Vitis HLS Migration Guide

UG1391 (v2020.2) November 24, 2020

XILINX®

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
11/24/2020 Version 2020.2	
Unsupported Features updates	N/A
07/28/2020 Version 2020.1	
Release updates.	N/A
06/03/2020 Version 2020.1	
Initial release.	N/A

Table of Contents

Chapter 1

Migrating to Vitis HLS

When migrating a kernel module implemented in 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.

Key Considerations:

- Behavioral Differences
- Unsupported Features
- Deprecated Commands

HLS Behavioral Differences

Vitis™ HLS brings some fundamental changes in the way HLS synthesizes the C code, supports language constructs, and supports existing commands, pragmas, and directives. For example, the `std::complex<long double >` data type is not supported in Vitis HLS, and should not be used. These changes have implications on the application QoR. Xilinx® recommends reviewing this section before using the tool.

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 application acceleration development flow or Vivado IP development flow.

```
open_solution -flow_target [vitis | vivado]
```

This global option is replacing the old config option (`config_sdx`).

Vivado 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.

```
open_solution -flow_target vivado
```

Table 1: **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_export -vivado_optimization_level`	20	255
`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

Vitis Flow (Kernel Mode):

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`).

```
open_solution -flow_target vitis
```

Table 2: **Default Control Settings**

Default Control Settings	Vivado HLS	Vitis HLS
interface pragma defaults	ip mode	kernel mode (check default interfaces)
`config_interface -m_axi_alignment_byte_size`	N/A	64
`config_interface -m_axi_max_widen_bitwidth`	N/A	512
`config_compile -name_max_length`	256	255
`config_compile -pipeline_loops`	64	64
`set_clock_uncertainty`	27%	27%
`config_rtl -register_reset_num`	3	3

Table 2: **Default Control Settings** *(cont'd)*

Default Control Settings	Vivado HLS	Vitis HLS
`config_interface -m_axi_latency`	0	64

Default Loop II Constraint Settings

In Vivado HLS, the default loop II constraint is set to 1, but in Vitis HLS it is set to auto. For example, if the tool could not achieve default II, it will try to achieve the best II possible.

Default Interfaces

The type of the interfaces that are created by interface synthesis depends on the data type of C argument, the default interface mode, and the interface directives. In Vitis HLS, the default interface changes depending on the data types used on the C arguments and configurations. The user selection of the `open_solution -flow_target [vitis | vivado]` will dictate the default interface settings.

The following figures shows the changes when the default interface protocol is enabled.

Argument type definitions (used in below tables):

- I: Input only (can only read from arg)
- O: Output only (can only write to arg)
- IO: Input & output (can read and write to arg)
- Return: Return data output
- Block: Block-level control
- D: Default mode for each typed.

***Note*:** If an illegal interface is specified, Vitis HLS issues a warning message and implements the default interface mode.

Vitis Flow (Accelerator Mode)

If HLS is used in the Vitis flow, the tool will automatically set the following configurations.

```
open_solution -flow_target vitis
```

Table 3: **Argument Types**

Argument Type	**Scalar**		**Pointer to an Array**			**Hls::stream**
Interface Mode	**Input**	**Return**	**I**	**I/O**	**O**	**I and O**
ap_ctrl_none						
ap_ctrl_hs						
ap_ctrl_chain		D				
axis						D
m_axi			D	D	D	

Notes:

1. D = Default Interface

The AXI4-Lite slave interface directive will change the behavior of the interface pragmas as shown below.

```
config_interface -default_slave_interface s_slave
```

Table 4: **Argument Types**

Argument Type	**Scalar**		**Pointer to an Array**			**Hls::stream**
Interface Mode	**Input**	**Return**	**I**	**I/O**	**O**	**I and O**
s_axi_lite	D	D	D	D	D	

Notes:

1. D = Default Interface

***Note*:** These default interface pragma settings can be overridden by a user-specified interface pragma.

Vivado Design Flow

Vitis HLS can be used in standalone mode to create IP. The tool will run this flow by default for which it sets the following global options:

- `open_solution -flow_target vivado`
- `config_interface -default_slave_interface s_axilite`

In this case, the following default interfaces are applied.

