision floating-point multiplication operation
- hsqrt: half precision floating-point square root operation
- facc: single precision floating-point accumulate operation
- fmacc: single precision floating-point multiply-accumulate operation
- fmadd: single precision floating-point multiply-add operation



TIP: Comparison operators, such as `dcmp`, are implemented in LUTs and cannot be implemented outside of the fabric, or mapped to DSPs, and so are not configurable with the `config_op` or `bind_op` commands.

Options

- **-impl** [dsp | fabric | meddsp | fulldsp | maxdsp | primitivedsp | auto | none | all]: Defines the default implementation style for the specified operation. The default is to let the tool choose which implementation to use. The selections include:
 - all: All implementations. This is the default setting.
 - dsp: Use DSP resources
 - fabric: Use non-DSP resources
 - meddsp: Floating Point IP Medium Usage of DSP resources
 - fulldsp: Floating Point IP Full Usage of DSP resources
 - maxdsp: Floating Point IP Max Usage of DSP resources
 - primitivedsp: Floating Point IP Primitive Usage of DSP resources
 - auto: enable inference of combined `facc` | `fmacc` | `fmadd` operators
 - none: disable inference of combined `facc` | `fmacc` | `fmadd` operators
- **-latency <value>**: Defines the default latency for the binding of the type to the implementation resource. The valid value range varies for each implementation (-`impl`) of the operation. The default is -1, which applies the standard latency for the implementation resource.



TIP: The latency can be specified for a specific operation without specifying the implementation detail. This leaves Vitis HLS to choose the implementation while managing the latency.

- **-precision** [low | high | standard]: Applies to `facc`, `fmacc`, and `fmadd` operators. Specify the precision for the given operator.
 - low: Use a low precision (60 bit and 100 bit integer) accumulation implementation when available. This option is only available on certain non-Versal devices, and may cause RTL/Co-Sim mismatches due to insufficient precision with respect to C++ simulation. However, it can always be pipelined with an `II=1` without source code changes, though it uses approximately 3X the resources of `standard` precision floating point accumulation.
 - high: Use high precision (one extra bit) fused multiply-add implementation when available. This option is useful for high-precision applications and is very efficient on Versal devices, although it may cause RTL/Co-Sim mismatches due to the extra precision with respect to C ++ simulation. It uses more resources than `standard` precision floating point accumulation.

- **standard**: standard precision floating point accumulation and multiply-add is suitable for most uses of floating-point, and is the default setting. It always uses a true floating-point accumulator that can be pipelined with $l=1$ on Versal devices, and l that is typically between 3 and 5 (depending on clock frequency and target device) on non-Versal devices.

Example 1

The following example binds the addition operation to the fabric, with the specified latency:

```
config_op add -impl fabric -latency 2
```

Example 2

The following example enables the floating point accumulator with low-precision to achieve $l=1$ on a non-Versal device:

```
config_op facc -impl auto -precision low
```

config_rtl

Description

Configures various attributes of the output RTL, the type of reset used, and the encoding of the state machines. It also allows you to use specific identification in the RTL.

By default, these options are applied to the top-level design and all RTL blocks within the design. You can optionally specify a specific RTL model.

Syntax

```
config_rtl [OPTIONS]
```

Options

- **-deadlock_detection <none | sim | hw>**: Enables simulation or synthesis deadlock detection in top level RTL of exported IP/XO file. The options are as follows:
 - **none** : Deadlock detection disabled
 - **sim** : Enables deadlock detection only for simulation/emulation (default)
 - **hw** : Deadlock detection enabled in synthesized and simulatable RTL IP. Adds `ap_local_deadlock` and `ap_local_block` signals to the IP to enable local and global deadlock detection.

- **-deadlock_diagnosis** : Enable deadlock detection diagnosis for Vitis kernels (.xo) during hardware emulation in the Vitis tool.
- **-header <string>**: Places the contents of file <string> at the top (as comments) of all output RTL and simulation files.



TIP: Use this option to ensure that the output RTL files contain user specified identification.

- **-kernel_profile**: Add top level event and stall ports required by kernel profiling.
- **-module_auto_prefix**: Specifies the top level function name as the prefix value. This option is ignored if `config_rtl -module_prefix` is also specified. This is enabled by default.
- **-module_prefix <string>**: Specifies a user-defined prefix to be added to all RTL entity/module names.
- **-mult_keep_attribute**: Enable keep attribute.
- **-register_all_io**: Register all I/O signals by default. The default is `false`. This is enabled when the option is specified.
- **-register_reset_num <int>**: Specifies the number of registers to add to the reset signal. In the Vivado IP flow the default is 0. For the Vitis kernel flow, the default value is 3.
- **-reset [none | control | state | all]**: Variables initialized in the C/C++ code are always initialized to the same value in the RTL and therefore in the bitstream. This initialization is performed only at power-on. It is not repeated when a reset is applied to the design.

The setting applied with the `-reset` option determines how registers and memories are reset.

- **none**: No reset is added to the design.
- **control**: Resets control registers, such as those used in state machines and those used to generate I/O protocol signals. This is the default setting.
- **state**: Resets control registers and registers or memories derived from static or global variables in the C/C++ code. Any static or global variable initialized in the C/C++ code is reset to its initialized value.
- **all**: Resets all registers and memories in the design. Any static or global variable initialized in the C/C++ code is reset to its initialized value.
- **-reset_async**: Causes all registers to use a asynchronous reset. If this option is not specified, a synchronous reset is used.
- **-reset_level (low | high)**: Allows the polarity of the reset signal to be either active-Low or active-High. The default is `High`.

Examples

Configures the output RTL to have all registers reset with an asynchronous active-Low reset.

```
config_rtl -reset all -reset_async -reset_level low
```

Adds the contents of `my_message.txt` as a comment to all RTL output files.

```
config_rtl -header my_message.txt
```

config_schedule

Description

Configures the default type of scheduling performed by Vitis HLS.

Syntax

```
config_schedule [OPTIONS]
```

Options

- `-enable_dsp_full_reg[=true|false]`: Specifies that the DSP signals should be fully registered. The default is `true`.

Examples

The following example disables registering DSP signals:

```
config_schedule -enable_dsp_full_reg=false
```

config_storage

Description

Sets the global default options for Vitis HLS micro-architecture binding of FIFO storage elements to memory resources.

The default configuration defined by `config_storage` for FIFO storage can be overridden by specifying the `BIND_STORAGE` pragma or directive for a specific design element, or specifying the `storage_type` option for the `INTERFACE` pragma or directive for objects on the interface.

Syntax

```
config_storage [OPTIONS] <type>
```

- **<type>**: Configures the `fifo` type.

Options

- **-auto_srl_max_bits <value>**: Specifies the maximum allowed SRL total bits (depth * width) for auto-srl implementations (`-impl autosrl`). The default is 1024.
- **-auto_srl_max_depth <value>**: Specifies the maximum allowed SRL depth for auto-srl implementation (`-impl autosrl`). The default is 2.
- **-impl [auto | bram | lutram | uram | memory | srl]**: Defines the device resource to use in binding the specified type.

Examples

The following example configures the default binding of `fifo`:

```
config_storage fifo -impl uram
```

config_unroll

Description

Automatically unroll loops based on the loop index limit (or tripcount).

Syntax

```
config_unroll [OPTIONS] <value>
```

Options

- **-tripcount_threshold <value>**: All loops which have fewer iterations than the specified value are automatically unrolled. The default value is 0.

Example

The following command ensures all loops which have fewer than 18 iterations are automatically unrolled during scheduling.

```
config_unroll -tripcount_threshold 18
```

Optimization Directives

Directives, or the `set_directive_*` commands, can be specified as Tcl commands that are associated with a specific solution, or set of solutions. Allowing you to customize the synthesis results for the same source code across different solutions. This lets you preserve the original code while engaging in what-if analysis of the design.

Directives must be run in the interactive mode, `vitis_hls -i`, or can be run as a script using the `-f` option as described in [vitis_hls Command](#).

Pragmas are directives that you can apply in the source code, rather than as a Tcl script, and so change the synthesis results for all implementations of your code. There are HLS pragmas for every `set_directive` command, so you can choose how you want to work with your Vitis™ HLS project. Refer to [HLS Pragmas](#) for information on the different pragmas.

Directives and pragmas are also available through the Vitis HLS IDE for assignment to specific elements of your source code, as described in [Adding Pragmas and Directives](#).



TIP: When running the commands through the IDE, the Tcl commands are added to a script of your project written to `solution/constraints/script.tcl`.

set_directive_aggregate

Description

This directive collects the data fields of a struct into a single wide scalar. Any arrays declared within the struct and Vitis HLS performs a similar operation as `set_directive_array_reshape`, and completely partitions and reshapes the array into a wide scalar and packs it with other elements of the struct.



TIP: Arrays of structs are restructured as arrays of aggregated elements.

The bit alignment of the resulting new wide-word can be inferred from the declaration order of the struct elements. The first element takes the least significant sector of the word and so forth until all fields are mapped.

Note: The AGGREGATE optimization does not pack the structs, and cannot be used on structs that contain other structs.

Syntax

```
set_directive_aggregate [OPTIONS] <location> <variable>
```

- <location> is the location (in the format `function[/label]`) which contains the variable which will be packed.
- <variable> is the struct variable to be packed.

Options

- `-compact [bit | byte | none | auto]`: Specifies the alignment of the aggregated struct. Alignment can be on the bit-level (packed), the byte-level (padded), none, or automatically determined by the tool which is the default behavior.

Examples

Aggregates struct pointer `AB` with three 8-bit fields (`typedef struct {unsigned char R, G, B;} pixel`) in function `func`, into a new 24-bit pointer, aligning data at the bit-level.

```
set_directive_aggregate func AB -compact bit
```

See Also

- [pragma HLS aggregate](#)
- [set_directive_array_reshape](#)
- [set_directive_disaggregate](#)

set_directive_alias

Description

Specify that two or more M_AXI pointer arguments point to the same underlying buffer in memory (DDR or HBM) and indicate any aliasing between the pointers by setting the distance or offset between them.



IMPORTANT! The ALIAS pragma applies to top-level function arguments mapped to M_AXI interfaces.

Vitis HLS considers different pointers to be independent channels and generally does not provide any dependency analysis. However, in cases where the host allocates a single buffer for multiple pointers, this relationship can be communicated through the ALIAS pragma or directive and dependency analysis can be maintained. The ALIAS pragma enables data dependence analysis in Vitis HLS by defining the distance between pointers in the buffer.

Requirements for ALIAS:

- All ports assigned to an ALIAS pragma must be assigned to M_AXI interfaces and assigned to different bundles, as shown in the example below
- Each port can only be used in one ALIAS pragma or directive
- The depth of all ports assigned to an ALIAS pragma must be the same
- When offset is specified, the number of ports and number of offsets specified must be the same: one offset per port
- The offset for the INTERFACE must be specified as slave or direct, offset=off is not supported

Syntax

```
set_directive_alias [OPTIONS] <location> <ports>
```

- <location> is the location string in the format function[/label] that the ALIAS pragma applies to.
- <ports> specifies the ports to alias.

Options

- **-distance <integer>**: Specifies the difference between the pointer values passed to the ports in the list.
- **-offset <string>**: Specifies the offset of the pointer passed to each port in the ports list with respect to the origin of the array.

Example

For the following function top:

```
void top(int *arr0, int *arr1, int *arr2, int *arr3, ...) {  
    #pragma HLS interface M_AXI port=arr0 bundle=hbm0 depth=0x40000000  
    #pragma HLS interface M_AXI port=arr1 bundle=hbm1 depth=0x40000000  
    #pragma HLS interface M_AXI port=arr2 bundle=hbm2 depth=0x40000000  
    #pragma HLS interface M_AXI port=arr3 bundle=hbm3 depth=0x40000000
```

The following command defines aliasing for the specified array pointers, and defines the distance between them:

```
set_directive_alias "top" arr0,arr1,arr2,arr3 -distance 10000000
```

Alternatively, the following command specifies the offset between pointers, to accomplish the same effect:

```
set_directive_alias top arr0,arr1,arr2,arr3 -offset  
00000000,10000000,20000000,30000000
```

See Also

- [pragma HLS alias](#)
- [set_directive_interface](#)

set_directive_allocation

Description

Specifies instance restrictions for resource allocation.

The ALLOCATION pragma or directive can limit the number of RTL instances and hardware resources used to implement specific functions, loops, or operations. For example, if the C/C++ source has four instances of a function `foo_sub`, the `set_directive_allocation` command can ensure that there is only one instance of `foo_sub` in the final RTL. All four instances are implemented using the same RTL block. This reduces resources used by the function, but negatively impacts performance by sharing those resources.

The operations in the C/C++ code, such as additions, multiplications, array reads, and writes, can also be limited by the `set_directive_allocation` command.

Syntax

```
set_directive_allocation [OPTIONS] <location> <instances>
```

- `<location>` is the location string in the format `function[/label]`.
- `<instances>` is a function or operator.

The function can be any function in the original C/C++ code that has not been either inlined by the `set_directive_inline` command or inlined automatically by Vitis HLS.

For a complete list of operations that can be limited using the ALLOCATION pragma, refer to the [config_op](#) command.

Options

- `-limit <integer>:`

Sets a maximum limit on the number of instances (of the type defined by the `-type` option) to be used in the RTL design.

- `-type [function|operation]`: The instance type can be `function` (default) or `operation`.

Examples

Given a design `foo_top` with multiple instances of function `foo`, limits the number of instances of `foo` in the RTL to 2.

```
set_directive_allocation -limit 2 -type function foo_top foo
```

Limits the number of multipliers used in the implementation of `My_func` to 1. This limit does not apply to any multipliers that might reside in sub-functions of `My_func`. To limit the multipliers used in the implementation of any sub-functions, specify an allocation directive on the sub-functions or inline the sub-function into function `My_func`.

```
set_directive_allocation -limit 1 -type operation My_func mul
```

See Also

- [pragma HLS allocation](#)
- [set_directive_inline](#)

set_directive_array_partition

Description



IMPORTANT! *Array_Partition and Array_Reshape pragmas and directives are not supported for M_AXI Interfaces on the top-level function. Instead you can use the hls::vector data types as described in [Vector Data Types](#).*

Partitions an array into smaller arrays or individual elements.

This partitioning:

- Results in RTL with multiple small memories or multiple registers instead of one large memory.
- Effectively increases the amount of read and write ports for the storage.
- Potentially improves the throughput of the design.
- Requires more memory instances or registers.

Syntax

```
set_directive_array_partition [OPTIONS] <location> <array>
```

- <location> is the location (in the format `function[/label]`) which contains the array variable.
- <array> is the array variable to be partitioned.

Options

- `-dim <integer>:`

Note: Relevant for multi-dimensional arrays only.

Specifies which dimension of the array is to be partitioned.

- If a value of 0 is used, all dimensions are partitioned with the specified options.
- Any other value partitions only that dimension. For example, if a value 1 is used, only the first dimension is partitioned.

- `-factor <integer>:`

Note: Relevant for type `block` or `cyclic` partitioning only.

Specifies the number of smaller arrays that are to be created.

- `-type (block|cyclic|complete):`

- `block` partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks where N is the integer defined by the `-factor` option.
- `cyclic` partitioning creates smaller arrays by interleaving elements from the original array. For example, if `-factor 3` is used:
 - Element 0 is assigned to the first new array.
 - Element 1 is assigned to the second new array.
 - Element 2 is assigned to the third new array.
 - Element 3 is assigned to the first new array again.
- `complete` partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. For multi-dimensional arrays, specify the partitioning of each dimension, or use `-dim 0` to partition all dimensions.

The default is `complete`.

Example 1

Partitions array AB[13] in function `func` into four arrays. Because four is not an integer factor of 13:

- Three arrays have three elements.
- One array has four elements (AB[9:12]).

```
set_directive_array_partition -type block -factor 4 func AB
```

Partitions array AB[6][4] in function `func` into two arrays, each of dimension [6][2].

```
set_directive_array_partition -type block -factor 2 -dim 2 func AB
```

Partitions all dimensions of AB[4][10][6] in function `func` into individual elements.

```
set_directive_array_partition -type complete -dim 0 func AB
```

Example 2

Partitioned arrays can be addressed in your code by the new structure of the array, as shown in the following code example;

```
struct SS
{
    int x[N];
    int y[N];
};

int top(SS *a, int b[4][6], SS &c) {...}

set_directive_array_partition top b -type complete -dim 1
set_directive_interface -mode ap_memory top b[0]
set_directive_interface -mode ap_memory top b[1]
set_directive_interface -mode ap_memory top b[2]
set_directive_interface -mode ap_memory top b[3]
```

See Also

- [pragma HLS array_partition](#)
- [set_directive_array_reshape](#)

set_directive_array_reshape

Description

 **IMPORTANT!** *Array_Partition and Array_Reshape pragmas and directives are not supported for M_AXI Interfaces on the top-level function. Instead you can use the hls::vector data types as described in [Vector Data Types](#).*

Combines array partitioning with vertical array mapping to create a single new array with fewer elements but wider words.

The `set_directive_array_reshape` command has the following features:

- Splits the array into multiple arrays (like `set_directive_array_partition`).
- Automatically recombine the arrays vertically to create a new array with wider words.

Syntax

```
set_directive_array_reshape [OPTIONS] <location> <array>
```

- `<location>` is the location (in the format `function[/label]`) that contains the array variable.
- `<array>` is the array variable to be reshaped.

Options

- `-dim <integer>:`

Note: Relevant for multi-dimensional arrays only.

Specifies which dimension of the array is to be reshaped.

- If the value is set to 0, all dimensions are partitioned with the specified options.
- Any other value partitions only that dimension. The default is 1.

- `-factor <integer>:`

Note: Relevant for type `block` or `cyclic` reshaping only.

Specifies the number of temporary smaller arrays to be created.

- `-object:`

Note: Relevant for container arrays only.

Applies reshape on the objects within the container. If the option is specified, all dimensions of the objects will be reshaped, but all dimensions of the container will be kept.

- **-type (block|cyclic|complete):**
 - **block** reshaping creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks where N is the integer defined by the **-factor** option and then combines the N blocks into a single array with **word-width*N**. The default is **complete**.
 - **cyclic** reshaping creates smaller arrays by interleaving elements from the original array. For example, if **-factor 3** is used, element 0 is assigned to the first new array, element 1 to the second new array, element 2 is assigned to the third new array, and then element 3 is assigned to the first new array again. The final array is a vertical concatenation (word concatenation, to create longer words) of the new arrays into a single array.
 - **complete** reshaping decomposes the array into temporary individual elements and then recombines them into an array with a wider word. For a one-dimension array this is equivalent to creating a very-wide register (if the original array was N elements of M bits, the result is a register with $N*M$ bits). This is the default.

Example 1

Reshapes 8-bit array AB[17] in function `func` into a new 32-bit array with five elements.

Because four is not an integer factor of 17:

- Index 17 of the array, AB[17], is in the lower eight bits of the reshaped fifth element.
- The upper eight bits of the fifth element are unused.

```
set_directive_array_reshape -type block -factor 4 func AB
```

Partitions array AB[6][4] in function `func`, into a new array of dimension [6][2], in which dimension 2 is twice the width.

```
set_directive_array_reshape -type block -factor 2 -dim 2 func AB
```

Reshapes 8-bit array AB[4][2][2] in function `func` into a new single element array (a register), $4*2*2*8$ (= 128)-bits wide.

```
set_directive_array_reshape -type complete -dim 0 func AB
```

Example 2

Partitioned arrays can be addressed in your code by the new structure of the array, as shown in the following code example;

```
struct SS
{
    int x[N];
    int y[N];
};
```

```
int top(SS *a, int b[4][6], SS &c) {...}  
  
set_directive_array_reshape top b -type complete -dim 0  
set_directive_interface -mode ap_memory top b[0]
```

See Also

- [pragma HLS array_reshape](#)
- [set_directive_array_partition](#)

set_directive_bind_op

Description

Vitis HLS implements the operations in the code using specific implementations. The `set_directive_bind_op` command specifies that for a specified variable, an operation (`mul`, `add`, `sub`) should be mapped to a specific device resource for implementation (`impl`) in the RTL. If this command is not specified, Vitis HLS automatically determines the resource to use.

For example, to indicate that a specific multiplier operation (`mul`) is implemented in the device fabric rather than a DSP, you can use the `set_directive_bind_op` command.

You can also specify the latency of the operation using the `-latency` option.



IMPORTANT! To use the `-latency` option, the operation must have an available multi-stage implementation. The HLS tool provides a multi-stage implementation for all basic arithmetic operations (`add`, `subtract`, `multiply`, and `divide`), and all floating-point operations.

Syntax

```
set_directive_bind_op [OPTIONS] <location> <variable>
```

- `<location>` is the location (in the format `function[/label]`) which contains the variable.
- `<variable>` is the variable to be assigned. The variable in this case is one that is assigned the result of the operation that is the target of this directive.

Options

- `-op <value>`: Defines the operation to bind to a specific implementation resource. Supported functional operations include: `mul`, `add`, `sub`

Supported floating point operations include: fadd, fsub, fdiv, fexp, flog, fmul, frsqrt, frecip, fsqrt, dadd, dsub, ddiv, dexp, dlog, dmul, drsqrt, drecip, dsqrt, hadd, hsub, hdiv, hmul, and hsqrt



TIP: Floating-point operations include single precision (f), double-precision (d), and half-precision (h).

- **-impl <value>**: Defines the implementation to use for the specified operation. Supported implementations for functional operations include fabric and dsp. Supported implementations for floating point operations include: fabric, meddsp, fulldsp, maxdsp, and primitivedsp.

Note: primitivedsp is only available on Versal devices.

- **-latency <int>**: Defines the default latency for the implementation of the operation. The valid latency varies according to the specified op and impl. The default is -1, which lets Vitis HLS choose the latency. The tables below reflect the supported combinations of operation, implementation, and latency.

Table 27: Supported Combinations of Functional Operations, Implementation, and Latency

Operation	Implementation	Min Latency	Max Latency
add	fabric	0	4
add	dsp	0	4
mul	fabric	0	4
mul	dsp	0	4
sub	fabric	0	4
sub	dsp	0	0



TIP: Comparison operators, such as `dcmp`, are implemented in LUTs and cannot be implemented outside of the fabric, or mapped to DSPs, and so are not configurable with the `config_op` or `bind_op` commands.

Table 28: Supported Combinations of Floating Point Operations, Implementation, and Latency

Operation	Implementation	Min Latency	Max Latency
fadd	fabric	0	13
fadd	fulldsp	0	12
fadd	primitivedsp	0	3
fsub	fabric	0	13
fsub	fulldsp	0	12
fsub	primitivedsp	0	3
fdiv	fabric	0	29
fexp	fabric	0	24

Table 28: Supported Combinations of Floating Point Operations, Implementation, and Latency (cont'd)

Operation	Implementation	Min Latency	Max Latency
fexp	meddsp	0	21
fexp	fulldsp	0	30
flog	fabric	0	24
flog	meddsp	0	23
flog	fulldsp	0	29
fmul	fabric	0	9
fmul	meddsp	0	9
fmul	fulldsp	0	9
fmul	maxdsp	0	7
fmul	primitivedsp	0	4
fsqrt	fabric	0	29
frsqrt	fabric	0	38
frsqrt	fulldsp	0	33
frecip	fabric	0	37
frecip	fulldsp	0	30
dadd	fabric	0	13
dadd	fulldsp	0	15
dsub	fabric	0	13
dsub	fulldsp	0	15
ddiv	fabric	0	58
dexp	fabric	0	40
dexp	meddsp	0	45
dexp	fulldsp	0	57
dlog	fabric	0	38
dlog	meddsp	0	49
dlog	fulldsp	0	65
dmul	fabric	0	10
dmul	meddsp	0	13
dmul	fulldsp	0	13
dmul	maxdsp	0	14
dsqrt	fabric	0	58
drsqrt	fulldsp	0	111
drecip	fulldsp	0	36
hadd	fabric	0	9
hadd	meddsp	0	12
hadd	fulldsp	0	12
hsub	fabric	0	9
hsub	meddsp	0	12

Table 28: Supported Combinations of Floating Point Operations, Implementation, and Latency (cont'd)

Operation	Implementation	Min Latency	Max Latency
hsub	fulldsp	0	12
hdiv	fabric	0	16
hmul	fabric	0	7
hmul	fulldsp	0	7
hmul	maxdsp	0	9
hsqrt	fabric	0	16

Examples

In the following example, a two-stage pipelined multiplier using fabric logic is specified to implement the multiplication for variable `<c>` of the function `foo`.

```
int foo (int a, int b) {
    int c, d;
    c = a*b;
    d = a*c;
    return d;
}
```

And the `set_directive` command is as follows:

```
set_directive_bind_op -op mul -impl fabric -latency 2 "foo" c
```



TIP: The HLS tool selects the core to use for variable `<d>`.

See Also

- [pragma HLS bind_op](#)
- [set_directive_bind_storage](#)

set_directive_bind_storage

Description

The `set_directive_bind_storage` command assigns a variable (array, or function argument) in the code to a specific memory type (`type`) in the RTL. If the command is not specified, the Vitis HLS tool determines the memory type to assign. The HLS tool implements the memory using specified implementations (`impl`) in the hardware. For example, you can use the `set_directive_bind_storage` command to specify which type of memory, and which implementation to use for an array variable. Also, this allows you to control whether the array is implemented as a single or a dual-port RAM.

 **IMPORTANT!** This feature is important for arrays on the top-level function interface, because the memory type associated with the array determines the number and type of ports needed in the RTL, as discussed in [Arrays on the Interface](#). However, for variables assigned to top-level function arguments you must assign the memory type and implementation using the `-storage-type` and `-storage-impl` options of the INTERFACE pragma or directive.

You can use the `-latency` option to specify the latency of the implementation. For block RAMs on the interface, the `-latency` option allows you to model off-chip, non-standard SRAMs at the interface, for example supporting an SRAM with a latency of 2 or 3. For internal operations, the `-latency` option allows the operation to be implemented using more pipelined stages. These additional pipeline stages can help resolve timing issues during RTL synthesis.

 **IMPORTANT!** To use the `-latency` option, the operation must have an available multi-stage implementation. The HLS tool provides a multi-stage implementation for all block RAMs.

For best results, Xilinx recommends that you use `-std=c99` for C and `-fno-builtins` for C and C++. To specify the C compile options, such as `-std=c99`, use the Tcl command `add_files` with the `-cflags` option. Alternatively, select the **Edit CFLAGs** button in the Project Settings dialog box as described in [Creating a New Vitis HLS Project](#).

Syntax

```
set_directive_bind_storage [OPTIONS] <location> <variable>
```

- `<location>` is the location (in the format `function[/label]`) which contains the variable.
- `<variable>` is the variable to be assigned.

 **TIP:** If the variable is an argument of a top-level function, then use the `-storage-type` and `-storage-impl` options of the INTERFACE pragma or directive.

Options

- **-type:** Defines the type of memory to bind to the specified variable. Supported types include: `fifo`, `ram_1p`, `ram_1wnr`, `ram_2p`, `ram_s2p`, `ram_t2p`, `rom_1p`, `rom_2p`, and `rom_np`.

Table 29: Storage Types

Type	Description
FIFO	A FIFO. Vitis HLS determines how to implement this in the RTL, unless the <code>-impl</code> option is specified.
RAM_1P	A single-port RAM. Vitis HLS determines how to implement this in the RTL, unless the <code>-impl</code> option is specified.
RAM_1WNR	A RAM with 1 write port and N read ports, using N banks internally.
RAM_2P	A dual-port RAM that allows read operations on one port and both read and write operations on the other port.
RAM_S2P	A dual-port RAM that allows read operations on one port and write operations on the other port.
RAM_T2P	A true dual-port RAM with support for both read and write on both ports.
ROM_1P	A single-port ROM. Vitis HLS determines how to implement this in the RTL, unless the <code>-impl</code> option is specified.
ROM_2P	A dual-port ROM.
ROM_NP	A multi-port ROM.

- **-impl <value>:** Defines the implementation for the specified memory type. Supported implementations include: `bram`, `bram_ecc`, `lutram`, `uram`, `uram_ecc`, `srl`, `memory`, and `auto` as described below.

Table 30: Supported Implementation

Name	Description
MEMORY	Generic memory for FIFO, lets the Vivado tool choose the implementation.
URAM	UltraRAM resource
URAM_ECC	UltraRAM with ECC
SRL	Shift Register Logic resource
LUTRAM	Distributed RAM resource
BRAM	Block RAM resource
BRAM_ECC	Block RAM with ECC
AUTO	Vitis HLS automatically determine the implementation of the variable.

Table 31: Supported Implementations by FIFO/RAM/ROM

Type	Command/Pragma	Scope	Supported Implementations
FIFO	<code>bind_storage</code> ¹	local	<code>AUTO</code> , BRAM, LUTRAM, URAM, MEMORY, SRL

Table 31: Supported Implementations by FIFO/RAM/ROM (cont'd)

Type	Command/Pragma	Scope	Supported Implementations
FIFO	config_storage	global	AUTO , BRAM, LUTRAM, URAM, MEMORY, SRL
RAM* ROM*	bind_storage	local	AUTO BRAM, BRAM_ECC, LUTRAM, URAM, URAM_ECC
RAM* ROM*	config_storage ²	global	N/A
RAM_1P	set_directive_interface s_axilite - storage_impl	local	AUTO , BRAM, URAM
	config_interface - m_axi_buffer_impl	global	AUTO , BRAM , LUTRAM, URAM

Notes:

- When no implementation is specified the directive uses AUTOSRL behavior as a default. However, this value cannot be specified.
 - `config_storage` only supports FIFO types.
- latency <int>**: Defines the default latency for the binding of the storage type to the implementation. The valid latency varies according to the specified `type` and `impl`. The default is -1, which lets Vitis HLS choose the latency.

Table 32: Supported Combinations of Memory Type, Implementation, and Latency

Type	Implementation	Min Latency	Max Latency
FIFO	BRAM	0	0
FIFO	LUTRAM	0	0
FIFO	MEMORY	0	0
FIFO	SRL	0	0
FIFO	URAM	0	0
RAM_1P	AUTO	1	3
RAM_1P	BRAM	1	3
RAM_1P	LUTRAM	1	3
RAM_1P	URAM	1	3
RAM_1WNR	AUTO	1	3
RAM_1WNR	BRAM	1	3
RAM_1WNR	LUTRAM	1	3
RAM_1WNR	URAM	1	3
RAM_2P	AUTO	1	3
RAM_2P	BRAM	1	3
RAM_2P	LUTRAM	1	3
RAM_2P	URAM	1	3
RAM_S2P	BRAM	1	3
RAM_S2P	BRAM_ECC	1	3
RAM_S2P	LUTRAM	1	3

Table 32: Supported Combinations of Memory Type, Implementation, and Latency (cont'd)

Type	Implementation	Min Latency	Max Latency
RAM_S2P	URAM	1	3
RAM_S2P	URAM_ECC	1	3
RAM_T2P	BRAM	1	3
RAM_T2P	URAM	1	3
ROM_1P	AUTO	1	3
ROM_1P	BRAM	1	3
ROM_1P	LUTRAM	1	3
ROM_2P	AUTO	1	3
ROM_2P	BRAM	1	3
ROM_2P	LUTRAM	1	3
ROM_NP	BRAM	1	3
ROM_NP	LUTRAM	1	3

 **IMPORTANT!** Any combinations of memory type and implementation that are not listed in the prior table are not supported by `set_directive_bind_storage`.

Examples

In the following example, the `coeffs[128]` variable is an argument to the function `func1`. The directive specifies that `coeffs` uses a single port RAM implemented on a BRAM core from the library.

```
set_directive_bind_storage -impl bram "func1" coeffs RAM_1P
```



TIP: The ports created in the RTL to access the values of `coeffs` are defined in the `RAM_1P` core.

See Also

- [pragma HLS bind_storage](#)

set_directive_dataflow

Description

Specifies that dataflow optimization be performed on the functions or loops, improving the concurrency of the RTL implementation.

All operations are performed sequentially in a C/C++ description. In the absence of any directives that limit resources (such as `set_directive_allocation`), Vitis HLS seeks to minimize latency and improve concurrency. Data dependencies can limit this. For example, functions or loops that access arrays must finish all read/write accesses to the arrays before they complete. This prevents the next function or loop that consumes the data from starting operation.

It is possible for the operations in a function or loop to start operation before the previous function or loop completes all its operations. When the DATAFLOW optimization is specified, the HLS tool analyzes the dataflow between sequential functions or loops and creates channels (based on ping-pong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL.



TIP: The `config_dataflow` command specifies the default memory channel and FIFO depth used in DATAFLOW optimization.

If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, Vitis HLS attempts to minimize the initiation interval and start operation as soon as data is available.

For the DATAFLOW optimization to work, the data must flow through the design from one task to the next. The following coding styles prevent the HLS tool from performing the DATAFLOW optimization. Refer to [Limitations of Control-Driven Task-Level Parallelism](#) for additional details.

- Single-producer-consumer violations
- Feedback between tasks
- Conditional execution of tasks
- Loops with multiple exit conditions



IMPORTANT! If any of these coding styles are present, the HLS tool issues a message and does not perform DATAFLOW optimization.

Finally, the DATAFLOW optimization has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from the DATAFLOW optimization, you must apply the optimization to the loop, the sub-function, or inline the sub-function.

Syntax

```
set_directive_dataflow <location> -disable_start_propagation
```

- <location> is the location (in the format `function[/label]`) at which dataflow optimization is to be performed.

- `-disable_start_propagation` disables the creation of a start FIFO used to propagate a start token to an internal process. Such FIFOs can sometimes be a bottleneck for performance.

Examples

Specifies dataflow optimization within function `foo`.

```
set_directive_dataflow foo
```

See Also

- [pragma HLS dataflow](#)
- [set_directive_allocation](#)
- [config_dataflow](#)

set_directive_dependence

Description

Vitis HLS detects dependencies within loops: dependencies within the same iteration of a loop are loop-independent dependencies, and dependencies between different iterations of a loop are loop-carried dependencies.

These dependencies are impacted when operations can be scheduled, especially during function and loop pipelining.

- **Loop-independent dependence:** The same element is accessed in a single loop iteration.

```
for ( i=0 ; i<N ; i++ ) {  
    A[ i ] = x;  
    y = A[ i ];  
}
```

- **Loop-carried dependence:** The same element is accessed from a different loop iteration.

```
for ( i=0 ; i<N ; i++ ) {  
    A[ i ] = A[ i-1 ] * 2;  
}
```

Under certain circumstances, such as variable dependent array indexing or when an external requirement needs to be enforced (for example, two inputs are never the same index), the dependence analysis might be too conservative and fail to filter out false dependencies. The `set_directive_dependence` command allows you to explicitly define the dependencies and eliminate a false dependence.



IMPORTANT! Specifying a false dependency when the dependency is not false can result in incorrect hardware. Ensure dependencies are correct (true or false) before specifying them.

Syntax

```
set_directive_dependence -dependent <arg> [OPTIONS] <location>
```

- **-dependent (true | false)**: This argument should be specified to indicate whether a dependence is true and needs to be enforced, or is false and should be removed. However, when not specified, the tool will return a warning that the value was not specified and will assume a value of false.
- **<location>**: The location in the code, specified as `function[/label]`, where the dependence is defined.

Options

- **-class (array | pointer)**: Specifies a class of variables in which the dependence needs clarification. This is mutually exclusive with the `-variable` option.
- **-dependent (true | false)**: Specify if a dependence needs to be enforced (true) or removed (false).
- **-direction (RAW | WAR | WAW)**:

Note: Relevant only for loop-carried dependencies.

Specifies the direction for a dependence:

- **RAW (Read-After-Write - true dependence)**: The write instruction uses a value used by the read instruction.
- **WAR (Write-After-Read - anti dependence)**: The read instruction gets a value that is overwritten by the write instruction.
- **WAW (Write-After-Write - output dependence)**: Two write instructions write to the same location, in a certain order.
- **-distance <integer>**:

Note: Relevant only for loop-carried dependencies where `-dependent` is set to true.

Specifies the inter-iteration distance for array access.

- **-type (intra | inter)**: Specifies whether the dependence is:
 - Within the same loop iteration (`intra`), or
 - Between different loop iterations (`inter`) (default).

- **-variable <variable>**: Defines a specific variable to apply the dependence directive. Mutually exclusive with the **-class** option.



IMPORTANT! You cannot specify a *dependence* for function arguments that are bundled with other arguments in an *m_axi* interface. This is the default configuration for *m_axi* interfaces on the function. You also cannot specify a dependence for an element of a struct, unless the struct has been disaggregated.

Examples

Removes the dependence between `Var1` in the same iterations of `loop_1` in function `func`.

```
set_directive_dependence -variable Var1 -type intra \
-dependent false func/loop_1
```

The dependence on all arrays in `loop_2` of function `func` informs Vitis HLS that all reads must happen *after* writes in the same loop iteration.

```
set_directive_dependence -class array -type intra \
-dependent true -direction RAW func/loop_2
```

See Also

- [pragma HLS dependence](#)
- [set_directive_disaggregate](#)
- [set_directive_pipeline](#)

set_directive_disaggregate

Description

The `set_directive_disaggregate` command lets you deconstruct a `struct` variable into its individual elements. The number and type of elements created are determined by the contents of the struct itself.



IMPORTANT! Structs used as arguments to the top-level function are aggregated by default, but can be disaggregated with this directive or pragma. Refer to [AXI4-Stream Interfaces](#) for important information about disaggregating structs associated with streams.

Syntax

```
set_directive_disaggregate <location> <variable>
```

- <location> is the location (in the format `function[/label]`) where the variable to disaggregate is found.
- <variable> specifies the struct variable name.

Options

This command has no options.

Example 1

The following example shows the struct variable `a` in function `top` will be disaggregated:

```
set_directive_disaggregate top a
```

Example 2

Disaggregated structs can be addressed in your code by the using standard C/C++ coding style as shown below. Notice the different methods for accessing the pointer element (`a`) versus the reference element (`c`);

```
struct SS
{
    int x[N];
    int y[N];
};

int top(SS *a, int b[4][6], SS &c) {

    set_directive_disaggregate top a
    set_directive_interface -mode s_axilite top a->x
    set_directive_interface -mode s_axilite top a->y

    set_directive_disaggregate top c
    set_directive_interface -mode ap_memory top c.x
    set_directive_interface -mode ap_memory top c.y
}
```

Example 3

The following example shows the `Dot` struct containing the `RGB` struct as an element. If you apply `set_directive_disaggregate` to variable `Arr`, then only the top-level `Dot` struct is disaggregated.

```
struct Pixel {
char R;
char G;
char B;
};

struct Dot {
Pixel RGB;
unsigned Size;
};
```

```
#define N 1086
void DUT(Dot Arr[N]) {
...
}
set_directive_disaggregate DUT Arr
```

If you want to disaggregate the whole struct, Dot and RGB, then you can assign the `set_directive_disaggregate` as shown below.

```
void DUT(Dot Arr[N]) {
#pragma HLS disaggregate variable=Arr->RGB
...
}
set_directive_disaggregate DUT Arr->RGB
```

The results in this case will be:

```
void DUT(char Arr_RGB_R[N], char Arr_RGB_G[N], char Arr_RGB_B[N], unsigned
Arr_Size[N]) {
...
}
```

See Also

- [pragma HLS disaggregate](#)
- [set_directive_aggregate](#)

set_directive_expression_balance

Description

Sometimes C/C++ code is written with a sequence of operations, resulting in a long chain of operations in RTL. With a small clock period, this can increase the latency in the design. By default, the Vitis HLS tool rearranges the operations using associative and commutative properties. As described in [Optimizing Logic Expressions](#), this rearrangement creates a balanced tree that can shorten the chain, potentially reducing latency in the design at the cost of extra hardware.

Expression balancing rearranges operators to construct a balanced tree and reduce latency.

- For integer operations expression balancing is on by default but may be disabled.
- For floating-point operations, expression balancing is off by default but may be enabled.

The `set_directive_expression_balance` command allows this expression balancing to be turned off, or on, within a specified scope.

Syntax

```
set_directive_expression_balance [OPTIONS] <location>
```

- <location> is the location (in the format `function[/label]`) where expression balancing should be disabled, or enabled.

Options

- `-off`: Turns off expression balancing at the specified location. Specifying the `set_directive_expression_balance` command enables expression balancing in the specified scope. Adding the `-off` option disables it.

Examples

Disables expression balancing within function `My_Func`.

```
set_directive_expression_balance -off My_Func
```

Explicitly enables expression balancing in function `My_Func2`.

```
set_directive_expression_balance My_Func2
```

See Also

- [pragma HLS expression_balance](#)

set_directive_function_instantiate

Description

By default:

- Functions remain as separate hierarchy blocks in the RTL, or are decomposed (inlined) into higher-level functions.
- All instances of a function, at the same level of hierarchy, uses the same RTL implementation (block).

The `set_directive_function_instantiate` command is used to create a unique RTL implementation for each instance of a function, allowing each instance to be optimized around a specific argument or variable.

By default, the following code results in a single RTL implementation of function `func_sub` for all three instances, or if `func_sub` is a small function it is inlined into function `func`.



TIP: By default, the Vitis HLS tool automatically inlines small functions. This is true even for function instantiations. Using the `set_directive_inline off` option can be used to prevent this automatic inlining.

```
char func_sub(char inval, char incr)
{
    return inval + incr;
}
void func(char inval1, char inval2, char inval3,
          char *outval1, char *outval2, char * outval3)
{
    *outval1 = func_sub(inval1, 1);
    *outval2 = func_sub(inval2, 2);
    *outval3 = func_sub(inval3, 3);
}
```

Using the directive as shown in the example section below results in three versions of function `func_sub`, each independently optimized for variable `incr`.

Syntax

```
set_directive_function_instantiate <location> <variable>
```

- `<location>` is the location (in the format `function[/region label]`) where the instances of a function are to be made unique.
- `<variable>` specifies the function argument to be specified as a constant in the various function instantiations.

Options

This command has no options.

Examples

For the example code shown above, the following Tcl (or pragma placed in function `func_sub`) allows each instance of function `func_sub` to be independently optimized with respect to input `incr`.

```
set_directive_inline -off func_sub
set_directive_function_instantiate func_sub incr
```

See Also

- [pragma HLS function_instantiate](#)
- [set_directive_allocation](#)
- [set_directive_inline](#)

set_directive_inline

Description

Removes a function as a separate entity in the RTL hierarchy. After inlining, the function is dissolved into the calling function, and no longer appears as a separate level of hierarchy.



IMPORTANT! *Inlining a child function also dissolves any pragmas or directives applied to that function. In Vitis HLS, any directives applied in the child context are ignored.*

In some cases, inlining a function allows operations within the function to be shared and optimized more effectively with the calling function. However, an inlined function cannot be shared or reused, so if the parent function calls the inlined function multiple times, this can increase the area and resource utilization.

By default, inlining is only performed on the next level of function hierarchy.

Syntax

```
set_directive_inline [OPTIONS] <location>
```

- <location> is the location (in the format `function[/label]`) where inlining is to be performed.

Options

- **-off:** By default, Vitis HLS performs inlining of smaller functions in the code. Using the `-off` option disables inlining for the specified function.
- **-recursive:** By default, only one level of function inlining is performed. The functions within the specified function are not inlined. The `-recursive` option inlines all functions recursively within the specified function hierarchy.

Examples

The following example inlines function `func_sub1`, but no sub-functions called by `func_sub1`.

```
set_directive_inline func_sub1
```

This example inlines function `func_sub1`, recursively down the hierarchy, except function `func_sub2`:

```
set_directive_inline -recursive func_sub1
set_directive_inline -off func_sub2
```

See Also

- [pragma HLS inline](#)
- [set_directive_allocation](#)

set_directive_interface

Description



IMPORTANT! The INTERFACE pragma or directive is only supported for use on the top-level function, and cannot be used for sub-functions of the HLS design. The Vitis HLS tool automatically determines the I/O protocol used by any sub-functions.

The INTERFACE pragma or directive specifies how RTL ports are created from the function arguments during interface synthesis. For more information, see [Defining Interfaces](#). The ports in the RTL implementation are derived from:

- Any function-level protocol that is specified.
- Function arguments and return.



TIP: Global variables required on the interface must be explicitly defined as an argument of the top-level function as described in [Global Variables](#). If a global variable is accessed, but all read and write operations are local to the design, the resource is created in the design. There is no need for an I/O port in the RTL.

Function-level handshakes:

- Control when the function starts operation.
- Indicate when function operation:
 - Ends
 - Is idle
 - Is ready for new inputs

The implementation of a function-level protocol:

- Is controlled by modes ap_ctrl_chain, ap_ctrl_hs, or ap_ctrl_none.
- Requires only the top-level function name.

Each function argument can be specified to have its own I/O protocol (such as valid handshake or acknowledge handshake).

Syntax

```
set_directive_interface [OPTIONS] <location> <port>
```

- <location> is the location (in the format `function[/label]`) where the function interface or registered output is to be specified.
- <port> is the parameter (function argument) for which the interface has to be synthesized. The port name is not required when block control modes are specified: `ap_ctrl_chain`, `ap_ctrl_hs`, or `ap_ctrl_none`.

Options



TIP: Many of the options specified below have global values that are defined in the [config_interface](#) command. Set local values for the interface defined here to override the global values.

- **-bundle <string>:** By default, the HLS tool groups or bundles function arguments with compatible options into interface ports in the RTL code. All AXI4-Lite (`s_axilite`) interfaces are bundled into a single AXI4-Lite port whenever possible. Similarly, all function arguments specified as an AXI4 (`m_axi`) interface are bundled into a single AXI4 port by default. All interface ports with compatible options, such as `mode`, `offset`, and `bundle`, are grouped into a single interface port. The port name is derived automatically from a combination of the mode and bundle, or is named as specified by `-name`.



IMPORTANT! When specifying the `bundle` name you should use all lower-case characters.

- **-clock <string>:** By default, the AXI4-Lite interface clock is the same clock as the system clock. This option is used to set specify a separate clock for an AXI4-Lite interface. If the `-bundle` option is used to group multiple top-level function arguments into a single AXI4-Lite interface, the clock option need only be specified on one of bundle members.
- **-depth <int>:** Specifies the maximum number of samples for the test bench to process. This setting indicates the maximum size of the FIFO needed in the verification adapter that the HLS tool creates for RTL co-simulation.



TIP: While `depth` is usually an option, it is required for `m_axi` interfaces and determines the amount of resources allocated for the adapter as explained in [AXI4 Master Interface](#).

- **-latency <value>:** This option can be used on `ap_memory` and `M_AXI` interfaces.
 - In an `ap_memory` interface, the interface option specifies the read latency of the RAM resource driving the interface. By default, a read operation of 1 clock cycle is used. This option allows an external RAM with more than 1 clock cycle of read latency to be modeled.
 - In an `M_AXI` interface, this option specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request `<value>` number of cycles (latency) before the read or write is expected. If this figure is too low, the design will be ready too soon and may stall waiting for the bus. If this figure is too high, bus access may be idle waiting on the design to start the access.

- `-max_read_burst_length <int>`: For use with the M_AXI interface, this option specifies the maximum number of data values read during a burst transfer.
- `-max_widen_bitwidth <int>`: Specifies the maximum bit width available for the interface when automatically widening the interface. This overrides the global value specified by the `config_interface -m_axi_max_bitwidth` command.
- `-max_write_burst_length <int>`: For use with the M_AXI interface, this option specifies the maximum number of data values written during a burst transfer.
- `-mode (ap_none|ap_vld|ap_ack|ap_hs|ap_ovld|ap_fifo|ap_memory|bram|axis|s_axilite|m_axi|ap_ctrl_none|ap_ctrl_hs|ap_ctrl_chain|ap_stable)`: Following is a summary of how Vitis HLS implements the `-mode` options.
 - `ap_none`: No protocol. The interface is a data port.
 - `ap_vld`: Implements the data port with an associated `valid` port to indicate when the data is valid for reading or writing.
 - `ap_ack`: Implements the data port with an associated `acknowledge` port to acknowledge that the data was read or written.
 - `ap_hs`: Implements the data port with associated `valid` and `acknowledge` ports to provide a two-way handshake to indicate when the data is valid for reading and writing and to acknowledge that the data was read or written.
 - `ap_ovld`: Implements the output data port with an associated `valid` port to indicate when the data is valid for reading or writing.

Note: Vitis HLS implements the input argument or the input half of any read/write arguments with mode `ap_none`.

- `ap_fifo`: Implements the port with a standard FIFO interface using data input and output ports with associated active-Low FIFO `empty` and `full` ports.

Note: You can only use this interface on read arguments or write arguments. The `ap_fifo` mode does not support bidirectional read/write arguments.

- `ap_memory`: Implements array arguments as a standard RAM interface. If you use the RTL design in Vivado IP integrator, the memory interface appears as discrete ports.
- `bram`: Implements array arguments as a standard RAM interface. If you use the RTL design in Vitis IP integrator, the memory interface appears as a single port.
- `axis`: Implements all ports as an AXI4-Stream interface.
- `s_axilite`: Implements all ports as an AXI4-Lite interface. Vitis HLS produces an associated set of C driver files during the Export RTL process.
- `m_axi`: Implements all ports as an AXI4 interface. You can use the `config_interface` command to specify either 32-bit (default) or 64-bit address ports and to control any address offset.
- `ap_ctrl_none`: No block-level I/O protocol.

Note: Using the `ap_ctrl_none` mode might prevent the design from being verified using the C/C++/RTL co-simulation feature.

- `ap_ctrl_hs`: Implements a set of block-level control ports to `start` the design operation and to indicate when the design is `idle`, `done`, and `ready` for new input data.

Note: The `ap_ctrl_hs` mode is the default block-level I/O protocol.

- `ap_ctrl_chain`: Implements a set of block-level control ports to `start` the design operation, `continue` operation, and indicate when the design is `idle`, `done`, and `ready` for new input data.
- `ap_stable`: No protocol. The interface is a data port. Vitis HLS assumes the data port is always stable after reset, which allows internal optimizations to remove unnecessary registers.

- `-name <string>`: Specifies a name for the port which will be used in the generated RTL.
- `-num_read_outstanding <int>`: For use with the M_AXI interface, this option specifies how many read requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, and a FIFO of size:

```
num_read_outstanding*max_read_burst_length*word_size
```

- `-num_write_outstanding <int>`: For use with the M_AXI interface, this option specifies how many write requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, and a FIFO of size:

```
num_read_outstanding*max_read_burst_length*word_size
```

- `-offset <string>`: Controls the address offset in AXI4-Lite (`s_axilite`) and AXI4 memory mapped (`m_axi`) interfaces for the specified port.
 - In an `s_axilite` interface, `<string>` specifies the address in the register map.
 - In an `m_axi` interface this option overrides the global option specified by the `config_interface -m_axi_offset` option, and `<string>` is specified as:
 - `off`: Do not generate an offset port.
 - `direct`: Generate a scalar input offset port.
 - `slave`: Generate an offset port and automatically map it to an AXI4-Lite slave interface. This is the default offset.
- `-register`: Registers the signal and any associated protocol signals and instructs the signals to persist until at least the last cycle of the function execution. The `-register_io` option of the `config_interface` command globally controls registering all inputs/outputs on the top function. This option applies to the following scalar interfaces for the top-level function:
 - `s_axilite`
 - `ap_fifo`

- ap_none
- ap_stable
- ap_hs
- ap_ack
- ap_vld
- ap_ovld

 **TIP:** The use of the register option on the return port of the function (`port=return`) is not supported. Instead use the `LATENCY` pragma or directive:

```
#pragma HLS LATENCY min=1 max=1
```

- **-register_mode** (`both|forward|reverse|off`): This option applies to AXI4-Stream interfaces, and specifies if registers are placed on the forward path (TDATA and TVALID), the reverse path (TREADY), on both paths (TDATA, TVALID, and TREADY), or if none of the ports signals are to be registered (`off`). The default is `both`. AXI4-Stream side-channel signals are considered to be data signals and are registered whenever the TDATA is registered.
- **-storage_impl=<impl>**: For use with `s_axilite` only. This options defines a storage implementation to assign to the interface. Supported implementation values include `auto`, `bram`, and `uram`. The default is `auto`.

