



Vitis High-Level Synthesis User Guide (UG1399)

HLS Pragmas

pragma HLS aggregate
pragma HLS allocation
pragma HLS array_partition
pragma HLS array_reshape
pragma HLS bind_op
pragma HLS bind_storage
pragma HLS dataflow
pragma HLS dependence
pragma HLS disaggregate
pragma HLS expression_balance
pragma HLS function_instantiate
pragma HLS inline
pragma HLS interface
pragma HLS latency
pragma HLS loop_flatten
pragma HLS loop_merge
pragma HLS loop_tripcount
pragma HLS occurrence
pragma HLS pipeline
pragma HLS protocol
pragma HLS reset
pragma HLS shared
pragma HLS stable
pragma HLS stream
pragma HLS top
pragma HLS unroll

HLS Pragma

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. For detailed pragma information, refer to the [Vitis HLS Flow](#).

Table: Vitis HLS Pragmas by Type

Type	Attributes
Kernel Optimization	<ul style="list-style-type: none">• pragma HLS aggregate• pragma HLS bind_op• pragma HLS bind_storage• pragma HLS expression_balance• pragma HLS latency• pragma HLS reset• pragma HLS top
Function Inlining	<ul style="list-style-type: none">• pragma HLS inline
Interface Synthesis	<ul style="list-style-type: none">• pragma HLS interface

Type	Attributes
Task-level Pipeline	<ul style="list-style-type: none">• <code>pragma HLS dataflow</code>• <code>pragma HLS shared</code>• <code>pragma HLS stream</code>
Pipeline	<ul style="list-style-type: none">• <code>pragma HLS pipeline</code>• <code>pragma HLS occurence</code>
Loop Unrolling	<ul style="list-style-type: none">• <code>pragma HLS unroll</code>• <code>pragma HLS dependence</code>
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>

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>
```

Where:

variable=<variable>

Specifies the variable to be grouped.

Example 1

Aggregates struct pointer AB with three 8-bit fields (typedef struct {unsigned char R, G, B;} pixel) in function foo, into a new 24-bit pointer.

```
typedef struct{
    unsigned char R, G, B;
} pixel;

pixel AB;
#pragma HLS aggregate variable=AB
```

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 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

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> factor=<int> dim=<int>
```

Where:

variable=<name>

A required argument that specifies the array variable to be partitioned.

<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 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 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 complete dim=2
```

See Also

- [set_directive_array_partition](#)
- [pragma HLS array_reshape](#)

pragma HLS array_reshape

Description

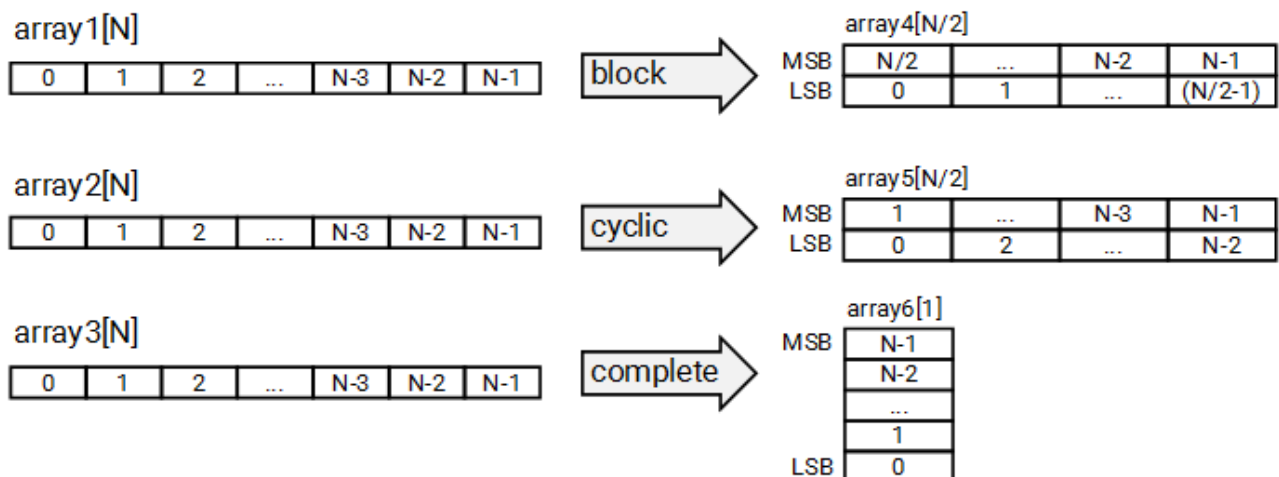
The ARRAY_RESHAPE pragma reforms the array with vertical remapping and concatenating elements of arrays by increasing bit-widths. This reduces the number of block RAM consumed while providing parallel access to the data. This pragma creates a new array with fewer elements but with greater bit-width, 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 block factor=2
  dim=1
  #pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2
  dim=1
  #pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
  ...
}
```

