

Vitis HLS 移植指南

UG1391 (v2020.1) 2020 年 7 月 28 日

条款中英文版本如有歧义，概以英文版本为准。



修订历史

章节	修订综述
2020 年 7 月 28 日 2020.1 版	
部分章节。	新增信息
2020 年 6 月 3 日 2020.1 版	
初始版本。	不适用

目录

修订历史.....	2
第 1 章：移植到 Vitis HLS.....	4
HLS 行为差异.....	4
数据流.....	15
第 2 章：不受支持的功能.....	16
顶层函数实参.....	16
结构体.....	17
HLS 视频库.....	17
C 语言任意精度类型.....	17
第 3 章：已弃用的 Tcl 命令选项和不受支持的 Tcl 命令选项.....	18
附录 A：附加资源与法律提示.....	20
赛灵思资源.....	20
Documentation Navigator 与设计中心.....	20
参考资料.....	20
请阅读：重要法律提示.....	21

移植到 Vitis HLS

对任一版本的 Vivado HLS 中已实现的内核模块进行移植时，必须明确掌握 HLS 版本间的差异以及这些差异对于设计的影响。

关键考虑因素：

- 行为差异
- 不受支持的功能
- 已弃用的命令

HLS 行为差异

Vitis HLS 对 HLS 综合 C 语言代码的方法进行了一些基础性更改。这些更改会影响应用的 QoR。强烈建议在使用工具前仔细审查行为差异。

默认用户控制设置

默认全局选项用于为 Vitis 应用加速开发流程或 Vivado IP 开发流程配置解决方案。

```
open_solution -flow_target <vitis | vivado>
```

此全局选项将替代旧配置选项 (`config_sdx`)。

Vivado 流程：

配置解决方案，以便在支持 Vivado IP 生成流程的前提下运行，要求严格按标准使用编译指示和指令，并将结果导出为 Vivado IP。

```
open_solution -flow_target vivado
```

表 1: 默认控制设置表

默认控制设置	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	不适用	0
config_interface -m_axi_max_widen_bitwidth	不适用	0

表 1：默认控制设置表 (续)

默认控制设置	Vivado_hls	Vitis_hls
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 编译指示默认设置	ip 模式	ip 模式

Vitis 流程 (内核模式)：

配置解决方案，以供在 Vitis 应用加速开发流程中使用。这样即可将 Vitis HLS 工具配置为无需指定 INTERFACE 编译指示或指令即可正确推断函数实参的接口，并将综合后的 RTL 代码作为 Vitis 内核对象文件 (.xo) 来输出。

```
open_solution -flow_target vitis
```

表 2：默认控制设置表

默认控制设置	Vivado_hls	Vitis_hls
interface 编译指示默认设置	ip 模式	内核模式 (检查默认接口)
config_interface -m_axi_alignment_byte_size	不适用	64
config_interface -m_axi_max_widen_bitwidth	不适用	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
config_interface -m_axi_latency	0	64

默认循环 II 约束设置

在 Vivado HLS 中，默认循环 II 约束设置为 1，但在 Vitis HLS 中，则设置为 auto。例如，如果工具无法实现默认 II，它将尝试实现可能实现的最佳 II。

默认接口

由接口综合所创建的接口类型取决于 C 语言实参的数据类型、默认接口模式以及接口指令。在 Vitis HLS 中，默认接口根据 C 语言实参和配置所使用的数据类型而变。用户所选的 `open_solution -flow_target <vitis | vivado>` 将复制默认接口设置。

下图显示了启用默认接口协议时的变化。

下表中使用的实参类型定义：

- I：仅输入 (仅限从实参读取)
- O：仅输出 (仅限写入实参)
- IO：输入和输出 (可从实参读取，也可写入实参)