 **TIP:** `uram` is a synchronous memory with only a single clock for two ports. Therefore `uram` cannot be specified for an `s_axilite` adapter with a second clock.

- **-storage_type=<type>**: For use with `ap_memory` and `bram` interfaces only. This options defines a storage type (for example, `RAM_T2P`) to assign to the variable. Supported types include: `ram_1p`, `ram_1wnr`, `ram_2p`, `ram_s2p`, `ram_t2p`, `rom_1p`, `rom_2p`, and `rom_np`.

 **TIP:** This can also be specified using the `BIND_STORAGE` pragma or directive for the variable for objects that are not defined on the interface.

Examples

Turns off function-level handshakes for function `func`.

```
set_directive_interface -mode ap_ctrl_none func
```

Argument `InData` in function `func` is specified to have a `ap_vld` interface and the input should be registered.

```
set_directive_interface -mode ap_vld -register func InData
```

See Also

- [pragma HLS interface](#)
- [set_directive_bind_storage](#)

set_directive_latency

Description

Specifies a maximum or minimum latency value, or both, on a function, loop, or region.

Vitis HLS always aims for minimum latency. The behavior of the tool when minimum and maximum latency values are specified is as follows:

- Latency is less than the minimum: If Vitis HLS can achieve less than the minimum specified latency, it extends the latency to the specified value, potentially enabling increased sharing.
- Latency is greater than the minimum: The constraint is satisfied. No further optimizations are performed.
- Latency is less than the maximum: The constraint is satisfied. No further optimizations are performed.
- Latency is greater than the maximum: If Vitis HLS cannot schedule within the maximum limit, it increases effort to achieve the specified constraint. If it still fails to meet the maximum latency, it issues a warning. Vitis HLS then produces a design with the smallest achievable latency.



TIP: You can also use the LATENCY pragma or directive to limit the efforts of the tool to find an optimum solution. Specifying latency constraints for scopes within the code: loops, functions, or regions, reduces the possible solutions within that scope and can improve tool runtime. Refer to [Improving Synthesis Runtime and Capacity](#) for more information.

If the intention is to limit the total latency of all loop iterations, the latency directive should be applied to a region that encompasses the entire loop, as in this example: `set_directive Region_All_Loop_A`

```
Region_All_Loop_A: {  
Loop_A: for (i=0; i<N; i++)  
{  
    ..Loop Body...  
}  
}
```

In this case, even if the loop is unrolled, the latency directive sets a maximum limit on all loop operations.

If Vitis HLS cannot meet a maximum latency constraint it relaxes the latency constraint and tries to achieve the best possible result.

If a minimum latency constraint is set and Vitis HLS can produce a design with a lower latency than the minimum required it inserts dummy clock cycles to meet the minimum latency.

Syntax

```
set_directive_latency [OPTIONS] <location>
```

- <location> is the location (function, loop or region) (in the format `function[/label]`) to be constrained.

Options

- `-max <integer>`: Specifies the maximum latency.
- `-min <integer>`: Specifies the minimum latency.

Examples

Function `foo` is specified to have a minimum latency of 4 and a maximum latency of 8.

```
set_directive_latency -min=4 -max=8 foo
```

In function `foo`, loop `loop_row` is specified to have a maximum latency of 12. Place the pragma in the loop body.

```
set_directive_latency -max=12 foo/loop_row
```

See Also

- [pragma HLS latency](#)

set_directive_loop_flatten

Description

Flattens nested loops into a single loop hierarchy.

In the RTL implementation, it costs a clock cycle to move between loops in the loop hierarchy. Flattening nested loops allows them to be optimized as a single loop. This saves clock cycles, potentially allowing for greater optimization of the loop body logic.



RECOMMENDED: Apply this directive to the inner-most loop in the loop hierarchy. Only perfect and semi-perfect loops can be flattened in this manner.

- Perfect loop nests
 - Only the innermost loop has loop body content.
 - There is no logic specified between the loop statements.
 - All loop bounds are constant.
- Semi-perfect loop nests
 - Only the innermost loop has loop body content.
 - There is no logic specified between the loop statements.
 - The outermost loop bound can be a variable.
- Imperfect loop nests

When the inner loop has variables bounds (or the loop body is not exclusively inside the inner loop), try to restructure the code, or unroll the loops in the loop body to create a perfect loop nest.

Syntax

```
set_directive_loop_flatten [OPTIONS] <location>
```

- <location> is the location (inner-most loop), in the format `function[/label]`.

Options

- `-off`: Option to prevent loop flattening from taking place, and can prevent some loops from being flattened while all others in the specified location are flattened.



IMPORTANT! The presence of the `LOOP_FLATTEN` pragma or directive enables the optimization. The addition of `-off` disables it.

Examples

Flattens `loop_1` in function `foo` and all (perfect or semi-perfect) loops above it in the loop hierarchy, into a single loop. Place the pragma in the body of `loop_1`.

```
set_directive_loop_flatten foo/loop_1
#pragma HLS loop_flatten
```

Prevents loop flattening in `loop_2` of function `foo`. Place the pragma in the body of `loop_2`.

```
set_directive_loop_flatten -off foo/loop_2
#pragma HLS loop_flatten off
```

See Also

- [set_directive_loop_flatten](#)

- `pragma HLS loop_merge`
 - `pragma HLS unroll`
-

set_directive_loop_merge

Description

Merges all loops into a single loop. Merging loops:

- Reduces the number of clock cycles required in the RTL to transition between the loop-body implementations.
- Allows the loops be implemented in parallel (if possible).

The rules for loop merging are:

- If the loop bounds are variables, they must have the same value (number of iterations).
- If loops bounds are constants, the maximum constant value is used as the bound of the merged loop.
- Loops with both variable bound and constant bound cannot be merged.
- The code between loops to be merged cannot have side effects. Multiple execution of this code should generate the same results.
 - `a = b` is allowed
 - `a = a + 1` is not allowed.
- Loops cannot be merged when they contain FIFO reads. Merging changes the order of the reads. Reads from a FIFO or FIFO interface must always be in sequence.

Syntax

```
set_directive_loop_merge <location>
```

- `<location>` is the location (in the format `function[/label]`) at which the loops reside.

Options

- `-force`: Forces loops to be merged even when Vitis HLS issues a warning. You must assure that the merged loop will function correctly.

Examples

Merges all consecutive loops in function `foo` into a single loop.

```
set_directive_loop_merge foo
```

All loops inside `loop_2` of function `foo` (but not `loop_2` itself) are merged by using the `-force` option.

```
set_directive_loop_merge -force foo/loop_2
```

See Also

- [pragma HLS loop_merge](#)
- [set_directive_loop_flatten](#)
- [set_directive_unroll](#)

set_directive_loop_tripcount

Description

The *loop tripcount* is the total number of iterations performed by a loop. Vitis HLS reports the total latency of each loop (the number of cycles to execute all iterations of the loop). This loop latency is therefore a function of the tripcount (number of loop iterations).



IMPORTANT! The `LOOP_TRIPCOUNT` pragma or directive is for analysis only, and does not impact the results of synthesis.

The tripcount can be a constant value. It might depend on the value of variables used in the loop expression (for example, `x < y`) or control statements used inside the loop.

Vitis HLS cannot determine the tripcount in some cases. These cases include, for example, those in which the variables used to determine the tripcount are:

- Input arguments, or
- Variables calculated by dynamic operation

In the following example, the maximum iteration of the for-loop is determined by the value of input `num_samples`. The value of `num_samples` is not defined in the C function, but comes into the function from the outside.

```
void foo (num_samples, ...) {
    int i;
    ...
    loop_1: for(i=0;i< num_samples;i++) {
        ...
        result = a + b;
    }
}
```

In cases where the loop latency is unknown or cannot be calculated, `set_directive_loop_tripcount` allows you to specify minimum, maximum, and average iterations for a loop. This lets the tool analyze how the loop latency contributes to the total design latency in the reports and helps you determine appropriate optimizations for the design.



TIP: If a C assert macro is used to limit the size of a loop variable, Vitis HLS can use it to both define loop limits for reporting and create hardware that is exactly sized to these limits.

Syntax

```
set_directive_loop_tripcount [OPTIONS] <location>
```

- `<location>` is the location of the loop (in the format `function[/label]`) at which the tripcount is specified.

Options

- `-avg <integer>`: Specifies the average number of iterations.
- `-max <integer>`: Specifies the maximum number of iterations.
- `-min <integer>`: Specifies the minimum number of iterations.

Examples

`loop_1` in function `foo` is specified to have a minimum tripcount of 12, and a maximum tripcount of 16:

```
set_directive_loop_tripcount -min 12 -max 16 -avg 14 foo/loop_1
```

See Also

- [pragma HLS loop_tripcount](#)

set_directive_occurrence

Description

When pipelining functions or loops, the OCCURRENCE directive specifies that the code in a pipelined function call within the pipelined function or loop is executed at a lower rate than the surrounding function or loop. This allows the pipelined call that is executed at the lower rate to be pipelined at a slower rate, and potentially shared within the top-level pipeline. For example:

- A loop iterates N times.
- Part of the loop is protected by a conditional statement and only executes M times, where N is an integer multiple of M .
- The code protected by the conditional is said to have an occurrence of N/M .

Identifying a region with an OCCURRENCE rate allows the functions and loops in this region to be pipelined with an initiation interval that is slower than the enclosing function or loop.

Syntax

```
set_directive_occurrence [OPTIONS] <location>
```

- $<\text{location}>$ specifies the block of code that contains the pipelined function call(s) with a slower rate of execution.

Options

- `-cycle <int>`: Specifies the occurrence N/M where:
 - N is the number of times the enclosing function or loop is executed.
 - M is the number of times the conditional region is executed.



IMPORTANT! N must be an integer multiple of M .

Examples

Region Cond_Region in function foo has an occurrence of 4. It executes at a rate four times slower than the code that encompasses it.

```
set_directive_occurrence -cycle 4 foo/Cond_Region
```

See Also

- [pragma HLS occurrence](#)
- [set_directive_pipeline](#)

set_directive_performance

Description

Note: `set_directive_performance` applies to loops and loop nests, and requires a known loop tripcount to determine the performance. If your loop has a variable tripcount then you must also specify `set_directive_tripcount`.

The `set_directive_performance` lets you specify a high-level constraint (`target_ti`) defining the number of clock cycles between successive starts of a loop, and lets the tool infer lower-level UNROLL, PIPELINE, ARRAY_PARTITION, and INLINE directives needed to achieve the desired result. The `set_directive_performance` does not guarantee the specified value will be achieved, and so it is only a target.

Note: `set_directive_inline` is applied automatically to functions inside any pipelined loop that has `II=1` to improve throughput. If you apply the `PERFORMANCE` directive that infers a pipeline with `II=1`, it will also trigger the auto-inline optimization. You can disable this for specific functions by using `set_directive_inline off`.

The target transaction interval (`target_ti`) specifies a performance target for loops, where a transaction is a complete set of loop iterations (tripcount) and the interval is the time between when the first transaction starts and the second transaction starts.

- **Target Transaction Interval (`target_ti`):** Specifies the number of clock cycles from the first transaction of a loop, or nested loop, and the start of the next transaction of the loop. Specified as the cycles needed for the loop to complete all iterations and begin the next transaction.

The transaction interval is the initiation interval (`II`) of the loop times the number of iterations, or tripcount: $\text{target_ti} = \text{II} * \text{loop tripcount}$. Conversely, $\text{target_ti} = \text{FreqHz} / \text{Operations per second}$.

For example, assuming an image processing function that processes a single frame per invocation with a throughput goal of 60 fps, then the target throughput for the function is 60 invocations per second. If the clock frequency is 180 MHz, then `target_ti` is $180\text{M}/60$, or 3 million clock cycles per function invocation.

Vitis HLS always tries to achieve the specified performance target in the design. When `set_directive_performance` is specified, the tool automatically applies pragmas or directives such as PIPELINE, UNROLL, or ARRAY_PARTITION to achieve the `target_ti`.

Syntax

Specify the directive for a function, or a labeled loop, or region of code.

```
set_directive_performance <label> -target_ti=<value>
```

Where:

- `-target_tti=<value>`: Specifies a target transaction interval defined as the number of clock cycles for the loop to complete an iteration. The `<value>` can be specified as an integer, floating point, or constant expression that is resolved by the tool as an integer.

Note: A warning will be returned if truncation occurs.

Example 1

The loop labeled `loop_1` is specified to have target transaction interval of 4 clock cycles:

```
set_directive_performance loop_1-target_tti=4
```

See Also

- [pragma HLS performance](#)
- [set_directive_inline](#)

set_directive_pipeline

Description

Reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations as described in [Pipelining Loops](#). A pipelined function or loop can process new inputs every N clock cycles, where N is the initiation interval (II). An II of 1 processes a new input every clock cycle.

As a default behavior, with the PIPELINE pragma or directive Vitis HLS will generate the minimum II for the design according to the specified clock period constraint. The emphasis will be on meeting timing, rather than on achieving II unless the `-II` option is specified.

The default type of pipeline is defined by the `config_compile -pipeline_style` command, but can be overridden in the PIPELINE pragma or directive.

If Vitis HLS cannot create a design with the specified II, it:

- Issues a warning.
- Creates a design with the lowest possible II.

You can then analyze this design with the warning messages to determine what steps must be taken to create a design that satisfies the required initiation interval.

Syntax

```
set_directive_pipeline [OPTIONS] <location>
```

Where:

- <location> is the location (in the format `function[/label]`) to be pipelined.

Options

- **-II <integer>**: Specifies the desired initiation interval for the pipeline. Vitis HLS tries to meet this request. Based on data dependencies, the actual result might have a larger II.
- **-off**: Turns off pipeline for a specific loop or function. This can be used when `config_compile -pipeline_loops` is used to globally pipeline loops.
- **-rewind**:

Note: Applicable only to a loop.

Optional keyword. Enables rewinding as described in [Rewinding Pipelined Loops for Performance](#). This enables continuous loop pipelining with no pause between one execution of the loop ending and the next execution starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop:

- Is considered as initialization.
 - Is executed only once in the pipeline.
 - Cannot contain any conditional operations (`if-else`).
- **-style <stp | frp | f1p>**: Specifies the type of pipeline to use for the specified function or loop. For more information on pipeline styles refer to [Flushing Pipelines and Pipeline Types](#). The types of pipelines include:
- **stp**: Stall pipeline. Runs only when input data is available otherwise it stalls. This is the default setting, and is the type of pipeline used by Vitis HLS for both loop and function pipelining. Use this when a flushable pipeline is not required. For example, when there are no performance or deadlock issue due to stalls.
 - **f1p**: This option defines the pipeline as a flushable pipeline. This type of pipeline typically consumes more resources and/or can have a larger II because resources cannot be shared among pipeline iterations.
 - **frp**: Free-running, flushable pipeline. Runs even when input data is not available. Use this when you need better timing due to reduced pipeline control signal fanout, or when you need improved performance to avoid deadlocks. However, this pipeline style can consume more power as the pipeline registers are clocked even if there is no data.



IMPORTANT! This is a hint not a hard constraint. The tool checks design conditions for enabling pipelining. Some loops might not conform to a particular style and the tool reverts to the default style (*stp*) if necessary.

Examples

Function `func` is pipelined with the specified initiation interval.

```
set_directive_pipeline func II=1
```

See Also

- [pragma HLS pipeline](#)
- [set_directive_dependence](#)
- [config_compile](#)

set_directive_protocol

Description

This command specifies a region of code, a protocol region, in which no clock operations will be inserted by Vitis HLS unless explicitly specified in the code. Vitis HLS will not insert any clocks between operations in the region, including those which read from or write to function arguments. The order of read and writes will therefore be strictly followed in the synthesized RTL.

A region of code can be created in the C/C++ code by enclosing the region in braces "{}" and naming it. The following defines a region named `io_section`:

```
io_section:{  
...  
lines of code  
...  
}
```

A clock operation can be explicitly specified in C code using an `ap_wait()` statement, and can be specified in C++ code by using the `wait()` statement. The `ap_wait` and `wait` statements have no effect on the simulation of the design.

Syntax

```
set_directive_protocol [OPTIONS] <location>
```

The <location> specifies the location (in the format `function[/label]`) at which the protocol region is defined.

Options

- `-mode [floating | fixed]:`
 - `floating`: Lets code statements outside the protocol region overlap and execute in parallel with statements in the protocol region in the final RTL. The protocol region remains cycle accurate, but outside operations can occur at the same time. This is the default mode.
 - `fixed`: The fixed mode ensures that statements outside the protocol region do not execute in parallel with the protocol region.

Examples

The example code defines a protocol region, `io_section` in function `foo`. The following directive defines that region as a fixed mode protocol region:

```
set_directive_protocol -mode fixed foo/io_section
```

See Also

- [pragma HLS protocol](#)

set_directive_reset

Description

Adds or removes resets for specific state variables (global or static). The reset port is used to restore the registers and block RAM, connected to the port, to an initial value any time the reset signal is applied. The presence and behavior of the RTL reset port is controlled using the `config_rtl` settings.

Greater control over reset is provided through the RESET pragma. If a variable is a static or global, the RESET pragma is used to explicitly add a reset, or the variable can be removed from the reset by turning `off` the pragma. This can be particularly useful when static or global arrays are present in the design.

Syntax

```
set_directive_reset [OPTIONS] <location> <variable>
```

- <location> is the location (in the format `function[/label]`) at which the variable is defined.

- <variable> is the variable to which the directive is applied.

Options

- **-off:**
 - If `-off` is specified, reset is *not* generated for the specified variable.

Examples

Adds reset to variable `a` in function `foo` even when the global reset setting is `none` or `control`.

```
set_directive_reset foo a
```

Removes reset from variable `static int a` in function `foo` even when the global reset setting is `state` or `all`.

```
set_directive_reset -off foo a
```

See Also

- [pragma HLS reset](#)
- [config_rtl](#)

set_directive_stable

Description

The STABLE pragma is applied to arguments of a DATAFLOW or PIPELINE region and is used to indicate that an input or output of this region can be ignored when generating the synchronizations at entry and exit of the DATAFLOW region. This means that the reading processes (resp. read accesses) of that argument do not need to be part of the “first stage” of the task-level (resp. fine-grain) pipeline for inputs, and the writing process (resp. write accesses) do not need to be part of the last stage of the task-level (resp. fine-grain) pipeline for outputs.

The pragma can be specified at any point in the hierarchy, on a scalar or an array, and automatically applies to all the DATAFLOW or PIPELINE regions below that point. The effect of STABLE for an input is that a DATAFLOW or PIPELINE region can start another iteration even though the value of the previous iteration has not been read yet. For an output, this implies that a write of the next iteration can occur although the previous iteration is not done.

Syntax

```
set_directive_stable <location> <variable>
```

- <location> is the function name or loop name where the directive is to be constrained.
- <variable> is the name of the array to be constrained.

Examples

In the following example, without the STABLE directive, proc1 and proc2 would be synchronized to acknowledge the reading of their inputs (including A). With the directive, A is no longer considered as an input that needs synchronization.

```
void dataflow_region(int A[...], int B[...] ...  
    proc1(...);  
    proc2(A, ...);
```

The directives for this example would be scripted as:

```
set_directive_stable dataflow_region variable=A  
set_directive_dataflow dataflow_region
```

See Also

- [pragma HLS stable](#)
- [set_directive_dataflow](#)
- [set_directive_pipeline](#)

set_directive_stream

Description

By default, array variables are implemented as RAM:

- Top-level function array parameters are implemented as a RAM interface port.
- General arrays are implemented as RAMs for read-write access.
- Arrays involved in sub-functions, or loop-based DATAFLOW optimizations are implemented as a RAM ping-pong buffer channel.

If the data stored in the array is consumed or produced in a sequential manner, a more efficient communication mechanism is to use streaming data, where FIFOs are used instead of RAMs. When an argument of the top-level function is specified as INTERFACE type ap_fifo, the array is automatically implemented as streaming. See [Defining Interfaces](#) for more information.



IMPORTANT! To preserve the accesses, it might be necessary to prevent compiler optimizations (in particular dead code elimination) by using the `volatile` qualifier as described in [Type Qualifiers](#).

Syntax

```
set_directive_stream [OPTIONS] <location> <variable>
```

- <location> is the location (in the format `function[/label]`) which contains the array variable.
- <variable> is the array variable to be implemented as a FIFO.

Options

- `-depth <integer>`:

Note: Relevant only for array streaming in dataflow channels.

By default, the depth of the FIFO implemented in the RTL is the same size as the array specified in the C code. This option allows you to modify the size of the FIFO.

When the array is implemented in a DATAFLOW region, it is common to use the `-depth` option to reduce the size of the FIFO. For example, in a DATAFLOW region where all loops and functions are processing data at a rate of $\text{II} = 1$, there is no need for a large FIFO because data is produced and consumed in each clock cycle. In this case, the `-depth` option may be used to reduce the FIFO size to 2 to substantially reduce the area of the RTL design.

This same functionality is provided for all arrays in a DATAFLOW region using the `config_dataflow` command with the `-depth` option. The `-depth` option used with `set_directive_stream` overrides the default specified using `config_dataflow`.

- **-type <arg>**: Specify a mechanism to select between FIFO, PIPO, synchronized shared (`shared`), and un-synchronized shared (`unsync`). The supported types include:
 - `fifo`: A FIFO buffer with the specified `depth`.
 - `piro`: A regular Ping-Pong buffer, with as many “banks” as the specified depth (default is 2).
 - `shared`: A shared channel, synchronized like a regular Ping-Pong buffer, with depth, but without duplicating the array data. Consistency can be ensured by setting the depth small enough, which acts as the distance of synchronization between the producer and consumer.



TIP: The default depth for `shared` is 1.

- `unsync`: Does not have any synchronization except for individual memory reads and writes. Consistency (read-write and write-write order) must be ensured by the design itself.

Examples

Specifies array A[10] in function func to be streaming and implemented as a FIFO.

```
set_directive_stream func A -type fifo
```

Array B in named loop loop_1 of function func is set to streaming with a FIFO depth of 12. In this case, place the pragma inside loop_1.

```
set_directive_stream -depth 12 -type fifo func/loop_1 B
```

Array C has streaming implemented as a PIPO.

```
set_directive_stream -type pipo func C
```

See Also

- [pragma HLS stream](#)
- [set_directive_dataflow](#)
- [set_directive_interface](#)
- [config_dataflow](#)

set_directive_top

Description

Attaches a name to a function, which can then be used by the `set_top` command to set the named function as the top. This is typically used to synthesize member functions of a class in C++.



RECOMMENDED: Specify the directive in an active solution. Use the `set_top` command with the new name.

Syntax

```
set_directive_top [OPTIONS] <location>
```

- `<location>` is the function to be renamed.

Options

- `-name <string>`: Specifies the name of the function to be used by the `set_top` command.

Examples

Function `foo_long_name` is renamed to `DESIGN_TOP`, which is then specified as the top-level. If the pragma is placed in the code, the `set_top` command must still be issued in the top-level specified in the GUI project settings.

```
set_directive_top -name DESIGN_TOP foo_long_name
```

Followed by the `set_top DESIGN_TOP` command.

See Also

- [pragma HLS top](#)
- [set_top](#)

set_directive_unroll

Description

Transforms loops by creating multiples copies of the loop body.

A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations may also be impacted by logic inside the loop body (for example, break or modifications to any loop exit variable). The loop is implemented in the RTL by a block of logic representing the loop body, which is executed for the same number of iterations.

The `set_directive_unroll` command allows the loop to be fully unrolled. Unrolling the loop creates as many copies of the loop body in the RTL as there are loop iterations, or partially unrolled by a factor N , creating N copies of the loop body and adjusting the loop iteration accordingly.

If the factor N used for partial unrolling is not an integer multiple of the original loop iteration count, the original exit condition must be checked after each unrolled fragment of the loop body.

To unroll a loop completely, the loop bounds must be known at compile time. This is not required for partial unrolling.

Syntax

```
set_directive_unroll [OPTIONS] <location>
```

- `<location>` is the location of the loop (in the `format function[/label]`) to be unrolled.

Options

- **-factor <integer>**: Specifies a non-zero integer indicating that partial unrolling is requested.

The loop body is repeated this number of times. The iteration information is adjusted accordingly.

- **-skip_exit_check**: Effective only if a factor is specified (partial unrolling).

- Fixed bounds

No exit condition check is performed if the iteration count is a multiple of the factor.

If the iteration count is *not* an integer multiple of the factor, the tool:

- Prevents unrolling.
- Issues a warning that the exit check must be performed to proceed.

- Variable bounds

The exit condition check is removed. You must ensure that:

- The variable bounds is an integer multiple of the factor.
- No exit check is in fact required.

Examples

Unrolls loop L1 in function foo. Place the pragma in the body of loop L1.

```
set_directive_unroll foo/L1
```

Specifies an unroll factor of 4 on loop L2 of function foo. Removes the exit check. Place the pragma in the body of loop L2.

```
set_directive_unroll -skip_exit_check -factor 4 foo/L2
```

Unrolls all loops inside loop L3 in function foo, but not loop L3 itself. The **-region** option specifies the location be considered an enclosing region and not a loop label.

```
set_directive_unroll -region foo/L3
```

See Also

- [pragma HLS unroll](#)
- [set_directive_loop_flatten](#)
- [set_directive_loop_merge](#)

HLS Pragmas

Optimizations in Vitis HLS

In the Vitis™ software platform, a kernel defined in the C/C++ language, or OpenCL™ C, must be compiled into the register transfer level (RTL) that can be implemented into the programmable logic of a Xilinx® device. The `v++` compiler calls the Vitis High-Level Synthesis (HLS) tool to synthesize the RTL code from the kernel source code.

The HLS tool is intended to work with the Vitis IDE project without interaction. However, the HLS tool also provides pragmas that can be used to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code. These pragmas can be added directly to the source code for the kernel.

The HLS pragmas include the optimization types specified in the following table.

Table 33: Vitis HLS Pragmas by Type

Type	Attributes
Kernel Optimization	<ul style="list-style-type: none">• <code>pragma HLS aggregate</code>• <code>pragma HLS alias</code>• <code>pragma HLS disaggregate</code>• <code>pragma HLS expression_balance</code>• <code>pragma HLS latency</code>• <code>pragma HLS performance</code>• <code>pragma HLS protocol</code>• <code>pragma HLS reset</code>• <code>pragma HLS top</code>• <code>pragma HLS stable</code>
Function Inlining	<ul style="list-style-type: none">• <code>pragma HLS inline</code>
Interface Synthesis	<ul style="list-style-type: none">• <code>pragma HLS interface</code>• <code>pragma HLS stream</code>
Task-level Pipeline	<ul style="list-style-type: none">• <code>pragma HLS dataflow</code>• <code>pragma HLS stream</code>
Pipeline	<ul style="list-style-type: none">• <code>pragma HLS pipeline</code>• <code>pragma HLS occurrence</code>
Loop Unrolling	<ul style="list-style-type: none">• <code>pragma HLS unroll</code>• <code>pragma HLS dependence</code>

Table 33: Vitis HLS Pragmas by Type (cont'd)

Type	Attributes
Loop Optimization	<ul style="list-style-type: none">• <code>pragma HLS loop_flatten</code>• <code>pragma HLS loop_merge</code>• <code>pragma HLS loop_tripcount</code>
Array Optimization	<ul style="list-style-type: none">• <code>pragma HLS array_partition</code>• <code>pragma HLS array_reshape</code>
Structure Packing	<ul style="list-style-type: none">• <code>pragma HLS aggregate</code>• <code>pragma HLS dataflow</code>
Resource Utilization	<ul style="list-style-type: none">• <code>pragma HLS allocation</code>• <code>pragma HLS bind_op</code>• <code>pragma HLS bind_storage</code>• <code>pragma HLS function_instantiate</code>

pragma HLS aggregate

Description

Collects and groups the data fields of a struct into a single scalar with a wider word width.

The AGGREGATE pragma is used for grouping all the elements of a struct into a single wide vector to allow all members of the struct to be read and written to simultaneously. The bit alignment of the resulting new wide-word can be inferred from the declaration order of the struct elements. The first element takes the LSB of the vector, and the final element of the struct is aligned with the MSB of the vector.

If the struct contains arrays, the AGGREGATE pragma performs a similar operation as ARRAY reshape, and combines the reshaped array with the other elements in the struct. Any arrays declared inside the struct are completely partitioned and reshaped into a wide scalar and packed with other scalar elements.



IMPORTANT! You should exercise some caution when using the AGGREGATE optimization on struct objects with large arrays. If an array has 4096 elements of type int, this will result in a vector (and port) of width $4096 \times 32 = 131072$ bits. The Vitis HLS tool can create this RTL design, however it is very unlikely logic synthesis will be able to route this during the FPGA implementation.

Syntax

Place the pragma near the definition of the struct variable to aggregate:

```
#pragma HLS aggregate variable=<variable> compact=<arg>
```

Where:

- `variable=<variable>`: Specifies the variable to be grouped.
- `compact=[bit | byte | none | auto]`: Specifies the alignment of the aggregated struct. Alignment can be on the bit-level, the byte-level, none, or automatically determined by the tool which is the default behavior.

Example 1

Aggregates struct pointer `AB` with three 8-bit fields (`typedef struct {unsigned char R, G, B;} pixel`) in function `func`, into a new 24-bit pointer aligned on the bit-level.

```
typedef struct{
unsigned char R, G, B;
} pixel;

pixel AB;
#pragma HLS aggregate variable=AB compact=bit
```

Example 2

Aggregates struct array `AB[17]` with three 8-bit field fields (R, G, B) into a new 17 element array of 24-bits.

```
typedef struct{
unsigned char R, G, B;
} pixel;

pixel AB[17];
#pragma HLS aggregate variable=AB
```

See Also

- [set_directive_aggregate](#)
- [pragma HLS array_reshape](#)
- [pragma HLS disaggregate](#)

pragma HLS alias

Description

Specify that two or more `M_AXI` pointer arguments point to the same underlying buffer in memory (DDR or HBM) and indicate any aliasing between the pointers by setting the distance or offset between them.



IMPORTANT! The ALIAS pragma applies to top-level function arguments mapped to M_AXI interfaces.

Vitis HLS considers different pointers to be independent channels and generally does not provide any dependency analysis. However, in cases where the host allocates a single buffer for multiple pointers, this relationship can be communicated through the ALIAS pragma or directive and dependency analysis can be maintained. The ALIAS pragma enables data dependence analysis in Vitis HLS by defining the distance between pointers in the buffer.

Requirements for ALIAS:

- All ports assigned to an ALIAS pragma must be assigned to M_AXI interfaces and assigned to different bundles, as shown in the example below
- Each port can only be used in one ALIAS pragma or directive
- The depth of all ports assigned to an ALIAS pragma must be the same
- When offset is specified, the number of ports and number of offsets specified must be the same: one offset per port
- The offset for the INTERFACE must be specified as slave or direct, offset=off is not supported

Syntax

```
pragma HLS alias ports=<list> [distance=<int> | offset=<list...>]
```

Where:

- **ports=<list>**: specifies the ports to alias.
- **distance=<integer>**: Specifies the difference between the pointer values passed to the ports in the list.
- **offset=<list>**: Specifies the offset of the pointer passed to each port in the ports list with respect to the origin of the array.

Note: offset and distance are mutually exclusive.

Example

For the following function top:

```
void top(int *arr0, int *arr1, int *arr2, int *arr3, ...) {  
    #pragma HLS interface mode=m_axi port=arr0 bundle=hbm0 depth=0x40000000  
    #pragma HLS interface mode=m_axi port=arr1 bundle=hbm1 depth=0x40000000  
    #pragma HLS interface mode=m_axi port=arr2 bundle=hbm2 depth=0x40000000  
    #pragma HLS interface mode=m_axi port=arr3 bundle=hbm3 depth=0x40000000
```

The following pragma defines aliasing for the specified array pointers, and defines the distance between them:

```
#pragma HLS ALIAS ports=arr0,arr1,arr2,arr3 distance=10000000
```

Alternatively, the following pragma specifies the offset between pointers, to accomplish the same effect:

```
#pragma HLS ALIAS ports=arr0,arr1,arr2,arr3  
offset=00000000,10000000,20000000,30000000
```

See Also

- [set_directive_alias](#)
- [pragma HLS interface](#)

pragma HLS allocation

Description

Specifies restrictions to limit resource allocation in the implemented kernel. The ALLOCATION pragma or directive can limit the number of RTL instances and hardware resources used to implement specific functions, loops, or operations. The ALLOCATION pragma is specified inside the body of a function, a loop, or a region of code.

For example, if the C source has four instances of a function `foo_sub`, the ALLOCATION pragma can ensure that there is only one instance of `foo_sub` in the final RTL. All four instances of the C function are implemented using the same RTL block. This reduces resources used by the function, but negatively impacts performance by sharing those resources.

Template functions can also be specified for ALLOCATION by specifying the function pointer instead of the function name, as shown in the examples below.

The operations in the C code, such as additions, multiplications, array reads, and writes, can also be limited by the ALLOCATION pragma.

Syntax

Place the pragma inside the body of the function, loop, or region where it will apply.



IMPORTANT! The order of the arguments below is important. The `<type>` as operation or function must follow the `allocation` keyword.

```
#pragma HLS allocation <type> instances=<list>
limit=<value>
```

Where:

- **<type>:** The type is specified as one of the following:
 - **function:** Specifies that the allocation applies to the functions listed in the `instances=` list. The function can be any function in the original C or C++ code that has not been:
 - Inlined by the `pragma HLS inline`, or the `set_directive_inline` command, or
 - Inlined automatically by the Vitis HLS tool.
 - **operation:** Specifies that the allocation applies to the operations listed in the `instances = list`.
- **instances=<list>:** Specifies the names of functions from the C code, or operators.

For a complete list of operations that can be limited using the ALLOCATION pragma, refer to the [config_op](#) command.

- **limit=<value>:** Optionally specifies the limit of instances to be used in the kernel.

Example 1

Given a design with multiple instances of function `foo`, this example limits the number of instances of `foo` in the RTL for the hardware kernel to two.

```
#pragma HLS allocation function instances=foo limit=2
```

Example 2

Limits the number of multiplier operations used in the implementation of the function `my_func` to one. This limit does not apply to any multipliers outside of `my_func`, or multipliers that might reside in sub-functions of `my_func`.



TIP: To limit the multipliers used in the implementation of any sub-functions, specify an allocation directive on the sub-functions or inline the sub-function into function `my_func`.

```
void my_func(data_t angle) {
    #pragma HLS allocation operation instances=mul limit=1
    ...
}
```

Example 3

The ALLOCATION pragma can also be used on template functions as shown below. The identification is generally based on the function name, but in the case of template functions it is based on the function pointer:

```
template <typename DT>
void foo(DT a, DT b){
}
// The following is valid
#pragma HLS ALLOCATION function instances=foo<DT>
...
// The following is not valid
#pragma HLS ALLOCATION function instances=foo
```

See Also

- [set_directive_allocation](#)
- [pragma HLS inline](#)

pragma HLS array_partition

Description



IMPORTANT! *Array_Partition* and *Array_Reshape* pragmas and directives are not supported for *M_AXI* Interfaces on the top-level function. Instead you can use the *hls::vector* data types as described in [Vector Data Types](#).

Partitions an array into smaller arrays or individual elements and provides the following:

- Results in RTL with multiple small memories or multiple registers instead of one large memory.
- Effectively increases the amount of read and write ports for the storage.
- Potentially improves the throughput of the design.
- Requires more memory instances or registers.

Syntax

Place the pragma in the C source within the boundaries of the function where the array variable is defined.

```
#pragma HLS array_partition variable=<name> \
type=<type>  factor=<int>  dim=<int>
```

Where:

- **variable=<name>**: A required argument that specifies the array variable to be partitioned.

- **type=<type>**: Optionally specifies the partition type. The default type is `complete`. The following types are supported:
 - **cyclic**: Cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. For example, if `factor=3` is used:
 - Element 0 is assigned to the first new array
 - Element 1 is assigned to the second new array.
 - Element 2 is assigned to the third new array.
 - Element 3 is assigned to the first new array again.
 - **block**: Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks, where N is the integer defined by the `factor=` argument.
 - **complete**: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. This is the default <type>.
- **factor=<int>**: Specifies the number of smaller arrays that are to be created.



IMPORTANT! For complete type partitioning, the factor is not specified. For block and cyclic partitioning, the `factor=` is required.

- **dim=<int>**: Specifies which dimension of a multi-dimensional array to partition. Specified as an integer from 0 to <N>, for an array with <N> dimensions:
 - If a value of 0 is used, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
 - Any non-zero value partitions only the specified dimension. For example, if a value 1 is used, only the first dimension is partitioned.

Example 1

This example partitions the 13 element array, AB[13], into four arrays using block partitioning:

```
#pragma HLS array_partition variable=AB type=block factor=4
```



TIP: Because four is not an integer factor of 13:

- Three of the new arrays have three elements each
- One array has four elements (AB[9:12])

Example 2

This example partitions dimension two of the two-dimensional array, AB[6][4] into two new arrays of dimension [6][2]:

```
#pragma HLS array_partition variable=AB type=block factor=2 dim=2
```

Example 3

This example partitions the second dimension of the two-dimensional `in_local` array into individual elements.

```
int in_local[MAX_SIZE][MAX_DIM];
#pragma HLS ARRAY_PARTITION variable=in_local type=complete dim=2
```

Example 4

Partitioned arrays can be addressed in your code by the new structure of the array, as shown in the following code example;

```
struct SS
{
    int x[N];
    int y[N];
};

int top(SS *a, int b[4][6], SS &c) {
#pragma HLS array_partition type=complete dim=1 variable=b
#pragma HLS interface mode=ap_memory port = b[0]
#pragma HLS interface mode=ap_memory port = b[1]
#pragma HLS interface mode=ap_memory port = b[2]
#pragma HLS interface mode=ap_memory port = b[3]
```

See Also

- [set_directive_array_partition](#)
- [pragma HLS array_reshape](#)

pragma HLS array_reshape

Description



IMPORTANT! *Array_Partition and Array_Reshape pragmas and directives are not supported for M_AXI Interfaces on the top-level function. Instead you can use the `hls::vector` data types as described in [Vector Data Types](#).*

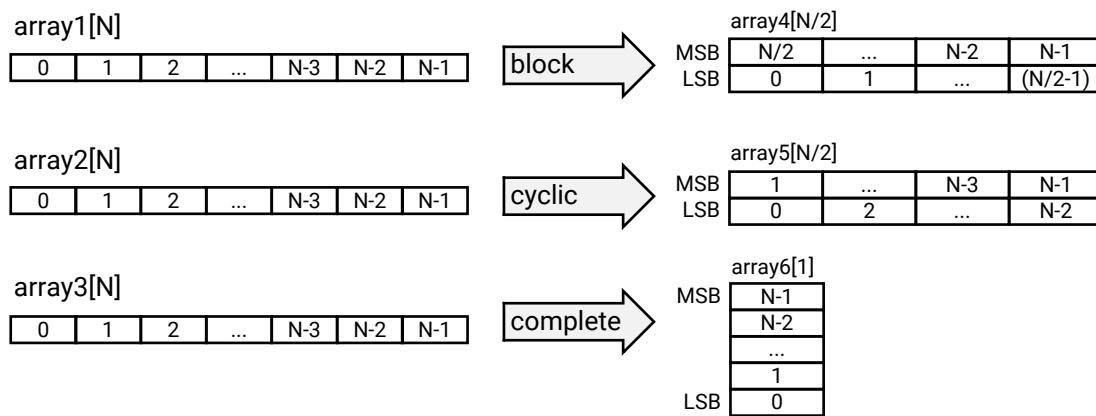
The `ARRAY_RESHAPE` pragma reforms the array with vertical remapping and concatenating elements of arrays by increasing bit-widths. This reduces the number of block RAM consumed while providing parallel access to the data. This pragma creates a new array with fewer elements but with greater bit-width, allowing more data to be accessed in a single clock cycle.

Given the following code:

```
void foo ( . . . ) {
    int array1[N];
    int array2[N];
    int array3[N];
    #pragma HLS ARRAY_RESHAPE variable=array1 type=block factor=2 dim=1
    #pragma HLS ARRAY_RESHAPE variable=array2 type=cycle factor=2 dim=1
    #pragma HLS ARRAY_RESHAPE variable=array3 type=complete dim=1
    .
}
```

The `ARRAY_RESHAPE` pragma transforms the arrays into the form shown in the following figure.

Figure 128: ARRAY_RESHAPE Pragma



X14307-110217

Syntax

Place the pragma in the C source within the region of a function where the array variable is defined.

```
#pragma HLS array_reshape variable=<name> \type=<type> factor=<int>
dim=<int>
```

Where:

- `variable=<name>`: Required argument that specifies the array variable to be reshaped.
- `type=<type>`: Optionally specifies the partition type. The default type is `complete`. The following types are supported:

- **cyclic**: Cyclic reshaping creates smaller arrays by interleaving elements from the original array. For example, if `factor=3` is used, element 0 is assigned to the first new array, element 1 to the second new array, element 2 is assigned to the third new array, and then element 3 is assigned to the first new array again. The final array is a vertical concatenation (word concatenation, to create longer words) of the new arrays into a single array.
- **block**: Block reshaping creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into $<N>$ equal blocks where $<N>$ is the integer defined by `factor=`, and then combines the $<N>$ blocks into a single array with `word-width*N`.
- **complete**: Complete reshaping decomposes the array into temporary individual elements and then recombines them into an array with a wider word. For a one-dimension array this is equivalent to creating a very-wide register (if the original array was N elements of M bits, the result is a register with $N*M$ bits). This is the default type of array reshaping.
- **factor=<int>**: Specifies the amount to divide the current array by (or the number of temporary arrays to create). A factor of 2 splits the array in half, while doubling the bit-width. A factor of 3 divides the array into three, with triple the bit-width.



IMPORTANT! For complete type partitioning, the `factor` is not specified. For block and cyclic reshaping, the `factor=` is required.

- **dim=<int>**: Specifies which dimension of a multi-dimensional array to partition. Specified as an integer from 0 to $<N>$, for an array with $<N>$ dimensions:
 - If a value of 0 is used, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
 - Any non-zero value partitions only the specified dimension. For example, if a value 1 is used, only the first dimension is partitioned.
- **object**: A keyword relevant for container arrays only. When the keyword is specified the `ARRAY_reshape` pragma applies to the objects in the container, reshaping all dimensions of the objects within the container, but all dimensions of the container itself are preserved. When the keyword is not specified the pragma applies to the container array and not the objects.

Example 1

Reshapes an 8-bit array with 17 elements, `AB[17]`, into a new 32-bit array with five elements using block mapping.

```
#pragma HLS array_reshape variable=AB type=block factor=4
```



TIP: `factor=4` indicates that the array should be divided into four; this means that 17 elements are reshaped into an array of five elements, with four times the bit-width. In this case, the last element, `AB[17]`, is mapped to the lower eight bits of the fifth element, and the rest of the fifth element is empty.

Example 2

Reshapes the two-dimensional array `AB [6] [4]` into a new array of dimension `[6][2]`, in which dimension 2 has twice the bit-width.

```
#pragma HLS array_reshape variable=AB type=block factor=2 dim=2
```

Example 3

Reshapes the three-dimensional 8-bit array, `AB [4] [2] [2]` in function `func`, into a new single element array (a register), 128-bits wide ($4 \times 2 \times 2 \times 8$).

```
#pragma HLS array_reshape variable=AB type=complete dim=0
```



TIP: `dim=0` means to reshape all dimensions of the array.

Example 4

Partitioned arrays can be addressed in your code by the new structure of the array, as shown in the following code example;

```
struct SS
{
    int x[N];
    int y[N];
};

int top(SS *a, int b[4][6], SS &c) {
#pragma HLS array_reshape type=complete dim=0 variable=b
#pragma HLS interface mode=ap_memory port=b[0]
```

See Also

- [pragma HLS array_reshape](#)
- [pragma HLS array_partition](#)

pragma HLS bind_op

Description

Vitis HLS implements the operations in the code using specific implementations. The `BIND_OP` pragma specifies that for a specific variable, an operation (`mul`, `add`, `div`) should be mapped to a specific device resource for implementation (`impl`) in the RTL. If the `BIND_OP` pragma is not specified, Vitis HLS automatically determines the resources to use for operations.

For example, to indicate that a specific multiplier operation (`mul`) is implemented in the device fabric rather than a DSP, you can use the `BIND_OP` pragma.

You can also specify the latency of the operation using the `latency` option.



IMPORTANT! To use the `latency` option, the operation must have an available multi-stage implementation. The HLS tool provides a multi-stage implementation for all basic arithmetic operations (add, subtract, multiply, and divide), and all floating-point operations.

Syntax

Place the pragma in the C source within the body of the function where the variable is defined.

```
#pragma HLS bind_op variable=<variable> op=<type>\  
impl=<value> latency=<int>
```

Where:

- `variable=<variable>`: Defines the variable to assign the `BIND_OP` pragma to. The variable in this case is one that is assigned the result of the operation that is the target of this pragma.
- `op=<type>`: Defines the operation to bind to a specific implementation resource. Supported functional operations include: `mul`, `add`, and `sub`. Supported floating point operations include: `fadd`, `fsub`, `fdiv`, `fexp`, `flog`, `fmul`, `frsqrt`, `frexp`, `fsqrt`, `dadd`, `dsub`, `ddiv`, `dexp`, `dlog`, `dmul`, `drsqrt`, `drrecip`, `dsqrt`, `hadd`, `hsub`, `hdiv`, `hmul`, and `hsqrt`



TIP: Floating point operations include single precision (`f`), double-precision (`d`), and half-precision (`h`).

- `impl=<value>`: Defines the implementation to use for the specified operation. Supported implementations for functional operations include `fabric`, and `dsp`. Supported implementations for floating point operations include: `fabric`, `meddsp`, `fulldsp`, `maxdsp`, and `primitivedsp`.

Note: `primitivedsp` is only available on Versal® devices.

- `latency=<int>`: Defines the default latency for the implementation of the operation. The valid latency varies according to the specified `op` and `impl`. The default is -1, which lets Vitis HLS choose the latency. The tables below reflect the supported combinations of operation, implementation, and latency.

Table 34: Supported Combinations of Functional Operations, Implementation, and Latency

Operation	Implementation	Min Latency	Max Latency
add	fabric	0	4
add	dsp	0	4
mul	fabric	0	4

Table 34: Supported Combinations of Functional Operations, Implementation, and Latency (cont'd)

Operation	Implementation	Min Latency	Max Latency
mul	dsp	0	4
sub	fabric	0	4
sub	dsp	0	0



TIP: Comparison operators, such as `dcmp`, are implemented in LUTs and cannot be implemented outside of the fabric, or mapped to DSPs, and so are not configurable with the `config_op` or `bind_op` commands.

Table 35: Supported Combinations of Floating Point Operations, Implementation, and Latency

Operation	Implementation	Min Latency	Max Latency
fadd	fabric	0	13
fadd	fulldsp	0	12
fadd	primitivedsp	0	3
fsub	fabric	0	13
fsub	fulldsp	0	12
fsub	primitivedsp	0	3
fdiv	fabric	0	29
fexp	fabric	0	24
fexp	meddsp	0	21
fexp	fulldsp	0	30
flog	fabric	0	24
flog	meddsp	0	23
flog	fulldsp	0	29
fmul	fabric	0	9
fmul	meddsp	0	9
fmul	fulldsp	0	9
fmul	maxdsp	0	7
fmul	primitivedsp	0	4
fsqrt	fabric	0	29
frsqrt	fabric	0	38
frsqrt	fulldsp	0	33
frecip	fabric	0	37
frecip	fulldsp	0	30
dadd	fabric	0	13
dadd	fulldsp	0	15
dsub	fabric	0	13
dsub	fulldsp	0	15

Table 35: Supported Combinations of Floating Point Operations, Implementation, and Latency (cont'd)

Operation	Implementation	Min Latency	Max Latency
ddiv	fabric	0	58
dexp	fabric	0	40
dexp	meddsp	0	45
dexp	fulldsp	0	57
dlog	fabric	0	38
dlog	meddsp	0	49
dlog	fulldsp	0	65
dmul	fabric	0	10
dmul	meddsp	0	13
dmul	fulldsp	0	13
dmul	maxdsp	0	14
dsqrt	fabric	0	58
drsqrt	fulldsp	0	111
drecip	fulldsp	0	36
hadd	fabric	0	9
hadd	meddsp	0	12
hadd	fulldsp	0	12
hsub	fabric	0	9
hsub	meddsp	0	12
hsub	fulldsp	0	12
hdiv	fabric	0	16
hmul	fabric	0	7
hmul	fulldsp	0	7
hmul	maxdsp	0	9
hsqrt	fabric	0	16

Example

In the following example, a two-stage pipelined multiplier using fabric logic is specified to implement the multiplication for variable `c` of the function `foo`.

```
int foo (int a, int b) {
int c, d;
#pragma HLS BIND_OP variable=c op=mul impl=fabric latency=2
c = a*b;
d = a*c;
return d;
}
```



TIP: The HLS tool selects the implementation to use for variable `d`.

See Also

- [set_directive_bind_op](#)
 - [pragma HLS bind_storage](#)
-

pragma HLS bind_storage

Description

The BIND_STORAGE pragma assigns a variable (array, or function argument) in the code to a specific memory type (`type`) in the RTL. If the pragma is not specified, the Vitis HLS tool determines the memory type to assign. The HLS tool implements the memory using specified implementations (`impl`) in the hardware.

For example, you can use the pragma to specify which memory type to use to implement an array. This lets you control whether the array is implemented as a single or a dual-port RAM for example. Also, this allows you to control whether the array is implemented as a single or a dual-port RAM.



IMPORTANT! This feature is important for arrays on the top-level function interface, because the memory type associated with the array determines the number and type of ports needed in the RTL, as discussed in [Arrays on the Interface](#). However, for variables assigned to top-level function arguments you must assign the memory type and implementation using the `-storage_type` and `-storage_impl` options of the INTERFACE pragma or directive.

You can also specify the latency of the implementation. For block RAMs on the interface, the `latency` option lets you model off-chip, non-standard SRAMs at the interface, for example supporting an SRAM with a latency of 2 or 3. For internal operations, the `latency` option allows the memory to be implemented using more pipelined stages. These additional pipeline stages can help resolve timing issues during RTL synthesis.