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

Figure: ARRAY_RESHAPE Pragma



X14307-1.10217

Syntax

Place the pragma in the C source within the region of a function where the array variable is defines.

```
#pragma HLS array_reshape variable=<name> \  
<type> factor=<int> dim=<int>
```

Where:

<name>

Required argument that specifies the array variable to be reshaped.

<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 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 block factor=2 dim=2
```

Example 3

Reshapes the three-dimensional 8-bit array, `AB [4] [2] [2]` in function `foo`, into a new single element array (a register), 128-bits wide ($4 \times 2 \times 2 \times 8$).

```
#pragma HLS array_reshape variable=AB complete dim=0
```

★ **Tip:** `dim=0` means to reshape all dimensions of the array.

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.

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, 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:** Primitive DSP 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: Supported Combinations of Functional Operations, Implementation, and Latency

Operation	Implementation	Min Latency	Max Latency
add	fabric	0	4
add	dsp	0	0
mul	fabric	0	4
mul	dsp	0	4
sub	fabric	0	4
sub	dsp	0	0

Table: 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

Operation	Implementation	Min Latency	Max Latency
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

Operation	Implementation	Min Latency	Max Latency
dmul	meddsp	0	13
dmul	fulldsp	0	13
dmul	maxdsp	0	14
dsqrt	fabric	0	58
drsqr	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. This usage 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](#).

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.

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: 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`, `lutr`, `uram`, `uram_ecc`, and `sr` as described below.

Table: Supported Implementation

Name	Description
URAM	UltraRAM resource
URAM	UltraRAM with ECC
SRL	Shift Register Logic resource
LUTRAM	Distributed RAM resource
BRAM	Block RAM resource
BRAM	Block RAM with ECC
AUTO	Vitis HLS automatically determine the implementation of the variable.

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, which lets Vitis HLS choose the latency.

Table: Supported Combinations of Memory Type, Implementation, and Latency

Type	Implementation	Min Latency	Max Latency
FIFO	MEMORY	0	0
FIFO	BRAM	0	0
FIFO	LUTRAM	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

Type	Implementation	Min Latency	Max Latency
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
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

Type	Implementation	Min Latency	Max Latency
ROM_nP	LUTRAM	1	3

!! Important: As shown in the following table, some combinations of memory type and implementation are not supported.

Table: Unsupported Combinations of Memory Type and Implementation

Memory Type	Unsupported Implementation
FIFO	URAM
RAM_1P	SRL
RAM_2P	SRL
RAM_S2P	SRL
RAM_T2P	SRL, LUTRAM
RAM_1WnR	SRL
ROM_1P	SRL, URAM
ROM_2P	SRL, URAM
ROM_nP	SRL, URAM

Example

In the following example, the `coeffs [128]` variable is an argument to the top-level function `foo_top`. The pragma specifies that `coeffs` uses a single port RAM implemented on a BRAM.