- 返回：返回数据输出
- 块：块级控制
- D：每一种类型的默认模式。

注释：如果指定非法接口，Vitis HLS 会发出警告消息，并实现默认接口模式。

Vitis 流程（内核模式）

如果在 Vitis 流程中使用 HLS，那么该工具将自动设置如下配置。

```
open_solution -flow_target vitis
```

表 3: 实参类型

实参类型	标量		指向数组的指针			Hls::stream
	接口模式	输入	返回	I	I/O	
ap_ctrl_none						
ap_ctrl_hs						
ap_ctrl_chain		D				
axis						D
m_axi			D	D	D	

AXI4-Lite 从接口指令将更改 interface 编译指示的行为，如下所示。

```
config_interface -default_slave_interface s_slave
```

表 4: 实参类型

实参类型	标量		指向数组的指针			Hls::stream
	接口模式	输入	返回	I	I/O	
s_axi_lite	D	D	D	D	D	

注释：用户指定的 interface 编译指示可覆盖这些默认 interface 编译指示设置。

Vivado 设计流程

Vitis HLS 可在独立模式下用于创建 IP。默认情况下，该工具将运行该流程并为其设置以下全局选项：

- `open_solution -flow_target vivado`
- `config_interface default_slave_interface off`（这意味着默认情况下，不会对源代码自动应用任何 AXI4-Lite 接口指示信息。）

在此情况下，将应用以下默认接口。

图 1：实参类型

Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none									
ap_ctrl_hs		D							
ap_ctrl_chain									
axis									
s_axilite									
m_axi									
ap_none	D					D			
ap_stable									
ap_ack									
ap_vld								D	
ap_ovld							D		
ap_hs									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									

Supported
 D = Default Interface
 Not Supported

X14293

结构体

代码中的结构体（例如，内部变量和全局变量）默认情况下处于解聚 (disaggregated) 状态。这些结构体分解为其各自的成员元素。创建的元素数量和类型取决于结构体本身的内容。结构体数组作为多个数组来实现，每个结构体成员都具有独立的数组。编译器会以额外字节来填充 C/C++ 语言中的结构体以实现数据对齐。为了使 Vitis HLS 中的内核代码与 gcc 兼容，将以额外字节来填充内核代码中的结构体。在“结构体填充与对齐”部分中显示了结构体示例：

任意精度类型 (ap_int 库) 的数据布局

对于含定制数据宽度的结构体中的数据类型（如，`ap_int`），为其分配的大小为 2 的指数。Vitis HLS 添加了填充位，用于将数据类型的大小对齐到 2 的指数。

在以下示例中，结构体中 `varA` 的大小将填充到 8 位（而不是 5 位）。

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



提示： Vitis HLS 还将填充 `bool` 数据类型，以将其对齐到 8 位。

结构体填充与对齐

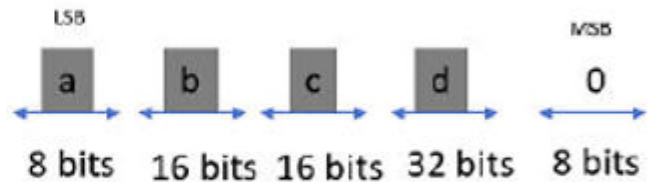
Vitis HLS 中的结构体根据所使用的 `__attributes__` 或 `#pragmas`，可能采用不同类型的填充和对齐方式。这些功能如下所述。

- 解聚：

默认情况下，代码中作为内部变量的结构体处于解聚状态。而内核接口上定义的结构体则并非如此。解聚结构体细分为各独立元素，如 `set_directive_disaggregate` 或 `pragma HLS disaggregate` 中所述。您无需应用 `DISAGGREGATE` 编译指示或指令，因为这是结构体的默认行为。

图 2：解聚结构体

```
Struct example __attribute__((aligned(X)))
{
    ap_int<5> a;
    Unsigned short int b;
    Unsigned short int c;
    int d;
};
```