IMPORTANT! To use the `latency` option, the operation must have an available multi-stage implementation. The HLS tool provides a multi-stage implementation for all block RAMs.

Syntax

Place the pragma in the C/C++ source within the body of the function where the variable is defined.

```
#pragma HLS bind_storage variable=<variable> type=<type>\  
[ impl=<value> latency=<int> ]
```

Where:

- `variable=<variable>`: Defines the variable to assign the BIND_STORAGE pragma to. This is required when specifying the pragma.



TIP: If the variable is an argument of a top-level function, then use the `-storage_type` and `-storage_impl` options of the INTERFACE pragma or directive.

- `type=<type>`: Defines the type of memory to bind to the specified variable. Supported types include: `fifo`, `ram_1p`, `ram_1wnr`, `ram_2p`, `ram_s2p`, `ram_t2p`, `rom_1p`, `rom_2p`, `rom_np`.

Table 36: Storage Types

Type	Description
FIFO	A FIFO. Vitis HLS determines how to implement this in the RTL, unless the <code>-impl</code> option is specified.
RAM_1P	A single-port RAM. Vitis HLS determines how to implement this in the RTL, unless the <code>-impl</code> option is specified.
RAM_1WNR	A RAM with 1 write port and N read ports, using N banks internally.
RAM_2P	A dual-port RAM that allows read operations on one port and both read and write operations on the other port.
RAM_S2P	A dual-port RAM that allows read operations on one port and write operations on the other port.
RAM_T2P	A true dual-port RAM with support for both read and write on both ports.
ROM_1P	A single-port ROM. Vitis HLS determines how to implement this in the RTL, unless the <code>-impl</code> option is specified.
ROM_2P	A dual-port ROM.
ROM_NP	A multi-port ROM.

- `impl=<value>`: Defines the implementation for the specified memory type. Supported implementations include: `bram`, `bram_ecc`, `lutram`, `uram`, `uram_ecc`, `srl`, `memory`, and `auto` as described below.

Table 37: Supported Implementation

Name	Description
MEMORY	Generic memory lets the Vivado tool choose the implementation.
URAM	UltraRAM resource
URAM_ECC	UltraRAM with ECC
SRL	Shift Register Logic resource
LUTRAM	Distributed RAM resource
BRAM	Block RAM resource
BRAM_ECC	Block RAM with ECC
AUTO	Vitis HLS automatically determine the implementation of the variable.

Table 38: Supported Implementations by FIFO/RAM/ROM

Type	Command/Pragma	Scope	Supported Implementations
FIFO	bind_storage ¹	local	AUTO , BRAM, LUTRAM, URAM, MEMORY, SRL
FIFO	config_storage	global	AUTO , BRAM, LUTRAM, URAM, MEMORY, SRL
RAM* ROM*	bind_storage	local	AUTO BRAM, BRAM_ECC, LUTRAM, URAM, URAM_ECC
RAM* ROM*	config_storage ²	global	N/A
RAM_1P	set_directive_interface s_axilite - storage_impl	local	AUTO , BRAM, URAM
	config_interface - m_axi_buffer_impl	global	AUTO , BRAM , LUTRAM, URAM

Notes:

- When no implementation is specified the directive uses AUTOSRL behavior as a default. However, this value cannot be specified.
 - `config_storage` only supports FIFO types.
- `latency=<int>`: Defines the default latency for the binding of the type. As shown in the following table, the valid latency varies according to the specified `type` and `impl`. The default is -1, that lets Vitis HLS choose the latency.

Table 39: Supported Combinations of Memory Type, Implementation, and Latency

Type	Implementation	Min Latency	Max Latency
FIFO	BRAM	0	0
FIFO	LUTRAM	0	0
FIFO	MEMORY	0	0
FIFO	SRL	0	0
FIFO	URAM	0	0
RAM_1P	AUTO	1	3
RAM_1P	BRAM	1	3
RAM_1P	LUTRAM	1	3
RAM_1P	URAM	1	3
RAM_1WNR	AUTO	1	3
RAM_1WNR	BRAM	1	3
RAM_1WNR	LUTRAM	1	3
RAM_1WNR	URAM	1	3
RAM_2P	AUTO	1	3
RAM_2P	BRAM	1	3
RAM_2P	LUTRAM	1	3
RAM_2P	URAM	1	3
RAM_S2P	BRAM	1	3

Table 39: Supported Combinations of Memory Type, Implementation, and Latency (cont'd)

Type	Implementation	Min Latency	Max Latency
RAM_S2P	BRAM_ECC	1	3
RAM_S2P	LUTRAM	1	3
RAM_S2P	URAM	1	3
RAM_S2P	URAM_ECC	1	3
RAM_T2P	BRAM	1	3
RAM_T2P	URAM	1	3
ROM_1P	AUTO	1	3
ROM_1P	BRAM	1	3
ROM_1P	LUTRAM	1	3
ROM_2P	AUTO	1	3
ROM_2P	BRAM	1	3
ROM_2P	LUTRAM	1	3
ROM_NP	BRAM	1	3
ROM_NP	LUTRAM	1	3



IMPORTANT! Any combinations of memory type and implementation that are not listed in the prior table are not supported by `set_directive_bind_storage`.

Example

The pragma specifies that the variable `coeffs` uses a single port RAM implemented on a BRAM core from the library.

```
#pragma HLS bind_storage variable=coeffs type=RAM_1P impl=bram
```



TIP: The ports created in the RTL to access the values of `coeffs` are defined in the `RAM_1P`.

See Also

- [set_directive_bind_storage](#)
- [pragma HLS bind_op](#)

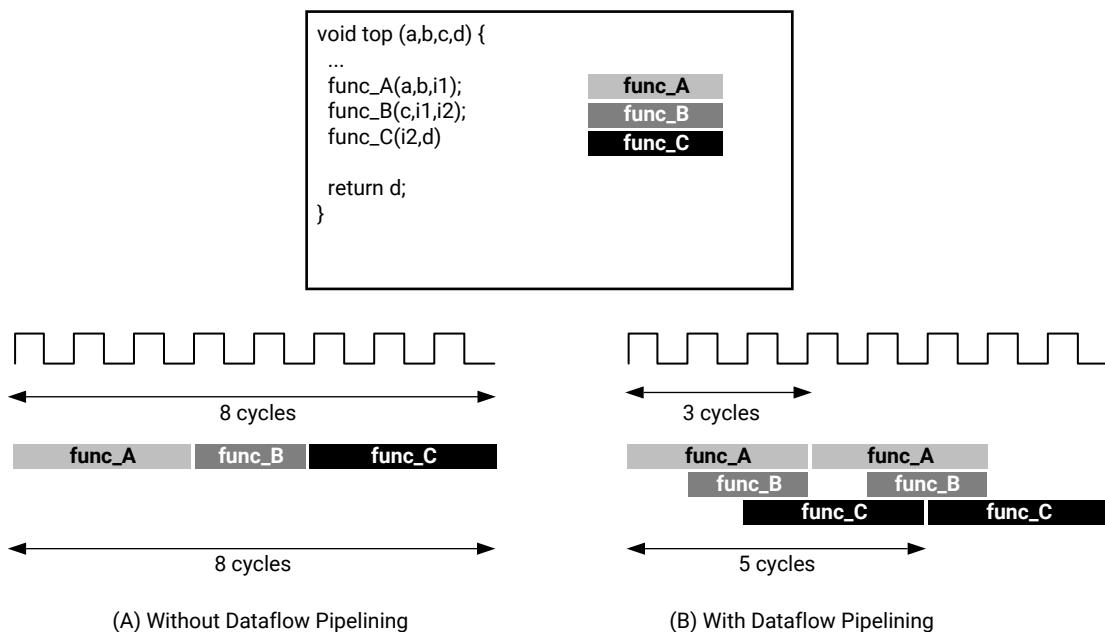
pragma HLS dataflow

Description

The DATAFLOW pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation and increasing the overall throughput of the design.

All operations are performed sequentially in a C description. In the absence of any directives that limit resources (such as `pragma HLS allocation`), the Vitis HLS tool seeks to minimize latency and improve concurrency. However, data dependencies can limit this. For example, functions or loops that access arrays must finish all read/write accesses to the arrays before they complete. This prevents the next function or loop that consumes the data from starting operation. The DATAFLOW optimization enables the operations in a function or loop to start operation before the previous function or loop completes all its operations.

Figure 129: DATAFLOW Pragma



X14266-110217

When the DATAFLOW pragma is specified, the HLS tool analyzes the dataflow between sequential functions or loops and creates channels (based on ping pong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL.

If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, the HLS tool attempts to minimize the initiation interval and start operation as soon as data is available.



TIP: The `config_dataflow` command specifies the default memory channel and FIFO depth used in dataflow optimization.

For the DATAFLOW optimization to work, the data must flow through the design from one task to the next. The following coding styles prevent the HLS tool from performing the DATAFLOW optimization. Refer to [Limitations of Control-Driven Task-Level Parallelism](#) for additional details.

- Single-producer-consumer violations
- Feedback between tasks
- Conditional execution of tasks
- Loops with multiple exit conditions



IMPORTANT! If any of these coding styles are present, the HLS tool issues a message and does not perform DATAFLOW optimization.

Finally, the DATAFLOW optimization has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from the DATAFLOW optimization, you must apply the optimization to the loop, the sub-function, or inline the sub-function.

Syntax

Place the pragma in the C source within the boundaries of the region, function, or loop.

```
#pragma HLS dataflow [disable_start_propagation]
```

- `disable_start_propagation`: Optionally disables the creation of a start FIFO used to propagate a start token to an internal process. Such FIFOs can sometimes be a bottleneck for performance.

Example

Specifies DATAFLOW optimization within the loop `wr_loop_j`.

```
        wr_loop_j: for (int j = 0; j < TILE_PER_ROW; ++j) {  
#pragma HLS DATAFLOW  
            wr_buf_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {  
                wr_buf_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {  
#pragma HLS PIPELINE  
                    // should burst TILE_WIDTH in WORD beat  
                    outFifo >> tile[m][n];  
                }  
            }  
            wr_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {  
                wr_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {
```

```
#pragma HLS PIPELINE
    outx[TILE_HEIGHT*TILE_PER_ROW*TILE_WIDTH*i
+TILE_PER_ROW*TILE_WIDTH*m+TILE_WIDTH*j+n] = tile[m][n];
}
}
```

See Also

- [set_directive_dataflow](#)
- [config_dataflow](#)
- [pragma HLS allocation](#)
- [pragma HLS pipeline](#)

pragma HLS dependence

Description

Vitis HLS detects dependencies within loops: dependencies within the same iteration of a loop are loop-independent dependencies, and dependencies between different iterations of a loop are loop-carried dependencies. The DEPENDENCE pragma allows you to provide additional information to define, negate loop dependencies, and allow loops to be pipelined with lower intervals.

- **Loop-independent dependence:** The same element is accessed in a single loop iteration.

```
for (i=0;i<N;i++) {
    A[i]=x;
    y=A[i];
}
```

- **Loop-carried dependence:** The same element is accessed from a different loop iteration.

```
for (i=0;i<N;i++) {
    A[i]=A[i-1]*2;
}
```

These dependencies impact when operations can be scheduled, especially during function and loop pipelining.

Under some circumstances, such as variable dependent array indexing or when an external requirement needs to be enforced (for example, two inputs are never the same index), the dependence analysis might be too conservative and fail to filter out false dependencies. The DEPENDENCE pragma allows you to explicitly define the dependencies and eliminate a false dependence.



IMPORTANT! Specifying a false dependency, when in fact the dependency is not false, can result in incorrect hardware. Ensure dependencies are correct (true or false) before specifying them.

Syntax

Place the pragma within the boundaries of the function where the dependence is defined.

```
#pragma HLS dependence variable=<variable> <class> \
<type> <direction> distance=<int> <dependent>
```

Where:

- **variable=<variable>**: Optionally specifies the variable to consider for the dependence.



IMPORTANT! You cannot specify a dependence for function arguments that are bundled with other arguments in an `m_axi` interface. This is the default configuration for `m_axi` interfaces on the function. You also cannot specify a dependence for an element of a struct, unless the struct has been disaggregated.

- **class=[array | pointer]**: Optionally specifies a class of variables in which the dependence needs clarification. Valid values include `array` or `pointer`.



TIP: `<class>` and `variable=` should not be specified together as you can specify dependence for a variable, or a class of variables within a function.

- **type=[inter | intra]**: Valid values include `intra` or `inter`. Specifies whether the dependence is:

- **intra**: Dependence within the same loop iteration. When dependence `<type>` is specified as `intra`, and `<dependent>` is false, the HLS tool might move operations freely within a loop, increasing their mobility and potentially improving performance or area. When `<dependent>` is specified as true, the operations must be performed in the order specified.

- **inter**: dependence between different loop iterations. This is the default `<type>`. If dependence `<type>` is specified as `inter`, and `<dependent>` is false, it allows the HLS tool to perform operations in parallel if the function or loop is pipelined, or the loop is unrolled, or partially unrolled, and prevents such concurrent operation when `<dependent>` is specified as true.

- **direction=[RAW | WAR | WAW]**: This is relevant for loop-carry dependencies only, and specifies the direction for a dependence:

- **RAW (Read-After-Write - true dependence)**: The write instruction uses a value used by the read instruction.
- **WAR (Write-After-Read - anti dependence)**: The read instruction gets a value that is overwritten by the write instruction.

- **WAW (Write-After-Write - output dependence):** Two write instructions write to the same location, in a certain order.
- **distance=<int>:** Specifies the inter-iteration distance for array access. Relevant only for loop-carry dependencies where dependence is set to `true`.
- **<dependent>:** The `<dependent>` argument should be specified to indicate whether a dependence is `true` and needs to be enforced, or is `false` and should be removed. However, when not specified, the tool will return a warning that the value was not specified and will assume a value of `false`. The accepted values are `true` or `false`.

Example 1

In the following example, the HLS tool does not have any knowledge about the value of `cols` and conservatively assumes that there is always a dependence between the write to `buff_A[1][col]` and the read from `buff_A[1][col]`. In an algorithm such as this, it is unlikely `cols` will ever be zero, but the HLS tool cannot make assumptions about data dependencies. To overcome this deficiency, you can use the `DEPENDENCE` pragma to state that there is no dependence between loop iterations (in this case, for both `buff_A` and `buff_B`).

```
void foo(int rows, int cols, ...){  
    for (row = 0; row < rows + 1; row++) {  
        for (col = 0; col < cols + 1; col++) {  
            #pragma HLS PIPELINE II=1  
            #pragma HLS dependence variable=buf_A type=inter false  
            #pragma HLS dependence variable=buf_B type=inter false  
            if (col < cols) {  
                buf_A[2][col] = buf_A[1][col]; // read from buf_A[1][col]  
                buf_A[1][col] = buf_A[0][col]; // write to buf_A[1][col]  
                buf_B[1][col] = buf_B[0][col];  
                temp = buf_A[0][col];  
            }  
        }  
    }  
}
```

Example 2

Removes the dependence between `Var1` in the same iterations of `loop_1` in function `func`.

```
#pragma HLS dependence variable=Var1 type=intra false
```

Example 3

Defines the dependence on all arrays in `loop_2` of function `func` to inform the HLS tool that all reads must happen after writes (RAW) in the same loop iteration.

```
#pragma HLS dependence class=array type=intra direction=RAW true
```

See Also

- [set_directive_dependence](#)
- [pragma HLS disaggregate](#)

- [pragma HLS pipeline](#)
-

pragma HLS disaggregate

Description

The DISAGGREGATE pragma lets you deconstruct a `struct` variable into its individual elements. The number and type of elements created are determined by the contents of the struct itself.



IMPORTANT! Structs used as arguments to the top-level function are aggregated by default, but can be disaggregated with this directive or pragma. Refer to [AXI4-Stream Interfaces](#) for important information about disaggregating structs associated with streams.

Syntax

Place the pragma in the C source within the boundaries of the region, function, or loop.

```
#pragma HLS disaggregate variable=<variable>
```

Options

Where:

- `variable=<variable>`: Specifies the struct variable to disaggregate.

Example 1

The following example shows the struct variable `a` in function `top` will be disaggregated:

```
#pragma HLS disaggregate variable=a
```

Example 2

Disaggregated structs can be addressed in your code by the using standard C/C++ coding style as shown below. Notice the different methods for accessing the pointer element (`a`) versus the reference element (`c`);

```
struct SS
{
    int x[N];
    int y[N];
};

int top(SS *a, int b[4][6], SS &c) {
#pragma HLS disaggregate variable = a
#pragma HLS interface s_axilite port = a->x
#pragma HLS interface s_axilite port = a->y
```

```
// Following is now supported
#pragma HLS disaggregate variable = c
#pragma HLS interface ap_memory port = c.x
#pragma HLS interface ap_memory port = c.y
```

Example 3

The following example shows the `Dot` struct containing the `RGB` struct as an element. If you apply the `DISAGGREGATE` pragma to variable `Arr`, then only the top-level `Dot` struct is disaggregated.

```
struct Pixel {
char R;
char G;
char B;
};

struct Dot {
Pixel RGB;
unsigned Size;
};

#define N 1086
void DUT(Dot Arr[N]) {
#pragma HLS disaggregate variable=Arr
...
}
```

If you want to disaggregate the whole struct, `Dot` and `RGB`, then you can assign the `DISAGGREGATE` pragma as shown below.

```
void DUT(Dot Arr[N]) {
#pragma HLS disaggregate variable=Arr->RGB
...
}
```

The results in this case will be:

```
void DUT(char Arr_RGB_R[N], char Arr_RGB_G[N], char Arr_RGB_B[N], unsigned
Arr_Size[N]) {
#pragma HLS disaggregate variable=Arr->RGB
...
}
```

See Also

- [set_directive_disaggregate](#)
- [pragma HLS aggregate](#)

pragma HLS expression_balance

Description

Sometimes C/C++ code is written with a sequence of operations, resulting in a long chain of operations in RTL. With a small clock period, this can increase the latency in the design. By default, the Vitis HLS tool rearranges the operations using associative and commutative properties. This rearrangement creates a balanced tree that can shorten the chain, potentially reducing latency in the design at the cost of extra hardware.

Expression balancing rearranges operators to construct a balanced tree and reduce latency.

- For integer operations expression balancing is on by default but may be disabled.
- For floating-point operations, expression balancing is off by default but may be enabled.

The EXPRESSION_BALANCE pragma allows this expression balancing to be disabled, or to be expressly enabled, within a specified scope.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS expression_balance off
```

Where:

- **off**: Turns off expression balancing at this location. Specifying `#pragma HLS expression_balance` enables expression balancing in the specified scope. Adding `off` disables it.

Example 1

Disables expression balancing within function `my_Func`.

```
void my_func(char inval, char incr) {  
    #pragma HLS expression_balance off
```

Example 2

This example explicitly enables expression balancing in function `my_Func`.

```
void my_func(char inval, char incr) {  
    #pragma HLS expression_balance
```

See Also

- [set_directive_expression_balance](#)

pragma HLS function_instantiate

Description

The FUNCTION_INSTANTIATE pragma is an optimization technique that has the area benefits of maintaining the function hierarchy but provides an additional powerful option: performing targeted local optimizations on specific instances of a function. This can simplify the control logic around the function call and potentially improve latency and throughput.

By default:

- Functions remain as separate hierarchy blocks in the RTL, or is decomposed (or inlined) into a higher level function.
- All instances of a function, at the same level of hierarchy, make use of a single RTL implementation (block).

The FUNCTION_INSTANTIATE pragma is used to create a unique RTL implementation for each instance of a function, allowing each instance to be locally optimized according to the function call. This pragma exploits the fact that some inputs to a function can be a constant value when the function is called, and uses this to both simplify the surrounding control structures and produce smaller more optimized function blocks.

Without the FUNCTION_INSTANTIATE pragma, the following code results in a single RTL implementation of function `func_sub` for all three instances of the function in `func`. Each instance of function `func_sub` is implemented in an identical manner. This is fine for function reuse and reducing the area required for each instance call of a function, but means that the control logic inside the function must be more complex to account for the variation in each call of `func_sub`.

```
char func_sub(char inval, char incr) {
#pragma HLS INLINE OFF
#pragma HLS FUNCTION_INSTANTIATE variable=incr
    return inval + incr;
}
void func(char inval1, char inval2, char inval3,
          char *outval1, char *outval2, char * outval3)
{
    *outval1 = func_sub(inval1, 1);
    *outval2 = func_sub(inval2, 2);
    *outval3 = func_sub(inval3, 3);
}
```



TIP: The Vitis HLS tool automatically decomposes (or inlines) small functions into higher-level calling functions. This is true even for function instantiations. Using the `INLINE` pragma with the `OFF` option can be used to prevent this automatic inlining.

In the code sample above, the FUNCTION_INSTANTIATE pragma results in three different implementations of function `func_sub`, each optimized for the specified values of `incr`, reducing the area and improving the performance of the function implementation.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS FUNCTION_INSTANTIATE variable=<variable>
```

Where:

- `variable=<variable>`: A required argument that defines the function argument to use as a constant.

Examples

In the following example, the FUNCTION_INSTANTIATE pragma, placed in function `swInt`, allows each instance of function `swInt` to be independently optimized with respect to the `maxv` function argument.

```
void swInt(unsigned int *readRefPacked, short *maxr, short *maxc, short
*maxv){
    #pragma HLS FUNCTION_INSTANTIATE variable=maxv
        uint2_t d2bit[MAXCOL];
        uint2_t q2bit[MAXROW];
    #pragma HLS array partition variable=d2bit,q2bit cyclic factor=FACTOR
        intTo2bit<MAXCOL/16>((readRefPacked + MAXROW/16), d2bit);
        intTo2bit<MAXROW/16>(readRefPacked, q2bit);
        sw(d2bit, q2bit, maxr, maxc, maxv);
}
```

See Also

- [set_directive_function_instantiate](#)
- [pragma HLS allocation](#)
- [pragma HLS inline](#)

pragma HLS inline

Description

Removes a function as a separate entity in the hierarchy. After inlining, the function is dissolved into the calling function and no longer appears as a separate level of hierarchy in the RTL.

 **IMPORTANT!** Inlining a child function also dissolves any pragmas or directives applied to that function. In Vitis HLS, any pragmas applied in the child context are ignored.

In some cases, inlining a function allows operations within the function to be shared and optimized more effectively with the calling function. However, an inlined function cannot be shared or reused, so if the parent function calls the inlined function multiple times, this can increase the area required for implementing the RTL.

The `INLINE` pragma applies differently to the scope it is defined in depending on how it is specified:

- `INLINE`: Without arguments, the pragma means that the function it is specified in should be inlined upward into any calling functions.
- `INLINE OFF`: Specifies that the function it is specified in should *not* be inlined upward into any calling functions. This disables the inline of a specific function that can be automatically inlined or inlined as part of recursion.
- `INLINE RECURSIVE`: Applies the pragma to the body of the function it is assigned in. It applies downward, recursively inlining the contents of the function.

By default, inlining is only performed on the next level of function hierarchy, not sub-functions. However, the `recursive` option lets you specify inlining through levels of the hierarchy.

Syntax

Place the pragma in the C source within the body of the function or region of code.

```
#pragma HLS inline <recursive | off>
```

Where:

- `recursive`: By default, only one level of function inlining is performed, and functions within the specified function are not inlined. The `recursive` option inlines all functions recursively within the specified function or region.
- `off`: Disables function inlining to prevent specified functions from being inlined. For example, if `recursive` is specified in a function, this option can prevent a particular called function from being inlined when all others are.



TIP: The Vitis HLS tool automatically inlines small functions, and using the `INLINE` pragma with the `off` option can be used to prevent this automatic inlining.

Example 1

The following example inlines all functions within the body of `func_top` inlining recursively down through the function hierarchy, except function `func_sub` is not inlined. The recursive pragma is placed in function `func_top`. The pragma to disable inlining is placed in the function `func_sub`:

```
func_sub (p, q) {  
    #pragma HLS inline off  
    int q1 = q + 10;  
    func(p1,q); // foo_3  
    ...  
}  
void func_top { a, b, c, d } {  
    #pragma HLS inline recursive  
    ...  
    func(a,b); // func_1  
    func(a,c); // func_2  
    func_sub(a,d);  
    ...  
}
```



TIP: Notice in this example that `INLINE RECURSIVE` applies downward to the contents of function `func_top`, but `INLINE OFF` applies to `func_sub` directly.

Example 2

This example inlines the `copy_output` function into any functions or regions calling `copy_output`.

```
void copy_output(int *out, int out_lcl[OSize * OSize], int output) {  
    #pragma HLS INLINE  
    // Calculate each work_item's result update location  
    int stride = output * OSize * OSize;  
  
    // Work_item updates output filter/image in DDR  
    writeOut: for(int itr = 0; itr < OSize * OSize; itr++) {  
        #pragma HLS PIPELINE  
        out[stride + itr] = out_lcl[itr];  
    }  
}
```

See Also

- [set_directive_inline](#)
- [pragma HLS allocation](#)

pragma HLS interface

Description



IMPORTANT! The INTERFACE pragma or directive is only supported for use on the top-level function, and cannot be used for sub-functions of the HLS design. The Vitis HLS tool automatically determines the I/O protocol used by any sub-functions.

In C/C++ code, all input and output operations are performed, in zero time, through formal function arguments. In a RTL design, these same input and output operations must be performed through a port in the design interface and typically operate using a specific input/output (I/O) protocol. For more information, see [Defining Interfaces](#).

The INTERFACE pragma specifies how RTL ports are created from the function definition during interface synthesis. The ports in the RTL implementation are derived from the following:

- Block-level I/O protocols: Provide signals to control when the function starts operation, and indicate when function operation ends, is idle, and is ready for new inputs. The implementation of a block-level protocol is:
 - Specified by the <mode> values ap_ctrl_none, ap_ctrl_hs, or ap_ctrl_chain. The ap_ctrl_chain block protocol is the default.
 - Associated with the function name.
- Function arguments: Each function argument can be specified to have its own port-level (I/O) interface protocol, such as valid handshake (ap_vld), or acknowledge handshake (ap_ack). Port-level interface protocols are created for each argument in the top-level function and the function return, if the function returns a value. The default I/O protocol created depends on the type of C argument. After the block-level protocol has been used to start the operation of the block, the port-level I/O protocols are used to sequence data into and out of the block.



TIP: Global variables required on the interface must be explicitly defined as an argument of the top-level function as described in [Global Variables](#). If a global variable is accessed, but all read and write operations are local to the design, the resource is created in the design. There is no need for an I/O port in the RTL.

Specifying Burst Mode

When specifying burst-mode for interfaces, using the `max_read_burst_length` or `max_write_burst_length` options (as described in the [Syntax](#) section) there are limitations and related considerations that are derived from the AXI standard:

1. The burst length should be less than, or equal to 256 words per transaction, because ARLEN & AWLEN are 8 bits; the actual burst length is AxLEN+1.
2. In total, less than 4 KB is transferred per burst transaction.

3. Do not cross the 4 KB address boundary.
4. The bus width is specified as a power of 2, between 32 bits and 512 bits (that is, 32, 64, 128, 256, 512 bits) or in bytes: 4, 8, 16, 32, 64.

With the 4 KB limit, the max burst length for a bus width of:

- 4 bytes (32 bits) is 256 words transferred in a single burst transaction. In this case, the total bytes transferred per transaction would be 1024.
- 8 bytes (64 bits) is 256 words transferred in a single burst transaction. The total bytes transferred per transaction would be 2048.
- 16 bytes (128 bits) is 256 words transferred in a single burst transaction. The total bytes transferred per transaction would be 4096.
- 32 bytes (256 bits) is 128 words transferred in a single burst transaction. The total bytes transferred per transaction would be 4096.
- 64 bytes (512 bits) is 64 words transferred in a single burst transaction. The total bytes transferred per transaction would be 4096.



TIP: The IP generated by the HLS tool might not actually perform the maximum burst length as this is design dependent. Refer to [AXI Burst Transfers](#) for more information.

For example, pipelined accesses from a `for`-loop of 100 iterations will not fill the max burst length when `max_read_burst_length` or `max_write_burst_length` is set to 128.

However, if the design is doing longer accesses than the specified maximum burst length, the access will be split into multiple bursts. For example, a pipelined `for`-loop with 100 accesses, and `max_read_burst_length` or `max_write_burst_length` of 64, will be split into 2 transactions: one sized to the max burst length (64), and one with the remaining data (burst of length 36 words).

Syntax

Place the pragma within the boundaries of the function.

```
#pragma HLS interface mode=<mode> port=<name> bundle=<string> \
register register_mode=<mode> depth=<int> offset=<string> latency=<value> \
clock=<string> name=<string> storage_type=<value> \
num_read_outstanding=<int> num_write_outstanding=<int> \
max_read_burst_length=<int> max_write_burst_length=<int>
```

Where:

- `mode=<mode>`: Specifies the interface protocol mode for function arguments used by the function, or the block-level control protocols. The mode can be specified as one of the following:
 - `ap_none`: No protocol. The interface is a data port.

- **ap_stable**: No protocol. The interface is a data port. The HLS tool assumes the data port is always stable after reset, which allows internal optimizations to remove unnecessary registers.
- **ap_vld**: Implements the data port with an associated `valid` port to indicate when the data is valid for reading or writing.
- **ap_ack**: Implements the data port with an associated `acknowledge` port to acknowledge that the data was read or written.
- **ap_hs**: Implements the data port with associated `valid` and `acknowledge` ports to provide a two-way handshake to indicate when the data is valid for reading and writing and to acknowledge that the data was read or written.
- **ap_ovld**: Implements the output data port with an associated `valid` port to indicate when the data is valid for reading or writing.



IMPORTANT! The HLS tool implements the input argument or the input half of any read/write arguments with mode `ap_none`.

- **ap_fifo**: Implements the port with a standard FIFO interface using data input and output ports with associated active-Low FIFO `empty` and `full` ports.

Note: You can only use this interface on read arguments or write arguments. The `ap_fifo` mode does not support bidirectional read/write arguments.

- **ap_memory**: Implements array arguments as a standard RAM interface. If you use the RTL design in the Vivado IP integrator, the memory interface appears as discrete ports.
- **bram**: Implements array arguments as a standard RAM interface. If you use the RTL design in the IP integrator, the memory interface appears as a single port.
- **axis**: Implements all ports as an AXI4-Stream interface.
- **s_axilite**: Implements all ports as an AXI4-Lite interface. The HLS tool produces an associated set of C driver files during the Export RTL process.
- **m_axi**: Implements all ports as an AXI4 interface. You can use the `config_interface` command to specify either 32-bit (default) or 64-bit address ports and to control any address offset.
- **ap_ctrl_chain**: Implements a set of block-level control ports to start the design operation, continue operation, and indicate when the design is idle, done, and ready for new input data.

Note: The `ap_ctrl_chain` interface mode is similar to `ap_ctrl_hs` but provides an additional input signal `ap_continue` to apply back pressure. Xilinx recommends using the `ap_ctrl_chain` block-level I/O protocol when chaining the HLS tool blocks together.

Note: The `ap_ctrl_chain` is the default block-level I/O protocol.

- **ap_ctrl_hs**: Implements a set of block-level control ports to start the design operation and to indicate when the design is idle, done, and ready for new input data.
- **ap_ctrl_none**: No block-level I/O protocol.

Note: Using the `ap_ctrl_none` mode might prevent the design from being verified using the C//RTL co-simulation feature.

- **port=<name>**: Specifies the name of the function argument or function return which the INTERFACE pragma applies to.



TIP: Block-level I/O protocols (`ap_ctrl_none`, `ap_ctrl_hs`, or `ap_ctrl_chain`) can be assigned to a port for the function `return` value.

- **bundle=<string>**: By default, the HLS tool groups or bundles function arguments with compatible options into interface ports in the RTL code. All AXI4-Lite (`s_axilite`) interfaces are bundled into a single AXI4-Lite port whenever possible. Similarly, all function arguments specified as an AXI4 (`m_axi`) interface are bundled into a single AXI4 port by default. All interface ports with compatible options, such as `mode`, `offset`, and `bundle`, are grouped into a single interface port. The port name is derived automatically from a combination of the mode and bundle, or is named as specified by `-name`.



IMPORTANT! When specifying the `bundle` name you should use all lower-case characters.

- **register**: An optional keyword to register the signal and any relevant protocol signals, and causes the signals to persist until at least the last cycle of the function execution. The `-register_io` option of the `config_interface` command globally controls registering all inputs/outputs on the top function. This option applies to the following interface modes:

- `s_axilite`
- `ap_fifo`
- `ap_none`
- `ap_hs`
- `ap_ack`
- `ap_vld`
- `ap_ovld`
- `ap_stable`



TIP: The use of the `register` option on the return port of the function (`port=return`) is not supported. Instead use the LATENCY pragma or directive:

```
#pragma HLS LATENCY min=1 max=1
```

- `register_mode=<forward|reverse|both|off>`: This option applies to AXI4-Stream interfaces, and specifies if registers are placed on the forward path (`TDATA` and `TVALID`), the reverse path (`TREADY`), on both paths (`TDATA`, `TVALID`, and `TREADY`), or if none of the ports signals are to be registered (`off`). The default is `both`. AXI4-Stream side-channel signals are considered to be data signals and are registered whenever the `TDATA` is registered.
- `depth=<int>`: Specifies the maximum number of samples for the test bench to process. This setting indicates the maximum size of the FIFO needed in the verification adapter that the HLS tool creates for RTL co-simulation.



TIP: While `depth` is usually an option, it is required for `m_axi` interfaces and determines the amount of resources allocated for the adapter as explained in [AXI4 Master Interface](#).

- `offset=<string>`: Controls the address offset in AXI4-Lite (`s_axilite`) and AXI4 memory mapped (`m_axi`) interfaces for the specified port.
 - In an `s_axilite` interface, `<string>` specifies the address in the register map.
 - In an `m_axi` interface this option overrides the global option specified by the `config_interface -m_axi_offset` option, and `<string>` is specified as:
 - `off`: Do not generate an offset port.
 - `direct`: Generate a scalar input offset port.
 - `slave`: Generate an offset port and automatically map it to an AXI4-Lite slave interface. This is the default offset.
- `clock=<name>`: Optionally specified only for interface mode `s_axilite`. This defines the clock signal to use for the interface. By default, the AXI4-Lite interface clock is the same clock as the system clock. This option is used to specify a separate clock for the AXI4-Lite (`s_axilite`) interface.



TIP: If the `bundle` option is used to group multiple top-level function arguments into a single AXI4-Lite interface, the `clock` option need only be specified on one of the bundle members.

- `name=<string>`: Specifies a name for the port which will be used in the generated RTL.
- `latency=<value>`: When `mode` is `m_axi`, this specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request a number of cycles (latency) before the read or write is expected. If this figure is too low, the design will be ready too soon and might stall waiting for the bus. If this figure is too high, bus access can be granted but the bus might stall waiting on the design to start the access.
- `storage_impl=<impl>`: For use with `s_axilite` only. This option defines a storage implementation to assign to the interface. Supported implementation values include `auto`, `bram`, and `uram`. The default is `auto`.



TIP: `uram` is a synchronous memory with only a single clock for two ports. Therefore `uram` cannot be specified for an `s_axilite` adapter with a second clock.

- **storage_type=<value>**: For use with `ap_memory` and `bram` interfaces only. This option specifies a storage type (that is, `RAM_T2P`) to assign to the variable. Supported types include: `ram_1p`, `ram_1wnr`, `ram_2p`, `ram_s2p`, `ram_t2p`, `rom_1p`, `rom_2p`, and `rom_np`.



TIP: *This can also be specified using the `BIND_STORAGE` pragma or directive for an object not on the interface.*

- **num_read_outstanding=<int>**: For AXI4 (`m_axi`) interfaces, this option specifies how many read requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size:

`num_read_outstanding * max_read_burst_length * word_size.`

- **num_write_outstanding=<int>**: For AXI4 (`m_axi`) interfaces, this option specifies how many write requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size:

`num_write_outstanding * max_write_burst_length * word_size.`

- **max_read_burst_length=<int>**:

- For AXI4 (`m_axi`) interfaces, this option specifies the maximum number of data values read during a burst transfer.

- **max_write_burst_length=<int>**:

- For AXI4 (`m_axi`) interfaces, this option specifies the maximum number of data values written during a burst transfer.



TIP: *If the port is a read-only port, then set the `num_write_outstanding=1` and `max_write_burst_length=2` to conserve memory resources. For write-only ports, set the `num_read_outstanding=1` and `max_read_burst_length=2`.*

- **-max_widen_bitwidth <int>**: Specifies the maximum bit width available for the interface when automatically widening the interface. This overrides the global value specified by the `config_interface -m_axi_max_bitwidth` command.

Example 1

In this example, both function arguments are implemented using an AXI4-Stream interface:

```
void example(int A[50], int B[50]) {
    //Set the HLS native interface types
    #pragma HLS INTERFACE mode=axis port=A
    #pragma HLS INTERFACE mode=axis port=B
    int i;
    for(i = 0; i < 50; i++){
        B[i] = A[i] + 5;
    }
}
```

Example 2

The following turns off block-level I/O protocols, and is assigned to the function return value:

```
#pragma HLS interface mode=ap_ctrl_none port=return
```

The function argument `InData` is specified to use the `ap_vld` interface and also indicates the input should be registered:

```
#pragma HLS interface mode=ap_vld register port=InData
```

Example 3

This example defines the INTERFACE standards for the ports of the top-level transpose function. Notice the use of the `bundle=` option to group signals.

```
// TOP LEVEL - TRANSPOSE
void transpose(int* input, int* output) {
    #pragma HLS INTERFACE mode=m_axi port=input offset=slave bundle=gmem0
    #pragma HLS INTERFACE mode=m_axi port=output offset=slave bundle=gmem1

    #pragma HLS INTERFACE mode=s_axilite port=input bundle=control
    #pragma HLS INTERFACE mode=s_axilite port=output bundle=control
    #pragma HLS INTERFACE mode=s_axilite port=return bundle=control

    #pragma HLS dataflow
```

See Also

- [set_directive_interface](#)
- [pragma HLS bind_storage](#)

pragma HLS latency

Description

Specifies a minimum or maximum latency value, or both, for the completion of functions, loops, and regions.

- **Latency:** Number of clock cycles required to produce an output.
- **Function latency:** Number of clock cycles required for the function to compute all output values, and return.
- **Loop latency:** Number of cycles to execute all iterations of the loop.

Vitis HLS always tries to minimize latency in the design. When the LATENCY pragma is specified, the tool behavior is as follows:

- Latency is greater than the minimum, or less than the maximum: The constraint is satisfied. No further optimizations are performed.
- Latency is less than the minimum: If the HLS tool can achieve less than the minimum specified latency, it extends the latency to the specified value, potentially enabling increased sharing.
- Latency is greater than the maximum: If the HLS tool cannot schedule within the maximum limit, it increases effort to achieve the specified constraint. If it still fails to meet the maximum latency, it issues a warning, and produces a design with the smallest achievable latency in excess of the maximum.



TIP: You can also use the LATENCY pragma to limit the efforts of the tool to find an optimum solution.

Specifying latency constraints for scopes within the code: loops, functions, or regions, reduces the possible solutions within that scope and can improve tool runtime. Refer to [Improving Synthesis Runtime and Capacity](#) for more information.

If the intention is to limit the total latency of all loop iterations, the latency directive should be applied to a region that encompasses the entire loop, as in this example:

```
Region_All_Loop_A: {  
#pragma HLS latency max=10  
Loop_A: for (i=0; i<N; i++)  
{  
    ..Loop Body...  
}  
}
```

In this case, even if the loop is unrolled, the latency directive sets a maximum limit on all loop operations.

If Vitis HLS cannot meet a maximum latency constraint it relaxes the latency constraint and tries to achieve the best possible result.

If a minimum latency constraint is set and Vitis HLS can produce a design with a lower latency than the minimum required it inserts dummy clock cycles to meet the minimum latency.

Syntax

Place the pragma within the boundary of a function, loop, or region of code where the latency must be managed.

```
#pragma HLS latency min=<int> max=<int>
```

Where:

- **min=<int>:** Optionally specifies the minimum latency for the function, loop, or region of code.
- **max=<int>:** Optionally specifies the maximum latency for the function, loop, or region of code.

Note: Although both min and max are described as optional, at least one must be specified.

Example 1

Function `foo` is specified to have a minimum latency of 4 and a maximum latency of 8.

```
int foo(char x, char a, char b, char c) {
    #pragma HLS latency min=4 max=8
    char y;
    y = x*a+b+c;
    return y
}
```

Example 2

In the following example, `loop_1` is specified to have a maximum latency of 12. Place the pragma in the loop body as shown.

```
void foo (num_samples, ...) {
    int i;
    ...
    loop_1: for(i=0;i< num_samples;i++) {
        #pragma HLS latency max=12
        ...
        result = a + b;
    }
}
```

Example 3

The following example creates a code region and groups signals that need to change in the same clock cycle by specifying zero latency.

```
// create a region { } with a latency = 0
{
    #pragma HLS LATENCY max=0 min=0
    *data = 0xFF;
    *data_vld = 1;
}
```

See Also

- [set_directive_latency](#)

pragma HLS loop_flatten

Description

Allows nested loops to be flattened into a single loop hierarchy with improved latency.

In the RTL implementation, it requires one clock cycle to move from an outer loop to an inner loop, and from an inner loop to an outer loop. Flattening nested loops allows them to be optimized as a single loop. This saves clock cycles, potentially allowing for greater optimization of the loop body logic.

Apply the LOOP_FLATTEN pragma to the loop body of the inner-most loop in the loop hierarchy. Only perfect and semi-perfect loops can be flattened in this manner:

- **Perfect loop nests:**

- Only the innermost loop has loop body content.
- There is no logic specified between the loop statements.
- All loop bounds are constant.

- **Semi-perfect loop nests:**

- Only the innermost loop has loop body content.
- There is no logic specified between the loop statements.
- The outermost loop bound can be a variable.

- **Imperfect loop nests:** When the inner loop has variable bounds (or the loop body is not exclusively inside the inner loop), try to restructure the code, or unroll the loops in the loop body to create a perfect loop nest.

Syntax

Place the pragma in the C source within the boundaries of the nested loop.

```
#pragma HLS loop_flatten off
```

Where:

- **off:** Optional keyword. Prevents flattening from taking place, and can prevent some loops from being flattened while all others in the specified location are flattened.



IMPORTANT! The presence of the LOOP_FLATTEN pragma or directive enables the optimization. The addition of *off* disables it.

Example 1

Flattens `loop_1` in function `foo` and all (perfect or semi-perfect) loops above it in the loop hierarchy, into a single loop. Place the pragma in the body of `loop_1`.

```
void foo (num_samples, ...) {
    int i;
    ...
loop_1: for(i=0;i< num_samples;i++) {
    #pragma HLS loop_flatten
    ...
    result = a + b;
}
}
```

Example 2

Prevents loop flattening in `loop_1`.

```
loop_1: for(i=0;i< num_samples;i++) {
    #pragma HLS loop_flatten off
    ...
}
```

See Also

- [set_directive_loop_flatten](#)
- [pragma HLS loop_merge](#)
- [pragma HLS unroll](#)

pragma HLS loop_merge

Description

Merges consecutive loops into a single loop to reduce overall latency, increase sharing, and improve logic optimization. Merging loops:

- Reduces the number of clock cycles required in the RTL to transition between the loop-body implementations.
- Allows the loops be implemented in parallel (if possible).

The LOOP_MERGE pragma will seek to merge all loops within the scope it is placed. For example, if you apply a LOOP_MERGE pragma in the body of a loop, the Vitis HLS tool applies the pragma to any sub-loops within the loop but not to the loop itself.

The rules for merging loops are:

- If the loop bounds are variables, they must have the same value (number of iterations).

- If the loop bounds are constants, the maximum constant value is used as the bound of the merged loop.
- Loops with both variable bounds and constant bounds cannot be merged.
- The code between loops to be merged cannot have side effects. Multiple execution of this code should generate the same results ($a=b$ is allowed, $a=a+1$ is not).
- Loops cannot be merged when they contain FIFO reads. Merging changes the order of the reads. Reads from a FIFO or FIFO interface must always be in sequence.

Syntax

Place the pragma in the C source within the required scope or region of code.

```
#pragma HLS loop_merge force
```

Where:

- **force**: Optional keyword to force loops to be merged even when the HLS tool issues a warning.



IMPORTANT! *In this case, you must manually ensure that the merged loop will function correctly.*

Examples

Merges all consecutive loops in function `foo` into a single loop.

```
void foo (num_samples, ...) {
    #pragma HLS loop_merge
    int i;
    ...
    loop_1: for(i=0;i< num_samples;i++) {
        ...
    }
```

All loops inside `loop_2` (but not `loop_2` itself) are merged by using the `force` option. Place the pragma in the body of `loop_2`.

```
loop_2: for(i=0;i< num_samples;i++) {
    #pragma HLS loop_merge force
    ...
}
```

See Also

- [set_directive_loop_merge](#)
- [pragma HLS loop_flatten](#)
- [pragma HLS unroll](#)

pragma HLS loop_tripcount

Description

When manually applied to a loop, specifies the total number of iterations performed by a loop.



IMPORTANT! The `LOOP_TRIPCOUNT` pragma or directive is for analysis only, and does not impact the results of synthesis.

The Vitis HLS tool reports the total latency of each loop, which is the number of clock cycles to execute all iterations of the loop. Therefore, the loop latency is a function of the number of loop iterations, or tripcount.

The tripcount can be a constant value. It can depend on the value of variables used in the loop expression (for example, `x < y`), or depend on control statements used inside the loop. In some cases, the HLS tool cannot determine the tripcount, and the latency is unknown. This includes cases in which the variables used to determine the tripcount are:

- Input arguments or
- Variables calculated by dynamic operation.

In the following example, the maximum iteration of the for-loop is determined by the value of input `num_samples`. The value of `num_samples` is not defined in the C function, but comes into the function from the outside.

```
void foo (num_samples, ...) {
    int i;
    ...
loop_1: for(i=0;i< num_samples;i++) {
    ...
    result = a + b;
}
```

In cases where the loop latency is unknown or cannot be calculated, the `LOOP_TRIPCOUNT` pragma lets you specify minimum, maximum, and average iterations for a loop. This lets the tool analyze how the loop latency contributes to the total design latency in the reports, and helps you determine appropriate optimizations for the design.



TIP: If a C assert macro is used in to limit the size of a loop variable Vitis HLS can use it to both define loop limits for reporting, and create hardware that is exactly sized to these limits.

Syntax

Place the pragma in the C source within the body of the loop.

```
#pragma HLS loop_tripcount min=<int> max=<int> avg=<int>
```

Where:

- `max= <int>`: Specifies the maximum number of loop iterations.
- `min=<int>`: Specifies the minimum number of loop iterations.
- `avg=<int>`: Specifies the average number of loop iterations.

Examples

In the following example, `loop_1` in function `foo` is specified to have a minimum tripcount of 12, and a maximum tripcount of 16:

```
void foo ( num_samples, ... ) {
    int i;
    ...
    loop_1: for(i=0;i< num_samples;i++) {
        #pragma HLS loop_tripcount min=12 max=16
        ...
        result = a + b;
    }
}
```

See Also

- [set_directive_loop_tripcount](#)

pragma HLS occurrence

Description

When pipelining functions or loops, the OCCURRENCE pragma specifies that the code in a pipelined function call within the pipelined function or loop is executed less frequently than the code in the enclosing function or loop. This allows the pipelined call that is executed less often to be pipelined at a slower rate, thus potentially improving the resource sharing potential within the top-level pipeline. To determine the OCCURRENCE pragma, do the following:

- A loop iterates $<N>$ times.
- However, part of the loop body is enabled by a conditional statement, and as a result only executes $<M>$ times, where $<N>$ is an integer multiple of $<M>$.
- The conditional code has an occurrence that is N/M times slower than the rest of the loop body.

For example, in a loop that executes 10 times, a conditional statement within the loop only executes two times has an occurrence of 5 (or 10/2).

Identifying a region with the OCCURRENCE pragma allows the functions and loops in that region to be pipelined with a higher initiation interval that is slower than the enclosing function or loop.

Syntax

Place the pragma in the C source within a block of code that contains the pipelined function call(s).

```
#pragma HLS occurrence cycle=<int>
```

Where:

- **cycle=<int>**: Specifies the occurrence N/M.
 - <N>: Number of times the enclosing function or loop is executed.
 - <M>: Number of times the conditional region is executed.



IMPORTANT! <N> must be an integer multiple of <M>.

Examples

In this example, the call of subfunction within the if statement (but not the memory read and write, since they are not inside a pipelined function) has an occurrence of 4 (it executes at a rate four times less often than the surrounding code that contains it). Hence, while without the occurrence pragma it would be pipelined with the same II as the caller, with the occurrence pragma it will be pipelined with an II=4. This will expose more resource sharing opportunities within it and with other functions.

```
void subfunction(...) {
#pragma HLS pipeline II=...
// Without the occurrence pragma,
// this will be automatically pipelined with an II=1,
// regardless of its own pipeline pragma,
// since it is called in a pipeline with II=1
// With the pragma, it has an II=4.
...
}
for (int i = 0; i < 16; i++) {
#pragma HLS pipeline II=1
    if (i % 4 == 0) {
#pragma HLS occurrence cycle=4
        subfunction(...);
        a[i+1] = b[i];
    }
}
```

See Also

- [set_directive_occurrence](#)
- [pragma HLS pipeline](#)

pragma HLS performance

Description

Note: The PERFORMANCE pragma applies to loops and loop nests, and requires a known loop tripcount to determine the performance. If your loop has a variable tripcount then you must also specify the TRIPCOUNT pragma.

The PERFORMANCE pragma lets you specify a high-level constraint (`target_ti`) defining the number of clock cycles between successive starts of a loop, and lets the tool infer lower-level UNROLL, PIPELINE, ARRAY_PARTITION, and INLINE pragmas needed to achieve the desired result. The PERFORMANCE pragma does not guarantee the specified value will be achieved, and so it is only a target.

Note: The INLINE pragma is applied automatically to functions inside any pipelined loop that has II=1 to improve throughput. If you apply the PERFORMANCE pragma that infers a pipeline with II=1, it will also trigger the auto-inline optimization. You can disable this for specific functions by using `#pragma HLS INLINE OFF`

The target transaction interval (`target_ti`) specifies a performance target for loops, where a transaction is a complete set of loop iterations (tripcount) and the interval is the time between when the first transaction starts and the second transaction starts.

- **Target Transaction Interval (`target_ti`):** Specifies the number of clock cycles between successive starts of the loop. In other words, the clock cycles from the start of the first transaction of a loop, or nested loop, and the start of the next transaction of the loop.

The transaction interval is the initiation interval (II) of the loop times the number of iterations, or tripcount: $\text{target_ti} = \text{II} * \text{loop tripcount}$. Conversely, $\text{target_ti} = \text{FreqHz} / \text{Operations per second}$.

For example, assuming an image processing function that processes a single frame per invocation with a throughput goal of 60 fps, then the target throughput for the function is 60 invocations per second. If the clock frequency is 180 MHz, then `target_ti` is $180M/60$, or 3 million clock cycles per function invocation.