```
#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 as described in Exploiting Task Level Parallelism: Dataflow Optimization, 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: DATAFLOW Pragma

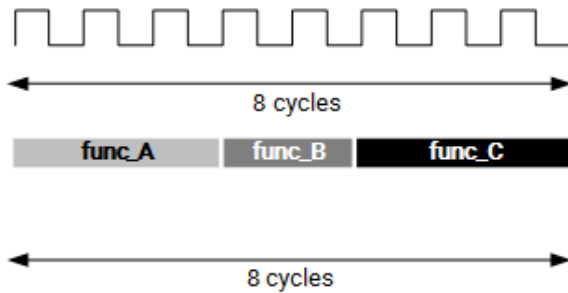
```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```

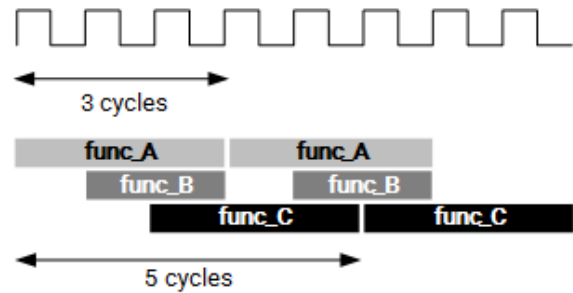
func_A

func_B

func_C



(A) Without Dataflow Pipelining



(B) With Dataflow Pipelining

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 Dataflow Optimization Limitations 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 as described in Managing Pipeline Dependencies.

!! 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>

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>

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>`

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>`

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`.

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=buff_A inter false
            #pragma HLS dependence variable=buff_B inter false
            if (col < cols) {
                buff_A[2][col] = buff_A[1][col]; // read from buff_A[1]
[col]
                buff_A[1][col] = buff_A[0][col]; // write to buff_A[1]
[col]
                buff_B[1][col] = buff_B[0][col];
                temp = buff_A[0][col];
            }
        }
    }
}
```

Example 2

Removes the dependence between `Var1` in the same iterations of `loop_1` in function `foo`.

```
#pragma HLS dependence variable=Var1 intra false
```

Example 3

Defines the dependence on all arrays in `loop_2` of function `foo` to inform the HLS tool that all reads must happen after writes (RAW) in the same loop iteration.

```
#pragma HLS dependence array intra 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.

Examples

The following example shows the struct variable `a` in function `foo_top` will be disaggregated:

```
#pragma HLS disaggregate variable=a
```

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.
- 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 `foo_sub` for all three instances of the function in `foo`. Each instance of function `foo_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 `foo_sub`.

```
char foo_sub(char inval, char incr) {  
    #pragma HLS function_instantiate variable=incr  
    return inval + incr;  
}  
void foo(char inval1, char inval2, char inval3,  
    char *outval1, char *outval2, char * outval3)  
{  
    *outval1 = foo_sub(inval1, 1);  
    *outval2 = foo_sub(inval2, 2);  
    *outval3 = foo_sub(inval3, 3);  
}
```

In the code sample above, the `FUNCTION_INSTANTIATE` pragma results in three different implementations of function `foo_sub`, each independently optimized for the `incr` argument, reducing the area and improving the performance of the function. After `FUNCTION_INSTANTIATE` optimization, `foo_sub` is effectively be transformed into three separate functions, each optimized for the specified values of `incr`.

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 `foo_top` inlining recursively down through the function hierarchy, except function `foo_sub` is not inlined. The recursive pragma is placed in function `foo_top`. The pragma to disable inlining is placed in the function `foo_sub`:

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

★ **Tip:** Notice in this example that `INLINE` applies downward to the contents of function `foo_top`, but applies upward to the code calling `foo_sub`.

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

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 [Managing Interface Synthesis](#).

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 function-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.
- Global variables accessed by the top-level function, and defined outside its scope:
 - If a global variable is accessed, but all read and write operations are local to the function, the resource is created in the RTL design. There is no need for an I/O port in the RTL. If the global variable is expected to be an external source or destination, specify its interface in a similar manner as standard function arguments. See the following [Examples](#).

★ **Tip:** The Vitis HLS tool automatically determines the I/O protocol used by any sub-functions. You cannot specify the `INTERFACE` pragma or directive for sub-functions.

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.

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> 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>

Specifies the interface protocol mode for function arguments, global variables used by the function, or the block-level control protocols. Refer to Details of Interface Synthesis for more information. 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, function return, or global variable 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. 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 `-register_io` option of the `config_interface` command globally controls registering all inputs/outputs on the top function.

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.

offset=<string>

Controls the address offset in AXI4-Lite (`s_axilite`) and AXI4 (`m_axi`) interfaces.

- For the `s_axilite` interface, `<string>` specifies the address in the register map.
- For the `m_axi` interface, `<string>` specifies one of the following values:
 - `direct`: Generate a scalar input offset port.
 - `slave`: Generate an offset port and automatically map it to an AXI4-Lite slave interface.
 - `off`: Do not generate an offset port.

★ **Tip:** The `-m_axi_offset` option of the `config_interface` command globally controls the offset ports of all M_AXI interfaces in the design.

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_type=<value>

For use with `ap_memory` and `bram` interfaces only. This options 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:
 $\text{num_read_outstanding} * \text{max_read_burst_length} * \text{word_size}$.

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:
 $\text{num_write_outstanding} * \text{max_write_burst_length} * \text{word_size}$.

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`.

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 axis port=A
    #pragma HLS INTERFACE 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 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 ap_vld register port=InData
```