- 聚合：

这是接口上的结构体的默认行为（如[接口综合与结构体](#)中所述）。Vitis HLS 将结构体的元素连接在一起，从而将此结构体聚合为单一数据单元。此默认操作是根据 `pragma HLS aggregate` 完成的，但您无需指定编译指示，因为对于接口上的结构体，这是默认行为。聚合过程可能还包含对结构体元素进行数据填充，以便按默认 4 字节对齐方式来实现字节结构对齐。

注释： 指定 `-Wpadded` 作为编译器标志来添加用于填充结构体的多个位时，该工具会发出警告。

- 已对齐：

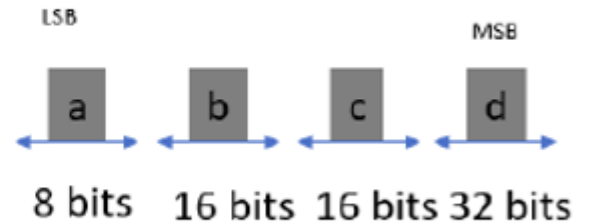
默认情况下，Vitis HLS 会按 4 字节对齐方式来对齐结构体，通过填充结构体元素来将其对齐到 32 位宽。但您可使用 `__attribute__((aligned(X)))` 来为结构体元素添加填充，以便将其对齐到“X”字节边界。在下图中，结构体对齐到 2 字节边界



重要提示！ 请注意，“X”只能定义为 2 的指数。

图 3：已对齐的结构体实现

```
Struct example __attribute__((packed))
{
    ap_int<5> a;
    Unsigned short int b;
    Unsigned short int c;
    int d;
};
```



所使用的填充取决于结构体元素的顺序和大小。在以下代码示例中，结构体对齐为 4 字节，Vitis HLS 将在第一个元素 `varA` 之后添加 2 字节用于填充，在第三个元素 `varC` 之后再添加 2 字节用于填充。结构体的总计大小将为 96 位。

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

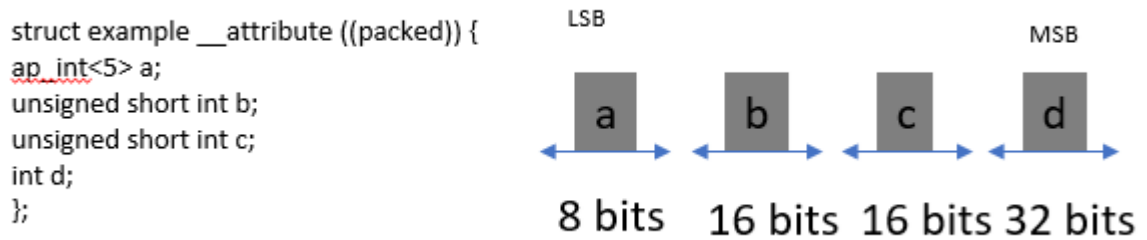
但如果您按如下方式重写该结构体，则无需填充，而结构体的总计大小将为 64 位。

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

- 已封装：

Vitis HLS 通过指定 `__attribute__((packed(X)))` 来封装结构体元素，在此情况下结构体大小取决于结构体每个元素的实际大小。在以下示例中，这意味着结构体大小为 72 位：

图 4：已封装的结构体实现



```

struct __attribute__((packed)) data_t {
    short varA;
    int varB;
    short varC;
};
    
```

接口捆绑

接口选项包含捆绑选项，用于将函数实参分组到各 AXI 接口端口。下表列出了将用户指定的捆绑名称或无捆绑名称与默认捆绑名称混用情况下的行为更改。

S_axilite 捆绑

用户定义的捆绑名称与默认捆绑名称

- 工具将使用用户指定的 s_axi lite 捆绑名称来创建特定名称的 axi lite 端口。

```

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
}
    
```

- 日志文件：**可在日志文件或 RTL 文件中查看已综合的 axi lite 端口名称。

```

INFO: [RTGEN 206-100] Bundling port 'return' 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'
    
```