Syntax

Place the pragma within the boundary a loop, or the outer loop of a loop nest.

```
#pragma HLS performance target_ti=<value>
```

Where:

- `target_ti=<value>`: Specifies a target transaction interval defined as the number of clock cycles for the function, loop, or region of code to complete an iteration. The <value> can be specified as an integer, floating point, or constant expression that is resolved by the tool as an integer.

Note: A warning will be returned if truncation occurs.

Example 1

The outer loop is specified to have target transaction interval of 1000 clock cycles.

```
for (int i =0; i < 1000; ++i) {  
    #pragma HLS performance target_ti=1000  
    for (int j = 0; j < 8; ++j) {  
        int tmp = b_buf[j].read();  
        b[i * 8 + j] = tmp + 2;  
    }  
}
```

See Also

- [set_directive_performance](#)
- [pragma HLS inline](#)

pragma HLS pipeline

Description

Reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations. The default type of pipeline is defined by the `config_compile - pipeline_style` command, but can be overridden in the PIPELINE pragma or directive.

A pipelined function or loop can process new inputs every <N> clock cycles, where <N> is the II of the loop or function. An II of 1 processes a new input every clock cycle. You can specify the initiation interval through the use of the II option for the pragma.

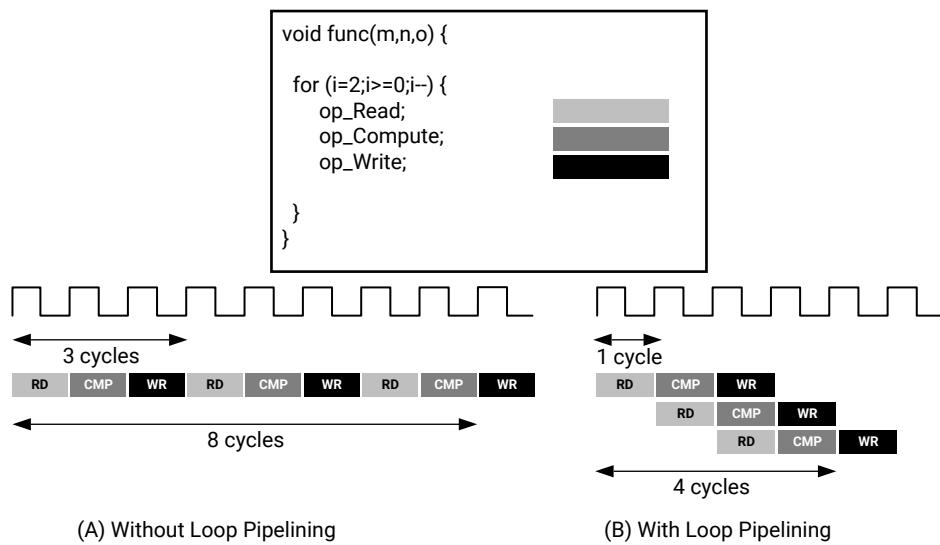
As a default behavior, with the PIPELINE pragma or directive Vitis HLS will generate the minimum II for the design according to the specified clock period constraint. The emphasis will be on meeting timing, rather than on achieving II unless the II option is specified.

If the Vitis HLS tool cannot create a design with the specified II, it issues a warning and creates a design with the lowest possible II.

You can then analyze this design with the warning message to determine what steps must be taken to create a design that satisfies the required initiation interval.

Pipelining a loop allows the operations of the loop to be implemented in a concurrent manner as shown in the following figure. In the figure, (A) shows the default sequential operation where there are three clock cycles between each input read (II=3), and it requires eight clock cycles before the last output write is performed. (B) shows the pipelined operations that show one cycle between reads (II=1), and 4 cycles to the last write.

Figure 130: Loop Pipeline



X14277-110217



IMPORTANT! Loop carry dependencies can prevent pipelining. Use the DEPENDENCE pragma or directive to provide additional information to overcome loop-carry dependencies, and allow loops to be pipelined (or pipelined with lower intervals).

Syntax

Place the pragma in the C source within the body of the function or loop.

```
#pragma HLS pipeline II=<int> off rewind style=<value>
```

Where:

- **II=<int>:** Specifies the desired initiation interval for the pipeline. The HLS tool tries to meet this request. Based on data dependencies, the actual result might have a larger initiation interval.
- **off:** Optional keyword. Turns off pipeline for a specific loop or function. This can be used to disable pipelining for a specific loop when `config_compile -pipeline_loops` is used to globally pipeline loops.

- **rewind:** Optional keyword. Enables rewinding as described in [Rewinding Pipelined Loops for Performance](#). This enables continuous loop pipelining with no pause between one execution of the loop ending and the next execution starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop:
 - Is considered as initialization.
 - Is executed only once in the pipeline.
 - Cannot contain any conditional operations (if-else).



TIP: *This feature is only supported for pipelined loops; it is not supported for pipelined functions.*

- **style=<stp | frp | f1p>:** Specifies the type of pipeline to use for the specified function or loop. For more information on pipeline styles refer to [Flushing Pipelines and Pipeline Types](#). The types of pipelines include:
 - **stp:** Stall pipeline. Runs only when input data is available otherwise it stalls. This is the default setting, and is the type of pipeline used by Vitis HLS for both loop and function pipelining. Use this when a flushable pipeline is not required. For example, when there are no performance or deadlock issue due to stalls.
 - **f1p:** This option defines the pipeline as a flushable pipeline. This type of pipeline typically consumes more resources and/or can have a larger II because resources cannot be shared among pipeline iterations.
 - **frp:** Free-running, flushable pipeline. Runs even when input data is not available. Use this when you need better timing due to reduced pipeline control signal fanout, or when you need improved performance to avoid deadlocks. However, this pipeline style can consume more power as the pipeline registers are clocked even if there is no data.



IMPORTANT! *This is a hint not a hard constraint. The tool checks design conditions for enabling pipelining. Some loops might not conform to a particular style and the tool reverts to the default style (stp) if necessary.*

Examples

In this example, function `func` is pipelined with an initiation interval of 1.

```
void func { a, b, c, d} {
    #pragma HLS pipeline II=1
    ...
}
```

See Also

- [set_directive_pipeline](#)
- [pragma HLS dependence](#)

- [config_compile](#)
-

pragma HLS protocol

Description

This command specifies a region of code, a protocol region, in which no clock operations will be inserted by Vitis HLS unless explicitly specified in the code. Vitis HLS will not insert any clocks between operations in the region, including those which read from or write to function arguments. The order of read and writes will therefore be strictly followed in the synthesized RTL.

A region of code can be created in the C/C++ code by enclosing the region in braces "{}" and naming it. The following defines a region named `io_section`:

```
io_section:{  
...  
lines of code  
...  
}
```

A clock operation can be explicitly specified in C/C++ code using an `ap_wait()` statement, and may be specified in C++ code by using the `wait()` statement. The `ap_wait` and `wait` statements have no effect on the simulation of the design.

Syntax

Place the pragma in the C source within the body of the function or protocol region.

```
#pragma HLS protocol [floating | fixed]
```

Options

- **floating:** Lets code statements outside the protocol region overlap and execute in parallel with statements in the protocol region in the final RTL. The protocol region remains cycle accurate, but outside operations can occur at the same time. This is the default mode.
- **fixed:** The fixed mode ensures that statements outside the protocol region do not execute in parallel with the protocol region.

Examples

This example defines a protocol region, `io_section` in function `foo` where the pragma defines that region as a floating protocol region as the default mode:

```
io_section: {  
#pragma HLS protocol  
...  
}
```

See Also

- [set_directive_protocol](#)

pragma HLS reset

Description

Adds or removes resets for specific state variables (global or static).

The reset port is used to restore the registers and block RAM, connected to the port, to an initial value any time the reset signal is applied. The presence and behavior of the RTL reset port is controlled using the `config_rtl` settings. The reset settings include the ability to set the polarity of the reset, and specify whether the reset is synchronous or asynchronous, but more importantly it controls, through the reset option, which registers are reset when the reset signal is applied. For more information, see [Controlling Initialization and Reset Behavior](#).

Greater control over reset is provided through the RESET pragma. If a variable is a static or global, the RESET pragma is used to explicitly add a reset, or the variable can be removed from the reset by turning `off` the pragma. This can be particularly useful when static or global arrays are present in the design.

Syntax

Place the pragma in the C source within the boundaries of the variable life cycle.

```
#pragma HLS reset variable=<a> off
```

Where:

- `variable=<a>`: Specifies the variable to which the RESET pragma is applied.
- `off`: Indicates that reset is not generated for the specified variable.

Example 1

This example adds reset to the variable `a` in function `foo` even when the global reset setting is `none` or `control`.

```
void foo(int in[3], char a, char b, char c, int out[3]) {  
    #pragma HLS reset variable=a
```

Example 2

Removes reset from variable `a` in function `foo` even when the global reset setting is `state` or `all`.

```
void foo(int in[3], char a, char b, char c, int out[3]) {  
    #pragma HLS reset variable=a off
```

See Also

- [set_directive_reset](#)
- [config_rtl](#)

pragma HLS stable

Description

The STABLE pragma is applied to arguments of a DATAFLOW or PIPELINE region and is used to indicate that an input or output of this region can be ignored when generating the synchronizations at entry and exit of the DATAFLOW region. This means that the reading processes (resp. read accesses) of that argument do not need to be part of the “first stage” of the task-level (resp. fine-grain) pipeline for inputs, and the writing process (resp. write accesses) do not need to be part of the last stage of the task-level (resp. fine-grain) pipeline for outputs.

The pragma can be specified at any point in the hierarchy, on a scalar or an array, and automatically applies to all the DATAFLOW or PIPELINE regions below that point. The effect of STABLE for an input is that a DATAFLOW or PIPELINE region can start another iteration even though the value of the previous iteration has not been read yet. For an output, this implies that a write of the next iteration can occur although the previous iteration is not done.

Syntax

```
#pragma HLS stable variable=<a>
```

Where:

- `variable=<a>`: Specifies the variable to which the STABLE pragma is applied.

Examples

In the following example, without the STABLE pragma, proc1 and proc2 would be synchronized to acknowledge the reading of their inputs (including A). With the pragma, A is no longer considered as an input that needs synchronization.

```
void dataflow_region(int A[...], int B[...] ...  
#pragma HLS stable variable=A  
#pragma HLS dataflow  
    proc1(...);  
    proc2(A, ...);
```

See Also

- [set_directive_stable](#)
- [pragma HLS dataflow](#)
- [pragma HLS pipeline](#)

pragma HLS stream

Description

By default, array variables are implemented as RAM:

- Top-level function array parameters are implemented as a RAM interface port.
- General arrays are implemented as RAMs for read-write access.
- Arrays involved in sub-functions, or loop-based DATAFLOW optimizations are implemented as a RAM ping pong buffer channel.

If the data stored in the array is consumed or produced in a sequential manner, a more efficient communication mechanism is to use streaming data as specified by the STREAM pragma, where FIFOs are used instead of RAMs.



IMPORTANT! When an argument of the top-level function is specified as INTERFACE type `ap_fifo`, the array is automatically implemented as streaming. See [Defining Interfaces](#) for more information.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS stream variable=<variable> type=<type> depth=<int>
```

Where:

- **variable=<variable>**: Specifies the name of the array to implement as a streaming interface.
- **depth=<int>**: Relevant only for array streaming in DATAFLOW channels. By default, the depth of the FIFO implemented in the RTL is the same size as the array specified in the C code. This option lets you modify the size of the FIFO to specify a different depth.

When the array is implemented in a DATAFLOW region, it is common to use the `depth` option to reduce the size of the FIFO. For example, in a DATAFLOW region when all loops and functions are processing data at a rate of `II=1`, there is no need for a large FIFO because data is produced and consumed in each clock cycle. In this case, the `depth` option can be used to reduce the FIFO size to 1 to substantially reduce the area of the RTL design.



TIP: The `config-dataflow -depth` command provides the ability to stream all arrays in a DATAFLOW region. The `depth` option specified in the STREAM pragma overrides the `config-dataflow -depth` setting for the specified `<variable>`.

- **type=<arg>**: Specify a mechanism to select between FIFO, PIPO, synchronized shared (`shared`), and un-synchronized shared (`unsync`). The supported types include:
 - `fifo`: A FIFO buffer with the specified `depth`.
 - `piro`: A regular Ping-Pong buffer, with as many “banks” as the specified depth (default is 2).
 - `shared`: A shared channel, synchronized like a regular Ping-Pong buffer, with depth, but without duplicating the array data. Consistency can be ensured by setting the depth small enough, which acts as the distance of synchronization between the producer and consumer.



TIP: The default depth for `shared` is 1.

- `unsync`: Does not have any synchronization except for individual memory reads and writes. Consistency (read-write and write-write order) must be ensured by the design itself.

Example 1

The following example specifies array `A[10]` to be streaming, and implemented as a FIFO.

```
#pragma HLS STREAM variable=A
```

Example 2

In this example, array `B` is set to streaming with a FIFO depth of 12.

```
#pragma HLS STREAM variable=B depth=12 type=fifo
```

Example 3

Array C has streaming implemented as a PIPO.

```
#pragma HLS STREAM variable=C type=piro
```

See Also

- [set_directive_stream](#)
- [pragma HLS dataflow](#)
- [pragma HLS interface](#)
- [config_dataflow](#)

pragma HLS top

Description

Attaches a name to a function, which can then be used with the `set_top` command to synthesize the function and any functions called from the specified top-level. This is typically used to synthesize member functions of a class in C/C++.

Specify the TOP pragma in an active solution, and then use the `set_top` command with the new name.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS top name=<string>
```

Where:

- `name=<string>`: Specifies the name to be used by the `set_top` command.

Examples

Function `foo_long_name` is designated the top-level function, and renamed to `DESIGN_TOP`. After the pragma is placed in the code, the `set_top` command must still be issued from the Tcl command line, or from the top-level specified in the IDE project settings.

```
void foo_long_name () {  
    #pragma HLS top name=DESIGN_TOP  
    ...  
}
```

Followed by the `set_top DESIGN_TOP` command.

See Also

- [set_directive_top](#)
- [set_top](#)

pragma HLS unroll

Description

You can unroll loops to create multiple independent operations rather than a single collection of operations. The UNROLL pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel.

Loops in the C/C++ functions are kept rolled by default. When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations might also be impacted by logic inside the loop body (for example, `break` conditions or modifications to a loop exit variable). Using the UNROLL pragma you can unroll loops to increase data access and throughput.

The UNROLL pragma allows the loop to be fully or partially unrolled. Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently. Partially unrolling a loop lets you specify a factor N , to create N copies of the loop body and reduce the loop iterations accordingly.



TIP: To unroll a loop completely, the loop bounds must be known at compile time. This is not required for partial unrolling.

Partial loop unrolling does not require N to be an integer factor of the maximum loop iteration count. The Vitis HLS tool adds an exit check to ensure that partially unrolled loops are functionally identical to the original loop. For example, given the following code:

```
for(int i = 0; i < X; i++) {  
    pragma HLS unroll factor=2  
    a[i] = b[i] + c[i];  
}
```

Loop unrolling by a factor of 2 effectively transforms the code to look like the following code where the `break` construct is used to ensure the functionality remains the same, and the loop exits at the appropriate point.

```
for(int i = 0; i < X; i += 2) {  
    a[i] = b[i] + c[i];  
    if (i+1 >= X) break;  
    a[i+1] = b[i+1] + c[i+1];  
}
```

In the example above, because the maximum iteration count, `X`, is a variable, the HLS tool might not be able to determine its value, so it adds an exit check and control logic to partially unrolled loops. However, if you know that the specified unrolling factor, 2 in this example, is an integer factor of the maximum iteration count `X`, the `skip_exit_check` option lets you remove the exit check and associated logic. This helps minimize the area and simplify the control logic.



TIP: When the use of pragmas like `ARRAY_PARTITION` or `ARRAY_RESHAPE` let more data be accessed in a single clock cycle, the HLS tool automatically unrolls any loops consuming this data, if doing so improves the throughput. The loop can be fully or partially unrolled to create enough hardware to consume the additional data in a single clock cycle. This automatic unrolling is controlled using the `config_unroll` command.

Syntax

Place the pragma in the C source within the body of the loop to unroll.

```
#pragma HLS unroll factor=<N> region skip_exit_check
```

Where:

- **factor=<N>**: Specifies a non-zero integer indicating that partial unrolling is requested. The loop body is repeated the specified number of times, and the iteration information is adjusted accordingly. If `factor=` is not specified, the loop is fully unrolled.
- **skip_exit_check**: Optional keyword that applies only if partial unrolling is specified with `factor=`. The elimination of the exit check is dependent on whether the loop iteration count is known or unknown:
 - **Fixed bounds**
No exit condition check is performed if the iteration count is a multiple of the factor.
If the iteration count is not an integer multiple of the factor, the tool:
 - Prevents unrolling.
 - Issues a warning that the exit check must be performed to proceed.
 - **Variable bounds**
The exit condition check is removed. You must ensure that:

- The variable bounds is an integer multiple of the factor.
- No exit check is in fact required.

Example 1

The following example fully unrolls `loop_1` in function `foo`. Place the pragma in the body of `loop_1` as shown.

```
loop_1: for(int i = 0; i < N; i++) {  
    #pragma HLS unroll  
    a[i] = b[i] + c[i];  
}
```

Example 2

This example specifies an unroll factor of 4 to partially unroll `loop_2` of function `foo`, and removes the exit check.

```
void foo (...) {  
    int8 array1[M];  
    int12 array2[N];  
    ...  
    loop_2: for(i=0;i<M;i++) {  
        #pragma HLS unroll skip_exit_check factor=4  
        array1[i] = ...;  
        array2[i] = ...;  
        ...  
    }  
    ...  
}
```

Example 3

The following example fully unrolls all loops inside `loop_1` in function `foo`, but not `loop_1` itself because the presence of the `region` keyword.

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {  
    int temp1[N];  
    loop_1: for(int i = 0; i < N; i++) {  
        #pragma HLS unroll region  
        temp1[i] = data_in[i] * scale;  
        loop_2: for(int j = 0; j < N; j++) {  
            data_out1[j] = temp1[j] * 123;  
        }  
        loop_3: for(int k = 0; k < N; k++) {  
            data_out2[k] = temp1[k] * 456;  
        }  
    }  
}
```

See Also

- [set_directive_unroll](#)

- [pragma HLS loop_flatten](#)
- [pragma HLS loop_merge](#)

Vitis HLS C Driver Reference

This section contains the following chapter:

- [AXI4-Lite Slave C Driver Reference](#)

AXI4-Lite Slave C Driver Reference

X<DUT>_Initialize

Syntax

```
int X<DUT>_Initialize(X<DUT> *InstancePtr, u16 DeviceId);  
int X<DUT>_Initialize(X<DUT> *InstancePtr, const char* InstanceName);
```

Description

int X<DUT>_Initialize(X<DUT> *InstancePtr, u16 DeviceId): For use on standalone systems, initialize a device. This API will write a proper value to `InstancePtr` which then can be used in other APIs. Xilinx recommends calling this API to initialize a device except when an MMU is used in the system, in which case refer to function `X<DUT>_CfgInitialize`.

int X<DUT>_Initialize(X<DUT> *InstancePtr, const char* InstanceName): For use on Linux systems, initialize a specifically named `uio` device. Create up to five memory mappings and assign the slave base addresses by `mmap`, utilizing the `uio` device information in `sysfs`.

- `InstancePtr`: A pointer to the device instance.
- `DeviceId`: Device ID as defined in `xparameters.h`.
- `InstanceName`: The name of the `uio` device.

Return

`XST_SUCCESS` indicates success, otherwise fail.

X<DUT>_CfgInitialize

Syntax

```
X<DUT>_CfgInitialize int X<DUT>_CfgInitialize(X<DUT> *InstancePtr,  
X<DUT>_Config *ConfigPtr);
```

Description

Initialize a device when an MMU is used in the system. In such a case the effective address of the AXI4-Lite slave is different from that defined in `xparameters.h` and API is required to initialize the device.

- `InstancePtr`: A pointer to the device instance.
- `DeviceId`: A pointer to a `X<DUT>_Config`.

Return

`XST_SUCCESS` indicates success, otherwise fail.

X<DUT>_LookupConfig

Syntax

```
X<DUT>_Config * X<DUT>_LookupConfig(u16 DeviceId);
```

Description

This function is used to obtain the configuration information of the device by ID.

- `DeviceId`: Device ID as defined in `xparameters.h`.

Return

A pointer to a `X<DUT>_LookupConfig` variable that holds the configuration information of the device whose ID is `DeviceId`. `NULL` if no matching `DeviceId` is found.

X<DUT>_Release

Syntax

```
int X<DUT>_Release(X<DUT> *InstancePtr);
```

Description

Release the `uio` device. Delete the mappings by `munmap`. The mapping will automatically be deleted if the process is terminated.

- `InstanceName`: The name of the `uio` device.

Return

`XST_SUCCESS` indicates success, otherwise fail.

X<DUT>_Start

Syntax

```
void X<DUT>_Start(X<DUT> *InstancePtr);
```

Description

Start the device. This function will assert the `ap_start` port on the device. Available only if there is `ap_start` port on the device.

- `InstancePtr`: A pointer to the device instance.
-

X<DUT>_IsDone

Syntax

```
void X<DUT>_IsDone(X<DUT> *InstancePtr);
```

Description

Check if the device has finished the previous execution: this function will return the value of the `ap_done` port on the device. Available only if there is an `ap_done` port on the device.

- `InstancePtr`: A pointer to the device instance.
-

X<DUT>_IsIdle

Syntax

```
void X<DUT>_IsIdle(X<DUT> *InstancePtr);
```

Description

Check if the device is in idle state: this function will return the value of the `ap_idle` port. Available only if there is an `ap_idle` port on the device.

- `InstancePtr`: A pointer to the device instance.
-

X<DUT>_IsReady

Syntax

```
void X<DUT>_IsReady(X<DUT> *InstancePtr);
```

Description

Check if the device is ready for the next input: this function will return the value of the `ap_ready` port. Available only if there is an `ap_ready` port on the device.

- `InstancePtr`: A pointer to the device instance.
-

X<DUT>_Continue

Syntax

```
void XExample_Continue(XExample *InstancePtr);
```

Description

Assert port `ap_continue`. Available only if there is an `ap_continue` port on the device.

- `InstancePtr`: A pointer to the device instance.

X<DUT>_EnableAutoRestart

Syntax

```
void X<DUT>_EnableAutoRestart(X<DUT> *InstancePtr);
```

Description

Enables “auto restart” on device. When this is enabled,

- Port `ap_start` will be asserted as soon as `ap_done` is asserted by the device and the device will auto-start the next transaction.
- Alternatively, if the block-level I/O protocol `ap_ctrl_chain` is implemented on the device, the next transaction will auto-restart (`ap_start` will be asserted) when `ap_ready` is asserted by the device and if `ap_continue` is asserted when `ap_done` is asserted by the device.

Available only if there is an `ap_start` port.

- `InstancePtr`: A pointer to the device instance.

X<DUT>_DisableAutoRestart

Syntax

```
void X<DUT>_DisableAutoRestart(X<DUT> *InstancePtr);
```

Description

Disable the “auto restart” function. Available only if there is an `ap_start` port.

- `InstancePtr`: A pointer to the device instance.

X<DUT>_Set_ARG

Syntax

```
void X<DUT>_Set_ARG(X<DUT> *InstancePtr, u32 Data);
```

Description

Write a value to port ARG (a scalar argument of the top-level function). Available only if ARG is an input port.

- `InstancePtr`: A pointer to the device instance.
- `Data`: Value to write.

X<DUT>_Set_ARG_vld

Syntax

```
void X<DUT>_Set_ARG_vld(X<DUT> *InstancePtr);
```

Description

Assert port ARG_vld. Available only if ARG is an input port and implemented with an ap_hs or ap_vld interface protocol.

- `InstancePtr`: A pointer to the device instance.

X<DUT>_Set_ARG_ack

Syntax

```
void X<DUT>_Set_ARG_ack(X<DUT> *InstancePtr);
```

Description

Assert port ARG_ack. Available only if ARG is an output port and implemented with an ap_hs or ap_ack interface protocol.

- `InstancePtr`: A pointer to the device instance.

X<DUT>_Get_ARG

Syntax

```
u32 X<DUT>_Get_ARG(X<DUT> *InstancePtr);
```

Description

Read a value from ARG. Only available if port ARG is an output port on the device.

- `InstancePtr`: A pointer to the device instance.

Return

Value of ARG.

X<DUT>_Get_ARG_vld

Syntax

```
u32 X<DUT>_Get_ARG_vld(X<DUT> *InstancePtr);
```

Description

Read a value from ARG_vld. Only available if port ARG is an output port on the device and implemented with an ap_hs or ap_vld interface protocol.

- `InstancePtr`: A pointer to the device instance.

Return

Value of ARG_vld.

X<DUT>_Get_ARG_ack

Syntax

```
u32 X<DUT>_Get_ARG_ack(X<DUT> *InstancePtr);
```

Description

Read a value from ARG_ack. Only available if port ARG is an input port on the device and implemented with an ap_hs or ap_ack interface protocol.

- `InstancePtr`: A pointer to the device instance.

Return

Value of ARG_ack.

X<DUT>_Get_ARG_BaseAddress

Syntax

```
u32 X<DUT>_Get_ARG_BaseAddress(X<DUT> *InstancePtr);
```

Description

Return the base address of the array inside the interface. Only available when ARG is an array grouped into the AXI4-Lite interface.

- `InstancePtr`: A pointer to the device instance.

Return

Base address of the array.

X<DUT>_Get_ARG_HighAddress

Syntax

```
u32 X<DUT>_Get_ARG_HighAddress(X<DUT> *InstancePtr);
```

Description

Return the address of the uppermost element of the array. Only available when ARG is an array grouped into the AXI4-Lite interface.

- `InstancePtr`: A pointer to the device instance.

Return

Address of the uppermost element of the array.

X<DUT>_Get_ARG_TotalBytes

Syntax

```
u32 X<DUT>_Get_ARG_TotalBytes(X<DUT> *InstancePtr);
```

Description

Return the total number of bytes used to store the array. Only available when ARG is an array grouped into the AXI4-Lite interface.

If the elements in the array are less than 16-bit, Vitis™ HLS groups multiple elements into the 32-bit data width of the AXI4-Lite interface. If the bit width of the elements exceeds 32-bit, Vitis HLS stores each element over multiple consecutive addresses.

- `InstancePtr`: A pointer to the device instance.

Return

The total number of bytes used to store the array.

X<DUT>_Get_ARG_BitWidth

Syntax

```
u32 X<DUT>_Get_ARG_BitWidth(X<DUT> *InstancePtr);
```

Description

Return the bit width of each element in the array. Only available when ARG is an array grouped into the AXI4-Lite interface.

If the elements in the array are less than 16-bit, Vitis HLS groups multiple elements into the 32-bit data width of the AXI4-Lite interface. If the bit width of the elements exceeds 32-bit, Vitis HLS stores each element over multiple consecutive addresses.

- `InstancePtr`: A pointer to the device instance.

Return

The bit-width of each element in the array.

X<DUT>_Get_ARG_Depth

Syntax

```
u32 X<DUT>_Get_ARG_Depth(X<DUT> *InstancePtr);
```

Description

Return the total number of elements in the array. Only available when `ARG` is an array grouped into the AXI4-Lite interface.

If the elements in the array are less than 16-bit, Vitis HLS groups multiple elements into the 32-bit data width of the AXI4-Lite interface. If the bit width of the elements exceeds 32-bit, Vitis HLS stores each element over multiple consecutive addresses.

- `InstancePtr`: A pointer to the device instance.

Return

The total number of elements in the array.

X<DUT>_Write_ARG_Words

Syntax

```
u32 X<DUT>_Write_ARG_Words(X<DUT> *InstancePtr, int offset, int *data, int length);
```

Description

Write the length of a 32-bit word into the specified address of the AXI4-Lite interface. This API requires the offset address from `BaseAddress` and the length of the data to be stored. Only available when `ARG` is an array grouped into the AXI4-Lite interface.

- `InstancePtr`: A pointer to the device instance.
- `offset`: The address in the AXI4-Lite interface.
- `data`: A pointer to the data value to be stored.

- `length`: The length of the data to be stored.

Return

Write length of data from the specified address.

X<DUT>_Read_ARG_Words

Syntax

```
u32 X<DUT>_Read_ARG_Words(X<DUT> *InstancePtr, int offset, int *data, int length);
```

Description

Read the length of a 32-bit word from the array. This API requires the data target, the offset address from `BaseAddress`, and the length of the data to be stored. Only available when `ARG` is an array grouped into the AXI4-Lite interface.

- `InstancePtr`: A pointer to the device instance.
- `offset`: The address in the `ARG`.
- `data`: A pointer to the data buffer.
- `length`: The length of the data to be stored.

Return

Read length of data from the specified address.

X<DUT>_Write_ARG_Bytes

Syntax

```
u32 X<DUT>_Write_ARG_Bytes(X<DUT> *InstancePtr, int offset, char *data, int length);
```

Description

Write the length of bytes into the specified address of the AXI4-Lite interface. This API requires the offset address from `BaseAddress` and the length of the data to be stored. Only available when `ARG` is an array grouped into the AXI4-Lite interface.

- `InstancePtr`: A pointer to the device instance.
- `offset`: The address in the ARG.
- `data`: A pointer to the data value to be stored.
- `length`: The length of the data to be stored.

Return

Write length of data from the specified address.

X<DUT>_Read_ARG_Bytes

Syntax

```
u32 X<DUT>_Read_ARG_Bytes(X<DUT> *InstancePtr, int offset, char *data, int length);
```

Description

Read the length of bytes from the array. This API requires the data target, the offset address from Base Address, and the length of data to be loaded. Only available when ARG is an array grouped into the AXI4-Lite interface.

- `InstancePtr`: A pointer to the device instance.
- `offset`: The address in the ARG.
- `data`: A pointer to the data buffer.
- `length`: The length of the data to be loaded.

Return

Read length of data from the specified address.

X<DUT>_InterruptGlobalEnable

Syntax

```
void X<DUT>_InterruptGlobalEnable(X<DUT> *InstancePtr);
```

Description

Enable the interrupt output. Interrupt functions are available only if there is `ap_start`.

- `InstancePtr`: A pointer to the device instance.

X<DUT>_InterruptGlobalDisable

Syntax

```
void X<DUT>_InterruptGlobalDisable(X<DUT> *InstancePtr);
```

Description

Disable the interrupt output.

- `InstancePtr`: A pointer to the device instance.

X<DUT>_InterruptEnable

Syntax

```
void X<DUT>_InterruptEnable(X<DUT> *InstancePtr, u32 Mask);
```

Description

Enable the interrupt source. There can be at most two interrupt sources (`source 0` for `ap_done` and `source 1` for `ap_ready`).

- `InstancePtr`: A pointer to the device instance.
- `Mask`: Bit mask.
 - Bit $n = 1$: enable interrupt source n .
 - Bit $n = 0$: no change.

X<DUT>_InterruptDisable

Syntax

```
void X<DUT>_InterruptDisable(X<DUT> *InstancePtr, u32 Mask);
```

Description

Disable the interrupt source.

- **InstancePtr:** A pointer to the device instance.
- **Mask:** Bit mask.
 - Bit n = 1: disable interrupt source n.
 - Bit n = 0: no change.

X<DUT>_InterruptClear

Syntax

```
void X<DUT>_InterruptClear(X<DUT> *InstancePtr, u32 Mask);
```

Description

Clear the interrupt status.

- **InstancePtr:** A pointer to the device instance.
- **Mask:** Bit mask.
 - Bit n = 1: toggle interrupt source n.
 - Bit n = 0: no change.

X<DUT>_InterruptGetEnabled

Syntax

```
u32 X<DUT>_InterruptGetEnabled(X<DUT> *InstancePtr);
```

Description

Check which interrupt sources are enabled.

- `InstancePtr`: A pointer to the device instance.

Return

Bit mask.

- Bit n = 1: enabled.
- Bit n = 0: disabled.

X<DUT>_InterruptGetStatus

Syntax

```
u32 X<DUT>_InterruptGetStatus(X<DUT> *InstancePtr);
```

Description

Check which interrupt sources are triggered.

- `InstancePtr`: A pointer to the device instance.

Return

Bit mask.

- Bit n = 1: triggered.
- Bit n = 0: not triggered.

Vitis HLS Libraries Reference

This section contains the following chapters:

- [Arbitrary Precision Data Types Library](#)
- [Vitis HLS Math Library](#)
- [HLS Stream Library](#)
- [HLS IP Libraries](#)

C/C++ Builtin Functions

Vitis HLS supports the following C/C++ builtin functions:

- `__builtin_clz(unsigned int x)`: Returns the number of leading 0-bits in `x`, starting at the most significant bit position. If `x` is 0, the result is undefined.
- `__builtin_ctz(unsigned int x)`: Returns the number of trailing 0-bits in `x`, starting at the least significant bit position. If `x` is 0, the result is undefined.

The following example shows how these functions may be used. This example returns the sum of the number of leading zeros in `in0` and trailing zeros in `in1`:

```
int foo (int in0, int in1) {
    int ldz0 = __builtin_clz(in0);
    int ldz1 = __builtin_ctz(in1);
    return (ldz0 + ldz1);
}
```

Arbitrary Precision Data Types Library

C-based native data types are on 8-bit boundaries (8, 16, 32, 64 bits). RTL buses (corresponding to hardware) support arbitrary lengths. HLS needs a mechanism to allow the specification of arbitrary precision bit-width and not rely on the artificial boundaries of native C data types: if a 17-bit multiplier is required, you should not be forced to implement this with a 32-bit multiplier.

Vitis™ HLS provides both integer and fixed-point arbitrary precision data types for C++. The advantage of arbitrary precision data types is that they allow the C code to be updated to use variables with smaller bit-widths and then for the C simulation to be re-executed to validate that the functionality remains identical or acceptable.

Using Arbitrary Precision Data Types

Vitis HLS provides arbitrary precision integer data types that manage the value of the integer numbers within the boundaries of the specified width, as shown in the following table.

Table 40: Arbitrary Precision Data Types

Language	Integer Data Type	Required Header
C++	ap_[u]int<W> (1024 bits) Can be extended to 4K bits wide as explained in C++ Arbitrary Precision Integer Types .	#include "ap_int.h"
C++	ap_[u]fixed<W,I,Q,O,N>	#include "ap_fixed.h"

The header files define the arbitrary precision types are also provided with Vitis HLS as a standalone package with the rights to use them in your own source code. The package, `xilinx_hls_lib_<release_number>.tgz`, is provided in the `include` directory in the Vitis HLS installation area.

Arbitrary Integer Precision Types with C++

The header file `ap_int.h` defines the arbitrary precision integer data type for the C++ `ap_[u]int` data types. To use arbitrary precision integer data types in a C++ function:

- Add header file `ap_int.h` to the source code.
- Change the bit types to `ap_int<N>` for signed types or `ap_uint<N>` for unsigned types, where `N` is a bit-size from 1 to 1024.

The following example shows how the header file is added and two variables implemented to use 9-bit integer and 10-bit unsigned integer types:

```
#include "ap_int.h"

void foo_top () {
    ap_int<9> var1;           // 9-bit
    ap_uint<10> var2;         // 10-bit unsigned
```

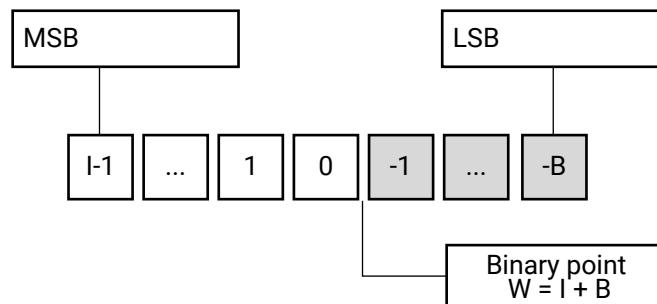
 **IMPORTANT!** One disadvantage of AP data types is that arrays are not automatically initialized with a value of 0. You must manually initialize the array if desired.

Arbitrary Precision Fixed-Point Data Types

In Vitis HLS, it is important to use fixed-point data types, because the behavior of the C++ simulations performed using fixed-point data types match that of the resulting hardware created by synthesis. This allows you to analyze the effects of bit-accuracy, quantization, and overflow with fast C-level simulation.

These data types manage the value of real (non-integer) numbers within the boundaries of a specified total width and integer width, as shown in the following figure.

Figure 131: Fixed-Point Data Type



X14268-100620

Fixed-Point Identifier Summary

The following table provides a brief overview of operations supported by fixed-point types.

Table 41: Fixed-Point Identifier Summary

Identifier	Description																	
W	Word length in bits																	
I	The number of bits used to represent the integer value, that is, the number of integer bits to the <i>left</i> of the binary point. When this value is negative, it represents the number of <i>implicit</i> sign bits (for signed representation), or the number of <i>implicit</i> zero bits (for unsigned representation) to the <i>right</i> of the binary point. For example, <pre>ap_fixed<2, 0> a = -0.5; // a can be -0.5, ap_ufixed<1, 0> x = 0.5; // 1-bit representation. x can be 0 or 0.5 ap_ufixed<1, -1> y = 0.25; // 1-bit representation. y can be 0 or 0.25 const ap_fixed<1, -7> z = 1.0/256; // 1-bit representation for z = 2^-8</pre>																	
Q	Quantization mode: This dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result. <table border="1"> <thead> <tr> <th>ap_fixed Types</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>AP_RND</td> <td>Round to plus infinity</td> </tr> <tr> <td>AP_RND_ZERO</td> <td>Round to zero</td> </tr> <tr> <td>AP_RND_MIN_INF</td> <td>Round to minus infinity</td> </tr> <tr> <td>AP_RND_INF</td> <td>Round to infinity</td> </tr> <tr> <td>AP_RND_CONV</td> <td>Convergent rounding</td> </tr> <tr> <td>AP_TRN</td> <td>Truncation to minus infinity (default)</td> </tr> <tr> <td>AP_TRN_ZERO</td> <td>Truncation to zero</td> </tr> </tbody> </table>		ap_fixed Types	Description	AP_RND	Round to plus infinity	AP_RND_ZERO	Round to zero	AP_RND_MIN_INF	Round to minus infinity	AP_RND_INF	Round to infinity	AP_RND_CONV	Convergent rounding	AP_TRN	Truncation to minus infinity (default)	AP_TRN_ZERO	Truncation to zero
ap_fixed Types	Description																	
AP_RND	Round to plus infinity																	
AP_RND_ZERO	Round to zero																	
AP_RND_MIN_INF	Round to minus infinity																	
AP_RND_INF	Round to infinity																	
AP_RND_CONV	Convergent rounding																	
AP_TRN	Truncation to minus infinity (default)																	
AP_TRN_ZERO	Truncation to zero																	
O	Overflow mode: This dictates the behavior when the result of an operation exceeds the maximum (or minimum in the case of negative numbers) possible value that can be stored in the variable used to store the result. <table border="1"> <thead> <tr> <th>ap_fixed Types</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>AP_SAT¹</td> <td>Saturation</td> </tr> <tr> <td>AP_SAT_ZERO¹</td> <td>Saturation to zero</td> </tr> <tr> <td>AP_SAT_SYM¹</td> <td>Symmetrical saturation</td> </tr> <tr> <td>AP_WRAP</td> <td>Wrap around (default)</td> </tr> <tr> <td>AP_WRAP_SM</td> <td>Sign magnitude wrap around</td> </tr> </tbody> </table>		ap_fixed Types	Description	AP_SAT ¹	Saturation	AP_SAT_ZERO ¹	Saturation to zero	AP_SAT_SYM ¹	Symmetrical saturation	AP_WRAP	Wrap around (default)	AP_WRAP_SM	Sign magnitude wrap around				
ap_fixed Types	Description																	
AP_SAT ¹	Saturation																	
AP_SAT_ZERO ¹	Saturation to zero																	
AP_SAT_SYM ¹	Symmetrical saturation																	
AP_WRAP	Wrap around (default)																	
AP_WRAP_SM	Sign magnitude wrap around																	
N	This defines the number of saturation bits in overflow wrap modes.																	

Notes:

- Using the AP_SAT* modes can result in higher resource usage as extra logic will be needed to perform saturation and this extra cost can be as high as 20% additional LUT usage.
- Fixed-point math functions from the `hls_math` library do not support the `ap_[u]fixed` template parameters Q,O, and N, for quantization mode, overflow mode, and the number of saturation bits, respectively. The quantization and overflow modes are only effective when an `ap_[u]fixed` variable is on the left hand of assignment or being initialized, but not during the calculation.

Example Using ap_fixed

In this example the Vitis HLS `ap_fixed` type is used to define an 18-bit variable with 6 bits representing the numbers above the decimal point and 12-bits representing the value below the decimal point. The variable is specified as signed, the quantization mode is set to round to plus infinity and the default wrap-around mode is used for overflow.

```
#include <ap_fixed.h>
...
ap_fixed<18, 6, AP_RND> my_type;
...
```

C++ Arbitrary Precision Integer Types

The native data types in C++ are on 8-bit boundaries (8, 16, 32, and 64 bits). RTL signals and operations support arbitrary bit-lengths.

Vitis HLS provides arbitrary precision data types for C++ to allow variables and operations in the C++ code to be specified with any arbitrary bit-widths: 6-bit, 17-bit, 234-bit, up to 1024 bits.



TIP: The default maximum width allowed is 1024 bits. You can override this default by defining the macro `AP_INT_MAX_W` with a positive integer value less than or equal to 4096 before inclusion of the `ap_int.h` header file.

Arbitrary precision data types have two primary advantages over the native C++ types:

- Better quality hardware: If for example, a 17-bit multiplier is required, arbitrary precision types can specify that exactly 17-bit are used in the calculation.

Without arbitrary precision data types, such a multiplication (17-bit) must be implemented using 32-bit integer data types and result in the multiplication being implemented with multiple DSP modules.

- Accurate C++ simulation/analysis: Arbitrary precision data types in the C++ code allows the C++ simulation to be performed using accurate bit-widths and for the C++ simulation to validate the functionality (and accuracy) of the algorithm before synthesis.

The arbitrary precision types in C++ have none of the disadvantages of those in C:

- C++ arbitrary types can be compiled with standard C++ compilers (there is no C++ equivalent of `apcc`).
- C++ arbitrary precision types do not suffer from Integer Promotion Issues.

It is not uncommon for users to change a file extension from `.c` to `.cpp` so the file can be compiled as C++, where neither of these issues are present.

For the C++ language, the header file `ap_int.h` defines the arbitrary precision integer data types `ap_(u)int<W>`. For example, `ap_int<8>` represents an 8-bit signed integer data type and `ap_uint<234>` represents a 234-bit unsigned integer type.

The `ap_int.h` file is located in the directory `$HLS_ROOT/include`, where `$HLS_ROOT` is the Vitis HLS installation directory.

The code shown in the following example is a repeat of the code shown in the Basic Arithmetic example in [Standard Types](#). In this example, the data types in the top-level function to be synthesized are specified as `dinA_t`, `dinB_t`, and so on.

```
#include "cpp_ap_int_arith.h"

void cpp_ap_int_arith(din_A inA, din_B inB, din_C inC, din_D inD,
    dout_1 *out1, dout_2 *out2, dout_3 *out3, dout_4 *out4
) {

    // Basic arithmetic operations
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;

}
```

In this latest update to this example, the C++ arbitrary precision types are used:

- Add header file `ap_int.h` to the source code.
- Change the native C++ types to arbitrary precision types `ap_int<N>` or `ap_uint<N>`, where `N` is a bit-size from 1 to 1024 (as noted above, this can be extended to 4K-bits if required).

The data types are defined in the header `cpp_ap_int_arith.h`.

Compared with the Basic Arithmetic example in [Standard Types](#), the input data types have simply been reduced to represent the maximum size of the real input data (for example, 8-bit input `inA` is reduced to 6-bit input). The output types have been refined to be more accurate, for example, `out2`, the sum of `inA` and `inB`, need only be 13-bit and not 32-bit.

The following example shows basic arithmetic with C++ arbitrary precision types.

```
#ifndef _CPP_AP_INT_ARITH_H_
#define _CPP_AP_INT_ARITH_H_

#include <stdio.h>
#include "ap_int.h"

#define N 9

// Old data types
//typedef char dinA_t;
//typedef short dinB_t;
//typedef int dinC_t;
//typedef long long dinD_t;
//typedef int dout1_t;
```

```
//typedef unsigned int dout2_t;
//typedef int32_t dout3_t;
//typedef int64_t dout4_t;

typedef ap_int<6> dinA_t;
typedef ap_int<12> dinB_t;
typedef ap_int<22> dinC_t;
typedef ap_int<33> dinD_t;

typedef ap_int<18> dout1_t;
typedef ap_uint<13> dout2_t;
typedef ap_int<22> dout3_t;
typedef ap_int<6> dout4_t;

void cpp_ap_int_arith(dinA_t inA,dinB_t inB,dinC_t inC,dinD_t inD,dout1_t
*out1,dout2_t *out2,dout3_t *out3,dout4_t *out4);

#endif
```

If [C++ Arbitrary Precision Integer Types](#) are synthesized, it results in a design that is functionally identical to [Standard Types](#). Rather than use the C++ `cout` operator to output the results to a file, the built-in `ap_int` method `.to_int()` is used to convert the `ap_int` results to integer types used with the standard `fprintf` function.

```
fprintf(fp, %d*%d=%d; %d+%d=%d; %d/%d=%d; %d mod %d=%d;\n,
    inA.to_int(), inB.to_int(), out1.to_int(),
    inB.to_int(), inA.to_int(), out2.to_int(),
    inC.to_int(), inA.to_int(), out3.to_int(),
    inD.to_int(), inA.to_int(), out4.to_int());
```

C++ Arbitrary Precision Integer Types: Reference Information

For comprehensive information on the methods, synthesis behavior, and all aspects of using the `ap_(u)int<N>` arbitrary precision data types, see [C++ Arbitrary Precision Types](#). This section includes:

- Techniques for assigning constant and initialization values to arbitrary precision integers (including values greater than 1024-bit).
- A description of Vitis HLS helper methods, such as printing, concatenating, bit-slicing and range selection functions.
- A description of operator behavior, including a description of shift operations (a negative shift value, results in a shift in the opposite direction).

C++ Arbitrary Precision Types

Vitis HLS provides a C++ template class, `ap_[u]int<>`, that implements arbitrary precision (or bit-accurate) integer data types with consistent, bit-accurate behavior between software and hardware modeling.

This class provides all arithmetic, bitwise, logical and relational operators allowed for native C integer types. In addition, this class provides methods to handle some useful hardware operations, such as allowing initialization and conversion of variables of widths greater than 64 bits. Details for all operators and class methods are discussed below.

Compiling ap_[u]int<> Types

To use the `ap_[u]int<>` classes, you must include the `ap_int.h` header file in all source files that reference `ap_[u]int<>` variables.

When compiling software models that use these classes, it may be necessary to specify the location of the Vitis HLS header files, for example by adding the `-I/<HLS_HOME>/include` option for `g++` compilation.

Declaring/Defining ap_[u] Variables

There are separate signed and unsigned classes:

- `ap_int<int_W>` (signed)
- `ap_uint<int_W>` (unsigned)

The template parameter `int_W` specifies the total width of the variable being declared.

User-defined types may be created with the C/C++ `typedef` statement as shown in the following examples:

```
include "ap_int.h">// use ap_[u]fixed<> types
typedef ap_uint<128> uint128_t; // 128-bit user defined type
ap_int<96> my_wide_var; // a global variable declaration
```

The default maximum width allowed is 1024 bits. This default may be overridden by defining the macro `AP_INT_MAX_W` with a positive integer value less than or equal to 4096 before inclusion of the `ap_int.h` header file.



CAUTION! Setting the value of `AP_INT_MAX_W` too High can cause slow software compile and runtimes.

Following is an example of overriding `AP_INT_MAX_W`:

```
#define AP_INT_MAX_W 4096 // Must be defined before next line
#include "ap_int.h"

ap_int<4096> very_wide_var;
```

Initialization and Assignment from Constants (Literals)

The class constructor and assignment operator overloads, allows initialization of and assignment to `ap_[u]int<>` variables using standard C/C++ integer literals.

This method of assigning values to `ap_[u]int<>` variables is subject to the limitations of C++ and the system upon which the software will run. This typically leads to a 64-bit limit on integer literals (for example, for those `LL` or `ULL` suffixes).

To allow assignment of values wider than 64-bits, the `ap_[u]int<>` classes provide constructors that allow initialization from a string of arbitrary length (less than or equal to the width of the variable).

By default, the string provided is interpreted as a hexadecimal value as long as it contains only valid hexadecimal digits (that is, 0-9 and a-f). To assign a value from such a string, an explicit C++ style cast of the string to the appropriate type must be made.

Following are examples of initialization and assignments, including for values greater than 64-bit, are:

```
ap_int<42> a_42b_var(-1424692392255LL); // long long decimal format
a_42b_var = 0x14BB648B13FLL; // hexadecimal format

a_42b_var = -1; // negative int literal sign-extended to full width

ap_uint<96> wide_var("76543210fedcba9876543210", 16); // Greater than 64-bit
wide_var = ap_int<96>("0123456789abcdef01234567", 16);
```

Note: To avoid unexpected behavior during co-simulation, do not initialize `ap_uint<N> a = {0}`.

The `ap_[u]<>` constructor may be explicitly instructed to interpret the string as representing the number in radix 2, 8, 10, or 16 formats. This is accomplished by adding the appropriate radix value as a second parameter to the constructor call.

A compilation error occurs if the string literal contains any characters that are invalid as digits for the radix specified.

The following examples use different radix formats:

```
ap_int<6> a_6bit_var("101010", 2); // 42d in binary format
a_6bit_var = ap_int<6>("40", 8); // 32d in octal format
a_6bit_var = ap_int<6>("55", 10); // decimal format
a_6bit_var = ap_int<6>("2A", 16); // 42d in hexadecimal format

a_6bit_var = ap_int<6>("42", 2); // COMPILE-TIME ERROR! "42" is not binary
```

The radix of the number encoded in the string can also be inferred by the constructor, when it is prefixed with a zero (0) followed by one of the following characters: "b", "o" or "x". The prefixes "0b", "0o" and "0x" correspond to binary, octal and hexadecimal formats respectively.

The following examples use alternate initializer string formats:

```
ap_int<6> a_6bit_var("0b101010", 2); // 42d in binary format
a_6bit_var = ap_int<6>("0o40", 8); // 32d in octal format
a_6bit_var = ap_int<6>("0x2A", 16); // 42d in hexidecimal format

a_6bit_var = ap_int<6>("0b42", 2); // COMPILE-TIME ERROR! "42" is not binary
```

If the bit-width is greater than 53-bits, the `ap_[u]int<>` value must be initialized with a string, for example:

```
ap_uint<72> Val("2460508560057040035.375");
```

Support for Console I/O (Printing)

As with initialization and assignment to `ap_[u]fixed<>` variables, Vitis HLS supports printing values that require more than 64-bits to represent.

Using the C++ Standard Output Stream

The easiest way to output any value stored in an `ap_[u]int` variable is to use the C++ standard output stream:

```
std::cout (#include <iostream> or <iostream.h>)
```

The stream insertion operator (`<<`) is overloaded to correctly output the full range of values possible for any given `ap_[u]fixed` variable. The following stream manipulators are also supported:

- `dec` (decimal)
- `hex` (hexadecimal)
- `oct` (octal)

These allow formatting of the value as indicated.

The following example uses `cout` to print values:

```
#include <iostream.h>
// Alternative: #include <iostream>

ap_ufixed<72> Val("10fedcba9876543210");

cout << Val << endl; // Yields: "313512663723845890576"
cout << hex << val << endl; // Yields: "10fedcba9876543210"
cout << oct << val << endl; // Yields: "41773345651416625031020"
```

Using the Standard C Library

You can also use the standard C library (`#include <stdio.h>`) to print out values larger than 64-bits:

1. Convert the value to a C++ `std::string` using the `ap_[u]fixed` classes method `to_string()`.
2. Convert the result to a null-terminated C character string using the `std::string` class method `c_str()`.

Optional Argument One (Specifying the Radix)

You can pass the `ap[u]int::to_string()` method an optional argument specifying the radix of the numerical format desired. The valid radix argument values are:

- 2 (binary) (default)
- 8 (octal)
- 10 (decimal)
- 16 (hexadecimal)

Optional Argument Two (Printing as Signed Values)

A second optional argument to `ap_[u]int::to_string()` specifies whether to print the non-decimal formats as signed values. This argument is boolean. The default value is false, causing the non-decimal formats to be printed as unsigned values.

The following examples use `printf` to print values:

```
ap_int<72> Val("80fedcba9876543210");

printf("%s\n", Val.to_string().c_str()); // => "80FEDCBA9876543210"
printf("%s\n", Val.to_string(10).c_str()); // => "-2342818482890329542128"
printf("%s\n", Val.to_string(8).c_str()); // => "401773345651416625031020"
printf("%s\n", Val.to_string(16, true).c_str()); // => "-7F0123456789ABCDF0"
```

Expressions Involving `ap_[u]>` types

Variables of `ap_[u]>` types may generally be used freely in expressions involving C/C++ operators. Some behaviors may be unexpected. These are discussed in detail below.

Zero- and Sign-Extension on Assignment From Narrower to Wider Variables

When assigning the value of a narrower bit-width signed (`ap_int<>`) variable to a wider one, the value is sign-extended to the width of the destination variable, regardless of its signedness.

Similarly, an unsigned source variable is zero-extended before assignment.

Explicit casting of the source variable may be necessary to ensure expected behavior on assignment. See the following example:

```
ap_uint<10> Result;

ap_int<7> Val1 = 0x7f;
ap_uint<6> Val2 = 0x3f;

Result = Val1; // Yields: 0x3ff (sign-extended)
Result = Val2; // Yields: 0x03f (zero-padded)

Result = ap_uint<7>(Val1); // Yields: 0x07f (zero-padded)
Result = ap_int<6>(Val2); // Yields: 0x3ff (sign-extended)
```

Truncation on Assignment of Wider to Narrower Variables

Assigning the value of a wider source variable to a narrower one leads to truncation of the value. All bits beyond the most significant bit (MSB) position of the destination variable are lost.

There is no special handling of the sign information during truncation. This may lead to unexpected behavior. Explicit casting may help avoid this unexpected behavior.

Class Methods and Operators

The `ap_[u]int` types do not support implicit conversion from wide `ap_[u]int (>64bits)` to builtin C/C++ integer types. For example, the following code example return `s1`, because the implicit cast from `ap_int[65]` to `bool` in the if-statement returns a 0.