Table 5: **Argument Types**

Argument Type	Scalar		Array			Pointer or Reference			Hls::stream
Interface Mode	**Input**	**Return**	**I**	**I/O**	**O**	**I**	**I/O**	**O**	**I and O**
ap_ctrl_none	3	1	3	3	3	3	3	3	3
ap_ctrl_hs	3	1	3	3	3	3	3	3	3
ap_ctrl_chain	3	D[1]	3	3	3	3	3	3	3
axis	1	3	1	3	1	1	3	1	1
s_axilite	1	1	1	1	1	1	1	1	3
m_axi	3	3	1	1	1	1	1	1	3
ap_none	D[1]	3	3	3	3	D[1]	1	1	3
ap_stable	1	3	3	3	3	1	3	3	3
ap_ack	1	3	3	3	3	1	1	1	3
ap_vld	1	3	3	3	3	1	1	D[1]	3
ap_ovld	3	3	3	3	3	3	D[1]	1	3
ap_hs	1	3	1	3	1	1	1	1	1
ap_memory	3	3	D[1]	D[1]	D[1]	3	3	3	3
bram	3	3	1	1	1	3	3	3	3
ap_fifo	3	3	1	3	1	1	3	1	D[1]

Notes:
1. Supported
2. D = Default Interface
3. Not Supported

Structs

Structs in the code, for instance internal and global variables, are disaggregated by default. They are decomposed into 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.

Structs in C/C++ are padded with extra bytes by the compiler for data alignment. In order to make kernel code in Vitis HLS compliant with `gcc`, structs in kernel code are padded with extra bytes. Refer to Structs for more information.

Data Layout for Arbitrary Precision Types (ap_int library)

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.

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;
};
```

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

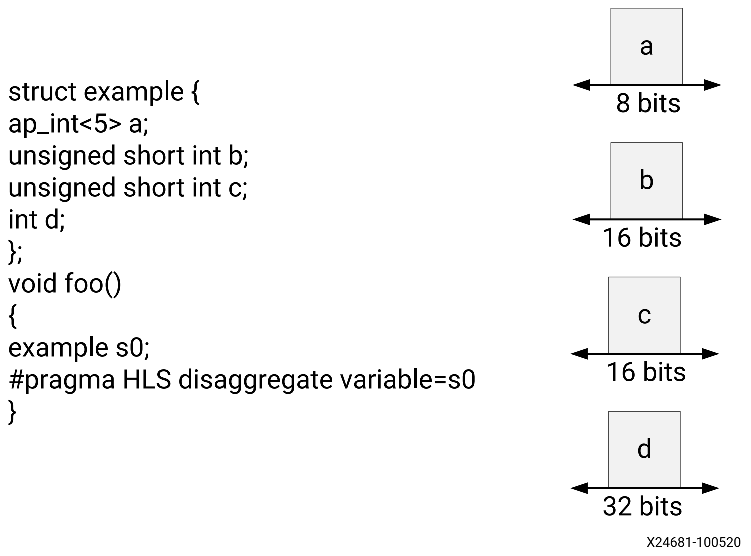
Struct Padding and Alignment

Structs in Vitis HLS can have different types of padding and alignment depending on the use of `__attributes__` 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 set_directive_aggregate pragma or directive to prevent the default disaggregation of structs in the code.*

Figure 1: **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
}
```

- **Aggregate:** This is the default behavior for structs on the interface, as discussed in Interface Synthesis and Structs. Vitis HLS joins the elements of the struct, aggregating the struct into a single data unit. This is done in accordance with pragma HLS aggregate, 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 data padding for elements of the struct, to align the byte structures on a default 4-byte alignment. You can disaggregate structs as described in the AGGREGATE pragma or directive.

  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 to elements of the struct, to align it on "X" byte boundaries. In the figure below, the struct is aligned on a 2-byte boundary.