- 如果未指定捆绑名称，工具将使用默认捆绑名称 -control 来创建含此特定名称的 axi lite 端口。

```

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
}
    
```

- **日志文件：**可在日志文件或 RTL 文件中查看已综合的 axi lite 端口名称。

```
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'.
```

- 以下情况下，工具将使用新默认名称 - control_r：如果用户混用默认捆绑名称 -control 和用户定义的名称来创建 axi lite 端口。

```
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
}
```

- **日志文件：**可在日志文件或 RTL 文件中查看已综合的 axi lite 端口名称。

```
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 捆绑选项

- 如果用户指定 config_interface - maxi_auto_max_ports = true，那么未显式捆绑的所有 M-AXI 接口都将映射到单独的 axi mm 端口。

```
#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'.
```

- 如果用户设置 config_interface m_axi_auto_max_ports=false，则将发生以下行为之一。
 - 如果用户将所有 MAXI interface 编译指示都映射到某一捆绑名称，例如“terry”。HLS 会生成名为“terry”的单一 MAXI 端口。

```
#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'.
```

- 如果用户在 interface 编译指示上不指定捆绑名称。HLS 会生成默认 MAXI 端口名称 - gmem。

```
#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'.
```

- 如果用户为部分接口指定捆绑名称而忽略其它接口。HLS 将把无捆绑的所有端口都捆绑到默认端口 - gmem，并且将使用用户指定的捆绑来生成 axi MM 端口。

```
#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'.
```

接口偏移

接口选项包含偏移选项，用于控制 AXI4-Lite (s_axilite) 接口和 AXI4 (m_axi) 接口中的地址偏移。下表列出了使用用户指定的捆绑名称或无捆绑名称与使用默认捆绑名称的行为差异。

Maxi 偏移

捆绑规则

工具将使用用户指定的偏移（如已提供）。

```
void top(char *a, char *b) {
#pragma HLS INTERFACE maxi port=a offset=direct
#pragma HLS INTERFACE maxi port=b offset=off
}
```

如果用户不指定偏移选项。

- 如果 default_slave_interface=off - 偏移将为直接偏移。
- 如果 default_slave_interface=s_axilite - 偏移将为从偏移。

```
void top(char *a, char *b) {
#pragma HLS INTERFACE maxi port=a bundle=my_maxi
#pragma HLS INTERFACE maxi port=b bundle=my_maxi
}
```

S_axilite 偏移

S_axilite 标量

将标量端口/偏移绑定到 axilite 适配器时，存在以下 3 种编译指示场景：

偏移的捆绑规则

- **完全指定**：HLS 将使用指定的 axilite 适配器。
- **未指定**：默认情况下，对于 Vitis 流程，HLS 将创建 axilite 适配器以自动发射标量端口和偏移。
- **部分指定**：任一未指定的偏移或标量都将绑定到 1 个额外的 axilite 适配器。

接口上的存储器属性

interface 编译指示或指令上的 storage_type 选项允许用户显式定义所使用的 RAM 类型以及所创建的 RAM 端口（单端口或双端口）。如果不指定 storage_type，Vitis HLS 会使用：

- 单端口 RAM（默认情况下）。
- 双端口 RAM，前提是这样可缩短启动时间间隔或减少时延。

对于 Vivado 流程，用户可以在指定接口上指定 RAM 存储类型，并将旧的 resource 编译指示替换为 storage_type。

```
#pragma HLS INTERFACE bram port = in1 storage_type=RAM_2P
#pragma HLS INTERFACE bram port = out storage_type=RAM_1P latency=3
```

含旁路的 AXI4-Stream 接口

含旁路的 AXI4-Stream：旁路为属于 AXI4-Stream 标准的一部分的可选信号。在 C 语言代码中可使用结构体直接引用和控制旁路信号，前提是结构体的成员元素与 AXI4-Stream 旁路信号的名称相匹配。AXI-Stream 旁路信号被视为数据信号，随 TDATA 一起寄存。

不建议用户使用自己的结构体作为 AXIS 旁路。建议使用编译器所提供的 ap_axis_u.h 类。