```
bool nonzero(ap_uint<65> data) {
    return data; // This leads to implicit truncation to 64b int
}

int main() {
    if (nonzero((ap_uint<65>)1 << 64)) {
        return 0;
    }
    printf(FAIL\n);
    return 1;
}
```

To convert wide `ap_[u]int` types to built-in integers, use the explicit conversion functions included with the `ap_[u]int` types:

- `to_int()`
- `to_long()`
- `to_bool()`

In general, any valid operation that can be done on a native C/C++ integer data type is supported using operator overloading for `ap_[u]int` types.

In addition to these overloaded operators, some class specific operators and methods are included to ease bit-level operations.

Binary Arithmetic Operators

Standard binary integer arithmetic operators are overloaded to provide arbitrary precision arithmetic. These operators take either:

- Two operands of `ap_[u]int`, or
- One `ap_[u]int` type and one C/C++ fundamental integer data type

For example:

- `char`

- short
- int

The width and signedness of the resulting value is determined by the width and signedness of the operands, before sign-extension, zero-padding or truncation are applied based on the width of the destination variable (or expression). Details of the return value are described for each operator.

When expressions contain a mix of `ap_[u]int` and C/C++ fundamental integer types, the C++ types assume the following widths:

- char (8-bits)
- short (16-bits)
- int (32-bits)
- long (32-bits)
- long long (64-bits)

Addition

```
ap_(u)int::RType ap_(u)int::operator + (ap_(u)int op)
```

Returns the sum of:

- Two `ap_[u]int`, or
- One `ap_[u]int` and a C/C++ integer type

The width of the sum value is:

- One bit more than the wider of the two operands, or
- Two bits if and only if the wider is unsigned and the narrower is signed

The sum is treated as signed if either (or both) of the operands is of a signed type.

Subtraction

```
ap_(u)int::RType ap_(u)int::operator - (ap_(u)int op)
```

Returns the difference of two integers.

The width of the difference value is:

- One bit more than the wider of the two operands, or
- Two bits if and only if the wider is unsigned and the narrower signed

This is true before assignment, at which point it is sign-extended, zero-padded, or truncated based on the width of the destination variable.

The difference is treated as signed regardless of the signedness of the operands.

Multiplication

```
ap_(u)int::RType ap_(u)int::operator * (ap_(u)int op)
```

Returns the product of two integer values.

The width of the product is the sum of the widths of the operands.

The product is treated as a signed type if either of the operands is of a signed type.

Division

```
ap_(u)int::RType ap_(u)int::operator / (ap_(u)int op)
```

Returns the quotient of two integer values.

The width of the quotient is the width of the dividend if the divisor is an unsigned type. Otherwise, it is the width of the dividend plus one.

The quotient is treated as a signed type if either of the operands is of a signed type.

Modulus

```
ap_(u)int::RType ap_(u)int::operator % (ap_(u)int op)
```

Returns the modulus, or remainder of integer division, for two integer values.

The width of the modulus is the minimum of the widths of the operands, if they are both of the same signedness.

If the divisor is an unsigned type and the dividend is signed, then the width is that of the divisor plus one.

The quotient is treated as having the same signedness as the dividend.



IMPORTANT! Vitis HLS synthesis of the modulus (%) operator will lead to instantiation of appropriately parameterized Xilinx LogiCORE divider cores in the generated RTL.

Following are examples of arithmetic operators:

```
ap_uint<71> Rslt;  
  
ap_uint<42> Val1 = 5;  
ap_int<23> Val2 = -8;
```

```
Rslt = Val1 + Val2; // Yields: -3 (43 bits) sign-extended to 71 bits
Rslt = Val1 - Val2; // Yields: +3 sign extended to 71 bits
Rslt = Val1 * Val2; // Yields: -40 (65 bits) sign extended to 71 bits
Rslt = 50 / Val2; // Yields: -6 (33 bits) sign extended to 71 bits
Rslt = 50 % Val2; // Yields: +2 (23 bits) sign extended to 71 bits
```

Bitwise Logical Operators

The bitwise logical operators all return a value with a width that is the maximum of the widths of the two operands. It is treated as unsigned if and only if both operands are unsigned. Otherwise, it is of a signed type.

Sign-extension (or zero-padding) may occur, based on the signedness of the expression, not the destination variable.

Bitwise OR

```
ap_(u)int::RType ap_(u)int::operator | (ap_(u)int op)
```

Returns the bitwise OR of the two operands.

Bitwise AND

```
ap_(u)int::RType ap_(u)int::operator & (ap_(u)int op)
```

Returns the bitwise AND of the two operands.

Bitwise XOR

```
ap_(u)int::RType ap_(u)int::operator ^ (ap_(u)int op)
```

Returns the bitwise XOR of the two operands.

Unary Operators

Addition

```
ap_(u)int ap_(u)int::operator + ()
```

Returns the self copy of the `ap_[u]int` operand.

Subtraction

```
ap_(u)int::RType ap_(u)int::operator - ()
```

Returns the following:

- The negated value of the operand with the same width if it is a signed type, or
- Its width plus one if it is unsigned.

The return value is always a signed type.

Bitwise Inverse

```
ap_(u)int::RType ap_(u)int::operator ~ ()
```

Returns the bitwise-NOT of the operand with the same width and signedness.

Logical Invert

```
bool ap_(u)int::operator ! ()
```

Returns a Boolean `false` value if and only if the operand is *not* equal to zero (0).

Returns a Boolean `true` value if the operand is equal to zero (0).

Ternary Operators

When you use the ternary operator with the standard C `int` type, you must explicitly cast from one type to the other to ensure that both results have the same type. For example:

```
// Integer type is cast to ap_int type
ap_int<32> testc3(int a, ap_int<32> b, ap_int<32> c, bool d) {
    return d?ap_int<32>(a):b;
}
// ap_int type is cast to an integer type
ap_int<32> testc4(int a, ap_int<32> b, ap_int<32> c, bool d) {
    return d?a+1:(int)b;
}
// Integer type is cast to ap_int type
ap_int<32> testc5(int a, ap_int<32> b, ap_int<32> c, bool d) {
    return d?ap_int<32>(a):b+1;
}
```

Shift Operators

Each shift operator comes in two versions:

- One version for *unsigned* right-hand side (RHS) operands
- One version for *signed* right-hand side (RHS) operands

A negative value supplied to the signed RHS versions reverses the shift operations direction. That is, a shift by the absolute value of the RHS operand in the opposite direction occurs.

The shift operators return a value with the same width as the left-hand side (LHS) operand. As with C/C++, if the LHS operand of a shift-right is a signed type, the sign bit is copied into the most significant bit positions, maintaining the sign of the LHS operand.

Unsigned Integer Shift Right

```
ap_(u)int ap_(u)int::operator >> (ap_uint<int_W2> op)
```

Integer Shift Right

```
ap_(u)int ap_(u)int::operator >> (ap_int<int_W2> op)
```

Unsigned Integer Shift Left

```
ap_(u)int ap_(u)int::operator << (ap_uint<int_W2> op)
```

Integer Shift Left

```
ap_(u)int ap_(u)int::operator << (ap_int<int_W2> op)
```



CAUTION! When assigning the result of a shift-left operator to a wider destination variable, some or all information may be lost. Xilinx recommends that you explicitly cast the shift expression to the destination type to avoid unexpected behavior.

Following are examples of shift operations:

```
ap_uint<13> Rslt;  
  
ap_uint<7> Val1 = 0x41;  
  
Rslt = Val1 << 6; // Yields: 0x0040, i.e. msb of Val1 is lost  
Rslt = ap_uint<13>(Val1) << 6; // Yields: 0x1040, no info lost  
  
ap_int<7> Val2 = -63;  
Rslt = Val2 >> 4; //Yields: 0x1ffc, sign is maintained and extended
```

Compound Assignment Operators

Vitis HLS supports compound assignment operators:

```
*=  
/=  
%=  
+=  
-=  
<<=  
>>=  
&=  
^=  
|=
```

The RHS expression is first evaluated then supplied as the RHS operand to the base operator, the result of which is assigned back to the LHS variable. The expression sizing, signedness, and potential sign-extension or truncation rules apply as discussed above for the relevant operations.

```
ap_uint<10> Val1 = 630;
ap_int<3> Val2 = -3;
ap_uint<5> Val3 = 27;

Val1 += Val2 - Val3; // Yields: 600 and is equivalent to:

// Val1 = ap_uint<10>(ap_int<11>(Val1) +
// ap_int<11>((ap_int<6>(Val2) -
// ap_int<6>(Val3))));
```

Increment and Decrement Operators

The increment and decrement operators are provided. All return a value of the same width as the operand and which is unsigned if and only if both operands are of unsigned types and signed otherwise.

Pre-Increment

```
ap_(u)int& ap_(u)int::operator ++ ()
```

Returns the incremented value of the operand.

Assigns the incremented value to the operand.

Post-Increment

```
const ap_(u)int ap_(u)int::operator ++ (int)
```

Returns the value of the operand before assignment of the incremented value to the operand variable.

Pre-Decrement

```
ap_(u)int& ap_(u)int::operator -- ()
```

Returns the decremented value of, as well as assigning the decremented value to, the operand.

Post-Decrement

```
const ap_(u)int ap_(u)int::operator -- (int)
```

Returns the value of the operand before assignment of the decremented value to the operand variable.

Relational Operators

Vitis HLS supports all relational operators. They return a Boolean value based on the result of the comparison. You can compare variables of `ap_[u]int` types to C/C++ fundamental integer types with these operators.

Equality

```
bool ap_(u)int::operator == (ap_(u)int op)
```

Inequality

```
bool ap_(u)int::operator != (ap_(u)int op)
```

Less than

```
bool ap_(u)int::operator < (ap_(u)int op)
```

Greater than

```
bool ap_(u)int::operator > (ap_(u)int op)
```

Less than or equal to

```
bool ap_(u)int::operator <= (ap_(u)int op)
```

Greater than or equal to

```
bool ap_(u)int::operator >= (ap_(u)int op)
```

Other Class Methods, Operators, and Data Members

The following sections discuss other class methods, operators, and data members.

Bit-Level Operations

The following methods facilitate common bit-level operations on the value stored in `ap_[u]int` type variables.

Length

```
int ap_(u)int::length ()
```

Returns an integer value providing the total number of bits in the `ap_[u]int` variable.

Concatenation

```
ap_concat_ref ap_(u)int::concat (ap_(u)int low)
ap_concat_ref ap_(u)int::operator , (ap_(u)int high, ap_(u)int low)
```

Concatenates two `ap_[u]int` variables, the width of the returned value is the sum of the widths of the operands.

The High and Low arguments are placed in the higher and lower order bits of the result respectively; the `concat()` method places the argument in the lower order bits.

When using the overloaded comma operator, the parentheses are required. The comma operator version may also appear on the LHS of assignment.



RECOMMENDED: To avoid unexpected results, explicitly cast C/C++ native types (including integer literals) to an appropriate `ap_[u]int` type before concatenating.

```
ap_uint<10> Rslt;
ap_int<3> Val1 = -3;
ap_int<7> Val2 = 54;

Rslt = (Val2, Val1); // Yields: 0x1B5
Rslt = Val1.concat(Val2); // Yields: 0x2B6
(Val1, Val2) = 0xAB; // Yields: Val1 == 1, Val2 == 43
```

Bit Selection

```
ap_bit_ref ap_(u)int::operator [] (int bit)
```

Selects one bit from an arbitrary precision integer value and returns it.

The returned value is a reference value that can set or clear the corresponding bit in this `ap_[u]int`.

The bit argument must be an `int` value. It specifies the index of the bit to select. The least significant bit has index 0. The highest permissible index is one less than the bit-width of this `ap_[u]int`.

The result type `ap_bit_ref` represents the reference to one bit of this `ap_[u]int` instance specified by bit.

Range Selection

```
ap_range_ref ap_(u)int::range (unsigned Hi, unsigned Lo)
ap_range_ref ap_(u)int::operator () (unsigned Hi, unsigned Lo)
```

Returns the value represented by the range of bits specified by the arguments.

The `Hi` argument specifies the most significant bit (MSB) position of the range, and `Lo` specifies the least significant bit (LSB).

The LSB of the source variable is in position 0. If the `Hi` argument has a value less than `Lo`, the bits are returned in reverse order.

```
ap_uint<4> Rslt;  
  
ap_uint<8> Val1 = 0x5f;  
ap_uint<8> Val2 = 0xaa;  
  
Rslt = Val1.range(3, 0); // Yields: 0xF  
Val1(3,0) = Val2(3, 0); // Yields: 0x5A  
Val1(3,0) = Val2(4, 1); // Yields: 0x55  
Rslt = Val1.range(4, 7); // Yields: 0xA; bit-reversed!
```

Note: The object returned by range select is not an `ap_(u)int` object and lacks operators, but can be used for assignment. To use the range select result in a chained expression with `ap_(u)int` methods, add an explicit constructor like below.

```
ap_uint<32> v = 0x8fff0000;  
bool r = ap_uint<16>(v.range(23, 8)).xor_reduce();
```

AND reduce

```
bool ap_(u)int::and_reduce()
```

- Applies the AND operation on all bits in this `ap_(u)int`.
- Returns the resulting single bit.
- Equivalent to comparing this value against -1 (all ones) and returning `true` if it matches, `false` otherwise.

OR reduce

```
bool ap_(u)int::or_reduce()
```

- Applies the OR operation on all bits in this `ap_(u)int`.
- Returns the resulting single bit.
- Equivalent to comparing this value against 0 (all zeros) and returning `false` if it matches, `true` otherwise.

XOR reduce

```
bool ap_(u)int::xor_reduce()
```

- Applies the XOR operation on all bits in this `ap_int`.
- Returns the resulting single bit.
- Equivalent to counting the number of 1 bits in this value and returning `false` if the count is even or `true` if the count is odd.

NAND reduce

```
bool ap_(u)int::nand_reduce ()
```

- Applies the NAND operation on all bits in this `ap_int`.
- Returns the resulting single bit.
- Equivalent to comparing this value against -1 (all ones) and returning `false` if it matches, `true` otherwise.

NOR reduce

```
bool ap_int::nor_reduce ()
```

- Applies the NOR operation on all bits in this `ap_int`.
- Returns the resulting single bit.
- Equivalent to comparing this value against 0 (all zeros) and returning `true` if it matches, `false` otherwise.

XNOR reduce

```
bool ap_(u)int::xnor_reduce ()
```

- Applies the XNOR operation on all bits in this `ap_(u)int`.
- Returns the resulting single bit.
- Equivalent to counting the number of 1 bits in this value and returning `true` if the count is even or `false` if the count is odd.

Bit Reduction Method Examples

```
ap_uint<8> Val = 0xaa;  
  
bool t = Val.and_reduce(); // Yields: false  
t = Val.or_reduce(); // Yields: true  
t = Val.xor_reduce(); // Yields: false  
t = Val.nand_reduce(); // Yields: true  
t = Val.nor_reduce(); // Yields: false  
t = Val.xnor_reduce(); // Yields: true
```

Bit Reverse

```
void ap_(u)int::reverse ()
```

Reverses the contents of `ap_[u]int` instance:

- The LSB becomes the MSB.
- The MSB becomes the LSB.

Reverse Method Example

```
ap_uint<8> Val = 0x12;  
Val.reverse(); // Yields: 0x48
```

Test Bit Value

```
bool ap_(u)int::test (unsigned i)
```

Checks whether specified bit of `ap_(u)int` instance is 1.

Returns true if Yes, false if No.

Test Method Example

```
ap_uint<8> Val = 0x12;  
bool t = Val.test(5); // Yields: true
```

Set Bit Value

```
void ap_(u)int::set (unsigned i, bool v)  
void ap_(u)int::set_bit (unsigned i, bool v)
```

Sets the specified bit of the `ap_(u)int` instance to the value of integer `v`.

Set Bit (to 1)

```
void ap_(u)int::set (unsigned i)
```

Sets the specified bit of the `ap_(u)int` instance to the value 1 (one).

Clear Bit (to 0)

```
void ap_(u)int:: clear(unsigned i)
```

Sets the specified bit of the `ap_(u)int` instance to the value 0 (zero).

Invert Bit

```
void ap_(u)int:: invert(unsigned i)
```

Inverts the bit specified in the function argument of the `ap_(u)int` instance. The specified bit becomes 0 if its original value is 1 and vice versa.

Example of bit set, clear and invert bit methods:

```
ap_uint<8> Val = 0x12;
Val.set(0, 1); // Yields: 0x13
Val.set_bit(4, false); // Yields: 0x03
Val.set(7); // Yields: 0x83
Val.clear(1); // Yields: 0x81
Val.invert(4); // Yields: 0x91
```

Rotate Right

```
void ap_(u)int:: rrotate(unsigned n)
```

Rotates the `ap_(u)int` instance `n` places to right.

Rotate Left

```
void ap_(u)int:: lrotate(unsigned n)
```

Rotates the `ap_(u)int` instance `n` places to left.

```
ap_uint<8> Val = 0x12;

Val.rrotate(3); // Yields: 0x42
Val.lrotate(6); // Yields: 0x90
```

Bitwise NOT

```
void ap_(u)int:: b_not()
```

- Complements every bit of the `ap_(u)int` instance.

```
ap_uint<8> Val = 0x12;

Val.b_not(); // Yields: 0xED
```

Bitwise NOT Example

Test Sign

```
bool ap_int:: sign()
```

- Checks whether the `ap_(u)int` instance is negative.
- Returns `true` if negative.
- Returns `false` if positive.

Explicit Conversion Methods

To C/C++ “(u)int”

```
int ap_(u)int::to_int ()  
unsigned ap_(u)int::to_uint ()
```

- Returns native C/C++ (32-bit on most systems) integers with the value contained in the `ap_[u]int`.
- Truncation occurs if the value is greater than can be represented by an `[unsigned]` `int`.

To C/C++ 64-bit “(u)int”

```
long long ap_(u)int::to_int64 ()  
unsigned long long ap_(u)int::to_uint64 ()
```

- Returns native C/C++ 64-bit integers with the value contained in the `ap_[u]int`.
- Truncation occurs if the value is greater than can be represented by an `[unsigned]` `int`.

To C/C++ “double”

```
double ap_(u)int::to_double ()
```

- Returns a native C/C++ `double` 64-bit floating point representation of the value contained in the `ap_[u]int`.
- If the `ap_[u]int` is wider than 53 bits (the number of bits in the mantissa of a `double`), the resulting `double` may not have the exact value expected.



RECOMMENDED: Xilinx recommends that you explicitly call member functions instead of using C-style cast to convert `ap_[u]int` to other data types.

Sizeof

The standard C++ `sizeof()` function should not be used with `ap_[u]int` or other classes or instance of object. The `ap_int<>` data type is a class and `sizeof` returns the storage used by that class or instance object. `sizeof(ap_int<N>)` always returns the number of bytes used. For example:

```
sizeof(ap_int<127>)=16  
sizeof(ap_int<128>)=16  
sizeof(ap_int<129>)=24  
sizeof(ap_int<130>)=24
```

Compile Time Access to Data Type Attributes

The `ap_[u]int<>` types are provided with a static member that allows the size of the variables to be determined at compile time. The data type is provided with the static const member `width`, which is automatically assigned the width of the data type:

```
static const int width = _AP_W;
```

You can use the `width` data member to extract the data width of an existing `ap_[u]int<>` data type to create another `ap_[u]int<>` data type at compile time. The following example shows how the size of variable `Res` is defined as 1-bit greater than variables `Val1` and `Val2`:

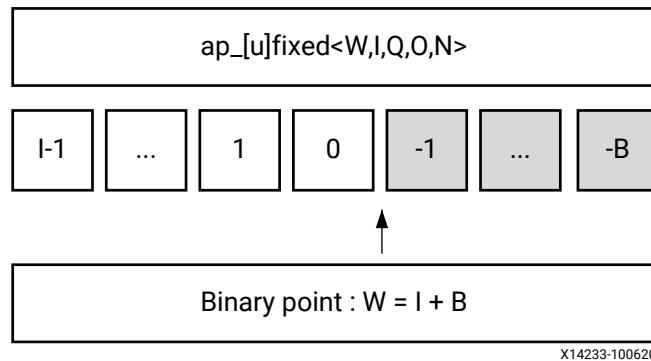
```
// Definition of basic data type
#define INPUT_DATA_WIDTH 8
typedef ap_int<INPUT_DATA_WIDTH> data_t;
// Definition of variables
data_t Val1, Val2;
// Res is automatically sized at compile-time to be 1-bit greater than data
// type
data_t
ap_int<data_t::width+1> Res = Val1 + Val2;
```

This ensures that Vitis HLS correctly models the bit-growth caused by the addition even if you update the value of `INPUT_DATA_WIDTH` for `data_t`.

C++ Arbitrary Precision Fixed-Point Types

C++ functions can take advantage of the arbitrary precision fixed-point types included with Vitis HLS. The following figure summarizes the basic features of these fixed-point types:

- The word can be signed (`ap_fixed`) or unsigned (`ap_ufixed`).
- A word with of any arbitrary size `W` can be defined.
- The number of places above the decimal point `I`, also defines the number of decimal places in the word, `W-I` (represented by `B` in the following figure).
- The type of rounding or quantization (`Q`) can be selected.
- The overflow behavior (`O` and `N`) can be selected.

Figure 132: Arbitrary Precision Fixed-Point Types

TIP: The arbitrary precision fixed-point types can be used when header file `ap_fixed.h` is included in the code.

Arbitrary precision fixed-point types use more memory during C simulation and if you are using very large arrays of `ap_[u]fixed` types.

The advantages of using fixed-point types are:

- They allow fractional number to be easily represented.
- When variables have a different number of integer and decimal place bits, the alignment of the decimal point is handled.
- There are numerous options to handle how rounding should happen: when there are too few decimal bits to represent the precision of the result.
- There are numerous options to handle how variables should overflow: when the result is greater than the number of integer bits can represent.

These attributes are summarized by examining the code in the example below. First, the header file `ap_fixed.h` is included. The `ap_fixed` types are then defined using the `typedef` statement:

- A 10-bit input: 8-bit integer value with 2 decimal places.
- A 6-bit input: 3-bit integer value with 3 decimal places.
- A 22-bit variable for the accumulation: 17-bit integer value with 5 decimal places.
- A 36-bit variable for the result: 30-bit integer value with 6 decimal places.

The function contains no code to manage the alignment of the decimal point after operations are performed. The alignment is done automatically.

The following code sample shows ap_fixed type.

```
#include "ap_fixed.h"

typedef ap_ufixed<10,8, AP_RND, AP_SAT> din1_t;
typedef ap_fixed<6,3, AP_RND, AP_WRAP> din2_t;
typedef ap_fixed<22,17, AP_TRN, AP_SAT> dint_t;
typedef ap_fixed<36,30> dout_t;

dout_t cpp_ap_fixed(din1_t d_in1, din2_t d_in2) {
    static dint_t sum;
    sum += d_in1;
    return sum * d_in2;
}
```

Using ap_(u)fixed types, the C++ simulation is bit accurate. Fast simulation can validate the algorithm and its accuracy. After synthesis, the RTL exhibits the identical bit-accurate behavior.

Arbitrary precision fixed-point types can be freely assigned literal values in the code. This is shown in the test bench (see the example below) used with the example above, in which the values of `in1` and `in2` are declared and assigned constant values.

When assigning literal values involving operators, the literal values must first be cast to ap_(u)fixed types. Otherwise, the C compiler and Vitis HLS interpret the literal as an integer or float/double type and may fail to find a suitable operator. As shown in the following example, in the assignment of `in1 = in1 + din1_t(0.25)`, the literal 0.25 is cast to an ap_fixed type.

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;
#include "ap_fixed.h"

typedef ap_ufixed<10,8, AP_RND, AP_SAT> din1_t;
typedef ap_fixed<6,3, AP_RND, AP_WRAP> din2_t;
typedef ap_fixed<22,17, AP_TRN, AP_SAT> dint_t;
typedef ap_fixed<36,30> dout_t;

dout_t cpp_ap_fixed(din1_t d_in1, din2_t d_in2);
int main()
{
    ofstream result;
    din1_t in1 = 0.25;
    din2_t in2 = 2.125;
    dout_t output;
    int retval=0;

    result.open(result.dat);
    // Persistent manipulators
    result << right << fixed << setbase(10) << setprecision(15);

    for (int i = 0; i <= 250; i++)
    {
```

```
output = cpp_ap_fixed(in1,in2);

result << setw(10) << i;
result << setw(20) << in1;
result << setw(20) << in2;
result << setw(20) << output;
result << endl;

in1 = in1 + din1_t(0.25);
in2 = in2 - din2_t(0.125);
}
result.close();

// Compare the results file with the golden results
retval = system(diff --brief -w result.dat result.golden.dat);
if (retval != 0) {
printf(Test failed !!!\n);
retval=1;
} else {
printf(Test passed !\n);
}

// Return 0 if the test passes
return retval;
}
```

Fixed-Point Identifier Summary

The following table shows the quantization and overflow modes.



TIP: Quantization and overflow modes that do more than the default behavior of standard hardware arithmetic (wrap and truncate) result in operators with more associated hardware. It costs logic (LUTs) to implement the more advanced modes, such as round to minus infinity or saturate symmetrically.

Table 42: Fixed-Point Identifier Summary

Identifier	Description
W	Word length in bits
I	The number of bits used to represent the integer value, that is, the number of integer bits to the <i>left</i> of the binary point. When this value is negative, it represents the number of <i>implicit</i> sign bits (for signed representation), or the number of <i>implicit</i> zero bits (for unsigned representation) to the <i>right</i> of the binary point. For example, <pre>ap_fixed<2, 0> a = -0.5; // a can be -0.5, ap_ufixed<1, 0> x = 0.5; // 1-bit representation. x can be 0 or 0.5 ap_ufixed<1, -1> y = 0.25; // 1-bit representation. y can be 0 or 0.25 const ap_fixed<1, -7> z = 1.0/256; // 1-bit representation for z = 2^-8</pre>

Table 42: Fixed-Point Identifier Summary (cont'd)

Identifier	Description	
Q	Quantization mode: This dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result.	
	ap_fixed Types	Description
	AP_RND	Round to plus infinity
	AP_RND_ZERO	Round to zero
	AP_RND_MIN_INF	Round to minus infinity
	AP_RND_INF	Round to infinity
	AP_RND_CONV	Convergent rounding
	AP_TRN	Truncation to minus infinity (default)
	AP_TRN_ZERO	Truncation to zero
O	Overflow mode: This dictates the behavior when the result of an operation exceeds the maximum (or minimum in the case of negative numbers) possible value that can be stored in the variable used to store the result.	
	ap_fixed Types	Description
	AP_SAT ¹	Saturation
	AP_SAT_ZERO ¹	Saturation to zero
	AP_SAT_SYM ¹	Symmetrical saturation
	AP_WRAP	Wrap around (default)
	AP_WRAP_SM	Sign magnitude wrap around
N	This defines the number of saturation bits in overflow wrap modes.	

Notes:

- Using the AP_SAT* modes can result in higher resource usage as extra logic will be needed to perform saturation and this extra cost can be as high as 20% additional LUT usage.
- Fixed-point math functions from the `hls_math` library do not support the `ap_[u]fixed` template parameters Q,O, and N, for quantization mode, overflow mode, and the number of saturation bits, respectively. The quantization and overflow modes are only effective when an `ap_[u]fixed` variable is on the left hand of assignment or being initialized, but not during the calculation.

C++ Arbitrary Precision Fixed-Point Types: Reference Information

For comprehensive information on the methods, synthesis behavior, and all aspects of using the `ap_(u)fixed<N>` arbitrary precision fixed-point data types, see [C++ Arbitrary Precision Fixed-Point Types](#). This section includes:

- Techniques for assigning constant and initialization values to arbitrary precision integers (including values greater than 1024-bit).
- A detailed description of the overflow and saturation modes.
- A description of Vitis HLS helper methods, such as printing, concatenating, bit-slicing and range selection functions.

- A description of operator behavior, including a description of shift operations (a negative shift value, results in a shift in the opposite direction).



IMPORTANT! For the compiler to process, you must use the appropriate header files for the language.

C++ Arbitrary Precision Fixed-Point Types

Vitis HLS supports fixed-point types that allow fractional arithmetic to be easily handled. The advantage of fixed-point arithmetic is shown in the following example.

```
ap_fixed<11, 6> Var1 = 22.96875; // 11-bit signed word, 5 fractional bits
ap_ufixed<12,11> Var2 = 512.5; // 12-bit word, 1 fractional bit
ap_fixed<16,11> Res1; // 16-bit signed word, 5 fractional bits

Res1 = Var1 + Var2; // Result is 535.46875
```

Even though `Var1` and `Var2` have different precisions, the fixed-point type ensures that the decimal point is correctly aligned before the operation (an addition in this case), is performed. You are not required to perform any operations in the C code to align the decimal point.

The type used to store the result of any fixed-point arithmetic operation must be large enough (in both the integer and fractional bits) to store the full result.

If this is not the case, the `ap_fixed` type performs:

- overflow handling (when the result has more MSBs than the assigned type supports)
- quantization (or rounding, when the result has fewer LSBs than the assigned type supports)

The `ap_[u]fixed` type provides various options on how the overflow and quantization are performed. The options are discussed below.

ap_[u]fixed Representation

In `ap[u]fixed` types, a fixed-point value is represented as a sequence of bits with a specified position for the binary point.

- Bits to the left of the binary point represent the integer part of the value.
- Bits to the right of the binary point represent the fractional part of the value.

`ap_[u]fixed` type is defined as follows:

```
ap_[u]fixed<int W,
          int I,
          ap_q_mode Q,
          ap_o_mode O,
          ap_sat_bits N>;
```

Quantization Modes

Rounding to plus infinity	AP_RND
Rounding to zero	AP_RND_ZERO
Rounding to minus infinity	AP_RND_MIN_INF
Rounding to infinity	AP_RND_INF
Convergent rounding	AP_RND_CONV
Truncation	AP_TRN
Truncation to zero	AP_TRN_ZERO

AP_RND

- Round the value to the nearest representable value for the specific `ap_[u]fixed` type.

```
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.5
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.0
```

AP_RND_ZERO

- Round the value to the nearest representable value.
- Round towards zero.
 - For positive values, delete the redundant bits.
 - For negative values, add the least significant bits to get the nearest representable value.

```
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.0
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.0
```

AP_RND_MIN_INF

- Round the value to the nearest representable value.
- Round towards minus infinity.
 - For positive values, delete the redundant bits.
 - For negative values, add the least significant bits.

```
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.0
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.5
```

AP_RND_INF

- Round the value to the nearest representable value.
- The rounding depends on the least significant bit.
 - For positive values, if the least significant bit is set, round towards plus infinity. Otherwise, round towards minus infinity.

- For negative values, if the least significant bit is set, round towards minus infinity.
Otherwise, round towards plus infinity.

```
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.5
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.5
```

AP_RND_CONV

- Round to the nearest representable value with "ties" rounding to even, that is, the least significant bit (after rounding) is forced to zero.
- A "tie" is the midpoint of two representable values and occurs when the bit following the least significant bit (after rounding) is 1 and all the bits below it are zero.

```
// For the following examples, bit3 of the 8-bit value becomes the
// LSB of the final 5-bit value (after rounding).
// Notes:
//   * bit7 of the 8-bit value is the MSB (sign bit)
//   * the 3 LSBs of the 8-bit value (bit2, bit1, bit0) are treated as
//     guard, round and sticky bits.
//   * See http://pages.cs.wisc.edu/~david/courses/cs552/S12/handouts/
//     guardbits.pdf

ap_fixed<8,3> p1 = 1.59375; // p1 = 001.10011
ap_fixed<5,3,AP_RND_CONV> rconv1 = p1; // rconv1 = 1.5 (001.10)

ap_fixed<8,3> p2 = 1.625; // p2 = 001.10100 => tie with bit3 (LSB-to-be)
= 0
ap_fixed<5,3,AP_RND_CONV> rconv2 = p2; // rconv2 = 1.5 (001.10) => lsb is
already zero, just truncate

ap_fixed<8,3> p3 = 1.375; // p3 = 001.01100 => tie with bit3 (LSB-to-be)
= 1
ap_fixed<5,3,AP_RND_CONV> rconv3 = p3; // rconv3 = 1.5 (001.10) => lsb is
made zero by rounding up

ap_fixed<8,3> p3 = 1.65625; // p3 = 001.10101
ap_fixed<5,3,AP_RND_CONV> rconv3 = p3; // rconv3 = 1.75 (001.11) => round
up
```

AP_TRN

- Always round the value towards minus infinity.

```
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.0
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.5
```

AP_TRN_ZERO

Round the value to:

- For positive values, the rounding is the same as mode AP_TRN.
- For negative values, round towards zero.

```
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.0
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.0
```

Overflow Modes

Saturation	AP_SAT
Saturation to zero	AP_SAT_ZERO
Symmetrical saturation	AP_SAT_SYM
Wrap-around	AP_WRAP
Sign magnitude wrap-around	AP_WRAP_SM

AP_SAT

Saturate the value.

- To the maximum value in case of overflow.
- To the negative maximum value in case of negative overflow.

```
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0; // Yields: 7.0
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0; // Yields: -8.0
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0; // Yields: 15.0
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0; // Yields: 0.0
```

AP_SAT_ZERO

Force the value to zero in case of overflow, or negative overflow.

```
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0; // Yields: 0.0
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0; // Yields: 0.0
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0; // Yields: 0.0
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0; // Yields: 0.0
```

AP_SAT_SYM

Saturate the value:

- To the maximum value in case of overflow.
- To the minimum value in case of negative overflow.
 - Negative maximum for signed `ap_fixed` types
 - Zero for unsigned `ap_ufixed` types

```
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0; // Yields: 7.0
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0; // Yields: -7.0
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0; // Yields: 15.0
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0; // Yields: 0.0
```

AP_WRAP

Wrap the value around in case of overflow.

```
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 31.0; // Yields: -1.0
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; // Yields: -3.0
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 19.0; // Yields: 3.0
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; // Yields: 13.0
```

If the value of N is set to zero (the default overflow mode):

- All MSB bits outside the range are deleted.
- For unsigned numbers. After the maximum it wraps around to zero.
- For signed numbers. After the maximum, it wraps to the minimum values.

If N>0:

- When N > 0, N MSB bits are saturated or set to 1.
- The sign bit is retained, so positive numbers remain positive and negative numbers remain negative.
- The bits that are not saturated are copied starting from the LSB side.

AP_WRAP_SM

The value should be sign-magnitude wrapped around.

```
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = 19.0; // Yields: -4.0
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = -19.0; // Yields: 2.0
```

If the value of N is set to zero (the default overflow mode):

- This mode uses sign magnitude wrapping.
- Sign bit set to the value of the least significant deleted bit.
- If the most significant remaining bit is different from the original MSB, all the remaining bits are inverted.
- If MSBs are same, the other bits are copied over.
 1. Delete redundant MSBs.
 2. The new sign bit is the least significant bit of the deleted bits. 0 in this case.
 3. Compare the new sign bit with the sign of the new value.
- If different, invert all the numbers. They are different in this case.

If N>0:

- Uses sign magnitude saturation

- N MSBs are saturated to 1.
- Behaves similar to a case in which N = 0, except that positive numbers stay positive and negative numbers stay negative.

Compiling ap_[u]fixed<> Types

To use the `ap_[u]fixed<>` classes, you must include the `ap_fixed.h` header file in all source files that reference `ap_[u]fixed<>` variables.

When compiling software models that use these classes, it may be necessary to specify the location of the Vitis HLS header files, for example by adding the “`-I/<HLS_HOME>/include`” option for `g++` compilation.

Declaring and Defining ap_[u]fixed<> Variables

There are separate signed and unsigned classes:

- `ap_fixed<W, I>` (signed)
- `ap_ufixed<W, I>` (unsigned)

You can create user-defined types with the C/C++ `typedef` statement:

```
#include "ap_fixed.h" // use ap_[u]fixed<> types
typedef ap_ufixed<128, 32> uint128_t; // 128-bit user defined type,
// 32 integer bits
```

Initialization and Assignment from Constants (Literals)

You can initialize `ap_[u]fixed` variable with normal floating point constants of the usual C/C++ width:

- 32 bits for type `float`
- 64 bits for type `double`

That is, typically, a floating point value that is single precision type or in the form of double precision. Note that the value assigned to the fixed-point variable will be limited by the precision of the constant.

```
#include <ap_fixed.h>

ap_ufixed<30, 15> my15BitInt = 3.1415;
ap_fixed<42, 23> my42BitInt = -1158.987;
ap_ufixed<99, 40> = 287432.0382911;
ap_fixed<36, 30> = -0x123.456p-1;
```

You can also use string initialization to ensure that all bits of the fixed-point variable are populated according to the precision described by the string.

```
ap_ufixed<2, 0> x      = "0b0.01";           // 0.25 in "dot" format
ap_ufixed<2, 0> y      = "0b01p-2";           // 0.25 in binary "scientific"
format
ap_ufixed<2, 0> z      = "0x4p-4";           // 0.25 in hex "scientific" format
ap_ufixed<62, 2> my_pi =
"0b11.001001000011111011010100010001000010110100011000010001101";
// pi with 60 fractional bits
```

The `ap_[u]fixed` types do not support initialization if they are used in an array of `std::complex` types.

```
typedef ap_fixed<DIN_W, 1, AP_TRN, AP_SAT> coeff_t; // MUST have IW >= 1
std::complex<coeff_t> twid_rom[REAL_SZ/2] = {{ 1, -0 }, { 0.9, -0.006 }, etc.}
```

The initialization values must first be cast to `std::complex`:

```
typedef ap_fixed<DIN_W, 1, AP_TRN, AP_SAT> coeff_t; // MUST have IW >= 1
std::complex<coeff_t> twid_rom[REAL_SZ/2] = {std::complex<coeff_t>( 1,
-0 ), std::complex<coeff_t>(0.9, -0.006 ), etc.}
```

Support for Console I/O (Printing)

As with initialization and assignment to `ap_[u]fixed<>` variables, Vitis HLS supports printing values that require more than 64 bits to represent.

The easiest way to output any value stored in an `ap_[u]fixed` variable is to use the C++ standard output stream, `std::cout` (`#include <iostream>` or `<iostream.h>`). The stream insertion operator, “`<<`”, is overloaded to correctly output the full range of values possible for any given `ap_[u]fixed` variable. The following stream manipulators are also supported, allowing formatting of the value as shown.

- `dec` (decimal)
- `hex` (hexadecimal)
- `oct` (octal)

```
#include <iostream.h>
// Alternative: #include <iostream>

ap_fixed<6,3, AP_RND, AP_WRAP> Val = 3.25;

cout << Val << endl;      // Yields: 3.25
```

Using the Standard C Library

You can also use the standard C library (`#include <stdio.h>`) to print out values larger than 64-bits:

1. Convert the value to a C++ `std::string` using the `ap_[u]fixed` classes method `to_string()`.
2. Convert the result to a null-terminated C character string using the `std::string` class method `c_str()`.

Optional Argument One (Specifying the Radix)

You can pass the `ap[u]int::to_string()` method an optional argument specifying the radix of the numerical format desired. The valid radix argument values are:

- 2 (binary)
- 8 (octal)
- 10 (decimal)
- 16 (hexadecimal) (default)

Optional Argument Two (Printing as Signed Values)

A second optional argument to `ap_[u]int::to_string()` specifies whether to print the non-decimal formats as signed values. This argument is boolean. The default value is false, causing the non-decimal formats to be printed as unsigned values.

```
ap_fixed<6, 3, AP_RND, AP_WRAP> Val = 3.25;  
  
printf("%s \n", in2.to_string().c_str()); // Yields: 0b011.010  
printf("%s \n", in2.to_string(10).c_str()); //Yields: 3.25
```

The `ap_[u]fixed` types are supported by the following C++ manipulator functions:

- `setprecision`
- `setw`
- `setfill`

The `setprecision` manipulator sets the decimal precision to be used. It takes one parameter `f` as the value of decimal precision, where `n` specifies the maximum number of meaningful digits to display in total (counting both those before and those after the decimal point).

The default value of `f` is 6, which is consistent with native C float type.

```
ap_fixed<64, 32> f = 3.14159;  
cout << setprecision (5) << f << endl;  
cout << setprecision (9) << f << endl;  
f = 123456;  
cout << setprecision (5) << f << endl;
```

The example above displays the following results where the printed results are rounded when the actual precision exceeds the specified precision:

```
3 . 1416
3 . 14159
1 . 2346e+05
```

The `setw` manipulator:

- Sets the number of characters to be used for the field width.
- Takes one parameter `w` as the value of the width
 - `w` determines the minimum number of characters to be written in some output representation.

If the standard width of the representation is shorter than the field width, the representation is padded with fill characters. Fill characters are controlled by the `setfill` manipulator which takes one parameter `f` as the padding character.

For example, given:

```
ap_fixed<65, 32> aa = 123456;
int precision = 5;
cout << setprecision(precision) << setw(13) << setfill('T') << aa << endl;
```

The output is:

```
TTT1 . 2346e+05
```

Expressions Involving `ap_[u]fixed<>` types

Arbitrary precision fixed-point values can participate in expressions that use any operators supported by C/C++. After an arbitrary precision fixed-point type or variable is defined, their usage is the same as for any floating point type or variable in the C/C++ languages.

Observe the following caveats:

- Zero and Sign Extensions

All values of smaller bit-width are zero or sign-extended depending on the sign of the source value. You may need to insert casts to obtain alternative signs when assigning smaller bit-widths to larger.

- Truncations

Truncation occurs when you assign an arbitrary precision fixed-point of larger bit-width than the destination variable.

Class Methods, Operators, and Data Members

In general, any valid operation that can be done on a native C/C++ integer data type is supported (using operator overloading) for `ap_[u]fixed` types. In addition to these overloaded operators, some class specific operators and methods are included to ease bit-level operations.

Binary Arithmetic Operators

Addition

```
ap_[u]fixed::RType ap_[u]fixed::operator + (ap_[u]fixed op)
```

Adds an arbitrary precision fixed-point with a given operand `op`.

The operands can be any of the following integer types:

- `ap_[u]fixed`
- `ap_[u]int`
- C/C++

The result type `ap_[u]fixed::RType` depends on the type information of the two operands.

```
ap_fixed<76, 63> Result;  
  
ap_fixed<5, 2> Val1 = 1.125;  
ap_fixed<75, 62> Val2 = 6721.35595703125;  
  
Result = Val1 + Val2; //Yields 6722.480957
```

Because `Val2` has the larger bit-width on both integer part and fraction part, the result type has the same bit-width and plus one to be able to store all possible result values.

Specifying the data's width controls resources by using the power functions, as shown below. In similar cases, Xilinx recommends specifying the width of the stored result instead of specifying the width of fixed point operations.

```
ap_ufixed<16,6> x=5;  
ap_ufixed<16,7>y=hl::rsqrt<16,6>(x+x);
```

Subtraction

```
ap_[u]fixed::RType ap_[u]fixed::operator - (ap_[u]fixed op)
```

Subtracts an arbitrary precision fixed-point with a given operand `op`.

The result type `ap_[u]fixed::RTType` depends on the type information of the two operands.

```
ap_fixed<76, 63> Result;  
  
ap_fixed<5, 2> Val1 = 1625.153;  
ap_fixed<75, 62> Val2 = 6721.355992351;  
  
Result = Val2 - Val1; // Yields 6720.23057
```

Because `Val2` has the larger bit-width on both integer part and fraction part, the result type has the same bit-width and plus one to be able to store all possible result values.

Multiplication

```
ap_[u]fixed::RTType ap_[u]fixed::operator * (ap_[u]fixed op)
```

Multiplies an arbitrary precision fixed-point with a given operand `op`.

```
ap_fixed<80, 64> Result;  
  
ap_fixed<5, 2> Val1 = 1625.153;  
ap_fixed<75, 62> Val2 = 6721.355992351;  
  
Result = Val1 * Val2; // Yields 7561.525452
```

This shows the multiplication of `Val1` and `Val2`. The result type is the sum of their integer part bit-width and their fraction part bit width.

Division

```
ap_[u]fixed::RTType ap_[u]fixed::operator / (ap_[u]fixed op)
```

Divides an arbitrary precision fixed-point by a given operand `op`.

```
ap_fixed<84, 66> Result;  
  
ap_fixed<5, 2> Val1 = 1625.153;  
ap_fixed<75, 62> Val2 = 6721.355992351;  
  
Val2 / Val1; // Yields 5974.538628
```

This shows the division of `Val2` and `Val1`. To preserve enough precision:

- The integer bit-width of the result type is sum of the integer bit-width of `Val2` and the fraction bit-width of `Val1`.
- The fraction bit-width of the result type is equal to the fraction bit-width of `Val2`.

Bitwise Logical Operators

Bitwise OR

```
ap_[u]fixed::RType ap_[u]fixed::operator | (ap_[u]fixed op)
```

Applies a bitwise operation on an arbitrary precision fixed-point and a given operand `op`.

```
ap_fixed<75, 62> Result;  
  
ap_fixed<5, 2> Val1 = 1625.153;  
ap_fixed<75, 62> Val2 = 6721.355992351;  
  
Result = Val1 | Val2; // Yields 6271.480957
```

Bitwise AND

```
ap_[u]fixed::RType ap_[u]fixed::operator & (ap_[u]fixed op)
```

Applies a bitwise operation on an arbitrary precision fixed-point and a given operand `op`.

```
ap_fixed<75, 62> Result;  
  
ap_fixed<5, 2> Val1 = 1625.153;  
ap_fixed<75, 62> Val2 = 6721.355992351;  
  
Result = Val1 & Val2; // Yields 1.00000
```

Bitwise XOR

```
ap_[u]fixed::RType ap_[u]fixed::operator ^ (ap_[u]fixed op)
```

Applies an `xor` bitwise operation on an arbitrary precision fixed-point and a given operand `op`.

```
ap_fixed<75, 62> Result;  
  
ap_fixed<5, 2> Val1 = 1625.153;  
ap_fixed<75, 62> Val2 = 6721.355992351;  
  
Result = Val1 ^ Val2; // Yields 6720.480957
```

Increment and Decrement Operators

Pre-Increment

```
ap_[u]fixed ap_[u]fixed::operator ++ ()
```

This operator function prefix increases an arbitrary precision fixed-point variable by 1.

```
ap_fixed<25, 8> Result;  
ap_fixed<8, 5> Val1 = 5.125;  
  
Result = ++Val1; // Yields 6.125000
```

Post-Increment

```
ap_[u]fixed ap_[u]fixed::operator ++ (int)
```

This operator function postfix:

- Increases an arbitrary precision fixed-point variable by 1.
- Returns the original val of this arbitrary precision fixed-point.

```
ap_fixed<25, 8> Result;  
ap_fixed<8, 5> Val1 = 5.125;  
  
Result = Val1++; // Yields 5.125000
```

Pre-Decrement

```
ap_[u]fixed ap_[u]fixed::operator -- ()
```

This operator function prefix decreases this arbitrary precision fixed-point variable by 1.

```
ap_fixed<25, 8> Result;  
ap_fixed<8, 5> Val1 = 5.125;  
  
Result = --Val1; // Yields 4.125000
```

Post-Decrement

```
ap_[u]fixed ap_[u]fixed::operator -- (int)
```

This operator function postfix:

- Decreases this arbitrary precision fixed-point variable by 1.
- Returns the original val of this arbitrary precision fixed-point.

```
ap_fixed<25, 8> Result;  
ap_fixed<8, 5> Val1 = 5.125;  
  
Result = Val1--; // Yields 5.125000
```

Unary Operators

Addition

```
ap_[u]fixed ap_[u]fixed::operator + ()
```

Returns a self copy of an arbitrary precision fixed-point variable.