This exposes the global variable `lookup_table` as a port on the RTL design, with an `ap_memory` interface:


```
pragma HLS interface ap_memory port=lookup_table
```

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 m_axi port=input offset=slave
    bundle=gmem0
    #pragma HLS INTERFACE m_axi port=output offset=slave
    bundle=gmem1

    #pragma HLS INTERFACE s_axilite port=input
    bundle=control
    #pragma HLS INTERFACE s_axilite port=output
    bundle=control
    #pragma HLS INTERFACE 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.

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 region is executed less frequently than the code in the enclosing function or loop. This allows the code that is executed less often to be pipelined at a slower rate, and potentially shared within the top-level pipeline. To determine the OCCURRENCE pragma, do the following:

- A loop iterates $\langle N \rangle$ times.
- However, part of the loop body is enabled by a conditional statement, and as a result only executes $\langle M \rangle$ times, where $\langle N \rangle$ is an integer multiple of $\langle M \rangle$.
- 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 region of code.

```
#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: $\langle N \rangle$ must be an integer multiple of $\langle M \rangle$.

Examples

In this example, the region `Cond_Region` has an occurrence of 4 (it executes at a rate four times less often than the surrounding code that contains it).

```
Cond_Region: {  
  #pragma HLS occurrence cycle=4  
  ...  
}
```

See Also

- [set_directive_occurrence](#)
- [pragma HLS pipeline](#)

pragma HLS pipeline

Description

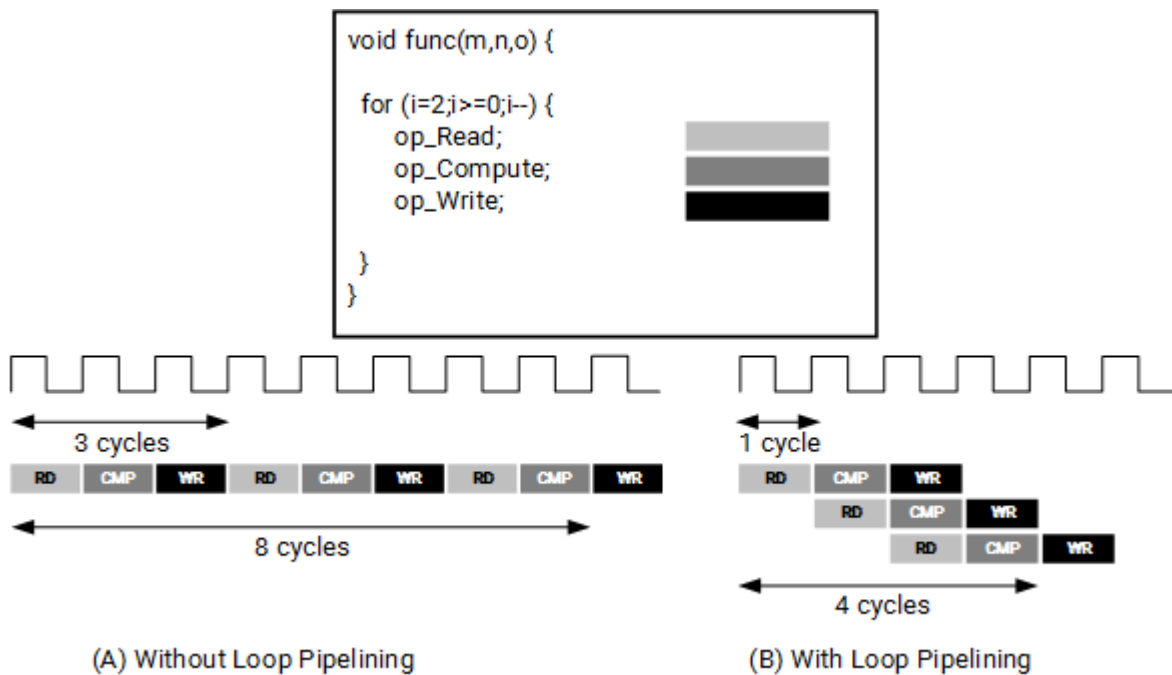
Reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations. The default type of pipeline is defined by the `config_compile -pipeline_style` command, but can be overridden in the PIPELINE pragma or directive.

A pipelined function or loop can process new inputs every `<N>` clock cycles, where `<N>` is the II of the loop or function. The default II for the PIPELINE pragma is 1, which processes a new input every clock cycle. You can also specify the initiation interval through the use of the II option for the pragma.

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: 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 enable_flush
```

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. The default II is 1.

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, or continuous loop pipelining with no pause between one loop iteration ending and the next iteration 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 | flp>

Specifies the type of pipeline to use for the specified function or loop. For more information on pipeline styles refer to Flushing Pipelines. 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.

flp