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

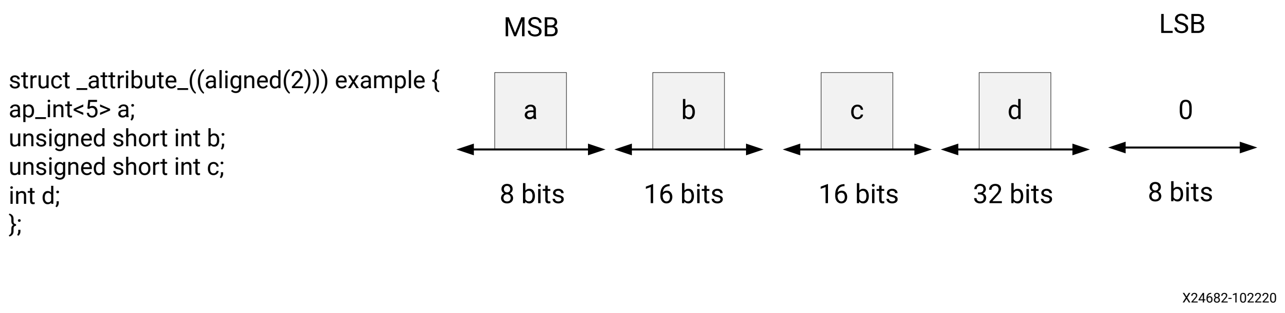
  Data types in struct with custom data widths, such as `ap_int`, are allocated with sizes which are powers of 2. Vitis HLS adds padding bits for aligning the size of the data type to a power of 2.

  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;
};
```

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

Figure 2: **Aligned Struct Implementation**



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.

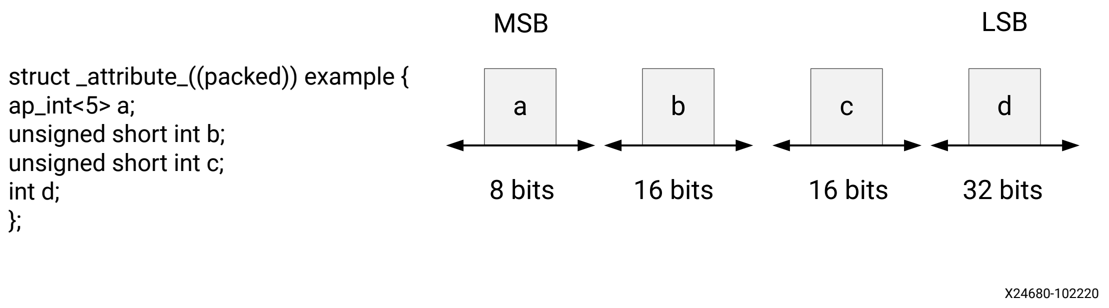
```
struct data_t {
  short varA;
  int varB;
  short varC;
};
```

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 3: **Packed Struct Implementation**



Interface Bundle Rules

The interface pragma contains a bundle option that groups function arguments to the respective individual AXI interface ports. The following sections list the rules if there is a mix of user-defined/not used and default bundle option.

S_AXI Lite

User-Defined and Default Bundle Names

- **Rule 1: User-specified Bundle Name:** This rule explicitly groups all interface ports with the same `bundle=<string>` into the same AXI4-Lite interface port and names the RTL port the value specified by `s_axi_<string>`.

```
void top(char *a, char *b, char *c, char *d) {
#pragma HLS INTERFACE s_axilite port=a bundle=terry
#pragma HLS INTERFACE s_axilite port=b bundle=terry
#pragma HLS INTERFACE s_axilite port=c bundle=stephen
#pragma HLS INTERFACE s_axilite port=d bundle=jim
}
```

```
INFO: [RTGEN 206-100] Bundling port 'd' to AXI-Lite port jim.
INFO: [RTGEN 206-100] Bundling port 'c' to AXI-Lite port stephen.
INFO: [RTGEN 206-100] Bundling port 'a' and 'b' to AXI-Lite port terry.
INFO: [RTGEN 206-100] Finished creating RTL model for 'example'
```

- **Rule 2: Default Bundle Name:** This rule explicitly groups all interface ports with no bundle name into the same AXI4-Lite interface port, and uses tool default `bundle=<deafult>`, and names the RTL port `s_axi_<default>`.