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = +Val1; // Yields 5.125000
```

Subtraction

```
ap_[u]fixed::RType ap_[u]fixed::operator - ()
```

Returns a negative value of an arbitrary precision fixed-point variable.

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = -Val1; // Yields -5.125000
```

Equality Zero

```
bool ap_[u]fixed::operator ! ()
```

This operator function:

- Compares an arbitrary precision fixed-point variable with 0,
- Returns the result.

```
bool Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = !Val1; // Yields false
```

Bitwise Inverse

```
ap_[u]fixed::RType ap_[u]fixed::operator ~ ()
```

Returns a bitwise complement of an arbitrary precision fixed-point variable.

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = ~Val1; // Yields -5.25
```

Shift Operators

Unsigned Shift Left

```
ap_[u]fixed ap_[u]fixed::operator << (ap_uint<_W2> op)
```

This operator function:

- Shifts left by a given integer operand.
- Returns the result.

The operand can be a C/C++ integer type:

- `char`
- `short`
- `int`
- `long`

The return type of the shift left operation is the same width as the type being shifted.

Note: Shift does not support overflow or quantization modes.

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val = 5.375;

ap_uint<4> sh = 2;

Result = Val << sh; // Yields -10.5
```

The bit-width of the result is (`W` = 25, `I` = 15). Because the shift left operation result type is same as the type of `Val`:

- The high order two bits of `Val` are shifted out.
- The result is -10.5.

If a result of 21.5 is required, `Val` must be cast to `ap_fixed<10, 7>` first -- for example, `ap_ufixed<10, 7>(Val)`.

Signed Shift Left

```
ap_[u]fixed ap_[u]fixed::operator << (ap_int<-W2> op)
```

This operator:

- Shifts left by a given integer operand.
- Returns the result.

The shift direction depends on whether the operand is positive or negative.

- If the operand is positive, a shift right is performed.
- If the operand is negative, a shift left (opposite direction) is performed.

The operand can be a C/C++ integer type:

- `char`

- short
- int
- long

The return type of the shift right operation is the same width as the type being shifted.

```
ap_fixed<25, 15, false> Result;
ap_uint<8, 5> Val = 5.375;

ap_int<4> Sh = 2;
Result = Val << sh; // Shift left, yields -10.25

Sh = -2;
Result = Val << sh; // Shift right, yields 1.25
```

Unsigned Shift Right

```
ap_[u]fixed ap_[u]fixed::operator >> (ap_uint<_W2> op)
```

This operator function:

- Shifts right by a given integer operand.
- Returns the result.

The operand can be a C/C++ integer type:

- char
- short
- int
- long

The return type of the shift right operation is the same width as the type being shifted.

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val = 5.375;

ap_uint<4> sh = 2;

Result = Val >> sh; // Yields 1.25
```

If it is necessary to preserve all significant bits, extend fraction part bit-width of the Val first, for example `ap_fixed<10, 5>(Val)`.

Signed Shift Right

```
ap_[u]fixed ap_[u]fixed::operator >> (ap_int<_W2> op)
```

This operator:

- Shifts right by a given integer operand.
- Returns the result.

The shift direction depends on whether operand is positive or negative.

- If the operand is positive, a shift right performed.
- If operand is negative, a shift left (opposite direction) is performed.

The operand can be a C/C++ integer type (`char`, `short`, `int`, or `long`).

The return type of the shift right operation is the same width as type being shifted. For example:

```
ap_fixed<25, 15, false> Result;
ap_uint<8, 5> Val = 5.375;

ap_int<4> Sh = 2;
Result = Val >> sh; // Shift right, yields 1.25

Sh = -2;
Result = Val >> sh; // Shift left, yields -10.5

1.25
```

Relational Operators

Equality

```
bool ap_[u]fixed::operator == (ap_[u]fixed op)
```

This operator compares the arbitrary precision fixed-point variable with a given operand.

Returns `true` if they are equal and `false` if they are *not* equal.

The type of operand `op` can be `ap_[u]fixed`, `ap_int` or C/C++ integer types. For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 == Val2; // Yields true
Result = Val1 == Val3; // Yields false
```

Inequality

```
bool ap_[u]fixed::operator != (ap_[u]fixed op)
```

This operator compares this arbitrary precision fixed-point variable with a given operand.

Returns `true` if they are *not* equal and `false` if they are equal.

The type of operand `op` can be:

- `ap_[u]fixed`
- `ap_int`
- C or C++ integer types

For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 != Val2; // Yields false
Result = Val1 != Val3; // Yields true
```

Greater than or equal to

```
bool ap_[u]fixed::operator >= (ap_[u]fixed op)
```

This operator compares a variable with a given operand.

Returns `true` if they are equal or if the variable is greater than the operator and `false` otherwise.

The type of operand `op` can be `ap_[u]fixed`, `ap_int` or C/C++ integer types.

For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 >= Val2; // Yields true
Result = Val1 >= Val3; // Yields false
```

Less than or equal to

```
bool ap_[u]fixed::operator <= (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return `true` if it is equal to or less than the operand and `false` if not.

The type of operand `op` can be `ap_[u]fixed`, `ap_int` or C/C++ integer types.

For example:

```
bool Result;  
  
ap_ufixed<8, 5> Val1 = 1.25;  
ap_fixed<9, 4> Val2 = 17.25;  
ap_fixed<10, 5> Val3 = 3.25;  
  
Result = Val1 <= Val2; // Yields true  
Result = Val1 <= Val3; // Yields true
```

Greater than

```
bool ap_[u]fixed::operator > (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return `true` if it is greater than the operand and `false` if not.

The type of operand `op` can be `ap_[u]fixed`, `ap_int`, or C/C++ integer types.

For example:

```
bool Result;  
  
ap_ufixed<8, 5> Val1 = 1.25;  
ap_fixed<9, 4> Val2 = 17.25;  
ap_fixed<10, 5> Val3 = 3.25;  
  
Result = Val1 > Val2; // Yields false  
Result = Val1 > Val3; // Yields false
```

Less than

```
bool ap_[u]fixed::operator < (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return `true` if it is less than the operand and `false` if not.

The type of operand `op` can be `ap_[u]fixed`, `ap_int`, or C/C++ integer types. For example:

```
bool Result;  
  
ap_ufixed<8, 5> Val1 = 1.25;  
ap_fixed<9, 4> Val2 = 17.25;  
ap_fixed<10, 5> Val3 = 3.25;  
  
Result = Val1 < Val2; // Yields false  
Result = Val1 < Val3; // Yields true
```

Bit Operator

Bit-Select and Set

```
af_bit_ref ap_[u]fixed::operator [] (int bit)
```

This operator selects one bit from an arbitrary precision fixed-point value and returns it.

The returned value is a reference value that can set or clear the corresponding bit in the `ap_[u]fixed` variable. The `bit` argument must be an integer value and it specifies the index of the bit to select. The least significant bit has index 0. The highest permissible index is one less than the bit-width of this `ap_[u]fixed` variable.

The result type is `af_bit_ref` with a value of either 0 or 1. For example:

```
ap_int<8, 5> Value = 1.375;  
  
Value[3]; // Yields 1  
Value[4]; // Yields 0  
  
Value[2] = 1; // Yields 1.875  
Value[3] = 0; // Yields 0.875
```

Bit Range

```
af_range_ref af_(u)fixed::range (unsigned Hi, unsigned Lo)  
af_range_ref af_(u)fixed::operator [] (unsigned Hi, unsigned Lo)
```

This operation is similar to bit-select operator `[]` except that it operates on a range of bits instead of a single bit.

It selects a group of bits from the arbitrary precision fixed-point variable. The `Hi` argument provides the upper range of bits to be selected. The `Lo` argument provides the lowest bit to be selected. If `Lo` is larger than `Hi` the bits selected are returned in the reverse order.

The return type `af_range_ref` represents a reference in the range of the `ap_[u]fixed` variable specified by `Hi` and `Lo`. For example:

```
ap_uint<4> Result = 0;  
ap_ufixed<4, 2> Value = 1.25;  
ap_uint<8> Repl = 0xAA;  
  
Result = Value.range(3, 0); // Yields: 0x5  
Value(3, 0) = Repl(3, 0); // Yields: -1.5  
  
// when Lo > Hi, return the reverse bits string  
Result = Value.range(0, 3); // Yields: 0xA
```

Range Select

```
af_range_ref af_(u)fixed::range ()  
af_range_ref af_(u)fixed::operator []
```

This operation is the special case of the range select operator []. It selects all bits from this arbitrary precision fixed-point value in the normal order.

The return type `af_range_ref` represents a reference to the range specified by $Hi = W - 1$ and $Lo = 0$. For example:

```
ap_uint<4> Result = 0;  
  
ap_ufixed<4, 2> Value = 1.25;  
ap_uint<8> Repl = 0xAA;  
  
Result = Value.range(); // Yields: 0x5  
Value() = Repl(3, 0); // Yields: -1.5
```

Length

```
int ap_[u]fixed::length()
```

This function returns an integer value that provides the number of bits in an arbitrary precision fixed-point value. It can be used with a type or a value. For example:

```
ap_ufixed<128, 64> My128APFixed;  
  
int bitwidth = My128APFixed.length(); // Yields 128
```

Explicit Conversion Methods

Fixed to Double

```
double ap_[u]fixed::to_double()
```

This member function returns this fixed-point value in form of IEEE double precision format. For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;  
double Result;  
  
Result = MyAPFixed.to_double(); // Yields 333.789
```

Fixed to Float

```
float ap_[u]fixed::to_float()
```

This member function returns this fixed-point value in form of IEEE float precision format. For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;  
float Result;  
  
Result = MyAPFixed.to_float(); // Yields 333.789
```

Fixed to Half-Precision Floating Point

```
half ap_[u]fixed::to_half()
```

This member function return this fixed-point value in form of HLS half-precision (16-bit) float precision format. For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;  
half Result;  
  
Result = MyAPFixed.to_half(); // Yields 333.789
```

Fixed to ap_int

```
ap_int ap_[u]fixed::to_ap_int ()
```

This member function explicitly converts this fixed-point value to `ap_int` that captures all integer bits (fraction bits are truncated). For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;  
ap_uint<77> Result;  
  
Result = MyAPFixed.to_ap_int(); //Yields 333
```

Fixed to Integer

```
int ap_[u]fixed::to_int ()  
unsigned ap_[u]fixed::to_uint ()  
ap_slong ap_[u]fixed::to_int64 ()  
ap_ulong ap_[u]fixed::to_uint64 ()
```

This member function explicitly converts this fixed-point value to C built-in integer types. For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;  
unsigned int Result;  
  
Result = MyAPFixed.to_uint(); //Yields 333  
  
unsigned long long Result;  
Result = MyAPFixed.to_uint64(); //Yields 333
```



RECOMMENDED: Xilinx recommends that you explicitly call member functions instead of using C-style cast to convert `ap_[u]fixed` to other data types.

Compile Time Access to Data Type Attributes

The `ap_[u]fixed<>` types are provided with several static members that allow the size and configuration of data types to be determined at compile time. The data type is provided with the static const members: `width`, `iwidth`, `qmode` and `omode`:

```
static const int width = _AP_W;
static const int iwidth = _AP_I;
static const ap_q_mode qmode = _AP_Q;
static const ap_o_mode omode = _AP_O;
```

You can use these data members to extract the following information from any existing `ap_[u]fixed<>` data type:

- `width`: The width of the data type.
- `iwidth`: The width of the integer part of the data type.
- `qmode`: The quantization mode of the data type.
- `omode`: The overflow mode of the data type.

For example, you can use these data members to extract the data width of an existing `ap_[u]fixed<>` data type to create another `ap_[u]fixed<>` data type at compile time.

The following example shows how the size of variable `Res` is automatically defined as 1-bit greater than variables `Val1` and `Val2` with the same quantization modes:

```
// Definition of basic data type
#define INPUT_DATA_WIDTH 12
#define IN_INTG_WIDTH 6
#define IN_QMODE AP_RND_ZERO
#define IN_OMODE AP_WRAP
typedef ap_fixed<INPUT_DATA_WIDTH, IN_INTG_WIDTH, IN_QMODE, IN_OMODE>
data_t;
// Definition of variables
data_t Val1, Val2;
// Res is automatically sized at run-time to be 1-bit greater than
INPUT_DATA_WIDTH
// The bit growth in Res will be in the integer bits
ap_int<data_t::width+1, data_t::iwidth+1, data_t::qmode, data_t::omode> Res
= Val1 +
Val2;
```

This ensures that Vitis HLS correctly models the bit-growth caused by the addition even if you update the value of `INPUT_DATA_WIDTH`, `IN_INTG_WIDTH`, or the quantization modes for `data_t`.

Vitis HLS Math Library

The Vitis™ HLS Math Library (`hls_math.h`) provides coverage of math functions from C++ (`cmath`) libraries, and can be used in both C simulation and synthesis. It offers floating-point (single-precision, double-precision, and half-precision) for all functions and fixed-point support for the majority of the functions. The functions in `hls_math.h` is grouped in `hls` namespace, and can be used as in-place replacement of function of `std` namespace from the standard C++ math library (`cmath`).



IMPORTANT! Using `hls_math.h` header in C code is not supported.

HLS Math Library Accuracy

The HLS math functions are implemented as synthesizable bit-approximate functions from the `hls_math.h` library. Bit-approximate HLS math library functions do not provide the same accuracy as the standard C function. To achieve the desired result, the bit-approximate implementation might use a different underlying algorithm than the standard C math library version. The accuracy of the function is specified in terms of ULP (Unit of Least Precision). This difference in accuracy has implications for both C simulation and C/RTL co-simulation.

The ULP difference is typically in the range of 1-4 ULP.

- If the standard C math library is used in the C source code, there may be a difference between the C simulation and the C/RTL co-simulation due to the fact that some functions exhibit a ULP difference from the standard C math library.
- If the HLS math library is used in the C source code, there will be no difference between the C simulation and the C/RTL co-simulation. A C simulation using the HLS math library, may however differ from a C simulation using the standard C math library.

In addition, the following seven functions might show some differences, depending on the C standard used to compile and run the C simulation:

- `copysign`
- `fpclassify`
- `isinf`

- `isfinite`
- `isnan`
- `isnormal`
- `signbit`

C90 mode

Only `isinf`, `isnan`, and `copysign` are usually provided by the system header files, and they operate on doubles. In particular, `copysign` always returns a double result. This might result in unexpected results after synthesis if it must be returned to a float, because a double-to-float conversion block is introduced into the hardware.

C99 mode (-std=c99)

All seven functions are usually provided under the expectation that the system header files will redirect them to `__isnan(double)` and `__isnan(float)`. The usual GCC header files do not redirect `isnormal`, but implement it in terms of `fpclassify`.

C++ Using math.h

All seven are provided by the system header files, and they operate on doubles.

`copysign` always returns a double result. This might cause unexpected results after synthesis if it must be returned to a float, because a double-to-float conversion block is introduced into the hardware.

C++ Using cmath

Similar to C99 mode (`-std=c99`), except that:

- The system header files are usually different.
- The functions are properly overloaded for:
 - `float().snan(double)`
 - `isinf(double)`

`copysign` and `copysignf` are handled as built-ins even when using `namespace std;`.

C++ Using cmath and namespace std

No issues. Xilinx recommends using the following for best results:

- `-std=c99` for C
- `-fno-builtins` for C and C++

Note: To specify the C compile options, such as `-std=c99`, use the Tcl command `add_files` with the `-cflags` option. Alternatively, use the **Edit CFLAGS** button in the Project Settings dialog box.

HLS Math Library

The following functions are provided in the HLS math library. Each function supports half-precision (type `half`), single-precision (type `float`) and double precision (type `double`).



IMPORTANT! For each function `func` listed below, there is also an associated half-precision only function named `half_func` and single-precision only function named `funcf` provided in the library.

When mixing half-precision, single-precision and double-precision data types, check for common synthesis errors to prevent introducing type-conversion hardware in the final FPGA implementation.

Trigonometric Functions

<code>acos</code>	<code>acospi</code>	<code>asin</code>	<code>asinpi</code>
<code>atan</code>	<code>atan2</code>	<code>atan2pi</code>	<code>cos</code>
<code>cosp</code>	<code>sin</code>	<code>sincos</code>	<code>sinpi</code>
<code>tan</code>	<code>tanpi</code>		

Hyperbolic Functions

<code>acosh</code>	<code>asinh</code>	<code>atanh</code>	<code>cosh</code>
<code>sinh</code>	<code>tanh</code>		

Exponential Functions

<code>exp</code>	<code>exp10</code>	<code>exp2</code>	<code>expm1</code>
<code>frexp</code>	<code>ldexp</code>	<code>modf</code>	

Logarithmic Functions

<code>ilogb</code>	<code>log</code>	<code>log10</code>	<code>log1p</code>

Power Functions

<code>cbrt</code>	<code>hypot</code>	<code>pow</code>	<code>rsqrt</code>
<code>sqrt</code>			

Error Functions

<code>erf</code>	<code>erfc</code>

Rounding Functions

ceil	floor	llrint	llround
lrint	lround	nearbyint	rint
round	trunc		

Remainder Functions

fmod	remainder	remquo
------	-----------	--------

Floating-point

copysign	nan	nextafter	nexttoward
----------	-----	-----------	------------

Difference Functions

fdim	fmax	fmin	maxmag
minmag			

Other Functions

abs	divide	fabs	fma
fract	mad	recip	

Classification Functions

fpclassify	isfinite	isinf	isnan
isnormal	signbit		

Comparison Functions

isgreater	isgreaterequal	isless	islessequal
islessgreater	isunordered		

Relational Functions

all	any	bitselect	isequal
isnotequal	isordered	select	

Fixed-Point Math Functions

Fixed-point implementations are also provided for the following math functions.

All fixed-point math functions support `ap_[u]fixed` and `ap_[u]int` data types with following bit-width specification,

1. `ap_fixed<W, I>` where $I \leq 33$ and $W-I \leq 32$
2. `ap_ufixed<W, I>` where $I \leq 32$ and $W-I \leq 32$
3. `ap_int<I>` where $I \leq 33$
4. `ap_uint<I>` where $I \leq 32$



IMPORTANT! Fixed-point math functions from the `hls_math` library do not support the `ap_[u]fixed` template parameters `Q,O`, and `N`, for quantization mode, overflow mode, and the number of saturation bits, respectively. The quantization and overflow modes are only effective when an `ap_[u]fixed` variable is on the left hand of assignment or being initialized, but not during the calculation.

Trigonometric Functions

cos	sin	tan	acos	asin	atan	atan2	sincos
cospi	sinpi						

Hyperbolic Functions

cosh	sinh	tanh	acosh	asinh	atanh
------	------	------	-------	-------	-------

Exponential Functions

exp	frexp	modf	exp2	expm1
-----	-------	------	------	-------

Logarithmic Functions

log	log10	ilogb	log1p
-----	-------	-------	-------

Power Functions

pow	sqrt	rsqrt	cbrt	hypot
-----	------	-------	------	-------

Error Functions

erf	erfc
-----	------

Rounding Functions

ceil	floor	trunc	round	rint	nearbyint
------	-------	-------	-------	------	-----------

Floating Point

nextafter	nexttoward
-----------	------------

Difference Functions

erf erfc fdim fmax fmin maxmag minmag

Other Functions

fabs recip abs fract divide

Classification Functions

signbit

Comparison Functions

isgreater isgreaterequal isless islessequal islessgreater

Relational Functions

isequal isnotequal any all bitselect

The fixed-point type provides a slightly-less accurate version of the function value, but a smaller and faster RTL implementation.

The methodology for implementing a math function with a fixed-point data types is:

1. Determine if a fixed-point implementation is supported.
2. Update the math functions to use `ap_fixed` types.
3. Perform C simulation to validate the design still operates with the required precision. The C simulation is performed using the same bit-accurate types as the RTL implementation.
4. Synthesize the design.

For example, a fixed-point implementation of the function `sin` is specified by using fixed-point types with the math function as follows:

```
#include "hls_math.h"
#include "ap_fixed.h"

ap_fixed<32,2> my_input, my_output;

my_input = 24.675;
my_output = sin(my_input);
```

When using fixed-point math functions, the result type must have the same width and integer bits as the input.

Verification and Math Functions

If the standard C math library is used in the C source code, the C simulation results and the C/RTL co-simulation results may be different: if any of the math functions in the source code have an ULP difference from the standard C math library it may result in differences when the RTL is simulated.

If the `hls_math.h` library is used in the C source code, the C simulation and C/RTL co-simulation results are identical. However, the results of C simulation using `hls_math.h` are not the same as those using the standard C libraries. The `hls_math.h` library simply ensures the C simulation matches the C/RTL co-simulation results. In both cases, the same RTL implementation is created. The following explains each of the possible options which are used to perform verification when using math functions.

Verification Option 1: Standard Math Library and Verify Differences

In this option, the standard C math libraries are used in the source code. If any of the functions synthesized do have exact accuracy the C/RTL co-simulation is different than the C simulation. The following example highlights this approach.

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

typedef float data_t;

data_t cpp_math(data_t angle) {
    data_t s = sinf(angle);
    data_t c = cosf(angle);
    return sqrtf(s*s+c*c);
}
```

In this case, the results between C simulation and C/RTL co-simulation are different. Keep in mind when comparing the outputs of simulation, any results written from the test bench are written to the working directory where the simulation executes:

- C simulation: Folder `<project>/<solution>/csim/build`
- C/RTL co-simulation: Folder `<project>/<solution>/sim/<RTL>`

where `<project>` is the project folder, `<solution>` is the name of the solution folder and `<RTL>` is the type of RTL verified (Verilog or VHDL). The following figure shows a typical comparison of the pre-synthesis results file on the left-hand side and the post-synthesis RTL results file on the right-hand side. The output is shown in the third column.

Figure 133: Pre-Synthesis and Post-Synthesis Simulation Differences

	result.dat	proj_cpp_math.prj/solution1/sim/systemc/result.dat
1	0.000000000000000	0.00999999776483
2	1.000000000000000	0.10999999403954
3	2.000000000000000	0.209999993443489
4	3.000000000000000	0.310000002384186
5	4.000000000000000	0.409999996423721
6	5.000000000000000	0.509999990463257
7	6.000000000000000	0.61000014305115
8	7.000000000000000	0.71000038146973
9	8.000000000000000	0.81000061988831
10	9.000000000000000	0.91000085830688
11	10.0000000000000	1.01000109672546
12	11.0000000000000	1.10000133514404
13	12.0000000000000	1.21000157356262
14	13.0000000000000	1.31000181198120
15	14.0000000000000	1.41000205039978
16	15.0000000000000	1.51000228881836
17	16.0000000000000	1.61000252723694
18	17.0000000000000	1.71000276565552
19	18.0000000000000	1.81000300407410
20	19.0000000000000	1.91000324249268
21	20.0000000000000	2.01000228881836
22	21.0000000000000	2.11000133514404
23	22.0000000000000	2.21000038146973
24	23.0000000000000	2.30999942779541
25	24.0000000000000	2.409999847412109
26	25.0000000000000	2.509999752044678
27	26.0000000000000	2.609999656677246
28	27.0000000000000	2.709999561309814
29	28.0000000000000	2.80999465942383
30	29.0000000000000	2.90999370574951

The results of pre-synthesis simulation and post-synthesis simulation differ by fractional amounts. You must decide whether these fractional amounts are acceptable in the final RTL implementation.

The recommended flow for handling these differences is using a test bench that checks the results to ensure that they lie within an acceptable error range. This can be accomplished by creating two versions of the same function, one for synthesis and one as a reference version. In this example, only function `cpp_math` is synthesized.

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

typedef float data_t;

data_t cpp_math(data_t angle) {
    data_t s = sinf(angle);
    data_t c = cosf(angle);
    return sqrtf(s*s+c*c);
}

data_t cpp_math_sw(data_t angle) {
    data_t s = sinf(angle);
    data_t c = cosf(angle);
    return sqrtf(s*s+c*c);
}
```

The test bench to verify the design compares the outputs of both functions to determine the difference, using variable `diff` in the following example. During C simulation both functions produce identical outputs. During C/RTL co-simulation function `cpp_math` produces different results and the difference in results are checked.

```
int main() {
    data_t angle = 0.01;
    data_t output, exp_output, diff;
    int retval=0;

    for (data_t i = 0; i <= 250; i++) {
        output = cpp_math(angle);
        exp_output = cpp_math_sw(angle);

        // Check for differences
        diff = ( (exp_output > output) ? exp_output - output : output - exp_output );
        if (diff > 0.0000005) {
            printf("Difference %.10f exceeds tolerance at angle %.10f \n", diff, angle);
            retval=1;
        }

        angle = angle + .1;
    }

    if (retval != 0) {
        printf("Test failed !!!\n");
        retval=1;
    } else {
        printf("Test passed !\n");
    }
    // Return 0 if the test passes
    return retval;
}
```

If the margin of difference is lowered to 0.00000005, this test bench highlights the margin of error during C/RTL co-simulation:

```
Difference 0.0000000596 at angle 1.1100001335
Difference 0.0000000596 at angle 1.2100001574
Difference 0.0000000596 at angle 1.5100002289
Difference 0.0000000596 at angle 1.6100002527
etc..
```

When using the standard C math libraries (`math.h` and `cmath.h`) create a “smart” test bench to verify any differences in accuracy are acceptable.

Verification Option 2: HLS Math Library and Validate Differences

An alternative verification option is to convert the source code to use the HLS math library. With this option, there are no differences between the C simulation and C/RTL co-simulation results. The following example shows how the code above is modified to use the `hls_math.h` library.

Note: This option is only available in C++.

- Include the `hls_math.h` header file.
- Replace the math functions with the equivalent `hls::` function.

```
#include <cmath>
#include "hls_math.h"
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

typedef float data_t;

data_t cpp_math(data_t angle) {
    data_t s = hls::sinf(angle);
    data_t c = hls::cosf(angle);
    return hls::sqrtf(s*s+c*c);
}
```

Verification Option 3: HLS Math Library File and Validate Differences

Including the HLS math library file `lib_hlsm.cpp` as a design file ensures Vitis HLS uses the HLS math library for C simulation. This option is identical to option2 however it does not require the C code to be modified.

The HLS math library file is located in the `src` directory in the Vitis HLS installation area. Simply copy the file to your local folder and add the file as a standard design file.

Note: This option is only available in C++.

As with option 2, with this option there is now a difference between the C simulation results using the HLS math library file and those previously obtained without adding this file. These difference should be validated with C simulation using a “smart” test bench similar to option 1.

Common Synthesis Errors

The following are common use errors when synthesizing math functions. These are often (but not exclusively) caused by converting C functions to C++ to take advantage of synthesis for math functions.

C++ cmath.h

If the C++ `cmath.h` header file is used, the floating point functions (for example, `sinf` and `cosf`) can be used. These result in 32-bit operations in hardware. The `cmath.h` header file also overloads the standard functions (for example, `sin` and `cos`) so they can be used for float and double types.

C math.h

If the `C math.h` library is used, the single-precision functions (for example, `sinf` and `cosf`) are required to synthesize 32-bit floating point operations. All standard function calls (for example, `sin` and `cos`) result in doubles and 64-bit double-precision operations being synthesized.

Cautions

When converting C functions to C++ to take advantage of `math.h` support, be sure that the new C++ code compiles correctly before synthesizing with Vitis HLS. For example, if `sqrtf()` is used in the code with `math.h`, it requires the following code `extern` added to the C++ code to support it:

```
#include <math.h>
extern "C" float sqrtf(float);
```

To avoid unnecessary hardware caused by type conversion, follow the warnings on mixing double and float types discussed in [Floats and Doubles](#).

HLS Stream Library

Streaming data is a type of data transfer in which data samples are sent in sequential order starting from the first sample. Streaming requires no address management.

Modeling designs that use streaming data can be difficult in C. The approach of using pointers to perform multiple read and/or write accesses can introduce issues, because there are implications for the type qualifier and how the test bench is constructed.



IMPORTANT! The `hls::stream` class is only used in C++ designs.

Vitis HLS provides a C++ template class `hls::stream<>` for modeling streaming data structures. The streams implemented with the `hls::stream<>` class have the following attributes.

- In the C code, an `hls::stream<>` behaves like a FIFO of infinite depth. There is no requirement to define the size of an `hls::stream<>`.
- They are read from and written to sequentially. That is, after data is read from an `hls::stream<>`, it cannot be read again.
- An `hls::stream<>` on the top-level interface is by default implemented with an `ap_fifo` interface for the Vivado® IP flow, or as an axis interface for the Vitis™ kernel flow.
- Streams may be defined either locally or globally and are always implemented as internal FIFOs. Streams defined in the global scope follow the same rules as any other global variables.
- There are two possible stream declarations:
 - `hls::stream<Type>`: specify the data type for the stream.

An `hls::stream<>` internal to the design is implemented as a FIFO with a default depth of 2. The **STREAM** pragma or directive can be used to change the depth.

- `hls::stream<Type, Depth>`: specify the data type for the stream, and the FIFO depth.

Set the depth to prevent stalls. If any task in the design can produce or consume samples at a greater rate than the specified depth, the FIFOs might become empty (or full) resulting in stalls, because it is unable to read (or write).

This section shows how the `hls::stream<>` class can more easily model designs with streaming data. The topics in this section provide:

- An overview of modeling with streams and the RTL implementation of streams.
- How to use streams.
- Blocking reads and writes.
- Non-Blocking Reads and writes.
- Controlling the FIFO depth.

Note: The `hls::stream` class should always be passed between functions as a C++ reference argument. For example, `&my_stream`.

C Modeling and RTL Implementation

Streams are modeled as an infinite queue in software (and in the test bench during RTL co-simulation). There is no need to specify any depth to simulate streams in C++. Streams can be used inside functions and on the interface to functions. Internal streams may be passed as function parameters.

Streams can be used only in C++ based designs. Each `hls::stream<>` object must be written by a single process and read by a single process.

If an `hls::stream` is used on the top-level interface in the Vivado IP flow it is implemented in the RTL as a FIFO interface (`ap_fifo`) by default, but can be optionally implemented as an AXI4-Stream interface (`axis`). In the Vitis kernel flow it is by default implemented as an AXI4-Stream interface (`axis`).

If an `hls::stream` is used inside the design function and synthesized into hardware, it is implemented as a FIFO with a default depth of 2. In some cases, such as when interpolation is used, the depth of the FIFO might have to be increased to ensure the FIFO can hold all the elements produced by the hardware. Failure to ensure the FIFO is large enough to hold all the data samples generated by the hardware can result in a stall in the design (seen in C/RTL co-simulation and in the hardware implementation). The depth of the FIFO can be adjusted using the STREAM directive with the `depth` option. An example of this is provided in the example design `hls_stream`.



IMPORTANT! Ensure `hls::stream` variables are correctly sized when used in the default non-DATAFLOW regions.

If an `hls::stream` is used to transfer data between tasks (sub-functions or loops), you should immediately consider implementing the tasks in a DATAFLOW region where data streams from one task to the next. The default (non-DATAFLOW) behavior is to complete each task before starting the next task, in which case the FIFOs used to implement the `hls::stream` variables must be sized to ensure they are large enough to hold all the data samples generated by the producer task. Failure to increase the size of the `hls::stream` variables results in the error below:

```
ERROR: [XFORM 203-733] An internal stream xxxx.xxxx.V.user.V' with default
size is
used in a non-dataflow region, which may result in deadlock. Please
consider to
resize the stream using the directive 'set_directive_stream' or the 'HLS
stream'
pragma.
```

This error informs you that in a non-DATAFLOW region, the default FIFOs depth of 2 may not be large enough to hold all the data samples written to the FIFO by the producer task, and deadlock may occur.

Using HLS Streams

To use `hls::stream<>` objects in your code include the header file `hls_stream.h` as shown below. Streaming data objects are defined by specifying the type and variable name. In this example, a 128-bit unsigned integer type is defined and used to create a stream variable called `my_wide_stream`.

```
#include "ap_int.h"
#include "hls_stream.h"

typedef ap_uint<128> uint128_t; // 128-bit user defined type
hls::stream<uint128_t> my_wide_stream; // A stream declaration
```

Streams must use scoped naming. Xilinx recommends using the scoped `hls::` naming shown in the example above. However, if you want to use the `hls` namespace, you can rewrite the preceding example as:

```
#include <ap_int.h>
#include <hls_stream.h>
using namespace hls;

typedef ap_uint<128> uint128_t; // 128-bit user defined type
stream<uint128_t> my_wide_stream; // hls:: no longer required
```

Given a stream specified as `hls::stream<T>`, the type T can be:

- Any C++ native data type
- A Vitis HLS arbitrary precision type (for example, `ap_int<>`, `ap_ufixed<>`)

- A user-defined struct containing either of the above types

Note: General user-defined classes (or structures) that contain methods (member functions) should not be used as the type (T) for a stream variable.

A stream can also be specified as `hls::stream<Type, Depth>`, where Depth indicates the depth of the FIFO needed in the verification adapter that the HLS tool creates for RTL co-simulation.

Streams can be optionally named. Providing a name for the stream allows the name to be used in reporting. For example, Vitis HLS automatically checks to ensure all elements from an input stream are read during simulation. Given the following two streams:

```
hls::stream<uint8_t> bytestr_in1;
hls::stream<uint8_t> bytestr_in2("input_stream2");
```

Any warning on elements of the streams are reported as follows, where it is clear that `input_stream2` refers to `bytestr_in2`:

```
WARNING: Hls::stream 'hls::stream<unsigned char>.1' contains leftover data,
which
may result in RTL simulation hanging.
WARNING: Hls::stream 'input_stream2' contains leftover data, which may
result in RTL
simulation hanging.
```

When streams are passed into and out of functions, they must be passed-by-reference as in the following example:

```
void stream_function (
    hls::stream<uint8_t> &strm_out,
    hls::stream<uint8_t> &strm_in,
    uint16_t strm_len
)
```

Streaming examples are provided in the [Vitis-HLS-Introductory-Examples](#) repository on GitHub, under [Interface/Streaming](#). Additional design examples using streams are provided in the [Vitis_Accel_Examples](#) also on GitHub.

Vitis HLS also supports both blocking and non-blocking access methods for `hls::stream` objects, as described in the following sections.

Blocking API

The term blocking means that operation stalls until fresh data is available on the streaming channels. The blocking API automatically checks if the FIFO is full or empty and performs the read/write operation on the FIFO. There is no need to manually check for full and empty conditions of the FIFO. The blocking API is fully deterministic, which means that C-simulation and RTL/Co-simulation will behave exactly the same way. This blocking API should only be used in the scenarios described in the following sections. Other uses could lead to non-deterministic behavior.



TIP: When used in blocking mode the `hls::stream` may deadlock due to insufficiently sized data streams.

Blocking Write Methods

In this example, the value of variable `src_var` is pushed into the stream.

```
// Usage of void write(const T & wdata)

hls::stream<int> my_stream;
int src_var = 42;

my_stream.write(src_var);
```

The `<<` operator is overloaded such that it may be used in a similar fashion to the stream insertion operators for C++ stream (for example, iostreams and filestreams). The `hls::stream<>` object to be written to is supplied as the left-hand side argument and the value to be written as the right-hand side.

```
// Usage of void operator << (T & wdata)

hls::stream<int> my_stream;
int src_var = 42;

my_stream << src_var;
```

Blocking Read Methods

This method reads from the head of the stream and assigns the values to the variable `dst_var`.

```
// Usage of void read(T &rdata)

hls::stream<int> my_stream;
int dst_var;

my_stream.read(dst_var);
```

Alternatively, the next object in the stream can be read by assigning (using for example `=`, `+=`) the stream to an object on the left-hand side:

```
// Usage of T read(void)

hls::stream<int> my_stream;

int dst_var = my_stream.read();
```

The `>>` operator is overloaded to allow use similar to the stream extraction operator for C++ stream (for example, iostreams and filestreams). The `hls::stream` is supplied as the LHS argument and the destination variable the RHS.

```
// Usage of void operator >> (T & rdata)

hls::stream<int> my_stream;
int dst_var;

my_stream >> dst_var;
```

Stream.full() Method

This method of the `hls::stream` objects returns `true`, if and only if the object is full, as shown below:

```
hls::stream<int> my_stream;
int src_var = 42;
bool stream_full;

stream_full = my_stream.full();
```

This method can be used to ensure deterministic behavior of the blocking API.

Stream.empty() Method

This method of `hls::stream` objects returns `true` if the stream is empty, as shown below.

```
hls::stream<int> my_stream;
int dst_var;
bool stream_empty;

stream_empty = my_stream.empty();
```

The following example shows how a combination of non-blocking accesses and full/empty tests can provide error handling functionality when the RTL FIFOs are full or empty:

```
#include "hls_stream.h"
using namespace hls;

typedef struct {
    short    data;
    bool     valid;
    bool     invert;
};
```

```
    } input_interface;

    bool invert(stream<input_interface>& in_data_1,
               stream<input_interface>& in_data_2,
               stream<short>& output
    ) {
        input_interface in;
        bool full_n;

        // Read an input value or return
        if (!in_data_1.read_nb(in))
            if (!in_data_2.read_nb(in))
                return false;

        // If the valid data is written, return not-full (full_n) as true
        if (in.valid) {
            if (in.invert)
                full_n = output.write_nb(~in.data);
            else
                full_n = output.write_nb(in.data);
        }
        return full_n;
    }
```

Deterministic Behavior

As discussed in [Blocking API](#), the blocking API can have both deterministic and non-deterministic behavior as shown in the following code example:

```
func1()
{
    while(!s.empty()) {
        s.read();
    }
}
```

During C-simulation the data to the stream is always available so the `while` loop runs through to completion. However, if the data stream has a single bubble when running in hardware the `while` loop will exit, and `func1` will return prematurely. This will lead to non-deterministic behavior between C-simulation and RTL execution.

The proper way to implement this loop is to use the side-channel signal `tlast`, as shown in the following example. Refer to [AXI4-Stream Interfaces with Side-Channels](#) for more information.

```
func1()
{
    while(!tlast) {
        s1.read();
    }
}
```

The blocking API can be deterministic when used in one of the following ways.

- Case 1 - Simple read/write to the FIFO:

```
int data = in.read();
if (data >= 10)
    out1.write(data);
else
    out2.write(data);
```

- Case 2 - FULL and EMPTY check using blocking API:

This case does not perform any computation with side effects, i.e. read or write a stream, memory, or static variable is still considered to use only blocking stream accesses.

```
void df(hls::stream<...> &s1, hls::stream<...> &s2, ...) {
#pragma HLS dataflow

    p1(s1, ...);
    p2(s2, ...);
    ...
}
void p1(hls::stream<...> &s1, ...) {
    if (s1.empty())
        return;
    ... = s1.read();
    ...
}
void p2(hls::stream<...> &s2, ...) {
    if (s2.full())
        return;
    ...
    s2.write(...);
}
```

Non-Blocking API



IMPORTANT! Non-blocking API is only supported on interfaces using the *ap_fifo* protocol. More specifically, the AXI-Stream (*axis*) protocol does not support non-blocking accesses.

The term non-blocking means that lack of data (or too much data) on the stream does not block the operation of a function or the iteration of a loop. Non-blocking methods return a Boolean value indicating the status of a read or write: `true` if successful, `false` otherwise. However, using non-blocking APIs can lead to non-deterministic behavior which cannot be fully validated during either C-simulation or RTL/Co-simulation, and requires an RTL testbench to test it exhaustively. Non-deterministic behavior can occur when reading from an empty FIFO or writing to a full FIFO.

During C simulation, streams have an infinite size. It is therefore not possible to validate with C simulation if the stream is full. These methods can be verified only during RTL simulation when the FIFO sizes are defined (either the default size of 1, or an arbitrary size defined with the STREAM directive).



IMPORTANT! If the design is specified to use the block-level I/O protocol `ap_ctrl_none` and the design contains any `hls::stream` variables that employ non-blocking behavior, C/RTL co-simulation is not guaranteed to complete.

Non-Blocking Writes

Non-Blocking Writes

This method attempts to push variable `src_var` into the stream `my_stream`, returning a boolean `true` if successful. Otherwise, `false` is returned and the queue is unaffected.

```
// Usage of bool write_nb(const T & wdata)

hls::stream<int> my_stream;
int src_var = 42;

if (my_stream.write_nb(src_var)) {
    // Perform standard operations
    ...
} else {
    // Write did not occur
    return;
}
```

Non-Blocking Writes with Full Check

Non-blocking behavior can also be modeled using non-blocking writes with full checking conditions, as explained in [Stream.full\(\) Method](#). This can lead to non-deterministic behavior and should be verified in RTL simulation with a sophisticated test bench.

```
hls::stream<int> my_stream;
int src_var = 42;
bool stream_full;

stream_full = my_stream.full();
if(!stream_full)
    my_stream.write_nb(src_var);
```

Non-Blocking Read

Non-Blocking Reads

This method attempts to read a value from the stream, returning `true` if successful. Otherwise, `false` is returned and the queue is unaffected.

```
// Usage of bool read_nb(const T & wdata)

hls::stream<int> my_stream;
int dst_var;

if (my_stream.read_nb(dst_var)) {
    // Perform standard operations
```

```
...
} else {
    // Read did not occur
    return;
}
```

Non-Blocking Reads with Empty Check

Non-blocking behavior can also be modeled using non-blocking read with a check for an empty stream, as described in [Stream.empty\(\) Method](#). This can lead to non-deterministic behavior and should be verified in RTL simulation with a sophisticated test bench.

```
READ_ONLY_LOOP:
while (check != 0) {
    if ( !addr_strm.empty() )
    {
        addr_strm.read_nb(addr_for_HBM);
        hbm[addr_for_HBM] = some_data;
        check[0] = 1;
        ...
    }
    ...
    ...
    check = (check << 1);
}
```

Controlling the RTL FIFO Depth

For most designs using streaming data, the default RTL FIFO depth of 2 is sufficient. Streaming data is generally processed one sample at a time.

For multirate designs in which the implementation requires a FIFO with a depth greater than 2, you must determine (and set using the STREAM directive) the depth necessary for the RTL simulation to complete. If the FIFO depth is insufficient, RTL co-simulation stalls.

Because stream objects cannot be viewed in the GUI directives pane, the STREAM directive cannot be applied directly in that pane.

Right-click the function in which an `hls::stream<>` object is declared (or is used, or exists in the argument list) to:

- Select the STREAM directive.
- Populate the `variable` field manually with name of the stream variable.

Alternatively, you can:

- Specify the STREAM directive manually in the `directives.tcl` file, or
- Add it as a pragma in `source`.

HLS Vector Library



IMPORTANT! To use `hls::vector` objects in your code include the header file `hls_vector.h`.

Vector Data Type Usage

The vector data type is provided to easily model and synthesize SIMD-type vector operations. Vitis™ HLS vector data type can be defined as follows, where `T` is a primitive or user-defined type with most of the arithmetic operations defined on it. `N` is an integer greater than zero. Once a vector type variable is declared it can be used like any other primitive type variable to perform arithmetic and logic operations.

```
#include <hls_vector.h>
hls::vector<T,N> aVec;
```



TIP: The best performance is achieved when both the bit-width of `T` and `N` are integer powers of 2.

Memory Layout

For any vector type defined as `hls::vector<T,N>`, the storage is guaranteed to be contiguous of size `sizeof(T) * N` and aligned to the greatest power of 2 such that the allocated size is at least `sizeof(T) * N`. In particular, when `N` is a power of 2 and `sizeof(T)` is a power of 2, `vector<T, N>` is aligned to its total size. This matches vector implementation on most architectures.



TIP: When `sizeof(T) * N` is an integer power of 2, the allocated size will be exactly `sizeof(T) * N`, otherwise the allocated size will be larger to make alignment possible.

The following example shows the definition of a vector class that aligns itself as described above.

```
constexpr size_t gp2(size_t N)
{
    return (N > 0 && N % 2 == 0) ? 2 * gp2(N / 2) : 1;
}

template<typename T, size_t N> class alignas(gp2(sizeof(T) * N)) vector
{
    std::array<T, N> data;
};
```

Following are different examples of alignment:

```
hls::vector<char,8> char8Vec; // aligns on 8 Bytes boundary
hls::vector<int,8> int8Vec; // aligns on 32 byte boundary
```

Requirements and Dependencies

Vitis HLS vector types requires support for C++ 14 or later. It has the following dependencies on the standard headers:

- <array>
 - std::array<T, N>
- <cassert>
 - assert
- <initializer_list>
 - std::initializer_list<T>

Supported Operations

- Initialization:

```
hls::vector<int, 4> x; // uninitialized
hls::vector<int, 4> y = 10; // scalar initialized: all elements set to 10
hls::vector<int, 4> z = {0, 1, 2, 3}; // initializer list (must have 4 elements)
hls::vector<ap_int, 4> a; // uninitialized arbitrary precision data type
```

- Access:

The operator[] enables access to individual elements of the vector, similar to a standard array:

```
x[i] = ...; // set the element at index i
... = x[i]; // value of the element at index i
```

- Arithmetic:

They are defined recursively, relying on the matching operation on T.

Table 43: Arithmetic Operation

Operation	In Place	Expression	Reduction (Left Fold)
Addition	<code>+=</code>	<code>+</code>	<code>reduce_add</code>
Subtraction	<code>-=</code>	<code>-</code>	<i>non-associative</i>
Multiplication	<code>*=</code>	<code>*</code>	<code>reduce_mult</code>
Division	<code>/=</code>	<code>/</code>	<i>non-associative</i>
Remainder	<code>%=</code>	<code>%</code>	<i>non-associative</i>
Bitwise AND	<code>&=</code>	<code>&</code>	<code>reduce_and</code>

Table 43: Arithmetic Operation (cont'd)

Operation	In Place	Expression	Reduction (Left Fold)
Bitwise OR	<code> =</code>	<code> </code>	<code>reduce_or</code>
Bitwise XOR	<code>^ =</code>	<code>^</code>	<code>reduce_xor</code>
Shift Left	<code><<=</code>	<code><<</code>	<i>non-associative</i>
Shift Right	<code>>>=</code>	<code>>></code>	<i>non-associative</i>
Pre-increment	<code>+ + x</code>	<i>none</i>	<i>unary operator</i>
Pre-decrement	<code>- - x</code>	<i>none</i>	<i>unary operator</i>
Post-increment	<code>x + +</code>	<i>none</i>	<i>unary operator</i>
Post-decrement	<code>x - -</code>	<i>none</i>	<i>unary operator</i>

- Comparison:

Lexicographic order on vectors (returns bool):

Table 44: Operation

Operation	Expression
Less than	<code><</code>
Less or equal	<code><=</code>
Equal	<code>= =</code>
Different	<code>! =</code>
Greater or equal	<code>>=</code>
Greater than	<code>></code>

HLS Task Library



IMPORTANT! To use `hls::task` objects in your code include the header file `hls_task.h`.

The `hls::task` library provides a simpler way of modeling purely streaming kernels, allowing static instantiation of tasks with only streaming I/O (`hls::stream` or `hls::stream_of_blocks`). This reduces the need for checks for empty stream needed to model concurrent processes in C++.

The following is a simple example that can be found at [simple_data_driven](#) on GitHub:

```
void odds_and_evens(hls::stream<int> &in, hls::stream<int> &out1,
hls::stream<int> &out2) {
    hls_thread_local hls::stream<int> s1; // channel connecting t1 and
t2
    hls_thread_local hls::stream<int> s2; // channel connecting t1 and t3

    // t1 infinitely runs splitter, with input in and outputs s1 and s2
    hls_thread_local hls::task t1(splitter, in, s1, s2);
    // t2 infinitely runs function odds, with input s1 and output out1
    hls_thread_local hls::task t2(odds, s1, out1);
    // t3 infinitely runs function evens, with input s2 and output
    hls_thread_local hls::task t3(evens, s2, out2);
}
```

Notice the top-level function, `odds_and_evens` uses streaming input and output interfaces. This is a purely streaming kernel. The top-level function includes the following:

- `s1` and `s2` are thread-local streams (`hls_thread_local`) and are used to connect the task-channel tasks `t1` and `t2`. These streams need to be thread-local so that they can be kept alive across top-level calls.
- `t1`, `t2`, and `t3` are the thread-local `hls::task` that execute the functions (`splitter`, `odds`, and `evens` respectively). The tasks run infinitely and just process data on their input streams. No synchronization is needed.

However, this type of model does have some restrictions such as:

- You cannot access non-local memory
- Non-stream data, such as scalar and array variables, must all be local to the processes and cannot be passed as arguments
- You must explicitly describe the parallelism in the design by the specification of parallel tasks

The `hls::task` objects can be mixed freely with standard dataflow-style function calls, which can move data in and out of memories (DRAM and BRAM). Tasks also support splitting channels (`hls::split`) to support one-to-many data distributions to build pools of workers that process streams of data, and merging channels (`hls::merge`) to support many-to-one data aggregation.

Tasks and Channels

The original DATAFLOW model lets you write sequential functions, and then requires the Vitis™ HLS tool to identify dataflow processes (tasks) and make them parallel, analyze and manage dependencies, perform scalar propagation and optimizations such as array-to-stream.

Alternatively, the use of `hls::task` objects requires you to explicitly instantiate tasks and channels, managing parallelization yourself in your algorithm design. The purpose of `hls::task` is to define a programming model that supports parallel tasks using only streaming data channels. Tasks are not controlled by function call/return, but run whenever the input streams are not empty.



TIP: The `hls::task` library provides concurrent semantics so that the C-simulation will be consistent with the RTL. This eliminates some of the problems with the sequential dataflow model.

The following is an example of tasks and channels. You can see that only streaming interfaces (`hls::stream` or `hls::stream_of_blocks`) are used. You can also see that the top-level function defines the tasks and stream channels using the `hls_thread_local` keyword.

```
void func1(hls::stream<int> &in, hls::stream<int> &out1, hls::stream<int>
&out2) {
    int data = in.read();
    if (data >= 10)
        out1.write(data);
    else
        out2.write(data);
}
void func2(hls::stream<int> &in, hls::stream<int> &out) {
    out.write(in.read() + 1);
}
void func3(hls::stream<int> &in, hls::stream<int> &out) {
    out.write(in.read() + 2);
}
void top-func(hls::stream<int> &in, hls::stream<int> &out1,
hls::stream<int> &out2) {
    hls_thread_local hls::stream<int> s1; // channel connecting t1 and t2
    hls_thread_local hls::stream<int> s2; // channel connecting t1 and t3

    hls_thread_local hls::task t1(func1, in, s1, s2); // t1 infinitely runs
func1, with input in and outputs s1 and s2
    hls_thread_local hls::task t2(func2, s1, out1); // t2 infinitely runs
func2, with input s1 and output out1
    hls_thread_local hls::task t3(func3, s2, out2); // t3 infinitely runs
func3, with input s2 and output out2
}
```

The `hls::task` objects are variables that should be declared as `hls_thread_local` in order to keep the variable and the underlying thread alive across multiple calls of the instantiating function (`top_func`) in the example above. The task objects implicitly manage a thread that runs a function continuously, such as `func1`, `func2`, or `func3` in the example above. The function is the task body, and has an implicit infinite loop around it.

Each `hls::task` must be passed a set of arguments that include the function name, input and output channels `hls::streams` or `hls::stream_of_blocks`. The channels must also be declared `hls_thread_local` to keep them alive across calls of the top-level function. Non-stream data, such as scalar and array variables, must all be local to the task functions and cannot be passed as arguments.