```
#include "ap_axi_sdata.h"
#include "ap_int.h"
#include "hls_stream.h"

#define DWIDTH 32

typedef ap_axiu<DWIDTH, 0, 0, 0> trans_pkt;

extern "C" {
void krnl_stream_vmult(
    hls::stream<trans_pkt> &b,
    hls::stream<trans_pkt> &output) {
    #pragma HLS INTERFACE axis port=b
    #pragma HLS INTERFACE axis port=output
    #pragma HLS INTERFACE s_axilite port=return bundle=control

    bool eos = false;
vmult:
    do {
```

```

#pragma HLS PIPELINE II=1
trans_pkt t2 = b.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 = t1.get_last();
t_out.keep = -1; // Enabling all bytes

// Writing packet to output stream
output.write(t_out);

if (t2.get_last()) {
    eos = true;
}
} while (eos == false);
}
}
    
```

限制

将 AXI4-Stream 接口与旁路配合使用并且函数实参为结构体时，Vitis HLS 会自动将该结构体的所有元素封装到单宽数据矢量中。该接口将作为单宽数据矢量来实现。此工具限制不允许用户在整个 axis 接口中发送结构体（例如，RGBPixel）。以下变通方法使用范围运算符，这样应可为用户实现相同的行为。

```

struct RGBPixel
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
    unsigned char a;

    RGBPixel(ap_int<32> d) : r(d.range(7,0)), g(d.range(15,8)),
b(d.range(23,16)), a(d.range(31,24)) {
#pragma HLS INLINE
}

    operator ap_int<32>() {
#pragma HLS INLINE
        return (ap_int<8>(a), ap_int<8>(b), ap_int<8>(g), ap_int<8>(r));
    }

}__attribute__((aligned(4)));
    
```

config_rtl -module_auto_prefix

config_rtl -module_auto_prefix 的行为更改

在 Vivado HLS 中，启用 config_rtl -module_auto_prefix 时，顶层 RTL 模块名称将采用其自己的名称作为前缀。在 2020.1 版 Vitis HLS 中，此 auto 前缀将仅应用于子模块。

启用 `-module_prefix` 选项则不会导致行为发生更改，如果使用该选项，那么指定的前缀值将置于所有模块（包括顶层模块）之前。并且，`-module_prefix` 选项也优先于 `-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
```

数据流

针对 `std::complex` 的支持：

在 Vivado HLS 中，无法在数据流 (DATAFLOW) 中直接使用 `std::complex` 数据类型，因为存在多重读写程序问题。此多重读写程序问题原因在于调用了 `std` 类构造函数来初始化该值。如果同时在数据流中使用该变量作为通道，就会导致以上问题。但 Vitis 支持通过使用 `no_ctor` 属性来使用 `std::complex`，如下所示。

```
// 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; // not here! (hls::stream has
it internally)

    proc_1(buffer, in);
    proc_2(fifo, buffer, acc);
    proc_3(out, fifo, acc);
}
```

不受支持的功能

在此版本中，不支持以下功能。



重要提示！ HLS 将对本章节中所提到的所有不受支持的功能发出警告或错误消息。

顶层函数实参

编译指示

- 如果实参同时包含编译指示 DEPENDENCE 和编译指示 INTERFACE，并且后者包含带有 2 个以上端口的 m_axi 捆绑，则不支持此类编译指示。

```
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
}
```

- 不支持应用于顶层函数实参的编译指示：
 - ARRAY_PARTITION (仅支持 dim =1, cyclic/block/complete)
 - ARRAY_RESHAPE
 - Resource

数据类型

- 在此版本中不支持在顶层函数实参上采用以下数据类型。
 - enum 或 enum 的任意用法 (结构体、enum 的数组指针)
 - _Complex
 - _Half 和 _fp16