```
void top(char *a, char *b, char *c, char *d) {
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c
#pragma HLS INTERFACE s_axilite port=d
}
```

```
Log fileINFO: [RTGEN 206-100] Bundling port 'a', 'b', 'c' to AXI-Lite
port control.
INFO: [RTGEN 206-100] Finished creating RTL model for 'example'.
```

- **Rule 3: Partially Specified Bundle Name:** If the bundle names are partially specified, then the tool will create more than `s_axi lite` interface ports, see the following bundle rules:

  - This rule explicitly groups all interface ports into the same AXI4-Lite Interface port which uses the default= control "and."

- This rule also explicitly groups all the remaining un-specified bundle names to the new default name which does not conflict with any user names.

```
void top(char *a, char *b, char *c, char *d) {
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c bundle=control
#pragma HLS INTERFACE s_axilite port=d bundle=control
}
```

```
INFO: [RTGEN 206-100] Bundling port 'c' and 'return' to AXI-Lite port
control.
INFO: [RTGEN 206-100] Bundling port 'a' and 'b' to AXI-Lite port
control_r.
INFO: [RTGEN 206-100] Finished creating RTL model for 'example'.
```

MAXI

- **Rule 0: No Global bundle configuration**: The global config option `config_interface -m_axi_auto_max_ports false` will impact the bundle rules as follows: The global config option `config_interface -m_axi_auto_max_ports false` will impact the bundle rules as follows:
    - **Rule 1: User-specified Bundle Name**:
    - This rule explicitly groups all interface ports with the same `bundle=<string>` into the same AXI MAXI interface port and names the RTL port the value specified by `m_axi_<string>`.

```
#pragma HLS INTERFACE m_axi port=a depth=50 bundle=terry
#pragma HLS INTERFACE m_axi port=a depth=50
#pragma HLS INTERFACE m_axi port=a depth=50
Log file
INFO: [RTGEN 206-500] Setting interface mode on port 'example/terry' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem1' to
'm_axi'.
```

```
#pragma HLS INTERFACE m_axi port=a depth=50 bundle=terry
#pragma HLS INTERFACE m_axi port=a depth=50 bundle=terry
#pragma HLS INTERFACE m_axi port=a depth=50 bundle=terry
Log file
INFO: [RTGEN 206-500] Setting interface mode on port 'example/terry' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/terry' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/terry' to
'm_axi'.
```

    - **Rule 2: Default Bundle Name**:

- This rule explicitly groups all interface ports with no bundle name into the same AXI interface port, and uses tool default `bundle=<default>`, and names the RTL port `<default>_m_axi`.

```
#pragma HLS INTERFACE m_axi port=a depth=50
#pragma HLS INTERFACE m_axi port=a depth=50
#pragma HLS INTERFACE m_axi port=a depth=50
Log file
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
```

```
#pragma HLS INTERFACE m_axi port=a depth=50 bundle=terry
#pragma HLS INTERFACE m_axi port=a depth=50
#pragma HLS INTERFACE m_axi port=a depth=50

Log file
INFO: [RTGEN 206-500] Setting interface mode on port 'example/terry' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
```

- **Rule 3: Global Bundle Configuration**:
- The global config option `config_interface -m_axi_auto_max_ports true` will impact the bundle rules.
    - This rule explicitly maps all the unspecified bundle interface ports into individual different AXI MAXI interface ports and names the RTL port sequentially as `gmem_0, geme_1, geme_2`.

Interface Offset

The interface option contains an offset option that controls the address offset in the AXI4-Lite (`s_axilite`) and AXI4 (`m_axi`) interfaces. The following list the rules if there is a mix of user-defined/not used and default offset option.

SAXI Lite

- **Rule 1: Fully Specified:** This rule explicitly groups all the scalar and offsets into the user-specified AXI4-Lite ports.
- **Rule 2: Default Specified:** This rule explicitly groups all the scalar and offsets without offset settings updating the default AXI4-Lite ports.
- **Rule 3: Partially Specified:** If the offsets are partially specified, then the tool will create more than one `s_axi lite` interface ports.

MAXI Offset

- **Fully Specified Offset**: This rule adheres to the user-specified offset settings in the pragma.
- **No-offset Specified**: If maxi offset is not specified in the pragma, the global config option: `config_interface -m_axi_offset <off/direct/slave>` will impact the offset rules.
  - **Rule 1**: User-specified SAXI Lite: This rule explicitly groups all the maxi offsets into user-specified AXI_lite port, for which the MAXI offset=<slave>

    ```
    void top(int *a) {
    #pragma HLS interface m_axi port=a
    #pragma HLS interface s_axilite port=a
    }
    ```

  - **Rule 2**: No SAXI LITE: This rule explicitly groups all the maxi offsets into the tool default offset.
    - For Vitis: `- offset = slave`
    - For Vivado: `- offset = off`
    - User Specified: `config_interface -m_axi_offset direct`

    ```
    void top(int *a, int *b) {
    #pragma HLS interface m_axi port=a bundle=M0
    #pragma HLS interface m_axi port=b bundle=M0
    }
    ```

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 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. Refer to AXI4-Stream Interfaces with Side-Channels for more information.

Behavior Changes to config_rtl -module_auto_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. In 2020.1 Vitis HLS, 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
```