IMPORTANT! Inclusion of `hls_task.h` makes `hls::stream` and `hls::stream_of_blocks` read calls blocking in C-simulation. This means that code that previously relied on reading an empty stream will now result in deadlock during simulation.

Supported I/O types

`hls::task` objects can only read and write streaming channels `hls::stream` and `hls::stream_of_blocks`. Note that both `hls::task` and the channels that connect to them must be declared as `hls_thread_local`.

Use of flushing pipelines

In general, `hls::task` designs must always use flushing pipelines (fpl) or free-running pipelines (frp), which also flush, because non-flushing pipelines introduce dependencies between process executions and thus may result in unexpected deadlocks.

Nested Tasks

In the following example, there are two instances of `task1` used in `task2`, both also instantiated as `hls::task` instances. This demonstrates that in addition to sequential functions the body of an `hls::task` can be functions containing only `hls::task` objects.

```
void task1(hls::stream<int> &in, hls::stream<int> &out) {
    hls_thread_local hls::stream<int> s1;

    hls_thread_local hls::task t1(func2, in, s1);
    hls_thread_local hls::task t2(func3, s1, out);
}

void task2(hls::stream<int> &in1, hls::stream<int> &in2, hls::stream<int>
&out1, hls::stream<int> &out2) {
    hls_thread_local hls::task tA(task1, in1, out1);
    hls_thread_local hls::task tB(task1, in2, out2);
}
```

The use of `hls_thread_local` is still required to ensure safe multiple instantiation of the intermediate network (`tA` and `tB`, both instances of `task1` in this example; and safe instances of the leaf-level processes `t1` inside `tA` and `tB`, both executing different copies of `func2`, and `t2` inside `tA` and `tB`).

Simulation and Co-simulation

C-simulation behavior for tasks and channels model will be the same as in C/RTL Co-simulation. Reading from an empty stream was previously allowed with only a warning informing that this condition can cause hangs during simulation. In Vitis HLS 2022.2 reading from an empty stream can cause deadlock even in C-simulation and therefore is now an error condition with the following messages:

- In designs containing `hls::task` objects:

```
ERROR [HLS SIM]: deadlock detected when simulating hls::tasks.  
Execute C-simulation in debug mode in the GUI and examine the source code  
location of all the blocked hls::stream::read() calls
```

- In designs that do not use `hls::task`:

```
ERROR [HLS SIM]: an hls::stream is read while empty, which may result in  
RTL simulation hanging. If this is not expected, execute C simulation in  
debug mode  
in the GUI and examine the source code location of the blocked  
hls::stream::read()  
call to debug. If this is expected, add -  
DHLS_STREAM_READ_EMPTY_RETURNS_GARBAGE to  
-cflags to turn this error into a warning and allow empty hls::stream  
reads to return  
the default value for the data type.
```



TIP: add `-DHLS_STREAM_READ_EMPTY_RETURNS_GARBAGE` to `-cflags` to turn this error into a warning

Tasks and Dataflow

`hls::task` also supports the definition of tasks inside of dataflow regions. The dataflow region allows the definition of processes that access external arrays mapped to M_AXI, scalar, or PIPO arguments from upper levels of the design hierarchy. This requires dataflow processes identified by the `#pragma HLS dataflow` statement, synchronized via `ap_ctrl_chain`, that read data from any non-streamed C++ data structure and output it as `hls::stream` or `hls::stream_of_blocks` channels for connection to `hls::tasks`. Tasks can then output streams that are read by other dataflow processes and written to M_AXI, scalar, or PIPO arguments.



IMPORTANT! Because `hls::task` objects cannot read or write M_AXI, scalar, or PIP0 arguments dataflow processes must read or write these interfaces and write or read stream channels to `hls::tasks` as shown in the example below.

The following example illustrates tasks and dataflow processes together. The top-level function (`top-func`) is a dataflow region that defines sequential functions `write_out()` and `read_in()`, as well as `hls::task` objects and `hls::stream` channels.

```
#include "hls_task.h"

// This is an I/O dataflow process
void write_out(int* out, int n, hls::stream<int> &s2) {
    for (int i=0; i<n; i++)
        out[i] = s2.read();
}

// This is an I/O dataflow process
void read_in(int* in, int n, hls::stream<int> &s1) {
    for (int i=0; i<n; i++)
        s1.write(in[i]);
}

// This is an hls::task body
void func1(hls::stream<int> &s1, hls::stream<int> &s3) {
    // No while(1) needed! This will be a task
    s3.write(... + s1.read());
}

// This is an hls::task body
void func2(hls::stream<int> &s3, hls::stream<int> &s2) {
    // No while(1) needed! This will be a task
    s2.write(... * s3.read());
}

// This could legally be at the top of the design hierarchy
void top-func(int *in, int *out, int n) {
#pragma HLS dataflow
    hls_thread_local hls::stream<int> sk3;
    hls_thread_local hls::task t2(func2, sk3, sk2);
    hls_thread_local hls::stream<int> sk2;

    read_in(in, n, sk1);                                // can access stream,
    scalar or array; calling order matters
    hls_thread_local hls::task t2(func2, sk3, sk2);      // can access only
    stream; instance order does not matter
    hls_thread_local hls::task t1(func1, sk1, sk3);      // can access only
    stream; instance order does not matter
    write_out(out, n, sk2);                            // can access stream,
    scalar or array; calling order matters
}
```

`#pragma HLS DATFLOW` is required for the two sequential functions, but the `hls::task` objects do not require it. Internally, Vitis HLS will automatically split `top-func`, including both regular dataflow processes and KPN processes into two dataflow regions:

1. One dataflow region using `ap_ctrl_chain` that contains regular dataflow processes, like `read_in()` and `write_out()`, in the order in which they appear in the C++ code, and a call to the `ap_ctrl_none` region below
2. A second dataflow region using `ap_ctrl_none`, containing the task and channels. The task declaration order does not matter.

As a result of this, you can expect to see two levels of hierarchy in the Dataflow viewer in the Vitis HLS GUI.

HLS Split/Merge Library



IMPORTANT! To use `hls::split<>` or `hls::merge<>` objects in your code include the header file `hls_np_channel.h` as shown in the example below.

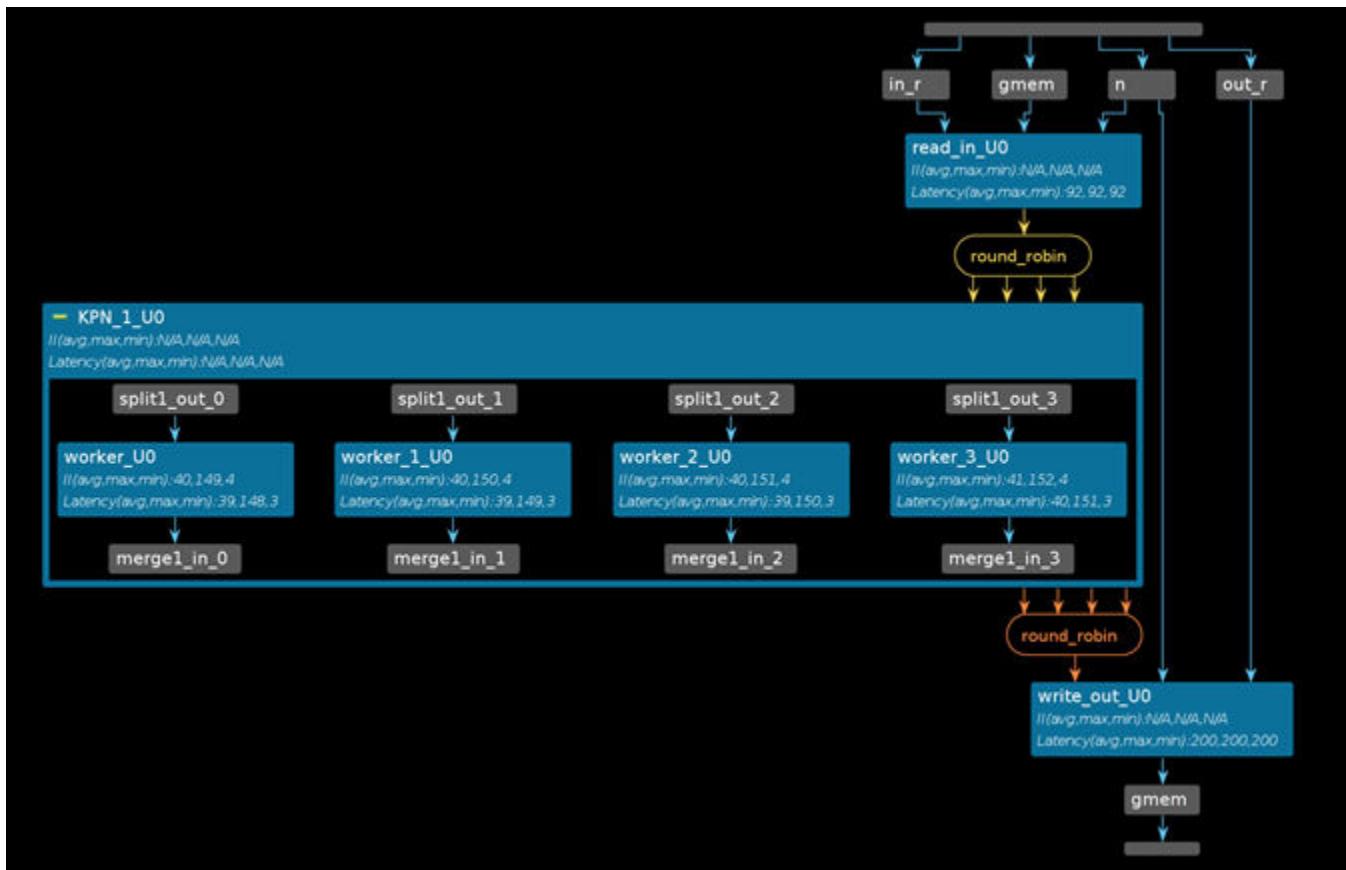
For use in Dataflow processes, split/merge channels let you create one-to-many or many-to-one type channels to distribute data to multiple tasks, or aggregate data from multiple tasks. These channels have a built-in job scheduler using either a round-robin approach in which data are sequentially distributed or gathered across the channels, or a load balancing approach that is determined based on channel availability.



TIP: Load balancing can lead to non-deterministic results in RTL/Co-simulation. In this case, you will need to write a test bench that is agnostic as to the order of results.

As shown in the figure below, data is read from an input stream and split through the round-robin scheduler mechanism, and distributed to associated worker tasks. After a worker completes the task, it writes the output which is merged also using the round-robin scheduler, into a single stream.

Figure 134: Split/Merge Dataflow



A split channel has one producer and many consumers, and can be typically used to distribute tasks to a set of workers, abstracting and implementing in RTL the distribution logic, and thus leading to both better performance and fewer resources. The distribution of an input to one of the N outputs can be:

- Round-robin, where the consumers read the input data in a fixed rotating order, thus ensuring deterministic behavior, but not allowing load sharing with dynamically varying computational loads for the workers.
- Load balancing, where the first consumer to attempt a read will read the first input data, thus ensuring good load balancing, but with non-deterministic results.

A merge channel has many producers and a single consumer, and operates based on the reverse logic:

- Round-robin, where the producer output data is merged using a fixed rotating order, thus ensuring deterministic behavior, but not allowing load sharing with dynamically varying computational loads for the workers.
- The load balancing merge channel, where the first producer that completes the work will write first into the channel with non-deterministic results.

The general idea of split and merge is that with the round_robin scheduler data are distributed around to workers for the split, and read from workers for the merge, in a deterministic fashion. So if all workers compute the same function the result is the same as with a single worker, but the performance is better.

If the workers perform different functions, then your design must ensure that the correct data item is sent to the correct function in the round-robin order of workers, starting from out[0] or in[0] respectively.

Specification

Specification of split/merge channels is as follows:

```
hls::split::load_balancing<DATATYPE, NUM_PORTS[, DEPTH, N_PORT_DEPTH]>
name;
hls::split::round_robin<DATATYPE, NUM_PORTS[, DEPTH]> name
hls::merge::load_balancing<DATATYPE, NUM_PORTS[, DEPTH]> name
hls::merge::round_robin<DATATYPE, NUM_PORTS[, DEPTH]> name
```

Where:

- **round_robin/load_balancing:** Specifies the type of scheduler mechanism used for the channel.
- **DATATYPE:** Specifies the data type on the channel. This has the same restrictions as standard `hls::stream`. The DATATYPE can be:
 - Any C++ native data type
 - A Vitis HLS arbitrary precision type (for example, `ap_int<>`, `ap_ufixed<>`)
 - A user-defined struct containing either of the above types
- **NUM_PORTS:** Indicates the number of write ports required for split (`1:num`) or read-ports required for merge (`num:1`) operation.
- **DEPTH:** Optional argument is the depth of the main buffer, located before the split or after the merge. This is optional, and the default depth is 2 when not specified.
- **N_PORT_DEPTH:** Optional field for round-robin to specify the depth of output buffers applied after split, or before merge. This is optional and the default depth is 0 when not specified.



TIP: To specify the optional `N_PORT_DEPTH` value, you must also specify `DEPTH`.

-
- **name:** Indicates the name of the created channel object

Following is an example which can be found at [mixed_control_and_data_driven](#) available on GitHub:

```
#include "hls_np_channel.h"

const int N = 16;
const int NP = 4;

void dut(int in[N], int out[N], int n) {
#pragma HLS dataflow
    hls::split::round_robin<int, NP> split1;
    hls::merge::round_robin<int, NP> merge1;

    read_in(in, n, split1.in);

    // Task-Channels
    hls_thread_local hls::task t[NP];
    for (int i=0; i<NP; i++) {
#pragma HLS unroll
        t[i](worker, split1.out[i], merge1.in[i]);
    }

    write_out(merge1.out, out, n);
}
```



TIP: The example above shows the workers implemented as `hls::task` objects. However, this is simply a feature of the example and not a requirement of `split/merge` channels.

Application of Split/Merge

The main use of split and merge is to support multiple compute engine instantiation to fully exploit the bandwidth of a DDR or HBM port. In this case, the producer is a load process that reads a burst of data from MAXI, and then passes the individual packets of data to be processed to a number of workers via the split channel. Use the round-robin protocol if the workers take similar amounts of time, or load balancing the execution time per input if variable. The consumer performs the reverse, writing data back into DRAM.



TIP: The write back address can be passed through the split and the merge, along with the data, in the case of load balancing.

These channels are modeled as implementing `hls::stream` objects at both ends of the split or the merge channel. This means that a split or merge channel end can be connected to any process that takes an `hls::stream` as an input or an output. The process does not need to be aware of the type of channel connection. Therefore, they can be used both for standard dataflow and for `hls::task` objects.

The following example shows how split can be used by a single produce and multiple consumers:

```
#include "hls_np_channel.h"

void producer(hls::stream<int> &s) {
    s.write(xxx);
}
```

```
void consumer1(hls::stream<int> &s) {
    ... = s.read();
}

void consumer2(hls::stream<int> &s) {
    ... = s.read();
}

void top-func() {
#pragma HLS dataflow
    hls::split::load_balancing<int, 4, 6> s; // NUM_PORTS=4, DEPTH=6

    producer(s.in, ...);
    consumer1(s.out[0], ...);
    consumer2(s.out[1], ...);
    consumer3(s.out[2], ...);
    consumer4(s.out[3], ...);
}
```

HLS Stream of Blocks Library



IMPORTANT! To use `hls::stream_of_blocks` objects in your code include the header file `hls_streamofblocks.h`.

The `hls::stream_of_blocks` type provides a user-synchronized stream that supports streaming blocks of data for process-level interfaces in a dataflow context, where each block is an array or multidimensional array. The intended use of stream-of-blocks is to replace array-based communication between a pair of processes within a dataflow region. Refer to the [using_stream_of_blocks](#) example on Github.

Currently, Vitis™ HLS implements arrays written by a producer process and read by a consumer process in a dataflow region by mapping them to ping pong buffers (PIPOs). The buffer exchange for a PIPO buffer occurs at the return of the producer function and the calling of the consumer function in C++.

While this ensures a concurrent communication semantic that is fully compliant with the sequential C++ execution semantics, it also implies that the consumer cannot start until the producer is done, as shown in the following code example.

```
void producer (int b[M][N], ...) {
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            b[i][f(j)] = ...;
}

void consumer(int b[M][N], ...) {
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            ... = b[i][g(j)] ...;
}

void top(...){
#pragma HLS dataflow
    int b[M][N];
#pragma HLS stream off variable=b

    producer(b, ...);
    consumer(b, ...);
}
```

This can unnecessarily limit throughput and/or increase resources if the producer generates data for the consumer in smaller blocks, for example by writing one row of the buffer output inside a nested loop, and the consumer uses the data in smaller blocks by reading one row of the buffer input inside a nested loop, as the example above does. In this example, due to the non-sequential buffer column access in the inner loop, you cannot simply stream the array `b`. However, the row access in the outer loop is sequential thus supporting `hls::stream_of_blocks` communication where each block is a 1-dimensional array of size `N`.

The main purpose of the `hls::stream_of_blocks` feature is to provide PIPO-like functionality, but with user-managed explicit synchronization, accesses, and a better coding style. Stream-of-blocks lets you avoid the use of dataflow in a loop containing the producer and consumer, which would have been a way to optimize the example above. However, in this case, the use of the dataflow loop containing the producer and consumer requires the use of a PIPO buffer ($2 \times N$) as shown in the following example:

```
void producer (int b[N], ...) {
    for (int j = 0; j < N; j++)
        b[f(j)] = ...;
}

void consumer(int b[N], ...) {
    for (int j = 0; j < N; j++)
        ... = b[g(j)];
}

void top(...){
// The loop below is very constrained in terms of how it must be written
    for (int i = 0; i < M; i++) {
#pragma HLS dataflow
        int b[N];
#pragma HLS stream off variable=b

        producer(b, ...); // writes b
        consumer(b, ...); // reads b
    }
}
```

The dataflow-in-a-loop code above is also not desirable because this structure has several limitations in Vitis HLS, such as the loop structure must be very constrained (single induction variable, starting from 0 and compared with a constant or a function argument and incremented by 1).

Stream-of-Blocks Modeling Style

On the other hand, for a stream-of-blocks the communication between the producer and the consumer is modeled as a stream of array-like objects, providing several advantages over array transfer through PIPO.

The use of stream-of-blocks in your code requires the following include file:

```
#include "hls_streamofblocks.h"
```

The stream-of-blocks object template is: `hls::stream_of_blocks<block_type, depth> v`

Where:

- `<block_type>` specifies the datatype of the array or multidimensional array held by the stream-of-blocks
- `<depth>` is an optional argument that provides depth control just like `hls::stream` or PIPOs, and specifies the total number of blocks, including the one acquired by the producer and the one acquired by the consumer at any given time. The default value is 2
- `v` specifies the variable name for the stream-of-blocks object

Use the following steps to access a block in a stream of blocks:

1. The producer or consumer process that wants to access the stream first needs to acquire access to it, using a `hls::write_lock` or `hls::read_lock` object.
2. After the producer has acquired the lock it can start writing (or reading) the acquired block. Once the block has been fully initialized, it can be released by the producer, when the `write_lock` object goes out of scope.

Note: The producer process with a `write_lock` can also read the block as long as it only reads from already written locations, because the newly acquired buffer must be assumed to contain uninitialized data. The ability to write and read the block is unique to the producer process, and is not supported for the consumer.

3. Then the block is queued in the stream-of-blocks in a FIFO fashion, and when the consumer acquires a `read_lock` object, the block can be read by the consumer process.

The main difference between `hls::stream_of_blocks` and the PIPO mechanism seen in the prior examples is that the block becomes available to the consumer as soon as the `write_lock` goes out of scope, rather than only at the return of the producer process. Hence the size of storage required to manage the original example (without the dataflow loop) is much less with stream-of-blocks than with just PIPOs: namely $2N$ instead of $2xMxN$ in the example.

Rewriting the prior example to use `hls::stream_of_blocks` is shown in the example below. The producer acquires the block by constructing an `hls::write_lock` object called `b`, and passing it the reference to the stream-of-blocks object, called `s`. The `write_lock` object provides an overloaded array access operator, letting it be accessed as an array to access underlying storage in random order as shown in the example below.

The acquisition of the lock is performed by constructing the `write_lock/read_lock` object, and the release occurs automatically when that object is destructed as it goes out of scope. This approach uses the common *Resource Acquisition Is Initialization (RAII)* style of locking and unlocking.

```
#include "hls_streamofblocks.h"
typedef int buf[N];
void producer (hls::stream_of_blocks<buf> &s, ...) {
    for (int i = 0; i < M; i++) {
        // Allocation of hls::write_lock acquires the block for the producer
        hls::write_lock<buf> b(s);
        for (int j = 0; j < N; j++)
            b[f(j)] = ...;
        // Deallocation of hls::write_lock releases the block for the consumer
    }
}

void consumer(hls::stream_of_blocks<buf> &s, ...) {
    for (int i = 0; i < M; i++) {
        // Allocation of hls::read_lock acquires the block for the consumer
        hls::read_lock<buf> b(s);
        for (int j = 0; j < N; j++)
            ... = b[g(j)] ...;
        // Deallocation of hls::write_lock releases the block to be reused by
        the producer
    }
}

void top(...){
#pragma HLS dataflow
    hls::stream_of_blocks<buf> s;

    producer(b, ...);
    consumer(b, ...);
}
```

The key features of this approach include:

- The expected performance of the outer loop in the producer above is to achieve an overall Initiation Interval (II) of 1
- A locked block can be used as though it were private to the producer or the consumer process until it is released.
- The initial state of the array object for the producer is undefined, whereas it contains the values written by the producer for the consumer.
- The principal advantage of stream-of-blocks is to provide overlapped execution of multiple iterations of the consumer and the producer to increase throughput.

Resource Usage

The resource cost when increasing the depth beyond the default value of 2 is similar to the resource cost of PIPOs. Namely, each increment of 1 will require enough memory for a block, e.g., in the example above $N * 32$ -bit words.

The stream of blocks object can be bound to a specific RAM type, by placing the `BIND_STORAGE` pragma where the stream-of-blocks is declared, for example in the top-level function. The stream of blocks uses 2-port BRAM (`type=RAM_2P`) by default.



IMPORTANT! `ARRAY_RESHAPE` and `ARRAY_PARTITION` are not supported for stream-of-blocks.

Checking for Full and Empty Blocks

The `read_lock` and `write_lock` are like `while(empty)` or `while(full)` loops - they keep trying to acquire the resource until they get the resource - so the code execution will stall until the lock is acquired. You can use the `empty()` and `full()` methods as shown in the following example to determine if a call to `read_lock` or `write_lock` will stall due to the lack of available blocks to be acquired.

```
#include "hls_streamofblocks.h"

void reader(hls::stream_of_blocks<buf> &in1, hls::stream_of_blocks<buf>
&in2, int out[M][N], int c) {
    for(unsigned j = 0; j < M;) {
        if (!in1.empty()) {
            hls::read_lock<ppbuf> arr1(in1);
            for(unsigned i = 0; i < N; ++i) {
                out[j][i] = arr1[N-1-i];
            }
            j++;
        } else if (!in2.empty()) {
            hls::read_lock<ppbuf> arr2(in2);
            for(unsigned i = 0; i < N; ++i) {
                out[j][i] = arr2[N-1-i];
            }
            j++;
        }
    }
}

void writer(hls::stream_of_blocks<buf> &out1, hls::stream_of_blocks<buf>
&out2, int in[M][N], int d) {
    for(unsigned j = 0; j < M; ++j) {
        if (d < 2) {
            if (!out1.full()) {
                hls::write_lock<ppbuf> arr(out1);
                for(unsigned i = 0; i < N; ++i) {
                    arr[N-1-i] = in[j][i];
                }
            }
        } else {
            if (!out2.full()) {
                hls::write_lock<ppbuf> arr(out2);
                for(unsigned i = 0; i < N; ++i) {
                    arr[N-1-i] = in[j][i];
                }
            }
        }
    }
}
```

```
    }
}

void top(int in[M][N], int out[M][N], int c, int d) {
#pragma HLS dataflow
    hls::stream_of_blocks<buf, 3> strm1, strm; // Depth=3
    writer(strm1, strm2, in, d);
    reader(strm1, strm2, out, c);
}
```

The producer and the consumer processes can perform the following actions within any scope in their body. As shown in the various examples, the scope will typically be a loop, but this is not required. Other scopes such as conditionals are also supported. Supported actions include:

- Acquire a block, i.e. an array of any supported data type.
 - In the case of the producer, the array will be empty, i.e. initialized according to the constructor (if any) of the underlying data type.
 - In the case of the consumer, the array will be full (of course in as much as the producer has filled it; the same requirements as for PIPO buffers, namely full writing if needed apply).
- Use the block for both reading and writing as if it were private local memory, up to its maximum allocated number of ports based on a `BIND_STORAGE` pragma or directive specified for the stream of blocks, which specifies what ports each side can see:
 - 1 port means that each side can access only one port, and the final stream-of-blocks can use a single dual-port memory for implementation.
 - 2 ports means that each side can use 1 or 2 ports depending on the schedule:
 - If the scheduler uses 2 ports on at least one side, merging will not happen
 - If the scheduler uses 1 port, merging can happen
 - If the pragma is not specified, the scheduler will decide, based on the same criteria currently used for local arrays. Moreover:
 - The producer can both write and read the block it has acquired
 - The consumer can only read the block it has acquired
- Automatically release the block when exiting the scope in which it was acquired. A released block:
 - If released by the producer, can be acquired by the consumer.
 - If released by the consumer, can be acquired to be reused by the producer, after being re-initialized by the constructor, if any. This initialization may slow down the design, hence often it is not desired. You may use the `__no_ctor__` attribute (explained earlier for `std::complex`) to prevent calling the constructor for the array elements.

A stream-of-blocks is very similar in spirit to a PIPO buffer. In the case of a PIPO, acquire is the same as calling the producer or consumer process function, while the release is the same as returning from it. This means that:

- the handshakes for a PIPO are
 - ap_start/ap_ready on the consumer side and
 - ap_done/ap_continue on the producer side.
- the handshakes of a stream of blocks are
 - its own read/empty_n on the consumer side and
 - write/full_n on the producer side.

Modeling Feedback in Dataflow Regions

One main limitation of PIPO buffers is that they can flow only forward with respect to the function call sequence in C++. In other words, the following connection is not supported with PIPOs, while it can be supported with `hls::stream_of_blocks`:

```
void top(...) {
    int b[N];
    for (int i = 0; i < M; i++) {
#pragma HLS dataflow
#pragma HLS stream off variable=b
        consumer(b, ...); // reads b
        producer(b, ...); // writes b
    }
}
```

The following code example is contrived to demonstrate the concept:

```
#include "hls_streamofblocks.h"
typedef int buf[N];
void producer (hls::stream_of_blocks<buf> &out, ...) {
    for (int i = 0; i < M; i++) {
        hls::write_lock<buf> arr(out);
        for (int j = 0; j < N; j++)
            arr[f(j)] = ...;
    }
}

void consumer(hls::stream_of_blocks<buf> &in, ...) {
    if (in.empty()) // execute only when producer has already generated some
meaningful data
        return;

    for (int i = 0; i < M; i++) {
        hls::read_lock<buf> arr(in);
        for (int j = 0; j < N; j++)
            ... = arr[g(j)];
        ...
    }
}

void top(...){
    // Note the large non-default depth.
```

```
// The producer must complete execution before the consumer can start
again, due to ap_ctrl_chain.
// A smaller depth would require ap_ctrl_none
hls::stream_of_blocks<buf, M+2> backward;

for (int i = 0; i < M; i++) {
#pragma HLS dataflow
    consumer(backward, ...); // reads backward
    producer(backward, ...); // writes backward
}
```

Limitations

There are some limitations with the use of `hls::stream_of_blocks` that you should be aware of:

- Each `hls::stream_of_blocks` object must have a single producer and consumer process, and each process must be different. In other words, local streams-of-blocks within a single process are not supported.
- You cannot use `hls::stream_of_blocks` within a sequential region. The producer and consumer must be separate concurrent processes in a dataflow region.
- You cannot use multiple nested acquire/release statements (`write_lock/read_lock`), for example in the same or nested scopes, as shown in the following example:

```
using ppbuf = int[N];
void readerImplicitNested(hls::stream_of_blocks<ppbuf>& in, ...) {
    for(unsigned j = 0; j < M; ++j) {
        hls::read_lock<ppbuf> arrA(in); // constructor would acquire A
    first
        hls::read_lock<ppbuf> arrB(in); // constructor would acquire B
    second
        for(unsigned i = 0; i < N; ++i)
            ... = arrA[f(i)] + arrB[g(i)];
        // destructor would release B first
        // destructor would release A second
    }
}
```

However, you can use multiple sequential or mutually exclusive acquire/release statements (`write_lock/read_lock`), for example inside IF/ELSE branches or in two subsequent code blocks. This is shown in the following example:

```
void readerImplicitNested(hls::stream_of_blocks<ppbuf>& in, ...) {
    for(unsigned j = 0; j < M; ++j) {
    {
        hls::read_lock<ppbuf> arrA(in); // constructor acquires A
        for(unsigned i = 0; i < N; ++i)
            ... = arrA[f(i)];
        // destructor releases A
    }
    {
        hls::read_lock<ppbuf> arrB(in); // constructor acquires B
        for(unsigned i = 0; i < N; ++i)
```

```
    ... = arrB[g(i)];  
    } // destructor releases B  
}
```

- Explicit release of locks in producer and consumer processes are not recommended, as they are automatically released when they go out of scope. However, you can use these by adding `#define EXPLICIT_ACQUIRE_RELEASE` before `#include "hls_streamofblocks.h"` in your source code.

HLS IP Libraries

Vitis™ HLS provides C++ libraries to implement a number of Xilinx® IP blocks. The C libraries allow the following Xilinx IP blocks to be directly inferred from the C++ source code ensuring a high-quality implementation in the FPGA.

Table 45: HLS IP Libraries

Library Header File	Description
hls_fft.h	Allows the Xilinx LogiCORE IP FFT to be simulated in C and implemented using the Xilinx LogiCORE block.
hls_fir.h	Allows the Xilinx LogiCORE IP FIR to be simulated in C and implemented using the Xilinx LogiCORE block.
hls_dds.h	Allows the Xilinx LogiCORE IP DDS to be simulated in C and implemented using the Xilinx LogiCORE block.
ap_shift_reg.h	Provides a C++ class to implement a shift register which is implemented directly using a Xilinx SRL primitive.

FFT IP Library

The Xilinx FFT IP block can be called within a C++ design using the library `hls_fft.h`. This section explains how the FFT can be configured in your C++ code.



RECOMMENDED: Xilinx highly recommends that you review the *Fast Fourier Transform LogiCORE IP Product Guide (PG109)* for information on how to implement and use the features of the IP.

To use the FFT in your C++ code:

1. Include the `hls_fft.h` library in the code
2. Set the default parameters using the predefined struct `hls::ip_fft::params_t`
3. Define the runtime configuration
4. Call the FFT function
5. Optionally, check the runtime status

The following code examples provide a summary of how each of these steps is performed. Each step is discussed in more detail below.

First, include the FFT library in the source code. This header file resides in the include directory in the Vitis HLS installation area which is automatically searched when Vitis HLS executes.

```
#include "hls_fft.h"
```

Define the static parameters of the FFT. This includes such things as input width, number of channels, type of architecture, which do not change dynamically. The FFT library includes a parameterization struct `hls::ip_fft::params_t`, which can be used to initialize all static parameters with default values.

In this example, the default values for output ordering and the widths of the configuration and status ports are over-ridden using a user-defined struct `param1` based on the predefined struct.

```
struct param1 : hls::ip_fft::params_t {  
    static const unsigned ordering_opt = hls::ip_fft::natural_order;  
    static const unsigned config_width = FFT_CONFIG_WIDTH;  
    static const unsigned status_width = FFT_STATUS_WIDTH;  
};
```

Define types and variables for both the runtime configuration and runtime status. These values can be dynamic and are therefore defined as variables in the C code which can change and are accessed through APIs.

```
typedef hls::ip_fft::config_t<param1> config_t;  
typedef hls::ip_fft::status_t<param1> status_t;  
config_t fft_config1;  
status_t fft_status1;
```

Next, set the runtime configuration. This example sets the direction of the FFT (Forward or Inverse) based on the value of variable “direction” and also set the value of the scaling schedule.

```
fft_config1.setDir(direction);  
fft_config1.setSch(0x2AB);
```

Call the FFT function using the HLS namespace with the defined static configuration (`param1` in this example). The function parameters are, in order, input data, output data, output status and input configuration.

```
hls::fft<param1> (xn1, xk1, &fft_status1, &fft_config1);
```

Finally, check the output status. This example checks the overflow flag and stores the results in variable “ovflo”.

```
*ovflo = fft_status1->getOvflo();
```



TIP: The example above shows the use of scalar values and arrays, but the FFT function also supports the use of `hls::stream` for arguments. Refer to [Using FFT Function with Streaming Interface](#) for more information.

FFT Static Parameters

The static parameters of the FFT define how the FFT is configured and specifies the fixed parameters such as the size of the FFT, whether the size can be changed dynamically, whether the implementation is pipelined or `radix_4_burst_io`.

The `hls_fft.h` header file defines a struct `hls::ip_fft::params_t` which can be used to set default values for the static parameters. If the default values are to be used, the parameterization struct can be used directly with the FFT function.

```
hls::fft<hls::ip_fft::params_t >
    (xn1, xk1, &fft_status1, &fft_config1);
```

A more typical use is to change some of the parameters to non-default values. This is performed by creating a new user-defined parameterization struct based on the default parameterization struct and changing some of the default values.

In the following example, a new user struct `my_fft_config` is defined with a new value for the output ordering (changed to `natural_order`). All other static parameters to the FFT use the default values.

```
struct my_fft_config : hls::ip_fft::params_t {
    static const unsigned ordering_opt = hls::ip_fft::natural_order;
};

hls::fft<my_fft_config >
    (xn1, xk1, &fft_status1, &fft_config1);
```

The parameters used for the FFT struct `hls::ip_fft::params_t` are explained below. The default values for the parameters and a list of possible values are provided.

Table 46: FFT Struct Parameters

Parameter	Description	C Type	Default Value	Valid Values
input_width	Data input port width.	unsigned	16	8-34
output_width	Data output port width.	unsigned	16	input_width to (input_width + max_nfft + 1)
status_width	Output status port width.	unsigned	8	Depends on FFT configuration
config_width	Input configuration port width.	unsigned	16	Depends on FFT configuration
max_nfft	The size of the FFT data set is specified as <code>1 << max_nfft</code> .	unsigned	10	3-16
has_nfft	Determines if the size of the FFT can be runtime configurable.	bool	false	True, False
channels	Number of channels.	unsigned	1	1-12

Table 46: FFT Struct Parameters (cont'd)

Parameter	Description	C Type	Default Value	Valid Values
arch_opt	The implementation architecture.	unsigned	pipelined_streaming_io	automatically_select pipelined_streaming_io radix_4_burst_io radix_2_burst_io radix_2_lite_burst_io
phase_factor_width	Configure the internal phase factor precision.	unsigned	16	8-34
ordering_opt	The output ordering mode.	unsigned	bit_reversed_order	bit_reversed_order natural_order
ovflo	Enable overflow mode.	bool	true	true false
scaling_opt	Define the scaling options.	unsigned	scaled	scaled unscaled block_floating_point
rounding_opt	Define the rounding modes.	unsigned	truncation	truncation convergent_rounding
mem_data	Specify using block or distributed RAM for data memory.	unsigned	block_ram	block_ram distributed_ram
mem_phase_factors	Specify using block or distributed RAM for phase factors memory.	unsigned	block_ram	block_ram distributed_ram
mem_reorder	Specify using block or distributed RAM for output reorder memory.	unsigned	block_ram	block_ram distributed_ram
stages_block_ram	Defines the number of block RAM stages used in the implementation.	unsigned	(max_nfft < 10) ? 0 : (max_nfft - 9)	0-11
mem_hybrid	When block RAMs are specified for data, phase factor, or reorder buffer, mem_hybrid specifies where or not to use a hybrid of block and distributed RAMs to reduce block RAM count in certain configurations.	bool	false	true false
complex_mult_type	Defines the types of multiplier to use for complex multiplications.	unsigned	use_mults_resources	use_luts use_mults_resources use_mults_performance
butterfly_type	Defines the implementation used for the FFT butterfly.	unsigned	use_luts	use_luts use_xtremedsp_slices



IMPORTANT! When specifying parameter values which are not integer or boolean, the HLS FFT namespace should be used. For example, the possible values for parameter `butterfly_type` in the following table are `use_luts` and `use_xtremedsp_slices`. The values used in the C program should be `butterfly_type = hls::ip_fft::use_luts` and `butterfly_type = hls::ip_fft::use_xtremedsp_slices`.

FFT Runtime Configuration and Status

The FFT supports runtime configuration and runtime status monitoring through the configuration and status ports. These ports are defined as arguments to the FFT function, shown here as variables `fft_status1` and `fft_config1`:

```
hls::fft<param1> (xn1, xk1, &fft_status1, &fft_config1);
```

The runtime configuration and status can be accessed using the predefined structs from the FFT C library:

- `hls::ip_fft::config_t<param1>`
- `hls::ip_fft::status_t<param1>`

Note: In both cases, the struct requires the name of the static parameterization struct, shown in these examples as `param1`. Refer to the previous section for details on defining the static parameterization struct.

The runtime configuration struct allows the following actions to be performed in the C code:

- Set the FFT length, if runtime configuration is enabled
- Set the FFT direction as forward or inverse
- Set the scaling schedule

The FFT length can be set as follows:

```
typedef hls::ip_fft::config_t<param1> config_t;
config_t fft_config1;
// Set FFT length to 512 => log2(512) =>9
fft_config1.setNfft(9);
```



IMPORTANT! The length specified during runtime cannot exceed the size defined by `max_nfft` in the static configuration.

The FFT direction can be set as follows:

```
typedef hls::ip_fft::config_t<param1> config_t;
config_t fft_config1;
// Forward FFT
fft_config1.setDir(1);
// Inverse FFT
fft_config1.setDir(0);
```

The FFT scaling schedule can be set as follows:

```
typedef hls::ip_fft::config_t<param1> config_t;
config_t fft_config1;
fft_config1.setSch(0x2AB);
```

The output status port can be accessed using the pre-defined struct to determine:

- If any overflow occurred during the FFT
- The value of the block exponent

The FFT overflow mode can be checked as follows:

```
typedef hls::ip_fft::status_t<param1> status_t;
status_t fft_status1;
// Check the overflow flag
bool *ovflo = fft_status1.getOvflo();
```



IMPORTANT! After each transaction completes, check the overflow status to confirm the correct operation of the FFT.

And the block exponent value can be obtained using:

```
typedef hls::ip_fft::status_t<param1> status_t;
status_t fft_status1;
// Obtain the block exponent
unsigned int *blk_exp = fft_status1.getBlkExp();
```

Using the FFT Function with Array Interface

The FFT function with array arguments is defined in the HLS namespace and can be called as follows:

```
hls::fft<STATIC_PARAM> (
    INPUT_DATA_ARRAY,
    OUTPUT_DATA_ARRAY,
    OUTPUT_STATUS,
    INPUT_RUN_TIME_CONFIGURATION);
```

The STATIC_PARAM is the static parameterization struct that defines the static parameters for the FFT.

Both the input and output data are supplied to the function as arrays (INPUT_DATA_ARRAY and OUTPUT_DATA_ARRAY). In the final implementation, the ports on the FFT RTL block will be implemented as AXI4-Stream ports. Xilinx recommends always using the FFT function in a region using dataflow optimization (set_directive_dataflow), and to specify both arrays as streaming using the set_directive_stream command.



IMPORTANT! The FFT cannot be used in a region which is pipelined. If high-performance operation is required, pipeline the loops or functions before and after the FFT then use dataflow optimization on all loops and functions in the region.

The data types for the arrays can be float or ap_fixed.

```
typedef float data_t;
complex<data_t> in_fft[FFT_LENGTH];
complex<data_t> out_fft[FFT_LENGTH];
```

To use fixed-point data types, the Vitis HLS arbitrary precision type `ap_fixed` should be used.

```
#include "ap_fixed.h"
typedef ap_fixed<FFT_INPUT_WIDTH,1> data_in_t;
typedef ap_fixed<FFT_OUTPUT_WIDTH,FFT_OUTPUT_WIDTH-FFT_INPUT_WIDTH+1>
data_out_t;
#include <complex>
typedef hls::x_complex<data_in_t> cmplxData;
typedef hls::x_complex<data_out_t> cmplxDataOut;
```

In both cases, the FFT should be parameterized with the same correct data sizes. In the case of floating point data, the data widths will always be 32-bit and any other specified size will be considered invalid.



IMPORTANT! The input and output width of the FFT can be configured to any arbitrary value within the supported range. The variables which connect to the input and output parameters must be defined in increments of 8-bit. For example, if the output width is configured as 33-bit, the output variable must be defined as a 40-bit variable.

Multi-Channel FFT

The multichannel functionality of the FFT can be used by using two-dimensional arrays for the input and output data. In this case, the array data should be configured with the first dimension representing each channel and the second dimension representing the FFT data.

```
typedef float data_t;
static complex<data_t> in_fft[FFT_CHANNELS][FFT_LENGTH];
static complex<data_t> out_fft[FFT_CHANNELS][FFT_LENGTH];
```

The FFT core consumes and produces data as interleaved channels (for example, ch0-data0, ch1-data0, ch2-data0, etc, ch0-data1, ch1-data1, ch2-data1, etc.). Therefore, to stream the input or output arrays of the FFT using the same sequential order that the data was read or written, you must fill or empty the two-dimensional arrays for multiple channels by iterating through the channel index first, as shown in the following example:

```
cmplxData in_fft[FFT_CHANNELS][FFT_LENGTH];
cmplxData out_fft[FFT_CHANNELS][FFT_LENGTH];

// Read data into FFT Input Array
for (unsigned i = 0; i < FFT_LENGTH; i++) {
    for (unsigned j = 0; j < FFT_CHANNELS; ++j) {
        in_fft[j][i] = in.read();
    }
}

// Write data out from FFT Output Array
for (unsigned i = 0; i < FFT_LENGTH; i++) {
    for (unsigned j = 0; j < FFT_CHANNELS; ++j) {
        out.write(out_fft[j][i]);
    }
}
```

Using FFT Function with Streaming Interface

The FFT function with streaming interfaces is defined in the HLS namespace similarly to this:

```
template <typename PARAM_T>
void fft(hls::stream<complex<float or ap_fixed>> &xn_s,
          hls::stream<complex<float or ap_fixed>> &xk_s,
          hls::stream<ip_fft::status_t<CONFIG_T>> &status_s,
          hls::stream<ip_fft::config_t<CONFIG_T>> &config_s);
```

and can be called as follows:

```
hls::fft<STATIC_PARAM> (
    INPUT_DATA_STREAM,
    OUTPUT_DATA_STREAM,
    OUTPUT_STATUS_STREAM,
    INPUT_RUN_TIME_CONFIGURATION_STREAM);
```

The STATIC_PARAM is the static parameterization struct that defines the static parameters for the FFT.

All input and outputs are supplied to the function as a `hls::stream<>`. In the final implementation, the ports on the FFT RTL block will be implemented as AXI4-Stream ports. Xilinx recommends always using the FFT function in a dataflow region using `set_directive_dataflow` or `#pragma HLS dataflow`.



IMPORTANT! The FFT cannot be used in a region which is pipelined. If high-performance operation is required, pipeline the loops or functions before and after the FFT then use dataflow optimization on all loops and functions in the region.

The data types for input data and output data streams can be `float` or `ap_fixed`.

```
void fft_top(
    bool direction,
    complex<data_in_t> in[FFT_LENGTH],
    complex<data_out_t> out[FFT_LENGTH],
    bool &ovflo)
{
    #pragma HLS dataflow
    hls::stream<complex<data_in_t>> in_fft;
    hls::stream<complex<data_out_t>> out_fft;
    hls::stream<config_t> fft_config;
    hls::stream<status_t> fft_status;

    // convert inputs to hls::stream<> and generates fft_config stream
    based on input arguments
    proc_fe(direction, fft_config, in, in_fft);
    // FFT IP
    hls::fft<config1>(in_fft, out_fft, fft_status, fft_config);
    // convert fft result to outputs data type and sets output ovflo
    according to fft_status stream
    proc_be(fft_status, ovflo, out_fft, out);
}
```

To use fixed-point data types, the Vitis HLS arbitrary precision type `ap_fixed` should be used.

```
#include "ap_fixed.h"
typedef ap_fixed<FFT_INPUT_WIDTH,1> data_in_t;
typedef ap_fixed<FFT_OUTPUT_WIDTH,FFT_OUTPUT_WIDTH-FFT_INPUT_WIDTH+1>
data_out_t;
#include <complex>
typedef complex<data_in_t> cmplxData;
typedef complex<data_out_t> cmplxDataOut;
```

In both cases, the FFT should be parameterized with the same correct data sizes. In the case of floating point data, the data widths will always be 32-bit and any other specified size will be considered invalid.

Comparing Stream and Array

The following table shows the array example and stream example usage of the FFT IP.



TIP: The top functions in each example are using array interfaces for consistency, but could be changed to use `hls::stream<>` or use the STREAM pragma or directive to the same effect.

Table 47: FFT Arguments

FFT with Array Arguments	FFT with <code>hls::stream<></code> Arguments
<pre>#include "fft_top.h" void proc_fe(bool direction, config_t* config, cpxDataIn in[FFT_LENGTH], cpxDataIn out[FFT_LENGTH]) { int i; config->setDir(direction); config->setSch(0x2AB); for (i=0; i< FFT_LENGTH; i++) out[i] = in[i]; } void proc_be(status_t* status_in, bool* ovflo, cpxDataOut in[FFT_LENGTH], cpxDataOut out[FFT_LENGTH]) { int i; for (i=0; i< FFT_LENGTH; i++) out[i] = in[i]; *ovflo = status_in->getOvflo() & 0x1; } // TOP function void fft_top(bool direction, complex<data_in_t> in[FFT_LENGTH], complex<data_out_t> out[FFT_LENGTH], bool* ovflo) { #pragma HLS interface ap_hs port=direction #pragma HLS interface ap_fifo depth=1 port=ovflo #pragma HLS interface ap_fifo depth=FFT_LENGTH port=in,out #pragma HLS data_pack variable=in #pragma HLS data_pack variable=out #pragma HLS dataflow complex<data_in_t> xn[FFT_LENGTH]; complex<data_out_t> xk[FFT_LENGTH]; config_t fft_config; status_t fft_status; // convert inputs to arrays and generates fft_config // based on input arguments proc_fe(direction, &fft_config, in, xn); // FFT IP hls::fft<config1>(xn, xk, &fft_status, &fft_config); // convert fft results to outputs data type and // sets output ovflo according to fft_status proc_be(&fft_status, ovflo, xk, out); } </pre>	<pre>#include "fft_top.h" void proc_fe(bool direction, hls::stream<config_t> &config_s, cpxDataIn in[FFT_LENGTH], hls::stream<cpxDataIn> &out_s) { int i; config_t config; config.setDir(direction); config.setSch(0x2AB); config_s.write(config); for (i=0; i< FFT_LENGTH; i++) { #pragma HLS pipeline II=1 rewind out_s.write(in[i]); } } void proc_be(hls::stream<status_t> &status_in_s, bool &ovflo, hls::stream<cpxDataOut> &in_s, cpxDataOut out[FFT_LENGTH]) { int i; for (i=0; i< FFT_LENGTH; i++) { #pragma HLS pipeline II=1 rewind out[i] = in_s.read(); } status_t status_in = status_in_s.read(); ovflo = status_in.getOvflo() & 0x1; } // TOP function void fft_top(bool direction, complex<data_in_t> in[FFT_LENGTH], complex<data_out_t> out[FFT_LENGTH], bool &ovflo) { #pragma HLS dataflow hls::stream<complex<data_in_t>> xn; hls::stream<complex<data_out_t>> xk; hls::stream<config_t> fft_config; hls::stream<status_t> fft_status; // convert inputs to hls::stream<> // and generates fft_config stream based on input arguments proc_fe(direction, fft_config, in, xn); // FFT IP hls::fft<config1>(xn, xk, fft_status, fft_config); // convert fft result to outputs data type // and sets output ovflo according to fft_status stream proc_be(fft_status, ovflo, xk, out); } </pre>

FIR Filter IP Library

The Xilinx FIR IP block can be called within a C++ design using the library `hls_fir.h`. This section explains how the FIR can be configured in your C++ code.



RECOMMENDED: Xilinx highly recommends that you review the FIR Compiler LogiCORE IP Product Guide ([PG149](#)) for information on how to implement and use the features of the IP.

To use the FIR in your C++ code:

1. Include the `hls_fir.h` library in the code.
2. Set the static parameters using the predefined struct `hls::ip_fir::params_t`.
3. Call the FIR function.
4. Optionally, define a runtime input configuration to modify some parameters dynamically.

The following code examples provide a summary of how each of these steps is performed. Each step is discussed in more detail below.

First, include the FIR library in the source code. This header file resides in the include directory in the Vitis HLS installation area. This directory is automatically searched when Vitis HLS executes. There is no need to specify the path to this directory if compiling inside Vitis HLS.

```
#include "hls_fir.h"
```

Define the static parameters of the FIR. This includes such static attributes such as the input width, the coefficients, the filter rate (`single`, `decimation`, `hilbert`). The FIR library includes a parameterization struct `hls::ip_fir::params_t` which can be used to initialize all static parameters with default values.

In this example, the coefficients are defined as residing in array `coeff_vec` and the default values for the number of coefficients, the input width and the quantization mode are over-ridden using a user-defined struct `myconfig` based on the predefined struct.

```
struct myconfig : hls::ip_fir::params_t {
    static const double coeff_vec[sg_fir_srrc_coeffs_len];
    static const unsigned num_coeffs = sg_fir_srrc_coeffs_len;
    static const unsigned input_width = INPUT_WIDTH;
    static const unsigned quantization = hls::ip_fir::quantize_only;
};
```

Create an instance of the FIR function using the HLS namespace with the defined static parameters (`myconfig` in this example) and then call the function with the `run` method to execute the function. The function arguments are, in order, input data and output data.

```
static hls::FIR<param1> fir1;
fir1.run(fir_in, fir_out);
```

Optionally, a runtime input configuration can be used. In some modes of the FIR, the data on this input determines how the coefficients are used during interleaved channels or when coefficient reloading is required. This configuration can be dynamic and is therefore defined as a variable. For a complete description of which modes require this input configuration, refer to the *FIR Compiler LogiCORE IP Product Guide* ([PG149](#)).

When the runtime input configuration is used, the FIR function is called with three arguments: input data, output data and input configuration.

```
// Define the configuration type
typedef ap_uint<8> config_t;
// Define the configuration variable
config_t fir_config = 8;
// Use the configuration in the FFT
static hls::FIR<param1> fir1;
fir1.run(fir_in, fir_out, &fir_config);
```



TIP: The example above shows the use of scalar values and arrays, but the FIR function also supports the use of `hls::stream` for arguments. Refer to [Vitis HLS Introductory Examples](#) for more information.

FIR Static Parameters

The static parameters of the FIR define how the FIR IP is parameterized and specifies non-dynamic items such as the input and output widths, the number of fractional bits, the coefficient values, the interpolation and decimation rates. Most of these configurations have default values: there are no default values for the coefficients.

The `hls_fir.h` header file defines a `struct hls::ip_fir::params_t` that can be used to set the default values for most of the static parameters.



IMPORTANT! There are no defaults defined for the coefficients. Therefore, Xilinx does not recommend using the pre-defined struct to directly initialize the FIR. A new user defined struct which specifies the coefficients should always be used to perform the static parameterization.

In this example, a new user `struct my_config` is defined and with a new value for the coefficients. The coefficients are specified as residing in array `coeff_vec`. All other parameters to the FIR use the default values.

```
struct myconfig : hls::ip_fir::params_t {
    static const double coeff_vec[sg_fir_srrc_coeffs_len];
};
static hls::FIR<myconfig> fir1;
fir1.run(fir_in, fir_out);
```

Fir Struct Parameters

The following table describes the parameters used for the parametrization struct `hls::ip_fir::params_t` and lists the default values for the parameters as well as the possible values.

Table 48: FIR Struct Parameter Values

Parameter	Description	C Type	Default Value	Valid Values
input_width	Data input port width	unsigned	16	No limitation
input_fractional_bits	Number of fractional bits on the input port	unsigned	0	Limited by size of input_width
output_width	Data output port width	unsigned	24	No limitation
output_fractional_bits	Number of fractional bits on the output port	unsigned	0	Limited by size of output_width
coeff_width	Bit-width of the coefficients	unsigned	16	No limitation
coeff_fractional_bits	Number of fractional bits in the coefficients	unsigned	0	Limited by size of coeff_width
num_coeffs	Number of coefficients	bool	21	Full
coeff_sets	Number of coefficient sets	unsigned	1	1-1024
input_length	Number of samples in the input data	unsigned	21	No limitation
output_length	Number of samples in the output data	unsigned	21	No limitation
num_channels	Specify the number of channels of data to process	unsigned	1	1-1024
total_num_coeff	Total number of coefficients	unsigned	21	num_coeffs * coeff_sets
coeff_vec[total_num_coeff]	The coefficient array	double array	None	Not applicable
filter_type	The type implementation used for the filter	unsigned	single_rate	single_rate, interpolation, decimation, hilbert_filter, interpolated
rate_change	Specifies integer or fractional rate changes	unsigned	integer	integer, fixed_fractional
interp_rate	The interpolation rate	unsigned	1	1-1024
decim_rate	The decimation rate	unsigned	1	1-1024
zero_pack_factor	Number of zero coefficients used in interpolation	unsigned	1	1-8
rate_specification	Specify the rate as frequency or period	unsigned	period	frequency, period
hardware_oversampling_rate	Specify the rate of over-sampling	unsigned	1	No Limitation
sample_period	The hardware oversample period	bool	1	No Limitation
sample_frequency	The hardware oversample frequency	unsigned	0.001	No Limitation
quantization	The quantization method to be used	unsigned	integer_coefficients, quantize_only, maximize_dynamic_range	integer_coefficients, quantize_only, maximize_dynamic_range
best_precision	Enable or disable the best precision	unsigned	false	false true

Table 48: FIR Struct Parameter Values (cont'd)

Parameter	Description	C Type	Default Value	Valid Values
coeff_structure	The type of coefficient structure to be used	unsigned	non_symmetric	inferred, non_symmetric, symmetric, negative_symmetric, half_band, hilbert
output_rounding_mode	Type of rounding used on the output	unsigned	full_precision	full_precision, truncate_lsbs, non_symmetric_rounding_down, non_symmetric_rounding_up, symmetric_rounding_to_zero, symmetric_rounding_to_infinity, convergent_rounding_to_even, convergent_rounding_to_odd
filter_arch	Selects a systolic or transposed architecture	unsigned	systolic_multiply_accumulate	systolic_multiply_accumulate, transpose_multiply_accumulate
optimization_goal	Specify a speed or area goal for optimization	unsigned	area	area, speed
inter_column_pipe_length	The pipeline length required between DSP columns	unsigned	4	1-16
column_config	Specifies the number of DSP module columns	unsigned	1	Limited by number of DSP macrocells used
config_method	Specifies how the DSP module columns are configured	unsigned	single	single, by_channel
coeff_padding	Specifies if zero padding is added to the front of the filter	bool	false	false, true

 **IMPORTANT!** When specifying parameter values which are not integer or boolean, the HLS FIR namespace should be used. For example the possible values for `rate_change` are shown in the table to be `integer` and `fixed_fractional`. The values used in the C program should be `rate_change = hls::ip_fir::integer` and `rate_change = hls::ip_fir::fixed_fractional`.

Using the FIR Function with Array Interface

The FIR function is defined in the HLS namespace and can be called as follows:

```
// Create an instance of the FIR
static hls::FIR<STATIC_PARAM> fir1;
// Execute the FIR instance fir1
fir1.run(INPUT_DATA_ARRAY, OUTPUT_DATA_ARRAY);
```

The STATIC_PARAM is the static parameterization struct that defines most static parameters for the FIR.

Both the input and output data are supplied to the function as arrays (INPUT_DATA_ARRAY and OUTPUT_DATA_ARRAY). In the final implementation, these ports on the FIR IP will be implemented as AXI4-Stream ports. Xilinx recommends always using the FIR function in a region using the dataflow optimization (set_directive_dataflow), because this ensures the arrays are implemented as streaming arrays. An alternative is to specify both arrays as streaming using the set_directive_stream command.



IMPORTANT! The FIR cannot be used in a region which is pipelined. If high-performance operation is required, pipeline the loops or functions before and after the FIR then use dataflow optimization on all loops and functions in the region.

The multichannel functionality of the FIR is supported through interleaving the data in a single input and single output array.

- The size of the input array should be large enough to accommodate all samples:
`num_channels * input_length.`
- The output array size should be specified to contain all output samples: `num_channels * output_length.`

The following code example demonstrates, for two channels, how the data is interleaved. In this example, the top-level function has two channels of input data (`din_i`, `din_q`) and two channels of output data (`dout_i`, `dout_q`). Two functions, at the front-end (fe) and back-end (be) are used to correctly order the data in the FIR input array and extract it from the FIR output array.

```
void dummy_fe(din_t din_i[LENGTH], din_t din_q[LENGTH], din_t
out[FIR_LENGTH]) {
    for (unsigned i = 0; i < LENGTH; ++i) {
        out[2*i] = din_i[i];
        out[2*i + 1] = din_q[i];
    }
}
void dummy_be(dout_t in[FIR_LENGTH], dout_t dout_i[LENGTH], dout_t
dout_q[LENGTH]) {
    for(unsigned i = 0; i < LENGTH; ++i) {
        dout_i[i] = in[2*i];
        dout_q[i] = in[2*i+1];
    }
}
void fir_top(din_t din_i[LENGTH], din_t din_q[LENGTH],
            dout_t dout_i[LENGTH], dout_t dout_q[LENGTH]) {

    din_t fir_in[FIR_LENGTH];
    dout_t fir_out[FIR_LENGTH];
    static hls::FIR<myconfig> fir1;

    dummy_fe(din_i, din_q, fir_in);
    fir1.run(fir_in, fir_out);
    dummy_be(fir_out, dout_i, dout_q);
}
```

Optional FIR Runtime Configuration

In some modes of operation, the FIR requires an additional input to configure how the coefficients are used. For a complete description of which modes require this input configuration, refer to the *FIR Compiler LogiCORE IP Product Guide (PG149)*.

This input configuration can be performed in the C code using a standard ap_int.h 8-bit data type. In this example, the header file `fir_top.h` specifies the use of the FIR and `ap_fixed` libraries, defines a number of the design parameter values and then defines some fixed-point types based on these:

```
#include "ap_fixed.h"
#include "hls_fir.h"

const unsigned FIR_LENGTH    = 21;
const unsigned INPUT_WIDTH   = 16;
const unsigned INPUT_FRACTIONAL_BITS = 0;
const unsigned OUTPUT_WIDTH  = 24;
const unsigned OUTPUT_FRACTIONAL_BITS = 0;
const unsigned COEFF_WIDTH   = 16;
const unsigned COEFF_FRACTIONAL_BITS = 0;
const unsigned COEFF_NUM     = 7;
const unsigned COEFF_SETS    = 3;
const unsigned INPUT_LENGTH  = FIR_LENGTH;
const unsigned OUTPUT_LENGTH = FIR_LENGTH;
const unsigned CHAN_NUM     = 1;
typedef ap_fixed<INPUT_WIDTH, INPUT_WIDTH - INPUT_FRACTIONAL_BITS> s_data_t;
typedef ap_fixed<OUTPUT_WIDTH, OUTPUT_WIDTH - OUTPUT_FRACTIONAL_BITS>
m_data_t;
typedef ap_uint<8> config_t;
```

In the top-level code, the information in the header file is included, the static parameterization struct is created using the same constant values used to specify the bit-widths, ensuring the C code and FIR configuration match, and the coefficients are specified. At the top-level, an input configuration, defined in the header file as 8-bit data, is passed into the FIR.

```
#include "fir_top.h"

struct param1 : hls::ip_fir::params_t {
    static const double coeff_vec[total_num_coeff];
    static const unsigned input_length = INPUT_LENGTH;
    static const unsigned output_length = OUTPUT_LENGTH;
    static const unsigned num_coeffs = COEFF_NUM;
    static const unsigned coeff_sets = COEFF_SETS;
};

const double param1::coeff_vec[total_num_coeff] =
{6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,-6,6,5,-3,-4,0,6};

void dummy_fe(s_data_t in[INPUT_LENGTH], s_data_t out[INPUT_LENGTH],
              config_t* config_in, config_t* config_out)
{
    *config_out = *config_in;
    for(unsigned i = 0; i < INPUT_LENGTH; ++i)
        out[i] = in[i];
}

void dummy_be(m_data_t in[OUTPUT_LENGTH], m_data_t out[OUTPUT_LENGTH])
```

```
{  
    for(unsigned i = 0; i < OUTPUT_LENGTH; ++i)  
        out[i] = in[i];  
  
}  
  
// DUT  
void fir_top(s_data_t in[INPUT_LENGTH],  
             m_data_t out[OUTPUT_LENGTH],  
             config_t* config)  
{  
  
    s_data_t fir_in[INPUT_LENGTH];  
    m_data_t fir_out[OUTPUT_LENGTH];  
    config_t fir_config;  
    // Create struct for config  
    static hls::FIR<param1> fir1;  
  
    //=====  
// Dataflow process  
    dummy_fe(in, fir_in, config, &fir_config);  
    fir1.run(fir_in, fir_out, &fir_config);  
    dummy_be(fir_out, out);  
    //=====  
}
```

Using FIR Function with Streaming Interface

The `run()` function with streaming interfaces and without config input is defined in the HLS namespace similar to this:

```
void run(  
    hls::stream<in_data_t> &in_V,  
    hls::stream<out_data_t> &out_V);
```

With config input, it is defined similar to this:

```
void run(  
    hls::stream<in_data_t> &in_V,  
    hls::stream<out_data_t> &out_V,  
    hls::stream<config_t> &config_V);
```

The FIR function is defined in the HLS namespace and can be called as follows:

```
// Create an instance of the FIR  
static hls::FIR<STATIC_PARAM> fir1;  
// Execute the FIR instance fir1  
fir1.run(INPUT_DATA_STREAM, OUTPUT_DATA_STREAM);
```

The `STATIC_PARAM` is the static parameterization struct that defines most static parameters for the FIR. Both the input and output data are supplied to the function as `hls::stream<>`. These ports on the FIR IP will be implemented as AXI4-Stream ports.

Xilinx recommends always using the FIR function in a dataflow region using `set_directive_dataflow` or `#pragma HLS dataflow`.



IMPORTANT! The FIR cannot be used in a region which is pipelined. If high-performance operation is required, pipeline the loops or functions before and after the FIR then use dataflow optimization on all loops and functions in the region.

The multichannel functionality of the FIR is supported through interleaving the data in a single input and single output stream.

- The size of the input stream should be large enough to accommodate all samples:
`num_channels * input_length`
- The output stream size should be specified to contain all output samples: `num_channels * output_length`

The following code example demonstrates how the FIR IP function can be used.

```
template<typename data_t, int LENGTH>
void process_fe(data_t in[LENGTH], hls::stream<data_t> &out)
{
    for(unsigned i = 0; i < LENGTH; ++i)
        out.write(in[i]);
}

template<typename data_t, int LENGTH>
void process_be(hls::stream<data_t> &in, data_t out[LENGTH])
{
    for(unsigned i = 0; i < LENGTH; ++i)
        out[i] = in.read();
}

// TOP function
void fir_top(
    data_t in[FIR1_LENGTH],
    data_out_t out[FIR2_LENGTH])
{
    #pragma HLS dataflow

    hls::stream<data_t> fir1_in;
    hls::stream<data_intern_t> fir1_out;
    hls::stream<data_out_t> fir2_out;

    // Create FIR instance
    static hls::FIR<config1> fir1;
    static hls::FIR<config2> fir2;

    =====
    // Dataflow process
    process_fe<data_t, FIR1_LENGTH>(in, fir1_in);
    fir1.run(fir1_in, fir1_out);
    fir2.run(fir1_out, fir2_out);
    process_be<data_out_t, FIR2_LENGTH>(fir2_out, out);
    =====
}
```

DDS IP Library

You can use the Xilinx Direct Digital Synthesizer (DDS) IP block within a C++ design using the `hls_dds.h` library. This section explains how to configure DDS IP in your C++ code.



RECOMMENDED: Xilinx highly recommends that you review the DDS Compiler LogiCORE IP Product Guide ([PG141](#)) for information on how to implement and use the features of the IP.



IMPORTANT! The C IP implementation of the DDS IP core supports the fixed mode for the `Phase_Increment` and `Phase_Offset` parameters and supports the `none` mode for `Phase_Offset`, but it does not support `programmable` and `streaming` modes for these parameters.

To use the DDS in the C++ code:

1. Include the `hls_dds.h` library in the code.
2. Set the default parameters using the pre-defined struct `hls::ip_dds::params_t`.
3. Call the DDS function.

First, include the DDS library in the source code. This header file resides in the include directory in the Vitis HLS installation area, which is automatically searched when Vitis HLS executes.

```
#include "hls_dds.h"
```

Define the static parameters of the DDS. For example, define the phase width, clock rate, and phase and increment offsets. The DDS C library includes a parameterization struct `hls::ip_dds::params_t`, which is used to initialize all static parameters with default values. By redefining any of the values in this struct, you can customize the implementation.

The following example shows how to override the default values for the phase width, clock rate, phase offset, and the number of channels using a user-defined struct `param1`, which is based on the existing predefined struct `hls::ip_dds::params_t`:

```
struct param1 : hls::ip_dds::params_t {
    static const unsigned Phase_Width = PHASEWIDTH;
    static const double DDS_Clock_Rate = 25.0;
    static const double PINC[16];
    static const double POFF[16];
};
```

Create an instance of the DDS function using the HLS namespace with the defined static parameters (for example, `param1`). Then, call the function with the `run` method to execute the function. Following are the data and phase function arguments shown in order:

```
static hls::DDS<config1> dds1;
dds1.run(data_channel, phase_channel);
```

DDS Static Parameters

The static parameters of the DDS define how to configure the DDS, such as the clock rate, phase interval, and modes. The `hls_dds.h` header file defines an `hls::ip_dds::params_t` struct, which sets the default values for the static parameters. To use the default values, you can use the parameterization struct directly with the DDS function.

```
static hls::DDS< hls::ip_dds::params_t > dds1;
dds1.run(data_channel, phase_channel);
```

The following table describes the parameters for the `hls::ip_dds::params_t` parameterization struct.



RECOMMENDED: Xilinx highly recommends that you review the *DDS Compiler LogiCORE IP Product Guide* ([PG141](#)) for details on the parameters and values.

Table 49: DDS Struct Parameters

Parameter	Description
<code>DDS_Clock_Rate</code>	Specifies the clock rate for the DDS output.
<code>Channels</code>	Specifies the number of channels. The DDS and phase generator can support up to 16 channels. The channels are time-multiplexed, which reduces the effective clock frequency per channel.
<code>Mode_of_Operation</code>	Specifies one of the following operation modes: Standard mode for use when the accumulated phase can be truncated before it is used to access the SIN/COS LUT. Rasterized mode for use when the desired frequencies and system clock are related by a rational fraction.
<code>Modulus</code>	Describes the relationship between the system clock frequency and the desired frequencies. Use this parameter in rasterized mode only.
<code>Spurious_Free_Dynamic_Range</code>	Specifies the targeted purity of the tone produced by the DDS.
<code>Frequency_Resolution</code>	Specifies the minimum frequency resolution in Hz and determines the Phase Width used by the phase accumulator, including associated phase increment (PINC) and phase offset (POFF) values.
<code>Noise_Shaping</code>	Controls whether to use phase truncation, dithering, or Taylor series correction.
<code>Phase_Width</code>	Sets the width of the following: <ul style="list-style-type: none">• PHASE_OUT field within <code>m_axis_phase_tdata</code>• Phase field within <code>s_axis_phase_tdata</code> when the DDS is configured to be a SIN/COS LUT only• Phase accumulator• Associated phase increment and offset registers• Phase field in <code>s_axis_config_tdata</code> For rasterized mode, the phase width is fixed as the number of bits required to describe the valid input range [0, Modulus-1], that is, $\log_2 (\text{Modulus}-1)$ rounded up.
<code>Output_Width</code>	Sets the width of SINE and COSINE fields within <code>m_axis_data_tdata</code> . The SFDR provided by this parameter depends on the selected Noise Shaping option.

Table 49: DDS Struct Parameters (cont'd)

Parameter	Description
Phase_Increment	Selects the phase increment value.
Phase_Offset	Selects the phase offset value.
Output_Selection	Sets the output selection to SINE , COSINE , or both in the <code>m_axis_data_tdata</code> bus.
Negative_Sine	Negates the SINE field at runtime.
Negative_Cosine	Negates the COSINE field at runtime.
Amplitude_Mode	Sets the amplitude to full range or unit circle.
Memory_Type	Controls the implementation of the SIN/COS LUT.
Optimization_Goal	Controls whether the implementation decisions target highest speed or lowest resource.
DSP48_Use	Controls the implementation of the phase accumulator and addition stages for phase offset, dither noise addition, or both.
Latency_Configuration	Sets the latency of the core to the optimum value based upon the Optimization Goal.
Latency	Specifies the manual latency value.
Output_Form	Sets the output form to two's complement or to sign and magnitude. In general, the output of SINE and COSINE is in two's complement form. However, when quadrant symmetry is used, the output form can be changed to sign and magnitude.
PINC[XIP_DDS_CHANNELS_MAX]	Sets the values for the phase increment for each output channel.
POFF[XIP_DDS_CHANNELS_MAX]	Sets the values for the phase offset for each output channel.

DDS Struct Parameter Values

The following table shows the possible values for the `hls::ip_dds::params_t` parameterization struct parameters.

Table 50: DDS Struct Parameter Values

Parameter	C Type	Default Value	Valid Values
DDS_Clock_Rate	double	20.0	Any double value
Channels	unsigned	1	1 to 16
Mode_of_Operation	unsigned	XIP_DDS_MOO_CONVENTIONAL	XIP_DDS_MOO_CONVENTIONAL truncates the accumulated phase. XIP_DDS_MOO_RASTERIZED selects rasterized mode.
Modulus	unsigned	200	129 to 256
Spurious_Free_Dynamic_Range	double	20.0	18.0 to 150.0
Frequency_Resolution	double	10.0	0.000000001 to 125000000

Table 50: DDS Struct Parameter Values (cont'd)

Parameter	C Type	Default Value	Valid Values
Noise_Shaping	unsigned	XIP_DDS_NS_NONE	XIP_DDS_NS_NONE produces phase truncation DDS. XIP_DDS_NS_DITHER uses phase dither to improve SFDR at the expense of increased noise floor. XIP_DDS_NS_TAYLOR interpolates sine/cosine values using the otherwise discarded bits from phase truncation XIP_DDS_NS_AUTO automatically determines noise-shaping.
Phase_Width	unsigned	16	Must be an integer multiple of 8
Output_Width	unsigned	16	Must be an integer multiple of 8
Phase_Increment	unsigned	XIP_DDS_PINCPOFF_FIXED	XIP_DDS_PINCPOFF_FIXED fixes PINC at generation time, and PINC cannot be changed at runtime. This is the only value supported.
Phase_Offset	unsigned	XIP_DDS_PINCPOFF_NONE	XIP_DDS_PINCPOFF_NONE does not generate phase offset. XIP_DDS_PINCPOFF_FIXED fixes POFF at generation time, and POFF cannot be changed at runtime.
Output_Selection	unsigned	XIP_DDS_OUT_SIN_AND_COS	XIP_DDS_OUT_SIN_ONLY produces sine output only. XIP_DDS_OUT_COS_ONLY produces cosine output only. XIP_DDS_OUT_SIN_AND_COS produces both sin and cosine output.
Negative_Sine	unsigned	XIP_DDS_ABSENT	XIP_DDS_ABSENT produces standard sine wave. XIP_DDS_PRESENT negates sine wave.
Negative_Cosine	bool	XIP_DDS_ABSENT	XIP_DDS_ABSENT produces standard sine wave. XIP_DDS_PRESENT negates sine wave.
Amplitude_Mode	unsigned	XIP_DDS_FULL_RANGE	XIP_DDS_FULL_RANGE normalizes amplitude to the output width with the binary point in the first place. For example, an 8-bit output has a binary amplitude of 10000000 - 10 giving values between 01111110 and 11111110, which corresponds to just less than 1 and just more than -1, respectively. XIP_DDS_UNIT_CIRCLE normalizes amplitude to half full range, that is, values range from 01000 .. (+0.5). to 110000 .. (-0.5).

Table 50: DDS Struct Parameter Values (cont'd)

Parameter	C Type	Default Value	Valid Values
Memory_Type	unsigned	XIP_DDS_MEM_AUTO	XIP_DDS_MEM_AUTO selects distributed ROM for small cases where the table can be contained in a single layer of memory and selects block ROM for larger cases. XIP_DDS_MEM_BLOCK always uses block RAM. XIP_DDS_MEM_DIST always uses distributed RAM.
Optimization_Goal	unsigned	XIP_DDS_OPTGOAL_AUTO	XIP_DDS_OPTGOAL_AUTO automatically selects the optimization goal. XIP_DDS_OPTGOAL_AREA optimizes for area. XIP_DDS_OPTGOAL_SPEED optimizes for performance.
DSP48_Use	unsigned	XIP_DDS_DSP_MIN	XIP_DDS_DSP_MIN implements the phase accumulator and the stages for phase offset, dither noise addition, or both in FPGA logic. XIP_DDS_DSP_MAX implements the phase accumulator and the phase offset, dither noise addition, or both using DSP slices. In the case of single channel, the DSP slice can also provide the register to store programmable phase increment, phase offset, or both and thereby, save further fabric resources.
Latency_Configuration	unsigned	XIP_DDS_LATENCY_AUTO	XIP_DDS_LATENCY_AUTO automatically determines the latency. XIP_DDS_LATENCY_MANUAL manually specifies the latency using the Latency option.
Latency	unsigned	5	Any value
Output_Form	unsigned	XIP_DDS_OUTPUT_TWOS	XIP_DDS_OUTPUT_TWOS outputs two's complement. XIP_DDS_OUTPUT_SIGN_MAG outputs signed magnitude.
PINC[XIP_DDS_CHANNELS_MAX]	unsigned array	{0}	Any value for the phase increment for each channel
POFF[XIP_DDS_CHANNELS_MAX]	unsigned array	{0}	Any value for the phase offset for each channel

SRL IP Library

C code is written to satisfy several different requirements: reuse, readability, and performance. Until now, it is unlikely that the C code was written to result in the most ideal hardware after high-level synthesis.

Like the requirements for reuse, readability, and performance, certain coding techniques or pre-defined constructs can ensure that the synthesis output results in more optimal hardware or to better model hardware in C for easier validation of the algorithm.

Mapping Directly into SRL Resources

Many C algorithms sequentially shift data through arrays. They add a new value to the start of the array, shift the existing data through array, and drop the oldest data value. This operation is implemented in hardware as a shift register.

This most common way to implement a shift register from C into hardware is to completely partition the array into individual elements, and allow the data dependencies between the elements in the RTL to imply a shift register.

Logic synthesis typically implements the RTL shift register into a Xilinx SRL resource, which efficiently implements shift registers. The issue is that sometimes logic synthesis does not implement the RTL shift register using an SRL component:

- When data is accessed in the middle of the shift register, logic synthesis cannot directly infer an SRL.
- Sometimes, even when the SRL is ideal, logic synthesis may implement the shift-resister in flip-flops, due to other factors. (Logic synthesis is also a complex process).

Vitis HLS provides a C++ class (`ap_shift_reg`) to ensure that the shift register defined in the C code is always implemented using an SRL resource. The `ap_shift_reg` class has two methods to perform the various read and write accesses supported by an SRL component.

Read from the Shifter

The read method allows a specified location to be read from the shifter register.

The `ap_shift_reg.h` header file that defines the `ap_shift_reg` class is also included with Vitis HLS as a standalone package. You have the right to use it in your own source code. The package `xilinx_hls_lib_<release_number>.tgz` is located in the `include` directory in the Vitis HLS installation area.

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1;

// Read location 2 of Sreg into var1
var1 = Sreg.read(2);
```

Read, Write, and Shift Data

A `shift` method allows a read, write, and shift operation to be performed.

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1;

// Read location 3 of Sreg into var1
// THEN shift all values up one and load In1 into location 0
var1 = Sreg.shift(In1,3);
```

Read, Write, and Enable-Shift

The `shift` method also supports an enabled input, allowing the shift process to be controlled and enabled by a variable.

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1, In1;
bool En;
```

```
// Read location 3 of Sreg into var1
// THEN if En=1
// Shift all values up one and load In1 into location 0
var1 = Sreg.shift(In1,3,En);
```

When using the ap_shift_reg class, Vitis HLS creates a unique RTL component for each shifter. When logic synthesis is performed, this component is synthesized into an SRL resource.

Vitis HLS Migration Guide

This section contains the following chapters:

- [Migrating to Vitis HLS](#)
- [Unsupported Features](#)
- [Deprecated and Unsupported Features](#)

Migrating to Vitis HLS

Vivado® HLS was the older generation HLS solution that has its last official release in 2020.1. Vitis™ HLS represents the next-generation HLS solution from AMD® and has several improvements over the older Vivado HLS technology. First and foremost, Vitis™ HLS has a new compiler that uses an updated version of the LLVM compiler standard and supports C/C++ 11/14 for compilation/simulation. When migrating a kernel module or IP implemented with one version of Vivado HLS, it is essential to understand the difference between the versions of HLS, and the impact that these differences have on the design.

The key differences between these version of HLS versions can be categorized as follows:

- Key Behavioral Differences
- Deprecated Commands
- Unsupported Features

In addition, Vitis HLS now supports the following new and stricter syntax checking for both source code and pragma usage:

- Syntax Checker: Vitis HLS will error out on unconnected ports (either in dataflow regions or during RTL generation).
- Pragma Conflict Checker: New warnings and errors are reported for pragma conflicts and/or typos in pragma options.
- New and Improved Messages when user-specified pragmas are ignored by the tool.

Key Behavioral Differences

This section highlights some of the key behavioral differences between HLS and Vitis™ HLS. These key differences can be the source of big quality of results (QoR) differences and therefore, it is necessary to understand these key differences when comparing QoR between different versions of HLS tools. AMD recommends reviewing this section before using Vitis HLS.



TIP: Due to the behavioral differences between Vitis HLS and Vivado HLS, you might need to differentiate your code for use in the Vitis tool. To enable the same source code to be used in both tools, Vitis HLS supports the `--VITIS_HLS--` predefined macro to encapsulate source code written specifically for use in that tool. Use `#if defined(--VITIS_HLS--) type` pre-processor declarations to encapsulate tool specific code.

Default User Control Settings

The default global option configures the solution for either Vitis kernel acceleration development flow or Vivado IP development flow.

```
open_solution -flow_target [vitis | vivado]
```

This global option is replacing the previous config option (`config_sdx`).

Vivado IP Development Flow

Configures the solution to run in support of the Vivado IP generation flow, requiring strict use of pragmas and directives, and exporting the results as Vivado IP. The command to set up the project solution for the Vivado IP flow is:

```
open_solution -flow_target vivado
```

The table below shows the original default settings of command options in the Vivado HLS tool, and the new defaults applied in the Vitis HLS tool.

Table 51: Default Control Settings

Default Control Settings	Vivado HLS	Vitis HLS
<code>config_compile -pipeline_loops</code>	0	64
<code>config_export -vivado_optimization_level</code>	2	0
<code>set_clock_uncertainty</code>	12.5%	27%
<code>config_interface -m_axi_alignment_byte_size</code>	N/A	0
<code>config_interface -m_axi_max_widen_bitwidth</code>	N/A	0
<code>config_export -vivado_phys_opt</code>	place	none
<code>config_interface -m_axi_addr64</code>	false	true
<code>config_schedule -enable_dsp_full_reg</code>	false	true
<code>config_rtl -module_auto_prefix</code>	false	true
interface pragma defaults	ip mode	ip mode

Vitis Application Acceleration Development Flow (Kernel Mode)

Configures the solution to run in support of the Vivado IP generation flow, requiring strict use of pragmas and directives, and exporting the results as Vivado IP. The command to set up the project solution for the Vivado IP flow is:

```
open_solution -flow_target vivado
```

The table below shows the original default settings of command options in the Vivado HLS tool, and the new defaults applied in the Vitis HLS tool.

Table 52: Default Control Settings

Default Control Settings	Vivado HLS	Vitis HLS
config_compile -pipeline_loops	0	64
config_export -vivado_optimization_level	2	0
set_clock_uncertainty	12.5%	27%
config_interface -m_axi_alignment_byte_size	N/A	0
config_interface -m_axi_max_widen_bitwidth	N/A	0
config_export -vivado_phys_opt	place	none
config_interface -m_axi_addr64	false	true
config_schedule -enable_dsp_full_reg	false	true
config_rtl -module_auto_prefix	false	true
interface pragma defaults	ip mode	ip mode

Default Interfaces

Both Vivado® HLS and Vitis™ HLS allow the user to customize the interfaces to the top-level function using pragma/directions. Vitis HLS additionally supports the ability to automatically infer the right interface. The type of interfaces that Vitis HLS creates depends on:

- The data type of the C arguments in the top-level function.
- The target flow.
- The default interface mode.
- Any user-specified **INTERFACE** pragmas or directives.

This is fully described in the [Defining Interfaces](#) section of the user guide (UG1399) and should be reviewed before proceeding further.

Interface Bundle Rules

By default, Vitis™ HLS groups function arguments with compatible options into a single `m_axi`/`s_axilite` interface adapter. Bundling ports into a single interface helps save device resources by eliminating AXI4 logic, which can be necessary when working in congested designs. However, a single interface bundle can limit the performance of the kernel because all the memory transfers have to go through a single interface. For example, the `m_axi` interface has independent READ and WRITE channels, so a single interface can read and write simultaneously, though only at one location. Using multiple bundles lets you increase the bandwidth and throughput of the kernel by creating multiple interfaces to connect to memory banks.

Bundle rules for AXI interfaces are fully described in [M_AXI Bundles/S_AXILITE Bundles](#) section in the user guide (UG1399) and should be reviewed before proceeding further.

Memory Property on Interface

The `storage_type` option on the interface pragma or directive lets the user explicitly define which type of RAM is used, and which RAM ports are created (single-port or dual-port). If no `storage_type` is specified, Vitis HLS uses:

- A single-port RAM by default.
- A dual-port RAM if it reduces the initiation interval or latency.

For the Vivado flow, the user can specify a RAM storage type on the specified interface, replacing the deprecated `RESOURCE` pragma with the `storage_type`.

```
#pragma HLS INTERFACE bram port = in1 storage_type=RAM_2P
#pragma HLS INTERFACE bram port = out storage_type=RAM_1P latency=3
```

AXI4-Stream Interfaces with Side-Channels

Side-channels are optional signals, which are part of the AXI4-Stream standard. The side-channel signals can be directly referenced and controlled in the C/C++ code using a struct, provided that the member elements of the struct match the names of the AXI4-Stream side-channel signals. The AXI4-Stream side-channel signals are considered data signals and are registered whenever `TDATA` is registered. There are significant differences in how this is handled in Vivado HLS vs Vitis HLS. The [AXI4-Stream Interfaces with Side-Channels](#) section in the user guide (UG1399) details the limitations and restrictions in Vitis HLS.

Memory Model

One key difference between Vivado® HLS and Vitis™ HLS is how the internal memory model is implemented. The memory model defines the way data is arranged and accessed in computer memory. It consists of two separate but related issues: data alignment and data structure padding. Vitis HLS uses a different memory model from Vivado HLS and the key differences are as follows:

- Vitis HLS follows the GCC Compiler standard for data alignment
- Vitis HLS differs significantly from Vivado HLS in how it aggregates and disaggregates structs/classes in the interface
 - Vivado HLS used to disaggregate structs in the interface by default. Vitis HLS instead keeps structs aggregated. You can use the AGGREGATE/[pragma HLS disaggregate](#) pragma/directive to match the Vivado HLS behavior.
 - The exception is when there is a `hls::stream` object in the struct or an array inside the struct is partitioned, Vitis HLS will disaggregate the struct in both of these cases.

- Structs in the code, both internal and global variables, are disaggregated by default and decomposed into their member elements, as described in [Structs](#). The number and type of elements created are determined by the contents of the struct itself. Arrays of structs are implemented as multiple arrays, with a separate array for each member of the struct.
- Vitis HLS also differs in how data alignment is handled for interface protocols (such as AXI).
- Changes also include deprecated pragmas such as DATA_PACK.

These changes are fully described in the [Vitis HLS Memory Layout Model](#) section of the user guide (UG1399) and should be reviewed before proceeding further.

Unconnected Ports

Vivado HLS accepted unconnected output signals for streamed single and multidimensional arrays.

Vitis HLS allows the following types of unconnected outputs:

- Unconnected output scalars
- Unconnected output single and multidimensional arrays implemented as PIPOs when used inside a dataflow region

Vitis HLS does not allow:

- Unconnected output streams
- Unconnected output streamed single or multidimensional arrays

Global Variables on the Interface

Both Vivado HLS and Vitis™ HLS allow global variables to be freely used in the code and are fully synthesizable. However, while Vivado HLS supported exporting global variables in the top-level function interface, Vitis™ HLS does not.

Global variables can not be used as arguments to the top-level function, or exposed as ports on the RTL interface in Vitis™ HLS. Variables needed on the interface of the top-level function must be explicitly declared in the interface.

Behavior Changes to Module Names and Module Prefix

In Vivado HLS, when `config_rtl -module_auto_prefix` was enabled the top RTL module would have its name prefixed with its own name. Since Vitis HLS 2020.1, this auto prefix will only be applied to sub-modules.

There is no change to the `-module_prefix` behavior. If this option is used, the specified prefix value will be prepended to all modules including the top module. The `-module_prefix` option also still takes precedence over `-module_auto_prefix`.

```
# vivado HLS 2020.1 generated module names (top module is "top")
top_top.v
top_submodule1.v
top_submodule2.v

# Vitis HLS 2020.1 generated module names
top.v           <-- top module no longer has prefix
top_submodule1.v
top_submodule2.v
```

Updated Memory (RAM/ROM) Module Name and RTL File Name

The new naming of the module is appended with the type of memory as shown below.

Table 53: Module and RTL File Names

OLD Module and RTL File Names	NEW Module and RTL File Names
ncp_encoder_func_parbits_memcore_ram	ncp_encoder_func_parbits_RAM_1P_LUTRAM_1R1W
test_A_V_ROM	test_A_V_ROM_1P_BRAM_1R, test_A_V_ROM_1P_BRAM_1v

Behavior differences in Inlining of functions

Automatic inlining of functions in Vivado HLS was governed by the size of the function. However, in Vitis HLS the automatic inlining optimization is driven by user pragma preferences or by a cost model. For example, consider the case of a loop that is being pipelined, and calls to sub-functions are being made inside this loop body. In Vivado HLS, these sub-functions may have been automatically inlined due to their small size. In Vitis HLS they will only be inlined if the user requests a pipeline of `II=1`. So the automatic inlining will only occur if the Vitis HLS scheduler determines that there is a benefit to inlining these sub-functions to achieve `II=1` in this pipelined loop.

Due to this change you may see differences in the QoR achieved by Vitis HLS and Vivado HLS. You can overcome these differences in Vitis HLS by manually inlining the sub-functions to match the prior behavior of Vivado HLS.

Dataflow

Support of std::complex:

In Vivado HLS, std::complex data type could not be used directly inside the DATAFLOW, because of multiple writers issue. This multiple reader and writer issue is coming from the std class constructor being called to initialize the value. When this variable is also used inside the dataflow as a channel, it leads to the above issue. However, Vitis supports the use of `std::complex` with support of an attribute `no_ctor` as shown below.

```
// Nothing to do here.
void proc_1(std::complex<float> (&buffer)[50], const std::complex<float>
*in);
void proc_2(hls::stream<std::complex<float>> &fifo, const
std::complex<float> (&buffer)[50], std::complex<float> &acc);
void proc_3(std::complex<float> *out, hls::stream<std::complex<float>>
&fifo, const std::complex<float> acc);

void top(std::complex<float> *out, const std::complex<float> *in) {
#pragma HLS DATAFLOW
    std::complex<float> acc __attribute__((no_ctor)); // here
    std::complex<float> buffer[50] __attribute__((no_ctor)); // here
    hls::stream<std::complex<float>, 5> fifo; // no need here (hls::stream
has it internally)

    proc_1(buffer, in);
    proc_2(fifo, buffer, acc);
    proc_3(out, fifo, acc);
}
```

Deprecated and Unsupported Features

Vitis™ HLS has deprecated a number of Vivado® HLS commands. These deprecated commands will be discontinued in a future release, and are not recommended for use; these are listed in the table below.

Table 54: Vivado HLS Commands Deprecated in Vitis HLS

Type	Command	Option	Vitis HLS	Details
config	config_interface	-m_axi_max_data_size	Deprecated	
config	config_interface	-m_axi_min_data_size	Deprecated	
config	config_interface	-m_axi_alignment_byte_size	Deprecated	
config	config_interface	-m_axi_offset=slave	Unsupported	This is now handled through a combination of -m_axi_offset=direct and -default_slave_interface=s_axilite
config	config_interface	-expose_global	Unsupported	Global variables are not supported in Vitis HLS for exposure as top-level ports in the IP or kernel.
config	config_interface	-trim_dangling_port	Unsupported	
config	config_array_partition	-auto_promotion_threshold	Deprecated	
config	config_array_partition	-auto_partition_threshold	Deprecated	Has been renamed to -complete_threshold
config	config_array_partition	-scalarize_all	Unsupported	
config	config_array_partition	-throughput_driven	Unsupported	

Table 54: Vivado HLS Commands Deprecated in Vitis HLS (cont'd)

Type	Command	Option	Vitis HLS	Details
config	config_array_partition	-maximum_size	Unsupported	
config	config_array_partition	-include_extern_globals	Unsupported	
config	config_array_partition	-include_ports	Unsupported	
config	config_schedule	All options but -enable_dsp_fillreg	Deprecated	
config	config_bind	* (all options)	Deprecated	
config	config_rtl	-encoding	Deprecated	
config	config_sdx	* (all options)	Deprecated	
config	config_flow	* (all options)	Deprecated	
config	config_dataflow	-disable_start_propagation	Deprecated	
config	config_rtl	-auto_prefix	Deprecated	Replaced by config_rtl -module_prefix.
config	config_rtl	-prefix	Deprecated	Replaced by config_rtl -module_prefix.
config	config_rtl	-m_axi_conservative_mode	Deprecated	Use config_interface -m_axi_conservative_mode
directive/pragma	set_directive_pipeline	-enable_flush	Deprecated	
directive/pragma	CLOCK	*	Unsupported	
directive/pragma	DATA_PACK	*	Unsupported	Use AGGREGATE pragma or directive, and __attribute__(packed(X)) if needed.
directive/pragma	INLINE	-region	Deprecated	
directive/pragma	INTERFACE	-mode ap_bus	Unsupported	Use m_axi instead.
directive/pragma	ARRAY_MAP	*	Unsupported	
directive/pragma	RESOURCE	*	Deprecated	Replaced by BIND_OP and BIND_STORAGE pragmas and directives. Use INTERFACE pragma or directive with storage_type option for top function arguments.

Table 54: Vivado HLS Commands Deprecated in Vitis HLS (cont'd)

Type	Command	Option	Vitis HLS	Details
directive/pragma	SHARED	*	Deprecated	The SHARED pragma or directive has been moved to the type=shared option of the STREAM pragma or directive.
directive/pragma	STREAM	-dim	Unsupported	
directive/pragma	STREAM	-off	Deprecated	STREAM off has become STREAM type=pipo
project	csim_design	-clang_sanitizer	Add/Rename	
project	export_design	-use_netlist	Deprecated	Replaced by: export_design -format ip_catalog
project	export_design	-xo	Deprecated	Replaced by: export_design -format xo
project	add_files		Unsupported	System-C files are not supported by Vitis HLS.
config	config_export	- disable_deadlock_detection	Deprecated	Replaced by: config_export -deadlock_detection_n_sim

Notes:

1. Deprecated: A warning message for discontinuity of the pragma in a future release will be issued.
2. Unsupported: Vitis HLS errors out with a valid message.
3. *: All the options in the command.

The following libraries are deprecated.

Table 55: Deprecated Libraries

Library	API	Deprecated	Suggested Replacement / Action
Linear Algebra Lib hls_linear_algebra.h hls/hls_axi_io.h hls/ hls_linear_algebra_io.h hls/linear_algebra --- deprecated -- x_complex_back_substitute.h -- x_complex_cholesky.h -- x_complex_cholesky_inverse.h -- x_complex_matrix_multiply.h -- x_complex_matrix_tb_utils.h -- x_complex_matrix_utils.h -- x_complex_qr_inverse.h -- x_complex_qrf.h `-- x_complex_svd.h -- hls_back_substitute.h -- hls_cholesky.h -- hls_cholesky_inverse.h -- hls_matrix_multiply.h -- hls_qr_inverse.h -- hls_qrf.h -- hls_svd.h `-- utils -- std_complex_utils.h -- x_hls_complex.h -- x_hls_matrix_tb_utils.h `-- x_hls_matrix_utils.h	cholesky float, ap_fixed, x_complex<float>, x_complex<ap_fixed> cholesky_inverse float, ap_fixed, x_complex<float>, x_complex<ap_fixed> matrix_multiply float, ap_fixed, x_complex<float>, x_complex<ap_fixed> qrf float, x_complex<float> qr_inverse float, x_complex<float> svd float, x_complex<float>	deprecated	Vitis Solver potrf float, double
			Vitis Solver pomatrixinverse float, double
			Vitis Solver geqrf float, double
			Vitis Solver gesvdj float, double

Table 55: Deprecated Libraries (cont'd)

Library	API	Deprecated	Suggested Replacement / Action
DSP Lib	atan2 input: std::complex<ap_fixed> output: ap_ufixed	deprecated	HLS Math atan2 ap_fixed/ap_ufixed/float/ double
	sqrt unsigned binary fraction with 1 bit integer, unsigned int		HLS Math sqrt ap_fixed/ap_ufixed/float/ double
	awgn (Additive white Gaussian noise) input: ap_ufixed output: ap_int		
	cmpy (complex number multiply) input: std::complex< ap_fixed > output: std::complex< ap_fixed >		
	convolution_encoder (for data transferring in channel with error correcting, used in pair with viterbi decoder) input: ap_uint output: ap_uint		
	viterbi_decoder (for data transferring in channel with error correcting, used in pair with convolution encoder) input: ap_uint output: ap_uint		
	nco (Numerically-Controlled Oscillator) input: ap_uint output: std::complex< ap_int >		

Unsupported Features

The following features are not supported in this release.



IMPORTANT! HLS will either issue a warning or error for all the unsupported features mentioned in this section.

Assertions

The `assert` macro in C/C++ is not supported in Vitis HLS.

Usage of assertions can have unintended consequences that are not obvious to the user, causing bad logic to be created. They can also prevent compiler optimizations depending on the complexity of the code.

Pragmas

- Pragma DEPENDENCE on an argument that also has an `m_axi INTERFACE` pragma specifying a bundle with two or more ports is not supported.

```
void top(int *a, int *b) { // both a and b are bundled to m_axi port gmem
    #pragma HLS interface m_axi port=a offset=slave bundle=gmem
    #pragma HLS interface m_axi port=b offset=slave bundle=gmem
    #pragma HLS dependence variable=a false
}
```

- Pragma INTERFACE no longer supports the `ap_bus` mode that was supported in Vivado HLS. You should use the `m_axi` interface instead.

Top-Level Function Argument



IMPORTANT! Vitis™ HLS does not support the INTERFACE pragma inside sub-functions.

Top-level argument with C data type of:

- enum or any use of enum (struct, array pointer of enum)
 - ap_int<N> (where N is 1-32768)
 - _Complex
 - _Half, __fp16
-

HLS Video Library

The `hls_video.h` for video utilities and functions has been deprecated and replaced by the Vitis vision library. See the [Migrating HLS Video Library to Vitis vision](#) on GitHub for more details.

C Arbitrary Precision Types

Vitis HLS does not support C arbitrary precision types. Xilinx recommends using C++ types with arbitrary precision.

In addition, C++ arbitrary precision types in Vitis HLS are limited to a maximum width of 4096 bits, instead of 32K bits supported by Vivado HLS.

C Constructs

- Pointer cast are not supported.
- Virtual functions are not supported.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help**→**Documentation and Tutorials**.
- On Windows, select **Start**→**All Programs**→**Xilinx Design Tools**→**DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on DocNav, see the [Documentation Navigator](#) page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *Introduction to FPGA Design with Vivado High-Level Synthesis* ([UG998](#))
2. *Vivado Design Suite Tutorial: High-Level Synthesis* ([UG871](#))
3. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* ([UG973](#))
4. *Floating-Point Design with Vivado HLS* ([XAPP599](#))
5. *Floating-Point Operator LogiCORE IP Product Guide* ([PG060](#))
6. *Fast Fourier Transform LogiCORE IP Product Guide* ([PG109](#))
7. *FIR Compiler LogiCORE IP Product Guide* ([PG149](#))
8. *DDS Compiler LogiCORE IP Product Guide* ([PG141](#))
9. *Vivado Design Suite: AXI Reference Guide* ([UG1037](#))
10. *Accelerating OpenCV Applications with Zynq-7000 SoC Using Vivado HLS Video Libraries* ([XAPP1167](#))
11. *UltraFast Vivado HLS Methodology Guide* ([UG1197](#))
12. Option Summary page on the GCC website (gcc.gnu.org/onlinedocs/gcc/Option-Summary.html)
13. Accellera website (<http://www.accellera.org/>)
14. AWGN page on the MathWorks website (<http://www.mathworks.com/help/comm/ug/awgn-channel.html>)
15. *Vivado® Design Suite Documentation*

Revision History

The following table shows the revision history for [Section III: Using Vitis HLS](#).

Section	Revision Summary
12/07/2022 Version 2022.2	
Introduction to Vitis HLS Re-Architecting the Design Code	Updated Sections and New Section.
10/19/2022 Version 2022.2	
Impact of Struct Size on Pipelining Vitis HLS Memory Layout Model Vitis HLS Alignment Rules and Semantics Using Manual Burst	Updated Sections

Section	Revision Summary
06/07/2022 Version 2022.1	
Impact of Struct Size on Pipelining Vitis HLS Memory Layout Model Vitis HLS Alignment Rules and Semantics Using Manual Burst	Updated Sections
04/20/2022 Version 2022.1	
Entire section	No changes to this section.

The following table shows the revision history for [Section II: HLS Programmers Guide](#).

Section	Revision Summary
12/07/2022 Version 2022.2	
Mixing Data-Driven and Control-Driven Models Execution Modes of HLS Designs	Updated Sections
06/07/2022 Version 2022.1	
Entire section	Editorial Updates.
04/20/2022 Version 2022.1	
Interfaces for Vitis Kernel Flow Interfaces for Vivado IP Flow S_AXILITE and Port-Level Protocols	Updated Sections
04/20/2022 Version 2022.1	
FIFO Interfaces Arbitrary Precision (AP) Data Types Floats and Doubles Composite Data Types Updated Memory (RAM/ROM) Module Name and RTL File Name Multi-Access Pointers on the Interface Defining Interfaces AXI4-Lite Interface	Updated Sections

The following table shows the revision history for [Section IV: Vitis HLS Command Reference](#).

Section	Revision Summary
12/07/2022 Version 2022.2	
Entire section	Updated Sections
06/07/2022 Version 2022.1	
Entire section	Updated Sections

Section	Revision Summary
04/20/2022 Version 2022.1	
set_directive_inline set_directive_interface pragma HLS inline	Updated Sections
04/20/2022 Version 2022.1	
cosim_design get_project get_solution list_part config_array_partition config_compile config_export config_interface config_op config_storage set_directive_aggregate set_directive_array_partition set_directive_array_reshape set_directive_bind_op set_directive_bind_storage set_directive_dependence set_directive_disaggregate set_directive_function_instantiate set_directive_interface set_directive_pipeline set_directive_stream pragma HLS aggregate pragma HLS alias pragma HLS array_partition pragma HLS array_reshape pragma HLS bind_op pragma HLS bind_storage pragma HLS dependence pragma HLS disaggregate pragma HLS interface pragma HLS pipeline pragma HLS protocol pragma HLS stream	Minor Updates

The following table shows the revision history for [Section VI: Vitis HLS Libraries Reference](#).

Section	Revision Summary
12/07/2022 Version 2022.2	
Entire section	No changes to this section.

Section	Revision Summary
06/07/2022 Version 2022.1	
Entire section	No changes to this section.
04/20/2022 Version 2022.1	
Entire section	No changes to this section.
04/20/2022 Version 2022.1	
Vitis HLS Libraries Reference: C++ Arbitrary Precision Fixed-Point Types HLS Stream Library FFT IP Library	Minor Updates

The following table shows the revision history for [Section VII: Vitis HLS Migration Guide](#).

Section	Revision Summary
12/07/2022 Version 2022.2	
Entire section	No changes to this section.
06/07/2022 Version 2022.1	
Entire section	No changes to this section.
04/20/2022 Version 2022.1	
Deprecated and Unsupported Features	Minor Update
04/20/2022 Version 2022.1	
Migrating to Vitis HLS Key Behavioral Differences Updated Memory (RAM/ROM) Module Name and RTL File Name Interface Bundle Rules Behavior Changes to Module Names and Module Prefix Default User Control Settings Top-Level Function Argument Deprecated and Unsupported Features	Minor Updates

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including

negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2020-2022 Advanced Micro Devices, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Kria, Spartan, Versal, Vitis, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. All other trademarks are the property of their respective owners.