结构体

- 不支持应用于特定结构体字段的 `dependence`、`resource` 和 `interface` 等的编译指示。变通方法是，用户可以指定 `disaggregate` 编译指示，并对结构体元素应用 `dependence`。

Unsupported Form

```
struct ST {
  int B[100];
  int C;
};
struct ST aa;
#pragma HLS dependence variable=aa.B false
workaround
struct ST {
  int B[100];
  int C;
};
struct ST aa;#pragma HLS disaggregate variable=aa
#pragma HLS dependence variable=aa.B false
```

- 如果编译指示所指定到的类/结构体实例属于该编译指示内部的方法/变量的目标，则不支持此类编译指示。变通方法是将此编译指示移至类/结构体定义内部，如以下代码所示。此限制将在后续版本中得到解决。

```
class ST {
  int B[100];
  ST () {
    #pragma HLS array_partition variable=B
    #pragma HLS inline
  }
};
```

HLS 视频库

- 适用于视频实用工具和功能的 `hls_video.h` 已弃用，并替换为 Vitis 视觉库。请参阅 [Github 视频库移植指南](#)，以获取更多详细信息。

C 语言任意精度类型

Vitis HLS 不支持 C 语言任意精度类型，建议用户改用 C++ 类型作为任意精度类型。

已弃用的 Tcl 命令选项和不受支持的 Tcl 命令选项

Vitis HLS 包含已弃用的 Tcl 命令选项和不受支持的 Tcl 命令选项。

表 5: 已弃用的 Tcl 命令表

类型	命令	选项	Vitis HLS 2020.1	详情
配置	config_interface	-expose_global	不支持	
配置	config_interface	-trim_dangling_port	不支持	
配置	config_array_partition	-scalarize_all	不支持	
配置	config_array_partition	-throughput_driven	不支持	
配置	config_array_partition	-maximum_size	不支持	
配置	config_array_partition	-include_extern_globals	不支持	
配置	config_array_partition	-include_ports	不支持	
配置	config_schedule	* (所有选项)	已弃用	
配置	config_bind	* (所有选项)	已弃用	
配置	config_rtl	-encoding	已弃用	
配置	config_sdx	-target	已弃用	
配置	config_flow	*	已弃用	
配置	config_sdx	-profile	已弃用	重命名为 config_export_vivado_optimization_level
配置	config_sdx	-optimization_level	已弃用	
指令	Clock	*	不支持	
指令	Data_pack	*	不支持	使用 pack 属性。
指令	Function_instantiate	不适用	已弃用	
指令	Inline	-region	已弃用	
指令	Array_map	*	不支持	
指令	Resource	*	已弃用	替换为 bind_op 和 bind_storage。
指令	Interface	*	已弃用	
指令	Stream	-dim	不支持	
指令	Top	-name	重命名	
配置	config_dataflow	-disable_start_propagation	已弃用	

表 5：已弃用的 Tcl 命令表 (续)

类型	命令	选项	Vitis HLS 2020.1	详情
配置	config_rtl	-auto_prefix	已弃用	替换为 config_rtl - module_prefix。
配置	config_rtl	-prefix	已弃用	替换为 config_rtl - module_prefix。
指令	Dependence	-class (array pointer)	已弃用	变通方法：指定变量名称。
指令	Dependence	dependent (-true false)	已弃用	变通方法使用大于 0 的“distance”值来替换“dependent=true”。
指令	Dependence	-direction (RAW WAR WAW)	已弃用	
指令	Dependence	-distance <integer>	已弃用	
tcl	csim_design	-clang_sanitizer	添加/重命名	
tcl	export_design	-use_netlist	已弃用	
tcl	export_design	-xo	已弃用	
tcl	add_files	system-c	不支持	

1. **已弃用**：在后续版本中将发出终止使用相关编译指示的警告消息。
2. **不支持**：Vitis HLS 将报错，并输出有效消息。
3. *：命令中的所有选项。

附加资源与法律提示

赛灵思资源

如需了解答复记录、技术文档、下载以及论坛等支持性资源，请参阅[赛灵思技术支持](#)。

Documentation Navigator 与设计中心

赛灵思 Documentation Navigator (DocNav) 提供了访问赛灵思文档、视频和支持资源的渠道，您可以在其中筛选搜索信息。打开 DocNav 的方法：

- 在 Vivado® IDE 中，单击“Help” → “Documentation and Tutorials”。
- 在 Windows 中，单击“Start” → “All Programs” → “Xilinx Design Tools” → “DocNav”。
- 在 Linux 命令提示中输入“docnav”。

赛灵思设计中心提供了根据设计任务和其他话题整理的文档链接，您可以使用链接了解关键概念以及常见问题解答。访问设计中心：

- 在 DocNav 中，单击“Design Hub View”标签。
- 在赛灵思网站上，查看[设计中心](#)页面。

注释：如需了解更多有关 DocNav 的信息，请参阅赛灵思网站上的 [Documentation Navigator](#)。

参考资料

以下技术文档是非常实用的补充资料，可配合本指南一起使用：

1. 《Vivado Design Suite 用户指南：采用 IP 进行设计》(UG896)
2. 《Vivado Design Suite：AXI 参考指南》(UG1037)

请阅读：重要法律提示

本文向贵司/您所提供的信息（下称“资料”）仅在对赛灵思产品进行选择和使用参考。在适用法律允许的最大范围内：（1）资料均按“现状”提供，且不保证不存在任何瑕疵，赛灵思在此声明对资料及其状况不作任何保证或担保，无论是明示、暗示还是法定的保证，包括但不限于对适销性、非侵权性或任何特定用途的适用性的保证；且（2）赛灵思对任何因资料发生的或与资料有关的（含对资料的使用）任何损失或赔偿（包括任何直接、间接、特殊、附带或连带损失或赔偿，如数据、利润、商誉的损失或任何因第三方行为造成的任何类型的损失或赔偿），均不承担责任，不论该等损失或者赔偿是何种类或性质，也不论是基于合同、侵权、过失或是其他责任认定原理，即便该损失或赔偿可以合理预见或赛灵思事前被告知有发生该损失或赔偿的可能。赛灵思无义务纠正资料中包含的任何错误，也无义务对资料或产品说明书发生的更新进行通知。未经赛灵思公司的事先书面许可，贵司/您不得复制、修改、分发或公开展示本资料。部分产品受赛灵思有限保证条款的约束，请参阅赛灵思销售条款：<https://china.xilinx.com/legal.htm#tos>；IP 核可能受赛灵思向贵司/您签发的许可证中所包含的保证与支持条款的约束。赛灵思产品并非为故障安全保护目的而设计，也不具备此故障安全保护功能，不能用于任何需要专门故障安全保护性能的用途。如果把赛灵思产品应用于此类特殊用途，贵司/您将自行承担风险和责任。请参阅赛灵思销售条款：<https://china.xilinx.com/legal.htm#tos>。

关于与汽车相关用途的免责声明

如将汽车产品（部件编号中含“XA”字样）用于部署安全气囊或用于影响车辆控制的应用（“安全应用”），除非有符合 ISO 26262 汽车安全标准的安全概念或冗余特性（“安全设计”），否则不在质保范围内。客户应在使用或分销任何包含产品的系统之前为了安全的目的全面地测试此类系统。在未采用安全设计的条件下将产品用于安全应用的所有风险，由客户自行承担，并且仅在适用的法律法规对产品责任另有规定的情况下，适用该等法律法规的规定。

版权声明

© Copyright 2020 赛灵思公司版权所有。Xilinx、赛灵思标识、Alveo、Artix、Kintex、Spartan、Versal、Virtex、Vivado、Zynq 本文提到的其它指定品牌均为赛灵思在美国及其它国家的商标。所有其它商标均为各自所有方所属财产。