This option defines the pipeline as a flushable pipeline as described in Flushing Pipelines. 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 `foo` is pipelined with an initiation interval of 1, which is the default.

```
void foo { a, b, c, d} {  
    #pragma HLS pipeline II=1  
    ...  
}
```

 **Note:** The default value for `II` is 1, so `II=1` is not required in this example.

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 mode=[floating | fixed]
```

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

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 the 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 shared

Description

The SHARED pragma specifies that an array local variable or argument in a given scope is viewed as a single shared memory, distributing the available ports to the processes that access it.

If the array is a local variable in a dataflow region, the resulting memory implementation depends whether it is also stable or not. If a variable is also marked **STABLE** it 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.

However, if a variable is not STABLE it is synchronized like a regular Ping-Pong buffer, with depth but without a duplication of 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.

!! Important: There is no checking for the SHARED pragma or directive, so you must ensure the functional correctness of the sharing, including during co-simulation.

Syntax

Place the pragma in the C source within the boundaries of the function, where the variable is defined, or the global variable is used.

```
#pragma HLS shared variable=<variable>
```

Options

Where:

- `variable=<variable>`: Specifies the name of the variable that does not change during execution of the function.

Examples

In this example, variable `a` is shared among processes under the `p1` dataflow region:

```
#pragma HLS shared variable=a
```

See Also

- [set_directive_shared](#)
- [pragma HLS dataflow](#)
- [pragma HLS stable](#)

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 Managing Interface Synthesis for more information.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS stream variable=<variable> depth=<int> off
```

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 `ll=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>`.

off

Disables streaming data. Relevant only for array streaming in dataflow channels.

★ **Tip:** The `config_dataflow -default_channel fifo` command globally implies a `STREAM` pragma on all arrays in the design. The `off` option specified here overrides the `config_dataflow` command for the specified `<variable>`, and restores the default of using a RAM ping pong buffer-based channel.

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
```

Example 3

Array C has streaming disabled. In this example, it is assumed to be enabled by `config_dataflow -default_channel`.

```
#pragma HLS STREAM variable=C off
```

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](#)