Pragmas Syntax

Allocation

- The Syntax of the allocation pragma must be followed..

  Example:

  ```
  #pragma HLS allocation instances=mul limit=1 operation // Invalid format
  #pragma HLS allocation operation instances=mul limit=1 // Valid format
  #pragma HLS allocation instances=foo limit=2 function // Invalid format
  #pragma HLS allocation function instances=foo limit=2 // Valid format

  IMPORTANT: If the referenced function/operation is not located after
  "allocation", it will be ignored with a warning message:
  WARNING: [HLS 207-1604] unexpected pragma argument 'instances', expects
  function/operation.
  ```

- When Using Template functions, the allocation pragma syntax has to be adhered as shown below.

  ```
  template <typename DT>
  void foo(DT a, DT b){
  }



  Invalid syntax
  // Below is invalid
  #pragma HLS ALLOCATION function instances = foo
  Valid syntax

  // Below is valid
  #pragma HLS ALLOCATION function instances = foo<DT>
  ```

Dependence

- DEPENDENCE pragma syntax always requires the true/false option.

  ```
  // Below will emit a warning
  #pragma HLS dependence variable=a

  // Below will not have a warning
  #pragma HLS dependence variable=a false
  ```

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 old 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
```

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.*

Top-level Function Arguments

Pragmas

- Pragma DEPENDENCE on an argument that also has an INTERFACE pragma with a `m_axi` 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
#prgama HLS interface m_axi port=a offset=slave bundle=gmem
#prgama 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.

Data Types

The following data types on the top-level function arguments are not supported in this release.

- enum or any use of enum (struct, array pointer of enum)
- 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 no longer supported.
- Virtual functions are no longer supported.

Deprecated and Unsupported Tcl Command Options

Vitis™ HLS has deprecated a number of Vivado® HLS commands. These deprecated commands are no longer supported in the Vitis HLS tool, and are listed in the table below.

Table 6: **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_size`	Deprecated	
config	`config_interface`	`-expose_global`	Unsupported	
config	`config_interface`	`-trim_dangling_port`	Unsupported	
config	`config_array_partition`	`-scalarize_all`	Unsupported	
config	`config_array_partition`	`-throughput_driven`	Unsupported	
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_fill_reg`	Deprecated	
config	`config_bind`	* (all options)	Deprecated	
config	`config_rtl`	`-encoding`	Deprecated	
config	`config_sdx`	* (all options)	Deprecated	
config	`config_flow`	* (all options)	Deprecated	

Table 6: **Vivado HLS Commands Deprecated in Vitis HLS** *(cont'd)*

Type	Command	Option	Vitis HLS	Details
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	`set_directive_resource`	`-location`	Deprecated	
directive/pragma	`CLOCK`	*	Unsupported	
directive/pragma	`DATA_PACK`	*	Unsupported	Use aggregate attribute.
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.
directive/pragma	`STREAM`	`-dim`	Unsupported	
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.

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.

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. *Vivado Design Suite User Guide: Designing with IP* (UG896)
2. *Vivado Design Suite: AXI Reference Guide* (UG1037)

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright